N° d'ordre : 77

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir le titre de

Docteur de l'École normale supérieure de Lyon
Spécialité : Informatique
au titre de la formation doctorale de : Informatique de Lyon

par Frédéric VIVIEN

Détection de parallélisme dans les boucles imbriquées

Soutenue le 17 décembre 1997

Après avis de : Monsieur Paul FEAUTRIER
Monsieur Robert SCHREIBER

Devant la commission d'examen formée de :
Monsieur Philippe CHRÉTIENNE (président du jury)
Monsieur Alain DARTE (directeur de thèse)
Monsieur Paul FEAUTRIER
Madame Catherine MONGENET
Monsieur Sanjay RAJOPADHYE
Monsieur Yves ROBERT (directeur de thèse)

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

À mes parents.

À Laëtitia.

Oh please believe me
I'd hate to miss the train

I've got a feeling, **The Beatles**

I'll be your doctor
I'm on my way

Sun hits the sky, **Supergrass**

Remerciements

Avertissement: je tiens à préciser à ceux qui ne me connaissent pas ou peu qu'il est toujours dangereux de me prendre au pied de la lettre...

- À tout seigneur tout honneur : je commencerai par remercier Jocelyne, Marie, Marie-Jeanne, Sylvie, Valérie et toutes les secrétaires de l'école qui ont pu un jour s'occuper de moi ou auront encore un jour à le faire, Giles, Jean-Louis et Philippe et tous ceux responsables de la bonne marche des ordinateurs.
- Paul Feautrier (celui-là même qui a conçu l'algorithme éponyme) m'a fait l'immense honneur d'accepter d'être rapporteur. Qu'il me pardonne d'avoir eu l'outrecuidance d'oser comparer mes maigres heuristiques à ses puissants algorithmes.
- Rob Schreiber, nonobstant tout ce dont je lui étais déjà redevable, a relevé le défi et a accepté d'être rapporteur.
- Catherine Mongenet, pour assister à ma soutenance, est venue des confins du pays et ce n'est pas peu dire...
- Philippe Chrétienne a eu la lourde responsabilité de diriger le jury tout en ignorant les us et coutumes de la maison.
- Sanjay Rajopadhye a comblé à la soutenance une des lacunes de ce mémoire en rappelant l'historique de nos travaux.
- Yves « Chef » Robert. Dans l'introduction de ce mémoire, je justifie très brièvement l'intérêt de mes travaux mais je n'explique pas pourquoi j'ai choisi de m'intéresser à la parallélisation automatique. Juste avant d'attaquer le DEA, j'ai passé trois mois de stage à programmer. Bergen offrait un cadre on ne peut plus agréable, mais le travail en lui-même fut loin d'être satisfaisant. À mon retour en France, je ne pouvais plus voir un ordinateur en peinture et quand il s'est agi de chercher un stage de DEA, j'étais fermement décidé à verser dans la théorie. J'avais envie de faire de « l'algorithmique » et de rester à Lyon. Je suis tout naturellement allé voir Yves. Je ne connaissais pas ses travaux, mais je lui faisais confiance. Faisant fi des désaccords que nous avions pu avoir par le passé, il prit le risque de m'accepter dans son équipe et me jeta dans les bras de ses ex-thésards. Il me vanta aussi les charmes offerts par la position de « premier thésard » d'un jeune recruté, et j'ai eu l'occasion de vérifier par la suite la justesse de son conseil. Depuis il m'a prodigué moult autres conseils mais il ne s'est pas cantonné à son rôle de directeur de thèse administratif : nous avons aussi beaucoup travaillé ensemble, ce qui m'a permis de me rendre compte que l'expérience qu'il a acquise au fil des ans compense largement les quelques neurones qu'il a pu égarer chemin faisant. Pour résumer, je me rallierai à l'opinion des Beatles : *He help you to understand. He does everything he can. Doctor Robert...*

– Alain « chef » Darte. J'ai choisi les bras d'Alain, non parce que je les jugeais plus confortables, mais parce que son sujet de DEA ne comportait pas le mot « expérimentations ». Je ne comprenais pas grand chose au-dit sujet mais j'étais attiré par son côté « sans filet ». On ne peut pas dire que ma frangine l'appréciait autant que moi : elle garde un souvenir effroyable du repas qu'elle partagea avec moi et un Alain qui a passé son temps à m'expliquer mon sujet de DEA en général et, en particulier, les tenants et aboutissants de l'optimalité de l'ordonnancement linéaire des nids uniformes à plusieurs instructions.

  La prophétie d'Yves s'est révélée exacte : j'ai eu en Alain, plus qu'un directeur de thèse, un collègue de travail pour ne pas dire un compagnon de jeu, avec tout ce que cela implique. Je garde un souvenir attendri des heures que nous avons passées ensemble au tableau à confronter nos idées, nos visions, nos intuitions... Sans nous concerter, nous avons adopté une méthode de travail particulièrement efficace : l'un expose au tableau des idées ou des résultats pendant que l'autre essaye de prouver qu'il ne s'agit que d'un tissu de conneries, chacun défendant fermement ses positions. Une tel crible nous a permis de débusquer nombre erreurs et imprécisions. Au final, et quelle que soit la conclusion de la séance, il apparaît invariablement que les deux protagonistes avaient raison depuis le début ! Il nous est même arrivé de parier sur la véracité d'une propriété (à ce propos, Alain, tu me dois toujours une tablette de chocolat pour l'optimalité prouvée au paragraphe 4.4.2). Malgré notre longue collaboration, il est quand même des questions que nous avons dû laisser ouvertes. Par exemple : lequel de nous deux est de plus mauvaise foi ?

  Lors du choix du stage de DEA, mes copains de promo ont plaisanté sur le mode « il a choisi le stage du seul chercheur plus petit que lui ». À croire que c'était proportionnel à la taille, il s'est avéré que mon directeur de stage était aussi plus exigeant que moi, ce qui n'était pas gagné d'avance, quoi qu'il en pense. Il m'a parfois été dur d'être aussi royaliste que le roi, mais Alain a toujours su me ramener dans le droit chemin quand j'ai risqué de me fourvoyer. Il a aussi su m'encourager : « Pourquoi ne fais-tu pas que du théâtre ? ».

  Je conclurai la demi-page de remerciements exigée par une nouvelle citation musicale. Lors de mon séjour chez Hewlett-Packard j'ai découvert Alanis Morissette. Certains de ses vers m'ont rappelé Alain : *I am short but I am healthy [...] I am young and I am underpaid. I am tired but I am working...*

– Les membres de la communauté, avec qui j'ai eu plaisir à discuter.

– Dimitri, Luc, Odile et Yves qui ont encadré mon enseignement. L'expérience fut pour le moins instructive.

– Jean-Alex qui se met en quatre pour tous les élèves de l'ENS.

– Georges-André va enfin disposer d'Alain pour lui (presque) tout seul. J'espère qu'il me pardonnera de ne pas lui avoir laissé la place dans l'état dans lequel je l'avais trouvée...

– Thomas, Michèle, Pierre, Pierre-Yves, Georges-André, Olivier et David eurent l'honneur d'être mes différents co-bureaux et de devoir, à ce titre, répondre à mes questions stupides et supporter mes plaisanteries (même s'ils n'avaient parfois que très peu d'affinités avec mon humour).

– Arnaud, le roi des thésards, qui est allé expliquer ce que l'on pensait à la direction. Depuis, aucun chercheur n'a osé embaucher un thésard faisant le poids.

– Vincent et Séverine qui m'ont fourni, avec le club théâtre de l'ENS, plein de raisons de ne pas travailler. Ils m'ont appris à être sérieux comme un pape tout en racontant de manière claire et intelligible les pires bobards. Cet enseignement est loin d'être négligeable pour un futur (enseignant-)chercheur.

– Patrice m'a beaucoup appris sur la photo, mais c'est à Edgar que je dois la réputation de partir faire de la photo pendant les groupes de travail.

– Jameusse (ça dit tout, non ?).

– Walter ne supporte pas que l'on puisse penser du bien de ses toiles mais il eut suffisamment de patience et d'abnégation pour démontrer qu'il était possible de cohabiter avec moi plus de deux ans sans qu'aucune mort ne s'ensuive. Il s'agit là du seul résultat « pratique » établi par le-dit noctambule cette dernière décennie. Que la barbe du prophète parsème sa longue route de jeunes filles en fleurs.

– Sir Arthur Chagras, l'Ægagropyle, se méprenant sur le sens de la précédente phrase se mit à saupoudrer ces pages de ses poils. Il serait temps que le cher tychonobézoard pense à se faire débrediner.

– *An Dour Beo*, *Le Cheval Blanc* et *Le Chalet Pierre Termier* ont partiellement hébergé la rédaction de ce mémoire.

– La bande son a été assurée, dans l'ordre alphabétique, par : The Beatles, Cast, The Charlatans, Geneva, Kula Shaker, The Manic Street Preachers, Alanis Morissette, Oasis, Pink Floyd, Radiohead, The Seahorses, Alan Stivell, Supergrass.

– Et que ceux que j'ai oublié me pardonnent...

Table des matières

Introduction

Malheureux qui se fie à femme après cela !
La meilleure est toujours en malice féconde;
C'est un sexe engendré pour damner tout le monde.
J'y renonce à jamais à ce sexe trompeur,
Et je le donne tout au diable de bon cœur.

Sganarelle *in* L'école des maris, **Molière**

Nombre d'utilisateurs de l'informatique — principalement ceux qui ont besoin de beaucoup de puissance de calcul — aimeraient pouvoir exécuter sur des ordinateurs parallèles leurs programmes « séquentiels », c'est-à-dire écrits pour être exécutés sur des ordinateurs classiques. Il est donc devenu nécessaire de savoir transformer, « paralléliser », les programmes séquentiels en programmes exécutables efficacement sur machines parallèles. Cette parallélisation doit être automatique, ou au moins semi-automatique, pour deux raisons : elle est requise bien moins souvent par des informaticiens que par de simples utilisateurs, fréquemment néophytes; la complexité des transformations en jeu rend difficile une application manuelle quand elle ne l'interdit pas.

Avant d'initier toute transformation du programme originel, il est nécessaire de détecter et de quantifier le parallélisme qu'il contient implicitement, ce qui requiert la connaissance des relations de dépendance existant entre les différents calculs. Ultérieurement, il sera entre autres nécessaire de réordonner les calculs en explicitant le parallélisme découvert. Cette thèse a pour objet la détection automatique du parallélisme implicite et la recherche d'ordonnancements explicitant ce parallélisme pour des structures de programmes particulières. Il n'est en effet pas envisageable de pouvoir paralléliser n'importe quel programme. Nous nous restreignons donc à l'étude de programmes relativement réguliers et susceptibles de contenir beaucoup de parallélisme. Nous focaliserons notre attention sur les constructions répétitives des langages impératifs : les boucles imbriquées. Ces boucles décrivent en peu de lignes un grand nombre de calculs et représentent une grande partie du temps d'exécution des programmes de calcul scientifique.

Depuis plus de vingt-cinq ans, des chercheurs ont proposé des algorithmes de détection du parallélisme dans les boucles imbriquées, les plus importants étant dus à : Lamport [55], Allen et Kennedy [4], Wolf et Lam [77], Feautrier [39, 40] et Lim et Lam [59]. Le simple utilisateur qui en aurait le choix et l'informaticien développant un paralléliseur sont à même de se demander quel algorithme choisir, d'autant plus que ces algorithmes sont pour la plupart déclarés optimaux par leurs auteurs. Il nous a paru nécessaire de comprendre les méthodes, les modèles, les avantages et les inconvénients de ces différents algorithmes. Nous en avons étudié l'efficacité. Dans sa thèse [83], Yi-Qing Yang a entrepris de déterminer la représentation minimale des relations de dépendance nécessaire à la vérification de la validité d'une transformation de programmes. Nous répondons, en partie, à la question duale : étant donnée une représentation des dépendances, quel est l'algorithme le plus simple capable d'en extraire tout le parallélisme implicite?

L'étude des algorithmes existants montre clairement des lacunes dans la panoplie des solutions proposées : schématiquement, nous avons le choix entre des algorithmes simples dont les (contre-)

performances théoriques sont mathématiquement établies et des algorithmes complexes à l'efficacité inconnue. Nous avons voulu proposer une solution intermédiaire. Pour ce faire, nous sommes partis de l'étude théorique d'un modèle théorique : les systèmes d'équations récurrentes uniformes (SERU). Nous avons unifié les résultats accumulés depuis trente ans dans le domaine de la détection du parallélisme et de l'ordonnancement de ces systèmes d'équations. Nous avons ensuite montré comment transposer les résultats obtenus aux boucles imbriquées dont les dépendances sont représentées de manière approchée. Nous avons ainsi pu construire un algorithme de détection du parallélisme et d'ordonnancement pour les boucles imbriquées, algorithme que nous avons comparé aux algorithmes antérieurs.

L'ordonnancement est une des multiples étapes d'un paralléliseur automatique, comme l'analyse de dépendance, le placement des calculs et des données, le partitionnement, la génération du code, etc. Nous nous sommes donc naturellement intéressés aux interactions entre l'ordonnancement et deux des autres phases de la parallélisation : le placement des calculs et l'élimination de dépendances. Les problèmes de placement et d'ordonnancement sont en effet intimement liés et il n'est pas possible de les résoudre totalement indépendamment. Les techniques d'élimination des « fausses » dépendances — c'est-à-dire des dépendances qui ne sont pas dues au calcul étudié mais à son implantation en machine — ont pour finalité de permettre de trouver plus de parallélisme en relâchant les contraintes sur l'ordonnancement. Il ne sert donc à rien d'éliminer des dépendances qui ne contraignent pas l'ordonnancement.

Au chapitre 1, nous définissons les principaux objets et les notions que nous aurons à manipuler et tout particulièrement l'analyse de dépendance et les principales représentations des dépendances. Nous proposons des définitions formelles des notions de degré de parallélisme et d'optimalité d'un algorithme de détection du parallélisme.

Au chapitre 2, nous passons en revue les principaux algorithmes de détection du parallélisme : Lamport, Allen et Kennedy, Wolf et Lam, Feautrier, et Lim et Lam. Nous en décrivons les principes et nous en étudions l'efficacité et les limitations. Nous proposons un certain nombre de résultats d'optimalité nouveaux. Nous motivons l'intérêt d'une nouvelle étude du domaine et celui d'une représentation particulière des dépendances : les graphes de dépendance réduits polyédriques.

Le chapitre 3 est consacré aux problèmes de la calculabilité et de l'ordonnancement des systèmes d'équations récurrentes uniformes. Nous proposons un algorithme optimal d'ordonnancement de ces systèmes.

Le chapitre 4 est consacré aux graphes de dépendance réduits polyédriques. Nous montrons comment uniformiser ces graphes afin de les rendre manipulables et de pouvoir y transposer les résultats obtenus pour les systèmes d'équations récurrentes uniformes. Nous présentons un algorithme optimal d'ordonnancement de ces graphes. Cet algorithme peut aussi bien être utilisé pour une parallélisation à grain fin que pour une parallélisation à grain moyen. Nous le comparons aux solutions préexistantes.

Au chapitre 5, nous considérons l'interaction entre notre algorithme d'ordonnancement et les placements affines des calculs, dans le cadre défini au chapitre précédent. Nous proposons un algorithme qui prend en entrée un graphe de dépendance réduit polyédrique et un placement affine des calculs et qui construit un ordonnancement affine compatible avec le placement, s'il en existe un.

Au chapitre 6, nous nous intéressons à l'élimination des fausses dépendances. Nous étudions deux transformations initialement proposées de manière informelle par Padua et Wolfe. Nous proposons également un schéma d'intégration des techniques d'élimination de dépendances au sein des algorithmes de détection du parallélisme. Nous considérons l'application de ce schéma à trois algorithmes.

Chapitre 1

Concepts et définitions

On a tort de consacrer à la spéculation pure un temps que l'on pourrait occuper à l'examen des réalités tangibles, audibles, en un mot accessibles à nos organes de perception.

Le père *in* Les bâtisseurs d'empire ou Le schmürz, Boris **Vian**

Comme nous l'avons annoncé, notre étude porte sur les « boucles imbriquées ». Nous commençons donc par définir à la section 1.1 ces objets et les notions qui s'y rapportent. Nous énonçons également les hypothèses faites sur les programmes que nous nous proposons d'étudier.

Détecter du parallélisme c'est détecter des ensembles d'opérations exécutables simultanément. Fondamentalement, la détection du parallélisme est une étude des relations de dépendance entre les différentes opérations du programme. Elle est donc fortement tributaire de l'analyse et de la représentation des dépendances, véritables clefs de voûte de la parallélisation automatique : une analyse trop approximative ou une représentation trop grossière masquent irréversiblement le parallélisme effectivement contenu dans le programme étudié et vouent la parallélisation à l'échec. Nous aborderons à la section 1.2 le problème de l'analyse et surtout de la représentation des dépendances. Nous verrons que l'ensemble des dépendances d'un nid de boucles est généralement exprimé au moyen d'un graphe (pour toutes les notions relatives aux graphes nous renvoyons le lecteur au livre de Gondran et Minoux [36]). Dans ce cas, la détection de parallélisme est une étude de propriétés de graphes valués.

Une fois l'analyse de dépendance réalisée et le parallélisme détecté (s'il existe une phase de détection indépendante de la phase d'expression du parallélisme), le parallélisme du programme peut être explicité de deux manières différentes :

- Transformations de programme. Une ou plusieurs transformations sont successivement appliquées au code originel dans le but d'exhiber une ou plusieurs boucles parallèles. Parmi ces transformations figurent la distribution de boucles, la fusion de boucles imbriquées (*loop coalescing*) ou consécutives (*loop fusion*), la scission d'une boucle en boucles imbriquées, le déroulement, la normalisation, le renversement d'une boucle, l'interversion de boucles, les transformations linéaires, le partitionnement, la permutation des instructions, etc. Plusieurs articles et livres passent en revue toutes ces techniques, en particulier [84], [6] et [82].

- Ordre d'énumération. Le parallélisme, au lieu d'être exposé à l'aide d'une suite de manipulations élémentaires, est défini par la donnée d'un nouvel ordre d'énumération des opérations décrites par les boucles. Cet ordre peut être total (pour la génération de boucles permutables) ou partiel (pour la génération de boucles parallèles). Le nouvel ordre d'énumération est déterminé par des techniques d'ordonnancement ou des techniques dérivées. Cet ordre attribue à chaque calcul sa date logique d'évaluation. La donnée de l'ordre définit une transformation globale des boucles imbriquées.

Nous nous intéresserons principalement aux méthodes d'expression du parallélisme au moyen d'ordonnancements, mais nous considérerons néanmoins les suites de transformations, soit indirectement lors de l'étude des algorithmes classiques de détection du parallélisme (chapitre 2), soit directement dans le cadre de l'élimination des dépendances (chapitre 6). Notre but étant la détection maximale du parallélisme, nous nous intéresserons surtout à la parallélisation à grain fin (exécution simultanée d'opérations élémentaires). Nous considérerons également la parallélisation à grain moyen (exécution simultanée de groupes d'opérations élémentaires). Une fois donné l'ordonnancement, il reste encore à réécrire le programme. Nous ne nous intéressons pas ici à ce problème de génération de code. Nous renvoyons le lecteur à [13, 14] pour une revue des techniques existantes. À la section 1.3, nous définirons formellement les ordonnancements et les notations qui s'y rapportent.

Finalement, nous aborderons à la section 1.4 le problème de la définition de l'optimalité des algorithmes d'extraction du parallélisme. Nous allons en effet étudier plusieurs algorithmes qui n'ont de commun que le but. Nous avons donc besoin de pouvoir comparer leurs performances au moyen d'une mesure bien définie or il existe dans la littérature autant de définition de l'« optimalité » que d'algorithmes...

1.1 Nids de boucles : définitions et notations

Définition 1 (Nid de boucles).

Nous appelons **instruction** *une structure de programme finie qui peut être l'évaluation d'une expression mathématique et l'assignation du résultat à une variable, un appel de procédure ou l'exécution conditionnelle d'une séquence d'instructions.*

Nous appelons **nid de boucles**, **nid**, *ou ensemble de* **boucles imbriquées**, *une structure de programme finie de la forme :*

For $i = l(N_1, \dots, N_p)$ to $u(N_1, \dots, N_p)$ step r do
  B_1
  $\vdots$
  B_s
EndFor

où B_j est une instruction ou un nid de boucles, i une variable entière appelée **indice** *ou* **compteur** *de boucle, r une variable entière appelée* **incrément** *ou* **pas** *de la boucle, et l et u sont des fonctions définissant les* **bornes de boucle**. *L'ensemble des blocs $B_1, \dots, B_s$ constitue le* **corps** *de la boucle.*

Dans ce mémoire, nous ne considérons pas des nids aussi généraux. Nous supposons que la fonction l (resp. u) est une fonction affine de paramètres de structure et de compteurs de boucles englobantes ou est le maximum (resp. minimum) d'un ensemble de telles fonctions. De même, nous ne nous intéressons qu'à des nids dont toutes les boucles sont de pas unitaire ($r = 1$) et nous omettons généralement de spécifier la valeur de ce pas. Sauf indication contraire nous supposons que les pas sont toujours strictement positifs et nous nous servons de cette propriété tout au long de ce mémoire, implicitement. Cette restriction n'est pas contraignante, un simple changement de variable transformant une boucle à incrément négatif en une boucle à un incrément positif. Pour des raisons toutes autres, nous excluons du corps de boucle toute fonction d'arrêt intempestif ou de transfert de contrôle (goto, appel de procédure). Contrairement au modèle des *programmes à contrôle statique* de Paul Feautrier [38, 22], nous ne faisons aucune hypothèse *a priori* ni sur les types de données manipulés par le programme, ni sur les éventuelles fonctions d'accès à ces données et nous n'interdisons pas la présence de tests dans le corps de boucle.

Nous appelons **nid de boucles parfait**, **nid parfait**, ou **boucles parfaitement imbriquées**, un nid dont le corps de toute boucle contient exactement, soit un nid, soit un bloc d'instructions. Avec nos hypothèses, un nid parfait est donc de la forme :

```
For i_1 = l(N_1, ..., N_p) to u(N_1, ..., N_p) do
  For i_2 = l(N_1, ..., N_p, i_1) to u(N_1, ..., N_p, i_1) do
    ⋮
    For i_n = l(N_1, ..., N_p, i_1, ..., i_{n-1}) to u(N_1, ..., N_p, i_1, ..., i_{n-1}) do
      S_1
      ⋮
      S_s
    EndFor
    ⋮
  EndFor
EndFor
```

où $S_1, \ldots, S_s$ sont des instructions. Que le nid étudié soit parfait ou non, nous notons toujours ses instructions de la sorte et la variable s représente le nombre total d'instructions du nid. n est appelé la **profondeur** du nid. Dans le cas de nids non parfaits, la profondeur d'une instruction est égale au nombre de boucles qui l'entourent. Afin de simplifier les notations, nous omettons les délimiteurs EndFor dans nos exemples, l'imbrication des boucles et instructions étant alors spécifiée par les niveaux d'indentation.

Nous identifions les itérations des n_S boucles englobants l'instruction S avec les vecteurs de $\mathbb{Z}^{n_S}$, appelés **vecteurs d'itération**, contenus dans un polyèdre convexe $\mathcal{D}_S$, appelé **domaine d'itération**, borné par les bornes de boucles. Dans le cas d'un nid parfait, toutes les instructions ont le même domaine d'itération $\mathcal{D}$. La $i^{ème}$ composante d'un vecteur d'itération est égale à la valeur du $i^{ème}$ compteur de boucle, en comptant à partir de la boucle la plus externe. Dans un nid, les différentes itérations (instances) d'une même instruction sont donc exécutées dans l'ordre lexicographique[1] de leurs vecteurs d'itération. Nous notons $\prec$ l'**ordre lexicographique** strict :

$$\forall \boldsymbol{i}, \boldsymbol{j} \in \mathbb{Z}^n, \quad \boldsymbol{i} \prec \boldsymbol{j} \quad \Leftrightarrow \quad \exists k \leq n, \forall l \leq k-1, \quad \boldsymbol{i}_l = \boldsymbol{j}_l \text{ et } \boldsymbol{i}_k < \boldsymbol{j}_k$$

et $\preccurlyeq$ l'ordre lexicographique large.

Deux **boucles** sont dites **permutables** si l'une entoure l'autre et si leur interversion ne modifie pas la sémantique du code. Un **ensemble de boucles totalement permutables** est constitué de boucles imbriquées deux à deux permutables.

1. Cette propriété n'est bien évidemment vérifiée que par les nids dont toutes les boucles sont à incréments positifs, ce qui est notre cas.

Exemple 1

```
For i = 1 to N do
  For j = 1 to N do
    For k = 1 to N do
      S1 : b(i) = (3b(i) + 1)/4
      S2 : a(i,j,k) = a(i,k,j) + b(i)
```

L'exemple 1 présente un nid de boucles parfait. Ce nid contient deux instructions, S_1 et S_2, de même domaine d'itération : $\mathcal{D} = \{(\mathsf{i},\mathsf{j},\mathsf{k}) \mid 1 \leq \mathsf{i},\mathsf{j},\mathsf{k} \leq N\}$. Ses deux boucles externes sont permutables. Pour s'en convaincre, considérons deux opérations $S(\mathsf{i},\mathsf{j},\mathsf{k})$ et $T(\mathsf{i}',\mathsf{j}',\mathsf{k}')$, la deuxième dépendant de la première. Une opération exécutée pour une certaine valeur de l'indice de boucle i n'utilise pas une variable servant à une opération exécutée pour une autre valeur de l'indice i. Nous avons donc les relations : $\mathsf{i}' = \mathsf{i}$ et $(\mathsf{j},\mathsf{k}) \preccurlyeq (\mathsf{j}',\mathsf{k}')$. Après interversion des boucles, $S(\mathsf{i},\mathsf{j},\mathsf{k})$ (resp. $T(\mathsf{i},\mathsf{j}',\mathsf{k}')$) est exécutée quand le nouveau vecteur d'itération vaut $(\mathsf{j},\mathsf{i},\mathsf{k})$ (resp. $(\mathsf{j}',\mathsf{i},\mathsf{k}')$). Après permutation, $S(\mathsf{i},\mathsf{j},\mathsf{k})$ est donc toujours exécutée avant $T(\mathsf{i},\mathsf{j}',\mathsf{k}')$ et la relation de dépendance est respectée.

1.2 Les dépendances et leur représentation

Nous avons besoin de connaître les relations de dépendance existantes pour pouvoir déterminer quelles opérations d'un nid de boucles sont indépendantes et exécutables en parallèle ou pour vérifier la légalité d'une transformation. Nous rappelons ici les différents concepts de graphes de dépendance : graphe de dépendance étendu (GDE), graphe de dépendance réduit (GDR) et graphe de dépendance apparent (GDA), et la notion de vecteur de distance. Nous introduisons une nouvelle représentation approchée des dépendances, dite par polyèdres de dépendance, et nous montrons qu'elle englobe les deux représentations classiques : les niveaux de dépendance et les vecteurs de direction.

Graphes de dépendance et vecteurs de distance

Deux opérations sont indépendantes et peuvent être exécutées dans un ordre quelconque à condition, si elles accèdent à une même cellule mémoire, qu'elles n'y accèdent qu'en lecture.

Cette condition proposée par Bernstein [10] est suffisante, mais non nécessaire en théorie. Elle sert de critère de dépendance dans la majorité, sinon la totalité, des analyseurs. De nombreux tests de dépendance ont été mis au point, dont le test de Banerjee [7], l'I-test [53, 66], le Δ-test [45], le λ-test [57, 46], celui de Feautrier [38], celui de PIPS [48], l'Omega-test [67], etc. (voir [84, 82] pour une revue de ces tests, et [38] pour l'analyse de dépendance exacte).

Une dépendance est orientée suivant l'ordre séquentiel d'exécution, de l'opération *source* vers l'opération *puits* ou *cible*. Nous distinguons trois types de dépendances. Une écriture suivie d'une lecture dans la même zone mémoire génère une **dépendance de flot**, une écriture suivie d'une autre écriture engendre une **dépendance de sortie** et une lecture suivie d'une écriture occasionne une **anti-dépendance**. Nous écrivons $S(\boldsymbol{i}) \Longrightarrow T(\boldsymbol{j})$ si l'itération $\boldsymbol{j}$ de l'instruction T dépend de l'itération $\boldsymbol{i}$ de l'instruction S. Dans ce cas, il est à noter que le vecteur[2] $\boldsymbol{j} - \boldsymbol{i}$ est toujours **lexicographiquement positif.** L'ordre partiel sur les opérations défini par la relation $\Longrightarrow$ décrit le **graphe de dépendance étendu** (GDE, parfois appelé *graphe de dépendance détaillé*). À la figure 1.1, nous présentons un exemple de GDE au moyen de l'exemple 2.

2. Nous ne pouvons former la différence $\boldsymbol{j} - \boldsymbol{i}$ que si les deux vecteurs ont même dimension. Nous reviendrons plus loin sur ce point.

Exemple 2

```
For i=1 to N
  For j=1 to N
    a(i, j) = 1 + a(i, j-1) + a(i-1, N)
```

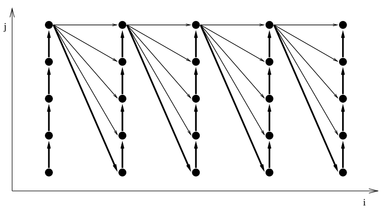


FIG. 1.1: *Exemple 2 et son graphe de dépendance étendu.*

En général, le GDE n'est pas effectivement généré à la compilation, soit parce qu'il manque des informations (la valeur de certains paramètres peut n'être fixée qu'à l'exécution, la localisation précise de certains accès mémoire peut ne pas être déterminable autrement que par exécution du programme, notamment dans le cas d'indirections[3], etc.), soit parce que la génération du graphe dans son intégralité est trop coûteuse.

De ce fait, les dépendances sont souvent capturées par un graphe orienté plus petit (souvent cyclique) qui contient autant de sommets que le nid d'instructions (c'est-à-dire s) et est appelé **graphe de dépendance réduit** (GDR, parfois appelé *graphe de dépendance généralisé*). Le GDR est un graphe de dépendance entre instructions. Dans le GDR, deux instructions S et T sont dites dépendantes (ce que nous notons $S \rightarrow T$) s'il existe au moins une paire $(\boldsymbol{i}, \boldsymbol{j})$ de vecteurs d'itérations telle que $S(\boldsymbol{i}) \implies T(\boldsymbol{j})$. De plus, l'arc[4] e du GDR de S vers T est étiquetée par l'ensemble $\{(\boldsymbol{i}, \boldsymbol{j}) \in \mathcal{D}^2 \mid S(\boldsymbol{i}) \implies T(\boldsymbol{j})\}$ ou par une sur-approximation D_e de cet ensemble. La précision et la représentation de cette approximation définissent la puissance de l'analyse de dépendance.

En d'autres termes, le graphe de dépendance réduit décrit sous une forme condensée un graphe de dépendance entre itérations appelé **graphe de dépendance apparent** (GDA). Ce graphe est une sur-approximation du graphe de dépendance étendu. Le GDE et le GDA ont les mêmes sommets, mais le GDA contient plus d'arcs :

$$(S, \boldsymbol{i}) \implies (T, \boldsymbol{j}) \text{ (dans le GDA)} \quad \iff \quad \exists\, e = (S, T) \text{ (dans le GDR) tel que } (\boldsymbol{i}, \boldsymbol{j}) \in D_e.$$

Pour une certaine classe de nids de boucles (cf. [38], [67] ou [61]), il est possible d'exprimer exactement cet ensemble de paires $(\boldsymbol{i}, \boldsymbol{j})$: $\boldsymbol{i}$ est défini par un *quast* ou *quasi affine selection tree*, c'est-à-dire une expression conditionnelle à plusieurs niveaux, dont les prédicats sont des inégalités affines des compteurs de boucles et des paramètres de structure, et qui renvoie une fonction quasi-affine du vecteur d'itération $\boldsymbol{j}$ [38, 22].

Retour sur l'exemple 1. Considérons les auto-dépendances en sortie de l'instruction S_1. Nous recherchons l'instance de S_1 dont dépend en sortie l'instance (i,j,k). Nous cherchons donc la dernière instance de S_1 ayant accédé en écriture b(i) avant que S_1(i,j,k) ne le fasse. Si nous supposons que N est supérieur à deux, cette instance nous est donnée par le *quast* :

3. Dans une étude empirique de la régularité des fonctions d'accès aux tableaux dans les programmes FORTRAN, Li, Yew et Zhu [58] ont montré que, dans les programmes qu'ils ont étudié, 53% de ces fonctions étaient des fonctions affines des seuls indices de boucle et que 8% des fonctions d'accès restantes faisaient intervenir des indirections.

4. En fait, le GDR contient un arc pour chaque paire d'accès mémoire qui induit une dépendance.

Si j = 1 et k = 1
  Alors $\perp$
  Sinon si k = 1
    Alors (i,j-1,N)
    Sinon (i,j,k-1)

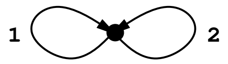
où le symbole $\perp$ indique qu'il n'y a pas de dépendance. Considérons les auto-dépendances de flot de l'instruction S_2. S_2(i,j,k) lit une donnée produite à l'itération :

Si j > k
  Alors (i,k,j)
  Sinon $\perp$

La plupart des algorithmes d'analyse de dépendance ne calculent pas l'ensemble des paires $(\boldsymbol{i}, \boldsymbol{j})$ mais plutôt l'ensemble $E_{S,T}$ de toutes les valeurs possibles de la différence $(\boldsymbol{j} - \boldsymbol{i})$ [5]. $E_{S,T}$ est appelé ensemble des **vecteurs de distance** ou **ensemble de distances** :

$$E_{S,T} = \{(\boldsymbol{j} - \boldsymbol{i}) \mid S(\boldsymbol{i}) \Longrightarrow T(\boldsymbol{j})\}$$

À la figure 1.2 nous présentons *un* graphe de dépendance réduit pour l'exemple 2. Nous avons choisi de représenter l'ensemble des distances par des niveaux de dépendance (nous définissons cette notion au paragraphe suivant). Comme le montre le graphe de dépendance apparent à la même figure, le GDR choisi contient en fait toutes les dépendances possibles.



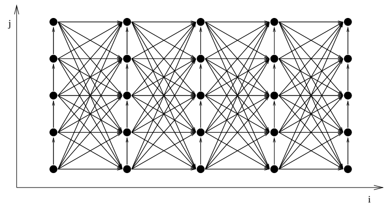


FIG. 1.2: *GDR par niveaux et GDA correspondant pour l'exemple 2.*

Il peut en fait y avoir plusieurs ensembles de vecteurs de distance attachés à un même couple d'instruction. Dans le cas de l'exemple 1, nous aurons trois ensembles de distances pour décrire les auto-dépendances de l'instruction S_1, un par type de dépendance (sortie, flot ou anti). L'ensemble des vecteurs de distance pour les auto-dépendances en sortie de l'instruction S_1 est ici la paire $\{(0,1,1-N),(0,0,1)\}$.

Quand une analyse de dépendance exacte est réalisable et que toutes les feuilles des arbres de sélection sont des fonctions affines et non pas seulement quasi-affine, l'ensemble des vecteurs de distance est la projection des points entiers d'une union de polyèdre. Cet ensemble peut donc être approché par son enveloppe convexe, par une description plus ou moins précise d'un polyèdre la contenant, voire par une union de polyèdres. Quand l'ensemble des distances est représenté par une union finie, l'arc de dépendance correspondant est décomposé en plusieurs arcs de mêmes extrémités.

5. Formellement, nous ne pouvons former la différence que si les deux vecteurs sont de même dimension. En règle générale, n'est considérée que la différence de la restriction des vecteurs d'itération à leur ensemble de boucles communes. Supposons que dans un nid non parfait $T(\boldsymbol{j})$ dépende de $S(\boldsymbol{i})$, les instructions T et S ayant exactement $n_{S,T}$ boucles englobantes communes. Le vecteur de distance associé à cette dépendance sera alors : $\boldsymbol{j}|_{n_{S,T}} - \boldsymbol{i}|_{n_{S,T}}$, où $\boldsymbol{i}|_k$ désigne le vecteur formé des k premières composantes du vecteur $\boldsymbol{i}$, et ce même si $\boldsymbol{i}$ et $\boldsymbol{j}$ sont de même dimension ($n_S = n_T$). Sauf indication contraire, nous ne parlerons de vecteurs de distance que pour des nids parfaits.

Il est à noter que la représentation par vecteurs de distance n'est pas équivalente à la représentation par paires, l'information sur la **localisation** du vecteur de distance dans le GDE étant perdue. Cette perte d'information peut même entraîner une perte de parallélisme, comme le montrera l'exemple 17. Ce type de représentation reste néanmoins important, notamment quand une analyse de dépendance exacte est impossible ou d'un coût prohibitif.

Représentations classiques par ensemble de distances (par ordre de précision croissante) :

- **niveaux de dépendance**, introduits pour l'algorithme de parallélisation d'Allen et Kennedy [2, 4].
- **vecteurs de direction**, ébauchés par Lamport [55] et définis par Wolfe [78, 79], ils sont utilisés dans l'algorithme de Wolf et Lam [77].
- **polyèdres de dépendance**, introduits par Irigoin et Triolet [49] et utilisés dans leur algorithme *supernode partitioning* [50]. Pour plus de détails le lecteur peut aussi se référer au logiciel PIPS [48].

Nous rappelons la définition de ces trois types de représentations et nous montrons que la troisième englobe les deux premières.

Dans tout ce mémoire, un **graphe orienté** est noté $G = (V, E)$, où V désigne l'ensemble des **nœuds** du graphe et E l'ensemble de ses **arcs**. Un arc e est dirigé de sa **queue** $q(e)$ vers sa **tête** $t(e)$: « $e : q(e) \rightarrow t(e)$ ». Un graphe est dit **valué** si une valeur (poids) est associée à chacun de ses arcs. Nous noterons $\boldsymbol{w}$ la fonction qui associe à un arc e son poids $\boldsymbol{w}(e)$. La valuation d'un chemin d'un graphe de dépendance est égale à la somme, si elle est définie, des valuations de ses arcs constitutifs. Nos graphes de dépendance seront valués par des entiers, des vecteurs d'entiers, ou des polyèdres, comme nous le présentons maintenant.

Niveaux de dépendance

La représentation par niveaux est la moins précise des approximations. Dans un nid de profondeur n, l'ensemble des vecteurs de distance $E_{S,T}$ est approché par un ensemble d'entiers de $[1, n] \cup \{\infty\}$ appelés niveaux. Dans ce contexte, l'entier $l \leq n$ représente tous les vecteurs de distance dont les $l - 1$ premières composantes sont nulles, mais non la $l^{ème}$. Le symbole ∞ représente le vecteur de distance nul.

Retour sur l'exemple 1. Considérons une nouvelle fois les auto-dépendances en sortie de l'instruction S_1. Nous avons vu que l'ensemble des vecteurs de distance était exactement : $\{(0, 1, 1 - N), (0, 0, 1)\}$. Nous pouvons représenter le premier vecteur de distance par le niveau « 2 » et le deuxième par le niveau « 3 ».

Une dépendance est de niveau $l \leq n$ si elle apparaît à la profondeur l dans le nid, c'est-à-dire pour une instance donnée des $l - 1$ boucles les plus externes. Dans ce cas, la dépendance est dite **portée par les boucles** au niveau l. Si $l = \infty$, la dépendance est interne au corps de boucle et elle relie les instances, pour le même vecteur d'itération, de deux instruction différentes. Une telle dépendance est dite **indépendante des boucles**. Un graphe de dépendance réduit dont les arcs sont étiquetés par des niveaux de dépendance est appelé **graphe de dépendance réduit par niveaux** (GDRN).

Vecteurs de direction

Quand l'ensemble des vecteurs de distance est un singleton, la dépendance est dite uniforme et l'unique vecteur de distance est appelé **vecteur de dépendance uniforme**.

Quand l'ensemble des vecteurs de distance comporte plusieurs éléments, il peut encore être représenté par un vecteur de dimension n, le **vecteur de direction**, dont les composantes appartiennent à $\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+,-\})$. La $i^{ème}$ composante de ce vecteur est une approximation des $i^{ème}$ composantes de tous les vecteurs de distance : elle vaut l'entier z si toutes les $i^{ème}$ composantes sont égales à z, elle vaut $z+$ (resp. $z-$) si toutes les $i^{ème}$ composantes sont supérieures (resp. inférieures) ou égales à z, et elle vaut « $*$ » si les $i^{ème}$ composantes peuvent prendre n'importe quelles valeurs. En général le symbole « $+$ » (resp. « $-$ ») est utilisé comme raccourci pour « $1+$ » (resp. « $(-1)-$ »).

La plupart des définitions de vecteurs de direction de la littérature n'autorisent que des valeurs des composantes prises dans l'ensemble : $\mathbb{Z} \cup \{*\} \cup \{0+, +, 0-, -\}$. Par ailleurs, les symboles « $+$ » et « $-$ » sont parfois notés « $>$ » et « $<$ ».

Retour sur l'exemple 1. L'ensemble des vecteurs de distance pour les auto-dépendances en sortie de l'instruction S_1 peut être représenté par les vecteurs de direction : $(0,1,-)$ et $(0,0,1)$.

L'ensemble des vecteurs de distance pour les auto-dépendances de flot de l'instruction S_2 est l'ensemble : $\{(0, j-k, k-j) \mid 1 \leq k < j \leq N\}$ qui peut être représenté par le vecteur : $(0,+,-)$.

Polyèdres de dépendance

Nous rappelons préalablement la définition mathématique des polyèdres et leurs propriétés de décomposition en sommets, rayons et lignes.

Définition 2 (Polyèdre, polytope).
Un ensemble P de vecteurs de $\mathbb{Q}^n$ est appelé polyèdre (convexe) s'il existe une matrice entière A et un vecteur entier b tels que :

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

Un polytope est un polyèdre borné.

Notre définition suppose que le vecteur b est un vecteur entier et n'est donc pas paramétré.

Un polyèdre peut toujours être décomposé en la somme d'un polytope et d'un cône polyédrique, appelé cône caractéristique (pour plus de détails, se référer à [74]). Un polytope est défini par la donnée de ses sommets : tout point du polytope est combinaison barycentrique positive des sommets. Un cône polyédrique est généré par un ensemble fini et est défini par ses rayons et lignes. Tout point du cône est la somme d'un combinaison positive des rayons et d'une combinaison quelconque des lignes. Par conséquent, un polyèdre $\mathcal{P}$ peut être défini de manière équivalente par son ensemble de *sommets* (notés $\{v_1, \ldots, v_\omega\}$), de *rayons* (notés $\{r_1, \ldots, r_\rho\}$) et de *lignes* (notés $\{l_1, \ldots, l_\lambda\}$). $\mathcal{P}$ est alors l'ensemble de tous les vecteurs $\boldsymbol{p}$ vérifiant :

$$\boldsymbol{p} = \sum_{i=1}^{\omega} \mu_i v_i + \sum_{i=1}^{\rho} \nu_i r_i + \sum_{i=1}^{\lambda} \xi_i l_i \qquad (1.1)$$

avec $\mu_i \in \mathbb{Q}^+$, $\nu_i \in \mathbb{Q}^+$, $\xi_i \in \mathbb{Q}$ et $\sum_{i=1}^{\omega} \mu_i = 1$.

Note. Afin d'éviter tout risque de confusion entre les **sommets** des polyèdres et ceux des graphes, ces derniers seront désignés par le terme de **nœud** dans l'ensemble de ce mémoire.

Nous sommes maintenant à même de définir les graphes de dépendance réduits polyédriques (GDRP), c'est-à-dire les graphes de dépendance réduits dont les arcs sont étiquetés par des polyèdres de dépendance. En fait, nous ne sommes intéressés que par les points entiers du polyèdre, les vecteurs de distance étant des vecteurs entiers.

Définition 3 (Graphe de dépendance réduit polyédrique (GDRP)).
Un graphe de dépendance réduit polyédrique est un graphe de dépendance réduit dont chaque arc $e : S \to T$ *est étiqueté par un polyèdre de dépendance* $\mathcal{P}(e)$ *qui est (ou qui fait partie d'un ensemble de polyèdres qui est) une sur-approximation de l'ensemble des vecteurs de distance : le graphe de dépendance apparent associé contient un arc d'une instance* $\boldsymbol{i}$ *de l'instruction* S *vers une instance* $\boldsymbol{j}$ *de l'instruction* T *si et seulement si* $(\boldsymbol{j} - \boldsymbol{i})$ *appartient à* $\mathcal{P}(e)$.

Nous étudierons en détail les graphes de dépendance réduits polyédriques au chapitre 4 après avoir prouvé leur intérêt au chapitre 2 (section 2.6). En première approximation, les polyèdres de dépendance peuvent être vus comme des vecteurs de direction dont les « directions » ne sont pas nécessairement parallèles aux axes. Nous montrons ci-dessous que les niveaux de dépendance et les vecteurs de direction peuvent être représentés de manière exacte par des polyèdres de dépendance.

Retour sur l'exemple 1. L'ensemble des vecteurs de distance pour les auto-dépendances de flot de l'instruction S_2 est l'ensemble : $\{(0, j-k, k-j) = (j-k)(0,1,-1) \mid 1 \leq k < j \leq N\}$. Cet ensemble peut donc être approché par le polyèdre d'unique sommet $(0,1,-1)$ et d'unique rayon $(0,1,-1)$.

Simulation des représentations classiques au moyen de polyèdres

Vecteurs de direction

Notons $\boldsymbol{e}_i$ le $i^{\text{ème}}$ vecteur de la base canonique, c'est-à-dire le vecteur de $\mathbb{Q}^n$ dont toutes les composantes sont nulles sauf la $i^{\text{ème}}$ qui est égale à 1. Un vecteur de direction n'est rien d'autre qu'une approximation par un polyèdre à un seul sommet et dont les hypothétiques rayons et lignes sont des vecteurs de la base canonique. Soit un arc e étiqueté par le vecteur de direction $\boldsymbol{d}$. $\boldsymbol{I}^+$, $\boldsymbol{I}^-$ et $\boldsymbol{I}^*$ désignent respectivement l'ensemble des composantes de $\boldsymbol{d}$ qui sont de la forme $z+$ (pour un entier z quelconque), $z-$ et « $*$ ». Finalement, nous notons $\boldsymbol{d}_c$ le vecteur de dimension n dont la $i^{\text{ème}}$ composante est égale à z si la $i^{\text{ème}}$ composante de $\boldsymbol{d}$ vaut z, $z+$ ou $z-$, et est égale à 0 sinon.

Par définition des symboles « $+$ », « $-$ » et « $*$ », le vecteur de direction $\boldsymbol{d}$ représente exactement tous les vecteurs entiers $\boldsymbol{p}$ de dimension n pour lesquels il existe des vecteurs (ν, ν', ξ) de $\mathbb{N}^{|\boldsymbol{I}^+|} \times \mathbb{N}^{|\boldsymbol{I}^-|} \times \mathbb{Z}^{|\boldsymbol{I}^*|}$ tels que :

$$\boldsymbol{p} = \boldsymbol{d}_c + \sum_{i \in \boldsymbol{I}^+} \nu_i \boldsymbol{e}_i - \sum_{i \in \boldsymbol{I}^-} \nu'_i \boldsymbol{e}_i + \sum_{i \in \boldsymbol{I}^*} \xi_i \boldsymbol{e}_i \tag{1.2}$$

Autrement dit, le vecteur de direction $\boldsymbol{d}$ représente tous les points entiers du polyèdre d'unique sommet $\boldsymbol{d}_c$, de rayons $\boldsymbol{e}_i$ pour $i \in \boldsymbol{I}^+$ et $-\boldsymbol{e}_i$ for $i \in \boldsymbol{I}^-$, et de lignes $\boldsymbol{e}_i$ pour $i \in \boldsymbol{I}^*$. Par exemple, le vecteur de direction $(2+, *, -, 3)$ équivaut au polyèdre de sommet $(2, 0, -1, 3)$, de rayons $(1,0,0,0)$ et $(0,0,-1,0)$, et de ligne $(0,1,0,0)$.

Niveaux de dépendance

Soit un arc de dépendance e de niveau $l < \infty$. Par définition du niveau, la première composante non nulle de tout vecteur de distance associé à e est la $l^{\text{ème}}$. Cette composante peut prendre n'importe quelle valeur entière strictement positive. En outre, nous n'avons aucune connaissance sur les composantes restantes. Par conséquent, une dépendance de niveau $l < \infty$ est équivalente au vecteur de direction $(\overbrace{0, \ldots, 0}^{l-1}, 1+, \overbrace{*, \ldots, *}^{n-l})$, alors qu'une dépendance de niveau ∞ correspond à un vecteur de distance nul. Comme tout vecteur de direction admet un polyèdre de dépendance équivalent, il en va de même d'une représentation par niveaux de dépendance. Par exemple, une dépendance de niveau 2 dans un nid de profondeur 3 équivaut au vecteur de direction $(0, 1+, *)$ qui correspond au polyèdre de sommet $(0, 1, 0)$, de rayon $(0, 1, 0)$ et de ligne $(0, 0, 1)$.

1.3 Ordonnancement : définitions et notations

Une fonction d'ordonnancement d'un ensemble d'opérations attribue à chaque opération une valeur prise dans un ensemble totalement ordonné $\mathcal{E}$. Pour notre problème spécifique, les ordonnancements doivent satisfaire des contraintes de précédence : si l'opération q dépend de l'opération p, alors p doit être ordonnée avant q, c'est-à-dire $\Theta(p) < \Theta(q)$, où Θ est la fonction d'ordonnancement et « $<$ » l'ordre total sur $\mathcal{E}$. Les ordonnancements définissent des ordres partiels.

Nous ne nous intéressons pas à n'importe quel type d'ordonnancement. Nous ne considérons que des fonctions relativement régulières à image dans un ensemble discret, et ce pour faciliter leur recherche, leur manipulation et la réécriture du code. Les ordonnancements les plus classiques sont à valeur dans $\mathbb{Z}^m$, où m est un entier. Nous manipulerons surtout des ordonnancements à valeurs dans $\mathbb{Q}^m$. Suivant la valeur de m nous parlerons d'ordonnancement monodimensionnel ($m = 1$) ou multidimensionnel ($m > 1$).

Définition 4 (Ordonnancement monodimensionnel).
Un ordonnancement monodimensionnel est une fonction Θ qui associe à chaque itération de chaque instruction d'un nid de boucles un entier (date) choisi de manière à préserver les dépendances :

$$\Theta : V \times \mathbb{Z}^n \to \mathbb{Z} \textit{ tel que } S(\boldsymbol{i}) \Rightarrow T(\boldsymbol{j}) \quad \Longrightarrow \quad \Theta(S, \boldsymbol{i}) < \Theta(T, \boldsymbol{j})$$

Par conséquent, une fonction Θ à valeur dans $\mathbb{Z}$ est un ordonnancement si et seulement si, pour tout couple de vecteurs d'itération $(\boldsymbol{i}, \boldsymbol{j})$ et tout couple d'instructions (S, T), elle vérifie :

$$\Theta(T, \boldsymbol{j}) \geq 1 + \Theta(S, \boldsymbol{i}) \tag{1.3}$$

si $T(\boldsymbol{j})$ dépend de $S(\boldsymbol{i})$.

La notion d'ordonnancement est étendue aux fonctions à valeurs dans $\mathbb{Q}$: est un ordonnancement (cf. [68]) la partie entière d'une fonction à valeur rationnelle qui satisfait l'équation 1.3. Par abus de notation, nous appellerons « ordonnancement » toute fonction rationnelle dont la partie entière est un ordonnancement. Sauf indication contraire, nous ne considérerons que des ordonnancements à valeurs rationnelles.

Nous ne nous préoccupons que d'ordonnancement réguliers[6] : ordonnancements linéaires, linéaires décalés (ou affine de partie linéaire unique) et affines. Ce sont des extensions des hyperplans introduits par Lamport [55] (cf. section 2.1).

Définition 5 (Ordonnancement affine).
Un ordonnancement affine est une application Θ de la forme :

$$\begin{array}{rrcl} & V \times \mathbb{Z}^n & \to & \mathbb{Q} \\ \Theta : & (v, \boldsymbol{i}) & \mapsto & X_v \boldsymbol{i} + \rho_v \end{array}$$

6. L'image par une fonction affine d'un ensemble discret, tel que l'ensemble des points entiers d'un polyèdre, est un ensemble discret. Nous pourrions donc définir directement les ordonnancements affines rationnels par une définition analogue à la définition 4, c'est-à-dire sans se ramener aux ordonnancements à valeurs entières. Nous ne le faisons pas car nous préférons manipuler des ordonnancements entiers et des inégalités larges, et pouvoir utiliser les techniques de programmation linéaire, plutôt que des ordonnancements rationnels et des inégalités strictes.

où les variables X_v représentent des vecteurs de $\mathbb{Q}^n$, les variables ρ_v des constantes rationnelles et où le produit $X_v\boldsymbol{i}$ est le produit scalaire.

Si tous les vecteurs X_v sont identiques, l'ordonnancement est dit **affine de partie linéaire unique** ou **linéaire décalé**. Si, de plus, toutes les constantes sont nulles ou toutes égales, l'ordonnancement est dit **linéaire**. Pour un ordonnancement linéaire décalé, l'inéquation 1.3 se transcrit :

$$\forall e \in G,\ X\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 1 \tag{1.4}$$

La formule $X\boldsymbol{w}(e)+\rho_{t(e)}-\rho_{q(e)}$ exprime le **délai** induit par l'ordonnancement sur l'arc e. Un vecteur X satisfaisant l'ensemble d'inéquations (1.4) est appelé **hyperplan fortement séparateur**. Un vecteur qui, complété par des constantes idoines, n'induit qu'un délai positif ou nul sur les arcs de dépendance est un **hyperplan faiblement séparateur**.

Nous définissons maintenant la notion d'ordonnancement multidimensionnel, c'est-à-dire d'ordonnancement à valeurs dans un espace $\mathbb{Q}^m$ de dimension éventuellement strictement supérieure à un. Ces ordonnancements ont été introduits et étudiés par Karp, Miller et Winograd [51], Lamport [55], Rao [70], Roychowdhury [72] et Feautrier [40].

Définition 6 (Ordonnancement multidimensionnel).
Un ordonnancement multidimensionnel de dimension d est une fonction qui définit un ordre d'exécution par l'association d'un vecteur entier de dimension d à chaque itération de chaque instruction. Les vecteurs image sont ordonnés entre eux grâce à l'ordre lexicographique (noté $\prec$) et sont choisis de manière à respecter les dépendances. Un tel ordonnancement est donc une application de $V \times \mathbb{Z}^n$ dans $\mathbb{Z}^d$.

Les différentes composantes de l'ordonnancement peuvent être vues comme des jours, des heures, des minutes, des secondes, etc. sachant que tous les jours n'ont pas forcément autant d'heures, les heures de minutes, etc. L'ordre séquentiel d'exécution d'un nid de boucles de profondeur supérieure ou égale à deux est un ordonnancement multidimensionnel.

Comme pour les ordonnancements monodimensionnels, nous étendons la notion d'ordonnancement aux fonctions à valeurs rationnelles et nous ne nous intéressons qu'à des fonctions un tant soit peu régulières.

Définition 7 (Ordonnancement affine multidimensionnel).
Un ordonnancement affine multidimensionnel de dimension d est une application Θ de la forme :

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \rightarrow & \mathbb{Q}^d \\ \Theta :\ (v, \boldsymbol{i}) & \mapsto & \left(X_v^1\boldsymbol{i} + \rho_v^1, \ldots, X_v^d\boldsymbol{i} + \rho_v^d\right) \\ S(\boldsymbol{i}) \Rightarrow T(\boldsymbol{j}) & \Longrightarrow & \left\lfloor\left(X_S^1\boldsymbol{i} + \rho_S^1, \ldots, X_S^d\boldsymbol{i} + \rho_S^d\right)\right\rfloor \prec \left\lfloor\left(X_T^1\boldsymbol{j} + \rho_T^1, \ldots, X_T^d\boldsymbol{j} + \rho_j^d\right)\right\rfloor \end{array}$$

où les variables X_v^i représentent des vecteurs de $\mathbb{Q}^n$ et les variables ρ_v^i des constantes rationnelles.

Quand, à toute profondeur i, les vecteurs X_v^i sont égaux pour les deux extrémités d'un arc e ($X^i = X^i_{q(e)} = X^i_{t(e)}$), le système de la définition 7 est vérifié si le système ci-dessous l'est :

$$\exists k_e \leq d \quad \forall i \in [1, k_e - 1] \quad \begin{array}{r} X^i\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 0 \\ X^{k_e}\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 1 \end{array}$$

Les ordonnancements multidimensionnels affines apparaissent donc liés aux notions d'hyperplans faiblement et fortement séparateurs. Les chapitres 3 et 4 s'étendront longuement sur le problème.

1.4 Optimalité des algorithmes de détection de parallélisme

Dans ce mémoire nous ne nous occupons pas du parallélisme exhibé par les boucles dites DOACROSS (cf. [84]) ou par les techniques d'ordonnancement d'une boucle unique sous contraintes de ressources (*software pipelining*). Nous ne nous intéressons qu'à la détection de grandes quantités de calculs indépendants qui peuvent être décrits par des boucles régulières. Si nous prenons l'exemple du langage HPF [43], nous ne considérons que des boucles qui peuvent être précédées de la directive `!HPF independent`. Dans nos exemples, de telles boucles seront désignées par le mot-clé **ForPar** [7] (nous noterons **ForSeq** une boucle séquentielle d'un programme sous forme parallélisée).

Dans un tel contexte, il serait tentant de définir naïvement la détection maximale de parallélisme comme la recherche, ou la génération, du plus grand nombre possible de boucles **ForPar**. Comme le montre l'exemple 3, un nid parfait de profondeur deux dont toutes les boucles sont parallèles (ForPar) est équivalent à une simple boucle parallèle.

Exemple 3

```
ForPar i = 1 to 1000
  ForPar j = 1 to 100
    a(i,j) = 0
```

```
ForPar I = 0 to 99999
  a((I div 100) + 1, (I mod 100) + 1) = 0
```

Les deux nids ci-dessus sont équivalents et révèlent le même parallélisme, mais le premier contient plus de boucles parallèles. La définition intuitive est donc rendue inutilisable par des transformations telles que la fusion de boucles imbriquées (*loop coalescing* en anglais) qui peuvent modifier le nombre de boucles imbriquées. Nous ne voulons faire aucune hypothèse sur les manipulations autorisées ou interdites aux algorithmes de parallélisation, ni sur la forme des codes générés, car nous voulons pouvoir évaluer n'importe quel algorithme. Nous avons donc besoin d'une définition précise pour mesurer la quantité de parallélisme que contient un programme.

Nous supposons que les algorithmes de détection de parallélisme que nous manipulons ne connaissent du nid $\mathcal{L}$ étudié que son graphe de dépendance réduit. Ils cherchent donc le parallélisme intrinsèque du GDR et non pas celui du nid. Tout algorithme de détection de parallélisme qui transforme $\mathcal{L}$ en un nid équivalent $\mathcal{L}_t$ se doit de préserver toutes les dépendances contenues dans le GDR, c'est-à-dire toutes les dépendances décrites par le GDA : si $S_i(\boldsymbol{i}) \Rightarrow S_j(\boldsymbol{j})$ dans le GDA, alors $S_i(\boldsymbol{i})$ doit être calculé avant $S_j(\boldsymbol{j})$ dans $\mathcal{L}_t$. Notre définition de l'optimalité nécessite un certain nombre de définitions préliminaires, à commencer par une définition théorique de la latence d'un nid parallélisé :

Définition 8 (Latence).
Nous définissons la latence $\mathcal{T}(\mathcal{L}_t)$ du nid transformé $\mathcal{L}_t$ comme le nombre minimal de cycles d'horloge requis pour l'exécuter si :

- *un nombre illimité de processeurs est disponible;*
- *l'exécution d'une instance d'une instruction nécessite un unique cycle d'horloge;*
- *tout autre opération ne coûte aucun cycle.*

7. Notre mot-clé ForPar ne modifie pas la sémantique du programme contrairement au Forall de Fortran90 qui correspond en général à une affectation sur des tableaux.

La latence calculée par la définition 8 ne sera évidemment pas égale au temps d'exécution du programme. Il s'agit juste d'une évaluation de la séquentialité du nid $\mathcal{L}_t$. La latence nous permet cependant de définir une notion de **degré de parallélisme** indépendante de l'architecture cible et qui révèle les qualités intrinsèques du GDR du code originel. Il est en fait indispensable, pour notre étude, d'utiliser une définition plus précise de la latence qui dépende d'une instruction S considérée : la **S-latence**. La S-latence est définie comme la latence excepté le fait que seules les opérations qui sont instances de l'instruction S requièrent un cycle pour s'exécuter (les autres s'exécutent instantanément).

Nous sommes maintenant à même de définir le degré de parallélisme détecté par un algorithme, pour une représentation donnée des dépendances. Cependant, afin d'éviter les problèmes du type de ceux illustrés par l'exemple 3, nous nous restreignons à l'étude de programmes paramétrés. Nous supposons que dans les définitions qui suivent tous les GDR et les GDA sont décrits par la même représentation des dépendances.

Définition 9 (S-degré de séquentialité extraite, de parallélisme extrait).
Soit A *un algorithme de détection de parallélisme. Soient $\mathcal{L}$ un nid de boucles et G son GDR. Nous parallélisons G à l'aide de* A. *Nous notons $\mathcal{D}_S$ le domaine d'itération de l'instruction S. Nous supposons que $\mathcal{D}_S$ contient (resp. est contenu dans) un hypercube de dimension n_S et de taille $\Omega(N)$ (resp. $O(N)$), où N est un paramètre. Le S-degré de séquentialité extraite (resp. de parallélisme extrait) de G par* A *vaut alors d_S (resp. $n_S - d_S$) où d_S est le plus petit entier positif tel que la S-latence du code transformé soit $O(N^{d_S})$.*

Rappels. Soient f et g deux fonctions. $f = O(g)$ si il existe k et N_0 tels que $f(N) \leq kg(N)$ pour N supérieur à N_0. $f = \Omega(g)$ si $g = O(f)$ et $f = \Theta(g)$ si $f = \Omega(g)$ et $g = O(f)$.

Nous avons supposé que les algorithmes étudiés ne connaissaient du nid à paralléliser que le graphe de dépendance réduit. En particulier, ils n'ont aucune information sur le domaine des instructions et la parallélisation est donc indépendante de ces domaines. L'hypothèse sur $\mathcal{D}_S$ n'est donc pas restrictive.

La définition proposée est encore loin d'être parfaite : elle ne distingue pas un code purement séquentiel d'un nid contenant un ensemble d'opérations indépendantes de taille $\log(N)$. Elle nous permet néanmoins de relier la latence et la longueur des chemins de dépendance du GDA, la S-latence et la S-longueur des chemins de dépendance du GDA (la S-longueur d'un chemin est le nombre de ses sommets qui sont des instances de l'instruction S). En effet, deux opérations reliées par une arête dans le GDA ne peuvent être calculées au même cycle d'horloge dans le nid transformé $\mathcal{L}_t$. La latence de $\mathcal{L}_t$, quel que soit l'algorithme de parallélisation utilisé, est donc supérieure au plus long chemin de dépendance du GDA. Plus précisément :

- Si un algorithme réussit à transformer le nid initial en un nid dont la S-latence est $O(N^{d_S})$ pour l'instruction S, alors la S-longueur de tout chemin de dépendance est également $O(N^{d_S})$.
- De manière équivalente, si le GDA contient un chemin d'une S-longueur qui ne soit pas $O(N^{d_S})$, alors, quel que soit l'algorithme de parallélisation utilisé, la latence du code transformé ne pourra pas être $O(N^{d_S})$.

Ceci nous amène à poser les définitions suivantes :

Définition 10 (S-degré de séquentialité intrinsèque, de parallélisme intrinsèque).
Soit G un GDR. Supposons que le domaine d'itération $\mathcal{D}_S$ contient (resp. est contenu dans) un hypercube de dimension n_S et de taille $\Omega(N)$ (resp. $O(N)$), où N est un paramètre. Soit d_S le plus petit entier positif tel que la S-longueur de tout chemin de dépendance du GDA est $O(N^{d_S})$.

Le S-degré de séquentialité intrinsèque (resp. de parallélisme intrinsèque) de G est alors d_S (resp. $n_S - d_S$).

De par les définitions 9 et 10, le S-degré de parallélisme extrait d'un GDR est toujours inférieur à son S-degré de parallélisme intrinsèque, ce qui est heureux !

Définition 11 (Extraction optimale de parallélisme).
Un algorithme A *extrait optimalement le parallélisme si pour chaque graphe de dépendance réduit G, et pour chaque instruction S de G, le S-degré de parallélisme extrait de G par* A *est égal au S-degré de parallélisme intrinsèque de G.*

Définir l'optimalité de la détection de parallélisme à partir de la S-latence au lieu de la latence, c'est-à-dire regarder les performances pour chacune des instructions plutôt que globalement, nous permet de discuter de la qualité des algorithmes de parallélisation même pour des instructions qui n'appartiennent pas à la partie la plus séquentielle du code originel.

Cette définition a le même défaut que la définition 9 : considérer comme séquentiel un code qui ne contient pas un degré de parallélisme. Considérons en effet une unique boucle de taille N dont le plus long chemin de dépendance est en $log(N)$. Avec notre définition, ne pas transformer le programme est considéré optimal même s'il est possible de le rendre exécutable en $log(N)$! Dans tous les cas étudiés aux chapitres 2, 3 et 4, nous aurons en fait un S-degré de séquentialité intrinsèque en $\Omega(N^{d_S})$ et un S-degré de séquentialité extraite en $O(N^{d_S})$.

Avec toutes ces définitions, nous pouvons prouver comme suit l'optimalité d'un algorithme A de détection de parallélisme. Soit $\mathcal{L}$ un nid de boucles de GDR G, pour une certaine représentation des dépendances, et de GDA G_a. Soit d_S le S-degré de séquentialité extraite de G par A. Nous avons deux moyens de prouver l'optimalité de A :

1. Construire pour chaque instruction S un chemin de G_a dont la S-longueur n'est pas $O(N^{d_S-1})$.

2. Construire un nid de boucles $\mathcal{L}$' dont le GDR est également G, pour la représentation des dépendances choisie, et dont le graphe de dépendance étendu contient, pour chaque instruction S, un chemin dont la S-longueur n'est pas $O(N^{d_S-1})$. Le nid $\mathcal{L}$' est appelé **nid apparent** (car l'algorithme qui parallélise $\mathcal{L}$ se comporte comme s'il parallélisait $\mathcal{L}$').

En fait (2) implique (1), le GDE de $\mathcal{L}$' étant un sous-ensemble de G_a ($\mathcal{L}$ et $\mathcal{L}$' ont même GDR). Par conséquent, prouver (2) établit un résultat plus fort. En particulier, le nid apparent révèle les limitations intrinsèques, pour l'extraction de parallélisme, de la représentation des dépendances choisie : même si le parallélisme présent dans $\mathcal{L}$ et $\mathcal{L}$' est différent à l'exécution (s'ils n'ont pas le même graphe de dépendance étendu), ils ne peuvent être distingués par l'algorithme de parallélisation puisqu'ils ont le même graphe de dépendance réduit. En d'autres termes, tout algorithme de détection de parallélisme (prenant en entrée la représentation des dépendances choisie) parallélisera de même $\mathcal{L}$ et $\mathcal{L}$'. Comme $\mathcal{L}$' est parallélisé optimalement, l'algorithme est considéré optimal *pour* la représentation des dépendances choisie.

La première méthode établit un résultat plus faible que la seconde mais elle est plus facile à établir. Nous l'utilisons surtout quand nous ne sommes pas capables de produire un nid apparent.

La première technique est utilisée au chapitre 4 pour prouver l'optimalité de l'algorithme d'ordonnancement des graphes de dépendance réduits polyédriques. La deuxième technique est utilisée pour établir l'optimalité de l'algorithme de Allen et Kennedy (cf. section 2.2 et [32, 33]). La figure 1.3 rappelle les relations entre le nid originel $\mathcal{L}$, le nid apparent $\mathcal{L}$' et leurs graphes de dépendance étendus, réduits et apparents respectifs.

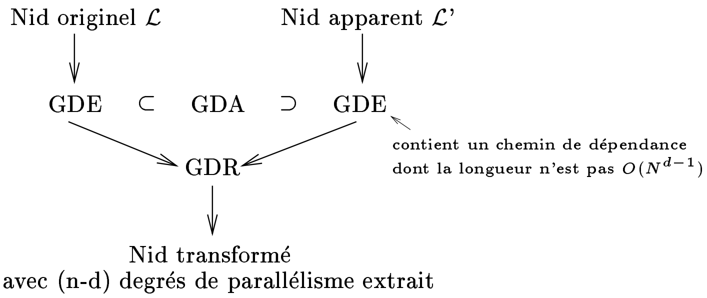


FIG. 1.3: *Relations entre $\mathcal{L}$, $\mathcal{L}$', et leurs différents graphes de dépendance.*

D'aucuns pourraient prétendre que la latence et la S-latence du code transformé ne sont pas faciles à calculer. Dans le cas général, elles ne peuvent en effet être calculées que par exécution du code pour une valeur donnée du paramètre N. Il est cependant possible, pour la majorité des algorithmes de parallélisation, de calculer le S-degré de parallélisme extrait (mais pas nécessairement la S-latence) par un simple examen de la structure du code transformé, ce que formalise le lemme 1. Nous retrouvons dans ce cas la définition intuitive du S-degré de parallélisme comme nombre de boucles imbriquées et parallèles qui englobent l'instruction S.

Lemme 1. *Nous supposons que le code transformé $\mathcal{L}_t$ ne contient pas d'autres structures de contrôle que des boucles, que les variables et paramètres apparaissant dans les bornes de boucles ne sont pas modifiés dans le corps du nid, et que chaque instruction S du code originel $\mathcal{L}$ apparaît une unique fois dans le code transformé $\mathcal{L}_t$ et est entourée dans les deux nids par exactement n_S boucles. Nous supposons également que le domaine d'itération $\mathcal{D}_S$ décrit par ces n_S boucles contient un hypercube $\underline{\mathcal{D}}$ de dimension n_S et de taille $\Omega(N)$ et est contenu dans un hypercube $\overline{\mathcal{D}}$ de dimension n_S et de taille $O(N)$. Sous ces hypothèses, le nombre de boucles parallèles englobant S est égal au S-degré de parallélisme extrait.*

DÉMONSTRATION. Considérons une instruction S du code originel $\mathcal{L}$. Pour simplifier l'argument de la démonstration, notons $\mathcal{L}_r$ le code obtenu par suppression de $\mathcal{L}_t$ de tout ce qui ne concerne pas les instances de S. La latence de $\mathcal{L}_r$ est, par définition, la S-latence de $\mathcal{L}_t$. De plus, $\mathcal{L}_r$ est un nid parfait de profondeur n_S qui contient une unique instruction : S. Soit $\overline{L}$ (resp. $\underline{L}$) le nid parfait dont les boucles décrivent le domaine d'itération $\overline{\mathcal{D}}$ (resp. $\underline{\mathcal{D}}$) et qui contient l'unique instruction S.

Comme $\underline{\mathcal{D}} \subset \mathcal{D}_t \subset \overline{\mathcal{D}}$, la latence de $\mathcal{L}_r$ est supérieure à celle de $\underline{L}$ et inférieure à celle de $\overline{L}$. En outre, comme $\overline{\mathcal{D}}$ et $\underline{\mathcal{D}}$ sont des hypercubes de dimension n_S, leur latence est facile à calculer. La latence de $\underline{L}$ est $\Omega(N^d)$ et celle de $\overline{L}$ vaut $O(N^d)$, où d est le nombre de boucles séquentielles entourant S. Par conséquent, la latence de $\mathcal{L}_r$ est $\Theta(N^d)$ et le S-degré de parallélisme extrait de $\mathcal{L}_r$ (et donc le S-degré de parallélisme extrait de $\mathcal{L}$) est $(n_S - d)$, c'est-à-dire le nombre de boucles parallèles englobant l'instruction S. $\square$

Chapitre 2

Algorithmes classiques d'ordonnancement

> Quelle nuit magnifique ! Les sons lointains de la musique et des rires, le chant des grillons, le murmure de l'eau. Ah ! jardin d'Éden, auquel il manque un élément ! [...] L'adorable présence d'une femme que l'on désire. Il manque à la brise du soir le son de sa voix. Les murmures de la terre réclament les protestations de son amour. O femmes...
>
> Abram Abramovitch Vengerovitch *in* Ce fou de Platonov, Anton **Tchekov**

Dans ce chapitre nous passons en revue les principaux algorithmes de détection du parallélisme : ceux de Lamport (section 2.1), d'Allen et Kennedy (section 2.2), de Wolf et Lam (section 2.3), de Feautrier (section 2.4), et de Lim et Lam (section 2.5). À la section 2.6 nous motivons l'étude que nous mènerons aux chapitres 3 et 4. Dans ce mémoire, nous nous intéresserons surtout aux algorithmes d'Allen et Kennedy, de Wolf et Lam, et de Feautrier. Nous décrivons l'algorithme de Lamport pour son intérêt historique et parce qu'il est beaucoup plus général que l'image qui en est couramment répandue. Nous citons l'algorithme de Lim et Lam par souci d'exhaustivité bien qu'il sorte quelque peu du cadre de notre étude.

2.1 Algorithme de Lamport

Leslie Lamport est connu en parallélisation automatique pour sa « méthode de l'hyperplan ». Dans son article *The parallel execution of DO loops* [55], il a également présenté une « méthode du plan », généralisation de la précédente, et une « méthode coordonnée ». Les deux premières méthodes construisent des ordonnancements généraux quand la troisième ne recherche que des permutations de boucles. Nous ne nous intéresserons donc qu'aux deux premiers algorithmes.

Lamport s'intéresse à des nids de boucles parfaits où les fonctions d'accès aux tableaux sont des fonctions affines multidimensionnelles des indices de boucle. Il suppose que toutes les fonctions d'accès à un même tableau ont la même partie linéaire et que chaque composante de cette partie linéaire est égale à un indice de boucle. Les parties linéaires des fonctions d'accès aux tableaux ne contiennent pas forcément tous les indices de boucle. Ces restrictions n'ont été posées que pour rendre possible, à l'époque, l'analyse des dépendances. En pratique, la « méthode de l'hyperplan » s'applique à tout nid dont les vecteurs de distance sont uniformes, sauf pour leur première composante, et la « méthode du plan » s'applique à une représentation des dépendances par vecteurs de direction.

2.1.1 Méthode de l'hyperplan

Cette méthode, qui repose sur la lexico-positivité des vecteurs de distance, est en fait un algorithme de construction d'ordonnancements linéaires, c'est-à-dire d'*hyperplans* fortement séparateurs. Elle considère des nids dont les dépendances sont représentées par des vecteurs de direction dont seule la première composante peut ne pas être constante.

Nous notons $\mathcal{V}$ l'ensemble des vecteurs de distance[1]. $\mathcal{V}_j$ est le sous-ensemble des vecteurs de $\mathcal{V}$ dont la première composante non nulle est celle de rang j. $\mathcal{V}_j$ est donc l'ensemble des vecteurs de distance de niveau j. L'hyperplan X est construit par récurrence (X_k désigne la $k^{ème}$ composante de l'hyperplan).

Algorithme

- $X = 0$
- $X_n = 1$
- Pour i allant de $n - 1$ à 2 faire :
  - Soit X_i le plus petit entier positif tel que $X\boldsymbol{d} \geq 1$ pour tout élément $\boldsymbol{d}$ de $\mathcal{V}_i$.
- Soit x le plus petit entier strictement positif tel que $x + X\boldsymbol{d} \geq 1$ pour tout élément $\boldsymbol{d}$ de $\mathcal{V}_1$. Nous posons finalement $X_1 = x$.

La preuve de correction de l'algorithme est simple. La première étape et l'induction repose sur le fait que la $i^{ème}$ composante de tout vecteur de $\mathcal{V}_i$ est strictement positive par définition de $\mathcal{V}_i$ et par lexico-positivité des vecteurs de distance. Le calcul de X_1 est valide car la première composante de tout vecteur de $\mathcal{V}_1$ est soit un entier naturel non nul, soit le symbole « + » des vecteurs de direction, et est donc toujours supérieure à un.

Lamport avait remarqué que ses ordonnancements étaient solutions d'un programme linéaire mais avouait ne pas connaître de méthodes pratiques permettant de résoudre ce programme dans le cas général. Son algorithme est optimal, au sens défini à la section 1.4, pour les nids de boucles dont les dépendances sont uniformes, sauf peut être pour la première composante, et dont le graphe de dépendance est fortement connexe.

2.1.2 Méthode du plan

Cette seconde méthode généralise la première. Au lieu de construire un ordonnancement linéaire elle construit un ordonnancement multidimensionnel linéaire, et ce pour des nids parfaits dont les dépendances sont représentées par des vecteurs de direction.

Comme précédemment, $\mathcal{V}$ représente l'ensemble des vecteurs de direction et $\mathcal{V}_j$ son sous-ensemble composé des vecteurs dont la première composante non nulle est la $j^{ème}$. Soit $\mathcal{J} = \{j_1, \ldots, j_m\}$ l'ensemble des rangs des composantes qui ne sont pas constantes pour tous les vecteurs de direction, ensemble auquel a été rajouté « 1 » s'il n'était déjà contenu. En d'autres termes, k appartient à $\mathcal{J}$ si $k = 1$ ou s'il existe un vecteur de direction $\boldsymbol{d} \in \mathcal{V}$ dont la $k^{ème}$ composante n'est pas un entier.

La méthode du plan construit un ordonnancement multidimensionnel linéaire (le corps de boucle est considéré comme étant atomique) de dimension m. La $i^{ème}$ dimension de l'ordonnancement

1. $\mathcal{V}$ ne contient pas les vecteurs nuls, c'est-à-dire les vecteurs de distance correspondant aux dépendances non portées par les boucles. La structure du corps de boucle n'est pas modifiée par les ordonnancements linéaires qui satisfont donc toujours les dépendances indépendantes des boucles.

satisfait les vecteurs appartenant à $\mathcal{V}_{j_i} \cup \ldots \cup \mathcal{V}_{j_{i+1}-1}$ (avec la convention $j_{m+1} = n+1$), c'est-à-dire les vecteurs distance de niveaux compris entre j_i et $j_{i+1} - 1$. Elle est construite par application de la méthode de l'hyperplan à $\mathcal{V}_{j_i} \cup \ldots \cup \mathcal{V}_{j_{i+1}-1}$ en ne considérant que les composantes de rangs compris entre j_i et $j_{i+1} - 1$ (les autres composantes de l'hyperplan construit sont nulles). Nous illustrons cette méthode sur un exemple.

Exemple 4

```
For i = 1 to N do
  For j = 2 to i do
    For k = 7 to N do
      a(i,j,k) = a(i-j,j,k) + a(i-1,j+3,k-7) + a(i,j-2,k+i) + a(i,j,k-1)
```

Cet exemple contient quatre dépendances auxquelles sont associées les vecteurs de direction : $(+,0,0)$, $(1,-3,7)$, $(0,2,*)$ et $(0,0,1)$ d'après l'analyseur Petit [52]. L'ensemble $\mathcal{J}$ vaut donc $\mathcal{J} = \{1,3\}$. La première dimension de l'ordonnancement doit donc satisfaire les vecteurs de distance $(+,0,0)$, $(1,-3,7)$ et $(0,2,*)$. Nous cherchons donc, au moyen de la méthode de l'hyperplan, un vecteur $X = (x,y)$ de dimension 2 et satisfaisant les vecteurs $(+,0)$, $(1,-3)$ et $(0,2)$. Nous commençons par chercher y. y doit vérifier la condition $y \times 2 \geq 1$, d'où $y = 1$. x est défini comme le plus petit entier tel que $x \geq 1$, $x - 3 \geq 1$ et $2 \geq 1$. Nous obtenons $x = 4$. Nous trouvons donc l'hyperplan $(4,1)$ ce qui nous donne le vecteur $(4,1,0)$ comme première dimension de notre ordonnancement. La deuxième dimension doit satisfaire le vecteur $(0,0,1)$. Nous trouvons naturellement l'hyperplan $(0,0,1)$.

Cet algorithme est bien évidemment optimal quand la méthode de l'hyperplan l'est, mais il ne l'est pas dans le cas général (cf. section 2.3). Dans la présentation originelle il n'est pas question de vecteurs de directions mais les symboles « + » et « $*$ » sont utilisés avec la signification que nous leur avons attribuée section 1.2. La méthode de l'hyperplan de Lamport est donc bien plus générale que la simple recherche d'un ordonnancement linéaire pour un nid uniforme.

2.2 Algorithme d'Allen et Kennedy

L'algorithme d'Allen et Kennedy [2, 4] a été originellement proposé pour la vectorisation de boucles. Il a été ultérieurement étendu pour maximiser le nombre de boucles parallèles[2] générées [1] et pour minimiser le nombre de synchronisations requises [3, 62]. Cet algorithme prend en entrée un graphe de dépendance réduit par niveaux (GDRN). Il repose sur les deux propriétés suivantes :

1. Une boucle est parallèle si elle ne porte pas de dépendances, c'est-à-dire s'il n'existe pas de dépendances dont le niveau soit égal à la profondeur de cette boucle et concernant une instruction entourée par cette boucle.

2. Toutes les itérations de l'instruction S peuvent être exécutées avant que ne soit évaluée une seule instance de l'instruction S' si le graphe de dépendance réduit ne contient pas de dépendances de S' vers S.

La première propriété nous permet de marquer une boucle comme étant séquentielle (ForSeq) ou parallèle (ForPar). La seconde suggère que la détection du parallélisme peut être menée indépendamment dans les différentes composantes fortement connexes du GDRN. L'extraction du parallélisme se fait au moyen d'une simple distribution de boucles. Nous présentons l'algorithme au paragraphe 2.2.1 avant d'en étudier l'efficacité au paragraphe 2.2.2.

2. Ce qui ne veut pas dire grand chose comme nous l'avons vu à la section 1.4.

2.2.1 Algorithme

Nous présentons ici une forme simple de l'algorithme d'Allen et Kennedy qui ne se préoccupe que de générer un maximum de boucles parallèles. Pour d'autres versions de cet algorithme, qui ne prennent par exemple en compte qu'une distribution partielle, le lecteur peut se référer aux articles originaux [1, 2, 3, 4] ou au livre de Zima et Chapman [84].

L'appel initial de l'algorithme est de la forme ALLEN-KENNEDY(G, 1) où G est le GDRN du nid à paralléliser.

ALLEN-KENNEDY(G, k)

1. Supprimer de G les arcs de niveau strictement inférieur à k.
2. Calculer les composantes fortement connexes de G.
3. **Pour chaque** composante fortement connexe C de G dans l'ordre topologique **faire** :
   (a) **si** C ne contient qu'une seule instruction et aucun arc **alors** générer des boucles ForPar dans toutes les dimensions restantes, c'est-à-dire du niveau k au niveau n_S, et générer le code de l'instruction S.
   (b) **sinon**
      i. Soit $l = \min_{e \in C} \boldsymbol{w}(e)$.[3]
      ii. Générer des boucles ForPar du niveau k au niveau $l - 1$ et une boucle ForSeq au niveau l.
      iii. Appeler récursivement ALLEN-KENNEDY(C, $l + 1$).

Exemple d'utilisation

Nous illustrons l'algorithme sur l'exemple 5 (figure 2.1). Le nid étudié contient 3 instructions, notées S_1, S_2 et S_3 dans l'ordre textuel. Le premier appel à l'algorithme, ALLEN-KENNEDY(G, 1), détecte deux composantes fortement connexes : $\mathcal{V}_1 = \{S_1\}$ et $\mathcal{V}_2 = \{S_2, S_3\}$. La première, $\mathcal{V}_1$, ne contient aucun arc. Par conséquent, l'algorithme génère deux boucles parallèles entourant S_1. La deuxième composante, $\mathcal{V}_2$, contient un arc de niveau 1. L'algorithme produit donc une boucle séquentielle et s'appelle récursivement sur $\mathcal{V}_2$: ALLEN-KENNEDY($\mathcal{V}_2$, 2). Le code produit jusqu'à présent est présenté à la figure 2.2.

Exemple 5

```
For i=1 to N
  For j=1 to N
    S1 : a(i, j) = i
    S2 : b(i, j) = b(i, j-1) + a(i, j) × c(i-1, j)
    S3 : c(i, j) = 2 × b(i, j) + a(i, j)
```

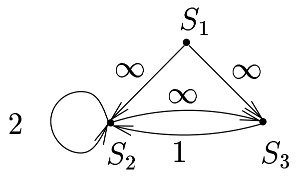


FIG. 2.1: *Exemple 5 et son GDRN.*

3. Nous rappelons que la fonction $\boldsymbol{w}$ associe à un arc d'un graphe de dépendance réduit sa valuation qui est, dans le cas présent, son niveau de dépendance.

Les arcs de niveau strictement inférieur à 2 sont supprimés de G. Le graphe obtenu contient deux composantes fortement connexes : $\mathcal{V}_{2,1} = \{S_2\}$ et $\mathcal{V}_{2,2} = \{S_3\}$, dans l'ordre topologique. La composante $\mathcal{V}_{2,1}$ contient un arc de niveau 2. ALLEN-KENNEDY engendre donc une boucle séquentielle et s'appelle récursivement sur cette composante (par l'invocation ALLEN-KENNEDY($\mathcal{V}_{2,1}$, 3)), ce qui générera le code pour l'instruction S_2. La seconde composante, $\mathcal{V}_{2,2} = \{S_3\}$, ne contient aucun arc. Par conséquent, l'algorithme produit une boucle parallèle autour de S_3 et génère le code de l'instruction. Le code finalement obtenu est celui de la figure 2.3.

```
ForPar i=1 to N
  ForPar j=1 to N
    a(i, j) = i
ForSeq i=1 to N
  ALLEN-KENNEDY(V2, 2)
```

FIG. 2.2: *Résultat du premier appel.*

```
ForPar i=1 to N
  ForPar j=1 to N
    a(i, j) = i
ForSeq i=1 to N
  ForSeq j=1 to N
    b(i, j) = b(i, j-1) + a(i, j) × c(i-1, j)
  ForPar j=1 to N
    c(i, j) = 2 × b(i, j) + a(i, j)
```

FIG. 2.3: *Code parallélisé.*

2.2.2 Optimalité

Il a été prouvé que pour chaque instruction du code originel l'algorithme ALLEN-KENNEDY détectait autant de boucles englobantes parallèles que possible (cf. [17, 84]). Cependant, les détails de la démonstration montrent que la propriété prouvée est en fait : « soient une instruction S et une boucle L_i l'englobant, alors L_i est marquée parallèle si et seulement si il n'y a pas de dépendances au niveau i entre deux instances de S ». La preuve établit donc seulement que l'algorithme d'Allen et Kennedy est optimal parmi les algorithmes de parallélisation qui génèrent un code parallélisé dans lequel les instances d'une instruction quelconque sont énumérées avec les mêmes boucles que dans le code initial, à l'étiquetage « parallèle » ou « séquentielle » près. Nous sommes donc loin de l'optimalité définie à la section 1.4. La démonstration de Callahan ne prouve pas qu'il est impossible de trouver plus de parallélisme dans les graphes de dépendance réduits par niveaux en appliquant des techniques plus sophistiquées que la distribution et la fusion de boucles. Pourtant, ALLEN-KENNEDY est optimal au sens fort, ce que nous prouvons en utilisant la technique des nids apparents (cf. section 1.4). Autrement dit, **aucun** algorithme ne peut trouver plus de parallélisme dans un graphe de dépendance réduit par niveaux que n'en trouve l'algorithme d'Allen et Kennedy.

Nid de boucles apparent

Considérons un nid de boucles $\mathcal{L}$. Nous présentons ici une méthode de construction d'un nid de boucles apparent $\mathcal{L}$', c'est-à-dire d'un nid de boucles de même graphe de dépendance réduit et parallélisé optimalement (cf. section 1.4). L'algorithme GÉNÉRATION_DU_NID_APPARENT prend en entrée un GDRN G et construit un nid de boucles parfait $\mathcal{L}$' de GDRN G.

Soit $G = (E, V)$ un GDRN. Nous supposons que G a été construit de manière cohérente à partir d'un nid de boucles. Par conséquent, ses nœuds peuvent être numérotés par l'ordre topologique défini par les arcs de niveau infini, c'est-à-dire par les arcs correspondant à des dépendances indépendantes des boucles. Nous numérotons donc les nœuds en respectant la loi : $v_i \xrightarrow{\infty} v_j \Rightarrow i < j$. Nous notons d la **dimension** de G : $d = \max\{\boldsymbol{w}(e) \mid e \in E \text{ et } \boldsymbol{w}(e) < \infty\}$.

Le nid apparent $\mathcal{L}$' correspondant à G est formé de d boucles parfaitement imbriquées englobant $|V|$ instructions notées $T_1, \ldots, T_{|V|}$. Chaque instruction est de la forme $\mathsf{a}_i[\boldsymbol{k}] = \mathsf{rhs}(i)$ où a_i est un tableau de dimension d, $\mathsf{rhs}(i)$ est le membre droit qui produit les valeurs stockées dans le tableau a_i et $\boldsymbol{k}$ le vecteur d'itération.

Durant les appels récursifs à ALLEN-KENNEDY, nous sélectionnons des arcs que nous appelons **critiques** et qui seront traités spécialement lors de la génération du nid apparent. Toutes les dépendances du nid $\mathcal{L}$' seront uniformes, sauf celles qui correspondront à ces arcs critiques. La sélection de ces arcs se fait à l'étape 3(b)i de ALLEN-KENNEDY (cf. section 2.2.1) : un des arcs de niveau $l = \min_{e \in C} \boldsymbol{w}(e)$ est arbitrairement élu « arc critique ». Dans la suite, E_c désignera l'ensemble des arcs critiques de G et "@" représentera l'opérateur de concaténation d'expressions.

GÉNÉRATION_DU_NID_APPARENT(G)

Initialisation :
Pour $i = 1$ **à** $|V|$ **faire** $\mathsf{rhs}(i) \leftarrow$ "1"

Calcul des instructions de $\mathcal{L}$' :
Pour chaque arc $e = (v_i, v_j) \in E$ **faire**
si $l(e) = \infty$ **alors** $\mathsf{rhs}(j) \leftarrow \mathsf{rhs}(j)$ @ "$+\mathsf{a}_i[\boldsymbol{k}_1, \ldots, \boldsymbol{k}_d]$"
si $l(e) < \infty$ et $e \notin E_c$ **alors** $\mathsf{rhs}(j) \leftarrow \mathsf{rhs}(j)$ @ "$+\mathsf{a}_i[\boldsymbol{k}_1, \ldots, \boldsymbol{k}_{l(e)-1}, \boldsymbol{k}_{l(e)} - 1, \boldsymbol{k}_{l(e)+1}, \ldots, \boldsymbol{k}_d]$"
si $e \in E_c$ **alors** $\mathsf{rhs}(j) \leftarrow \mathsf{rhs}(j)$ @ "$+\mathsf{a}_i[\boldsymbol{k}_1, \ldots, \boldsymbol{k}_{l(e)-1}, \boldsymbol{k}_{l(e)} - 1, \underbrace{N, \ldots, N}_{d-l(e)}]$"

Génération du code de $\mathcal{L}$' :
Pour $i = 1$ **à** d **faire** générer("**For** $\boldsymbol{k}_i = 1$ **to** N **do**")
Pour $i = 1$ **à** $|V|$ **faire** générer("$\mathsf{a}_i[\boldsymbol{k}_1, \ldots, \boldsymbol{k}_d] =$" @ $\mathsf{rhs}(i)$)

Lemme 2. *G est le graphe de dépendance réduit par niveaux de $\mathcal{L}$'.*

DÉMONSTRATION. Soit G' le GDRN associé à $\mathcal{L}$'. Dans $\mathcal{L}$' il y a une unique *écriture* pour chaque valeur du vecteur d'itération $\boldsymbol{k}$ et pour chaque tableau a_i : cette *écriture* a lieu à l'itération $\boldsymbol{i}$ de T_i. Par conséquent, les dépendances de $\mathcal{L}$' impliquant le tableau a_i correspondent à des dépendances entre cette unique *écriture* et des *lectures* dans ce tableau. Chaque *lecture* du tableau a_i dans le membre droit d'une instruction correspond, par construction de $\mathcal{L}$', à un arc particulier e du graphe G. Par conséquent, les graphes G et G' ont mêmes nœuds et mêmes arcs. Il reste à vérifier que les niveaux de dépendance des arcs de G' correspondent à ceux de G, ce qui est évident. □

Retour sur l'exemple 5. Le graphe de dépendance réduit par niveaux de l'exemple 5 est présenté à la figure 2.1. Il comporte deux composantes fortement connexes. La première ne contient aucun arc. La seconde, $\mathcal{V}_2 = \{S_2, S_3\}$, est de niveau 1, et ne contient qu'un seul arc de ce niveau, l'arc e de S_3 vers S_2, qui est donc marqué « critique ». Cet arc générera donc un accès en lecture c(i-1, N) dans le membre droit de l'instruction T_2. L'arc e éliminé (lors de l'appel récursif), la composante $\mathcal{V}_2$ est cassée en deux composantes fortement connexes, l'une contenant S_2 et l'autre S_3. L'auto-dépendance de S_2 est de niveau 2 et est également sélectionnée comme étant critique puisqu'elle est le seul arc restant. Elle génère une lecture b(i, j-1) dans le membre droit de T_2. Finalement, le nid de boucles de l'exemple 5 est représenté figure 2.4. Le lecteur peut vérifier sur la figure 2.4 que le graphe de dépendance étendu du nid apparent contient bien un chemin de S_1-longueur $\Omega(1)$, un chemin de S_2-longueur $\Omega(N^2)$ et un chemin de S_3-longueur $\Omega(N)$, et que le nid apparent contient donc uniquement le parallélisme qu'y trouve ALLEN-KENNEDY. Les arcs en gras forment simultanément ces trois chemins.

```
For i=1 to N
  For j=1 to N
    a(i, j) = 1
    b(i, j) = 1 + b(i, j-1) + a(i, j) + c(i-1, N)
    c(i, j) = 1 + b(i, j) + a(i, j)
```

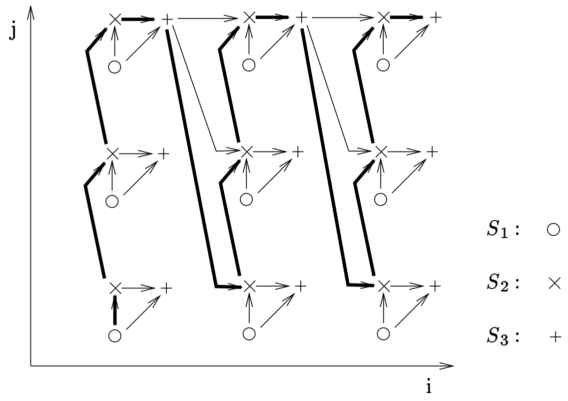


FIG. 2.4: *Nid apparent pour l'exemple 5 et son GDE pour $N = 3$.*

Nous notons d_S le nombre d'invocations de ALLEN-KENNEDY (l'appel terminal excepté) qui concernent l'instruction S. d_S est donc le nombre d'appels ALLEN-KENNEDY(H, k) pour lesquels S appartient à une composante fortement connexe de H contenant au moins un arc. Une boucle séquentielle englobant S est générée par chacun de ces appels et seul ce type d'appels en génère. d_S est donc également le nombre de boucles séquentielles qui entourent S dans le code parallélisé. $n_S - d_S$ est donc le S-degré de parallélisme extrait, si le domaine d'itération $\mathcal{D}_S$ est supposé contenir un hypercube $\underline{\mathcal{D}}$ de dimension n_S et de taille $\Omega(N)$ et être contenu dans un hypercube $\overline{\mathcal{D}}$ de dimension n_S et de taille $O(N)$ (cf. lemme 1). Nous avons prouvé le résultat suivant [32, 33] :

Théorème 1. *Soit un nid de boucles $\mathcal{L}$ et son graphe de dépendance réduit par niveaux G. Soit $\mathcal{L}$' le nid de boucles apparent généré par* GÉNÉRATION_DU_NID_APPARENT *à partir de G. Pour chaque composante fortement connexe G_i de G il existe un chemin dans le graphe de dépendance étendu du nid apparent $\mathcal{L}$' qui visite $\Omega(N^{d_S})$ fois chaque nœud S de G_i.*

La démonstration est longue, technique et indigeste. Nous n'avons pas jugé utile de la rappeler ici. La construction du chemin est similaire dans sa structure à celle utilisée pour démontrer le théorème 4 (paragraphe 3.2.2). Le théorème nous permet d'établir l'optimalité de ALLEN-KENNEDY :

Corollaire 1. *L'algorithme d'Allen et Kennedy est optimal, au sens établi à la section 1.4, pour l'extraction du parallélisme dans les graphes de dépendance réduits par niveaux.*

DÉMONSTRATION. Soit G le GDRN d'un nid de boucles $\mathcal{L}$ de profondeur n. $n - d_S$ est le S-degré de parallélisme extrait de G. L'algorithme GÉNÉRATION_DU_NID_APPARENT produit un nid de boucles parfait $\mathcal{L}$' de profondeur d dont le GDRN est exactement G (lemme 2). De plus, pour chaque composante fortement connexe G_i de G, il existe un chemin dans le GDE de $\mathcal{L}$' qui contient $\Omega(N^{d_S})$ occurrences de toute instruction S de G_i (théorème 1). Si $d = n$, le corollaire est démontré.

Il est possible que d soit strictement inférieur à n : $d < n$. Dans ce cas, nous définissons un nid $\mathcal{L}$" de profondeur n à partir du nid $\mathcal{L}$' de profondeur d, en ajoutant simplement à $\mathcal{L}$' $n - d$ boucles internes (englobées par les d boucles existantes) et nous complétons tous les accès aux tableaux par l'expression : $[\boldsymbol{k}_{d+1}, \ldots, \boldsymbol{k}_n]$. Le GDRN reste inchangé toutes les dépendances portées par des boucles l'étant par une des d boucles les plus externes, celles communes avec $\mathcal{L}$'. De fait, les $n - d$ boucles les plus internes sont parallèles et le chemin construit dans le GDE de $\mathcal{L}$' par le théorème 1 peut être immédiatement traduit en un chemin de même structure du GDE de $\mathcal{L}$" en fixant à une même valeur les $n - d$ dernières composantes de tous les vecteurs d'itérations considérés (le GDE de $\mathcal{L}$' est la projection du GDE de $\mathcal{L}$" parallèlement aux $n - d$ dernières dimensions). □

En résumé, nous prouvons que pour tout nid $\mathcal{N}_1$ il existe un nid $\mathcal{N}_2$, de même graphe de dépendance réduit par niveaux, tel que pour toute instruction S de $\mathcal{N}_1$ entourée après parallélisation par d_S boucles séquentielles, il existe un chemin de dépendance dans $\mathcal{N}_2$ incluant $\Omega(N^{d_S})$ instances de S. En d'autres termes, l'algorithme d'Allen et Kennedy ne peut distinguer $\mathcal{N}_1$ de $\mathcal{N}_2$ puisqu'ils ont même GDRN et $\mathcal{N}_2$ est parallélisé optimalement au sens fort (cf. section 1.4), ALLEN-KENNEDY exhibant pour chaque instruction autant de parallélisme que le permet la borne supérieure définie par la longueur des plus longs chemins de dépendance.

Par conséquent, si nous ne connaissons d'un nid que son GDRN, nous ne pouvons pas y détecter et en extraire plus de parallélisme que l'algorithme d'Allen et Kennedy. Cet algorithme est donc parfaitement adapté à une représentation des dépendances par niveaux. Pour détecter dans un nid plus de parallélisme que ALLEN-KENNEDY il nous faut donc plus d'information. Si une représentation plus précise des dépendances est fournie, il est possible de battre ALLEN-KENNEDY sur des exemples où une simple interversion de boucles (exemple 6, figure 2.5) ou une torsion de boucle suivie d'une interversion (exemple 7, figure 2.6) suffisent à exhiber un parallélisme maximal.

Exemple 6

```
For i=1,n
  For j=1,n
    a(i, j) = a(i-1, j-1) + a(i, j-1)
```

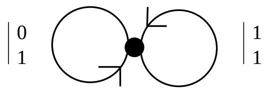


FIG. 2.5: *Exemple 6 et son graphe de dépendance réduit.*

Exemple 7

```
For i=1,n
  For j=1,n
    a(i, j) = a(i-1, j) + a(i, j-1)
```

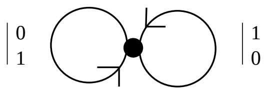


FIG. 2.6: *Exemple 7 et son graphe de dépendance réduit.*

Nous pouvons nous demander s'il est possible de détecter plus de parallélisme si la structure du code est mieux connue, par exemple si nous savons comment les boucles sont imbriquées (en ne supposant aucune connaissance supplémentaire sur les bornes de boucles). En fait, ce n'est pas possible : nous pouvons obliger $\mathcal{L}$' à avoir la même structure que $\mathcal{L}$. L'algorithme de génération du nid est similaire à GÉNÉRATION_DU_NID_APPARENT aux modifications suivantes près :

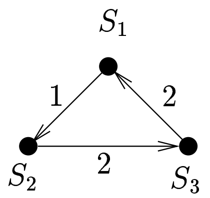
- Le membre gauche de l'instruction T_i est $\mathsf{a}_i(\boldsymbol{k})$ où $\boldsymbol{k}$ est le vecteur d'itération qui correspond aux boucles qui englobent S_i dans le code originel $\mathcal{L}$. Par conséquent, la dimension du tableau associé à l'instruction T_i est égale aux nombres de boucles entourant l'instruction.
- Le membre droit de l'instruction T_i est défini comme dans GÉNÉRATION_DU_NID_APPARENT, sauf que les vecteurs d'itérations sont tronqués ou complétés par des composantes égales à N pour avoir la taille requise.

Le théorème 1 peut alors être généralisé aux nids non parfaits, avec une preuve similaire. Nous ne rentrerons pas plus avant dans les détails le cas parfait étant déjà suffisamment compliqué.

Par contre, il est possible de trouver plus de parallélisme si le domaine est connu comme le montre l'exemple 8. Le GDRN du nid est présenté figure 2.7. Nous n'avons pas spécifié le code

Exemple 8

```
For i = 1 to N do
  For j = 1 to 2 do
    S1(i, j)
    S2(i, j)
    S3(i, j)
```




```
For i = 1 to N do
  T1 = S1(i, 1)
  T2 = S2(i, 1)
  T3 = S3(i, 1)
  T4 = S1(i, 2)
  T5 = S2(i, 2)
  T6 = S3(i, 2)
```

FIG. 2.7: *Exemple 8 : code originel et après déroulement de la boucle interne.*

de l'exemple pour montrer sa généralité. Le GDRN contient un unique circuit qui est de niveau un. ALLEN-KENNEDY trouve donc que la boucle externe est séquentielle et que la boucle interne est parallèle, ce qui révèle en pratique peu de parallélisme puisque la boucle parallèle ne décrit que deux itérations. Connaissant cette propriété du nid nous pouvons dérouler la boucle interne et obtenir le deuxième nid présenté figure 2.7. Considérons ses dépendances.

- De S_1 vers S_2. Dans le nid initial, il y a une dépendance de niveau un de S_1 vers S_2, donc S_2(i', j') dépend de S_1(i, j) pour i' > i et j' quelconque. Dans le nid déroulé il y a donc des dépendances de niveau un de T_1 vers T_2 et vers T_3, et de T_4 vers T_2 et vers T_3.

- De S_2 vers S_3. Dans le nid initial, il y a une dépendance de niveau deux de S_2 vers S_3, donc S_3(i, j') dépend de S_2(i, j) pour j' > j, c'est-à-dire pour j' = 2 et j = 1. Dans le nid déroulé il y a donc une dépendance de niveau infini de T_2 vers T_6.

- De S_3 vers S_1. Avec un raisonnement similaire à celui du cas précédent, nous montrons qu'il y a une dépendance de niveau infini de T_3 vers T_4.

Le nouveau graphe de dépendance est donc sans circuits et ALLEN-KENNEDY trouve que l'unique boucle est parallèle. Dans notre définition de l'optimalité nous supposons que le domaine d'itération est inconnu et que les algorithmes étudiés cherchent du parallélisme indépendant du domaine. C'est pourquoi nous considérons que ALLEN-KENNEDY est optimal alors qu'il est loin de trouver tout le parallélisme contenu dans cet exemple, même quand les dépendances sont représentées par niveaux.

2.3 Algorithme de Wolf et Lam

Les exemples 6 et 7 contiennent du parallélisme, même si ALLEN-KENNEDY est incapable de le détecter. Le corollaire 1 affirme que ce parallélisme ne peut être découvert si les dépendances sont représentées par niveaux. Pour contourner ce problème, Michael Wolf et Monica Lam ont proposé un algorithme qui accepte en entrée une approximation des dépendances par vecteurs de direction [77]. Leurs travaux unifient les algorithmes antérieurs basés sur des opérations matricielles élémentaires, telles que la torsion (*skew* en anglais), l'interversion (*interchange*) et le renversement (*reversal*) de boucles, en se plaçant dans le cadre unique des *transformations unimodulaires* valides. L'algorithme de Wolf et Lam généralise la « méthode du plan » de Lamport (section 2.1) qui parallélise optimalement les exemples 6 et 7.

2.3.1 Objectif

Wolf et Lam ont pour objectif de construire des ensembles de boucles totalement permutables. Ces boucles totalement permutables sont à la base de toutes les techniques de partitionnement (*tiling* en anglais) [50, 73, 77, 12, 20]. Le partitionnement est utilisé pour exposer du parallélisme à grain moyen. Cependant, un ensemble de d boucles imbriquées totalement permutables peut toujours être transformé en un ensemble de $d-1$ boucles parallèles imbriquées entourées d'une boucle séquentielle. Les boucles permutables sont donc aussi liées à la parallélisation à grain fin.

Wolf et Lam cherchent, au moyen de transformations unimodulaires, le plus grand[4] ensemble de boucles externes totalement permutables[5]. Ils considèrent récursivement les dimensions restantes et les vecteurs non satisfaits par ces boucles. La version de l'algorithme originellement présentée [77] construit l'ensemble de boucles permutables via une étude de cas simples. Elle repose sur une heuristique pour les nids de profondeur supérieure ou égale à cinq. Dans le reste de cette section, nous présentons l'algorithme d'un point de vue théorique et nous en proposons une version générale.

Pour que des boucles soient permutables, il faut qu'elles soient imbriquées. Wolf et Lam ne considèrent donc que des ensembles de boucles parfaitement imbriquées.

2.3.2 Interprétation théorique

Les transformations unimodulaires ont deux avantages principaux : la linéarité et l'inversibilité dans les entiers. Étant donnée une transformation unimodulaire T, sa linéarité permet de tester simplement sa validité. En effet, dans le cadre des nids à incréments positifs, une transformation est valide si elle transforme tout vecteur de distance en un vecteur lexicographiquement positif. La transformation unimodulaire T est donc valide si et seulement si $T\boldsymbol{d} \succ 0$ pour tout vecteur de distance $\boldsymbol{d}$ non nul[6]. L'inversibilité facilite la réécriture du code, la transformation devenant un simple changement de base dans $\mathbb{Z}^n$.

Il n'est en général pas possible de tester la véracité de l'inégalité $T\boldsymbol{d} \succ 0$ pour tous les vecteurs de distance, car ils sont trop nombreux. Le paralléliseur se contente donc de vérifier cette propriété pour tous les vecteurs de direction non nuls, avec l'arithmétique usuelle sur $\mathbb{Z} \cup \{*\} \cup (\mathbb{Z} \times \{+,-\})$. Dans la suite, nous ne considérons que les vecteurs de direction non nuls. Ces vecteurs sont lexicographiquement positifs[7].

Nous notons $T_1, \ldots, T_n$ les lignes de T et Γ la fermeture[8] du cône engendré par l'ensemble des vecteurs de direction. Pour tout vecteur de direction $\boldsymbol{d}$:

$$T\boldsymbol{d} \succ 0 \;\Leftrightarrow\; \exists k_{\boldsymbol{d}},\ 1 \leq k_{\boldsymbol{d}} \leq n \mid \forall i,\ 1 \leq i < k_{\boldsymbol{d}},\ T_i\boldsymbol{d} = 0 \text{ et } T_{k_{\boldsymbol{d}}}\boldsymbol{d} > 0,$$

ce qui signifie que les dépendances représentées par $\boldsymbol{d}$ sont portées par une boucle de profondeur $k_{\boldsymbol{d}}$. Considérons le cône Γ^+ des hyperplans faiblement séparateurs : $\Gamma^+ = \{Y \mid \forall X \in \Gamma,\ YX \geq 0\}$. Si Γ^+ est de dimension pleine, ici n, nous pouvons construire T à partir de n vecteurs linéairement indépendants de Γ^+ et transformer les boucles originelles en n boucles totalement permutables. De plus, si Γ^+ est de dimension pleine, Γ est un cône pointé (il ne contient aucun espace affine de

4. Nous pouvons compter sans risque le nombre de boucles car les transformations unimodulaires ne modifient pas le nombre de boucles d'un nid.

5. Dans notre cadre de boucles dont tous les incréments sont positifs, les k boucles externes sont totalement permutables si les k premières composantes de tous les vecteurs de direction sont positives.

6. Les transformations unimodulaires ne modifient pas la structure du corps de boucle qui est en quelque sorte considéré comme étant atomique. Les dépendances indépendantes des boucles sont donc automatiquement satisfaites par toutes les transformations unimodulaires, qu'elles soient ou non valides, et nous n'avons pas à nous en préoccuper.

7. Nous supposons donc ici que l'analyse de dépendance est bien faite. Par exemple, nous supposons qu'elle ne nous renverra jamais le vecteur de direction $(0+,*)$ qui contient des distances non lexicographiquement positives et dont l'ensemble des valeurs légales est exactement représenté par le couple de vecteurs : $\{(+,*),(0,0+)\}$.

8. La fermeture d'un ensemble E est l'ensemble des points qui sont limites d'une suite de points de E. Dans notre cas, prenons le vecteur de direction $(1,*)$. Il engendre le cône $\gamma = \{(x,y) \mid x > 0 \text{ ou } x = y = 0\}$. La fermeture de ce cône vaut alors $\Gamma = \{(x,y) \mid x \geq 0\}$.

dimension non nulle, cf. [74] pour plus de détails sur les cônes et les notions adjacentes). Dans ce cas il existe un vecteur T_1 tel que $k_{\boldsymbol{d}} = 1$ pour tout vecteur de direction $\boldsymbol{d}$. Toutes les dépendances portées par une boucle le sont alors par la boucle la plus externe et toutes les boucles internes sont parallèles. T_1 est appelé **vecteur de temps** ou **hyperplan fortement séparateur**.

La notion de vecteur de temps, ou d'ordonnancement, est au cœur de la méthode de l'hyperplan (section 2.1) et de ses variantes et extensions (l'ordonnancement des systèmes d'équations récurrentes uniformes [27], l'algorithme de Wolf et Lam, celui de Feautrier présenté à la section 2.4, l'algorithme des chapitres 3 et 4, etc.). Elle est tout particulièrement intéressante pour l'expression du parallélisme à grain fin. La notion de boucles totalement permutables est quant à elle la base des techniques de partitionnement. Dans le cas des transformations unimodulaires, ces deux notions sont donc reliées par le cône Γ^+ et ses propriétés.

Quand Γ n'est pas un cône pointé, Γ^+ est de dimension r, $1 \leq r < n$. $r = n - s$ où s est la dimension de l'« espace linéaire »[9] de Γ. À l'aide de r vecteurs de Γ^+ linéairement indépendants, nous pouvons transformer le nid initial en un nid aux r boucles externes totalement permutables. La transformation est récursivement appliquée aux $n-r$ boucles les plus internes en ne considérant que les vecteurs de direction non portés par une des r boucles les plus externes, c'est-à-dire en ne considérant que les vecteurs de direction inclus dans l'espace linéaire de Γ. Tel est le principe de l'algorithme de Wolf et Lam, même s'ils ne l'ont pas exprimé en ces termes dans leur article [77].

2.3.3 Version générale de l'algorithme

La discussion théorique est résumée par l'algorithme WOLF-LAM ci-dessous. Cet algorithme prend en entrée un ensemble D de vecteurs de direction et une suite E de vecteurs indépendants et renvoie une séquence E' de vecteurs indépendants. La matrice de transformation est construite à partir de l'ensemble E' : le $i^{ème}$ vecteur-ligne de la matrice est le $i^{ème}$ vecteur de la suite E'. L'appel initial est WOLF-LAM $(\Delta, \emptyset)$ où Δ est l'ensemble de tous les vecteurs de direction.

WOLF-LAM (D, E)

1. Soit Γ la fermeture du cône généré par les vecteurs de direction de D.

2. $\Gamma^+ = \{\mathbf{y} \mid \forall \mathbf{x} \in \Gamma,\ \mathbf{yx} \geq 0\}$. Soit r sa dimension.

3. Compléter E en un ensemble E' de r vecteurs de Γ^+ linéairement indépendants (nous avons toujours $E \subset \Gamma^+$ par construction).

4. Soit D' le sous-ensemble de D défini par : $\boldsymbol{d} \in D' \Leftrightarrow \forall v \in E',\ v\boldsymbol{d} = 0$ (c'est-à-dire $D' = D \cap E'^{\perp} = D \cap \text{espace.linéaire}(\Gamma)$)[10].

5. Si $D' = \emptyset$ renvoyer E' sinon appeler récursivement WOLF-LAM (D', E').

9. Traduction pour le moins légère de l'expression « lineality space » qui dénote l'espace vectoriel associé au plus grand espace affine inclus dans le cône. Si $\Gamma = \{x \mid Ax \leq b\}$ alors l'espace linéaire de Γ est $\{x \mid Ax = 0\}$ (cf. [74]).

10. Nous montrons que $E'^{\perp} = \Gamma^{+\perp}$ est égal à l'espace linéaire de Γ. Soient $\boldsymbol{d}$ un point de l'espace linéaire de Γ et y un point de Γ. Alors, pour tout rationnel λ, $y + \lambda\boldsymbol{d} \in \Gamma$ et, si $x \in \Gamma^+$, $x(y + \lambda\boldsymbol{d}) \geq 0$. $x\boldsymbol{d}$ est donc forcément nul et $\boldsymbol{d} \in \Gamma^{+\perp}$. Réciproquement, nous écrivons le cône Γ sous forme matricielle : $\Gamma = \{x \mid Ax \leq 0\}$. Les opposés des vecteurs-lignes de A appartiennent donc à Γ^+. Donc, si $\boldsymbol{d}$ est un point de $\Gamma^{+\perp}$, $A\boldsymbol{d} = 0$ et $\boldsymbol{d}$ appartient à l'espace linéaire de Γ par définition de l'espace linéaire d'un polyèdre (cf. [74]).

La partie heuristique de l'algorithme originellement présenté par Wolf et Lam concerne le calcul de Γ et de $\Gamma+$ (étapes 1 et 2), calcul que nous n'explicitons pas. L'étape 3 nous oblige à vérifier que E' est bien inclus dans $\Gamma^+(D')$, or $E' \subset \Gamma^+(D) \subset \Gamma^+(D')$.

L'algorithme peut en fait produire une matrice non unimodulaire. Pour réellement obtenir une matrice unimodulaire, nous procédons comme suit :

- Soit Δ l'ensemble des vecteurs de direction. Invoquer WOLF-LAM $(\Delta, \emptyset)$.
- Construire une matrice inversible T_1 dont les premières lignes sont les vecteurs de E' pris dans leur ordre de production par WOLF-LAM. Soit $T_2 = pT_1^{-1}$, où p est choisi de sorte que T_2 soit entière.
- Calculer la forme de Hermite gauche de T_2, $T_2 = QH$, où Q est unimodulaire et H triangulaire inférieure positive.
- Q^{-1} est la matrice de transformation attendue, car si $\boldsymbol{d}$ est un vecteur de distance, $T_1\boldsymbol{d}$ est lexicographiquement positif et, H étant triangulaire inférieure positive, $pQ^{-1}\boldsymbol{d} = HT_1\boldsymbol{d} \succcurlyeq 0$.

Exemple d'utilisation

Exemple 9

```
For i=1,n
  For j=1,n
    For k=1,n
      a(i, j, k) = a(i-1, j+i, k) + a(i, j, k-1) + a(i, j-1, k+1)
```

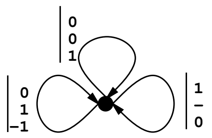


FIG. 2.8: *Exemple 9 et son graphe de dépendance réduit.*

Nous illustrons l'algorithme avec l'exemple 9 (figure 2.8). L'ensemble des vecteurs de direction est $\Delta = \{(1,-,0),(0,0,1),(0,1,-1)\}$. $\Gamma(\Delta) = \{(x,y,z) \mid x \geq 0\}$ et l'espace linéaire de $\Gamma(\Delta)$ est bidimensionnel (généré par $(0,1,0)$ et $(0,0,1)$). Par conséquent, $\Gamma^+(\Delta)$ est monodimensionnel et engendré par $E_1 = \{(1,0,0)\}$. Pour trouver directement un hyperplan faiblement séparateur $X = (x,y,z)$, nous aurions dû résoudre le système : x+«−»$y \geq 0, z \geq 0, y - z \geq 0$. De la première inéquation nous obtenons que y doit être négatif, des deux dernières inéquations nous concluons qu'il doit être positif. Nous trouvons finalement $x \geq 0, y = z = 0$. $D' = \{(0,0,1),(0,1,-1)\}$ et $\Gamma(D')$ est pointé. Nous complétons E_1 par deux vecteurs de $\Gamma^+(D')$, par exemple $E_2 = \{(0,1,0),(0,1,1)\}$. Pour notre exemple, la matrice de transformation, dont les premières lignes sont les vecteurs de E_1 et les autres les vecteurs de E_2, est déjà unimodulaire et correspond à une simple torsion. Pour exposer une boucle ForPar, nous choisissons le premier vecteur de E_2 dans l'« intérieur relatif »[11] de Γ^+ : nous voulons en effet un vecteur X tel que $X \in \Gamma^+, \forall \boldsymbol{d} \in \Gamma, X\boldsymbol{d} > 0$. Nous prenons par exemple $E_2 = \{(0,2,1),(0,1,0)\}$. En termes de transformations de boucles, ce choix équivaut à appliquer à la troisième boucle une torsion de facteur 2 puis à intervertir les deux dernières boucles. Nous obtenons de la sorte le code suivant :

11. L'intérieur relatif d'un polyèdre est l'ensemble de ses points dont l'intersection d'un voisinage avec l'enveloppe affine du polyèdre est incluse dans le polyèdre [74].

```
ForSeq i=1 to n do
  ForSeq k=3 to 3n do
    ForPar j=max(1, ⌈(k-n)/2⌉) to min(n, ⌊(k-1)/2⌋) do
      a(i, j, k-2j) = a(i-1, j+i, k-2j) + a(i, j, k-2j-1) + a(i, j-1, k-2j+1)
```

2.3.4 Optimalité

Wolf et Lam ont montré que leur méthodologie était optimale (Théorème B.6 in [77]) : « un algorithme qui trouve le maximum de parallélisme à gros grain et s'appelle récursivement sur les boucles internes produit le degré maximal possible de parallélisme ». Étonnamment, ils n'ont besoin d'aucune hypothèse pour établir ce théorème. Si nous nous reportons à la démonstration, nous nous rendons compte que l'optimalité de l'algorithme est une optimalité *parmi les algorithmes ne produisant que des transformations unimodulaires*. De plus, elle doit être comprise encore une fois « pour la représentation des dépendances choisie », c'est-à-dire pour des dépendances approchées par des vecteurs de direction mais sans qu'aucune information sur la structure du graphe de dépendance ne soit connue. Nous avons donc affaire à une optimalité au sein d'une classe d'algorithmes et pour une représentation des dépendances donnée. Il est en fait possible de démontrer l'optimalité de WOLF-LAM en utilisant notre définition :

Théorème 2. *L'algorithme* WOLF-LAM *est optimal parmi tous les algorithmes de détection du parallélisme qui prennent en entrée un ensemble de vecteurs de direction (considérant implicitement que le nid contient une seule instruction ou que le corps de boucle est atomique).*

Nous présentons une preuve indirecte de ce théorème (section 4.5). Comme pour ALLEN-KENNEDY, la sous-optimalité de WOLF-LAM dans le cas général n'est pas tant due à la méthodologie mise en jeu qu'à la faiblesse de la représentation des dépendances étudiée : bien sûr les vecteurs de direction ne sont qu'une approximation des dépendances exacte mais, surtout, ne pas utiliser la structure du graphe de dépendance réduit peut induire une perte de parallélisme. Nous présentons ci-dessous deux exemples montrant les faiblesse de WOLF-LAM. Pour le premier nid, il suffit de rajouter une étape de distribution de boucles dans l'algorithme pour le rendre optimal. ALLEN-KENNEDY et WOLF-LAM sont sous-optimaux pour le deuxième nid, même quand nous effectuons des distributions de boucles au sein de WOLF-LAM. Nous avons donc clairement besoin de techniques plus puissantes pour détecter efficacement le parallélisme contenu dans les graphes de dépendance réduits étiquetés par des vecteurs de direction.

Exemple 10 (où WOLF-LAM a besoin de distribution de boucles)

Contrairement à l'algorithme d'Allen et Kennedy, celui de Wolf et Lam est incapable de trouver le moindre parallélisme dans cet exemple, présenté figure 2.9, à cause des vecteurs de direction $(1, -, 0)$, $(0, 1, -)$ et $(0, 0, 1)$. Il suffit de rajouter une étape de distribution de boucles avant l'appel récursif pour pouvoir trouver du parallélisme. Comme précédemment, le premier ensemble de boucles totalement permutables ne contient que la boucle externe. Cette boucle satisfait une auto-dépendance de S_1 et la dépendance de S_1 vers S_2. L'ensemble des arcs non satisfaits engendre un graphe contenant deux composantes fortement connexes : la première contient S_1 et le vecteur $(0, 0, 1)$, la deuxième contient S_2 et le vecteur $(0, 1, -)$. Dans chacune des deux composantes, WOLF-LAM trouve ensuite que les deux boucles restantes peuvent être rendues totalement permutables.

Exemple 11 (où ALLEN-KENNEDY et WOLF-LAM sont sous-optimaux)

L'exemple est présenté figure 2.10 avec son graphe de dépendance réduit valué par des vecteurs de direction. Dans cet exemple, même la version étendue de WOLF-LAM utilisant la distribution de boucles ne trouve pas plus d'un degré de parallélisme, le graphe de dépendance restant fortement

```
For i=1,n
 For j=1,n
  For k=1,n
   S1 : a(i, j, k) = a(i-1, j+i, k) + a(i, j, k-1) + b(i, j-1, k)
   S2 : b(i, j, k) = b(i, j-1, k+j) + a(i-1, j, k)
```

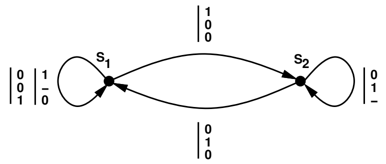


FIG. 2.9: *Exemple 10 et son graphe de dépendance réduit.*

```
For i = 1 to n do
  For j = 1 to n do
    For k = 1 to n do
      S1: a(i, j, k) = b(i-1, j+i, k) + b(i, j-1, k+2)
      S2: b(i, j, k) = a(i, j-1, k+j) + a(i, j, k-1)
```

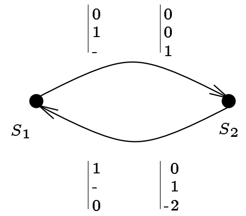


FIG. 2.10: *Exemple 11 et son graphe de dépendance réduit.*

connexe après le calcul du premier ensemble de boucles externes totalement permutables. WOLF-LAM est gêné par les deux dépendances ayant une composante égale à « $-$ » et le premier ensemble de boucles permutables ne contient que la boucle externe. Une seule dépendance est portée par cette boucle et le graphe résultant reste fortement connexe. Le second ensemble de boucles permutables ne contient que la boucle sur j. Deux dépendances sont éliminées ce qui déconnecte le graphe. WOLF-LAM trouve la troisième boucle parallèle après qu'elle ait été distribuée.

De même, ALLEN-KENNEDY ne trouve qu'un degré de parallélisme le graphe étant fortement connexe et contenant un circuit de niveau un et un de niveau deux.

Le lecteur pourra vérifier la validité de l'ordonnancement linéaire décalé qui fait exécuter $S_1(i,j,k)$ à $4i-2k$ et $S_2(i,j,k)$ à $4i-2k+3$. Cet ordonnancement expose deux degrés de parallélisme et montre la sous-optimalité de ALLEN-KENNEDY et de WOLF-LAM pour les graphes de dépendance réduits étiquetés par des vecteurs de direction.

2.4 Algorithme de Feautrier

Contrairement à Lamport, Allen et Kennedy, et Wolf et Lam, Paul Feautrier n'a pas essayé de trouver du parallélisme dans une représentation approchée des dépendances : il ne s'occupe que de dépendances exactes. Dans le cadre des programmes à contrôle statique, il a d'abord décrit un algorithme de construction d'ordonnancements affines monodimensionnels [39], puis multidimensionnels [40].

2.4.1 Cas monodimensionnel

Une analyse de dépendance exacte est réalisable sur les programmes dits « à contrôle statique » dont les fonctions d'accès sont des fonctions affines des indices de boucle et des paramètres de structure [38]. Pour de tels programmes, l'itération $\boldsymbol{j}$ de l'instruction T dépend de l'itération $\boldsymbol{i}$ de l'instruction S si $\boldsymbol{i} = h_{S,T}(\boldsymbol{j})$ et $\boldsymbol{j} \in \mathcal{P}_{S,T}$, où $h_{S,T}$ est une fonction quasi-affine et $\mathcal{P}_{S,T}$ un polyèdre [39].[12] Nous supposerons, comme Paul Feautrier dans son exposé [39], que les fonctions quasi-affines $h_{S,T}$ que nous serons amenés à manipuler sont en fait toujours des fonctions affines. L'équation de satisfaction d'une dépendance entre S et T par l'ordonnancement s'écrit alors :

$$\boldsymbol{j} \in \mathcal{P}_{S,T} \Rightarrow \Theta(T, \boldsymbol{j}) \geq 1 + \Theta(S, h_{S,T}(\boldsymbol{j})). \tag{2.1}$$

Feautrier recherche des ordonnancements affines à valeurs positives compatibles avec les dépendances affines[13] et de la forme :

$$\Theta(S, \boldsymbol{i}) = X_S \boldsymbol{i} + Y_S \boldsymbol{n} + \rho_S$$

où $\boldsymbol{n}$ est le vecteur des paramètres de structure. Il n'est *a priori* pas possible d'énumérer une par une toutes les contraintes incluses dans l'équation (2.1) et qui sont à vérifier par l'ordonnancement, $\mathcal{P}_{S,T}$ étant potentiellement paramétré. Pour contourner cette difficulté, Feautrier a recours au lemme de Farkas affine [39, 74] :

Théorème 3 (Forme affine du lemme de Farkas).

Soit $\mathcal{D}$ un polyèdre non vide défini par p inégalités : $a_k x + b_k \geq 0, k \in \{1, \ldots, p\}$. Une forme affine Φ est positive en tout point de $\mathcal{D}$ si et seulement si elle est une combinaison affine positive des formes affines définissant $\mathcal{D}$:

$$\Phi(x) = \lambda_0 + \sum_{k=1}^{p} \lambda_k (a_k x + b_k), \forall k \in [0, p] \ \lambda_k \geq 0.$$

Les valeurs positives λ_k sont appelées multiplicateurs de Farkas.

Le lemme de Farkas affine permet donc d'exprimer une fonction affine positive comme une combinaison affine des formes affines qui définissent son domaine de positivité. Feautrier l'utilise pour exprimer les fonctions d'ordonnancement (la fonction associée à l'instruction S est positive sur le domaine de S) et l'équation 2.1 qui décrit les contraintes de précédence à respecter (la fonction $\Theta(T, \boldsymbol{j}) - \Theta(S, h_{S,T}(\boldsymbol{j})) - 1$ est affine et positive). Il obtient ainsi un système d'équations linéaires dont les inconnues sont des multiplicateurs de Farkas et qui peut être résolu par programmation linéaire. Cependant, ce système peut ne pas être satisfiable. En ce cas, le programme étudié n'admet pas d'ordonnancements affines monodimensionnels et il faut rechercher des ordonnancements multidimensionnels.

2.4.2 Cas multidimensionnel

Pour résoudre ce problème plus général, Feautrier propose l'algorithme ci-dessous qui est appelé sur le graphe de dépendance réduit.

12. L'existence de dépendances dans un couple d'instructions peut être définie par la donnée de plusieurs fonctions $h^k_{S,T}$ et de leurs polyèdres $\mathcal{P}^k_{S,T}$ associés, et non d'une seule. $\boldsymbol{i}$ est alors source de dépendance si $\boldsymbol{i} = h^k_{S,T}(\boldsymbol{j})$ et $\boldsymbol{j} \in \mathcal{P}^k_{S,T}$ pour une certaine valeur de k.

13. Comme les ordonnancements recherchés sont affines ils n'opèrent pas sur la relation de dépendance exacte mais sur sa fermeture convexe. En d'autres termes, les ordonnancements doivent être valides sur l'intégralité de la fermeture convexe de la relation de dépendance et pas seulement sur ses points entiers.

FEAUTRIER(G)

1. Calculer les composantes fortement connexes de G.

2. **Pour chaque** composante fortement connexe G_i de G dans l'ordre topologique **faire**

   (a) Trouver un ordonnancement affine qui induit un délai positif ou nul sur tous les arcs et qui satisfait l'équation (2.1) pour le plus grand nombre possible de dépendances.

   (b) Construire le sous-graphe G'_i généré par les arcs non satisfaits. Si $G'_i \neq \emptyset$, appeler récursivement FEAUTRIER(G'_i).

Les ordonnancements requis par l'étape 2a sont construits par le procédé mis au point pour les ordonnancements monodimensionnels. Seul l'entier « 1 » de l'équation 2.1 est remplacé par une variable rationnelle $\delta_{S,T}$ à valeurs dans $[0,1]$ qui code si la dépendance entre S et T est ou non satisfaite. Un système d'équations linéaires est obtenu comme précédemment. Il suffit de le résoudre tout en maximisant la somme $\sum_{S,T} \delta_{S,T}$ pour obtenir le résultat escompté.

2.4.3 Application aux graphes de dépendance réduits polyédriques

L'algorithme élaboré par Paul Feautrier à partir du lemme de Farkas affine peut en fait s'appliquer à une représentation des dépendances par des polyèdres. En effet, soient un nid de boucles parfait, deux instructions S et T de même domaine $\mathcal{D}$, T dépendant de S par une dépendance e décrite par le polyèdre $\mathcal{P}_e$. L'ordonnancement doit respecter cette dépendance, c'est-à-dire vérifier l'équation :

$$\forall \boldsymbol{j} \in \mathcal{D},\ \forall \boldsymbol{d} \in \mathcal{P}_e,\ (\boldsymbol{j} - \boldsymbol{d}) \in \mathcal{D} \quad \Rightarrow \quad \Theta(T, \boldsymbol{j}) \geq 1 + \Theta(S, \boldsymbol{j} - \boldsymbol{d}) \tag{2.2}$$

L'ensemble des couples de vecteurs $(\boldsymbol{j}, \boldsymbol{d})$ satisfaisant aux conditions énoncées par le membre gauche de l'implication est trivialement un polyèdre. Nous pouvons donc appliquer à cette équation le lemme de Farkas affine comme il a été utilisé pour l'équation (2.1) qui régissait la situation équivalente pour la représentation exacte des dépendances d'un programme à contrôle statique. Les représentations approchées classiques des dépendances pouvant être simulées de manière équivalente par des polyèdres (cf. section 1.2), l'algorithme de Feautrier s'applique aussi bien à une représentation exacte des dépendances qu'aux représentations approchées.

2.4.4 Optimalité

Dans le cadre de l'analyse de dépendance exacte, les seuls résultats d'optimalité connus pour l'algorithme de Feautrier concernent les nids de boucles uniformes à une ou plusieurs instructions (cf. paragraphe 3.2.2), ce qui est peu. Nous présentons ici un exemple tiré de [39] et pour lequel cet algorithme est sous optimal. Le lecteur trouvera d'autres exemples dans les deux articles originaux [39, 40].

L'algorithme de Feautrier ne trouve pas de parallélisme dans l'exemple 12 (figure 2.11). Le nid de boucles est laissé inchangé alors qu'il contient un degré de parallélisme, comme le montre la forme parallélisé présentée figure 2.11. Pour cet exemple, il n'existe pas d'ordonnancement linéaire optimal. Le problème ici ne provient pas tant du fait que nous cherchons un ordonnancement linéaire que du fait que nous voulons un ordonnancement valable pour le domaine d'itération tout entier.

Exemple 12

```
For i=0,2n
  x(i) = x(2n-i)
```

```
ForPar i=0,n
  x(i) = x(2n-i)
ForPar i=n+1,2n
  x(i) = x(2n-i)
```

FIG. 2.11: *Exemple 12 : nid originel et forme parallélisée.*

2.5 Algorithme de Lim et Lam

Les algorithmes précédemment décrits cherchaient tous à exhiber le maximum de parallélisme. Celui de Amy Lim et Monica Lam [59] recherche quant à lui le maximum de parallélisme exposable sans dépasser un volume de synchronisations donné.

Cet algorithme travaille dans les mêmes conditions que celui de Paul Feautrier dont il peut être considéré comme une extension : il s'intéresse aux dépendances exactes des nids à contrôle statique dont les fonctions d'accès sont affines. Il produit un placement des calculs et un ordonnancement de ceux-ci au moyen de fonctions affines. Les systèmes linéaires sont également obtenus au moyen du lemme de Farkas affine. Plutôt que de résoudre des programmes linéaires comme l'algorithme de Feautrier, celui de Lim et Lam calcule des noyaux de matrices. Bien que le principe de leur algorithme apparaisse clairement dans leur article, nous ne pouvons malheureusement pas en dire autant du choix des transformations.

L'algorithme recherche le parallélisme qui peut être obtenu avec une quantité croissante de synchronisations : aucune synchronisation, $O(1)$, $O(N)$, etc. synchronisations, où N est le paramètre de taille du domaine d'itération.

Parallélisme sans synchronisations. L'algorithme recherche d'abord le plus grand degré de parallélisme qui peut être obtenu sans aucune synchronisation. Pour cela il recherche un placement des calculs de plus grande dimension possible pour lequel deux opérations en relation de dépendance sont systématiquement placées sur le même processeur.

Parallélisme avec $O(1)$ synchronisations. L'algorithme recherche le parallélisme qui peut être exhibé avec un nombre de synchronisations constant, c'est-à-dire indépendant de la taille du domaine. L'algorithme décompose le graphe de dépendance réduit en composantes fortement connexes qu'il ordonnance suivant l'ordre topologique. Il applique à chaque composante fortement connexe l'algorithme de recherche de parallélisme ne nécessitant pas de synchronisations.

Parallélisme avec $O(N)$ synchronisations. Pour exposer du parallélisme qui requiert un nombre de synchronisations linéaire en la taille du domaine, Lim et Lam recherchent un ordonnancement monodimensionnel affine qui satisfait le plus grand nombre de dépendances possible. Cet « ordonnancement » partitionne les opérations suivant leurs dates d'exécution. L'algorithme de recherche du parallélisme nécessitant $O(1)$ synchronisations est appliqué dans chaque ensemble de cette partition.

Ce principe se généralise pour obtenir du parallélisme nécessitant $O(N^k)$ synchronisations, où N est le paramètre de taille du domaine.

Lim et Lam affirment que leur algorithme est optimal : « L'algorithme trouve tous les degrés de parallélisme du programme étudié, tout le parallélisme étant à grain aussi gros que possible » (théorème 8.1 de [59]). Peut-être est-ce vrai en rajoutant la précision « parmi les algorithmes produisant des transformations affines » mais le résultat est faux dans le cas général.

Exemple 13

For i = 1 to N do
  S_1 : a(i) = f(i)
  S_2 : a(N+i-1) = a(i)

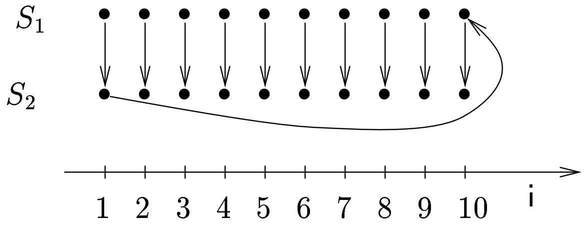


FIG. 2.12: *Contre-exemple à l'optimalité de l'algorithme de Lim et Lam et son GDE.*

Contre-exemple à l'optimalité

L'exemple 13 est un contre-exemple à cette optimalité. Son graphe de dépendance étendu est présenté avec le code à la figure 2.12. Dans ce nid, $S_2(i)$ dépend toujours de $S_1(i)$. Par contre il n'existe qu'une seule dépendance d'une itération de S_2 vers une itération de S_1 : $S_2(1) \Rightarrow S_1(N)$. Les deux instructions sont donc dans la même composante fortement connexe du GDR.

Il n'existe pas de placement affine des calculs qui soit sans synchronisations comme en recherchent Lim et Lam et qui expose du parallélisme. En effet, elles commencent par rechercher un placement sans synchronisations. Une telle fonction place sur un même processeur des instructions en relation de dépendance. Donc $S_1(1)$ et $S_2(1)$ doivent être sur le même processeur, tout comme $S_2(1)$ et $S_1(N)$. Le placement affine doit mettre sur le même processeur deux instances différentes de S_1 et ce dans un espace de dimension un. Toutes les instances de S_1 sont donc placées sur le même processeur et il en va de même de S_2 et il n'y a pas de parallélisme exposable par des fonctions affines de placement et qui serait sans synchronisations. Le graphe de dépendance réduit étant fortement connexe, cela revient ici au même de rechercher comme elles le font du parallélisme sans ou avec un nombre borné de synchronisations. Leur algorithme est donc incapable de trouver ici du parallélisme qui ne requiert qu'un nombre constant de synchronisations.

Une première solution consiste pourtant à placer $S_1(i)$ et $S_2(i)$ sur le processeur i, pour $1 \leq i \leq N-1$, et à placer $S_1(N)$ et $S_2(N)$ sur le processeur 1. Nous obtenons ainsi une solution qui n'est pas affine mais qui ne requiert aucune synchronisation. Une deuxième solution consiste simplement à placer $S_1(i)$ et $S_2(i)$ sur le processeur i, pour $1 \leq i \leq N$. Cette solution est affine et requiert une seule synchronisation : après l'exécution des processeurs de numéro inférieur à $N-1$ et avant celle du processeur N. Ce découpage contredit la démonstration de leur théorème 6.3 en fournissant un contre-exemple à l'argument : « tout schéma de parallélisation qui assigne à des processeurs différents des opérations d'une même composante fortement connexe du graphe de dépendance réduit requiert $\Omega(N)$ synchronisations où N est le nombre d'itérations d'une boucle »[14]. Cette proposition serait vraie si nous avions une représentation approchée des dépendances, tout vecteur de distance existant alors potentiellement dans l'ensemble du domaine d'itération et non pas en un point précis. Mais ici, par hypothèse, nous sommes dans le cadre des dépendances exactes... Le lecteur peut reprocher une certaine irrégularité de la génération des synchronisations nécessaires à notre contre-exemple. Il suffirait peut être de rajouter au théorème 6.3 des hypothèses de régularité sur le placement et sur la génération des synchronisations pour assurer sa validité. Dans l'état actuel de nos connaissances, il est simplement faux.

14. Je ne résiste pas au plaisir de citer la phrase originelle : « Any parallelization scheme that assigns data dependent operations in an SCC [(Strongly Connected Component)] to different processors requires at least $O(n)$ [sic] synchronizations, where n is the number of iterations in a loop ».

2.6 Conclusion

Nous venons d'étudier cinq algorithmes de détection de parallélisme. Nous ne revenons pas ici sur l'algorithme de Lamport qui est généralisé par l'algorithme de Wolf et Lam, ni sur celui de Lim et Lam qui est équivalent pour la détection de parallélisme à l'algorithme de Feautrier.

L'algorithme de Paul Feautrier, tel qu'il a été originellement présenté, travaille sur une représentation exacte des dépendances. De ce fait, aucune perte de parallélisme ne peut être imputable au calcul de dépendances. C'est le principal avantage de l'approche. Mais un algorithme de détection peut « laisser s'échapper » du parallélisme et son efficacité doit être étudiée très précisément. Nous n'avons malheureusement aucun résultat d'optimalité pour cet algorithme hors de la classe très restreinte des nids à dépendances uniformes. Ce que nous appelons « algorithme » est d'ailleurs qualifié d'« heuristique » par son auteur. Cet algorithme requiert qu'une analyse de dépendance exacte soit réalisable, ce qui impose de très fortes restrictions sur les nids étudiés. De fait, seuls les programmes à contrôle statique et à fonctions d'accès affines sont traitables par cette méthode. Cette limitation sur les codes constitue le principal inconvénient de l'approche. Des travaux récents ont tentés d'affaiblir ces contraintes en généralisant l'analyse de dépendance exacte en une *analyse floue du flot de donnée* [23].

L'algorithme d'Allen et Kennedy et celui de Wolf et Lam travaillent sur une représentation approchée des dépendances. L'utilisation d'une représentation approchée à l'avantage de nous éviter d'avoir à faire des hypothèses sur les codes et donc d'étudier une classe très générale de programmes. L'inconvénient d'une représentation approchée, c'est le coût, en terme de perte de parallélisme, de l'approximation des dépendances. Les dépendances sont en effet approchées conservativement, c'est-à-dire sur-approximées : la représentation choisie peut représenter plus de dépendances que le code n'en contient. Les dépendances additionnelles induisent une plus grande séquentialité apparente du code originel. Ce surcroît de séquentialité peut engendrer une perte de parallélisme. Le problème est de déterminer si une telle perte est ou non tolérable. La question fondamentale est alors : si l'algorithme de détection utilisé n'a trouvé aucun parallélisme dans notre nid de boucles, une autre technique en aurait-elle trouvé ? À l'heure actuelle il n'y a pas de réponse complète à une question aussi générale. Parfois, WOLF-LAM découvre de parallélisme quand ALLEN-KENNEDY ne trouve que de la séquentialité, et vice versa, comme le montrent les exemples 6 et 10. Ces exemples prouvent, s'il en était besoin, qu'une représentation des dépendances par niveaux est trop grossière et une représentation par vecteurs de direction sans notion de localisation est trop lacunaire pour qu'elles permettent une détection efficace du parallélisme. Nous avons tenté de combiner les deux types de représentation (vecteurs de direction et graphe de dépendance) et d'étendre l'algorithme de Wolf et Lam. Nous avons décrit de la sorte une nouvelle version de WOLF-LAM qui s'est révélée être sous-optimale. Cette situation illustre la difficulté à combiner les méthodes et concepts utilisés par les algorithmes de Allen et Kennedy et de Wolf et Lam.

Nous avons donc d'un côté un algorithme utilisant des techniques puissantes sur une représentation exacte des dépendances mais sans que nous ayons de garantie sur son efficacité. De l'autre côté, nous avons des algorithmes plus simples travaillant sur une représentation approchée des dépendances et dont l'(in)efficacité est bien définie. Schématiquement, nous aimerions disposer d'un algorithme travaillant sur une représentation approchée des dépendances afin de pouvoir traiter des programmes plus généraux que ceux étudiés par l'algorithme de Paul Feautrier. Nous voudrions utiliser des techniques suffisamment puissantes pour extraire tout le parallélisme (au sens défini section 1.4) contenu dans la représentation des dépendances choisie. Cependant, nous aimerions générer des codes aussi simples que faire se peut. Nous aimerions que cet algorithme idyllique trouve du parallélisme à chaque fois que ALLEN-KENNEDY ou une des versions de WOLF-LAM en trouve.

Il existe bien un algorithme qui remplit cette dernière condition : celui qui essaye ALLEN-KENNEDY et les deux versions de WOLF-LAM et renvoie la meilleure réponse ! La manière dont cet algorithme a été construit est sa principale faiblesse : il utilise la structure du graphe de dépendance (ALLEN-KENNEDY) ou la précision des vecteurs de direction (WOLF-LAM), mais il ne profite jamais simultanément des deux types d'informations. Ainsi, l'extension de WOLF-LAM que nous avons proposée utilise la structure du graphe de dépendance avant ou après le calcul d'un nouvel ensemble de boucles permutables, mais jamais *pendant* ce calcul.

Wolf et Lam ont eu pour but de « combiner la rigueur mathématique du modèle des transformations matricielles avec la généralité de l'approche des compilateurs vectoriseurs et paralléliseurs ». Nous avons le même but, mais nous voulons mettre l'accent sur l'utilisation de la structure des graphes de dépendances.

Il y a un fossé entre la complexité des algorithmes ALLEN-KENNEDY et WOLF-LAM d'une part et FEAUTRIER de l'autre, tant en termes de représentation des dépendances que pour les techniques utilisées. Nous aimerions le combler et proposer un algorithme intermédiaire, notamment en complexité, mais néanmoins optimal pour toutes les approximations classiques des dépendances. Nous aimerions générer des codes simples pour ne pas perdre à l'exécution ce que nous aurions théoriquement gagné à utiliser un algorithme de parallélisation plus puissant. Nous voulons donc utiliser des techniques intermédiaires entre celles mises en jeu par ALLEN-KENNEDY et WOLF-LAM, qui génèrent des codes très (trop) simples, et celles utilisées par FEAUTRIER, qui sont nettement plus expressives mais peuvent engendrer des codes très complexes.

Nous allons répondre imparfaitement à ces désirs dans le reste de ce mémoire, principalement au chapitre 4. Notre algorithme repose sur une étude systématique des propriétés fondamentales des systèmes d'équations récurrentes uniformes exposée au chapitre 3. Cette étude nous permettra de produire un algorithme de parallélisation des graphes de dépendance réduits polyédriques. Nous avons déjà montré (section 1.2) la généralité de cette forme d'approximation. Nous motivons ci-dessous son intérêt pratique.

De l'intérêt d'une approximation polyédrique des dépendances

L'exemple 14 est présenté figure 2.13 où sont énumérées les relations exactes de dépendance. La figure 2.14 montre les graphes de dépendance réduits où les dépendances sont représentées par (a) des niveaux de dépendances et (b) des vecteurs de direction. Nous considérons la parallélisation de l'exemple 14 au moyen des algorithmes de Allen et Kennedy, de Wolf et Lam et de Feautrier.

Exemple 14

```
For i = 1 to n do
  For j = 1 to n do
    S: a(i, j) = a(j, i) + a(i, j - 1)
```

$$\begin{cases} \text{if } 1 \leq i \leq n,\ 1 \leq j < n & S(i,j) \xrightarrow{\text{flot}} S(i,j+1) \\ \text{if } 1 \leq i < j \leq n & S(i,j) \xrightarrow{\text{flot}} S(j,i) \\ \text{if } 1 \leq j < i \leq n & S(j,i) \xrightarrow{\text{anti}} S(i,j) \end{cases}$$

FIG. 2.13: *Exemple 14 et ses relations de dépendance exactes.*

ALLEN-KENNEDY. Les trois dépendances sont respectivement de niveaux 2, 1 et 1. Il y a un circuit de dépendance au niveau un et un au niveau deux. Aucun parallélisme n'est détecté.

WOLF-LAM. Les vecteurs de direction valent respectivement $(0,1)$, $(+,-)$ et $(+,-)$. Dans la deuxième dimension le « 1 » et les « $-$ » sont incompatibles et préviennent la détection d'un ensemble de deux boucles totalement permutables. Le code reste inchangé et aucun parallélisme n'est détecté.

FEAUTRIER. Cet algorithme trouve l'ordonnancement affine : $\Theta(i,j) = 2i + j - 3$ et un degré de parallélisme.

Dans notre exemple, une représentation des dépendances par des niveaux ou des vecteurs de direction ne contient pas de parallélisme. C'est pourquoi ALLEN-KENNEDY et WOLF-LAM ont échoué. Une analyse de dépendance exacte associée à des méthodes de programmation linéaire qui nécessitent la résolution de gros[15] programmes linéaires paramétrés permet, quant à elle, la découverte d'un degré de parallélisme. Le code parallélisé est produit figure 2.15.

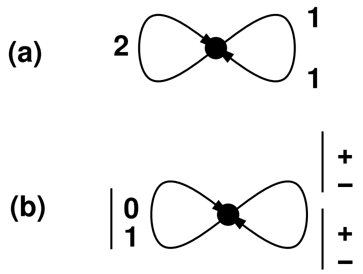


FIG. 2.14: *GDR de l'exemple 14.*

For j = 3 to 3 n do
  ForPar $i = \max\left(1, \lceil \frac{j-n}{2} \rceil\right)$ to $\min\left(n, \lfloor \frac{j-1}{2} \rfloor\right)$ do
    $a(i,\, j-2i) = a(j-2i,\, i) + a(i,\, j-2i-1)$

FIG. 2.15: *Version parallélisée de l'exemple 14.*

Il n'est cependant pas nécessaire de recourir à une représentation exacte des dépendances pour pouvoir découvrir du parallélisme dans l'exemple 14. En effet, il existe une dépendance uniforme $\boldsymbol{u} = (0,1)$ et un ensemble de vecteurs de distance $\{(j-i, i-j) = (j-i)(1,-1) \mid 1 \leq j-i \leq n-1\}$ qui peut être sur-approché par l'ensemble $\mathcal{P} = \{(1,-1) + \lambda(1,-1) \mid \lambda \geq 0\}$. $\mathcal{P}$ est un polyèdre avec un sommet, $\boldsymbol{v} = (1,-1)$, et un rayon, $\boldsymbol{r} = (1,-1)$. Supposons que nous recherchions un ordonnancement linéaire $\Theta(i,j) = x_1 i + x_2 j$. Soit $X = (x_1, x_2)$. Pour que Θ soit valide, il nous faut avoir l'inégalité $X\boldsymbol{d} \geq 1$ pour tout vecteur de dépendance $\boldsymbol{d}$, c'est-à-dire $X(0,1) \geq 1$ et $X\boldsymbol{d} \geq 1$ pour tout $\boldsymbol{d} \in \mathcal{P}$. La dernière inégalité se réécrit : $X(1,-1) + \lambda X(1,-1) \geq 1$ où $\lambda \geq 0$, ce qui équivaut au système : $X(1,-1) \geq 1$ et $X(1,-1) \geq 0$, c'est-à-dire $X\boldsymbol{v} \geq 1$ et $X\boldsymbol{r} \geq 0$.

Un ordonnancement linéaire valide est donc défini par un vecteur X satisfaisant les trois inégalités :

$$X\boldsymbol{u} \geq 1 \quad X\boldsymbol{v} \geq 1 \quad X\boldsymbol{r} \geq 0 \quad \text{i.e.} \quad X\begin{pmatrix} 0 \\ 1 \end{pmatrix} \geq 1 \quad X\begin{pmatrix} 1 \\ -1 \end{pmatrix} \geq 1 \quad X\begin{pmatrix} 1 \\ -1 \end{pmatrix} \geq 0\ ,$$

ce qui nous donne, en minimisant la norme de X, $X = (2,1)$ et nous trouvons le même résultat que l'algorithme de Feautrier, à une constante près.

Dans cet exemple, une approximation des dépendances par des niveaux ou des vecteurs de direction ne suffit pas à permettre la détection de parallélisme. Au contraire, avec une approximation par polyèdres de dépendance, nous avons trouvé autant de parallélisme qu'avec une représentation exacte, mais en résolvant un système d'inéquations plus simple.

La pierre angulaire de notre manipulation est l'« uniformisation » qui nous a permis de passer de l'inégalité sur l'ensemble $\mathcal{P}$ aux inégalités sur $\boldsymbol{v}$ et sur $\boldsymbol{r}$. Grâce à cette uniformisation, les contraintes

15. Le nombre d'inégalité et de variables est lié au nombre des contraintes qui définissent le domaine de validité de chaque relation de dépendance.

affines ont disparu et, contrairement à l'algorithme de Feautrier, nous n'avons pas eu besoin du lemme de Farkas affine. Pour mieux comprendre le principe de l'uniformisation, raisonnons en terme de chemins de dépendance. L'idée est de considérer un arc e, d'une instruction S vers une instruction T, étiqueté du vecteur de distance $\boldsymbol{d} = \boldsymbol{v} + \lambda \boldsymbol{r}$, comme un chemin ϕ utilisant une fois le vecteur de dépendance « uniforme » $\boldsymbol{v}$ et λ fois le vecteur de dépendance « uniforme » $\boldsymbol{r}$. Cette simulation est illustrée figure 2.16 : pour simuler ϕ, nous introduisons un nouveau nœud S' et un arc de poids nul de S' vers le nœud initial T. L'algorithme de parallélisation qui sera présenté au chapitre 4 repose sur ce principe d'uniformisation.

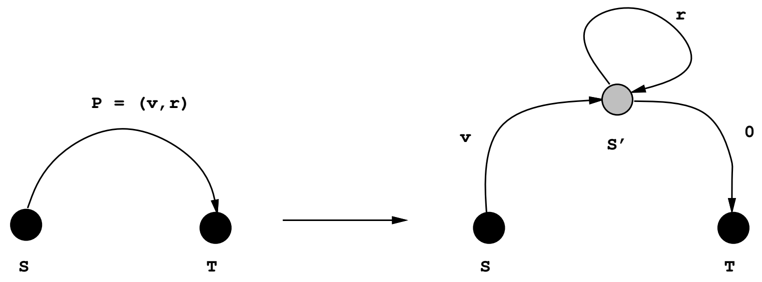


FIG. 2.16: *Simulation d'un arc valué par un polyèdre contenant un sommet et un rayon.*

L'uniformisation des dépendances a, de fait, uniformisé les contraintes et transformé le problème d'ordonnancement affine sous-jacent en un simple problème d'ordonnancement où toutes les dépendances sont uniformes (0, $\boldsymbol{v}$ et $\boldsymbol{r}$ de la figure 2.16). Il existe cependant deux différences fondamentales entre le problème d'ordonnancement ainsi défini et le cadre classique des nids de boucles uniformes :

- Les vecteurs de dépendance ne sont pas nécessairement lexicographiquement positifs (un rayon peut être égal à $(0, -1)$). Le problème en devient beaucoup plus compliqué. Sans la lexico-positivité des vecteurs de distance, la méthode de l'hyperplan de Lamport ne s'applique plus et nous ne sommes plus assurés de trouver $n - 1$ degrés de parallélisme dans un nid de profondeur n. Par contre, un graphe de dépendance réduit dont les arcs sont valués par des vecteurs entiers quelconques est typiquement le GDR d'un *système d'équations récurrentes uniformes*. Pour ordonnancer nos graphes de dépendance réduits polyédriques, nous allons donc tout naturellement nous servir des techniques développées pour déterminer la calculabilité et ordonnancer les systèmes d'équations récurrentes uniformes, techniques qui font l'objet du chapitre 3.

- La contrainte imposée par un rayon $\boldsymbol{r}$ est plus faible que la contrainte imposée par une dépendance uniforme classique : l'inégalité à satisfaire est simplement $X\boldsymbol{r} \geq 0$ au lieu de $X\boldsymbol{r} \geq 1$. La liberté supplémentaire ainsi octroyée doit être prise en compte par l'algorithme de parallélisation.

Nous étudions donc au chapitre 3 les systèmes d'équations récurrentes uniformes et nous en appliquerons les résultats au graphes de dépendance réduits polyédriques au chapitre 4.

Chapitre 3

Équations récurrentes uniformes

> Oui, monsieur le président. Je plumote à l'aile d'oie; l'aile d'oie est assurément la plus élégante et la plus efficace des plumes à plumeau
>
> Louis *in* Il faut que le Sycomore coule, Jean-Michel **Ribes**

Richard Karp, Raymond Miller et Shmuel Winograd ont introduit [51] en 1967 les concepts d'*Équation Récurrente Uniforme* (ERU) et de *Système d'Équations Récurrentes Uniformes* (SERU). Ces équations récurrentes permettent de décrire une répétition régulière de calculs. Dans leur article fondateur, Karp, Miller et Winograd ont principalement étudié le problème de calculabilité de ces équations et systèmes. Ils n'ont pas proposé de solution au problème d'ordonnancement. Néanmoins, tous les résultats présentés dans ce chapitre étaient souvent préfigurés dans leur article fondateur et ils en découlent plus ou moins directement. Les principaux prolongements des travaux de Karp, Miller et Winograd sont dus à Sailesh Rao [70] et à Vwani Roychowdhury [72] qui ont tous deux étudié la latence et l'ordonnancement des systèmes d'équations récurrentes, rebaptisés pour l'occasion *algorithmes itératifs réguliers* (Regular Iterative Algorithms, en anglais).

Nous rappelons la définition des systèmes d'équations récurrentes uniformes à la section 3.1. Nous nous penchons à la section 3.2 sur leur calculabilité et nous lions celle-ci à l'existence de (multi-)circuits de poids nul. Nous présentons l'algorithme de Karp, Miller et Winograd qui détermine la calculabilité d'un SERU et nous nous en servons pour déterminer la latence théorique de ces systèmes d'équations. À partir de l'algorithme de détermination de la calculabilité, nous construisons à la section 3.3 un algorithme optimal d'ordonnancement multidimensionnel.

Nous avons brièvement motivé notre intérêt pour ces systèmes d'équations à la section 2.6. Le présent chapitre ne se justifie pas *a priori* mais *a posteriori*, quand nous aurons transposé aux graphes de dépendance réduits polyédriques les résultats obtenus pour les systèmes d'équations récurrentes uniformes. Il en va de même de la section sur la calculabilité qui ne se justifie pleinement que parce qu'elle nous donne naturellement une solution au problème de l'ordonnancement.

3.1 Définition

Définition 12 (Système d'équations récurrentes uniformes).
Un système d'équations récurrentes uniformes est constitué :

- *d'un domaine $\mathcal{D}$, polyèdre de $\mathbb{Q}^n$;*
- *de m variables $V_1, \ldots, V_m$ définies sur les points entiers du polyèdre $\mathcal{D}$;*

– *de m équations :*

$$V_i(z) = f_i(V_{i_1}(z - d_{1,i}), \ldots, V_{i_j}(z - d_{j,i}), \ldots, V_{i_{m_i}}(z - d_{m_i,i})) \tag{3.1}$$

où z est un point de $\mathcal{D}$, où, pour tout couple (i, j), $d_{i,j}$ désigne un vecteur constant de $\mathbb{Z}^n$, et où f_i est une fonction stricte à m_i arguments.

De plus, la valeur de la variable V_{i_j} est supposée connue aux points $(z - d_{j,i})$ extérieurs à $\mathcal{D}$ et pour lesquels z appartient à $\mathcal{D}$.

La figure 3.1 propose un exemple de SERU (exemple 15).

Les systèmes d'équations récurrentes uniformes que nous utiliserons émaneront de nids de boucles, or, dans un nid de boucles, le domaine d'une instruction est toujours borné. Nous supposerons donc que *le domaine $\mathcal{D}$ est borné*.

Le système d'équations est dit *uniforme* car les distances de dépendance, définies par les vecteurs $d_{i,j}$, sont constantes. Un système d'équations récurrentes uniformes ressemble fortement à un nid de boucles à dépendances uniformes. Néanmoins, il existe une différence fondamentale entre ces deux objets : le nid de boucles contient intrinsèquement un ordre d'exécution des calculs qu'il décrit au contraire du système d'équations récurrentes uniformes qui peut même ne pas être calculable. En effet, l'équation (3.1) ne permet de calculer la valeur de la variable V_i au point z qu'une fois connues les valeurs des variables $V_{i_1}, \ldots, V_{i_{m_i}}$ aux points $(z - d_{1,i}), \ldots, (z - d_{m_i,i})$ respectivement. Si la valeur d'une variable en un point donné dépend d'elle-même, le système ne sera pas calculable ce qu'illustre le cas extrême de l'équation récurrente uniforme : $V(z) = V(z), z \in \mathbb{Z}^n$. Nous étudierons en détail ce problème de calculabilité à la section 3.2. La différence entre SERU et nids uniformes s'explique par la non lexico-positivité des vecteurs de dépendance d'un système d'équations récurrentes uniformes. Les vecteurs $d_{i,j}$ peuvent prendre n'importe quelle valeur.

Tout comme pour les nids de boucles, nous définissons pour les systèmes d'équations récurrentes uniformes les notions de graphe de dépendance réduit, de graphe de dépendance étendu, ainsi que toutes les notions afférentes. La figure 3.1 présente le GDR du système d'équations de l'exemple 15.

Exemple 15

$$\begin{cases} \mathsf{a(i, j, k) = b(i\text{-}1, j\text{-}1, k) + a(i, j, k\text{-}1)} \\ \mathsf{b(i, j, k) = a(i\text{-}1, j, k) + b(i, j, k\text{+}1)} \end{cases}$$

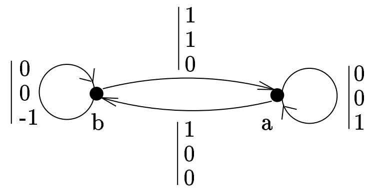


FIG. 3.1: *Exemple de système d'équations récurrentes uniformes et son graphe de dépendance.*

3.2 Calculabilité et latence

Nous avons déjà mentionné le problème de calculabilité des systèmes d'équations récurrentes uniformes. Karp, Miller et Winograd ont montré qu'un système d'équations récurrentes uniformes sur un domaine fini est calculable si et seulement si une instance d'une variable ne dépend jamais d'elle-même, c'est-à-dire si et seulement si le graphe de dépendance étendu ne contient pas de circuits. Si le graphe de dépendance réduit ne contient pas de circuits de poids nul alors le graphe de dépendance étendu ne contient pas de circuits. La réciproque n'est vraie que si le domaine $\mathcal{D}$ est suffisamment grand. En effet, si le graphe de dépendance réduit contient un circuit de poids nul il faut encore que le domaine $\mathcal{D}$ soit suffisamment grand pour pouvoir construire dans le graphe de dépendance étendu un circuit qui lui corresponde et qui rende le système non calculable. Le lecteur trouvera dans [51] une borne supérieure sur la taille nécessaire de $\mathcal{D}$. Nous nous intéressons uniquement à des systèmes d'équations avec des domaines larges car nous avons en tête la parallélisation de nids de boucles, et que cette parallélisation ne se justifie que si le nid étudié décrit un grand nombre de calculs. Pour déterminer si un système est ou non calculable, nous devons donc chercher dans son graphe de dépendance réduit d'éventuels circuits de poids nul. Nous présentons au paragraphe 3.2.1 un algorithme de recherche de tels circuits qui fut originellement proposé par Karp, Miller et Winograd [51], puis fut étudié par Rao [70], Roychowdhury [72], et Kosaraju et Sullivan [54]. Tous les résultats présentés dans ce chapitre reposent sur les propriétés de la solution proposée par Karp, Miller et Winograd. Nous nous attarderons sur la réalisation de l'algorithme à la paragraphe 3.2.3. Auparavant, nous construirons à la paragraphe 3.2.2 des chemins de dépendance qui se révéleront ultérieurement être les plus longs en ordre de grandeur. La longueur de ces chemins bornent le parallélisme des SERU.

3.2.1 Détection de circuits de poids nul

Définitions et notations

Dans toute cette section, G dénote le graphe de dépendance réduit d'un système d'équations récurrentes uniformes.

Définition 13 (Matrices de dépendance et de connexion).
Nous notons C la matrice de connexion*, ou* matrice d'incidence*, du graphe de dépendance réduit G. Chacune des lignes de C correspond à un nœud de G et chacune de ses colonnes à un arc. Si le $j^{\text{ème}}$ arc de G est une auto-dépendance, la $j^{\text{ème}}$ colonne de C est entièrement nulle. Sinon, si le $j^{\text{ème}}$ arc de G est orienté du $i^{\text{ème}}$ nœud vers le $k^{\text{ème}}$, $C_{i,j}$ est égal à -1, $C_{k,j}$ à $+1$, et le reste de la $j^{\text{ème}}$ colonne est nul ($l \notin \{i,k\} \Rightarrow C_{l,j} = 0$).*
La $j^{\text{ème}}$ colonne de la matrice de dépendance D contient le vecteur de dépendance associé au $j^{\text{ème}}$ arc de G.
Nous noterons B la matrice par blocs dont les premières lignes sont les lignes de C et les dernières lignes celles de D.

Nous illustrons ces définitions à l'aide de l'exemple 15 :

$$C = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \end{bmatrix} \quad B = \begin{bmatrix} C \\ D \end{bmatrix}$$

Multi-circuits de poids nul

Définition 14 (Multi-circuit).
Un multi-circuit est une union de circuits élémentaires, non nécessairement connexe. Dans le cas de graphes valués, le poids d'un multi-circuit est égal à la somme des valuations de ses circuits constitutifs.

Nous pouvons définir de manière équationnelle la notion de *multi-circuit de poids nul.* Tout vecteur non nul, entier et positif q satisfaisant l'équation $Cq = 0$ décrit un multi-circuit, et réciproquement. Si ce vecteur appartient de plus au noyau à droite de D : $Dq = 0$ (et donc $Bq = 0$), le multi-circuit est de poids nul. Détecter l'existence de multi-circuits de poids nul équivaut donc à rechercher l'existence de solutions entières du système :

$$\{q \geq 0, \quad q \neq 0, \quad Bq = 0\}.$$

Toute solution rationnelle de ce système multipliée par un entier idoine fournit une solution entière impliquant les mêmes arcs. Comme seule nous importe l'appartenance d'arcs à des multi-circuits de poids nul, nous nous contentons de chercher des solutions rationnelles du système. Nous savons donc construire des multi-circuits de poids nul. Nous étudions maintenant leurs propriétés et nous montrons que leur existence est liée au problème de la calculabilité. À la section 3.2.3, nous présentons des méthodes de construction des multi-circuits de poids nul « efficaces », au contraire de la méthode proposée par Karp, Miller et Winograd et utilisée ci-dessous.

Notation. *Par la suite, G' désignera le sous-graphe de $G = (V, E)$ généré par les arcs de G qui appartiennent à au moins un multi-circuit de poids nul, c'est-à-dire généré par l'ensemble :* $\{e \in E \mid \exists q \neq 0, q \geq 0, Bq = 0 \text{ et } q_e \neq 0\}$.

Notre problème est la recherche de circuits de poids nul. Nous ne savons pas détecter directement la présence de circuits de poids nul dans un graphe, mais nous savons détecter la présence de multi-circuits de poids nul. Or *un circuit de poids nul est un multi-circuit de poids nul*, même si la réciproque est fausse. Nous allons donc rechercher les multi-circuits de poids nul et regarder si, parmi ces multi-circuits, il y a ou non des circuits de poids nul. Si notre graphe G ne contient pas de multi-circuits de poids nul (G' est vide), il ne contient *a fortiori* pas de circuits de poids nul. L'autre cas extrême est celui où G' est fortement connexe. Dans ce cas, G contient un circuit de poids nul comme nous le montrons. Un circuit est toujours inclus dans une composante fortement connexe. Si G' n'est ni vide ni fortement connexe, nous considérons chacune des composantes fortement connexes de G' et à y rechercher récursivement des circuits de poids nul. Formellement, les lemmes et corollaires suivants, qui furent originellement établis par Karp, Miller et Winograd, nous permettent de ramener la recherche de circuits de poids nul à l'étude des multi-circuits de poids nul.

Lemme 3. *Un graphe G contient un circuit de poids nul si et seulement si son sous-graphe G' en contient un.*

Démonstration. Par définition, G' est généré par les arcs de G qui participent à au moins un multi-circuit de poids nul. Un circuit de poids nul étant un tel multi-circuit, G' contient les circuits de poids nul de G. Réciproquement, G' étant inclus dans G, ses circuits sont des circuits de G. □

Corollaire 2. *Si le sous-graphe G' de G est vide, G ne contient pas de circuits de poids nul.*

Corollaire 3. *G contient un circuit de poids nul si et seulement si l'une des composantes fortement connexes de G' en contient un.*

Lemme 4. *Si le sous-graphe G' de G est fortement connexe, alors il contient un circuit de poids nul.*

Démonstration. Si G' est fortement connexe, il contient un circuit $\mathcal{C}$ qui passe par tous ses nœuds. $\mathcal{C}$, considéré comme un chemin, est formé des arcs e_1, ..., e_i, ..., e_p. Par définition de G', chacun de ces arcs appartient à un multi-circuit de poids nul $\mathcal{C}_i$. $\mathcal{C}_i$ se décompose sous la forme : $\mathcal{C}_i = e_i + \pi_i + \mathcal{C}'_i$ où $e_i + \pi_i$ est un circuit de G' et $\mathcal{C}'_i$ un multi-circuit. π_i est un chemin du nœud tête de l'arc e_i ($t(e_i)$) au nœud queue de e_i ($q(e_i)$). Nous construisons un circuit de poids nul de

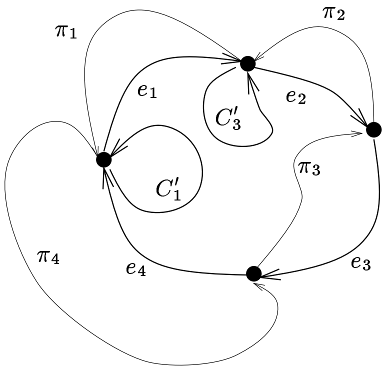


FIG. 3.2: *Construction d'un circuit de poids nul dans un graphe G' fortement connexe.*

G' comme suit : nous parcourons les arcs e_1, ..., e_i, ..., e_p puis nous revenons au point de départ en suivant successivement les chemins π_p, ..., π_i, ..., π_1. Comme ce circuit visite tous les nœuds de G', nous pouvons y insérer les unions de circuits $\mathcal{C}'_i$. Le poids du circuit obtenu est nul puisqu'il est la somme des poids des unions $\mathcal{C}_i$. La construction de ce circuit de poids nul est illustrée par la figure 3.2. □

Nous rappelons maintenant l'algorithme de Karp, Miller et Winograd qui permet de décider si un système d'équations récurrentes uniformes est ou non calculable. Cet algorithme est la transcription formelle de la décomposition récursive du graphe de dépendance réduit que nous avons évoquée plus haut.

Algorithme de détection (KMW)

Cet algorithme détermine la calculabilité d'un système d'équations récurrentes uniformes de GDR G : l'appel KMW(G) renvoie VRAI si et seulement si G est calculable.

booléen KMW(G)

1. Décomposer G en ses s composantes fortement connexes, G_1, ..., G_i, ..., G_s. Appeler l'étape 2 sur chacun des sous-graphes G_i et renvoyer le ET logique des réponses.

2. Construire G'_i le sous-graphe de G_i généré par les arcs de G_i qui appartiennent à au moins un multi-circuit de poids nul.

   – Si G'_i est vide, renvoyer VRAI.
   – Si G'_i est fortement connexe, renvoyer FAUX.
   – Sinon, renvoyer KMW(G'_i).

Lemme 5. *L'algorithme de décomposition* KMW *est correct.*

DÉMONSTRATION. L'algorithme se termine toujours quand il est invoqué sur un graphe fini. En effet, soit l'étape 2 de l'algorithme appliquée au graphe F se termine, F' étant vide ou fortement connexe, soit l'algorithme est appelé récursivement sur les composantes fortement connexes de F' qui sont des sous-graphes stricts de F.

Si la décomposition se conclut sur un sous-graphe G'_i fortement connexe, le lemme 4 nous assure que G'_i, et donc G, contient un circuit de poids nul, et le système d'équations est par conséquent non calculable. En revanche, si tous les appels récursifs aboutissent à des sous-graphes G'_i vides, le corollaire 3 nous permet de conclure que G ne contient pas de circuits de poids nul. □

Avant d'étudier au paragraphe 3.2.3 la construction et les propriétés de G', nous allons nous intéresser à la longueur des chemins de dépendance du graphe de dépendance étendu d'un SERU. Au préalable, nous devons introduire la notion de profondeur de décomposition.

Définition 15 (Profondeur de décomposition). *À tout nœud S de G nous associons une profondeur de décomposition d_S égale au nombre d'appels récursifs de l'algorithme* KMW *qui concernent un graphe contenant le nœud v.*

La profondeur de décomposition d_G du graphe G est elle égale à la plus grande profondeur de décomposition d'un nœud de G. Elle est notée d s'il n'y a pas ambiguïté.

Si le premier appel se termine directement (chaque sous-graphe G'_i est soit vide soit fortement connexe), aucun appel récursif n'est généré et tous les nœuds de G sont de profondeur un. À une profondeur strictement plus grande que un, c'est-à-dire après un premier appel récursif, le graphe F traité par KMW ne contient pas de nœuds *isolés*, c'est-à-dire de nœuds qui n'appartiennent à aucun circuit. Par souci d'homogénéité, nous éliminerons de G les éventuels nœuds isolés, ceux-ci ne jouant aucun rôle dans les problèmes de calculabilité et d'ordonnancement, et nous dirons que la profondeur de décomposition de G est nulle ($d = 0$) si tous ses nœuds sont isolés, c'est-à-dire si toutes ses composantes fortement connexes sont réduites à des graphes contenant un unique nœud et aucun arc.

Nous allons maintenant relier la profondeur de décomposition et la longueur des plus longs chemins de dépendance, ceux-ci nous donnant une mesure du parallélisme du système.

3.2.2 Plus longs chemins de dépendance

Une fois connue la calculabilité d'un système d'équations récurrentes uniformes, il est intéressant de considérer la longueur de ses chemins de dépendance car ceux-ci fournissent une borne inférieure sur la séquentialité du système et donc une borne supérieure sur son parallélisme.

Alain Darte, Leonid Khachiyan et Yves Robert ont montré [27] que le plus long chemin de dépendance d'une simple équation récurrente uniforme était de longueur équivalente à la latence de l'ordonnancement linéaire optimal, pour des domaines de dimension pleine dont la taille tend vers l'infini. Remarquez que la latence d'un ordonnancement linéaire est le nombre d'itérations de la boucle séquentielle externe du nid de boucles qui décrit l'équation récurrente uniforme ordonnancée. Sommairement, pour des domaines de taille paramétrée par N, la latence de l'ordonnancement linéaire optimal est équivalente à λN, pour une certaine constante λ, et il en va de même de la longueur du plus long chemin de dépendance : les deux grandeurs sont équivalentes à λN pour une même valeur de λ. Patrick Le Gouëslier d'Argence a étendu ce résultat [26, 25] aux systèmes d'équations récurrentes uniformes qui admettent des ordonnancements monodimensionnels et dont les graphes de dépendance réduits sont fortement connexes.

Nous verrons qu'il n'existe pas toujours d'ordonnancements monodimensionnels et que la longueur des plus longs chemins n'est plus nécessairement en $O(N)$, mais peut être équivalente à kN^p, pour certaines constantes k et p, où p est entière. Nous n'essayerons pas d'être précis sur le facteur multiplicatif k, mais seulement sur la puissance p, conformément à la notion d'optimalité que nous avons défini à la section 1.4. Le théorème 4 indique que la puissance p est au moins égale à la profondeur de décomposition du graphe de dépendance réduit. Nous établirons l'égalité des deux grandeurs à la section 3.3. Ce théorème a été initialement présenté par Sailesh Rao [70]. Comme l'a écrit Borgès [11], la preuve existentielle de Sailesh Rao « n'entraîne pas la moindre conviction, mais n'admet pas la moindre réfutation ». Nous proposons ici une démonstration constructive qui repose sur l'idée de la preuve de Rao.

Théorème 4 (Plus longs chemins d'un système d'équations récurrentes uniformes).
Si le domaine contient un hypercube de dimension n et de taille $\Omega(N)$ et si d_S est la profondeur de décomposition du nœud S de G, alors il existe pour chaque composante fortement connexe G_i de G un chemin dans le graphe de dépendance étendu qui contient $\Omega(N^{d_S})$ instances de S pour chaque nœud S de G_i.

PRINCIPE DE LA DÉMONSTRATION. Nous considérons une composante fortement connexe G_i de G et nous construisons un circuit $\mathcal{C}_{1,i}$ qui passe par tous ses nœuds. Nous construisons également un ensemble de circuits $\mathcal{C}_{2,j}$ de G'_i qui forment un multi-circuit de poids nul. Nous parcourons $\Omega(N)$ fois $\mathcal{C}_{1,i}$ et, pour chaque visite, dès que nous rencontrons un circuit $\mathcal{C}_{2,j}$ nous le traversons $\Omega(N)$ fois. La construction est récursive et lors de chaque traversée d'un circuit $\mathcal{C}_{2,j}$ seront visités $\Omega(N)$ fois les hypothétiques circuits $\mathcal{C}_{3,j}$ produits lors de la décomposition de la composante fortement connexe de G'_i contenant $\mathcal{C}_{2,j}$.

L'ensemble des circuits $\mathcal{C}_{2,j}$ est de poids total nul. Par conséquent, parcourir $\Omega(N)$ fois chacun de ces circuits lors du parcours de $\mathcal{C}_{1,i}$ produit un chemin de dépendance de longueur $\Omega(N)$ mais entre deux instances $\boldsymbol{j}$ et $\boldsymbol{i}$ d'un nœud S telles que $\boldsymbol{j} - \boldsymbol{i}$ soit borné indépendamment de N. De ce fait, nous pouvons parcourir le chemin $\mathcal{C}_{1,i}$ $\Omega(N)$ fois et construire une chemin contenant $\Omega(N^2)$ instances de chacun des nœuds des circuits $\mathcal{C}_{2,j}$.

DÉMONSTRATION. La preuve met en œuvre des chemins du graphe de dépendance réduit et du graphe de dépendance étendu. Afin d'éviter toute confusion, les nœuds des premiers chemins seront désignés par la lettre v et ceux des seconds par la lettre $\boldsymbol{p}$ (par abus de notation, nous identifierons un point du GDE et le point du domaine $\mathcal{D}$ qui lui est associé).

Pour tout chemin l de G, $l = \sum_{i=1}^{r} e_i$ où $t(e_i) = q(e_{i+1})$, nous définissons les deux grandeurs $m(l)$ et $M(l)$:

$$m(l) = \max\left\{ t \in \mathbb{Z}^- \mid t\mathbf{1} \leq \sum_{i=1}^{s} \boldsymbol{w}(e_i), \forall s \in [1, r] \right\},$$

$$M(l) = \min\left\{ t \in \mathbb{Z}^+ \mid \sum_{i=1}^{s} \boldsymbol{w}(e_i) \leq t\mathbf{1}, \forall s \in [1, r] \right\},$$

où $\mathbf{1}$ désigne le vecteur dont toutes les composantes sont égales à un. $m(l)$ et $M(l)$ représentent les bornes inférieure et supérieure d'un hypercube de dimension n dans lequel nous pourrons facilement construire un chemin du graphe de dépendance étendu correspondant dans G au chemin l.

À chaque appel, l'algorithme KMW produit un certain nombre de composantes fortement connexes. Pour chacune de ces composantes F, l'étape 2 construit un sous-graphe F'. Par la suite, $H_{i,j}$ désignera le $j^{ème}$ sous-graphe de G généré à la profondeur i. Par exemple, $H_{1,j}$ est la $j^{ème}$ composante fortement connexe de G. À chaque graphe $H_{1,j}$ est associé un sous-graphe $H'_{1,j}$ qui, s'il est non vide et non fortement connexe, générera certains des graphes $H_{2,j}$.

Pour tout entier i strictement inférieur à d ($i < d$), nous associons au graphe $H_{i,j}$ un multi-circuit de poids nul $q_{i,j}$ qui définit[1] $H'_{i,j}$. $q_{i,j}$ définit un circuit $C_{i+1,j'}$ pour chaque composante fortement connexe $H_{i+1,j'}$ de $H'_{i,j}$. Par construction de ces circuits, la somme de leurs poids est nulle : $\sum_{j'} w(C_{i+1,j'}) = Dq_{i,j} = 0$.

1. Pour obtenir un tel multi-circuit il suffit de prendre pour chaque arc de $H'_{i,j}$ un multi-circuit de poids nul qui le contient et de sommer tous ces multi-circuits de poids nul.

Pour chaque composante fortement connexe $H_{1,j}$ de G, nous définissons un circuit $C_{1,j}$ qui visite tous ses nœuds.

Il nous reste encore à définir l'origine $v_{i,j}$ du circuit $C_{i,j}$ considéré comme un chemin :

- Choisir arbitrairement un nœud $v_{1,j}$ dans la $j^{ème}$ composante fortement connexe de G.
- Suivre les arcs de $C_{1,j}$ à partir de $v_{1,j}$. $v_{2,k}$ est alors le premier nœud de $C_{2,k}$ rencontré lors de ce parcours.
- Définir récursivement toutes les origines $v_{i+1,k}$ en parcourant $C_{i,j}$ à partir de $v_{i,j}$.

Finalement, soit $m_{i,j} = m(C_{i,j})$ et $M_{i,j} = M(C_{i,j})$, où les arcs de $C_{i,j}$ sont considérés à partir du nœud $v_{i,j}$.

Nous sommes maintenant à même de construire le chemin désiré. Soit $\mathcal{N}$ un entier strictement positif dont nous fixerons la valeur ultérieurement.

Chemin $C_j(\mathcal{N})$ de G

Dans chaque composante fortement connexe G_j de G, définir un circuit $C_j(\mathcal{N})$ comme suit : parcourir $\mathcal{N}$ fois le circuit $C_{1,j}$ à partir de $v_{1,j}$ et, à chaque fois qu'est atteint pour la première fois un nœud $v_{2,j'}$, traverser $\mathcal{N}$ fois le circuit $C_{2,j'}$ avant de poursuivre plus avant sur $C_{1,j}$. Récursivement, parcourir de même tous les circuits $C_{i,j}$. Un tel chemin traverse $\mathcal{N}^i$ fois le circuit $C_{i,j}$.

Soit S un nœud de G de profondeur $d_S \geq 1$. Par définition de la profondeur de décomposition, il appartient à l'un sous-graphes $C_{d_S,j}$ de G et le chemin $C_j(\mathcal{N})$ contient $\mathcal{N}^{d_S}$ instances de S.

Nous considérons à présent la composante G_{j_0} et son circuit $C_{j_0}(\mathcal{N})$. Nous allons construire un chemin dans le graphe de dépendance étendu, c'est-à-dire une suite de points $\boldsymbol{p}$, *tous contenus dans le domaine $\mathcal{D}$*, qui corresponde au circuit $C_{j_0}(\mathcal{N})$. Cette construction est le point faible de la preuve de Rao qui n'explique pas comment il garantit que son chemin ne sort pas des limites du domaine.

Chemin $P(\mathcal{N})$ du graphe de dépendance étendu

Nous partons d'un point $\boldsymbol{p}(\mathcal{N})$ de $\mathcal{D}$ et nous traduisons le chemin dans G en un chemin dans $\mathcal{D}$ par addition successive de vecteurs de dépendance idoines.

Soit un point $\boldsymbol{p}$ du chemin $P(\mathcal{N})$, soit $\theta(i,j)$ le nombre de parcours de $C_{i,j}$ qui ont été complétés avant que $\boldsymbol{p}$ ne soit atteint et soit $l_{i,j}$ la fraction du circuit $C_{i,j}$ parcourue au moment où $\boldsymbol{p}$ fut rencontré :

$$\boldsymbol{p} = \boldsymbol{p}(\mathcal{N}) + \boldsymbol{w}\left(\sum_{i,j}\theta(i,j)C_{i,j} + \sum_{i,j} l_{i,j}\right) \tag{3.2}$$

(ici $\boldsymbol{p}$ dénote abusivement le point du domaine $\mathcal{D}$ qui est associé au point du graphe de dépendance étendu). Nous devons montrer qu'un tel point $\boldsymbol{p}$ appartient à $\mathcal{D}$ si $\mathcal{N}$ et $p(\mathcal{N})$ sont judicieusement choisis. Pour cela, nous allons évaluer la valeur du point de $\mathcal{D}$ associé à $\boldsymbol{p}$. Remarquez que $\theta(i,j)$ est obligatoirement nul pour les circuits $C_{i,j}$ qui ne participent pas à $C_j(\mathcal{N})$.

Soit i un entier strictement plus grand que 1, $i > 1$. Considérons $H_{i-1,j}$ et les composantes fortement connexes $H_{i,j'}$ de $H'_{i-1,j}$. Le circuit $C_{i-1,j}$ a été entièrement parcouru $\theta(i-1,j)$ fois. Par conséquent, chaque circuit $C_{i,j'}$ a été visité au moins $\mathcal{N} \times \theta(i-1,j)$ fois : $\theta(i,j') \geq \mathcal{N} \times \theta(i-1,j)$. Il a peut-être été parcouru plus souvent si le circuit $C_{i-1,j}$ était suivi au moment où $\boldsymbol{p}$ fut rejoint,

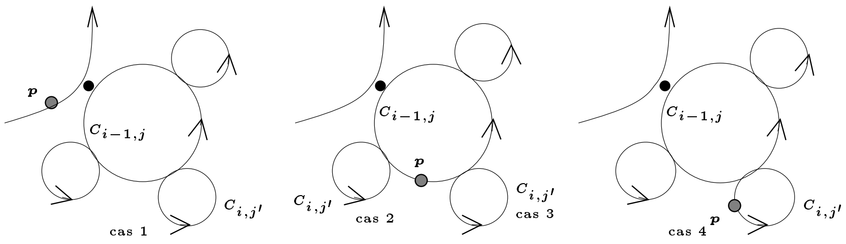


FIG. 3.3: *Positions relatives possibles de* $\boldsymbol{p}$*, de* $C_{i,j}$ *et de* $C_{i+1,j'}$*.*

c'est-à-dire si $l_{i-1,j}$ est non vide (le cas 1 de la figure 3.3 illustre la situation $l_{i-1,j} = \emptyset$). Dans ce cas, lors de la dernière traversée de $C_{i-1,j}$, $C_{i,j'}$ a été de nouveau parcouru un certain nombre de fois, compris entre 0 et $\mathcal{N}$, dépendant de la configuration des différents acteurs (cf. figure 3.3) : suivant que $\boldsymbol{p}$ a été atteint avant (cas 3), pendant (cas 4) ou après (cas 2) d'éventuelles circonvolutions le long de $C_{i,j'}$. Par conséquent, $\theta(i,j') \leq \mathcal{N} \times \theta(i-1,j) + \mathcal{N}$. Précisément, il ne peut y avoir égalité dans la dernière inéquation que si $C_{i,j'}$ a été entièrement visité, c'est-à-dire que si $l_{i,j'}$ est vide.

Nous posons $\theta'(i,j') = \theta(i,j') - \mathcal{N} \times \theta(i-1,j)$ pour tout entier i strictement supérieur à un et $\theta'(1,j) = \theta(1,j)$. Nous déduisons de l'étude précédente que :

$$\theta'(i,j') \leq \mathcal{N} \text{ si } l_{i,j'} = \emptyset \text{ et } \theta'(i,j') \leq \mathcal{N} - 1 \text{ sinon.}$$

Nous rappelant que pour tout graphe $H_{i-1,j}$ la somme $\sum_{j'} \boldsymbol{w}(C_{i,j'})$ est nulle par construction, nous obtenons de ce qui précède pour tout $i > 1$:

$$\begin{aligned} \boldsymbol{w}\left(\textstyle\sum_{j'} \theta'(i,j')C_{i,j'}\right) &= \boldsymbol{w}\left(\textstyle\sum_{j'}(\theta(i,j') - \mathcal{N}\theta(i-1,j))C_{i,j'}\right) \\ &= \boldsymbol{w}\left(\textstyle\sum_{j'} \theta(i,j')C_{i,j'}\right) - \mathcal{N}\theta(i-1,j) \times 0 \quad \Longrightarrow \end{aligned}$$

$$\boldsymbol{w}\left(\sum_{j'} \theta'(i,j')C_{i,j'}\right) = \boldsymbol{w}\left(\sum_{j'} \theta(i,j')C_{i,j'}\right) \tag{3.3}$$

De plus, l'égalité 3.3 est vérifiée pour $i = 1$ par définition de $\theta'(1,j)$. En sommant les équations 3.3 pour i variant de 1 à $d_{G_{j_0}}$, nous établissons l'égalité :

$$\boldsymbol{w}\left(\sum_{i,j} \theta(i,j)C_{i,j} + \sum_{i,j} l_{i,j}\right) = \boldsymbol{w}\left(\sum_{i,j} \theta'(i,j)C_{i,j} + \sum_{i,j} l_{i,j}\right).$$

Finalement, grâce aux inégalités ci-dessous :

- $m_{i,j}\mathbf{1} \leq \boldsymbol{w}(C_{i,j}) \leq M_{i,j}\mathbf{1}$.
- $m_{i,j}\mathbf{1} \leq \boldsymbol{w}(l_{i,j}) \leq M_{i,j}\mathbf{1}$.
- $\theta'(i,j') \leq \mathcal{N}$ si $l_{i,j'} = \emptyset$ et $\theta'(i,j') \leq \mathcal{N} - 1$ sinon.

nous déduisons de ce qui précède l'inégalité :

$$\left(\mathcal{N} \times \sum_{i,j} m_{i,j}\right)\mathbf{1} \leq \boldsymbol{w}\left(\sum_{i,j} \theta(i,j)C_{i,j} + \sum_{i,j} l_{i,j}\right) \leq \left(\mathcal{N} \times \sum_{i,j} M_{i,j}\right)\mathbf{1}$$

qui implique :

$$\boldsymbol{p}(\mathcal{N}) + \mathcal{N}\left(\sum_{i,j} m_{i,j}\right)\mathbf{1} \leq \boldsymbol{p} \leq \boldsymbol{p}(\mathcal{N}) + \mathcal{N}\left(\sum_{i,j} M_{i,j}\right)\mathbf{1} \tag{3.4}$$

Le domaine $\mathcal{D}$ contient un hypercube de taille $\Omega(N)$ par hypothèse. Soient $p_0(N)$ l'origine de cet hypercube et $D(N)$ sa taille : $p_0(N) \leq \boldsymbol{p} \leq p_0(N) + D(N) \Rightarrow \boldsymbol{p} \in \mathcal{D}$. Nous posons $K = \sum_{i,j}(M_{i,j} - m_{i,j})$. K est donc indépendant de N. Nous posons également $\mathcal{N} = \lfloor \frac{D(N)}{K} \rfloor$ et $\boldsymbol{p}(\mathcal{N}) = -\mathcal{N} \times \sum_{i,j} m_{i,j}\mathbf{1} + p_0(N)$. Avec ces définitions, tout point $\boldsymbol{p}$ vérifiant l'équation (3.4) appartient au domaine $\mathcal{D}$. Comme $D(N) = \Omega(N)$, nous avons donc construit un chemin du graphe de dépendance étendu qui visite $\Omega(N^{d_S})$ fois le nœud S de G_{j_0}, pour une composante fortement connexe G_{j_0} quelconque. □

3.2.3 Construction de G'

Nous nous intéressons maintenant à la construction de G', clef de voûte de l'algorithme de Karp, Miller et Winograd. Nous allons montrer que G' peut être obtenu par résolution d'un simple programme linéaire en nombre rationnels.

Construction théorique

Nous allons prouver que les arcs de G' sont très exactement les arcs e_i pour lesquels la variable v_i associée est nulle dans toute solution optimale rationnelle du programme linéaire (3.5) :

$$\min\left\{ \ \textstyle\sum_i v_i \quad | \quad q \geq 0, \quad v \geq 0, \quad q + v \geq 1, \quad Bq = 0 \ \right\} \tag{3.5}$$

Le vecteur q, transformé en vecteur entier par multiplication éventuelle par un entier, représente un multi-circuit de poids nul, comme au paragraphe 3.2.1. Remarquons préalablement que ce programme linéaire admet toujours une solution définie par le couple $q = 0, v = 1$. Dans toute solution optimale, les valeurs respectives de q et v sont reliées par les correspondances suivantes :

Lemme 6. *Dans toute solution optimale rationnelle (q, v) du programme 3.5 :*

- $q_i \neq 0 \Leftrightarrow v_i = 0$.
- $q_i = 0 \Leftrightarrow v_i = 1$.

Démonstration. Considérons une solution optimale rationnelle (q, v). L'unique contrainte sur v_i provient de l'inéquation $q_i + v_i \geq 1$. Par conséquent, $q_i = 0 \Rightarrow v_i \geq 1 \Rightarrow v_i = 1$ par minimalité de la somme $\sum_i v_i$. De plus, $v_i = 0 \Rightarrow q_i \neq 0$. Il nous reste à montrer que $q_i \neq 0 \Rightarrow v_i = 0$.

Soit q_j la plus petite composante non nulle de q : $q_j = \min_i\{q_i \mid q_i \neq 0\}$. Nous construisons une nouvelle solution (q', v') définie par $q' = \frac{1}{q_j}q$, $v'_i = 0$ si q_i est non nul et $v'_i = v_i$ sinon. Les composantes non nulles de q' sont supérieures à un par définition de q_j, ce qui nous assure que l'inégalité $q' + v' \geq 1$ est satisfaite. Pour tout i, nous avons $v'_i \leq v_i$. Par construction, nous avons diminué la valeur de la fonction objective, $\sum_i v'_i \leq \sum_i v_i$. Si $v' \neq v$, nous avons trouvé une meilleure solution, au sens strict, ce qui est impossible puisque nous sommes parti d'une solution optimale. Par conséquent $v' = v$ et v_i est nul quand q_i est non nul : $q_i \neq 0 \Rightarrow v_i = 0$. □

Nous sommes maintenant à même de relier G' et le programme linéaire.

Lemme 7. *Dans toute solution optimale rationnelle (q, v) du programme 3.5 :*

- $v_i = 0 \Leftrightarrow e_i \in G'$

Démonstration. Soit (q, v) une solution du programme linéaire 3.5. q, multiplié par un entier idoine définit un multi-circuit de poids nul. Donc, grâce au lemme 6, $v_i = 0 \Rightarrow q_i \neq 0 \Rightarrow e_i \in G'$.

Réciproquement, soit e_i un arc de G'. Par définition de G', il existe un multi-circuit de poids nul $\tilde{q}$ (c'est-à-dire un vecteur entier $\tilde{q}$ vérifiant $B\tilde{q} = 0$) contenant e_i : $\tilde{q}_i > 0$ (en fait $\tilde{q}_i \geq 1$ car $\tilde{q}$ est entier). Considérons une solution optimale (q, v) du programme 3.5. Nous formons une nouvelle solution (q', v') définie par $q' = q + \tilde{q}$, $v'_j = v_j$ si $j \neq i$ et $v'_i = 0$. Cette solution est meilleure sauf si $v_i = 0$. □

Par conséquent, la résolution en rationnels du programme linéaire 3.5 nous fournit directement G' : les arcs de G' sont les arcs e_i pour lesquels $v_i = 0$.

Construction pratique

Nous proposons ici un programme linéaire qui permet de calculer G' mais qui met en jeux moins de variables que le programme 3.5. G' est le sous-graphe de G engendré par les arcs participant à des multi-circuits de poids nul. Par définition, un multi-circuit est une union de circuits élémentaires. Rechercher l'ensemble des arcs inclus dans des multi-circuits de poids nul équivaut alors à rechercher les circuits élémentaires contenus dans des unions de poids nul. Dans le cas général, il n'est pas réaliste de penser générer tous les circuits élémentaires puisque leur quantité peut être exponentielle en le nombre de nœuds et d'arcs du graphe. Ce problème est aisément contourné par l'introduction de *base de cycles*, dont nous rappelons ici la définition.

Définition 16 (Cycles indépendants, base de cycles, nombre cyclomatique). *p cycles $\mu^1, \ldots, \mu^p$ sont dits* dépendants *s'il existe une relation vectorielle de la forme :*

$$\sum_{i=1}^{p} \lambda_i \boldsymbol{\mu}^i = 0$$

où $\boldsymbol{\mu}^i$ est le vecteur associé au cycle μ^i, c'est-à-dire le vecteur de dimension $|E|$ dont la $j^{\text{ème}}$ composante indique par sa valeur absolue le nombre d'occurrences de l'arc e_j dans μ^i et par son signe le sens dans lequel μ^i parcourt e_j. Des cycles qui ne sont pas dépendants sont dits indépendants*.*

Une base de cycles *est un ensemble minimal de cycles indépendants, $\mu^1, \ldots, \mu^p$, tel que tout vecteur représentatif d'un cycle μ puisse se mettre sous la forme :*

$$\boldsymbol{\mu} = \sum_{i=1}^{p} \lambda_i \boldsymbol{\mu}^i.$$

Le nombre cyclomatique *$\nu(G)$ du graphe est la dimension d'une base de cycles. Il vaut : $\nu(G) = |E| - |V| + c$, où c est le nombre de composantes connexes.*

Rappelons nous qu'un circuit est un cycle, mais que la réciproque est fausse : un cycle peut traverser un arc (orienté) de la tête vers la queue, c'est-à-dire à contresens (cf. [36] pour une définition formelle). Un multi-circuit étant une union de cycles, il est également décomposable sur une base de cycles.

La méthode classique de construction d'une base de cycles utilise un arbre recouvrant maximal, les deux objets étant duaux [36]. Ce procédé prouve au passage qu'une base de cycles contient exactement $\nu(G) = p + |E| - |V|$ éléments ($\nu(G)$ est le *nombre cyclomatique* du graphe G), où p est le nombre de composantes connexes du graphe. Comme nous travaillons toujours sur graphes réduits contenant une une unique composante fortement connexe, p vaudra invariablement un dans notre cas.

Dans le programme linéaire 3.5, les contraintes $Cq = 0$ et $q \geq 0$ nous assure que la variable q représente un multi-circuit. q peut donc être décomposé sur la base de cycles $\mathcal{BC} = \{C_1, \dots, C_{\nu(G)}\}$: $q = \sum_{i=1}^{\nu(G)} \lambda_i C_i$, pour certains rationnels $\lambda_1, \dots, \lambda_{\nu(G)}$. Nous obtenons ainsi un nouveau programme linéaire :

$$\min\left\{ \textstyle\sum_i v_i \quad | \quad Q\lambda \geq 0 \quad v \geq 0, \quad Q\lambda + v \geq 1, \quad \mathcal{W}\lambda = 0 \right\} \tag{3.6}$$

où Q dénote la matrice $|E| \times \nu(G)$ dont les colonnes sont les cycles C_i de la base et où $\mathcal{W}$ représente la matrice $n \times \nu(G)$ dont les colonnes sont les poids des cycles C_i. Autrement dit, $\mathcal{W} = WQ$. Nous avons donc remplacé les contraintes $q \geq 0$ et $Cq = 0$ par l'inéquation $Q\lambda \geq 0$. Les arcs de G' sont encore exactement les arcs e_i vérifiant $v_i = 0$ dans une quelconque solution optimale du programme linéaire (3.6). Ce nouveau programme linéaire comprend $2|E| - |V| + 1$ variables (au lieu de $2|E|$ pour le programme (3.5)) et $3|E| + n$ égalités et inégalités (au lieu de $3|E| + n$).

3.3 Ordonnancement

Nous allons montrer au paragraphe 3.3.1 que la construction de G' est intimement reliée à la construction des hyperplans (faiblement) séparateurs de G, ce qui nous amènera tout naturellement à proposer au paragraphe 3.3.2 un algorithme d'ordonnancement des systèmes d'équations récurrentes uniformes.

3.3.1 Hyperplans séparateurs

Pour mieux comprendre le programme linéaire 3.5, nous considérons son dual, au sens de la dualité de la programmation linéaire (nous renvoyons au livre d'Alexander Schrijver [74] pour tous les résultats de programmation linéaire). Pour cela, nous réécrivons le programme 3.5 originel sous forme canonique :

$$\min\left\{ \textstyle\sum_i v_i \quad | \quad q \geq 0, \quad v \geq 0, \quad w \geq 0, \quad q + v = 1 + w, \quad Bq = 0 \right\}. \tag{3.7}$$

Son dual peut alors s'écrire :

$$\max\left\{ \textstyle\sum_i z_i \quad | \quad 0 \leq z_i \leq 1, \quad X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} \geq z_i \right\} \tag{3.8}$$

où l'inégalité $z_i \geq 0$ correspond (au sens de la propriété sur les écarts complémentaires ou *complementary slackness* en anglais) à la variable w_i, $z_i \leq 1$ à v_i et $X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} \geq z_i$ à la variable q_i. Le programme dual à la propriété suivante :

Lemme 8. *Dans toute solution optimale (z, X, ρ) du programme dual 3.8 :*

- $e_i \in G' \Leftrightarrow X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} = 0$
- $e_i \notin G' \Leftrightarrow X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} \geq 1$

DÉMONSTRATION. La preuve repose sur le théorème des écarts complémentaires [74] (*complementary slackness theorem* en anglais), qui relie les inégalités du dual strictement satisfaites et la nullité de variables du primal.

Variable v : il existe une solution optimale du programme 3.8 avec $z_i < 1$ si et seulement si il n'existe pas de solution optimale du programme 3.7 avec $v_i > 0$, c'est-à-dire si et seulement toute solution optimale du programme 3.7 vérifie l'égalité $v_i = 0$, c'est-à-dire si et seulement si $e_i \in G'$ (lemme 7).

Variable w : $q_i + v_i \geq 1$ est toujours une égalité (c'est-à-dire $w_i = 0$) quand $q_i = 0$. Quand $q_i \neq 0$, il existe une solution optimale avec $q_i + v_i > 1$ (par simple multiplication de q par un nombre positif suffisamment grand). Par le théorème des écarts complémentaires, il existe une solution optimale de 3.8 avec $z_i > 0$ si et seulement si il n'existe pas de solution optimale du programme 3.7 avec $w_i > 0$, c'est-à-dire si et seulement si toute solution optimale du programme 3.7 satisfait $q_i = 0$, c'est-à-dire si et seulement si $e_i \notin G'$.

Variable q : il existe une solution optimale du programme 3.8 avec une inégalité stricte dans la contrainte $X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} \geq z_i$, si et seulement si il n'existe pas de solution optimale du programme 3.7 avec $q_i > 0$, c'est-à-dire si et seulement si $e_i \notin G'$.

Ceci nous permet d'établir le résultat. □

Le lemme 8 nous montre que le programme linéaire dual nous procure des hyperplans séparateurs qui sont strictement séparateurs pour les arcs qui ne participent pas à des multi-circuits de poids nul et qui sont faiblement séparateurs pour les arcs de G'. De plus, pour tout graphe apparaissant dans la décomposition, ces hyperplans sont stricts le plus souvent possible : ils sont fortement séparateurs pour le plus grand nombre d'arcs possible. Ce caractère glouton fait naturellement penser à l'algorithme de Paul Feautrier (section 2.4). Par contre le programme dual fait apparaître des hyperplans faiblement séparateurs, c'est-à-dire des germes d'ordonnancements affines de *partie linéaire unique*. C'est pourquoi, dans ce chapitre et le suivant, nous ne rechercherons jamais d'ordonnancements affines pour lesquels chaque instruction a sa partie linéaire propre, comme le fait Paul Feautrier. Il ne s'agira pas là d'un parti pris, mais de l'utilisation d'une propriété mathématique des systèmes que nous étudions.

Ce lemme peut être généralisé au cas où un coût $\lambda_i > 0$ (« délai » le long de e_i) est associé à chaque arc e_i :

Lemme 9. *Soit $G = (V, E)$ un graphe de dépendance. Pour tout $|E|$-uplet de valeurs strictement positives $\lambda_1, \ldots, \lambda_{|E|}$, il existe des constantes $\rho_1, \rho_2, \ldots, \rho_{|V|}$ et un vecteur X tels que :*

- $e_i \in G' \Leftrightarrow X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} = 0$
- $e_i \notin G' \Leftrightarrow X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} \geq \lambda_i$

DÉMONSTRATION. Le programme linéaire :

$$\min \left\{ \textstyle\sum_i \lambda_i v_i \;\mid\; q \geq 0, \;\; v \geq 0, \;\; w \geq 0, \;\; q + v = 1 + w, \;\; Bq = 0 \right\} \tag{3.9}$$

a les mêmes solutions optimales que le programme (3.7) si les λ sont tous strictement positifs. Son programme dual s'écrit :

$$\max \left\{ \textstyle\sum_i z_i \;\mid\; 0 \leq z_i \leq \lambda_i, \;\; X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} \geq z_i \right\} \tag{3.10}$$

La preuve est alors similaire à la preuve du lemme 8. □

Plus important, durant l'algorithme de décomposition, il est possible d'associer à un nœud v de G, une suite d'hyperplans $X_v^{(1)}, \ldots, X_v^{(d_v)}$ obtenue grâce au programme dual 3.8 (d_v est la profondeur de décomposition de v). Si x est un nœud ou un arc de G, nous notons $G_x^{(1)}$ la composante fortement connexe de G qui contient x et $G_x^{(i)}$ le sous-graphe de G généré à profondeur i et incluant x. $G_x^{(i)}$ est une composante fortement connexe de $(G_x^{(i-1)})'$. La suite d'hyperplans a une propriété importante :

Théorème 5 (Indépendance des hyperplans).
Les $d_v - 1$ premiers hyperplans, $X_v^{(1)}, \ldots, X_v^{(d_v-1)}$, associés au nœud v de G sont linéairement indépendants. De plus, si $G_v^{(d_v)}$ est calculable (et donc aussi si G est calculable) tous les hyperplans, $X_v^{(1)}, \ldots, X_v^{(d_v)}$, associés à v sont linéairement indépendants.

DÉMONSTRATION. Soit $\rho_w^{(i)}$ les constantes associées au vecteur $X_v^{(i)}$ dans le programme linéaire dual 3.8. Supposons qu'il existe un entier $k \leq d_v$, tel que les vecteurs $X_v^{(1)}, \ldots, X_v^{(k-1)}$ soient linéairement indépendants et les vecteurs $X_v^{(1)}, \ldots, X_v^{(k-1)}, X_v^{(k)}$ liés. Alors, soit $X_v^{(k)} = 0$ si $k = 1$, soit il existe des nombres rationnels $\{\lambda_i\}_{1 \leq i \leq k-1}$ tels que $X_v^{(k)} = \sum_{i=1}^{k-1} \lambda_i X_v^{(i)}$.

Soit $\mathcal{C}$ un circuit de $G_v^{(k)}$. Montrons que $X_v^{(k)}$ induit un délai nul sur $\mathcal{C}$: $X_v^{(k)}\boldsymbol{w}(\mathcal{C}) = 0$. La propriété est trivialement vérifiée si $k = 1$ puisque $X_v^{(k)} = 0$ dans ce cas. Sinon, par construction des vecteurs $X_v^{(i)}$ le lemme 8 nous assure l'égalité :

$$\forall i \in [1, k-1] \forall e \in G_v^{(k)} \ : \ X_v^{(i)}\boldsymbol{w}(e) - \rho_{t(e)}^{(i)} + \rho_{q(e)}^{(i)} = 0 \tag{3.11}$$

puisque nous avons l'inclusion $G_v^{(k)} \subset (G_v^{(i)})'$ pour i compris entre 1 et $k-1$.

Sommant, à i fixé, les équations (3.11) le long de $\mathcal{C}$, nous obtenons l'égalité $X_v^{(i)}\boldsymbol{w}(\mathcal{C}) = 0$ pour tout i, $1 \leq i \leq k-1$. Comme $X_v^{(k)}$ est combinaison linéaire des vecteurs $X_v^{(i)}$, $X_v^{(k)}\boldsymbol{w}(\mathcal{C}) = 0$. Cependant, par construction de $X_v^{(k)}$, nous avons l'inégalité :

$$\forall e \in G_v^{(k)} \ : \ X_v^{(k)}\boldsymbol{w}(e) - \rho_{t(e)}^{(i)} + \rho_{q(e)}^{(i)} \geq z_e \tag{3.12}$$

où $z_e = 1$ si $e \notin (G_v^{(k)})'$ et $z_e = 0$ si $e \in (G_v^{(k)})'$. L'inéquation (3.12) montre que $X_v^{(k)}\boldsymbol{w}(\mathcal{C})$ est supérieur au nombre d'arcs de $\mathcal{C}$ qui n'appartiennent pas à $(G_v^{(k)})'$. Par conséquent, $X_v^{(k)}\boldsymbol{w}(\mathcal{C})$ étant nul, tous les arcs de $\mathcal{C}$ sont contenus dans $(G_v^{(k)})'$, et ce pour tout circuit $\mathcal{C}$ de $G_v^{(k)}$. $G_v^{(k)}$ étant fortement connexe par construction, $G_v^{(k)} = (G_v^{(k)})'$ et $(G_v^{(k)})'$ est fortement connexe. L'algorithme de décomposition s'est donc arrêté ici à la profondeur k, donc $k = d_v$, et il a conclu que $G_v^{(d_v)}$ n'est pas calculable et donc que G ne l'est pas non plus. □

Construction pratique des hyperplans et des constantes

Tout comme nous l'avons fait pour la construction de G', nous introduisons des bases de cycles dans l'espoir de réduire la complexité des programmes linéaires utilisés. Le programme linéaire (3.13), dual du programme (3.6), nous permet de calculer le vecteur X, mais pas les constantes ρ, contrairement au programme (3.8).

$$\max \left\{ \ \textstyle\sum_i z_i \ \ \middle| \ \ x \geq 0, \ \ 0 \leq z \leq 1, \ \ X\mathcal{W} = (x+z)Q \ \right\} \tag{3.13}$$

Ce programme contient $2|E| + n$ variables (au lieu de $n + |E| + |V|$) et $4|E| - |V| + 1$ égalités et inégalités (au lieu de $3|E|$). Le progrès n'est pas évident ! De plus, il nous reste encore à calculer les constantes ρ. Nous ne nous attardons pas sur la complexité de résolution des programmes linéaires qui dépend aussi des logiciels utilisés. Le principal avantage de cette approche réside dans la détermination séparée des valeurs des constantes, la méthode proposée nous en garantissant une certaine régularité dont l'intérêt deviendra évident quand il s'agira d'appliquer ces résultats à la génération d'ordonnancements de nids de boucles. Dans ce cas, la régularité et la propreté du code généré sont importantes.

Remarquons en outre que la valeur des variables z_i du programme (3.13) nous est donnée par le programme primal (3.6). Nous pouvons donc attribuer une autre fonction objective à notre programme linéaire et optimiser comme bon nous semble le choix du vecteur X.

Le calcul des constantes se fait au moyen de l'algorithme de Bellman et Ford [24] qui résout le problème des plus courts chemins dans les graphes contenant une unique source. Cet algorithme détermine s'il existe un circuit de poids strictement négatif atteignable à partir de la source. Si le graphe ne contient pas un tel circuit, l'algorithme produit les plus courts chemins et leurs poids en $O(|V||E|)$ étapes. Nous construisons une copie H du graphe G. Nous lui rajoutons un nœud « source » et un arc de poids nul de ce nœud vers chacun des autres nœuds du graphe. Nous remplaçons le vecteur de dépendance $\boldsymbol{w}(e)$ associé à chaque arc e par le délai induit $\boldsymbol{w}'(e) : \boldsymbol{w}'(e) = X\boldsymbol{w}(e_i) - z_i$. Comme nous voulons utiliser l'algorithme de Bellman et Ford, nous devons prouver que H ne contient pas de circuits de poids strictement négatif.

Lemme 10. *Tout circuit $\mathcal{C}$ de H est à poids positif.*

Démonstration. Soit un circuit $\mathcal{C}$ de H. $\mathcal{C}$ se décompose sur la base de cycles $\mathcal{BC} : \mathcal{C} = Q\lambda$, λ étant un vecteur rationnel. Le poids de $\mathcal{C}$ vaut :

$$\boldsymbol{w}'(\mathcal{C}) = \boldsymbol{w}'(Q\lambda) = (X\mathcal{W} - zQ)\lambda = xQ\lambda = x\mathcal{C}.$$

Les vecteurs x et $\mathcal{C}$ étant tous deux positifs par définition, $x\mathcal{C}$ est positif. □

Nous pouvons donc utiliser l'algorithme de Bellman et Ford sur H et nous obtenons ce faisant les plus courts chemins et leurs poids. Nous notons ρ_S l'opposé du poids du plus court chemin menant au nœud S.

Lemme 11. *Le vecteur X et les constantes ρ_S forment une solution optimale du programme 3.13.*

Démonstration. Soit l'arc e_i de G. Par définition des plus courts chemins, nous avons :

$$(-\rho_{t(e_i)}) \leq (-\rho_{q(e_i)}) + \boldsymbol{w}'(e_i) \Leftrightarrow X\boldsymbol{w}(e_i) + \rho_{t(e_i)} - \rho_{q(e_i)} \geq z_i.$$

□

Remarque : Le programme linéaire 3.13 montre clairement que seuls les circuits[2] du graphe de dépendance jouent un rôle dans le choix de la partie linéaire de l'ordonnancement et que les valeurs des dépendances entre instructions importe peu. En d'autres termes, seules les auto-dépendances d'une instruction comptent pour son ordonnancement. Lors de la recherche d'un ordonnancement,

2. Dans ce programme linéaire, nous n'avons pas des circuits mais des cycles. Cependant, dans un graphe fortement connexe il est possible de prendre une base de cycles qui soit uniquement composée de circuits. Nous verrons, à la section 4.3.3, une autre caractérisation des parties linéaires des ordonnancements énoncée exclusivement au moyen de circuits élémentaires.

il faut donc considérer le poids des circuits de dépendance et non des arcs. Les valuation des arcs n'interviendront que pour le choix des constantes. Les arcs de dépendance n'influent donc sur le parallélisme présent qu'au travers des circuits qui les comprend. Nous sommes très loin de l'approche de Wolf et Lam où seules les valeurs des arcs étaient considérées. Nous reviendrons sur ce point aux sections 4.3.3 et 4.4.1.

3.3.2 Ordonnancement multidimensionnel

Nous sommes maintenant à même de proposer un algorithme d'ordonnancement des systèmes d'équations récurrentes uniformes calculables qui est optimal au sens défini à la section 1.4.

Construction des ordonnancements

Nous considérons un graphe de dépendance G, supposé fortement connexe (nous indiquerons ultérieurement comment traiter les graphes non fortement connexes). Pour l'ordonnancer, nous appliquons à G l'algorithme de décomposition, en nous intéressant cette fois-ci au programme linéaire dual 3.8.

Nous construisons un ordonnancement pour G en sélectionnant, pour chacun de ses nœuds v, les suites de vecteurs $(X_v^{(1)}, \dots, X_v^{(d_v)})$ et de constantes $(\rho_v^{(1)}, \dots, \rho_v^{(d_v)})$ produites par le programme (3.8) tout au long de la décomposition. Pour chaque nœud, nous complétons ces suites par le nombre de zéros nécessaire à l'obtention de suites de longueur n. L'algorithme résultant est invoqué par l'appel ORDONNANCEUR_DE_SERU $(G,1)$.

booléen ORDONNANCEUR_DE_SERU $(G,\ k)$

1. Décomposer G en ses s composantes fortement connexes G_1, ...G_i, ..., G_s. Appeler les étapes 2, 3 puis 4 sur chacun de ces sous-graphes G_i et renvoyer le ET logique des réponses.

2. Construire G'_i le sous-graphe de G_i généré par les arcs de G_i qui appartiennent à au moins un multi-circuit de poids nul.

3. Choisir un vecteur X et, pour chaque nœud v de G_i, une constante ρ_v tels que :

   $$\begin{cases} e = (q(e), t(e)) \in G'_i & \Rightarrow \quad X\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} = 0 \\ e = (q(e), t(e)) \notin G'_i & \Rightarrow \quad X\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 1 \end{cases}$$

   Pour tout nœud v de G_i, poser $\rho_v^{(k)} = \rho_v$ et $X_v^{(k)} = X$.

4. – Si G'_i est vide, renvoyer VRAI.
   – Si G'_i est fortement connexe , renvoyer FAUX.
   – Sinon, renvoyer ORDONNANCEUR_DE_SERU $(G'_i,\ k+1)$.

La correction de l'algorithme est immédiate et découle de la correction de l'algorithme KMW et du lemme 8. Il nous faut juste prouver que la fonction affine multidimensionnelle construite est bien un ordonnancement.

Lemme 12. *Si G est calculable, la fonction multidimensionnelle Θ produite par l'algorithme* ORDONNANCEUR_DE_SERU *:*

$$\begin{array}{rcl} V \times \mathcal{D} & \rightarrow & \mathbb{Z}^n \\ (v,\ \boldsymbol{p}) & \mapsto & (X_v^{(1)}\boldsymbol{p} + \rho_v^{(1)},\ \dots,\ X_v^{(d_v)}\boldsymbol{p} + \rho_v^{(d_v)},\ 0,\ \dots,\ 0) \end{array}$$

définit un ordonnancement multidimensionnel.

DÉMONSTRATION. Soit e un arc de G. Nous devons prouver que, pour tout point $\boldsymbol{p}$ du domaine ($\boldsymbol{p} \in \mathcal{D}$) tel que $\boldsymbol{p} - \boldsymbol{w}(e)$ soit également point du domaine ($\boldsymbol{p} - \boldsymbol{w}(e) \in \mathcal{D}$), l'équation suivante est vérifiée :

$$\lfloor T(t(e), \boldsymbol{p}) \rfloor \succ \lfloor T(q(e), \boldsymbol{p} - \boldsymbol{w}(e)) \rfloor .$$

Pour ce faire nous allons établir l'existence d'un entier k, $k \in [1 \ldots n]$, tel que :

$$\begin{cases} X^i_{t(e)}\boldsymbol{p} + \rho^i_{t(e)} \geq X^i_{q(e)}(\boldsymbol{p} - \boldsymbol{w}(e)) + \rho^i_{q(e)} & \text{pour tout } i, \ 1 \leq i \leq k-1 \\ X^k_{t(e)}\boldsymbol{p} + \rho^k_{t(e)} \geq X^k_{q(e)}(\boldsymbol{p} - \boldsymbol{w}(e)) + \rho^k_{q(e)} + 1 \end{cases}$$

ce qui démontrera la propriété recherchée.

G étant calculable, les appels récursifs terminaux de l'algorithme de décomposition aboutissent tous à des graphes ne contenant pas de multi-circuits de poids nul. Pour tout arc e de G il existe donc un niveau minimal k pour lequel e n'appartient pas à un multi-circuit de poids nul : $e \in G_e^{(k)}$ et $e \notin (G_e^{(k)})'$. Par construction, les nœuds extrémités de e, $t(e)$ et $q(e)$, appartiennent aux mêmes sous-graphes générés par la décomposition de G, au moins jusqu'à la profondeur k incluse. Par conséquent, leurs suites respectives de vecteurs, $X^{(i)}_{t(e)}$ et $X^{(i)}_{q(e)}$ coïncident au moins jusqu'au rang k :

$$X^{(1)}_{t(e)} = X^{(1)}_{q(e)} = X^{(1)}, \ \ldots, \ X^{(k)}_{t(e)} = X^{(k)}_{q(e)} = X^{(k)}.$$

De plus, par définition de k, les vecteurs $X^{(i)}$ sont faiblement séparateurs pour e jusqu'au niveau $k-1$ et ne sont strictement séparateurs qu'au niveau k. Pour i inférieur à $k-1$, la $i^{ème}$ composante de $\Theta(t(e), \boldsymbol{p}) - \Theta(q(e), \boldsymbol{p} - \boldsymbol{w}(e))$ vaut donc :

$$\left(X^{(i)}\boldsymbol{p} + \rho^{(i)}_{t(e)}\right) - \left(X^{(i)}\left(\boldsymbol{p} - \boldsymbol{w}(e)\right) + \rho^{(i)}_{q(e)}\right) = X^{(i)}\boldsymbol{w}(e) + \rho^{(i)}_{t(e)} - \rho^{(i)}_{q(e)} = 0$$

et la $k^{ème}$:

$$\left(X^{(k)}\boldsymbol{p} + \rho^{(k)}_{t(e)}\right) - \left(X^{(k)}\left(\boldsymbol{p} - \boldsymbol{w}(e)\right) + \rho^{(k)}_{q(e)}\right) = X^{(k)}\boldsymbol{w}(e) + \rho^{(k)}_{t(e)} - \rho^{(k)}_{q(e)} \geq 1$$

Finalement, quelles que soient les valeurs des suites $X^{(i)}_{t(e)}$, $X^{(i)}_{q(e)}$, $\rho^{(i)}_{t(e)}$ et $\rho^{(i)}_{q(e)}$ au delà du rang k, nous avons bien la propriété désirée. □

Les ordonnancements obtenus ne sont pas tout à fait quelconques : ce sont des ordonnancements affines dont la $i^{ème}$ partie linéaire est commune aux nœuds d'une même composante fortement connexe générée à la profondeur i. Ces ordonnancement affines sont à partie linéaire *localement* unique. Cette propriété de régularité est importante en pratique. Les ordonnancements obtenus étant relativement simples, la génération de code pour les nids de boucles ne sera pas trop compliquée et les codes produits seront relativement réguliers (cf. [13, 14]).

Il nous reste à considérer le cas de graphes non fortement connexes. Nous appliquons la même technique qu'Allen et Kennedy et que Feautrier : nous commençons par ordonner topologiquement les composantes fortement connexes et nous ordonnançons ensuite chaque composante fortement connexe au moyen de ORDONNANCEUR_DE_SERU, comme s'il s'agissait d'un graphe fortement connexe. Nous nous contentons donc de séquentialiser, si besoin est, les différentes composantes fortement connexes de G, ce qui n'est pas forcément la solution la plus efficace.

Efficacité des ordonnancements

Nous montrons ici que la dimension de nos ordonnancements est minimale.

Lemme 13. *Si le domaine est contenu dans un hypercube de dimension n et de taille $O(N)$ et si d_S est la profondeur de décomposition du nœud S de G, alors l'ordonnancement multidimensionnel associé à S à une latence en $O(N^{d_S})$.*

DÉMONSTRATION. $\mathcal{D}$ est contenu dans un hypercube de dimension n et de taille au plus λN pour une certaine constante λ. La latence d'un ordonnancement affine de partie linéaire non nulle est alors au plus linéaire en N et la latence d'un ordonnancement multidimensionnel de dimension k est alors en $O(N^k)$. □

Le théorème 4 et le lemme 13 montrent que les ordonnancements multidimensionnels que nous obtenons sont optimaux :

Corollaire 4 (Optimalité de l'ordonnancement).
L'ordonnancement proposé est optimal : si le domaine contient (resp. est contenu dans) un hypercube de dimension n et de taille $\Omega(N)$ (resp. $O(N)$) et si d est la profondeur de décomposition de G, alors la latence de l'ordonnancement est en $O(N^d)$ et la longueur du plus long chemin en $\Omega(N^d)$.
Plus précisément, si d_S est la profondeur de décomposition du nœud S de G, alors l'ordonnancement associé à S à une latence en $O(N^{d_S})$ et il existe dans le graphe de dépendance apparent un chemin qui contient $\Omega(N^{d_S})$ occurrences de S.

Plus simplement, les chemins de dépendance exhibés par le théorème 4 définissaient une borne inférieure sur la séquentialité du système d'équations récurrentes uniformes. Nos ordonnancements atteignent cette borne, en ordre de grandeur. Les équations récurrentes contiennent donc exactement $(n-d)$ degrés de parallélisme et nous sommes capables de les expliciter au moyen d'un ordonnancement régulier.

Question : est-il possible d'être plus précis sur le facteur multiplicatif de N^d ? En d'autres termes, est-il possible de construire un ordonnancement de latence équivalente à kN^d chaque fois que le plus long chemin de dépendance est de longueur équivalente à kN^d, c'est-à-dire d'avoir un résultat aussi précis pour les ordonnancements multidimensionnels que pour les ordonnancements monodimensionnels ?

3.3.3 Complexité de l'algorithme d'ordonnancement

Nous présentons ici une borne supérieure de la complexité de l'algorithme.

Théorème 6 (Complexité de l'algorithme d'ordonnancement).
Pour un graphe de dépendance G, contenant $|V|$ nœuds et $|E|$ arcs valués par des vecteurs de dimension n, l'algorithme d'ordonnancement a une complexité en $O((n+1)Z)$ où Z est la complexité de la résolution d'un programme linéaire ayant $(|E|+|V|+n)$ variables et $(3\times|E|)$ inéquations.

DÉMONSTRATION. Soit Z la complexité en temps de la résolution du programme 3.8 de construction de G'. Ce programme met en œuvre $|E|+|V|+n$ variables et $3|E|$ inégalités.

Pour tout nœud v, les vecteurs $X_v^{(1)}, \ldots, X_v^{(d_v-1)}$ sont linéairement indépendants d'après le théorème 5. Nous en déduisons que n majore d_v-1 $(d_v-1\leq n)$ et même d_v $(d_v\leq n)$ quand G est calculable. Par conséquent, la profondeur de récursion est au plus égale à $n+1$.

En outre, les différentes résolutions du programme linéaire (3.8) à la profondeur i ont pour arguments des sous-graphes disjoints de G. La complexité totale de ces résolutions à profondeur i est alors du même ordre Z qu'une résolution similaire sur le graphe G dans son intégralité, puisque toutes ces résolutions peuvent être faites simultanément par fusion des différentes instances, disjointes, des programmes linéaires (3.8) (pour de plus amples détails, se référer à l'article de Kosaraju et Sullivan [54]).

Finalement, une fois calculé G', il ne nous reste plus qu'à construire ses composantes fortement connexes, ce qui peut être fait en $O(|E| + |V|)$ et est donc négligeable devant Z. □

La complexité temporelle ici proposée est meilleure que celle présentée par Kosaraju et Sullivan [54]. L'amélioration provient du théorème 5 qui nous procure une bien meilleure borne sur la profondeur de décomposition que celle dont disposait les chercheurs sus-cités et du système 3.8 qui nous permet d'obtenir G' par la résolution d'un seul programme linéaire au lieu d'un programme par arc.

Conclusion

Nous avons présenté dans ce chapitre les principaux résultats connus concernant la calculabilité, la latence et l'ordonnancement des systèmes d'équations récurrentes uniformes. Les travaux sur la calculabilité sont dus principalement à Karp, Miller et Winograd. La latence a été étudiée par Sailesh Rao. Il a également tenté d'ordonnancer les SERUS à l'aide d'ordonnancement polynomiaux, mais sans succès[3]. Ses travaux ont été repris et étendus par Vwani Roychowdhury qui a réussi à construire des ordonnancements polynomiaux optimaux. Le seul résultat vraiment original de ce chapitre est le programme linéaire (3.7) de construction de G'. Ce programme met en évidence, à travers son dual, le caractère glouton de l'ordonnancement.

Pour nous, les systèmes d'équations récurrentes uniformes ne sont pas une fin en soit. Nous ne sommes réellement intéressés que par les nids de boucles. Nous allons nous servir de notre étude des SERUS pour ordonnancer les nids de boucles dont les dépendances sont représentées par des graphes de dépendance réduits polyédriques.

3. La démonstration du théorème 2.7 de sa thèse [70], théorème de construction d'ordonnancement polynomiaux, est fausse. En effet, son inéquation (2.21) sur le délai induit sur les arcs de G' ne peut être déduite des précédentes comme il le prétend. Il lui a juste manqué le lemme 8 pour pouvoir réellement conclure. Son théorème en lui-même est juste puisqu'il est un corollaire du corollaire 4.

Chapitre 4

Graphes de dépendance réduits polyédriques

Bonsoir, chers clients !

L'acteur *in* Le petit canard, Philippe **Meyer**

Nous avons défini (cf. section 1.2) la notion de graphe de dépendance réduit polyédrique (GDRP) et nous avons justifié l'intérêt que nous portons à ce mode de représentation des dépendances (cf. section 2.6). Nous allons étudier ici les propriétés intrinsèques de ces graphes valués. Afin d'en faciliter la manipulation, nous commencerons par présenter à la section 4.1 un algorithme d'uniformisation qui nous ramènera des graphes de dépendance réduits polyédriques aux graphes de dépendance réduits à dépendances uniformes. Cette transformation nous permet d'utiliser les résultats précédemment établis pour les systèmes d'équations récurrentes uniformes. À partir de l'étude des relations entre un graphe polyédrique et sa forme uniformisée, nous proposerons à la section 4.2 un algorithme pour déterminer la calculabilité d'un GDRP et nous évaluerons la taille des plus longs chemins de dépendance de ces graphes. Nous présenterons un algorithme optimal d'ordonnancement des graphes de dépendance réduits polyédriques (section 4.3) que nous étendrons à la génération de boucles totalement permutables et à l'ordonnancement des nids non parfaits (section 4.4). Finalement, nous comparerons à la section 4.5 notre algorithme et ceux présentés au chapitre 2.

4.1 Uniformisation

Nous montrons ici comment un graphe de dépendance réduit polyédrique peut être représenté par un graphe de dépendance uniforme, c'est-à-dire un graphe de dépendance dont les arcs sont valués par des vecteurs constants. L'uniformisation réalisée est une simple généralisation de ce que nous avions remarqué sur l'exemple 14 à la section 2.6. Il s'agit ici de rendre les dépendances uniformes, non pas en transformant le programme originel comme cela se fait dans le cadre des réseaux systoliques, mais en transformant simplement le graphe de dépendance.

4.1.1 Algorithme

Afin d'éviter de possibles confusions, nous appellerons *graphe originel* et noterons $G_o = (V, E)$ le graphe de dépendance réduit polyédrique qui décrit les dépendances du code que nous voulons paralléliser. Le graphe produit par l'algorithme d'uniformisation à partir du graphe originel sera quant à lui noté $G_u = (W, F)$ et baptisé *graphe uniformisé*.

Pour générer le graphe uniformisé G_u, l'algorithme passe en revue les arcs du graphe originel G_o. G_u ne contient initialement que les nœuds de G_o et aucun arc. G_u vaut donc primitivement : $G_u = (W, F) = (V, \emptyset)$. Pour chaque arc e de G_o, de nouveaux arcs, et parfois un nouveau nœud, sont ajoutés à G_u suivant la valeur de $\mathcal{P}(e)$, le polyèdre de dépendance associé à l'arc e. Nous appelons *nœuds virtuels* les nœuds créés lors de l'uniformisation, par opposition aux nœuds *réels* qui correspondent aux nœuds de G_o.

Pour décrire les polyèdres de dépendance nous utilisons les notations de la section 1.2 : ω, ρ et λ désignent respectivement le nombre de sommets v_i, de rayons r_i et de lignes l_i du polyèdre $\mathcal{P}(e)$.

UNIFORMISATION(GDRP G)

- Initialiser W à V et F à $\emptyset$.
- Pour tout arc $e : q(e) \to t(e) \in E$ faire :
  1. Si $\rho = 0$ et $\lambda = 0$ (le polyèdre est réduit à un polytope)
     - Ajouter à F ω arcs orientés de $q(e)$ vers $t(e)$ et de poids respectifs $v_1, v_2, \ldots, v_\omega$.
  2. Si $\rho \neq 0$ ou $\lambda \neq 0$
     - Ajouter à W un nœud virtuel n_e.
     - Ajouter à F ω arcs orientés de $q(e)$ vers n_e et de poids respectifs $v_1, v_2, \ldots, v_\omega$.
     - Ajouter à F ρ auto-dépendances de n_e de poids respectifs $r_1, r_2, \ldots, r_\rho$.
     - Ajouter à F λ auto-dépendances de n_e de poids respectifs $l_1, l_2, \ldots, l_\lambda$.
     - Ajouter à F λ auto-dépendances de n_e de poids respectifs $-l_1, -l_2, \ldots, -l_\lambda$.
     - Ajouter à F un arc de poids nul orienté de n_e vers $t(e)$.

La figure 4.1 illustre l'uniformisation sur un polyèdre dépourvu de lignes.

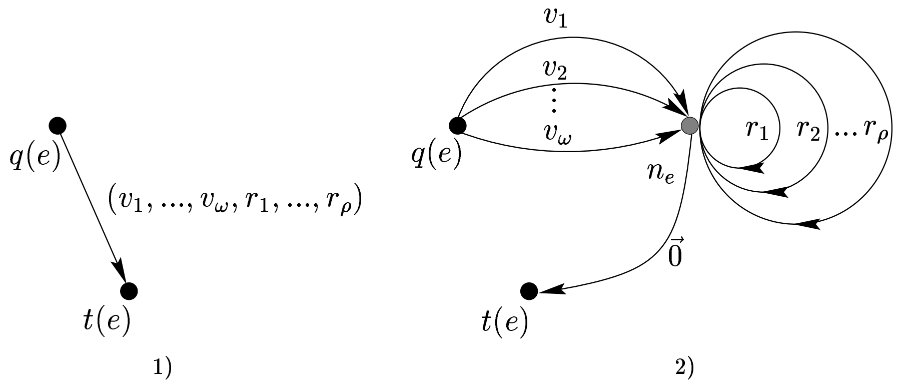


FIG. 4.1: *Exemple d'uniformisation : 1) graphe originel, 2) graphe uniformisé.*

Exemple d'utilisation

Nous illustrons l'algorithme UNIFORMISATION sur l'exemple 16 présenté à la figure 4.2 avec son graphe de dépendance réduit polyédrique. Les dépendances ont été obtenues grâce à l'analyseur Tiny [80] qui les approche par des polyèdres de dépendance particuliers : des vecteurs de direction dont les composantes sont exclusivement prises dans l'ensemble $\mathbb{Z} \cup \{+, 0+, -, 0-, *\}$ (cf. section 1.2).

Le graphe uniformisé associé au GDRP de l'exemple 16 est dessiné à la figure 4.3. Les nœuds virtuels y sont représentés en grisé et les nœuds réels en noir. Ce graphe contient trois nœuds virtuels qui correspondent au symbole « + » et aux deux symboles « − » des vecteurs de direction initiaux.

Exemple 16

```
For i = 1 to n do
  For j = 1 to n do
    For k = 1 to j do
      a(i, j, k) = c(i, j, k - 1) + 1
      b(i, j, k) = a(i - 1, j + i, k) + b(i, j - 1, k)
      c(i, j, k + 1) = c(i, j, k) + b(i, j - 1, k + i)
                   + a(i, j - k, k + 1)
```

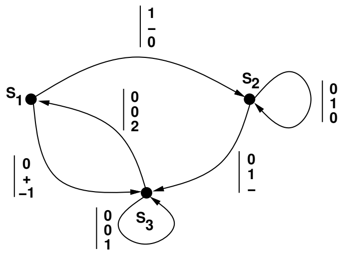


FIG. 4.2: *Exemple 16 et son graphe de dépendance réduit avec vecteurs de direction.*

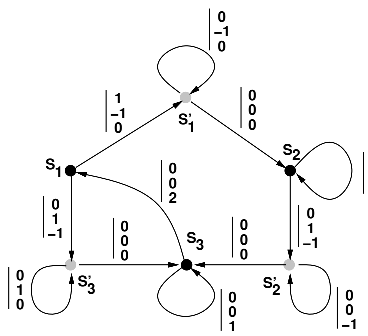


FIG. 4.3: *Graphe uniformisé pour l'exemple 16.*

4.1.2 Relations entre le GDRP G_o et le graphe uniformisé G_u

Dans ce paragraphe nous explorons et relions les structures respectives des graphes G_o et G_u. Cette étude montre l'équivalence des deux graphes. Au lieu de manipuler G_o nous pourrons donc nous contenter de travailler sur G_u qui est somme toute de facture plus classique !

Nous supposons que le graphe originel G_o est fortement connexe (il en va alors de même de G_u). Si tel n'était pas le cas, les résultats présentés ci-dessous ne seraient valables que sur les composantes fortement connexes de G_o.

Souvenons-nous que les arcs de G_o ne sont pas valués par un unique vecteur, comme c'est le cas pour les arcs G_u, mais par un polyèdre, c'est-à-dire un ensemble de vecteurs. Afin d'éviter que l'hypothétique lecteur ne confonde un arc étiqueté par un polyèdre et une des instanciations de cet arc, c'est-à-dire un arc de mêmes extrémités dont le poids est un point du polyèdre, nous appellerons *arc polyédrique* l'arc e étiqueté par le polyèdre $\mathcal{P}(e)$ et *arc de dépendance* une de ses instanciations, f, à qui est donc associé un poids $\boldsymbol{w}(f)$ qui est un point entier du polyèdre : $\boldsymbol{w}(f) \in \mathcal{P}(e)$. Autrement dit, à un arc polyédrique e correspond autant d'arcs de dépendance qu'il y a de points entiers dans le polyèdre $\mathcal{P}(e)$. Un *chemin de dépendance* est naturellement défini comme une suite d'arcs de dépendance dont les arcs polyédriques associés forment un chemin de G_o. Le poids d'un tel chemin est tout simplement la somme des poids des arcs de dépendance qui le composent. Nous sommes maintenant à même de relier les chemins de dépendance de G_u aux chemins polyédriques de G_o.

De G_u à G_o

Remarquons au préalable une particularité de G_u : aucun de ses arcs ne relie deux nœuds virtuels distincts. Par conséquent, un chemin qui raccorde deux nœuds réels et dont tous les nœuds intermédiaires sont virtuels visite au plus un nœud virtuel, éventuellement plusieurs fois. Nous qualifierons d'*atomique* un tel chemin. Remarquez qu'il peut très bien ne contenir aucun nœud virtuel ... De même, un chemin qui relie deux nœuds virtuels distincts passe nécessairement par un nœud réel. La relation entre chemins atomiques de G_u et arcs de dépendance de G_o est immédiate :

Lemme 14. *Un chemin atomique Π_u de G_u correspond à un unique arc polyédrique e de G_o et son poids est un point entier du polyèdre associé à e : $\boldsymbol{w}(\Pi_u) \in \mathcal{P}(e)$.*

Démonstration. Comme nous l'avons déjà observé, le chemin Π_u visite au plus un seul nœud virtuel qu'il peut traverser plusieurs fois. Deux cas de figure se présentent suivant que Π_u contient ou non un nœud virtuel :

- Π_u *traverse un nœud virtuel.* Ce nœud virtuel n_e a été crée lors de l'uniformisation d'un arc polyédrique e. Le schéma ci-dessous illustre la structure de Π_u :

$$\Pi_u = q(e) \xrightarrow{v_i} n_e \dots n_e \xrightarrow{r_i} n_e \dots n_e \xrightarrow{l_j} n_e \dots n_e \xrightarrow{-l_k} n_e \dots n_e \xrightarrow{0} t(e)$$

  Le premier arc de Π_u part de $q(e)$ pour arriver en n_e et il correspond à un sommet du polyèdre $\mathcal{P}(e)$. Le dernier arc de Π_u quitte n_e pour aboutir en $t(e)$ et est de poids nul. Tous les autres arcs constitutifs de Π_u sont des auto-dépendances de n_e dont le poids est soit un rayon de $\mathcal{P}(e)$, soit une ligne (ou son opposé). Le poids de Π_u est donc la somme d'un sommet, d'une combinaison entière positive de rayons et d'une combinaison entière quelconque de lignes du polyèdre $\mathcal{P}(e)$. Par définition des sommets, rayons et lignes d'un polyèdre (cf. section 1.2), le poids de Π_u est bien un point de $\mathcal{P}(e)$.

- Π_u *ne traverse aucun nœud virtuel.* Π_u étant atomique, tous ses nœuds intermédiaires sont virtuels et il ne contient donc qu'un seul arc qui relie deux nœuds réels. Cet arc correspond donc au sommet d'un polyèdre sans rayons ni lignes (cas 1 de l'algorithme Uniformisation) et son poids est donc un point de $\mathcal{P}(e)$.

□

Le lemme 14, se généralise aisément à tout chemin :

Corollaire 5. *Un chemin Π_u de G_u qui relie deux nœuds réels définit un chemin de dépendance Π_o de G_o équivalent : chaque sous-chemin atomique de Π_u correspond exactement à un arc polyédrique de G_o dont le polyèdre de dépendance contient le poids du sous-chemin atomique.*

De G_o à G_u

La réciproque du lemme 14 n'est pas toujours vérifiée ni, *a fortiori*, celle du corollaire 5. Il n'est en effet pas toujours possible de construire un chemin atomique de G_u correspondant à un arc de dépendance donné de G_o, un point entier d'un polyèdre pouvant être obtenu par combinaison linéaire à coefficients rationnels mais non entiers des sommets, rayons et lignes du polyèdre. Néanmoins, lorsque le chemin de dépendance est en fait un circuit, il est possible d'établir une correspondance entre les deux graphes. C'est l'objet du lemme 15. Par la suite, si $\mathcal{C}$ est un circuit et m un entier, $m\mathcal{C}$ représente le circuit formé de m parcours de $\mathcal{C}$.

Lemme 15. *Soit $\mathcal{C}$ un circuit de dépendance de G_o, c'est-à-dire un circuit formé d'arcs de dépendance. Il existe un entier m et un circuit $\mathcal{C}_u$ de G_u tels que les circuits $m\mathcal{C}$ de G_o et $\mathcal{C}_u$ de G_u soient équivalents, c'est-à-dire aient même poids et même structure.*

DÉMONSTRATION. Soit e un arc de dépendance du circuit $\mathcal{C}$. Par définition, $\boldsymbol{w}(e)$ est un point du polyèdre $\mathcal{P}(e)$ et satisfait donc à l'équation :

$$\boldsymbol{w}(e) = \sum_{i=1}^{\omega(e)} \mu(e,i)v(e,i) + \sum_{i=1}^{\rho(e)} \nu(e,i)r(e,i) + \sum_{i=1}^{\lambda(e)} \xi(e,i)l(e,i)$$

où $\mu(e,i) \in \mathbb{Q}^+$, $\nu(e,i) \in \mathbb{Q}^+$, $\xi(e,i) \in \mathbb{Q}$, $\sum_{i=1}^{\omega(e)} \mu(e,i) = 1$ et où $v(e,i)$, $r(e,i)$ et $l(e,i)$ dénotent respectivement les $\omega(e)$ sommets, $\rho(e)$ rayons et $\lambda(e)$ lignes de $\mathcal{P}(e)$.

Soit Δ le plus petit dénominateur commun de tous les rationnels $\mu(e,i)$, $\nu(e,i)$ et $\xi(e,i)$ qui apparaissent dans la décomposition des poids de tous les arcs de dépendance de $\mathcal{C}$. Par construction, nous avons pour tout arc e de $\mathcal{C}$:

$$\Delta\boldsymbol{w}(e) = \sum_{i=1}^{\Delta} v(e,k_i) + \sum_{i=1}^{\rho(e)} \nu'(e,i)r(e,i) + \sum_{i=1}^{\lambda(e)} \xi'(e,i)l(e,i)$$

où $1 \leq k_i \leq \omega(e)$, $\nu'(e,i) \in \mathbb{N}$, et $\xi'(e,i) \in \mathbb{Z}$. Nous pouvons maintenant construire le circuit $\mathcal{C}_u$ désiré qui sera équivalent à $\Delta\mathcal{C}$. Soient $e_1, \ldots, e_n$ les arcs constitutifs de $\mathcal{C}$, dans l'ordre de parcours. Pour chaque entier i, $1 \leq i \leq \Delta$, nous définissons un circuit $\mathcal{C}_u^i$ comme suit :

$$\mathcal{C}_u^i = q(e_1) \xrightarrow{v(e_1,k_i)} n_{e_1} \xrightarrow{0} t(e_1) = q(e_2) \xrightarrow{v(e_2,k_i)} n_{e_2} \xrightarrow{0} t(e_2) \ldots t(e_{n-1}) = q(e_n) \xrightarrow{v(e_n,k_i)} t(e_n) = q(e_1)$$

En fait, si l'arc e_j a été introduit par uniformisation d'un polyèdre sans rayons ni lignes (cas 1 de l'algorithme UNIFORMISATION), le nœud virtuel n_{e_j} n'existe pas et l'élément de chemin : $q(e_j) \xrightarrow{v(e_j,k_i)} n_{e_j} \xrightarrow{0} t(e_j)$ devient $q(e_j) \xrightarrow{v(e_j,k_i)} t(e_j)$.

Finalement, nous concaténons tous les circuits C_u^i et nous ajoutons au circuit ainsi fabriqué, pour chaque arc e de $\mathcal{C}$, $\nu'(e,i)$ fois l'auto-dépendance de n_e de poids $r(e,i)$ et $\xi'(e,i)$ fois (resp. $-\xi'(e,i)$ fois) l'auto-dépendance de n_e de poids $l(e,i)$ (resp. $-l(e,i)$) si $\xi'(e,i)$ est positif (resp. négatif). Le circuit ainsi produit est équivalent à $\Delta\mathcal{C}$ tant par son poids que par les arcs qu'il emploie. □

Conclusion

Le corollaire 5 et le lemme 15 nous permettent de passer des circuits de G_u à ceux de G_o et réciproquement, à un facteur multiplicatif près. Nous allons maintenant nous pencher sur G_u

et tenter d'appliquer les résultats établis pour les systèmes d'équations récurrentes uniformes aux graphes de dépendances générés par uniformisation de graphes de dépendance réduits polyédriques. Les lemmes que nous venons de démontrer nous serviront à transposer les résultats des graphes uniformisés aux graphes originels.

4.2 Calculabilité

Nous nous intéressons au problème de calculabilité des graphes de dépendance réduits polyédriques comme nous l'avons fait au chapitre précédent pour les systèmes d'équations récurrentes uniformes. Contrairement au cas des équations récurrentes, les poids qui valuent les arcs des graphes de dépendance ne sont pas, ici, quelconques. Ils proviennent en effet d'un nid de boucles et sont des approximations de vecteurs de distance, or les vecteurs de distance sont toujours lexicographiquement positifs, comme nous l'avons déjà précisé. Nous pourrions donc nous attendre à ce que l'ordre originel des opérations dans le nid de boucles respecte les dépendances représentées par le GDRP et donc à ce que nos GDRP soient toujours calculables. Tel est effectivement le cas quand le GDRP représente exactement et uniquement les dépendances présentes dans le nid originel. Mais, la plupart du temps, il y a eu approximation des dépendances, approximation sur laquelle nous n'avons fait d'autres hypothèses que son caractère conservatif. Nous ne pouvons donc postuler la calculabilité *a priori* de nos graphes de dépendance. En pratique, les programmes d'analyse de dépendance à notre disposition étant suffisamment intelligents, tous les graphes que nous serons amenés à manipuler représenteront des systèmes calculables : les polyèdres de dépendance seront inclus dans les polyèdres engendrés par les niveaux de dépendance ou les vecteurs de distance seront simplement lexicographiquement positifs, et l'ordre séquentiel original définira un ordonnancement valide. La calculabilité n'en reste pas moins un problème important puisque son étude nous mènera naturellement à un algorithme d'ordonnancement.

Nous établissons la condition de calculabilité des GDRP au paragraphe 4.2.1 et nous présentons aux paragraphes 4.2.2 et 4.2.3 deux algorithmes qui la testent. Au paragraphe 4.2.4 nous étudions les plus longs chemins de dépendance de nos graphes.

4.2.1 Condition de calculabilité

Un graphe de dépendance réduit est dit « calculable » si et seulement si il décrit un calcul bien défini. Dans le cas de systèmes à domaine borné, un graphe de dépendance réduit est calculable si et seulement si son graphe de dépendance apparent ne contient pas de circuits, c'est-à-dire si et seulement si une instance d'une variable ne dépend jamais d'elle-même. Comme pour les équations récurrentes (chapitre 3), cette propriété est équivalente, dès lors que le domaine est suffisamment grand, à l'absence dans le graphe de dépendance réduit de circuits de poids nul. Cette caractérisation qui s'applique directement aux GDRP, et donc au graphe originel G_o, se transpose aux graphes uniformisés grâce au corollaire 5 et au lemme 15 :

Théorème 7 (Calculabilité des graphes de dépendance réduits polyédriques).
G_o est calculable si et seulement si G_u ne contient pas de circuits de poids nul incluant des nœuds réels.

4.2.2 Algorithme de décomposition (AD)

Au paragraphe précédent, nous avons déplacé le problème de calculabilité du graphe originel G_o au graphe uniformisé G_u. La recherche de circuits de poids nul dans G_u est effectivement beaucoup plus simple qu'une inspection directe de G_o, les graphes uniformisés ressemblant comme deux gouttes d'eau aux graphes de dépendance des systèmes d'équations récurrentes uniformes. La seule différence, notable, entre les deux types de graphes est la présence de deux types de nœuds dans les graphes uniformisés de GDRP : nœuds réels et nœuds virtuels. Ici, la calculabilité n'interdit plus tous les circuits de poids nul, mais seulement ceux qui contiennent au moins un nœud réel. L'algorithme de Karp, Miller et Winograd n'a donc besoin que d'être très légèrement modifié pour satisfaire aux nouvelles contraintes et la preuve de sa correction (cf. paragraphe 3.2.1) s'étend immédiatement à l'algorithme de décomposition obtenu.

booléen AD(Graphe uniformisé G)

1. Décomposer G en ses composantes fortement connexes et ne retenir que les composantes G_1, ..., G_i, ..., G_s qui contiennent au moins un nœud réel. Appeler les étapes 2 et 3 sur chacun de ces sous-graphes G_i et renvoyer le ET logique des réponses.

2. Construire G'_i, le sous-graphe de G_i généré par les arcs de G_i qui appartiennent à au moins un multi-circuit de poids nul.

3.
   - Si G'_i est vide ou ne contient que des nœuds virtuels, renvoyer VRAI.
   - Si G'_i est fortement connexe et contient au moins un nœud réel, renvoyer FAUX.
   - Sinon, renvoyer AD(G'_i).

Cet algorithme est bien évidemment appelé sur un graphe uniformisé et il retourne VRAI si et seulement si ce graphe est calculable, c'est-à-dire si et seulement si il ne contient pas de circuits de poids nul incluant un nœud réel. Par conséquent, le GDRP G_o est calculable si et seulement si l'appel AD(G_u) renvoie VRAI.

Nous définissons pour un graphe uniformisé G_u et pour ses nœuds une profondeur de décomposition comme nous l'avions fait pour les systèmes d'équations récurrentes uniformes (cf. paragraphe 3.2.1). La profondeur de décomposition d'un nœud d'un GDRP G_o est égale à la profondeur du nœud de G_u qui lui est associé. La profondeur du graphe originel G_o est la profondeur maximale de ses nœuds. Comme pour les équations récurrentes, nous relierons les notions de profondeur de décomposition, de longueur des chemins de dépendance et de degré de parallélisme.

4.2.3 Variante de l'algorithme de décomposition (AD*)

Pour des raisons techniques qui apparaîtront au moment d'ordonnancer les GDRP (section 4.3), nous proposons une deuxième version de l'algorithme de décomposition. Les deux algorithmes sont équivalents en ce qu'ils produisent la même réponse.

Lors de la décomposition, nous voulons garder ensemble les arcs du graphe uniformisé qui proviennent de la traduction du même arc polyédrique, et ce tant qu'au moins un des arcs correspondant à un **sommet** du polyèdre participe à un multi-circuit de poids nul. Pour ce faire, pour chaque arc polyédrique e, si G' contient une arc correspondant à un sommet du polyèdre $\mathcal{P}(e)$, nous rajoutons à G' tous les arcs issus de la transcription de $\mathcal{P}(e)$ qu'il ne contient pas déjà. Nous obtenons l'algorithme suivant :

booléen AD*(Graphe uniformiséG)

1. Décomposer G en ses composantes fortement connexes et ne retenir que les composantes G_1, ..., G_i, ..., G_s qui contiennent au moins un nœud réel. Appeler les étapes 2, 3 et 4 sur chacun de ces sous-graphes G_i.

2. Construire G_i' le sous-graphe de G_i généré par les arcs de G_i qui appartiennent à au moins un multi-circuit de poids nul.
3. Initialiser G_i'' à G_i'. Pour tout arc polyédrique e, si G_i' contient un arc correspondant à un sommet du polyèdre $\mathcal{P}(e)$, rajouter à G_i'' tous les arcs issus de la transcription de $\mathcal{P}(e)$ qu'il ne contient pas déjà. Nous
4.
   - Si G_i'' est vide ou ne contient que des nœuds virtuels, renvoyer VRAI.
   - Si G_i'' est fortement connexe et contient au moins un nœud réel, renvoyer FAUX.
   - Sinon, renvoyer $\mathrm{AD}^*(G_i'')$.

Nous ne rajoutons à une composante fortement connexe G_i' que des arcs reliant des nœuds qu'elle contient [1]. Nous ne risquons donc pas de connecter des composantes fortement connexes disjointes et de modifier le comportement de l'algorithme de décomposition. De plus, les arcs rajoutés ne participent à aucun multi-circuit de poids nul à la profondeur où ils sont rajoutés. Il n'y a donc aucun risque qu'ils appartiennent à plus grande profondeur à de tels multi-circuits. Par conséquent, les algorithmes AD et AD* se comportent de la même manière vis-à-vis des circuits recherchés et des nœuds. En particulier, un nœud a même profondeur de décomposition pour les deux algorithmes.

4.2.4 Plus longs chemins de dépendance des GDRP

Nous allons évaluer la taille des chemins du graphe de dépendance apparent et nous obtiendrons comme précédemment une borne inférieure sur la séquentialité, et donc une borne supérieure sur le parallélisme, contenus dans la représentation des dépendances. Nous appliquons au graphe uniformisé G_u le théorème sur les plus longs chemins d'un système d'équations récurrentes uniformes et nous en transposons le résultat à G_o au moyen des lemmes du paragraphe 4.1.2. Ce faisant, nous obtenons un théorème similaire à celui établi pour les équations récurrentes :

Théorème 8 (Plus longs chemins d'un GDRP).
Soient G_o un GDRP. Si le domaine d'itération de G_o contient un hypercube de dimension n et de taille $\Omega(N)$ et si d_S est la profondeur de décomposition du nœud S de G_o par l'algorithme AD *(resp.* AD**) appliqué à G_u, alors il existe pour chaque composante fortement connexe G_i de G_o un chemin dans le graphe de dépendance apparent de G_o dont la projection sur G_o est un circuit qui contient $\Omega(N^{d_S})$ instances de chaque nœud S de G_i.*

DÉMONSTRATION. Remarquons tout d'abord que la profondeur de décomposition d'un nœud réel de G_u est la même pour les algorithmes KMW et AD, le premier ne distinguant pas les nœuds réels des virtuels. Nous utilisons alors le théorème 4 pour construire, pour chaque composante fortement connexe G_i de G_u contenant au moins un nœud réel, un chemin du graphe de dépendance apparent dont la projection sur G_u est un circuit qui traverse $\Omega(N^{d_S})$ fois chaque nœud S de G_i. La projection sur G_u étant un circuit contenant au moins un nœud réel, nous concluons en utilisant le corollaire 5 qui nous permet d'obtenir un circuit de G_o et qui nous assure qu'un nœud réel a le même nombre d'occurrences dans les deux circuits. □

1. L'arc de poids nul, reliant les nœuds n_e et $t(e)$ d'un arc polyédrique e, appartient à G' si et seulement si un des arcs correspondant à un sommet de $\mathcal{P}(e)$ y appartient. En effet, les seuls arcs entrants de n_e qui ne soient pas des auto-dépendances sont ceux qui correspondent aux sommets du polyèdre et le seul arc sortant de n_e qui ne soit pas une auto-dépendance est l'arc de poids nul vers $t(e)$. Donc, comme G' est fortement connexe par construction, G' contient un arc correspondant à un sommet de $\mathcal{P}(e)$ si et seulement si il contient l'arc de poids nul reliant les nœuds n_e et $t(e)$.

Nous n'avons pas prouvé que les chemins ainsi fabriqués étaient « les plus longs » comme l'intitulé du théorème l'annonçait. Nous le montrerons indirectement une fois que nous aurons proposé et étudié à la section suivante un algorithme d'ordonnancement des graphes de dépendance réduits polyédriques. Sur ce point nous suivons une fois de plus l'étude réalisée pour les systèmes d'équations récurrentes uniformes.

4.3 Ordonnancement

Grâce au procédé d'uniformisation, nous avons pu adapter aux graphes de dépendance réduits polyédriques (GDRP) les résultats de calculabilité obtenus pour les systèmes d'équations récurrentes uniformes. Nous suivons, ici, la même démarche pour pour l'ordonnancement. Au paragraphe 4.3.1 nous proposons un algorithme d'ordonnancement dont nous prouvons la correction et l'optimalité au paragraphe 4.3.2. Au paragraphe 4.3.3, nous étudions quelques propriétés des parties linéaires des algorithmes produits. Nous illustrons l'algorithme au paragraphe 4.3.4.

4.3.1 Algorithme

Nous pourrions considérer le graphe uniformisé G_u comme le graphe de dépendance d'un système d'équations récurrentes uniformes et l'ordonnancer à l'aide de l'algorithme présenté au paragraphe 3.3.2. Ce faisant, nous n'obtiendrions malheureusement pas obligatoirement un ordonnancement valide de G_o. Nous avons vu au paragraphe 4.1.2 que tout chemin de G_o ne pouvait pas forcément être simulé par un chemin de G_u à cause des combinaisons barycentriques de sommets de polyèdres et des combinaisons non entières de lignes et de rayons. Cette même propriété nous interdit la réutilisation bête de l'algorithme ORDONNANCEUR_DE_SERU. Ci-dessous, nous explicitons le problème sur un exemple.

Supposons que G_o contienne un arc e dont le polyèdre de dépendance a au moins deux sommets σ_1 et σ_2. Supposons de plus que les arcs f_1 et f_2 de G_u associés respectivement à ces deux sommets ne soient pas satisfaits à la même profondeur par l'ordonnancement trouvé. Par exemple, f_1 est satisfait par la première dimension de l'ordonnancement et f_2 par la deuxième. G_o contient l'arc de dépendance f de poids $(\sigma_1 + \sigma_2)/2$. La première dimension de l'ordonnancement induit sur f_1 un délai supérieur à un et seulement un délai positif sur f_2. Par conséquent, nous sommes juste assurés que le délai induit sur f est supérieur à un demi. L'arc f n'est donc pas satisfait par la première dimension de l'ordonnancement (pour ce faire le délai doit être supérieur à un, cf. section 1.3). L'arc f_1 étant satisfait par la première dimension de l'ordonnancement, il n'est pas considéré pour le calcul de la deuxième dimension. Nous n'avons donc *aucune* information sur le délai induit par cette deuxième dimension sur f_1, et donc *a fortiori* sur f. En résumé, nous ne savons pas si l'arc de dépendance f est ou non satisfait, ce qui est pour le moins gênant ! Comme G_o contient tous les arcs de poids $(\lambda\sigma_1 + (1 - \lambda)\sigma_2)$ pour $\lambda \in [0, 1]$, nous ne pouvons pas nous en sortir en augmentant arbitrairement le délai induit sur f_1 par la première composante de l'ordonnancement.

Nous rencontrons exactement le même problème dans le cas de combinaisons non entières de rayons et de lignes. Tout polyèdre admet une représentation, appelée base de Hilbert, dans laquelle tout point du polyèdre se décompose en combinaison entière de sommets, rayons et lignes. Typiquement, les vecteurs de direction sont représentés, par définition, sur une base de Hilbert (cf. section 1.2). Si nous sommes assurés que tous les polyèdres que nous serons amenés à manipuler seront représentés dans une base de Hilbert, nous pouvons réutiliser bêtement l'algorithme ORDONNANCEUR_DE_SERU. Nous ne voulons pas faire cette hypothèse et nous présentons donc une solution au problème général d'ordonnancement des GDRP.

Pour que notre problème soit résolu, il suffit qu'une des dimensions de l'ordonnancement induise **simultanément** un délai strictement positif sur tous les sommets d'un polyèdre et que les dimensions précédentes induisent un délai positif sur ces sommets. Comme nous n'aurons pas cette propriété dans le cas général, nous l'imposons. L'algorithme de décomposition AD* nous permet d'assurer cette propriété et l'algorithme d'ordonnancement repose sur cet algorithme de décomposition. L'algorithme d'ordonnancement différera légèrement d'une fusion simple des algorithmes AD* et ORDONNANCEUR_DE_SERU. Nous utiliserons en effet la liberté dont nous disposons sur les rayons : nous n'avons besoin d'y induire qu'un délai positif et non strictement positif (cf. la remarque finale de la section 2.6 et le paragraphe 4.3.2).

L'algorithme construit pour chaque nœud réel S de G_u une séquence de vecteurs $(X_S^1, \ldots, X_S^k, \ldots, X_S^{d_S})$ et une séquence de constantes $(\rho_S^1, \ldots, \rho_S^k, \ldots, \rho_S^{d_S})$. Ces séquences définissent un ordonnancement multidimensionnel valide comme nous le prouverons au prochain paragraphe. L'algorithme est initialement invoqué sur le graphe uniformisé (appel ORDONNANCEUR_DE_GDRP$(G_u, 1)$). Nous supposons que G_u est fortement connexe. Si tel n'était pas le cas, nous utiliserions l'algorithme ci-dessous pour ordonnancer chacune de ses composantes fortement connexes et nous ordonnancerons ces dernières entre elles topologiquement, comme nous l'avons fait au paragraphe 3.3.2 pour les SERU.

booléen ORDONNANCEUR_DE_GDRP(Graphe uniformisé G, k)

1. Décomposer G en ses composantes fortement connexes et ne conserver que les composantes G_1, ..., G_i, ..., G_s qui contiennent au moins un nœud réel. Appeler les étapes 2, 3, 4, puis 5 sur chacun de ces sous-graphes G_i et renvoyer le ET logique des réponses.

2. Construire G_i' le sous-graphe de G_i généré par les arcs qui appartiennent à au moins un multi-circuit de poids nul.

3. Poser $G_i'' = G_i'$. Pour tout arc polyédrique e de G_o, si G_i' contient un arc correspondant à un sommet du polyèdre $\mathcal{P}(e)$, rajouter à G_i'' tous les arcs issus de la transcription de $\mathcal{P}(e)$ qu'il ne contient pas déjà.

4. Choisir un vecteur X et, pour chaque nœud S de G_i, une constante ρ_S tels que :

$$\left\{\begin{array}{lll} e = (q(e), t(e)) \in G_i'' \text{ ou } q(e) \text{ est un nœud virtuel} & \Rightarrow & X\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 0 \\ e = (q(e), t(e)) \notin G_i'' \text{ et } q(e) \text{ est un nœud réel} & \Rightarrow & X\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 1 \end{array}\right. \quad (4.1)$$

   Pour tout nœud réel S de G_i, nous posons $\rho_S^k = \rho_S$ et $X_S^k = X$.

5. – Si G_i'' est vide ou ne contient que des nœuds virtuels, renvoyer VRAI.
   – Si G_i'' est fortement connexe et contient au moins un nœud réel, renvoyer FAUX.
   – Sinon, renvoyer ORDONNANCEUR_DE_GDRP$(G_i''$,$k + 1)$.

Remarques :

– Nous verrons que l'étape 3 n'est nécessaire que pour les GDRP généraux. En particulier, elle peut être ignorée pour les représentations par vecteurs de direction. Lorsque l'étape 3 est superflue, les étapes 2 et 4 peuvent être résolus par un unique programme linéaire.

– Nous ne précisons pas, volontairement, comment choisir le vecteur X et les constantes ρ de l'étape 4 de manière à laisser toute liberté à l'utilisateur. Les techniques décrites au paragraphe 3.3.1 et employant des bases de cycles sont toujours utilisables. Pour ce qui est de la fonction objective, nous pouvons minimiser classiquement la norme du vecteur X, $\sum_{i=1}^{n} |X_i|$, grâce à une nouvelle variable Y, deux nouvelles contraintes, $X \leq Y$ et $-X \leq Y$, et la fonction objective min $\sum_{i=1}^{n} Y_i$. Nous pouvons aussi minimiser la latence induite par X sur le domaine.

Retour sur l'exemple 16. Considérons le graphe de dépendance uniforme présenté à la figure 4.3. Il contient deux circuits élémentaires de poids $(1, 0, 1)$ et $(0, 1, 1)$ et cinq auto-dépendances de poids respectifs $(0, 0, 1)$, $(0, 0, -1)$, $(0, 1, 0)$ (deux fois) et $(0, -1, 0)$. Par conséquent, tous les arcs participent à des multi-circuits de poids nul[2], exceptés ceux qui n'appartiennent qu'au circuit de poids $(1, 0, 1)$. Le sous-graphe G' est exposé à la figure 4.4.

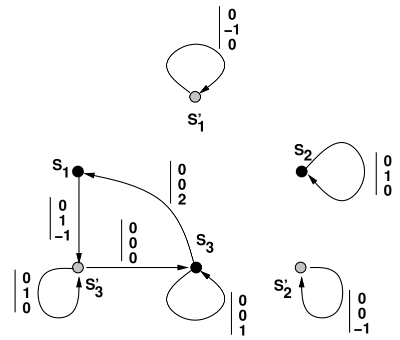


FIG. 4.4: *Sous-graphe des multi-circuits de poids nul de l'exemple 16.*

Les contraintes d'ordonnancement provenant de G' obligent la partie linéaire, $X = (x, y, z)$, à être orthogonale au poids de tous les circuits de G' (nous reviendrons en détail sur ce point à la section 4.3.3). Par conséquent, ces deux dernières coordonnées sont nulles : $y = z = 0$. Considérant les autres contraintes, nous trouvons (par exemple) la solution $X = (2, 0, 0)$, $\rho_{S_1} = \rho_{S_3} = 1$ et $\rho_{S_2} = 0$.

G' contient quatre composantes fortement connexes. Deux d'entre elles ne contiennent que des nœuds virtuels et nous les ignorons donc. Les deux autres ne comportent pas de multi-circuits de poids nul. La composante fortement connexe ne contenant que le nœud S_2 peut être ordonnancée avec le vecteur $X = (0, 1, 0)$, tandis que l'étude de l'autre composante nous donne, comme solution possible, le vecteur $X = (0, 0, 2)$ et les constantes $\rho_{S_1} = 0$ et $\rho_{S_3} = 3$.

En résumé, nous trouvons l'ordonnancement bidimensionnel : $(2i, j)$ pour S_2, $(2i+1, 2k)$ pour S_1 et $(2i+1, 2k+3)$ pour S_3. Utilisant la procédure « codegen » de l'« Omega Calculator » délivré avec Petit [52] nous avons obtenu le code parallèle exhibé à la figure 4.5. Le code proposé est « virtuel » dans le sens où il se contente de révéler le parallélisme contenu dans le programme originel. Nous ne prétendons pas qu'il doive être exécuté tel quel. Nous n'avons, entre autres, fait aucun effort pour tenter d'éliminer les tests (au moyen de « loop peeling »). Nous n'avons pas plus précisé quel processeur exécutait quel calcul, où se trouvait les données, etc.

Le lecteur pourra remarquer que ni ALLEN-KENNEDY ni la version originelle de WOLF-LAM ne sont capables de trouver ici tout le parallélisme, l'instruction S_3 leur paraissant purement séquentielle à tous les deux.

2. Car $(0, 1, 1) + (0, -1, 0) + (0, 0, -1) = (0, 0, 1) + (0, 0, -1) = (0, 1, 0) + (0, -1, 0) = (0, 0, 0)$.

```
ForSeq i = 1 to n do
  ForSeq j = 1 to n do
    ForPar k = 1 to j do
      b(i, j, k) = a(i - 1, j + i, k) + b(i, j - 1, k)
  ForSeq k = 1 to n+1 do
    If (k ≤ n) Then
      ForPar j = k to n do
        a(i, j, k) = c(i, j, k - 1) + 1
    If (k ≥ 2) Then
      ForPar j = k-1 to n do
        c(i, j, k) = c(i, j, k - 1) + b(i, j - 1, k + i - 1) + a(i, j - k + 1, k)
```

FIG. 4.5: *Version parallélisée de l'exemple 16.*

4.3.2 Correction et optimalité de l'algorithme

Nous démontrons d'abord la correction puis l'optimalité de ORDONNANCEUR_DE_GDRP. À la fin de ce paragraphe nous établirons quelques propriétés sur les ordonnancements produits par l'algorithme.

Correction

Nous devons montrer que l'algorithme présenté au paragraphe précédent est correct, c'est-à-dire que toutes ses étapes sont réalisables, qu'il se termine et qu'il produit le résultat escompté. Il est à remarquer que son squelette n'est autre que l'algorithme AD*. La correction de ce dernier ayant été prouvée au paragraphe 4.2.3, il nous suffit de démontrer que :

- l'étape 4 est possible, c'est-à-dire qu'il existe toujours un vecteur X et des constantes ρ qui répondent à nos conditions;
- les séquences de vecteurs et de constantes définissent véritablement un ordonnancement.

Le premier point est clair. En effet, le lemme 8 nous assure de l'existence d'un vecteur X et de constantes ρ_S qui vérifient le système :

$$\begin{cases} X\boldsymbol{w}(e) + \rho_{y_e} - \rho_{x_e} = 0 & \forall e \in G' \\ X\boldsymbol{w}(e) + \rho_{y_e} - \rho_{x_e} \geq 1 & \forall e \notin G' \end{cases}$$

Entre autres, toute solution optimale du programme linéaire dual 3.8 nous fournit de tels vecteurs et constantes. Les contraintes du système linéaire de l'étape 4 sont par conséquent toujours satisfiables puisqu'elles sont plus faibles : certaines des contraintes « est supérieur à un » ont été relâchées pour être remplacées par « est positif ».

Pour faciliter la preuve du second point, nous complétons chaque séquence de vecteurs X_S^i avec des vecteurs nuls et chaque séquence de constantes ρ_S^i avec des zéros, pour que toutes soient de longueur d, la profondeur de décomposition de G_u par AD*.

Lemme 16. *Si $G_o = (V, E)$ est calculable, la fonction affine multidimensionnelle Θ produite par* ORDONNANCEUR_DE_GDRP *:*

$$\begin{array}{rcl} V \times \mathcal{D} & \to & \mathbb{Z}^d \\ (S,\ \boldsymbol{i}) & \mapsto & (\lfloor X_S^1 \boldsymbol{i} + \rho_S^1 \rfloor,\ \ldots,\ \lfloor X_S^{d_S} \boldsymbol{i} + \rho_S^{d_S} \rfloor,\ 0,\ \ldots,\ 0) \end{array}$$

définit un ordonnancement multidimensionnel de G_o.

DÉMONSTRATION. Pour prouver que la fonction Θ définit bien un ordonnancement, nous devons montrer qu'elle vérifie toutes les dépendances représentées par le GDRP G_o, c'est-à-dire que pour tout arc polyédrique $e = (q(e), t(e))$ de G_o, pour tout point $\boldsymbol{w}(e)$ du polyèdre $\mathcal{P}(e)$ et tout point $\boldsymbol{i}$ du domaine $\mathcal{D}$, nous avons : $\Theta(t(e), \boldsymbol{i}) \succ \Theta(q(e), \boldsymbol{i} - \boldsymbol{w}(e))$, où $\succ$ représente l'ordre lexicographique strict (cf. section 1.3). Pour ce faire nous allons établir l'existence d'un entier k, $k \in [1 \dots d]$, tel que :

$$\begin{cases} X^i_{t(e)}\boldsymbol{i} + \rho^i_{t(e)} \geq X^i_{q(e)}(\boldsymbol{i} - \boldsymbol{w}(e)) + \rho^i_{q(e)} & \text{pour tout } i,\ 1 \leq i \leq k-1 \\ X^k_{t(e)}\boldsymbol{i} + \rho^k_{t(e)} \geq X^k_{q(e)}(\boldsymbol{i} - \boldsymbol{w}(e)) + \rho^k_{q(e)} + 1 & \end{cases} \tag{4.2}$$

ce qui démontrera la propriété recherchée.

Les valeurs possibles de $\boldsymbol{w}(e)$ nous sont données par l'équation suivante :

$$\boldsymbol{w}(e) = \sum_{j=1}^{\omega} \mu_j v_j + \sum_{j=1}^{\rho} \nu_j r_j + \sum_{j=1}^{\lambda} \xi_j l_j \text{ avec } (\mu, \nu, \xi) \in \mathbb{Q}^{+\omega} \times \mathbb{Q}^{+\rho} \times \mathbb{Q}^{\lambda} \text{ et } \sum_{j=1}^{\omega} \mu_j = 1 \tag{4.3}$$

Nous supposons dans le reste de la preuve que l'arc considéré a engendré un nœud virtuel lors de l'uniformisation. La démonstration est similaire pour l'autre cas.

Si G_o est calculable, tous les appels récursifs de AD* se terminent et débouchent sur un sous-graphe G' soit vide, soit qui ne contient que des nœuds virtuels. Tout arc du graphe correspondant au sommet d'un polyèdre est donc éliminé au cours de la décomposition. L'étape 3 de l'algorithme garde ensemble dans le graphe tous les arcs issus de l'uniformisation d'un même arc polyédrique jusqu'à la profondeur minimale k où plus aucun arc correspondant à un sommet de $\mathcal{P}(e)$ n'appartient à un multi-circuit de poids nul. Les arcs issus de l'uniformisation de $\mathcal{P}(e)$ sont donc toujours dans la même composante fortement connexe et les séquences de vecteurs $X^i_{t(e)}$, $X^i_{q(e)}$ et $X^i_{n_e}$ sont confondues jusqu'au niveau k inclus. Nous oublions donc l'indice et écrivons simplement X^i. Le système 4.2 se réécrit alors :

$$\begin{cases} X^i\boldsymbol{w}(e) + \rho^i_{t(e)} - \rho^i_{q(e)} \geq 0 \quad \forall i,\ 1 \leq i \leq k-1 \\ X^k\boldsymbol{w}(e) + \rho^k_{t(e)} - \rho^k_{q(e)} \geq 1 \end{cases} \tag{4.4}$$

L'étape 4 de l'algorithme nous assure de la validité, pour $i \in \{1, \dots, k\}$, des inégalités :

$$\begin{cases} X^i l_j = X^i l_j + \rho^i_{n_e} - \rho^i_{n_e} \geq 0 \text{ et } X^i(-l_j) \geq 0, \quad \text{c'est-à-dire } X^i l_j = 0 \\ X^i r_j = X^i r_j + \rho^i_{n_e} - \rho^i_{n_e} \geq 0 \\ X^i v_j + \rho^i_{n_e} - \rho^i_{q(e)} \geq 0 \text{ si } i \leq k-1 \text{ et } X^k v_j + \rho^k_{n_e} - \rho^k_{q(e)} \geq 1 \\ X^i\, 0 + \rho_{t(e)} - \rho_{n_e} \geq 0 \end{cases}$$

Par sommation *ad-hoc* de ces inégalités et en utilisant la définition de $\boldsymbol{w}(e)$, nous obtenons la satisfaction du système 4.4. □

Remarque. La modification apportée à AD* pour garder ensemble tous les arcs issus d'un même polyèdre nous assure que le système 4.2 est satisfait même pour des poids $\boldsymbol{w}(e)$ qui seraient combinaisons rationnelles de sommets, lignes et rayons. Si, pour tout point entier $\boldsymbol{w}(e)$ de $\mathcal{P}(e)$, la décomposition de l'équation 4.3 est à coefficients entiers (et non simplement rationnels), $\mathcal{P}(e)$ est dit décomposé sur une **base de Hilbert** [74]. Dans ce cas, l'algorithme peut être simplifié : les arcs f de G_u correspondant à $\mathcal{P}(e)$ et vérifiant de fait l'inégalité $X\boldsymbol{w}(f) + \rho_{y_f} - \rho_{x_f} \geq 1$ n'ont pas besoin d'être gardés dans G'. Supprimer ainsi des arcs de G' permet de relâcher les contraintes sur les composantes suivantes de l'ordonnancement.

Les vecteurs de direction expriment une décomposition sur base de Hilbert comme le montre la simulation des vecteurs de direction par des polyèdres à la section 1.2. Dans le cas d'un graphe de dépendance réduit uniquement valué par des vecteurs de direction nous pouvons donc tout bonnement ignorer l'étape 3 de ORDONNANCEUR_DE_GDRP.

Optimalité

Pour prouver l'optimalité de notre algorithme d'ordonnancement, nous nous contentons de recopier ce que nous avons écrit pour l'ordonnancement des systèmes d'équations récurrentes uniformes.

Lemme 17. *Si le domaine d'itération est contenu dans un hypercube de dimension n et de taille $O(N)$, et si d_S est la profondeur de décomposition du nœud S de G, alors l'ordonnancement multidimensionnel associé à S à une latence en $O(N^{d_S})$.*

La démonstration de ce lemme est similaire à celle du lemme 13.

Le théorème 8 et le lemme 17 montrent que les ordonnancements multidimensionnels que nous obtenons sont optimaux :

Corollaire 6 (Optimalité de l'ordonnancement).
L'ordonnancement proposé est optimal : si le domaine d'itération contient (resp. est contenu dans) un hypercube de dimension n et de taille $\Omega(N)$ (resp. $O(N)$) et si d est la profondeur de décomposition de G, alors la latence de l'ordonnancement est en $O(N^d)$ et la longueur du plus long chemin en $\Omega(N^d)$.

Plus précisément, si d_S est la profondeur de décomposition du nœud S de G, alors l'ordonnancement associé à S à une latence en $O(N^{d_S})$ et il existe dans le graphe de dépendance apparent un chemin qui contient $\Omega(N^{d_S})$ occurrences de S.

Plus simplement, les chemins de dépendance exhibés par le théorème 8 définissaient une borne inférieure sur la séquentialité du graphe de dépendance réduit polyédrique. Nos ordonnancements atteignent cette borne, en ordre de grandeur. Les GDRP contiennent donc exactement $(n-d)$ degrés de parallélisme et nous sommes capables de les expliciter. Comme précédemment, les causes de sous-optimalité de l'algorithme dans le cas général ne sont pas à rechercher dans les techniques employées mais dans la représentation des dépendances et le manque d'information sur les domaines d'itérations.

Considérons l'exemple 17 (figure 4.6). Si les dépendances sont représentées par des vecteurs de distance, le GDR contient deux auto-dépendances de poids $(0,1)$ et deux arcs étiquetés par un polyèdre ayant un sommet ($(1,2)$ et $(1,-2)$ respectivement) et un rayon ($(0,1)$ et $(0,-1)$ respectivement). Le GDR comprend par conséquent un multi-circuit de poids nul. De plus, ce multi-circuit visite les deux nœuds réels. Par conséquent, l'algorithme ORDONNANCEUR_DE_GDRP ne trouve aucun parallélisme dans cet exemple.

Exemple 17

For i=1 to n do
  For j=1 to n do
    S_1 : a(i, j) = b(i-1, j+i) + a(i, j-1)
    S_2 : b(i, j) = a(i-1, j-i) + b(i, j-1)

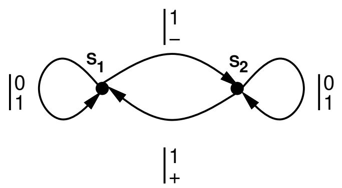


FIG. 4.6: *Exemple 17 et son graphe de dépendance réduit.*

Il est cependant tout à fait valide de calculer l'itération (i, j) de S_1 (resp. S_2) à la date $2i + j$ (resp. $i + j$). L'ordonnancement proposé met en évidence un degré de parallélisme que notre algorithme ne pouvait pas trouver : l'approximation des dépendances a rendu le nid séquentiel en apparence. L'algorithme de Feautrier trouve un tel ordonnancement, si les dépendances exactes lui sont fournies.

La preuve d'optimalité proposée pour notre algorithme est moins forte que celle établie pour ALLEN-KENNEDY. En effet, nous n'avons pas prouvé que pour chaque GDRP G il existait un nid admettant G pour graphe de dépendance et dont le graphe de dépendance étendu contiendrait, pour toute instruction S, un chemin de dépendance visitant $\Omega(N^{d_S})$ instances de S.

4.3.3 Propriétés des hyperplans séparateurs

Nous étudions ici les parties linéaires des ordonnancements produits par ORDONNANCEUR_DE_-GDRP. Nous caractérisons l'ensemble des valeurs possibles de chacun de ces hyperplans et l'ensemble des ordonnancements constructibles par notre algorithme. Les propriété obtenues seront utilisées à la section 4.4 et au chapitre 5.

Nous considérons un graphe uniformisé G_u supposé calculable. Lors de son traitement par l'algorithme, un ensemble de sous-graphes fortement connexes est produit. Nous notons $G_u(S, i)$ celui de ces sous-graphes engendré à profondeur i et qui contient S. $G_u(S, 1)$ est dont la composante fortement connexe de G_u qui contient S et $G_u(S, i + 1)$ est la composante fortement connexe de $(G_u(S, i))'$ qui contient S (s'il en existe une).

Nous définissons une grandeur l qui indique le délai induit par l'ordonnancement sur un arc donné de $G_u(S, i)$. Pour tout arc e de $G_u(S, i)$, $l(e)$ vaut 1 si et seulement si e n'appartient pas à $(G_u(S, i))''$ ($e \notin (G_u(S, i))''$) et si e est issu d'un nœud réel ($q(e)$ réel). $l(e)$ vaut 0 sinon. Si C est un ensemble d'arcs, $l(C)$ est la somme des valeurs de l sur les arcs constitutifs de C.

La partie linéaire de chacune des dimensions d'un ordonnancement peut en fait prendre n'importe quelle valeur dans un polyèdre dont la dimension est liée à celle de l'ensemble des poids des circuits inclus dans un multi-circuit de poids nul.

Lemme 18. *Soit une instruction S et une profondeur i inférieure à d_S ($i \leq d_S$).*

1. *L'ensemble des valeurs que peut prendre la partie linéaire de la $i^{\text{ème}}$ composante de l'ordonnancement de S est un polyèdre noté $\mathsf{P}(G_u(S, i))$:*

$$\mathsf{P}(G_u(S, i)) = \{X \mid \textit{pour tout circuit élémentaire } \mathcal{C} \textit{ de } G_u(S, i), X\boldsymbol{w}(\mathcal{C}) \geq l(\mathcal{C})\}. \tag{4.5}$$

2. *Tout point de ce polyèdre est inclus dans le cône caractéristique du polyèdre.*
3. *L'espace vectoriel engendré par $\mathsf{P}(G_u(S, i))$, noté $\mathsf{E}(G_u(S, i))$, est l'ensemble des vecteurs orthogonaux aux poids des circuits de $(G_u(S, i))'$:*

$$\mathsf{E}(G_u(S, i)) = \{X \mid \mathcal{C} \textit{ circuit élémentaire de } (G_u(S, i))' \Rightarrow X\boldsymbol{w}(\mathcal{C}) = 0\}. \tag{4.6}$$

4. *Si G_u est calculable, nous avons l'inclusion : $\mathsf{E}(G_u(S, i)) \subsetneq \mathsf{E}(G_u(S, i + 1))$.*
5. *$\mathsf{E}(G_u(S, i))$, l'espace vectoriel engendré par le polyèdre $\mathsf{P}(G_u(S, i))$, est égal à l'enveloppe affine de ce polyèdre. Nous pourrons donc confondre la dimension du polyèdre $\mathsf{P}(G_u(S, i))$, qui est par définition [74] la dimension de son enveloppe affine, et la dimension de l'espace vectoriel qu'il engendre.*

L'espace vectoriel $\mathsf{E}(G_u(S, i))$ rappelle l'ensemble Γ^+ utilisé par l'algorithme de Wolf et Lam (cf. section 2.3). La principale différence provient du fait que Γ^+ était construit à partir d'arcs de dépendance alors que $\mathsf{E}(G_u(S,i))$ est construit à partir de circuits de dépendance. Certaines de ces propriétés ont été mises en évidence par Georges-André Silber et le lecteur curieux en trouvera d'autres dans son mémoire de DEA [75].

DÉMONSTRATION. Nous démontrons une par une les assertions du lemme.

1. Il est immédiat que l'ensemble $\mathsf{P}(G_u(S,i))$ contient toutes les parties linéaires possibles par sommation le long de $\mathcal{C}$ des inéquations du système 4.1. Réciproquement, pour tout point X de $\mathsf{P}(G_u(S,i))$, il est possible de déterminer une constante ρ_T pour chaque instruction T de $G_u(S,i)$ pour que le vecteur et les constantes satisfassent le système 4.1. Pour cela, il suffit d'utiliser l'algorithme de détermination des constantes qui a été présenté à la fin du paragraphe 3.3.1 en pondérant chaque arc e d'un poids $\boldsymbol{w}'(e) = X\boldsymbol{w}(e) - l(e)$ (le poids d'un circuit élémentaire est alors toujours positif par définition de X).

   L'ensemble $\mathsf{P}(G_u(S,i))$ est un polyèdre (cf. définition 2, section 1.2).

2. Comme tout polyèdre, l'ensemble $\mathsf{P}(G_u(S,i))$ peut être décomposé en la somme d'un polytope et d'un cône: $\mathsf{P}(G_u(S,i)) = P(G_u(S,i)) + \Gamma(G_u(S,i))$. Dans notre cas, le cône $\Gamma(G_u(S,i))$ a la particularité de contenir tout point du polyèdre. En effet, d'après l'équation 4.5, si X est un élément de $\mathsf{P}(G_u(S,i))$, $(1+\lambda)X$ l'est également pour $\lambda \geq 0$ et tout point du polyèdre appartient au cône de $\mathsf{P}(G_u(S,i))$ [74].

3. Notons V l'espace vectoriel candidat : $V = \{X \mid \mathcal{C}$ circuit élémentaire de $(G_u(S,i))' \Rightarrow X\boldsymbol{w}(\mathcal{C}) = 0\}$. Par définition, tout vecteur de $\mathsf{P}(G_u(S,i))$ induit un délai positif sur tout circuit de $G_u(S,i)$. Autrement dit, tout vecteur X de $\mathsf{P}(G_u(S,i))$ a un produit scalaire positif avec le poids de tout circuit de $G_u(S,i)$. Prenons un ensemble $C_1, \ldots, C_p$ de circuits élémentaires de $G_u(S,i)$ qui forment un multi-circuit de poids nul. Nous avons : $\forall i \in [1,p], X\boldsymbol{w}(C_i) \geq 0$ et $\sum_{i=1}^p \boldsymbol{w}(C_i) = 0$. Donc $\sum_{i=1}^p X\boldsymbol{w}(C_i) = 0$ et $X\boldsymbol{w}(C_i) = 0$. Par conséquent, tout vecteur de $\mathsf{P}(G_u(S,i))$ a un produit scalaire nul avec le poids d'un circuit participant à un multi-circuit de poids nul. $\mathsf{P}(G_u(S,i))$ est donc inclus dans l'espace vectoriel V. Par conséquent, $\mathsf{E}(G_u(S,i)) \subset V$.

   Réciproquement, soit Y un point de cet ensemble V et X un point de $\mathsf{P}(G_u(S,i))$. Il existe alors un rationnel λ tel que $(\lambda X + Y)\boldsymbol{w}(\mathcal{C}) \geq l(\mathcal{C})$ pour tout circuit élémentaire $\mathcal{C}$ de $G_u(S,i)$ puisque Y induit un délai nul sur tous les circuits sur lesquels X induit un délai nul, c'est-à-dire sur les circuits de $(G_u(S,i))'$. $\mathsf{E}(G_u(S,i))$ est un espace vectoriel contenant X et $\lambda X + Y$. Il contient donc Y. $V \subset \mathsf{E}(G_u(S,i))$.

4. $(G_u(S,i+1))'$ est un sous-graphe de $G_u(S,i+1)$ qui est lui-même une composante fortement connexe de $(G_u(S,i))'$. Par conséquent, $(G_u(S,i+1))'$ est un sous-graphe de $(G_u(S,i))'$ et tout vecteur orthogonal au poids de tout circuit de $(G_u(S,i))'$ est alors orthogonal au poids de tout circuit de $(G_u(S,i+1))'$: $\mathsf{E}(G_u(S,i)) \subset \mathsf{E}(G_u(S,i+1))$.

   Raisonnons par l'absurde et supposons que l'inclusion n'est pas stricte. Nous avons donc l'égalité : $\mathsf{E}(G_u(S,i)) = \mathsf{E}(G_u(S,i+1))$. Soit X l'hyperplan produit par ORDONNANCEUR_DE_GDRP lors de l'appel sur $\mathsf{E}(G_u(S,i+1))$. Il existe une famille $X_1, \ldots, X_m$ de vecteurs de $\mathsf{P}(G_u(S,i))$ telle que $X = \sum_{i=1}^m X_i$. Comme nous l'avons montré au point précédent, chaque vecteur de $\mathsf{E}(G_u(S,i))$, donc chaque vecteur X_i, a un produit scalaire nul avec tout circuit de $(G_u(S,i))'$. Comme $(G_u(S,i))'$ contient $G_u(S,i+1)$, nous en déduisons que X induit un délai nul sur tous les circuits de $\mathsf{E}(G_u(S,i+1))$ ce qui contredit l'hypothèse sur la calculabilité de G_u.

5. L'enveloppe affine $\mathsf{A}(G_u(S,i))$ de $\mathsf{P}(G_u(S,i))$ est définie par :

$$\mathsf{A}(G_u(S,i)) = \{x \mid \exists m \in N^*, x = \sum_{j=1}^{m} \lambda_j x_j, \sum_{j=1}^{m} \lambda_j = 1, \forall j \in [1,m], x_j \in \mathsf{P}(G_u(S,i)), \lambda_j \in \mathbb{Q}\}.$$

La définition de l'espace vectoriel engendré par $\mathsf{P}(G_u(S,i))$ est identique excepté le fait qu'elle ne comporte pas la condition sur la somme des λ_i. Par conséquent : $\mathsf{A}(G_u(S,i)) = \mathsf{E}(G_u(S,i))$. Réciproquement, si x appartient à $\mathsf{P}(G_u(S,i))$, alors $2x$ appartient aussi à $\mathsf{P}(G_u(S,i))$ (voir le point 2). Donc $2 \times (x) + (-1) \times (2x)$ appartient à $\mathsf{A}(G_u(S,i))$. Donc l'origine appartient à $\mathsf{A}(G_u(S,i))$. Soit $y = \sum_{i=1}^{m} \lambda_i y_i$ un point de $\mathsf{E}(G_u(S,i))$. $y = \sum_{i=1}^{m} \lambda_i y_i + (1 - \sum_{i=1}^{m} \lambda_i) \times (2 \times (x) + (-1) \times (2x))$ et y est un point de $\mathsf{A}(G_u(S,i))$.

$\square$

La caractérisation proposée est « pratique ». En effet, le polyèdre $\mathsf{P}(G_u(S,i))$ peut être effectivement déterminé, ainsi que son cône et les générateurs de celui-ci. Pour ce faire nous pouvons utiliser par exemple la *Polyhedral Library* [76]. Cependant, le nombre de rayons et de lignes du polyèdre peuvent être en nombre exponentiel. En pratique, nous n'aurons en fait besoin que d'un point du polyèdre, qui peut être obtenu par programmation linéaire, et de $\mathsf{E}(G_u(S,i))$ qui peut être calculé directement et à moindre coût.

Dans la suite nous dirons d'un ordonnancement qu'il est « *constructible* par ORDONNANCEUR_DE_GDRP », ou simplement « *constructible* », s'il est une réponse possible de l'algorithme ORDONNANCEUR_DE_GDRP. Une fois connus les graphes $G_u(S,i)$ et les polyèdres $\mathsf{P}(G_u(S,i))$, nous pouvons choisir arbitrairement, pour tout graphe $G_u(S,i)$, un point du polyèdre $\mathsf{P}(G_u(S,i))$ et en faire la partie linéaire de la $i^{ème}$ composante de l'ordonnancement de toutes les instructions contenues dans $G_u(S,i)$. Pour obtenir définitivement un ordonnancement, il nous suffit de calculer les constantes comme suggéré dans la preuve du lemme 18. Les polyèdres $\mathsf{P}(G_u(S,i))$ définissant l'ensemble des valeurs possibles des différents hyperplans, cette méthode nous permet d'obtenir exactement tous les ordonnancements *constructibles*.

Au paragraphe 4.4.1 nous utiliserons notre connaissance du polyèdre $\mathsf{P}(G_u(S,i))$ et de sa dimension pour construire des boucles totalement permutables. Au chapitre 5, nous utiliserons toute la liberté sur le choix des parties linéaires pour construire un ordonnancement compatible avec un placement des calculs donné.

4.3.4 Exemple d'utilisation

Nous illustrons une nouvelle fois l'utilisation de notre algorithme. Cette fois-ci nous considérons un nid de boucles dont tout le parallélisme ne peut être détecté qu'avec une approximation des dépendances plus fine que les vecteurs de direction. Dans un graphe de dépendance réduit avec polyèdres de dépendance nous obtenons une S_1-latence en $O(N)$ et une S_2-latence en $O(N^2)$.

L'exemple 18 est présenté à la figure 4.7 avec son graphe de dépendance réduit G_o valué par des vecteurs de direction. Celui-ci a été déterminé par l'analyseur de dépendance Tiny [80]. L'algorithme d'uniformisation nous permet de construire le graphe uniformisé G_u (figure 4.8).

Le graphe uniformisé possède un multi-circuit de poids nul qui implique tous les arcs dont le poids est orthogonal à $(1,0,0)$ (cf. figure 4.8). Dans G', les instructions S_1 et S_2 sont incluses dans la même composante fortement connexe qui contient un multi-circuit de poids nul qui traverse le nœud réel S_2. S_1 disparaît à profondeur deux mais S_2 n'est éliminé qu'à profondeur trois. S_2 est donc

Exemple 18

```
For i = 1 to n do
  For j = 1 to n do
    For k = 1 to n do
      a(i, j, k) = a(i, j - 1, k + j) + b(j, i - 1, k)
      b(i, j, k) = b(i, j, k -1) + a(i, j, k)
```

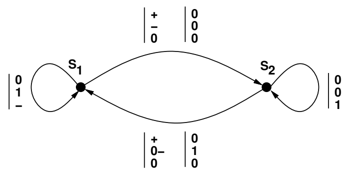


FIG. 4.7: *Exemple 18 et son graphe de dépendance réduit avec vecteurs de direction.*

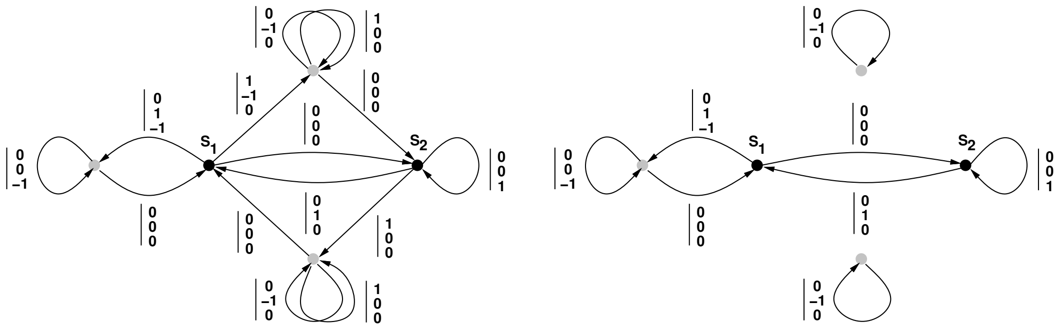


FIG. 4.8: *Graphe uniformisé de l'exemple 18 et son sous-graphe des multi-circuits de poids nul.*

trouvé purement séquentiel quand un degré de parallélisme est détecté pour S_1. Un ordonnancement multidimensionnel possible est alors : $(i, 2j)$ pour S_1 et $(i, 2j+1, k)$ pour S_2. Le code résultant est présenté à la figure 4.9.

```
ForSeq i = 1 to n do
  ForSeq j = 1 to n do
    ForPar k = 1 to n do
      a(i, j, k) = a(i, j - 1, k + j) + b(j, i - 1, k)
    ForSeq k = 1 to n do
      b(i, j, k) = b(i, j, k -1) + a(i, j, k)
```

FIG. 4.9: *Version parallélisée de l'exemple 18.*

Il est à remarquer que c'est exactement le code qu'aurait trouvé ALLEN-KENNEDY ou une version de WOLF-LAM avec distribution de boucles entre les étapes de recherche d'ensemble de boucles permutables. Cependant, si les dépendances de l'exemple sont l'objet d'une analyse plus fine, il apparaît que le GDRP uniformisé de la figure 4.10 en propose une approximation conservative qui contient plus de parallélisme. La référence au tableau b en membre droit de S_1 y génère deux dépendances : une dépendance de flot dont le polyèdre de dépendance a un sommet, $(0, 1, 0)$, et un rayon, $(1, -1, 0)$; une anti-dépendance dont le polyèdre de dépendance a un sommet, $(1, -2, 0)$, et le même rayon que l'autre polyèdre, à savoir $(1, -1, 0)$.

Remarquez à la figure 4.10 comment la précision accrue sur les dépendances à changé la structure de G', le sous-graphe des multi-circuits de poids nul. S_1 est maintenant éliminé au premier niveau de décomposition et S_2 au deuxième. Nous détectons donc un degré de parallélisme en plus pour chacune des instructions et nous obtenons comme ordonnancement possible : $(4i + 2j)$ pour S_1 et $(4i + 2j + 1, k)$ pour S_2. Le nouveau code parallélisé est présenté à la figure 4.11.

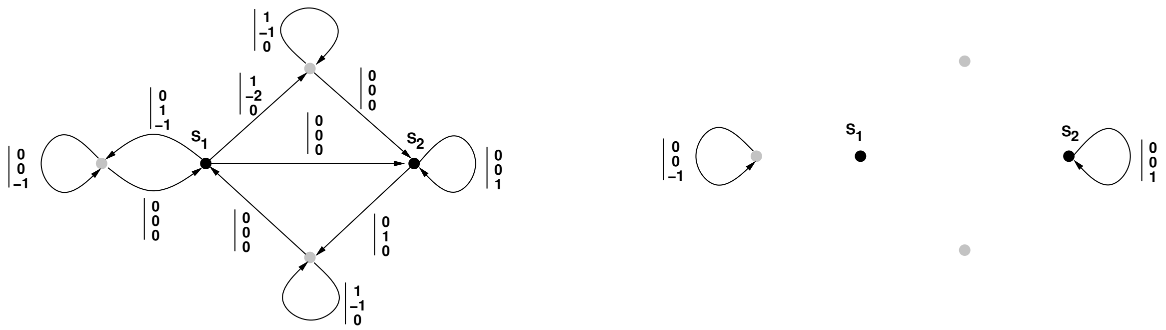


FIG. 4.10: *GDRP uniformisé G_u pour l'exemple 18 et son graphe des multi-circuits de poids nul.*

```
ForSeq j = 3 to 3n do
  ForPar k = 1 to n do
    ForPar i = max(1, ⌈(j−n)/2⌉) to min(n, ⌊(j−1)/2⌋) do
      a(i, j-2i, k) = a(i, j - 2i - 1, k + j -2i) + b(j - 2i, i - 1, k)
  ForSeq k = 1 to n do
    ForPar i = max(1, ⌈(j−n)/2⌉) to min(n, ⌊(j−1)/2⌋)
      b(i, j - 2i, k) = b(i, j - 2i, k -1) + a(i, j - 2i, k)
```

FIG. 4.11: *Deuxième version parallélisée de l'exemple 18.*

4.4 Extensions

Nous proposons ici deux extensions de notre algorithme. Nous montrons d'abord comment le modifier pour qu'il génère les boucles permutables nécessaires au partitionnement. Cette première extension est naturelle et se fait aisément. La deuxième étend notre algorithme pour qu'il puisse traiter des nids non parfaits. Cette extension est beaucoup plus technique que la première, non pas tant dans sa formulation — le nouvel algorithme n'est pas plus compliqué que l'ancien — mais dans sa justification.

4.4.1 Génération de boucles totalement permutables

Nous pouvons facilement reconvertir l'algorithme ORDONNANCEUR_DE_GDRP pour qu'il produise des boucles totalement permutables plutôt que des ordonnancements. La présentation ci-dessous est sommaire. Nous voulons juste montrer qu'il est effectivement possible d'étendre notre algorithme. Le lecteur trouvera plus de détails dans [75, 30, 31].

Considérons un appel ORDONNANCEUR_DE_GDRP(G, i) et une composante fortement connexe G_j de G. Soit S une instruction quelconque de G_j ($G_j = G_u(S, i)$, avec les notations du paragraphe 4.3.3). À l'étape 4 de l'algorithme, au lieu de construire un unique hyperplan faiblement séparateur, nous en construisons $h_i = \dim(\mathsf{E}(G_u(S, i))) - \dim(\mathsf{E}(G_u(S, i-1)))$ (avec la convention $\dim(\mathsf{E}(G_u(S, 0))) = 0$). Pour cela, nous prenons h_i vecteurs de $\mathsf{P}(G_u(S, i))$ qui forment une famille libre avec les $\dim(\mathsf{E}(G_u(S, i-1)))$ hyperplans précédemment choisis. Nous complétons chacun de ces h_i vecteurs en une fonction affine satisfaisant le système 4.1, comme si nous voulions en faire la $i^{ème}$ composante d'un ordonnancement. Les h_i fonctions affines ainsi obtenues engendrent autant de boucles qui sont communes à toutes les instructions de G_j et qui sont totalement permutables, chacune des fonctions affines induisant par définition un délai positif sur tous les arcs de G_j.

Il est tout à fait possible que le nombre total de boucles ainsi produites pour l'instruction S ($\dim(\mathsf{E}(G_u(S,d_S)))$) soit strictement inférieur à la dimension du domaine d'itération (n). Dans ce cas, nous complétons arbitrairement l'ensemble des boucles pour qu'il décrive le domaine d'itération dans son ensemble.

Remarque : Dans la pratique, nous n'avons pas besoin de construire $\mathsf{P}(G_u(S,i))$ pour obtenir nos boucles permutables. Nous avons besoin d'un seul point X de $\mathsf{P}(G_u(S,i))$, c'est-à-dire d'une solution du système 4.1, et d'une base $b_1,\ldots,b_{\dim(\mathsf{E}(G_u(S,i)))}$ de $\mathsf{E}(G_u(S,i))$, qui peut être directement calculée à partir de $(G_u(S,i))'$. À partir du vecteur X et d'un vecteur b_j nous pouvons obtenir un point $\mu_j X + b_j$ de $\mathsf{P}(G_u(S,i))$ (cf. la démonstration du lemme 18). La famille des vecteurs $\mu_j X + b_j$ est libre si $\sum_j \lambda_j \mu_j \neq -1$ où $X = \sum_j \lambda_j b_j$. Cette propriété est facile à vérifier puisque, si $\mu_j X + b_j$ est un point de $\mathsf{P}(G_u(S,i))$, il en va de même de $\alpha_j X + b_j$ pour $\alpha_j \geq \mu_j$. Nous n'avons plus qu'à prendre, parmi les $\mu_j X + b_j$, les h_i vecteurs nécessaires.

Cette méthode nous permet d'exhiber des boucles totalement permutables qui permettront ultérieurement d'appliquer au nid des techniques de partitionnement. Il n'est cependant pas nécessaire d'exiger comme nous l'avons fait que chaque partie linéaire satisfasse toutes les dépendances satisfiables à son niveau. Cette exigence nous prive entre autres des boucles parallèles, qui sont engendrés par des fonctions affines qui induisent un délai nul sur chacune des dépendances. Au chapitre 7, nous évoquerons un relâchement possible des contraintes.

4.4.2 Nids non parfaits

Jusqu'à présent nous n'avons considéré que des nids de boucles parfaits. Pour appliquer nos techniques à un nid de boucles non parfait il faudrait commencer par le rendre parfait. Plutôt que d'adapter les nids non parfaits à notre algorithme, nous montrons comment faire l'inverse.

Nous avons vu à la section 1.2 (note 5) que les vecteurs de distance d'un nid non parfait n'étaient définis que pour les boucles englobantes communes aux instructions en relation de dépendance : si S et T ont exactement $n_{S,T}$ boucles englobantes communes, alors le vecteur de distance associé à la dépendance $S(\boldsymbol{i}) \Rightarrow T(\boldsymbol{j})$ est $\boldsymbol{j}|_{n_{S,T}} - \boldsymbol{i}|_{n_{S,T}}$, où $\boldsymbol{p}|_k$ désigne le vecteur formé des k premières composantes du vecteur $\boldsymbol{p}$.

La dimension du polyèdre associé à un arc du graphe de dépendance réduit polyédrique d'un nid non parfait dépend donc de l'arc considéré et n'est plus une constante du graphe. Nous voulons néanmoins ordonnancer de tels graphes. Pour ce faire, nous allons appliquer l'algorithme « ensemble de boucles parfaitements imbriquées » par « ensemble de boucles parfaitements imbriquées ».

Considérons un sous-graphe fortement connexe G du graphe uniformisé G_u. Comme précédemment, G_u est produit par UNIFORMISATION à partir de G_o. Notons $n(G)$ la plus petite dimension du poids d'un arc de G : $n(G) = \min_{e\in G} \dim(\boldsymbol{w}(e))$. $n(G)$ est exactement le nombre de boucles englobant toutes les instructions incluses dans G, si G contient au moins un arc.[3]

Intuition

La description de l'intuition contient de nombreux abus de langage et de notations, et oublie certains détails. De plus, elle ne démontre rien.

3. Considérons un arc e tel que $\dim(\boldsymbol{w}(e)) = n(G)$. Les instructions reliées par l'arc e ont exactement $n(G)$ boucles communes, par définition des vecteurs de distance, et ces boucles sont les boucles les plus externes entourant ces instructions. Il y a donc au plus $n(G)$ boucles englobant toutes les instructions incluses dans G. Nous procédons de proche en proche pour montrer qu'il y a exactement $n(G)$ boucles englobants toutes les instructions du nid et que ce sont les boucles les plus externes. Soit f un arc de G, $\dim(\boldsymbol{w}(f)) \geq n(G)$. Les instructions reliées par l'arc f ont au moins $n(G)$ boucles communes. Si une des extrémités de f est entouré par les $n(G)$ boucles communes aux extrémités de e, comme ces boucles sont les plus externes entourant cette extrémité de f, il en va de même de l'autre extrémité.

Soient deux opérations $S(\boldsymbol{i})$ et $T(\boldsymbol{j})$, la deuxième dépendant de la première. Le vecteur de distance $\boldsymbol{d} = \boldsymbol{j}|_{n_{S,T}} - \boldsymbol{i}|_{n_{S,T}}$ ne nous donne aucune information sur les composantes des vecteurs d'itération qui ne correspondent pas à des boucles communes. Nous aimerions avoir un « vecteur de distance » que nous pourrions ajouter à $\boldsymbol{i}$ pour obtenir $\boldsymbol{j}$ (en oubliant les problèmes de différence de dimension des deux vecteurs). Nous ne pouvons pas utiliser $\boldsymbol{d}$ de cette manière puisqu'il lui manque des dimensions. Si nous n'avons pas d'autres informations sur la dépendance que la valeur de $\boldsymbol{d}$, nous ne pouvons approcher les composantes manquantes que par le symbole « $*$ » des vecteurs de direction. Nous complétons donc la représentation polyédrique des dépendances en ajoutant des « $*$ » pour compléter tous les vecteurs de cette représentation en vecteurs de dimension n (la profondeur maximale du nid). Les vecteurs d'itération d'une instruction S sont *a priori* eux aussi étendus pour devenir de dimension n. En fait, seules les n_S premières composantes des « nouveaux » vecteurs d'itération ont un sens.

Grâce à l'ajout des dimensions, nous obtenons une nouvelle représentation polyédrique des dépendances que nous pouvons ordonnancer au moyen de ORDONNANCEUR_DE_GDRP puisque tous ses polyèdres ont même dimension. Les deux représentations étant équivalentes, nous conservons même la propriété d'optimalité.

Un symbole « $*$ » dans la $i^{ème}$ composante d'un vecteur de direction oblige la $i^{ème}$ composante de tout hyperplan a être nulle (le polyèdre correspondant à la dépendance contient une ligne dont la seule composante non nulle est la $i^{ème}$). Par conséquent, seules les $n(G)$ premières composantes de l'hyperplan recherché peuvent ne pas être nulles. L'algorithme va donc rechercher un ordonnancement en ne considérant que les $n(G)$ boucles entourant toutes les instructions, c'est-à-dire en ne considérant que les $n(G)$ premières composantes des poids des arcs.

Nous obtenons ainsi l'algorithme ci-dessous que nous étudions formellement dans le reste de cette section.

ORDONNANCEUR_DE_GDRP_NP(Graphe uniformisé G, k)

1. Décomposer G en ses composantes fortement connexes et ne conserver que les composantes G_1, ..., G_i, ..., G_s qui contiennent au moins un nœud réel. Appeler les étapes 2, 3, 4, 5, puis 6 sur chacun de ces sous-graphes G_i et renvoyer le ET logique des réponses.

2. Calculer $n(G_i)$.

3. Construire G'_i le sous-graphe de G_i généré par les arcs qui appartiennent à au moins un multi-circuit dont le poids est nul quand ne sont considérées que les $n(G_i)$ premières composantes de chaque poids.

4. Poser $G''_i = G'_i$. Pour tout arc polyédrique e de G_o, si G'_i contient un arc correspondant à un sommet du polyèdre $\mathcal{P}(e)$, rajouter à G''_i tous les arcs issus de la transcription de $\mathcal{P}(e)$ qu'il ne contient pas déjà.

5. Choisir un vecteur X de $\mathbb{Q}^{n(G_i)}$ et, pour chaque nœud S de G_i, une constante ρ_S tels que :

$$\begin{cases} e = (q(e), t(e)) \in G''_i \text{ ou } q(e) \text{ est un nœud virtuel} & \Rightarrow \quad X\boldsymbol{w}(e)|_{n(G_i)} + \rho_{t(e)} - \rho_{q(e)} \geq 0 \\ e = (q(e), t(e)) \notin G''_i \text{ et } q(e) \text{ est un nœud réel} & \Rightarrow \quad X\boldsymbol{w}(e)|_{n(G_i)} + \rho_{t(e)} - \rho_{q(e)} \geq 1 \end{cases}$$

Pour tout nœud S de G_i, nous posons $\rho^k_S = \rho_S$, $X^k_S|_{n(G_i)} = X$ et nous fixons à zéro les $n_S - n(G_i)$ dernières composantes de X^k_S.

6. – Si G_i'' est vide ou ne contient que des nœuds virtuels, renvoyer VRAI.
   – Si G_i'' est égal à G_i, renvoyer FAUX.
   – Sinon, renvoyer ORDONNANCEUR_DE_GDRP_NP(G_i'', $k+1$).

Remarque. À l'étape 6 nous avons remplacé le test « si G_i'' est fortement connexe et contient au moins un nœud réel » par le test « si G_i'' est égal à G_i » car, ici, G_i'' peut être fortement connexe et G être néanmoins calculable comme le montre le graphe de la figure 4.12. Dans cet exemple $n(G)$ vaut un. Quand nous ne considérons que la première composante des vecteurs, la dépendance de S_1 vers S_2 et celle de S_2 vers S_1 valent un. L'auto-dépendance de S_1 est de poids zéro. Par conséquent, G' et G'' sont égaux et ne contiennent que S_1 et son auto-dépendance. G'' est fortement connexe mais G est calculable puisqu'il admet l'ordonnancement (i, j) pour S_1 et (i) pour S_2.

Pour comprendre le problème il nous faut revenir à la description intuitive et aux vecteurs de direction. En fait, nous avons étendu les vecteurs de distance entre S_1 et S_2 en leur rajoutant une deuxième composante égale à « $*$ ». Nous appliquons ORDONNANCEUR_DE_GDRP à ce graphe. G' contient alors, en plus de S_1 et de son auto-dépendance, les nœuds virtuels issus de l'uniformisation des « $*$ » et leurs auto-dépendances. Ici, G' n'est donc pas fortement connexe. La simplification qui consiste à ne considérer que les $n(G)$ premières composantes de chaque vecteur à fait disparaître artificiellement des nœuds virtuels de G' n'appartenant pas à une composante fortement connexe contenant un nœud réel.

L'étape 6 peut maintenant appeler récursivement l'algorithme sur un graphe $H = G_i''$ dont aucune arête n'est satisfiable. Pour un tel graphe $H'' = H$ et l'algorithme se termine.

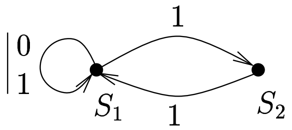


FIG. 4.12: *Exemple de graphe calculable pour lequel G' est fortement connexe.*

Nous devons montrer que l'algorithme renvoie VRAI si et seulement si le graphe étudié est calculable et qu'il produit bien un ordonnancement. Nous voulons également prouver l'optimalité de cet ordonnancement.

Pour démontrer la calculabilité comme l'optimalité, nous avons besoin de construire des chemins de dépendance dans le graphe de dépendance apparent de G_o. Nous ne pouvons pas utiliser directement G_u comme nous l'avons fait précédemment, le poids d'un chemin de G_u n'étant pas clairement défini. En effet, prenons un arc e de G_u. Pour tout couple de points $(\boldsymbol{i}, \boldsymbol{j})$ de $(\mathcal{D}_{q(e)}, \mathcal{D}_{t(e)})$, $S_{t(e)}(\boldsymbol{j})$ dépend de $S_{q(e)}(\boldsymbol{i})$ si et seulement si $\boldsymbol{j}|_{n_{q(e),t(e)}} - \boldsymbol{i}|_{n_{q(e),t(e)}} = \boldsymbol{w}(e)$. Par conséquent, si toutes les boucles englobant $S_{t(e)}$ n'entourent pas $S_{q(e)}$, plusieurs instances différentes de $S_{t(e)}$ peuvent dépendre de $S_{q(e)}(\boldsymbol{i})$ avec le vecteur de distance $\boldsymbol{w}(e)$. Pour pouvoir construire facilement nos chemins nous avons besoin d'une structure plus régulière.

Construction d'un graphe polyédrique équivalent à G_o.

Nous créons ci-dessous un nouveau graphe de dépendance réduit polyédrique, noté H_o, « équivalent » à G_o. H_o est une version formelle du graphe avec les « $*$ » qui figure dans la présentation de l'intuition. Nous posons $n = \max_S n_S$. Le graphe H_o est définit comme suit :

- H_o a les mêmes nœuds et arcs que G_o;
- l'arc f de H_o correspondant à l'arc e de G_o est étiqueté par le polyèdre $\mathcal{P}(f)$:
  - dont les sommets et rayons sont ceux de $\mathcal{P}(e)$ auxquels ont été rajoutés $n - n_{q(e),t(e)}$ zéros terminaux;
  - qui contient les lignes de $\mathcal{P}(e)$ auxquels ont été rajoutés $n - n_{q(e),t(e)}$ zéros terminaux;
  - qui contient $n - n_{q(e),t(e)}$ lignes dites *supplémentaires*, les coordonnées de la $i^{ème}$ de ces lignes étant toutes nulles sauf celle de rang $i + n_{q(e),t(e)}$ qui est égale à un (ces lignes codent les « $*$ »).

Nous notons H_u le graphe uniformisé de H_o, obtenu grâce à l'algorithme UNIFORMISATION. Nous pouvons identifier un circuit $\mathcal{C}$ de G_u à un circuit de H_u, appelé *circuit correspondant.* En effet, G_o et H_o ont exactement même structure et l'uniformisation de leurs polyèdres respectifs donne naissance aux mêmes arcs, aux lignes *supplémentaires* et à un cas particulier près : si un polyèdre de G_u ne contient ni rayons ni lignes il ne donnera pas naissance à un nœud virtuel contrairement au polyèdre équivalent de H_u, et ce à cause des lignes *supplémentaires.* Si $\mathcal{C}$ contient un arc dû à l'uniformisation d'un sommet d'un tel polyèdre $\mathcal{P}(e)$, son circuit correspondant contient l'arc issu de l'uniformisation du sommet équivalent du polyèdre correspondant et il contient aussi l'arc de poids nul de n_e vers $t(e)$ (pour conserver les propriétés de connectivité).

Pour construire H_o nous avons étendu tous les vecteurs de G_o en des vecteurs de dimension $n = \max_S n_S$. Les dimensions rajoutées nous permettent de simuler la liberté sur les composantes des vecteurs d'itérations qui ne correspondent pas à des boucles englobantes communes aux instructions en relation de dépendance. L'inconvénient de ces dimensions, c'est qu'elles n'ont pas de signification réelle et que le domaine d'itération d'un sommet S de H_o est de dimension n alors que seules les n_S premières composantes ont un sens. L'avantage, c'est que pour construire un chemin de dépendance du graphe de dépendance apparent de G_o il suffit de parcourir H_u : si f est un chemin atomique de H_u (f relie deux nœuds réels et ne contient aucun nœud réel intermédiaire) alors $S_{t(f)}((\boldsymbol{i} + \boldsymbol{w}(f))|_{n_{S_{t(f)}}})$ dépend de $S_{q(f)}(\boldsymbol{i}|_{n_{S_{q(f)}}})$ dans G_o.

Calculabilité

Remarquons au préalable que l'algorithme se termine toujours puisqu'un appel sur un graphe G ne génère des appels récursifs que sur des graphes strictement inclus dans G.

Un graphe de dépendance réduit est calculable si et seulement si son graphe de dépendance apparent ne contient pas de circuits, c'est-à-dire si et seulement si il n'existe pas un chemin de longueur non nulle dont les extrémités sont la même instance de la même instruction.

Considérons un graphe G_o non calculable. Son GDA contient donc un circuit C. Soit $n(C)$ la plus petite dimension d'un vecteur de distance valuant un arc constitutif de C. Nous notons C_u un circuit de G_u équivalent à C. C_u contient au moins un nœud réel. C (resp. C_u) est de poids nul quand ne sont considérées que les j premières composantes des poids de ses arcs, pour $j \leq n(C)$, car C décrit une dépendance d'une instance d'une instruction vers la même instance de la même instruction. Soit G un sous-graphe fortement connexe de G_u contenant C_u. Alors $n(G) \leq n(C)$ et $C_u \subset G' \subset G''$. Par conséquent, un appel de l'algorithme sur un graphe G contenant C_u générera un appel de l'algorithme sur un graphe H contenant C_u ou conclura à la non calculabilité de G. Comme l'algorithme se termine toujours, ORDONNANCEUR_DE_GDRP_NP appelé sur un graphe non calculable renvoie FAUX.

Réciproquement, considérons un graphe pour lequel notre algorithme renverrait FAUX. À un moment de la décomposition il apparaît donc une composante fortement connexe G_i telle que

$(G_i)'' = G_i$. G_i contient forcément un nœud réel puisque l'algorithme n'est jamais appelé sur un graphe qui ne contient que des nœuds virtuels. Par conséquent, il existe un multi-circuit de poids nul de G_i, quand ne sont considérées que les $n(G_i)$ premières coordonnées de chaque vecteur, contenant tous les nœuds, virtuels ou réels, de G_i. Ce multi-circuit est en fait un circuit, noté $\mathcal{C}$. Le circuit $\mathcal{C}'$ correspondant de H_u contient également tous les nœuds de H_u et en particulier le nœud virtuel T issu de l'uniformisation de l'arc dont le poids était de plus petite dimension ($n(G_i)$). Les $n(G_i)$ premières composantes de $\mathcal{C}'$ sont nulles. Nous pouvons transformer $\mathcal{C}'$ en un circuit de poids nul en lui ajoutant le bon nombre de fois les auto-dépendances idoines de T correspondant à des lignes supplémentaires : si la $j^{ème}$ composante du poids de $\mathcal{C}'$, $j \geq 1 + n(G_i)$, est égale à $x \neq 0$, nous ajoutons à $\mathcal{C}'$ $|x|$ fois l'auto-dépendance de T dont le poids est nul à l'exception de la $j^{ème}$ composante qui vaut « 1 » (resp. « -1 ») si x est positif (resp. négatif). Nous avons construit un circuit de H_u de poids nul et contenant un nœud réel, c'est-à-dire un chemin de dépendance de H_u dont les extrémités sont la même instance de la même instruction et qui contient un nœud réel. Or, comme nous l'avons déjà fait remarquer, pour construire un chemin de dépendance du graphe de dépendance apparent de G_o il suffit de parcourir H_u : si f est un chemin atomique de H_u (f relie deux nœuds réels et ne contient aucun nœud réel intermédiaire) alors $S_{t(f)}((\boldsymbol{i} + \boldsymbol{w}(f))|_{n_{S_{t(f)}}})$ dépend de $S_{q(f)}(\boldsymbol{i}|_{n_{S_{q(f)}}})$ dans G_o. Par conséquent, G_o n'est pas calculable.

Nous avons pu transformer le circuit $\mathcal{C}'$ en un circuit de poids nul uniquement parce que $\mathcal{C}'$ contenait le sommet virtuel correspondant à un arc dont le poids est de plus petite dimension. Nous n'aurions pas pu être assuré de cette propriété si nous avions mis comme condition d'arrêt : « si G_i'' est fortement connexe et contient au moins un nœud réel » (cf. l'exemple de la figure 4.12).

Validité de l'ordonnancement

Remarquons tout d'abord que l'étape 5 de l'algorithme est toujours réalisable puisque l'étape correspondante de l'algorithme ORDONNANCEUR_DE_GDRP l'est. En effet, cette étape 5 construit (en apparence) un hyperplan faiblement séparateur pour un GDRP classique, le fait que ce GDRP a été construit à partir du graphe de dépendance d'un nid non-parfait n'apparaît pas dans les équations.

Lemme 19. *Si $G_o = (V, E)$ est calculable, la fonction affine multidimensionnelle Θ produite par* ORDONNANCEUR_DE_GDRP_NP *:*

$$(S,\ \boldsymbol{i}) \ \mapsto\ (X_S^1\boldsymbol{i} + \rho_S^1,\ \ldots,\ X_S^{d_S}\boldsymbol{i} + \rho_S^{d_S},\ 0,\ \ldots,\ 0),$$

où $\boldsymbol{i}$ appartient à $\mathcal{D}_S$, définit un ordonnancement multidimensionnel de G_o.

DÉMONSTRATION. Cette démonstration est une adaptation de celle écrite pour le lemme 16.

Pour prouver que la fonction Θ définit bien un ordonnancement, nous devons montrer qu'elle vérifie toutes les dépendances représentées par le GDRP G_o. Pour tout arc polyédrique $e = (q(e), t(e))$ de G_o, pour tout point $\boldsymbol{w}(e)$ du polyèdre $\mathcal{P}(e)$ et tout points $\boldsymbol{i}$ et $\boldsymbol{j}$ de domaines respectifs $\mathcal{D}_{q(e)}$ et $\mathcal{D}_{t(e)}$, nous devons donc assurer l'implication :

$$\boldsymbol{j}|_{n_{q(e),t(e)}} - \boldsymbol{i}|_{n_{q(e),t(e)}} = \boldsymbol{w}(e) \quad \Rightarrow \quad \lfloor\Theta(t(e), \boldsymbol{j})\rfloor \succ \lfloor\Theta(q(e), \boldsymbol{i})\rfloor.$$

Pour ce faire il nous suffit d'établir l'existence d'un entier k, $k \in [1, d]$, tel que :

$$\boldsymbol{j}|_{n_{q(e),t(e)}} - \boldsymbol{i}|_{n_{q(e),t(e)}} = \boldsymbol{w}(e) \ \Rightarrow\ \begin{cases} X_{t(e)}^i\boldsymbol{j} + \rho_{t(e)}^i \geq X_{q(e)}^i\boldsymbol{i} + \rho_{q(e)}^i & \forall i,\ 1 \leq i \leq k-1 \\ X_{t(e)}^k\boldsymbol{j} + \rho_{t(e)}^k \geq X_{q(e)}^k\boldsymbol{i} + \rho_{q(e)}^k + 1 & \end{cases} \tag{4.7}$$

Si G_o est calculable, tout arc du graphe correspondant au sommet d'un polyèdre est éliminée au cours de la décomposition. L'étape 4 de l'algorithme garde ensemble dans le graphe tous les arcs issus de l'uniformisation d'un même arc polyédrique jusqu'à la profondeur minimale k où plus aucun arc correspondant à un sommet de $\mathcal{P}(e)$ n'appartient à un multi-circuit de poids nul. Ces arcs sont donc toujours dans la même composante fortement connexe et les séquences de vecteurs $X^i_{t(e)}|_{n_{q(e),t(e)}}$, $X^i_{q(e)}|_{n_{q(e),t(e)}}$ et $X^i_{n_e}|_{n_{q(e),t(e)}}$ sont confondues jusqu'au niveau k inclus. Nous omettons donc l'indice et écrivons simplement $X^i|_{n_{q(e),t(e)}}$. De plus, les composantes des vecteurs $X^i_{t(e)}$, $X^i_{q(e)}$ et $X^i_{n_e}$ [4] de rang supérieurs à $n_{q(e),t(e)}$ sont toutes nulles.

Donc, si $\boldsymbol{j}|_{n_{q(e),t(e)}} - \boldsymbol{i}|_{n_{q(e),t(e)}} = \boldsymbol{w}(e)$, la différence $(X^i_{t(e)}\boldsymbol{j} + \rho^i_{t(e)}) - (X^i_{q(e)}\boldsymbol{i} + \rho^i_{q(e)})$ se réécrit :

$$(X^i|_{n_{q(e),t(e)}}\boldsymbol{j}|_{n_{q(e),t(e)}} + \rho^i_{t(e)}) - (X^i|_{n_{q(e),t(e)}}\boldsymbol{i}|_{n_{q(e),t(e)}} + \rho^i_{q(e)}) = X^i|_{n_{q(e),t(e)}}\boldsymbol{w}(e) + \rho^i_{t(e)} - \rho^i_{q(e)}.$$

Le système 4.7 se réécrit alors :

$$\begin{cases} X^i|_{n_{q(e),t(e)}}\boldsymbol{w}(e) + \rho^i_{t(e)} - \rho^i_{q(e)} \geq 0 & \forall i,\ 1 \leq i \leq k-1 \\ X^k|_{n_{q(e),t(e)}}\boldsymbol{w}(e) + \rho^k_{t(e)} - \rho^k_{q(e)} \geq 1 \end{cases}$$

Nous prouvons qu'il est satisfait comme nous l'avons fait dans la démonstration du lemme 16. □

Optimalité

Nous pouvons donc simplement appliquer à H_u le théorème 8 sur la longueur des chemins des GDRP pour obtenir un résultat équivalent pour G_u. Nous prouvons ainsi que la latence d'une instruction pour l'ordonnancement est du même ordre de grandeur que les plus longs chemins de dépendances concernant cette instruction. Pour cela il nous suffit de prouver que la profondeur de décomposition d'un nœud de G_u par ORDONNANCEUR_DE_GDRP_NP, qui définit la magnitude de la latence de l'ordonnancement, est égale à la profondeur de décomposition du nœud équivalent de H_u par ORDONNANCEUR_DE_GDRP, qui définit la magnitude de la longueur des chemins.

Passage des multi-circuits de poids nul de G_u à ceux de H_u. Soit G_i une composante fortement connexe de G_u, et H_i la composante fortement connexe correspondante de H_u. Considérons l'arc e de G_i dont le poids est de dimension minimale $(n(G_i))$ et l'arc f_u de H_u correspondant. f_u est issu de l'uniformisation d'un arc polyédrique f de H_o. $\mathcal{P}(f)$ contient donc une ligne de poids : $\boldsymbol{l}(j) = (\underbrace{0,\dots,0}_{j+n(G_i)-1}, 1, \underbrace{0,\dots,0}_{n-n(G_i)-j})$, pour tout $j \in [1, n-n(G_i)]$. Le nœud n_f de H_u porte donc une auto-dépendance de poids $\boldsymbol{l}(j)$ et une de poids $-\boldsymbol{l}(j)$ pour tout j dans $[1, n-n(G_i)]$.

Soit C un multi-circuit de G_i, de poids nul quand ne sont considérées que les $n(G_i)$ premières composantes de chaque poids. Soit C' le multi-circuit de H_i correspondant. Les $n(G_i)$ premières composantes du poids de C' sont nulles. Il suffit donc d'ajouter à C' des auto-dépendances de n_f bien choisies pour le transformer en un multi-circuit de poids nul de H_i.

Réciproquement, soit C' un multi-circuit de poids nul de H_i. Nous notons C'' le multi-circuit de H_i obtenu en éliminant de C' toutes les auto-dépendances de nœuds virtuels provenant de l'uniformisation de lignes dites *supplémentaires*. Les $n(G_i)$ premières composantes du poids des auto-dépendances éliminées sont nulles. Donc $\boldsymbol{w}(C'')|_{n(G_i)} = \boldsymbol{w}(C')|_{n(G_i)} = 0$. Par conséquent, C,

4. Nous décidons arbitrairement que le vecteur $X^i_{n_e}$ est de même dimension que $X^i_{t(e)}$, ce qui n'a aucune importance vu qu'au plus seules les composantes de rangs inférieurs à $n_{q(e),t(e)}$ comptent.

le multi-circuit de G_i correspondant à C'', est de poids nul pour G_u, c'est-à-dire quand ne sont considérées que les $n(G_i)$ premières composantes.

Nous déduisons de ce qui précède que l'ensemble des multi-circuits de poids nuls de H_i est équivalent à l'ensemble des multi-circuits de poids nuls de G_i, à des lignes *supplémentaires* près. Vu l'étape 4, $(H_i)'$ est égal à $(G_i)'$ plus quelques lignes *supplémentaires* qui sont des auto-dépendances de nœuds virtuels issus d'arcs polyédriques n'appartenant pas à $(H_i)'$. L'ensemble des composantes fortement connexes de $(H_i)'$ contenant au moins un nœud réel et l'ensemble équivalent de (G'_i) sont donc égaux. Après l'étape 1 suivant l'appel récursif, les calculs se font sur les mêmes composantes fortement connexes et l'égalité des profondeurs de décomposition s'établit par récurrence.

Conclusion

Nous avons proposé un algorithme qui prend en entrée le GDRP d'un nid non-parfait, les vecteurs de distance n'étant définis que sur les composantes des vecteurs d'itérations qui correspondent à des boucles communes aux instructions en dépendance. L'ordonnancement généré par notre algorithme est optimal. Nous sommes donc à même de détecter tout le parallélisme contenu dans notre modélisation des dépendances, ce qui ne veut pas dire que nous n'aurions pas pu trouver plus de parallélisme en transformant le nid de boucles en un nid parfait avant d'en analyser les dépendances et de le paralléliser. Au contraire, l'analyse des dépendances sur une version parfaite du nid révélerait plus de parallélisme car serait plus précise.

4.5 Retour sur les algorithmes classiques d'ordonnancement

Pour finir, nous nous devons de comparer notre algorithme à ceux que nous avons présentés au chapitre 2 et de revenir sur les souhaits que nous avons exprimés dans ce même chapitre.

Algorithme d'Allen et Kennedy

Notre algorithme ordonnance un graphe de dépendance réduit par niveaux exactement comme le fait ALLEN-KENNEDY. Pour s'en rendre compte, il suffit de considérer la représentation des niveaux de dépendances par des vecteurs de direction (cf. section 1.2). Soit l le plus petit niveau d'un arc du GDRN G étudié. Une dépendance de niveau $l < \infty$ est équivalente au vecteur de direction $(\overbrace{0,\ldots,0}^{l-1},1+,\overbrace{*,\ldots,*}^{n-l})$. Les $n-l$ dernières composantes de tout hyperplan sont donc obligatoirement nulles et un circuit dont tous les arcs sont de niveaux strictement supérieurs à l est donc inclus dans G'. Nous effectuons donc les mêmes distributions de boucles que ALLEN-KENNEDY et marquons les mêmes boucles séquentielles. Par contre, toutes les boucles parallèles que nous générons sont internes au nid et ne peuvent englober des boucles séquentielles. Nous pouvons tout à fait adapter notre algorithme pour qu'il génère d'abord toutes les boucles parallèles possibles puis une boucle séquentielle, avant de s'appeler récursivement.

Nous pouvons donc considérer que notre algorithme généralise donc celui d'Allen et Kennedy.

Algorithme de Wolf et Lam

Pour se placer dans le modèle de cet algorithme, nous considérons des nids à une seule instruction. Dans ce cadre, nos équations se simplifient. Il nous faut bien évidemment comparer WOLF-LAM avec l'extension de ORDONNANCEUR_DE_GDRP qui génère des boucles totalement permutables. Les deux algorithmes ont exactement le même comportement : leurs ensembles de boucles totalement permutables respectifs ont même taille, en effet le polyèdre lié à notre algorithme et décrit par l'équation 4.5 est inclus dans le polyèdre Γ^+ (paragraphe 2.3.2) lié à WOLF-LAM, ce dernier polyèdre étant inclus dans le cône du polyèdre 4.5.

Notre algorithme généralise donc celui de Wolf et Lam.

Algorithme de Feautrier

Nous ne pouvons comparer les performances de notre algorithme et de celui de Feautrier que sur un même objet. C'est pourquoi nous nous référons ici à la version de l'algorithme de Feautrier ordonnançant des GDRP. Cet algorithme recherche de manière gloutonne des ordonnancements plus généraux que les nôtres. Le corollaire 6 nous assure que cet algorithme ne peut pas trouver plus de parallélisme que le nôtre, en ordre de grandeur. Par contre, il pourrait *a priori* satisfaire les arcs de dépendance différemment et trouver moins de parallélisme, en ordre de grandeur. En fait, il se comporte exactement comme notre algorithme en satisfaisant les mêmes arcs aux mêmes niveaux. Cet algorithme est donc lui aussi optimal pour la détection de parallélisme dans les graphes de dépendance réduits polyédriques. Par essence, l'algorithme de Feautrier utilise la connaissance des domaines d'itération. Nous n'avons pas supposé qu'il en était privé ici.

Théorème 9 (Optimalité pour les graphes de dépendance réduits polyédriques). *Soit un nid de boucles parfait dont les dépendances sont représentées par un GDRP G fortement connexe et dont le domaine d'itération contient (resp. est contenu dans) un hypercube de dimension n et de taille $\Omega(N)$ (resp. $O(N)$). Alors, l'ordonnancement de G proposé par* FEAUTRIER *est optimal : si d est le nombre d'appels récursifs à* FEAUTRIER *nécessaires pour complètement ordonnancer G (en comptant le premier appel), alors la latence de l'ordonnancement est en $O(N^d)$ et la longueur du plus long chemin en $\Omega(N^d)$.*

Plus précisément, si d_S est le nombre d'appels récursifs à FEAUTRIER *nécessaires pour complètement ordonnancer le nœud S de G (en comptant le premier appel), alors l'ordonnancement associé à S à une latence en $O(N^{d_S})$ et il existe dans le graphe de dépendance apparent un chemin qui contient $\Omega(N^{d_S})$ occurrences de S.*

DÉMONSTRATION DU THÉORÈME 9. Nous considérons un GDRP G_o fortement connexe dont nous connaissons le domaine $\mathcal{D}$. Nous allons montrer que la première composante de l'ordonnancement multidimensionnel produit par FEAUTRIER satisfait exactement les mêmes arcs polyédriques que l'ordonnancement produit par ORDONNANCEUR_DE_GDRP. Ceci prouvé, le résultat découlera immédiatement du corollaire 6.

L'algorithme de Paul Feautrier essaye de satisfaire le maximum d'arcs polyédriques à chaque étape de la récursion. Il recherche donc un ordonnancement affine qui, pour chaque arc e de G_o, satisfait l'équation :

$$\forall \boldsymbol{j} \in \mathcal{D},\ \forall \boldsymbol{d} \in \mathcal{P}_e,\ \boldsymbol{j} - \boldsymbol{d} \in \mathcal{D}\ \Rightarrow\ \Theta(S_j, \boldsymbol{j}) \geq \Theta(S_i, \boldsymbol{j} - \boldsymbol{d}) + \delta_e, \tag{4.8}$$

δ_e vérifiant toujours $\delta_e \geq 0$ et le plus souvent possible $\delta_e \geq 1$. Nous notons $\Theta(S_i, \boldsymbol{j}) = X_i\boldsymbol{j} + Y_i\boldsymbol{n} + \rho_i$, où $\boldsymbol{n}$ est le vecteur des paramètres de structure.

Par hypothèse, G_o est fortement connexe. Soit $\mathcal{C}$ un circuit de dépendance qui traverse tous ses sommets. Soit $q_1, \ldots, q_l$ un ensemble de l circuits élémentaires formant un multi-circuit de poids nul. Si nous ignorons le domaine, nous pouvons brancher q_i sur $\mathcal{C}$ autant de fois que nous le désirons, puisque $\mathcal{C}$ visite tous les sommets de G_o. Soit λ le plus grand entier tel que chacun des circuits q_i puisse être rattaché λ fois au circuit $\mathcal{C}$ et que l'ensemble forme un chemin de G_o dont il existe une instanciation qui demeure dans les limites du domaine $\mathcal{D}$. Par hypothèse, le domaine contient un hypercube de taille $f(N) = \Omega(N)$. Pour un arc de dépendance e, nous notons $H(e)$ la valeur absolue de son poids, c'est-à-dire $H(e) = |\boldsymbol{w}(e)|$. Nous étendons cette notion de valeur absolue aux chemins comme suit : si $p = e_1, \ldots, e_l$, $H(p) = \sum_{i=1}^{l} H(e_i)$. Avec ces notations, il est immédiat que si $H(\mathcal{C}) + \mu \sum_{i=1}^{l} H(q_i) \leq f(N)\mathbf{1}$, il existe dans le graphe de dépendance apparent de G_o *muni de son domaine* un chemin correspondant au circuit $\mathcal{C}$ sur lequel ont été branchées μ occurrences de chacun des circuits q_i. Il existe alors aussi un chemin correspondant au circuit $\mathcal{C}$ sur lequel ont été branchées μ occurrences d'un circuit q_j donné. Nous en déduisons une borne inférieure de λ :

$$\lambda \geq \min \left\{ \left\lfloor \frac{f(N)_j - H(\mathcal{C})_j}{\sum_{i=1}^{l} H(q_i)_j} \right\rfloor \middle| 1 \leq j \leq n \text{ et } \sum_{i=1}^{l} H(q_i)_j \neq 0 \right\} \tag{4.9}$$

Le graphe de dépendance apparent de G_o contient donc une dépendance de $S_j(\boldsymbol{j})$ vers $S_j(\boldsymbol{j} + \boldsymbol{w}(\mathcal{C}) + \lambda \boldsymbol{w}(q_i))$, pour au moins une valeur de $\boldsymbol{j}$. Le délai induit sur cette dépendance par la première composante de l'ordonnancement est forcément positif (δ_e est toujours positif dans l'équation 4.8). Nous en déduisons :

$$\forall i,\ X_j(\boldsymbol{j}) + Y_j \boldsymbol{n} + \rho_j \leq X_j(\boldsymbol{j} + \boldsymbol{w}(\mathcal{C}) + \lambda \boldsymbol{w}(q_i)) + Y_j \boldsymbol{n} + \rho_j \Leftrightarrow \forall i,\ 0 \leq X_j(\boldsymbol{w}(\mathcal{C}) + \lambda \boldsymbol{w}(q_i)).$$

Une somme de termes tous positifs étant supérieure à chacun de ses termes, nous en déduisons :

$$\forall i,\ 0 \ \leq\ X_j(\boldsymbol{w}(\mathcal{C}) + \lambda \boldsymbol{w}(q_i)) \ \leq\ \sum_{i=1}^{l} X_j(\boldsymbol{w}(\mathcal{C}) + \lambda \boldsymbol{w}(q_i)) \ \leq\ l X_j \boldsymbol{w}(\mathcal{C}) \Leftrightarrow$$

$$\forall i,\ -X_j(\boldsymbol{w}(\mathcal{C}) \ \leq\ \lambda \boldsymbol{w}(q_i)) \ \leq\ (l-1) X_j \boldsymbol{w}(\mathcal{C})$$

puisque $\sum_{i=1}^{l} X_j \lambda \boldsymbol{w}(q_i) = \lambda X_j \sum_{i=1}^{l} \boldsymbol{w}(q_i)$ et que la dernière somme est nulle, les circuits q_i formant un multi-circuit de poids nul. L'expression $l X_j \boldsymbol{w}(\mathcal{C})$ est indépendante de la taille du domaine, tout comme le vecteur nul. Nous obtenons ainsi un encadrement par des termes constants du produit $\lambda X_j \boldsymbol{w}(q_i)$ qui contient le facteur λ qui est minorée par une fonction croissante linéaire en la taille du domaine (équation 4.9). Par conséquent, $X_j \boldsymbol{w}(q_i) = 0$. Par conséquent, pour toute instruction S_j, la partie linéaire X_j de son ordonnancement est orthogonale au poids de tout circuit élémentaire participant à un multi-circuit de poids nul.

Soit q un circuit élémentaire d'arcs de dépendance qui participe à un multi-circuit de poids nul. Nous décomposons q en arcs de dépendance : $q = e_1, \ldots, e_k$. En supposant que le domaine est suffisamment grand pour contenir une instanciation de q, nous obtenons par instanciation de l'équation 4.8 :

$$X_{t(e_i)} \left(\boldsymbol{j} + \sum_{j=1}^{i} \boldsymbol{w}(e_j) \right) + \rho_{t(e_i)} \geq X_{q(e_i)} \left(\boldsymbol{j} + \sum_{j=1}^{i-1} \boldsymbol{w}(e_j) \right) + \rho_{q(e_i)} + \delta_{e_i}$$

où δ_{e_i} est la variable qui indique si l'arc polyédrique, dont e_i est une instanciation, est ou non satisfait par l'ordonnancement. En sommant ces équations pour i variant de 1 à k, nous aboutissons à l'inégalité :

$$\sum_{i=1}^{k} \delta_{e_i} \leq X_{t(e_k)} \sum_{i=1}^{k} \boldsymbol{w}(e_i).$$

Le membre gauche de l'inégalité est positif par définition. Quant au membre droit, il est nul vu ce qui a été prouvé ci-dessus : c'est le produit scalaire de la partie linéaire d'un ordonnancement et du poids d'un circuit élémentaire participant à un multi-circuit de poids nul. Par conséquent, pour tout i $\delta_{e_i} = 0$ et aucun des arcs polyédriques visités par ce circuit n'est satisfait. Ces arcs polyédriques sont exactement ceux qui ne sont pas satisfaits par ORDONNANCEUR_DE_GDRP : si un arc polyédrique est visité par un circuit élémentaire participant à un multi-circuit de poids nul, alors au moins un des arcs de G_u correspondant à l'un des sommets de son polyèdre participe à un multi-circuit de poids nul (pour passer des circuits du graphe originel au graphe uniformisé, voir le lemme 15). Par conséquent, la première composante de l'ordonnancement produit par FEAUTRIER ne satisfait aucun des arcs polyédriques qui ne sont pas satisfaits par la première composante des ordonnancements produits par ORDONNANCEUR_DE_GDRP. Réciproquement, tous les arcs polyédriques satisfaits par ORDONNANCEUR_DE_GDRP sont satisfiables et FEAUTRIER les satisfaits puisqu'il cherche à satisfaire le plus d'arcs possibles à l'aide d'ordonnancements plus généraux que ceux manipulés par ORDONNANCEUR_DE_GDRP. Le corollaire 6 nous permet de conclure. □

L'algorithme de Feautrier généralise donc notre algorithme.

Avons-nous rempli nos objectifs ?

Il est clair que nous avons rempli nos objectifs relatifs aux algorithmes d'Allen et Kennedy et de Wolf et Lam : nous généralisons ces deux algorithmes, nous utilisons une représentation des dépendances plus générale et nous en extrayons tout le parallélisme existant, en ordre de grandeur.

Notre position vis-à-vis de l'algorithme de Feautrier est moins nette. Certes nous générons des ordonnancements affines plus simples, et même les plus simples qui permettent de découvrir tout le parallélisme, mais Feautrier pourrait contraindre son algorithme à ne rechercher que ce type d'ordonnancements. Pour ce qui est de la complexité des deux ordonnancements, nous nous référons au développement des deux algorithmes qui a été mené à l'Université de Passau par l'équipe de Christian Lengauer au sein du paralléliseur « LooPo » [60]. Les deux algorithmes ont été comparés sur quatre exemples [63]. À chaque fois, l'algorithme de Feautrier a utilisé les dépendances exactes et le nôtre une approximation par des polyèdres. Deux des nids étaient non parfaits [5]. Notre algorithme a toujours été plus rapide, le gain relatif en temps croissant avec le nombre de dépendances du programme étudié. Cependant, la vitesse d'exécution d'un paralléliseur n'est pas une fin en soit. Ici elle indique que ORDONNANCEUR_DE_GDRP résout des systèmes linéaires beaucoup plus petits que FEAUTRIER. Sur le dernier exemple, non parfait, qui contenait trente cinq dépendances, notre algorithme a trouvé un ordonnancement en moins de deux secondes alors qu'il a fallu plus de quarante huit minutes pour celui de Feautrier. Il est donc à craindre que les systèmes que cherche à satisfaire FEAUTRIER atteignent rapidement des tailles qui en interdisent la résolution. Nous pouvions nous attendre à la différence de complexité des systèmes linéaires résolus par les deux algorithmes. Il est beaucoup plus surprenant que sur ces quatre exemples les deux algorithmes aient à chaque fois trouvé le même ordonnancement. Ce qui nous invite à poser la question iconoclaste : « avons-nous besoin d'ordonnancements aussi compliqués que ceux générés par l'algorithme de Paul Feautrier ? ». En théorie, la réponse est « oui », mais en pratique la question est ouverte. Pour pouvoir y répondre il faudrait savoir à quoi ressemble un programme dans la « réalité ».

L'algorithme de Feautrier n'est pas optimal dans le cas général. Il serait intéressant de savoir pour quelles classes de programmes il l'est. Le théorème 9 fourni un élément de réponse d'autant

5. Le rapport en question ne précise pas exactement les conditions de l'expérimentation. Il décrit une méthode d'extension aux nids non parfaits qui produit un résultat identique à celui de l'algorithme ORDONNANCEUR_DE_GDRP_NP.

plus important qu'il n'existait jusqu'à présent aucun résultat sur le comportement de l'algorithme dans le cas multidimensionnel. Un des points forts de l'approche de Feautrier est sa capacité à ne pas faire de différence entre nids parfaits et non parfaits.

Conclusion

L'étude théorique des graphes de dépendance réduits polyédriques nous a permis de produire un algorithme d'ordonnancement optimal de ces graphes qui généralise simultanément les algorithmes d'Allen et Kennedy et de Wolf et Lam. Elle nous a également permis de découvrir que l'algorithme proposé par Paul Feautrier possédait déjà ces qualités. L'algorithme de Feautrier est plus fin que le nôtre, puisqu'il sait tirer parti de la connaissance du domaine des instructions et des dépendances, et il est plus général, puisqu'il peut aussi bien fonctionner avec une représentation exacte des dépendances qu'avec une représentation approchée, voire travailler simultanément sur les deux types de représentations. Le lecteur pourrait donc être tenté de penser que notre algorithme et la présente étude sont sans intérêts. Cette conclusion nous semble hâtive.

Avant tout, l'étude théorique nous a permis de comprendre les limitations potentielles de la parallélisation des nids de boucles dont les dépendances sont représentées par un graphe de dépendance réduit polyédrique, en mettant en évidence les schémas de dépendance responsables de la séquentialité intrinsèque des GDRP. Nous avons vu que les circuits de dépendance et leurs valeurs importaient beaucoup plus que les arcs de dépendances, contrairement à ce que laisse penser l'approche de Wolf et Lam. Il s'est aussi avéré que nos graphes s'ordonnançaient naturellement à l'aide d'ordonnancements linéaires décalés et non à l'aide d'ordonnancements affines généraux.

Les propriétés de nos ordonnancements nous indiquent des extensions possibles de l'algorithme. En particulier, nous disposons d'une certaine liberté de choix pour nos ordonnancements : nous connaissons toutes les parties linéaires possibles. Cette connaissance nous a permis de passer de la parallélisation à grain fin à celle à grain moyen en générant des boucles totalement permutables. Elle va maintenant nous permettre de coupler fortement notre recherche d'ordonnancements avec celle de placements des calculs (chapitre 5).

Chapitre 5

Ordonnancement en présence de placement des calculs

> Alors dramatique. Problème insoluble. Dramatique. Les deux belles-mères veulent figurer sur le carton d'invitation !
>
> Yvan *in* « Art », Yasmina **Reza**

Dans le cadre de la parallélisation, nous devons spécifier non seulement la date d'exécution des différentes opérations (ordonnancement) mais aussi leur lieu (placement des calculs). Le placement des calculs associe à chaque opération du programme le processeur (virtuel) qui doit l'évaluer. Les chercheurs qui ont étudié le problème du placement ont dans leur majorité supposé que les fonctions de placement étaient affines [69, 5, 28, 41, 9, 29, 35, 34]. Dans ce cadre, les processeurs virtuels forment un grille de dimension δ. Cette hypothèse se justifie totalement pour des machines dont la topologie est une grille, ce qui devient rarissime, ou lorsque la parallélisation a pour langage cible un langage tel que HPF qui ne connaît pas d'autres topologies que la grille [43]. En fait, cette hypothèse se justifie surtout par la simplicité du modèle : nous ne sommes capables d'étudier que des fonctions simples et, comme l'écrit Paul Feautrier [42], « ce n'est que pour le cas des fonctions linéaires qu'il a été possible de trouver des algorithmes de placement automatiques ». Les grilles ont aussi l'avantage de pouvoir être placées sur d'autres structures [71, 64], mais ce modèle n'en est pas moins éloigné de la réalité. Cependant, faute de disposer de placements plus « réalistes », nous supposerons que le problème de placement des calculs a été résolu au moyen d'un algorithme ou d'une heuristique produisant des fonctions affines.

Nous supposons que toutes les instructions ont un placement de leurs calculs, c'est-à-dire de leurs instances, de même dimension δ. Le placement des calculs d'une instruction S est une projection de l'espace d'itération $\mathcal{D}_S$ vers la grille des processeurs virtuels de dimension δ. Nous supposons également que le degré de parallélisme désiré par le paralléliseur est exactement égal à la dimension du placement fourni. Nous nous devons donc de trouver des ordonnancements pour lesquels les δ dimensions du placement sont effectivement parallèles : à tout instant défini par l'ordonnancement (en régime permanent), un ensemble d'opérations peut être exécuté en parallèle et nous voulons que la projection de cet ensemble par le placement soit effectivement un ensemble de processeurs de dimension δ. Un ordonnancement vérifiant cette propriété sera dit *compatible* avec le placement. Nous illustrons le problème de compatibilité au travers de l'exemple 19 (figure 5.1).

Exemple de motivation

Exemple 19

```
For i = 1 to n do
  For j = 1 to n do
    S1 : a(i, j) = b(i-1, 1)
    S2 : b(i, j) = a(i-1, 1)
```

$$1 \le i \le n-1, 1 \le j \le n, \quad S_1(i,1) \xrightarrow{\text{flot}} S_2(i+1,j)$$
$$1 \le i \le n-1, 1 \le j \le n, \quad S_2(i,1) \xrightarrow{\text{flot}} S_1(i+1,j)$$

FIG. 5.1: *Exemple 19 et ses relations de dépendance.*

Les dépendances du code de l'exemple 19 sont présentées à la figure 5.1. Par transitivité, S_1(i+2, 1) dépend de S_1(i, 1), pour i $\in$ [1, n-2], et la première composante de la partie linéaire de tout ordonnancement affine de S_1 doit être supérieure à 1. Il en va de même pour S_2. Par ailleurs ordonnancer les itérations i de S_1 et S_2 à la date i respecte les deux dépendances. Par conséquent, un tel ordonnancement est de latence minimale.

Considérons maintenant le problème du placement en cherchant à minimiser les communications. Si nous voulons que la dépendance de S_1 vers S_2 ne génère pas de communications, pour i fixé appartenant à [1, n-1] nous devons placer toutes les opérations S_2(i,j) sur le même processeur que S_1(i,1). La partie linéaire du placement de S_2 doit donc être colinéaire au vecteur (1, 0). Il en va de même pour le placement de S_1 si nous ne voulons pas que la dépendance de S_2 vers S_1 engendre de communications. Dans cet exemple, il n'est pas possible d'annuler l'ensemble des communications sans placer tous les calculs sur le même processeur. Il est par contre possible d'annuler toutes les communications dues à une des dépendances tout en rendant locales les autres communications et tout en conservant un degré de parallélisme. Dans ce cas les parties linéaires des deux placements sont égales et colinéaires à (1, 0). Par contre, si aucune de ces parties linéaires n'est colinéaire à (1, 0), pour chaque valeur de i dans [1,n-1] la dépendance de S_1 vers S_2 (resp. de S_2 vers S_1) génère une communication pour au moins n-1 valeurs de j.

Par conséquent, l'ordonnancement de latence minimale et le placement qui minimise les communications sont incompatibles dans notre exemple : les opérations S_1(i, j) et S_2(i, j), pour j $\in$ [1,n], doivent être exécutées à la date i or, à i fixé, toutes les opérations S_1(i, j) (resp. S_2(i, j)) doivent être exécutées sur le même processeur puisque les placements sont parallèles à la direction (1, 0). La solution obtenue ne contient donc aucun degré de parallélisme. Il n'y a pas de raisons *a priori* pour qu'il existe un ordonnancement compatible avec un placement des calculs donné, et il y en a encore moins pour qu'un ordonnancement et un placement choisis indépendamment soient compatibles. Par contre, étant donné un ordonnancement, il existe toujours un placement compatible[1]. Mais la recherche d'un placement une fois un ordonnancement donné contraint le choix du placement : dans notre exemple un placement compatible avec l'ordonnancement de latence minimale engendre au moins $(\mathsf{n}-1)^2$ communications (distantes). La recherche d'un ordonnancement une fois un placement donné permet de garantir des propriétés sur le placement : ici il existe des ordonnancements compatibles avec le meilleur placement, l'ordonnancement linéaire défini par le vecteur (1, 1) en est un. L'idée est de considérer le « meilleur » placement, d'essayer de construire un ordonnancement compatible et de recommencer en cas d'échec avec un placement de moindre qualité. Nous nous intéressons à la recherche d'ordonnancements compatibles avec un placement donné, principale-

1. Dans notre cadre de transformations affines, il suffit de compléter l'ensemble E_S des parties linéaires des composantes de l'ordonnancement d'une instruction S en une base B_S de l'espace et de tirer au hasard δ vecteurs de $B_S \setminus E_S$ pour obtenir un placement de S compatible.

ment parce qu'il s'agit d'un problème d'ordonnancement, en espérant pouvoir tirer parti de notre connaissance de tous les ordonnancements *constructibles* par notre algorithme.

L'ordonnancement E_S et le placement C_S d'une instruction S sont compatibles si les vecteurs lignes de leurs parties linéaires forment une famille libre de $\mathbb{Q}^n$. Ce chapitre est en fait une étude des implications de cette propriété pour nos ordonnancements particuliers.

Nous nous intéressons à des nids de boucles parfaits de profondeur n dont les dépendances sont représentées par un GDRP. Nous supposons qu'un placement C_S de rang et de dimension δ nous a été donné pour chaque instruction S du nid. Nous cherchons un ordonnancement *constructible* par ORDONNANCEUR_DE_GDRP et compatible avec ces placements.

Le placement affine C_S des instances de l'instruction S est noté :

$$\begin{array}{rccl} C_S: & \mathcal{D} & \longrightarrow & \mathbb{P} \\ & \boldsymbol{i} & \longmapsto & C_S(\boldsymbol{i}) = \mathbf{C}_S\,\boldsymbol{i} + \boldsymbol{c}_S \end{array}$$

où $\mathbb{P}$ est la grille virtuelle de processeurs de dimension δ. $\mathbf{C}_S$ est une matrice de taille $\delta \times n$, de rang δ et de vecteurs-lignes $\mathbf{C}_S^1, \dots, \mathbf{C}_S^\delta$. $\boldsymbol{c}_S$ est un vecteur de taille δ. L'ordonnancement associé à l'instruction S est noté quant à lui :

$$\begin{array}{rccl} E_S: & \mathcal{D} & \longrightarrow & \mathbb{T} \\ & \boldsymbol{i} & \longmapsto & E_S(\boldsymbol{i}) = \mathbf{E}_S\,\boldsymbol{i} + \boldsymbol{e}_S \end{array}$$

où $\mathbb{T}$ est un espace de dimension $n - \delta$. La matrice $\mathbf{E}_S$ est de taille $(n - \delta) \times n$ et de vecteurs-lignes $\mathbf{E}_S^1, \dots, \mathbf{E}_S^{d_S}$. Le vecteur $\boldsymbol{e}_S$ est de taille $(n - \delta)$. L'ordonnancement E_S et le placement C_S sont compatibles si et seulement si :

$$\operatorname{rang}\begin{pmatrix} \mathbf{E}_S \\ \mathbf{C}_S \end{pmatrix} = n. \tag{5.1}$$

À la section 5.1, nous montrons qu'au lieu de rechercher directement les parties linéaires des ordonnancements, c'est-à-dire des points des polyèdres $\mathsf{P}(G_u(S,i))$ (avec les notations définies au paragraphe 4.3.3), nous pouvons nous contenter de chercher des points des espaces vectoriels $\mathsf{E}(G_u(S,i))$. Nous montrons qu'une fois connus des points des espaces vectoriels $\mathsf{E}(G_u(S,i))$ satisfaisant les bonnes conditions, nous savons construire un ordonnancement compatible à partir de ces points. À la section 5.2, nous démontrons une condition nécessaire et suffisante d'existence des points des espaces vectoriels $\mathsf{E}(G_u(S,i))$ qui « satisfont les bonnes conditions» et donc d'existence d'un ordonnancement constructible compatible avec un placement donné. Nous montrons également comment construire les « bons » points des espaces. Nous illustrons notre méthode sur un exemple à la section 5.3. À la section 5.4, nous montrons que notre condition nécessaire et suffisante d'existence d'un ordonnancement *constructible* par ORDONNANCEUR_DE_GDRP et compatible est en fait une condition nécessaire et suffisante d'existence d'un ordonnancement affine *quelconque* compatible, c'est-à-dire *constructible* ou non par ORDONNANCEUR_DE_GDRP.

Nous reprenons dans ce chapitre les notations définies au paragraphe 4.3.3.

5.1 Réduction de l'espace de recherche

D'après l'équation 5.1 et les résultats établis au paragraphe 4.3.3, un ordonnancement *constructible* compatible est exactement défini par la donnée d'un vecteur $\mathbf{E}_S^i$ de $\mathsf{P}(G_u(S,i))$ pour chacun des graphes $G_u(S,i)$ à condition que la famille $\{\mathbf{E}_S^1, \dots, \mathbf{E}_S^{d_S}, \mathbf{C}_S^1, \dots, \mathbf{C}_S^\delta\}$ soit libre et que $\mathbf{E}_S^i = \mathbf{E}_T^i$, pour tout nœud T de $G_u(S,i)$ et tout $i \in [1, d_S]$.

$\mathsf{E}(G_u(S,i))$ est l'espace vectoriel engendré par $\mathsf{P}(G_u(S,i))$. Nous montrons dans cette section que si nous ne connaissons qu'un ensemble de vecteurs des espaces vectoriels $\mathsf{E}(G_u(S,i))$, et non un ensemble de vecteurs des polyèdres $\mathsf{P}(G_u(S,i))$, qui satisfait ces conditions alors nous savons quand même construire un ordonnancement compatible. Nous supposons donc dans cette section que pour toute instruction S et tout entier i de $[1, d_S]$ nous avons un vecteur $\mathbf{E}_S^i$ de $\mathsf{E}(G_u(S,i))$ tel que la famille $\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{d_S}, \mathbf{C}_S^1, \ldots, \mathbf{C}_S^{\delta}\}$ soit libre et que $\mathbf{E}_S^i = \mathbf{E}_T^i$, pour tout nœud T de $G_u(S,i)$. Nous supposons également que, pour tout ensemble $G_u(S,i)$, nous connaissons un hyperplan $p(G_u(S,i))$ qui satisfait tous les arcs satisfiables de $G_u(S,i)$, même ceux appartenant à $(G_u(S,i))''$. $p(G_u(S,i))$ est donc un point du polyèdre $\mathsf{P}(G_u(S,i))$ qui vérifie le système :

$$\left\{ \begin{array}{lcl} e = (q(e), t(e)) \in (G_u(S,i))' & \Rightarrow & p(G_u(S,i))\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 0 \\ e = (q(e), t(e)) \notin (G_u(S,i))' & \Rightarrow & p(G_u(S,i))\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 1 \end{array} \right. \tag{5.2}$$

Pour chaque instruction S, nous complétons la famille libre $\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{d_S}, \mathbf{C}_S^1, \ldots, \mathbf{C}_S^{\delta}\}$ avec $n - \delta - d_S$ vecteurs $\mathbf{L}_S^1, \ldots, \mathbf{L}_S^{n-\delta-d_S}$ pour obtenir une famille libre de dimension n. Nous formons la matrice :

$$\mathbf{P}_{S,0} = \begin{pmatrix} \mathbf{E}_S \\ \mathbf{L}_S \\ \mathbf{C}_S \end{pmatrix},$$

où $\mathbf{E}_S$ (resp. $\mathbf{L}_S$) est la matrice dont le $i^{\grave{e}me}$ vecteur-ligne est le vecteur $\mathbf{E}_S^i$ (resp. $\mathbf{L}_S^i$).

Considérons un graphe $G_u(S,i)$. Nous voulons que toutes les instructions représentées dans ce graphe aient la même $i^{\grave{e}me}$ partie linéaire et que cette partie linéaire soit un point de $\mathsf{P}(G_u(S,i))$. Pour cela nous allons ajouter à la $i^{\grave{e}me}$ ligne des matrices P_T, pour tout nœud T de $G_u(S,i)$, un même nombre de fois le point $p(G_u(S,i))$ du polyèdre $\mathsf{P}(G_u(S,i))$.

Algorithme de construction des parties linéaires

Pour $i = 1$ à $\max_{S \in G_u} d_S$ faire :

– Pour chaque composante fortement connexe $G_u(S,i)$ de G_i faire :

1. Pour tout nœud T de $G_u(S,i)$ nous définissons la matrice $\mathbf{P}'_{T,i-1}$ comme la matrice $\mathbf{P}_{T,i-1}$ à laquelle nous avons rajouté le vecteur $p(G_u(S,i))$ sur la $i^{\grave{e}me}$ ligne. Soient $\mu_T = \det(\mathbf{P}_{T,i-1})$ et $\mu'_T = \det(\mathbf{P}'_{T,i-1})$.
2. Calculer l'ensemble :
$$\mu = \left\{ \frac{-\mu_T}{\mu'_T - \mu_T} \;\middle|\; \mu_T \neq \mu'_T \right\}$$
Chercher un rationnel ν tel qu'il existe des constantes ρ_S qui vérifient :
$$\left\{ \begin{array}{lcl} e \in (G_u(S,i))'' \text{ ou } q(e) \text{ est virtuel} & \Rightarrow & (\mathbf{E}_S^i + \nu\, p(G_u(S,i)))\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 0 \\ e \notin (G_u(S,i))'' \text{ et } q(e) \text{ est réel} & \Rightarrow & (\mathbf{E}_S^i + \nu\, p(G_u(S,i)))\boldsymbol{w}(e) + \rho_{t(e)} - \rho_{q(e)} \geq 1 \end{array} \right.$$
3. $\mathbf{P}_{T,i}$ est égale à la matrice $\mathbf{P}_{T,i-1}$ à la $i^{\grave{e}me}$ ligne de laquelle a été rajouté λ fois le vecteur $p(G_u(S,i))$, où $\lambda \geq \nu$ et $\lambda \notin \mu$.

Pour chaque instruction S, les d_S premières lignes de la matrice $\mathbf{P}_{S,d_S}$ définissent les d_S parties linéaires de son ordonnancement.

Correction de l'algorithme

Pour prouver la correction de l'algorithme, nous devons montrer que μ et ν sont bien définis, que, pour tout $i \in [1, d_T]$, la matrice $\mathbf{P}_{T,i}$ est inversible et que sa $i^{\grave{e}me}$ ligne est bien un point de $\mathsf{P}(G_u(S,i))$. Nous montrons que l'avant dernière propriété est garantie par $\lambda \notin \mu$ et que la dernière est assurée par $\lambda \geq \nu$.

Pour montrer que ν existe nous n'utilisons pas la formulation de l'algorithme, qui explicite un mode de calcul, mais la formulation équivalente proposée par le lemme 18 : pour tout circuit élémentaire $\mathcal{C}$ de $G_u(S,i)$, $(\mathbf{E}_S^i + \nu\, p(G_u(S,i)))\boldsymbol{w}(\mathcal{C}) \geq l(\mathcal{C})$, où $l(C)$ est le nombre d'arcs du circuit C dont la queue est un nœud réel et qui n'appartiennent pas à $(G_u(S,i))''$. Soit un circuit C de $G_u(S,i)$. Si C contient un arc n'appartenant pas à $(G_u(S,i))'$ l'hyperplan $p(G_u(S,i))$, qui est par hypothèse solution du système (5.2), induit un délai strictement positif sur C. Tout rationnel ν supérieur à $(l(\mathcal{C}) - \mathbf{E}_S^i\boldsymbol{w}(\mathcal{C}))/(p(G_u(S,i))\boldsymbol{w}(\mathcal{C}))$ vérifie alors l'équation : $(\mathbf{E}_S^i + \nu\, p(G_u(S,i)))\boldsymbol{w}(\mathcal{C}) \geq l(\mathcal{C})$. Sinon, C est inclus dans $(G_u(S,i))'$. C est alors inclus dans $(G_u(S,i))''$ et $l(C)$ est nul. Par ailleurs, comme le vecteur $\mathbf{E}_S^i$ appartient à $\mathsf{E}(G_u(S,i))$, il induit un délai nul sur tout les circuits de $(G_u(S,i))'$ et donc sur C. $p(G_u(S,i))$ est un point de $\mathsf{P}(G_u(S,i))$ et donc de $\mathsf{E}(G_u(S,i))$, il induit lui aussi un délai nul sur C. L'équation $(\mathbf{E}_S^i + \nu\, p(G_u(S,i)))\boldsymbol{w}(\mathcal{C}) \geq l(\mathcal{C})$ est alors vérifiée par tout rationnel ν. Un graphe ne contenant qu'un nombre fini de circuits élémentaires, il existe bien un rationnel ν tel que l'algorithme en recherche un.

Nous devons prouver qu'en modifiant la $i^{\grave{e}me}$ ligne de la matrice nous ne l'avons pas rendu non inversible. Initialement $\mathbf{P}_{T,0}$ est inversible. Nous montrons que s'il en va de même de $\mathbf{P}_{T,i-1}$, alors $\mathbf{P}_{T,i}$ est aussi inversible.

Considérons la matrice paramétrée $\mathbf{Q}_{T,i-1}(\alpha)$ définie comme étant égale à la matrice $\mathbf{P}_{T,i-1}$ à laquelle nous avons rajouté α fois le vecteur $p(G_u(S,i))$ sur la $i^{\grave{e}me}$ ligne. Nous avons donc $\mathbf{Q}_{T,i-1}(0) = \mathbf{P}_{T,i-1}$ et $\mathbf{Q}_{T,i-1}(1) = \mathbf{P}'_{T,i-1}$. Le déterminant de la matrice $\mathbf{Q}_{T,i-1}(\alpha)$ est un polynôme en α de degré au plus un : $\det(\mathbf{Q}_{T,i-1}(\alpha)) = [\det(\mathbf{Q}_{T,i-1}(1)) - \det(\mathbf{Q}_{T,i-1}(0))]\alpha + \det(\mathbf{Q}_{T,i-1}(0)) = (\mu'_T - \mu_T)\alpha + \mu_T$. Ce polynôme est non identiquement nul puisque $\mathbf{Q}_{T,i-1}(0) = \mathbf{P}_{T,i-1}$ est inversible par hypothèse. Ce polynôme s'annule donc en au plus une valeur. Si $\mu_T \neq \mu'_T$, il s'annule en $-\mu_T/(\mu'_T - \mu_T)$, et il ne s'annule pas sinon. Les graphes $G_u(S,i)$ ayant un nombre fini de nœuds, μ est bien défini. Vu ce qui précède, pour tout rationnel $\alpha \notin \mu$, $\mathbf{Q}_{T,i-1}(\alpha)$ est inversible pour tout nœud T de $G_u(S,i)$. En particulier, $\mathbf{P}_{T,i}$ est inversible.

5.2 Construction de parties linéaires dans $\mathsf{E}(G_u(S,i))$

Nous avons vu à la section précédente qu'il nous suffisait de connaître, pour tout nœud S de G_u et tout entier i de $[1, d_S]$, un vecteur non nul $\mathbf{E}_S^i$ de $\mathsf{E}(G_u(S,i))$ pour pouvoir construire un ordonnancement compatible avec le placement, à la condition que la famille $\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{d_S}, \mathbf{C}_S^1, \ldots, \mathbf{C}_S^\delta\}$ soit libre et que $\mathbf{E}_S^i = \mathbf{E}_T^i$, pour tout nœud T de $G_u(S,i)$. Nous nous intéressons ici à la construction des vecteurs $\mathbf{E}_S^i$. Le lemme 20 nous donne une condition nécessaire et suffisante à l'existence des ensembles de vecteurs recherchés. Vu l'algorithme de la section précédente, ce lemme nous donne une condition nécessaire et suffisante à l'existence d'un ordonnancement constructible compatible. Nous verrons à la fin de cette section comment construire les vecteurs nécessaires.

Lemme 20. *$\mathsf{C}(S)$ désigne l'espace vectoriel engendré par la famille de vecteurs $\{\mathbf{C}_S^1, \ldots, \mathbf{C}_S^\delta\}$.*

Pour toute instruction S, il existe une suite de vecteurs $\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{d_S}$ telle que $\mathbf{E}_S^i \in \mathsf{E}(G_u(S,i))$, telle que toutes les instructions T de $G_u(S,i)$ aient le même $i^{\grave{e}me}$ vecteur $\mathbf{E}_S^i$ et telle que la famille $\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{d_S}, \mathbf{C}_S^1, \ldots, \mathbf{C}_S^\delta\}$ soit libre si et seulement si pour toute instruction S et tout entier i de

$[1, d_S]$, *nous avons :*

$$i + \dim(\mathsf{E}(G_u(S,i)) \cap \mathsf{C}(S)) \leq \dim(\mathsf{E}(G_u(S,i))), \tag{5.3}$$

Pour démontrer ce lemme nous avons besoin du résultat suivant :

Lemme 21. *Soit un ensemble de sous-espaces vectoriels E, $F_1, \ldots, F_m$ de $\mathbb{Q}^n$. Nous supposons que, pour tout $i \in [1, m]$, il existe un sous-espace vectoriel V_i de E, de dimension $d \geq 1$, et en somme directe avec F_i : $F_i \oplus V_i$. Alors, il existe un sous-espace vectoriel V de E, de dimension d, et en somme directe avec tous les espaces F_i.*

DÉMONSTRATION. La démonstration se fait par récurrence sur la valeur de d. Nous commençons par montrer qu'il existe un sous-espace vectoriel de dimension un en somme directe avec tous les espaces F_i, si $d \geq 1$. Cela revient à prouver qu'il existe un élément x de E qui n'appartient à aucun des espaces F_i. Raisonnons par l'absurde et supposons qu'il n'existe pas un tel élément x de E. Alors $E \subset \cup_{i=1}^m F_i$ et $E = \cup_{i=1}^m (F_i \cap E)$. Donc, comme un espace vectoriel n'est jamais la réunion finie d'un ensemble de ses sous-espaces vectoriels stricts, il existe i tel que $E \subset F_i$, ce qui est impossible car il existe par hypothèse un sous-espace vectoriel V_i de E de dimension non nulle et en somme directe avec F_i. Il existe donc un élément x de E qui n'appartient à aucun des espaces F_i.

Nous avons montré le résultat cherché pour $d = 1$. Supposons que nous l'avons démontré jusqu'au rang $j \geq 1$. Soit un ensemble de sous-espaces vectoriels E, $F_1, \ldots, F_m$ de $\mathbb{Q}^n$ tels que, pour tout $i \in [1, m]$, il existe un sous-espace vectoriel V_i de E, de dimension $j + 1$, et en somme directe avec F_i. En utilisant ce que nous venons de démontrer, nous prenons un élément x de E qui n'appartient à aucun des espaces F_i. Nous considérons les espaces $G_i = F_i \oplus \mathrm{Vect}(x)$ où $\mathrm{Vect}(x)$ désigne l'espace vectoriel engendré par x. Il existe un sous-espace vectoriel W_i de E en somme directe avec G_i et de dimension j. Pour le construire, il suffit de prendre une base B_i de F_i et de compléter $B_i \cup \{x\}$, qui est une base de $F_i \oplus \mathrm{Vect}(x)$, en une base B'_i de $F_i + \mathrm{Vect}(x) + V_i$. $F_i + \mathrm{Vect}(x) + V_i$ contient $F_i \oplus V_i$ qui est de dimension $\dim(F_i) + j + 1$. Donc, $B'_i \setminus (B_i \cup \{x\})$ contient j ou $j + 1$ vecteurs. Les vecteurs de $B'_i \setminus (B_i \cup \{x\})$ ont été choisis en complément d'une base de $F_i \oplus \mathrm{Vect}(x)$, ils engendrent donc un espace vectoriel W_i supplémentaire à $G_i = F_i \oplus \mathrm{Vect}(x)$.

Nous avons donc maintenant un espace vectoriel E et des espaces G_i, tels que, pour tout $i \in [1, m]$, il existe un sous-espace vectoriel W_i de E, de dimension j, en somme directe avec G_i. Nous appliquons à ces espaces l'hypothèse de récurrence pour obtenir un sous-espace W de E en somme directe avec chacun des $G_i = F_i \oplus \mathrm{Vect}(x)$. $W \oplus \mathrm{Vect}(x)$ est alors le sous-espace de E recherché. □

DÉMONSTRATION DU LEMME 20.
La condition est suffisante.

Nous supposons que l'inégalité (5.3) est satisfaite.

Nous construisons les vecteurs $\mathbf{E}_T^j$ de manière gloutonne. Soit un graphe $G_u(S, i)$. Nous supposons que, pour $j \in [1, i-1]$, les vecteurs $\mathbf{E}_S^j$ ont déjà été sélectionnés, qu'ils forment une famille libre, que $\mathbf{E}_S^j$ appartient à $\mathsf{E}(G_u(S, j))$, que, pour tout nœud T de $G_u(S, j)$, $\mathsf{C}(T)$ et $\mathrm{Vect}\{\mathbf{E}_T^1, \ldots, \mathbf{E}_T^j\}$ sont en somme directe et que $\mathbf{E}_T^j$ est égal à $\mathbf{E}_S^j$.

Il existe un sous-espace vectoriel V_T de $\mathsf{E}(G_u(S, i))$, de dimension au moins :

$$\dim(\mathsf{E}(G_u(S,i))) - \dim(\mathsf{E}(G_u(S,i)) \cap \mathsf{C}(T)) - (i-1),$$

en somme directe avec $\text{Vect}\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}\} \oplus \mathsf{C}(T)$: il suffit de prendre un espace supplémentaire de $[\text{Vect}\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}\} \oplus \mathsf{C}(T)] \cap \mathsf{E}(G_u(S,i))$ dans $\mathsf{E}(G_u(S,i))$. Le lemme 21 nous assure alors de l'existence d'un sous-espace vectoriel $H(G_u(S,i))$ de $\mathsf{E}(G_u(S,i))$, de dimension :

$$m(G_u(S,i)) = \dim(\mathsf{E}(G_u(S,i))) - \max\{\dim(\mathsf{E}(G_u(S,i)) \cap \mathsf{C}(T)) \mid T \text{ nœud de } G_u(S,i)\} - (i-1),$$

en somme directe avec les espaces $\text{Vect}\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}\} \oplus \mathsf{C}(T)$ pour tout nœud T de $G_u(S,i)$. Donc, $m(G_u(S,i)) \geq i - (i-1) = 1$ à cause de l'inégalité (5.3). $H(G_u(S,i))$ est donc de dimension non nulle et nous en prenons au hasard un élément x non nul. x est un élément de $\mathsf{E}(G_u(S,i))$ tel que : $\text{Vect}(x) \oplus \text{Vect}\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}\} \oplus \mathsf{C}(T)$, pour tout nœud T de $G_u(S,i)$. Par conséquent, la famille $\{x, \mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}\}$ est libre et engendre un sous-espace vectoriel de $\mathsf{E}(G_u(S,i))$ en somme directe avec $\mathsf{C}(T)$ pour tout nœud T de $G_u(S,i)$. Nous pouvons poser $\mathbf{E}_T^i = x$ pour tout nœud T de $G_u(S,i)$.

La condition est nécessaire.

Nous supposons qu'il existe des vecteurs $\mathbf{E}_S^i$ qui vérifient les hypothèses énoncées par le lemme. Considérons un graphe $G_u(S,i)$. Par hypothèse, la famille $\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^i, \mathbf{C}_S^1, \ldots, \mathbf{C}_S^\delta\}$ est libre. Pour tout $j \in [1,i]$, $\mathbf{E}_S^j$ est un point de $\mathsf{E}(G_u(S,j))$ qui est un sous-espace de $\mathsf{E}(G_u(S,i))$ (cf. le lemme 18). La famille $\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^i\}$ est libre et engendre donc un sous-espace vectoriel de $\mathsf{E}(G_u(S,i))$ de dimension i. Ce sous-espace vectoriel est de plus en somme directe avec $\mathsf{C}(S)$, le sous-espace vectoriel engendré par $\{\mathbf{C}_S^1, \ldots, \mathbf{C}_S^\delta\}$. Donc $\mathsf{C}(S)$ admet un sous-espace vectoriel de $\mathsf{E}(G_u(S,i))$ en somme directe et de dimension au moins i. Donc $i \leq \dim(\mathsf{E}(G_u(S,i))) - \dim(\mathsf{E}(G_u(S,i)) \cap \mathsf{C}(S))$. □

Construction pratique

Nous présentons deux méthodes de construction de solutions. La première méthode est basée sur la démonstration de l'existence de solutions. La deuxième utilise la programmation linéaire et construit directement des parties linéaires d'ordonnancement, c'est-à-dire des points des polyèdres $\mathsf{P}(G_u(S,j))$.

Construction basée sur la démonstration du lemme 20

Pour effectivement pouvoir construire nos vecteurs $\mathbf{E}_S^i$, il nous suffit de savoir construire les espaces $H(G_u(S,k))$ apparaissant dans la démonstration du lemme 20. Ces espaces $H(G_u(S,k))$ sont en fait fournis par le lemme 21. Il nous suffit donc de savoir construire une solution du lemme 21 pour résoudre notre problème. L'observation de la démonstration du lemme 21 montre qu'il nous suffit de savoir trouver un élément z de $\mathsf{E}(G_u(S,i))$qui n'appartient pas à $\cup_{i=1}^m F_i$ pour savoir construire une solution.

Construction d'une solution du lemme 21. Soit un ensemble de sous-espaces vectoriels E, $F_1, \ldots, F_m$ de $\mathbb{Q}^n$. Nous supposons que, pour tout $i \in [1,m]$, il existe un point x_i de E n'appartenant pas à F_i : $F_i \oplus \text{Vect}(x_i)$. Par récurrence sur m nous construisons un point x de E qui n'appartient pas à $\cup_{i=1}^m F_i$.

Pour $m = 1$, la propriété est vraie : il suffit de tester l'appartenance à F_1 des vecteurs d'une base de E. Supposons la propriété vérifiée pour i sous-espaces vectoriels. Nous avons donc un élément x_i de E tel que $x_i \notin \cup_{j=1}^i F_j$. Nous savons construire un point y de $E \setminus F_{i+1}$. Nous considérons un ensemble $H_{i+1} = \{p_k = \lambda_k x_i + (1-\lambda_k)y\}_{1 \leq k \leq i+2}$ où $\lambda_1, \ldots, \lambda_{i+2}$ sont $i+2$ rationnels distincts quelconques. x_i et y sont des points de l'espace vectoriel E, donc H_{i+1} est un sous-ensemble de E. Un des $i+2$ points de H_{i+1} n'appartient pas à $\cup_{j=1}^{i+1} F_j$. En effet, supposons que tel ne soit pas le cas. Chacun des $i+2$ points de H_{i+1} appartient alors à l'un des $i+1$ ensembles F_j, $1 \leq j \leq i+1$. Un de ces ensembles, noté F_k, contient alors deux des points de H_{i+1}, notés p_l et p_m. F_k étant un espace vectoriel, il contient $\lambda_m p_l - \lambda_l p_m = (\lambda_m - \lambda_l)y$. Or, $\lambda_m \neq \lambda_l$ par définition. Donc l'espace vectoriel F_k contient y. Nous montrons de même que F_k contient x_i. Or, si $k \in [1, i]$, F_k ne contient pas x_i par définition de x_i, et si $k = I+1$, F_k ne contient pas y par définition de y. Dans tous les cas il y a contradiction et un des éléments de H_{i+1} est solution de notre problème.

Inconvénient de cette approche. Cette méthode de construction de solutions a l'inconvénient de contraindre la recherche. La construction des espaces $H(G_u(S,k))$ repose sur des choix arbitraires. Le choix du point de $\mathsf{P}(G_u(S,i))$ qui permet de transformer le point de $\mathsf{E}(G_u(S,i))$ en point de $\mathsf{P}(G_u(S,i))$ est lui aussi arbitraire. Nous proposons maintenant une construction de solutions directe mais beaucoup plus coûteuse.

Construction directe d'ordonnancements compatibles

Nous supposons que la condition énoncée par le lemme 20 est satisfaite. Le lemme 20 et l'algorithme de la section précédente nous assure de l'existence d'un ordonnancement compatible avec le placement.

Considérons un graphe $G_u(S,i)$ et supposons que les vecteurs $\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}$ ont déjà été construits. Nous recherchons un point $\mathbf{E}_S^i$ de $\mathsf{P}(G_u(S,i))$ tel que la famille $\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}, \mathbf{E}_S^i, \mathbf{C}_T^1, \ldots, \mathbf{C}_T^\delta$ soit libre pour tout nœud T de $G_u(S,i)$. La démonstration du lemme 20 nous assure seulement de l'existence d'un tel vecteur dans $\mathsf{E}(G_u(S,i))$, mais pas nécessairement dans $\mathsf{P}(G_u(S,i))$. Une fois obtenu le point de $\mathsf{E}(G_u(S,i))$ nous pouvons le transformer en un point de $\mathsf{P}(G_u(S,i))$ en utilisant la démarche utilisée par l'algorithme de la section précédente. Nous sommes donc assurés de l'existence d'un tel vecteur $\mathbf{E}_S^i$.

Pour chaque instruction T de $G_u(S,i)$ nous construisons une base B_T de l'espace dont les $\delta+i-1$ premiers vecteurs forment une base de $\mathsf{C}(T) \oplus \mathrm{Vect}\{\mathbf{E}_S^1, \ldots, \mathbf{E}_S^{i-1}\}$. Une solution de notre problème est exactement un vecteur $\mathbf{E}_S^i$ qui satisfait les contraintes d'un ordonnancement (système 4.1) et dont un des $n-(\delta+i-1)$ derniers coefficients de la décomposition dans la base B_T est non nul, pour tout nœud T de $G_u(S,i)$. Pour chaque nœud T de $G_u(S,i)$, nous choisissons arbitrairement un entier j_T dans $[\delta+i, n]$ et un signe σ_T dans $\{-1, 1\}$. Nous formons un système d'équations et d'inéquations linéaires défini à partir du système 4.1 auquel nous ajoutons les équations de décomposition de $\mathbf{E}_S^i$ sur chacune des bases B_T ainsi qu'une inéquation obligeant la composante de rang j_T de la décomposition de $\mathbf{E}_S^i$ sur B_T à être supérieure à un si $\sigma_T = 1$ et à être inférieure à moins un sinon. Ce système définit un polyèdre de solutions. L'ensemble des choix sur les j_T et les σ_T définit $2^{m(n-(\delta+i-1))}$ systèmes linéaires, où m est le nombre de placements C_T distincts, et donc $2^{m(n-(\delta+i-1))}$ polyèdres de solutions. Il est immédiat que si X est une solution à notre problème, il existe un rationnel $\lambda \geq 1$ tel que λX appartienne à l'un de ces polyèdres. Cette méthode, quelque soit sa complexité, nous donne accès à l'ensemble de toutes les solutions constructibles de manière gloutonne.

5.3 Exemple d'utilisation

Nous nous servons de l'exemple 16 du chapitre 4. La figure 5.2 rappelle son graphe de dépendance uniformisé et son sous-graphe des multi-circuits de poids nul.

Comme nous l'avons indiqué lors de l'étude de cet exemple, la partie linéaire de la première composante de l'ordonnancement doit être choisi dans le polyèdre : $\mathsf{P}(G_u(S_1,1)) = \mathsf{P}(G_u(S_2,1)) =$

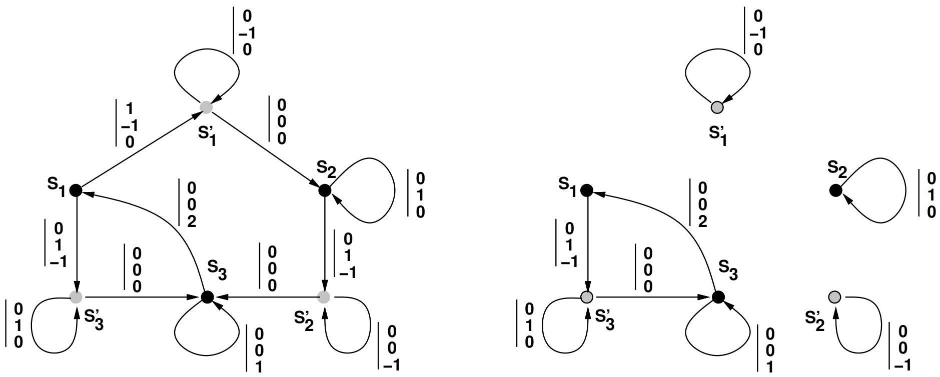


FIG. 5.2: *Graphe uniformisé et graphe des multi-circuits de poids nul de l'exemple 16.*

$\mathsf{P}(G_u(S_3,1)) = \{(x,0,0) \mid x \geq 2\}$. D'où les espaces vectoriels : $\mathsf{E}(G_u(S_1,1)) = \mathsf{E}(G_u(S_2,1)) = \mathsf{E}(G_u(S_3,1)) = \mathrm{Vect}\{(1,0,0)\}$.

Pour la deuxième composante de l'ordonnancement, nous devons considérer séparément les deux composantes fortement connexes :

- La première composante contient les instructions S_1 et S_3 et trois circuits de poids respectifs $(0,1,1)$, $(0,0,1)$ et $(0,1,0)$ et de longueur respectives deux, un et zéro (au sens défini par la fonction l). Nous obtenons donc : $\mathsf{P}(G_u(S_1,2)) = \mathsf{P}(G_u(S_3,2)) = \{(x,y,z) \mid y \geq 0, z \geq 1, y+z \geq 2\}$ et $\mathsf{E}(G_u(S_1,2)) = \mathsf{E}(G_u(S_3,2)) = \mathbb{Q}^3$.
- La deuxième composante contient l'instruction S_2 et un circuit de poids $(0,1,0)$ et de longueur un (au sens défini par la fonction l). Nous obtenons donc : $\mathsf{P}(G_u(S_2,2)) = \{(x,y,z) \mid y \geq 1\}$ et $\mathsf{E}(G_u(S_2,2)) = \mathbb{Q}^3$.

Nous supposons que nous ont été donnés les placements $C_{S_1} = (0,0,1)$, $C_{S_2} = (0,1,0)$ et $C_{S_3} = (0,1,1)$. Par résolution du programme linéaire (5.2) nous obtenons, par exemple, les hyperplans : $p(G_u(S_1,1)) = p(G_u(S_2,1)) = p(G_u(S_3,1)) = (4,0,0)$, $p(G_u(S_1,2)) = p(G_u(S_3,2)) = (0,1,2)$ et $p(G_u(S_2,2)) = (0,1,0)$.

Il existe un ordonnancement compatible car l'inégalité (5.3) est vérifiée pour $i = 1$ (aucun des vecteurs de placement n'est colinéaire à l'unique direction que peut prendre la partie linéaire de la première composante de l'ordonnancement) et $i = 2$ (les espaces vectoriels de recherche sont de dimension trois et les placements de dimension un).

Dans un premier temps nous cherchons des points des espaces vectoriels $\mathsf{E}(G_u(S_j,i))$:

- Nous commençons par rechercher $\mathbf{E}^1_{S_1} = \mathbf{E}^1_{S_2} = \mathbf{E}^1_{S_3}$ dans $\mathsf{E}(G_u(S_1,1)) = \mathsf{E}(G_u(S_2,1)) = \mathsf{E}(G_u(S_3,1))$. Nous devons trouver un élément z de $\mathrm{Vect}\{(1,0,0)\}$ qui ne soit pas élément de $\mathrm{Vect}(C_{S_1}) \cup \mathrm{Vect}(C_{S_2}) \cup \mathrm{Vect}(C_{S_3})$. Nous vérifions simplement que $z = (1,0,0)$ est solution et nous posons : $\mathbf{E}^1_{S_1} = \mathbf{E}^1_{S_2} = \mathbf{E}^1_{S_3} = (1,0,0)$.
- Nous cherchons $\mathbf{E}^2_{S_1} = \mathbf{E}^2_{S_3}$ dans $\mathsf{E}(G_u(S_1,2)) = \mathsf{E}(G_u(S_3,2)) = \mathbb{Q}^3$. Nous devons trouver un vecteur z qui ne soit pas élément de $\mathrm{Vect}(C_{S_1},(1,0,0)) \cup \mathrm{Vect}(C_{S_3},(1,0,0))$, c'est-à-dire de $\mathrm{Vect}((0,0,1),(1,0,0)) \cup \mathrm{Vect}((0,1,1),(1,0,0))$. Nous prenons par exemple $z = (0,1,0)$.
- Nous cherchons $\mathbf{E}^2_{S_3}$ dans $\mathsf{E}(G_u(S_2,2)) = \mathbb{Q}^3$. Nous devons trouver un vecteur z qui ne soit pas élément de $\mathrm{Vect}(C_{S_2},(1,0,0)) = \mathrm{Vect}((0,1,0),(1,0,0))$. Nous prenons par exemple $z = (0,0,1)$.

Nous construisons ensuite des points des polyèdres $\mathsf{P}(G_u(S_j,i))$ à l'aide de l'algorithme de la section 5.1 :

- Nous avons initialement les trois matrices :

$$P_{S_1,0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, P_{S_2,0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \text{ et } P_{S_3,0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

  Comme $p(G_u(S_1,1))$ vaut $(2,0,0)$, il est immédiat que l'ensemble μ vaut $\{-1/2\}$. De même, nous trouvons immédiatement $\nu = 1/2$. Nous obtenons le vecteur $\mathbf{E}^1_{S_1} = \mathbf{E}^1_{S_2} = \mathbf{E}^1_{S_3} = (2,0,0)$ en prenant arbitrairement $\lambda = 1/2$.

- Nous recherchons $\mathbf{E}^2_{S_1} = \mathbf{E}^2_{S_3}$ connaissant $p(G_u(S_1,2)) = (0,1,2)$. Dans ce cas μ est l'ensemble $\{-1;1\}$ et $\nu = 1/2$. Nous obtenons le vecteur $\mathbf{E}^2_{S_1} = \mathbf{E}^2_{S_3} = (0,3/2,1)$ en prenant arbitrairement $\lambda = 1/2$.

- Nous recherchons $\mathbf{E}^2_{S_2}$ connaissant $p(G_u(S_2,2)) = (0,1,0)$. Dans ce cas μ est l'ensemble vide et $\nu = 1$. Nous obtenons le vecteur $\mathbf{E}^2_{S_2} = (0,1,1)$ en prenant arbitrairement $\lambda = 1$.

Finalement nous obtenons les matrices de transformation temps-espace :

$$P_{S_1,2} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3/2 & 1 \\ 0 & 0 & 1 \end{bmatrix}, P_{S_2,2} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \text{ et } P_{S_3,2} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3/2 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

Ces matrices sont régulières et leurs deux premières lignes définissent les parties linéaires d'un ordonnancement. Nous trouvons les constantes nécessaires à la complétion de l'ordonnancement, soit par programmation linéaire, soit au moyen d'un algorithme de calcul des plus courts chemins. Nous obtenons par exemple : $\rho^1_{S_1} = 1$, $\rho^1_{S_2} = 0$, $\rho^1_{S_3} = 1$, $\rho^2_{S_1} = 0$, $\rho^2_{S_2} = 0$ et $\rho^2_{S_3} = 1/2$. L'itération (i, j, k) de l'instruction S_1 doit donc être exécutée au temps $\lfloor$(2i+1, 3/2j+k)$\rfloor$ sur le processeur k, l'itération (i, j, k) de S_2 doit être exécutée au temps $\lfloor$(2i, j+k)$\rfloor$ sur le processeur j, et l'itération (i, j, k) de S_3 doit être exécutée au temps $\lfloor$(2i+1, 3/2j+k+1/2)$\rfloor$ sur le processeur j+k. La figure 5.3 présente le code après transformation (« intDiv » y désigne la division euclidienne).

```
ForSeq t1 = 1 to n do
  ForSeq t2 = 2 to 5n+1 do
    ForPar p = intDiv(t2+1,2) to min(t2-1,n) do
      b(t1, p, t2-p) = a(t1-1, p+t1, t2-p) + b(t1, p-1, t2-p)
    ForPar p = max(1+(-t2-1)%3,3*intDiv(intDiv(t2-3n+1,2) +t2+2,3)-t2) to intDiv(t2,5) step 3 do
      a(t1, (2p-t2)/-3, p) = c(t1, (2p-t2)/-3, p-1) + 1
    ForPar p = max(intDiv(t2+2,3),intDiv(t2-n,2)) to intDiv(2t2-2,5) do
      c(t1, t2-2p-1, 3p-t2+2) = c(t1, t2-2p-1, 3p-t2+1) + b(t1, t2-2p-2, 3p-t2+t1+1)
                                + a(t1, -5p+2t2-2, 3p-t2+2)
```

FIG. 5.3: *Exemple 16 après application de la transformation.*

5.4 Généralité de l'approche

Nous venons de proposer une méthode de construction d'ordonnancements compatibles avec un placement des calculs donné. L'algorithme d'ordonnancement sous-jacent est celui du chapitre 4.

Notre méthode ne teste l'existence et ne construit un ordonnancement compatible avec tous les placements que s'il en existe un *constructible* par ORDONNANCEUR_DE_GDRP. Réciproquement, s'il existe un tel ordonnancement nous le trouvons.

Nous recherchons des ordonnancements affines particuliers, qui sont localement des ordonnancements linéaires décalés. Nous pourrions donc craindre l'existence de cas pour lesquels il existerait des ordonnancements affines compatibles bien que nos algorithmes soient incapables d'en trouver. En fait, le théorème 10 nous assure du succès de notre approche en cas d'existence d'un ordonnancement affine quelconque compatible avec le placement. Nous sommes bien sûr toujours dans le cadre où les dépendances sont représentées par un graphe de dépendance réduit polyédrique.

Théorème 10. *Soit un nid de boucles parfait dont les dépendances sont représentées par un graphe de dépendance réduit polyédrique et soit un placement affine de ses calculs. Nous supposons que le domaine d'itération contient (resp. est contenu dans) un hypercube de dimension* n *et de taille* $\Omega(N)$ *(resp.* $O(N)$*). S'il existe un ordonnancement multidimensionnel affine compatible avec le placement, alors il existe un tel ordonnancement constructible par l'algorithme* ORDONNANCEUR_DE_GDRP.

Avant de démontrer ce théorème, nous en déduisons immédiatement la condition d'existence d'un ordonnancement affine compatible, avec les notations définies précédemment :

Théorème 11. *Soit un nid de boucles parfait dont les dépendances sont représentées par un graphe de dépendance réduit polyédrique et soit un placement affine de ses calculs. Nous supposons que le domaine d'itération contient (resp. est contenu dans) un hypercube de dimension* n *et de taille* $\Omega(N)$ *(resp.* $O(N)$*).*

Il existe un ordonnancement multidimensionnel affine compatible avec le placement si et seulement si, pour toute instruction S *et tout entier* i *de* $[1, d_S]$*, nous avons :*

$$i + \dim(\mathsf{E}(G_u(S,i)) \cap \mathsf{C}(S)) \leq \dim(\mathsf{E}(G_u(S,i))).$$

DÉMONSTRATION DU THÉORÈME 10.

Cette démonstration repose sur celle du lemme 9 qui prouve, en étudiant les propriétés d'un ordonnancement affine quelconque sur un GDRP, que l'algorithme de Paul Feautrier se comporte exactement de la même manière que le nôtre sur des GDRP

Soit E' un ordonnancement affine multidimensionnel compatible avec le placement des calculs. Nous notons $\mathbf{E}'^i_S$ la partie linéaire de la $i^{ème}$ composante de l'ordonnancement de S défini par E'.

Considérons le graphe G_i défini comme le sous-graphe de G_u généré par les dépendances qui ne sont pas satisfaites par une des $i-1$ premières dimensions d'un ordonnancement produit par ORDONNANCEUR_DE_GDRP : $G_i = \cup_{S \in G_u} G_u(S,i)$. Considérons le graphe $\mathcal{G}_i$ des dépendances qui ne sont satisfaites par aucune des $i-1$ premières dimensions de E', notre ordonnancement affine quelconque. Initialement, nous avons $\mathcal{G}_1 = G_1 = G$. Nous prouvons par récurrence l'inclusion $G_i \subset \mathcal{G}_i$. Nous supposons donc la propriété vérifiée jusqu'au rang i inclus.

Soient F une composante fortement connexe de G_i et S une instruction de F. Nous avons vu dans la démonstration du lemme 9 qu'une fonction affine induisant un délai positif sur toutes les arêtes d'un GDRP ne pouvait satisfaire aucune arête qui ne serait pas satisfaite par un hyperplan faiblement séparateur produit par ORDONNANCEUR_DE_GDRP. Par conséquent, parmi les arêtes de $\mathcal{G}_i$, la $i^{ème}$ composante de E' ne peut satisfaire aucune arête de G_{i+1} car $G_{i+1} \subset G_i \subset \mathcal{G}_i$. D'où l'inclusion $G_{i+1} \subset \mathcal{G}_{i+1}$.

Nous montrons que $\mathbf{E}'^i_S$ appartient à $\mathsf{E}(G_u(S,i))$, pour $i \leq d_j$. Nous avons vu, toujours dans la démonstration du lemme 9, que la partie linéaire $\mathbf{E}'^i_S$ était obligatoirement orthogonale au poids de tout circuit élémentaire participant à un multi-circuit de poids nul inclus dans la composante fortement connexe de $\mathcal{G}_i$ contenant S. En particulier, $\mathbf{E}'^i_S$ doit être orthogonal au poids de tout circuit élémentaire participant à un multi-circuit de poids nul inclus dans la composante fortement connexe de G_i contenant S. Ce qui veut exactement dire que $\mathbf{E}'^i_S$ appartient à $\mathsf{E}(G_u(S,i))$ (cf. la définition de $\mathsf{E}(G_u(S,i))$ par le lemme 9). Nous ne pouvons pas conclure directement car les différents nœuds de $G_u(S,i)$ n'ont pas forcément la même $i^{ème}$ partie linéaire.

Nous avons montré que s'il existe un ordonnancement multidimensionnel E' compatible avec le placement, alors la partie linéaire de la $i^{ème}$ composante de l'ordonnancement d'une instruction S, $\mathbf{E}'^i_S$, appartient à $\mathsf{E}(G_u(S,i))$. Pour toute instruction S et tout entier i de $[1, d_S]$, la famille $\{\mathbf{E}'^1_S, \ldots, \mathbf{E}'^i_S, \mathbf{C}^1_S, \ldots, \mathbf{C}^\delta_S\}$ est libre et $\mathrm{Vect}(\{\mathbf{E}'^1_S, \ldots, \mathbf{E}'^i_S\}) \subset \mathsf{E}(G_u(S,i))$, nous avons donc :

$$i + \dim(\mathsf{E}(G_u(S,i)) \cap \mathsf{C}(S)) \leq \dim(\mathsf{E}(G_u(S,i))).$$

C'est tout ce dont à besoin le lemme 20 pour assurer l'existence d'un ordonnancement compatible. □

Le théorème 10 et son corollaire sont bien évidemment également vrais si nous supposons que l'algorithme d'ordonnancement étudié n'a aucune connaissance du domaine d'itération du nid étudié.

5.5 Conclusion

Nous avons présenté une méthode qui prend en entrée le graphe de dépendance réduit polyédrique d'un nid de boucles et un placement affine de ses calculs. Nous savons construire un ordonnancement du nid compatible avec le placement si et seulement si il existe un ordonnancement affine quelconque compatible avec ce placement et satisfaisant le GDRP. Cette extension de ORDONNANCEUR_DE_GDRP montre encore à quel point cet algorithme est adapté aux approximations polyédriques des dépendances. Elle illustre également les possibilités offertes par la connaissance de l'ensemble des ordonnancements constructibles par nos méthodes.

Une fois le placement des calculs donné, le paralléliseur peut se rendre compte que certaines dépendances vont engendrer des communications plus ou moins coûteuses. Il lui est tout à fait possible d'obliger les ordonnancements à induire un délai plus longs sur ces dépendances en remplaçant dans le système 4.1 la valeur « 1 » par celle de son choix et retarder relativement l'exécution de certaines opérations.

Chapitre 6

De l'élimination des fausses dépendances

> Monsieur Valette ! Monsieur Valette ! Que Monsieur Valette me foute la paix, bon Dieu de bon Dieu, ou qu'il prenne un avocat. S'il vous plaît ! J'ai fini mon travail, moi. Monsieur Valette ! C'est intolérable, que d'histoires pour un licenciement ! C'est ridicule ! Quand je pense qu'à l'heure actuelle il y a des assurances comme jamais auparavant dans l'histoire de l'humanité... Oui! Qu'il prenne un avocat. S'il vous plaît ! Moi aussi je prendrai un avocat. Exiger une participation à sa découverte ! Mais qu'il ouvre le robinet du gaz ou qu'il prenne un avocat — s'il vous plaît ! — si Monsieur Valette peut se payer le luxe de perdre un procès ou de le gagner. S'il vous plaît ! S'il vous plaît !
>
> M Bonhomme *in* Monsieur Bonhomme et les incendiaires, Max **Frisch**

Nous avons vu au chapitre 1.2 qu'il existait trois types de dépendances de données : les dépendances de *flot*, les *anti*-dépendances et les dépendances de *sortie*. Les dépendances de flot sont dues à la réutilisation d'une valeur précédemment calculée. Elles sont inhérentes à l'algorithme, ce qui leur vaut le qualificatif de *vraies* dépendances. Au contraire, les anti-dépendances et les dépendances de sortie sont dues à la réutilisation de variables, c'est-à-dire de la mémoire. Elles ne sont pas inhérentes au schéma de calcul de l'algorithme, mais à son implémentation. D'où leur surnom de *fausses* dépendances. Alors que les vraies dépendances sont présentes dans toute implémentation d'un algorithme donné, les fausses dépendances sont spécifiques de l'implémentation étudiée. Ces dépendances surnuméraires peuvent entraîner une séquentialisation partielle ou totale du nid de boucles. Mais, dans un certains nombre de cas, il est possible de les éliminer en introduisant de nouvelles variables, c'est-à-dire plus de mémoire. Face au problème posé par ces fausses dépendances, le paralléliseur peut adopter deux positions extrêmes : ignorer le problème et ne pas différencier vraies et fausses dépendances au risque de ne pas pouvoir extraire tout le parallélisme contenu dans l'algorithme, voire de croire le programme purement séquentiel; ou bien éliminer toutes les fausses dépendances (passage en assignation unique) quelqu'en soit le coût en mémoire. Aucune de ces deux attitudes n'est satisfaisante. Il serait de loin préférable de ne supprimer que les fausses dépendances qui gênent la parallélisation.

Nous revenons à la section 6.1 sur les fausses dépendances et les faux chemins de dépendances. Nous insisterons particulièrement sur leur côté artificiel. Section 6.2 nous formaliserons et étudierons deux techniques d'élimination de chemins de dépendance originellement présentées par David Padua et Michael Wolfe. Nous présenterons finalement à la section 6.3 un schéma général d'intégration des techniques d'élimination des dépendances au sein des algorithmes de détection du parallélisme et d'ordonnancement. Cette intégration a pour but de ne supprimer que les fausses dépendances gênant la détection de parallélisme. Nous essayerons d'appliquer ce schéma aux algorithmes d'Allen et Kennedy, de Wolf et Lam et à celui présenté au chapitre 4.

6.1 Fausses dépendances, faux chemins de dépendance

Cette section ne présente que des généralités. Elle a pour but de préciser le problème. Nous ne nous intéressons qu'aux nids de boucles. Notre but étant la parallélisation de ces structures de calcul, nous ne considérerons que les dépendances entre instructions du nid étudié, ignorant les dépendances en entrée et en sortie du nid.

Remarque. Dans toutes les figures, les dépendances de flot, de sortie et les anti-dépendances sont respectivement annotées f, o et a. Cette annotation est complétée par l'indice *in* ou *out*[1] suivant que la dépendance se produit en entrée ou en sortie de l'instruction considérée.

Il existe trois types de dépendances :

- *Dépendance de flot.* Une dépendance de flot est générée par l'accès en lecture d'une variable (zone mémoire) précédemment accédée en écriture. Une dépendance de flot est donc due à l'utilisation d'une valeur calculée au sein même du nid de boucles. Elle est par conséquent inhérente au schéma de calcul. C'est une *vraie* dépendance.

  Une dépendance de flot en entrée d'une instruction peut concerner n'importe quelle variable accédée en lecture au sein de l'instruction et même une variable lue dans la fonction d'accès du tableau qui serait affecté par l'instruction.

  Une dépendance de flot en sortie d'une instruction ne peut avoir comme source que la variable affectée.

- *Dépendance de sortie.* Une dépendance de sortie est générée par l'accès en écriture d'une variable (zone mémoire) précédemment accédée en écriture. Une dépendance de sortie est donc due à la réaffectation d'une variable déjà affectée par le nid. Intuitivement, pour supprimer la dépendance de sortie, il suffirait que la deuxième affectation ait lieu à un emplacement mémoire inutilisé, et que les accès originels en lecture à la valeur stockée par la seconde affectation soit modifiés en conséquence. Une dépendance de sortie n'est donc pas propre au calcul mais à son implémentation. Il s'agit d'une *fausse* dépendance.

  Une dépendance de sortie en entrée ou en sortie d'une instruction ne peut concerner que la variable affectée par l'instruction.

- *Anti-dépendance.* Une anti-dépendance est générée par l'accès en écriture d'une variable (zone mémoire) précédemment accédée en lecture. Une anti-dépendance est donc due à la réaffectation d'une variable précédemment lue par le nid. Un traitement similaire à celui proposé pour les dépendances de sortie permettrait d'éliminer ce type de dépendance. Une anti-dépendance est donc également une *fausse* dépendance.

  Une anti-dépendance en entrée d'une instruction ne peut concerner que la variable affectée par l'instruction.

1. Cette annotation suit donc la dénomination anglo-saxonne, pour des raisons non scientifiques.

Une anti-dépendance en sortie d'une instruction peut concerner n'importe quelle variable accédée en lecture au sein de l'instruction, et en particulier une variable lue dans la fonction d'accès du tableau qui serait affecté par l'instruction.

De cette énumération répétitive, il ressort que les anti-dépendances et dépendances de sortie peuvent théoriquement être éliminées par le même type de manipulation. Il apparaît aussi qu'une variable accédée au sein d'une instruction ne peut pas être concernée par les six formes de dépendances (les trois types en entrée et en sortie). La figure 6.1 illustre le lien entre les différents types d'accès aux variables et les différents types de dépendance.

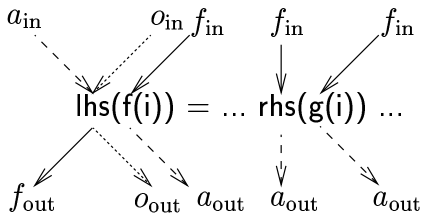


FIG. 6.1: *Les différents types d'accès aux variables et les différents types de dépendance.*

De ces trivialités il découle que le chemin de dépendance de S_1 vers S_3 du graphe de dépendance de l'exemple 20 (figure 6.2) est un mirage. En effet, ce chemin est constitué d'une dépendance de sortie suivie d'une anti-dépendance. La figure 6.1 nous montre qu'une même variable ne peut pas être la cible d'une dépendance de sortie et la source d'une anti-dépendance. Ce chemin de dépendance n'est donc pas une conséquence des accès mémoire au sein du nid mais de la représentation des dépendances. Dans le graphe de dépendance de la figure 6.3 nous avons décomposé les instructions de l'exemple 20, séparant les différents accès mémoire (le symbole S_j représente ici l'unique accès en écriture au sein de cette instruction; le symbole "lect b(i)" désigne l'accès en lecture au tableau b). Notre chemin de dépendance a disparu.

Exemple 20

For i = 1 to N do
  S_1 : a(i) = b(i)
  S_2 : a(i) = c(i)
  S_3 : c(i) = d(i)

$$S_1 \xrightarrow{o} S_2 \xrightarrow{a} S_3$$

FIG. 6.2: *Exemple 20 et son graphe de dépendance.*

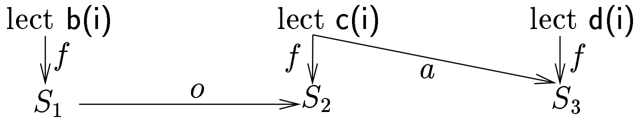


FIG. 6.3: *Graphe de dépendance entre opérations pour l'exemple 20.*

Nous avons vu que considérer le corps de boucle comme étant atomique sur-contraignait la parallélisation et faisait perdre du parallélisme (sous-optimalité de WOLF-LAM illustrée à la section 2.3). Nous venons d'illustrer le même type de problème : une représentation trop grossière de l'origine des dépendances (l'instruction toute entière plutôt qu'une de ses opérations élémentaires constitutives) fait apparaître de faux chemins de dépendances. Les deux transformations que nous allons maintenant étudier essayent entre autres de placer dans des instructions différentes des opérations créant des chemins de dépendance qui ne sont dus qu'à la granularité de notre représentation des dépendances.

6.2 Deux techniques d'élimination de chemins de dépendance

David Padua et Michael Wolfe ont présenté deux techniques d'élimination de chemins de dépendance dans un article traitant d'optimisations de programmes [65]. Ces techniques n'étaient pas formalisées mais simplement présentées chacune sur un exemple. Nous commençons par rappeler leur présentation originelle au paragraphe 6.2.1, puis nous formalisons et étudions respectivement les deux transformations (paragraphes 6.2.2 et 6.2.3). Finalement, nous considérons l'efficacité de ces deux transformations.

6.2.1 Présentation originelle

Nous reprenons ici les exemples de l'article de Padua et Wolfe [65] et leur dénomination des transformations.

Anti-dépendances

Exemple 21

For i = 1 to N do
  S_1 : a(i) = b(i) + c(i)
  S_2 : d(i) = (a(i) + a(i+1)) / 2

$$\begin{tikzcd} S_1 \arrow[r, bend right, "f"'] & S_2 \arrow[l, bend right, "a"'] \end{tikzcd}$$

FIG. 6.4: *Exemple 21 et son graphe de dépendance* avant *transformation.*

Considérons l'exemple 21 (figure 6.4). S_1 écrivant a(i) juste avant que S_2 ne l'utilise, il y a une dépendance de flot de S_1 vers S_2. Il y a aussi une anti-dépendance de S_2 vers S_1, la valeur de a(i+1) devant être lue par S_2 avant d'être réécrite par S_1 à l'itération suivante. Il existe donc un circuit de dépendances comme le montre le graphe de dépendance réduit de la figure 6.4. Ce circuit peut être supprimé par l'introduction d'un tableau temporaire, comme le montre la figure 6.5.

For i = 1 to N do
  S_2' : temp(i) = a(i+1)
  S_1 : a(i) = b(i) + c(i)
  S_2 : d(i) = (a(i) + temp(i)) / 2

$$\begin{tikzcd} S_2' \arrow[r, "a"] \arrow[rr, bend right, "f"'] & S_1 \arrow[r, "f"] & S_2 \end{tikzcd}$$

FIG. 6.5: *Exemple 21 et son graphe de dépendance* après *transformation.*

La transformation du programme n'a pas éliminé de dépendances. Au contraire, elle en a fait apparaître une nouvelle : une dépendance de flot de S_2' vers S_2 due au temporaire. Mais le nouveau graphe de dépendance est sans circuits. Le nouveau nid de boucles est donc immédiatement vectorisable :

S_2' : temp(1:N) = a(2:N+1)
S_1 : a(1:N) = b(1:N) + c(1:N)
S_2 : d(1:N) = (a(1:N) + temp(1:N)) / 2

Dépendances de sortie

Nous pouvons également détruire un circuit de dépendances contenant une dépendance de sortie. Considérons l'exemple 22 (figure 6.6). La zone mémoire a(i+1) est écrite par l'instruction S_2 avant d'être réécrite par S_1 à l'itération suivante, d'où une dépendance de sortie de S_2 vers S_1. Il existe également une dépendance de flot de S_1 vers S_2 due aux deux accès a(i). Nous introduisons un tableau temporaire comme indiqué figure 6.7. Le nouveau nid de boucles est sans circuits et peut donc, par exemple, être vectorisé.

Exemple 22

For i = 1 to N do
  S_1 : a(i) = b(i) + c(i)
  S_2 : a(i+1) = a(i) + 2 × d(i)

$$\begin{tikzcd} S_1 \arrow[r, bend right, "f"'] & S_2 \arrow[l, bend right, "o"'] \end{tikzcd}$$

FIG. 6.6: *Exemple 22 et son graphe de dépendance* avant *transformation.*

For i = 1 to N do
  S'_1 : temp(i) = b(i) + c(i)
  S_2 : a(i+1) = temp(i) + 2 × d(i)
  S_1 : a(i) = temp(i)

$$\begin{tikzcd} S'_1 \arrow[r, "f"] \arrow[rr, bend right, "f"'] & S_2 \arrow[r, "o"] & S_1 \end{tikzcd}$$

FIG. 6.7: *Exemple 22 et son graphe de dépendance* après *transformation.*

En résumé, les deux transformations ont supprimé des circuits de dépendances, permettant la vectorisation et/ou facilitant la parallélisation, au prix d'une augmentation de la mémoire requise par le programme. Dans les deux cas nous avons dû introduire un tableau temporaire. Padua et Wolfe [65] n'ont décrit leurs méthodes qu'au moyen des exemples ci-dessus. Nous allons maintenant définir formellement ces deux transformations.

6.2.2 Définition formelle et étude de la première transformation

Définition formelle

Dans l'exemple 21, Padua et Wolfe ont introduit un tableau temporaire et y ont stocké la valeur contenue dans l'emplacement mémoire responsable de l'anti-dépendance. L'expression a(i+1) dans l'instruction S_2 de l'exemple 21 représente l'accès à un emplacement mémoire précis et la lecture d'une valeur indispensable au calcul. Dans cette instruction, il est nécessaire d'accéder à un emplacement mémoire où a été sauvegardée une copie de la valeur enregistrée en a(i+1), mais cette zone mémoire n'a aucunement besoin d'être a(i+1). Padua et Wolfe ont donc décomposé l'accès a(i+1) en deux parties : dans la nouvelle instruction S'_2, ils accèdent effectivement à la zone mémoire a(i+1) et copient son contenu dans un nouveau tableau; dans l'instruction S_2 originelle, ils remplacent la référence à a(i+1) par une référence au nouveau tableau. Ils séparent ainsi la lecture en mémoire de l'utilisation de la valeur lue.

Nous définissons formellement la première transformation de Padua et Wolfe à la figure 6.8. Cette transformation peut être appliquée à n'importe quel accès en lecture[2]. Ici nous l'appliquons à l'accès rhs(g(i)). Elle provoque l'apparition d'une nouvelle instruction et d'un tableau temporaire qui ne doit pas être utilisé ailleurs dans le programme.

2. Notez qu'elle peut même être appliquée à un accès en lecture dans le membre gauche de l'instruction S_k, c'est-à-dire, pour notre exemple, à une lecture effectuée au sein même de la fonction f(i).

```
For i = 1 to N do
   ...
   S_k : lhs(f(i)) = ... rhs(g(i)) ...
   ...
```

Nid avant transformation.

```
For i = 1 to N do
   ...
   S'_k : temp(i) = rhs(g(i))
   S_k : lhs(f(i)) = ... temp(i) ...
   ...
```

Nid après transformation.

FIG. 6.8: *Définition formelle de la première transformation de Padua et Wolfe.*

Remarque : cette transformation s'applique à des nids de profondeur quelconque, parfaits ou non, même si tous nos exemples ne contiennent qu'une seule boucle.

Conséquences

Cette transformation a pour but la destruction de chemins de dépendance qui traversent l'instruction S_k. Considérons S_k avant transformation (figure 6.8). Il peut y avoir des dépendances de flot, de sortie et des anti-dépendances aboutissant ou trouvant leur origine en S_k, soit six types différents d'arcs de dépendance (cf. figure 6.11). Comme nous n'avons modifié qu'un accès en lecture de S_k, la transformation n'a pu affecter qu'une dépendance de flot en entrée ou une anti-dépendance en sortie. Nous allons discuter l'impact de la transformation sur ces dépendances, sans oublier la nouvelle dépendance de flot f_{new} de S'_k vers S_k.

- **Dépendance de flot en entrée** (figure 6.9). La donnée rhs(g(i)) lue dans l'instruction S_k a été précédemment produite dans une quelconque instruction S_l. Après transformation, cette donnée est lue dans l'instruction S'_k et la dépendance de flot est donc entre S_l et S'_k.

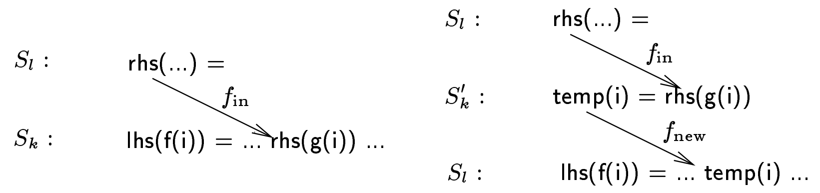


FIG. 6.9: *Dépendance de flot en entrée : avant et après transformation.*

- **Anti-dépendance en sortie** (figure 6.10). La donnée rhs(g(i)) lue dans l'instruction S_k est ultérieurement réécrite dans une quelconque instruction S_l. Après transformation, cette donnée est lue dans l'instruction S'_k et l'anti-dépendance est donc entre S'_k et S_l.

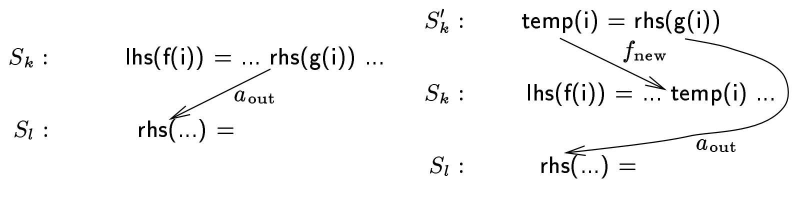


FIG. 6.10: *Anti-dépendance en sortie : avant et après transformation.*

Notez que les dépendances de flot en entrée et les anti-dépendances en sortie, qui ne concernent pas rhs(g(i)) mais un autre accès en lecture de S_k, ne sont pas modifiées par la transformation.

Ainsi l'exemple 21 contient une dépendance de flot en entrée de l'instruction S_2. Cette dépendance n'est pas due à l'accès a(i+1), cible de la transformation, mais à l'accès a(i). La dépendance de flot de S_1 vers S_2 n'est donc pas modifiée pas la transformation (cf. figure 6.5).

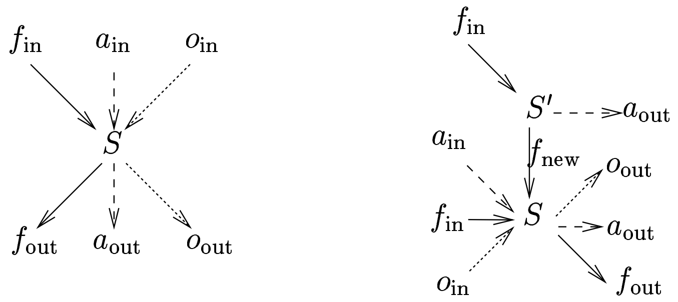


FIG. 6.11: *Dépendances avant et après application de la première transformation à S.*

La figure 6.11 résume l'impact de la transformation. Les auto-dépendances sont traitées comme les autres dépendances. Considérons une auto-anti-dépendance de l'instruction S_k. Celle-ci, initialement orientée de S_k vers S_k, est remplacée par une dépendance de S'_k vers S_k.

Dans l'optique de l'élimination de chemins de dépendance, la transformation d'une instruction S donnée se justifie si S a en entrée une anti-dépendance ou dépendance de sortie et en sortie une anti-dépendance (cf. figure 6.11) :

$$? \xrightarrow{a,o} S \xrightarrow{a} ?$$

Dans ce cas, la transformation doit être appliquée à l'accès en lecture source de l'anti-dépendance. La transformation est également intéressante si S a en entrée une dépendance de flot et en sortie une anti-dépendance, à la condition que les accès en lecture au sein de l'instruction S responsables de ces deux dépendances soient différents et à la condition que la transformation soit appliquée à l'accès en lecture source de l'anti-dépendance. Cette propriété est également illustrée par l'exemple 21 : la dépendance de flot en entrée de S_2 est due à l'accès a(i) alors que l'anti-dépendance en sortie est due à a(i+1).

6.2.3 Définition formelle et étude de la seconde transformation

Nous définissons la transformation avant de l'étudier, d'abord dans le cas des nids dont les fonctions d'accès en écriture ne génèrent pas de dépendances, puis dans le cas général. Finalement, nous mettons en évidence l'importance de l'instruction qui recopie la valeur calculée dans son lieu de stockage originel.

Définition formelle

Dans leur transformation de l'exemple 22, Padua et Wolfe ont en fait séparé la production et l'utilisation d'une valeur de son enregistrement en mémoire.

Nous définissons formellement la deuxième transformation de Padua et Wolfe à la figure 6.12. L'instruction S_k considérée est scindée en deux instructions : dans la première, notée S'_k, le membre droit de l'instruction originelle est évalué et affecté dans un tableau temporaire temp; dans la deuxième, notée S_k, la valeur calculée par S'_k est recopiée du tableau temporaire dans l'emplacement mémoire où elle était originellement stockée, ici le tableau lhs.

Nous décidons que les affectations dans les deux nouvelles instructions se font au moyen de la même fonction d'accès que dans l'instruction originelle. Nous aurions pu décider, comme pour la première transformation, que l'affectation dans S'_k se ferait sous la forme temp(i) $=$ quelle que soit la fonction d'accès originelle. Ces deux solutions peuvent en effet être considérées comme des extensions « naturelles » de l'exemple de Padua et Wolfe. Cependant, pour la deuxième transformation, nous allons remplacer certains accès en lecture aux tableaux lhs par des accès aux tableaux temp. Le fait que l'affectation dans temp au sein de S'_k se fasse avec la même fonction d'accès que l'assignation originelle dans lhs nous permet de réutiliser partout les fonctions d'accès utilisées dans le code originel et de ne pas à avoir à en inverser certaines.

Toutes les lectures[3] lhs(g(i)) dans le tableau lhs qui concernent une valeur calculée par S'_k sont remplacées par des lectures temp(g(i)) dans le temporaire. Cette substitution d'accès nécessite une bonne connaissance du flot de données et donc l'utilisation de puissants analyseurs de dépendances. Nous devons en effet savoir si la valeur lue dans un accès en lecture à lhs a ou non été produite par l'instruction S_k. La réécriture peut requérir l'utilisation de conditionnelles comme le montre l'exemple 23 au paragraphe 6.2.5. En fait, nous ne sommes pas obligés de remplacer toutes les lectures dans le tableau lhs qui concernent une valeur calculée par S'_k. Le code est tout à fait correct si un ou plusieurs accès en lecture sont laissés inchangés, grâce à l'instruction $_k$. Il s'agit donc une fois de plus d'un choix. Comme nous voulons étudier la puissance de la transformation nous considérons sa forme qui provoque le plus de modifications.

For i = 1 to N do
  ...
  S_k : lhs(f(i)) = rhs(...)
  ...

Nid avant transformation.

For i = 1 to N do
  ...
  S'_k : temp(f(i)) = rhs(...)
  S_k : lhs(f(i)) = temp(f(i))
  ...

Nid après transformation.

FIG. 6.12: *Définition formelle de la deuxième transformation de Padua et Wolfe.*

Cette deuxième transformation provoque également l'apparition d'une nouvelle instruction et d'un tableau temporaire qui ne doit pas être utilisé ailleurs dans le programme.

Conséquences

Nous étudions l'impact de cette transformation sur les six types de dépendances impliquant l'instruction S_k considérée (cf. figure 6.19). Comme précédemment, il y a une nouvelle dépendance de flot f_{new} de S'_k vers S_k. Nous supposerons pour cette étude que les fonctions d'accès en écriture aux tableaux et variables sont injectives (cf. le cas « dépendance de sortie en sortie »). Nous supposerons également que ces fonctions ne génèrent pas de dépendances mais nous montrerons ultérieurement comment lever cette dernière limitation.

- **Dépendance de flot en entrée** (figure 6.13). Par hypothèse, cette dépendance ne peut pas concerner la fonction d'accès f(i) de la variable lhs. Elle concerne donc une donnée lue dans le membre droit de l'instruction S_k et précédemment produite dans une quelconque instruction S_l. Après transformation, la donnée n'est lue que dans le membre droit de l'instruction S'_k et la dépendance de flot est donc entre S'_k et S_l.

3. Seuls les accès au tableau lhs au sein du nid de boucles considéré sont modifiés.

$$\begin{array}{ll} S_l : & \mathsf{rhs}(\ldots) = \\ & \qquad f_{\text{in}} \\ S_k : & \mathsf{lhs}(\mathsf{f}(\mathsf{i})) = \mathsf{rhs}(\ldots) \end{array} \qquad\qquad \begin{array}{ll} S_l : & \mathsf{rhs}(\ldots) = \\ & \qquad f_{\text{in}} \\ S'_k : & \mathsf{temp}(\mathsf{f}(\mathsf{i})) = \mathsf{rhs}(\ldots) \\ & \qquad f_{\text{new}} \\ S_l : & \mathsf{lhs}(\mathsf{f}(\mathsf{i})) = \mathsf{temp}(\mathsf{f}(\mathsf{i})) \end{array}$$

FIG. 6.13: *Dépendance de flot en entrée : avant et après transformation.*

- **Anti-dépendance en entrée** (figure 6.14). Une quelconque instruction S_l accède en lecture la zone mémoire $\mathsf{lhs}(\mathsf{f}(\mathsf{i}))$ avant que S_k n'y écrive. Après transformation, $\mathsf{lhs}(\mathsf{f}(\mathsf{i}))$ est toujours réécrite par S_k et l'anti-dépendance est inchangée.

$$\begin{array}{ll} S_l : & \qquad = \mathsf{lhs}(\ldots) \\ & \qquad a_{\text{in}} \\ S_k : & \mathsf{lhs}(\mathsf{f}(\mathsf{i})) = \mathsf{rhs}(\ldots) \end{array} \qquad\qquad \begin{array}{ll} S_l : & \qquad = \mathsf{lhs}(\ldots) \\ & \qquad a_{\text{in}} \\ S'_k : & \mathsf{temp}(\mathsf{f}(\mathsf{i})) = \mathsf{rhs}(\ldots) \\ & \qquad f_{\text{new}} \\ S_k : & \mathsf{lhs}(\mathsf{f}(\mathsf{i})) = \mathsf{temp}(\mathsf{f}(\mathsf{i})) \end{array}$$

FIG. 6.14: *Anti-dépendance en entrée : avant et après transformation.*

- **Dépendance de sortie en entrée** (figure 6.15). Une quelconque instruction S_l accède en écriture la zone mémoire $\mathsf{lhs}(\mathsf{f}(\mathsf{i}))$ avant que S_k n'y réécrive. Après transformation, $\mathsf{lhs}(\mathsf{f}(\mathsf{i}))$ est toujours réécrite par S_k et la dépendance de sortie est inchangée.

$$\begin{array}{ll} S_l : & \mathsf{lhs}(\ldots) = \\ & o_{\text{in}} \\ S_k : & \mathsf{lhs}(\mathsf{f}(\mathsf{i})) = \mathsf{rhs}(\ldots) \end{array} \qquad\qquad \begin{array}{ll} S_l : & \mathsf{lhs}(\ldots) = \\ & o_{\text{in}} \\ S'_k : & \mathsf{temp}(\mathsf{f}(\mathsf{i})) = \mathsf{rhs}(\ldots) \\ & \qquad f_{\text{new}} \\ S_k : & \mathsf{lhs}(\mathsf{f}(\mathsf{i})) = \mathsf{temp}(\mathsf{f}(\mathsf{i})) \end{array}$$

FIG. 6.15: *Dépendance de sortie en entrée : avant et après transformation.*

- **Dépendance de flot en sortie** (figure 6.16). S_k écrit la donnée $\mathsf{lhs}(\mathsf{f}(\mathsf{i}))$ avant qu'une quelconque instruction S_l ne la lise. L'accès à lhs dans S_l, noté $\mathsf{lhs}(\mathsf{g}(\mathsf{i}))$, a donc été remplacé par l'accès $\mathsf{temp}(\mathsf{g}(\mathsf{i}))$ au temporaire. Comme la donnée $\mathsf{temp}(\mathsf{f}(\mathsf{i}))$ est écrite par S'_k, la dépendance de flot est alors entre S'_k et S_l.

- **Anti-dépendance en sortie** (figure 6.17). Par hypothèse, cette dépendance ne peut pas concerner la fonction d'accès $\mathsf{f}(\mathsf{i})$ de la variable lhs. Elle concerne donc une donnée lue dans le membre droit de l'instruction S_k et ultérieurement réécrite par une quelconque instruction S_l. Après transformation, la donnée n'est lue que dans le membre droit de S'_k et l'anti-dépendance est entre S'_k et S_l.

- **Dépendance de sortie en sortie** (figure 6.18). Une quelconque instruction S_l accède en écriture la zone mémoire $\mathsf{lhs}(\mathsf{f}(\mathsf{i}))$ après que S_k y a écrit. Après transformation, $\mathsf{lhs}(\mathsf{f}(\mathsf{i}))$ est toujours écrit par S_k. La dépendance reste inchangée. Si la fonction d'accès $\mathsf{f}(\mathsf{i})$ au tableau lhs n'avait pas été supposée injective, nous aurions également pu avoir des auto-dépendances de sortie sur l'instruction S'_k.[4]

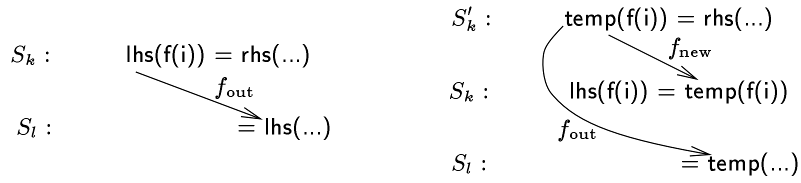


FIG. 6.16: *Dépendance de flot en sortie : avant et après transformation.*

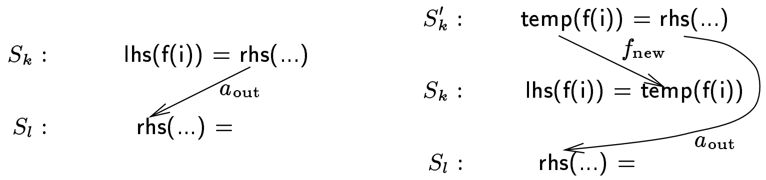


FIG. 6.17: *Anti-dépendance en sortie : avant et après transformation.*

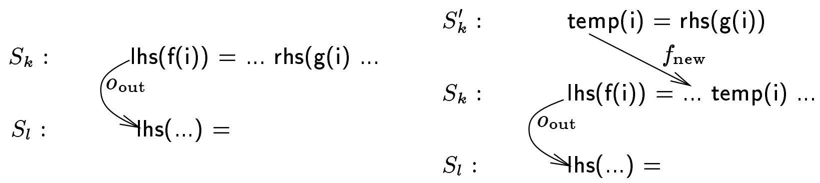


FIG. 6.18: *Dépendance de sortie en sortie : avant et après transformation.*

Cette étude de cas est résumée par la figure 6.19. Il en ressort que la seconde transformation est utile à l'élimination de chemins de dépendance si elle est appliquée à une instruction S cible d'une anti-dépendance ou d'une dépendance de sortie et source d'une dépendance de flot ou d'une anti-dépendance :

$$? \xrightarrow{a,o} S \xrightarrow{f,a} ?$$

Nous reviendrons sur la portée de cette transformation au paragraphe 6.2.4. Nous montrons d'abord comment lever la deuxième hypothèse (les fonctions d'accès en écriture aux tableaux ne sont pas l'objet de dépendances). Nous nous intéresserons ensuite à l'instruction de recopie : S_k : lhs(f(i)) = temp(f(i)).

Dépendances dues à des lectures en membre gauche

Nous avons précédemment supposé que les fonctions d'accès en écriture aux tableaux ne pouvaient être l'objet ni de dépendances de flot (en entrée) ni d'anti-dépendances (en sortie). Nous considérons maintenant ces deux types de dépendance.

Dans les deux cas, la dépendance concerne une donnée lue dans la fonction d'accès f(i) du tableau lhs présent dans le membre gauche de l'instruction S_k. Dans le cas d'une dépendance de flot en entrée (respectivement, d'une anti-dépendance en sortie) cette donnée a été précédemment produite (resp. sera ultérieurement réécrite) par une quelconque instruction S_l. Après transformation, la donnée

4. Ce problème ne serait pas apparu si nous avions décidé que l'affectation dans S'_k serait de la forme temp(i) =... au lieu de temp(f(i)) =....

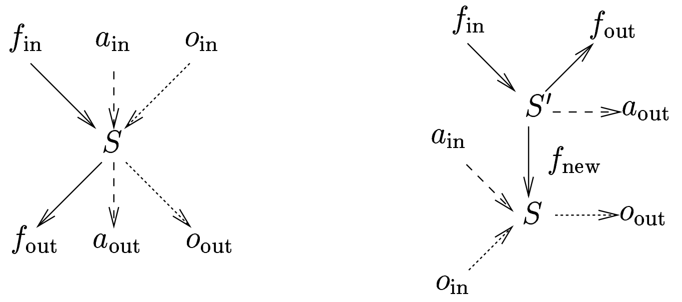


FIG. 6.19: *Dépendances avant et après application de la seconde transformation à l'instruction S.*

est utilisée pour calculer les accès des tableaux temp dans S'_k, et lhs et temp dans S_k. La dépendance initiale est remplacée par trois dépendances du même type : une entre S_l et S'_k et deux entre S_l et S_k. Les figures 6.20 et 6.21 illustrent les deux types de dépendance.

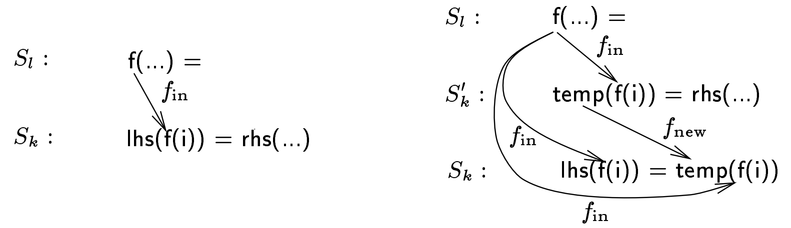


FIG. 6.20: *Dépendance de flot en entrée du membre gauche : avant et après transformation.*

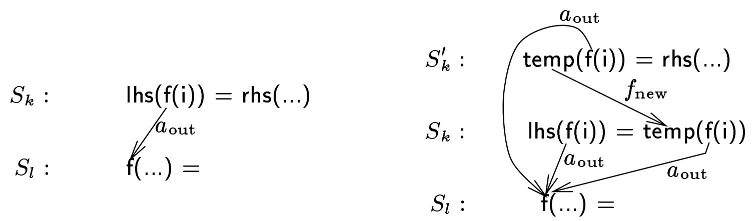


FIG. 6.21: *Anti-dépendance en sortie du membre gauche : avant et après transformation.*

Nous résumons à la figure 6.22 ces résultats avec ceux obtenus pour les autres types de dépendances (figure 6.19). Il pourrait sembler à première vue que, dans le cas général, la seconde transformation n'a d'intérêt que pour une instruction cible d'une anti-dépendance ou d'une dépendance de sortie et source d'une dépendance de flot, à cause des nouvelles anti-dépendances en sortie de S_k. Mais ces nouvelles dépendances ont une forme particulière : elles sont dues à des lectures en membre gauche. Nous pouvons les éliminer en appliquant à la fonction d'accès fautive une transformation similaire à la première transformation ou simplement en appliquant la première transformation à l'accès responsable de la dépendance et non à tout la fonction d'accès au tableau. Dans tous les cas, nous pouvons appliquer la première transformation avant ou après la deuxième, nous avons choisi sur l'exemple proposé à la figure 6.23 d'appliquer d'abord la première transformation. La figure 6.24 décrit l'ensemble des dépendances possibles avant et après la double transformation. Le lecteur peut vérifier que la double transformation proposée détruit les mêmes chemins de dépendances : $? \xrightarrow{a,o} S \xrightarrow{f,a} ?$. Elle permet aussi de casser les chemins de la forme $? \xrightarrow{f} S \xrightarrow{a} ?$ où la dépendance de flot atteint le membre droit de l'instruction originelle et où l'anti-dépendance a pour origine le membre gauche. Par la suite, nous appellerons « seconde transformation » la double transformation proposée ici.

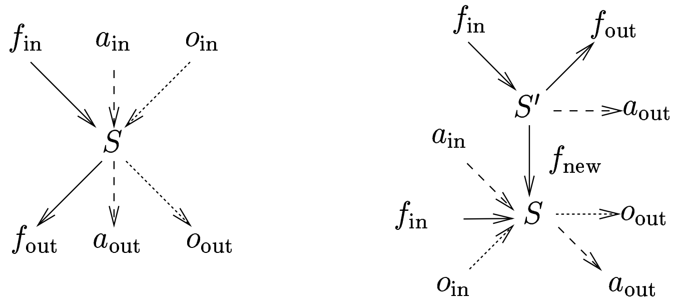


FIG. 6.22: *Dépendances avant et après application de la seconde transformation en présence de dépendances dues à des lectures en membre gauche.*

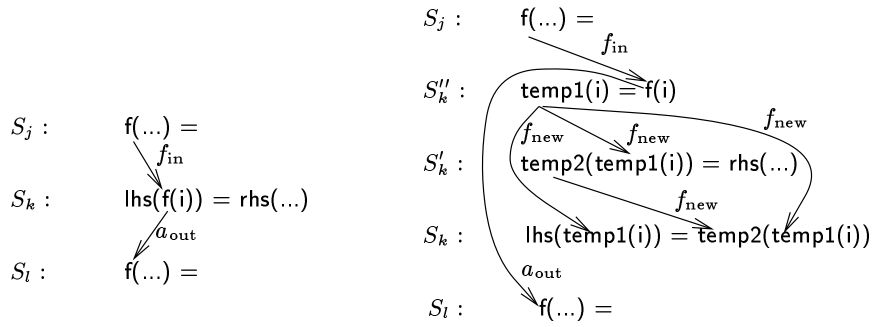


FIG. 6.23: *Dépendances avant et après application de la double transformation.*

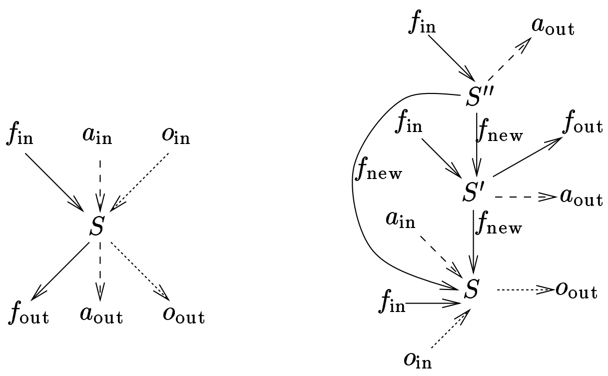


FIG. 6.24: *Dépendances avant et après application de la seconde transformation, cas général.*

Importance de l'instruction de recopie

Après transformation, l'instruction S_k est une simple recopie : lhs(f(i)) = temp(...). Apparemment, cette instruction n'est plus source d'aucune dépendance de flot (cf. figure 6.24), ne produit plus de valeurs utiles et peut être éliminée. Ce serait oublier les dépendances externes au nid que, volontairement, nous n'avons pas représentées. Si, effectivement, aucune des valeurs stockées dans le tableau lhs par S_k n'est utilisée, son exécution est une pure perte de temps et l'instruction peut être supprimée. La transformation devient alors un simple renommage. En revanche, si l'instruction S_k est source d'au moins une dépendance de flot, nous nous devons de la garder ou d'ajouter au programme des instructions équivalente de mise à jour du tableau lhs.

6.2.4 Efficacité des deux transformations

Efficacité de la seconde transformation

L'application de la seconde transformation à l'ensemble des instructions d'un nid de boucles détruit tous les circuits de dépendance, à l'exception de ceux uniquement formés de dépendances de flot et de ceux entièrement composés de dépendances de sortie, comme l'énonce le théorème 12.

Théorème 12. *Soit un nid de boucles L de graphe de dépendance réduit G. Nous supposons que les fonctions d'accès en écriture aux variables sont toutes injectives. Soit G' le graphe de dépendance réduit du nid obtenu par application de la seconde transformation à l'intégralité des instructions de L.*

Soit C un circuit de G'. Il est soit exclusivement constitué de dépendances de flot soit exclusivement constitué d'anti-dépendances. De plus, G contient un circuit équivalent à C.

DÉMONSTRATION. Supposons que G' contienne un circuit C et examinons un de ses arcs noté e. e peut *a priori* être une dépendance de flot, de sortie, ou une anti-dépendance :

- e est une **dépendance de sortie**. D'après la figure 6.24, l'arc e relie deux instructions de recopie $e \ : S_k \longrightarrow S_l$. Après transformation, l'instruction S_l n'est plus la source que de dépendances de sortie. L'arc de C qui suit e est donc également une dépendance de sortie et le circuit ne contient que ce type de dépendance. De plus, si G' contient une dépendance de sortie orientée de S_k vers S_l, il en va exactement de même de G.

- e est une **anti-dépendance**. Elle aboutit donc en une instruction de recopie S_l. Comme nous l'avons vu précédemment, une instruction de recopie n'est la source que de dépendances de sortie. Donc C contient une dépendance de sortie, l'arc qui suit e, et est donc un circuit uniquement composé de dépendances de sortie, ce qui est absurde.

- e est une **dépendance de flot**. C ne peut contenir de dépendances de sortie ou d'anti-dépendances, d'après les deux études de cas précédentes. C est donc uniquement constitué de dépendances de flot. La figure 6.24 montre qu'à toute dépendance de flot de G' aboutissant en un nœud S'_k (resp. S''_k) correspond une dépendance de flot de G aboutissant en S_k. De même, à toute dépendance de flot de G' originaire de S'_k correspond une dépendance de flot de G originaire de S_k. Il existe donc un circuit de G correspondant à G' et qui est exclusivement composé de dépendances de flot.

□

Par conséquent, le graphe de dépendance d'un nid qui ne comporte pas de circuits uniquement formés soit de dépendances de flot, soit de dépendances de sortie, est converti en un graphe sans circuits par application de la seconde transformation à l'ensemble de ses instructions.

Il n'est cependant pas forcément nécessaire de transformer toutes les instructions d'un nid pour éliminer un maximum de circuits. Nous reviendrons en détail sur ce point au paragraphe 6.2.5. Auparavant nous comparons les deux transformations.

Comparaison des deux transformations

La première transformation s'applique à chacun des accès en lecture de l'instruction originelle, chaque application nécessitant l'introduction d'un nouveau temporaire. Au contraire, la seconde transformation requiert au plus deux nouveaux temporaires par instruction : au lieu de stocker dans une nouvelle variable le résultat d'une seule lecture mémoire, toutes les lectures du membre droit sont effectuées et seul le résultat de l'évaluation de ce membre est stockée dans une nouvelle variable. Il en va de même pour les fonctions d'accès du membre gauche. Par conséquent, du point de vue du coût mémoire, la seconde transformation semble plus intéressante.

Nous avons vu que la seconde transformation était à même de détruire tout circuit de dépendances qui n'était ni un circuit de dépendances de flot, ni un circuit de dépendances de sortie. La première transformation n'est pas aussi puissante, comme le montre l'exemple de la figure 6.25. Le graphe de dépendance du nid originel contient les circuits :

$$S_1 \xrightarrow{f} S_2 \xrightarrow{f} S_3 \xrightarrow{a} S_1 \text{ et } S_1 \xrightarrow{f} S_2 \xrightarrow{o} S_1$$

L'application systématique de la première transformation aux instructions du nid (deuxième code de la figure 6.25) se révèle incapable d'éliminer les circuits, la transformation ne se justifiant en fait pour aucune instruction d'après l'étude du paragraphe 6.2.2. Au contraire, l'application de la seconde transformation à l'instruction S_1 rend le graphe de dépendance sans circuits (troisième code de la figure 6.25).

For i = 1 to N do	For i = 1 to N do	For i = 1 to N do
S_1 : a(i-1) = b(i)	S_1' : tmp1(i) = b(i)	S_1' : tmp(i-1) = b(i)
	S_1 : a(i-1) = tmp1(i)	S_1 : a(i-1) = tmp(i-1)
S_2 : a(i) = a(i-1)	S_2' : tmp2(i) = a(i-1)	S_2 : a(i) = tmp(i-1)
	S_2 : a(i) = tmp2(i)	
S_3 : c(i) = a(i)	S_3' : tmp3(i) = a(i)	S_3 : c(i) = a(i)
	S_3 : c(i) = tmp3(i)	

FIG. 6.25: *Nid originel, après la première transformation, et après la deuxième.*

La seconde transformation peut donc détruire tous les circuits de dépendances que la première peut supprimer, la réciproque étant fausse. Cependant, la première transformation n'est pas incluse dans la deuxième comme le montre l'exemple 21 : pour rendre le graphe de dépendance sans circuits la première transformation doit être appliquée à l'instruction S_2 et la deuxième à S_1. Dans la suite de cette section nous ne considérerons plus que la seconde transformation puisqu'elle est globalement moins gourmande en mémoire et plus puissante.

6.2.5 Minimisation du coût d'élimination des circuits

Nous nous intéressons ici au nombre minimal d'applications de la seconde transformation nécessaire à l'élimination de tous les circuits destructibles par cette transformation. Nous commençons par l'étude d'un exemple. Nous montrerons ensuite que le problème est NP-complet et nous proposerons deux heuristiques.

Exemple de référence

La figure 6.26 présente l'exemple 23 et son graphe de dépendance qui contient six dépendances : 3 dépendances de flot (une de S_3 vers S_2 due au tableau a, une de S_2 vers S_4 due au tableau b et

Exemple 23

For i = 4 to N do
  S_1 : a(i+5) = c(i-3) + b(2i+2)
  S_2 : b(2i) = a(i-1) + 1
  S_3 : a(i) = c(i+5) - 1
  S_4 : c(i) = b(2i-4)

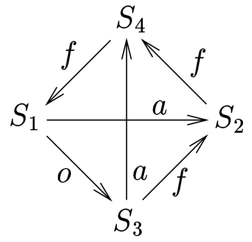


FIG. 6.26: *Exemple 23 et son graphe de dépendance* avant *transformation.*

une de S_4 vers S_1 due au tableau c), 2 anti dépendances (une de S_1 vers S_2 due au tableau b et une de S_3 vers S_4 due au tableau c) et 1 dépendance de sortie (de S_1 vers S_3 due au tableau a).

anti	S_1	b(2i+2)	$\longrightarrow$	S_2	b(2i)	
flot	S_1	a(i+5)	$\longrightarrow$	S_2	a(i-1)	[recouverte]
sortie	S_1	a(i+5)	$\longrightarrow$	S_3	a(i)	
flot	S_2	b(2i)	$\longrightarrow$	S_4	b(2i-4)	
anti	S_3	c(i+5)	$\longrightarrow$	S_4	c(i)	
flot	S_3	a(i)	$\longrightarrow$	S_2	a(i-1)	
flot	S_4	c(i)	$\longrightarrow$	S_1	c(i-3)	

TAB. 1: *Dépendances trouvées par Tiny.*

Le tableau 1 montre l'ensemble des dépendances trouvées par le logiciel Tiny pour cet exemple. En fait, Tiny trouve une septième dépendance mais reconnaît que cette dépendance est recouverte (killed) par d'autres. Nous aurions en effet pu trouver une dépendance de flot de S_1 vers S_2, a(i+5) étant écrit par $S_1(i)$ et lue par $S_2(i+6)$. Mais entre-temps, a(i+5) est réécrit par $S_3(i+5)$ et c'est uniquement le résultat de cette écriture qui est lu par $S_2(i+6)$. Par conséquent la dépendance de flot de S_1 vers S_2 est recouverte par la dépendance de sortie de S_1 vers S_3 suivie de la dépendance de flot de S_3 vers S_2. Il est ici capital que l'analyseur de dépendances sache reconnaître que la dépendance de flot de S_1 vers S_2 est recouverte, afin d'éviter la création artificielle d'un circuit de dépendances de flot !

D'après l'étude du paragraphe 6.2.3, il peut être intéressant de transformer les instructions S_2, S_3 et S_4. La figure 6.27 montre qu'il n'est pas nécessaire d'appliquer trois fois la transformation pour rendre le graphe de dépendance sans circuits : ici nous n'avons besoin que de toucher à S_2 et S_3. Nous avons introduit des conditionnelles pour réécrire le code, certaines lectures accédant à des valeurs produites tantôt par des instructions transformées, tantôt hors du nid. Nous aurions aussi bien pu dérouler partiellement le nid pour éliminer les cas pathologiques.

Nous avons pu rendre le graphe de dépendance sans circuits grâce à l'introduction de seulement deux nouveaux tableaux (là où une mise en assignation unique en eut requis quatre si nous considérons que les tableaux a, b et c sont accédés en écriture ailleurs dans le programme). Dans cet exemple, la transformation de trois des instructions semblait *a priori* intéressante mais en fait il nous a suffit de transformer deux de ces instructions pour obtenir un graphe de dépendance réduit sans circuits. Nous montrons maintenant que la minimisation du nombre d'instructions à transformer pour détruire tous les circuits qui ne sont ni des circuits de dépendance de flot, ni des circuits de dépendances de sortie, est un problème NP-complet.

For i = 4 to N do

S_1 : a(i+5) = c(i-3) + b(2i+2)

S'_2 : b-temp(2i) = $\begin{cases} \text{if i} \geq 5 \text{ then a-temp(i-1) + 1} \\ \text{else a(i-1) + 1} \end{cases}$

S_2 : b(2i) = b-temp(2i)

S'_3 : a-temp(i) = c(i+5) - 1

S_3 : a(i) = a-temp(i)

S_4 : c(i) = $\begin{cases} \text{if i} \geq 6 \text{ then b-temp(2i-4)} \\ \text{else b(2i-4)} \end{cases}$

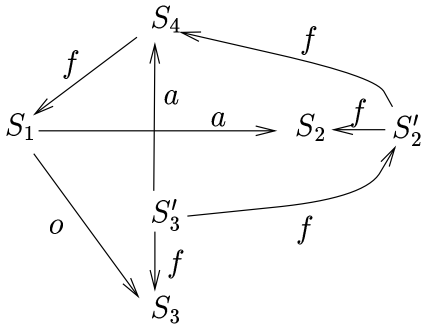


FIG. 6.27: *Exemple 23 et son graphe de dépendance* après *transformation.*

NP-complétude

Nous démontrons ici que déterminer le nombre minimal d'instructions à transformer est NP-difficile. Pour ce faire, nous définissons formellement notre problème et nous montrons que le problème de décision associé est NP-complet par une réduction du problème de satisfiabilité 3-SAT. Nous utiliserons dans la démonstration des nids ne contenant que des anti-dépendances. En dépit de cette hypothèse le problème reste NP-difficile.

Problématique. Soit $G = (V, E, \ell)$ le graphe de dépendance du nid L. Le type de la dépendance associée à un arc est donné par la fonction $\ell\ : E \longrightarrow \{f, a, o\}$ (dépendance de flot, anti-dépendance ou dépendance de sortie). Nous cherchons le nombre minimal d'instructions sur lesquelles appliquer la seconde transformation pour supprimer tous les circuits destructibles de la sorte. Pour pouvoir formuler le problème de décision associé nous avons besoin de quelques définitions.

Définition 17.

- *Étant donnés un nid de boucles L, son graphe de dépendance $G = (V, E, \ell)$ et un entier positif K, pouvons-nous trouver un ensemble d'au plus K instructions de L dont la transformation donnerait un nid dont le graphe de dépendance G' ne contiendrait que des circuits de dépendances de flot et des circuits de dépendances de sortie ? Si la réponse est positive, L appartient à* CIRCUITS-PURS(K).

- *Un nid de boucles L est dit* admissible *si et seulement si son graphe de dépendance $G = (V, E, \ell)$ ne contient que des anti-dépendances : $\forall e \in E,\ \ell(e) = a$.*

- *Étant donné un nid de boucles admissible L et un entier positif K, pouvons-nous trouver un ensemble d'au plus K instructions de L dont la transformation donnerait un nid de graphe de dépendance G' sans circuits ? Si la réponse est positive, L appartient à* NID-SANS-CIRCUITS*(K).*

Nous allons prouver que NID-SANS-CIRCUITS(K) est NP-complet au sens fort. Il en découlera que CIRCUITS-PURS(K) est aussi NP-complet au sens fort. Nous utilisons une transcription en terme de graphes de notre transformation. Dans la suite nous ne considérons en fait des graphes dont les arcs ne sont pas étiquetés a, f ou o, toutes les dépendances étant des anti-dépendances.

Définition 18. *Soit $G = (V, E)$ un graphe orienté. Transformer le nœud $S \in V$ équivaut*[5] *à créer un nouveau graphe $G' = (V', E')$ tel que :*

1. *G' contient un nouveau nœud S' en plus des nœuds de G : $V' = V \cup \{S'\}$;*
2. *E' contient un nouvel arc $e = (s', s)$;*
3. *Soit un arc $e = (u, v) \in E$:*
   (a) *Si $u \neq S$ alors $e \in E'$,*
   (b) *Si $u = S$, alors $e' = (S', v) \in E'$,*

Soit un chemin $R \to S \to T$. Si S est transformé, cet ensemble d'arcs devient $R \to S \leftarrow S' \to T$ et le chemin de R à T est détruit. La transformation de plusieurs nœuds du graphe originel peut se faire dans n'importe quel ordre. Finalement, nous définissons le problème de théorie des graphes :

Définition 19. *Étant donné un graphe G et un entier positif K, pouvons-nous trouver un ensemble d'au plus K nœuds de G dont la transformation (par la transformation de la définition 18) produirait un graphe G' sans circuits ? Si la réponse est positive, G appartient à* GRAPHE-SANS-CIRCUITS*(K).*

Nous allons préalablement démontrer que GRAPHE-SANS-CIRCUITS est un problème NP-complet par réduction du problème de satisfiabilité 3-SAT. Nous prouverons ensuite que NID-SANS-CIRCUITS possède la même propriété. Pour cela, nous prouvons que si G est un graphe il existe un nid de boucles admissible dont G est le graphe de dépendance.

Théorème 13. GRAPHE-SANS-CIRCUITS *est NP-complet.*

DÉMONSTRATION. Premièrement, GRAPHE-SANS-CIRCUITS appartient à NP : étant donnés un graphe $G = (V, E)$, un entier K et une liste de nœuds à transformer, nous pouvons vérifier en temps polynomial si le nouveau graphe G' est ou non sans circuits (la vérification se fait en temps linéaire $O(|V| + |E|)$ par simple traversée du graphe).

Nous procédons par réduction à partir du problème 3-SAT. Une instance du problème 3-SAT [44] est formée d'une expression booléenne B sous forme normale conjonctive :

$$B = \bigwedge_{i=1}^{t} C_i$$

- où $C_j = \ell_j^1 \vee \ell_j^2 \vee \ell_j^3$, $1 \leq j \leq t$, est une clause,
- et où chaque littéral ℓ_j^k est une variable de l'ensemble $X = \{x_1, \cdots, x_r\}$ ou sa négation.

La problème de décision associé est : y a-t-il une fonction de vérité $w \ : X \to \{\text{VRAI, FAUX}\}$ telle que B s'évalue à VRAI? Si la réponse est positive, B appartient à 3-SAT.

Nous utiliserons l'exemple ci-dessous tout au long de la preuve : $t = 3$, $r = 4$, et

$$B = (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4).$$

Étant donné B nous devons construire une instance de notre problème (un graphe $g(B) = (V, E)$ et un entier K) telle que l'appartenance de $g(B)$ à GRAPHE-SANS-CIRCUITS(K) soit équivalente à

5. Pour s'en convaincre, regarder la figure 6.19 en ne considérant que les anti-dépendances.

l'appartenance de B à 3-SAT. La fonction de construction g doit en outre être de coût polynomial en la taille de B, c'est-à-dire en le nombre de clauses et de variables.

Construction. Nous construisons un graphe $g(B)$ contenant $2 \times t \times r$ nœuds. Pour chaque variable x_i nous introduisons un motif W_i comprenant $2 \times t$ nœuds, étiquetés $V(i,j)$ et $F(i,j)$, pour j compris entre 1 et t (cf. figure 6.28). Intuitivement, pour connecter les motifs, nous utiliserons le nœud $V(i,j)$ ou le nœud $F(i,j)$ si la variable x_i apparaît dans la clause C_j : nous nous servirons de $F(i,j)$ si la variable x_i figure, niée, dans C_j et de $V(i,j)$ si elle s'y trouve mais n'est pas niée. Comme le montre la figure 6.28, le motif est un graphe biparti complet : il y a un arc de tout nœud $V(i,j)$ vers tout nœud $F(i,k)$, et réciproquement, pour $1 \leq j,k \leq t$. Un motif contient donc $2 \times t^2$ arcs.

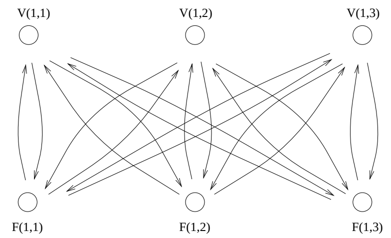


FIG. 6.28: *Le motif W_1.*

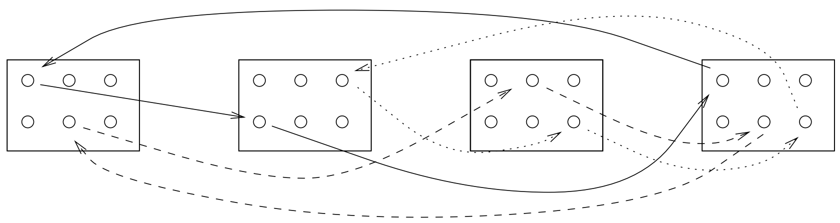


FIG. 6.29: *Connexions des motifs dictées par les clauses.*

Les motifs sont connectés entre eux suivant la composition des clauses. Considérons la clause C_1 de l'exemple : $C_1 = x_1 \vee \neg x_2 \vee x_4$. Nous relions les nœuds $T(1,1)$, $F(2,1)$ et $T(4,1)$ afin d'obtenir le circuit $T(1,1) \to F(2,1) \to T(4,1) \to T(1,1)$ (cf. figure 6.29). De même, comme $C_2 = \neg x_1 \vee x_3 \vee \neg x_4$, nous relions $F(1,2)$, $T(3,2)$ et $F(4,2)$ pour former un deuxième circuit. De la sorte, nous introduisons 3 nouveaux arcs pour chaque clause, ce qui nous donne un total de $2 \times r \times t^2 + 3 \times t$ arcs dans le graphe $g(B)$ qui est donc bien de taille polynomiale en r et t.

Nous posons finalement, $K = r \times t$. Autrement dit nous nous demandons s'il est possible de rendre le graphe sans circuits par transformation de la moitié de ses nœuds.

Équivalence. Nous prouvons maintenant l'équivalence : $g(B) \in$GRAPHE-SANS-CIRCUITS$(K) \Leftrightarrow B \in$ 3-SAT.

Supposons initialement que B appartienne à 3-SAT. Soit $w\ : X \to \{$VRAI, FAUX$\}$ la fonction de vérité permettant à B de s'évaluer à VRAI. Si w assigne la valeur VRAI à la variable x_i ($w(x_i) =$ VRAI), nous transformons les t nœuds $V(i,j), 1 \leq j \leq t$. Sinon, nous transformons les t nœuds $F(i,j), 1 \leq j \leq t$. Nous transformons donc au total $K = r \times t$ nœuds.

Transformer tous les nœuds $V(i,*)$ ou tous les nœuds $F(i,*)$ permet de détruire tout circuit interne au motif W_i. En effet, le graphe étant biparti, tout circuit interne traverse au moins un des nœuds $V(i,*)$ et un des nœuds $F(i,*)$.

Il nous reste à montrer que tous les circuits correspondant aux clauses sont également éliminés.

Comme B appartient à 3-SAT, chaque clause C_j s'évalue à VRAI sous w. Il existe donc au moins une variable x_i présente dans C_j dont la valeur déterminée par w met C_j à VRAI. Si x_i apparaît niée, $w(x_i) =$ FAUX et nous avons transformé le nœud $F(i,j)$. Comme le circuit associé à la clause C_j emprunte par construction ce nœud, il a bien été éliminé. Le raisonnement est identique si x_i n'est pas niée. Par conséquent, G' est sans circuits et $g(B) \in$ GRAPHE-SANS-CIRCUITS(K).

Réciproquement, soit $g(B)$ appartenant à GRAPHE-SANS-CIRCUITS(K). Autrement dit, nous supposons qu'il est possible de rendre $g(B)$ sans circuits en transformant moins de $K = r \times t$ de ses nœuds. Nous devons construire une fonction de vérité w qui permette d'évaluer B à VRAI.

Considérons un motif W_i. Comme G', le graphe obtenu par transformation des nœuds, est sans circuits, tous les nœuds $V(i,j), 1 \leq j \leq t$, ou tous les nœuds $F(i,j), 1 \leq j \leq t$, ont été transformés. Sinon, il resterait un nœud $V(i,j_0)$ et un nœud $F(i,j_1)$ intacts et G' contiendrait le circuit : $V(i,j_0) \rightarrow F(i,j_1) \rightarrow V(i,j_0)$. Par conséquent, puisque $K = r \times t$ nœuds ont été transformés et que t nœuds de chacun des r motifs ont dû être transformés, exactement t nœuds du motif W_i ont été transformés : tous les nœuds $V(i,*)$ ou tous les nœuds $F(i,*)$. Nous en déduisons la fonction de vérité w en posant $w(x_i) =$ VRAI si ce sont les nœuds $V(i,*)$ qui ont été transformés, et $w(x_i) =$ FAUX sinon. Il reste à montrer que B s'évalue à VRAI pour la fonction de vérité w.

Soit une clause C_j. Le circuit de $g(B)$ qui lui correspond a été détruit. Par conséquent, au moins l'un de ses trois nœuds a été transformé. Notons x_{i_0} la variable associée à l'un des nœuds modifiés. Ce nœud est soit $F(i_0,j)$ soit $V(i_0,j)$, suivant que x_{i_0} apparaissait dans C_j niée ou non. Si $V(i_0,j)$ a été transformé, alors $w(x_i) =$ VRAI, sinon $F(i_0,j)$ l'a été et $w(x_i) =$ FAUX. Par conséquent, C_j prend la valeur VRAI pour w et il en va de même de B. Donc B appartient à 3-SAT. □

Théorème 14. NID-SANS-CIRCUITS *est NP-complet.*

Pour démontrer la NP-complétude de NID-SANS-CIRCUITS il nous suffit de montrer que tout graphe est le graphe de dépendance réduit d'un nid de boucles admissible.

DÉMONSTRATION. Avant tout, NID-SANS-CIRCUITS appartient à NP. Soit un nid de boucles admissible L et son graphe de dépendance $G = (V,E)$. Les nœuds à transformer étant donnés, nous pouvons vérifier en temps polynomial si le graphe est ou non sans circuits après transformation par une simple traversée du graphe.

Nous utilisons une réduction à partir de GRAPHE-SANS-CIRCUITS. Pour tout graphe $G = (V,E)$, nous construisons un nid de boucles admissible admettant G comme graphe de dépendance. Toutes les arcs de G doivent correspondre à des anti-dépendances du nid L. Nous associons à chaque nœud $v \in V$ un tableau monodimensionnel *tab.v* de taille 100, par exemple. Le nid L est constitué d'une seule boucle entourant $|V|$ instructions. Pour chaque nœud v nous construisons une instruction $S(v)$ de la forme « $tab.v(i) = \ldots$ ». Pour chaque arc $e = (u,v) \in E$ nous produisons une anti-dépendance de S_u vers S_v en plaçant une référence $tab.v(i+1)$ dans le membre droit de l'instruction S_u, comme suit :

```
For i = 1 to 99 do
  Su : tab.u(i) = ... + tab.v(i+1) + ...
  ...
  Sv : tab.v(i) = ...
```

Le nid de boucles L est notoirement admissible, son graphe de dépendance ne contenant pas de dépendances de flot ni de sortie. La construction du nid est de coût polynomial en la taille de G. D'où le résultat. □

Corollaire 7. CIRCUITS-PURS *est un problème NP-complet.*

Heuristiques

Nous présentons ici deux heuristiques pour déterminer quelles instructions doivent être transformées afin de détruire tous les circuits de dépendances éliminables grâce à la deuxième transformation. La première heuristique est de coût exponentiel dans le pire cas mais peut être néanmoins intéressante pour de petits graphes de dépendances. La deuxième s'exécute en temps polynomial : $O(t^2(|V|+|E|))$, où t est le nombre d'instructions à transformer, d'où un pire cas en $O(|V|^2(|V|+|E|))$.

Heuristique basée sur l'hypergraphe des circuits. L'heuristique la plus naturelle consiste sans doute à raisonner sur l'hypergraphe $H=(V,F)$ des circuits du graphe de dépendance G. L'ensemble F des arcs de l'hypergraphe est défini comme une collection de sous-ensembles f de V. Un « arc » de H est le sous-ensemble f des nœuds visités par un circuit élémentaire C de G. La figure 6.30 présente l'hypergraphe de notre exemple de référence. Un nœud v de f est marqué *cassable* pour C si le circuit C est détruit par transformation de v. Le nœud v est, entre autres, marqué *cassable* s'il est la cible dans C d'une anti-dépendance ou d'une dépendance de sortie et s'il est la source dans C d'une anti-dépendance ou d'une dépendance de flot (cf. figure 6.24).

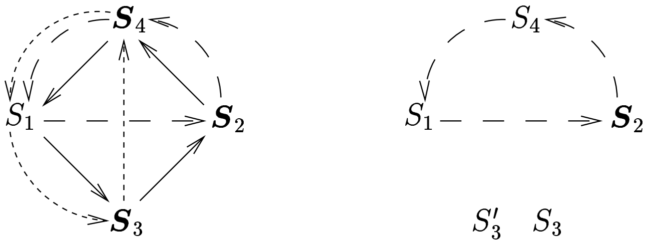


FIG. 6.30: *Hypergraphe de l'exemple 23, avant et après transformation de S_3.*

La recherche de l'ensemble minimal de sommets à casser est un problème de recherche de recouvrement minimal d'un hypergraphe, problème pour lequel il n'y a pas d'algorithme polynomial connu [36].

Une fois l'hypergraphe construit, nous lui appliquons une approche gloutonne : nous transformons le nœud v_0 qui est *cassable* pour le plus grand nombre de sous-ensembles $f \in F$; tous les circuits pour lesquels v_0 était cassable sont alors éliminés de l'hypergraphe et l'opération est répétée tant qu'il demeure des nœuds cassables. Cette heuristique peut être améliorée en transformant prioritairement à chaque étape tout nœud qui serait le seul nœud cassable d'un circuit.

Le principal inconvénient de cette heuristique, c'est le coût de son pire cas. En effet, les circuits élémentaires du graphe peuvent être en nombre exponentiel en la taille du graphe et rendre prohibitive la construction de l'hypergraphe.

Application de l'heuristique à l'exemple de référence. L'exemple et son graphe de dépendance sont présentés à la figure 6.26. Son hypergraphe des circuits contient 3 circuits comme le montre la figure 6.30 où les différents circuits sont symbolisés par des arcs de corps différents et où les nœuds cassables sont en gras. Le tableau ci-dessous récapitule pour chaque nœud le nombre de circuits qui le contiennent et pour lesquels il est *cassable*.

S_1	S_2	S_3	S_4
0	1	2	1

L'heuristique transforme d'abord l'instruction S_3. L'hypergraphe du nid après transformation est présenté à la figure 6.30. Comme il ne contient qu'un seul nœud cassable, S_2, l'heuristique le transforme et se termine. Nous obtenons le graphe présenté à la figure 6.27.

Heuristique de coût polynomial. La transformation d'une instruction est intéressante si l'instruction est cible d'une anti-dépendance ou d'une dépendance de sortie et source d'une anti-dépendance ou dépendance de flot, ou si le membre droit de l'instruction est cible d'une dépendance de flot et le membre gauche source d'une anti-dépendance. À chaque nœud v nous associons le nombre $Util(v)$ de couples (arc-entrant, arc-sortant) qui rendent intéressante la transformation de v (nous ne nous préoccupons pas de savoir si un tel couple appartient ou non à un circuit).

Nous transformons un nœud v pour lequel $Util(v)$ est maximal. Nous obtenons un nouveau graphe G' dont nous enlevons tous les arcs qui ne participent pas à une composante fortement connexe. Nous appliquons récursivement l'heuristique au graphe G' s'il contient un nœud v pour lequel $Util(v)$ est non nul.

Les composantes fortement connexes de G' peuvent être construites en $O(|V| + |E|)$. Une fois un nœud transformé, nous appliquons récursivement l'heuristique si une composante fortement connexe contient une anti-dépendance ou un circuit élémentaire contenant une dépendance de flot et une dépendance de sortie. Partant d'un nœud v cible d'une dépendance de sortie et source d'une dépendance de flot, nous pouvons tester l'appartenance de v à un circuit élémentaire contenant une dépendance de flot et une dépendance de sortie en une traversée « intelligente » du graphe, soit en $O(|V| + |E|)$. Le graphe contenant au plus $|V|$ nœuds, tester si l'heuristique doit encore être appliquée se fait en $O(|V|(|V| + |E|))$. Il y a autant d'appel à l'heuristique que de nœuds transformés. Dans le pire cas tous les nœuds seront transformés et l'heuristique est de complexité $O(|V|^2(|V| + |E|))$.

Application de l'heuristique à l'exemple de référence. L'exemple et son graphe de dépendance sont décrits figure 6.26. Le tableau ci-dessous présente les valeurs de la fonction *Util* pour les différents nœuds.

	S_1	S_2	S_3	S_4
$Util(v)$	0	1	2	1

Une fois de plus, nous commençons par transformer l'instruction S_3. Le nouveau graphe de dépendance a une unique composante fortement connexe. Nous lui appliquons récursivement l'heuristique, la fonction *Util* prenant maintenant les valeurs :

	S_1	S_2	S_4
$Util(v)$	0	1	0

Ce qui nous amène à transformer derechef l'instruction S_2 et à trouver le même résultat que précédemment.

6.2.6 Conclusion

Nous avons étudié deux transformations de corps de nids de boucles qui s'apparentent à un renommage de tableau. Nous avons montré leur efficacité et leurs limitations pour la suppression de circuits de dépendances. En pratique, ces transformations ne seront pas utilisées seules mais avec d'autres méthodes d'élimination de fausses dépendances ou de faux chemins de dépendance, en vue d'améliorer la parallélisation (cf. la section suivante). Le résultat de NP-complétude est donc beaucoup moins important que la caractérisation exacte des chemins de dépendances cassables par ces transformations.

6.3 Élimination de fausses dépendances et détection du parallélisme

Tout un ensemble de techniques d'élimination de fausses dépendances a été proposé, comme le renommage de scalaires [65] ou de tableaux (cf. section 6.2), l'expansion de scalaires [16] ou de tableaux [37], la privatisation [47] ou la mise en assignation unique [38]. Pour une description de ces techniques, nous renvoyons le lecteur à l'état de l'art de Banerjee, Eigenmann, Nicolau et Padua [8], à celui de Bacon, Graham et Sharp [6] et aux livres de Wolfe [82], et Zima et Chapman [84].

Nous proposons ici un schéma d'intégration de ces techniques d'élimination des dépendances au sein des algorithmes de détection du parallélisme. Nous commençons par motiver cette approche au paragraphe 6.3.1. Nous formaliserons ensuite l'intégration (paragraphe 6.3.2) avant de considérer son application à Allen-Kennedy (paragraphe 6.3.3), à Wolf-Lam (paragraphe 6.3.4), puis à Ordonnanceur_de_gdrp (paragraphe 6.3.5). Dans la conclusion, au paragraphe 6.3.6, nous reviendrons sur les travaux réalisés dans ce domaine par d'autres chercheurs.

6.3.1 Motivation

Nous étudions sur un exemple l'interaction entre les techniques d'élimination de dépendance et celles de détection du parallélisme. Nous présentons trois schémas différents de parallélisation d'un exemple au moyen de l'algorithme d'Allen et Kennedy : l'application directe de Allen-Kennedy, l'application après mise en assignation unique et l'intégration des techniques d'élimination au sein même de Allen-Kennedy.

Exemple de référence

La figure 6.31 présente l'exemple qui nous servira tout au long de ce paragraphe de motivation. Comme nous allons le paralléliser au moyen de l'algorithme d'Allen et Kennedy (cf. section 2.2) nous utiliserons une approximation des dépendances par niveaux.

Exemple 24

```
For i = 1 to N do
  For j = 1 to N do
    For k = 1 to N do
      S1 : a(i,j,k) = a(i,j,k+1) + c(i,j+1,k) + c(i,j-1,k) + b
      S2 : b = a(i-1,j,k) + a(i,j-1,k)
      S3 : c(i,j,k) = c(i+1, j, k) + c(i,j,k+1)
```

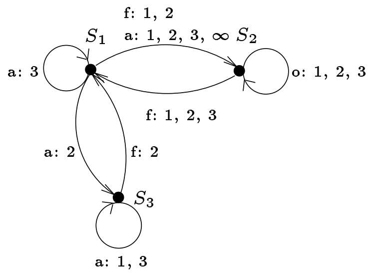
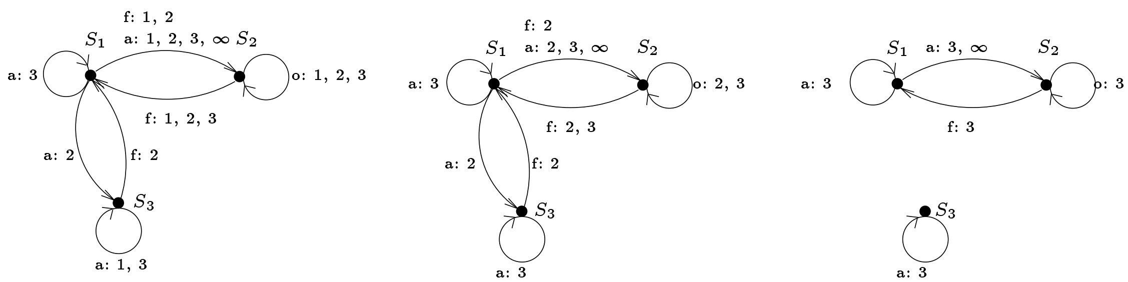


Fig. 6.31: *Exemple 24 et son graphe de dépendance par niveaux.*

Parallélisation classique

La parallélisation classique est l'application directe de l'algorithme Allen-Kennedy sur le graphe de dépendance de l'exemple étudié.

Le graphe de dépendance réduit par niveaux est rappelé à la figure 6.32. Il a été établi à partir de l'analyse fournie par le logiciel Tiny [81]. Certaines de ces dépendances ne sont pas des dépendances directes. Elles sont recouvertes pas d'autres dépendances et nous pourrions donc les omettre. Il n'y a par exemple d'anti-dépendance de S_1 vers S_2 due au scalaire b qu'au niveau ∞, les dépendances aux niveaux 1, 2 et 3 étant recouvertes.



(a) (b) (c)

FIG. 6.32: *GDRN: originel (a), sans dépendances de niveau 1 (b), ni de niveau 2 (c).*

Ce graphe de dépendance réduit par niveaux contient une unique composante fortement connexe incluant les trois instructions et des dépendances de niveau 1. Par conséquent, la boucle externe, sur i, est marquée « séquentielle » par ALLEN-KENNEDY. Après élimination des arcs de niveau 1, toutes les instructions font toujours partie d'une même composante fortement connexe qui inclut des dépendances de niveau 2. Par conséquent la deuxième boucle, sur j, est marquée « séquentielle ». Une fois supprimés les arcs de niveau 2, il demeure deux composantes fortement connexes, chacune comprenant une dépendance de niveau 3. La troisième boucle, sur k, est donc elle aussi marquée « séquentielle » pour toutes les instructions. En résumé, ALLEN-KENNEDY ne trouve pas de parallélisme dans notre exemple quand les dépendances de sortie et autre anti-dépendances sont prises en compte. Nous montrons maintenant que l'exemple 24 contient en fait du parallélisme dont l'exposition nécessite l'élimination de fausses dépendances.

Parallélisation après mise en assignation unique

L'autre solution, extrême du point de vue des fausses dépendances, consiste à mettre dans un premier temps le programme en assignation unique, éliminant de la sorte toutes les anti-dépendances et les dépendances de sortie, puis à le paralléliser. La figure 6.33 montre l'exemple 24 mis en assignation unique (en supposant que le paramètre N est plus grand que 1). En toute rigueur, il faudrait aussi recopier atemp(i,j,k) dans a(i,j,k), ctemp(i,j,k) dans c(i,j,k) et btemp(N,N,N) dans b après exécution du nid.

```
For i = 1 to N do
  For j = 1 to N do
    For k = 1 to N do
      S1 : atemp(i,j,k) = a(i,j,k+1) + c(i,j+1,k)
                          + if j≥ 2 then ctemp(i,j-1,k) else c(i,0,k)
                          + if k≥ 2 then btemp(i, j, k-1)
                                    else if j≥ 2 then btemp(i, j-1, N)
                                              else if i≥ 2 then btemp(i-1, N, N)
                                                        else b
      S2 : btemp(i,j,k) = if i≥ 2 then atemp(i-1, j, k) else a(0, j, k)
                          + if j≥ 2 then atemp(i, j-1, k) else a(i, 0, k)
      S3 : ctemp(i, j, k) = c(i+1, j, k) + c(i,j,k+1)
```

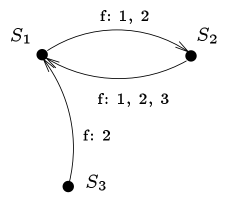


FIG. 6.33: *Exemple 24 mis en assignation unique et son graphe de dépendance.*

Les deux boucles les plus externes (les boucles sur i et sur j) entourant les instructions S_1 et S_2 seront marquées « séquentielles » par ALLEN-KENNEDY. Toutes les autres boucles seront considérées parallèles, ce que transcrit la figure 6.34.

```
ForPar i = 1 to N do
  ForPar j = 1 to N do
    ForPar k = 1 to N do
      S3 : ctemp(i, j, k) = c(i+1, j, k) + c(i,j,k+1)
ForSeq i = 1 to N do
  ForSeq j = 1 to N do
    ForPar k = 1 to N do
      S2 : btemp(i,j,k) = if i≥ 2 then atemp(i-1, j, k) else a(0, j, k)
                        + if j≥ 2 then atemp(i, j-1, k) else a(i, 0, k)
    ForPar k = 1 to N do
      S1 : atemp(i,j,k) = a(i,j,k+1) + c(i,j+1,k)
                        + if j≥ 2 then ctemp(i,j-1,k) else c(i,0,k)
                        + if k≥ 2 then btemp(i, j, k-1)
                                  else if j≥ 2 then btemp(i, j-1, N)
                                            else if i≥ 2 then btemp(i-1, N, N)
                                                      else b
```

FIG. 6.34: *Exemple 24 après mise en assignation unique puis parallélisation.*

Le programme obtenu à une latence en $\Theta(N^2)$ au lieu de $\Theta(N^3)$ pour le programme obtenu après parallélisation directe. Notre exemple contient donc bien du parallélisme qui était masqué par les fausses dépendances. Afin d'exposer le maximum de parallélisme, nous avons dû introduire trois nouveaux tableaux de taille N^3 (la taille du domaine d'itération). Nous montrons maintenant comment l'intégration des techniques d'élimination au sein de l'algorithme de parallélisation permet d'exhiber le maximum de parallélisme à un moindre coût mémoire.

Intégration des techniques d'élimination au sein de l'algorithme de parallélisation

Le graphe de flot de données (le graphe de dépendance réduit par niveaux où seules les dépendances de flot sont conservées, figure 6.35(a)) nous indique exactement le parallélisme contenu dans le programme. Nous voulons exhiber tout ce parallélisme en introduisant le minimum de mémoire.

Première boucle. À cause des dépendances de flot, la boucle la plus externe (la boucle sur i) est obligatoirement séquentielle pour les instructions S_1 et S_2. Au contraire, elle peut être rendue parallèle pour l'instruction S_3. Pour ce faire, il faut que S_3 n'appartienne plus à aucune composante fortement connexe contenant des dépendances de niveau 1. Par conséquence, nous devons éliminer deux fausses dépendances *incompatibles* avec la parallélisation (cf. figure 6.35(b)) :

- l'anti-dépendance de S_1 vers S_3. Cette dépendance et la dépendance de flot de S_3 vers S_1 raccorde S_1 à la composante fortement connexe de S_1 et S_2 qui inclut obligatoirement des dépendances de niveau 1.
- l'auto-anti-dépendance de S_3 de niveau 1.

Il est à noter qu'il n'est aucunement nécessaire de toucher à l'anti-dépendance de niveau 1 de S_1 vers S_2 ou à l'auto-dépendance de sortie de S_2 de niveau 1.

Les deux arcs *incompatibles* peuvent être supprimés par application sur S_3 de la « seconde transformation » (cf. paragraphe 6.2.3) qui n'introduit qu'un seul nouveau tableau tridimensionnel. Le nouveau graphe de dépendance réduit par niveaux est présenté figure 6.35(c). La première étape de ALLEN-KENNEDY nous fournit alors le programme suivant :

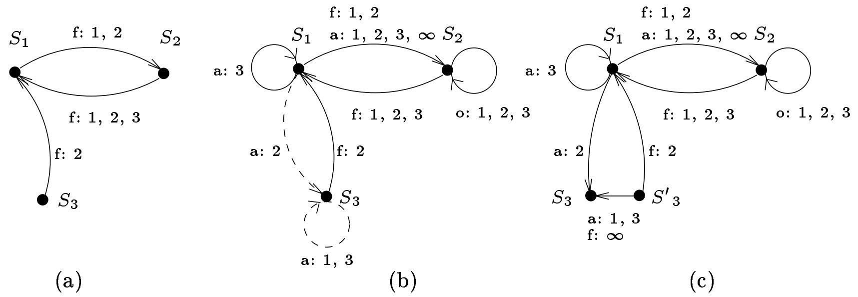


FIG. 6.35: *Exemple 24 : a) graphe de flot de données par niveaux; b) GDRN avec arcs incompatibles en pointillés; c) GDRN après transformation de S_3.*

```
ForPar i = 1 to N do
  ForPar j = 1 to N do
    ForPar k = 1 to N do
      S'3 : ctemp(i,j,k) = c(i+1, j, k) + c(i,j,k+1)
ForSeq i = 1 to N do
  For j = 1 to N do
    For k = 1 to N do
      S1 : a(i,j,k) = a(i,j,k+1) + b + c(i,j+1,k)
                    + if j≥ 2 then ctemp(i,j-1,k) else c(i,0,k)
      S2 : b = a(i-1,j,k) + a(i,j-1,k)
ForPar i = 1 to N do
  ForPar j = 1 to N do
    ForPar k = 1 to N do
      S3 : c(i,j,k) = ctemp(i,j,k)
```

Deuxième boucle. Nous ne nous occupons plus que de S_1 et S_2, le code correspondant à S_3 ayant déjà été intégralement généré. La figure 6.36 récapitule l'ensemble des dépendances non encore satisfaites.

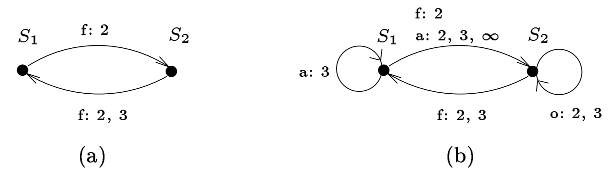


FIG. 6.36: *Graphe de flot de données (a) et GDRN (b) après la première étape de parallélisation.*

La deuxième boucle englobant S_1 et S_2 est rendue séquentielle par les dépendances de flot de niveaux 2. L'élimination de l'anti-dépendance de niveau 2 de S_1 vers S_2 est donc inutile car elle ne nous permettrait pas de trouver plus de parallélisme. La deuxième boucle est donc marquée « séquentielle » sans autre forme de procès.

Troisième boucle. Le graphe de flot de données de la figure 6.37(a) nous montre que la troisième boucle peut être rendue parallèle puisqu'il ne contient pas de circuit de niveau 3. Pour pouvoir exhiber ce parallélisme, il faut que S_1 et S_2 n'appartiennent plus à aucune composante fortement connexe contenant des dépendances de niveau 3. Nous devons donc supprimer les trois fausses dépendances incompatibles, en pointillés sur la figure 6.37(b) :

- l'auto dépendance de sortie de S_2;
- l'anti-dépendance de S_1 vers S_2;
- l'auto anti-dépendance de S_1.

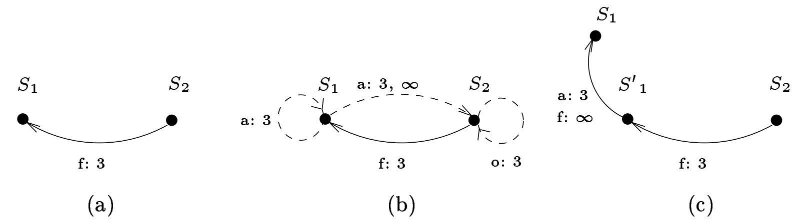


FIG. 6.37: *Après la deuxième étape de parallélisation : (a) graphe de flot de données, (b) GDRN aux arcs incompatibles en pointillés, (c) GDRN après transformation.*

Les deux premières dépendances peuvent être éliminées par expansion du scalaire b. La troisième est détruite par application de la seconde transformation à S_1 (cf. paragraphe 6.2.3). Nous n'avons cependant pas besoin d'introduire deux tableaux de tridimensionnels. Nous pouvons nous contenter de tableaux linéaires de taille N puisque que les deux boucles externes englobant S_1 et S_2 sont séquentielles. Nous n'avons en effet besoin d'éliminer les dépendances incompatibles que *pour chaque itération des boucles externes*. Nous procédons comme si les indices correspondant aux boucles externes étaient constants. Nous obtenons ainsi le code :

```
ForSeq i = 1 to N do
  ForSeq j = 1 to N do
    btemp(0) = b
    For k = 1 to N do
      S'1 : atemp(k) = a(i,j,k+1) + btemp(k-1) + c(i,j+1,k)
                       + if j≥ 2 then ctemp(i,j-1,k) else c(i,0,k)
      S1 : a(i,j,k) = atemp(k)
      S2 : btemp(k) = a(i-1,j,k) + a(i,j-1,k)
    b = btemp(N)
```

dont le graphe de dépendance réduit par niveaux au niveau 3 est représenté à la figure 6.37(c). Nous appliquons la troisième et dernière étape de ALLEN-KENNEDY pour obtenir finalement :

```
ForSeq i = 1 to N do
  ForSeq j = 1 to N do
    btemp(0) = b
    ForPar k = 1 to N do
      S2 : btemp(k) = a(i-1,j,k) + a(i,j-1,k)
    ForPar k = 1 to N do
      S'1 : atemp(k) = a(i,j,k+1) + btemp(k-1) + c(i,j+1,k)
                       + if j≥ 2 then ctemp(i,j-1,k) else c(i,0,k)
    ForPar k = 1 to N do
      S1 : a(i,j,k) = atemp(k)
    b = btemp(N)
```

L'intégration des techniques d'élimination au sein de ALLEN-KENNEDY nous a permis de trouver autant de parallélisme à un moindre coût mémoire (un tableau tridimensionnel et deux tableaux linéaires au lieu de trois tableaux tridimensionnels). Nous allons maintenant tenter de formaliser cette approche.

6.3.2 Intégration des techniques d'élimination dans les algorithmes d'ordonnancement

Nous proposons ici un schéma général d'insertion des techniques d'élimination de fausses dépendances et de faux chemins de dépendances au sein des algorithmes d'ordonnancement, dans le but d'exhiber le maximum de parallélisme tout en réduisant la quantité de mémoire introduite. Nous présentons une description théorique du procédé et nous discuterons brièvement ses détails. Dans les paragraphes suivants nous étudierons son application à certains algorithmes d'ordonnancement.

Schéma d'insertion

Nous supposons que l'algorithme d'ordonnancement utilisé possède les propriétés suivantes :

- Chaque instruction S entourée de n_S boucles dans le nid originel est également entourée de n_S boucles après parallélisation.
- Les boucles englobant une instruction sont générées en commençant par la plus externe et en terminant par la plus interne.

Ces hypothèses sont peu restrictives puisqu'elles sont vérifiées par tous les algorithmes d'ordonnancement présentés aux chapitres 2 et 4.

Nous notons G le graphe de dépendance réduit du nid considéré. Le schéma est le suivant.

INTÉGRATION(G)

- Déterminer l'ensemble des fausses dépendances et faux chemins de dépendance supprimables au moyen des techniques d'élimination et les détruire. Soit F le graphe de dépendance ainsi obtenu. F expose autant de parallélisme que faire se peut.
- « Appliquer » l'algorithme de parallélisation à F. Soit PP le programme obtenu. Chaque instruction S est entourée dans PP par n_S boucles, chacune étant marquée soit séquentielle, soit parallèle.
- **Pour** $d = 1$ à $d = \max_S(n_S)$ **faire**
  1. Marquer *incompatible* toute dépendance de G n'appartenant pas à F et qui induirait une perte de parallélisme si elle n'était pas éliminée à l'étape d, PP étant pris comme référence. Autrement dit, les dépendances incompatibles sont celles qui interdisent de générer autant de boucles parallèles que dans PP, parmi les $n_S - d + 1$ boucles restantes.
  2. Éliminer toutes les dépendances incompatibles en introduisant des tableaux de dimension aussi petite que possible.
  3. Générer la boucle.

Discussion

L'algorithme présenté ci-dessus est juste un schéma. Nous n'essayerons pas de le préciser d'avantage. Son imprécision est une conséquence de sa généralité. Il pose entre autres le problème de la caractérisation des arcs incompatibles, de la minimisation du nombre d'arcs à éliminer, de la minimisation de la dimension des tableaux introduits. L'exemple traité au paragraphe 6.3.1 donne des indications de réponses à ces questions. Nous revenons en détail ci-dessous sur le problème de la taille des tableaux. Nous étudierons dans les paragraphes suivants le problème de détermination des arcs incompatibles pour les algorithmes ALLEN-KENNEDY, WOLF-LAM et ORDONNANCEUR_DE_GDRP. Nous proposons cependant une définition d'un « ensemble de dépendance incompatible ».

Définition 20 (Ensemble de dépendances incompatible).
Lors de la génération de la $d^{ème}$ boucle par l'algorithme de parallélisation, un ensemble de dépendances est dit incompatible *s'il suffit de supprimer tous ses éléments à l'étape d pour ne pas perdre de parallélisme,* PP *étant pris comme référence.*

Intuitivement, un ensemble incompatible est un ensemble de dépendances qui contient les dépendances incompatibles. La taille de l'ensemble définit une borne supérieure du nombre de transformations à appliquer au programme et nous donne donc une indication de la mémoire requise. Désirant exposer au moindre coût tout le parallélisme contenu dans le nid étudié, nous nous intéressons donc aux ensembles incompatibles de taille minimale. Un ensemble de dépendances incompatible peut aussi bien contenir des arcs que des chemins de dépendance.

Dimensions des tableaux temporaires. En théorie, nous pouvons espérer n'introduire que des tableaux temporaires de dimension égale à la profondeur du nid moins le nombre de boucles séquentielles englobantes déjà générées au moment de leur introduction. Pour l'exemple de référence, nous avons pu appliquer la seconde transformation de Padua et Wolfe dans ces conditions (cf. figure 6.37) : nous nous sommes servis d'un tableau de dimension 1 dans un nid de profondeur 3 lors de la transformation de S_1. L'utilisation d'un tableau à une seule dimension a provoqué l'apparition d'auto-dépendances de sortie de niveaux 1 et 2, ce qui n'eut aucune conséquence, les deux boucles externes étant déjà marquées « séquentielles ».

En pratique, cependant, les dimensions des temporaires peuvent être très variables. D'un côté, nous pouvons avoir besoin de plus de mémoire si les techniques d'élimination des dépendances ne sont pas suffisamment puissantes, les fonctions d'accès aux tableaux étant, par exemple, trop compliquées. De l'autre, la quantité de mémoire requise peut être réduite par privatisation le long des dimensions parallèles. Ce qu'illustre le premier code que nous avons transformé : l'exemple 21 (figure 6.7). Dans cet exemple, le tableau `temp` introduit peut être remplacé par un scalaire privatisé.

6.3.3 Application du schéma d'intégration à ALLEN-KENNEDY

Pour une description complète de ALLEN-KENNEDY, nous renvoyons le lecteur à la section 2.2. L'algorithme INTÉGRATION ayant été conçu après l'étude de la parallélisation d'un cas particulier par ALLEN-KENNEDY, il n'est pas étonnant qu'il s'applique aisément à cet algorithme d'ordonnancement. La détermination de l'ensemble des arcs dits *incompatibles* est d'autant plus facile que ALLEN-KENNEDY n'utilise que la distribution de boucles qui ne requiert que la détection de composantes fortement connexes.

Soit H un graphe de dépendance réduit par niveaux. Nous notons H_l le sous-graphe de H obtenu par suppression de tous les arcs de niveau strictement inférieur à l.

Lemme 22 (Chemins incompatibles pour ALLEN-KENNEDY).
Les chemins incompatibles au niveau l sont exactement les circuits de G_l qui contiennent un arc de niveau l et un sommet qui n'appartient à aucun circuit de F_l contenant un arc de niveau l.

DÉMONSTRATION. Si une instruction S appartient à un circuit de G_l qui contient un arc de niveau l, elle appartient à une composante fortement connexe contenant un arc de niveau l et

ALLEN-KENNEDY entoure l'instruction par une $l^{ème}$ boucle qui est séquentielle. La réciproque est également vraie.

Pour que la $l^{ème}$ boucle englobant S soit reconnue parallèle par ALLEN-KENNEDY, il faut donc supprimer tous les circuits de G_l contenant S et un arc de niveau l. Cette condition est nécessaire et suffisante. □

Pour pouvoir exhiber le maximum de parallélisme, nous devons donc détruire chacun des circuits apparaissant dans la définition. *A priori*, le problème est compliqué puisque, si l'ensemble est non vide, il contient un nombre infini de circuits. En pratique, nous ne considérons que les circuits minimaux pour l'inclusion. Le problème de la minimisation du nombre de transformations à appliquer est un problème NP-complet si nous supposons qu'une transformation peut au plus déconnecter tous les arcs entrants et tous les arcs sortants de l'instruction à laquelle elle est appliquée (par NP-complétude de GRAPHE-SANS-CIRCUITS, cf. théorème 13).

Le besoin d'éliminer de faux circuits de dépendance justifie l'étude de la seconde transformation de Padua et Wolfe (section 6.2).

6.3.4 Application du schéma d'intégration à WOLF-LAM

Pour une description complète de WOLF-LAM, voir la section 2.3. Cet algorithme ne considère pas la structure du graphe de dépendance mais uniquement à la valeur des vecteurs de distance. Les techniques d'élimination de faux chemins de dépendance, comme celles de Padua et Wolfe, sont donc ici sans objet. Nous avons besoin de techniques d'éliminations de dépendances.

L'application du schéma d'intégration à WOLF-LAM est loin d'être immédiate. Nous illustrons cette difficulté au moyen de l'exemple 25 (figure 6.38). Cet exemple contient trois auto-dépendances : une flot de poids $(0, 1, 1)$, et deux anti de poids $(1, 0-, 0)$ et $(0, 1, 0-)$. Nous pouvons éliminer chacune des deux fausses dépendances par l'introduction d'un temporaire dans lequel serait préalablement recopié les données nécessaires, comme le montre la figure 6.38. Nous obtenons de la sorte deux nids. Le premier est un nid d'initialisations, sans dépendances internes, et qui est totalement parallèle. Le deuxième contient la dépendance de flot, trois boucles totalement permutables et deux degrés de parallélisme. Sur le nid original WOLF-LAM trouvera que le premier ensemble de boucles permutables ne contient que la boucle externe, et que le deuxième ensemble contient deux boucles permutables (et un degré de parallélisme). Par contre, il suffit d'éliminer n'importe laquelle des deux anti-dépendances pour que WOLF-LAM soit à même de trouver trois boucles externes permutables et deux degrés de parallélisme. Autrement dit, aucune des deux fausses dépendances n'est intrinsèquement *incompatible* avec la détection des boucles permutables mais leur coexistence est *incompatible* avec une extraction maximale du parallélisme.

Exemple 25

```
For i = 0 to N do
 For j = 0 to N do
  For k = 0 to N do
  a(i,j,k) = a(i+1,0,k) + a(i,j+1,0) + a(i,j-1,k-1)
```

```
For i = 0 to N do
 For j = 0 to N do
   b(i,j) = a(i+1,0,j)
   c(i,j) = a(i,j+1,0)

For i = 0 to N do
 For j = 0 to N do
  For k = 0 to N do
   a(i,j,k) = b(i,k) + c(i,j) + a(i,j-1.k-1)
```

FIG. 6.38: *Exemple 25 et une version sans fausses dépendances.*

Pour cet exemple 25, il y a donc deux ensembles de dépendances incompatibles et de taille minimale, chacun contenant une des deux anti-dépendances. Éliminer l'unique dépendance contenu dans l'un ou dans l'autre permet d'exposer le maximum de boucles permutables externes (et le maximum de parallélisme). Nous allons montrer que déterminer des ensembles de dépendances incompatibles minimaux est un problème NP-complet dans le cas de WOLF-LAM.

NP-complétude de la minimisation du nombre de dépendances à éliminer

Problématique. Nous disposons de deux ensembles de vecteurs de direction : $\mathcal{I}$, l'ensemble des dépendances qui sont inamovibles, ensemble qui comprend les dépendances de flot, c'est l'ensemble des dépendances de F; $\mathcal{F}$ l'ensemble des fausses dépendances éliminables. $(\mathcal{I}, \mathcal{F})$ est l'ensemble des dépendances de G. Nous recherchons le nombre minimal de vecteurs de direction qui doivent être supprimés de $\mathcal{F}$ pour obtenir un ensemble $\mathcal{F}'$ tel que l'ensemble des vecteurs de direction restants, $\mathcal{I} \cup \mathcal{F}'$, admette autant de boucles totalement permutables [6] externes que l'ensemble $\mathcal{I}$. Le problème de décision associé se définit comme suit :

Définition 21 (Problème de décision associé).
Soient deux ensembles de vecteurs de direction $\mathcal{I}$ et $\mathcal{F}$ et un entier positif K. $\mathcal{I} \cup \mathcal{F}$ (resp. $\mathcal{F}$) est l'ensemble des dépendances (resp. des dépendances supprimables par des techniques d'élimination) d'un nid de boucles. Pouvons-nous éliminer au plus K vecteurs de $\mathcal{F}$ pour obtenir un ensemble $\mathcal{F}'$ tel que l'ensemble de vecteurs $\mathcal{I} \cup \mathcal{F}'$ admette autant de boucles totalement permutables externes que l'ensemble $\mathcal{I}$? Si la réponse est positive, $(\mathcal{I}, \mathcal{F})$ appartient à WL-COMPATIBLE*(K).*

Nous démontrons que WL-COMPATIBLE est un problème NP-complet par réduction du problème de satisfiabilité 3-SAT. La démonstration de NP-complétude du problème analogue pour ALLEN-KENNEDY ne s'étend pas ici. En effet, il nous faudrait ne considérer que des nids à une seule instruction et à dépendances représentées par des niveaux, mais le problème devient trivial dans ce cas. La preuve de NP-complétude pour ALLEN-KENNEDY repose sur des propriétés de graphes or nous n'avons plus ici de structure de graphe. Nous ne pouvons donc raisonner que sur la valeur des vecteurs de distance. ALLEN-KENNEDY et WOLF-LAM travaillent sur des objets fondamentalement différents, l'un privilégiant la structure du graphe de dépendance, l'autre ne s'occupant que de la valeur des vecteurs de distance. Les preuves de NP-complétude des problèmes qui leur sont ici associés sont donc fondamentalement différentes.

Théorème 15. WL-COMPATIBLE *est un problème NP-complet.*

DÉMONSTRATION.
Il nous faut vérifier l'appartenance de WL-COMPATIBLE à NP. WOLF-LAM est de complexité polynomiale. Il suffit donc de l'exécuter sur les ensembles de vecteurs de direction $\mathcal{I}$ et $\mathcal{I} \cup \mathcal{F}'$ et de comparer le nombre de boucles de chaque ensemble de boucles totalement permutables. WL-COMPATIBLE appartient bien à NP.

Nous procédons par réduction à partir du problème 3-SAT. Une instance de ce problème est formée d'une expression booléenne B sous forme normale conjonctive :

$$B = \bigwedge_{i=1}^{t} C_i$$

6. La démonstration ne considère que des nids dont l'ensemble des boucles permutables externes contient toutes les boucles quand toutes les fausses dépendances sont éliminées. De plus, les vecteurs de direction des nids considérés ne contiennent pas d'étoiles « $*$ ». Dans ce cas, l'ensemble des boucles permutables externes contient toutes les boucles si et seulement si le nid contient $n-1$ degrés de parallélisme. La démonstration proposée est donc valable que nous nous intéressions à WOLF-LAM pour la génération de boucles permutables, ou pour la génération de boucles parallèles.

– où $C_j = \ell_j^1 \vee \ell_j^2 \vee \ell_j^3$, $1 \leq j \leq t$, est une clause,

– et où chaque littéral ℓ_j^k est une variable de l'ensemble $X = \{x_1, \cdots, x_r\}$ ou sa négation.

La problème de décision associé est : y a-t-il une fonction de vérité $w\ : X \rightarrow \{\text{Vrai}, \text{Faux}\}$ telle que B s'évalue à Vrai pour w ? Si la réponse est positive, B appartient à 3-Sat.

Étant donné B nous devons construire une instance de notre problème (deux ensembles de vecteurs de direction $\mathcal{I}$ et $\mathcal{F}$ et un entier K) telle que l'appartenance de $(\mathcal{I}, \mathcal{F})$ à WL-Compatible(K) soit équivalente à l'appartenance de B à 3-Sat. La fonction de construction doit en outre être de coût polynomial en la taille de B, c'est-à-dire en le nombre de clauses et de variables.

Construction de $\mathcal{F}$. $\mathcal{F}$ est un ensemble de $2r$ vecteurs $v_1, \ldots v_{2r}$ de dimension $1 + 3t + r$.

– Le vecteur v_{2i-1} correspond au littéral « x_i », pour i compris entre 1 et r. Sa première composante vaut 1, pour assurer sa lexico-positivité. Pour $1 \leq j \leq t$, sa composante de rang $1 + 3(j-1) + k$ vaut « 0- » si le littéral l_j^k est égal à x_i (l_j^k est le $k^{ème}$ littéral de la $j^{ème}$ clause) et vaut 0 sinon. Les r dernières composantes de v_{2i-1} sont nulles sauf la composante $1 + 3t + i$ qui vaut « 0+ ».

– Le vecteur v_{2i} correspondant au littéral « $\neg x_i$ », pour i compris entre 1 et r. Sa première composante vaut 1. Sa composante de rang $1 + 3(j-1) + k$ vaut « 0- » si le littéral l_j^k est égal à $\neg x_i$ et vaut 0 sinon. Les r dernières composantes de v_{2i-1} sont nulles sauf la composante $1 + 3t + i$ qui vaut « 0- ».

Les r dernières composantes des vecteurs v_j vont interdire la présence simultanée dans $\mathcal{F}'$ de v_{2i-1} et v_{2i}.

Construction de $\mathcal{I}$. $\mathcal{I}$ est un ensemble de t vecteurs $c_1, \ldots, c_t$ de la forme :

$$c_i = (0, \underbrace{0, \ldots, 0}_{3*(i-1)}, 1, 1, 1, \underbrace{0, \ldots, 0}_{3*(t-i)}, \underbrace{0, \ldots, 0}_{r})$$

Chacun de ces vecteurs correspond à une clause : c_i est associée à C_i. Les trois composantes consécutives égale à 1 serviront à lier les trois littéraux de la clause C_i.

Nous posons finalement $K = r$. Autrement dit nous nous demandons s'il est possible de trouver autant de boucles totalement permutables externes dans $\mathcal{I} \cup \mathcal{F}$ que dans $\mathcal{I}$ en supprimant la moitié des arcs de $\mathcal{F}$. La construction de $\mathcal{I}$, de $\mathcal{F}$ et de K est notoirement de polynomial.

Intuition. Les multi-circuits de poids nul de $\mathcal{I} \cup \mathcal{F}$ correspondent aux clauses (à l'exception, pour tout $i \in [1, r]$, de celui liant v_{2i-1} et v_{2i} par leur composante de rang $1 + 3t + i$). Si nous ne considérons que les composantes correspondants à la clause C_j, c'est-à-dire celles de rangs compris entre $1 + 3(j-1) + 1$ et $1 + 3(j-1) + 3$, nous obtenons uniquement quatre vecteurs non nuls de poids respectifs : $(1, 1, 1)$, $(0-, 0, 0)$, $(0, 0-, 0)$ et $(0, 0, 0-)$. Le premier vecteur correspond à c_j et les trois autres aux littéraux apparaissant dans C_j. Les trois derniers vecteurs forcent l'appartenance du premier à un multi-circuit de poids nul.

Nous attirons l'attention du lecteur sur le fait que les vecteurs de direction utilisés ne contiennent pas de lignes quand ils sont considérés comme des polyèdres.

Construction du nid. Nous construisons un nid à une seule instruction dont les dépendances de flot forment l'ensemble $\mathcal{I}$ et l'ensemble des dépendances amovibles par les techniques d'élimination constituent l'ensemble $\mathcal{F}$. Nous construisons un ensemble de $1 + 3t + r$ boucles, la $j^{ème}$ s'écrivant : `For` $i_j = 0$ `to` N `do`. L'unique instruction S est produite par l'affectation : $\mathsf{A}(i_1, \ldots, i_{1+3t+r}) = 1$. Nous rajoutons au membre droit de cette affectation l'expression :

« + $\mathsf{A}(i_1, \ldots, i_{1+3(j-1)}, i_{1+3(j-1)+1} - 1, i_{1+3(j-1)+2} - 1, i_{1+3(j-1)+3} - 1, i_{1+3j+1}, \ldots, i_{1+3t+r})$ », pour introduire une auto dépendance de flot de vecteur de direction c_j. Une auto anti-dépendance de vecteur de direction v_i est générée par l'addition au membre droit de S de l'expression $E(v_i)$: « + $\mathsf{A}(i_1 + 1, i_2, \ldots, i_{1+3t+r})$ » dans laquelle 0 (resp. N) remplace i_j dans la composante de rang j si la coordonnée correspondante de v_i vaut « 0- » (resp. « 0+ »). Cette anti-dépendance peut être éliminée par la pré-copie hors du nid des valeurs lues par $E(v_i)$, leur stockage dans un temporaire et le remplacement dans S de $E(v_i)$ par l'accès au temporaire (cette manipulation a été utilisée pour l'exemple 25).

Équivalence des problèmes. Nous prouvons maintenant l'équivalence : $(\mathcal{I}, \mathcal{F}) \in$WL-COMPATIBLE$(K) \Leftrightarrow B \in$ 3-SAT.

Supposons que B appartienne à 3-SAT. Soit $w : X \rightarrow \{$VRAI, FAUX$\}$ une fonction de vérité permettant à B de s'évaluer à VRAI. Si w assigne la valeur VRAI à la variable x_i ($w(x_i) =$ VRAI), nous supprimons le vecteur v_{2i-1} de $\mathcal{F}$. Sinon, nous supprimons v_{2i}. Nous supprimons donc en tout $K = r$ vecteurs. Autrement dit $\mathcal{F}'$ contient v_{2i} si x_i est vrai et v_{2i-1} sinon.

Il nous faut montrer qu'il y a autant de boucles permutables externes dans $\mathcal{I}$ que dans $(\mathcal{I}, \mathcal{F}')$. Les vecteurs de $\mathcal{I}$ étant tous positifs par construction, les $1 + 3t + r$ boucles sont totalement permutables pour $\mathcal{I}$. Nous allons montrer que toutes les boucles peuvent être également rendues permutables pour $(\mathcal{I}, \mathcal{F}')$. Pour cela, il nous suffit de prouver que cet ensemble de vecteurs de direction n'engendre pas de multi-circuits de poids nul. Nous allons construire un hyperplan fortement séparateur Y qui induit un délai strictement positif sur tous les vecteurs de direction et sur tous les rayons et toutes les lignes [7] des vecteurs de direction (quand ces derniers sont considérés comme des polyèdres) ce qui est la condition nécessaire et suffisante pour montrer que G' est vide.

Le produit scalaire de Y avec un vecteur $\boldsymbol{d}$ de $(\mathcal{I}, \mathcal{F}')$ se décompose comme la somme : du produit de la première composante de Y avec celle de $\boldsymbol{d}$; de la somme sur toutes les clauses du produit scalaire des composantes de Y correspondant à la clause C_j (composantes de rangs $1+3(j-1)+1$ à $1+3(j-1)+3$) avec celles de $\boldsymbol{d}$; du produit scalaire des r dernières composantes de Y avec celles de $\boldsymbol{d}$. Nous construisons Y de sorte que :

1. le premier terme de cette décomposition soit positif en général et strictement positif pour les vecteurs de $\mathcal{F}'$;

2. la somme provenant de la clause C_j soit positive pour tous les vecteurs et strictement positive pour c_j et pour les vecteurs qui correspondent à des littéraux présents dans la clause C_j;

3. le dernier terme de la décomposition soit positif.

Nous construisons Y pour qu'il vérifie ces trois propriétés :

1. La première composante de tous les vecteurs est positive par construction, $Y_1 = 1$ satisfait les conditions énoncées pour la première composante.

2. Notons E_j l'espace vectoriel engendré par les vecteurs dont seules les composantes de rangs $1+3(j-1)+1$, $1+3(j-1)+2$ et $1+3(j-1)+3$ sont non nulles. $Y|_{E_j}$ désigne la restriction de Y à E_j. Tous les vecteurs ont une projection nulle sur E_j exceptés c_j et les vecteurs qui correspondent à des littéraux présents dans la clause C_j. B appartenant à 3-SAT, au moins

7. Comme nous l'avons vu à la section 2.3, si l'ensemble des boucles externes totalement permutables contient toutes les boucles, il existe un hyperplan fortement séparateur. La réciproque n'est pas vrai dans le cas général. Il faut en plus que l'ensemble des multi-circuits de poids nul soit vide, ce qui est équivalent à l'existence d'un hyperplan induisant un délai strictement positif sur tous les vecteurs de direction et sur tous les rayons et toutes les lignes.

un des littéraux de C_j s'évalue à vrai pour w. Or si w assigne la valeur VRAI à la variable x_i, nous avons supprimé le vecteur v_{2i-1} de $\mathcal{F}$ et sinon le vecteur v_{2i}. Donc aucun vecteur correspondant à un littéral s'évaluant à VRAI n'appartient à $\mathcal{F}'$. Par conséquent, au plus deux des vecteurs qui correspondent à des littéraux présents dans la clause C_j appartiennent à $\mathcal{F}'$. À une renumérotation des lignes près, l'ensemble des vecteurs, dont nous voulons que le produit scalaire avec $Y|_{E_j}$ soit strictement positif, vaut donc $\{(1,1,1),(0-,0,0),(0,0-,0)\}$ ou $\{(1,1,1),(0-,0,0)\}$ ou $\{(1,1,1)\}$ suivant le nombre de littéraux de C_j s'évaluant à VRAI. Dans tous les cas nous pouvons poser $Y|_{E_j} = (-1,-1,3)$.

3. Exactement un seul des $t+r$ vecteurs de $\mathcal{I} \cup \mathcal{F}'$ a sa composante de rang $1+3t+j$ non nulle : celui de v_{2j} et de v_{2j-1} qui figure dans $\mathcal{F}'$. Si cette composante est égale à « 0- », $Y_{1+3t+j} = -1$, sinon $Y_{1+3t+j} = +1$.

En conclusion, nous avons construit un vecteur Y dont le produit scalaire avec tout vecteur, et avec tout rayon ou toute ligne de tout vecteur, $\boldsymbol{d}$ de $(\mathcal{I}, \mathcal{F}')$ est strictement positif.

Réciproquement, supposons que $(\mathcal{I}, \mathcal{F})$ *appartienne à* WL-COMPATIBLE(K). Nous devons construire une fonction de vérité w qui permette d'évaluer B à vrai.

Soit $\mathcal{F}'$ le sous-ensemble de $\mathcal{F}$ à au moins r éléments tel que $(\mathcal{I}, \mathcal{F}')$ admet autant de boucles totalement permutables externes que $\mathcal{I}$. Toutes les boucles sont bien évidemment totalement permutables pour $\mathcal{I}$ qui est un ensemble de vecteurs à coordonnées toutes positives. Il existe donc un vecteur Y dont le produit scalaire avec tous les vecteurs de $(\mathcal{I}, \mathcal{F}')$ est strictement positif.

$\mathcal{F}'$ possédant au moins r éléments, il contient exactement un des deux vecteurs v_{2i} et v_{2i-1} $(1 \leq i \leq r)$, ceux-ci étant incompatibles à cause de leurs composantes respectives de rang $1+3t+i$. Si v_{2i-1} appartient à $\mathcal{F}'$ nous posons $w(x_i) = $ FAUX et sinon nous posons $w(x_i) = $ VRAI. Supposons que C_j s'évalue à FAUX sous w, il en va alors de même de tous ses littéraux. Pour tout élément k de $\{1,2,3\}$, il existe une variable x_i, telle que $l_j^k = x_i$ ou $l_j^k = \neg x_i$. Si $l_j^k = x_i$ et s'évalue à faux, $w(x_i) =$FAUX et v_{2i-1} appartient à $\mathcal{F}'$. Si $l_j^k = \neg x_i$ et s'évalue à faux, $w(x_i) =$VRAI et v_{2i} appartient à $\mathcal{F}'$. Dans les deux cas, $\mathcal{F}'$ contient le vecteur correspondant au littéral l_j^k, c'est-à-dire le vecteur dont la coordonnée de rang $1+3(j-1)+k$ vaut « 0- ». Par conséquent, la coordonnée de Y de rang $1+3(j-1)+k$ est négative ou nulle. Or $(\mathcal{I} \cup \mathcal{F}')$ contient $c_j = (0, \underbrace{0, \dots, 0}_{3*(j-1)}, 1, 1, 1, \underbrace{0, \dots, 0}_{3*(t-j)}, \underbrace{0, \dots, 0}_{r})$, dont le produit scalaire avec Y est donc négatif ou nul. Il y a contradiction : C_j ne peut s'évaluer à faux. □

Ayant prouvé la complexité du problème, nous introduisons une heuristique pour tenter de le résoudre.

Heuristique

Nous proposons d'appliquer simplement une heuristique gloutonne à notre problème : partir de l'ensemble de vecteurs $\mathcal{I}$ et lui rajouter un par un des vecteurs de $\mathcal{F}$ en vérifiant qu'il est bien toujours possible de trouver k boucles externes totalement permutables. Nous préconisons de choisir en priorité des vecteurs de $\mathcal{F}$ qui ont un maximum de composantes constantes parmi celles qui correspondent aux boucles qui peuvent être rendues permutables, les composantes de la forme « $z+$ » et « $z-$ » contraignant fortement l'espace des solutions.

6.3.5 Application du schéma d'intégration à ORDONNANCEUR_DE_GDRP

Pour une description complète de ORDONNANCEUR_DE_GDRP, voir le chapitre 4. Comme nous l'avons montré à la section 4.5, cet algorithme trouve exactement la même quantité de parallélisme que WOLF-LAM pour les nids à une seule instruction dont les dépendances sont exprimées au moyen de vecteurs de direction. Nous sommes donc exactement dans le cadre de la preuve de NP-complétude de WL-COMPATIBLE. Minimiser la taille de l'ensemble des fausses dépendances et faux chemins de dépendance à éliminer pour que notre algorithme découvre le maximum de parallélisme est donc également un problème NP-complet. Parcourir l'espace des solutions nous amènerait à devoir résoudre un nombre exponentiel de programmes linéaires. Nous sommes donc amenés à proposer une fois de plus une heuristique.

Contrairement au cas de l'algorithme de Wolf et Lam, ce n'est pas le poids de l'arc que nous ajoutons ou enlevons au graphe de dépendance qui importe, mais le poids des circuits élémentaires incluant cet arc. D'où une difficulté plus grande à définir une heuristique. Nous proposons simplement d'éliminer dans un premier temps les chemins de dépendance entre les composantes fortement connexes de F, puis d'appliquer une heuristique gloutonne sur chacune des composantes fortement connexes. Pour améliorer l'heuristique il faudrait pouvoir orienter le choix de l'arc ou du chemin que nous essayons d'ajouter à l'ensemble des arcs que nous devons ou avons décidé de garder. Nous pouvons conserver sans problème toute auto-dépendance dont le poids est inclus dans l'espace vectoriel E engendré par les poids des circuits de G'. L'introduction d'une dépendance, qui ne serait pas une auto-dépendance, mais dont le poids appartiendrait à E, peut quant à elle entraîner une perte de parallélisme puisqu'elle peut former un circuit avec un chemin dont le poids n'est pas élément de E et augmenter ainsi la dimension de E...

6.3.6 Conclusion

À partir d'une étude de cas utilisant l'algorithme d'Allen et Kennedy nous avons établi un schéma d'intégration des techniques d'élimination de dépendances au sein des algorithmes de détection de parallélisme. Bien que ce schéma ait semblé prometteur dans son cadre d'étude, son application aux algorithmes WOLF-LAM et ORDONNANCEUR_DE_GDRP s'est révélée difficile. Cette difficulté croit avec la complexité des techniques mises en œuvre par les algorithmes de détection. Dans les deux cas, il est apparu que déterminer l'ensemble minimal d'arcs et de chemins de dépendance à supprimer était un problème NP-complet. Les heuristiques proposées dans les deux cas sont des tirages au sort à peine améliorés. En tout état de cause, seule une expérimentation conséquente pourra nous dire si le schéma proposé est ou non intéressant.

D'autres chercheurs se sont intéressés au problème du contrôle de l'expansion mémoire nécessaire à une parallélisation maximale. Chamski [21] a essayé de « replier » la mémoire après une parallélisation suivant une mise en assignation unique. Vincent Lefebvre et Paul Feautrier [56] commencent par trouver un ordonnancement en ne tenant compte que des dépendances de flot. Ils déterminent ensuite la mémoire minimale requise pour que l'exécution du programme dictée par l'ordonnancement soit correcte. Il ne s'agit pas ici d'expansion à proprement parler puisqu'ils ne tiennent pas compte des variables et tableaux du programme initial mais seulement de l'ensemble des données qui circulent. Il est donc tout à possible que Lefebvre et Feautrier produisent un programme parallèle utilisant moins de mémoire que le programme séquentiel initial. Ils cherchent la mémoire minimale requise « par » un ordonnancement donné. Au contraire, nous cherchons à minimiser la mémoire tout en conservant la dimension de l'espace des solutions du problème d'ordonnancement (pour les boucles permutables) ou tout en nous assurant que cet espace est simplement non vide (pour les boucles parallèles). C'est là la différence fondamentale entre nos approches respectives.

Chapitre 7

Autres travaux

Dans ce chapitre, nous décrivons très brièvement les travaux menés au cours de cette thèse et qui n'ont pas été développés dans le présent mémoire, pour des raisons de place ou d'homogénéité.

Resynchronisation de graphes sans circuits

La complexité grandissante des circuits numériques rend désormais impossible toute optimisation globale d'un circuit: les concepteurs ne considèrent plus que les parties critiques de leur circuit. Notre article [18, 19] traite des problèmes d'optimisation temporelle lorsque ces parties critiques peuvent être représentées par des graphes orientés sans circuits. Nous considérons différents problèmes, concernant les contraintes sur la période d'horloge, qui interviennent lors de la conception d'un circuit et qui découlent de la détermination de la période optimale du circuit. Nous proposons un algorithme de faible complexité, basé sur la technique de resynchronisation, pour résoudre ce problème dont nous donnons trois formulations différentes, selon le modèle de délai utilisé pour les portes combinatoires : même délai pour toutes les portes, délais différents ou délais non uniformes.

Combinaison des techniques d'ordonnancement et de resynchronisation

Classiquement, si nous voulons paralléliser un nid de boucles nous allons chercher un ordonnancement qui satisfait toutes les dépendances. Toutes les dépendances du nid après transformation seront alors portées par des boucles, si le corps de boucle n'a pas été considéré atomique comme c'est par exemple le cas pour WOLF-LAM. Rien n'interdit pourtant d'avoir des nids parallèles avec des dépendances internes au corps de boucle, sinon notre méthodologie. Nous montrons dans [30, 31] comment relâcher les contraintes, lors de la génération d'ordonnancements par ORDONNANCEUR_DE_GDRP, pour autoriser l'existence de dépendances indépendantes des boucles sans rien perdre de l'efficacité de l'algorithme, au contraire. Pour ce faire nous utilisons des techniques de resynchronisation de graphes (*retiming*). Cette modification de l'algorithme nous permet d'obtenir, après transformation du nid, des codes qui ne pouvaient pas être produits initialement. Nous présentons également le principe d'un algorithme générant des boucles totalement permutables et autorisant la présence de dépendances indépendantes des boucles.

Fusion de boucles parallèles

Dans [13, 14] nous passons en revue les principaux algorithmes de parallélisation. Nous nous y intéressons également au problème de la fusion de boucles parallèles. Cette fusion est importante, la réduction du nombre de boucles parallèles entraînant une diminution des synchronisations nécessaires à l'exécution du programme. Nous avons considéré ce problème dans le cadre de la génération du code correspondant aux ordonnancements multidimensionnels produits par la version de notre algorithme autorisant la présence de dépendances indépendantes des boucles. Nous avons montré que le problème était NP-complet mais nous avons proposé un programme linéaire en nombre entier pour le résoudre qui, étant donné la taille des graphes en pratique, est utilisable bien qu'exponentiel.

Placement de tuiles en milieu hétérogène

Les techniques de partitionnement produisent des codes où les instructions sont regroupées en *tuiles*. Une fois le code généré, il reste à répartir les tuiles entre les différents processeurs et à en ordonnancer l'exécution. Nous nous sommes intéressés [15] à ce problème dans le cadre des environnement hétérogènes, c'est-à-dire quand les différents processeurs disponibles n'ont pas la même vitesse. Nous avons proposé une solution optimale basée sur un programme linéaire en nombre entiers. Malheureusement, ce programme est trop gros pour qu'il soit envisageable de le résoudre dans la pratique. Nous avons alors conçu une heuristique simple, régulière et asymptotiquement optimale. Nous avons montré comment la mettre en pratique et nous avons mené des expérimentations sur un ensemble de stations de travail pour montrer son efficacité.

Conclusion

> En réalité, je ne supporte plus aucun discours rationnel. Tout ce qui a fait le monde, tout ce qui a été beau et grand un jour dans le monde, n'est jamais né d'un discours rationnel.
>
> Yvan *in* « Art », Yasmina **Reza**

Comme son intitulé l'indique judicieusement, cette thèse avait pour but la détection de parallélisme dans les boucles imbriquées. Avant de proposer de nouvelles techniques et de nouveaux algorithmes, nous avons passé en revue les solutions existantes. Pour pouvoir évaluer ces solutions, nous avons été amenés à définir une notion d'optimalité qui ne dépend que de la représentation des dépendances choisie.

Leslie Lamport a en quelque sorte inauguré le domaine avec ses méthodes de l'hyperplan et du plan. Au début des années 1970 et contrairement à ce qu'en a retenu l'imagerie populaire, il cherchait déjà des ordonnancements multidimensionnels. Il essayait donc d'ordonnancer des codes présentant des irrégularités. Très peu de résultats étaient connus à l'époque sur l'analyse de dépendance. Ses algorithmes travaillaient sur une représentation approchée, ancêtre des vecteurs de direction, et il considérait le corps de boucle comme étant atomique.

John Allen et Ken Kennedy ont proposé une approche totalement différente. Ils n'ont pas supposé comme Lamport que le corps de boucle était atomique. Au contraire, ils ont considéré indépendamment les différentes instructions. Plutôt que de se focaliser sur les vecteurs de distance et d'utiliser une approximation « précise » de leur valeur, ils se sont contentés d'une représentation grossière (les niveaux) et se sont surtout intéressés au graphe généré par les dépendances. Nous avons pu montrer que leur algorithme était optimal en ordre de grandeur : aucun algorithme qui ne connaît d'un nid de boucles que son graphe de dépendance réduit par niveaux ne peut trouver plus de parallélisme, en ordre de grandeur, que n'en trouve l'algorithme d'Allen et Kennedy. Pour pouvoir trouver plus de parallélisme il faut donc plus d'information, que ce soit sur la valeur des vecteurs de distance, sur le domaine d'existence des dépendances, sur les domaines d'itérations, etc.

Michael Wolf et Monica Lam se sont intéressés à un modèle proche de celui de Lamport : des vecteurs de direction représentant les dépendances d'un nid dont le corps de boucle est considéré comme étant atomique. Dans ce modèle, ils ont recherché des transformations unimodulaires générant des ensembles de boucles totalement permutables. Au contraire des deux algorithmes précédents qui recherchaient du parallélisme à grain fin, Wolf et Lam se sont intéressés au parallélisme à grain moyen : ils produisent des boucles permutables en vue d'un partitionnement ultérieur. Nous avons pu montrer que l'algorithme de Wolf et Lam était optimal pour son modèle : aucun algorithme, qui ne connaît d'un nid de boucles qu'une approximation des dépendances par des vecteurs de direction et qui n'a, entre autres, aucune connaissance du graphe de dépendance, ne peut trouver plus de parallélisme, en ordre de grandeur, que n'en trouve l'algorithme de Wolf et Lam.

Nous avons pu exhiber des exemples qui montrent que les algorithmes d'Allen et Kennedy et de Wolf et Lam ont leurs domaines d'excellence respectifs : aucun des deux algorithmes ne généralise l'autre. De plus, quand les dépendances sont représentées par un graphe de dépendance réduit étiqueté par des vecteurs de direction, les deux algorithmes sont sous-optimaux, c'est-à-dire qu'aucun des deux n'est capable de trouver systématiquement tout le parallélisme contenu, en ordre de grandeur, dans la représentation des dépendances.

Nous nous sommes intéressés à une représentation des dépendances un peu plus générale et englobant celles utilisées par Allen et Kennedy, et Wolf et Lam : les graphes de dépendance réduits étiquetés par des polyèdres. Nous sommes partis d'une étude théorique du modèle que forment les systèmes d'équations récurrentes uniformes. Nous avons ensuite transposé aux graphes de dépendance réduits polyédriques les résultats de calculabilité et d'ordonnancement obtenus pour ces systèmes. Nous avons obtenu de la sorte un algorithme d'ordonnancement optimal pour cette représentation des dépendances. La méthodologie employée permet de connaître exactement les dépendances responsables de la séquentialité du nid étudié. De plus, cet algorithme ne se contente pas de produire un ordonnancement possible : il détermine exactement l'ensemble de tous les ordonnancements qu'il est capable de construire. Enfin, cet algorithme, initialement destiné à la parallélisation à grain fin, peut être adapté pour générer des boucles permutables et du parallélisme à grain moyen. Nous avons pu montrer que notre algorithme généralisait ceux de Allen et Kennedy et de Wolf et Lam.

Avant que nous ne commencions nos travaux, Paul Feautrier avait proposé un algorithme d'ordonnancement qui ne prend pas en entrée une représentation approchée des dépendances mais une représentation exacte. Il était connu que son algorithme est optimal quand les dépendances sont uniformes mais qu'il est sous-optimal dans le cas général. La méthodologie utilisée est très générale et permet d'ordonnancer non seulement des dépendances affines mais également des représentations approchées des dépendances, que les nids soient ou non parfaits. Nous avons pu montrer que l'algorithme de Feautrier ordonnance comme le nôtre un nid dont n'est connu que le graphe de dépendance réduit polyédrique et qu'il était donc optimal pour les représentations polyédriques des dépendances. Ce résultat a pu affirmer un peu plus la puissance pressentie mais non démontrée de l'approche de Paul Feautrier.

Amy Lim et Monica Lam ont proposé un algorithme de parallélisation à gros grain qui repose sur une représentation exacte des dépendances. Cet algorithme peut être vu comme une extension de celui de Feautrier. Cet algorithme diffère des précédents en ce qu'il ne recherche pas le maximum de parallélisme mais le maximum de parallélisme exposable au moyen de transformations affines et sans dépasser un volume de synchronisations donné. Nous avons pu infirmer la démonstration d'optimalité de cet algorithme. La classe d'optimalité de cet algorithme est donc un problème totalement ouvert.

Nous résumons ces résultats dans la table 2. Les algorithmes sont classés par ordre croissant de leur complexité, qui est aussi l'ordre croissant de la complexité des codes générés. La colonne « Synchro. - Fusion » indique si un algorithme essaye de, ou peut être étendu pour, minimiser le nombre des synchronisations nécessaires à l'exécution des codes parallèles qu'il produit, que cette tentative de minimisation soit directement intégrée dans la détection de parallélisme ou qu'elle ait lieu *a posteriori* par fusion de boucles. La dernière colonne a pour objet le parallélisme à grain moyen et la génération de boucles permutables.

Notre étude offre une classification des algorithmes de détection de parallélisme que nous avons étudiés. Nous pensons avoir répondu à la question : « Étant donné une représentation des dépendances, quel est l'algorithme le plus simple capable d'en extraire le maximum de parallélisme ? » Les principales questions théoriques ouvertes concernent l'efficacité de l'algorithme de Feautrier : est-il

Algorithmes	**Représentation des dépendances**	**Transformation de boucles**	**Optimal (degré de parallélisme)**	**Synchro. Fusion**	**Grain moyen**
Allen-Kennedy	Niveaux de dépendance Plusieurs instructions Nids non parfaits	Distribution	Oui	Oui	Non
Wolf-Lam	Vecteurs de direction Une instruction Nids parfaits	Unimodulaire	Oui	Non	Oui
Darte-Vivien	Polyèdres Plusieurs instructions Nids non parfaits	Linéaire décalé	Oui	Oui	Oui
Feautrier	Affines (exactes) Plusieurs instructions Nids non parfaits	Affine	Non (oui pour polyèdres)	Non	Non
Lim-Lam	Affines (exactes) Plusieurs instructions Nids non parfaits	Affine	Non	Oui	Oui

TAB. 2: *Comparaison des différents algorithmes de détection de parallélisme.*

possible d'élaborer un algorithme de détection du parallélisme plus efficace, en ordre de grandeur, pour les dépendances affines ? Est-il possible de découper automatiquement les domaines d'itération pour permettre l'expression de plus de parallélisme ? etc. Nous pourrions aussi être tentés de mêler représentation exacte et représentation approchée des dépendances pour pouvoir traiter des codes un peu plus généraux que les programmes à contrôle statique. En fait, la méthodologie utilisée par Feautrier permet tout à fait de mélanger les deux types de représentations. Cependant, plus que des nouveaux travaux théoriques, il nous semble important de mener une étude pratique des différents algorithmes et surtout de comparer, dans la mesure du possible, l'algorithme de Paul Feautrier et celui que nous avons proposé. Une telle comparaison n'est pas évidente à mener. Il nous faudrait en effet un étalon de comparaison, c'est-à-dire un ensemble de programmes banc d'essai, qui représente fidèlement l'ensemble des cas de figure présent « dans la nature », avec leur fréquence d'apparition. Nous aimerions qu'une telle étude nous indique quel algorithme choisir en fonction des caractéristiques du code étudié. Nous voudrions savoir quand nos techniques se révèlent réellement impuissantes et quand il est absolument indispensable d'avoir recours à la méthodologie de Paul Feautrier.

Nous avons écrit qu'il nous semblait important de mener une étude pratique. Il reste cependant à mener des travaux théoriques dans le cadre des boucles totalement permutables et du partitionnement. Dans ce mémoire, nous avons juste montré qu'il était possible d'étendre notre algorithme pour qu'il génère ces boucles permutables. Georges-André Silber a commencé à étudier la liberté dont nous disposons sur le choix des boucles permutables et les critères qui doivent régir ce choix.

Notre algorithme nous offre une grande liberté de choix des ordonnancements car la méthodologie employée permet d'exhiber l'ensemble des ordonnancements qu'il est capable de construire. Nous nous sommes servi de cette liberté pour proposer une méthode de construction d'un ordonnancement compatible avec un placement affine des calculs donné, quand les dépendances du nid sont représentées par un graphe de dépendance réduit polyédrique. Nous avons pu montrer que, dans ce cadre, nous étions à même de construire un ordonnancement compatible à chaque fois qu'il existe un ordonnancement affine compatible avec le placement donné, que cet ordonnancement soit ou non constructible par notre algorithme. Mais que faire quand un tel ordonnancement n'existe pas ?

Finalement, nous avons considéré le problème de l'élimination des fausses dépendances. Nous avons formalisé et étudié la puissance de deux transformations originellement présentées par David Padua et Michael Wolfe. Nous avons également essayé d'intégrer les techniques d'élimination de fausses dépendances et de faux chemins de dépendance au sein des algorithmes de parallélisation, avec un succès pour l'instant très mitigé vu la difficulté du problème. Nous avons essayé d'introduire le moins de mémoire possible, et pour cela de conserver un maximum de fausses dépendances, tout en préservant certaines propriétés de l'espace des solutions du problème d'ordonnancement ou, plus généralement, du problème de détection de parallélisme. Dans une approche diamétralement opposée, Paul Feautrier et Vincent Lefebvre cherchent à introduire le minimum de mémoire nécessaire à la « bonne exécution » d'un programme dont la parallélisation est dictée par un ordonnancement choisi *a priori* à partir du graphe de flot de données.

La parallélisation d'un programme général ou d'un simple nid de boucles est formée d'un ensemble de transformations et de techniques. Si certaines des « étapes » de cette parallélisation sont bien ordonnées entre elles (il semble pour le moins peu réaliste de prétendre détecter du parallélisme avant de pratiquer la moindre analyse de dépendance), l'ordre obtenu n'est que partiel. Pour le problème de l'élimination des fausses dépendances, faut-il, comme Paul Feautrier et Vincent Lefebvre, ordonnancer le nid avant d'essayer d'en éliminer les fausses dépendances au risque d'avoir choisi l'ordonnancement le plus coûteux en mémoire ? Faut-il, comme nous avons tenter de le faire, éliminer les fausses dépendances pendant l'ordonnancement du nid au risque de se heurter à un problème théoriquement trop compliqué ? Faut-il passer en assignation unique, ordonnancer le nid à l'aide d'un ordonnancement obtenu par la deuxième approche puis demander à Paul Feautrier et Vincent Lefebvre d'éliminer la mémoire superflue ? Pour le problème de l'ordonnancement et du placement, faut-il ordonnancer avant de placer les calculs ou après ? Pouvons-nous rechercher un ordonnancement et un placement simultanément ? Nous voulions originellement mener une recherche simultanée. Si nous avons finalement proposé de placer les calculs puis d'ordonnancer le nid, c'est uniquement parce que nous ne savons pas résoudre les autres versions du problème...

Le problème de l'ordonnancement des nids de boucles semble en bonne partie compris et résolu à l'heure actuelle. L'ordonnancement des transformations à appliquer à ces nids pour les paralléliser semble quant à lui un problème beaucoup plus ouvert !

Bibliographie

[1] John Allen, David Callahan et Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual Symposium on Principles of Programming Languages*, pages 63–76, Munich, RFA, Janvier 1987.

[2] John R. Allen et Ken Kennedy. PFC: a program to convert Fortran to parallel form. Rapport de recherche MASC-TR82-6, Rice University, Houston, TX, USA, 1982.

[3] John R. Allen et Ken Kennedy. Automatic loop interchange. *Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6):233–246, Juin 1984.

[4] John R. Allen et Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Octobre 1987.

[5] Jennifer M. Anderson et Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *ACM Sigplan Notices*, 28(6):112–125, Juin 1993.

[6] David F. Bacon, Susan L. Graham et Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), 1994.

[7] Uptal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[8] Uptal Banerjee, Rudolph Eigenmann, Alexandru Nicolau et David Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[9] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali et P. Stodghill. Solving alignment using elementary linear algebra. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau et D. Padua, éditeurs, *Languages and Compilers for Parallel Computing, 7th International Workshop (LCPC '94)*, volume 892 des *LNCS*, pages 46–60, Ithaca, NY, USA, Août 1994. Springer-Verlag.

[10] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15:757–762, Octobre 1966.

[11] Jorge Luis Borgès. *Œuvres complètes,* Tome I. Bibliothèque de la Pléiade. Gallimard, 1994.

[12] Pierre Boulet, Alain Darte, Tanguy Risset et Yves Robert. (pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.

[13] Pierre Boulet, Alain Darte, Georges-André Silber et Frédéric Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. Rapport de recherche 97-17, LIP, ENS-Lyon, France, Juin 1997.

[14] Pierre Boulet, Alain Darte, Georges-André Silber et Frédéric Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Journal of Parallel Computing*, 1997. À paraître dans un numéro spécial intitulé « Languages and Compilers for Parallel Computers ».

[15] Pierre Boulet, Jack Dongarra, Yves Robert et Frédéric Vivien. Tiling for heterogeneous computing platforms. Rapport de recherche UT-CS-97-373, University of Tennessee, Knoxville, USA, Juillet 1997.

[16] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *International Conference of Supercomputing*, pages 407–417, 1988.

[17] David Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, Mars 1987.

[18] Pierre-Yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert et Frédéric Vivien. Retiming DAGs. Rapport de recherche 95-18, LIP, ENS-Lyon, France, Septembre 1995.

[19] Pierre-Yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert et Frédéric Vivien. Retiming DAGs. In *TAU 97*, Austin, Texas, 1997.

[20] Pierre-Yves Calland et Tanguy Risset. Precise tiling for uniform loop nests. In P. Cappello et al, éditeur, *Application Specific Array Processors ASAP 95*, pages 330–337. IEEE Computer Society Press, 1995.

[21] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle.* Mémoire de doctorat, Université de Rennes, Rennes, France, 1993. Numéro 957.

[22] Jean-François Collard. *Parallélisation automatique des programmes à contrôle dynamique.* Mémoire de doctorat, Université Paris VI, Janvier 1995.

[23] Jean-François Collard, Denis Barthou et Paul Feautrier. Fuzzy Array Dataflow Analysis. In *Proceedings of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Santa Barbara, CA, Juillet 1995.

[24] Thomas H. Cormen, Charles E. Leiserson et Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[25] Patrick Le Gouëslier d'Argence. An asymptotically optimal affine schedule on bounded convex polyhedric domains. In *Proceedings of Euro-Par'96 Parallel Processing*, volume 1124 des *LNCS*. Springer Verlag, Août 1996.

[26] Patrick Le Gouëslier d'Argence. *Contribution à l'étude des problèmes d'ordonnancement cyclique multidimensionnel.* Mémoire de doctorat, Université Paris VI, Janvier 1996.

[27] Alain Darte, Leonid Khachiyan et Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[28] Alain Darte et Yves Robert. A graph-theoretic approach to the alignment problem. Rapport de recherche 93-20, LIP, ENS-Lyon, France, Juillet 1993.

[29] Alain Darte et Yves Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 1994.

[30] Alain Darte, Georges-André Silber et Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. Rapport de recherche 96-34, LIP, ENS-Lyon, France, Novembre 1996.

[31] Alain Darte, Georges-André Silber et Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 1997. À paraître dans un numéro spécial.

[32] Alain Darte et Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Rapport de recherche 96-05, LIP, ENS-Lyon, France, Février 1996.

[33] Alain Darte et Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *Journal of Parallel Algorithms and Applications*, 12(1-3):83–112, 1997. Numéro spécial intitulé « Special issue on Optimizing Compilers for Parallel Languages ».

[34] Claude G. Diderich et Marc Gengler. The alignment problem in a linear algebra framework. In *Proceedings of the Hawaï International Conference on System Sciences (HICSS-30), Software Technology Track*, pages 586–595, Wailea, HI, USA, Janvier 1997. IEEE Computer Society Press.

[35] Michèle Dion et Yves Robert. Mapping affine loop nests: New results. In Bob Hertzberger et Guiseppe Serazzi, éditeurs, *High-Performance Computing and Networking, International Conference and Exhibition*, volume 919 des *LNCS*, pages 184–189. Springer-Verlag, 1995. Version étendue : rapport de recherche 94-30, LIP, ENS Lyon, France.

[36] M. Gondran et M. Minoux. *Graphes et algorithmes.* Eyrolles, 1985.

[37] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Septembre 1988.

[38] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[39] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part I: one-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, Octobre 1992.

[40] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II: multi-dimensional time. *Int. J. Parallel Programming*, 21(6):389–420, Décembre 1992.

[41] Paul Feautrier. Towards automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[42] Paul Feautrier. Distribution automatique des données et des calculs. *T.S.I.*, 15(5):529–557, 1996.

[43] High Performance Fortran Forum. High Performance Fortran Language Specification. Rapport de recherche 2.0, Rice University, Janvier 1997.

[44] Michael R. Garey et Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1991.

[45] G. Goff, K. Kennedy et C.W. Tseng. Practical dependence testing. In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, Juin 1991.

[46] D. Grunwald. Data dependence analysis: the λ test revisited. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[47] Junjie Gu, Zhiyan Li et Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing 95*, 1995.

[48] François Irigoin, Pierre Jouvelot et Rémy Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, RFA, Juin 1991.

[49] François Irigoin et Rémy 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.

[50] François Irigoin et Rémy Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. on Principles of Programming Languages*, pages 319–329, San Diego, CA, USA, Janvier 1988.

[51] R.M. Karp, R.E. Miller et S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, Juillet 1967.

[52] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman et D. Wonnacott. *New user interface for Petit and other interfaces: user guide.* University of Maryland, USA, Juin 1995.

[53] X.Y. Kong, D. Klappholz et K. Psarris. The I test: a new test for subscript data dependence. In Padua, éditeur, *Proceedings of 1990 International Conference of Parallel Processing*, Août 1990.

[54] S. Rao Kosaraju et Gregory F. Sullivan. Detecting cycles in dynamic graphs in polynomial time (preliminary version). In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 398–406. ACM Press, Mai 1988.

[55] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, Février 1974.

[56] V. Lefebvre et Paul Feautrier. Automatic storage management for parallel programs. *Journal on Parallel Computing*, 1997.

[57] Z.Y. Li, P.-C. Yew et C.Q. Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, pages 215–224, Crète, Grèce, Juin 1989.

[58] Z.Y. Li, P.-C. Yew et C.Q. Zhu. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–36, Juillet 1990.

[59] Amy W. Lim et Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, Janvier 1997.

[60] The polyhedral loop parallelizer : Loopo. World Wide Web document, URL: `http://brahms.fmi.uni-passau.de/cl/loopo/index.html`.

[61] V. Maslov. Lazy array data-flow dependence analysis. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. POPL*, pages 311–325, Janvier 1994.

[62] Kathryn S. McKinley et Ken Kennedy. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau et D. Padua, éditeurs, *The Sixth Annual Languages and Compilers for Parallelism Workshop*, volume 768 des *LNCS*, pages 301–320. Springer-Verlag, 1993.

[63] Wolfgang Meisl. Practical methods for scheduling and allocation in the polytope model. World Wide Web document, URL: `http://brahms.fmi.uni-passau.de/cl/loopo/doc`.

[64] B. Monien et I.H. Sudborough. Embedding one interconnection network in another. *Computing Suppl.*, 7:257–282, 1990.

[65] David A. Padua et Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Décembre 1986.

[66] K. Psarris, X.Y. Kong et D. Klappholz. Extending the I test to direction vectors. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, RFA, Juin 1991.

[67] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Août 1992.

[68] Patrice Quinton. The systematic design of systolic arrays. In Françoise Fogelman Soulie, Yves Robert et Maurice Tchuente, éditeurs, *Automata Networks in Computer Science*, chapitre 9, pages 229–260. Manchester University Press, 1987.

[69] J. Ramanujam et P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, Octobre 1991.

[70] Sailesh K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, Octobre 1985.

[71] S. Ravindran, A.M. Gibbons et M.S. Paterson. Dense edge-disjoint embedding of complete binary trees in interconnection networks. Rapport de recherche CS-RR-284, University of Warwick, Coventry, UK, Mars 1995.

[72] Vwani P. Roychowdhury. *Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms*. PhD thesis, Stanford University, Décembre 1988.

[73] Robert Schreiber et Jack J. Dongarra. Automatic blocking of nested loops. Rapport de recherche 90-38, The University of Tennessee, Knoxville, TN, Août 1990.

[74] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[75] Georges-André Silber. Recherche de parallélisme à grain moyen dans les nids de boucles. Rapport de Dea, École normale supérieure de Lyon, France, Juin 1996.

[76] Doran K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, Décembre 1993. Also published in IRISA technical report PI 785, Rennes, France; Dec, 1993.

[77] Michael E. Wolf et Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, Octobre 1991.

[78] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA, Octobre 1982.

[79] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge MA, 1989.

[80] Michael Wolfe. *TINY, a loop restructuring research tool.* Oregon Graduate Institute of Science and Technology, USA, Décembre 1990.

[81] Michael Wolfe. The Tiny loop restructuring research tool. In H.D. Schwetman, éditeur, *International Conference on Parallel Processing*, volume II, pages 46–53. CRC Press, 1991.

[82] Michael Wolfe. *High Performance Compilers For Parallel Computing.* Addison-Wesley Publishing Company, 1996.

[83] Yi-Qing Yang. *Tests des dépendances et transformations de programme.* Mémoire de doctorat, École Nationale Supérieure des Mines de Paris, Fontainebleau, France, 1993.

[84] Hans Zima et Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

Annexe A

Publications personnelles

Articles parus dans des revues internationales

- Pierre-Yves Calland, Alain Darte, Yves Robert et Frédéric Vivien. *On the removal of anti and output dependences.* À paraître en 1998 dans *International Journal of Parallel Programming.*

- Alain Darte et Frédéric Vivien. *Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. International Journal of Parallel Programming*, 25(6):447–496, 1997.

- Pierre Boulet, Alain Darte, Georges-André Silber et Frédéric Vivien. *Loop parallelization algorithms: from parallelism extraction to code generation.* À paraître dans un numéro spécial de *Parallel Computing* sur « Languages and Compilers for Parallel Computers ».

- Alain Darte, Georges-André Silber et Frédéric Vivien. *Combining retiming and scheduling techniques for loop parallelization and loop tiling.* À paraître dans un numéro spécial de *Parallel Processing Letters*, 1997.

- Alain Darte et Frédéric Vivien. *Parallelizing nested loops with approximation of distance vectors: a survey. Parallel Processing Letters*, 7(2):133–144, 1997.

- Alain Darte et Frédéric Vivien. *On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Journal of Parallel Algorithms and Applications*, 12(1-3):83–112, 1997. Numéro spécial : « Optimizing Compilers for Parallel Languages ».

- Pierre-Yves Calland, Alain Darte, Yves Robert et Frédéric Vivien. *Plugging anti and output dependence removal techniques into loop parallelization algorithms. Parallel Computing*, 23(1-2):251–266, 1997.

- Alain Darte et Frédéric Vivien. *Revisiting the decomposition of Karp, Miller, and Winograd. Parallel Processing Letters*, 5(4):551–562, Décembre 1995.

Conférences avec publication des actes

- Pierre-Yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert et Frédéric Vivien. *Retiming dags. In : TAU 97.* – Austin, Texas, 1997.

– Alain Darte et Frédéric Vivien. *Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs.* In *Proceedings of PACT'96*, Boston, MA, Octobre 1996. IEEE Computer Society Press.

– Alain Darte et Frédéric Vivien. *On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops.* In *Proceedings of Europar'96*, volume 1123 des *LNCS*, Lyon, France, Août 1996. Springer Verlag.

– Pierre-Yves Calland, Alain Darte, Yves Robert et Frédéric Vivien. *On the removal of anti and output dependences.* In J. Fortes, C. Mongenet, K. Parhi, and V. Taylor, editors, *Application Specific Systems, Architectures and Processors*, pages 353–364. IEEE Computer Society Press, 1996.

– Alain Darte et Frédéric Vivien. *Revisiting the decomposition of Karp, Miller and Winograd.* In P. Cappello, C. Mongenet, G.R. Perrin, P. Quinton, and Y. Robert, editors, *Application Specific Array Processors 95*, pages 13–25. IEEE Computer Society Press, 1995.

– Alain Darte et Frédéric Vivien. *A classification of nested loops parallelization algorithms.* In *INRIA-IEEE Symposium on Emerging Technologies and Factory Automation*, pages 217–224. IEEE Computer Society Press, 1995.

Rapports de recherche

– Pierre Boulet, Jack Dongarra, Yves Robert et Frédéric Vivien. *Tiling for heterogeneous computing platforms.* Rapport de recherche UT-CS-97-373, Université du Tennessee, Knoxville, USA, Juillet 1997.

– Pierre Boulet, Alain Darte, Georges-André Silber et Frédéric Vivien. *Loop parallelization algorithms: from parallelism extraction to code generation.* Rapport de recherche 97-17, LIP, ENS-Lyon, France, Juin 1997.

– Alain Darte, Georges-André Silber et Frédéric Vivien. *Combining retiming and scheduling techniques for loop parallelization and loop tiling.* Rapport de recherche 96-34, LIP, ENS-Lyon, France, Novembre 1996.

– Pierre-Yves Calland, Alain Darte, Yves Robert et Frédéric Vivien. *Plugging anti and output dependence removal techniques into loop parallelization algorithms.* Rapport de recherche 96-13, LIP, ENS-Lyon, France, Juin 1996.

– Alain Darte et Frédéric Vivien. *Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs.* Rapport de recherche 96-06, LIP, ENS-Lyon, France, Avril 1996.

– Alain Darte et Frédéric Vivien. *On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops.* Rapport de recherche 96-05, LIP, ENS-Lyon, France, Février 1996.

– Pierre-Yves Calland, Alain Darte, Yves Robert et Frédéric Vivien. *On the removal of anti and output dependences.* Research Report 96-04, LIP, LIP, ENS-Lyon, France, Février 1996.

– Pierre-Yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert et Frédéric Vivien. *Retiming DAGs.* Rapport de recherche 95-18, LIP, ENS-Lyon, France, Mai 1995.

– Alain Darte et Frédéric Vivien. *A classification of nested loops parallelization algorithms.* Rapport de recherche 95-11, LIP, ENS-Lyon, France, Mai 1995.

– Alain Darte et Frédéric Vivien. *Automatic parallelization based on multi-dimensional scheduling.* Rapport de recherche 94-24, LIP, ENS-Lyon, France, Septembre 1994.

Living is easy with time on my side

Cos I want to leave where the sun meets the sky

I am a doctor !

Sun hits the sky, **Supergrass**