N° d'ordre : 139
N° attribué par la bibliothèque : 99ENSL0139

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir le grade de

Docteur de l'École normale supérieure de Lyon
spécialité Informatique
au titre de la formation doctorale de MathIf

par Georges-André SILBER

Parallélisation automatique par insertion de directives

Présentée et soutenue publiquement le 15 décembre 1999

Après avis de : Monsieur Marc GENGLER
Madame Catherine MONGENET

Devant la commission d'examen formée de :
Monsieur Alain DARTE (directeur de thèse)
Monsieur Jean-Luc DEKEYSER
Monsieur Marc GENGLER
Monsieur François IRIGOIN
Madame Catherine MONGENET
Monsieur Yves ROBERT (directeur de thèse)

Thèse préparée à l'École normale supérieure de Lyon
au sein du Laboratoire de l'Informatique du Parallélisme.

Remerciements

Cette thèse a été préparée au sein du laboratoire de l'informatique du parallélisme (LIP) de l'École normale supérieure de Lyon, entre septembre 1996 et décembre 1999, sous la direction conjointe d'Alain Darte et Yves Robert. Je voudrais tout d'abord remercier Alain Darte (chargé de recherches au CNRS) qui m'a encadré pendant cette thèse et avec qui j'ai eu la chance de faire mes premiers pas dans le monde de la recherche scientifique. Sa rigueur, sa disponibilité sans faille, son enthousiasme communicatif, son perfectionnisme, son intégrité scientifique et sa franchise resteront toujours un exemple que j'essaierai de suivre lors de ma future carrière scientifique. Je garderai un excellent souvenir de ces années passées à travailler en sa compagnie. Je remercie également vivement Yves Robert (professeur à l'ENS de Lyon) pour ses conseils et ses remarques toujours pertinentes et éclairées tout au long du déroulement de cette thèse, sans oublier Neil Young.

Je suis très reconnaissant envers tous les membres de mon jury, tout particulièrement envers Catherine Mongenet (professeur à l'Université de Strasbourg I) et Marc Gengler (professeur à l'université de la Méditerrannée) pour leur relecture détaillée de mon manuscrit, et également envers François Irigoin (professeur à l'École des mines de Paris) et Jean-Luc Dekeyser (professeur à l'Université de Lille) pour l'attention qu'ils ont portée à mon travail.

J'aimerais également remercier tous les membres de l'équipe ReMaP avec qui j'ai eu la joie de travailler durant ces années, je pense notamment à Frédéric Vivien, Frédéric Desprez, Guillaume Huard, Pierre Boulet, Pierre-Yves Calland et Jean-Christophe Mignot. Parmi les membres du LIP, j'aimerais remercier tout particulièrement Jean-Michel Muller pour son attention et sa gentillesse, Jean Duprat pour son regard atypique, Marianne Delorme pour ses combats que je partage et de nombreuses autres personnes dont le nom me viendra lorsque ces lignes seront déjà imprimées. Merci à Serge Miguet et à Tao Yang pour m'avoir accueilli en stage. Un grand merci à Odile Millet-Botta qui a été mon tuteur lors de ma première année à l'ENS de Lyon et qui m'a beaucoup soutenu. Je tiens tout particulièrement à remercier Luc Bougé, pour sa droiture, sa gentillesse, son soutien et ses encouragements constants, ainsi que pour ses enseignements et son approche pédagogique au sein du Magistère d'informatique de l'ENS de Lyon qui sont un modèle que j'essaierai de suivre lors de ma carrière d'enseignant-chercheur. Je n'oublierai pas le personnel administratif et technique du LIP qui est pour beaucoup dans le fait que ce laboratoire soit très confortable et que les déboires administratifs soient limités au minimum.

Merci à tous mes proches, parents et amis. Une grande pensée pour ma grand mère qui m'a beaucoup donné. Je ne sais pas comment remercier mes parents qui sont des modèles pour moi et à qui je dois énormément. Ils m'ont toujours épaulé quels qu'aient été mes choix. J'aimerais qu'ils reçoivent toute mon affection et qu'ils sachent que je suis fier d'être leur fils. Je garde pour la fin ma femme Frédérique et ma fille Emma, à qui je dois la joie de vivre, un grand amour et un soutien gigantesque.

Table des matières

Chapitre 1

Introduction

Les compilateurs sont des éléments clés de l'informatique. En effet, ils permettent aux utilisateurs de décrire des méthodes pour résoudre leurs problèmes sous forme de *programmes* dans un langage « presque » naturel (en tous cas, relativement éloigné du langage de la machine). Un compilateur doit traduire au mieux ces programmes dans le langage de la machine pour que ceux-ci soient entre autres : 1) fidèles à la description donnée par le programmeur et 2) rapides à l'exécution. La priorité est bien sûr donnée au premier point, puisque le programmeur doit obtenir une solution correcte à son problème (en considérant que sa méthode de résolution est correcte). L'obtention de code machine s'exécutant rapidement est une tâche difficile. La rapidité dépend de nombreux paramètres, comme par exemple le type de machine pour laquelle le programme doit être traduit ou encore la quantité de mémoire rapide (mémoire cache) disponible, le nombre de registres, l'architecture de la machine. Il est donc très difficile de trouver des méthodes globales et absolues pouvant être appliquées systématiquement pour obtenir de bonnes performances, puisque chaque programme est différent et que les méthodes qu'il faudrait appliquer pour obtenir un code machine peuvent également changer selon la machine ciblée. Pour rendre le problème encore plus difficile, toutes les méthodes utilisées par le compilateur pour améliorer la rapidité des programmes doivent évidemment respecter le premier point énoncé plus haut.

La recherche en *parallélisation automatique*, domaine utopique de la recherche en informatique, consiste à trouver des algorithmes permettant à un compilateur de transformer automatiquement un programme conçu pour une machine à un seul flot d'instructions en un programme pour une machine comprenant plusieurs flots d'instructions, autrement appelée machine *parallèle*. Dans l'absolu, le but rêvé est que le compilateur trouve tout seul le code machine optimal pour la machine parallèle considérée, sans intervention du programmeur. Malheureusement, trouver le code machine optimal suppose de pouvoir analyser le programme très finement, pour savoir isoler les composantes du programmes qui peuvent être exécutées de manière concurrente. Cette analyse est extrêmement difficile, lorsqu'elle est seulement faisable. Une fois toutes les parties concurrentes isolées, il faut encore être capable de tirer parti au mieux des possibilités de la machine parallèle ciblée, ce qui peut être considéré comme aussi difficile que l'analyse précédente.

En même temps que les machines parallèles sont apparus des langages de programmation parallèle, c'est-à-dire des langages où le programmeur traite explicitement le parallélisme. Nous ne parlerons pas ici des bibliothèques de passage de messages (MPI, PVM), que nous considérons comme étant de trop bas niveau pour le programmeur, mais des langages où le programmeur peut spécifier les parties concurrentes de son programme à l'aide de directives de parallélisation (OpenMP, HPF, directives Cray). Le compilateur est alors chargé de traduire celui-ci en langage parallèle de bas niveau, comme par exemple avec du passage de message ou des processus légers (threads). Le compilateur doit toujours être relativement puissant pour savoir tirer parti au mieux de la machine parallèle, mais nous lui avons ôté une difficulté en identifiant pour lui les parties concurrentes et le placement éventuel des données. Cette traduction du langage à base de directives en code machine parallèle n'est pas traitée dans cette thèse.

Dans cette thèse, nous nous plaçons dans le cadre de la parallélisation automatique au niveau du code source. Les compilateurs destinés à accueillir nos transformations prennent en entrée un langage séquentiel et fournissent un sortie un langage parallèle avec des directives de parallélisation. Le langage cible des compilateurs que nous considérons n'est donc pas du langage machine mais reste du langage de haut-niveau. Nous laissons la génération du code de bas niveau au type de compilateur dont nous avons parlé au paragraphe précédent. Cette approche qui justifie le titre de cette thèse permet de se concentrer sur la détection des composantes parallèles des programmes.

Comme nous l'avons dit précédemment, cette détection suppose une analyse fine du programme par des moyens sophistiqués. Le programmeur connaît certaines propriétés de son programme. Il connaît également la machine sur laquelle il veut exécuter son programme. Nous avons donc choisi de nous placer dans le cadre de la parallélisation semi-automatique, où le programmeur peut agir sur la façon dont le parallélisme est détecté et exhibé dans certaines parties de son code. Cette approche suppose de développer des algorithmes de parallélisation qui soient suffisamment généraux pour pouvoir accepter un grand degré de liberté dans les codes parallèles qu'ils peuvent produire. Ainsi, lors de la parallélisation d'un code, l'utilisateur peut laisser le compilateur paralléliser automatiquement certaines parties de son programme (typiquement, les parties où le parallélisme est évident) et agir plus finement sur les parties de son code qui demandent plus d'attention et d'expertise.

Dans la première partie de cette thèse (§2, p. 5), nous allons présenter le cadre de notre étude. Nous ne donnerons pas de résultats nouveaux mais nous présenterons les concepts et définitions nécessaires à la compréhension de notre travail. Nous allons notamment parler des *boucles*, c'est-à-dire les structures itératives que nous allons chercher à paralléliser. Nous nous concentrons sur ces structures car elles monopolisent la plupart du temps de calcul des programmes scientifiques courants. Nous introduirons également la notion de *boucle parallèle* et nous donnerons la sémantique d'une telle boucle. Nous avons parlé au tout début de cette introduction de la fidélité à la description du programme donnée par le programmeur. Nous présenterons donc dans cette première partie un formalisme, les *dépendances* du programme, qui donne les conditions à respecter pour assurer cette fidélité.

La deuxième partie de cette thèse (§3, p. 27) est une extension de l'algorithme de parallélisation développé par Alain Darte et Frédéric Vivien [24, 69]. Nous avons caractérisé l'ensemble des transformations et donné un algorithme permettant de trouver le nombre maximal de boucles complètement permutables d'un ensemble de boucles imbriquées avec une représentation des dépendances par des polyèdres. La transformation d'un ensemble de boucles imbriquées en des boucles complètement permutables est une transformation nécessaire à l'application d'une opération ultérieure, le « tiling ». Cette opération consiste à regrouper des opérations pour augmenter le grain de la parallélisation, afin de mieux utiliser la hiérarchie mémoire ou d'éviter d'avoir un rapport calculs/communications défavorable sur une machine à mémoire distribuée. En fait, une fois que les boucles imbriquées sont sous la forme de boucles permutables, tous les grains de parallélisation sont possibles, du grain fin au gros grain, en passant par toutes les tailles intermédiaires. Cet algorithme laisse donc une grande liberté au programmeur ou au compilateur pour obtenir un code parallèle de bas niveau bien adapté à la machine parallèle ciblée.

Dans la troisième partie de cette thèse (§4, p. 45), nous nous intéressons aux codes comprenant des dépendances impliquant des expressions contenues dans des structures de contrôle (codes à contrôle non statique). La parallélisation de tels codes pose des problèmes particuliers puisqu'il faut prendre en compte les dépendances de contrôle. Les méthodes proposées antérieurement introduisaient systématiquement de la mémoire temporaire pour supprimer ces dépendances de contrôle [4, 36]. Nous introduisons un nouveau type de graphe, le *graphe des dépendances mixte*, qui permet de représenter exactement les dépendances à respecter dans ce type de code. Nous présentons également un mécanisme de transformation de ce graphe lors de l'introduction d'un temporaire pour supprimer une anti-dépendance issue d'une structure de contrôle. Nous montrons que notre approche permet de trouver tous les codes légaux possibles lorsque les seules opérations de transformation autorisées sont la fusion et la distribution. Nous présentons un algorithme de parallélisation se fondant sur celui d'Allen, Callahan et Kennedy [2] et qui est optimal en terme de parallélisme avec la représentation des dépendances par niveau considérée et avec les temporaires autorisés par le programmeur. Nous montrons également que la minimisation du nombre de temporaires est un problème NP-complet au sens fort mais qu'une recherche exhaustive est possible au vu de la taille des problèmes dans les programmes courants.

La quatrième partie de cette thèse (§5, p. 85) présente un environnement de compilation appelé Nestor. L'idée de cet environnement est apparu lors de l'implantation des algorithmes des deux chapitres précédents. Nous voulions valider ces algorithmes sur de vrais codes en Fortran 77 ou en Fortran 90, et générer du HPF, du Fortran avec OpenMP ou des directives Cray pour pouvoir tester ces codes parallélisés. Devant le manque d'outils permettant cette implantation et devant la difficulté d'utilisation et les limitations des outils existants, nous avons décidé de développer un nouvel environnement permettant l'implantation aisée de transformations de programme au niveau du code source. Après un long travail est né Nestor sous la forme actuelle : une bibliothèque C++ offrant un ensemble de classes décrivant hiérarchiquement un programme source, avec des méthodes pour parcourir, modifier ou augmenter la représentation interne, avec une documentation complète consultable en ligne, avec un support pour l'écriture de transformations dynamiques de code, avec une interface graphique permettant une vraie transformation semi-automatique. Nestor offre en entrée un support pour traiter les directives dirigeant la parallélisation, notamment pour spécifier les portions de code à traiter, et en entrée/sortie un support pour Fortran 77 et 90 avec HPF, OpenMP ou les directives Cray. Un ensemble de briques de base sont intégrées à l'environnement, comme l'analyse des dépendances de données ou la manipulation de graphes. L'algorithme précédent a été implanté dans cet environnement. Nestor est un logiciel libre et gratuit qui peut être téléchargé avec l'ensemble de ses sources [57] et qui a la vocation d'apporter une grande aide aux utilisateurs désirant paralléliser leurs codes et aux chercheurs en compilation désirant implanter leurs transformations.

Pour finir cette thèse, nous donnerons quelques conclusions et quelques pistes de recherche future dans la lignée des travaux que nous avons présentés. Mais avant tout cela, passons aux préliminaires.

Chapitre 2

Préliminaires

Trois langages de programmation sont actuellement « en vogue » dans le cadre du calcul à hautes performances, Fortran et ses dialectes (Fortran 90, 95, HPF [31], OpenMP [66]), C et ses variantes (C++), ainsi que le dernier né, Java. Pour des raisons historiques, le langage Fortran domine encore actuellement le calcul à hautes performances, de part son implantation dans le monde des physiciens. Loin des querelles qui font naître d'interminables débats quant à savoir quel langage est le « meilleur », nous pouvons noter que ces trois langages partagent le même modèle de calcul et à peu près les mêmes instructions et types de données. Ils possèdent tous trois les mêmes structures de contrôle : les conditionnelles, différents types de boucles et la récursion. Ils ont les mêmes types de base (avec des différences minimes), comme les entiers, les réels et les caractères et permettent de définir des types utilisateurs. Ainsi, les notions que nous allons présenter dans ce chapitre s'appliquent à n'importe lequel de ces langages. Dans cette thèse, nous nous intéressons plus particulièrement aux boucles, c'est-à-dire aux instructions itératives comme le `do` de Fortran ou le `for` de C et Java. Une structure itérative permet de définir succinctement un ensemble de calculs répétitifs : au sein d'un programme scientifique, l'ensemble de ces structures monopolise généralement la majorité du temps de calcul.

La première partie de ce chapitre définit formellement la notion de *boucle* ainsi que sa sémantique d'exécution. Dans la deuxième partie de ce chapitre, après avoir défini la notion d'exécution parallèle d'une boucle, nous verrons quels sont les différentes possibilités d'expression de ce parallélisme dans les différents langages que nous étudions : HPF pour Fortran, OpenMP pour Fortran et C, processus légers (threads) pour Java. La troisième partie est consacrée à une légère digression parlant des différentes possibilités de génération de code lorsqu'une boucle séquentielle a été mise sous la forme d'une boucle parallèle. Les transformations de nids de boucles permettant d'obtenir de nouvelles boucles où plusieurs instructions peuvent être exécutées en parallèle sont gouvernées par un certain nombre de contraintes. La plus importante d'entre elles est le respect de la sémantique du code originel : le résultat du code parallélisé doit être le même que celui du code séquentiel original. C'est pourquoi ces transformations devront respecter les dépendances de donnée et de contrôle entre les instructions. C'est l'objet de la quatrième et de la cinquième partie de ce chapitre. La cinquième partie clôt ce chapitre par une présentation du graphe des dépendances de donnée particulière appelée le *graphe des dépendances de donnée réduit polyédrique*.

2.1 Boucles

Les *instructions* d'un programme écrit dans un langage comme Fortran, C ou Java peuvent être rangées dans trois catégories. Les *actions*, les *structures de contrôle* et les *branchements*.

Les actions sont les instructions de base d'un programme. Elles peuvent induire un changement dans l'état du calcul ou produire un effet de bord. Ces actions comprennent l'affectation, c'est-à-dire l´évaluation d'une expression et son affectation à une variable, l'appel d'une procédure et les entrées/sorties.

Les structures de contrôle permettent de structurer l'exécution des différentes actions du programmes. Il y a deux types de structure de contrôle, les *structures itératives* et les *structures conditionnelles*. Les structures itératives (ou boucles) provoquent l'exécution répétitive d'une instruction ou d'un groupe d'instructions. Il y a diverses formes de structures itératives, comme le do de Fortran, le for de C, le while, etc... Les structures conditionnelles conditionnent l'exécution d'une instruction ou d'un groupe d'instructions en évaluant une expression booléenne. L'instruction if et l'instruction switch (qui peut être transformée en un série d'instructions if) sont les deux structures conditionnelles principales.

Les branchements provoquent un transfert de contrôle explicite à un autre endroit du programme. Parmi ces instructions on trouve le goto de C et Fortran ainsi que toutes ses variantes.

Parmi les structures de contrôle, les structures itératives sont celles qui permettent d'exprimer en peu de lignes un grand nombre de calculs, notamment en calcul scientifique. Les structures itératives telles que le do de Fortran ou le for de C ont été historiquement les structures qui ont le plus concentré l'attention dans le domaine de l'optimisation de programmes, principalement du fait de leur régularité et du gain que l'on peut obtenir en optimisant ces structures qui monopolisent la plupart du temps de calcul des applications scientifiques. Nous n'allons pas déroger à la règle et nous allons définir un modèle de structure itérative que nous nommerons avec beaucoup d'originalité une *boucle*.

Définition 1 (Boucle) *Nous appelons* boucle *et nous notons* do *une structure itérative de programme finie de la forme :*

$$\begin{array}{l} \mathsf{do}\ i = b_{\text{deb}}, b_{\text{fin}}, b_{\text{pas}} \\ \quad s_1 \\ \quad \vdots \\ \quad s_d \\ \mathsf{enddo} \end{array}$$

où s_j est une instruction, i une variable entière appelée indice, *b_{deb} et b_{fin} des expressions entières définissant les* bornes de boucle *et b_{pas} une expression entière dont la valeur est différente de zéro définissant le* pas *de la boucle. L'ensemble des instructions $s_1, \ldots, s_d$ est appelé le* corps *de la boucle. Le corps de la boucle ne doit pas contenir d'instruction de branchement.*

Cette définition est très générale et nous allons parfois la restreindre, notamment lorsque nous parlerons de recherche d'ordonnancement. Nous supposerons alors que les expressions b_{deb} et b_{fin} sont des expressions affines. De même, nous nous restreindrons parfois à des expressions b_{pas} constantes et égales à 1 et dans ce cas nous omettrons cette expression et supposerons que $b_{\text{deb}} \leq b_{\text{fin}}$. Dans ce cas nous parlerons de boucle *régulière*. La sémantique d'exécution d'une boucle est définie ci-dessous et correspond à l'exécution d'une boucle dans un langage impératif comme Fortran.

Définition 2 (Exécution d'un boucle) *Une boucle* do *est exécutée de la façon suivante :*

1. *La valeur des trois expressions entières b_{deb}, b_{fin} et b_{pas} est calculée.*
2. *Le compteur i prend comme valeur b_{deb}.*
3. *Si $b_{\text{pas}} > 0$ et $i \leq b_{\text{fin}}$ ou si $b_{\text{pas}} < 0$ et $i \geq b_{\text{fin}}$, alors pour la valeur courante de i les instructions du corps de la boucle sont exécutées dans l'ordre où elles apparaissent dans le texte du programme et l'exécution est reprise au point 3 après avoir ajouté b_{pas} à i ; sinon aller au point 4.*
4. *L'instruction suivant la boucle est exécutée.*

L'ordre dans lequel les instructions du corps de la boucle apparaissent dans le texte du programme est appelé l'*ordre textuel*. Cette sémantique d'exécution garantit que les bornes de la boucle ne seront évaluées qu'une seule fois avant de commencer à itérer. C'est la différence principale avec une structure itérative telle que le while ou le repeat, où l'expression contenue dans la condition d'arrêt est évaluée à chaque itération. D'autre part, dans le corps de la boucle, la variable i est considérée comme une constante et ne peut donc pas être modifiée par une instruction. Cette sémantique est celle du langage Fortran. Le langage C et la langage Java ne fournissent pas de boucle telle que nous l'avons définie : en effet pour ces deux langages la boucle for n'est qu'une boucle while cachée par du sucre syntaxique. Pour ces deux langages, nous allons uniquement considérer les boucles for écrites de telle façon qu'il soit possible de respecter cette sémantique. L'exemple 2.1 illustre le type de boucles qui ne respecte pas cette sémantique et qui peut amener à des résultats inattendus.

```
a(1) = 4
do i = 1, a(1)
   a(i) = 1
enddo
```

(a) Fortran

```
a[1] = 4 ;
for (i = 1 ; i ≤ a[1] ; i++)
   a[i] = 1 ;
```

(b) C

Exemple 2.1: deux fragments de programme apparemment similaires mais qui ne donnent pas le même résultat puisque la sémantique des boucles en Fortran et en C n'est pas la même. En Fortran, la boucle a quatre itérations alors qu'en C, la boucle a une seule itération.

Parmi les instructions qui composent le corps d'une boucle, il peut y avoir d'autres boucles. Dans ce cas, nous parlerons de *boucles imbriquées* ou de *nid de boucles*. Lorsque ces boucles sont parfaitement imbriquées, comme dans l'exemple de la figure 2.1, nous parlerons de *nid de boucles parfait* ou de *boucles parfaitement imbriquées*.

Nous noterons n la *profondeur* du nid de boucles, qui correspond au nombre de boucles imbriquées ou en d'autres termes au nombre de boucles englobant une instruction s_j du corps du nid de boucles. Un nid de boucle parfait régulier de profondeur n décrit un polyèdre fini P de l'espace d'itération $\mathbb{Z}^n$ borné par les bornes des boucles. Nous appellerons ce polyèdre le *domaine d'itération*. Chaque itération de la boucle correspond à un point entier du polyèdre et est identifié par un *vecteur d'itération* $\vec{p} = (p_1, ..., p_n)$ ($\vec{p}$ est un vecteur colonne) où p_l (aussi noté $[\vec{p}]_l$) est la valeur de l'indice de la boucle l, en partant de la boucle la plus

```
do i_1 = b_inf_1, b_sup_1          (* B_1 *)
   do i_2 = b_inf_2, b_sup_2       (* B_2 *)
     \
      do i_n = b_inf_n, b_sup_n    (* B_n *)
         s_1
         ⋮
         s_d
      enddo
     /
   enddo
enddo
```

Figure 2.1: un nid de boucles parfaitement imbriquées de profondeur n.

externe (nous parlerons parfois également de boucle de profondeur l). P peut être défini par un ensemble d'inégalités affines de la forme :

$$\{\vec{p} \in \mathbb{Z}^n \mid A\vec{p} \leq b\}.$$

Définition 3 (Opération) *On appellera* opération *et l'on notera $s_j(\vec{p})$ une instance d'instruction s_j, c'est-à-dire l'exécution d'une instruction lors d'une itération donnée.*

```
do i = 0, N
   do j = 0, N
      a(i,j) = b(i,j-6) + d(i-1,j+3)     (* s_1 *)
      b(i+1,j-1) = c(i+2,j+5)            (* s_2 *)
      c(i+3,j-1) = a(i,j-2)              (* s_3 *)
      d(i,j-1) = a(i,j-2)                (* s_4 *)
   enddo
enddo
```

$$\begin{aligned} P &= \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i,j \leq N\} \\ &= \{\vec{p} \in \mathbb{Z}^2 \mid A\vec{p} \leq b\} \end{aligned}$$

$$A = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \quad b = \begin{pmatrix} N \\ 0 \\ N \\ 0 \end{pmatrix}$$

Exemple 2.2: exemple de nid de boucle parfait avec son domaine d'itération.

L'exemple 2.1 donne le code d'un nid de boucle parfait régulier tiré de [47] avec son domaine d'itération. Dans le cas d'un nid de boucles parfait, toutes les instructions vont partager le même domaine d'itération P, à la différence d'un nid de boucles non parfait. D'une manière générale, nous pouvons attacher à chaque instruction s d'une boucle un domaine d'itération P_s.

Dans un nid de boucles parfait, les itérations sont exécutées dans *l'ordre lexicographique* de leurs vecteurs d'itération. Nous notons $\prec$ l'ordre lexicographique strict :

$$\forall \vec{p}, \vec{q} \in \mathbb{Z}^n, \quad \vec{p} \prec \vec{q} \iff \exists k \leqslant n, \quad \forall l < k, \quad [\vec{p}]_l = [\vec{q}]_l \text{ et } [\vec{p}]_k < [\vec{q}]_k$$

et $\preccurlyeq$ l'ordre lexicographique large.

Nous écrirons $\vec{p} \prec \vec{q}$ si $\vec{p}$ est inférieur lexicographiquement à $\vec{q}$ et $\vec{p} \preccurlyeq \vec{q}$ si $\vec{p} = \vec{q}$ ou $\vec{p} \prec \vec{q}$. On dira qu'un vecteur $\vec{p}$ est *lexicographiquement positif* (noté $\vec{p} \succ \vec{0}$) si la première composante non nulle de ce vecteur est strictement positive. L'ordre lexicographique agrémenté de l'ordre textuel des instructions au sein des boucles est l'ordre d'exécution standard des boucles imbriquées d'un langage impératif comme Fortran (l'ordre donné par la définition 2 lorsque les pas valent 1).

2.2 Parallélisme implicite

Les trois langages dont nous avons parlé plus haut offrent la possibilité d'exprimer le fait que toutes ou une partie des itérations d'une boucle peuvent être exécutées de manière concurrente ou parallèle. Cette expression se fait à travers des extensions du langage comme HPF [31], OpenMP [66], CrayMP, SunMP, pour ne citer qu'eux. Ces extensions permettent de placer des *directives* dans le programme, le plus souvent sous la forme de commentaires qui offrent une information au compilateur lui permettant de générer un code parallèle en distribuant les calculs sur différents processus. Nous allons nous concentrer dans la partie suivante sur les extensions permettant de modifier la sémantique d'exécution des boucles.

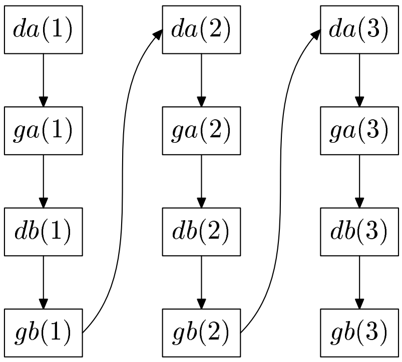
2.2.1 Boucle parallèle

Lorsque l'on entend parler de structure itérative parallèle dans la littérature, le terme « boucle parallèle » revient souvent, bien que cela soit un abus de langage : il n'y a pas à proprement parler de « boucle » dans une boucle parallèle. Les itérations sont exécutées de manière concurrente et cette structure n'est donc plus vraiment une structure itérative. Nous ne dérogerons cependant pas à la règle et emploierons le terme *boucle parallèle* pour désigner l'exécution concurrente des itérations d'une boucle. Lorsque nous emploierons le terme boucle tout seul, ce sera au sens des définitions 1 et 2. Avant de définir ce que nous entendons exactement par boucle parallèle, nous allons examiner quelques possibilités d'expression du parallélisme dans les boucles de différents langages.

En Fortran 95 est apparue une nouvelle construction itérative qui est le forall. Cette construction n'est pas à proprement parler une construction parallèle mais une implantation parallèle est possible. La sémantique d'exécution d'un forall est très différente de la sémantique d'exécution d'une boucle. Dans la figure 2.2, l'exécution d'une boucle do est illustrée par un graphe où les sommets sont les différentes composantes des instructions d'affectation du corps de la boucle et les arcs sont des contraintes de précédence entre ces composantes. L'exécution suit la définition 2 : pour la première itération, la partie droite de la première affectation ($da(1)$) est calculée puis affectée à la partie gauche ($ga(1)$), puis la partie droite de la deuxième affectation ($db(1)$) est calculée puis affectée à la partie gauche ($gb(1)$), et ainsi de suite pour toutes les itérations. Dans le cas d'un forall, le graphe illustrant l'exécution est plus compliqué : les parties droites de toutes les itérations de la première affectation ($da(1)$, $da(2)$, $da(3)$) sont calculées, puis elles sont affectées aux parties gauches de toutes les itérations de la première affectation, etc... La sémantique d'exécution du forall permet d'implanter le calcul des parties droites de la première itération en parallèle, puis toutes les affectations de la première instruction, puis de la même façon pour la deuxième instruction, avec la nécessité d'effectuer une synchronisation entre toutes ces phases de calcul. Le forall est l'instruction exécutée implicitement lors de l'écriture des expressions de tableau de Fortran 90 : le forall de la figure 2.2 est équivalent aux deux instructions d'affectation $ga = da$ et $gb = db$ de Fortran 90.

En Fortran, les extensions permettant d'exprimer du parallélisme dans les boucles sont HPF, OpenMP, CrayMP et SunMP (cette liste n'est pas exhaustive). OpenMP est également

do $i = 1, 3$
 $ga(i) = da(i)$
 $gb(i) = db(i)$
enddo



forall $(i = 1 : 3)$
 $ga(i) = da(i)$
 $gb(i) = db(i)$
endforall

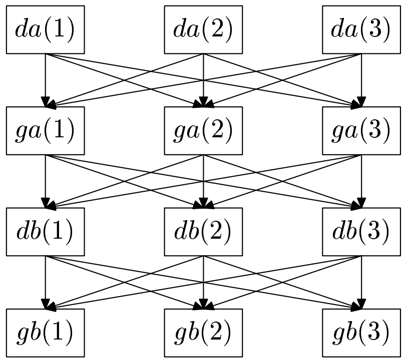
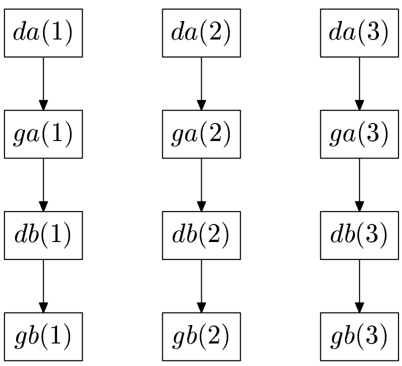


Figure 2.2: comparaison de la sémantique d'exécution d'une boucle et d'un forall.

disponible pour C et C++ et offre les mêmes possibilités que l'implantation Fortran, avec une restriction sur les boucles pouvant être désignées parallèles. Nous ne donnerons pas la syntaxe de toutes ces directives, se référer pour cela aux document suivants : la documentation de HPF 2.0 [31], la documentation d'OpenMP [66] et un excellent comparatif des différents dialectes parallèles [70].

La figure 2.3 donne la sémantique d'exécution d'une boucle en présence d'une directive d'exécution parallèle. Par rapport à la figure 2.2, un certain nombre d'arcs ont été enlevés dans les graphes qui illustrent l'exécution de ces boucles : les contraintes d'exécution ont été réduites et une exécution parallèle de ces structures itératives peut être implantée. Nous pouvons voir que la sémantique d'exécution est plus «lâche» que celle d'un forall : il n'y a plus de synchronisations entre les différentes phases de calcul.

!HPF$ independent
do $i = 1, 3$
 $ga(i) = da(i)$
 $gb(i) = db(i)$
enddo



!HPF$ independent
forall $(i = 1 : 3)$
 $ga(i) = da(i)$
 $gb(i) = db(i)$
endforall

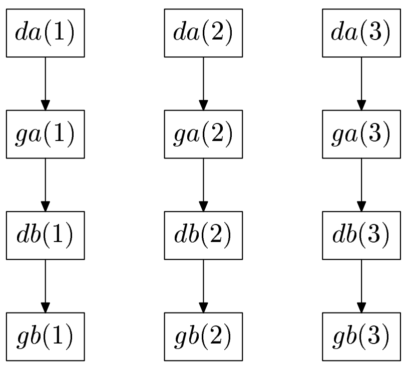


Figure 2.3: comparaison de la sémantique d'exécution d'une boucle et d'un forall en présence d'une directive d'exécution parallèle.

Plutôt que de décorer à chaque fois une boucle parallèle avec sa directive d'exécution parallèle, nous noterons une telle boucle doall dans la suite de ce document. La définition d'une boucle parallèle est la même que celle d'une boucle de la définition 1, par contre c'est la sémantique d'exécution qui change. Nous emploierons le terme *processus* pour désigner un contexte d'exécution d'instructions. Un *processeur* permet d'exécuter un processus. Deux processus peuvent s'exécuter en parallèle s'ils sont placés sur deux processeurs différents et un ensemble de processus peut se synchroniser. De part sa sémantique d'exécution spéciale, un doall peut être exécuté de différentes façons, notamment comme un forall ou un do.

Définition 4 (Exécution d'un boucle parallèle) *Une boucle parallèle doall est exécutée de la façon suivante :*

1. *La valeur des trois expressions entières b_{deb}, b_{fin} et b_{pas} est calculée.*
   (a) *En considérant qu'il y a autant de processus indépendants disponibles que d'itérations, chaque processus prend en charge une itération, c'est-à-dire une valeur de i (telle que $i = b_{\text{deb}} + \lambda b_{\text{pas}}$ avec λ entier et $i <= b_{\text{fin}}$) et exécute les instructions du corps de la boucle dans l'ordre où elles apparaissent dans le programme.*
   (b) *Les processus se synchronisent.*
2. *L'instruction suivant la boucle est exécutée.*

Il est important de noter qu'à la différence de la définition 2, la valeur de i à la sortie de la boucle est indéterminée. En suivant le point (a), toutes les itérations peuvent être exécutées en même temps, d'où le terme de « boucle parallèle ». Avec cette définition, nous partons du principe qu'il y a autant de processus que d'itérations de la boucle. S'il y a moins de processus que d'itérations, il est toujours possible de regrouper des itérations et de faire en sorte qu'un processus exécute plusieurs itérations. À l'extrême, cette boucle parallèle peut être exécutée par un seul processus, mais l'exécution peut être différente de la définition 2 puisque les itérations peuvent être exécutées dans n'importe quel ordre. (Certaines boucles parallèles, notamment celles générées automatiquement, offrent parfois encore plus de parallélisme sous la forme de parallélisme *intra*-itération : si nous reprenons la figure 2.3, il n'y a pas d'arc entre $ga(i)$ et $db(i)$.)

Les directives de parallélisation d'une boucle permettent de spécifier également quelles doivent être les variables locales à une itération de la boucle (on parle également de variable *privatisée* ou *privée*).

2.2.2 Autres directives

De nombreuses autres directives permettant de faciliter la programmation parallèle pour les machines à mémoire distribuée et partagée existent (notamment pour définir des tâches parallèles). Pour les machines à mémoire distribuée, HPF tient le haut du pavé [31] puisqu'il offre des directives de placement de données. Pour les machines à mémoire partagée, il existe de nombreux dialectes parallèles qui ressemblent tous plus ou moins à la définition du comité X3H5 [77] qui n'est malheureusement jamais devenue une norme ISO. Tous les constructeurs de machines parallèles à mémoire partagée fournissent leurs propres directives dans leurs compilateurs : Cray/SGI, HP, Digital, Sun, NEC, Fujitsu. Un rapport de Klaas Jan Wierenga résume parfaitement tous ces dialectes avec leurs différences [70]. Le modèle OpenMP [66, 30, 45] est une tentative prometteuse d'unifier tous ces dialectes et de rajouter des fonctionnalités à la définition du comité X3H5. Un compilateur pour Fortran/OpenMP est disponible sur la plupart des machines parallèles à mémoire partagée [45].

2.3 Parallélisme explicite

Dans cette partie, nous allons faire une légère digression pour examiner comment un compilateur qui reconnaît en entrée une boucle doall peut implanter le code parallèle correspondant. Si nous reprenons l'exemple utilisé précédemment dans la figure 2.3, un compilateur générant du code pour une machine multiprocesseurs à mémoire distribuée peut générer deux types de code, en suivant la règle qui dit que le processus qui effectue le calcul est celui qui possède la donnée modifiée (*owner computes rule*). Nous nous plaçons ici dans l'optique d'une exécution SPMD (Single Program Multiple Data, un seul programme avec des données différentes).

Le premier type de code est l'expression la plus simple de la « owner computes rule » sur l'espace d'itération entier. Pour chaque élément de calcul, le code teste si l'élément de donnée modifié par le calcul est possédé par le processus qui exécute (avec la fonction est_local()). Le code parallèle correspondant est le suivant :

```
do i = 1, n
    if (est_local(da(i)) envoyer(da(i))              (* s1 *)
    if (est_local(ga(i)) ga(i) = recevoir(da(i))     (* s2 *)

    if (est_local(db(i)) envoyer(db(i))              (* s3 *)
    if (est_local(gb(i)) gb(i) = recevoir(db(i))     (* s4 *)
enddo
```

La fonction est_local(d) teste si la donnée d est présente sur un processeur (une donnée est sur un seul processeur). La fonction envoyer(d) envoie une donnée d au processeur qui en a besoin. La fonction recevoir(d) reçoit une donnée d d'un processeur.

La première instruction s_1 veut dire que le processus qui possède la donnée $da(i)$ va l'envoyer au processus qui en a besoin. Le principe est le même pour l'instruction s_3. Les instructions s_2 et s_4 consistent pour les processus concernés en une réception de la donnée suivie de l'affectation de cette donnée à l'emplacement mémoire correspondant. La première remarque est que ce code ne sera pas du tout efficace puisque l'espace d'itération entier est couvert par chacun des processus. Pour rendre ce code plus efficace, les communications peuvent être sorties de la boucle si les accès aux tableaux sont connus au moment de la compilation. Néanmoins, il reste toujours le problème de la couverture de l'espace d'itération.

Si le code en entrée est assez simple et que le compilateur est assez intelligent, une deuxième approche est possible dans laquelle chaque processus ne va traiter que la ou les parties contiguës de chaque tableau qu'il possède, comme dans le code suivant :

```
(* Communication : récupérer les données de da *)
(* Communication : récupérer les données de db *)
do i = début_de_ma_partie_de_ga, fin_de_ma_partie_de_ga
    ga(i) = da(i)
enddo
do i = début_de_ma_partie_de_gb, fin_de_ma_partie_de_gb
    gb(i) = db(i)
enddo
```

La *distribution* de la boucle n'est pas forcément nécessaire si les deux tableaux sont distribués de la même façon sur les processus. Ce code permet de ne plus traiter tout l'espace d'itération et de faire toutes les communications à l'extérieur de la boucle.

Considérons maintenant le cas où le code est légèrement plus compliqué, par exemple en remplaçant la deuxième instruction de notre code exemple, $gb(i) = db(i)$, par l'instruction $gb(i-1) = ga(i)$. Avec cette modification, nous ne pouvons plus effectuer toutes les communications à l'extérieur de la boucle car une donnée calculée à l'intérieur de la boucle est utilisée par l'instruction suivante. Nous obtenons le code suivant :

```
do i = 1, N
   if (est_local(da(i)) envoyer(da(i))
   if (est_local(ga(i)) then
      ga(i) = recevoir(da(i))
      send(ga(i))
   endif
   if (est_local(gb(i − 1)) gb(i − 1) = recevoir(ga(i))
enddo
```

Une des possibilités d'amélioration du code est de distribuer la boucle pour obtenir le schéma général : une boucle parallèle, une communication/synchronisation globale, une boucle parallèle. Une autre possibilité est de choisir un bon placement des données tel que $ga(i)$ et $gb(i-1)$ soient sur le même processus, ce qui permettrait d'effectuer les deux calculs sur le même processus. Dans ce cas précis, le code suivant peut être généré :

```
(* Communication : récupérer les données de da *)
do i = début_de_ma_partie_de_ga, fin_de_ma_partie_de_ga
   ga(i) = da(i)
   gb(i − 1) = ga(i)
enddo
```

Si ga et gb ne sont pas plaçés de la même façon, il y a une dernière possibilité offerte par la norme HPF2 [31] qui permet de modifier la « owner computes rule », la directive ON HOME. Sur l'exemple précédent, nous pouvons forcer le compilateur à effectuer les calculs sur le même processeur, en générant des tableaux temporaires ou en dupliquant certains calculs. Ce type de code est plus difficile à produire car le compilateur doit avoir une connaissance très précise des accès aux tableaux pour générer les communications.

Comme nous venons de le voir, générer du code pour des boucles parallèles avec des dépendances entre les instructions dans la boucle est assez difficile. Nous pourrions penser que dans ce cas le compilateur devrait toujours générer plusieurs boucles avec un petit nombre d'instructions, avec une synchronisation entre chaque boucle, en d'autres termes, que le compilateur devrait toujours distribuer la boucle plutôt que de la laisser fusionnée. Ce n'est malheureusement pas aussi simple et nous verrons dans un chapitre ultérieur (chapitre 4) les avantages et les inconvénients respectifs de la fusion et de la distribution de boucle. En fait, nous avons présenté tous ces exemples de codes parallèles effectifs pour illustrer le fait qu'il est difficile pour un compilateur de produire un code parallèle efficace. Cet aspect fondamental de la parallélisation automatique doit être pris en compte : détecter des boucles parallèles n'est pas suffisant pour générer du code exécutable efficace.

Pour toute transformation de boucle, il y a un compromis à trouver entre différents facteurs (communications, calculs, hiérarchie mémoire, type de processeur, etc...) et il n'est pas toujours facile de trouver le meilleur compromis. Comme nous l'avons présenté dans notre introduction, notre approche de la parallélisation par insertion de directives nous permet d'une part de travailler à un haut niveau, mais nous permet également de pouvoir adapter la transformation de code à la machine cible, soit par l'intermédiaire de directives appropriées, soit par l'intermédiaire de regroupements appropriés des instructions au sein des boucles appropriés. Dans toutes les transformations que nous allons voir dans la suite, un critère important va être la flexibilité de l'algorithme utilisé, pour permettre d'obtenir un code efficace par rapport à la machine. Nous ne parlerons pas dans cette thèse de la recherche de ce compromis : notre approche va simplement consister à ne pas effectuer d'opérations trop « destructrices » sur le code, c'est-à-dire empêchant les optimisations ultérieures plus dépendantes de la machine.

2.4 Dépendances de contrôle

Dans cette partie, nous allons présenter les dépendances de contrôle dont nous aurons besoin au chapitre 4. Ces dépendances représentent les contraintes liées à la structure du programme, c'est-à-dire à la façon dont les différentes structures de contrôle sont imbriquées.

2.4.1 Graphe de flot de contrôle

Le *flot de contrôle* [28] d'un programme est une suite possible d'exécution séquentielle des instructions du programme. Un *bloc de base* est une séquence à sortie unique de zéro ou plus instructions sans instruction de branchement, à part peut être la dernière, et pas de cible de branchement (de cible d'instruction `goto`), exceptée peut être la première instruction. Si le flot de contrôle du programme entre dans un bloc de base, il va exécuter toutes les instructions du bloc sans s'arrêter. Une instruction peut être considérée comme un bloc de base.

Le *graphe de flot de contrôle* (CFG) est un graphe orienté qui capture le flot de contrôle du programme. Chaque sommet de ce graphe est un bloc de base et chaque arc représente un flot de contrôle potentiel entre deux blocs (il y a un arc entre le bloc B_1 et le bloc B_2 si B_2 peut immédiatement suivre B_1 lors d'une séquence d'exécution).

Le graphe de flot de contrôle permet d'effectuer diverses analyses. Il permet par exemple de déterminer la structure du programme, de mettre en avant des portions de code jamais exécutées dynamiquement (« code mort »), etc... Il permet également de construire le *graphe des dépendances de contrôle*, comme nous le verrons dans un prochain paragraphe. Une des caractéristiques principale du CFG est la présence d'un sommet source unique appelé DÉBUT et qui permet d'atteindre tous les sommets du graphe et qui incarne une condition déterminant le début de l'exécution du programme.

Définition 5 (graphe de flot de contrôle) *Un graphe de flot de contrôle (CFG) est un graphe orienté $G = (V, E)$ qui a un unique sommet source appelé* DÉBUT *et un unique sommet de sortie appelé* FIN. *Pour tout sommet $s \in V$, il existe un chemin orienté depuis* DÉBUT *jusqu'à s et un chemin orienté de s jusqu'à* FIN.

Certains sommets qui ont au moins deux successeurs se verront attribuer des étiquettes sur leurs arcs sortants. Dans le cas d'une instruction conditionnelle `if`, nous aurons deux étiquettes V (vrai) et F (faux) associées aux arcs sortants pour différencier les deux flots de contrôle associés à l'évaluation de la condition. Dans le cas d'une boucle `do`, un des deux arcs sortants se verra attribuer une étiquette V (vrai) pour signifier que la condition déterminant s'il y a encore des itérations à effectuer est vraie. L'autre arc n'a pas d'étiquette puisque nous partons du principe que cet arc sera toujours emprunté : nous ne prenons pas en considération les boucles infinies, bien que cela soit à priori possible en ajoutant un arc ayant une étiquette F. Si nous faisions ce choix, il s'ensuivrait qu'une instruction suivant une boucle serait toujours dépendante de celle-ci en terme de contrôle.

Nous nous ramènerons toujours au cas où nous n'avons pas plus de deux successeurs pour un sommet. Il est toujours possible de se ramener à ce cas : le seul sommet pouvant à priori avoir plusieurs arcs sortants est un sommet représentant une structure conditionnelle à choix multiples (un switch par exemple) qui peut toujours se ramener à une suite d'instructions conditionnelles simples.

Pour illustrer ce que représente un graphe de flot de contrôle, la figure 2.4 présente un morceau de programme ainsi que son graphe de flot de contrôle associé. L'arc allant de DÉBUT à s_0 représente la condition de départ du programme. L'arc allant de DÉBUT à FIN représente la non exécution du programme. Les sommets rectangulaires sont des structures de contrôle alors que les sommets ronds sont des actions. Nous prenons comme unité de bloc de base une instruction. Nous pouvons remarquer les deux arcs sortant d'un sommet représentant une boucle. L'arc n'ayant pas d'étiquette modélise la fin de la boucle.

```
(* DÉBUT *)
b(2,2) = 3                                (* s0 *)
do i = 2,n                                (* do1 *)
   do j = 2,n-1                           (* do2 *)
      if (a(i,j+1) == 0) then             (* if1 *)
         a(i,j) = b(i,j) + c              (* s1 *)
         d(i,j) = u(i,j) × e              (* s2 *)
      else
         d(i,j) = a(i,j+1) + 1            (* s3 *)
      endif
   enddo
   do j = 2,n-1                           (* do3 *)
      f(i,j) = f(i-1,j) + f(i,j+1) × 2    (* s4 *)
   enddo
enddo
(* FIN *)
```

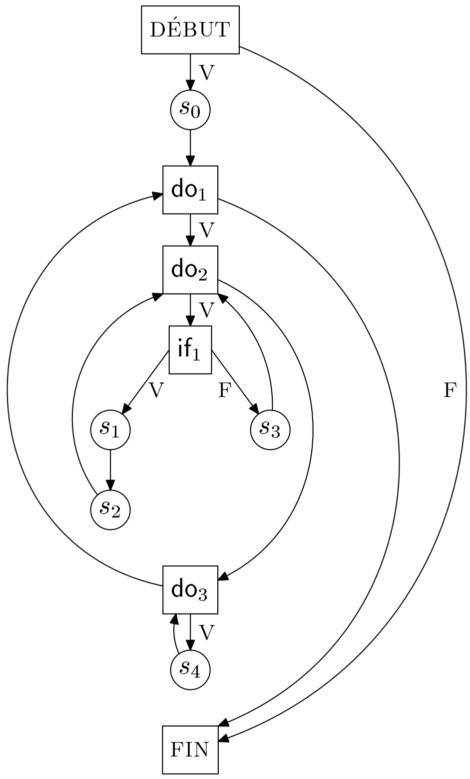


Figure 2.4: un exemple de graphe de flot de contrôle représentant les divers flots d'exécution possibles d'un morceau de programme.

2.4.2 Arbre de post-dominance

La notion de *dominance* dans un CFG exprime le fait qu'un bloc est dominé par un autre, c'est-à-dire que chaque chemin orienté qui arrive sur ce bloc passe par cet autre bloc. Concrètement, cela veut dire que si un bloc est exécuté, c'est que l'autre l'a été précédemment. La notion de dominance est présentée dans le livre de Aho, Sethi et Ullman [1]. De manière inverse, on peut définir la notion de *post-dominance*.

Définition 6 (Post-dominance) *Étant donné un graphe $G = (V, E)$ et deux sommets M et N (avec M différent de N), N est* post-dominé *par M si chaque chemin de N à* FIN *contient M.*

Si un bloc A est post-dominé par un bloc B, cela veut dire que si le bloc A est exécuté, le bloc B va forcément l'être. La définition 6 permet de construire un *arbre de post-dominance* d'un graphe de flot de contrôle. Cet arbre de post-dominance permet d'avoir une représentation canonique de toutes les relations de post-dominance. Par exemple, en considérant l'arbre de post-dominance de la figure 2.5, issu du graphe de flot de contrôle de la figure 2.4, nous pouvons remarquer que pour atteindre la fin du programme (le sommet FIN) depuis l'instruction s_1, le flot de contrôle va passer par les instructions s_2, do_2, do_3 et do_1.

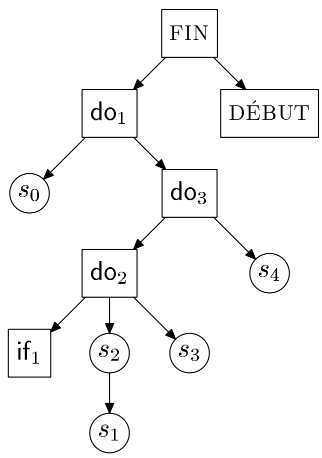


Figure 2.5: arbre de post-dominance correspondant au graphe de flot de contrôle (b) de la figure 2.4.

Le calcul des post-dominants dans un graphe de flot de contrôle est équivalent au calcul des dominants dans ce graphe inversé. La méthode de Lengauer et Tarjan [39] permet de faire ce calcul. L'arbre de post-dominance d'un graphe de flot de contrôle $G = (V, E)$ peut être construit en temps $O(|V|\alpha(|V|))$, comme expliqué dans l'article de Ferrante *et al.* [28].

2.4.3 Graphe des dépendances de contrôle

Quand l'exécution d'une instruction est une conséquence du flot de contrôle, nous dirons qu'il existe une *dépendance de contrôle*. Les relations de dépendances de contrôle capturent les conditions qui contrôlent l'exécution des instructions du programme. Par exemple, dans le fragment de programme suivant :

```
if (A) then
    c = c + d    (* s1 *)
endif
```

s_1 dépend du prédicat A puisque la valeur de A détermine si s_1 est exécutée ou non.

Définition 7 (Dépendance de contrôle) *Soit $G = (V, E)$ un graphe de flot de contrôle et $X, Y \in V$. Il y a une dépendance de contrôle de X vers Y si et seulement si :*

1. *il existe un chemin orienté P de X vers Y tel que tout Z appartenant à P (différent de X et Y) soit post-dominé par Y ;*
2. *X n'est pas post-dominé par Y.*

Une description moins formelle d'une dépendance de contrôle est de dire que les conditions nécessaires et suffisantes pour que Y soit dépendant de X en terme de contrôle sont que X doit avoir au moins deux successeurs dans le graphe de flot de contrôle et que le fait de suivre un de ces successeurs exécute toujours Y, tandis que suivre un autre successeur peut avoir comme résultat que Y ne soit pas exécuté. La condition 1 de la définition 7 peut être satisfaite par un chemin constitué d'un seul sommet et la condition 2 est toujours satisfaite quand X et Y sont le même sommet : ceci permet de modéliser correctement les boucles.

En se fondant sur cette définition, un *graphe des dépendances de contrôle* (CDG) peut être construit, les arcs étant étiquetés par l'étiquette du premier arc sur le chemin orienté P. La construction de ce graphe que nous présentons dans la suite est issue de l'article de Ferrante *et al.* [28].

La première étape pour déterminer les dépendances de contrôle est la construction de l'arbre de post-dominance depuis le graphe de flot de contrôle. Considérant ces deux graphes, soit S l'ensemble des arcs (A, B) du graphe de flot de contrôle (avec A et B des sommets) tels que B n'est pas un ancêtre de A dans l'arbre de post-dominance (c'est-à-dire que B ne post-domine pas A). Ces arcs sont ceux qui satisfont la condition 2 de la définition 7. Chacun de ces arcs a une étiquette puisque sinon B post-dominerait A. Nous trouverons par la suite les couples (A, B) tels que les deux conditions de la définition 7 soient vraies et tels qu'il n'y ait pas d'arc entre A et B dans le CFG. Les graphes des figures 2.4 et 2.5 permettent de construire S tel que

$$S = \{(\text{DÉBUT}, s_0), (\mathsf{do}_1, \mathsf{do}_2), (\mathsf{do}_2, \mathsf{if}_1), (\mathsf{if}_1, s_1), (\mathsf{if}_1, s_3), (\mathsf{do}_3, s_4)\}$$

La suite de la recherche des dépendances de contrôle consiste à examiner chaque arc (A, B) de S afin de déterminer s'il satisfait la condition 1 de la définition 7. Soit L le premier ancêtre commun de A et B dans l'arbre de post-dominance (lorsque l'on remonte dans l'arbre depuis A ou B). Par construction, L est différent de B (B n'est pas un ancêtre de A dans l'arbre de post-dominance).

Proposition 1 *L est le sommet A ou bien L est le père direct de A dans l'arbre de post-dominance.*

Preuve : appelons X le père de A dans l'arbre de post-dominance (par construction, X est différent de B). Montrons par contradiction que X post-domine B. Supposons que X ne post-domine pas B : nous aurions alors un chemin de B vers FIN qui ne passerait pas par X. Ajouter l'arc (A, B) à ce chemin nous donnerait un chemin qui va de A à FIN sans passer par X : ceci contredit le fait que X post-domine A. Sachant donc que X post-domine B, il doit être un ancêtre de B dans l'arbre de post-dominance. Comme X est le post-dominant immédiat de A, alors le premier ancêtre commun de A et B dans l'arbre de post-dominance est soit X soit A lui-même. □

Avec la propriété précédente, nous pouvons à présent énoncer deux cas qui permettent de dériver des conditions selon lesquelles un sommet est dépendant de A en terme de contrôle. Nous verrons que ces deux cas sont corrects et permettent bien de trouver toutes les dépendances de contrôle.

Cas 1. L est le père de A. Tous les sommets de l'arbre de post-dominance sur le chemin de L à B, incluant B mais pas L sont rendus dépendants de A en terme de contrôle.

Cas 2. L est le sommet A. Tous les sommets de l'arbre de post-dominance sur le chemin de A à B, incluant A et B, sont rendus dépendants de A en terme de contrôle (ce cas capture les dépendances de contrôle d'une boucle).

Étant donné un arc (A, B), une méthode pour utiliser ces deux cas et construire le graphe de contrôle est de faire un parcours arrière dans l'arbre de post-dominance depuis B jusqu'au sommet avant le père de A s'il existe ou jusqu'à A, en marquant chaque sommet recontré comme étant dépendant de A en terme de contrôle. L'examen complet de tous les arcs de S permet de déterminer toutes les dépendances de contrôle du code représenté par le CFG. Chaque dépendance est étiquetée par l'éventuelle étiquette de l'arc de S qui a provoqué l'ajout de cette dépendance. La table suivante donne les dépendances de contrôle découvertes par cette méthode en utilisant les graphes des figures 2.4 et 2.5. Le graphe de la figure 2.6 donne le graphe de contrôle correspondant, où nous avons choisi de représenter les arcs des dépendances de contrôle en pointillés.

Arc de S	Étiquette	Sommets	Dépendant de
(DÉBUT, s_0)	V	s_0, do_1	DÉBUT
(do_1, do_2)	V	do_1, do_2, do_3	do_1
(do_2, if_1)	V	do_2, if_1	do_2
(if_1, s_1)	V	s_1, s_2	if_1
(if_1, s_3)	F	s_3	if_1
(do_3, s_4)	V	do_3, s_4	do_3

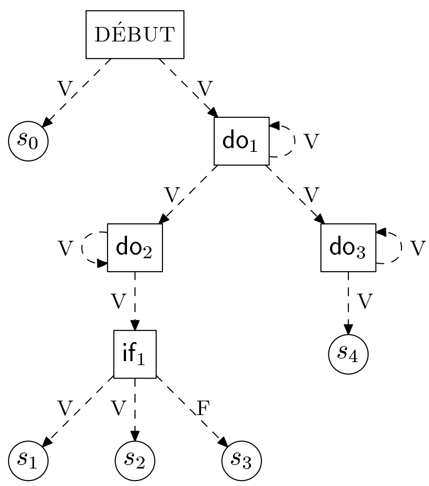


Figure 2.6: graphe des dépendances de contrôle correspondant au graphe de flot de contrôle de la figure 2.4.

Pour simplifier les graphes, nous omettrons volontairement dans la suite de ce document les arcs représentant une dépendance de contrôle d'une boucle vers elle-même. Nous omettrons également le label V sur les arcs qui sortent d'un do.

La correction de cette construction découle directement de la définition. Dans l'article de Ferrante *et al.* [28], il est montré que la construction du graphe de contrôle prend un temps en $O(|V^2|)$ pour un graphe de flot de contrôle $G = (V, E)$ et que toutes les dépendances de contrôle sont trouvées. Pour finir, voici la définition d'un graphe des dépendances de contrôle d'une portion de programme.

Définition 8 (Graphe des dépendances de contrôle) *Un graphe des dépendances de contrôle $G = (V, E)$ d'une portion de programme est un graphe orienté avec V l'ensemble des sommets représentant les instructions de la portion de programme et E l'ensemble des arcs représentant les dépendances de contrôle entre les instructions. Chaque arc de E a une étiquette qui appartient à l'ensemble* (V, F)*. Ce graphe a un unique sommet source appelé* DÉBUT *tel que pour tout sommet s appartenant à V différent de* DÉBUT*, il y a un chemin de* DÉBUT *vers s.*

2.5 Dépendances de données

En dehors de l'ordre d'exécution standard, d'autres ordres d'exécution des opérations peuvent aboutir au même résultat final. Une transformation de boucle visant l'optimisation d'un certain critère va chercher à modifier l'ordre d'exécution standard pour satisfaire au mieux ce critère en respectant des contraintes qui sont les *dépendances de donnée* entre les opérations.

Deux opérations peuvent être exécutées en parallèle si elles n'accèdent pas au même emplacement mémoire [7] ou qu'aucun de ces accès n'est une écriture. Dans ce cas, les deux opérations sont *indépendantes*, sinon il y a une relation de dépendance de donnée entre les deux opérations qui doit être respectée par toute transformation. Toutes les relations de dépendance de donnée existant dans un programme spécifient les contraintes sur l'ordre d'exécution des opérations qui permettent de conserver la sémantique originelle du programme.

Lorsqu'une opération x écrit dans un emplacement mémoire puis qu'une opération y lit cet emplacement mémoire, nous dirons qu'il y a une *dépendance de flot* entre x et y (notée f). Lorsqu'une opération x lit un emplacement mémoire puis qu'une opération y écrit dans cet emplacement, nous dirons qu'il y a une *anti-dépendance* entre x et y (notée a). Lorsqu'une opération x écrit dans un emplacement mémoire puis que x écrit dans ce même emplacement, nous dirons qu'il y a une *dépendance de sortie* entre x et y (notée o). Dans ces trois cas, nous dirons que x est la *source* de la dépendance et que y en est le *puits*.

2.5.1 Graphe des dépendances de données

Nous écrirons $s_i(\vec{p}) \Rightarrow s_j(\vec{q})$ si l'opération $s_j(\vec{q})$ dépend de l'opération $s_i(\vec{p})$. Notons que $\vec{q} - \vec{p}$ (sur la partie commune des vecteurs) est toujours lexicographiquement positif ou nul quand $s_i(\vec{p}) \Rightarrow s_j(\vec{q})$. L'exécution séquentielle fournit un sur-ordre total de $\Rightarrow$, ordre en général partiel qu'il va falloir respecter dans les transformations. Cet ordre peut être représenté par un graphe où les arcs représentent les dépendances et où il y a un sommet par opération. Nous appelons ce graphe le *graphe des dépendances de donnée étendu* (EDG). La définition exhaustive de toutes les dépendances d'un nid de boucles par un EDG peut être extrêmement grosse puisqu'il y a autant de sommets que d'opérations. Un autre graphe résumé peut être défini, où chaque sommet est cette fois-ci une *instruction* et où il existe un arc entre deux sommets s_i et s_j si et seulement s'il existe deux opérations telles que $s_i(\vec{p}) \Rightarrow s_j(\vec{q})$. Si tel est le cas, nous noterons la dépendance de la façon suivante : $s_i \rightarrow s_j$. Nous appelons ce graphe le *graphe des dépendances de donnée réduit* (RDG).

L'intérêt principal d'un graphe des dépendances de donnée réduit par rapport à un graphe des dépendances de donnée étendu est que le premier permet d'effectuer des transformations de manière symbolique, c'est-à-dire sans avoir à évaluer les bornes de boucle ni à « dérouler » la boucle. En effet, la construction d'un graphe des dépendances de donnée étendu fini nécessite la connaissance du nombre d'opérations qui vont être effectuées, ce qui n'est pas toujours possible au moment de l'exécution. Un graphe des dépendances de donnée réduit peut être cyclique alors que le graphe des dépendances de donnée étendu est acyclique par définition. Ces deux approches ont donné lieu à des recherches dans deux directions différentes qui ne se sont pas encore vraiment rejointes.

Un arc entre s_i et s_j sera pondéré par l'ensemble des paires $(\vec{p}, \vec{q}) \in P^2$ telles que $s_i(\vec{p}) \Rightarrow s_j(\vec{q})$, ou tout du moins par une approximation de cet ensemble que l'on notera D. L'exemple 2.3 montre un de nid de boucles avec ses dépendances. Voici la définition d'un graphe des dépendances de données réduit.

Définition 9 (Graphe des dépendances de données réduit (RDG)) *Un* graphe des dépendances de données réduit *d'une portion de programme est un graphe orienté $G = (V, E)$ tel que V est l'ensemble des sommets représentant les instructions de la portion de programme et E l'ensemble des arcs représentant les dépendances de donnée entre les instructions. Chaque arc de E est étiqueté par un ensemble D représentant une approximation de la dépendance.*

```
do i1 = 1, N
   do i2 = 1, N
      a(i1, i2) = a(i2, i1) + a(i1, i2 - 1)   (* s1 *)
   enddo
enddo
```

$$
\begin{aligned}
P &= \{(i_1, i_2) \in \mathbb{Z}^2 \mid 1 \leq i_1, i_2 \leq N\} \\
d_1 &: s_1(i_1, i_2) \rightarrow s_1(i_1, i_2 + 1) \\
d_2 &: s_1(i_1, i_2) \rightarrow s_1(i_2, i_1)
\end{aligned}
$$

Exemple 2.3: nid de boucles avec ses dépendances.

Il est important de noter que pour une portion de programme donnée, hormis le sommet DÉBUT du graphe de contrôle, il y a le même nombre de sommets dans le graphe des dépendances de contrôle et le graphe des dépendances de donnée. Dans leur article de 1987, Ferrante *et al.* présentent un graphe qui regroupe les graphes des dépendances de contrôle et de dépendances de donnée en un unique graphe appelé le *graphe des dépendances de programme* (PDG, Program Dependence Graph). Ce graphe permet de n'utiliser qu'un seul graphe pour effectuer des transformations sur un programme puisqu'il contient toutes les informations utiles sur les dépendances à respecter. Dans la suite, nous utiliserons une version simplifiée de ce graphe que nous appellerons tout simplement le *graphe des dépendances*. Le graphe original tel qu'il est défini dans leur article comporte un certain nombre de sommets qui résument l'information de contrôle, ce que nous n'utiliserons pas dans ce document. Nous présentons à la figure 2.7 un exemple de graphe des dépendances pour le graphe de flot de contrôle de la figure 2.4. Les arcs en pointillés sont les dépendances de contrôle alors que les arcs pleins sont les dépendances de donnée. Nous verrons plus loin (§2.5.3, p. 22) la signification de l'étiquette de ces arcs.

Nous allons introduire plus loin (chapitre 4) un nouveau type de graphe, le *graphe des dépendances mixte* qui ne comportera qu'un nombre de sommets égal au nombre d'actions

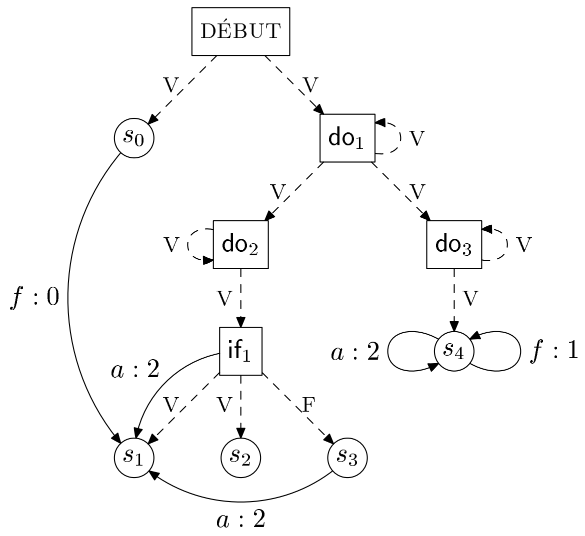


Figure 2.7: graphe des dépendances correspondant au graphe de flot de contrôle de la figure 2.4.

dans la portion de programme et qui offrira un résumé des dépendances à respecter pour s'assurer de la validité d'une transformation.

2.5.2 Représentation des dépendances de données

Nous allons nous intéresser dans cette partie aux différentes approximations des dépendances utilisées pour étiqueter les arcs d'un RDG. L'abstraction des dépendances la plus précise consiste en une énumération exhaustive des paires d'itérations $(\vec{p}, \vec{q})$ de $s_i(\vec{p})$ et $s_j(\vec{q})$ dépendantes. Appelons cet ensemble DI.

Définition 10 (Itérations de dépendances)

$$DI = \{(\vec{p}, \vec{q}) \mid s_i(\vec{p}) \Rightarrow s_j(\vec{q})\}$$

Dans quelques cas précis de nids de boucles, il est possible de représenter précisément l'ensemble DI [25], quand $\vec{p}$ peut être exprimé par l'intermédiaire d'une fonction affine $f_{i,j}$ de $\vec{p}$ où $\vec{q}$ varie dans un polyèdre $P_{i,j} \subset P$:

$$\{(\vec{p}, \vec{q}) \in P^2 \mid s_i(\vec{p}) \Rightarrow s_j(\vec{q})\} = \{(f_{i,j}(\vec{q}), \vec{q}) \in P^2 \mid \vec{q} \in P_{i,j} \subset P\}$$

Une analyse précise de cet ensemble peut être trop coûteuse, lorsqu'elle est possible. On change donc la représentation et la précision de cet ensemble, deux caractéristiques qui expriment la puissance d'une analyse de dépendances et qui conditionnent par là-même la puissance des transformations possibles. Au lieu de représenter exactement toutes les paires $(\vec{p}, \vec{q}) \in P^2$ induisant une dépendance, on exprime l'ensemble D des valeurs $\vec{d} = \vec{q} - \vec{p} \in P$ (sur la partie commune des vecteurs) que l'on appellera l'ensemble des *vecteurs de distance* ou encore l'ensemble des *vecteurs de dépendance* tellement cette représentation est usitée dans la littérature. L'utilisation d'un vecteur de distance $\vec{d}$ permet de représenter toutes les dépendances entre les itérations $\vec{p}$ et $(\vec{p} + \vec{d})$. Elle reflète explicitement le pas de dépendance entre les itérations mais ne tient pas compte de l'origine des dépendances.

Définition 11 (Vecteurs de distance)

$$D = \{\vec{d} \mid \exists(\vec{p}, \vec{q}) \in DI, \vec{d} = \vec{q} - \vec{p}\}$$

En ce qui concerne la précision de D, diverses approximations sont possibles, notamment par vecteurs de direction, niveaux de dépendance ou encore polyèdre, ce que nous allons voir dans le paragraphe suivant.

2.5.3 Précision des ensembles de vecteurs de distance

Dans le cadre de la représentation des dépendances par un ensemble de vecteurs de distance, divers degrés de précision sont possibles. En voici une liste non exhaustive classée par degré de précision décroissante.

Polyèdre On peut exprimer un ensemble de vecteurs de distance par un *polyèdre de dépendance* défini par ses sommets et ses rayons [32]. L'ensemble total des vecteurs de distance de toutes les instructions sera donc une union finie de polyèdres. Un polyèdre de dépendance sans sommets est appelé un *cône de dépendance*. Dans le cas de l'exemple 2.3, nous aurions les deux dépendances :

$$d_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \text{et} \quad d_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \ \lambda \geq 0$$

La dépendance d_2 est un polyèdre avec un sommet et un rayon.

Vecteurs de direction Il se peut que l'ensemble des vecteurs de distance se réduise à un singleton. Dans ce cas, on parle de *dépendance uniforme*. Sinon, on peut toujours exprimer cet ensemble de vecteurs par un seul vecteur dont les composantes appartiennent à $\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+, -\})$. Dans ce cas, on parlera de *vecteur de direction*. La composante d_i d'un tel vecteur est une approximation de toutes les composantes possibles des vecteurs de distance de l'ensemble. Elle est égale à $z+$ (resp. $z-$) si toutes les composantes d_i des vecteurs de distances sont supérieure à z (resp. inférieures), à $*$ si la composante peut prendre n'importe quelle valeur et à z si la dépendance est uniforme à ce niveau et a comme valeur unique z. $+$ est une abréviation de $1+$ et $-$ une abréviation de $(-1)-$. Ce type de précision a été introduit par Wolfe [74] et est le degré de précision maximal supporté par l'algorithme de parallélisation de Wolf et Lam présenté dans [73]. Avec l'exemple 2.3, nous aurions :

$$d_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \text{et} \quad d_2 = \begin{pmatrix} + \\ - \end{pmatrix}$$

Niveau de dépendance L'ensemble des vecteurs de distance est approximé par un entier $p \in [\![0, n]\!]$ défini comme étant le plus grand entier tel que les $p-1$ premières composantes du vecteur de distance soient égales à 0 (en fait, p est le numéro de la boucle qui crée la dépendance). Si $p \leq n$, cela signifie que la dépendance est *portée* par la boucle p du nid, c'est-à-dire à une itération donnée des $p-1$ boucles externes. Si $p = 0$, la dépendance se trouve au plus profond du nid de boucles parfait mais entre deux instructions différentes. Ce type de dépendance a été introduit par Allen et Kennedy [3, 2] dans le cadre de leur algorithme de parallélisation. Avec l'exemple 2.3, nous aurions :

$$d_1 = 2 \quad \text{et} \quad d_2 = 1$$

Pour plus de détail sur toutes ces différentes précisions et les différents algorithmes employés pour trouver les dépendances, se référer à [79].

Dans ce qui suit, nous allons parler plus en détail d'un graphe de dépendances réduit particulier, le *graphe de dépendance réduit polyédrique*. Cette représentation généralise la plupart des représentations de dépendance utilisées dans la littérature sans avoir le coût d'une recherche des dépendances exactes.

2.5.4 Graphe des dépendances réduit polyédrique

Rappelons tout d'abord la définition mathématique d'un polyèdre et quelques précisions sur ceux-ci [56].

Définition 12 (Polyèdre, polytope) *Un ensemble DP de vecteurs de $\mathbb{Q}^n$ est appelé polyèdre (convexe) s'il existe une matrice entière A et un vecteur entier b tel que :*

$$DP = \{x \mid x \in \mathbb{Q}^n,\ Ax \leq b\}$$

Un polytope est un polyèdre borné.

Un polyèdre peut toujours être décomposé en la somme d'un polytope convexe et d'un cône polyédrique (un ouvrage de référence sur le sujet est le livre de Schrijver [56]). Un polytope est défini par ses sommets et chaque point du polytope par une combinaison barycentrique positive de ses sommets. Un cône polyédrique est généré de façon finie et peut être défini par ses rayons et ses droites. Tout point du cône est la somme d'une combinaison positive de ses rayons et de ses droites. Ainsi, un polyèdre convexe DP peut être défini de manière équivalente à la définition 12 par un *système générateur* qui correspond à trois ensembles finis : sommets $\{\vec{s_i}\}$ de cardinal ω, rayons $\{\vec{r_j}\}$ de cardinal ρ et droites $\{\vec{d_k}\}$ de cardinal θ. Ainsi, DP est l'ensemble des vecteurs $\vec{p} \in \mathbb{Q}^n$ tels que :

$$\vec{p} = \sum_{i=1}^{\omega} \lambda_i \vec{s_i} + \sum_{j=1}^{\rho} \mu_j \vec{r_j} + \sum_{k=1}^{\theta} \nu_k \vec{d_k} \text{ avec } (\lambda, \mu, \nu) \in (\mathbb{Q}^+)^\omega \times (\mathbb{Q}^+)^\rho \times \mathbb{Q}^\theta \text{ et } \sum_{i=1}^{\omega} \lambda_i = 1$$

Un algorithme exposé par Chernikova [15] permet de passer d'une représentation à l'autre. En fait, nous allons nous intéresser uniquement aux points entiers du polyèdre de dépendance, puisque les vecteurs de dépendances sont entiers.

Définition 13 (Graphe des dépendances réduit polyédrique (PRDG)) *Un* graphe des dépendances réduit polyédrique *est un RDG, c'est-à-dire un graphe avec un sommet par instruction du nid de boucles et avec un arc du sommet s_i vers le sommet s_j s'il existe au moins une paire $(\vec{p}, \vec{q}) \in P^2$ telle que $s_i(\vec{p}) \Rightarrow s_j(\vec{q})$. Chaque arc e du PRDG est pondéré par un polyèdre $P(e)$ qui approxime l'ensemble des vecteurs de distance $(\vec{q} - \vec{p})$ tels que $s_i(\vec{p}) \Rightarrow s_j(\vec{q})$.*

La puissance d'approximation d'une représentation des ensembles de vecteurs de distance par un polyèdre, spécifié par un système générateur, permet d'englober les approximations par vecteurs de direction (DV) et niveaux de dépendance (DL). En fait, il y a la hiérarchie suivante au niveau de la précision [24, 79] :

$$\perp \subset DL \subset DV \subset DP \subset D \subset DI$$

où $\perp$ est une information nulle.

Un PRDG peut être capturé dans une structure équivalente mais plus simple à manipuler, la structure de graphe de dépendance uniforme (où chaque arc est pondéré par un vecteur de distance uniforme). L'uniformisation est effectuée à l'aide de l'algorithme présenté par Darte et Vivien [24]. Essayons d'en expliquer rapidement le principe. Sur la figure 2.8, le graphe de dépendance du haut a un arc pondéré par un polyèdre $p(e)$ comportant un sommet v et un rayon r. Par définition, $p(e) = \{v + \lambda r,\ \lambda \geq 0\}$. Si nous considérons l'arc e comme un chemin de dépendance $\phi = v + \lambda r$ qui utilise une fois le vecteur uniforme v et λ fois le vecteur r, le principe général de l'algorithme est clair : ce chemin de dépendance est *simulé* en rajoutant un sommet et un certain nombre d'arcs. Par exemple, sur la figure 2.8, le graphe du bas simule le polyèdre $p(e)$ en rajoutant un sommet virtuel, un arc pondéré par 0 entre le sommet virtuel et le sommet destination, un arc pondéré par v entre le sommet source et le sommet virtuel et un arc qui boucle pondéré par r sur le sommet virtuel.

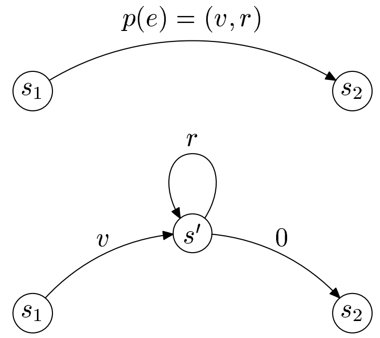


Figure 2.8: exemple d'uniformisation d'un PRDG.

En fait, ce graphe uniformisé décrit un système d'équations récurrentes uniformes (si on ne tient pas compte de la *virtualité* de certains sommets) introduit par Karp, Miller et Winograd [35]. Il y a un certain nombre de différences entre un nid de boucles et un système d'équations récurrentes uniformes (SURE). Dans un nid de boucles, il y a un ordre d'exécution *explicite* qui est en fait l'ordre séquentiel standard et toutes les dépendances sont lexicographiquement positives. Le graphe de dépendance étendu est acyclique et il n'y a pas de cycle de poids nul dans le RDG.Dans un SURE, l'ordre d'exécution est implicite : la partie gauche est évaluée quand la partie droite l'a été. La principale conséquence est qu'un SURE n'est pas toujours calculable (on ne peut pas toujours trouver d'ordonnancement explicite). Un SURE n'est pas calculable quand il existe un cycle de poids nul dans son graphe de dépendances réduit. Trouver un ordre d'exécution nécessite de détecter la présence de cycles de poids nul et de construire un ordonnancement fondé sur une décomposition récursive du graphe de dépendance [21, 24].

2.6 Détection de parallélisme

Toutes les représentations de graphe que nous avons introduites dans ce chapitre vont nous permettre de restructurer le code selon un certain objectif, dans notre cas la détection de parallélisme. Au vu des dépendances de données, nous pourrons déterminer si une boucle peut être exécutée avec la sémantique d'une boucle parallèle avec la définition suivante :

Définition 14 (Boucle $\longrightarrow$ boucle parallèle) *Une boucle peut être exécutée comme une boucle parallèle si et seulement si il n'y a pas de dépendance de donnée portée par cette boucle.*

La définition précédente est bien adaptée à une représentation des vecteurs de distance par un niveau de dépendance. Dans le cas général, la définition suivante peut être utilisée :

Définition 15 *Soit un nid de boucles de profondeur n et d'ensemble de vecteurs de dépendances D. La boucle i est parallèle si et seulement si*

$$\forall \vec{d} \in D,\ (d_1, ..., d_{i-1}) \succ \vec{0} \text{ ou } \forall j,\ 1 <= j <= i,\ d_j = 0$$

Dans les deux prochains chapitres, notre but va être de transformer des boucles afin de détecter du parallélisme. En d'autres termes, nous allons essayer de changer l'ordre dans lequel les actions du programme sont effectuées afin que des dépendances qui étaient portées auparavant par une boucle ne le soient plus, rendant ainsi la boucle parallèle. Il existe de nombreux autres algorithmes de parallélisation (voir le travail de synthèse présenté dans [22] ainsi que la thèse de Frédéric Vivien [69]), mais l'intérêt des deux algorithmes que nous allons présenter est qu'ils offrent la possibilité de gérer finement la mémoire.

Chapitre 3

Boucles complètement permutables

Les techniques d'ordonnancement affine, de la plus simple, la méthode de l'hyperplan de Lamport [38], à la plus compliquée, les ordonnancements multi-dimensionnels affines de Feautrier [27], sont utilisées pour transformer un ensemble de boucles imbriquées en un code sémantiquement équivalent, qui consiste en des boucles parallèles entourées de boucles séquentielles. Nous parlerons dans ce cas de transformations à grain fin et ces différentes méthodes seront présentées dans la première partie de ce chapitre. La méthode de Lamport et son extension, l'ordonnancement linéaire, transforme n boucles uniformes parfaitement imbriquées en $n-1$ boucles parallèles entourées d'une boucle séquentielle. Quand cette méthode n'est pas appliquable, un ordonnancement multi-dimensionnel peut être utilisé pour transformer les boucles originales en $n-r$ boucles séquentielles entourant r boucles parallèles internes, avec $n-r>1$. Le but est de rendre r (le *degré de parallélisme*) aussi grand que possible.

Le modèle de calcul dans lequel ces techniques sont utilisées n'est rien d'autre qu'une PRAM. Les contraintes additionnelles dont nous parlions au chapitre précédent, telles que le coût des communications, le nombre et le type des processeurs, la hiérarchie mémoire, ne sont pas prises en considération. L'espoir est que le code mis sous forme parallèle va pouvoir être optimisé a posteriori, par exemple en regroupant les processus PRAM virtuels sur des processeurs physiques. Pourtant, nous avons vu dans le chapitre précédent que la génération d'un code parallèle exécutable efficace n'était pas une chose facile. Ensuite, spécialement quand r est petit, la granularité des calculs peut être trop petite. Cette granularité était parfaite pour des machines vectorielles ou SIMD, mais pour les machines actuelles à mémoire distribuée ou partagée, nous aurions besoin d'une granularité supérieure pour obtenir des résultats satisfaisants et pour mieux exploiter la hiérarchie mémoire. La granularité peut être augmentée par une technique appelée *tiling*, introduite par Irigoin et Triolet [33] sous le nom de *supernode partitioning*.

Le tiling consiste en une agrégation de plusieurs itérations qui vont être considérées comme un calcul élémentaire. La taille et la forme d'une *tuile* (un regroupement d'itérations) devra être choisie selon différents critères, pour une meilleure utilisation des caches mémoires, pour une meilleure vectorisation des communications et des calculs, pour réduire des communications, etc... Tous ces critères sont dépendants de la machine et bien que de nombreuses stratégies aient déjà été proposées [33, 37, 51, 55, 61, 10], choisir un « bon » regroupement est toujours un problème ouvert.

Cependant, avant de penser à définir la taille et la forme des tuiles, nous devons être sûrs qu'elle seront atomiques, c'est-à-dire qu'elles pourront être calculées sans synchronisation ou communication. La propriété d'atomicité est remplie si le graphe des dépendances de données entre les tuiles est acyclique, ce qui est garanti si les tuiles partitionnent le domaine d'itération en des rectangles identiques et si le domaine d'itération est décrit par des *boucles complètement permutables*. Jusqu'à maintenant, tous les algorithmes proposés pour détecter des boucles complètement permutables ont les restrictions suivantes :

- les boucles doivent être parfaitement imbriquées ;
- les dépendances doivent être uniformes, excepté pour l'algorithme de Wolf et Lam [73] où les dépendances peuvent être approximées par des vecteurs de direction ;
- les instructions du corps de la boucle sont considérées comme une unique instruction au niveau du RDG, ce qui n'autorise pas un simple décalage des instructions même si cela aurait suffit pour rendre les boucles complètement permutables.

Prendre en compte la structure du graphe des dépendances de données réduit est très utile pour la détection de boucles parallèles : voir par exemple les algorithmes d'Allen, Callahan et Kennedy [2] (voir également le chapitre 4), Darte et Vivien [24], ou Feautrier [27]. Dans la troisième partie de ce chapitre, nous allons montrer que ceci est aussi utile pour la détection de boucles complètement permutables. Notre méthode va combiner de la retemporisation de graphe (graph retiming) et de l'ordonnancement de graphe.

Nous n'enlevons pas toutes les restrictions énoncées plus haut puisque nous considérons uniquement les boucles parfaitement imbriquées. Cependant, notre algorithme peut être appliqué même si les dépendances sont décrites par un approximation polyédrique des vecteurs de distance (ce qui est plus général que les vecteurs de direction), et nous exploitons le fait que le corps de la boucle peut contenir plusieurs instructions, c'est-à-dire que le graphe peut avoir plusieurs sommets.

L'organisation de ce chapitre est la suivante. Après avoir parlé des différentes transformations permettant d'obtenir des codes à grain fin dans la partie 3.1, nous présenterons le principe du parallélisme à grain moyen (ou variable) dans la partie 3.2, pour finalement expliquer dans la partie 3.3 notre algorithme qui permet de caractériser l'ensemble des transformations permettant d'obtenir le nombre maximal de boucles complètement permutables [20]. Pour finir, nous parlerons dans la dernière partie de ce chapitre des différents problèmes restant ouverts.

3.1 Transformations à grain fin

Nous allons nous intéresser dans ce chapitre aux transformations de l'ordre d'exécution séquentiel standard d'un nid de boucles. Dans ce qui suit, nous allons considérer le graphe $G = (V, E)$ avec V qui est l'ensemble des arcs et E l'ensemble des sommets, où G est un RDG.

3.1.1 Ordonnancement uni-dimensionnel

Un *ordonnancement uni-dimensionnel* est une application T de $V \times \mathbb{Z}^n \to \mathbb{Z}$ qui donne un ordre d'exécution à chaque $v \in V$ tout en préservant les dépendances. En pratique, on peut rechercher des applications de $V \times \mathbb{Z}^n$ dans $\mathbb{Q}$, puis prendre la partie entière de T, ceci pour simplifier le calcul. Cependant, dans notre cas, nous allons toujours travailler dans $\mathbb{Z}$ pour simplifier les écritures de formule.

Définition 16 (Ordonnancement uni-dimensionnel) *L'application T est un ordonnan-*

cement uni-dimensionnel valide si et seulement si pour tous les vecteurs d'itération $\vec{p}$ et $\vec{q}$ et toutes les instructions v et w,

$$T(v,\vec{p}) \geq T(w,\vec{q}) + 1 \text{ si } w(\vec{q}) \Rightarrow v(\vec{p})$$

Il existe plusieurs classes d'ordonnancement uni-dimensionnel dans la littérature. En voici une liste non exhaustive :

Ordonnancement linéaire. Introduit par Lamport [38]. T est défini par un hyperplan commun à toutes les instructions du nid de boucles.

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \longrightarrow & \mathbb{Z} \\ T : (s,\vec{p}) & \rightarrow & \vec{X}.\vec{p} \text{ où } \vec{X} \in \mathbb{Z}^n \end{array}$$

Ordonnancement linéaire décalé. Dans cette méthode, chaque instruction peut être décalée par rapport aux autres.

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \longrightarrow & \mathbb{Z} \\ T : (s,\vec{p}) & \rightarrow & \vec{X}.\vec{p} + \rho_s \text{ où } \vec{X} \in \mathbb{Z}^n \text{ et } \rho_s \in \mathbb{Z} \end{array}$$

Ordonnancement affine. Chaque instruction peut posséder une partie linéaire différente.

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \longrightarrow & \mathbb{Z} \\ T : (s,\vec{p}) & \rightarrow & \vec{X}_s.\vec{p} + \rho_s \text{ où } \vec{X}_s \in \mathbb{Z}^n \text{ et } \rho_s \in \mathbb{Z} \end{array}$$

Il est important de noter que dans le cas d'un ordonnancement linéaire décalé, les inéquations de la définition 16 se réduisent en :

$$\forall e = (x,y) \in E,\ \vec{X}.\vec{d_e} + \rho_y - \rho_x \geq 1$$

Le vecteur $\vec{X}$ est appelé *hyperplan strictement séparateur*. Quand on a uniquement la relation :

$$\forall e = (x,y) \in E,\ \vec{X}.\vec{d_e} + \rho_y - \rho_x \geq 0$$

on parle pour $\vec{X}$ d'*hyperplan faiblement séparateur* de l'arc e. Ces deux notions vont être importantes dans le cas de notre recherche de boucles complètement permutables.

3.1.2 Ordonnancement multi-dimensionnel

Nous allons rappeler ici la notion d'ordonnancement multi-dimensionnel introduite par Karp et Rao dans [52, 35] dans le contexte des systèmes d'équations récurrentes uniformes (SURE) et par Feautrier dans [25] dans le contexte des nids de boucles.

Un *ordonnancement multi-dimensionnel* de dimension d est une application de $V \times \mathbb{Z}^n$ dans $\mathbb{Z}^d$ qui donne un ordre d'exécution pour chaque opération $s_i(\vec{p})$ et qui préserve les relations de dépendance.

L'ordre d'exécution est un ordre à d dimensions qui est considéré comme l'ordre lexicographique strict $>_l$ (pour correspondre à l'ordre d'exécution de d boucles imbriquées). De manière métaphorique, on peut voir les composantes successives du vecteur d'ordonnancement comme étant des jours, heures, minutes, secondes, etc. Les contraintes à satisfaire pour préserver les relations de dépendances sont les suivantes, T étant la fonction d'ordonnancement :

$$T(s_i, \vec{p}) >_l T(s_j, \vec{q}) \quad \text{si} \quad s_j(\vec{q}) \Rightarrow s_i(\vec{p}) \tag{3.1}$$

qui peuvent se reformuler de la façon suivante :

$$\exists k,\ 1 \leq k \leq d,\ \forall l,\ l < k,\ [T(s_i, \vec{p})]_l = [T(s_j, \vec{q})]_l \text{ et } [T(s_i, \vec{p})]_k \geq [T(s_j, \vec{q})]_k + 1. \tag{3.2}$$

Le cas particulier des ordonnancements multidimensionnels que nous allons considérer ici est la classe des *ordonnancements multi-dimensionnels affines*, définis de la manière suivante :

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \longrightarrow & \mathbb{Z}^d \\ T : (s_l, \vec{p}) & \rightarrow & (\vec{X}^1_{s_l}.\vec{p} + \rho^1_{s_l},\ ...,\ \vec{X}^d_{s_l}.\vec{p} + \rho^d_{s_l}),\ \text{avec } \vec{X}^i_{s_l} \in \mathbb{Z}^n \text{ et } \rho^i_{s_l} \in \mathbb{Z}. \end{array}$$

Si on considère que tous les $\vec{X}^i_{s_l}$ sont égaux quel que soit l, la formule 3.2 peut se réduire en :

$$\begin{cases} \forall e = (x_e, y_e) \in E,\ \exists k_e,\ 1 \leq k_e \leq d \text{ tel que } \vec{X}^{k_e}.d(\vec{e}) + \rho^{k_e}_{y_e} - \rho^{k_e}_{x_e} \geq 1 \\ \text{et } \forall i,\ 1 \leq i < k_e,\ \vec{X}^i.d(\vec{e}) + \rho^i_{y_e} - \rho^i_{x_e} = 0 \end{cases}$$

Donc, l'existence de tels ordonnancements multi-dimensionnels est fortement reliée à l'existence d'hyperplans fortement séparateurs (≥ 1) et faiblement séparateurs ($= 0$).

L'algorithme présenté dans [21, 24] construit de tels ordonnancements, en se servant d'une décomposition du graphe de dépendances qui se fonde sur l'algorithme présenté par Karp, Miller et Winograd dans [35].

3.1.3 L'algorithme de Darte et Vivien

Cet algorithme se fonde sur la décomposition de Karp, Miller et Winograd [35] pour trouver un ordonnancement multi-dimensionnel du nid de boucles. Le principe général est de partir d'un PRDG G fortement connexe et uniformisé, de trouver le graphe G' des arêtes participant à au moins un multi-cycle de poids nul et de construire une partie de l'ordonnancement. Puis, on redécompose G' en composantes fortement connexes et on recommence la même opération : on cherche pour chaque composante fortement connexe le graphe des multi-cycles de poids nul et on construit une partie d'ordonnancement, etc...

La recherche des multi-cycles (union de cycles) de poids nuls est nécessaire dans le sens où le SURE correspondant au graphe est calculable si et seulement si il ne contient pas de cycle de poids nul. La recherche d'un cycle de poids nul est plus difficile que la recherche de multi-cycles de poids nul. C'est la raison pour laquelle on utilise ceux-ci, avec l'avantage de la propriété : tout cycle de poids nul est un multi-cycle de poids nul.

La partie d'ordonnancement que l'on va construire est un vecteur $\vec{X}$ commun à toutes les instructions de la composante connexe considérée pour la dimension correspondant au niveau de décomposition du graphe. On va chercher un vecteur $\vec{X}$ et des constantes pour chaque instruction de G satisfaisant les contraintes suivantes.

$$e = (x_e, y_e) \in G' \quad \Leftrightarrow \quad \vec{X}.\vec{d_e} + \rho_{y_e} - \rho_{x_e} \geq 0, \tag{3.3}$$

$$e = (x_e, y_e) \notin G' \quad \Leftrightarrow \quad \vec{X}.\vec{d_e} + \rho_{y_e} - \rho_{x_e} \geq 1. \tag{3.4}$$

Les articles [21, 24] donnent toutes les justifications théoriques. N'oublions pas que l'on cherche un ordonnancement multi-dimensionnel valide. D'après l'équation 3.2, nous voyons qu'il faut qu'au moins un des $\vec{X}$ trouvé dans une dimension de l'ordonnancement soit strictement séparateur. D'après la décomposition du graphe expliquée brièvement précédemment, on se rend compte qu'un sommet qui n'est pas dans G' ne sera plus dans le reste de la décomposition parce qu'on peut lui trouver un hyperplan strictement séparateur. A contrario, pour un sommet qui est dans G', on va « différer » la recherche d'un hyperplan strictement séparateur à une profondeur plus lointaine de la décomposition. C'est la raison de la recherche d'un hyperplan faiblement séparateur pour ceux-ci. En fait, il est facile de voir que l'on ne peut pas trouver d'hyperplan strictement séparateur pour un sommet de G'.

Lemme 1 (Darte et Vivien) *La fonction multi-dimensionnelle T :*

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \longrightarrow & \mathbb{Q}^d \\ T : (s_l, \vec{p}) & \rightarrow & (\vec{X}^1_{s_l}.\vec{p} + \rho^1_{s_l},\ \ldots,\ \vec{X}^d_{s_l}.\vec{p} + \rho^{d_l}_{s_l},\ 0,\ \ldots,\ 0) \end{array}$$

trouvée par l'algorithme de Darte et Vivien définit un ordonnancement multi-dimensionnel valide.

La preuve de ce lemme et toutes les justifications théoriques se trouvent dans les articles [21, 24]. Nous pouvons maintenant essayer d'étendre cette méthode de recherche de parallélisme de grain fin à une recherche de parallélisme à grain moyen. En effet, à un niveau de décomposition, au lieu de chercher un vecteur $\vec{X}$ et des constantes, on peut essayer de trouver une autre transformation, c'est ce que nous allons voir dans la partie 3.3.

3.2 Parallélisme à grain moyen

Les transformations présentées auparavant ne sont qu'un sous-ensemble des transformations de réordonnancement possibles pour les opérations d'un nid de boucles. De nombreuses transformations différentes ont été étudiées, telles entre autres l'échange de boucles, la distribution, la torsion, le tiling, le strip-mining [2, 76, 14, 69]. Chacune de ces transformation a ses propres règles de légalité et il est donc difficile de les composer et de prédire l'effet d'une composition.

Les transformations unimodulaires répondent partiellement à cette question en permettant d'unifier toute transformation pouvant être obtenue par composition d'échanges de boucles, de torsions et d'inversions [6, 72]. Le problème, c'est que ce contexte n'est pas assez général dans le sens où toutes les instructions sont modifiées de la même façon, c'est-à-dire que l'on considère toutes les instructions comme formant un bloc de base insécable. Nous verrons cet aspect plus en détail lorsque nous parlerons de l'article de Wolf et Lam [73].

Nous allons nous placer dans un contexte plus général que les transformations unimodulaires. Nous allons chercher une transformation T qui va consister en un ensemble de transformations affines, une par instruction du nid de boucle. Ainsi, pour chaque instruction s_p, on aura une fonction affine T_p de $\mathbb{Z}^n$ dans $\mathbb{Z}^m$:

$$T_p : [i_1,\ \ldots,\ i_n] \rightarrow [f^1_p,\ \ldots,\ f^m_p]$$

où les fonctions f^i_p sont des fonctions affines des indices de boucles et des constantes du programme. Dans notre cas, nous allons chercher pour chaque instruction s_p une matrice M_p (partie linéaire) et un vecteur $\vec{\rho}_p$ (partie constante). M_p sera une matrice carrée de taille $n \times n$ et $\vec{\rho}_p$ un vecteur de dimension n. Une instruction s_p exécutée à l'itération $\vec{i}$ sera exécutée, après transformation, à l'itération $M_p\vec{i} + \vec{\rho}_p$. Dans le cas général, une transformation est légale si et seulement si :

$$\forall s_p, s_q \in V,\ \{s_p(\vec{i}) \Rightarrow s_q(\vec{j})\} \ \Rightarrow \{T_p(\vec{i}) \leq_l T_q(\vec{j})\}$$

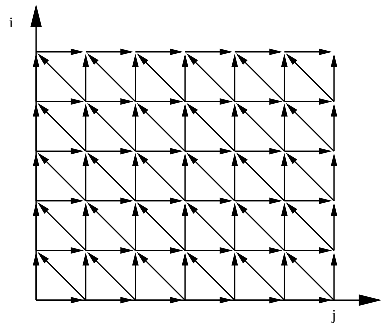
plus la prise en compte de l'ordre textuel si $T_p(\vec{i}) = T_q(\vec{j})$. Dit autrement, la transformation doit préserver les dépendances. Notre principale contribution à ce domaine est la caractérisation de l'ensemble des transformations de ce type qui donnent le nombre maximal de boucles complètement permutables, notion que nous allons définir précisément dans ce qui suit.

3.2.1 Boucles complètement permutables

La fonction objective que nous nous imposons est de trouver le nombre maximum de boucles complètement permutables dans un nid de boucles par une transformation affine multi-dimensionnelle gardant le même nombre de boucles. Le problème va donc être de trouver la transformation affine maximisant cette fonction objective, problème que nous traiterons dans la section 3.3. En attendant, nous allons définir ce qu'est un nid de boucles complètement permutable et son intérêt dans la parallélisation automatique. Considérons par exemple le nid de boucle de la figure 3.1.

do $i = 0, 5$
  do $j = 0, 6$
    $s_1 : a(j+1) = a(j) + a(j+1) + a(j+2)$
  enddo
enddo

$$D = \{(0,1), (1,0), (1,-1)\}$$



Exemple 3.1: nid de boucles avec ses dépendances (EDG).

Les dépendances peuvent se mettre sous forme de matrice :

$$D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & -1 \end{pmatrix}$$

où chaque ligne correspond à une boucle. Imaginons que l'on veuille échanger les deux boucles i et j par une transformation unimodulaire T :

$$D' = TD = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \end{pmatrix}$$

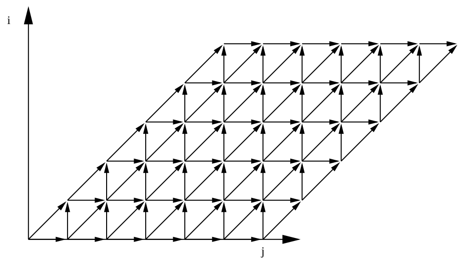
On peut voir que cette transformation n'est pas valide dans le sens où tous les vecteurs de dépendance ne sont plus lexicographiquement positifs.

Par contre, si on applique d'abord une torsion du nid de boucle par rapport à i à l'aide d'une autre transformation T' :

$$D'' = T'D = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

```
do i = 0,5
   do j = i,6+i
      s1 : a(j+1-i) = a(j-i) + a(j+1-i) + a(j+2-i)
   enddo
enddo
```

$$D = \{(0,1),(1,1),(1,0)\}$$



Exemple 3.2: nid de boucles transformé avec ses dépendances (EDG).

illustrée par l'exemple 3.2, on voit que l'on peut échanger les deux boucles et que l'on a toujours des vecteurs lexicographiquement positifs. On dit dans ce cas que les boucles sont *permutables* (*complètement* permutables si il y a plus de deux boucles). Cette caractéristique est importante à la fois pour des raisons de parallélisme et pour des raisons de changement de grain de parallélisation.

Théorème 1 (Boucles complètement permutables) *Les boucles consécutives de profondeur i à j d'un nid de boucles avec des vecteurs de dépendances D sont* complètement permutables *si et seulement si*

$$\forall \vec{d} \in D \ : \ (d_1, ..., d_{i-1}) \succ \vec{0} \ ou \ (\forall k, \, i \leq k \leq j \ : \ d_k \geq 0)$$

La preuve de ce théorème est immédiate. La condition nécessaire et suffisante pour échanger deux boucles est qu'on doit garder les vecteurs de dépendance lexicographiquement positifs en échangeant les coordonnées correspondantes. Voyons maintenant l'intérêt des boucles complètement permutables dans le cas de la parallélisation automatique. La synthèse qui suit tente de donner un aperçu de toutes les transformations possibles dans ce contexte.

3.2.2 Parallélisation

Les itérations d'une boucle peuvent s'exécuter en parallèle si et seulement s'il n'y a pas de dépendance portée par cette boucle. La maximisation du degré de parallélisme au sein d'un nid de boucles revient à trouver le nombre maximal de boucles parallèles (définition 15) au sein de ce nid de boucle (voir la discussion à ce sujet dans la thèse de Frédéric Vivien [69]). Nous allons voir dans cette partie les différents « types » de parallélisme que l'on peut trouver avec des transformations unimodulaires et le lien entre ces différents types et les boucles complètement permutables.

Une fois qu'un nid de boucle est complètement permutable, il est facile de trouver le degré de parallélisme à grain fin maximal. Nous allons voir qu'il est également facile de trouver le même degré de parallélisme pour du gros grain. Pour aller encore plus loin, nous allons voir que l'on peut encore trouver le degré maximal de parallélisme avec un *grain moyen*, c'est-à-dire entre du gros grain et du grain fin, cela avec un changement d'atomicité des instructions élémentaires.

Parallélisation à grain fin

Un ensemble de n boucles complètement permutables peut être transformé en un code contenant au moins $(n-1)$ boucles parallèles [33], ceci à l'aide d'une transformation « front de vagues» (wavefront) représentée par la matrice

$$\begin{pmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$$

Cette transformation «front de vagues» a comme effet de placer le maximum de boucles parallèles au plus profond du nid de boucles et est donc utile dans le cas d'une parallélisation à grain fin. Le résultat de cette transformation sur le nid de boucles transformé de l'exemple 3.2 est illustré par la figure 3.1. Les instructions d'une vague sont exécutées en parallèle. Nous avons donc un front de vagues successives faisant chacune s'exécuter un «hyperplan» d'instructions en parallèle. Après application de cette transformation sur un nid de boucles de profondeur n, on obtient une boucle séquentielle en tête et $(n-1)$ boucles parallèles.

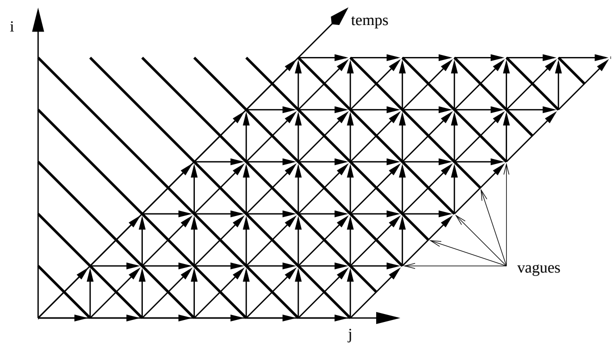


Figure 3.1: résultat de la transformation «front de vagues».

Parallélisation à gros grain

À l'inverse de l'optique à grain fin, nous allons chercher ici à rendre le plus de boucles externes parallèles. La dimension de l'espace d'itération d'un nid de boucles est n. Cependant, il se peut que la dimension de l'espace embrassé par les vecteurs de dépendance soit inférieure à n, disons par exemple k. Dans ce cas, il est possible de trouver $n-k$ boucles externes parallèles par une transformation appropriée [34]. En fait, il est possible de trouver une transformation T dont les $(n-k)$ premières lignes génèrent le *sous-espace orthogonal* de l'espace embrassé par les dépendances, ce qui va produire le nombre maximal de boucles parallèles externes.

Parallélisation à grain moyen

Une transformation importante et qui est la clef de notre travail est une transformation que nous pouvons appeler *tiling*, *blocking* ou encore *supernode partitioning*, pour rendre hommage à Irigoin et Triolet [33] qui ont introduit cette notion. L'idée générale est de changer l'atomicité du programme. Là où l'unité atomique d'exécution était l'instruction, on va passer à une unité atomique d'exécution qui va être un *ensemble d'instructions* mises dans une *tuile*. L'intérêt est évident dans de nombreux cas, notamment dans les deux cas suivants :

– dans le cas de machines à mémoire partagée, on augmente la localité des données et donc les chances d'accès à une donnée déjà présente dans le cache ;
– dans le cas de machines à mémoire distribuée, on va réduire le volume de communications et la synchronisation.

Ainsi, la figure 3.2 représente le nid de boucles complètement permutables de l'exemple 3.2. Nous ne changeons pas l'ordre d'exécution des instructions du nid de boucles en rajoutant les deux boucles I et J qui ne font que rajouter du contrôle. C'est la première étape de la mise sous forme de blocs des instructions du nid de boucles. Ce qu'il est important de remarquer, c'est que d'un nid de boucles à deux boucles complètement permutables, on est passé à un nid de boucles avec quatre boucles complètement permutables. C'est la condition

```
do I = 0,5,2
  do i = I,I+1
    do J = I,I+6,2
      do j = max(J,i),min(J+1,6+i)
        a(j+1-i) = a(j-i) + a(j+1-i) + a(j+2-i)
      enddo
    enddo
  enddo
enddo
```

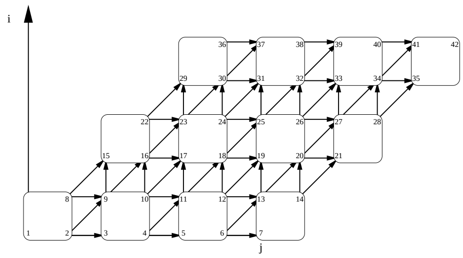


Figure 3.2: première étape de la mise sous forme de blocs du programme de l'exemple 3.2. L'ordre d'exécution des opérations n'est pas modifié.

sine qua non à la légalité de l'inversion présentée dans la figure 3.3. Cette fois-ci, on a vraiment changé l'ordre d'exécution des instructions. Chaque bloc est maintenant *atomique* dans le sens où dès que ce bloc à toutes les données dont il a besoin, il peut s'exécuter sans avoir besoin de communiquer à nouveau (il est maintenant possible d'exécuter en parallèle certaines tuiles, par exemple avec une transformation « front de vagues » comme présenté à la figure 3.4). En fait, tous les points de synchronisation sont les débuts et les fins de blocs. Cette décomposition est présentée dans un but illustratif : le code peut directement être mis sous forme de tuiles sans passer par toutes ces étapes (voir à ce sujet le travail de Xue [78]).

```
do I = 0,5,2
  do J = I,I+6,2
    do i = I,I+1
      do j = max(J,i),min(J+1,6+i)
        a(j+1-i) = a(j-i) + a(j+1-i) + a(j+2-i)
      enddo
    enddo
  enddo
enddo
```

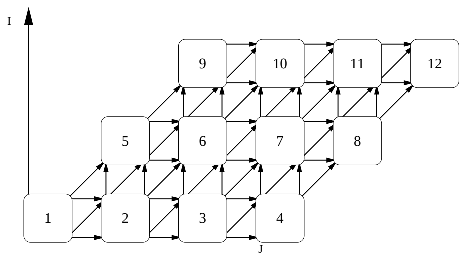


Figure 3.3: deuxième étape de la mise sous forme de blocs du programme de l'exemple 3.2. Inversion des boucles i et J, ce qui mofidie l'ordre d'exécution des opérations.

Travaux précédents

Le travail présenté par Wolf et Lam [73] avait unifié tous les algorithmes précédents se fondant sur des opérations de matrices élémentaires comme la torsion, l'échange ou la distribution dans un seul et même contexte qui sont les transformations unimodulaires valides.

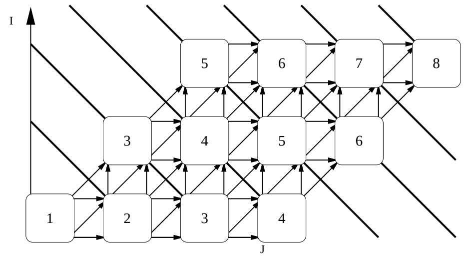


Figure 3.4: transformation « front de vagues » sur le nid de boucles 3.3 mis sous forme de blocs.

Nous allons voir dans le paragraphe suivant que l'on peut se placer dans un cadre encore plus général que les transformations unimodulaires.

De plus, ils ne considèrent dans ce travail que les ensembles de vecteurs de distance représentés par des vecteurs de direction. C'est une limitation que nous allons dépasser ici, puisque nous considérerons les graphes de dépendances polyédriques. Un problème très important de l'approche de Wolf et Lam est de ne pas prendre en compte la structure du graphe. En effet, ils ne traitent que des nids de boucles avec une seule instruction ou un ensemble d'instructions ne formant qu'un seul bloc. Ainsi, le fait de ne pas prendre en compte la structure du graphe en terme de composantes fortement connexes fait perdre du parallélisme [12].

L'algorithme de Wolf et Lam est directement adapté à la recherche de boucles complètement permutables. Cet algorithme est un cas particulier de la méthode de Darte et Vivien [24]. Il restait cependant à étendre cette méthode pour permettre la détection de boucles complètement permutables [20].

3.3 Nombre maximal de boucles permutables

L'algorithme de Darte et Vivien construit récursivement une fonction de transformation affine $T_s(\vec{p}) = M_s.\vec{p}+\vec{\rho}_s$ exhibant le maximum de boucles parallèles internes. Cette fonction à la propriété suivante : la première ligne de M_s est la même pour toutes les instructions d'une même composante fortement connexe du PRDG, ce qui permet d'écrire plus simplement les contraintes de dépendance (équations 3.3 et 3.4). Les dépendances vérifiant 3.4 ne sont plus considérées dans le reste de la décomposition, puisque déjà satisfaites à ce niveau (on a trouvé un hyperplan strictement séparateur pour elles). L'algorithme continue récursivement sur le sous-graphe composé des arcs vérifiant l'équation 3.3, avec la même propriété : la ligne de M_s correspondant au niveau de décomposition courant est la même pour toutes les instructions d'une même composante fortement connexe.

Nous allons étendre cette approche en cherchant, pour chaque instruction s, une transformation affine multi-dimensionnelle inversible exhibant le nombre maximal de boucles permutables externes. Pour cela, on va appliquer le même canevas que précédemment, à la différence près qu'à chaque niveau de la décomposition, au lieu de définir une ligne de la transformation (le vecteur $\vec{X}$ et les constantes ρ_s), on va définir *plusieurs* lignes de cette transformation (autant que possible en fait) : une *matrice* M de vecteurs $\vec{X}$ et un vecteur de constantes $\vec{\rho}_s$. En d'autres termes, on veut que les composantes de la matrice de dépendances après transformation soient toutes positives sur ces dimensions, ce qui peut se résumer par la formule suivante, qui est la base de notre recherche :

$$\forall e \in G,\ M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x \geq 0 \tag{3.5}$$

avec M de taille maximale et de rang plein. Ceci définit une partie de la transformation finale qui devra être valide, c'est-à-dire qui devra respecter toutes les dépendances. A cet effet, on applique à nouveau cette technique récursivement sur les arcs qui ne sont pas satisfaits, c'est-à-dire sur les arcs $e = (x, y)$ vérifiant :

$$M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x = 0$$

Nous verrons plus loin que cette approche est valide et que l'algorithme se termine.

Nous nous plaçons pour commencer au niveau le plus externe de cette décomposition, nous verrons par la suite comment l'adapter récursivement aux niveaux suivants. G est le PRDG courant, supposé fortement connexe. n est le nombre de boucles imbriquées. Le problème de la détection de boucles permutables externes revient à résoudre le problème suivant : caractériser les matrices M (**de rang plein et de taille maximale**) et les vecteurs ρ_s vérifiant la condition 3.5.

Remarquons que l'équation 3.5, condition nécessaire sur les arcs, peut se reformuler en une condition nécessaire sur les circuits :

Proposition 2 (Condition sur les circuits (Prop. 1)) *Etant donnée une matrice M, il existe des vecteurs de constantes $\vec{\rho}_s$ vérifiant le système d'inéquations 3.5 si et seulement si pour tout circuit C de G, $M.\vec{w}_C \geq 0$ où $\vec{w}_C$ est le poids de C, c'est-à-dire la somme des vecteurs de dépendances des arcs de C.*

Preuve :

$\Rightarrow$ soit C un circuit de G. En sommant la formule 3.5, les constantes s'annulent et on obtient, $M.\vec{w}_C \geq 0$.

$\Leftarrow$ réciproquement, on peut raisonner dimension par dimension, c'est-à-dire ligne par ligne de M. Pour une ligne $\vec{v}$ donnée de M, on doit résoudre un système d'inéquations de la forme :

$$\vec{v}.\vec{d_e} + \rho_y - \rho_x \geq 0$$

pour tous les arcs $e = (x, y)$ de G. La condition pour que ce système ait une solution est justement que $\vec{v}.\vec{w}_C \geq 0$ pour tout circuit C : les constantes ρ peuvent être déterminées par une technique « à la Bellman-Ford » [16], par exemple ρ_x peut être défini comme la longueur d'un plus long chemin d'une source au sommet x, chaque arc e étant pondérée par $-\vec{v}.\vec{d_e}$.

□

Cette propriété nous permet de mettre en évidence, comme dans l'algorithme de Darte et Vivien, le rôle fondamental du sous-graphe G' des multi-circuits de poids nul.

Proposition 3 (Condition sur les arcs (Prop. 1)) *La transformation composée d'une matrice M et de vecteurs de constantes $\vec{\rho}_s$ est valide et donne le nombre maximal de boucles complètement permutables si et seulement si*

$$\begin{array}{rcl} (i)\ \forall e \notin G',\ M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x & \geq & 0 \\ (ii)\ \forall e \in G',\ M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x & = & 0 \end{array}$$

et M est de rang plein et de rang maximal.

Preuve : soit C un multi-circuit de G, c'est-à-dire une union (somme) de circuits de G. On a donc $M.\vec{w}_C \geq 0$ comme somme de quantités positives (proposition 2). Maintenant, si C est un multi-circuit de poids nul, on a $M.\vec{w}_C = 0$, ce qui peut se réécrire en raisonnant sur les arcs :

$$\sum_{e \in C} M.\vec{d_e} + \vec{\rho_y} - \vec{\rho_x} = 0.$$

En vertu de la formule 3.5, on a une somme de termes positifs qui est nulle, donc tous les termes sont nuls. En particulier, si $e \in G'$, e appartient à un multi-circuit de poids nul, donc vérifie l'équation :

$$M.\vec{d_e} + \vec{\rho_y} - \vec{\rho_x} = 0$$

Ce qui termine la preuve. □

Notons que cette transformation est valide seulement si on peut compléter la transformation pour satisfaire les arcs résiduels (ce que nous allons montrer à la Proposition 5). La matrice M est composée de r vecteurs lignes $\vec{v_l}$, avec $1 \leq l \leq r$. C'est ce r que l'on veut maximiser. Soit U l'espace vectoriel engendré par les poids des circuits de G'. Appelons k sa dimension.

Lemme 2 **($\vec{v_l} \in U^{\perp}$)**

$$\forall \textit{circuit } C \in G',\ M.\vec{w}_C = 0$$

autrement dit chaque $\vec{v_l}$ est dans l'orthogonal de U et donc $r \leq n - k$.

Preuve : Si C est un circuit de G', tous ses arcs appartiennent à G' et donc, en vertu de la propriété 3, on a $M.\vec{w}_C = 0$ comme somme de termes nuls. De plus, si $\vec{u} \in U$, $\vec{u}$ est par définition combinaison linéaire de poids de circuit de G'. On a donc également $M.\vec{u} = 0$. □

Nous allons montrer qu'en fait $r = n - k$. La difficulté vient du fait qu'on a d'autres contraintes à respecter que celles de l'orthogonalité (les contraintes (i) de la Proposition 3) et que l'on veut une méthode constructive et polynômiale.

Lemme 3 **(Vect($\vec{v_l}$) $= U$)** *On peut trouver $n - k$ vecteurs $\vec{v_l}$ linéairement indépendants, autrement dit $r = n - k$ et les lignes de M génèrent l'orthogonal de U.*

Preuve : Pour prouver ce lemme, il suffit de montrer que l'on peut toujours construire $n-k$ vecteurs $\vec{v}$ linéairement indépendants satisfaisant :

$$\forall e \in G,\ \vec{v}.\vec{d_e} + \rho_y - \rho_x \geq 0$$

pour des constantes ρ « bien choisies ».

La première étape va être de trouver l'orthogonal de l'espace vectoriel U engendré par les poids des circuits de G'. En général, le nombre de circuits de G' est exponentiel en le nombre d'arcs de ce graphe. Pour éviter ce caractère non polynômial, on peut démontrer que U est également généré par les poids d'une *base de cycles* de G' (dont la construction est polynômiale), en considérant celui-ci comme un graphe non-orienté. (En effet, G' est une union de composantes fortement connexes et on peut démontrer que pour un graphe fortement connexe, il existe une base de cycle qui n'est constituée que de circuits.)

Notons W la matrice des poids de cette base de cycles. On doit trouver une matrice E de rang plein et maximal qui vérifie l'équation $E.W = 0$. Une méthode pour résoudre cette équation est de mettre la matrice W sous sa *forme normale de Smith* [56], c'est-à-dire de décomposer la matrice W en trois matrices Q_1, S et Q_2 telles que S soit une matrice « diagonale » de rang k et Q_1 et Q_2 deux matrices unimodulaires. On doit donc résoudre l'équation $E.Q_1.S = 0$ avec E de rang plein et maximal. Ceci revient à résoudre l'équation $H.S = 0$ avec H de rang plein et maximal, puis on obtient E en posant $E = H.Q_1^{-1}$. H est facile à trouver puisque S est pseudo-diagonale : H est la matrice de taille $(n-k) \times n$ dont les k premières colonnes sont nulles et les $(n-k)$ dernières forment la matrice identité de taille $(n-k)$.

Nous avons donc trouvé une base (formée par les lignes de E) de l'orthogonal de U. Il reste maintenant à trouver une matrice M, c'est-à-dire $n-k$ vecteurs $\vec{v}$, tels que

$$\forall e \in G,\ \vec{v}.\vec{d_e} + \rho_y - \rho_x \geq 0$$

pour des constantes « bien choisies » (Proposition 3), ou encore en termes de circuits $\forall$circuit $C \in G$, $\vec{v}.\vec{w}_C \geq 0$ (Proposition 2). On dispose d'un vecteur qui vérifie cette équation, et même de façon plus forte, c'est le vecteur d'ordonnancement $\vec{X}$ trouvé par l'algorithme de Darte et Vivien pour ce niveau de décomposition :

$$\begin{array}{rll} (i) & \forall\text{circuit } C \in G', & \vec{X}.\vec{w}_c = 0 \\ (ii) & \forall\text{circuit } C \notin G', & \vec{X}.\vec{w}_c \geq \#(e \in c, e \notin G') > 0 \end{array}$$

Pour plus de précisions, voir [24]. Soient $\vec{e}_1, \ldots, \vec{e}_{n-k}$ les lignes de la matrice E. Si on pose $\vec{v}_l = (\vec{e}_l + \lambda_l \vec{X})$, avec l le numéro de la ligne de M $(1 \leq l \leq (n-k))$, il suffit de trouver des λ_l tels que $\vec{v}_l.\vec{w}_c \geq 0$ et tels que les $\vec{v}_l$ soient linéairement indépendants.

Nous donnons pour commencer une condition sur les λ_l pour que les $\vec{v}_l$ soient linéairement indépendants : $\vec{X}$ appartient lui aussi à l'orthogonal de U, il est donc combinaison linéaire des vecteurs-ligne de E. $\vec{X} = \sum_{i=1}^{n-k} y_i \vec{e}_i$. Notons Λ, le vecteur-colonne de composantes les λ_l, Y le vecteur-ligne de composantes les y_i et X le vecteur-ligne de composantes les composantes du vecteur $\vec{X}$. Matriciellement, on a :

$$X = Y.E \text{ et } M = E + \Lambda.X = (I_{n-k} + \Lambda.Y)E$$

E étant de rang plein, M est de rang plein si et seulement si la matrice $I_{n-k} + \Lambda.Y$, carrée de taille $(n-k)$, est inversible. En fait, cette matrice est la matrice de passage de la base E à la base M. On peut montrer que son déterminant est égal à $1 + Y.\Lambda$, c'est-à-dire $1 + \sum_{i=1}^{n-k} y_i \lambda_i$. En d'autres termes, les vecteurs $\vec{v}_l$ sont linéairement indépendants si et seulement si

$$\sum_{i=1}^{n-k} y_i \lambda_i \neq -1 \tag{3.6}$$

Remarquons maintenant que $\vec{v}_l.\vec{w}_C = \vec{e}_l.\vec{w}_C + \lambda_l \vec{X}.\vec{w}_C$. Si C est un circuit de G', on a donc $\vec{v}_l.\vec{w}_C = 0$ quel que soit λ_l. Si C est un circuit élémentaire dont au moins un des arcs n'appartient pas à G', on a $\vec{X}.\vec{w}_C \geq 1$. Il suffit donc de prendre λ_l suffisamment grand (selon la valeur fixée de $\vec{v}_l.\vec{w}_C$) pour satisfaire l'inéquation. Si C est un circuit quelconque, il est somme de circuits élémentaires et l'inéquation pour ce circuit est satisfaite dès lors qu'elle est satisfaite pour tous les circuits élémentaires. Ceci prouve l'existence des λ_l : il suffit de les choisir suffisamment grands et vérifiant l'équation 3.6.

En pratique, on peut souhaiter trouver les λ_l les plus petits possibles. Dans ce cas, le principe général de l'algorithme consiste à résoudre, pour chaque ligne l de la matrice M,

$$\forall a = (x, y) \in G,\ (\vec{e_l} + \lambda_l \vec{X}).\vec{d_a} + [\vec{\rho_y}]_l - [\vec{\rho_x}]_l \geq 0.$$

On résout ceci en faisant une recherche dichotomique sur λ_l. Pour chaque valeur de λ_l, on applique un algorithme de type Bellman-Ford (voir la Proposition 2 et [16]) pour trouver les constantes quand c'est possible. Par cette méthode, on aura le plus petit λ_l qui marche avec les plus petites constantes correspondantes. En pratique, une recherche séquentielle doit être suffisante sur les exemples réels.

Nous avons donc prouvé que nous pouvons trouver $(n-k)$ vecteurs linéairement indépendants qui marchent. De plus, nous avons donné une méthode *constructive.* □

Remarquons que quelque soit la façon dont on complétera la transformation à partir de M, les dépendances correspondant à des arcs $e = (x, y)$ telles que $M.\vec{d_e} + \vec{\rho_y} - \vec{\rho_x} \geq 0$ et $M.\vec{d_e} + \vec{\rho_y} - \vec{\rho_x} \neq 0$ seront satisfaites car déjà portées par au moins une des boucles correspondant à M. Nous allons maintenant revenir aux autres arcs (ceux pour lesquels $M.\vec{d_e} + \vec{\rho_y} - \vec{\rho_x} = 0$) et montrer comment on pourra les satisfaire récursivement.

Nous avons besoin des lemmes suivants :

Lemme 4

$$C \in G' \Longleftrightarrow \vec{w}_C \in U$$

Preuve : $\Rightarrow$ est vrai par définition de U. Réciproquement, considérons le vecteur $\vec{X}$ donné par l'algorithme de Darte et Vivien. Ce vecteur a la propriété d'être dans l'orthogonal de U et de vérifier $\vec{X}.\vec{w}_C > 0$ pour tout circuit C non dans G'. Par conséquent, si C est un circuit non dans G', $\vec{w}_C$ n'est pas dans U sinon on aurait $\vec{X}.\vec{w}_C = 0$. □

Lemme 5 *Si M est solution de la Proposition 2, de rang plein et maximal, alors pour tout circuit C non dans G', on a :*

$$M.\vec{w}_C \geq 0 \text{ et } M.\vec{w}_C \neq 0$$

Preuve : On a quel que soit le circuit C appartenant à G, $M.\vec{w}_C \geq 0$. Si $M.\vec{w}_C = 0$, ceci implique que $\vec{w}_C$ est orthogonal à toutes les lignes $\vec{v_l}$ de M, c'est-à-dire à l'espace vectoriel V généré par les $\vec{v_l}$. Si M est de rang plein et maximal, V est égal à l'orthogonal de U (cf Lemme 3). Par conséquent $\vec{w}_C \in (U^{\perp})^{\perp} = U$. Le lemme 4 montre alors que $C \in G'$. □

Tous ces lemmes nous permettent maintenant de caractériser très précisément l'ensemble des transformations qui permettent de construire un ensemble maximal de boucles complètement permutables.

Proposition 4 (Condition sur les circuits (Prop. 2)) *La proposition 2 est équivalente aux conditions suivantes :*

$$\begin{aligned} (i)\ \forall \textit{circuit } c \notin G',\ M.\vec{w}_c &\geq 0 \\ &\neq 0 \\ (ii)\ \forall \textit{circuit } c \in G',\ M.\vec{w}_c &= 0 \end{aligned}$$

et M est de rang plein et maximal, c'est-à-dire composée de $n-k$ vecteurs linéairement indépendants.

En raisonnant sur les arcs, nous obtenons la propriété suivante :

Proposition 5 (Condition sur les arcs (Prop. 2)) *La transformation composée d'une matrice M et de vecteurs de constantes $\vec{\rho}_s$ est valide et donne le nombre maximal de boucles complètement permutables si et seulement si :*
- *M est de rang plein égal à $n-k$ où k est la dimension de l'espace vectoriel engendré par le poids des circuits de G' ;*
- *on a les deux propriétés suivantes :*

$$\begin{array}{rcl} (i)\ \forall e \notin G',\ M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x & \geq & 0 \\ (ii)\ \forall e \in G',\ M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x & = & 0 \end{array}$$

- *et le sous-graphe formé par les arcs $e = (x, y)$ vérifiant : $M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x = 0$ est une forêt de composantes fortement connexes de G', c'est-à-dire est un graphe dont les composantes fortement connexes sont exactement celles de G'.*

Preuve : Soit H le sous-graphe formé par les arcs $e = (x, y)$ vérifiant : $M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x = 0$. Il nous reste juste à démontrer que tout circuit de H est un circuit de G'. En effet, soit C un circuit de H. En sommant chacune des égalités $M.\vec{d_e} + \vec{\rho}_y - \vec{\rho}_x = 0$ pour les arcs e de C, on obtient $M.\vec{w}_C = 0$. Puisque C est aussi un circuit de G, la proposition 4 montre que C est dans G'. □

Cette dernière proposition nous permet de nous ramener à une approche récursive sur les composantes fortement connexes de G', comme dans l'algorithme de Darte et Vivien. La seule différence est qu'il peut rester certaines dépendances à satisfaire correspondant à des arcs qui ne sont pas dans G'. Mais on a montré dans la proposition précédente que le graphe formé par ces arcs est acyclique. Elles peuvent donc être satisfaites avant de passer aux dimensions suivantes par un simple tri topologique [16] qui se traduit dans le code par un nouvel ordre textuel sur les instructions et des distributions de boucle si on n'est pas au fond de la récursion. Deux remarques avant de conclure cette partie par un exemple.
- Rappelons qu'il est possible que l'espace vectoriel engendré par les poids des circuits de G ne soit pas de dimension pleine, c'est-à-dire de dimension $m < n$. Dans ce cas, on peut, en plus de l'aspect permutable, imposer que les $n-m$ lignes de transformation rendent les $n-m$ premières boucles parallèles (voir la recherche de gros grain, §3.2.2, p. 34).
- La recherche des boucles permutables lors des appels récursifs suivants doit permettre de construire des dimensions linéairement indépendantes de celles déjà construites. Pour cela, on raisonne de façon similaire à ce qui vient d'être exposé, simplement on travaille non plus dans l'espace de dimension n tout entier, mais dans un espace vectoriel supplémentaire à l'espace vectoriel engendré par les dimensions déjà construites. Ceci ne modifie pas les résultats précédents, mais doit être considéré au moment de l'implantation.

Nous venons donc de caractériser l'ensemble des transformations qui donnent le nombre maximum de boucles complètement permutables dans un nid de boucles par une transformation affine multi-dimensionnelle et la preuve constructive du Lemme 3 permet d'implanter cette transformation.

```
do i = 1, N
   do j = 1, N
      a(i, j) = b(i, j − 1) × 3        (* s1 *)
      b(i, j) = 6 − a(i − 1, j + 1)    (* s2 *)
   enddo
enddo
```

$$\begin{tikzcd} s_1 \arrow[r, bend left, "{\binom{1}{-1}}"] & s_2 \arrow[l, bend left, "{\binom{0}{1}}"] \end{tikzcd}$$

Exemple 3.3: exemple simple avec son graphe de dépendances.

3.3.1 Un exemple simple

L'algorithme de Wolf et Lam [73] appliqué sur l'exemple 3.3 transformerait le nid de boucles par une torsion. En fait, le graphe de dépendance contient un seul circuit $C = (1\ 0)^t$ qui vérifie la proposition 4 avec M égal à l'identité (notons qu'avec ce graphe nous pouvons également obtenir du parallélisme à gros grain). Il reste à trouver les vecteurs de constantes ρ de la proposition 3 : pour cela, on peut appliquer pour chaque dimension, une recherche de plus long chemin « à la Bellman-Ford ». On trouve alors pour la première dimension : $\rho_{s_1} = \rho_{s_2} = 0$ et pour la seconde $\rho_{s_1} = 0$ et $\rho_{s_2} = 1$. Les nouveaux vecteurs de dépendances sont donc :

$$\begin{pmatrix}1\\-1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} - \begin{pmatrix}0\\0\end{pmatrix} = \begin{pmatrix}1\\0\end{pmatrix} \quad \text{et} \quad \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix} - \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\0\end{pmatrix}$$

```
do i = 1, N
   do j = 1, N + 1
      if j ≥ 2 then
         b(i, j − 1) = 6 − a(i − 1, j)    (* s2 *)
      endif
      if N ≥ j then
         a(i, j) = b(i, j − 1) × 3        (* s1 *)
      endif
   enddo
enddo
```

Exemple 3.4: Code généré pour l'exemple 3.3.

Les nouveaux vecteurs de dépendances sont positifs. Le premier est porté par la boucle externe tandis que le second est interne aux deux boucles. Pour respecter la seconde dépendance, il faut donc placer s_2 *textuellement* avant s_1. On voit donc qu'avec un simple décalage inter-itération et un réordonnancement intra-itération, on a réussi à exposer 2 boucles permutables. Cette solution est par nature indétectable par les solutions proposées jusqu'ici dans la littérature, bien que l'exemple soit on ne peut plus simple. Le code résultant est présenté dans l'exemple 3.4.

3.4 Problêmes ouverts

Nous avons présenté une méthode permettant de caractériser l'ensemble des transformations permettant de trouver le nombre maximal de boucles complètement permutables dans un nid de boucles parfaitement imbriquées représenté par un graphe des dépendances de donnée réduit polyédrique. Nous avons généralisé l'algorithme de Wolf et Lam au cas de corps de boucles avec plusieurs instructions. La mise du nid de boucles sous la forme de boucles complètement permutables est une étape nécessaire au tiling (plus exactement, la dérivation de M et des vecteurs ρ est nécessaire, la génération effective du code intermédiaire ne l'est pas forcément [78]).

Par exemple, comment décrire un critère d'optimalité pour la détection des boucles complètement permutables ? Il se pose en effet un problème majeur : comment savoir ce qui est « optimal » (voir à ce sujet la thèse Frédéric Vivien [69]). Dans un autre ordre d'idée, comment trouver une bonne taille et un bon aspect pour une tuile ? Comment placer les données sur les différents processus par rapport à l'endroit où elles sont utilisées ? Toutes ces questions cruciales doivent trouver une réponse (même partielle) pour permettre une réelle génération de code parallèle efficace.

Chapitre 4

Le graphe des dépendances mixte

Les transformations de code doivent non seulement prendre en compte les dépendances de données pour s'assurer de leur respect, mais également les dépendances de contrôle dans le cas de codes à flot de contrôle complexe, comme par exemple avec des expression conditionnelles au sein des instructions `if` qui sont impliquées dans une dépendance de donnée. Comme nous allons le voir dans ce chapitre, le graphe des dépendances de donnée réduit ne se prête pas bien à la représentation de ce type de code. Des approches ont été proposées pour gérer ce genre de dépendances, notamment la « *if conversion* », mais elles ne sont pas exemptes de défaut (par exemple une complexité du code accrue). D'autres approches ont été proposées qui utilisent le graphe des dépendances de contrôle, mais comme nous allons le voir, elles ne permettent pas de trouver tous les codes légaux possibles.

En nous servant des deux transformations qui sont la distribution et la fusion de boucles, nous allons présenter dans ce chapitre un nouveau graphe des dépendances que nous avons appelé le *graphe des dépendances mixte* qui a la particularité de résumer sur chaque sommet le flot de contrôle attaché à une action du programme. Nous verrons que cette nouvelle représentation est une extension du RDG et qu'elle peut être utilisée de la même façon. Nous verrons également que l'introduction de variables temporaires dans le code généré permet de diriger la recherche de partitions valides dans le graphe, ce qui n'était pas le cas avec les approches précédentes où c'était une partition donnée qui déterminait l'utilisation d'un temporaire.

Nous allons présenter une version modifiée de l'algorithme d'Allen et Kennedy qui utilise le graphe des dépendances mixte et qui va nous permettre de mettre en avant la puissance et la souplesse d'utilisation de notre nouvelle représentation. Nous allons également montrer que la minimisation du nombre de variables temporaires introduites est un problème NP-complet.

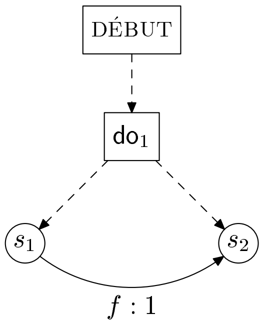
4.1 Distribution et fusion de structures de contrôle

La *distribution* d'une boucle (introduite par Muraoka [43]) convertit une boucle unique en plusieurs boucles, chacune d'elle ayant dans son corps un sous-ensemble des instructions de la boucle originale. Une instruction de la boucle originale se trouve dans une et une seule des nouvelles boucles. La figure 4.1 donne un exemple de distribution d'une boucle.

La distribution d'une boucle permet d'obtenir dans certains cas des nids de boucles parfaits, autorisant ainsi l'application de techniques utilisables uniquement dans ce cas là [54]. Elle permet également de réduire le nombre d'instructions dans le corps des boucles, ce qui permet parfois d'augmenter les performances par une réduction des défauts de cache et une meilleure utilisation des registres [76] (la fusion de boucles que nous verrons plus bas peut également avoir les mêmes conséquences). Un des aspects intéressants de la distribution de boucles dans un contexte de recherche de parallélisme est que cette opération permet dans certains cas de transformer des dépendances de données portées par une boucle en dépendances indépendantes de la boucle, ce qui permet d'exhiber une ou des boucles parallèles, comme dans l'exemple de la figure 4.1 : dans la boucle originale, il y a une dépendance de donnée entre $a(i)$ et $a(i-1)$ qui est portée par la boucle ; par la distribution, cette dépendance devient indépendante de la boucle, rendant ainsi les deux boucles parallèles.

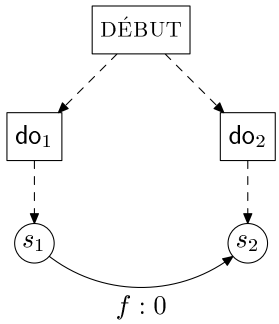
Malgré cette liste d'avantages, la distribution d'une boucle peut également avoir des inconvénients. Un premier inconvénient, qui est lié à l'augmentation de localité des données, est que la distribution peut parfois dégrader les performances puisque l'on modifie temporellement les accès aux cases mémoires. Ainsi, dans l'exemple donné par la figure 4.1, la boucle distribuée (b) peut faire décroître les performances, car l'accès à une case du tableau a ne se fait plus à la même itération mais n itérations plus loin.

do $i = 2, n$ (* do_1 *)
  $a(i) = b(i) + c(i)$ (* s_1 *)
  $d(i) = a(i-1) + e(i)$ (* s_2 *)
enddo



(a) boucle fusionnée

doall $i = 1, n$ (* do_1 *)
  $a(i) = b(i) + c(i)$ (* s_1 *)
enddoall
doall $i = 1, n$ (* do_2 *)
  $d(i) = a(i-1) + e(i)$ (* s_2 *)
enddoall



(b) boucle distribuée

Figure 4.1: un exemple de fusion/distribution de boucle avec les graphes de dépendances associés.

La *fusion* de boucles [46, 76, 5, 18] est la transformation inverse de la distribution et agrège plusieurs boucles pour n'en former qu'une seule. La *fusion par nom* [46] est une variante de la fusion qui tente de regrouper dans une même boucle les actions qui modifient les mêmes tableaux, espérant ainsi que l'utilisation de la hiérarchie mémoire sera meilleure. Nous allons voir dans le paragraphe suivant les critères de légalité de la fusion et de la distribution.

4.2 Légalité de la distribution et de la fusion

La définition suivante donne les conditions nécessaires et suffisantes pour la légalité d'une distribution de boucle. Pour l'instant, nous allons considérer que le RDG ne comporte pas de dépendance impliquant une structure de contrôle, ce sera l'objet de la suite de ce chapitre.

Définition 17 (Distribution d'une boucle) *Étant donnés une boucle et son graphe des dépendances de données réduit (RDG), une distribution donnée de cette boucle est légale si et seulement si*

1. *toutes les instructions impliquées dans un circuit de dépendances de donnée sont placées dans la même boucle ;*
2. *s'il y a une dépendance de s_1 vers s_2 et que s_1 et s_2 ne sont pas placées dans la même boucle, la boucle contenant s_1 est placée avant la boucle contenant s_2 dans le code généré ;*
3. *s'il y a une dépendance de s_1 vers s_2 indépendante de la boucle ou non portée et que s_1 et s_2 sont dans la même boucle, l'ordre textuel original entre s_1 et s_2 est conservé au sein de la boucle.*

Une distribution de boucle peut être modélisée sur le graphe des dépendances de donnée réduit (RDG) par une *partition*, c'est-à-dire un ensemble de groupes disjoints de sommets où chaque groupe représente des instructions devant être mises dans la même boucle. Avec la restriction précédente (le RDG ne comporte pas de dépendance impliquant une structure de contrôle), une partition ne concerne que les sommets représentant une action de la portion de code considérée. Une partition des sommets V d'un RDG $G = (V, E)$ donne un nouveau RDG $G_p = (V_p, E_p)$ où chaque sommet est un ensemble de sommets de l'ancien RDG et où il y a un arc entre en groupe c_1 et un groupe c_2 s'il y un arc $e = (u, v)$ tel que $u \in c_1$ et $v \in c_2$.

Étant donnée une partition, la génération du code correspondant doit suivre l'ordre topologique donné par le nouveau RDG pour que le code généré soit légal (condition 2 de la définition 17). La génération du code correspondant à un groupe de sommet doit suivre la condition 3 de la définition 17. Qu'en est-il de la condition 1 de cette même définition ? Tout d'abord, tout circuit de dépendances du RDG doit être inclus dans un groupe de sommets pour respecter ce circuit. Il manque cependant une restriction : il se peut qu'une partition crée un circuit de dépendances de donnée dans le nouveau RDG, rendant ainsi le code impossible à générer puisqu'il n'y a pas d'ordre topologique induit entre les composantes du circuit. Cette restriction nous donne la définition de la légalité d'une partition à un niveau k d'un RDG.

Définition 18 (Légalité d'une partition de distribution) *Étant donnés une boucle et son graphe des dépendances de données réduit (RDG), une* partition de distribution *de ce RDG est légale si et seulement si le nouveau RDG induit par la partition est sans circuit.*

L'ensemble des partitions légales d'un RDG permet donc de générer l'ensemble des codes légaux que nous pouvons obtenir par une distribution de boucle, en suivant bien sûr l'ordre topologique imposé par le graphe et l'ordre textuel sous-jacent quand cela est nécessaire lors de la génération du code.

En ce qui concerne la fusion de boucles, la principale difficulté est que les boucles soient compatibles entre elles, ce qui est résumé par la définition suivante.

Définition 19 (Fusion de boucles) *Soient deux boucles et leur graphes des dépendances de donnée réduits et leurs graphes des dépendances de contrôle correspondants. Ces deux boucles sont* compatibles*, c'est-à-dire qu'elles peuvent être fusionnées si et seulement si*

1. *elles sont exécutées selon les mêmes conditions d'un point de vue du flot de contrôle ;*
2. *les bornes des boucles sont compatibles ;*
3. *les dépendances de donnée sont respectées.*

Nous pouvons remarquer que dans l'absolu le respect de chacune de ces trois conditions est uniquement vérifiable à l'exécution. Néanmoins, nous pouvons trouver des conditions sur-approximées suffisant au respect de cette définition. La première condition est sur-approximée par le fait que les deux boucles ont le même ancêtre dans le graphe des dépendances de contrôle. En ce qui concerne la deuxième condition, la compatibilité des bornes de boucle, il suffit que celles-ci soient exactement identiques pour s'assurer du respect de la condition. Il est à noter que certaines transformations sont possibles pour rendre ces bornes de boucle identiques, comme par exemple le *décalage de boucle* (loop shifting), mais nous n'autorisons par de telles transformations ici [12]. Pour la troisième condition, pour n'importe quelle modélisation sur-approximée ou exacte des dépendances de données, elle est respectée si la fusion ne crée pas de dépendance de donnée portée par la nouvelle boucle allant dans le sens inverse de l'ordre d'exécution original des opérations [76]. Ce cas peut uniquement arriver quand il y a une dépendance de donnée non portée par une boucle entre deux instructions qui se trouvent chacune dans un boucle différente. Lors de la fusion, trois cas peuvent se produire : la dépendance reste non portée, la dépendance devient portée mais reste dans le sens de l'exécution des itérations ou la dépendance devient portée mais dans le sens inverse de l'exécution des itérations. C'est ce dernier cas qui est proscrit. Ce genre de dépendance dans un graphe des dépendances réduit est un arc qui empêche la fusion (*fusion preventing*).

Si nous fusionnons deux boucles dont l'une est parallèle et l'autre séquentielle, la boucle résultante sera séquentielle. Il y a donc des contextes où nous ne voudrons pas fusionner de telles boucles. En fait, la condition 1 et la condition 2, ainsi que la nature des boucles peuvent être rassemblées en un même attribut générique que nous appellerons le *type* de la boucle. Ainsi, si nous considérons un RDG qui représente des instructions de boucles différentes, nous pouvons attribuer à chaque sommet un type selon la boucle à laquelle l'instruction appartient, en choisissant selon le contexte ce que nous entendons par « type ».

De la même manière que pour la distribution, une fusion peut être modélisée par une partition des sommets du RDG. Ainsi, nous pouvons donner la définition générale d'une partition de fusion/distribution [18].

Définition 20 (Légalité d'une partition) *Étant donné un graphe des dépendances de données réduit (RDG), une* partition *de ce RDG est légale si et seulement si*

1. *le nouveau RDG induit par la partition est sans circuit ;*
2. *deux sommets connectés par un arc qui empêche la fusion ne sont pas dans le même groupe ;*
3. *deux sommets de types différents ne sont pas dans le même groupe.*

Cette définition donne les conditions générales permettant de définir toutes les partitions légales d'un RDG représentant une portion de code, donc d'obtenir tous les codes légaux résultant d'une série de fusions/distributions. Rappelons que tout ceci est défini avec la restriction que les seuls sommets du RDG reliés par des arcs sont des actions. Un algorithme comme celui d'Allen et Kennedy [2] exploite la distribution de boucle pour trouver du parallélisme et est optimal en terme de parallélisme au regard de la représentation des dépendances par niveau [23].

Une difficulté survient lorsque dans la boucle il n'y a pas que des instructions élémentaires, mais également des structures conditionnelles ou itératives. Par exemple, si la boucle que l'on veut distribuer contient une structure if avec deux instructions élémentaires à l'intérieur, dans quel cas avons nous le droit de distribuer ? En fait, cela revient à faire une distribution de if, sous des conditions assez similaires à la distribution de boucles.

Définition 21 (Distribution d'un if) *Étant donnés une structure conditionnelle if et son RDG, elle peut être distribuée si et seulement si le graphe induit par la partition du if est sans circuit.*

De la même manière, deux structures conditionnelles de type if peuvent être fusionnées sous des contraintes similaires à la fusion de boucles.

Définition 22 (Fusion de structures conditionnelles if) *Des structures conditionnelles if sont compatibles, c'est-à-dire qu'elles peuvent être fusionnées si et seulement si*

1. *elles sont exécutées selon les mêmes conditions d'un point de vue du flot de contrôle ;*
2. *les conditions des structures conditionnelles sont compatibles.*

Nous pouvons remarquer que la fusion de deux structures conditionnelles compatibles respecte toujours les dépendances de données, toujours sous la contrainte qui est que le RDG est uniquement composé de sommets qui représentent des actions. En fait, ceci est dû au fait que l'on ne change pas l'ordre d'exécution initial des instructions.

Par contre, à la différence d'un do, une structure conditionnelle de type if contrôle deux ensembles d'instructions distincts : les instructions exécutées lorsque la condition est vraie et celles exécutées lorsqu'elle est fausse. Lors de la distribution d'une telle structure, la sémantique d'exécution doit être respectée. Ainsi en présence d'un if comme celui de la partie (a) de la figure 4.2, les deux codes (b) et (c) peuvent être générés et sont corrects.

```
if (e) then
    s1
else
    s2
endif
```

(a)

```
if (e) then
    s1
endif
if (e) then
    (* rien *)
else
    s2
endif
```

(b)

```
if (e) then
    s1
endif
if (not(e)) then
    s2
endif
```

(c)

Figure 4.2: un exemple de distribution de structure conditionnelle.

Ainsi, si nous nous intéressons à la recherche d'une partition de distribution/fusion, puisque la condition pour qu'une distribution de structure conditionnelle if est la même que pour une distribution de boucle, la définition d'une partition reste la même que précédemment. En ce qui concerne la fusion de structures conditionnelles if, elle ne change pas non plus la définition de partition légale : c'est uniquement au moment de la génération du code que nous pouvons décider de fusionner ou non ces structures. Dans le paragraphe suivant, nous allons voir que les choses vont se compliquer en présence de dépendances de contrôle.

4.3 Flot de contrôle complexe

Lorsque nous transformons du code, nous cherchons à modifier l'ordre dans lequel les *actions* sont exécutées. La plupart des algorithmes de transformation de code opèrent sur des graphes des dépendances de données réduits où il n'y a des dépendances de données qu'entre les actions du code, comme au paragraphe précédent. En simplifiant, ces codes sont ceux que l'on appelle à *contrôle statique* [26]. Malheureusement, il arrive qu'il y ait également des codes où il y a des dépendances entre une expression contenue dans une structure de contrôle et une action. Ce type de code ne rendre pas dans le cadre des codes à contrôle statique et complique fortement la tâche de transformation du code. Considérons un exemple comme celui de la figure 4.3.

do $i = 2, n$ (* do_1 *)
    if $(a(i) == 0)$ then (* if_1 *)
        $a(i) = b(i) + c(i)$ (* s_1 *)
        $d(i) = a(i-1) + e(i)$ (* s_2 *)
    endif
enddo

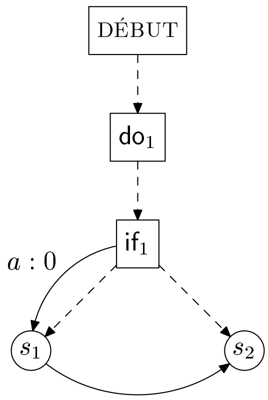


Figure 4.3: portion de programme où la distribution n'est pas possible à cause d'une dépendance impliquant une structure de contrôle.

Dans ce code, il y a une dépendance de donnée entre l'expression contenue dans la structure conditionnelle if_1 et l'action s_1. Ce genre de dépendance est très ennuyeuse pour une raison toute simple. Considérons une transformation de code comme la distribution que nous avons vue précédemment. La distribution va avoir comme effet de placer les actions dans différentes structures de contrôle résultantes. Il n'y a pas de duplication des accès à la mémoire au niveau des actions puisque celle-ci ne se trouvent toujours qu'à un seul endroit dans le code obtenu. Par contre, si nous considérons les expressions contenues dans un if ou un do, la distribution va avoir comme effet de dupliquer ces expressions dans toutes les nouvelles structures résultantes. L'évaluation de ces expressions est faite dans un certain état de la mémoire et détermine le flot de contrôle du programme. La duplication de ces expressions doit donc être autorisée uniquement si le nouveau flot de contrôle du programme est compatible avec le flot de contrôle original. Lorsque nous avons une anti-dépendance comme celle de la figure 4.3, partant de la structure conditionnelle if_1 et allant vers l'action s_1, nous pouvons voir que le résultat de l'évaluation de l'expression contenue dans le if peut ne pas être le même selon l'endroit où cette évaluation est faite : avant ou après s_1. Dans ce cas précis, si nous distribuons la boucle et que nous évaluons à nouveau l'expression contenue dans if_1, le code sera faux (présenté à la figure 4.4). En effet, la valeur de $a(i)$ utilisée dans if'_1 dans la deuxième boucle a été modifiée dans la boucle précédente et rend donc le code sémantiquement incorrect. Il est à noter que nous ne pouvons pas placer la boucle do'_1 avant la boucle do_1 car nous devons respecter la dépendance de donnée entre s_1 et s_2.

Afin de transformer la boucle englobante do_1 en plusieurs boucles parallèles par une

```
(* Attention, code sémantiquement faux ! *)
do i = 2, n                           (* do1 *)
   if (a(i) == 0) then                (* if1 *)
      a(i) = b(i) + c(i)              (* s1 *)
   endif
do i = 2, n                           (* do'1 *)
   if (a(i) == 0) then                (* if'1 *)
      d(i) = a(i − 1) + e(i)          (* s2 *)
   endif
enddo
```

Figure 4.4: distribution du code de la figure 4.3 donnant un code faux.

distribution et trouver un code qui soit légal, plusieurs approches ont été présentées dans la littérature, qui ont en commun la même idée sous-jacente : introduire de la mémoire supplémentaire pour détruire la dépendance de donnée qui pose problème.

4.3.1 La « if conversion »

La plus ancienne de ces approches, connue sous le nom de « *if conversion* » [4], consiste à rajouter une action d'affectation dans le corps de la boucle qui va stocker le résultat de l'évaluation de l'expression de la structure de contrôle dans une zone de mémoire temporaire de la taille adéquate, comme présenté à la figure 4.5. Ensuite, chaque instruction à l'intérieur de la structure conditionnelle va être transformée en instruction *gardée*, c'est-à-dire que l'instruction sera conditionnée par une garde qui sera l'évaluation du temporaire correspondant.

```
do i = 2, n                           (* do1 *)
   t(i) = (a(i) == 0)                 (* t1 *)
   if (t(i)) a(i) = b(i) + c(i)       (* s1 *)
   if (t(i)) d(i) = a(i − 1) + e(i)   (* s2 *)
enddo
```

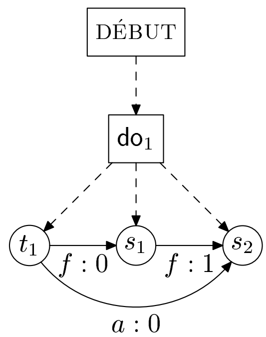


Figure 4.5: introduction d'un tableau temporaire pour stocker l'évaluation de l'expression de la structure conditionnelle pour le code de la figure 4.3.

Dans ce code, un tableau de dimension 1 et de taille n a été introduit pour stocker le résultat de l'évaluation de l'expression conditionnelle $a(i) == 0$, modifiant ainsi la dépendance qui ne part plus de la structure de contrôle. Une fois que le code est mis sous cette forme, la boucle peut être distribuée pour obtenir deux boucles parallèles, comme présenté dans la figure 4.6. Pour chacun de ces trois codes, nous avons associé le graphe des dépendances afin de visualiser l'effet des différentes étapes de transformation sur les dépendances de donnée et de contrôle.

La clé de l'utilisation de temporaires est de transformer une anti-dépendance qui sort d'une structure de contrôle en une dépendance de flot qui arrive sur cette structure. Ainsi,

```
doall i = 2, n                        (* do_1 *)
    t(i) = (a(i) == 0)                (* t_1 *)
    if (t(i)) a(i) = b(i) + c(i)      (* s_1 *)
enddoall
doall i = 2, n                        (* do_1'' *)
    if (t(i)) d(i) = a(i − 1) + e(i)  (* s_2 *)
enddoall
```

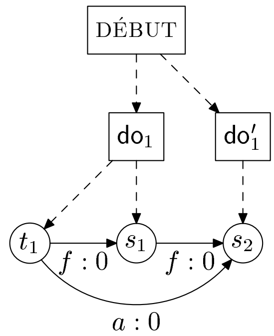


Figure 4.6: Distribution de la boucle grâce à l'introduction du tableau temporaire.

la liberté de transformation est plus grande car même si la structure est dupliquée, la valeur de l'évaluation de l'expression a été stockée, permettant ainsi sa réutilisation ultérieure. Une dépendance qui arrive sur une structure de contrôle est toujours une anti-dépendance par définition, puisqu'une expression dans une structure de contrôle ne modifie pas la mémoire. (Nous considérons ici que nous avons affaire à des langages de programmation « propres », comme par exemple le Fortran. Il est vrai qu'en C il est tout à fait possible de placer une affectation dans la condition d'un if ou dans des bornes de boucle.)

Ainsi, dans le code de la figure 4.6, nous pouvons en toute sécurité dupliquer la structure de contrôle if_1 puisque la valeur de sa condition a été évaluée au bon endroit, c'est-à-dire dans le bon contexte mémoire.

Il est à noter que le principe de la if conversion peut également s'appliquer à d'autres structures de contrôle, comme par exemple la boucle do. Par exemple, considérons le code suivant et sa version distribuée :

```
do i = a(1), n                (* do_1 *)
    a(i) = b(i) + c(i)        (* s_1 *)
    d(i) = a(i − 1) + e(i)    (* s_2 *)
enddo
```

```
t = a(1)                      (* t_1 *)
doall i = t, n                (* do_1 *)
    a(i) = b(i) + c(i)        (* s_1 *)
enddoall
doall i = t, n                (* do_1 *)
    d(i) = a(i − 1) + e(i)    (* s_2 *)
enddoall
```

Bien qu'elle soit utilisée dans de nombreux systèmes de compilation, il reste que cette approche a de nombreux inconvénients. Tout d'abord, la « if conversion » est une approche systématique, c'est-à-dire qu'elle ne cherche par à savoir si l'introduction d'un temporaire est vraiment nécessaire pour obtenir un code distribué, augmentant ainsi la taille de la mémoire utilisée. De plus, une fois que cette transformation a été effectuée, il n'est pas toujours facile de retrouver le code original. D'autre part, si le code a beaucoup d'instructions if imbriquées, les expressions qu'il faut stocker dans les temporaires deviennent rapidement très grosses. Il faut donc appliquer en plus des méthodes de simplification des expressions booléennes [4]. Pour finir, si nous avons une instruction if qui englobe une boucle, la if conversion va avoir comme effet de « pousser » le if sous la forme d'instructions gardées dans le corps de la boucle, pouvant ainsi modifier le nombre d'itérations effectuées, même si celles-ci sont vides de calculs effectifs.

4.3.2 L'approche de McKinley et Kennedy

Une approche plus récente, dédiée à la distribution de boucles, a été présentée par McKinley et Kennedy [36]. Elle tente de n'introduire les temporaires que quand c'est nécessaire au regard d'une partition donnée, c'est-à-dire selon un regroupement donné des actions au sein des boucles. Pour cela, ils utilisent les informations du graphe des dépendances de contrôle, à la différence de l'approche précédente. Ainsi, cette approche va considérer le graphe des dépendances (PDG, voir le §2.5.1, p. 20) et se décompose en trois étapes :

1. une partition est déterminée sur le graphe des dépendances, regroupant les instructions entre elles ;
2. le graphe est restructuré pour refléter la nouvelle organisation ;
3. un programme équivalent est généré qui respecte le graphe.

Avec leur approche, une partition est légale si et seulement si tout circuit du graphe des dépendances, c'est-à-dire un circuit composé de dépendances de données et/ou de dépendances de contrôle, est contenu dans une seule partition. En d'autres termes, le nouveau graphe après partitionnement, c'est-à-dire où les nouveaux sommets sont les groupes d'instructions, doit être sans circuit. D'après McKinley et Kennedy, cette condition est nécessaire et suffisante avec le type de partition qu'ils considèrent. Nous verrons plus loin que la définition d'une partition qu'ils utilisent restreint l'ensemble des partitions vraiment légales, puisque cette définition n'est pas la même que la définition 20 (ils considèrent les dépendances de contrôle lors de la recherche de circuits).

Si on reprend l'exemple donné à la figure 4.3, la partition donnant le même code qu'à la figure 4.6 est celle donnée par la figure 4.7.

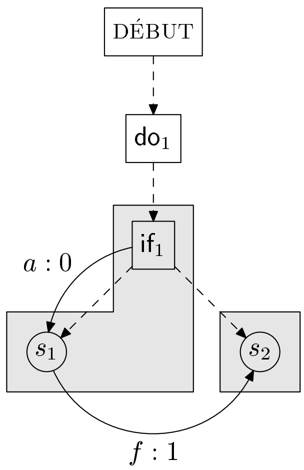
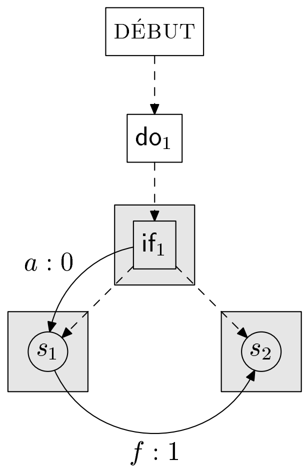


Figure 4.7: une partition du graphe des dépendances de la figure 4.3 avec l'approche de McKinley et Kennedy.

La condition qui provoque l'introduction d'un temporaire est qu'il y ait un arc de contrôle qui passe d'une partition à l'autre. Ainsi, l'arc de contrôle entre if_1 et s_2 provoque l'ajout d'un temporaire, pour finalement donner le code présenté à la figure 4.6. Cette approche est intéressante par rapport à la if conversion classique puisqu'elle permet de ne pas systématiquement introduire les temporaires mais uniquement quand c'est nécessaire. Il est important de noter que cette partition n'est pas la seule légale. Une autre solution consisterait à mettre if_1, s_1 et s_2 dans des groupes séparés, comme dans la figure 4.8.




```
doall i = 2, n                  (* do1 *)
   t(i) = (a(i) == 0)           (* t1 *)
enddoall
doall i = 2, n                  (* do1' *)
   if (t(i)) then               (* if1 *)
      a(i) = b(i) + c(i)        (* s1 *)
   endif
enddoall
doall i = 2, n                  (* do1'' *)
   if (t(i)) then               (* if1' *)
      d(i) = a(i − 1) + e(i)    (* s2 *)
   endif
enddoall
```

Figure 4.8: autre partition du graphe des dépendances de la figure 4.3 avec l'approche de McKinley et Kennedy et le code généré.

Cette solution exhibe trois boucles parallèles, en séparant le calcul de toutes les valeurs de l'expression contenue dans if_1 des autres boucles. Ce code nous permet une remarque importante. Dans l'absolu, cela n'a pas de sens de mettre le sommet if_1 dans un groupe. Une partition devrait uniquement être composée de groupes contenant des actions, puisque ce sont elles que l'on va placer dans des boucles différentes. Implicitement, le fait de mettre une structure de contrôle dans un groupe signifie avec l'approche de McKinley et Kennedy un calcul éventuel de l'expression de cette structure et son stockage dans un temporaire, tel que le calcul t_1 de la figure précédente. Ainsi, if_1 doit être considéré dans le partitionnement s'il y a une dépendance qui implique son expression. Bien qu'étant un sérieux progrès par rapport à la if conversion, cette particularité de leur approche met en avant un sérieux inconvénient : elle ne permet pas de trouver tous les regroupements d'instructions valides, notamment ceux sans temporaires. Utilisons un exemple pour illustrer ce fait. Nous allons prendre pour cela un extrait de la boucle présentée au chapitre 2 aux figures 2.4 et 2.7, pages 15 et 21. Cet extrait est présenté à la figure 4.9 avec son graphe des dépendances associé.

```
...
   do j = 2, n − 1                  (* do2 *)
      if (a(i, j + 1) == 0) then    (* if1 *)
         a(i, j) = b(i, j) + c      (* s1 *)
         d(i, j) = u(i, j) × e      (* s2 *)
      else
         d(i, j) = a(i, j + 1) + 1  (* s3 *)
      endif
   enddo
...
```

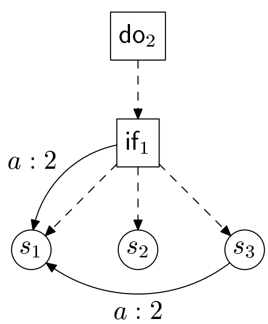


Figure 4.9: une boucle extraite du code de la figure 2.4, page 15, avec son PDG.

Examinons les partitions possibles de ce graphe avec leur approche. Afin d'obtenir du parallélisme, nous voulons faire en sorte que s_1 et s_3 ne soient pas dans la même partition. Ensuite, sans perte de généralité, nous allons considérer que nous mettons s_2 dans le même groupe que s_3 (s_2 n'est impliquée dans aucune dépendance, nous pouvons donc mettre cette instruction où nous voulons). Ainsi, sachant que s_2 et s_3 seront dans le même groupe, la question est de savoir où nous allons placer if_1. Nous avons trois possibilités, représentées par la figure 4.10 : soit nous laissons if_1 dans un groupe isolé (partition 1), soit nous le mettons avec s_1 (partition 2), soit nous le regroupons avec s_2 et s_3 (partition 3).

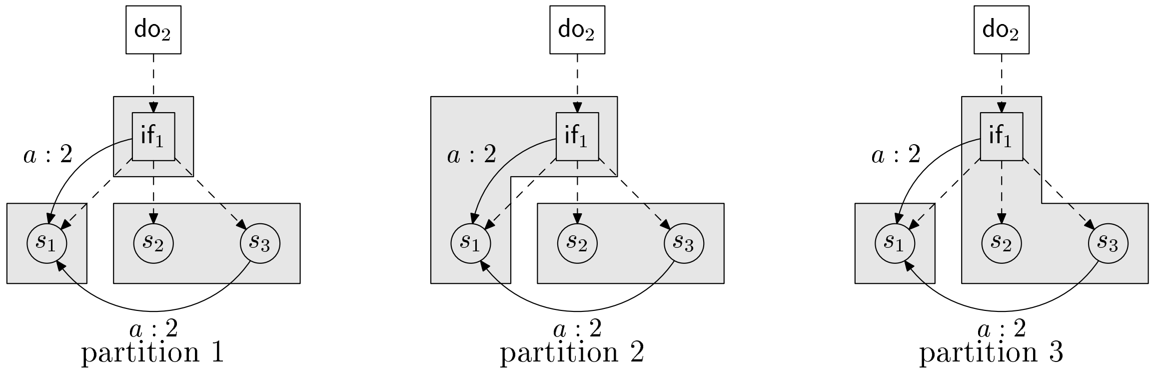


Figure 4.10: trois partitions du graphe des dépendances de la figure 4.9 avec l'approche de McKinley et Kennedy.

Avec la première partition, nous obtenons un code avec trois boucles parallèles et l'introduction d'un temporaire puisque des arcs de contrôle passent du groupe contenant if_1 aux deux autres groupes. La deuxième partition est illégale au sens de leur approche puisqu'il y a un circuit entre les deux partitions. La troisième partition donne deux boucles parallèles, toujours avec l'introduction d'un temporaire puisqu'il y a un arc de contrôle entre les deux partitions. En fait, nous pouvons voir qu'avec l'approche de McKinley et Kennedy, il n'est pas possible de trouver une partition de ce graphe représentant un code correct sans introduire de temporaire. Si nous considérons le code suivant :

```
doall j = 2, n − 1                  (* do2 *)
   if (a(i, j + 1) == 0) then       (* if1 *)
      d(i, j) = u(i, j) × e         (* s2 *)
   else
      d(i, j) = a(i, j + 1) + 1     (* s3 *)
   endif
enddoall
do j = 2, n − 1                     (* do'2 *)
   if (a(i, j + 1) == 0) then       (* if'1 *)
      a(i, j) = b(i, j) + c         (* s1 *)
enddo
```

nous voyons qu'il existe pourtant une solution de code sans temporaire qui est correcte, avec une boucle parallèle et une boucle séquentielle. Ce code n'est pas exprimable par une partition avec leur approche, ou plutôt, ce code serait exprimable par la partition 2, en oubliant le circuit qui rend la partition illégale. Ce code ressemble également à la partition 3 sauf que dans la partition 3, il y a forcément un temporaire qui est introduit. Tout cela pour dire que leur approche ne permet pas de trouver tous les codes légaux possibles, à cause de leur définition de la légalité d'une partition qui est trop restrictive. En voyant la partition 2, il découle que nous pourrions peut-être envisager de considérer les arcs de contrôle d'une autre façon et qu'ils ne devraient pas forcément toujours être respectés. C'est l'approche que nous allons exposer dans le paragraphe suivant, lorsque nous parlerons de la représentation que nous avons appelée le *graphe des dépendances mixte*. Cette nouvelle représentation prend en compte les dépendances de données et de contrôle dans un même graphe et permet de trouver tous les codes légaux possibles quand les seules transformations possibles sont la fusion et la distribution. Pour finir cet exemple sur l'approche de McKinley et Kennedy, nous pouvons dire que la démonstration selon laquelle le nombre de temporaires qu'ils introduisent au regard d'une partition est minimal est biaisée. En fait, il suffit de regarder la partition 3 précédente pour voir que leur approche introduit un temporaire alors qu'il n'y en a pas besoin pour que le code soit correct.

Un autre exemple de code où l'approche de McKinley et Kennedy ne permet pas de trouver toutes les partitions est celui présenté à la figure 4.11. Dans cet exemple, considérons la partition suivante contenant trois groupes : $\{(\mathsf{do}_1, s_1, s_3), (\mathsf{if}_1), (s_2)\}$. Cette partition est illégale selon leur approche à cause du circuit entre les groupes $(\mathsf{do}_1, s_1, s_3)$ et (if_1), créé par la dépendance de donnée entre s_1 et if_1 et la dépendance de contrôle entre if_1 et s_3. Pourtant, cette partition devrait permettre de générer le code (légal) de la figure 4.12 en ne considérant pas la dépendance de contrôle entre if_1 et s_3. Ce code ne correspond à aucune partition du graphe avec leur approche. Nous verrons dans la suite pourquoi nous n'avons pas besoin de respecter cettte dépendance et nous présenterons un nouveau type de graphe qui permettra de trouver tous les codes légaux.

```
do j = 2,n                (* do1 *)
   a(i) = a(i-1) + 1      (* s1 *)
   if (a(i) >= 0) then    (* if1 *)
      a(i-1) = 0          (* s2 *)
      b(i) = b(i-1) + 1   (* s3 *)
   endif
enddo
```

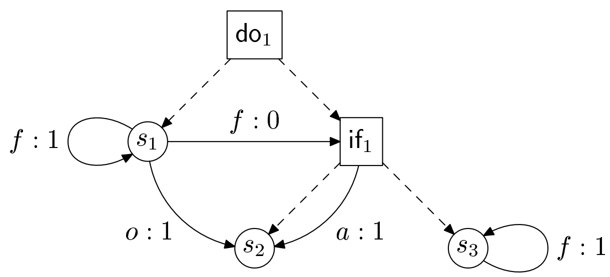


Figure 4.11: une autre boucle de code exemple avec son PDG.

Un autre apport certain du travail de McKinley et Kennedy mais relativement indépendant de leur approche est l'utilisation d'une logique à trois états qui va permettre de simplifier les expressions booléennes employées et de stocker de manière compacte des suites de décisions de contrôle [36]. Par exemple si nous considérons le code de la figure 4.13, avec son graphe des dépendances associé, nous voyons que soit l'action s_2 est exécutée, soit l'action s_3, soit aucune des deux. Nous avons donc besoin de stocker trois décisions possibles. Une logique à trois états permet de faire cela : les variables temporaires auront trois valeurs possibles, V, F et $\perp$. Cette approche permettra d'éviter l'explosion de la taille du code qu'on pourrait obtenir avec la « if conversion ». Nous y reviendrons plus loin lorsque nous présenterons notre algorithme (§4.7.3, p. 76). Un exemple de code obtenu avec une logique à trois états pour le code de la figure 4.13 est présenté plus loin à la figure 4.31, page 78.

```
do j = 2, n                  (* do1 *)
   a(i) = a(i − 1) + 1       (* s1 *)
   if (a(i) ≥ 0) then        (* if1 *)
      b(i) = b(i − 1) + 1    (* s3 *)
   endif
enddo
doall j = 2, n               (* do1 *)
      t(i) = (a(i) ≥ 0)      (* t1 *)
enddoall
doall j = 2, n               (* do1 *)
   if (t(i)) then            (* if1 *)
      a(i − 1) = 0           (* s2 *)
   endif
enddoall
```

Figure 4.12: code généré depuis une partition (illégale pour McKinley et Kennedy) du PDG de l'exemple 4.9.

```
do i = 2, n                  (* do1 *)
   if (a(i) > T) then        (* if1 *)
      a(i) = i               (* s1 *)
   else
      if (b(i) ≠ 0) then     (* if2 *)
         u = a(i − 1)/b(i)   (* s2 *)
      else
         c(i) = d(i) + 1     (* s3 *)
      endif
   endif
enddo
```

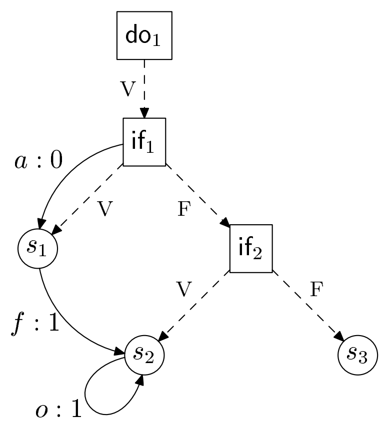


Figure 4.13: exemple de boucle où l'utilisation d'une logique à trois états permet de simplifier la réécriture du code. À la différence des autres graphes de dépendance, les arêtes de contrôle sont ici étiquetées pour mettre en évidence les décisions prises sur un chemin de contrôle.

4.4 Le graphe des dépendances mixte

Nous allons voir dans cette partie un nouveau type de graphe, le *graphe des dépendances mixte* qui est une extension du graphe des dépendances de donnée réduit (RDG) dans le cas de codes à contrôle non statique. Dans cette partie, nous ne parlerons pas de l'introduction de temporaires pour supprimer des anti-dépendances. Nous parlerons uniquement de la mécanique de ce nouveau graphe et notamment de certains de ses arcs représentant des *dépendances de chemin* qui sont des dépendances de donnée impliquant au moins une expression d'une structure de contrôle. Nous verrons plus loin (§4.6, p. 68) comment nous pouvons intégrer dans ce graphe une mécanique d'introduction de variables temporaires.

4.4.1 Définition d'un MDG

L'idée principale du *graphe des dépendances mixte* (MDG) est que chaque sommet de celui-ci va représenter un chemin dans le graphe des dépendances de contrôle, partant du sommet DÉBUT pour arriver sur une action. Nous allons voir que ce graphe est exactement un graphe des dépendances de donnée réduit (RDG) quand il n'y a pas de dépendance de donnée impliquant une structure de contrôle, c'est-à-dire lorsque nous sommes en présence d'un code à contrôle statique. La définition 20 d'une partition sera donc toujours valable. Le graphe des dépendances mixte est une extension du RDG dans le cas de codes à contrôle non statique. Nous allons donc montrer par la suite que la définition d'une partition valide est toujours la même.

Définition 23 (Graphe des dépendances mixte (MDG)) *Étant donné un graphe des dépendances $G = (V, E)$ d'une portion de programme avec un sommet source unique S, un* graphe des dépendances mixte *est un graphe orienté $G_a = (V_a, E_a)$ avec V_a les sommets et E_a les arcs. Chaque sommet de V_a est un chemin de contrôle de G partant du sommet S et arrivant sur une action de V. Si il existe une dépendance de donnée entre deux sommets X et Y de G, cette dépendance est reportée entre les deux sommets (représentant des chemins) de G_a qui contiennent respectivement X et Y. L'étiquetage des arêtes correspondant à des dépendances de données dans le graphe des dépendances est conservé sur la ou les arêtes correspondantes dans le MDG.*

Chaque arc de E_a représente une dépendance de donnée entre deux sommets V_a, c'est-à-dire entre deux sommets dans deux chemins de S à une action de G. Il y a exactement autant de sommets dans V_a que d'actions dans V. La définition précédente n'est valable qu'avec la restriction qui est que nous ne considérons que des codes sans branchement. D'autres part, nous négligeons les arcs du graphe de contrôle qui partent d'un sommet et qui arrivent sur ce même sommet, comme par exemple sur une boucle `do`. Ces deux conditions font qu'il est certain qu'étant donnée une action, il existe un unique chemin entre le sommet S (le sommet source de la définition précédente) et cette action dans le graphe des dépendances de contrôle. Nous pourrions étendre cette définition au cas général en considérant que chaque sommet du graphe des dépendances mixte représente un *ensemble* de chemins, mais nous n'aborderons pas cette extension ici. La présence d'un sommet source unique permet de se concentrer uniquement sur une portion de programme à une profondeur donnée. Ainsi, si nous considérons une portion de code qui n'est pas contenue dans une structure de contrôle, le sommet source sera le sommet DÉBUT. Par contre, si nous considérons du code contenu dans une structure de contrôle, le sommet source sera cette structure de contrôle.

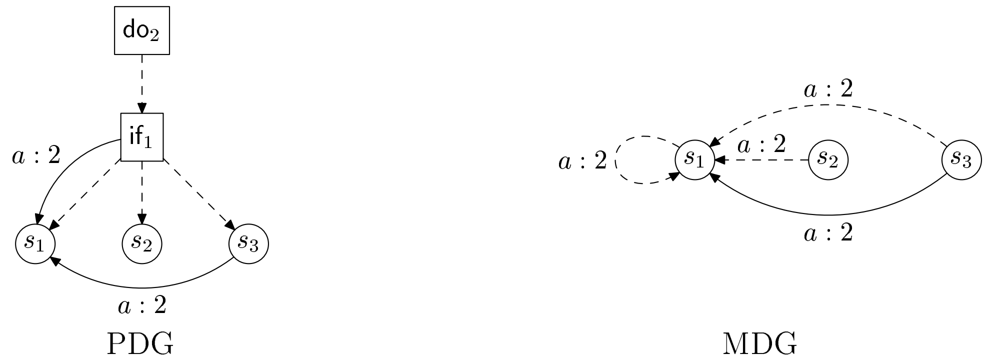


Figure 4.14: le MDG correspondant au PDG de la figure 4.9.

Dans un graphe des dépendances mixte, nous allons considérer deux types de dépendances : les dépendances de données entre deux actions du graphe des dépendances correspondant et les dépendances de donnée impliquant une structure de contrôle que nous appellerons *dépendances de chemin*. Si nous reprenons l'exemple du code présenté à la figure 4.9, nous obtenons selon la définition le graphe des dépendances mixte présenté dans la figure 4.14. Pour plus de clarté, nous avons également représenté à nouveau le graphe des dépendances (PDG) de cette portion de code. Dans le MDG, le sommet s_1 représente le chemin $\boxed{\mathsf{do}_2}\to\boxed{\mathsf{if}_1}\to\boxed{s_1}$, le sommet s_2 le chemin $\boxed{\mathsf{do}_2}\to\boxed{\mathsf{if}_1}\to\boxed{s_2}$ et le sommet s_3 le chemin $\boxed{\mathsf{do}_2}\to\boxed{\mathsf{if}_1}\to\boxed{s_3}$. Les arcs que nous avons représentés par des arcs pleins sont des dépendances de données. Les arcs que nous avons représentés en pointillés sont des anti dépendances de chemin impliquant l'expression $a(i, j+1) == 0$ de la structure de contrôle. Selon notre définition, trois arcs sont des dépendances de chemin et représentent une dépendance entre l'expression $a(i, j+1) == 0$ qui se trouve sur le chemin menant à chacune des actions s_1, s_2 et s_3.

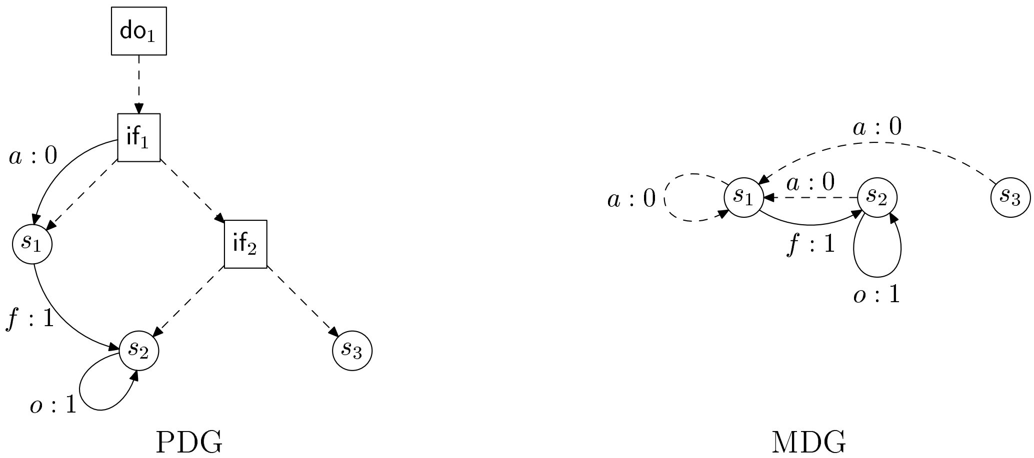


Figure 4.15: le MDG associé au PDG de la figure 4.13.

Pour donner un autre exemple, nous avons également représenté le MDG du code de la figure 4.13, avec son PDG associé, dans la figure 4.15. De manière analogue, il y a également trois arcs représentant une dépendance de chemin entre l'expression $a(i) > T$ et les trois actions s_1, s_2 et s_3. Le sommet s_1 représente le chemin $\boxed{\mathsf{do}_1}\to\boxed{\mathsf{if}_1}\to\boxed{s_1}$, le sommet s_2 le chemin $\boxed{\mathsf{do}_1}\to\boxed{\mathsf{if}_1}\to\boxed{\mathsf{if}_2}\to\boxed{s_2}$ et le sommet s_3 le chemin $\boxed{\mathsf{do}_1}\to\boxed{\mathsf{if}_1}\to\boxed{\mathsf{if}_2}\to\boxed{s_3}$.

Les dépendances de chemin permettent de prendre en compte le fait que si une expression contenue dans une instruction de contrôle (comme un `do` ou un `if`) est le puits d'une dépendance de flot, l'expression et toutes ses duplications par une éventuelle distribution doivent être exécutées **après** la modification d'une des composantes de l'expression par la source de la dépendance de flot. Par définition, une expression contenue dans une structure de contrôle ne peut pas être la source d'une dépendance de flot puisque l'expression ne modifie pas les valeurs qu'elle utilise. Une dépendance de flot de ce type va donc contraindre les codes valides en empêchant parfois une distribution à cause d'un circuit ou en rendant une boucle séquentielle.

Une anti-dépendance due à une expression contenue dans une instruction de contrôle contraint également les codes valides : la valeur de l'expression doit être calculée **avant** la modification d'une de ses composantes par le puits de la dépendance. Quand cette expression est dupliquée par une distribution de boucle do ou de if, toutes les boucles distribuées contenant l'expression doivent être exécutées **avant** la boucle contenant le puits de la dépendance, ou si la source est dans la même boucle que le puits, l'organisation originale des instructions source et puits doit être préservée d'un point de vue du contrôle.

4.4.2 Partition du MDG et génération de code

De même que pour le RDG, une partition du MDG définit un regroupement des actions au sein de nouvelles boucles. Ainsi, la partition du MDG de la figure 4.14 donnée par la figure 4.16 est une partition valide (nous verrons plus bas pourquoi cette partition est valide). De même, la partition du MDG de la figure 4.15 donnée par la figure 4.17 est une partition valide.

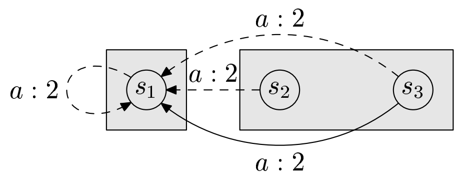


Figure 4.16: une partition légale du MDG de la figure 4.14.

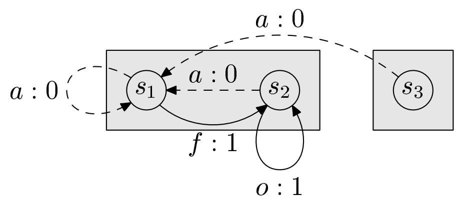


Figure 4.17: une partition légale du MDG de la figure 4.15.

Une partition est définie pour un certain niveau k. Typiquement, pour un nid de boucles imbriquées, nous allons commencer par définir une partition au niveau 1, puis chaque groupe de la partition va définir un nouveau MDG avec des chemins raccourcis et éventuellement des dépendances de chemin enlevées si elles n'existent plus (si la source ou le puits de la dépendance ne sont plus sur le chemin ou si la dépendance de donnée est portée comme c'est le cas pour les dépendances de données classiques) et des dépendances de données qui disparaissent de manière naturelle. Chaque groupe de la partition définit bien un nouveau

MDG puisque le sommet source est unique (la boucle contenant les instructions du groupe). Sans perte de généralité, nous pouvons donc nous contenter de raisonner à un certain niveau k puisque tout ce que nous allons énoncer marchera récursivement.

Ainsi, étant donnée une partition valide d'un MDG, examinons la façon dont nous allons générer le code. Tout d'abord, à l'intérieur d'un groupe, le code est généré de la même façon que pour le RDG : nous devons suivre l'ordre textuel original sous-jacent pour les dépendances de donnée indépendantes des boucles. En ce qui concerne les dépendances de chemin, celles-ci sont naturellement respectées si nous suivons l'ordre textuel sous-jacent et surtout la même imbrication des instructions (si nous respectons l'ordre des instructions sur le chemin). Nous partons du principe que nous n'autorisons pas la distribution des structures conditionnelles if, bien que cela soit possible en redéfinissant au sein de chaque groupe une « partition » valide de distribution des if.

Le code pour les différents groupes d'une partition est généré suivant l'ordre topologique imposé par les arcs entre les différents groupes du graphe induit par la partition. Nous avons vu juste au dessus qu'à l'intérieur d'un groupe, les dépendances de chemin étaient naturellement respectées et donc n'imposaient pas de contrainte particulière si nous respections l'ordre textuel original. Par contre, nous allons voir que le respect des dépendances de chemin par la génération du code suivant l'ordre topologique entre les groupes de la partition est une condition nécessaire et suffisante à la validité du code généré depuis la partition.

Lemme 6 (Dépendances de chemin entre des groupes) *Étant donnés deux groupes de sommets reliés par une dépendance de chemin, le code généré est légal si et seulement s'il suit l'ordre topologique induit par la dépendance.*

Preuve : soit A le premier groupe et B le second. S'il y a une dépendance de chemin entre A et B, il y a deux possibilités : 1) il y a une dépendance de flot entre une action qui est à la fin d'un chemin d'un sommet de A et une expression contenue dans un chemin d'un sommet de B, ou 2) il y a une anti-dépendance entre une expression contenue dans un chemin d'un sommet de A et une action qui est à la fin d'un chemin d'un sommet de B. Dans ces deux cas, si le groupe B est exécuté avant le groupe A, la dépendance de donnée est violée (la valeur de l'expression est potentiellement fausse). Donc respecter la dépendance de chemin est une condition nécessaire et suffisante à l'obtention d'un code légal. □

Ainsi, une partition légale d'un MDG suit exactement la même définition qu'une partition légale d'un RDG (définition 20). Dans cette approche sans introduction de temporaires, tout se passe comme dans le cas des distributions de do et de if lorsqu'il n'y a pas de dépendances de données impliquées dans une structure de contrôle. La génération du code est la même et les dépendances de chemin garantissent qu'une évaluation se fait dans le bon ordre par rapport aux actions qui la modifie (une légère différence est que les dépendances de chemin ne doivent pas être prises en compte dans un groupe pour la génération de code). Ainsi, par rapport à l'approche de McKinley et Kennedy présentée précédemment, nous autorisons implicitement le *recalcul* des expressions alors que leur approche impose l'introduction d'un temporaire. Nous verrons dans la partie suivante (§4.6, p. 68) que nous autorisons les deux possibilités : recalcul ou stockage dans un temporaire.

Définition 24 (Légalité d'une partition d'un MDG) *Étant donné un graphe des dépendances mixte (MDG), une* partition *de ce MDG est légale si et seulement si*

1. *le nouveau MDG induit par la partition est sans circuit ;*

2. *deux sommets connectés par un arc qui empêche la fusion ne sont pas dans le même groupe* (*voir* *§4.2, p. 48*) *;*
3. *deux sommets représentant deux structures de contrôle de types différents ne sont pas dans le même groupe.*

L'ensemble des partitions légales d'un MDG permet de trouver l'ensemble des codes légaux quand la distribution et la fusion sont les deux seules transformations possibles.

Les deux partitions des figures 4.16 et 4.17 sont donc des partitions valides selon la définition précédente. Les codes obtenus avec ces partitions sont présentés à la figure 4.18. Le premier de ces codes, le code présenté en (a) est celui que nous avions déjà vu lorsque nous avons parlé de l'approche de McKinley et Kennedy. Rappelons nous que ce code ne correspondait à aucune partition avec leur approche. Nous voyons qu'ici la partition correspondant à ce code est très naturelle. Nous avons une boucle parallèle qui contient s_2 et s_3 et une boucle séquentielle qui contient s_1. Cette boucle est séquentielle à cause de l'anti-dépendance de donnée entre l'expression $a(i, j + 1) == 0$ et s_1 portée par la boucle. Le deuxième code présenté en (b) contient une boucle parallèle contenant s_3 ainsi qu'une boucle séquentielle contenant s_1 et s_2. Cette boucle est séquentielle à cause de la dépendance portée par la boucle et nous avons du laisser s_1 et s_2 dans la même boucle à cause du circuit entre s_1 et s_2 composé d'une dépendance de donnée et d'une dépendance de chemin.

```
doall j = 2, n-1                (* do_2 *)
   if (a(i, j+1) == 0) then     (* if_1 *)
      d(i, j) = u(i, j) × e     (* s_2 *)
   else
      d(i, j) = a(i, j+1) + 1   (* s_3 *)
   endif
enddoall
do j = 2, n-1                   (* do'_2 *)
   if (a(i, j+1) == 0) then     (* if'_1 *)
      a(i, j) = b(i, j) + c     (* s_1 *)
   endif
enddo
```

(a) partition de la figure 4.16

```
doall i = 2, n                  (* do_1 *)
   if (not a(i) > T) then       (* if_1 *)
      if (not b(i) ≠ 0) then    (* if_2 *)
         c(i) = d(i) + 1        (* s_3 *)
      endif
   endif
enddoall
do i = 2, n                     (* do'_1 *)
   if (a(i) > T) then           (* if'_1 *)
      a(i) = i                  (* s_1 *)
   else
      if (b(i) ≠ 0) then        (* if'_2 *)
         u = a(i-1)/b(i)        (* s_2 *)
      endif
   endif
enddo
```

(b) partition de la figure 4.17

Figure 4.18: deux codes correspondant respectivement aux partitions des figures 4.16 et 4.17.

4.5 Algorithme de parallélisation

Pour illustrer l'utilisation du graphe des dépendances mixte, nous allons présenter un algorithme de parallélisation directement inspiré de l'algorithme d'Allen, Callahan et Kennedy [2]. Au lieu d'utiliser un RDG comme précédemment, cet algorithme va utiliser un MDG. Nous verrons plus loin (§4.7, p. 74) une légère modification de cet algorithme dans lequel nous introduirons des variables temporaires telles que celles de l'approche de McKinley et Kennedy sauf que nous utiliserons les dépendances de chemin du MDG pour décider quand nous introduirons ces variables.

Le principe général de cet algorithme est la décomposition du graphe en composantes fortement connexes pour un niveau k, c'est-à-dire la création d'une partition légale sans circuit pour ce niveau k. Ensuite, l'algorithme poursuit récursivement en considérant séparément au niveau $k+1$ les groupes de la partition du niveau k, et ainsi de suite. Le niveau est le nombre de boucles englobant le code représenté par le MDG, sachant que le sommet source DÉBUT correspond à une boucle au niveau 0. Donc, si on travaille à un niveau k, ceci veut dire que le sommet source du MDG est une boucle au niveau $k-1$. Les sommets fils du sommet DÉBUT aurons donc un niveau 1, et ainsi de suite.

Soit $G = (V, E)$ le MDG en entrée de l'algorithme, k le niveau courant et n_G la profondeur maximale du MDG G. Cette profondeur peut être déterminée en prenant la longueur maximale (en terme de nombre d'instructions do) d'un chemin représenté par un sommet du MDG. Chaque arc de E sera étiqueté par son niveau de dépendance que nous noterons l, ainsi que par sa profondeur que nous noterons p et qui sera la profondeur minimale entre la profondeur de l'expression source et la profondeur entre l'expression puits (par profondeur nous entendons nombre de boucles englobantes, sachant qu'une expression dans une borne de boucle d'un do est considéré comme étant englobée par ce do). Le niveau est également en quelque sorte une « profondeur » mais c'est la profondeur courante de l'algorithme, c'est-à-dire la profondeur des premiers sommets fils du MDG.

Nous noterons G_k le MDG où toutes les dépendances telles que $l < k$ $(l \neq 0)$ et $p < k$ ont été enlevées. G correspond à G_0. Passer de G_k à G_{k+1} revient à raccourcir les chemins de G_k. Nous devrons donc supprimer certaines dépendances dont la source ou le puits n'est plus sur un des chemins du graphe (c'est la raison de la condition $p < k$)

Une description de l'algorithme est présentée à la figure 4.1. Le premier appel à la procédure est **partition-mdg**$(G, 1)$. La condition d'arrêt de cet algorithme récursif est que le niveau k considéré soit supérieur à la profondeur n_G. Dans les paragraphes suivants, nous allons détailler les différentes étapes de cet algorithme.

```
procédure partition-mdg(G, k)
    Si k > n_G, stopper
    Détermination du graphe G_k depuis G                      (* voir §4.5.1, p. 64 *)
    Calcul des composantes fortement connexes G_ki de G_k     (* voir §4.5.2, p. 64 *)
    Choix d'une partition des G_ki                            (* voir §4.5.3, p. 65 *)
    Écriture du code de chaque G_ki selon l'ordre topologique (* voir §4.5.4, p. 66 *)
    Pour chaque G_ki appel de
      partition-mdg(G_ki, k + 1)                              (* voir §4.5.5, p. 67 *)
fin de la procédure
```

Algorithme 4.1: algorithme d'Allen et Kennedy avec un MDG.

4.5.1 Détermination de G_k

La première étape de l'algorithme est la détermination du MDG G_k étant donné un MDG G_{k+1} : toutes les arêtes telles que $l < k$ et $l \neq 0$ sont enlevées, ainsi que les arêtes telles que $p < k$. Chaque sommet du MDG va avoir un certain type d'instruction sur son chemin au niveau k. Rappelons que le niveau sur un chemin est exprimé en fonction du nombre d'instructions do rencontrées, sachant que le sommet DÉBUT est considéré comme une boucle et est au niveau 0.

En prenant le MDG de la figure 4.20, correspondant au code de la figure 4.19, nous pouvons voir que le sommet s_0 est une action au niveau 1. Les autres sommets ont tous une boucle (la boucle do_1) au niveau 1. Les chemins représentés par ces sommets partagent le même sommet source DÉBUT (de niveau 0).

```
b(2,2) = 3                                  (* s0 *)
do i = 2,n                                  (* do1 *)
   do j = 2,n-1                             (* do2 *)
      if (a(i,j+1) == 0) then               (* if1 *)
         a(i,j) = b(i,j) + c                (* s1 *)
         d(i,j) = u(i,j) × e                (* s2 *)
      else
         d(i,j) = a(i,j+1) + 1              (* s3 *)
      endif
   enddo
   do j = 2,n-1                             (* do3 *)
      f(i,j) = f(i-1,j) + f(i,j+1) × 2      (* s4 *)
   enddo
enddo
```

Figure 4.19: code de la figure 2.4 du chapitre 2.

4.5.2 Calcul des composantes fortement connexes

La deuxième étape est le calcul des composantes fortement connexes du MDG. Cette étape peut être accomplie en un temps linéaire en le nombre de sommets et d'arcs du graphe par un algorithme présenté par Tarjan (voir une très bonne description dans [16]). Une composante fortement connexe d'un graphe orienté est un ensemble maximal de sommets et d'arcs contenus dans un circuit. Les régions fortement connexes, parfois appelées π-blocs [75], sont les blocs de base de la distribution de boucle puisqu'elles représentent la partition légale maximale du MDG, c'est-à-dire la partition contenant les plus petits groupes de sommets. Après cette étape, si les composantes fortement connexes sont ordonnées suivant l'ordre topologique, le code peut déjà être généré en donnant la distribution maximale du code représenté par le MDG. Nous verrons dans le paragraphe suivant (§4.5.3, p. 65) que nous ne sommes pas obligé d'utiliser cette distribution maximale mais que nous pouvons envisager d'autres partitions.

Chaque boucle peut être marquée comme étant parallèle ou séquentielle selon que la composante fortement connexe comporte un arc de niveau k ou non (une dépendance portée par la boucle ou non). À ce point, nous avons le nombre maximal de boucles parallèles si la

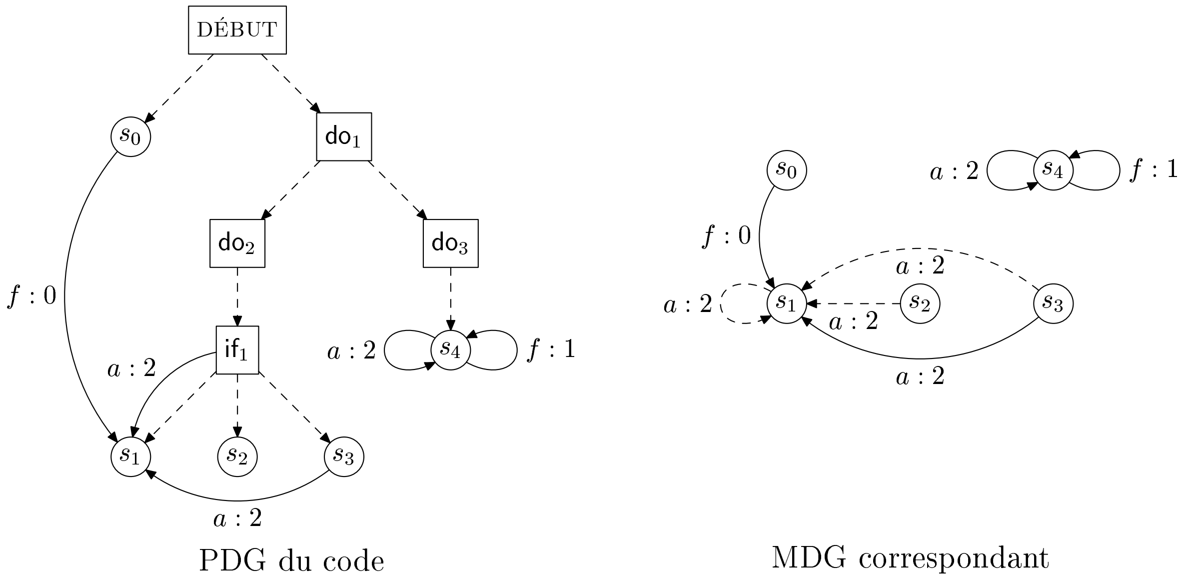


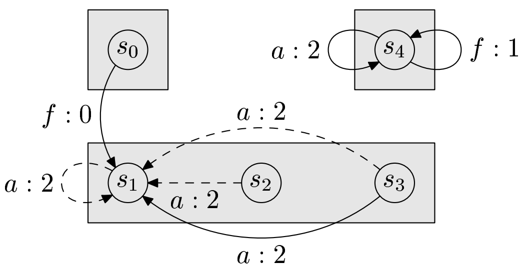
Figure 4.20: le PDG et le MDG du code de la figure 4.19.

distribution est la seule transformation possible [2, 23]. Prenons par exemple le MDG de la figure 4.20. Dans ce cas, les composantes fortement connexes sont les sommets eux-mêmes. Il est intéressant de noter qu'au niveau 1 considéré, une composante fortement connexe n'est pas une boucle, c'est le sommet s_0 qui représente l'instruction $b(2,2) = 3$. Trois composantes fortement connexes sont des boucles parallèles, issues de la boucle do_1 des sommets s_1, s_2 et s_3. Une composante fortement connexe est une boucle séquentielle issue de la boucle do_1 du sommet s_4. Cette boucle est séquentielle à cause de la dépendance de niveau 1 qui est portée par la boucle do_1.

4.5.3 Choix d'une partition

La troisième étape est une étape de « fusion », où plutôt de choix d'une partition selon un certain critère d'optimisation. Tout d'abord, des composantes fortement connexes peuvent être regroupées pour avoir moins de boucles résultant de la distribution (la même quantité de parallélisme peut éventuellement être exposée avec moins de boucles). De plus, des boucles séparées avant l'exécution de l'algorithme peuvent être fusionnées si nous voulons augmenter la taille des corps de boucle, à la condition que ces boucles soient compatibles et que la fusion soit autorisée. En fait, cette phase se ramène dans le cas général à un problème de fusion. Nous pouvons imaginer que nous attribuions un type à chaque composante fortement connexe, contenant des informations sur la boucle que représente la composante fortement connexe (si elle représente une boucle, sinon elle aura un type particulier). Ces informations porteront sur la borne de la boucle et sa nature (parallèle ou séquentielle). Nous n'avons pas besoin de nous soucier des conditions de contrôle selon lesquelles sont exécutées chacune des boucles puisque chaque chemin (sommet) du MDG a un sommet source qu'il partage avec tous les autres chemins. Il peut bien sûr y avoir dans le graphe formé par les composantes fortement connexes des arêtes empêchant la fusion. Ce problème est donc un problème de fusion avec plusieurs types (au moins 2) et des arêtes empêchant la fusion. Il a été montré que ce problème est NP-complet [18]. Par contre, nous ne travaillons pas toujours dans le cas général et il est souvent possible d'effectuer une recherche exhaustive, par exemple pour trouver une fusion maximale.

Revenons à l'exemple du MDG de la figure 4.20. Nous avions trouvé cinq composantes fortement connexes, correspondant aux sommets du MDG. Cependant, il peut être profitable de refusionner les trois composantes fortement connexes s_1, s_2 et s_3 pour ne représenter qu'une seule boucle parallèle, donnant ainsi la partition présentée à la figure 4.21. Nous ne pouvons pas fusionner le sommet s_0 avec une autre composante puisque celui-ci n'est pas une boucle au niveau 1. De même, si nous fusionnons s_4 avec la groupe de sommets s_1, s_2 et s_3, cela va avoir comme effet de rendre la boucle parallèle séquentielle en lui faisant porter une dépendance.



Partition au niveau 1

```
b(2,2) = 3          (* s0 *)
doall i = 2,n       (* do1 *)
   (* s1, s2, s3 *)
enddoall
do i = 2,n          (* do1 *)
   (* s4 *)
enddo
```

Code correspondant

Figure 4.21: partition au niveau 1 du MDG de la figure 4.20 ainsi que le code partiel associé.

4.5.4 Écriture du code

La quatrième étape est l'écriture du code dictée par la partition qui est donnée par les deux étapes précédentes. Le code correspondant à chaque groupe d'instructions est écrit en respectant l'ordre topologique entre les groupes, qui ne donne pas toujours un ordre total.

Le code pour chaque groupe va être réécrit en se fondant sur le chemin de chaque sommet, plus précisément en se servant de la deuxième instruction de ce chemin. Rappelons que la première instruction est partagée par tous les chemins et correspond à la structure de contrôle englobante au niveau supérieur. Le code va donc uniquement être écrit de manière partielle pour le niveau que nous sommes en train de traiter.

Pour chaque groupe, le code est généré pour chaque sommet en partant de la deuxième instruction du chemin correspondant jusqu'à ce que l'on rencontre une nouvelle instruction do (donc que l'on change de niveau) ou la fin du chemin. Avec notre principe de fusion, toutes les deuxièmes instructions des chemins des sommets d'un groupe sont identiques. Si cette deuxième instruction est une boucle parallèle, elle sera écrite avec la syntaxe particulière correspondant à une boucle parallèle dans le langage cible considéré. Nous utilisons la syntaxe doall.

La seule vraie contrainte est de respecter les dépendances de données au sein d'un groupe : pour cela, il suffit de respecter l'ordre textuel initial des instructions. Ensuite, nous pouvons partir du principe que nous gardons l'organisation initiale des instructions en terme de contrôle, ce qui garantit le respect des dépendances de chemin. Nous pouvons nous servir pour cela du graphe des dépendances de contrôle qui a déjà été construit pour permettre de générer le MDG. Dans le cas où deux sommets d'un groupe représentent une boucle qui n'était pas fusionnée initialement dans le MDG, nous fusionnons ces deux boucles lors de la génération du code : c'est bien pour cela qu'elles ont été mises dans le même groupe. Il y a également ici la possibilité d'effectuer des distributions ou des fusions éventuelles de structures conditionnelles if, à condition que cette transformation soit légale (voir les définitions 21 et 22). En fait, un principe plus général est de dire qu'il suffit de respecter les dépendances de niveau 0 dans un groupe et de les supprimer au fur et à mesure de la génération des structures de contrôle. Pour résumer, nous pouvons donc voir que la génération du code offre un certain nombre de libertés, du moment que les dépendances de données sont respectées. Celles-ci le sont automatiquement si l'ordre textuel et l'organisation des instructions initiaux sont respectés.

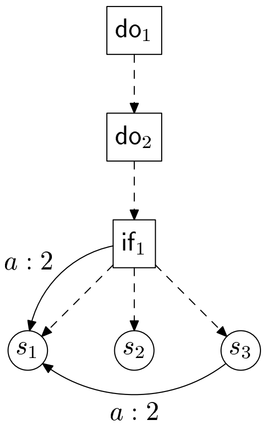
Pour revenir à la partition de la figure 4.21, une possibilité de génération de code est donnée dans cette même figure. L'autre possibilité aurait été de générer le code pour le groupe contenant le sommet s_4 avant le groupe contenant les sommets s_1, s_2 et s_3. Nous pouvons voir que le code généré est un code partiel, correspondant au niveau que nous sommes en train de traiter. Par exemple, pour le groupe contenant le sommet s_4, la deuxième instruction au niveau considéré est do_1 et l'instruction suivante sur le chemin est do_3 (qui est à un nouveau niveau) : la seule instruction écrite est donc do_1. Le principe est le même pour le groupe contenant s_1, s_2 et s_3.

4.5.5 Appel récursif

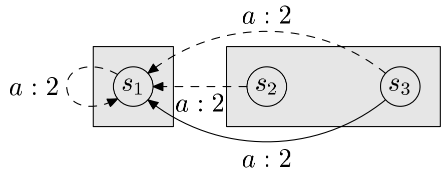
La fin de la procédure est un appel récursif à cette même procédure pour chaque groupe de la partition, avec un niveau supérieur. Ainsi, chaque groupe va être considéré comme un MDG à part entière au niveau supérieur, avec une instruction source unique pour chaque chemin de celui-ci au niveau directement inférieur.

Ainsi, avec la partition de la figure 4.21, l'algorithme va être appliqué récursivement aux différents groupes, avec k qui vaut 2. Voyons les différents appels récursifs :

- Le premier appel à la procédure va être pour le groupe contenant s_0 : la procédure ne s'exécute pas car k est strictement supérieur à 0, qui est la profondeur du MDG contenant s_0.
- Le deuxième appel est pour le groupe contenant s_1, s_2 et s_3 : dans ce cas la procédure va s'exécuter puisque la profondeur maximale du MDG correspondant est 2. Le MDG correspondant à ce groupe est présenté à la figure 4.22. Nous avons présenté à côté de celui-ci le PDG correspondant pour avoir une meilleure visualisation du chemin que représente chaque sommet du MDG. La partition que nous avons choisie est la même que celle que nous avions choisie pour la figure 4.16 (c'est le même code). La génération du code à été faite selon les principes énoncés plus haut. Nous avons donc deux groupes, l'un contenant s_1 (une boucle séquentielle) et l'autre contenant s_2 et s_3 (une boucle parallèle). L'ordre topologique nous impose d'écrire le groupe contenant s_2 et s_3 avant l'autre. Le code pour ce groupe est écrit en partant de l'instruction do_2 en deuxième position des deux chemins du groupe et en suivant les deux chemins jusqu'au bout. Nous avons respecté l'ordre textuel initial et l'organisation des instructions du graphe des dépendances de contrôle. Le principe est le même pour s_1. Il est à noter que cette boucle est séquentielle à cause de la dépendance entre l'expression de if_1 et l'action s_1. Les appels récursifs de la procédure pour les deux groupes de cette partition ne vont pas avoir d'effets puisque nous avons atteint la profondeur maximale pour chacun de ces sous graphes.
- Le troisième appel est pour le groupe contenant le sommet s_4. Comme précédemment, le PDG, le MDG et le code correspondant à l'unique partition possible sont présentés



PDG au niveau 2



Partition au niveau 2

```
doall i = 2, n                      (* do1 *)
   doall j = 2, n − 1               (* do2 *)
      if (a(i, j + 1) == 0) then    (* if1 *)
         d(i, j) = u(i, j) × e      (* s2 *)
      else
         d(i, j) = a(i, j + 1) + 1  (* s3 *)
      endif
   enddoall
   do j = 2, n − 1                  (* do'2 *)
      if (a(i, j + 1) == 0) then    (* if'1 *)
         a(i, j) = b(i, j) + c      (* s1 *)
      endif
   enddo
enddoall
```

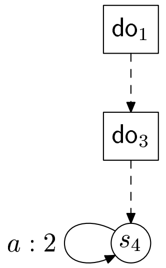
Code correspondant

Figure 4.22: PDG et MDG avec sa partition du groupe contenant s_1, s_2 et s_3 au niveau 2, ainsi que le code correspondant, toujours avec le code original de la figure 4.20.

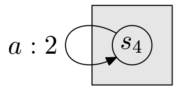
à la figure 4.23. Nous pouvons remarquer que la dépendance de niveau 1 qui existait précédemment sur s_4 a disparu à ce niveau. Après la génération du code de cette partition, l'algorithme est terminé. Finalement, le code global qui est obtenu est présenté à la figure 4.24. Ce code exhibe la quantité maximale de parallélisme que nous pouvons obtenir, en se fondant sur une représentation par niveau des dépendances. Il est à noter que ce n'est pas le seul code qu'il est possible d'obtenir : tout dépend des phases de fusion et de génération de code utilisées dans l'algorithme.

4.6 Variables temporaires

Nous avons présenté dans la partie précédente un algorithme de parallélisation qui se fonde sur le MDG. Nous allons voir dans cette partie que nous pouvons supprimer certains arcs de ce graphe à l'aide de variables temporaires. Les arcs qui pourront être supprimés sont ceux qui représentent des anti-dépendances de chemin. Nous verrons dans la dernière partie de ce chapitre (§4.9, p. 80) que nous pensons que nous pouvons supprimer toutes les



PDG au niveau 2



Partition au niveau 2

```
do i = 2, n                                  (* do1 *)
   do j = 2, n - 1                           (* do3 *)
      f(i,j) = f(i-1,j) + f(i,j+1) × 2       (* s4 *)
   enddo
enddo
```

Code correspondant

Figure 4.23: PDG et MDG avec sa partition du groupe contenant s_4 au niveau 2, ainsi que le code correspondant, toujours avec le code original de la figure 4.20.

anti-dépendances grâce à des temporaires.

4.6.1 Anti-dépendances et temporaires

Nous avons vu précédemment (§4.4.1, p. 59) qu'une anti-dépendance due à une expression contenue dans une instruction de contrôle contraignait les codes valides : la valeur de l'expression doit être calculée avant la modification d'une de ses composantes par le puits de la dépendance. Quand cette expression est dupliquée par une distribution de boucle do ou de if, toutes les boucles distribuées contenant l'expression doivent être exécutées avant la boucle contenant le puits de la dépendance, ou si la source est dans la même boucle que le puits, l'organisation originale des instructions source et puits doit être préservée d'un point de vue du contrôle. Il n'est pas toujours possible d'exécuter toutes les boucles contenant la source d'une telle dépendance avant le puits, cela dépend des éventuels circuits du MDG. Si il y a un circuit, la distribution est interdite. À la différence des dépendances de flot, les anti-dépendances de ce type vont pouvoir être levées en introduisant de la mémoire supplémentaire sous la forme de variables pour stocker l'évaluation de l'expression à une place appropriée et réutiliser le résultat de cette évaluation dans les éventuelles boucles distribuées qui seraient exécutées **après** la boucle qui modifie une des composantes de l'expression. C'est le même principe que la if conversion et la méthode de McKinley et Kennedy (voir le paragraphe sur cette méthode page 53).

Revenons à notre exemple de code de la figure 4.20 déjà tant présenté dans ce document et voyons ce qu'il est possible de faire en introduisant un temporaire. L'anti-dépendance de niveau 2 qui va de $a(i, j+1) == 0$ à s_1 doit retenir toute notre attention. Cette dépendance peut être supprimée par l'introduction d'un tableau de la taille appropriée pour stocker l'expression. Ainsi, nous pourrions obtenir plus de parallélisme et plus de liberté de partitionnement. L'introduction d'un temporaire revient à considérer le MDG de la figure 4.25. Notons que le PDG de gauche dans cette figure ne représente pas un « vrai code » mais est uniquement ici pour illustrer l'introduction du temporaire. Normalement, il devrait y avoir une dépendance entre t_1 et if$_1$ mais nous allons voir que nous n'avons pas besoin de

```
b(2,2) = 3                                 (* s0 *)
doall i = 2,n                              (* do1 *)
   doall j = 2,n-1                         (* do2 *)
      if (a(i,j+1) == 0) then              (* if1 *)
         d(i,j) = u(i,j) × e               (* s2 *)
      else
         d(i,j) = a(i,j+1) + 1             (* s3 *)
      endif
   enddoall
   do j = 2,n-1                            (* do2 *)
      if (a(i,j+1) == 0) then              (* if1 *)
         a(i,j) = b(i,j) + c               (* s1 *)
      endif
   enddo
enddoall
do i = 2,n                                 (* do1 *)
   do j = 2,n-1                            (* do3 *)
      f(i,j) = f(i-1,j) + f(i,j+1) × 2     (* s4 *)
   enddo
enddo
```

Figure 4.24: Code final obtenu par application de l'algorithme 4.1 sur le code de la figure 4.20.

cette dépendance puisque nous allons déterminer l'utilisation ou non du temporaire lors de la génération du code.

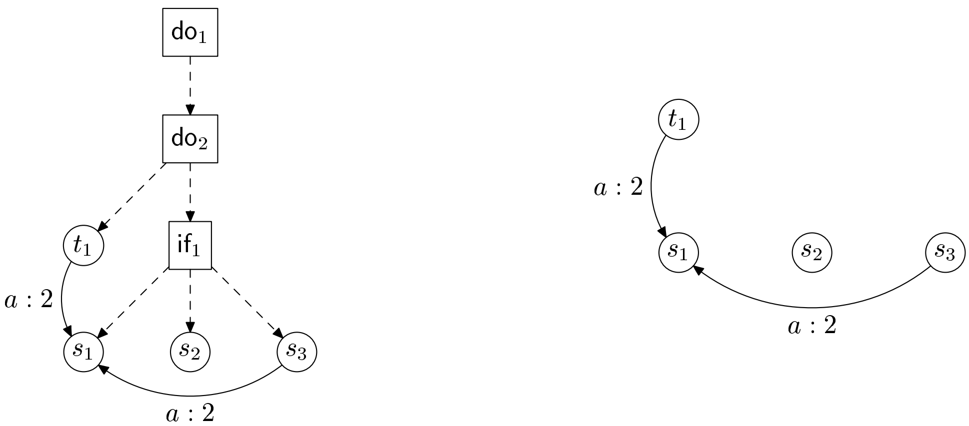


Figure 4.25: MDG transformé après l'introduction d'un temporaire.

4.6.2 Génération du code avec temporaires

Nous voyons que l'introduction de ce temporaire a comme conséquence la suppression de toutes les arêtes de chemin qui sont liées à cette anti-dépendance. Il y a a priori une dépendance entre le sommet représentant le temporaire et chaque utilisation de ce temporaire. Pourquoi cette dépendance n'est elle pas représentée sur le MDG ? Car nous ne savons pas encore et nous ne voulons pas savoir quels sont les sommets qui vont utiliser le temporaire, ou plus précisément, c'est la génération du code qui va déterminer si le temporaire est utilisé ou non. Le principe général est que tout code généré dans une boucle **avant** le calcul du temporaire **ne peut pas utiliser** le temporaire, que tout code généré dans une boucle **entre** le calcul du temporaire et le puits de la dépendance **peut utiliser** le temporaire mais n'est pas obligé de le faire et que tout code généré dans une boucle **après** le puits de la dépendance **doit utiliser** le temporaire sinon le code serait faux.

Il y a un petit choix à faire en ce qui concerne le chemin menant au puits de la dépendance : il y a un cas particulier où une dépendance de chemin n'est pas entre deux chemins mais sur le même chemin, comme par exemple dans l'exemple de la figure 4.20 (dépendance entre if_1 et s_1), et que cette dépendance est portée par la boucle (ce qui est le cas ici). Cette dépendance est une dépendance de chemin qui boucle sur un sommet du MDG (dans notre exemple du MDG de la figure 4.20 elle boucle sur le sommet s_1). La dépendance est de toute façon respectée par la structure du code, mais elle peut rendre la boucle séquentielle si le temporaire créé n'est pas utilisé. Pour simplifier, nous partons donc du principe que le puits de la dépendance utilise toujours le temporaire créé dans ce cas là si la dépendance est portée.

Le choix d'une partition se fait donc sans prendre en compte les dépendances de chemin qui ont été enlevées par le rajout du sommet correspondant à un temporaire. Par contre, nous devrons prendre en compte le temporaire dans le choix d'une partition et prouver qu'en faisant cela, le code est bien correct et qu'on a tous les codes. Ainsi, si nous revenons à la figure 4.25, nous pouvons choisir une partition maximale telle que chaque groupe soit un seul sommet. Cette partition illustre les différentes façons de générer le code. La seule contrainte temporelle du point de vue de la génération de code est que les instructions t_1 et s_3 soient exécutées avant l'instruction s_1. La figure 4.26 présente une première version du code.

```
doall j = 2, n − 1                 (* do2 *)
   t(i) = (a(i, j + 1) == 0)       (* t1 *)
enddoall
doall j = 2, n − 1                 (* do2 *)
   if (a(i, j + 1) == 0) then      (* if1 *)
      d(i, j) = u(i, j) × e        (* s2 *)
   endif
enddoall
doall j = 2, n − 1                 (* do2 *)
   if (not t(i)) then              (* if1 *)
      d(i, j) = a(i, j + 1) + 1    (* s3 *)
   endif
enddoall
doall j = 2, n − 1                 (* do2 *)
   if (t(i)) then                  (* if1 *)
      a(i, j) = b(i, j) + c        (* s1 *)
   endif
enddoall
```

Figure 4.26: première version du code pour la figure 4.25.

Nous aurions pu générer moins de boucles mais ce code a été choisi comme support d'explication. Tout d'abord, la première boucle parallèle est le calcul du temporaire. Il peut donc être utilisé dans toutes les boucles qui viennent après cette boucle. La deuxième boucle est une boucle correspondant au groupe contenant l'instruction s_2. Dans cette boucle, le temporaire n'est pas utilisé bien qu'il eût été possible de le faire. Par contre, dans la troisième boucle qui est le groupe correspondant à l'instruction s_3, nous avons choisi d'utiliser le temporaire, pour montrer que c'est possible de le faire. La dernière boucle est le groupe correspondant à l'instruction s_1. Nous utilisons dans ce cas le temporaire, bien qu'il eût été possible de ne pas le faire, au détriment du parallélisme. Considérons le code présenté à la figure 4.27 qui est toujours légal.

```
doall j = 2, n − 1                (* do2 *)
   t(i) = (a(i, j + 1) == 0)      (* t1 *)
enddoall
doall j = 2, n − 1                (* do2 *)
   if (not t(i)) then             (* if1 *)
      d(i, j) = a(i, j + 1) + 1   (* s3 *)
   endif
enddoall
doall j = 2, n − 1                (* do2 *)
   if (t(i)) then                 (* if1 *)
      a(i, j) = b(i, j) + c       (* s1 *)
   endif
enddoall
doall j = 2, n − 1                (* do2 *)
   if (t(i)) then                 (* if1 *)
      d(i, j) = u(i, j) × e       (* s2 *)
   endif
enddoall
```

Figure 4.27: deuxième version du code pour la figure 4.25.

Cette fois-ci, nous sommes obligé d'utiliser le temporaire pour s_2 qui se trouve après s_1, qui modifie une des composantes de l'expression $a(i, j+1) == 0$.

D'après ce que nous avons énoncé plus haut, il est également possible de savoir selon l'ordre topologique si le temporaire est utilisé ou non. Ainsi, s'il n'est pas utilisé, nous n'avons pas besoin de générer le code correspondant à la création du temporaire. Nous verrons néanmoins plus loin que la minimisation du nombre de temporaires tout en maintenant un parallélisme maximal est un problème NP-complet.

4.6.3 Introduction des nouveaux sommets dans le MDG

Toute expression contenue dans une structure de contrôle peut être stockée dans une variable ou un tableau temporaire de la taille et de la dimension appropriée. Pour chaque expression, un nouveau sommet peut être introduit dans le MDG (et le graphe des dépendances de contrôle associé), correspondant à la duplication du chemin partant du sommet source des chemins du MDG et menant à la structure de contrôle, en remplaçant cette structure de contrôle par le stockage de son expression dans un temporaire. Ainsi, si nous reprenons l'exemple précédent (figure 4.25), le sommet temporaire t_1 a été créé dans le graphe des dépendances de contrôle, juste avant l'évaluation de l'expression de if_1. Le chemin correspondant à ce sommet t_1 sera $\boxed{\mathsf{do}_1} \to \boxed{\mathsf{do}_2} \to \boxed{t_1}$. Chaque sommet temporaire introduit de cette façon a une dimension d qui lui est associée qui est la profondeur où se trouve l'action de stockage dans un temporaire (c'est le nombre de sommets do rencontrés sur le chemin complet qui va du sommet DÉBUT jusqu'à l'action de stockage dans le PDG). Cette dimension correspond à la dimension maximale dont nous avons besoin pour stocker l'évaluation de l'expression de la structure de contrôle.

La première remarque que nous pouvons faire est que nous pouvons montrer que toutes les dépendances de chemin issues d'une anti-dépendance peuvent être supprimées si nous utilisons un temporaire de la dimension maximale d. En fait, le point clé est que nous savons dire lors de la génération du code si nous sommes avant ou après l'expression qui modifie la source de l'anti-dépendance et donc nous savons si nous devons utiliser le temporaire ou recalculer l'expression. Nous allons cependant affiner cette introduction de temporaire pour permettre de contrôler et de diriger la recherche de partitions légales. Nous attribuons à chaque dépendance de chemin issue d'une anti-dépendance une dimension d correspondant à la dimension du temporaire maximal qui peut être créé éventuellement (cette dimension correspond à la profondeur du sommet temporaire éventuel) et une taille t (avec $-1 \leq t \leq d$) qui correspond à la dimension maximale que nous autorisons pour un éventuel temporaire visant à supprimer cette dépendance de chemin. Une valeur de t égale à -1 signifie que nous n'autorisons pas de temporaire pour cette dépendance de chemin. Ces deux valeurs vont permettre de déterminer à quel niveau k dans le déroulement de l'algorithme précédent un arc correspondant à une dépendance de chemin issue d'une anti-dépendance peut être supprimé grâce à un sommet temporaire (en d'autres termes, à quel niveau k la dimension du temporaire qui est introduit est suffisante pour supprimer cet arc).

Nous prenons comme restriction que le temporaire est indexé par les t dernières boucles internes englobantes et que ce temporaire ne peut être utilisé que dans ces t boucles. Comme toujours, nous supposons que l'imbrication des instructions en terme de contrôle est respectée.

Proposition 6 (Dimension du temporaire) *Soit un MDG et un arc représentant une anti-dépendance de chemin. Cet arc a un attribut d correspondant à la profondeur du sommet temporaire éventuel et un attribut t correspondant à la dimension maximale autorisée (avec $-1 \leq t \leq d$). Le temporaire à un dimension nécessaire et suffisante pour supprimer la dépendance de chemin dès que l'on partitionne à un niveau $k > l$ avec $l = d - t$.*

Preuve : puisque nous avons la relation $-1 \leq t \leq d$, il s'en suit que $0 \leq d - t \leq d + 1$. L'idée principale de cette démonstration est de savoir quelle « quantité » de l'évaluation de l'expression nous pouvons stocker dans la mémoire qui est disponible. Si $t = d$ ($l = 0$), toutes les expressions peuvent être évaluées et stockées dans la mémoire temporaire avant son utilisation : la dépendance de chemin peut donc être supprimée juste avant de commencer à partitionner (quand $k = 1 > l$). Si $t = -1$, nous obtenons une valeur de l qui est égale à $d + 1$, ce qui implique que nous ne pourrons jamais supprimer les arcs : nous n'avons pas de mémoire pour stocker l'évaluation de l'expression. Si $-1 < t < d$, cela veut dire que la mémoire temporaire ne peut pas stocker toutes les valeurs de l'évaluation de l'expression : il manque les $d - t$ premières dimensions de l'évaluation. Ceci implique que l'on pourra uniquement utiliser cette mémoire temporaire quand le niveau k de partitionnement sera strictement supérieur à $d - t$. $\square$

4.7 Modification de l'algorithme de parallélisation

Dans cette partie, nous présentons le modification de l'algorithme de parallélisation 4.1 pour permettre l'utilisation de temporaires. Étant donné un MDG, un travail préliminaire est d'attribuer à chaque expression contenue dans une structure de contrôle une dimension maximale t du temporaire autorisée. Puis, cette valeur va être reportée sur chaque arc représentant une dépendance de chemin issue d'une anti-dépendance, ainsi que la profondeur d de l'emplacement éventuel du temporaire.

4.7.1 Détermination de G_k

La première phase de l'algorithme qui est la détermination du MDG G_k étant donné un MDG G_{k+1} est modifiée de la façon suivante : les arcs de chemin issus d'une anti-dépendance avec des attributs d et t tels que $k > (d-t)$ sont supprimés. Un nouveau sommet est ajouté dans le MDG correspondant au calcul du temporaire, en reportant de manière naturelle toutes les dépendances de données impliquant l'expression évaluée dans ce calcul. On dit alors que le temporaire est *activé*.

Le reste de l'algorithme marche de la même façon. La génération du code est telle que nous l'avons décrite dans l'ancien algorithme avec en plus la prise en compte des temporaires (décrite à la page 70).

4.7.2 Privatisation des temporaires

Lorsque le temporaire est activé, celui-ci peut être (et doit être) *privatisé* pour les $d-t$ boucles englobantes si elles sont parallèles, par exemple avec la directive new d'HPF [31] ou son équivalent dans d'autres langages à base de directives [66, 70].

Ainsi, dans le code de la figure 4.28, l'anti-dépendance entre la borne supérieure du do et l'action s_1 peut être cassée par un temporaire. Nous avons fixé la dimension maximale du temporaire autorisée à $t_1 = 1$, ce qui correspond à un tableau de dimension 1. L'activation éventuelle du temporaire se fait à profondeur $d = 2$, c'est-à-dire juste avant le troisième do imbriqué. Lors de l'application de notre algorithme de parallélisation, les arcs peuvent être enlevés pour une partition au niveau $k = 2$, puisque $d - t_1 = 1$ est strictement inférieur à $k = 2$. Puisque nous avons t_1 qui est strictement inférieur à k, nous pouvons et devons privatiser le temporaire pour les boucles englobantes (si elles sont parallèles), celui-ci n'ayant pas une dimension égale à la profondeur à laquelle il se trouve. Si nous ne privatisions pas le temporaire, nous créerions une dépendance de sortie qui aurait comme effet de rendre la boucle séquentielle. Le MDG correspondant à l'introduction du temporaire ainsi que le code généré sont présentés dans la figure 4.29.

do $i = 1, n$
    do $j = 2, n$
        do $k = 1, a(i, j+1, 1)$ (* do$_3$ *)
            $a(i, j, k) = b(i, j, k) + a(i, j-1, k)$ (* s_1 *)
            $c(i, j, k) = a(i, j, k)^2$ (* s_2 *)
        enddo
    enddo
enddo

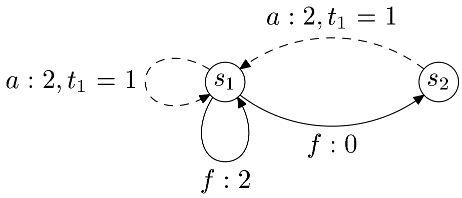
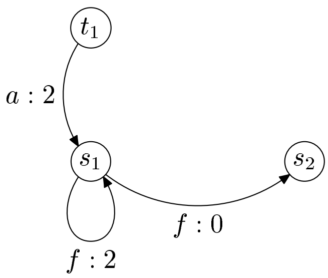


Figure 4.28: un exemple de code avec son MDG. L'anti-dépendance entre do$_3$ et s_1 va pouvoir être cassée en introduisant un temporaire.



doall $i = 1, n$ (* nst_0 est privatisé pour cette boucle *)
    doall $j = 2, n$
        $nst_0(j) = a(i, j+1, 1)$ (* t_1 *)
    enddoall
    do $j = 2, n$
        doall $k = 1, nst_0(j)$
            $a(i, j, k) = b(i, j, k) + a(i, j-1, k)$ (* s_1 *)
        enddoall
    enddo
    doall $j = 2, n$
        doall $k = 1, nst_0(j)$
            $c(i, j, k) = a(i, j, k)^2$ (* s_2 *)
        enddoall
    enddoall
enddoall

Figure 4.29: le MDG et le code généré correspondant au code de la figure précédente (4.28) où nous avons introduit un temporaire au niveau $k = 2$, avec $d - t_1 = 2 - 1 < 2$. Le temporaire peut être privatisé pour 1 boucle englobante ($d - t_1 = 1$).

Tailles des temporaires. Nous savons quelle *dimension* de temporaire est nécessaire pour stocker les conditions des structures de contrôle, mais nous ne savons pas encore quelle *taille* nous devons utiliser pour ces tableaux temporaires. À notre connaissance, il n'y a nulle part dans la littérature trace d'une méthode permettant de déterminer la taille du temporaire à utiliser. Une première possibilité est de laisser cette charge à l'utilisateur. Il se peut également que les bornes de boucle entourant l'initialisation du test soient évaluables à la compilation, auquel cas, il est facile de savoir quelle taille allouer pour le temporaire. Une troisième possibilité est d'allouer de la mémoire pour le temporaire au moment de l'exécution avec une procédure système comme allocate. Par exemple, la génération du code avec un temporaire pourrait se faire de la façon suivante :

```
(* nous allons utiliser un temporaire t1 *)
allocate(t1, ub − lb + 1)
do i = lb, ub
   t1(i) = ...
   ...
enddo
deallocate(t1)
```

4.7.3 Temporaires et recouvrement des tests.

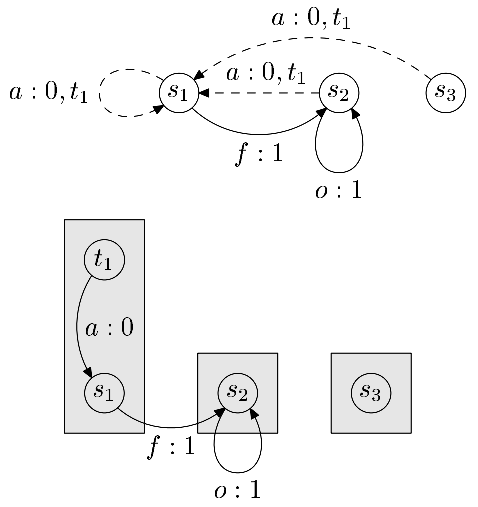
Plutôt que de ne permettre l'introduction de temporaires que pour les expressions qui sont source d'une anti-dépendance, nous pouvons très bien autoriser le stockage de n'importe quelle expression contenue dans une structure de contrôle dans un temporaire. Quel est l'intérêt d'une telle chose ? Une première raison pourrait être que le stockage de l'évaluation d'une expression coûteuse en temps de calcul permet d'échanger de la mémoire contre du temps, en ne répétant pas le calcul mais en retrouvant son résultat dans un temporaire.

Une deuxième raison qui va attirer toute notre attention dans cette partie est le recouvrement de certains tests par une expression d'une logique à trois états, comme dans l'approche de McKinley et Kennedy évoquée précédemment (page 56). Si nous reprenons l'exemple donné à la figure 4.15, page 59, nous allons pouvoir constater que l'utilisation d'un temporaire pour stocker l'expression contenue dans if_2 va simplifier la réécriture du code.

Dans le code de la figure 4.30, nous avons décidé d'attribuer une valeur $t_1 = 1$ où t_1 est la dimension du temporaire que nous autorisons pour stocker l'expression de if_1. Nous autorisons donc un temporaire de dimension 1, c'est-à-dire un tableau de dimension 1.

Lors du partitionnement au niveau $k = 1$, nous avons enlevé les arcs représentant des anti-dépendances de chemin, puisque nous pouvons introduire un temporaire de dimension suffisante : avec $d = 1$ et $t_1 = 1$, nous avons $d - t_1 = 0$ ce qui est strictement inférieur à k. Nous introduisons donc un temporaire dans le MDG en reportant les arêtes de manière correcte. Nous avons choisi arbitrairement de mettre le sommet t_1 et le sommet s_1 dans le même groupe. Jusqu'ici, c'est exactement l'algorithme que nous avons énoncé plus haut.

Si nous considérons maintenant une logique à trois états, nous pouvons recouvrir la condition contenue dans if_1 par la condition de if_2 pour les actions s_2 et s_3. Dans le code de la figure 4.31, nous avons décidé d'attribuer une valeur $t_1 = -1$ (pas de temporaire autorisé) et $t_2 = 1$ où t_2 est la dimension du temporaire que nous autorisons pour stocker l'expression de if_2. Nous autorisons donc cette fois-ci un tableau de dimension 1 pour stocker l'expression de if_2. Lors de l'écriture d'un code avec une logique à trois états, il doit y avoir une action supplémentaire au début du groupe qui contient l'affectation du temporaire. Cette action



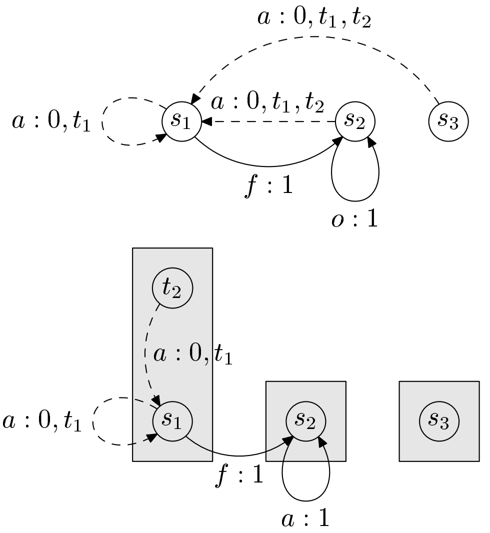

```
doall i = 2, n                  (* do1 *)
   t1(i) = (a(i) > T)           (* t1 *)
   if (t1(i)) then              (* if1 *)
      a(i) = i                  (* s1 *)
   endif
enddoall
do i = 2, n                     (* do'1 *)
   if (not t(i)) then           (* if'1 *)
      if (b(i) ≠ 0) then        (* if'2 *)
         u = a(i − 1)/b(i)      (* s2 *)
      endif
   endif
enddo
doall i = 2, n                  (* do1 *)
   if (not t1(i)) then          (* if1 *)
      if (not b(i) ≠ 0) then    (* if2 *)
         c(i) = d(i) + 1        (* s3 *)
      endif
   endif
enddoall
```

Figure 4.30: application de l'algorithme 4.1 sur l'exemple de la figure 4.15, avec une logique à deux états.

consiste à affecter le temporaire avec une valeur $\perp$ qui signifie que le temporaire est indéfini. Ainsi, le temporaire peut prendre trois valeurs : V, F et $\perp$.

Le code s'est simplifié pour les deux dernières boucles. Les instructions if qui étaient auparavant imbriquées sont maintenant devenues une seule instruction avec un seul test.

L'intégration de cette extension dans l'algorithme se fait très simplement en ajoutant sur les arcs de chemins concernés (ici les arcs allant de s_2 et s_3 vers s_1), une valeur de temporaire autorisée supplémentaire pour une condition susceptible de recouvrir le test. Lors de la détermination des arcs représentant des anti-dépendances de chemin à enlever grâce à des temporaires, toutes ces différentes valeurs de dimension vont être considérées en partant de celle représentant l'instruction la plus profonde. Dans l'exemple précédent, la valeur t_2 est d'abord examinée, puis la valeur t_1 si t_2 ne convient pas. Et ainsi de suite si il y a plus de valeurs. Lors de la génération du code, il faut également penser à rajouter l'affectation préliminaire du temporaire à $\perp$ au début du groupe qui contient l'initialisation du temporaire.




```
doall i = 2, n                (* do1 *)
   t2(i) =⊥
   if (a(i) > T) then         (* if1 *)
      a(i) = i                (* s1 *)
   else
      t2(i) = (b(i) ≠ 0)      (* t2 *)
   endif
enddoall
do i = 2, n                   (* do'1 *)
   if (t2(i) == V) then       (* if2 *)
      u = a(i − 1)/b(i)       (* s2 *)
   endif
enddo
doall i = 2, n                (* do'1 *)
   if (t2(i) == F) then       (* if2 *)
      c(i) = d(i) + 1         (* s3 *)
   endif
enddoall
```

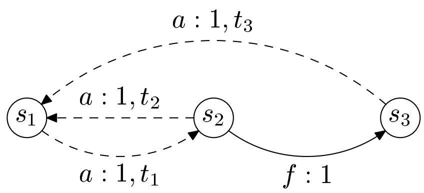
Figure 4.31: application de l'algorithme 4.1 sur l'exemple de la figure 4.15, avec une logique à trois états (avec recouvrement des tests).

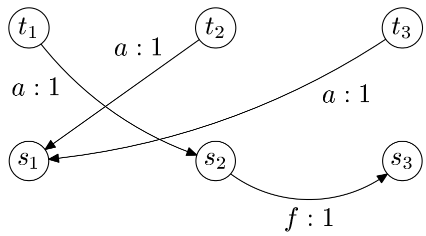
4.8 Minimisation du nombre de temporaires

4.8.1 Un exemple complet.

Étant donné un code ainsi que son graphe des dépendances mixte, rajouter tous les temporaires liés à une anti-dépendance de chemin permet de trouver le maximum de parallélisme dans le code, au regard de la représentation des dépendances par niveau employée [23]. Il se peut cependant que l'on puisse obtenir la même quantité de parallélisme avec moins de temporaires. Examinons pour illustrer notre propos le code présenté à gauche de la figure 4.32, avec son MDG. Les trois dépendances de chemin créent un circuit dans le graphe composant une seule composante fortement connexe. De plus, toutes les dépendances sont portées par la boucle, ce qui fait que tel quel il n'y pas de solution de partition qui exhibe du parallélisme. Si nous enlevons tous les arcs représentant des anti-dépendances de chemin en introduisant des temporaires, en prenant $t_1 = t_2 = t_3 = 1$, nous obtenons un code avec le maximum de parallélisme. Nous avons présenté le MDG correspondant à l'introduction de trois temporaires ainsi qu'un code parallèle issu d'une partition légale de ce MDG à droite de la figure 4.32.

Nous allons maintenant voir s'il est possible d'obtenir autant de parallélisme avec moins de temporaires. Essayons de voir ce que nous pouvons obtenir comme code avec deux temporaires. La première possibilité est d'autoriser un temporaire pour t_1 et t_2 mais d'empêcher l'introduction d'un temporaire pour t_3. Nous obtenons le premier MDG de la figure 4.33. Une partition possible est de mettre t_1 et t_2 dans une même boucle et de mettre chaque instruction s_1, s_2 et s_3 dans une boucle différente. Un code correspondant à cette partition est donné dans la même figure. Nous avons donc avec deux temporaires la même quantité de parallélisme qu'avec trois temporaires. Il y a deux autres possibilités de contraindre l'introduction de temporaires qui sont également présentées toutes deux à la figure 4.33. Ces deux MDG permettent encore une fois d'exhiber la même quantité de parallélisme. Ainsi, avec seulement






```
do i = 2,n − 1                (* do1 *)
   if (b(i + 1) > 0) then     (* if1 *)
      a(i) = 1                (* s1 *)
   endif
   if (a(i + 1) > 0) then     (* if2 *)
      b(i) = 2                (* s2 *)
   endif
   if (a(i + 1) ≥ 0) then     (* if3 *)
      c(i) = b(i − 1) + 3     (* s3 *)
   endif
enddo
```

```
doall i = 2,n − 1                   (* do1 *)
   t1(i) = (b(i + 1) > 0)           (* t1 *)
   t2(i) = (a(i + 1) > 0)           (* t2 *)
   t3(i) = (a(i + 1) ≥ 0)           (* t3 *)
enddoall
doall i = 2,n − 1                   (* do1 *)
   if (t1(i)) a(i) = 1              (* if1, s1 *)
   if (t2(i)) b(i) = 2              (* if2, s2 *)
enddoall
doall i = 2,n − 1                   (* do1 *)
   if (t3(i)) c(i) = b(i − 1) + 3   (* if3, s3 *)
enddoall
```

Figure 4.32: exemple de code avec trois dépendances de chemin qui créent des circuits dans le graphe. Il n'y a pas de solution parallèle sans temporaire. La solution avec trois temporaires est présentée à droite de la figure, avec le MDG associé (pour plus de concision, nous avons omis les mots clés then et endif).

deux temporaires, quelle que soit la façon dont nous contraignons leur introduction, il nous est possible d'obtenir la même quantité de parallélisme.

Considérons maintenant les trois possibilités d'introduction d'un seul temporaire qui sont présentées dans la figure 4.34. Le premier MDG en haut à gauche correspond à un temporaire pour t_2. Le meilleur code que nous puissions obtenir en terme de parallélisme pour ce MDG est celui présenté sous celui-ci, où la seule chose que nous pouvons calculer en parallèle est la condition de l'instruction if$_2$. Tout le reste est séquentiel. Le deuxième MDG en haut à droite correspond à un temporaire pour s_3. Ce MDG nous permet d'exhiber un petit peu plus de parallélisme puisque cette fois-ci nous pouvons calculer s_3 en parallèle. La troisième solution est la solution optimale en terme de nombre de temporaires tout en conservant la même quantité de parallélisme qu'avec le maximum de temporaires. Il suffit donc d'introduire un temporaire pour t_1 pour pouvoir tout exécuter en parallèle. Cet exemple nous permet de mettre en avant un point crucial de la minimisation du nombre de temporaires : enlever l'arc de s_1 à s_2 dans le MDG permet de casser tous ses circuits, rendant ainsi la liberté de partitionnement totale.

4.8.2 NP-complétude

Le problème que nous devons résoudre est donc le suivant : étant donné un code représenté par un MDG et connaissant la quantité maximale de parallélisme que nous pouvons obtenir pour ce code (avec le maximum de temporaires), combien de temporaires devons nous insérer au minimum pour obtenir la quantité maximale de parallélisme ? En terme de MDG, nous pouvons reformuler le problème en :

étant donné un MDG avec des arcs correspondant à des anti-dépendances de che-

min, combien de ces arcs faut-il enlever au minimum pour supprimer un maximum de circuits du graphe ?

Soit $G = (V, E)$ un MDG correspondant à une boucle. Supprimons tous les arcs de chemin issus d'anti-dépendances en introduisant les temporaires adéquats dans le graphe : nous obtenons un graphe G'. Considérons le graphe G'' des composantes fortement connexes de G', où il y a un arc d'un sommet c_1 vers un sommet c_2 de G'' s'il y a un arc d'un sommet s_1 vers un sommet s_2 de G' et que $s_1 \in c_1$ et $s_2 \in c_2$. Le code correspondant à G'' donne la quantité maximale de parallélisme (c'est l'application de l'algorithme de parallélisation présenté précédemment). Rajoutons à ce graphe G'' les arcs issus d'anti-dépendance que nous avions supprimés de G. Tout ce que nous avons fait jusqu'à présent se fait en temps polynômial. Le fait de rajouter des arcs entre les sommets de G'' peut avoir comme effet de créer des circuits dans celui-ci. Le problème que nous avons présenté précédemment va donc être de supprimer le minimum d'arcs pour que tous ces circuits soient supprimés. Ce problème a été démontré comme étant NP-complet et est présenté sous le nom de FEEDBACK-ARC-SET dans le livre de Garey et Johnson [29]. En toute rigueur, pour démontrer ce résultat, étant donnée une instance de FEEDBACK-ARC-SET, il faut montrer que l'on peut construire un code pour lequel minimiser le nombre de temporaires jusqu'à ce qu'il n'y ait plus de circuit permet de montrer qu'avec les mêmes arcs supprimés, on résoud l'instance de FEEDBACK-ARC-SET.

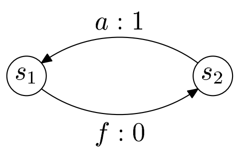
Nous pouvons espérer que le nombre de temporaires possible soit relativement petit, permettant ainsi une recherche exhaustive de la solution optimale. Ainsi, dans l'exemple précédent choisi pour son exemplarité, il n'y a que 8 possibilités à examiner, sachant que la recherche peut s'arrêter dès que la solution optimale a été trouvée (la recherche d'une solution peut commencer sans temporaire, puis avec un seul temporaire, puis avec deux et ainsi de suite).

4.9 Quelques remarques finales

Nous avons présenté un nouveau type de graphe permettant de prendre en compte les dépendances impliquant une structure de contrôle. Nous avons expliqué que ce graphe permettait d'obtenir par l'ensemble des partitions légales, l'ensemble des codes légaux. Nous avons également donné une méthode pour introduire des temporaires permettant de supprimer des arcs issus d'anti-dépendances de chemin et donné un algorithme de parallélisation utilisant cette méthode. Nous avons également expliqué que le problème de minimiser le nombre de temporaires était NP-complet, mais que nous pensons qu'une recherche exhaustive était tout à fait faisable en un temps raisonnable. L'algorithme de parallélisation présenté est implanté dans Nestor, que nous présentons au chapitre suivant.

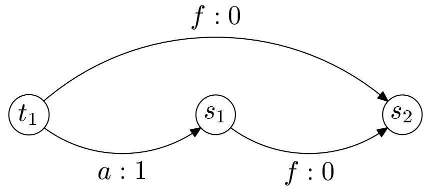
Pour terminer ce chapitre, nous pouvons remarquer que la méthode de Padua et Wolfe [46] permettant de « casser » des anti-dépendances en introduisant un temporaire repose sur le même principe que ce que nous venons de présenter dans ce chapitre. Néanmoins, ils présentent leur méthode avec une légère différence. Considérons le code suivant :

```
do i = 2, n
   a(i) = b(i) + c(i)       (* s1 *)
   d(i) = a(i) + a(i + 1)   (* s2 *)
enddo
```

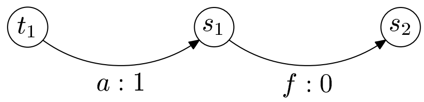


représenté avec son MDG. Il y a un circuit de dépendance composé d'une dépendance de flot et d'une anti-dépendance impliquant l'expression $a(i+1)$. Un tableau temporaire peut être introduit de la même façon que pour les instructions de contrôle :

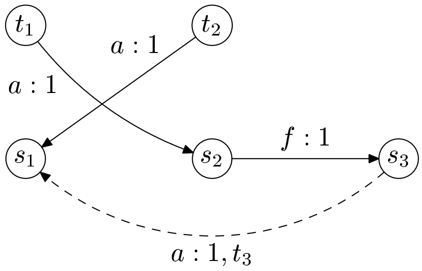
do $i = 2, n$
  $t1(i) = a(i+1)$ (* t_1 *)
  $a(i) = b(i) + c(i)$ (* s_1 *)
  $d(i) = a(i) + t1(i)$ (* s_2 *)
enddo



Le MDG que nous avons dessiné ci-dessus est celui issu de l'article de Padua et Wolfe. Avec notre approche, il serait sensiblement différent. En effet, avec les mêmes arguments que pour les structures de contrôle, nous pouvons dire que l'arc entre t_1 et s_2 n'a pas de raison d'être : lors de la génération de code, nous utilisons ou non le temporaire selon que nous sommes avant ou après le puits de l'anti-dépendance. Nous pouvons remarquer qu'une expression en partie droite d'une affectation peut uniquement être source d'une anti-dépendance ou puits d'une dépendance de flot : l'expression ne modifie pas la mémoire. Le graphe que nous obtiendrions avec notre approche serait le suivant :

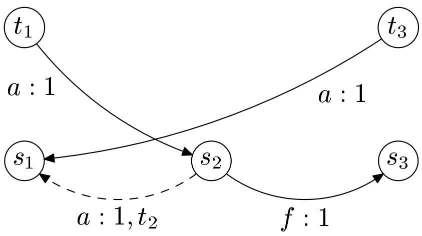


Par rapport à l'algorithme de parallélisation que nous avons présenté précédemment, nous pouvons tout à fait envisager de rajouter un paramètre de dimension de temporaire autorisé sur tous les arcs représentant une anti-dépendance et pas uniquement sur les arcs représentant une anti-dépendance de chemin. Ainsi, nous pensons que nous pouvons appliquer la même méthode que pour les structures de contrôle et obtenir un algorithme qui permet de supprimer toutes les anti-dépendances lors de la parallélisation d'un code. À notre connaissance, c'est la seule technique qui permet de prendre en compte le fait qu'on peut *utiliser* ou *recalculer* une expression. L'approche de Padua et Wolfe utilise toujours le temporaire introduit.



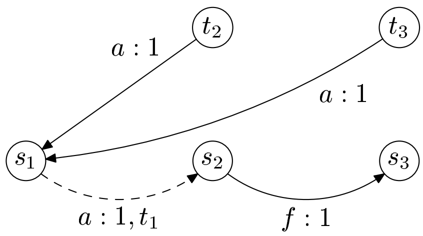
$$t_1 = t_2 = 1,\ t_3 = -1$$

doall $i = 2, n-1$	(* do_1 *)
$t1(i) = (b(i+1) > 0)$	(* t_1 *)
$t2(i) = (a(i+1) > 0)$	(* t_2 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(t2(i))$ $b(i) = 2$	(* if_2, s_2 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(a(i+1) \geq 0)$ $c(i) = b(i-1) + 3$	(* if_3, s_3 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(t1(i))$ $a(i) = 1$	(* if_1, s_1 *)
enddoall	



$$t_1 = t_3 = 1,\ t_2 = -1$$

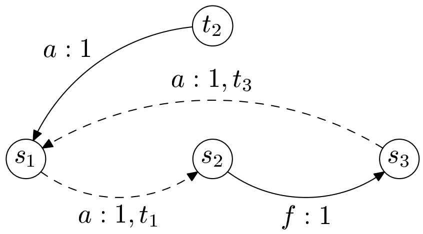
doall $i = 2, n-1$	(* do_1 *)
$t1(i) = (b(i+1) > 0)$	(* t_1 *)
$t3(i) = (a(i+1) \geq 0)$	(* t_3 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(a(i+1) > 0)$ $b(i) = 2$	(* if_2, s_2 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(t1(i))$ $a(i) = 1$	(* if_1, s_1 *)
if $(t3(i))$ $c(i) = b(i-1) + 3$	(* if_3, s_3 *)
enddoall	



$$t_2 = t_3 = 1,\ t_1 = -1$$

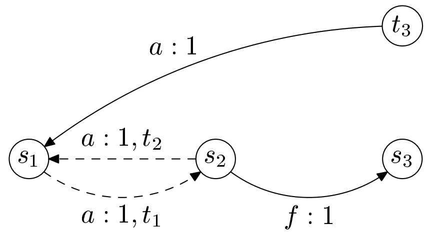
doall $i = 2, n-1$	(* do_1 *)
$t2(i) = (a(i+1) > 0)$	(* t_2 *)
$t3(i) = (a(i+1) \geq 0)$	(* t_3 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(b(i+1) > 0)$ $a(i) = 1$	(* if_1, s_1 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(t2(i))$ $b(i) = 2$	(* if_2, s_2 *)
enddoall	
doall $i = 2, n-1$	(* do_1 *)
if $(t3(i))$ $c(i) = b(i-1) + 3$	(* if_3, s_3 *)
enddoall	

Figure 4.33: trois MDGs et trois codes parallèles possibles avec deux temporaires, obtenus depuis le MDG de la figure 4.32.



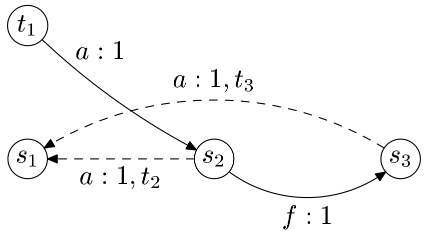
$$t_1 = t_3 = -1,\ t_2 = 1$$

doall $i = 2, n-1$ (* do$_1$ *)
    $t2(i) = (a(i+1) > 0)$ (* t$_2$ *)
enddoall
do $i = 2, n-1$ (* do$_1$ *)
    if $(b(i+1) > 0)$ $a(i) = 1$ (* if$_1$, s_1 *)
    if $(t2(i))$ $b(i) = 2$ (* if$_2$, s_2 *)
    if $(a(i+1) \geq 0)$ (* if$_3$ *)
        $c(i) = b(i-1) + 3$ (* s_3 *)
enddo



$$t_1 = t_2 = -1,\ t_3 = 1$$

doall $i = 2, n-1$ (* do$_1$ *)
    $t3(i) = (a(i+1) \geq 0)$ (* t$_3$ *)
enddoall
do $i = 2, n-1$ (* do$_1$ *)
    if $(b(i+1) > 0)$ $a(i) = 1$ (* if$_1$, s_1 *)
    if $(a(i+1) > 0)$ $b(i) = 2$ (* if$_2$, s_2 *)
enddo
doall $i = 2, n-1$ (* do$_1$ *)
    if $(t3(i))$ (* if$_3$ *)
        $c(i) = b(i-1) + 3$ (* s_3 *)
enddoall



$$t_2 = t_3 = -1,\ t_1 = 1$$

doall $i = 2, n-1$ (* do$_1$ *)
    $t1(i) = (b(i+1) > 0)$ (* t$_1$ *)
enddoall
doall $i = 2, n-1$ (* do$_1$ *)
    if $(a(i+1) > 0)$ $b(i) = 2$ (* if$_2$, s_2 *)
enddoall
doall $i = 2, n-1$ (* do$_1$ *)
    if $(a(i+1) \geq 0)$ $c(i) = b(i-1) + 3$ (* if$_3$, s_3 *)
enddoall
doall $i = 2, n-1$ (* do$_1$ *)
    if $(t1(i))$ $a(i) = 1$ (* if$_1$, s_1 *)
enddoall

Figure 4.34: trois MDGs et trois codes parallèles possibles avec un temporaire, obtenus depuis le MDG de la figure 4.32. Le dernier code exhibe le maximum de parallélisme avec le nombre minimal de temporaires.

Chapitre 5

Nestor

Dans les deux chapitres précédents, nous avons présenté des méthodes de transformation de code. Dans le domaine de la parallélisation automatique, de nombreuses autres méthodes de transformation ont été développées, certaines assez simples à implanter, d'autres très difficiles, à tel point qu'elles n'ont jamais été implantées dans un compilateur industriel. Ces algorithmes utilisent des manipulations de graphe, de l'algèbre linéaire, de l'énumération de points entiers de polyèdres, etc. (voir par exemple les algorithmes de détection de parallélisme présentés dans [27, 40, 73, 38, 24, 20], cette liste n'étant pas exhaustive). L'implantation de ces algorithmes est souvent un problème de recherche par lui-même et doit donc être facilitée par des outils fournissant les briques de base d'un tel développement et une représentation simple et facile à manipuler du code devant être modifié. De plus, la plupart de ces algorithmes doivent être validés sur de vrais codes : la représentation du code à modifier doit donc supporter un langage complet, comme par exemple Fortran 77 ou Fortran 90. Dans la mesure du possible, cette représentation doit cacher au maximum la représentation bas-niveau classique, telle que l'arbre abstrait bas-niveau fournit par un analyseur syntaxique, difficile à manipuler et à modifier. Dans le cas de transformations mettant en évidence du parallélisme, il est également utile de permettre l'expression de ce parallélisme à l'aide d'un paradigme de programmation parallèle par insertion de directives. Un tel paradigme permet de tester et d'évaluer immédiatement le code produit sur une machine parallèle après compilation par un compilateur approprié. Un programmeur peut également introduire des appels à une bibliothèque de passage de messages (PVM [50], MPI [41]) si les paradigmes proposés pour exprimer le parallélisme ne lui suffisent pas.

Les bibliothèques de Nestor [57, 60] fournissent un tel environnement de développement, permettant de construire des passes de transformation ou d'analyse de Fortran et de ses différents dialectes courants (Fortran 77, 90 et 95, OpenMP [66, 30], HPF 2.0 [31], directives Cray ou Sun [77, 70]). Nous avons choisi Fortran et ses dialectes car ce langage est riche de directives diverses et variées permettant d'exprimer du parallélisme dans les boucles, des tâches (portions concurrentes de code), des réductions, des distributions et du placement de données, et il est très utilisé pour les applications scientifiques qui sont les principales bénéficiaires de la parallélisation automatique (bien que Nestor ne soit pas limité au seul domaine de la parallélisation automatique). De plus, le langage Fortran est un langage relativement « propre » par rapport à un langage comme le langage C où il est très facile d'écrire des programmes difficiles à analyser.

Le premier intérêt à rester au niveau du code source est que le code est toujours lisible par le programmeur : celui-ci peut contrôler les optimisations faites par le compilateur directement, sans avoir à « éplucher » le code assembleur généré, ou le code Fortran 77 compliqué généré par un compilateur HPF comme Adaptor [13]. Notre approche des directives est duale : le programmeur peut insérer des directives dans le code séquentiel indiquant à l'outil de transformation ou d'analyse développé avec Nestor quelles sont les parties du code à examiner ou étant déjà parallèles, et un outil de transformation développé avec Nestor insère des directives indiquant quelles sont les parties du code parallèles, les réductions, etc... La figure 5.1 présente le schéma de compilation que nous considérons.

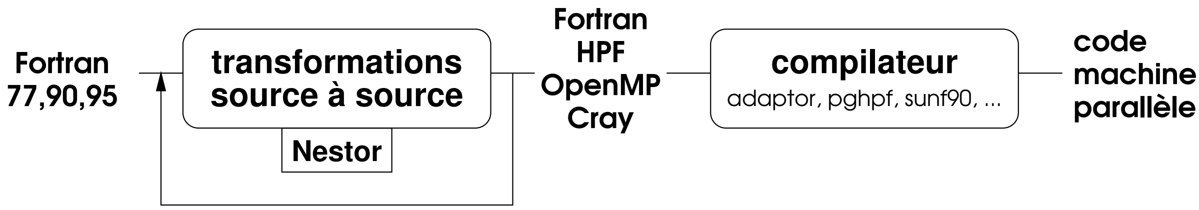


Figure 5.1: schéma de transformation source à source de programmes Fortran.

Dans la première partie de ce chapitre (§5.1, p. 86), nous allons survoler les différents aspects de Nestor, exposer les différences par rapport aux outils du même type existants et présenter l'organisation générale de Nestor. Dans la deuxième partie (§5.2, p. 88), nous parlerons de la documentation offerte par Nestor qui est une de ses caractéristiques principales. Dans la troisième partie (§5.3, p. 90), nous présenterons le noyau de Nestor qui est la bibliothèque contenant les classes de base, en expliquant les choix d'implantation, la structure d'une classe type et les différentes possibilités offertes par celle-ci. Ensuite, nous expliquerons pourquoi nous avons créé un « générateur de Nestor » que nous appelons métaNestor (§5.4, p. 114) et qui permet de générer les classes C++ qui représentent la structure interne d'un langage depuis des fichiers de description de la grammaire du langage. Nous parlerons ensuite de la bibliothèque d'analyse syntaxique et sémantique de programmes en Fortran (§5.5, p. 115), fondée sur la partie frontale du compilateur Adaptor [13], ainsi que la bibliothèque de manipulation de graphes de dépendances (§5.6, p. 117).

À ce stade, nous aurons fini la description de la structure de base de notre système. Nous pourrons alors décrire GTKNestor qui est une interface graphique permettant de parcourir la structure interne et d'appliquer des transformations de manière interactive sur n'importe quel élément (§5.7, p. 120). Dans la partie suivante (§5.8, p. 123), nous décrirons la bibliothèque permettant d'écrire des transformations sous la forme de code dynamique lié à l'exécution dans des outils comme GTKNestor. Nous finirons par des exemples d'utilisation de notre système (§5.9, p. 125) avant de parler des perspectives futures de celui-ci (§5.10, p. 138).

Après avoir lu le paragraphe suivant, le lecteur pressé pourra directement aller voir la dernière partie de ce chapitre (§5.9, p. 125) pour trouver des exemples d'utilisation de Nestor.

5.1 Aperçu de Nestor et outils existants

De nombreux outils intégrant des algorithmes de transformation de code ont été développés, par exemple Bouclettes [8], Petit [49] ou LooPo [65]. Leurs objectifs étaient de prouver que ces algorithmes pouvaient être implantés dans un vrai compilateur. Par exemple, Petit démontre la puissance du test Omega et de la librairie correspondante. Néanmoins, ils n'ont pas été conçus pour traiter de vrais codes : ils prennent en entrée un langage « jouet », habituellement un sous ensemble de Fortran ou de C.

Cependant, les méthodes théoriques doivent être également validées en testant leur comportement sur de vraies applications, par exemple des codes scientifiques. D'autres outils plus ambitieux ont donc été développés dans le passé avec cet objectif en tête :

- PIPS [48], qui est un outil complet de parallélisation de code traitant le Fortran 77, développé par l'équipe de François Irigoin à l'École des mines de Paris ;
- PAF, qui est un Paralléliseur Automatique de Fortran 77, développé au PRiSM par l'équipe de Paul Feautrier ;
- Polaris [67], qui est également un paralléliseur automatique de Fortran 77, développé à l'université d'Urbana-Champaign par l'équipe de David Padua ;
- SUIF [62], qui définit un format de représentation des programmes C et Fortran 77 et un ensemble de transformations, et qui a été développé à l'université Stanford par l'équipe de Monica Lam ;
- Sage++ [68], qui comme SUIF définit également un format de représentation des programmes C++ et Fortran 77 et qui a été développé à l'université de l'Indiana et à l'université de Rennes.
- sans oublier tous les outils commerciaux de parallélisation de code, Forge, Foresys, KAP Guide, etc... Ces outils ne rentrent pas dans le cadre de cette étude puisque par définition ce ne sont pas des outils « ouverts » et gratuits pouvant être étendus.

Ces outils ont été développés depuis plus de dix ans pour certains d'entre eux : ce sont maintenant de très gros systèmes (difficiles à maintenir et à étendre) et leur représentation interne est souvent à un niveau trop bas pour permettre une prise en main facile par les nouveaux utilisateurs. Par exemple, SUIF est en train d'évoluer vers sa version 2, ce qui implique un changement radical de sa structure interne. L'équipe développant Polaris pense également à changer la représentation interne qu'ils utilisent.

Tous ces outils sont souvent peu appropriés pour effectuer des transformations de haut niveau sur le code source. Par exemple, SUIF ne traite pas directement le Fortran mais transforme celui-ci en C avant de représenter le programme C en représentation interne. Celle-ci ne représente donc pas vraiment le programme Fortran et il est difficile de retrouver le programme Fortran original. D'autre part, cette représentation est d'un niveau beaucoup plus bas que la représentation de Nestor. Sage++ est la bibliothèque la plus proche de la notre en terme de fonctionnalités, mais les conceptions sont très différentes sur le fond. Sage++ se fonde sur une structure interne C de bas niveau et les classes C++ fournissent des méthodes pour accèder à cette structure. À l'inverse, il n'y a pas de structure interne sous-jacente dans Nestor : ses différentes classes *sont* la structure interne. Dans la pratique, il s'avère que l'on ne peut pas prévoir tout ce que l'utilisateur veut faire comme tranformations. Il faut donc lui donner la structure interne et non pas se contenter de lui donner des moyens d'accéder à celle-ci. Ainsi, l'ajout d'un élément dans notre structure interne se fait très simplement par l'ajout d'une classe, sans avoir à se soucier de la cohérence d'une structure interne sous-jacente où d'un mécanisme global d'écriture de la structure. En bref, Nestor est beaucoup plus approprié pour implanter des transformations au niveau du code source que tous ces outils. De plus, aucun de ces outils ne supporte le Fortran 90 et toutes les directives des différents dialectes parallèles.

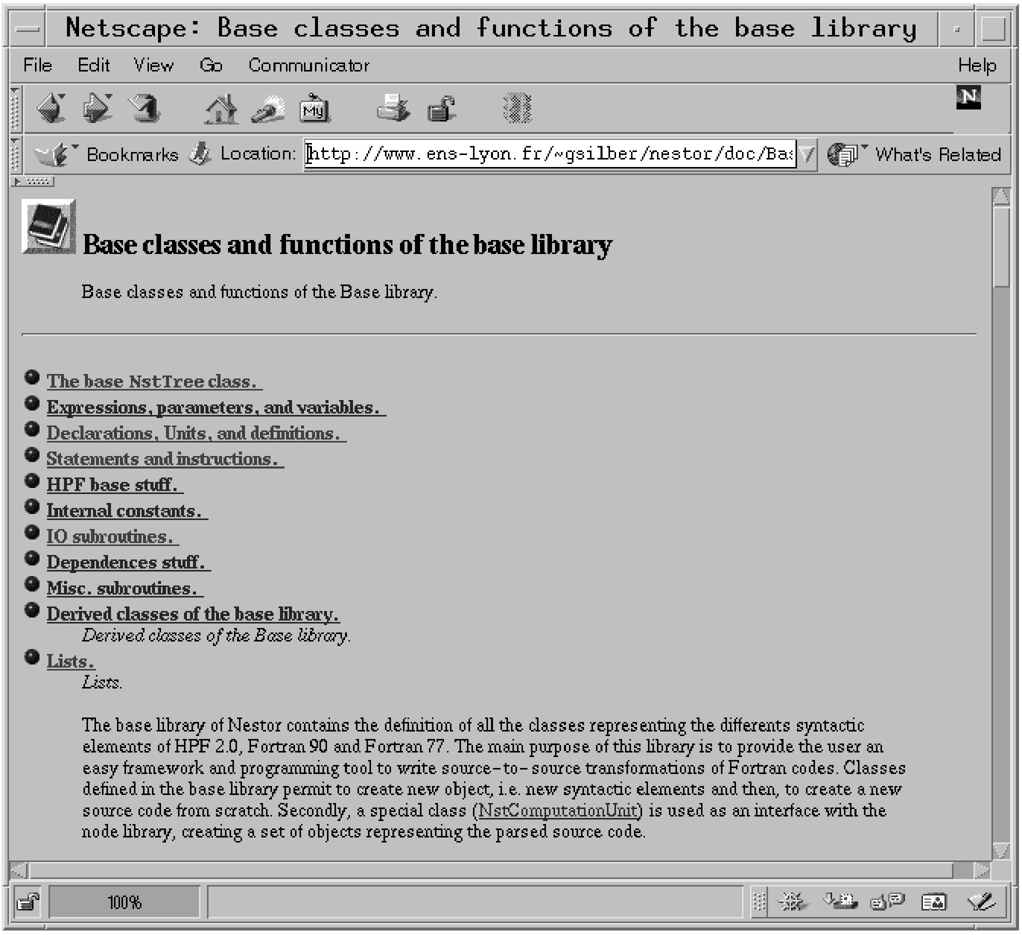
En plus de ces différences au niveau des langages en entrée et en sortie, Nestor a été pensé dès le départ comme n'étant pas un outil complet de parallélisation, mais comme un outil permettant d'implanter facilement des transformations. Nestor ne fait rien ! Il fournit cependant des moyens puissants pour faire de nombreuses choses facilement. Cet objectif qui peut paraître moins ambitieux que tous les outils précédents lui confère de nombreux avantages, puisqu'il a été conçu en vue de faciliter la programmation de transformations. Certaines des caractéristiques que nous présentons ci-dessous sont présentes dans des outils existants, mais aucun outil n'a toutes ces caractéristiques.

- Nestor se concentre sur les transformations au niveau du code source. Sa structure interne est donc le plus proche possible du code source, avec notamment une gestion des commentaires et des directives du code original qui permet de regénérer le code à l'identique.
- Nestor a une structure interne qui offre de nombreuses possibilités de parcours génériques dans tous les sens de la structure (en profondeur, en largeur, remontée, etc...).
- Nestor a par nature une gestion de la mémoire évitant les fuites. Il peut donc traiter de très gros programmes comportant de nombreuses lignes de code.
- Nestor supporte complètement les langages Fortran 77, 90, et 95, avec les directives HPF 2.0, OpenMP, Cray, Sun, etc...
- Nestor est petit (si on ne prend pas en compte les outils externes optionnels). Ses bibliothèques fournissent uniquement les briques de base pour construire des systèmes de transformation de code au niveau du code source. Sa taille permet une installation facile sur n'importe quel système UNIX. Il est développé et maintenu par une seule personne.
- Nestor est gratuit et librement distribuable. N'importe qui est libre de récupérer notre outil et de le modifier à sa guise [57]. Il est distribué sous licence GNU.
- Nestor est orienté objet. Ce paradigme de programmation offre de nombreux avantages pour la programmation de transformations et d'analyses : l'héritage, la surcharge des opérateurs, la facilité de gestion des entrées/sorties, l'encapsulation de méthodes spécifiques à un élément syntaxique, les méthodes virtuelles, etc... La bibliothèque standard C++ offre de nombreux types structurés utiles pour le programmeur. Le C++ offre également l'avantage d'un interfaçage facile avec de nombreuses bibliothèques existantes (Omega, la polylib, ...).
- Nestor est complètement documenté. Cette caractéristique pourtant indispensable pour le programmeur se retrouve rarement dans les outils existants. La documentation de notre outil évolue en même temps que le code et est consultable en HTML et en Postscript.
- Nestor a une interface graphique qui permet d'effectuer des transformations de manière interactive sur la structure interne.
- Nestor comporte une bibliothèque qui permet de créer des transformations sous la forme de code C++ utilisé dynamiquement.
- Nestor est facilement extensible. Il est très facile d'ajouter une classe à la structure interne. Nous avons également créé un outil qui permet de créer un ensemble de classes représentant un langage depuis une définition concise de la grammaire de ce langage.
- Nestor offre une analyse des dépendances et une bibliothèque de manipulation de graphes.

5.2 Documentation en ligne

Nestor a été conçu dès le départ comme un système devant être facile à utiliser et à programmer. Pour cela, il est nécessaire que le système soit documenté. Nous avons choisi de faire évoluer la documentation en même temps que le code source, puisque celle-ci se trouve dans le code même de Nestor. Un utilitaire du domaine public, doc++ [80], nous permet ensuite d'extraire cette documentation du code et de la mettre en forme en HTML agrémenté de Java ou en Postscript. Ainsi, la documentation la plus récente de Nestor est toujours disponible par l'intermédiaire d'un navigateur web à l'endroit même où Nestor est développé [57]. Notre système intègre également dans sa distribution publique la documentation HTML complète de celui-ci.

La navigation dans la documentation se fait à travers des liens hypertexte qui permettent de gagner énormément de temps lors du développement. Il est un effet très facile de trouver la définition de telle ou telle classe et de savoir de quelle classe celle-ci hérite ou quels sont ses héritières, d'avoir une documentation complète sur l'utilisation de tel ou tel aspect de la classe, etc... Java est utilisé dans la documentation pour représenter le graphe d'héritage de chaque classe, sous la forme de boutons reliés. Un dessin résumant de longues lignes de texte, le dessin suivant est un extrait de la documentation générale des classes de base dans un navigateur web (tous les morceaux de texte soulignés permettent d'accéder à la documentation correspondante) :



Le dessin suivant est un extrait de la documentation de la classe NstStatementIf :

5.3 La bibliothèque des classes de base

Le cœur de Nestor est la bibliothèque `nstbase` qui contient toutes les classes décrivant la structure interne. Chaque classe de la structure interne décrit un élément syntaxique du langage, sous une forme très simple à manipuler et à modifier grâce à toutes les méthodes fournies dans chaque classe.

Pour illustrer notre propos, nous allons utiliser un exemple (5.1) qui va nous servir de support tout au long de ce chapitre. Cet exemple contient trois déclarations de variable (`i`, `j` et `k`) et trois affectations de ces variables. Ce code va être représenté par une structure interne à Nestor. Les différents éléments syntaxiques vont être représentés par différents objets de différentes classes.

```
program dummy
  integer i
  integer j
  real k
  i = 3
  j = i * 3
  k = 3.2
end
```

Exemple 5.1: un exemple de programme Fortran très simple.

Dans ce qui suit, nous allons examiner tous les concepts fondamentaux liés à la définition d'une classe dans cette bibliothèque, en commençant par l'héritage.

5.3.1 Héritage et type des objets

En prenant l'exemple 5.1, partons des éléments les plus simples qui sont les deux constantes littérales `3` et `3.2`. La première de ces constantes va être représentée par un objet de la classe `NstInteger` qui contient un attribut de type entier. Voici un extrait de cette classe :

```
class NstInteger : public NstConstant {
  ...
  int value;
  ...
};
```

où dans notre cas l'attribut `value` contient le nombre 3. La deuxième constante est représentée par un objet de la classe `NstReal` qui contient un attribut de type `string` (les réels ne sont pas stockés dans un attribut `float` pour des raisons que nous n'exposerons pas ici). Voici un extrait de cette classe :

```
class NstReal : public NstConstant {
  ...
  string value;
  ...
};
```

où dans notre cas l'attribut `value` contient la chaîne de caractères `"3.2"`.

Le premier concept important est que les deux classes *héritent* de la même classe de base `NstConstant`. En fait, les deux classes représentent bien toutes deux des constantes, mais l'une est un entier alors que l'autre est un réel. Nous utilisons donc le principe d'*héritage* offert par la programmation orientée objet et par C++ pour modéliser la structure d'un programme. D'autres classes sont *dérivées* de la classe `NstConstant`. Le *graphe d'héritage* de cette classe est présenté à la figure 5.2.

Ainsi, les classes `NstString`, `NstBoolean` et `NstComplex` héritent également de la classe `NstConstant` et représentent respectivement des constantes de type chaîne de caractères, booléen et nombre complexe. La classe `NstConstant` hérite de la classe `NstTree`. Cette classe `NstTree` est la classe la plus importante de Nestor. Toutes les classes de la bibliothèque de

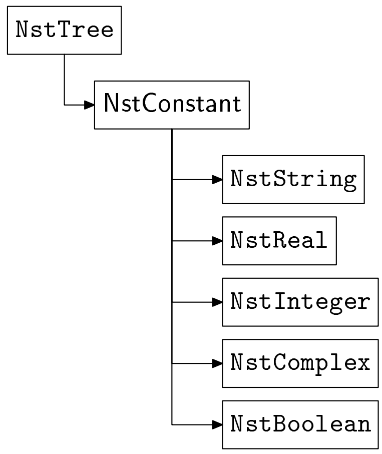


Figure 5.2: graphe d'héritage de la classe `NstConstant`.

base héritent de cette classe qui contient l'implantation de tous les mécanismes importants permettant de se déplacer dans la structure, de connaître l'identité d'un objet, etc...

Ainsi, chaque objet d'une classe de la bibliothèque `nstbase` contient au moins un sous-objet de type `NstTree`. Par exemple, un objet de la classe `NstInteger` contient un sous-objet de la classe `NstConstant` qui contient lui même un sous-objet de la classe `NstTree`. La figure 5.3 donne une vision abstraite d'un objet de la classe `NstInteger` avec les différents sous-objets de celui-ci.

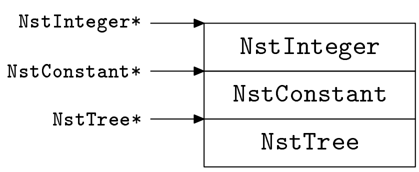


Figure 5.3: un objet de la classe `NstInteger` avec ses sous-objets.

Étant donné un pointeur sur un objet de la classe `NstConstant`, il est possible d'accéder à chacun de ses sous-objets avec un transtypage (cast) explicite. La fonction suivante récupère le sous-objet `NstConstant` et le sous-objet `NstTree` d'un objet `NstInteger` :

```
void exemple_de_cast(NstInteger* entier)
{
  NstConstant* cst_entier;
  NstTree* tree_entier;

  cst_entier  = (NstConstant*) entier;
  tree_entier = (NstTree*) entier;
}
```

S'il est possible de récupérer le sous-objet d'un objet, il est également possible, étant donné un objet, de récupérer l'objet dont celui-ci est sous-objet. Par exemple, si nous considérons une liste de constantes, comme nous ne pouvons pas faire de liste contenant des objets de différents types, nous allons plutôt faire une liste contenant des objets de la classe

`NstConstant`, sachant que chacun de ces objets sera le sous-objet d'un objet dérivé de la classe `NstConstant`. Voici un extrait du prototype de la classe `NstConstantList` que nous verrons plus en détail lorsque nous parlerons des listes :

```
class NstConstantList : public NstList<NstConstant> {
  ...
};
```

Ainsi, étant donnée une telle liste, nous pouvons retrouver l'objet dont chacun des éléments de la liste est sous-objet. Il y a pour cela plusieurs possibilités. La première est d'utiliser la méthode `type()` qui est définie dans la classe `NstTree` et qui renvoie un entier unique identifiant la classe de l'objet. Ainsi, étant donné un objet de la classe `NstConstant`, il est possible de trouver le bon objet avec la méthode suivante :

```
NstInteger* isNstInteger (NstConstant* cst)
{
  if (cst->type() == NST_CONSTANT_INT)
    return (NstInteger*)cst;
  else
    return NULL;
}
```

où la fonction `isNstInteger()` renverra `NULL` si l'objet `cst` n'est pas un sous-objet d'un objet de la classe `NstInteger` et un pointeur sur l'objet `NstInteger` sinon (voir figure 5.3).

Une autre méthode est d'utiliser le patron de fonction `dynamic_cast<T>()` offert par la bibliothèque standard du C++ [63]. Cette fonction s'utilise de la façon suivante :

```
NstInteger* isNstInteger (NstConstant* cst)
{
  return dynamic_cast<NstInteger*>(cst);
}
```

et est équivalente au corps de la fonction `isNstInteger()` précédente. Dans la suite, nous utiliserons uniquement cette méthode pour effectuer des transtypages. Le type explicite codé dans Nestor existe pour des raisons historiques. En effet, la première version de Nestor a été développée avec un compilateur qui ne supportait pas le transtypage dynamique avec `dynamic_cast<T>()`. Afin de rester compatible avec les anciens codes, nous avons donc conservé la méthode `type()`.

Ainsi, ce schéma d'héritage, avec toutes les classes de la structure interne qui héritent de la classe `NstTree` et où chaque classe est un raffinement d'une classe moins évoluée, offre plusieurs visions de la structure interne. Au plus bas niveau, la structure interne est une collection d'objets de la classe `NstTree`, ce qui permet notamment de faire des parcours génériques sur la structure (§5.3.9, p. 109). Au niveau le plus haut de l'abstraction, chaque objet de la structure interne est une instance de la classe représentant exactement son élément syntaxique.

5.3.2 Relations entre les classes

Nous avons vu au paragraphe précédent comment l'héritage permettait de représenter les différents niveaux d'abstractions de la structure interne. Nous allons voir dans ce paragraphe comment les classes s'organisent pour former une structure interne. Reprenons pour cela l'exemple 5.1. Une structure interne complète est contenue dans un objet de la classe `NstComputationUnit`, celle-ci représentant un fichier de programme contenant des fonctions, de procédures, des modules, etc... Voici un extrait de la classe `NstComputationUnit` :

```
class NstComputationUnit : public NstTree {
  NstUnitList units;
  ...
};
```

où `units` est la liste de toutes les unités du fichier (voir les listes §5.3.4, p. 100). Une unité peut être un programme, une fonction, une procédure, etc... Avec l'exemple 5.1, la liste contient un seul élément qui est un objet de la classe `NstUnitProgram` (`program dummy`) dont voici un extrait :

```
class NstUnitProgram : public NstUnit {
  NstDeclarationList declarations;
  NstStatementList statements;
  ...
};
```

Avec notre exemple, la liste `declarations` contient trois déclarations et la liste `statements` contient trois instructions. Examinons cette dernière liste. Le premier élément de cette liste correspond à l'affectation `i=3`. Cet élément est un objet de la classe `NstStatementBasic` qui est la classe représentant une action. Voici un extrait de cette classe :

```
class NstStatementBasic : public NstStatement {
  NstInstruction* instruction() const;
  ...
};
```

où la méthode `instruction()` renvoie un pointeur sur l'objet de la classe `NstInstruction` contenu dans l'objet de la classe `NstStatementBasic`. Nous n'expliquerons pas la signification du mot clé `const` dans ce document (se référer pour cela à l'ouvrage de référence sur le C++ [63]). Dans notre cas, l'objet de la classe `NstInstruction` est un sous-objet de la classe `NstInstructionAssign` qui représente une affectation et dont voici un extrait :

```
class NstInstructionAssign : public NstInstruction {
  NstVariable* lvalue() const;
  NstExpression* rvalue() const;
  ...
};
```

La méthode `lvalue()` renvoie l'objet représentant la partie gauche de l'affectation, c'est-à-dire la variable affectée (ici `i`), alors que la méthode `rvalue()` renvoie l'objet représentant la partie droite, c'est-à-dire l'expression évaluée et stockée dans la variable (ici `3`). Cette dernière expression est représentée par un objet de la classe `NstExpressionConstant` dont voici un extrait :

```
class NstExpressionConstant : public NstExpression {
  NstConstant* constant() const;
  ...
};
```

La méthode `constant()` renvoie dans notre cas un objet de la classe `NstInteger` que nous avons déjà présentée plus haut. Dans notre cas, cet objet représente la constante littérale entière 3.

Nous avons donc fini notre parcours, en profondeur d'abord, de la structure interne destiné à illustrer les relations entre les différentes classes qui permettent de former la structure. Ce parcours est résumé par la figure 5.4 où `front()` donne le premier objet d'une liste.

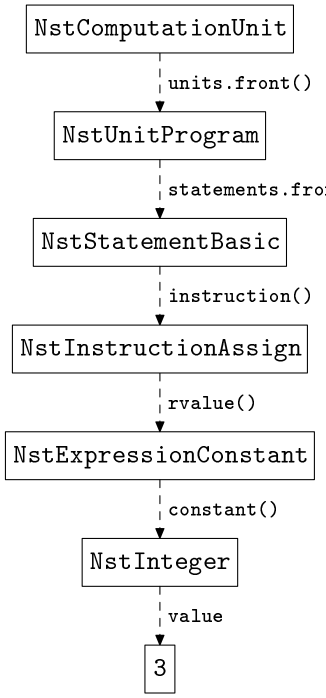


Figure 5.4: parcours en profondeur d'abord de la structure interne de l'exemple 5.1.

5.3.3 La classe `NstTree` et la paternité

La classe `NstTree` est la classe la plus importante de la bibliothèque de base de Nestor. Toutes les classes héritent de cette classe.

Graphe d'héritage de `NstTree`. La figure 5.5 représente une partie du graphe d'héritage de la classe `NstTree`. Cette classe n'hérite d'aucune classe. Nous allons rapidement décrire toutes les classes présentées dans ce schéma, sachant que de plus amples informations peuvent être trouvées dans la documentation en ligne [57].

`NstComputationUnit` Cette classe n'a pas d'héritière. Elle représente un fichier, c'est-à-dire une collection de procédures, de fonctions et éventuellement un programme principal. Cette classe contient deux tables des symboles pour les procédures intrinsèques et les procédures externes (§5.3.7, p. 107). Cette classe est principalement une liste d'objets de la classe `NstUnit`. Elle offre le seul moyen de transformer un fichier en une structure interne en offrant une interface avec la bibliothèque d'analyse syntaxique et sémantique (§5.5, p. 115).

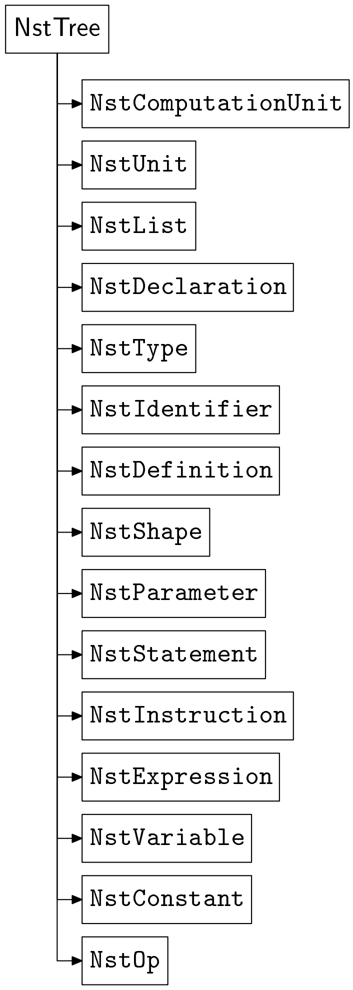


Figure 5.5: le graphe d'héritage partiel de `NstTree`. Les classes concernant les directives HPF et OpenMP ne sont pas représentées.

`NstUnit` Cette classe représente une unité, c'est-à-dire une fonction (`NstUnitFunction`), une procédure (`NstUnitSubroutine`), un programme (`NstUnitProgramme`), etc... Elle contient principalement une liste de déclarations de variables et une liste d'instructions. Un exemple de code Fortran représenté par un objet de la classe `NstUnitProgram` est celui de l'exemple 5.1.

`NstList` Cette classe est un patron de classe, c'est-à-dire une classe avec un paramètre de type générique. Elle permet donc de créer des listes contenant des objets de types différents. Par dérivation, voici quelques listes qui existent dans la bibliothèque de base : `NstUnitList`, `NstDeclarationList`, `NstDefinitionList`, `NstStatementList`. Cette classe fournit un grand nombre de possibilités pour parcourir la liste, rajouter des éléments, supprimer des éléments, réorganiser la liste. Elle est dérivée de la classe `list` de la bibliothèque C++ standard (§5.3.4, p. 100).

`NstDeclaration` Cette classe représente une déclaration de symbole, c'est-à-dire une variable `NstDeclarationVariable`, un paramètre `NstDeclarationParameter`, un type `NstDeclarationType`, etc... Associée à la classe `NstDefinition`, elle offre un support pour la déclaration et l'utilisation de symboles stockés à un endroit unique du programme (§5.3.7, p. 107). Un exemple de déclaration représentée par un objet de la classe `NstDeclarationVariable` :

```
integer*4 i
```

`NstType` Cette classe représente un type de variable. Le type peut être scalaire (`NstTypeInteger`, `NstTypeReal`, ...), tableau (`NstTypeArray`), pointeur (`NstTypePointer`), etc... La principale information contenue dans un type est la taille de ce type, notamment pour les scalaires. Un exemple de type représenté par un objet de la classe `NstTypeInteger` :

```
integer*4
```

`NstIdentifier` Cette classe offre un support pour le stockage d'identificateurs. Elle n'a pas d'héritière. Un identificateur est une chaîne de caractères et cette classe offre un support pour créer des identificateurs uniques par l'intermédiaire d'une table de hachage. Cet aspect est important quand un nouvel identificateur doit être rajouté dans un programme (§5.3.7, p. 107).

`NstDefinition` Cette classe est un peu particulière. Elle représente un symbole du code : une variable (`NstDefinitionVariable`), une fonction (`NstDefinitionFunction`), etc... Elle est la seule classe pour laquelle ses objets peuvent se trouver à plusieurs endroits dans la structure interne (§5.3.3, p. 99). Une définition est la seule à contenir l'identificateur associé à un symbole, permettant ainsi de changer globalement un nom de symbole en une seule instruction. Une définition est la fille de la déclaration qui a créé ce symbole (§5.3.3, p. 99).

`NstShape` Cette classe représente un intervalle, pour définir un tableau par exemple. Un exemple d'intervalle représenté par un objet de la classe `NstShapeExplicit` :

```
5:100
```

`NstParameter` Cette classe représente un paramètre de fonction et peut donc représenter une expression (`NstParameterValue`) ou encore une variable (`NstParameterVariable`).

`NstStatement` Cette classe représente une instruction du programme. Par exemple une boucle (`NstStatementDo`), une instruction conditionnelle (`NstStatementIf`), une boucle while (`NstStatementWhile`), une action (`NstStatementBasic`). Un exemple d'instruction représentée par un objet de la classe `NstStatementDo` :

```
do i = 1, n
  a(i) = 0
enddo
```

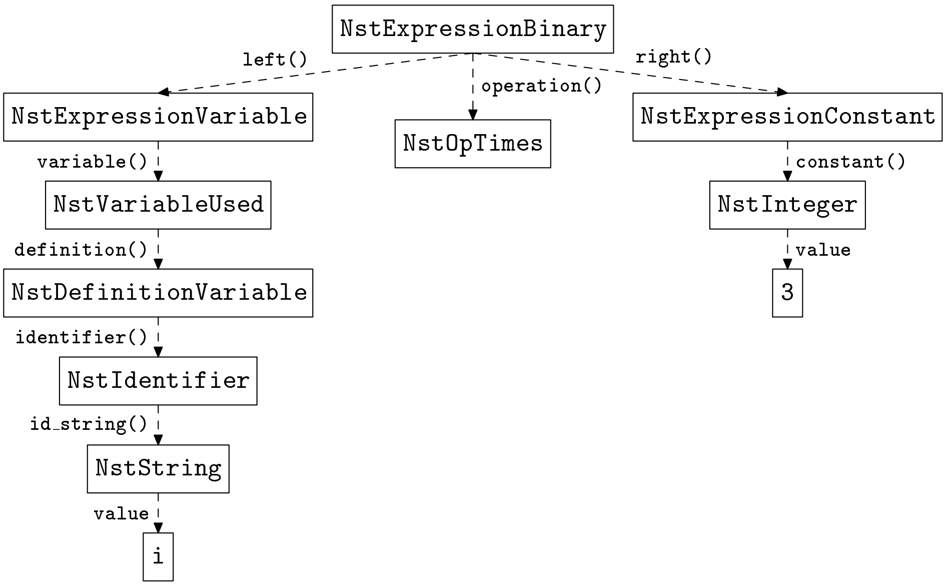
`NstInstruction` Cette classe représente une action du programme, c'est-à-dire une affectation (`NstInstructionAssign`), un appel de procédure (`NstInstructionCall`), un goto (`NstInstructionGoto`), etc... Voici un exemple d'action représentée par un objet de la classe `NstInstructionAssign` :

```
a(i) = 0
```

`NstExpression` Cette classe représente une expression du programme. Cette expression peut être binaire (`NstExpressionBinary`), peut être un accès en lecture à une variable (`NstExpressionVariable`), une valeur constante (`NstExpressionConstant`). Une expression peut contenir d'autres expressions, par exemple dans le cas d'expressions binaires : on obtient dans ce cas un arbre binaire d'expressions représentant l'expression dans le bon ordre d'évaluation. Les priorités entre les opérateurs sont gérées dans cette classe. Un exemple d'expression représentée par un objet de la classe `NstExpressionBinary` :

```
i * 3
```

Voici l'arbre de l'expression précédente (binaire si on ne considère par l'opérateur) :



`NstVariable` Cette classe représente l'utilisation d'une variable (en lecture ou en écriture). Cette variable peut être utilisée dans une expression (`NstVariableUsed`), peut être un compteur de boucle (`NstVariableLoop`), peut être indexée (`NstVariableIndexed`), etc... Cette classe contient principalement un pointeur vers la définition de la variable utilisée (§5.3.7, p. 107). Dans l'arbre binaire précédent, le sous-arbre gauche à partir de `NstVariableUsed`, contient une définition de variable (`NstDefinitionVariable`) : cette définition de variable est partagée par toutes les utilisations de cette variable.

`NstConstant` Cette classe représente une constante littérale. Nous l'avons déjà vue précédemment (voir la figure 5.2).

`NstOp` Cette classe représente un opérateur binaire ou unaire (+, ×, /, −, etc...).

Nous avons donc survolé les différentes classes de base composant la structure interne d'un programme avec Nestor. Nous n'avons pas parlé de toutes les classes permettant de matérialiser les directives de HPF ou de OpenMP. Nous en parlerons plus loin (§5.3.10, p. 113). Nous allons voir maintenant comment la classe `NstTree` supporte la notion de typage pour un objet.

Typage. La classe `NstTree` contient des méthodes donnant des informations sur le type de l'objet dont l'objet `NstTree` est le sous-objet. Rappelons qu'il est impossible de créer un objet de ce type, un objet `NstTree` est toujours sous-objet d'un autre objet. Voici un extrait de la classe `NstTree` concernant les informations sur le type :

```
class NstTree {
  ...
  virtual int type() const;
  virtual const char* class_type() const;
  virtual const char* class_name() const;
  ...
};
```

Ces trois méthodes sont des méthodes virtuelles, c'est-à-dire qu'elles sont redéfinies dans les classes qui héritent de `NstTree` et que la recherche de la bonne méthode à appeler est faite à l'exécution [63]. Ainsi, en considérant l'objet `NstInteger` de la figure 5.1, la méthode `type()` renvoie un entier correspondant à la constante `NST_CONSTANT_INT`, la méthode `class_type()` renvoie la chaîne de caractère `"NST_CONSTANT_INT"` et la méthode `class_name()` renvoie la chaîne de caractère `"NstInteger"`. Ces méthodes sont à priori superflues pour trouver le vrai type d'un sous-objet, puisque `dynamic_cast` permet de le faire facilement, mais elles sont utiles pour afficher le vrai type d'un objet lors de messages d'erreur par exemple.

Paternité. Un aspect important de la conception de cette bibliothèque est que chaque objet n'existe qu'à un seul endroit de la structure (hormis les symboles représentant une variable ou une fonction (§5.3.7, p. 107)). Ceci se traduit par une recopie des objets chaque fois qu'il sont mis à un endroit de la structure. Ce choix d'implantation permet à chaque objet de savoir dans quel objet il est contenu et de stocker une information donnant son type d'utilisation. Cette information est importante pour pouvoir remonter dans la structure. Ce choix de conception permet également d'éviter l'écriture de structures incorrectes et simplifie la gestion mémoire pour l'utilisateur qui n'a pas à se poser la question de savoir quand il doit allouer de la mémoire (nous verrons plus loin le mécanisme de copie d'objets et de déplacement d'objets (§5.3.6, p. 104) qui permet de gérer plus finement la mémoire). Cet aspect de conception permet également d'éviter les cycles dans la structure interne. Voici les méthodes de la classe `NstTree` permettant de retrouver ces informations :

```
class NstTree {
  ...
  NstTree* in() const;
  list <NstTree*> fathers() const;
  string whoami() const;
  ...
};
```

La première de ces trois méthodes, `in()`, renvoie le père d'un objet dans la structure interne. Ainsi, avec la figure 5.4, le père de l'objet `NstExpressionConstant` est l'objet `NstInstructionAssign` (ce dernier objet a également un autre fils qui est la partie gauche de l'affectation). Pour l'objet `NstStatementBasic` de cette même figure, la méthode `in()` ne renvoie pas l'objet `NstUnitProgram` mais la liste `statements` de celui-ci. Un élément d'une liste a comme père cette liste. La liste `statements` a comme père l'objet `NstUnitProgram`. En utilisant `in()`, on peut remonter dans la structure jusqu'à la racine de celle-ci qui est l'objet `NstComputationUnit`. Pour ce dernier, `in()` renverra `NULL`. Le champ `in()` est mis à jour par le père de l'objet à l'aide d'un mécanisme que nous ne décrirons pas ici (mais qui est relativement compliqué car un objet peut être adopté de différentes façons par son père).

La deuxième méthode, `fathers()`, construit la liste de pointeurs vers tous les ancêtres de l'objet dans la structure, ordonnée par la plus proche paternité. Cette méthode peut s'écrire très simplement avec `in()` mais elle offre un raccourci au programmeur.

La troisième méthode, `whoami()`, renvoie une chaîne de caractère décrivant l'utilisation de l'objet. Par exemple pour l'objet `NstExpressionConstant` de notre exemple, `whoami()` renverra `ASSIGNMENT_EXPRESSION`. Cette information est mise à jour par le père de l'objet. Notons que comme pour `in()`, cette information ne peut pas être modifiée par l'utilisateur, empêchant ainsi l'écriture de structures incorrectes en terme de code.

Toutes ces méthodes, ajoutées aux relations entre les classes (§5.3.2, p. 93) permettent de parcourir la structure très aisément, dans tous les sens, sans forcément se cantonner à des parcours en profondeur ou en largeur. Nous verrons plus loin (§5.3.9, p. 109) différents supports intégrés pour toutes sortes de parcours de la structure.

Informations diverses. Chaque objet créé par la bibliothèque standard a un numéro unique qui est stocké dans un attribut `tree_id` de son sous-objet `NstTree`. Voici un extrait de la définition de la classe `NstTree` :

```
class NstTree {
  unsigned long tree_id;
  NstUserInfo user_info;
  ...
};
```

Chaque objet a également un attribut `user_info` qui permet à l'utilisateur de stocker des informations diverses et variées. Voici la définition de la classe `NstUserInfo` :

```
class NstUserInfo {
  unsigned long tag;
  int user1;
  int user2;
  int user3;
  void* info;
  void (*custom_unparse)(ostream& s, const NstTree& t);
};
```

Elle comporte quatre champs entiers `tag`, `user1`, `user2`, `user3` ainsi qu'un pointeur générique `info` permettant de stocker n'importe quelle valeur. Le pointeur de fonction `custom_unparse`, s'il est mis à une valeur différente de `NULL`, désignera une fonction utilisée à la place de la fonction standard d'écriture de la structure dans un flot (§5.3.8, p. 108). Ce pointeur permet au programmeur de définir une fonction d'écriture différente pour chaque objet, si les fonctions d'écriture fournies ne lui conviennent pas où si un objet doit être écrit de manière spécifique. Ce mécanisme permet à l'utilisateur de gérer très finement la façon dont la structure est écrite et permet également de gérer des éléments syntaxiques pour des extensions du langage non gérées pas Nestor.

5.3.4 Les listes

Dans certains cas, un objet peut avoir un nombre variable de fils, comme par exemple l'objet `NstUnitProgram` de notre exemple qui a un nombre variable de déclarations et un nombre variable d'instructions dans son corps. Toutes les listes héritent de la classe `NstList` qui offre un certain nombre d'opérations sur les listes. Voici un extrait de cette classe :

```
template <class T> class NstList : public NstTree, private list<T*> {
  ..
  iterator begin();
  iterator end();
  ...
```

```
  size_type size() const;
  bool empty() const;
  ...
  void push_front(T& x);
  void push_back(T& x);
  iterator insert(iterator position, T& x);
  ...
  void pop_front();
  void pop_back();
  iterator erase(iterator position);
  void clear();
  void remove(T* value);
  ...
  iterator find(T* value);
  ...
  reference front();
  reference back();
};
```

Cette classe est un patron de classe, paramétré par le type d'objet `T` que doit contenir la liste. Elle hérite à la fois de la classe `NstTree` et de la classe `list` de la bibliothèque C++ standard [63], dont elle redéfinit la plupart des méthodes. Pour créer une liste `lst` contenant des instructions, nous pourrions écrire la déclaration suivante :

```
NstList<NstStatement> lst;
```

qui est cependant différente d'une déclaration comme

```
NstStatementList lst;
```

puisque `NstStatementList` hérite de `NstList<NstStatement>` mais rajoute des attributs et des méthodes à celle-ci.

Le concept central d'une liste est la notion d'*itérateur*. Un itérateur est une sorte de « pointeur » vers un élément particulier de la liste. Pour se déplacer d'élément en élément de la liste, un itérateur supporte les mêmes opérations que les pointeurs. Par exemple, si `it` est un itérateur, `it++` fera pointer celui-ci sur l'élément suivant de la liste, alors que `*it` donnera l'élément de la liste pointé par `it`. Les deux premières méthodes de notre liste sont `begin()` et `end()` qui renvoient respectivement un itérateur sur le premier élément de la liste et un itérateur sur la fin de la liste (pas sur le dernier élément mais sur la fin de la liste). Dans la classe `NstList` nous ne stockerons que des pointeurs sur des objets (remarquez le `list<T*>` dans la définition de la classe), ce qui fait que pour accéder à un objet et non pas à son pointeur, nous devrons déréférencer deux fois l'itérateur (`**it`). Voici par exemple une boucle qui itère sur tous les éléments d'une liste d'instructions `lst` et qui affiche celle-ci (nous verrons la signification de l'instruction `cout << **it << endl ;` plus loin (§5.3.8, p. 108)) :

```
NstStatementList::iterator it;
for (it = lst.begin(); it != lst.end(); it++)
  cout << **it << endl;
```

La méthode `size()` renvoie le nombre d'éléments de la liste. La méthode `empty()` renvoie `TRUE` si la liste est vide (auquel cas `begin()==end()` et `size()==0`).

Les méthodes `push_front()` et `push_back()` permettent de rajouter un élément dans la liste respectivement en tête et à la fin de la liste. La méthode `insert()` permet de rajouter un élément avant l'itérateur spécifié. Elle renvoie le nouvel itérateur créé. Il est important de noter que ces trois méthodes recopient récursivement l'objet passé en paramètre avant de l'insérer dans la liste, sauf si cet objet a été coupé de son emplacement d'origine par `cut()` (tout objet est recopié, voir §5.3.3, p. 99). Tous ces concepts de recopie sont expliqués plus loin (§5.3.6, p. 104). Voici par exemple un morceau de programme qui recopie une liste `lst` dans une liste `lst2` vide en inversant l'ordre des éléments :

```
NstStatementList::iterator it;
for (it = lst.begin(); it != lst.end(); it++)
  lst2.push_front(**it);
```

Les méthodes `pop_front()` et `pop_back()` permettent de supprimer respectivement le premier et le dernier élément de la liste. La méthode `erase()` permet d'effacer un élément spécifié par son itérateur. Elle retourne l'itérateur suivant. La méthode `remove()` permet de supprimer un élément spécifié par son adresse. La méthode `clear()` efface tous les éléments de la liste. Voici par exemple comment `clear()` pourrait être définie :

```
void clear()
{
  while (!empty()) pop_front();
}
```

La méthode `find()` permet de retrouver un élément particulier de la liste selon son adresse. Les méthodes `front()` et `back()` retournent respectivement le premier et le dernier élément de la liste (et non pas leurs itérateurs).

Nous avons fait le tour d'à peu près tous les éléments importants de la classe de liste. Nous avons choisi d'utiliser au maximum les patrons de classe standards offerts par le C++ afin d'assurer une meilleure portabilité et de ne pas risquer d'ajouter des erreurs en concevant une nouvelle classe liste, ce qui est toujours une source potentielle d'erreurs conceptuelles et de programmation. Ceci assure également un code plus compact et il existe une documentation complète pour tous les concepts de cette liste. Notons que la librairie standard C++ offre également d'autres types de données structurés utiles, comme les tas, les containers, les vecteurs, etc...

5.3.5 Création et modification des objets

Nous avons vu comment manipuler les objets et comment était organisée la structure. Nous n'avons cependant pas encore parlé de la création de nouveaux objets. Avec `Nestor` il y a deux possibilités de créer des objets. Soit par l'intermédiaire d'un programme Fortran lu par le constructeur `NstComputationUnit`, soit en utilisant les constructeurs des différents objets. En fait, pour écrire une passe de transformation de code, il faut souvent utiliser ces deux possibilités : on commence par créer une nouvelle structure interne représentant un programme Fortran et on rajoute des éléments à la structure (de nouveaux objets) selon la transformation effectuée (de nouvelles variables, de nouvelles instructions, etc...). Nous allons uniquement parler dans cette partie de la dernière possibilité, la première étant décrite dans la suite (§5.5, p. 115).

Chaque objet qui est une feuille dans le graphe d'héritage complet des classes de la bibliothèque de base de Nestor possède un ou plusieurs constructeurs permettant de créer de nouveaux objets. La documentation donne la liste des différents constructeurs associés à chaque classe feuille [57]. Prenons par exemple le type `NstExpressionBinary` dont nous avons parlé plus haut. Voici le constructeur de cette classe :

```
NstExpressionBinary(const NstExpression& l,
                    const NstExpression& r,
                    const NstOp& op)
```

où `l` est l'expression gauche, `r` est l'expression droite et `op` l'opérateur. Chaque paramètre du constructeur sera recopié avant d'être placé dans le nouvel objet créé, sauf ceux ayant fait l'objet d'un `cut()` (voir §5.3.6, p. 104). Cet exemple de constructeur montre que pour créer une expression binaire, il faut déjà avoir créé ses deux fils. Ce principe est appliqué pour toutes les classes de la structure. Il permet d'assurer que tout objet créé est cohérent et complet.

Reprenons l'arbre de l'expression binaire présentée précédemment (§5.3.3, p. 98) et voyons ce qu'il faut faire pour la créer avec Nestor. La première chose à faire est de créer les feuilles de cet arbre, c'est-à-dire les deux constantes littérales `3` et `i` :

```
NstInteger trois(3);
NstString  i_str = "i";
```

Une fois ces deux constantes littérales créées, nous pouvons créer les deux objets juste au dessus dans l'arbre :

```
NstIdentifier i_id(i_str);
NstExpressionConstant trois_expr(trois);
```

et ainsi de suite en remontant :

```
NstDefinitionVariable i_def(i_id);
NstVariableUsed i_used(i_def);
NstExpressionVariable i_expr(i_used);
```

À ce stade, nous pouvons enfin créer notre expression binaire puisque nous avons tous les éléments avec cet appel au constructeur de `NstExpressionBinary` qui va créer un objet `bin_expr` :

```
NstExpressionBinary bin_expr(i_expr, trois_expr, NstOpTimes());
```

Nous n'avons plus besoin de tous les objets qui ont été utilisés pour définir `bin_expr` puisque ceux-ci auront été recopiés récursivement par le constructeur. Cet objet est cohérent et complet. Nous aurions pu écrire cette création de manière plus condensée, par exemple en écrivant

```
NstDefinitionVariable i_def(NstIdentifier(NstString("i")));
NstExpressionVariable i_expr(NstVariableUsed(i_def));
```

ce qui nuit à la lisibilité mais permet d'éviter la création de variables intermédiaires. De la même manière on aurait pu écrire :

```
NstExpressionConstant trois_expr(NstInteger(3));
```

Lorsque nous manipulons une variable, il faut faire attention à ce que cette variable soit déclarée, sinon elle n'aura pas de père. Pour que sa définition ait un père, il est nécessaire de déclarer cette variable (§5.3.7, p. 107). Ainsi, nous devrions rajouter à notre création précédente la ligne :

```
NstDeclarationVariable i_decl(NstTypeInteger(), i_def);
```

qui va recopier `i_def` dans l'objet de la déclaration. Ainsi, le code complet est le suivant :

```
NstDefinitionVariable i_def(NstIdentifier(NstString("i")));
NstDeclarationVariable i_d(NstTypeInteger(), i_def);
NstExpressionVariable i_expr(NstVariableUsed(i_def));
NstExpressionConstant trois_expr(NstInteger(3));
NstExpressionBinary bin_expr(i_expr, trois_expr, NstOpTimes());
```

où la définition du symbole `i` ne sera pas recopiée dans l'objet `NstVariableUsed`, ni dans l'objet `NstDeclarationVariable`, mais seulement utilisée par l'intermédiaire d'un pointeur sur celui-ci. L'objet `NstDeclarationVariable` va cependant « adopter » la définition en devenant son père.

Nous savons donc maintenant comment créer de nouveaux objets. Nous allons voir à présent comment modifier un objet existant. Chaque classe a un certain nombre de méthodes permettant d'accéder aux fils de celle-ci. Par exemple, la classe `NstExpressionBinary` à trois méthodes `left()`, `right()` et `operation()` permettant de récupérer les objets correspondant respectivement à la partie gauche, à la partie droite et à l'opération binaire. Ces trois méthodes sont des méthodes permettant de *lire* les valeurs stockées dans un objet. La plupart du temps, chaque fois qu'il y a une méthode dans une classe permettant de récupérer un objet fils, il y a également une méthode permettant de *modifier* ce fils avec le même nom grâce au polymorphisme offert par C++ [63]. Ainsi, les trois méthodes précédentes ont chacune une méthode correspondante permettant cette fois-ci de modifier l'objet :

```
class NstExpressionBinary : public NstExpression {
  ...
  NstExpression* left(const NstExpression& n)
  NstExpression* right(const NstExpression& n)
  NstOp* operation(const NstOp& n);
  ...
};
```

Par exemple, étant donné l'objet `bin_expr` créé plus haut, il est possible de remplacer sa partie gauche par un autre objet avec la ligne de code ci-dessous :

```
bin_expr.left(NstExpressionConstant(NstInteger(10)));
```

où la variable `i` est remplacée par la constante `10`. Toutes ces méthodes fonctionnent de la même manière : l'objet précédent est détruit, le nouvel objet passé en paramètre est dupliqué récursivement et stocké. Le nouvel objet passé en paramètre n'est pas dupliqué s'il a fait l'objet précédemment d'un `cut()` (§5.3.6, p. 104).

Dans la partie suivante, nous allons explorer une nouvelle facette de la bibliothèque de base, les différents mécanismes qui permettent de déplacer des objets dans la structure sans pour cela les dupliquer.

5.3.6 Copier, couper, coller, remplacer

Nous avons vu à la partie précédente que chaque fois qu'un nouvel objet était créé ou modifié, les objets passés en paramètre étaient dupliqués avant d'être stockés. Ce choix de conception permet de maintenir différentes informations quant à la paternité de chaque objet et assure une moindre gestion de la mémoire de la part du programmeur. Il évite ainsi autant que possible les fuites de mémoire et la nécessité d'avoir un ramasse-miettes. Ce schéma a cependant une exception en ce qui concerne les définitions des symboles d'un programme (§5.3.7, p. 107).

Nous allons présenter dans cette partie tous les mécanismes permettant de dupliquer des objets explicitement et de déplacer des objets dans la structure sans les dupliquer. Ce dernier mécanisme permet de gagner en efficacité en économisant les duplications. Nous n'allons cependant pas déroger à notre règle qui est qu'un objet doit se trouver à un et un seul endroit de la structure, à part pour les définitions.

Duplication et copie d'objets. La classe `NstTree` fournit deux méthodes virtuelles pour la duplication et la copie, `clone()` et `copy()`, et chaque classe fournit un opérateur de copie (`=`) ainsi qu'un constructeur de copie. Voici par exemple l'extrait de la classe `NstExpressionBinary` qui concerne la duplication et la copie :

```
class NstExpressionBinary : public NstExpression {
  ...
  NstExpressionBinary(const NstExpressionBinary& o);
  void copy (const NstExpressionBinary& s);
  NstExpressionBinary& operator= (const NstExpressionBinary& s);
  NstExpressionBinary* clone () const;
  ...
};
```

La méthode `clone()` crée un nouvel objet qui est la duplication récursive de l'objet sur lequel la méthode est appelée. Par exemple, en reprenant l'objet `bin_expr` du paragraphe précédent, nous pouvons créer un nouvel objet qui sera son clone avec la commande :

```
NstExpressionBinary* clone_de_bin_expr = bin_expr->clone();
```

La méthode `copy()` copie tous les attributs d'un objet dans l'objet sur lequel la méthode est appelée, en dupliquant chaque attribut (sauf comme toujours les définitions). Cette méthode est utilisée par le constructeur de copie ainsi que pour la surcharge de l'opérateur `=`. Ainsi, pour créer un nouvel objet `bin_expr2`, qui serait une copie de `bin_expr`, nous pouvons utiliser l'instruction suivante :

```
NstExpressionBinary bin_expr2(bin_expr);
```

Déplacement d'objets. Chaque duplication d'objet est récursive. Cette opération peut donc être relativement coûteuse surtout si nous devons dupliquer des structures très profondes. Il existe donc un mécanisme dans Nestor pour déplacer des objets sans les dupliquer : le `cut()`. Ce mécanisme est une sorte de couper/coller inspiré des traitements de texte mais appliqué aux objets. Il sert à déplacer un objet dans un autre objet sans pour cela le dupliquer. Une méthode plus évoluée permet de couper un objet et de le remplacer en même temps par un autre objet : le `replace()`. Voici l'extrait de la classe `NstTree` contenant ces deux méthodes virtuelles :

```
class NstTree {
  ...
  virtual NstTree* cut();
```

```
  virtual void replace(NstTree& n);
  ...
};
```

La méthode `cut()` coupe le lien existant entre cet objet et son père, en remplaçant l'objet par un objet par défaut créé automatiquement de manière interne. Elle renvoie l'objet par défaut créé, ce qui permet par exemple d'appliquer un replacement (avec `replace()` sur cet objet. Dans le cas d'une liste, l'objet est supprimé de la liste et non remplacé. La méthode `replace()` est identique à `cut()` sauf que l'objet coupé est remplacé dans le père par un clone de l'objet passé en paramètre. Dans le cas d'une liste, l'objet est remplacé dans la liste par un copie de l'objet passé en paramètre (sauf bien sûr si cet objet a été victime d'un `cut()`).

Examinons un exemple simple. Soit par exemple les deux expressions binaires `i+(j+3)` et `k+1` représentées par les objets `e1` et `e2`. Nous voulons déplacer la partie droite de `e1` dans la partie droite de `e2` (et donc supprimer `1`). Comment allons nous procéder ? Tout d'abord, nous allons récupérer la partie droite de `e1` avec l'instruction suivante :

```
NstExpression* e1_d = e1.right();
```

puis nous allons faire un `cut()` sur cette partie droite :

```
e1_d->cut();
```

qui va avoir comme effet de détacher l'objet pointé par `e1_d` de son père, c'est-à-dire de `e1`, et de le remplacer dans celui-ci par un objet par défaut (créé de manière interne). L'expression `i+j+3` ressemblera donc maintenant à ceci :

```
i + <default>
```

et est dans un état non syntaxiquement correct. Maintenant, si nous remplaçons la partie droite de `e2` par l'objet `e1_d`, celui-ci ne sera pas dupliqué mais simplement attaché à son nouveau père :

```
e2.right(*e1_d);
```

après avoir bien sûr détruit la partie droite précédente de `e2`. Il ne nous reste plus qu'à remettre `e1` dans un état cohérent en lui fournissant une nouvelle partie droite :

```
e1.right(NstExpressionConstant(NstInteger(1)));
```

Les deux expressions sont maintenant `i+1` et `k+j+3`. Nous aurions pu éviter cette dernière instruction si nous avions utilisé `replace()` au lieu de `cut()` :

```
e1_d->replace(NstExpressionConstant(NstInteger(1)));
```

Quelques remarques. Un objet coupé est un objet dont le père est un objet global géré par le système (appelé `cut_buffer`). C'est grâce à ce mécanisme que le système est en mesure de déterminer si un objet a été coupé ou non. Chaque constructeur ou méthode de modification d'une classe teste si un objet passé en paramètre n'a pas comme père l'objet `cut_buffer` avant de recopier l'objet. Lorsqu'un objet est créé par l'utilisateur (avec `new` ou alors en tant que variable automatique), il n'a pas le statut d'objet ayant été coupé mais est tout simplement orphelin. L'utilisateur a la possibilité d'effectuer un `cut()` sur cet objet (on peut faire un `cut()` même sur un objet n'ayant pas de père), sachant que celui-ci ne sera alors pas recopié si il est passé en paramètre d'un constructeur ou d'une méthode de modification d'un autre objet. C'est alors la responsabilité de l'utilisateur de s'assurer que l'objet recevant l'objet coupé n'a pas une durée de vie supérieure à l'objet coupé (auquel cas l'objet coupé disparaitrait en laissant son père dans un état non cohérent). Ce mécanisme permet à l'utilisateur de gérer ses objets plus finement mais lui demande en contrepartie plus d'attention.

5.3.7 Les identificateurs et les symboles

Gestion des symboles. Divers problèmes se posent lorsque l'on considère les symboles dans une structure interne. Nous devons fournir une méthode suffisamment simple pour gérer une table des symboles, afin de retrouver le type et la déclaration d'une variable utilisée dans le code. Nous devons également avoir un mécanisme d'imbrication de ces tables des symboles pour permettre de déterminer la portée des variables. Dans Nestor, un symbole est représenté par un objet de la classe `NstDefinition` ou d'une de ses classes dérivées. Cet objet contiendra l'élément constitutif du symbole qui est son identificateur. Un identificateur est représenté par un objet de la classe `NstIdentifier`. Une table des symboles est tout simplement une liste de définitions (`NstDefinitionList`). Plutôt que d'imaginer un schéma compliqué de création des définitions et de gestion de la table des symboles, nous avons tenu à garder tout ceci aussi simple que possible, en gardant uniquement deux principes en tête :

- une définition est unique et n'est dupliquée nulle part ;
- le seul père possible d'une définition est la déclaration de son symbole.

Une définition est répétée à plusieurs endroits, il est difficile de savoir qui doit la détruire (on peut penser que c'est la déclaration qui devrait détruire l'objet, mais ce n'est pas elle qui l'a créé). Nous avons donc décidé pour plus de sûreté de laisser cette gestion au programmeur. La déclaration d'une variable peut cependant se faire simplement en utilisant la méthode `declare_variable()` de la classe `NstDeclarationList` qui prend en paramètre le type et le nom de la variable, et renvoie la définition du symbole créé et déclaré dans la liste. Si cette liste est dans un objet qui a une table des symboles, la définition est rajoutée à cette table. Rappelons que c'est ensuite à la charge du programmeur d'éventuellement détruire la définition créée.

Toute définition contient la liste de tous les objets qui utilisent cette définition. Cette liste est définie dans `NstDefinition` est s'appelle `uses`. Cet aspect confère un aspect très utile à une définition puisqu'elle permet de centraliser toutes les utilisations d'un symbole dans la structure. Cette information est utile pour déterminer le flot de données d'un programme, les dépendances de donnée, etc.

Pour déclarer proprement une variable dans un programme, comme par exemple la déclaration de la variable `i` dans l'exemple 5.1, voici ce qu'il faut écrire, avec `dummy` qui est un objet de la classe `NstUnitProgram`. Tout d'abord, il faut créer la définition du symbole :

```
NstIdentifier i_id(NstString("i"));
NstDefinitionVariable* i_def = new NstDefinitionVariable(i_id);
```

où nous utilisons `new` pour allouer de la mémoire explicitement. Ensuite, nous pouvons créer la déclaration de la variable :

```
NstDeclarationVariable i_decl(NstTypeInteger(), *i_def);
```

où l'objet `i_def` ne sera pas dupliqué mais simplement utilisé. Puisque nous sommes dans un programme, nous devons également insérer cette déclaration dans la liste des déclarations et dans la table des symboles du programme :

```
dummy.declarations.push_back(i_decl);
dummy.definition()->symbols.push_back(*i_def);
```

Nous aurions pu également utiliser la méthode `declare_variable()` pour effectuer le travail équivalent aux quatre dernières lignes. Voici une autre façon de créer une définition et de la déclarer en même temps :

```
NstIdentifier i_id(NstString("i"));
NstDefinitionVariable* i_def;
i_def = dummy.declarations.declare_variable(NstTypeInteger(), i_id);
```

Lorsque la structure ayant comme racine l'objet `dummy` sera détruite, il faudra également détruire explicitement `i_def` :

```
delete i_def;
```

Définitions d'unités. Une définition est attachée à chaque fonction, programme, procédure, etc... Cette définition contient la table des symboles de cette unité (une liste appelée `symbols`). Cette table des symboles contient la liste des symboles déclarés localement dans l'unité. Le père d'une définition d'unité (`NstDefinitionUnit`) est l'unité qu'elle définit (`NstUnit`). Ainsi, étant donnée la définition d'une unité avec sa table des symboles, il est possible de remonter à l'unité englobante grâce à la méthode `in()` des ancêtres successifs de la définition de l'unité. La gestion de l'imbrication des tables de symboles est donc présente naturellement dans la structure. La portée d'une variable définie dans la table des symboles d'une définition d'unité est cette unité et toutes les unités qu'elle englobe. La méthode virtuelle `search()` qui est définie dans `NstTree` permet de retrouver une définition selon un identificateur dans l'éventuelle table des symboles d'un objet.

Unicité des identificateurs. Certaines transformations de code imposent d'introduire de nouvelles variables dans le programme. Un problème se pose dans ce cas de figure : comment faire pour être sûr d'introduire une variable dont le nom n'existe pas déjà dans le programme ? La classe `NstIdentifier` permet de résoudre ce problème en gérant une table de hachage qui va contenir tous les identificateurs du processus et qui permet de tester facilement si un identificateur existe déjà ou non. Le constructeur par défaut de `NstIdentifier`, sans paramètre, permet de créer un identificateur de la forme `nst_`d où d est un entier unique (un compteur est géré globalement).

5.3.8 Affichage de la structure (« unparse »)

Tout objet de la structure interne peut être écrit sur un flot en écriture (`ostream`) grâce à l'opérateur de sortie `<<`. Cet opérateur n'est défini que dans la classe `NstTree` mais appelle une méthode virtuelle `unparse()` qui est définie dans chaque classe (cette méthode est privée). Ainsi, chaque classe possède sa propre fonction d'écriture de la structure (en anglais, cette écriture est appelée l'« unparse »). Par exemple, si le programme de l'exemple 5.1 est représenté par un objet `dummy`, la ligne suivante

```
cout << dummy << endl;
```

affiche le programme sur la sortie standard. L'écriture est paramétrable globalement par plusieurs fonctions. Par exemple, la fonction `nst_fortran_type()` permet de choisir entre divers types de directives en sortie : HPF 2.0, OpenMP, les directives Cray et les directives

Sun. La fonction `nst_unparse_mode()` permet de choisir le langage de sortie : Fortran ou Petit (§5.6, p. 117). Ainsi, chaque classe doit fournir deux méthodes pour écrire la structure, une pour Fortran et une pour Petit, sachant qu'il y aura un seul code si l'écriture est identique.

La redéfinition de l'opérateur d'écriture dans un flot (<<) permet d'avoir une grande souplesse pour écrire la structure. L'écriture sera la même que l'on écrive sur l'écran ou dans un fichier. D'autre part, cela facilite grandement l'écriture des méthodes `unparse()` dans les classes. Par exemple, voici la méthode `unparse()` pour le Fortran de la classe `NstStatementIf` :

```
void NstStatementIf::unparse_fortran (ostream& s) const
{
  my_label_indent (s, label);
  s << "if (" << *condition() << ") then" << endl;
    my_inc_indent ();
    s << then_part;
    my_dec_indent ();
  if (else_part.size() > 0)
    {
      my_indent(s);
      s << "else" << endl;
        my_inc_indent ();
        s << else_part;
        my_dec_indent ();
    }
  my_indent(s);
  s << "end if";
}
```

5.3.9 Parcours de la structure

Méthodes pour un parcours générique. Nous avons vu dans les parties précédentes qu'il y avait quantité de liens permettant de se déplacer dans la structure. Cependant, puisque chaque objet peut être d'une classe différente, il n'est pas aisé de définir un parcours générique de la structure. Chaque classe qui représente un élément syntaxique de Fortran a un certain nombre de fils. Par exemple, la classe `NstExpressionBinary` a trois fils : `left()`, `right()` et `operation()`. Si nous voulions faire un parcours générique de toute la structure et appliquer une certaine action à toutes les expressions binaires, il nous faudrait écrire une fonction `parcourir()` dont voici un extrait :

```
void parcourir(NstTree* t)
{
  ...
  if (NstExpressionBinary* eb =
         dynamic_cast<NstExpressionBinary*>(t))
    {
       action(eb);
       parcourir(eb->left());
       parcourir(eb->operation());
```

```
      parcourir(eb->right()):
    }
  ...
}
```

où il faudrait qu'il y ait une instruction `if` par type de classe de la structure interne (pour permettre un parcours complet). D'autre part, il faudrait réécrire cette fonction pour chaque `action()` que l'on veut voir effectuer durant le parcours. Encore plus contraignant, si une classe était rajoutée à la structure interne, il faudrait réécrire toutes les fonctions de parcours, ce qui est inimaginable. Nous avons donc choisi d'offrir des méthodes qui permettent de rendre ce parcours complètement générique. Dans cette optique, nous fournissons un accès générique à tous les fils d'une classe. Voici l'extrait de la classe `NstTree` qui contient les méthodes d'accès générique aux fils d'une classe :

```
class NstTree {
  ...
  virtual NstTree* init();
  virtual NstTree* init(NstTrees::iterator &it);
  virtual NstTree* next();
  virtual NstTree* next(NstTrees::iterator &it);
  ...
};
```

La méthode `init()` renvoie le premier fils d'une classe, ou `NULL` si cette classe n'a pas de fils (le premier fils est le premier dans l'ordre textuel des déclarations dans la classe ou dans la documentation). Les appels à la méthode `next()` consécutifs à un appel à `init()` renvoient les différents fils de la classe. Avec ces méthodes, nous pouvons réécrire notre fonction `parcourir()` beaucoup plus facilement :

```
void parcourir(NstTree *t)
{
  if (NstExpressionBinary* eb =
         dynamic_cast<NstExpressionBinary*>(t))
    action(eb);
  NstTree *c = t->init();
  while (c)
    {
      parcourir(c);
      c = t->next();
    }
}
```

Ces quelques lignes suffisent à définir un parcours complet de toute la structure et l'application d'une action sur tous les objets de la classe `NstExpressionBinary` de la structure interne.

Les deux méthodes `init()` et `next()` fonctionnent en conservant dans l'objet sur lequel elles sont appliquées un itérateur sur une liste interne contenant tous les fils. Cela peut poser des problèmes si le programmeur veut faire plusieurs parcours simultanés sur le même objet. C'est pour cette raison qu'il y a deux variantes d'`init()` et de `next()` qui prennent en paramètre l'itérateur sur cette liste externe (qui est de type `NstTrees`).

Afin de simplifier encore plus les parcours, la classe `NstTree` contient une méthode

```
class NstTree {
  ...
  void traversal(int (*action)(NstTree*, void*), void *extra = NULL);
  ...
};
```

qui permet de faire un parcours récursif depuis un objet, en appliquant la fonction `action()` sur chaque sommet de la structure, et qui a comme paramètre un pointeur sur un objet `NstTree` et un pointeur générique qui est passé à la méthode `traversal()` par le paramètre `extra` (qui est à `NULL` par défaut). Le parcours s'arrête lorsque `action()` renvoie une valeur différente de 0. Nous verrons plusieurs utilisation de cette méthode dans la suite, lorsque nous donnerons des exemples d'utilisation de Nestor (§5.9.2, p. 127, §5.9.3, p. 130).

Classe pour un parcours générique. Nous avons également dans la bibliothèque de base une classe dédiée au parcours de la structure interne qui s'appelle `NstTreeTraversal`. Par défaut, cette classe effectue un parcours de la structure interne depuis un objet passé en constructeur de la classe. Toutes les méthodes de cette classe sont virtuelles, ce qui permet de les redéfinir de manière aisée. Voici la définition complète de cette classe :

```
class NstTreeTraversal {
public:
  virtual int pre_action(NstTree* t);
  virtual int in_action(NstTree* t, NstTree* st, int num);
  virtual void post_action(NstTree* t);
  virtual void traversal(NstTree* t);
};
```

où `pre_action()` est une méthode appliquée sur chaque élément avant la récursion (si cette méthode renvoie 1, la récursion se termine), où `post_action()` est une méthode appliquée sur chaque élément après la récursion et où `in_action()` est une méthode appliquée sur chaque objet fils d'un élément (la récursion se termine si cette méthode renvoie 1). Le premier paramètre de cette méthode est l'élément courant, le deuxième est le fils courant et le troisième est le numéro du fils. Voici le code de la méthode `traversal()` de cette classe :

```
void
NstTreeTraversal::traversal(NstTree* t)
{
  assert(t);
  if (pre_action(t) == 0)
    {
      int stn = 0;
      NstTrees::iterator it;
      NstTree *current = t->init(it), *nc;
      while (current)
        {
          stn++;
          nc = t->next(it);
          if (in_action(t, current, stn) == 0)
            traversal(current);
          current = nc;
```

```
      }
    }
  post_action(t);
}
```

Cette classe est faite pour être étendue. Par exemple, la classe `NstTreeTraversalPrint` étend simplement la classe `NstTreeTraversal` en redéfinissant le deux méthodes virtuelles `pre_action()` et `post_action()` pour obtenir un affichage récursif de tous les types des objets depuis un objet de la structure interne, avec une gestion de l'indentation permettant de voir l'imbrication des objets :

```
class NstTreeTraversalPrint : public NstTreeTraversal {
public:
  int pre_action(NstTree* t)
    {
      my_indent(cout);
      cout << t->class_name() << ":" << t->class_type() << endl;
      my_inc_indent();
      return 0;
    }
  void post_action(NstTree* t)
    {
      my_dec_indent();
    }
};
```

Ce type de parcours, qui gère l'indentation, aurait été impossible à réaliser avec la simple méthode `traversal()` de `NstTree`, puisqu'il faut exécuter une action avant et après l'affichage des fils d'un objet. La classe `NstTreeTraversal` permet d'écrire simplement des parcours compliqués.

Méthode pour un parcours en « hauteur ». Tous les parcours que nous avons vus jusqu'à présent permettent de « descendre » dans la structure depuis un objet donné. Une dernière méthode de parcours permet cette fois-ci de « remonter » dans la structure grâce à la méthode `in()` et de rechercher un objet particulier selon une fonction `comptest` passée en paramètre. Le parcours s'arrête lorsque `in()` renvoie `NULL` ou lorsque `comptest` renvoie une valeur différente de 0. Le paramètre `extra` a la même signification que pour la méthode `traversal()`. Voici la définition de cette méthode :

```
class NstTree {
  ...
  NstTree* upsearch(int (*comptest)(NstTree*, void*), void* extra = NULL);
  ...
};
```

Nous avons donc vu tous les parcours génériques implantés dans la bibliothèque de base de Nestor. Ces différents parcours permettent un déplacement aisé dans la structure et forment un canevas puissant pour écrire des transformations sur des parties précises du code. Il est par exemple très facile de rechercher dans la structure toutes les parties « marquées » du code et de leur appliquer une transformation.

5.3.10 Les directives d'HPF et d'OpenMP

Directives supportées. Pour finir cette partie sur les classes de la bibliothèque de base de Nestor, nous allons présenter les différentes classes qui permettent de gérer les directives d'HPF et d'OpenMP. En ce qui concerne HPF, les distributions, alignements, processeurs, sont gérés par des classes héritières de `NstDeclaration`. Par exemple, la classe `NstDeclarationDistribute` permet de gérer les déclarations de distribution de tableau de la forme

```
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)
```

où `A` est le tableau à distribuer. De même, la classe `NstDeclarationAlign` permet de représenter les spécifications d'alignement de la forme

```
!HPF$ ALIGN B WITH A
```

où `B` doit être aligné avec `A`.

Les directives de parallélisation d'HPF et d'OpenMP, ainsi que la plupart de celles définies par le comité X3H5 [77] sont représentées dans Nestor par des classes héritières de la classe `NstStatement`. Par exemple, une région parallèle est définie par la classe `NstStatementTask`, et correspond à une tâche au sens d'HPF ou d'OpenMP. Les boucles parallèles sont spécifiées grâce à une classe `NstLoopInfo` qui contient des informations concernant l'exécution parallèle éventuelle de la boucle : les variables privées de la boucle, les variables partagées, les réductions, etc. Le drapeau `independent` de cette classe est un booléen permettant de dire si la boucle est parallèle ou non. Lors de l'écriture de la structure, la valeur de retour de la fonction `nst_fortran_type()` permet de déterminer dans quel dialecte une boucle parallèle devra être écrite. Ainsi, voici le résultat de l'écriture d'une boucle parallèle avec HPF, sachant que la variable c est privatisée (`nst_fortran_type()==HPF`) :

```
!HPF$ INDEPENDENT, NEW(c)
do i = 1,n
  c = ...
enddo
```

et la même sortie avec OpenMP (`nst_fortran_type()==OpenMP`) :

```
!$OMP PARALLEL DO SHARED(n) PRIVATE(i,c)
do i = 1,n
  c = ...
enddo
```

Il existe également deux instructions (`NstInstructionRedistribute` et `NstInstructionRealign`) pour représenter les redistributions et les réalignements en HPF 2.0. Les autres classes sont décrites dans la documentation [57].

Directives non supportées. Les directives non supportées ou exotiques peuvent toujours être spécifiées « à la main » par le programmeur. La première possibilité de spécification d'une directive est d'utiliser les deux classe `NstDeclarationComment` et `NstStatementComment` qui contiennent toutes deux une chaîne de caractères qui sera écrite telle quelle lors de l'écriture de la structure.

D'autre part, les classes `NstUnit`, `NstDeclaration` et `NstStatement` contiennent chacune deux attributs de type `string` appelés `above_comment` et `below_comment` qui contiennent respectivement du texte à écrire avant et après l'écriture de l'unité, de la déclaration ou de l'instruction. La bibliothèque d'analyse syntaxique et sémantique récupère les commentaires dans le code original, donc les directives non supportées par celle-ci sont quand même conservées sous la forme de commentaires (§5.5, p. 115).

5.4 MétaNestor

Après avoir développé les classes de la bibliothèque de base, nous nous sommes rendus compte que celles-ci pouvaient être générées depuis un fichier de description très simple : la régularité de chaque classe et la forme même de la représentation interne d'un programme permet de générer automatiquement la définition et le code des classes de la bibliothèque de base. Ainsi, nous avons développé un petit programme appelé `metanestor` qui prend en entrée un fichier de définition comme celui-ci :

```
NstStatement NST_STATEMENT NST\_STATEMENT NstTree nop
int f label LABEL label
int f line_number LINE_NUMBER line_number
int f is_commented COMMENTED_STATEMENT is_commented
string f above_comment ABOVE_COMMENT above_comment
string f below_comment ABOVE_COMMENT below_comment
q
NstStatementIf NST_STATEMENT_IF NST\_STATEMENT\_IF NstStatement nop
NstExpression p e IF_EXPRESSION condition
NstStatementList l then_part IF_THEN_PART then_part
NstStatementList l else_part IF_ELSE_PART else_part
q
NstStatementDo NST_STATEMENT_DO NST\_STATEMENT\_DO NstStatement nop
NstVariableLoop p e STATEMENT_DO_ID id
NstExpressionSlice p s STATEMENT_DO_RANGE range
NstStatementList l body STATEMENT_DO_BODY body
NstLoopinfo p li STATEMENT_DO_INFO info
q
NstStatementList NST_STATEMENT_LIST NST\_STATEMENT\_LIST
    NstList<NstStatement> nop
q
x
```

qui définit les classes `NstStatement`, `NstStatementIf`, `NstStatementDo` et `NstList`, et qui génère la définition et le code de toutes ces classes, avec un canevas de documentation pour chaque classe. La bibliothèque de base de Nestor peut donc être automatiquement regénérée depuis ces fichiers de définition. Bien sûr, il faut rajouter des méthodes ou des attributs dans certaines classes, mais cela simplifie énormément le travail.

Ainsi, dans l'exemple précédent, la définition de la classe `NstStatementIf` permet de générer la définition de classe dont voici un extrait :

```
class NstStatementIf : public NstStatement {
public:
  int type() const { return NST_STATEMENT_IF; }
  const char* class_type() const { return "NST_STATEMENT_IF"; }
  const char* class_name() const {return "NstStatementIf"; }
```

```
private:
  NstExpression* e;
public:
  NstExpression* condition() const { return e; }
  NstExpression* condition(const NstExpression& n);
public:
  NstStatementList then_part;
private:
  NstTree* then_part_ptr;
public:
  NstStatementList else_part;
private:
  NstTree* else_part_ptr;
  ...
};
```

et dont nous avons supprimé toute la documentation (qui est elle même générée automatiquement). Parallèlement, `metanestor` génère également les parties génériques du code de toutes les méthodes dans un autre fichier. Par exemple, tous les mécanismes de `cut()` et `replace()` sont pris en charge par `metanestor`.

Cette approche permet de rajouter des classes très facilement. Il suffit d'écrire un petit fichier de définition et de lancer le programme de génération de classe pour avoir une nouvelle classe qui peut être intégrée dans les sources.

5.5 La bibliothèque d'analyse syntaxique et sémantique

Nous avons vu dans la partie décrivant les classes de base de la bibliothèque comment créer une structure interne décrivant un programme. Nous allons voir dans cette partie comment créer une structure interne depuis le code source d'un programme en Fortran.

5.5.1 La partie frontale d'Adaptor

Adaptor est un compilateur HPF du domaine public développé par Thomas Brandes [13]. Il inclut une partie frontale permettant de construire un arbre syntaxique et d'effectuer une analyse sémantique depuis un programme en Fortran 77 ou 90 avec des directives HPF 2.0. Nous avons choisi d'utiliser cette partie frontale car elle est robuste, disponible gratuitement et qu'elle est construite grâce à une boîte à outils (également gratuite) qui permet de l'étendre et de la modifier facilement (GMD compiler toolbox [13]). Par exemple, nous avons ajouté à cette partie frontale des directives propres à Nestor et les directives d'OpenMP. Nous avions comme objectif dès le départ d'utiliser une partie frontale qui nous permettrait de traiter des vrais codes plutôt que de considérer un langage *ad-hoc* comme dans de nombreux outils expérimentaux.

Historiquement, Nestor a commencé à être développé en vue d'être une phase de parallélisation préalable du code pour Adaptor. Partant d'un programme Fortran séquentiel, Nestor suivi d'Adaptor devait pouvoir détecter du parallélisme dans les nids de boucles (le « nest » de Nestor), et générer un programme parallèle avec une bibliothèque de passage de messages. Cet objectif a été dépassé puisque Nestor permet effectivement de faire cela mais est également une structure ouverte et extensible d'implantation de nouvelles transformations. De plus, nous ne sommes pas limités à HPF.

La partie frontale d'Adaptor a été encapsulée dans une bibliothèque. Une fois que le code a été lu et que l'arbre a été créé en mémoire, cet arbre représente un programme Fortran syntaxiquement et sémantiquement correct.

5.5.2 Directives de marquage du code

Nous avons étendu la partie frontale d'Adaptor pour qu'elle conserve une trace des commentaires dans l'arbre syntaxique (ce qui n'était pas fait dans Adaptor). Nous voulions que ces commentaires soient conservés pour deux raisons. La première est que dans le cadre de transformations au niveau du code source, nous voulions que l'utilisateur retrouve un code après transformation qui ressemble dans la mesure du possible à son code original. La deuxième est que nous voulons pouvoir conserver des commentaires qui sont en fait des directives de parallélisation non reconnues comme telles par la partie frontale.

Afin de pouvoir marquer des portions de code devant être examinée en vue d'une transformation éventuelle, nous avons introduit trois directives qui sont prises en compte par la partie frontale et intégrées à l'arbre syntaxique. Ces trois directives commencent par la sentinelle `!NST$` ou `!NESTOR$`. La première de ces directives est la directive `SINGLE` qui permet de mettre un drapeau dans l'objet `NstLoopInfo` d'une boucle `do` ou `forall`. Elle s'utilise de la façon suivante :

```
!NST$ SINGLE
  do i = 1, n
    ...
  enddo
```

La deuxième de ces directives est plus précisément un couple de directives `BEGIN` et `END` qui permettent de spécifier une région contenant des instructions (un objet `NstStatement-Nestor` qui contient une liste d'instructions). Ces directives s'utilisent de la façon suivante :

```
!NST$ BEGIN
  do i = 1, n
    ...
  enddo
  a = ...
  do i = 0, n-1
    ...
  enddo
!NST$ END
```

La troisième de ces directives (`ALL`) doit être placée obligatoirement entre les déclarations et les instructions d'une routine. Elle est équivalente à créer une région avec `BEGIN` et `END` contenant toutes les instructions de la routine.

5.5.3 Interface avec la bibliothèque de base

Il y a un seul moyen de créer une structure interne Nestor depuis un programme Fortran, c'est d'utiliser le constructeur de la classe `NstComputationUnit` qui a la forme suivante :

```
NstComputationUnit(const char* file_name, unsigned long options = NoOptionsMask)
```

où `file_name` est le nom du fichier Fortran à lire et `options` représente les options de la partie frontale (format du programme, mode spécial équivalent à la présence d'une directive `ALL` au début de chaque routine). Si `file_name` est égal à `NULL`, la lecture du programme se fera sur l'entrée standard. Ce constructeur utilise la partie frontale pour créer un arbre syntaxique du programme. Une fois cet arbre créé, il crée une structure interne Nestor en parcourant cet arbre à l'aide de fonctions d'accès permettant d'accéder à tous les nœuds de l'arbre. Une fois que la structure interne est créée, l'arbre syntaxique peut être détruit. L'objet de la classe `NstComputationUnit` contient une table des symboles spéciale pour les routines externes, intrinsèques et les segments `common` du code en entrée.

Voici par exemple le constructeur de la classe `NstStatementIf` permettant de faire l'interface avec l'arbre syntaxique de la partie frontale :

```
NstStatementIf::NstStatementIf(void* t, NstDefinition* obj)
  : NstStatement(t, obj),
    then_part (nst_statement_body(t), obj),
    else_part (nst_statement_else(t), obj)
{
  e = newNstExpression(nst_statement_expr(t), obj);
  initsubtrees();
  init_fathers();
}
```

Ce constructeur est construit automatiquement depuis le fichier de définition de la classe `NstStatementIf`. Les fonctions d'interfaçage avec l'arbre syntaxique sont `nst_statement_-body()`, `nst_statement_else()` et `nst_statement_expr()` et sont à écrire « à la main ».

5.6 La bibliothèque de manipulation de graphes

Analyse des dépendances. L'analyse des dépendances de données (voir §2.5, p. 19) est la première étape avant toute optimisation qui modifie l'ordre des calculs dans un programme. Sans un analyseur de dépendance sophistiqué, les transformations de code comme les transformations de boucle, l'expansion scalaire, la privatisation de tableau, la suppression du code mort, sont impossibles. Tout outil de parallélisation se doit donc de contenir un analyseur de dépendances de données.

Il est avéré que le développement d'un analyseur de dépendances qui soit à la fois puissant et rapide est une tâche très difficile. Il peut cependant être résolu « rapidement » grâce à un ensemble de méthodes dédiées à des cas particuliers qui surviennent couramment dans les codes. Pour cette raison, nous avons décidé de nous reposer sur un outil logiciel libre appelé Petit [49] développé à l'université du Maryland par l'équipe de Bill Pugh. L'entrée de Petit est un programme écrit dans un langage propre qui ressemble à un sous-ensemble du C. Sa sortie est un fichier contenant des paires de références à des emplacements mémoire impliqués dans des dépendances de donnée. Chaque dépendance est représentée par une relation fondée sur l'arithmétique de Presburger.

Une possibilité d'intégrer Petit à Nestor aurait été d'introduire toutes les techniques de Petit directement dans la structure interne de Nestor. Cependant, comme ce dernier accepte le langage Fortran en entier et que Petit accepte uniquement un sous-ensemble de langage, ce n'est pas si simple de modifier les sources de Petit, même pour les personnes qui l'ont développé. Ainsi, en suivant les conseils de Bill Pugh, nous avons choisi d'utiliser Petit comme un outil indépendant à travers des fichiers. Cette stratégie n'est pas seulement plus simple, elle a aussi l'avantage de permettre le remplacement aisé de l'analyseur de dépendance (par exemple par une nouvelle version de Petit ou un autre analyseur). Elle apporte également un confort de poids dans le développement : les bogues potentiels de Petit et de Nestor sont séparés. Une fois cette stratégie choisie, il restait deux problèmes à résoudre : donner un bon programme à Petit en entrée et faire correspondre sa sortie avec la structure interne de Nestor.

Programme d'entrée de Petit. La première tâche était facile à accomplir, grâce à la structure même de Nestor et de ses possibilités de créer une nouvelle façon d'écrire le code (§5.3.8, p. 108). Ainsi, nous avons redéfini un nouveau langage de sortie dédié à Petit qu'il suffit de choisir par l'intermédiaire de la fonction globale `nst_unparse_mode(UnparsePetit)`. Ce nouveau langage de sortie a été rapide à définir puisqu'il partage une partie de sa structure avec Fortran. Par exemple les expressions s'écrivent de la même manière, il n'y a donc pas eu besoin de redéfinir une nouvelle méthode d'écriture. Certains éléments de la structure interne n'existent pas dans le langage de Petit. Dans ces cas là, l'utilisation de l'opérateur `<<` provoque une erreur.

Comme nous l'avons dit précédemment, Petit prend en entrée un programme complet écrit dans son langage. Pour ne pas être obligé d'analyser un code Fortran complet, c'est-à-dire toute la structure interne de Nestor, nous offrons la possibilité d'analyser une seule instruction (un objet `NstStatement`) et la méthode d'écriture de la structure (`<<`) génère un programme depuis celle-ci en rajoutant les déclarations de toutes les variables utilisées dans cette instruction. Il est important de noter que comme une instruction peut en fait être une région contenant des instructions, nous pouvons traiter n'importe quel taille de programme. Cette technique d'écriture « locale » permet de ne pas limiter la taille des codes analysés et offre une souplesse d'utilisation qui est en accord avec notre principe de traitement interactif du code.

Lors de l'écriture de la portion de structure interne qui doit être analysée, chaque accès à la mémoire par l'intermédiaire d'une variable est numéroté. Plus précisément, chaque objet `NstVariableUsed` qui représente un accès à une variable va voir son champ `user_info.tag` modifié pour contenir le numéro de l'accès mémoire. La numérotation se fait en suivant l'ordre d'apparition dans le code.

Sortie de Petit. Une fois que Petit a généré un fichier de dépendances d'une portion de structure interne de Nestor, il reste à faire correspondre ces dépendances avec la structure interne. L'utilisation des numéros de ligne n'est pas suffisante car les deux langages peuvent être formattés de manière différente et de plus, nous ne passons qu'une portion de code à Petit. Nous avons donc légèrement modifié la grammaire de Petit (2 lignes) pour que celui-ci numérote les accès aux tableaux dans le même ordre que la structure interne de Nestor lors de l'écriture de celle-ci. Ainsi, nous pouvons facilement faire la correspondance entre les accès mémoire des dépendances du fichier et les objets de la structure interne.

Nous avons modifié la façon dont les dépendances sont représentées dans le fichier de sortie de Petit. Dans la plupart des algorithmes de parallélisation, nous avons besoin d'une approximation des vecteurs de distance et pas d'une formule de Presburger. Nous avons écrit un petit outil qui se fonde sur la Polylib [71] (une bibliothèque de manipulation de polyèdres) pour extraire depuis une formule de Presburger une description des dépendances par une approximation par niveau, vecteur de direction et polyèdre, les trois représentations utilisées respectivement par les algorithmes de parallélisation d'Allen et Kennedy [2], Wolf et Lam [73] et Darte et Vivien [24]. L'exemple 5.2 illustre les relations de dépendance qui sont disponibles dans notre implantation de Petit. C'est le cas typique où les approximations par niveau et vecteur de direction ne sont pas suffisantes pour détecter du parallélisme.

do $i = 1, n$
  do $j = 1, n$
    $a(i,j) = a(j,i) + a(i,j-1)$ (* s_1 *)
  enddo
enddo

Exemple 5.2: notre outil détecte trois dépendances : une dépendance de flot due à la lecture $a(i,j-1)$, de niveau 2, de vecteur de direction $(0,1)$ et d'approximation polyédrique le singleton $(0,1)$, et deux autres dépendances, une dépendance de flot et une anti-dépendance, dues toutes deux à la lecture $a(j,i)$, de niveau 1, de vecteur de direction $(1+,1-)$ et de représentation polyédrique le polyèdre composé d'un sommet $(1,-1)$ et d'un rayon $(1,-1)$.

Construction d'un graphe des dépendances de donnée. Nous avons vu précédemment comment était réalisé l'interfaçage entre Nestor et Petit, il nous reste maintenant à voir comment créer effectivement un graphe des dépendances. La bibliothèque de manipulation de graphe contient une classe `NstGraph` qui représente un graphe orienté :

```
class NstGraph {
  ...
  NstEdgeList edges
  NstVertexList vertices
  ...
};
```

c'est-à-dire une liste d'arcs (`edges`) et une liste de sommets (`vertices`). Un arc contient une représentation de la dépendance par un niveau, un vecteur ou un polyèdre. Un sommet contient un pointeur sur la variable ou l'instruction impliquée dans la dépendance. Deux classes héritent de cette classe `NstGraph`. La classe `NstRDGVar` qui représente un graphe des dépendances de donnée où chaque sommet est une variable impliquée dans une dépendance et la classe `NstRDGStat` où chaque sommet est une instruction. Nous allons examiner ici le constructeur de `NstRDGVar`. Celui-ci accepte en paramètre l'instruction qui doit être analysée et construit le graphe des dépendances de donnée grâce à Petit :

```
NstRDGVar(const NstStatement& s);
```

Ce constructeur fonctionne toujours même si Petit n'arrive pas à effectuer son analyse. La méthode `built()` de la classe permet de savoir si le graphe a bien été construit.

La classe `NstGraph` permet d'effectuer un certain nombre d'opérations comme le tri topologique, la coloration des sommets du graphe et le regroupement de ces sommets selon leur couleur, le calcul des composantes fortement connexes, etc... Elle comporte également un opérateur `<<` permettant d'écrire le graphe et un support, écrit par Guillaume Huard, pour l'affichage graphique de celui-ci grâce à l'outil de visualisation gratuit `xvcg` [53].

5.7 GTKNestor

Pour utiliser et tester une transformation écrite grâce à Nestor, le programmeur doit a priori écrire à chaque fois les lignes de code permettant de lire un programme en Fortran et de transformer celui-ci en structure interne de Nestor. De plus, il faut que le programmeur gère les différents paramètres de la bibliothèque d'analyse syntaxique et sémantique, comme le type de format de programme Fortran (libre ou fixe) ou la gestion des directives.

Si le programmeur veut appliquer sa transformation sur des portions de programme prédéterminées, il faut également qu'il prévoie une gestion des directives de Nestor. Une fois sa transformation appliquée, il doit gérer l'écriture de son programme transformé, dans le dialecte de son choix. Dans le cas de la lecture et de l'écriture du programme, il faut également gérer l'ouverture, la lecture et l'écriture de divers fichiers.

GTKNestor permet de s'affranchir de ces différents travaux. En effet, cette interface graphique permet de lire un programme en Fortran avec différents paramètres, d'afficher la structure interne du programme lu sous une forme hiérarchique et pratique à utiliser, et d'écrire la structure interne dans un fichier en utilisant un langage choisi par l'utilisateur. La structure interne peut être visualisée entièrement ou par morceaux, sous les différents langages. L'interface reconnaît automatiquement tous les modules disponibles placés dans un répertoire particulier et permet l'application d'un de ces modules sur n'importe quel élément de la structure interne ou sur toute la structure, de manière interactive : l'utilisateur sélectionne un morceau de structure et clique sur un bouton pour appliquer le module. La sélection d'un module à appliquer créé l'interface graphique permettant de paramétrer le module (voir §5.8, p. 123).

Cette interface est un outil puissant et pratique pour les programmeurs et pour les utilisateurs. Pour les programmeurs, c'est-à-dire les personnes qui implantent de nouvelles transformations, notre interface permet de se concentrer uniquement sur la transformation proprement dite et de visualiser et tester immédiatement son résultat. Pour les utilisateurs, c'est-à-dire les personnes qui veulent transformer leur code, les différents modules composent un outil qui est un paralléliseur semi-automatique de code avec une visualisation de toutes les étapes.

Images de l'interface graphique. Afin d'illustrer notre propos, les figures 5.6 et 5.7 présentent deux fenêtres de GTKNestor. La première de ces fenêtres (figure 5.6) est la fenêtre qui permet de lire et d'écrire un programme en Fortran et de choisir les différents paramètres en entrée et en sortie. Ici, nous avons sélectionné un programme appelé `adi_nestor.f` (présenté plus loin à la figure 5.8). Notre programme en sortie s'appelle `adi_openmp.f` et nous avons sélectionné du Fortran avec des directives OpenMP en sortie. Un appui sur le bouton `Parse file` permet de créer la structure interne dont la représentation graphique est présentée à la figure 5.7. Cette représentation permet d'exhiber la structure hiérarchique de la représentation interne. Sur chaque branche, un petit bouton $\boxed{+}$ permet de « déplier » un morceau de la structure. Sur cette figure, nous avons appliqué sur la boucle surlignée un module élémentaire présenté plus bas (§5.9.3, p. 130), qui permet de décider si une boucle est parallèle ou non. Une boucle parallèle est représentée par le mot clé `DO PARALLEL`. Toutes les instructions parallèles sont présentées dans une couleur différente. Une autre fenêtre qui n'est pas présentée ici contient la liste des modules disponibles avec des possibilités d'affichage de la documentation de chaque module.

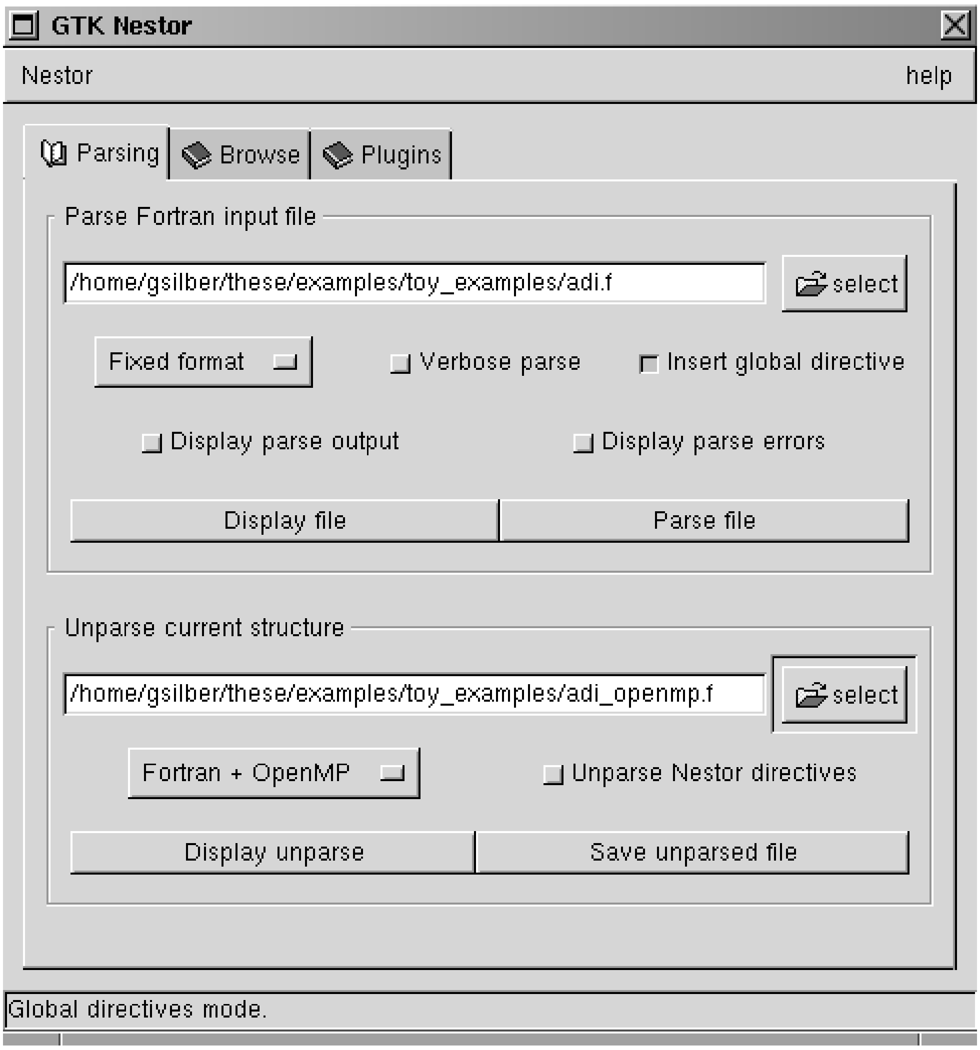


Figure 5.6: la fenêtre principale de GTKNestor permettant de lire et d'écrire un fichier dans différents dialectes.

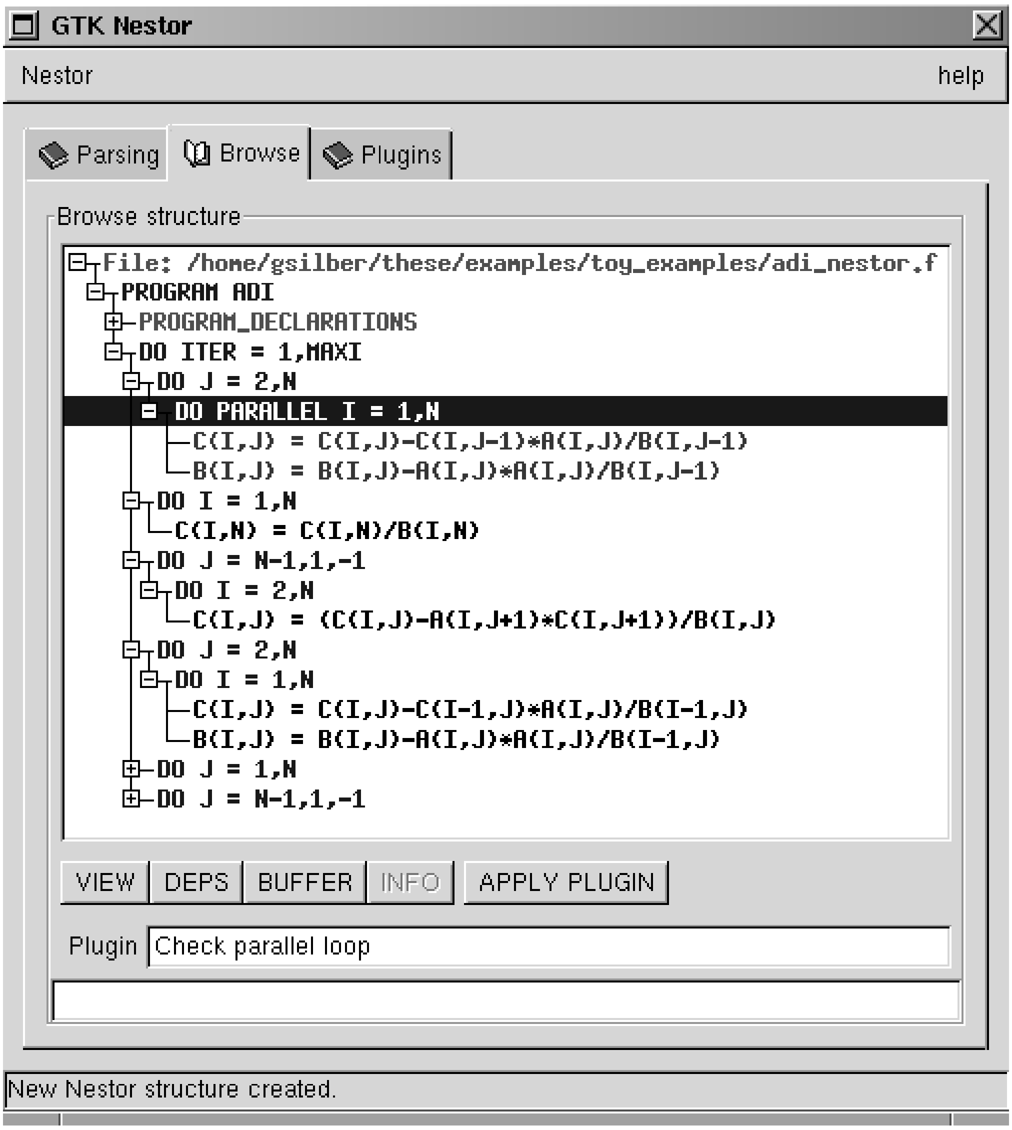


Figure 5.7: la fenêtre de GTKNestor permettant de consulter et de modifier la structure interne.

5.8 La bibliothèque de support pour les « plugins »

Afin de faciliter l'écriture de transformations de la structure interne, nous avons écrit une bibliothèque permettant la création de « plugins », c'est-à-dire de modules de code C++ indépendants qui prendront en entrée un pointeur `NstTree*` et un ensemble de paramètres et qui fourniront en sortie un pointeur sur la racine de la structure modifiée. Ces modules pourront notamment être utilisés par GTKNestor (§5.7, p. 120).

Un module utilisant cette bibliothèque devra comporter au moins deux fonctions. Une fonction `plugin_init()` qui contiendra le descriptif du module ainsi que ces paramètres et une fonction `plugin_run()` qui est le corps du module (la partie qui effectue la transformation).

La fonction `plugin_init()`. Voici une fonction `plugin_init()` type :

```
int plugin_init(struct arglist* desc)
{
  plug_set_name(desc, NAME);
  plug_set_description(desc, DESC);
  plug_set_summary(desc, SUMM);
  plug_set_copyright(desc, COPYRIGHT);
  ...
  return(0);
}
```

Le processus qui utilise le module va communiquer avec celui-ci par l'intermédiaire d'une structure `arglist`. La fonction `plugin_init()` va mettre des informations dans cette structure par l'intermédiaire de fonctions d'ajout qui ont la forme générale `plug_set_...()`, comme par exemple `plug_set_name()` qui va donner au processus appelant le nom du module. De la même manière, la fonction `plugin_init()` doit donner une courte description du module (`plug_set_description()`), un résumé plus conséquent (`plug_set_summary()`) ainsi que des informations sur l'auteur du module (`plug_set_copyright()`). Les paramètres `NAME`, `DESC`, `SUMM` et `COPYRIGHT` sont des chaînes de caractères. Finalement, la fonction `plugin_init()` doit renvoyer 0 pour signifier au processus appelant que tout a fonctionné correctement.

La fonction `plugin_init()` précédente ne spécifie pas de paramètres pour le module. Nous pourrions ajouter un ou des paramètres au module en utilisant la fonction `plug_set_arguments()` et en ajoutant tous les paramètres avec la fonction `plug_add_argument()` qui a le prototype suivant :

```
void plug_add_argument(struct arglist* l, char* n, int t, void* d);
```

où `n` est le nom du paramètre, `t` est son type et `d` sa valeur par défaut. Par exemple, pour ajouter un paramètre appelé `val` de type entier et ayant la valeur par défaut 1, nous utiliserions la ligne de code suivante :

```
plug_add_argument(desc, "val", P_INT, (void*)1);
```

Le type `t` peut prendre les valeurs `P_INT`, `P_BOOL`, `P_STRING`, `P_BUFFER`, `P_FILE` et `P_SET` qui désignent respectivement un entier, un booléen, une chaîne de caractères, un objet `NstTree`, un fichier et une liste à choix multiple. Dans ce dernier cas, le programmeur peut utiliser la fonction `plug_add_argument_element()` pour rajouter un choix à cette liste. Par exemple, si nous voulions créer une liste appelée `"Liste de choix"` avec les deux choix possibles `"Avant"` et `"Après"`, nous écririons les lignes suivantes :

```
plug_add_argument(desc, "Liste de choix", P_SET, NULL);
plug_add_argument_element(desc, "Liste de choix", "Avant");
plug_add_argument_element(desc, "Liste de choix", "Après");
```

Les types des arguments vont permettre au processus qui désire utiliser ce module de définir une interface, comme par exemple l'interface graphique générée par GTKNestor (voir les exemples donnés §5.9.4, p. 134 et §5.9.5, p. 136).

La fonction `plugin_run()`. Cette fonction est appelée pour exécuter le module. Voici une fonction `plugin_run()` type, correspondant aux paramètres que nous avons donnés en exemple dans la fonction `plugin_init()` précédente :

```
int plugin_run(struct arglist * args)
{
  NstTree* tree = plug_get_tree(args);
  char* report_file = plug_get_report(args);
  int val = (int)plug_get_argument(args, "val");
  int choix = (int)plug_get_argument(args, "Liste de choix");
  ...
  // Effectuer ici une operation sur tree avec les
  //  paramètres val et choix, en écrivant éventuellement
  //  des commentaires dans le fichier report_file.
  ...
  plug_set_tree(args, tree->in());
  plug_set_report(args, report_file);
  return 0;
}
```

La méthode `plug_get_tree()` permet de récupérer un objet de la structure interne sur lequel la transformation doit être effectuée. Ici nous récupérons un objet `tree`. La méthode `plug_get_report()` permet de récupérer le nom d'un fichier dans lequel la transformation va pouvoir écrire des commentaires sur le travail effectué.

Pour récupérer la valeur d'un argument parmi ceux spécifiés dans la définition du module (`plugin_init()`), il faut utiliser la fonction `plug_get_argument()` qui prend en paramètre le nom du paramètre et renvoie sa valeur courante. Par exemple, nous récupérons dans une variable `val` le paramètre qui a comme nom `"val"` et dans une variable `choix` le choix courant parmi les choix du paramètre `"Liste de choix"` (ce choix est un entier correspondant au numéro du choix, ceux-ci étant numérotés à partir de 1 dans l'ordre inverse de leur déclaration). Dans le cas de GTKNestor, la valeur de ces paramètres est celle contenue dans l'interface graphique générée automatiquement.

Une fois que les arguments ont été récupérés, une transformation ou une analyse quelconque peut être écrite. Une fois cette phase terminée, le module doit replacer dans la liste des arguments un pointeur sur l'endroit à partir duquel la structure interne doit être réaffichée. Ainsi, les deux lignes :

```
  plug_set_tree(args, tree->in());
  plug_set_report(args, report_file);
```

précisent que la structure interne doit être réaffichée à partir du père de l'objet passé en paramètre au module et que le fichier de rapport a effectivement été utilisé pour écrire quelque chose.

Pourquoi des modules ? L'intérêt des modules est de pouvoir rajouter des fonctionalités à Nestor sans avoir à recompiler la bibliothèque à chaque fois. Il suffit de créer le module, de le compiler grâce aux facilités offertes par le système et de le tester interactivement. D'autre part, un module permet à un programmeur de s'affranchir de la phase de création de la structure interne et de la phase d'écriture de celle-ci. De plus, cette approche permet de créer facilement des outils de transformation interactive du code comme par exemple GTKNestor, permettant un affichage de la structure interne et l'application d'un module quelconque sur n'importe quel élément de celle-ci. Un module peut également être intégré à un exécutable dédié à une seule transformation, pour par exemple créer un filtre comme celui présenté §5.9.3, p. 130.

5.9 Utilisation de Nestor

Dans cette partie, nous allons présenter divers aspects de l'utilisation de Nestor. Dans un premier temps, nous allons présenter trois exécutables effectuant des opérations sur un code source spécifié en entrée et dont certaines parties sont marquées par des directives : un filtre de base illustrant la construction d'une transformation (§5.9.1, p. 125), un analyseur de dépendances (§5.9.2, p. 127) et un détecteur de parallélisme (§5.9.3, p. 130). Dans un second temps, nous présenterons divers « plugins » intégrés à la distribution de base de Nestor et qui peuvent être utilisés dynamiquement sur n'importe quelle portion de code (par exemple avec l'interface graphique `gtknestor`). Le premier module permet d'ajouter une distribution de données (§5.9.4, p. 134). Le deuxième est un module de parallélisation (§5.9.5, p. 136) utilisant l'algorithme présenté dans le chapitre précédent (§4.7, p. 74). Tous ces exemples sont élémentaires et illustratifs, il est bien sûr possible de faire des choses bien plus puissantes avec notre bibliothèque.

5.9.1 Un filtre basique

La tâche la plus simple à accomplir dans un environnement de transformation source à source est de réaliser un compilateur de base qui réalise l'identité, c'est-à-dire un programme qui prenne en entrée un code source en Fortran et qui génère le même code source en Fortran, en étant passé par une représentation interne de ce code source. Nous ne devons avoir aucun phénomène de régression entre l'entrée et la sortie de ce programme, c'est-à-dire que l'on doit pouvoir utiliser la sortie comme entrée du programme un nombre indéfini de fois et toujours pouvoir compiler le code source en code machine avec le même résultat.

Avec notre bibliothèque, un tel programme est très facile à construire. Le code ci-dessous est le fichier `nstcat.cc` qui est fourni avec la distribution de Nestor (`src/exe/nstcat.cc`, ce code n'utilise que la bibliothèque `nstbase` et la bibliothèque `adaptor`).

```
# include <nestor.H>

int main(int argc, char** argv)
{
  char *fname = (argc < 2) ? NULL : argv[1];
```

```
  NstComputationUnit prog(fname);
  cout << prog << endl;
  return 0;
}
```

La première ligne de ce programme est l'initialisation d'une variable `fname` qui va contenir le nom du fichier texte à lire. Si l'utilisateur ne passe pas de paramètre, cette variable est affectée à `NULL`. La deuxième ligne est la création d'une structure interne de code source depuis le texte du fichier de `fname`. Si `fname` est à `NULL`, le constructeur de `NstComputationUnit` va lire du texte sur l'entrée standard jusqu'au caractère `EOF`. Ce constructeur peut échouer et stopper l'exécution du programme si le code source en entrée présente une erreur syntaxique ou sémantique. Après l'exécution de ce constructeur, `prog` représente la racine d'une structure interne de code source. L'avant dernière ligne provoque l'écriture sur la sortie standard de la structure interne sous la forme d'un code source en Fortran.

Après compilation, ce programme nous donne un exécutable appelé `nstcat` qui peut être simplement appelé depuis la ligne de commande de la façon suivante (les caractères `info%` correspondent à l'invite du shell) :

```
info% nstcat fichier.f
```

ou alors sans argument, ce qui correspond à l'attente de caractères sur l'entrée standard :

```
info% nstcat
```

Dans ce dernier cas, si nous entrons au clavier la suite de lignes suivante qui correspond à un tout petit programme Fortran :

```
integer i
i = 3
end
```

nous obtenons la sortie suivante :

```
      program NO_NAME
      integer*4 I
      I = 3
      end
```

Nous pouvons également enchaîner les appels à `nstcat` avec des tubes :

```
info% nstcat | nstcat | nstcat | nstcat | nstcat | nstcat
```

ce qui donnera exactement la même sortie avec le même programme en entrée que précédemment.

Cette construction est un filtre de base. Nous pouvons imaginer des filtres qui peuvent réaliser un certain nombre de transformations sur la structure interne générée avec le constructeur de `NstComputationUnit`. Il suffit d'insérer cette transformation entre le constructeur et l'écriture de la structure interne (réalisée ici avec l'opérateur surchargé `cout`). Nous pouvons donc avoir toute une série de filtres dédiés chacun à une transformation particulière et les enchaîner sur la ligne de commande. Par exemple, nous pouvons enchaîner un premier filtre qui réalise une évaluation partielle du code avec un deuxième filtre qui effectue de la mise en ligne de procédure (« procedure inlining »), puis un troisième qui supprime les portions de code mort, avant de finir par un filtre qui parallélise le code, par exemple avec l'algorithme d'Allen, Callahan et Kennedy. Cette approche permet une très grande souplesse d'utilisation et une simplification de la phase d'implantation d'une transformation. Tous ces filtres pourraient à terme former un ensemble de briques de base qui combinées permettraient d'obtenir un système interactif de transformation de code. À notre connaissance Nestor est le seul système à offrir une telle simplicité.

5.9.2 Un analyseur de dépendance

Notre bibliothèque a rajouté à la syntaxe de base de Fortran un certain nombre de directives permettant de spécifier certaines portions de code. Rappelons que ces directives sont `SINGLE` qui permet de marquer une boucle `do` et le couple de directives `BEGIN` et `END` qui permettent de spécifier une région de code.

Le programme que nous allons présenter dans ce chapitre fait partie de la distribution de Nestor sous le nom de `nstdep.cc` et permet de calculer les dépendances de données (et de les imprimer dans un fichier) des portions de code marquées. Voici le code (quelque peu raccourci, les ... indiquent des portions de programme manquantes) de `nstdep.cc` (`src/exe/nstdep.cc`, ce code utilise les bibliothèques `adaptor`, `nstbase` et `nstgraph`) :

```
# include <nestor.H>

NstRDGVar* compute_dependences(NstStatement* st)
{
  // définition de dep_file
  ...
  NstRDGVar* dg = new NstRDGVar(*st);
  if (dg->built())
    dep_file << *dg;
  else
    cout << "Dependence analysis failed. ";
  return dg;
}

int treat_only_marked(NstTree* t, void* ignored)
{
  if (NstStatement* s = dynamic_cast<NstStatement*>(t))
    {
      int nst_flag = 0;
      if (NstStatementDo* sd = dynamic_cast<NstStatementDo*>(s))
        {
          nst_flag = sd->info()->nestor_flag;
        }
      if (s->type() == NST_STATEMENT_NESTOR || nst_flag)
        {
          NstRDGVar* dg = compute_dependences(s);
          delete dg;
          return 1;
        }
    }
  return 0;
```

```
}

int main(int ac, char **av)
{
  ...
  char *fname = (argc < 2) ? NULL : argv[1];
  NstComputationUnit prog(fname, options);
  program->traversal(treat_only_marked, report);
  return 0;
}
```

La première fonction de ce programme, `compute_dependences()`, prend comme paramètre une instruction (`NstStatement`) sur laquelle l'analyse de dépendances doit être menée. Cette instruction doit être une boucle ou une région (`NstStatementNestor`). Le constructeur `NstRDGVar` construit un code dans le langage de Petit et lance celui-ci pour analyser les dépendances. La liste des dépendances se trouve dans l'objet créé. Les dépendances sont écrites dans un fichier avec l'instruction

```
dep_file << *dg;
```

La deuxième fonction, `treat_only_marked()`, est destinée à être appelée sur toutes les composantes de la structure interne par la méthode `traversal()`. Elle teste si cette composante est une boucle marquée ou une région et appelle `compute_dependences()` sur celle-ci.

La fonction `main()` construit une structure interne depuis le code source passé en entrée du programme (dans un fichier ou directement sur l'entrée standard) et effectue un parcours récursif sur la structure interne avec l'instruction

```
program->traversal(treat_only_marked, report);
```

Après compilation, nous obtenons un exécutable `nstdep` qui peut être utilisé de la façon suivante :

```
info% nstdep fichier.f
```

et qui va écrire les dépendances de toutes les portions de code marquées dans `fichier.f` dans des fichiers dont le nom est donné sur la sortie standard lors de l'exécution. Ce programme n'est pas un filtre puisqu'il ne fournit rien sur la sortie standard. Par exemple, avec le programme de la figure 5.8, voici les messages générés par un appel à `nstdep` :

```
info% nstdep adi.f
! [ Unit: ADI, line: 10. 3 dependences, file: /tmp/nst_2.10270.ADI.dep ]
! [ Unit: ADI, line: 18. No dependences. ]
! [ Unit: ADI, line: 23. 1 dependences, file: /tmp/nst_7.10270.ADI.dep ]
! [ Unit: ADI, line: 30. 3 dependences, file: /tmp/nst_10.10270.ADI.dep ]
! [ Unit: ADI, line: 38. No dependences. ]
! [ Unit: ADI, line: 43. 1 dependences, file: /tmp/nst_15.10270.ADI.dep ]
```

ainsi que le contenu du premier fichier `/tmp/nst_2.10270.ADI.dep` :

```
info% more /tmp/nst_2.10270.ADI.dep
from [2] to [4] level: 1 depth: 2 dir: (1,0) type: flow
Integral Vertices: (1,0)
from [7] to [6] level: 1 depth: 2 dir: (1,0) type: flow
```

```
      program adi
      integer i,j,iter
      parameter (N = 100, MAXI = 1000)
      integer N, MAXI
      real c(N,N), a(N,N), b(N,N)
      DO iter = 1, MAXI
!NESTOR$ SINGLE
         DO j = 2, N
            DO i = 1, N
               c(i,j) = c(i,j)-c(i,j-1)*a(i,j)/b(i,j-1)
               b(i,j) = b(i,j)-a(i,j)*a(i,j)/b(i,j-1)
            ENDDO
         ENDDO
!NESTOR$ SINGLE
         DO i = 1,N
            c(i,N) = c(i,N)/b(i,N)
         ENDDO
!NESTOR$ SINGLE
         DO j = N-1,1,-1
            DO i = 2,N
               c(i,j) = (c(i,j)-a(i,j+1)*c(i,j+1))/b(i,j)
            ENDDO
         ENDDO
!NESTOR$ SINGLE
         DO j = 2, N
            DO i = 1, N
               c(i,j) = c(i,j)-c(i-1,j)*a(i,j)/b(i-1,j)
               b(i,j) = b(i,j)-a(i,j)*a(i,j)/b(i-1,j)
            ENDDO
         ENDDO
!NESTOR$ SINGLE
         DO j = 1,N
            c(N,j) = c(N,j)/b(N,j)
         ENDDO
!NESTOR$ SINGLE
         DO j = N-1,1,-1
            DO i = 2,N
               c(i,j) = (c(i,j)-a(i+1,j)*c(i+1,j))/b(i,j)
            ENDDO
         ENDDO
      ENDDO
      end
```

Figure 5.8: le code source du programme `adi.f`.

```
Integral Vertices: (1,0)
from [7] to [11] level: 1 depth: 2 dir: (1,0) type: flow
Integral Vertices: (1,0)

cNESTOR$ SINGLE
        do J = 2,N
           do I = 1,N
              C[2](I,J) = C[3](I,J)-C[4](I,J-1)*A[5](I,J)/B[6](I,J-1)
              B[7](I,J) = B[8](I,J)-A[9](I,J)*A[10](I,J)/B[11](I,J-1)
           end do
        end do
```

où les nombre entre crochets dans les dépendances correspondent à des accès à des variables dans le code.

5.9.3 Un détecteur de parallélisme

Nestor possède un mécanisme permettant de déterminer quelles sont les boucles parallèles dans une section de code. De telles boucles seront marquées comme étant parallèles. La classe permettant de faire ceci est `CheckParallelLoop`.

La classe `CheckParallelLoop`. Cette classe hérite de `NstTreeTraversal` (§5.3.9, p. 111) puisque la recherche de boucles parallèles va nécessiter un parcours de la structure. Le constructeur de `CheckParallelLoop` lance le parcours, avec `nb_par` et `nb_seq` qui sont des variables recevant respectivement le nombre de boucles parallèles et le nombre de boucles séquentielles trouvées, et `cv` le graphe des dépendances courant :

```
CheckParallelLoop::CheckParallelLoop (NstTree& t, ofstream* rep = NULL)
{
  cv = NULL;
  nb_par = nb_seq = 0;
  traversal(&t);
}
```

La méthode `traversal()` est celle de la classe `NstTreeTraversal`. Nous avons simplement redéfini la méthode `post_action()` et `pre_action()`. Voici tout d'abord la méthode `pre_action()` (exécutée pour chaque élément de la structure) :

```
int
CheckParallelLoop::pre_action(NstTree* t)
{
  if (NstStatementDo* stdo = dynamic_cast<NstStatementDo*>(t))
    {
      if (!cv)
        {
          // creator contient l'élément qui a créé le graphe des dépendances
          creator = t;
          cv = new NstRDGVar(*stdo);
          c_level = 1;
          if (!cv->built())
            {
              ... // erreur
            }
        }
      else
        ++c_level;

      if (cv) {
        // Teste si la boucle courant ne porte pas de dépendance pour le niveau
        //   c_level courant.
        minmax.var_min = ULONG_MAX;
        minmax.var_max = 0UL;
        stdo->traversal(get_min_max, (void*)&minmax);
        int check = check_level();
        stdo->info()->is_independent = !check;
      }
      if (stdo->info()->is_independent)
```

```
        ++nb_par;
      else
        ++nb_seq;
    }
  return 0;
}
```

Ce code teste tout d'abord si l'élément courant est une boucle. Si c'est le cas, il crée un graphe des dépendances, sauf si il y en a déjà un qui a été créé (par la boucle englobante éventuelle). Une fois ce graphe créé (stocké dans l'attribut `cv` de la classe), l'attribut `c_level` contient le niveau de profondeur courant de la boucle que l'on examine.

La suite ce cette méthode consiste à déterminer si le graphe des dépendances contient une dépendance portée par la boucle de niveau `c_level` que l'on examine. La fonction `get_min_max()` permet de déterminer l'indice du plus petit et du plus grand élément de tableau présent dans la boucle (cet indice est l'ordre d'apparition du tableau dans le code et est celui utilisé par notre version modifiée de Petit (§5.6, p. 118)). Nous avons besoin de cet élément car le nid de boucle peut ne pas être parfait et le graphe peut avoir été calculé pour une boucle englobante. Voici la fonction `get_min_max()`, où l'indice de l'accès au tableau est stocké dans `user_info.tag` :

```
int get_min_max(NstTree* t, void* mm)
{
  if (t->type() == NST_VARIABLE_USED)
    {
      if (t->user_info.tag > 0)
        {
          if (t->user_info.tag < ((struct minmax*)mm)->var_min)
            ((struct minmax*)mm)->var_min = t->user_info.tag;
          if (t->user_info.tag > ((struct minmax*)mm)->var_max)
            ((struct minmax*)mm)->var_max = t->user_info.tag;
        }
      return 1;
    }
  return 0;
}
```

Une fois l'intervalle des indices d'accès au tableau déterminé, il suffit de tester si il n'y a pas de dépendance portée par la boucle. Ce test est décrit par la méthode `check_level()` suivante, qui parcourt la liste des dépendances du graphe `cv` et teste pour chaque dépendance si le puits et la source sont dans l'intervalle déterminé précédemment et si la dépendance est de niveau `c_level` :

```
int
CheckParallelLoop::check_level()
{
  NstEdgeList::iterator el_it;
  NstDependence* dep;
  int is_dependence = 0;

  for (el_it = cv->edges.begin(); el_it != cv->edges.end(); el_it++)
    {
      dep = (NstDependence*) (*el_it);
```

```
      if (dep->from_tag() >= minmax.var_min && dep->from_tag() <= minmax.var_max &&
          dep->to_tag() >= minmax.var_min && dep->to_tag() <= minmax.var_max &&
          dep->level == c_level)
        {
          is_dependence = 1;
        }
    }
  return is_dependence;
}
```

Finalement, la méthode `post_action()` détruit le graphe des dépendances créé si l'objet qui appelle `post_action()` est celui qui a créé ce graphe (ceci est déterminé grâce à l'attribut `creator`) et décrémente le niveau de 1 :

```
void
CheckParallelLoop::post_action(NstTree* t)
{
  if (t->type() == NST_STATEMENT_DO) {
    if (creator == t) {
      delete cv;
      cv = NULL;
      creator = NULL;
    }
    else
      --c_level;
  }
}
```

Nous avons donc présenté en détail cette classe `CheckParallelLoop` qui est intégrée à Nestor et que nous allons pouvoir utiliser dans le paragraphe suivant.

Un paralléliseur utilisant `ChechParallelLoop`**.** Nous allons présenter un exécutable appelé `nstpar` permettant de détecter le parallélisme élémentaire dans un code Fortran. Voici un extrait du code source de `nstpar.cc` (`src/exe/nstpar.cc`) :

```
# include <nestor.H>
int treat_only_marked(NstTree* t, void* ignored)
{
  NstStatementNestor* sn = dynamic_cast<NstStatementNestor*>(t);
  NstStatementDo* sd = dynamic_cast<NstStatementDo*>(t);
  if (sn || (sd && sd->info()->nestor_flag == 1))
    {
      CheckParallelLoop checker(*t, (ofstream*)ignored);
      return 1;
    }
  return 0;
}
int main(int ac, char **av)
{
  ...
  if (strcmp(umode, "hpf") == 0) nst_fortran_type(HPF);
```

```
  else if (strcmp(umode, "sunmp") == 0) nst_fortran_type(SunMP);
  else if (strcmp(umode, "craymp") == 0) nst_fortran_type(CrayMP);
  else if (strcmp(umode, "openmp") == 0) nst_fortran_type(OpenMP);
  ...
  char *fname = (argc < 2) ? NULL : argv[1];
  NstComputationUnit prog(fname, options);
  prog.traversal(treat_only_marked, report);
  cout << prog << endl;
  return 0;
}
```

La fonction `treat_only_marked()` est quasiment la même que dans l'exemple précédent. La fonction `main()` est presque identique hormis le choix du langage de sortie (HPF, OpenMP, CrayMP ou SunMP) et l'écriture de la structure interne sur la sortie standard.

Ce programme est un filtre qui prend en entrée un code source et génère un code avec des directives de parallélisation devant les boucles parallèles. Les seules boucles examinées sont celles qui sont choisies par l'utilisateur. Ce programme peut également servir de convertisseur entre différents types de directives (HPF vers OpenMP, OpenMP vers SunMP, etc...). Par exemple avec le code précédent (`adi.f`, figure 5.8) en entrée, nous obtenons après la commande

```
info% nstpar -u openmp adi.f
```

le code OpenMP suivant, où les boucles séquentielles ont toujours la directive `SINGLE` et les boucles parallèles la directive OpenMP `PARALLEL DO` :

```
      program adi
      integer i,j,iter
      parameter (N = 100, MAXI = 1000)
      integer N, MAXI
      real c(N,N), a(N,N), b(N,N)
cNESTOR$ SINGLE
         do J = 2,N
c$OMP PARALLEL DO
            do I = 1,N
               C(I,J) = C(I,J)-C(I,J-1)*A(I,J)/B(I,J-1)
               B(I,J) = B(I,J)-A(I,J)*A(I,J)/B(I,J-1)
            end do
         end do
c$OMP PARALLEL DO
         do I = 1,N
            C(I,N) = C(I,N)/B(I,N)
         end do
cNESTOR$ SINGLE
         do J = N-1,1,-1
c$OMP PARALLEL DO
            do I = 2,N
               C(I,J) = (C(I,J)-A(I,J+1)*C(I,J+1))/B(I,J)
            end do
         end do
c$OMP PARALLEL DO
```

```
        do J = 2,N
           do I = 1,N
              C(I,J) = C(I,J)-C(I-1,J)*A(I,J)/B(I-1,J)
              B(I,J) = B(I,J)-A(I,J)*A(I,J)/B(I-1,J)
           end do
        end do
c$OMP PARALLEL DO
        do J = 1,N
           C(N,J) = C(N,J)/B(N,J)
        end do
c$OMP PARALLEL DO
        do J = N-1,1,-1
           do I = 2,N
              C(I,J) = (C(I,J)-A(I+1,J)*C(I+1,J))/B(I,J)
           end do
        end do
     end do
     end
```

Nestor permet de construire en quelques lignes un détecteur de parallélisme élémentaire dans n'importe quel code Fortran.

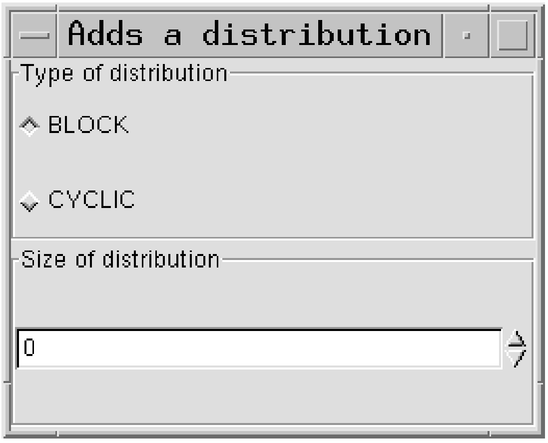
5.9.4 Ajout ou modification de distributions

Nous allons présenter ici un module permettant d'ajouter ou de modifier des distributions HPF pour des tableaux du code. L'entrée de notre module sera un élément de structure interne qui sera une déclaration de tableau. La distribution HPF pour ce tableau sera ajoutée à la fin de la liste de déclarations contenant notre déclaration en entrée. Voici la partie documentaire et de définition de notre module (le code complet se trouve sous `src/plugins/insert_distri/ins_dist.cc`) :

```
#include <nst_plug.H>

#define NAME "Adds a distribution"
#define DESC "\
Inserts an HPF distribution for an array that is already\n\
declared.\n"
#define COPYRIGHT "no copyright"
#define SUMM "Inserts or modify an HPF distribution for an array"

int plugin_init(struct arglist* desc)
{
  plug_set_name(desc, NAME);
  plug_set_description(desc, DESC);
  plug_set_summary(desc, SUMM);
  plug_set_copyright(desc, COPYRIGHT);

  plug_set_arguments(desc);
  plug_add_argument(desc, "Type of distribution", P_SET, NULL);
  plug_add_argument_element(desc, "Type of distribution", "BLOCK");
```

```
  plug_add_argument_element(desc, "Type of distribution", "CYCLIC");
  plug_add_argument(desc, "Size of distribution", P_INT, (void*)0);
  return(0);
}
```

Notre module a deux paramètres, l'un est le type de distribution (une liste de choix comprenant `BLOCK` et `CYCLIC`), l'autre est la taille éventuelle d'un bloc de la distribution (à 0 par défaut, pour signifier que la taille doit être choisie par le compilateur). Voici la fenêtre de paramétrage générée par `gtknestor` :



Le reste du module consiste simplement à récupérer les différents paramètres et à insérer la distribution au bon endroit ou de la modifier si celle-ci existe déjà.

```
int plugin_run(struct arglist * args)
{
  NstTree* tree = plug_get_tree(args);
  NstDeclarationList* father = dynamic_cast<NstDeclarationList*>(tree->in());
  NstDeclarationVariable* dv = dynamic_cast<NstDeclarationVariable*>(tree);
  ...
  int dtype = (int)plug_get_argument(args, "Type of distribution");
  int dsize = (int)plug_get_argument(args, "Size of distribution");

  if (NstDeclarationDistribute* old_dist
        = search_existing_distribution(father, dv))
    modify_distribution(old_dist, dv, dtype, dsize);
  else
    insert_distribution(dv, dtype, dsize);

  plug_set_tree(args, (NstTree*)father);
  return 0;
}
```

Nous ne donnerons pas ici le code source des trois autres fonctions de ce module mais uniquement le code source de la fonction `insert_distribution()` :

```
void insert_distribution(NstDeclarationVariable* dv, int dtype, int dsize)
{
  if (NstTypeArray* ta = dynamic_cast<NstTypeArray*>(dv->variable_type()))
    {
```

```
      int size = ta->shapes.size();
      NstDistributionNode def_dist;
      NstDistformat* df = create_dist(dtype, dsize);
      while(size--)
        def_dist.mapping.push_back(*df);
      NstProcessorDefault proc_def;
      NstDeclarationDistribute dist(*dv->definition(), def_dist, proc_def);
      NstDeclarationList *dl = dynamic_cast<NstDeclarationList*>(dv->in());
      ...
      dl->push_back(dist);
    }
}
```

Le code de cette fonction est assez ardu à comprendre ainsi nous en donnons le principe général. Après avoir vérifié si la variable de la déclaration était bien de type tableau

```
dynamic_cast<NstTypeArray*>(dv->variable_type())
```

cette fonction crée la déclaration de la distribution (`NstDeclarationDistribute`) et insère celle-ci à la fin de la liste des déclarations

```
dl->push_back(dist)
```

5.9.5 Algorithme d'Allen et Kennedy

Le dernier module que nous présentons est l'implantation de l'algorithme que nous avons présenté dans le chapitre précédent (§4.7, p. 74). Rappelons que cet algorithme permet de détecter du parallélisme dans les nids de boucles grâce à la distribution de boucles en présence de dépendances de contrôle. Plutôt que de présenter le code du module, nous avons uniquement présenté à la figure 5.9 la fenêtre de paramétrage crée par `gtknestor` que nous allons utiliser comme support de notre explication.

Le premier paramètre (« `Tests and bounds in temps` ») est une liste à choix multiple. Les différents choix possibles sont décrits ci-dessous.

Always introduit un temporaire pour toutes les expressions de toutes les structures de contrôle.

Sometimes introduit un temporaire uniquement pour les expressions contenues dans une structure de contrôle qui sont source d'une anti-dépendance.

Minimal introduit le nombre minimal de temporaires nécessaire à l'obtention de la quantité minimale de temporaires (recherche exhaustive présentée §4.8.2, p. 79).

User supplied l'utilisateur peut placer une directive de la forme

```
!NESTOR$ TP(n)=t
```

devant une structure de contrôle pour spécifier que l'expression `n` de la structure de contrôle peut être contenue dans un temporaire de dimension maximale `t`. Par exemple :

```
!NESTOR$ TP(1)=-1,TP(2)=1
do i = a(1), b(1)
  ...
```

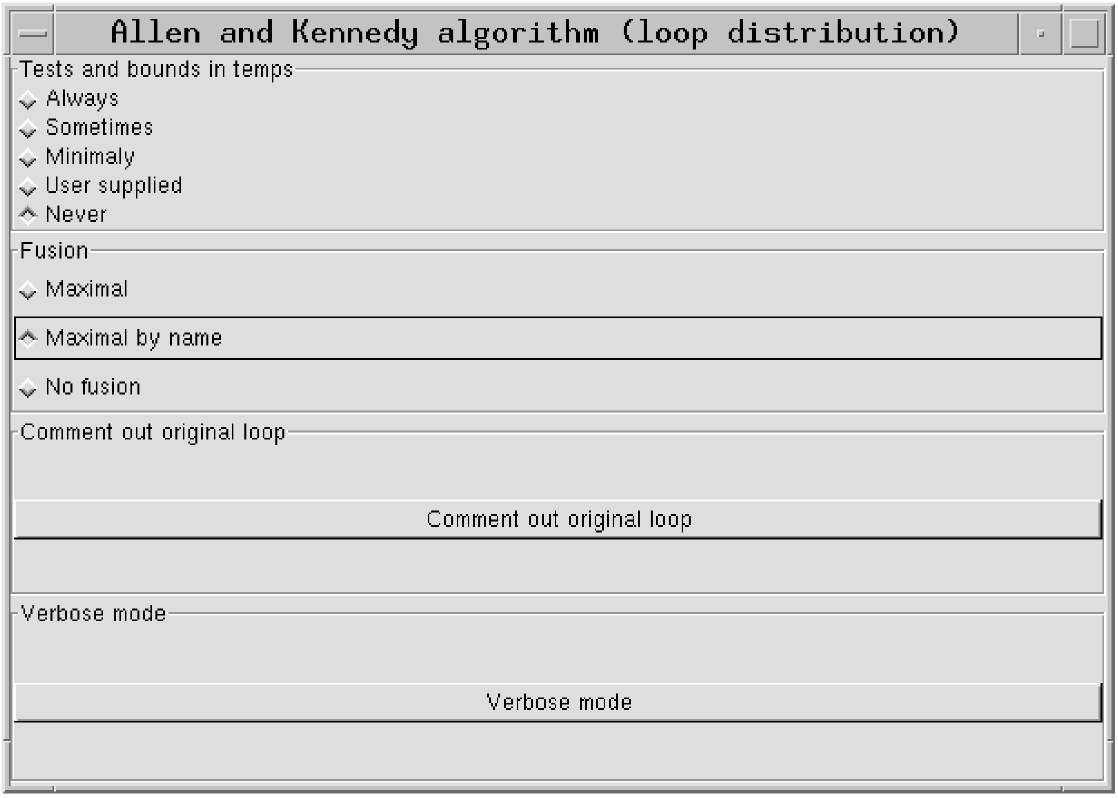


Figure 5.9: fenêtre de paramétrage de notre algorithme présenté §4.7, p. 74.

n'autorise pas de temporaire pour `a(1)` alors qu'un temporaire de dimension 1 est autorisé pour `b(1)`. Cette option permet à l'utilisateur de contrôler la parallélisation de son code. Une autre forme de cette même directive

```
!NESTOR$ TP(n)#t
```

indique que l'expression doit absolument être stockée dans un temporaire de dimension `t`.

Never n'autorise pas l'introduction de temporaires pour paralléliser le code.

La deuxième liste de choix décrit le type de fusion qui doit être effectuée dans l'algorithme après la phase de recherche des composantes fortement connexes.

Maximal essaie de fusionner au maximum les composantes fortement connexes tout en préservant le parallélisme.

Maximal by name essaye de fusionner au maximum les composantes qui ont des instructions impliquant le même tableau, tout en préservant le parallélisme. Ce type de fusion tente d'améliorer les effets de cache [46].

No fusion ne refusionne pas les composantes fortement connexes.

Les deux derniers paramètres permettent de choisir si le code original avant transformation doit être conservé dans le nouveau code sous forme de commentaire et si après exécution une fenêtre de texte doit expliciter toutes les étapes de l'éventuelle transformation du code.

Ce module est un exemple type de parallélisation semi-automatique : l'utilisateur peut agir sur de nombreux paramètres pour diriger la parallélisation de son code. Il peut également choisir de n'intervenir sur rien et laisser faire l'algorithme en espérant que celui-ci trouve la meilleure solution.

La figure 5.10 donne un exemple de code parallélisé à l'aide de ce module. L'anti-dépendance entre la borne de boucle et la première instruction du corps de cette boucle a été cassée par l'introduction d'un temporaire `nst_0` de dimension 1, ce temporaire étant privatisé pour la boucle la plus externe (voir §4.7.2, p. 74) avec la directive `NEW(nst_0)`. La taille de ce temporaire doit être précisée par l'utilisateur ou bien pourrait être déterminée par un autre module (qui n'est pas écrit).

```
do i = 1,n
  do j = 2,n
    do k = 2, a(i,j+1,3)
      a(i,j,k) = b(i,j,k)+a(i,j-1,k)
      c(i,j,k) = a(i,j-1,k-1)+d(i,j,k)
    enddo
  enddo
enddo
```

Code original.

```
!HPF$ INDEPENDENT, NEW(nst_0)
  do I = 1,N
!HPF$ INDEPENDENT
    do J = 2,N
      nst_0(J) = A(I,J+1,3)
    end do
    do J = 2,N
!HPF$ INDEPENDENT
      do K = 2,nst_0(J)
        A(I,J,K) = B(I,J,K)+A(I,J-1,K)
      end do
    end do
!HPF$ INDEPENDENT
    do J = 2,N
!HPF$ INDEPENDENT
      do K = 2,nst_0(J)
        C(I,J,K) = A(I,J-1,K-1)+D(I,J,K)
      end do
    end do
  end do
```

Code après parallélisation.

Figure 5.10: un code séquentiel parallélisé avec notre module de parallélisation selon l'algorithme présenté §4.7, p. 74 avec introduction de temporaires.

La copie d'écran présentée à la figure 5.11 est le résultat du module dans la fenêtre d'affichage et de modification du code de `gtknestor` après l'application du module.

5.10 Perspectives futures

Les perspectives futures pour notre outil sont nombreuses. Ne serait-ce que dans le domaine de la parallélisation de code, il reste de nombreux algorithmes de transformation de code à implanter et à valider. Notre outil est très adapté à cette tâche puisqu'il permet de lire n'importe quel code Fortran et de générer un code en HPF ou OpenMP pour tester le nouveau code. Il y a également de nombreux domaines ouverts dans la compilation de codes séquentiels pour lesquels notre outil serait également bénéfique. En dehors du parallélisme, Nestor pourrait être utilisé pour d'autres tâches, comme par exemple pour implanter des analyses pour prouver formellement qu'un code est correct. Nestor peut également servir de structure interne pour un environnement graphique de conception de code. Du côté de l'enseignement de l'informatique, notre outil peut être une bonne base de développement

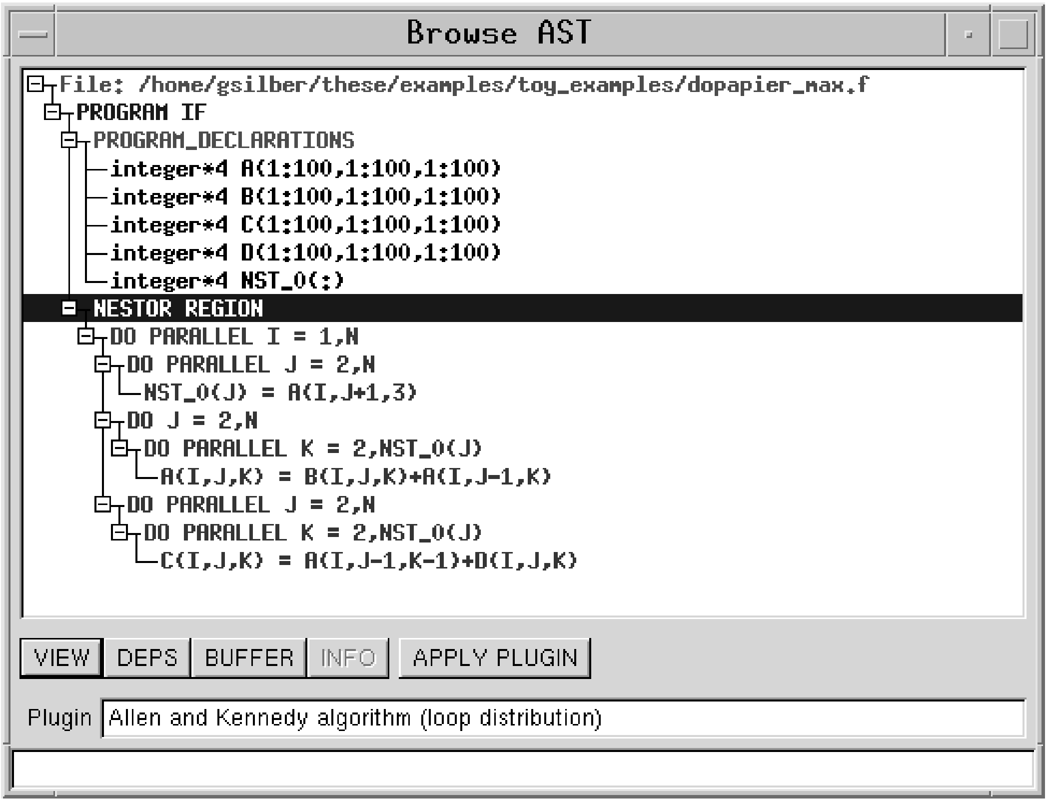


Figure 5.11: résultat de l'application de notre algorithme dans la fenêtre d'affichage de la structure.

d'optimisations dans le cadre d'un cours de compilation. Nous pensons qu'il y a quantité d'applications dans lesquelles Nestor serait très utile, grâce à sa conception claire.

Afin de ne pas se cantonner au langage Fortran, il serait tout à fait possible d'étendre Nestor afin que celui-ci supporte d'autres langages en entrée et en sortie. Il existe de nombreuses parties frontales dans le domaine public pour par exemple C et Java qui permettraient à Nestor, en développant avec métaNestor un nouvel ensemble de classes pour ces langages, de traiter indifféremment des codes en Fortran, C ou Java. Les transformations écrites pour le Fortran ne devraient pas forcément être modifiées en profondeur pour traiter ces nouveaux langages. Dans tous les cas, Nestor a été prévu dès le départ pour pouvoir être étendu grâce à sa conception modulaire.

Dans le futur, nous allons étendre Nestor pour avoir par exemple une représentation des domaines d'itération des boucles (grâce par exemple à la Polylib [71]), où encore intégrer un générateur de code depuis une représentation par des polyèdres (développé par Fabien Quilleré). Bref, de nombreux outils émergent dans la communauté française de la parallélisation automatique et Nestor pourrait utiliser ces outils pour offrir encore plus de fonctionnalités et offrir surtout une plate-forme commune de développement de transformations et d'analyses à la communauté. Nestor est disponible à l'URL suivante, avec différents outils comme Petit et `xvcg`, ainsi que la documentation complète (la version courante de Nestor, sans compter le code des modules et d'Adaptor, comporte environ 60000 lignes de code C++) :

`http://www.ens-lyon.fr/~gsilber/nestor/index.html`

Chapitre 6

Conclusions et perspectives

Dans cette thèse, nous avons présenté des algorithmes de transformation de code, un nouveau type de graphe des dépendances et un environnement de compilation. Ces apports sont une modeste contribution à ce domaine très vaste qu'est la compilation. Nous avons choisi comme cadre la transformation de code au niveau du code source, par insertion de directives de parallélisation, car celles-ci nous donnent suffisamment de latitude pour exprimer le résultat de nos algorithmes.

Le premier algorithme que nous avons présenté permet de trouver le nombre maximal de boucles complètement permutables dans le cas des graphes des dépendances réduits polyédriques, étape nécessaire à la recherche d'un pavage de l'espace d'itération pour augmenter le grain de la parallélisation. Nous n'avons accomplit que l'étape préliminaire : trouver un « bon » pavage reste un problème ouvert et générer le code efficace correspondant est également problématique. Notre travail permet néanmoins un grand degré de liberté quant au code pouvant être obtenu, ce qui devrait faciliter la génération d'un code efficace.

Le deuxième algorithme que nous avons présenté exploite le nouveau graphe des dépendances que nous avons introduit, le graphe des dépendances mixte. Ce graphe permet de représenter des codes à contrôle non statique et de prendre en compte l'introduction des temporaires pour supprimer des anti-dépendances. Cette introduction est dirigée par le programmeur qui décide où introduire ces temporaires et de quelle dimension ils doivent être, ou par le compilateur qui peut essayer tous les cas. Notre formalisme de graphe permet d'obtenir tous les codes légaux lorsque les seules transformations autorisées sont la distribution et la fusion. Nous pensons que ce formalisme pourrait être utilisé pour d'autres algorithmes de parallélisation, afin que ceux-ci puissent également traiter les codes ayant un flot de contrôle complexe.

Parallélisation semi-automatique. Nous nous sommes placés dans le cadre de la parallélisation semi-automatique, où les algorithmes de transformation doivent être suffisamment généraux pour pouvoir être paramétrés par le programmeur. Celui-ci, grâce à la connaissance de son programme et grâce à la connaissance de la machine cible, va pouvoir paramétrer la transformation pour obtenir un code qui lui paraît efficace. En restant au niveau du code source, il peut toujours voir le résultat de la transformation. Le paramétrage de l'algorithme peut se faire par l'intermédiaire de directives placées dans le code ou par l'intermédiaire d'une autre interface (voir Nestor et ses modules §5, p. 85).

Rester au niveau du code source permet d'effectuer un grand nombre de transformations et d'obtenir de très bons résultats. Le précompilateur ASTI d'IBM effectue des transformations au niveau du code source et obtient de très bonnes performances [54]. La très bonne synthèse de Bacon, Graham et Sharp [5] dresse une liste des transformations de haut niveau d'un hypothétique compilateur. Parmi ces transformations, on trouve les opérations sur les boucles, l'évaluation partielle, le clonage de procédures, la propagation de constantes, etc... Il nous semble donc tout à fait raisonnable de travailler au niveau du code source pour laisser un compilateur bas niveau générer un code parallèle depuis un code avec des directives (comme nous l'avons dit dans l'introduction, nous voyons les bibliothèques de passage de messages comme étant de trop bas niveau pour la programmation courante).

Ce choix consistant à rester au niveau du code repose bien entendu sur l'espoir que le compilateur de bas-niveau sera capable de générer un code efficace depuis le code parallèle. La société Tera [64] a récemment développé une machine appelée MTA à base de processeurs supportant le calcul multithreadé au niveau du matériel. Cette machine a été conçue dès le départ pour offrir un modèle de programmation qui est exactement celui que nous avons utilisé dans cette thèse et qui permet une correspondance totale entre le modèle et le matériel. Ainsi, la sémantique d'exécution d'un boucle parallèle telle que nous l'avons présentée dans la première partie (définition 4, page 11) décrit l'exécution réelle de cette boucle sur le processeur. Citons un extrait de la présentation de la Tera MTA [64] concernant le modèle de la machine :

> « *Tera provides a thread-based programming model that permits a mixture of implicit and explicit parallelism in the same program. The virtual machine has an unbounded number of processors with uniform access to all memory locations. The programmer may specify an unbounded number of threads interacting via shared data structures. The runtime system will acquire physical resources from the operating system and use these resources to implement the virtual machine.* »

Cette description correspond au modèle sous-jacent de la parallélisation automatique et c'est une des premières, si ce n'est la première machine à proposer un tel modèle au niveau du matériel.

> « (...) *Tera's compilers perform automatic parallelization of Fortran, C, and C++ source programs. Loop nests are restructured to enhance parallelism, with scalar expansion and array privatization used where beneficial.* (...) *The programmer can insert pragmas and directives into the program to help the compiler in automatic parallelization. The parallelism that the compiler discovers and exploits supplements whatever is explicitly generated by the user. Inner-loop parallelism is easy for the compilers to discover and manage, whereas outer-loop parallelism may only be evident to the programmer.* »

Nous en avions rêvé, Tera l'a fait. Voyons ce que Tera propose au niveau des compilateurs.

> « *Fortran 77 is supported, with a high degree of Cray compatibility. The MIL-STD intrinsics and a set of parallel directives are also provided. C is ANSI compliant, and except for templates and exceptions, C++ conforms to the ARM. Both C and C++ include pragmas for parallelism. All three languages may call each other freely. The compilers are implemented with a front-end for each language driving a common back-end. Fortran 90 support is planned for the second half of 1999. OpenMP support is provided, but support for MPI and pthreads is not currently scheduled.* »

La prudence s'impose cependant quant au devenir d'une telle machine. Il y eut de nombreuses tentatives par le passé d'architectures novatrices qui ont été un fiasco. Cette nouvelle machine nous conforte cependant dans l'idée que ce modèle de transformation de code est tout à fait valide, surtout que les performances sont au rendez-vous [54, 44, 42].

Nous pensons que notre outil Nestor sera utile dans le cadre de la parallélisation semi-automatique. Il reste un travail important à faire en ce qui concerne l'évaluation des performances des codes obtenus par les différents algorithmes de parallélisation. Maintenant que nous pouvons effectivement tester ces algorithmes sur de vrais codes, il reste à effectuer de nombreux tests. L'ambition de Nestor est de devenir un environnement de développement de code parallèle complet, gratuit et extensible, permettant aux programmeurs de paralléliser interactivement leurs codes et aux scientifiques de tester de nouvelles transformations.

Publications personnelles

Journaux internationaux

- Alain Darte, Georges-André Silber et Frédéric Vivien. « Combining retiming and scheduling techniques for loop parallelization and loop tiling ». *Parallel Processing Letters* **7**, numéro 4 (1997), 379–392.
- Pierre Boulet, Alain Darte, Georges-André Silber et Frédérique Vivien. « Loop parallelization algorithms : from parallelism extraction to code generation ». *Journal of Parallel Computing* **24**, numéro 3 (1998), 421–444. Special issue on Languages and Compilers for Parallel Computers.

Conférence internationale

- Georges-André Silber et Alain Darte. « The Nestor Library : A Tool for Implementing Fortran Source to Source Transformations ». Dans *High Performance Computing and Networking (HPCN'99)* (Avril 1999), volume 1593 des *Lecture Notes in Computer Science*, Springer Verlag, pp. 653–662.

Conférence nationale

- Georges-André Silber. « Nestor : un système de parallélisation automatique ». Dans *RenPar' 10 : dixièmes rencontres francophones du parallélisme* (Juin 1998).

Rapports de recherche

- Alain Darte, Georges-André Silber et Frédéric Vivien. « Combining retiming and scheduling techniques for loop parallelization and loop tiling ». Rapport de recherche 96-34, Laboratoire de l'Informatique du Parallélisme (LIP), École normale supérieure de Lyon, France, Novembre 1996.
  `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1996/RR1996-34.ps.Z`.
- Pierre Boulet, Alain Darte, Georges-André Silber et Frédérique Vivien. « Loop parallelization algorithms : from parallelism extraction to code generation ». Rapport de recherche 1997-17, Laboratoire de l'Informatique du Parallélisme (LIP), École normale supérieure de Lyon, France, Juin 1997.
  `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1997/RR1997-17.ps.Z`.
- Georges-André Silber et Alain Darte. « The Nestor library : a tool for implementing Fortran source to source transformations ». Rapport de recherche 1998-42, Laboratoire de l'Informatique du Parallélisme (LIP), École normale supérieure de Lyon, France, Septembre 1998.
  `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-42.ps.Z`.
- Georges-André Silber. « A programmer's guide for Nestor. Version 1. ». Rapport technique 1998-04, Laboratoire de l'informatique du parallélisme (LIP), École normale supérieure de Lyon, France, Octobre 1998.
  `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/TR/TR1998/TR1998-04.ps.Z`.

Document électronique

- Georges-André Silber. « Les bibliothèques de Nestor ». Document électronique WWW,

http://www.ens-lyon.fr/~gsilber/nestor/index.html.

Bibliographie

[1] A. Aho, R. Sethi et J. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] J. R. Allen, D. Callahan et K. Kennedy. « Automatic Decomposition of Scientific Programs for Parallel Execution ». Dans *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages* (Munich, Allemagne, Janvier 1987), pp. 63–76.

[3] J. R. Allen et K. Kennedy. « PFC : a Program to Convert Fortran to Parallel Form ». Dans *Proceedings of IBM Conference on Parallel Computing and Scientific Computations* (1982).

[4] J. R. Allen, K. Kennedy, C. Porterfield et J. Warren. « Conversion of Control Dependence to Data Dependence ». Dans *Proceedings of the Tenth Annual ACM Symposium on the Principles of Programming Language* (Austin, États-Unis, Janvier 1983).

[5] D. F. Bacon, S. L. Graham et O. J. Sharp. « Compiler Transformations for High-Performance Computing ». *ACM Computing Surveys* **26**, numéro 4 (1994).

[6] U. Banerjee. « Unimodular transformations of double loops ». Dans *Proceedings of the third Workshop on Programming Languages and Compilers for Parallel Computing* (Levine, CA, Août 1990), pp. 192–219.

[7] A. J. Bernstein. « Analysis of programs for parallel processing ». *IEEE Transactions on Electronic Computers* **15** (Octobre 1966), 757–762.

[8] P. Boulet. « Bouclettes : A Fortran Loop Parallelizer ». Dans *HPCN 96* (Bruxelles, Belgium, Juin 1996), Springer Verlag Lecture Notes in Computer Science, pp. 784–791.

[9] P. Boulet, A. Darte, T. Risset et Y. Robert. « (Pen)-ultimate tiling ? ». Rapport de recherche 1993-36, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Novembre 1993. `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1993/RR1993-36.ps.Z`.

[10] P. Boulet, A. Darte, T. Risset et Y. Robert. « (Pen)-ultimate tiling ? ». *Integration, the VLSI Journal* **17** (1994), 33–51. Voir également le rapport de recherche [9].

[11] P. Boulet, A. Darte, G.-A. Silber et F. Vivien. « Loop parallelization algorithms : from parallelism extraction to code generation ». Rapport de recherche 1997-17, Laboratoire de l'informatique de parallélisme (LIP), École normale supérieure de Lyon, France, Juin 1997. `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1997/RR1997-17.ps.Z`.

[12] P. Boulet, A. Darte, G.-A. Silber et F. Vivien. « Loop parallelization algorithms : from parallelism extraction to code generation ». *Journal of Parallel Computing* **24**, numéro 3 (1998), 421–444. Special issue on Languages and Compilers for Parallel Computers. Voir également le rapport de recherche [11].

[13] T. Brandes. « ADAPTOR, High Performance Fortran Compilation System ». Document électronique, `http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`.

[14] S. Carr et K. Kennedy. « Compiler blockability of numerical algorithms ». Dans *Supercomputing '92* (Novembre 1992), pp. 114–124.

[15] N. V. Chernikova. « Algorithm for discovering the set of all the solutions of a linear programming problem ». *USSR Computational Mathematics and Mathematical Physics* **8**, numéro 6 (1968).

[16] T. H. Cormen, C. E. Leiserson et R. L. Rivest. *Introduction to algorithms*. The MIT Press, 1990.

[17] A. Darte. « On the complexity of loop fusion ». Rapport technique 1998-50, LIP, ENS-Lyon, France, Octobre 1998. `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-50.ps.Z`.

[18] A. Darte. « On the complexity of loop fusion ». *Parallel Computing* (1999). Voir également le rapport de recherche [17].

[19] A. Darte, G.-A. Silber et F. Vivien. « Combining retiming and scheduling techniques for loop parallelization and loop tiling ». Rapport technique 96-34, LIP, ENS-Lyon, France, Novembre 1996. `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1996/RR1996-34.ps.Z`.

[20] A. Darte, G.-A. Silber et F. Vivien. « Combining retiming and scheduling techniques for loop parallelization and loop tiling ». *Parallel Processing Letters* **7**, numéro 4 (1997), 379–392. Voir également le rapport de recherche [19].

[21] A. Darte et F. Vivien. « Revisiting the decomposition of Karp, Miller and Winograd ». *Parallel Processing Letters* **5**, numéro 4 (1995), 551–562.

[22] A. Darte et F. Vivien. « A classification of nested loops parallelization algorithms ». *Parallel Processing Letters* **7**, numéro 2 (1997), 217–224.

[23] A. Darte et F. Vivien. « On the Optimality of Allen and Kennedy's Algorithm for Parallelism Extraction in Nested Loops ». *Parallel Algorithms and Applications* **12** (1997), 83–112.

[24] A. Darte et F. Vivien. « Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs ». *International Journal of Parallel Programming* **25**, numéro 6 (1997), 447–497.

[25] P. Feautrier. « Dataflow analysis of array and scalar references ». *International Journal of Parallel Programming* **20**, numéro 1 (1991), 23–51.

[26] P. Feautrier. « Some efficient solutions to the affine scheduling problem, part I : one-dimensional time ». *International Journal of Parallel Programming* **21**, numéro 5 (Octobre 1992), 313–348.

[27] P. Feautrier. « Some efficient solutions to the affine scheduling problem, part II : multidimensional time ». *International Journal of Parallel Programming* **21**, numéro 6 (Décembre 1992), 389–420.

[28] J. Ferrante, K. J. Ottenstein et J. D. Warren. « The Program Dependence Graph and its use in Optimization ». *ACM Transactions on Programming Languages and Systems* **9**, numéro 3 (Juillet 1987), 319–349.

[29] M. R. Garey et D. S. Johnson. *Computers and Intractability, a guide to the theory of NP-completeness*. W. H. Freeman and Company, 1979.

[30] P. Graham. « OpenMP, a parallel programming model for shared memory architectures ». Rapport technique, Edinburgh Parallel Computing Centre, The University of Edinburgh, Mars 1999. Disponible à l'url : `http://www.epcc.ed.ac.uk/epcc-tec/documents/`.

[31] High Performance Fortran Forum. « High Performance Fortran Language Specification ». Rapport technique 2.0, Rice University, Janvier 1997.

[32] F. Irigoin et R. Triolet. « Computing dependence direction vectors and dependence cones with linear systems ». Rapport de recherche ENSMP-CAI-87-E94, École des Mines de Paris, Fontainebleau, France, 1987.

[33] F. Irigoin et R. Triolet. « Supernode partitionning ». Dans *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA, Janvier 1988), pp. 319–329.

[34] F. Irigoin et R. Triolet. « Dependence approximation and global parallel code generation for nested loops ». Dans *Parallel Distributed Algorithms* (1989).

[35] R. M. Karp, R. E. Miller et S. Winograd. « The organization of computations for uniform recurrence equations ». *Journal of the ACM* **14**, numéro 3 (Juillet 1967), 563–590.

[36] K. Kennedy et K. S. McKinley. « Loop Distribution with Arbitrary Control Flow ». Dans *Supercomputing'90* (Août 1990).

[37] M. S. Lam et M. E. Wolf. « Automatic blocking by a compiler ». Dans *Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing* (1992), J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen et R. G. Voigt, éditeurs, pp. 537–542.

[38] L. Lamport. « The Parallel Execution of DO Loops ». *Communications of the ACM* **17**, numéro 2 (Février 1974), 83–93.

[39] T. Lengauer et R. E. Tarjan. « A Fast Algorithm for Finding Dominators in a Flowgraph ». *ACM Transactions on Programming Languages and Systems* **1**, numéro 1 (Juillet 1979), 121–141.

[40] A. W. Lim et M. S. Lam. « Maximizing Parallelism and Minimizing Synchronization with Affine Transforms ». Dans *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Janvier 1997), ACM Press.

[41] Message Passing Interface Forum. « MPI Forum Home Page ». Document électronique, `http://www.mpi-forum.org`.

[42] C. Miyamoto et C. Lin. « Evaluating Titanium SPMD Programs on the Tera MTA ». Dans *SuperComputing* (1999).

[43] Y. Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Février 1971.

[44] L. Oliker et R. Biswas. « Efficient Parallelization of a Dynamic Unstructured Application on the Tera MTA ». Dans *SuperComputing* (1999).

[45] OpenMP. « OpenMP Standard for Shared-memory Parallel Directives ». Document électronique, `http://www.openmp.org`.

[46] D. A. Padua et M. J. Wolfe. « Advanced compiler optimizations for supercomputers ». *Communications of the ACM* **29**, numéro 12 (Décembre 1986), 1184–1201.

[47] J. Peir et R. Cytron. « Minimum Distance : a Method for Partitioning Recurrences for Multiprocessors ». *IEEE Transactions on Computers* **38**, numéro 8 (Août 1989), 1203–1211.

[48] PIPS Team. « PIPS (Interprocedural Parallelizer for Scientific Programs) ». Document électronique, `http://www.cri.ensmp.fr/~pips/index.html`.

[49] W. Pugh. « Release 1.10 of Petit ». Document électronique, `http://www.cs.umd.edu/projects/omega/index.html`.

[50] PVM, Parallel Virtual Machine. « PVM Home Page ». Document électronique, `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[51] J. Ramanujam. « A Linear Algebraic View of Loop Transformations and Their Interaction ». Dans *Proceedings of the fifth SIAM Conference on Parallel Processing for Scientific Computing* (1992), J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen et R. G. Voigt, éditeurs, pp. 543–548.

[52] S. K. Rao. *Regular iterative algorithms and their implementations on processor arrays*. PhD thesis, Stanford University, Octobre 1985.

[53] G. Sander. « Visualization of Compiler Graphs ». Document électronique, `http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html`.

[54] V. Sarkar. « Automatic selection of high-order transformations in the IBM XL Fortran compilers ». *IBM Journal of Research and Development* **41**, numéro 3 (Mai 1997), 233–264.

[55] R. Schreiber et J. J. Dongarra. « Automatic Blocking of Nested Loops ». Rapport technique 90-38, The University of Tennessee, Knoxville, TN, Août 1990.

[56] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[57] G.-A. Silber. « Les bibliothèques de Nestor ». Document électronique, `http://www.ens-lyon.fr/~gsilber/nestor/index.html`. Voir également les deux documents [58, 59].

[58] G.-A. Silber. « Nestor : un système de parallélisation automatique ». Dans *RenPar' 10 : dixièmes rencontres francophones du parallélisme* (Juin 1998).

[59] G.-A. Silber. « A programmer's guide for Nestor. Version 1. ». Rapport technique 1998–04, Laboratoire de l'informatique du parallélisme (LIP), École normale supérieure de Lyon, France, Octobre 1998. `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/TR/TR1998/TR1998-04.ps.Z`.

[60] G.-A. Silber et A. Darte. « The Nestor Library : A Tool for Implementing Fortran Source to Source Transformations ». Dans *High Performance Computing and Networking (HPCN'99)* (Avril 1999), volume 1593 des *Lecture Notes in Computer Science*, Springer Verlag, pp. 653–662.

[61] B. Sinharoy et N. Szymanski. « Finding Optimum Wavefront of Parallel Computation ». Dans *Proc. of the Twenty-Sixth Annual Hawaii International Conference on System Sciences* (1993), H. El-Rewini, T. Lewis et B. D. Shriver, éditeurs, volume 2, IEEE Computer Society Press, pp. 225–234.

[62] Stanford Compiler Group. « The SUIF Compiler System ». Document électronique, `http://suif.stanford.edu/suif/suif.html`.

[63] B. Stroustrup. *The C++ programming language*, third edition. Addison-Wesley, 1997.

[64] Tera Computer Company. « TERA's multithreaded architecture ». Document électronique, http://www.tera.com.

[65] The LooPo Project. « The Polyhedral Loop Parallelizer : LooPo ». Document électronique, http://brahms.fmi.uni-passau.de/cl/loopo/index.html.

[66] The OpenMP Architecture Review Board. « OpenMP : A proposed industry standard API for shared memory programming ». White Paper, Octobre 1997. Voir également le document électronique [45].

[67] The Polaris Group. « Polaris Project Home Page ». Document électronique, http://polaris.cs.uiuc.edu/polaris/polaris.html.

[68] The Sage++ Development Team. « Sage++ Compiler toolkit ». Document électronique, http://www.extreme.indiane.edu/sage/index.html.

[69] F. Vivien. *Détection de parallélisme dans les boucles imbriquées*. Thèse de doctorat, École normale supérieure de Lyon, France, Décembre 1997.

[70] K. J. Wierenga. « Survey of compiler directives for shared memory programming ». Rapport technique, Edinburgh Parallel Computing Centre, The University of Edinburgh, Mars 1997. Disponible à l'url : http://www.epcc.ed.ac.uk/epcc-tec/documents/.

[71] D. Wilde et V. Loechner. « PolyLib - A library of polyhedral functions ». Document électronique, http://icps.u-strasbg.fr/PolyLib/index.html.

[72] M. E. Wolf et M. S. Lam. « A data locality optimizing algorithm ». Dans *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Juin 1991), pp. 30–44.

[73] M. E. Wolf et M. S. Lam. « A loop transformation theory and an algorithm to maximize parallelism ». *IEEE Transactions on Parallel and Distributed Systems* **2**, numéro 4 (Octobre 1991), 452–471.

[74] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Octobre 1982.

[75] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.

[76] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[77] X3H5 Committee. « Parallel extensions for Fortran ». Rapport technique X3H5/93-SD1-Revision M, Accredited Standards Committee X3, Avril 1994.

[78] J. Xue. « On tiling as a loop transformation ». *Parallel Processing Letters* **7**, numéro 4 (1997), 409–424.

[79] Y.-Q. Yang. *Tests des dépendances et transformations de programme*. Thèse de doctorat, Ecole Nationale Supérieure des Mines de Paris, Fontainebleau, France, 1993.

[80] M. Zöckler et R. Wunderling. « DOC++ ». Document électronique, http://www.zib.de/Visual/software/doc++/index.html.

Parallélisation automatique par insertion de directives

Résumé

Dans cette thèse, nous nous sommes placés dans le cadre de la parallélisation automatique par insertion de directives, c'est-à-dire des transformations de code au niveau du code source où le parallélisme est exprimé par des directives données au compilateur. Ce document expose trois apports à ce domaine. Les deux premiers apports sont des extensions au modèle polyédrique de représentation des nids de boucles imbriquées. La première extension se situe au niveau de la fonction objective. Notre objectif est la recherche de boucles complètement permutables, transformation nécessaire à une transformation appelée *tiling* permettant de modifier le grain de la parallélisation. La deuxième extension présente un nouveau type de graphe, le *graphe des dépendances mixte*, permettant de traiter les codes à contrôle non statique, c'est-à-dire les codes où des expressions de structures de contrôle (if ou do) sont impliquées dans des dépendances de données. En utilisant ce graphe, nous présentons une méthode pour introduire de la mémoire temporaire, ce qui permet de « casser » les anti dépendances impliquant des structures de contrôle. Le troisième apport de cette thèse est un outil appelé Nestor destiné à faciliter l'implantation de transformations source à source de programmes Fortran. Une extension de l'algorithme de parallélisation d'Allen, Callahan et Kennedy utilisant le graphe des dépendances mixte a été implantée dans notre outil.

Mots clés : compilation, parallélisation automatique, transformation de code, boucles, outils.

Automatic parallelization by directives insertion

Abstract

The context of this work is automatic parallelization by directives insertion, that is to say high level code source transformations where parallelism is expressed by directives given to the compiler. This text exposes three contributions to this domain. The first two contributions are extensions to the polyedric model used to represent loop nests. The first contribution extends the objective function to the research of fully permutable loops, preliminary step before the appliance of a transformation called *tiling* used to modify the grain of parallelization. The second extension is a new type of graph, the *Mixed Dependence Graph*, allowing the representation of non static control codes, where an expression contained into a control structure (if or do) is involved into a data dependence. We present a method using this graph to suppress anti dependences involving a control structure by adding extra memory. The third contribution of this thesis is a tool called Nestor aimed at easing the implantation of source to source Fortran code transformations. An extension of Allen, Callahan, and Kennedy's parallelisation algorithm using the mixed dependence graph has been implemented in our tool.

Keywords : compilation, automatic parallelization, code transformation, loops nests, tools.