Outils pour la parallélisation automatique

Pierre Boulet

N° d'ordre : 13

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 Pierre BOULET

Titre :
Outils pour la parallélisation automatique

Soutenue le 18/01/96

Après avis de : Monsieur Thomas BRANDES
Monsieur Jean-Luc DEKEYSER

Devant la Commission d'examen formée de :

Monsieur Thomas BRANDES
Monsieur Michel COSNARD
Monsieur Jean-Luc DEKEYSER
Monsieur Bernard DION
Monsieur François IRIGOIN
Monsieur Yves ROBERT

Je tiens à exprimer ici ma gratitude envers les membres du jury :

– à Thomas Brandes pour les discussion fructueuses et les rapports toujours cordiaux que nous avons eu, même quand un stupide bug récalcitrant nous a occupé plusieurs semaines, et aussi pour avoir rapporté une thèse en français qui n'est pas sa langue maternelle ;

– à Michel Cosnard pour l'intérêt qu'il marque au travail de chaque thésard du LIP et son énergie débordante ;

– à Jean-Luc Dekeyser d'avoir rapporté sur ma thèse et de son accueil au LIFL, ce qui a été ma première expérience déterminante de recherche ;

– à Bernard Dion de sa participation à mon jury et à l'intérêt qu'il porte à Bouclettes ;

– à François Irigoin pour ses questions toujours stimulantes ;

– et enfin à Yves Robert qui a été un directeur de thèse passionné, passionnant, plein d'humour et de conseils précieux, bref, un modèle à suivre.

Remerciements

Je me plie ici bien volontier à l'exercice des remerciements. Je voudrais remercier en premier lieu la personne sans laquelle cette thèse n'aurait pas été ce qu'elle est, à savoir Yves qui a su être un guide mais aussi un mentor et un ami et qui sous ses pointes d'humour cache toujours de très bons conseils.

Ensuite, je remercie Alain et Tanguy qui m'ont ouvert la voie avec leurs études sans lesquelles Bouclettes n'est rien et avec qui les conversations sont toujours stimulantes. Pour finir le tour des membres de l'équipe, merci à Michèle, Frédo et Pierre-Yves pour la bonne ambiance conciliant décontraction et travail.

Merci à José pour son accueil à Purdue où j'ai découvert la parallélisation automatique et où j'ai appris beaucoup de choses, et pas seulement en informatique... Merci à Thomas pour sa patience et son magnifique outil ADAPTOR.

Merci à tous les thésards du LIP, et en particulier à Jean-Marc, qui est bien plus qu'un copain, Cyril, avec qui j'ai partagé un bureau et la date de la soutenance de thèse, Xavier, Richard, Bernard, Thierry et Jaco pour leurs délires au coin café ou sur `thesards.lip`, et aussi Arnaud, Laurent, Didier, Jean-Marc, Thibault et tous les autres.

Merci aussi aux enseignants au contact de qui j'ai beaucoup appris, que ce soit en temps qu'étudiant jusqu'au DEA, qu'en temps que collègue après, et principalement à Luc et Odile. Merci aussi au CIES et Jean-Michel Caron et Véronique De Marchis pour leur dévouement et leur foi dans le monitorat.

Merci à tous les joueurs de l'équipe de rugby de l'ENS avec qui j'ai partagé tant de défaites (`:-)`) et de victoires et bien sûr à Jean-Claude sans qui le sport à l'ENS ne serai pas si vivant et qui fait qu'on peut avoir un *mens sana in corpore sano* dans cette belle école.

Merci aussi à tous les gens avec qui j'ai fait des jeux de rôle pendant lesquels on a souvent parlé science et en particulier Denis, Jean-Marc et Hélène, et Matthieu et Joëlle.

Et enfin et bien sûr, merci à toute ma famille, mes frères, mes parents et mes grands parents qui m'ont toujours soutenu même s'il n'ont pas toujours compris tout ce que je faisait. J'ai ici une pensée particulière pour ma grand mère qui ne m'aura pas vu docteur et à qui ça aurait fait tant plaisir.

En bref, merci à tous ceux que j'ai cité et aussi à ceux que j'ai oublié ici.

Table des matières

1 Introduction **11**

2 HPF et la parallélisation automatique **13**
2.1 Introduction . . . 13
2.2 Tour d'horizon (optimiste) . . . 13
2.3 Présentation d'HPF . . . 14
2.3.1 Directives de placement de données . . . 14
2.3.2 Constructions de spécifications d'opérations parallèles . . . 16
2.3.3 Autres fonctionnalités d'HPF . . . 17
2.4 Ce que ne peut pas faire HPF . . . 18
2.5 Parallélisation automatique . . . 20
2.5.1 Pourquoi les nids de boucles ? . . . 20
2.5.2 Méthode . . . 20
2.5.3 Analyse des dépendances . . . 21
2.5.4 Ordonnancement . . . 22
2.5.5 Allocation . . . 23
2.5.6 Partitionnement et génération de code . . . 24
2.6 Conclusion . . . 25

3 Motivation : étude expérimentale **29**
3.1 Introduction . . . 29
3.2 Méthodes d'ordonnancement considérées . . . 29
3.3 La MasPar MP-1 . . . 31
3.4 Étude de cas comparative . . . 32
3.4.1 Ordonnancement linéaire . . . 32
3.4.2 Ordonnancement par instruction . . . 33
3.4.3 Une approche non uniformisée . . . 35
3.4.4 La routine de MasPar . . . 36
3.4.5 Comparaisons de temps . . . 37
3.5 Partitionnement . . . 38
3.5.1 Partitionnement immédiat . . . 38
3.5.2 Un partitionnement plus élaboré . . . 38
3.5.3 La routine de MasPar . . . 39
3.5.4 Comparaisons de performances . . . 40
3.6 Conclusion . . . 40

4 Bouclettes **41**
4.1 Introduction . . . 41
4.1.1 Qu'est-ce que Bouclettes ? . . . 41
4.1.2 Autres outils . . . 41
4.2 Analyse des données et extraction du parallélisme . . . 42
4.2.1 Le langage d'entrée . . . 42
4.2.2 Phases de la parallélisation automatique . . . 42
4.2.3 Analyse de dépendances . . . 43
4.2.4 Ordonnancement . . . 43
4.2.5 Placement . . . 45
4.2.6 La génération du code . . . 46
4.3 Implémentation . . . 47
4.3.1 Programmation . . . 47
4.3.2 Interface utilisateur . . . 47
4.4 Génération du code dans Bouclettes . . . 48
4.4.1 Codage de l'ordonnancement linéaire . . . 48
4.4.2 Codage de l'ordonnancement linéaire décalé . . . 53
4.4.3 Commentaires . . . 58
4.5 Exemple détaillé . . . 58
4.5.1 Le programme d'entrée . . . 58
4.5.2 Ordonnancement linéaire sans redistribution . . . 59
4.5.3 Ordonnancement linéaire avec redistribution . . . 59
4.5.4 Ordonnancement linéaire décalé . . . 60
4.6 Évaluation expérimentale du code produit par Bouclettes . . . 62
4.6.1 Description rapide d'ADAPTOR . . . 64
4.6.2 Résultats . . . 65
4.6.3 Commentaires . . . 74
4.7 Conclusion . . . 74

5 Le pavage ultime ? **75**
5.1 Introduction . . . 75
5.1.1 Pavage : motivation . . . 75
5.1.2 Pavage: définition . . . 76
5.1.3 Organisation du chapitre . . . 78
5.2 Critère pour un pavage « optimal » . . . 79
5.2.1 Que voulons-nous optimiser ? . . . 79
5.2.2 Travaux liés . . . 80
5.2.3 Un critère extensible . . . 83
5.3 Minimiser $V(H)$ est un problème combinatoire . . . 85
5.3.1 Le cône $\Gamma = \{H | HD \geq 0\}$ . . . 85
5.3.2 Cas particulier : D est une matrice carrée . . . 86
5.3.3 Cas général : D est une matrice $n \times m$ de rang plein . . . 88
5.4 Conclusion . . . 91

6 Évaluation d'expressions de tableaux **93**
6.1 Introduction . . . 93
6.1.1 Les règles du jeu . . . 93
6.1.2 Expressions de tableaux d'HPF . . . 95
6.1.3 Organisation du chapitre . . . 97
6.2 Revue de travaux précédents . . . 97
6.3 NP-complétude du problème . . . 98
6.3.1 Définition du problème simplifié . . . 98
6.3.2 Preuve du théorème . . . 99
6.4 Heuristiques . . . 105
6.4.1 Introduction . . . 105
6.4.2 Quand toutes les feuilles sont dans des positions différentes . . . 105
6.4.3 Quand des feuilles peuvent partager la même localisation . . . 109
6.4.4 Extension . . . 113
6.5 Conclusion . . . 113

7 Conclusions et perspectives **117**

Bibliographie **119**

Publications personnelles **125**

Chapitre 1

Introduction

Avant de parler d'outils pour la parallélisation automatique, il convient de définir la parallélisation automatique et de motiver la construction de tels outils. Les deux dernières décennies ont vu l'émergence d'une nouvelle classe d'ordinateurs : les ordinateurs *parallèles*. La spécificité de ces machines est qu'elles possèdent plusieurs unités de calcul qui fonctionnent en parallèle, c'est-à-dire en même temps, pour résoudre plus rapidement de plus gros problèmes. En effet, ce type d'architecture permet, à relativement peu de frais, d'augmenter la puissance de traitement, non seulement en réduisant le temps nécessaire pour résoudre un problème donné, mais aussi en permettant la résolution de problèmes plus gourmands en mémoire.

Les principaux utilisateurs de ces machines sont les scientifiques qui ont toujours besoin de plus de puissance de calcul. Or, ces mêmes scientifiques ont déjà écrit des millions de lignes de code (principalement Fortran) pour programmer « l'ancienne » génération de machines. Essentiellement *utilisateurs* de l'informatique, ils n'ont pas forcément envie de s'investir dans de nouveaux langages de programmation, d'autant plus que les machines parallèles, devenues plus puissantes, sont aussi plus difficiles à programmer. En effet, il n'est pas aisé de décomposer le programme à exécuter en *tâches* qui seront chacune affectées aux *processeurs* (les unités de calcul) et qui coopéreront pour mener à bien la résolution du problème. Des difficultés importantes apparaissent lors de la détection du parallélisme ; et quand bien même cette étape est franchie avec succès, l'écriture du programme peut devenir un vrai défi quand le programmeur doit décrire toutes les communications qui ont lieu pour coordonner le travail des différentes tâches. Ces difficultés sont tellement importantes qu'il n'est pas rare de voir un programme aller moins vite en utilisant plusieurs processeurs qu'en en utilisant qu'un seul mais sans avoir à gérer de communications.

Il s'agit donc de rendre ces machines parallèles plus faciles à programmer et de permettre aux utilisateurs de tirer un parti maximum de leur puissance potentielle.

Langages de programmation parallèle

Une approche visant à simplifier la tâche de l'utilisateur est de l'aider à programmer en lui fournissant des langages de programmation parallèle plus faciles à utiliser. Les premiers langages parallèles disponibles étaient des langages à communications par messages. Ces langages d'assez bas niveau augmentent le C ou le Fortran avec des instructions permettant l'envoi et la réception de messages. Bien qu'ils soient encore très fréquemment employés car ils sont en général bien optimisés, ces langages sont relativement difficiles à manipuler car l'utilisateur doit programmer la répartition des données et des calculs sur les processeurs disponibles. Avec l'apparition de bibliothèques de gestion des communications standards (PVM et MPI, par exemple), un autre des inconvénients de ces lan-

gages tend à disparaître : ils deviennent plus facilement portables d'une machine à l'autre. D'autres bibliothèques, de calcul scientifique cette fois, viennent à l'aide du programmeur. Ces bibliothèques implémentent de manière optimisée les noyaux de calcul scientifique les plus couramment utilisés.

Une nouvelle famille de langages a fait son apparition un peu plus tardivement, ce sont les langages dits « data-parallèles ». Ce paradigme de programmation libère l'utilisateur de la gestion des communications. Le programmeur décrit la manière de répartir ses données sur les processeurs de la machine et le compilateur génère les communications induites par les calculs. Ce style de programmation est beaucoup plus simple pour un utilisateur profane mais laisse moins de liberté et de contrôle au programmeur expérimenté qui est prêt à passer une grande partie de son temps de programmation à l'optimisation de son code. Le représentant le plus important (à mon avis) de cette famille de langages est High Performance Fortran (HPF) car il a été conçu par un forum international constitué, à la fois, de spécialistes des langages, de spécialistes d'architecture de machines, de fabriquants et d'utilisateurs.

Cependant, malgré tous les efforts qui ont porté sur les langages de programmation parallèle, certains algorithmes restent difficiles à analyser pour en extraire le parallélisme.

Parallélisation automatique

En complément des langages parallèles, le programmeur a le choix d'utiliser la parallélisation automatique. Cette parallélisation consiste à adapter un programme écrit pour une machine *séquentielle* (qui n'a qu'un processeur) à une machine parallèle. L'intérêt de l'automatisation de cette parallélisation par un programme appelé *paralléliseur* est qu'on pourrait alors réutiliser tout le code déjà écrit en Fortran pour machines séquentielles, après parallélisation, sur des machines parallèles.

Où en est-on des méthodes de parallélisation automatique? Les systoliciens ont développé des méthodes de génération automatique de circuits systoliques. Ces méthodes d'analyse de dépendance, d'ordonnancement, d'allocation et de partitionnement se sont révélées particulièrement bien adaptées à la génération automatique de code parallèle. De nombreuses recherches sont menées pour construire des méthodes de plus en plus performantes et générales.

Plan de la thèse

Après une présentation détaillée de la parallélisation automatique et d'HPF (chapitre 2), une étude expérimentale (chapitre 3) de parallélisation sur un exemple motive la construction d'un paralléliseur automatique. Le chapitre 4 est consacré à la réalisation d'un logiciel, Bouclettes, qui parallélise automatiquement une classe réduite de programmes (les nids de boucles uniformes qui utilisent des translations comme accès aux tableaux de données) en HPF. J'insiste surtout sur la partie génération de code HPF, qui est la partie la plus novatrice de ce programme. Le code produit par Bouclettes est ensuite évalué expérimentalement. Outre la réalisation de Bouclettes, ma contribution au domaine est aussi théorique avec l'étude d'un partitionnement des données appelé « pavage par parallélépipèdes » (chapitre 5) et celle de l'optimisation des calculs d' « expressions de tableaux » du langage High Performance Fortran (chapitre 6). Le pavage est une technique permettant d'optimiser la taille des tâches réparties sur les processeurs pour diminuer le temps passé en communications. L'évaluation d'expressions de tableaux, quant à elle, est une étape d'optimisation du compilateur parallèle. Je conclus enfin dans le chapitre 7 et j'y présente quelques perspectives.

Chapitre 2

HPF et la parallélisation automatique

2.1 Introduction

Le but de ce chapitre[1] est de présenter au lecteur non spécialiste les techniques de parallélisation automatique : quel est leur champ d'application, et comment se situent-elles par rapport au développement des compilateurs HPF (High Performance Fortran) ?

Nous commençons par parler brièvement des machines parallèles, côté architecture et côté environnement de programmation. Puis nous décrivons HPF, et ce qu'il peut apporter à l'utilisateur de ces machines. Nous montrons ensuite ce qu'HPF ne peut pas faire, à savoir paralléliser efficacement un nid de boucles imbriquées (même tout simple), et comment les techniques de parallélisation automatique peuvent (pourront) remédier à cette inexcusable lacune !

2.2 Tour d'horizon (optimiste)

Les environnements de programmation des machines parallèles ont fait de gros progrès en 10 ans. Il suffit de comparer la situation d'hier et celle d'aujourd'hui pour titrer « *cool machines* » comme dans l'éditorial de *IEEE Parallel and Distributed Technology (Spring* 94*)*.

Où en est le matériel ? Le point technologique est aujourd'hui facile à bâtir: une architecture à mémoire distribuée, basée sur des processeurs du commerce de type RISC haut de gamme et un réseau d'interconnexion, soit très simple pour une orientation parallélisme à gros grain, soit à très forte valeur ajoutée pour la performance et la gestion matérielle d'une mémoire virtuellement partagée.

On se trouve ici devant le choix entre un réseau de stations de travail dédiées connectées par un réseau construit avec des éléments « du commerce », soit un réseau très spécialisé, performant et donc cher, qui accueille des processeurs puissants « du commerce » encore.

Où en est le logiciel ? Les environnements ont beaucoup progressé. La plupart des architectures sont proposées avec un micro-noyau « Unix light » sur chaque processeur.

Du coup, le débogage est facilité et les outils usuels du type `dbx` sont utilisables. Pour l'étude et l'amélioration des performances, l'analyse post-mortem (après exécution) est un passage incontournable. Les outils classiques comme `prof` sont disponibles sur la plupart des plate-formes, souvent même agrémentés de systèmes de visualisation très performants.

1. Écrit en collaboration avec Michèle Dion.

Quant à la programmation, sur certain ordinateurs parallèles, l'utilisateur peut s'abstraire de la vision « mémoire distribuée » grâce à des mécanismes (matériels ou logiciels) de mémoire virtuelle partagée (CONVEX Exemplar, CRAY T3D) et programmer simplement. Par contre, même dans ces cas favorables où les communications ne sont pas explicites, une bonne connaissance de l'architecture est nécessaire pour exploiter au mieux les machines. En effet, les primitives de répartition des données restent à la charge du programmeur. Les compilateurs parallèles HPF (High Performance Fortran) ne garantissent pas toujours de bonnes performances (nous verrons pourquoi dans la suite). De toute manière ils reportent une grande partie de la complexité des manipulations indispensables à une parallélisation performante dans des annotations astucieuses laissées à la charge du programmeur.

Du coté de la programmation parallèle explicite, le maître mot est *standardisation* : les bibliothèques de procédures de communications sont des standards de facto (PVM) ou par construction (MPI). Et pour le calcul scientifique, ScaLaPack joue le rôle de bibliothèque numérique incontournable. Ce point est d'importance car un code parallèle est maintenant facilement portable de la station de travail où a lieu le développement à la machine cible. Les efforts de parallélisation sont à la fois moindres et plus productifs.

2.3 Présentation d'HPF

Nous tentons l'impossible ici, à savoir présenter HPF en quelques pages. Nous renvoyons bien sûr à [KLS+94, Fos95] pour une présentation complète.

High Performance Fortran est un langage *data-parallèle*. Le *data-parallélisme* désigne le parallélisme qui provient de l'exécution concurrente de la même opération appliquée à une partie ou à tout un ensemble de données. Un programme data-parallèle est une séquence de telles opérations. Pour pouvoir écrire un tel programme, il faut procéder en deux phases : décomposer les données pour les répartir sur les processeurs puis spécifier les opérations parallèles à effectuer sur ces données partitionnées (réparties sur les processeurs).

Fortran 90 propose déjà des constructions pour exprimer des opérations parallèles simples avec la notation compacte de sections de tableaux. HPF étend Fortran 90 en lui apportant de nouvelles constructions parallèles et des directives de placement de données.

2.3.1 Directives de placement de données

Les directives[2] de placement de données sont le moyen dont dispose le programmeur pour contrôler la répartition des données sur les processeurs de sa machine parallèle. Ce placement des données impose aussi le placement des calculs car HPF utilise la règle des *écritures locales*. Cette règle veut dire que le processeur qui effectue le calcul est celui qui *possède* la donnée calculée.

Le placement des données se fait en trois étapes :

1. La mise en relation d'ensembles de données se fait au moyen de la directive `ALIGN`. Cette directive permet de spécifier les éléments de tableaux qui doivent être, si possible, alloués au même processeur. La forme générale de cette directive est :

   `!HPF$ ALIGN` *tableau* `WITH` *cible*

2. Attention les directives ne sont que des conseils donnés au compilateur, il n'est pas obligé d'en tenir compte!

Ceci indique que le *tableau* spécifié doit être aligné avec la *cible*. Cette cible peut être un autre tableau ou ce qu'on appelle un « template » qui est en fait un tableau fictif sur lequel sont alignés d'autres tableaux. Par exemple:

```
!HPF$ TEMPLATE T(n+1,n+1)
!HPS$ ALIGN A(i,j) WITH T(i,j)
!HPS$ ALIGN B(i,j) WITH T(j+1,i+1)
```

veut dire que pour toutes les valeurs possibles de `i` et `j`, les éléments `B(i,j)` et `A(j+1,i+1)` seront stockés dans la mémoire du même processeur.

2. Le partitionnement des données sur les processeurs (virtuels) se fait par la directive `DISTRIBUTE`. Cette directive permet de définir comment les données seront réparties dans les mémoires d'un tableau de processeurs virtuels défini par la directive `PROCESSORS`. La forme générale de ces directives est :

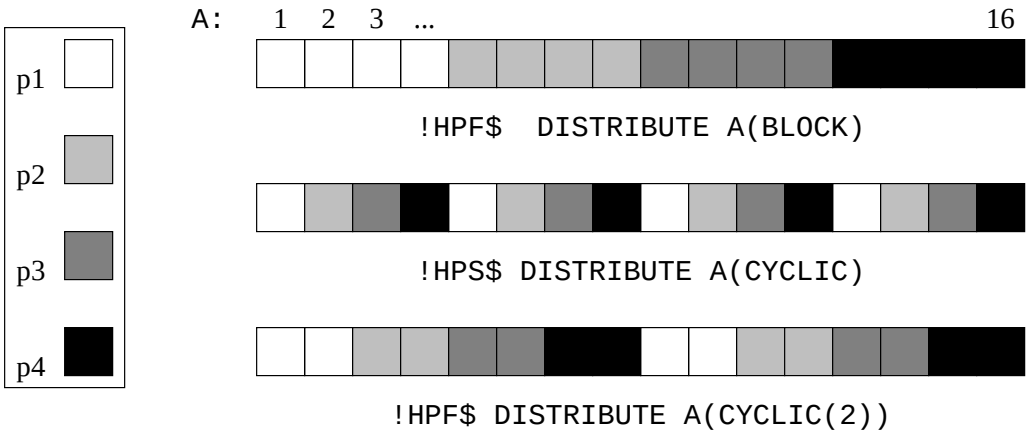
   `!HPF$ PROCESSORS` *tableau*
   `!HPF$ DISTRIBUTE` *liste-de-tableaux* `[ONTO` *nom-du-tableau*`]`

   Chaque tableau de la liste de tableaux à distribuer peut l'être de différentes manières. Le programmeur spécifie la manière de distribuer chaque dimension du tableau. Notons N le nombre d'éléments du tableau dans la direction considérée et P le nombre de processeurs virtuels dans cette même direction. Chaque dimension peut être distribuée de 3 façons différentes :

`*`	:	pas de distribution
`BLOCK(`n`)`	:	distribution en blocs (défaut: $n = N/P$)
`CYCLIC(`n`)`	:	distribution cyclique (défaut: $n = 1$)

   La distribution par blocs donnera dans chaque mémoire un bloc contigu de taille N/P, alors qu'une distribution cyclique alloue tous les éléments d'indices congrus modulo P au même processeur. L'argument entier optionnel de `BLOCK` et `CYCLIC` donne le nombre d'éléments dans un bloc. La figure 2.1 donne des exemples de distributions d'un tableau à une dimension de taille 16 sur 4 processeurs.

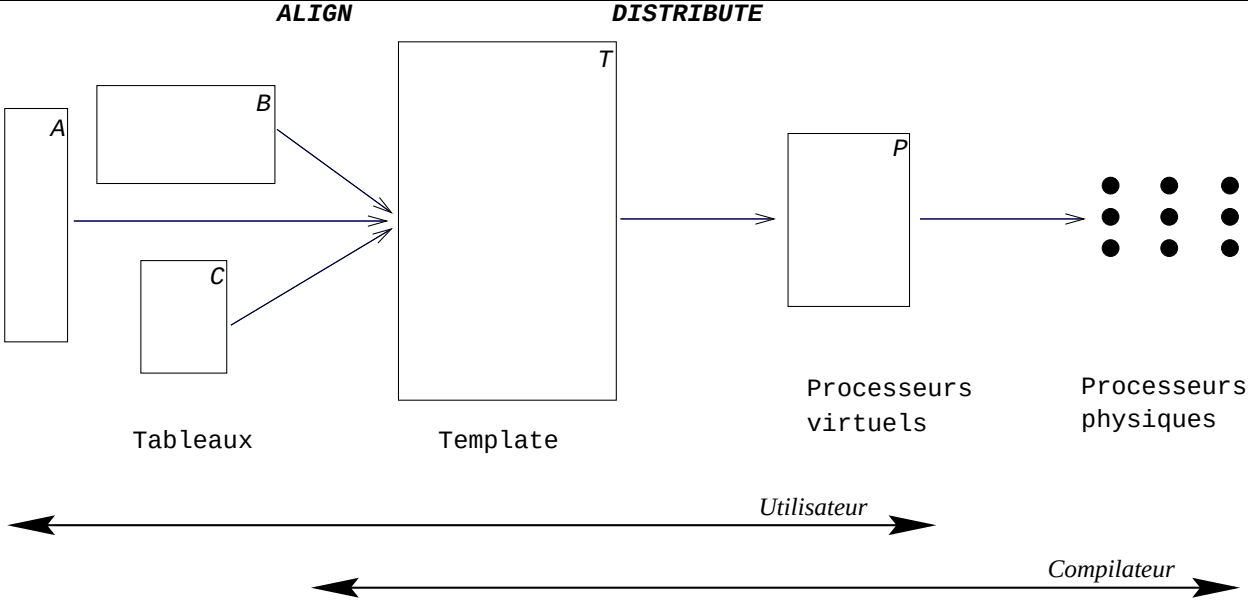
FIG. **2.1 -** Exemples de distributions de données



3. La dernière étape est le placement du tableau de processeurs virtuels sur les processeurs physiques de la machine cible. Ceci est réalisé par le compilateur HPF d'une façon non spécifiée et dépendante de la machine.

La figure 2.2 résume les différentes étapes du placement des données.

FIG. **2.2 -** Placement des données avec HPF



2.3.2 Constructions de spécifications d'opérations parallèles

Les constructions décrites ici permettent au programmeur de spécifier des opérations data-parallèles sur les tableaux répartis grâce aux directives décrites ci-dessus.

La notation de tableaux de Fortran 90

Fortran 90 autorise l'application d'un certain nombre d'opérations *scalaires* à des tableaux. Ceci se traduit par l'application de ces opérations à chaque élément des tableaux mis en jeux. Si plusieurs tableaux interviennent dans le calcul de l'opération, ils doivent être superposables (même dimension et même taille dans chaque dimension). Pour des exemples, voir le programme 2.1.

PROG. **2.1** Exemples d'expressions de tableaux de Fortran 90

```
real A(10,20), B(10,20)
logical L(10,20)
A = A + 1.0    ! Ajoute 1 à chaque élément de A
A = SQRT(A)    ! Prend la racine carrée de chaque élément de A
L = A .EQ. B   ! L(i,j) vaut .true. si A(i,j)=B(i,j)
               ! et .false. sinon
```

Une section de tableau de la forme souhaitée peut être substituée à un tableau dans une telle expression de tableaux. Une section de tableau est spécifiée dans chaque dimension par un triplet

borne-inférieure : *borne-supérieure* : *pas*. Par exemple, la section `A(3,1:7:3)` contient les éléments de la ligne 3, colonnes 1, 4 et 7 et est superposable à `B(6,2:7:2)`.

De plus, Fortran 90 possède des fonctions *de transformation* qui retournent un scalaire ou un tableau qui dépend des valeurs de plusieurs éléments du ou des tableaux arguments. On peut citer en exemple `SUM(A)` qui calcule la somme des éléments du tableau `A`, ou `MATMUL(A,B)` qui calcule le tableau produit des tableaux `A` et `B`.

Les extensions d'HPF

L'instruction `FORALL`. Elle permet des affectations plus générales à une section de tableau. Cette instruction a la forme générale :

`FORALL (`*triplet*`, ..., `*triplet*`, `*masque*`)` *affectation*

où *triplet* a la forme générale :

indice `=` *borne-inférieure* : *borne-supérieure* : *pas*

(où : *pas* est optionnel) et spécifie un ensemble d'indices.

L'ordre d'évaluation d'une instruction `FORALL` est le suivant :

1. L'expression à droite de l'affectation est évaluée pour toutes les valeurs d'indices décrites par les *triplets* et non cachées par le *masque*. Ces évaluations peuvent être faites dans n'importe quel ordre (ou en parallèle).
2. Les affectations sont ensuite réalisées, à nouveau dans n'importe quel ordre.

La directive `INDEPENDENT`. Elle sert à indiquer au compilateur que les itérations d'une boucle `DO` peuvent être effectuées indépendamment les unes des autres, c'est à dire dans n'importe quel ordre ou en parallèle sans changer le résultat calculé.

Par exemple, dans le morceau de programme 2.2, la boucle extérieure est « indépendante » alors que la boucle intérieure ne l'est pas.

PROG. **2.2** Illustration de la directive `INDEPENDENT`

```
!HPS$  INDEPENDENT
       do i=1,n      ! boucle sur i indépendante
          do j=2,n   ! boucle sur j non indépendante
             B(i)=B(i)+A(i,j)
          enddo
       enddo
```

2.3.3 Autres fonctionnalités d'HPF

HPF propose également des moyens de passer la distribution des arguments aux paramètres formels des procédures et de changer la distribution d'un tableau dans le cours du programme. HPF possède aussi des fonctions permettant de connaître le nombre de processeurs et la topologie du réseau d'interconnexion de la machine cible.

La construction `FORALL` est une extension de l'instruction `FORALL` qui permet d'exprimer plus de programmes parallèles. Des procédures déclarées `PURE` ne font pas d'effet de bord et peuvent ainsi être appelées dans des constructions `FORALL` pour encore plus d'expressivité.

Enfin, HPF ajoute d'autres fonctions data-parallèles dans sa bibliothèque et autorise l'appel à des procédures « *extrinsèques* », c'est à dire écrites dans un autre langage, par exemple des procédures écrites avec une bibliothèque comme MPI pour le passage de messages.

2.4 Ce que ne peut pas faire HPF

HPF requiert de l'utilisateur

– des directives pour la distribution des calculs et des données : `ALIGN, DISTRIBUTE`

– des directives pour identifier les boucles parallèles : `FORALL, INDEPENDENT`

C'est donc l'utilisateur qui a la charge d'extraire le parallélisme potentiel et de déterminer les bons alignements de tableaux. Quelquefois, c'est facile, comme dans l'exemple de processus de diffusion discrète unidimensionnel (voir le programme 2.3).

PROG. **2.3** Processus de diffusion discrète unidimensionnel

```
do i = 1, M
   do j = 1, N
      u(i,j) = f(u(i-1,j), u(i,j-1))
   enddo
enddo
```

Aucune des deux boucles n'est parallèle, en raison des dépendances: pour calculer le point (i, j), il faut avoir déjà calculé le point $(i-1, j)$ (donc la boucle sur i est séquentielle) et le point $(i, j-1)$ (donc la boucle sur j est séquentielle). Mais on voit que toutes les itérations (i, j) telles que $i + j = Cte$ sont indépendantes. Pour tirer parti de ce parallélisme à peine caché, on pose

$$\begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

et on *réécrit* le nid (voir le programme 2.4).

PROG. **2.4** Réécriture du processus de diffusion discrète unidimensionnel

```
do t = 2, M + N
   forall (p = max(1, t-N) : min(t, N))
          u(t-p, p) = f(u(t-p-1, p), u(t-p, p-1))
enddo
```

Il ne reste qu'à *distribuer* les colonnes de u aux processeurs pour obtenir une version qu'un compilateur HPF saura paralléliser avec efficacité.

Les transformations effectuées sur le nid d'origine de deux boucles, pour simples qu'elles soient en apparence, font tout de même appel à un arsenal mathématique sophistiqué. Les paramètres M et N ne sont pas instanciés à la compilation, et la réécriture du nid, notamment le calcul des nouvelles bornes de boucles, n'est pas immédiate. Également, le choix d'une matrice de changement de base $(i, j) \longrightarrow (t, p)$ de déterminant 1 (on dit *unimodulaire*) n'est pas innocent ...

D'autres exemples font la part moins belle à l'utilisateur. Ainsi le petit nid décrit dans le programme 2.5, emprunté à Peir et Cytron [PC89], cache bien son parallélisme.

– Quels points (i, j) peuvent être exécutés en parallèle ?

– Comment aligner les tableaux a, b, c et d pour minimiser les communications ?

Difficile de répondre directement ! Pourtant, ce deuxième nid de boucles ressemble au précédent: il est parfaitement imbriqué, et les dépendances entre les instructions sont faciles à calculer: ce sont des translations.

PROG. **2.5** Exemple de Peir et Cytron

```
do i = 0, N
   do j = 0, N
      {Instruction S1:} a(i,j) = b(i,j-6) + d(i-1,j+3)
      {Instruction S2:} b(i+1,j-1) = c(i+2,j+5)
      {Instruction S3:} c(i+3,j-1) = a(i,j-2)
      {Instruction S4:} d(i,j-1) = a(i,j-1)
   enddo
enddo
```

Pour un nid de boucles plus général, comme on en trouve en calcul scientifique, les choses se corsent encore. Regardons la factorisation de Choleski (programme 2.6).

PROG. **2.6** Factorisation de Choleski

```
do k = 1, N
   s = a(k,k)
   do i = 1, k-1
      s = s - a(k,i)**2
   enddo
   p(k) = 1.0 / sqrt(s)
   do j = k+1, N
      s = a(k,j)
      do i = 1, k-1
         s = s - a(j,i) * a(k,i)
      enddo
      a(j,k) = s * p(k)
   enddo
enddo
```

Ici, les tableaux n'ont pas tous la même dimension, les boucles ne sont pas parfaitement imbriquées, et les dépendances entre instructions font appel à des expressions affines. Bien sûr, l'algorithmicien qui connaît la factorisation de Choleski saurait indiquer les boucles parallèles, ou à défaut, proposer une réécriture du nid. Mais clairement, il suffirait de changer l'instruction `s = s - a(j,i) * a(k,i)` en `s = s - a(j+k,i-j) * a(k,i+j+k)` pour le laisser désemparé.

C'est le rôle des techniques de parallélisation automatique que de générer automatiquement des transformations espace-temps qui permettent d'une part de construire explicitement des boucles parallèles, et d'autre part de déterminer la répartition adéquate des données sur les processeurs (allocation et ordonnancement sont intimement liés) : directives de distribution et d'alignement.

Bien sûr, ces techniques ne s'appliquent, pour l'instant, qu'à des programmes très simples : les nids de boucles à contrôle statique (voire dynamique [Col95]) et dépendances uniformes et affines, pour reprendre la terminologie de Feautrier [Fea92a, Fea92b]. Nous détaillons état de l'art et perspectives dans le paragraphe suivant.

2.5 Parallélisation automatique

2.5.1 Pourquoi les nids de boucles ?

Les nids de boucles sont au coeur des compilateurs-paralléliseurs pour super-ordinateurs. Leur importance en termes d'applications est claire : pour de nombreux programmes scientifiques, le temps passé dans quelques boucles est une grande fraction du temps d'exécution total, et le parallélisme potentiel en est souvent très large. D'une part, la restriction aux nids de boucles uniformes ou affines ne constitue pas une limitation trop pénalisante : cette classe de programmes (qui contient notamment la plupart des programmes de traitement du signal [Kun88] ou d'algèbre linéaire numérique [GL89]) est à l'origine de nombreuses recherches depuis les travaux précurseurs de Lamport [Lam74], citons par exemple [SF91, HS91, IT88, LHS90, Lis90, PC89, Pol88, ST91] et [LER92, ZC90].

D'autre part, la structure régulière et répétitive des nids de boucles affines facilite la mise en œuvre des techniques d'analyse de dépendance et la recherche de fonctions d'ordonnancement et d'allocation. Le problème général de l'ordonnancement optimal d'un système de tâches sur une machine parallèle est connu comme un problème difficile [Chr89], même si l'on suppose disposer d'un nombre illimité de processeurs (à cause des communications). Pourtant, dans le cas de boucles imbriquées, il est possible de définir des algorithmes d'ordonnancement efficaces et d'établir leur optimalité en utilisant des outils mathématiques variés tels la programmation linéaire ou les manipulations de matrices à coefficients entiers. Les caractéristiques principales qui différencient les techniques d'ordonnancement des nids de boucles affines par opposition aux systèmes généraux de tâches sont :

La cyclicité La structure régulière et répétitive d'un nid de boucles permet de ne considérer que le *graphe réduit de dépendances* pour la recherche d'un ordonnancement optimal.

La généricité Il est possible de trouver des ordonnancements *génériques* , valables pour toutes les valeurs des paramètres (pas besoin de re-compiler à chaque fois que la valeur d'un paramètre change).

L'évaluabilité La qualité d'un ordonnancement peut être évaluée et des théorèmes d'optimalité ont été démontrés.

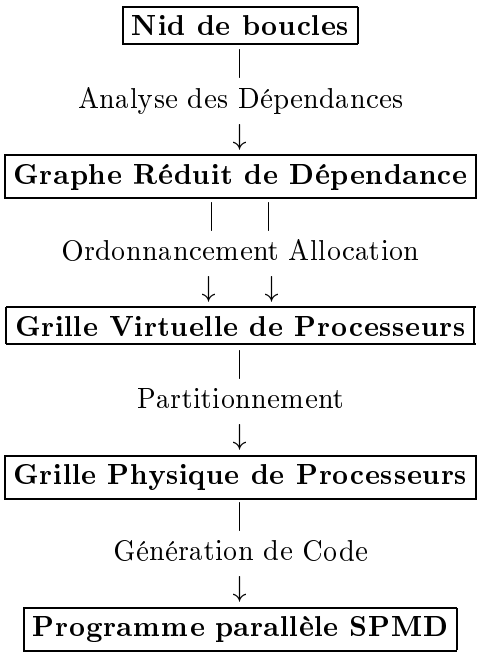
2.5.2 Méthode

Le principe de parallélisation automatique est résumé à la figure 2.3. L'analyse de dépendances en est le point de départ obligé. L'utilisation combinée des techniques d'ordonnancement et d'allocation donne une description spatio-temporelle de la répartition des calculs et des données qui se traduit par une réécriture du nid de boucles initial lors de la phase de génération de code.

Nous illustrons les différentes étapes dans les paragraphes suivants en utilisant l'exemple de Peir et Cytron. Rappelons que même si nous avons choisi un exemple simple parfait (toutes les instructions se situent au même niveau d'imbrication) et uniforme (les dépendances sont des translations), les techniques de transformation de boucles que nous décrivons ici s'appliquent à une classe plus générale de programmes appelée « nids de boucles affines à contrôle statique » [Fea92a, Fea92b]. Les restrictions sont les suivantes :

1. le programme peut dépendre de paramètres entiers définis une et une seule fois dans le programme par une instruction d'entrée-sortie ou par une relation avec d'autres paramètres déjà définis,

FIG. **2.3 -** Principe de parallélisation



2. les types de données sont restreints aux variables scalaires et aux tableaux multi-dimensionels de scalaires,

3. la seule structure de contrôle est la boucle `do`,

4. les seules instructions sont des affectations portant sur les scalaires ou les tableaux,

5. les bornes de boucles et les fonctions d'accès aux tableaux sont des *fonctions affines* des indices des boucles englobantes et des paramètres,

6. l'allocation de mémoire est supposée faite de façon unique (sans surnommage ou « aliasing »), c'est-à-dire que deux éléments de tableau correspondent à la même adresse mémoire si et seulement si ce sont les éléments d'un même tableau et que les indices sont identiques.

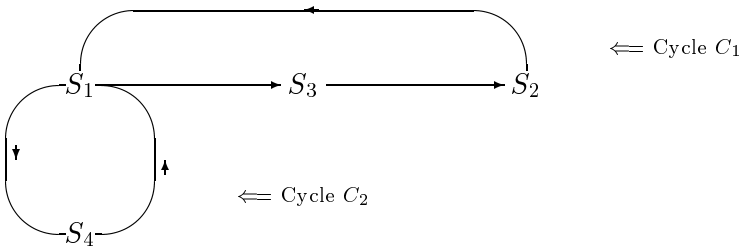
2.5.3 Analyse des dépendances

Le domaine de calcul de l'exemple de Peir et Cytron (voir programme 2.5 page 19) est un carré : $Dom = \{(i,j) \in Z^2, 0 \leq i,j \leq N\}$. Ce qui peut également s'écrire :

$$Dom(N) = \{x, Ax \leq Nb\}, \text{ avec } A = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \text{ et } b = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

D'autre part, la valeur $a(i,j)$, calculée à l'itération (i,j) de l'instruction S_1, est lue à l'itération $(i,j+2)$ de l'instruction S_3, d'où la dépendance de S_1 vers S_3 de vecteur $\binom{0}{2}$. En testant de façon

FIG. **2.4 -** Graphe réduit de dépendances



similaire toutes les autres dépendances, nous obtenons la matrice de dépendance suivante :

$$D = (d_{1,3}, d_{3,2}, d_{2,1}, d_{1,4}, d_{4,1}) = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 2 & -6 & 5 & 1 & -4 \end{pmatrix}$$

Remarquons que, dans notre exemple, le calcul est facile, car les vecteurs de dépendance ne dépendent pas des compteurs de boucle (on dit que le nid est uniforme). Le calcul des dépendances serait beaucoup plus difficile dans l'exemple de la factorisation de Choleski !

Les nids de boucles sont généralement représentés par un graphe orienté contenant k sommets (les instructions) reliés par des arêtes correspondant aux vecteurs de dépendances. La figure 2.4 représente le graphe de dépendance associé à notre exemple : c'est un graphe fortement connexe, il contient deux cycles $C_1 = (S_1, S_3, S_2)$ et $C_2 = (S_1, S_4)$.

2.5.4 Ordonnancement

Pour l'ordonnancement, c'est-à-dire pour l'extraction du parallélisme, on veut utiliser le fait que des points du domaine de calcul non reliés par des vecteurs de dépendance peuvent être exécutés simultanément. Schématiquement, l'idée la plus naturelle est de déterminer un vecteur de temps π et une famille d'hyperplans affines $H(t)$ tels que l'ensemble des points exécutés à un instant t soit égal à l'intersection du domaine de calcul avec l'hyperplan $H(t)$. On passe de $H(t)$ à $H(t+1)$ par une translation de vecteur π : les calculs progressent selon un front d'onde parallèle à la famille des hyperplans $H(t)$. Après réécriture, la nouvelle forme du nid sera, en utilisant les notations HPF, celle du programme 2.7.

PROG. **2.7** Forme générale d'un nid après réécriture

```
      do temps = temps_0, temps_N
!HPF$ INDEPENDENT
         do p ∈ E(temps)
            P(p)
         enddo
      enddo
```

$E(temps)$, l'ensemble des points calculés à l'itération $temps$, est défini par $E(temps) = \{p \in Dom; \lfloor \pi.p \rfloor = temps\}$. Ceci n'est rien d'autre que la méthode de l'hyperplan de Lamport [Lam74].

Pour garantir que la sémantique du nid de boucles initial est conservée, il faut que le sens des dépendances entre instructions soit préservé. Dans notre exemple, on cherche un vecteur π à composantes rationnelles tel que pour tout vecteur de dépendance d, on ait $\pi.d \geq 1$. En effet, si un

point q du domaine dépend d'un point p, on a $q = p + d$, où d est un vecteur de dépendance. La condition $\pi.d \geq 1$ assure que q sera calculé après p.

Mais parmi tous les vecteurs π qui remplissent la condition $\pi.d \geq 1$, lequel choisir ? Par exemple celui qui minimise le temps total d'exécution. Oublions les parties entières : le temps total d'exécution est donné par $\max\{\pi p - \pi q; p, q \in Dom\}$. Il faut alors résoudre un problème de programmation linéaire.

Pour le lecteur curieux, signalons que le meilleur ordonnancement linéaire pour notre exemple est donné par le vecteur de temps $\pi = (7 \quad 1)$, et qu'il conduit à un temps d'exécution égal à $8N$: voila qui n'était pas évident à trouver par simple inspection du nid !

Il existe des classes d'ordonnancement plus générales que les ordonnancements linéaires. Nous résumons les différentes solutions par la classification suivante :

(1) linéaire Un seul vecteur de temps, pas de constante. L'ordonnancement est donné par la fonction $p \mapsto \lfloor Xp \rfloor$.

(2) linéaire décalé Un seul vecteur de temps, une constante par équation. L'ordonnanement du sommet i est donné par la fonction $p \mapsto \lfloor Xp + c_i \rfloor$.

(2 bis) linéaire décalé paramétré Chaque constante est une fonction linéaire des paramètres. Dans le cas d'un seul paramètre N, l'ordonnancement du sommet i est donné par la fonction $p \mapsto \lfloor Xp + Nc'_i + c^*_i \rfloor$.

(3) affine par instruction Un vecteur et une constante par instruction. L'ordonnancement de l'instruction i est donné par la fonction $p \mapsto \lfloor X_i p + c_i \rfloor$.

(3 bis) affine paramétré par instruction Chaque constante est une fonction linéaire des paramètres. Dans le cas d'un seul paramètre N, l'ordonnancement du sommet i est donné par la fonction $p \mapsto \lfloor X_i p + Nc'_i + c^*_i \rfloor$.

Des résultats d'optimalité figurent dans [DRR93, DR94b]. Dans notre exemple, les solutions (2)bis et (3)bis sont équivalentes, car le graphe de dépendence est fortement connexe. La meilleure solution est donnée par $X = (\frac{11}{7} \quad -\frac{1}{7})$, $c'_1 = 0$, $c'_2 = -\frac{1}{7}$, $c'_3 = \frac{9}{7}$ et $c'_4 = \frac{8}{7}$, qui permettent d'aboutir à un temps total d'exécution de l'ordre de $\frac{12}{7}N$ (ce qui représente moins du quart de ce que l'on obtenait avec un ordonnancement linéaire).

Une description des méthodes utilisées pour calculer ces ordonnancements optimaux est présentée à la section 4.2.4

2.5.5 Allocation

Le problème général de trouver un placement optimal des instructions et des tableaux d'un nid de boucles affine sur une grille de processeurs virtuels est NP-complet [LC91, AL93]. De façon plus surprenante, ce résultat négatif reste valide pour les nids de boucles uniformes [DR93b]. Cependant, dans le cas simple des nids de boucles uniformes, il est possible de déterminer un placement optimal et d'appliquer des heuristiques efficaces [DR93b] (voir section 4.2.5 pour une présentation de ces heuristiques).

Pour résumer l'approche dans le cas d'un nid de boucles parfait et uniforme comme l'exemple de Peir et Cytron, partant d'un nid de profondeur n et contenant k instructions sur un domaine paramétré $Dom(N) = \{Ax \leq Nb\}$, la machine cible est une grille de processeurs virtuels de dimension $n - 1$ opérant en mode SPMD . Plus précisement, nous avons k grilles superposables

(une par instruction), et autant de cellules virtuelles que de projections distinctes des points de calculs $S_i(p), 1 \leq i \leq k, p \in Dom(N)$.

A chaque tableau et à chaque instruction est associée une fonction d'allocation ; il n'y a aucune raison a priori d'imposer la règle des écritures locales de HPF. La fonction d'allocation de l'élément (i, j) d'un tableau x de dimension q_x est :

$$\text{alloc}_x(i, j) = M_x \begin{pmatrix} i \\ j \end{pmatrix} + \alpha_x,$$

où M_x est une matrice de dimension $(n-1) \times q_x$ et α_x une constante.

De la même façon, la fonction d'allocation de l'instance (i, j) d'une instruction S est :

$$\text{alloc}_S(i, j) = M_S \begin{pmatrix} i \\ j \end{pmatrix} + \alpha_S,$$

où M_S est une matrice de dimension $(n-1) \times n$ et α_S une constante.

Pour effectuer le calcul de l'instance $S_1(i, j)$ de l'instruction S_1, le processeur $\text{alloc}_{S_1}(i, j)$ doit recevoir $b(i, j-6)$ du processeur $\text{alloc}_b(i, j-6)$ et $d(i-1, j+3)$ du processeur $\text{alloc}_d(i-1, j+3)$. Le résultat est ensuite envoyé par $\text{alloc}_{S_1}(i, j)$ au processeur $\text{alloc}_a(i, j)$.

Le **graphe de communication** de la figure 2.5 représente les communications nécessaires pour le calcul du nid de boucles. Il contient neuf arêtes, et donc autant de communications possibles. L'objectif est d'en minimiser le nombre. Même si on dispose de nombreux paramètres (8 constantes α et 8 matrices d'allocation M), on n'a pas autant de degrés de liberté à cause des cycles du graphe de communication. Pour résumer, dans le cas des nids de boucles affines, il est possible de choisir la même matrice d'allocation M pour tous les tableaux et toutes les instructions, et il reste ensuite à déterminer les meilleures constantes d'alignement (voir [DR93a, DR94c]). Pour les nids de boucles affines, c'est plus compliqué ! (Voir [AL93, HS91, LC91, Fea94] ...) Dans notre exemple, il peut ne rester finalement qu'une seule communication sur les neuf de départ en prenant par exemple la matrice de projection $M = (1 \quad 0)$ et les constantes d'alignement:

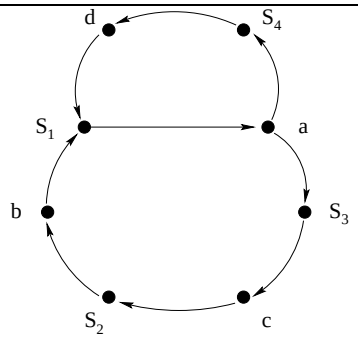
tableau	alignement	instruction	alignement
a	0	1	0
b	2	2	1
c	3	3	0
d	0	4	0

2.5.6 Partitionnement et génération de code

Partant d'un nid de boucles de profondeur n, l'architecture cible est une grille de dimension $n-1$. Dans notre exemple, $n = 2$ et l'architecture cible est un tableau linéaire ou un anneau de taille p, p étant le nombre de processeurs physiques. Les techniques d'ordonnancement et d'allocation nous donnent « un programme parallèle virtuel » qui s'exécute sur un nombre non borné de processeurs, appelés « processeurs virtuels ». Notre objectif est maintenant de faire correspondre à chaque processeur virtuel à un processeur physique, en partitionnant l'ensemble des processeurs virtuels et en allouant chaque élément de la partition à un processeur physique. Si on génère du code SPMD, chaque processeur va exécuter un programme dont le schéma est donné par le programme 2.8.

Si on génère du code HPF, c'est le moment d'utiliser le paramètre de distribution de données `CYCLIC(`*`taille-de-bloc`*`)`.

FIG. **2.5 -** Graphe de communication



PROG. **2.8** Schéma de programme SPMD

```
do t = temps_0, temps_N
   exécuter tous les calculs ordonnancés à l'instant t
   synchronisation
enddo
```

Nous voulons garantir que tous les processeurs soient toujours actifs (hormis pour des raisons d'effets de bord) de façon à obtenir un programme parallèle efficace. Ceci implique qu'à chaque étape de la boucle `do`, tous les processeurs ont le même nombre d'opérations à effectuer afin d'éviter d'avoir des processeurs en attente à la fin d'une phase de calcul.

Dans notre exemple, considérons la projection suivant la direction $s = \binom{1}{0}$ (cela revient à choisir $M = \begin{pmatrix} 0 & 1 \end{pmatrix}$). Avec $\pi = \begin{pmatrix} 7 & 1 \end{pmatrix}$, nous avons $c = \pi.s = 7$, ce qui signifie que chaque processeur virtuel est actif seulement tous les 7 tops. Il est donc utile de regrouper les processeurs en paquets de taille 7.

Pour résumer, voici le liste des différentes architectures que nous avons mentionnées, chaque architecture est obtenue par fusionnement des processeurs virtuels de l'architecture précédente :

1. k grilles de processeurs virtuels (obtenues par projection du nid de boucles),
2. k grilles de processeurs virtuels avec une efficacité de 100% (après regroupement des processeurs virtuels par paquets),
3. 1 grille de processeurs virtuels (plus petite grille commune pour tous les processeurs),
4. 1 grille de processeurs physiques (en partitionnant le grille précédente de façon à obtenir une grille qui « tient » dans l'architecture cible).

Le code final généré va être compliqué (voir programme 2.9). Il contient notamment la gestion des mémoires locales, les envois et réceptions de messages mais de telles primitives de bas-niveau sont gérées par les compilateurs HPF !

2.6 Conclusion

En résumé, les techniques de parallélisation automatique font appel à des outils mathématiques variés — programmation linéaire, manipulation de matrices à coefficients entiers, etc. Elles se

PROG. **2.9** Code HPF obtenu par parallélisation automatique de l'exemple de Peir et Cytron avec l'ordonnancement linéaire

```
      PROGRAM boucle

      INTEGER P
      INTEGER T
      PARAMETER (n = 100.0)
      REAL a(n,n)
      REAL b(n,n)
      REAL c(n,n)
      REAL d(n,n)

!HPF$ TEMPLATE BCLT_0_template(n+3,n+21)
!HPF$ DISTRIBUTE BCLT_0_template(CYCLIC,*)
!HPF$ ALIGN a(i1,i2) WITH BCLT_0_template(i1,i2+21)
!HPF$ ALIGN b(i1,i2) WITH BCLT_0_template(i1+2,i2+7)
!HPF$ ALIGN c(i1,i2) WITH BCLT_0_template(i1+3,i2)
!HPF$ ALIGN d(i1,i2) WITH BCLT_0_template(i1,i2+21)
      DO T = 21, 8*n-26
!HPF$ INDEPENDENT
        DO P = ceiling(max((-n+T+5.0)/7.0,2.0)),
     &         floor(min(T/7.0-1.0,n-3.0))
          a(P,T-7*P) = (b(P,T-7*P-6)+d(P-1,T-7*P+3))
          b(P+1,T-7*P-1) = c(P+2,T-7*P+5)
          c(P+3,T-7*P-1) = a(P,T-7*P-2)
          d(P,T-7*P-1) = a(P,T-7*P-1)
        END DO
      END DO
      END
```

limitent pour l'essentiel à des nids de boucles, donc à des fragments de programmes réguliers et structurés, mais en contrepartie elles tirent parti de la régularité de la structure pour offrir des solutions génériques avec un temps de compilation dépendant de la taille du programme et non du nombre de calculs.

Si le problème de l'ordonnancement est aujourd'hui bien compris, le problème de l'allocation est sans doute le plus mal compris surtout dans le cas des nids de boucles affines. Il ne suffit plus de distinguer les communications locales des communications non bornées : il faut prendre en compte la possibilité de primitives de communications globales propres à la machine.

La réalisation d'un outil de parallélisation automatique constitue le point central de cette thèse. L'autre partie, plus théorique étudie diverses techniques pour améliorer les performances des compilateurs-paralléliseurs.

Chapitre 3

Motivation : étude expérimentale

3.1 Introduction

Ce chapitre décrit une étude expérimentale ayant pour but d'évaluer dans quelle mesure les méthodes d'ordonnancement présentées brièvement au chapitre précédent sont adaptées à la parallélisation sur machines SIMD. L'algorithme étudié est à la fois simple et fondamental : le produit de matrices est en effet intensif en calcul et en communications. L'étude est menée sur une MasPar MP-1. La modélisation d'une telle machine s'est révélée très difficile à cause de ses possibilités de communication hiérarchiques. Les deux facteurs de performance principaux qui ont été identifiés sont le temps de communication et l'utilisation de la mémoire (registres contre accès indirect à la mémoire principale). La conclusion de cette étude est que les ordonnancements affines sont bien adaptés à une implémentation sur machine SIMD, mais qu'il y a aussi un réel besoin de modèles mathématiques adéquats pour décrire de telles machines afin d'en exploiter toute la puissance potentielle.

Les différentes méthodes d'ordonnancement sont présentées dans la section 3.2 : les ordonnancements linéaire et affine par instruction. Après une brève description de la MasPar dans la section 3.3, les expériences faites avec l'hypothèse d'un nombre infini de processeurs sont présentées dans la section 3.4. Le partitionnement en blocs du meilleur algorithme est ensuite détaillé dans la section 3.5. Les performances des programmes sont ensuite comparées à celles de la routine de la bibliothèque de calcul scientifique de la MasPar. Enfin, les conclusions de cette étude sont exposées dans la section 3.6.

3.2 Méthodes d'ordonnancement considérées

Le problème considéré ici est le produit de matrices uniformisé (voir programme 3.1).

Le domaine de calcul de ce problème peut être représenté par l'ensemble $Dom = \{p \mid Ap \leq b\}$ où $Ap \leq b$ décrit le cube $1 \leq i, j, k \leq n$, et ses dépendances par le graphe de dépendance de la figure 3.1.

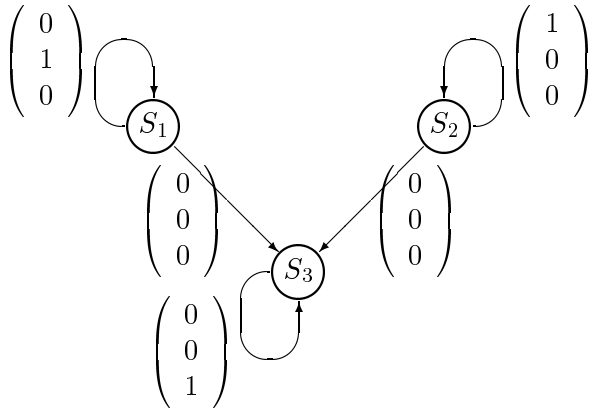
Comme indiqué au chapitre précédent, un ordonnancement pour un tel problème est une fonction des indices qui donne le temps auquel chaque instance des instructions va être exécutée. Nous considérons ici l'ordonnancement linéaire et l'ordonnancement affine par instructions présentés page 23.

Dans la cas qui nous intéresse, le produit de matrices, le meilleur vecteur de temps pour l'ordonnancement linéaire est $(1 \quad 1 \quad 1)$. Il donne un temps variant de 3 à $3n$.

PROG. **3.1** Le produit de matrices uniformisé

```
do i=1, n
   do j=1, n
      do k=1, n
         S1: a(i,j,k)=a(i,j-1,k)
         S2: b(i,j,k)=b(i-1,j,k)
         S3: c(i,j,k)=c(i,j,k-1)+a(i,j,k)*b(i,j,k)
      enddo
   enddo
enddo
```

FIG. **3.1 -** Graphe de dépendances réduit du produit de matrices.



Pour l'ordonnancement affine par instruction, on obtient l'ordonnancement suivant :

- L'instruction S_1 au point $p = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$ est calculée au temps $\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}.p$.
- L'instruction S_2 au point p est calculée au temps $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}.p$.
- L'instruction S_3 au point p est calculée au temps $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}.p + n$.

Cet ordonnancement donne une itération sur le temps allant de 1 à $2n$.

3.3 La MasPar MP-1

La machine utilisée pour cette étude de cas est une MasPar MP-1 [Bla90, Chr90, Nic90] à 16384 processeurs. C'est un ordinateur SIMD massivement parallèle. Le système est constitué d'un tableau de processeurs élémentaires (ici 128×128 PE) contrôlé par une unité de contrôle (ACU). Cette ACU est connectée à une station de travail UNIX jouant le rôle de frontal.

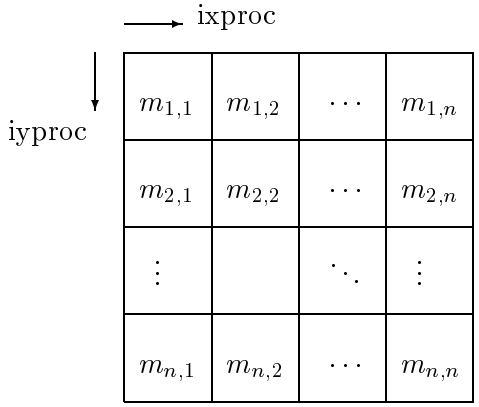
Les PEs dans le tableau de PE sont connectés en grille bidimensionnelle. De plus, chaque processeur peut communiquer avec ses huit voisins[1] par le réseau de communication Xnet. Xnet implémente aussi un rebouclage au bord de la grille de PE la transformant en tore. Il y a aussi un routeur global qui autorise des communications non régulières. Chaque groupe de 4×4 PE a 2 liens avec le routeur (une entrée et une sortie). Le routeur est un multiplexeur à trois étages extensible de la petite configuration de 32×32 PE à la plus grande de 128×128 PE.

Tous les programmes sont écrits en AMPL, le langage de programmation de MasPar basé sur le C ANSI. Les possibilités additionnelles principales sont :

- la déclaration `plural` : les variables definies avec le mot clé `plural` sont allouées de la même manière sur la grille de processuers, les autres sont allouées sur l'ACU. Les opérations sur des variables `plural`es sont exécutées en parallèle par tous les processeurs actifs.
- les expressions conditionnelles pour sélectionner les processeurs actifs : ce sont des extensions de l'utilisation des instructions `if`, `while`, `do` et `switch` qui opèrent sur des expressions `plural`es. Seuls les processeurs actifs exécutent les instructions parallèles. Ces conditions de sélection peuvent être imbriquées.
- le mot clé `xnet` permet l'utilisation du réseau de communication Xnet. `xnet` prend deux arguments : la direction de communication et la distance au processeur communicant. Il y a aussi les constructions `xnetp` et `xnetc` :
  - `xnetp` permet de « pipeliner » les données entre les deux processeurs communicants par les processeurs *inactifs* entre eux. La répartition des processeurs actifs est très importante quand on utilise cette instruction.
  - `xnetc` agit de la même façon que `xnetp` mais laisse une copie de chaque donnée pipelinée dans chaque processeur participant au pipeline.
- le mot clé `router` permet l'utilisation du routeur pour des communications arbitraires. Le paramètre de cette instruction est le numéro du processeur communicant.

1. Nord, nord-est, est, sud-est, sud, sud-ouest, ouest et nord-ouest.

FIG. **3.2 -** Stockage de la matrice $M = (m_{i,j})_{\{1 \leq i,j \leq n\}}$



un élément de tableau par processseur

– AMPL supporte aussi une large bibliothèque de fonctions opérant sur des données paralèles comme des vecteurs ou des matrices par exemple.

3.4 Étude de cas comparative

Cette section présente différentes implémentations du problème de multiplication de matrices présenté dans la section 3.2. On discute pour chacune l'utilisation des processeurs et l'efficacité du motif de communication. Dans tous les cas, on suppose un nombre illimité de processeurs et l'étude du partitionnement est reportée à la section suivante. Pour tous les programmes, la répartition des données est la répartition standard décrite par la figure 3.2. À la fin de cette section, les temps d'exécution sont comparés avec ceux de la routine de MasPar.

3.4.1 Ordonnancement linéaire

Plusieurs allocations des données ont été proposées pour l'ordonnancement linéaire. En effet, il n'y a qu'une contrainte sur le vecteur d'allocation A :

$$A \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \neq 0$$

Pour tirer avantage de la structure de la MasPar MP-1, les données sont projetées suivant $A = (\ 0 \quad 0 \quad 1\ )$. Cette allocation conduit au flux de données décrit par la figure 3.3. Pour avoir un programme où toutes les données sont réparties sur la grille de PE, les données sont initialement réparties comme sur la figure 3.4. La sélection des données utiles est faite en sélectionnant l'ensemble des processeurs actifs à chaque instant.

Ce programme souffre d'un inconvénient majeur : tous les processeurs ne sont pas actifs à un instant donné.

FIG. **3.3 -** Flux des données avec l'ordonnancement linéaire.

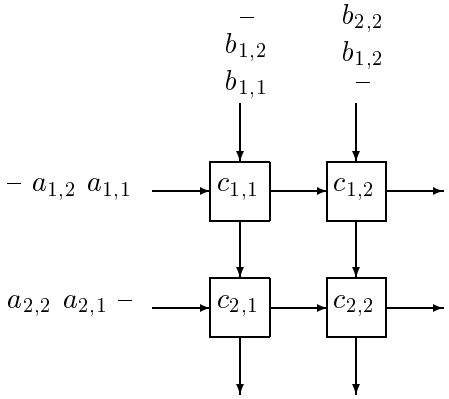


FIG. **3.4 -** Placement initial des données pour l'ordonnancement linéaire.

$a_{1,4}$	$a_{1,3}$	$a_{1,2}$	$a_{1,1}$
$a_{2,3}$	$a_{2,2}$	$a_{2,1}$	$a_{2,4}$
$a_{3,2}$	$a_{3,1}$	$a_{3,4}$	$a_{3,3}$
$a_{4,1}$	$a_{4,4}$	$a_{4,3}$	$a_{4,2}$

$b_{4,1}$	$b_{3,2}$	$b_{2,3}$	$b_{1,4}$
$b_{3,1}$	$b_{2,2}$	$b_{1,3}$	$b_{4,4}$
$b_{2,1}$	$b_{1,2}$	$b_{4,3}$	$b_{3,4}$
$b_{1,1}$	$b_{4,2}$	$b_{3,3}$	$b_{2,4}$

3.4.2 Ordonnancement par instruction

Trouver une allocation linéaire pour l'ordonnancement affine par instruction est plus difficile que pour l'ordonnancement linéaire car chaque instruction induit une contrainte sur le vecteur de projection. Pour l'ordonnancement affine par instruction on a choisi le vecteur de projection $A = (\ 1 \quad 1 \quad -1\ )$ car on doit avoir :

$$A \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \neq 0 \Rightarrow A_2 \neq 0,$$

$$A \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \neq 0 \Rightarrow A_1 \neq 0,$$

et

$$A \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \neq 0 \Rightarrow A_3 \neq 0.$$

(ce sont les conditions pour éviter de multiples calculs au même moment sur le même processeur). Le vecteur sélectionné a été celui qui, vérifiant ces conditions, a conduit à l'implémentation la plus simple.

On obtient alors le placement initial des données de la figure 3.5. Ensuite, pendant la première phase, les matrices A et B sont propagées comme indiqué sur la figure 3.6. La seconde phase est le calcul de la matrice C. Les processeurs actifs pendant cette phase sont indiqués sur la figure 3.7.

FIG. **3.5 -** Placement initial des données pour l'ordonnancement affine par instruction

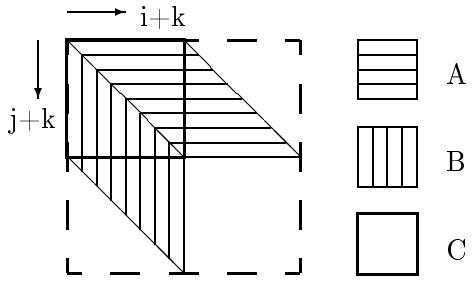


FIG. **3.6 -** Affine par instruction, première phase : propagation de A et B.

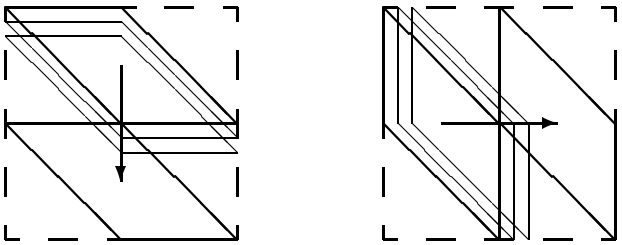


FIG. **3.7 -** Affine par instruction, deuxième phase: calcul de C

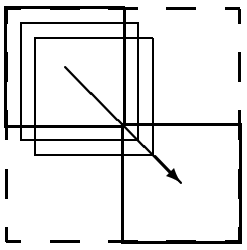
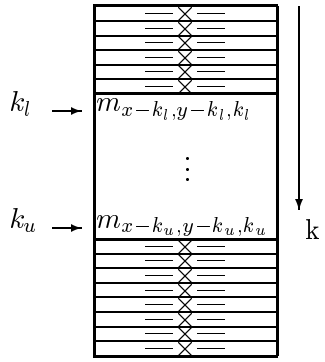


FIG. **3.8 -** Stockage de $m_{i,j,k}$ dans le processeur x, y pour l'ordonnancement par instruction.



où k_l et k_u sont définis par:

$$k_l = \min\{k \mid i + k = x \text{ et } j + k = y\}$$
$$k_u = \max\{k \mid i + k = x \text{ et } j + k = y\}$$
$$\text{avec } 0 \leq i, j, k \leq n, (n = 63)$$

Deux implémentations de cet algorithme ont été étudiées : pour la première, on a mis un élément de tableau par PE (voir figure 3.8 pour le stockage dans la mémoire des PE), utilisant ainsi seulement un quart des PE disponibles à chaque instant. Pour utiliser tous les PE à tout moment, on considère une grille virtuelle de processeurs quatre fois plus grande que la grille physique. C'est la deuxième implémentation. Comme il y a seulement un quart des processeurs virtuels actifs à chaque instant, tous les processeurs physiques sont actifs à chaque instant. Comme le réseau d'interconnexion se comporte comme un tore, les communications peuvent être faites avec `xnet`.

Les désavantages de cette implémentation sont :

- la distribution initiale torsadée (voir figure 3.5)
- la distribution après la propagation :
  - elle nécessite des tableaux pour stocker les données dans la mémoire des PE, il n'y a pas assez de registres pour retenir toute la propagation
  - soit il y a une mauvaise efficacité avec seulement un quart des processeurs actifs à chaque instant, soit l'utilisation des processeurs virtuels nécessite du contrôle qui prend du temps et de la mémoire parce qu'on doit stocker quatre tableaux de valeurs pour chaque processeur virtuel sur un processeur physique). En fait, il n'y a pas assez de mémoire pour stocker toutes les données nécessaires pour la propagation. Cette implémentation n'est pas extensible.

L'avantage est la séparation des phases de propagation et de calcul, mais comme on utilise des accès indirects à la mémoire, ce programme est plus lent que celui qui est dérivé de l'ordonnancement linéaire si on considère les données stockées comme le nécessite chaque algorithme.

3.4.3 Une approche non uniformisée

Comme dans l'algorithme précédent il y a toujours le même nombre de processeurs actifs, on a essayé d'utiliser tous les processeurs disponibles tout le temps. Pour ce faire, on a stocké chaque

tableau (A, B, C) de la façon standard.

En fait, avec ces projections, aucun des tableaux ne bouge. Cette solution peut être vue comme une implémentation du problème non uniformisé présenté par le programme 3.2.

PROG. **3.2** Problème non uniformisé

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

où $a(i,k,k) = a_{i,k}$ pour tout i et k, et $b(k,j,k) = b_{k,j}$ pour tout j et k.

Cela a plusieurs advantages :

- Comme il n'y a plus de propagation, il n'y a plus besoin de tableaux pour stocker les données. On peut simplement utiliser un registre par processeur pour stocker chacun des éléments locaux de A, B et C.
- Avec cette méthode d'allocation, tous les processeurs sont actifs en permanence.
- De plus, le programme est facile à lire et à écrire et ne nécessite pas de contrôle particulièrement compliqué.
- Le problème de cet algorithme est le motif de communication décrit par la figure 3.9 : à un instant donné, tous les processeurs d'une colonne doivent diffuser leur valeur de A à leur ligne et tous les processeurs d'une ligne doivent diffuser leur valeur de B à leur colonne. Par chance, la Maspar MP-1 a un outil puissant pour faire cela : `xnetc`, cette version pipelinée de `xnet` qui laisse une copie de la donnée qu'elle transmet et assez efficace pour cela.

Ce programme est bien plus efficace que ceux qui utilisent le problème uniformisé.

3.4.4 La routine de MasPar

La routine de MasPar est basée sur un algorithme de multiplication de matrices construit par Cannon [Can69]. Le placement initial des données est le placement standard avec un élément de la matrice par processeur comme pour l'algorithme précédent (voir figure 3.2). L'algorithme se décompose en deux phases : pour avoir tous les processeurs actifs et pour calculer le produit de deux éléments correspondants de A et de B, il est nécessaire de « prétordre[2] » A par lignes et B par colonnes comme décrit ci-dessous pour deux matrices 4×4 :

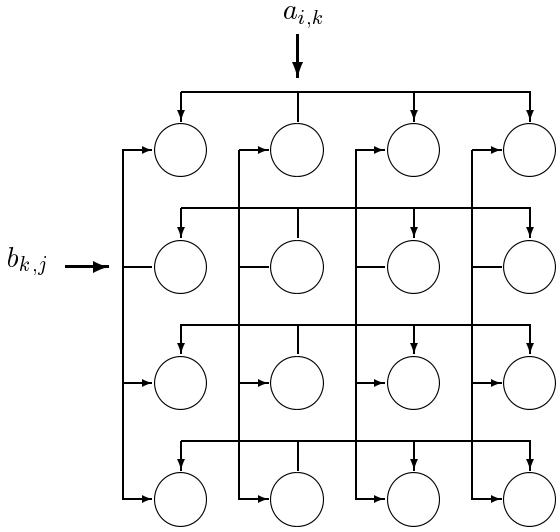
$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,1}$
$a_{3,3}$	$a_{3,4}$	$a_{3,1}$	$a_{3,2}$
$a_{4,4}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$

$b_{1,1}$	$b_{2,2}$	$b_{3,3}$	$b_{4,4}$
$b_{2,1}$	$b_{3,2}$	$b_{4,3}$	$b_{1,4}$
$b_{3,1}$	$b_{4,2}$	$b_{1,3}$	$b_{2,4}$
$b_{4,1}$	$b_{1,2}$	$b_{2,3}$	$b_{3,4}$

Après cette torsion, un flux de données simple est suffisant pour calculer le produit : après le calcul de chaque produit local et sa sommation avec la valeur précédente de C, A est décalé d'une

2. *skew* en anglais

FIG. **3.9 -** Les communications pour un tableau de PE de taille 4×4 avec $k = 2$ pour la version non uniformisée.



colonne vers la droite et B d'une ligne vers le bas. Après n étapes de cette séquence calcul-décalage, C contient le produit de A et B.

Ce programme a plusieurs avantages :

- il permet de n'utiliser que des registres pour stocker les matrices,
- pendant l'étape répétitive de calcul-décalages, toutes les communications sont des communications avec les voisins faites par `xnet`,
- et comme le routeur est bien adapté à la torsion, cette phase n'ajoute pas un grand délai au temps d'exécution.

Les temps donnés dans la section suivante pour cet algorithme sont mesurés avec un programme déduit de la routine de MasPar dont on a enlevé tout le contrôle dû au partitionnement. On a fait cela pour avoir une comparaison équitable entre les différents algorithmes. Le temps avec les données déjà placées est le temps de la seconde phase de l'algorithme, la pré-torsion étant le placement des données.

3.4.5 Comparaisons de temps

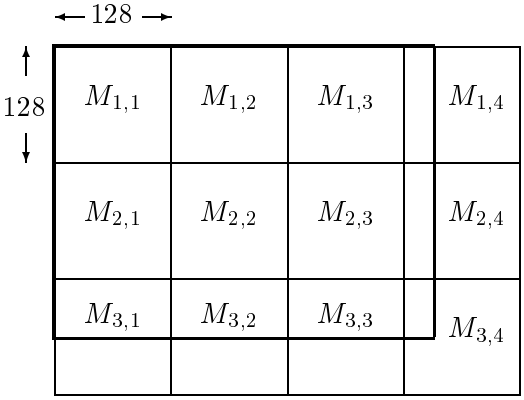
Nous donnons dans la table 3.1 les différents temps de calcul des programmes décrits dans cette section. Tous les temps sont donnés en secondes pour le produit le plus grand qui peut être fait sur une machine de 16384 processeurs, c'est-à-dire $n = 128$ pour tous les programmes, sauf pour la première version de l'affine par instruction pour lequel $n = 64$. Nous donnons deux temps pour chaque programme : le premier assume une distribution standard des données (voir figure 3.2) et le second ne mesure que le temps de calcul sans le placement initial de données.

Cette table confirme l'importance de l'utilisation de la mémoire et des mécanismes de communication. Par exemple, le programme de l'ordonnancement affine par instruction a une bonne utilisation des mécanismes de communication mais soit il utilise seulement un quart des processeurs disponibles (affine par instruction 1) ou il ne tient pas en mémoire (affine par instruction virtuel) ! La table confirme aussi qu'il est important que tous les processeurs soient actifs en permanence.

TAB. **3.1 -** Temps pour les programmes sans partitionnement.

ordonnancement	dist. std	calcul seul
linéaire	0,0358	0,0354
affine 1	0,0235	0,0224
affine virtuel	non disponible	non disponible
non uniforme	0,0159	0,0159
MasPar	0,0117	0,0110

FIG. **3.10 -** Partitionnement en blocs d'une matrice M.



3.5 Partitionnement

On considère dans cette section le partitionnement en blocs de notre algorithme le plus rapide (le non uniformisé) et de la routine de MasPar pour calculer des produits de matrices qui sont plus grandes que la taille de la grille de processeurs. On étudie d'abord deux partitionnements sur notre algorithme et ensuite celui de MasPar. Les temps d'exécutions de ces programmes sont donnés pour différentes tailles de matrices.

Pour tous ces partitionnements, on prend l'hypothèse que toutes les matrices sont partitionnées en blocs de la taille de la grille de PE (ici, 128×128), comme sur la figure 3.10, avec chacun de ces blocs répartis sur la grille de PE. Si la taille de la matrice n'est pas un nombre de blocs exact, on ajoute des zéros pour compléter les blocs extrémaux.

3.5.1 Partitionnement immédiat

Le premier partitionnement en blocs qui vient à l'esprit est de calculer le produit par blocs avec trois boucles imbriquées comme décrit par l'algorithme du programme 3.3. C'est très simple et n'ajoute pas beaucoup de contrôle. Cependant, on le verra dans la section suivante, ce n'est pas le plus efficace.

3.5.2 Un partitionnement plus élaboré

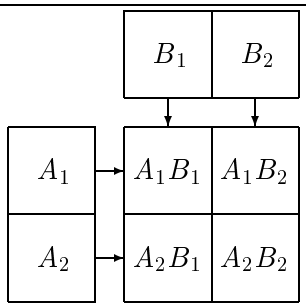
Un inconvénient du partitionnement précédent est que la propagation des données a lieu pour chaque produit de blocs. Comme chaque bloc de A doit être multiplié par une ligne de blocs de B, et vice-versa, beaucoup de propagations des mêmes données sont répétées.

Pour réduire la quantité de communications, on utilise un schéma de calcul où on peut la réduire par un facteur arbitraire jusqu'à la taille d'une ligne ou d'une colonne de la matrice. Cependant,

PROG. **3.3** Algorithme du partitionnement immédiat en blocs

```
do i=1, nblocks_i
   do j = 1, nblocks_j
      C_{i,j}=0
      do k=1, nblocks_k
         C_{i,j}=C_{i,j}+prod_block(A_{i,k}, B_{k,j})
      enddo
   enddo
enddo
```

FIG. **3.11 -** Une étape de calcul dans le partitionnement 2 blocs par 2 blocs



pour être utile, cette réduction ne doit pas augmenter le temps nécessaire aux accès aux données en mémoire : on est limité par le nombre de registres disponibles dans chaque PE. Comme chaque PE dispose de trente registres de 32 bits à la disposition de l'utilisateur, on n'a pu réduire les communications que d'un facteur deux.

Pour obtenir ce gain, on factorise la propagation d'un bloc de A pour calculer son produit avec deux blocs de B et réciproquement. Pour être efficace, on utilise deux blocs de A et deux blocs de B pour calculer quatre produits de blocs en une étape comme décrit par la figure 3.11. On utilise ainsi seulement quatre propagations (une pour chaque bloc) où on en utilisait huit avant (deux pour chaque produit). Pour gagner un facteur trois, on pourrait faire de même avec trois blocs de A et trois blocs de B, et ainsi de suite.

Si l'accès indirect en mémoire était plus rapide qu'une propagation, ce qui est le cas, on pourrait utiliser le même schéma de calcul avec la taille maximum et stocker toutes les variables nécessaires dans un tableau en mémoire. Cette option a été testée mais elle a un inconvénient majeur : elle devrait être plus rapide que les autres algorithmes pour de grandes matrices, mais la taille mémoire nécessaire pour stocker les propagations augmente avec la taille des matrices. On arrive à une taille si grande que pour les grandes tailles où cette option pourrait être plus rapide, elle nécessite plus de mémoire qu'il n'y en a disponible. Et dans le cas de petites matrices, le programme obtenu est beaucoup trop lent.

3.5.3 La routine de MasPar

La routine de MasPar pour le produit par blocs se fait aussi en deux étapes : la torsion et ensuite le calcul. On peut gagner du temps en factorisant la rotation au lieu de la faire à chaque fois qu'on utilise un bloc. Les programmeurs de MasPar on choisi de prétordre B une fois pour toutes au début de l'exécution et de le détordre à la fin. Pour A, ils utilisent une méthode différente, similaire à celle qui est décrite ci-dessus : ils factorisent la torsion d'un bloc de A pour trois produits de blocs

TAB. **3.2 -** Temps pour les programmes partitionnés.

taille:	1×1×1	3×2×2	4×5×5	10×10×10
simple	0,0161	0,192	1,59	15,9
2 par 2	0,0213	0,152	1,20	11,6
MasPar	0,0318	0,125	0,878	8,41

et multiplient trois blocs de A avec trois blocs de B comme décrit précédemment.

3.5.4 Comparaisons de performances

La table 3.2 montre les temps mesurés pour différentes tailles de matrices pour chaque programme décrit dans cette section. Tous les temps sont donnés en secondes et la taille est donnée comme les trois dimensions des matrices en nombre de blocs.

On peut voir que le partitionnement simple n'ajoute pas de délai important dû au contrôle, il est ainsi le plus performant pour de petites problèmes. Mais les deux autre stratégies ont de progressivement de meilleures performances par rapport au plus simple comme la taille des données augmente.

3.6 Conclusion

Le produit de matrices a été implémenté suivant différentes méthodes d'ordonnancement et ces implémentations ont été comparées avec la routine de MasPar. Notre but était de voir si les méthodologies développées originellement pour les réseaux systoliques étaient bien adaptées à la parallélisation automatique d'algorithmes sur des machines SIMD.

Les résultats montrent qu'il y a un potentiel d'utilisation pour de telles machines car les constructions purement systoliques utilisent très peu de mémoire et ainsi peuvent être implémentées avec uniquement des registres. De plus toutes les communications sont locales et peuvent ainsi être réalisées par des mécanismes rapides. Il y a besoin cependant d'améliorations pour prendre en compte toutes les possibilités et les limites d'une machine particulière comme les mécanismes de communication, le nombre de registres par processeur et le nombre de processeurs. Pour réussir dans cet effort, il y a besoin de modéliser mathématiquement ces contraintes. Si on veut trouver la meilleure implémentation d'un problème donné, il faut évaluer le temps non seulement en termes du nombre d'itérations d'une boucle, mais aussi comme le temps d'exécution réel d'une communication, d'un accès mémoire ou d'une opération arithmétique. C'est bien plus compliqué, mais aussi bien plus précis.

Remerciements

Nous remercions le « Parallel Processing Laboratory of Purdue University » et la subvention permettant l'utilisation de la MasPar qui y a été utilisée : NSF Parallel Infrastructure Grant # CDA-9015696.

Chapitre 4

Bouclettes

4.1 Introduction

Dans l'étude préliminaire du chapitre précédent, toutes les communications ont été écrites « à la main ». Ce qui est réalisable pour un exemple simple peut devenir bien plus difficile à gérer dans le cas général. De plus, les progammes écrits pour la MasPar ne sont pas portables car ils font appel à des primitives de communication propres à cette machine. Ce sont les raisons principales qui nous ont fait nous tourner vers le data-parallélisme et HPF en particulier quand nous avons voulu écrire un paralléliseur automatique[1]. Bouclettes est un tel outil. Il prend en entrée une certaine catégorie de boucles Fortran et retourne en sortie un programme HPF avec des constructions parallèles explicites.

4.1.1 Qu'est-ce que Bouclettes?

Bouclettes a été écrit pour valider des techniques d'ordonnancement et de placement de données basées sur des extensions de la méthode de l'hyperplan. Ces techniques sont décrites brièvement dans la section 4.2. Le but de la construction de Bouclettes était d'avoir un outil de parallélisation complètement automatique. Ce but a été atteint et l'utilisateur n'a plus qu'à choisir la stratégie de parallélisation qu'il veut appliquer.

Nous avons choisi HPF comme langage de sortie car nous croyons qu'il peut devenir un standard pour la programmation parallèle (et ainsi être largement utilisé). De plus, le data-parallélisme est un paradigme de programmation qui fourni un moyen simple d'exprimer des répartitions de données et de manipuler les communications induites par les calculs. Il libère ainsi le programmeur (ou l'outil de parallélisation) de la génération des communications de bas niveau du programme parallèle.

Ce chapitre est organisé comme suit: après l'introduction, nous présentons les différentes étapes de la parallélisation. L'outil et son implémentation sont décrits dans la section 4.3. Nous présentons ensuite la phase de génération du code dans la section 4.4. Et après un exemple complet (section 4.5), nous étudions comment des compilateurs HPF optimisent le code généré par Bouclettes dans la section 4.6. Nous concluons finalement dans la section 4.7.

4.1.2 Autres outils

La parallélisation automatique est étudiée par de nombreux groupes de recherche et plusieurs outils de parallélisation ont été écrits : SUIF [SCG], à l'université de Stanford en Californie, PIPS [Pip], à l'École Nationale Supérieure des Mines de Paris, la bibliothèque Omega [Pug+] à l'université du

1. Voir le chapitre 2 pour une introduction à HPF et à la parallélisation automatique.

Maryland, au Maryland, LooPo [Len+] à l'université de Passau, en Allemagne, le compilateur Paradigme [Ban+95], à l'université de l'Illinois, et PAF [Pri] à l'université de Versailles, entre autres.

Les particularités de Bouclettes par rapport à ces autres outils sont les méthodes de parallélisation employées (voir section 4.2) et le langage de sortie (HPF). On peut noter par exemple que Bouclettes est moins ambitieux que SUIF ou PAF, dans le sens qu'il traite un ensemble plus restreint de programmes, mais qu'il est plus précis dans son domaine, il exploite mieux le parallélisme potentiel des nids de boucles qu'il sait traiter.

4.2 Analyse des données et extraction du parallélisme

4.2.1 Le langage d'entrée

Bouclettes prend en entrée des boucles parfaitement imbriquées. Les bornes des indices des boucles sont des fonctions affines des indices des boucles englobantes et d'un paramètre, la taille du problème. Les seules instructions permises dans le corps du nid de boucles sont des affectations à des tableaux. Tous les tableaux apparaissant dans le corps du nid doivent être de dimension la profondeur d du nid.

De plus, toutes les fonctions d'accès aux tableaux doivent être des translations par rapport à l'index d'itération. Considérons le nid donné par le programme 4.1. C'est un nid que Bouclettes saura traiter. Si on avait remplacé la ligne `a(i,j,k)=a(i,j-1,k)` par `a(i,j,k)=a(j,i-1,k)`, les accès aux

PROG. **4.1** Exemple de programme d'entrée de Bouclettes

```
do i=2,n
  do j=2,n
    do k=2,n
      a(i,j,k)=a(i,j-1,k)
      b(i,j,k)=b(i-1,j,k)
      c(i,j,k)=c(i,j,k)+a(i,j-1,k)*b(i-1,j,k)
    end do
  end do
end do
```

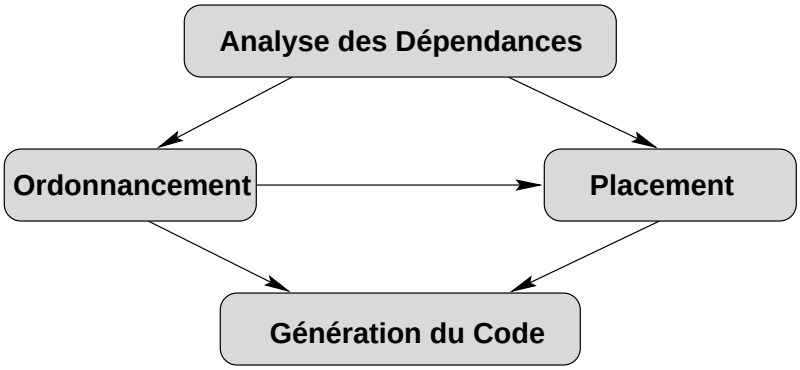
tableaux ne seraient plus tous des translations et ce nid ne serait pas transformable par Bouclettes.

4.2.2 Phases de la parallélisation automatique

Le processus de parallélisation peut être décomposé en plusieurs tâches interdépendantes. Voir la figure 4.1.

L'analyse des dépendances consiste en la création d'un graphe représentant les contraintes sur l'ordre d'exécution des instances des instructions. La phase d'ordonnancement utilise les dépendances pour construire une fonction qui associe un temps d'exécution à chaque instance de chaque instruction. La phase de placement des données répartit les tableaux et les calculs sur un ensemble de processeurs virtuels. Les deux phases précédentes sont interdépendantes : nous voulons que la transformation globale respecte les dépendances de données et que deux calculs qui sont ordonnancés au même moment soient effectués par deux processeurs virtuels différents. La dernière étape est la génération du code. Le code généré contient des boucles explicitement parallèles (directive `INDEPENDENT`) et une répartition des données qui est, elle aussi, explicite (directives `DISTRIBUTE` et `ALIGN`).

FIG. **4.1 -** Les étapes de la parallélisation dans Bouclettes



Le système Bouclettes est organisé comme une succession d'étapes :

1. le programme d'entrée est analysé et traduit dans une représentation interne,
2. cette représentation est utilisée pour calculer les dépendances de données ; dans notre cas les dépendances sont uniformes, donc un analyseur très simple est suffisamment puissant pour déterminer les dépendances exactes,
3. à partir de ces dépendances, un ordonnancement linéaire ou linéaire décalé est construit,
4. l'ordonnancement et la représentation interne sont utilisées pour calculer un placement des données compatible avec l'ordonnancement,
5. enfin, le code HPF est généré en utilisant la transformation qu'on vient de calculer.

4.2.3 Analyse de dépendances

L'analyse de dépendances est assez simple dans le contexte restreint dans lequel nous nous plaçons. Il faut en effet trouver toutes les dépendances entre les instructions du corps du nid : les trois sortes de dépendances (directes, de sortie et les antidépendances) peuvent être calculées de la même façon. Les vecteurs de dépendance sont des différences entre deux fonctions d'accès au même tableau ; et réciproquement, toutes les différences entre deux fonctions d'accès au même tableau sont des vecteurs de dépendance.

4.2.4 Ordonnancement

Darte et Robert ont présenté des techniques de calcul d'ordonnancements pour des nids de boucles uniformes [DKR91, DR94b]. Ces techniques sont une partie de la base théorique de Bouclettes.

Pour l'instant, l'utilisateur a le choix entre un ordonnancement linéaire et un ordonnancement linéaire décalé.

L'ordonnancement linéaire est une fonction qui associe un temps t à un point d'itération $\vec{i}$ ($\vec{i} = (i, j, k)$ si le nid de boucles est de dimension 3) comme suit :

$$t(\vec{i}) = \left\lfloor \frac{p}{q} \pi . \vec{i} \right\rfloor$$

où p, q sont des entiers et π est un vecteur d'entiers premiers entre eux de dimension la profondeur d du nid de boucles.

L'ordonnancement linéaire décalé est une extension de l'ordonnancement linéaire où chaque instruction du corps du nid a sa propre fonction d'ordonnancement. Toutes ces fonctions partagent la même partie linéaire et des constantes de décalage (qui peuvent différer entre les instructions) sont ajoutées pour chaque instruction. Le temps t_k pour l'instruction k est calculé comme suit :

$$t_k(\vec{i}) = \left\lfloor \frac{p}{q}\pi.\vec{i} + \frac{c_k}{q} \right\rfloor$$

où p, q, c_k sont des entiers et π est un vecteur d'entiers premiers entre eux de dimension d.

Le calcul de ces ordonnancements est effectué par des techniques (programmation linéaire ou calcul sur des graphes) qui garantissent que le résultat est optimal dans la classe d'ordonnancements considérée. Ici on entend par optimal que la latence totale est minimisée.

Recherche de l'ordonnancement linéaire optimal

On considère un domaine d'itération de la forme $Dom = \{p | Ap \leq b\}$. Le temps d'exécution total donné par un ordonnancement linéaire est la différence maximale entre les temps d'exécution de deux points du domaine :

$$\max_{(Ap \leq b, Aq \leq b)} (\lfloor \pi p \rfloor - \lfloor \pi q \rfloor)$$

avec p et q entiers, qu'on peut approcher par :

$$T(\pi) = \max_{(Ap \leq b, Aq \leq b)} \pi(p - q)$$

en supprimant les parties entières et en autorisant p et q à n'être plus nécessairement entiers.

La recherche du meilleur ordonnancement linéaire se traduit alors par le problème d'optimisation :

$$\min_{(\pi D \geq 1)} T(\pi) = \min_{(\pi D \geq 1)} \max_{(Ap \leq b, Aq \leq b)} \pi(p - q)$$

En fait, puisque trouver le vecteur de temps optimal revient à résoudre le problème :

$$\min_{(\pi D \geq 1)} \max_{(Ap \leq b, Aq \leq b)} \pi(p - q)$$

et que, par dualité,

$$\begin{cases} Ap \leq b \\ Aq \leq b \\ \max \pi(p - q) \end{cases}$$

est égal à :

$$\begin{cases} \pi_1 A = \pi \\ \pi_2 A = -\pi \\ \pi_1 \geq 0 \\ \pi_2 \geq 0 \\ \min (\pi_1 + \pi_2) b \end{cases}$$

la recherche du vecteur de temps optimal peut se faire en résolvant le problème linéaire suivant :

$$\begin{cases} \pi D \geq 1 \\ \pi_1 A = \pi \\ \pi_2 A = -\pi \\ \pi_1 \geq 0 \\ \pi_2 \geq 0 \\ \min (\pi_1 + \pi_2) b \end{cases}$$

Remarquons que ce problème est linéaire en b. La recherche du vecteur de temps optimal sur une famille de domaines de la forme $\{Ax \leq Nb\}$ se réduit à celle sur le domaine $\{Ax \leq b\}$ qui peut donc se faire sans connaissance du paramètre N, donc à la compilation. Pour des domaines à plusieurs paramètres, on peut utiliser un algorithme du simplexe paramétré comme PIP [FT90].

Recherche de l'ordonnancement linéaire décalé

Pour de tels ordonnancements, l'ensemble des contraintes de dépendances se réduit à :

$$\begin{cases} \forall i, 1 \leq i \leq k \\ \pi D_i \geq 1 \\ \forall i, 1 \leq i \leq k \\ \forall j \in Pred(i) \\ \pi d_{j,i} + c_i - c_j \geq 1 \end{cases}$$

D'autre part, comme toutes les instructions sont séquencées par le même vecteur de temps, elles ont le même délai :

$$\text{délai}(\pi) = \max_{(Ap \leq b, Aq \leq b)} \pi(p - q)$$

et le temps total d'exécution est donné par $\text{délai}(\pi) + \max_{i,j}(c_i - c_j)$. Négligeant $\max_{i,j}(c_i - c_j)$ (qui ne dépend pas de la taille du domaine) devant le délai qui est linéaire en N sur un domaine $\{Ax \leq Nb\}$, on est ramené à résoudre le problème linéaire suivant :

$$\begin{cases} \forall i, \pi D_i \geq 1 \\ \pi_1 A = \pi \\ \pi_2 A = -\pi \\ \pi_1 \geq 0 \\ \pi_2 \geq 0 \\ \\ \forall (i,j) \text{ tel que } d_{i,j} \text{ existe} \\ \pi d_{i,j} + c_j - c_i \geq 1 \\ \min (\pi_1 + \pi_2) b \end{cases}$$

L'algorithme utilisé dans Bouclettes est la résolution d'un problème d'optimisation raffinant celui qui est présenté ci-dessus pour réduire le nombre de contraintes.

4.2.5 Placement

Darte et Robert ont présenté une technique de détermination d'un placement des données et des calculs sur une grille de processeurs virtuels (voir [DR94a]). C'est cette technique simplifiée qui est utilisée dans Bouclettes.

En se basant sur le calcul du « graphe de communication », une structure qui représente toutes les communications qui peuvent avoir lieu dans un nid de boucles donné, on détermine une projection (représentée par une matrice M) et des constantes d'alignement. L'idée principale est de projeter les tableaux (et donc les calculs) sur une grille de processeurs virtuels de dimension $d-1$. Ensuite les tableaux et les calculs sont alignés (par les constantes d'alignement) pour supprimer des communications.

Plus précisément, M, la matrice de projection, est une matrice de taille $(d-1) \times d$ d'entiers, de rang plein, et les constantes α_x sont des vecteurs d'entiers de dimension $d-1$. À chaque tableau ou instruction est alors associée une fonction d'allocation définie par :

$$\mathrm{alloc}_x(\vec{i}) = M\vec{i} + \alpha_x$$

Comme les nids de boucles considérés sont uniformes, le fait de choisir une matrice différente pour chaque tableau ou instruction n'améliorerait pas le placement. De plus, l'ordonnancement doit être pris en compte pour choisir la matrice M. En effet, le domaine d'itération du nid de boucles transformé sera l'image du domaine d'itération initial par la transformation :

$$\vec{i} \mapsto \begin{pmatrix} \pi \\ M \end{pmatrix} \vec{i}$$

Il est indispensable d'avoir ce domaine d'itération inclus dans $\mathbf{Z}^d$, car, dans le cas contraire, on aurait des processeurs avec des index rationnels. Comme le choix de M n'a pas un grand impact sur le nombre de communications qui restent, $\left(\begin{smallmatrix} \pi \\ M \end{smallmatrix}\right)$ est juste calculée comme la complétion unimodulaire du vecteur π.

Une fois que M a été calculée, les constantes d'alignement sont déterminées afin de minimiser le nombre de communications. L'utilisateur peut choisir ici s'il veut respecter la règle des écritures locales (comme dans HPF) ou pas. S'il choisit de ne pas respecter cette règle, des tableaux temporaires seront générés au besoin pendant la phase suivante pour en tenir compte.

Le calcul des constantes se fait par la construction d'un sous-graphe sans cycle contenant tous les sommets du graphe de communication. Rappelons que le graphe de communication est un graphe biparti dont les sommets sont d'une part les tableaux de données et d'autre part les instructions et qu'il y a une arête entre deux sommets si il y a lecture ou écriture du tableau par l'instruction. Les arêtes sont orientées dans le sens de la communication potentielle et étiquetées par les différences d'index entre l'accès au tableau et l'index de l'occurrence de l'instruction. Les constantes sont alors calculées par la propagation des valeurs d'alignement à partir d'un nœud particularisé du sous-graphe sans cycle en annulant les communications de chaque arête de ce sous-graphe. C'est pendant la phase de construction du sous-graphe recouvrant qu'intervient le respect forcé ou non de la règle des écritures locales. En effet, si cette règle est forcée, alors toutes les arêtes correspondant à des écritures sont obligatoirement dans le sous-graphe construit.

Remarquons que c'est le fait de vouloir un graphe de communication uniforme (les étiquettes des arêtes sont des constantes) qui nous a conduit à prendre l'hypothèse que les fonctions d'accès aux tableaux sont des translations.

4.2.6 La génération du code

De nombreux problèmes apparaissent ici. Dans tous les cas, la génération du code implique la réécriture du nid en fonction d'une transformation unimodulaire. Cette technique de réécriture est décrite dans [CFR93] et fait appel au logiciel PIP [FT90] (Parallel Integer Programming). Une description complète du processus de génération du code peut être trouvée dans la section 4.4.

La génération du code produit essentiellement une boucle séquentielle, représentant l'itération sur le temps donné par l'ordonnancement, contenant $d-1$ boucles parallèles (`INDEPENDENT`) décrivant l'ensemble des processeurs actifs à chaque instant. Les tableaux sont répartis pour respecter le placement calculé précédemment.

Certains cas compliquent la génération du code :

la règle des écritures locales : quand le placement ne respecte pas cette règle, des tableaux temporaires sont ajoutés pour la simuler.

la direction de projection : le pouvoir d'expression de la directive HPF `DISTRIBUTE` est limité aux projections suivant un ou plusieurs axes du domaine. Quand le placement projette les données dans une autre direction, on redistribue les données. Cette redistribution consiste en une copie des tableaux dans de nouveaux tableaux temporaires, qui sont, eux, projetés suivant un axe du domaine d'itération, suivie du calcul du nid avec ces tableaux temporaires et de la recopie des résultats dans les tableaux d'origine.

les rationnels et les constantes de temps : ces paramètres compliquent beaucoup le code généré. Nous aurions en fait besoin de pouvoir exprimer du parallélisme de contrôle pour utiliser tout le parallélisme donné par de tels ordonnancements.

la stratégie de répartition : la meilleure répartition des données serait de les répartir de manière *bloc-cyclique* avec la taille des blocs dépendant de la machine cible. Cela partitionnerait les données de façon à équilibrer la charge de calcul entre les processeurs et dans le même temps autoriser le compilateur à regrouper des communications pour qu'elles soient plus rapides. Bouclettes peut générer n'importe quelle répartition. Comme les compilateurs actuels ne supportent que des distributions par blocs, pour pouvoir tester la sortie de Bouclettes, nous générerons des distributions par blocs.

4.3 Implémentation

4.3.1 Programmation

Bouclettes a été écrit dans le langage Caml Light[2] [LW94, Cri]. Nous avons choisi Caml Light parce qu'il permet de manipuler très facilement des structures de données complexes comme des arbres abstraits ou des expressions symboliques, et de faire des calculs symboliques facilement. Comme les analyseurs lexicaux et syntaxiques sont aussi faciles à écrire en Caml Light, ce langage est très bien adapté à l'écriture de compilateurs. Pour écrire Bouclettes, nous avons développé des modules utilitaires pour les calculs sur les rationnels, les manipulations d'expressions symboliques générales ou affines, ou le calcul matriciel, et une interface au logiciel PIP pour la programmation linéaire paramétrée. Nous avons interfacé Caml Light, grâce à des fichiers, avec les programmes écrits en C++ qui calculent les ordonnancements. Enfin une interface utilisateur graphique a été écrite en utilisant la bibliothèque CamlTk permettant la présentation d'un programme bien intégré. Le programme complet fait plus de 12500 lignes de code.

4.3.2 Interface utilisateur

Bouclettes a deux formes : une commande acceptant des options sur la ligne de commande qui fait toute la transformation en une seule fois sans interface graphique et un environnement

2. Caml Light est une implémentation de ML faite à l'INRIA

graphique qui permet à l'utilisateur de choisir interactivement les options qui l'intéressent et de voir la transformation étape par étape.

Les différentes options sont les suivantes :

- le choix du type d'ordonnancement : linéaire ou linéaire décalé,
- le choix de forcer ou non le respect de la règle des écritures locales,
- le choix de redistribuer les données même si ce n'est pas nécessaire (voir page 52).

Il y a encore une option dans Bouclettes : l'utilisateur peut choisir le langage de sortie pour tester le programme produit. Toutes les sorties autres que la sortie HPF standard comprennent le remplissage des tableaux et le calcul de la somme des éléments de chaque tableau après le calcul du nid transformé. Cela permet des comparaisons entre les versions et la vérification de la justesse du programme transformé. Les différents formats de sortie non standards sont :

- le programme d'entrée en Fortran 77,
- le programme de sortie en Fortran 77, toutes les directives HPF sont enlevées et les boucles parallèles (`INDEPENDENT` et `FORALL`) sont traduites en boucles `DO` séquentielles, permettant ainsi la vérification que le programme de sortie fait bien le même calcul que le programme d'entrée,
- le programme de sortie dans un sous-ensemble d'HPF qui est compris par les compilateurs HPF actuels. En effet, les compilateurs actuels n'implémentent qu'une partie du standard HPF 1 [HPF93]. En particulier, les boucles `INDEPENDENT` et les constructions `FORALL` sont traduites en instructions `FORALL`[3]. C'est aussi la raison pour laquelle on a choisi de générer des distributions par blocs.

Plus d'informations techniques ainsi que les sources du programmes sont disponibles à l'URL :

`http://www.ens-lyon.fr/~pboulet/bclt/bouclettes.html`

4.4 Génération du code dans Bouclettes

Cette section présente dans le détail la phase finale de la parallélisation dans Bouclettes : la génération du code HPF. Nous la présentons en deux étapes : d'abord, dans la section 4.4.1, les problèmes posés par l'ordonnancement linéaire et ensuite, dans la section 4.4.2, la réécriture de l'ordonnancement linéaire décalé. Enfin, nous ferons quelques commentaires sur l'impact qu'a eu le choix d'HPF comme langage de sortie dans la section 4.4.3.

4.4.1 Codage de l'ordonnancement linéaire

Réécriture de boucles

Bouclettes utilise des techniques présentées par Collard, Feautrier et Risset dans [CFR93] pour réécrire les boucles après réindexation. La réindexation produit un nouveau domaine d'itération qui est un polyèdre convexe défini par un ensemble de contraintes affines. La réécriture du nid de boucles nécessite l'énumération de tous les points entiers de ce convexe et l'algorithme utilisé

3. les premiers compilateurs HPF à accepter les constructions `FORALL` apparaissent

s'appuie sur une version paramétrée de la méthode du simplexe dual, implémentée dans le logiciel PIP (voir [Fea88]).

Considérons le nid de boucles parfait initial du programme 4.2, où $\vec{z}$ est un vecteur de paramètres de structure (voir [CFR93]).

PROG. **4.2** Nid de boucles parfait initial

```
do i1=i1^l(z), i1^u(z)
   do i2=i2^l(i1,z), i2^u(i1,z)
         ...
         do id=id^l(i1,...,id-1,z), id^u(i1,...,id-1,z)
            S1(i1,i2,...,id,z)
            ...
            Sk(i1,i2,...,id,z)
         enddo
   ...
   enddo
enddo
```

Nous réécrivons le nid en lui appliquant une transformation unimodulaire U. Les vecteurs de coordonnées $\vec{i} = (i_1, ..., i_d)^t$ et $\vec{j} = (j_1, ..., j_d)^t$, respectivement dans la vieille base et dans la nouvelle, sont liés par la relation :

$$\vec{j} = U\vec{i}$$

où U est une matrice unimodulaire ($\det(U) = \pm 1$) de taille $d \times d$. Comme nous nous intéressons aux nids parfaits, les domaines d'itération sont des polyèdres convexes finis de $\mathbf{Z}^d$ qui peuvent être définies par un ensemble d'inégalités comme :

$$D(\vec{z}) = \{\vec{i} | \vec{i} \in \mathbf{Z}^d, C\vec{i} + C'\vec{z} + \vec{b} \geq 0\},$$

où C, C' sont des matrices de contraintes et $\vec{b}$ est un vecteur constant. Dans le nouveau domaine d'itération, le polyèdre peut être défini par :

$$D(\vec{z}) = \{\vec{j} | \vec{j} \in \mathbf{Z}^d, CU^{-1}\vec{j} + C'\vec{z} + \vec{b} \geq 0\}.$$

Collard, Feautrier et Risset ont prouvé que le nid de boucles initial (programme 4.2) peut être réécrit avec le nouveau domaine d'itération dans la forme donnée par le programme 4.3 où les bornes de boucles j_n^l et j_n^u, $1 \leq n \leq d$ sont des expressions des paramètres de structure et des indices des boucles englobantes $j_1, ..., j_{n-1}$ suffisamment simples (pour un compilateur HPF). Pour un nid de boucles parfait de profondeur d, les nouvelles bornes de boucles sont obtenues après $2d$ appels successifs à PIP.

Non respect de la règle des écritures locales

Les compilateurs HPF respectent la règle des écritures locales : chaque processeur ne calcule que ses propres données. Dans la phase de placement de Bouclettes, l'utilisateur peut choisir de ne pas forcer le respect de cette règle. Considérons un élément de tableau $a(\vec{i} + c_a)$ calculé par une instruction $S(\vec{i})$. Le placement peut donner des fonctions d'allocation telles que : $(M(\vec{i}+c_a)+\alpha_a) \neq M\vec{i}+\alpha_S$. Pour rendre les deux fonctions d'allocation compatibles avec la règle des écritures locales, nous devons ajouter des tableaux temporaires pendant la phase de génération de code.

PROG. **4.3** Nid de boucles parfait réécrit

```
do j1=j1^l(z), j1^u(z)
   do j2=j2^l(j1,z), j2^u(j1,z)
         ...
         do jd=jd^l(j1,...,jd-1,z), jd^u(j1,...,jd-1,z)
            S1(U^-1 j, z)
            ...
            Sk(U^-1 j, z)
         enddo
   ...
   enddo
enddo
```

PROG. **4.4** Nid qui ne vérifie pas la règle des écritures locales

```
real a(n,n)

do i
      S(i)    a(i + c_a) = expr
enddo
```

PROG. **4.5** Nid qui vérifie la règle des écritures locales après ajout de tableaux temporaires

```
real a(n,n)
real a_tmp(n,n)

do i
      S1(i)   a_tmp(i + c_a) = expr
      S2(i)   a(i + c_a) = a_tmp(i + c_a)
enddo
```

Ainsi le nid du programme 4.4, où *expr* est une expression fonction d'autres éléments de tableaux du programme, est transformé en y ajoutant des tableaux temporaires (voir le programme 4.5).

Les nouvelles fonctions d'allocation sont déduites des fonctions initiales pour respecter la règle des écritures locales :

$$\begin{array}{lcl} \text{alloc}_a(\vec{i}) & = & M\vec{i} + \alpha_a \\ \text{alloc}_{S_1}(\vec{i}) & = & M\vec{i} + \alpha_s \\ \text{alloc}_{a_{tmp}}(\vec{i}) & = & M\vec{i} + \alpha_S - Mc_a \\ \text{alloc}_{S_2}(\vec{i}) & = & M\vec{i} + Mc_a + \alpha_a. \end{array}$$

Alignement des tableaux

En HPF, le programmeur peut spécifier le placement des données à deux niveaux. D'abord les tableaux sont alignés entre eux grâce à la directive `ALIGN`. Ensuite, ils sont répartis sur les processeurs (virtuels) avec la directive `DISTRIBUTE` [KLS+94].

Selon la matrice de projection M, nous adoptons deux stratégies pour aligner les données :

- si la projection se fait le long d'un des axes du domaine d'itération, nous pouvons aligner directement les tableaux comme c'est expliqué dans la suite,
- sinon, nous devons « redistribuer » les tableaux avant de les aligner (voir page 52).

Quand la projection se fait le long d'un des axes du domaine d'itération D

Soient a_k, $(1 \leq k \leq n)$ les tableaux d'un nid de boucles de profondeur d. Soit $\text{alloc}_{a_k}(\vec{i}) = M\vec{i} + \alpha_{a_k}$ la fonction d'allocation pour le tableau a_k. Soit π le vecteur d'ordonnancement linéaire pour le nid. Soit $\vec{i} \in D$ et $\vec{j} = \vec{i} + \left(\begin{smallmatrix}\pi\\M\end{smallmatrix}\right)^{-1}\left(\begin{smallmatrix}0\\\alpha_k\end{smallmatrix}\right)$. Nous avons

$$M\vec{j} = M\vec{i} + M\begin{pmatrix}\pi\\M\end{pmatrix}^{-1}\begin{pmatrix}0\\\alpha_k\end{pmatrix}.$$

Soit $\left(\begin{smallmatrix}\pi\\M\end{smallmatrix}\right)^{-1} = (X_1\ X_2)$, nous avons

$$\begin{pmatrix}\pi\\M\end{pmatrix}(X_1\ X_2) = \begin{pmatrix}\pi X_1 & \pi X_2\\ MX_1 & MX_2\end{pmatrix} = Id.$$

Ainsi, $MX_2 = Id$ et

$$M\vec{j} = M\vec{i} + M(X_1X_2)\begin{pmatrix}0\\\alpha_k\end{pmatrix}$$

$$M\vec{j} = M\vec{i} + \alpha_k$$

Soit $p = M\vec{i} + \alpha_k$ (le processeur p reçoit la valeur $a_k(\vec{i})$). p et l'image de $\vec{j}$ par M correspondent au même point dans l'espace des processeurs virtuels. Pour aligner tous les tableaux a_k entre eux, une possibilité est de déclarer un « template » `BCLT_template` de dimension d et d'aligner tous les tableaux avec la directive suivante :

`!HPF$ ALIGN` $a_k(\vec{i})$ `WITH BCLT_template`$\left(\vec{i} + \left(\begin{smallmatrix}\pi\\M\end{smallmatrix}\right)^{-1}\left(\begin{smallmatrix}0\\\alpha_k\end{smallmatrix}\right)\right)$

La répartition des données alignées sur les processeurs est alors spécifiée par la directive :

```
!HPF$ DISTRIBUTE BCLT_template (BLOCK,...,BLOCK,*,BLOCK,...,BLOCK)
```

L'« `*` » correspond à la direction de la projection. Notons que ceci est possible seulement parce que la projection est parallèle à une direction du domaine d'itération.

Rotation des tableaux

Quand la projection n'est pas parallèle à un axe, on doit « redistribuer » les tableaux pour écrire les directives HPF.

Soit $U = \binom{\pi}{M}$. Pour chaque tableau a_k du nid de boucles, nous définissons les nouveaux tableaux $a_{k,\mathrm{rot}}$ tels que

$$a_{k,\mathrm{rot}}(\vec{i}) = a_k(U^{-1}\vec{i}).$$

Nous calculons la nouvelle fonction d'allocation du tableau $a_{k,\mathrm{rot}}$ à partir de la fonction d'allocation du tableau a_k. Soit $\mathrm{alloc}_{a_k}(\vec{i}) = M\vec{i} + \alpha_k$, nous choisissons $\mathrm{alloc}_{a_{k,\mathrm{rot}}}(\vec{i}) = MU^{-1}\vec{i} + \alpha_k$ ($a_{k,\mathrm{rot}}(\vec{i})$ et $a_k(U^{-1}(\vec{i}))$ sont dans la mémoire du même processeur).

Comme à la page 51, soit $U^{-1} = (X_1\ X_2)$. Ainsi, nous avons

$$MU^{-1} = (MX_1\ MX_2)$$

et

$$UU^{-1} = \begin{pmatrix} \pi X_1 & \pi X_2 \\ MX_1 & MX_2 \end{pmatrix} = \mathrm{Id}.$$

Ainsi,

$$MU^{-1} = \left(\begin{array}{c|c} 0 & \\ \vdots & \mathrm{Id} \\ 0 & \end{array}\right).$$

La matrice de projection des nouveaux tableaux définit une projection parallèle à la première dimension de l'espace des processeurs.

D'autre part, après réécriture, l'accès au tableau a_k dans le nouveau nid est $a_k(U^{-1}\vec{j} + c_{a_k})$. Nous avons

$$\begin{aligned} a_k(U^{-1}\vec{j} + c_{a_k}) &= a_{k,\mathrm{rot}}(U(U^{-1}\vec{j} + c_{a_k})) \\ &= a_{k,\mathrm{rot}}(\vec{j} + Uc_{a_k}). \end{aligned}$$

Si nous remplaçons dans le nouveau nid toutes les occurrences du tableau a_k par les occurrences correspondantes du tableau $a_{k,\mathrm{rot}}$, nous obtenons à nouveau un nid réécrit avec uniquement des translations comme fonctions d'accès aux tableaux.

Résumé

Pour résumer notre approche, la stratégie pour générer le code après un ordonnancement linéaire est :

1. vérifier que le placement des données est compatible avec la règle des écriture locales, et si ce n'est pas le cas insérer des tableaux temporaires,
2. réécrire le nid de boucles,
3. vérifier si la matrice de projection correspond à une projection selon une dimension de l'espace d'itération, et si ce n'est pas le cas remplacer les tableaux initiaux par des tableaux qui ont subi une rotation, générer les boucles parallèles au début et à la fin du programme qui, respectivement, initialisent les tableaux tournés et recopient les résultats dans les tableaux initiaux,

4. générer les directives d'alignement de chaque tableau avec un « template »,

5. générer la directive de répartition pour placer le « template » sur les processeurs virtuels.

4.4.2 Codage de l'ordonnancement linéaire décalé

De l'ordonnancement linéaire à l'ordonnancement linéaire décalé

Comme expliqué dans la section précédente, nous sommes capables de réécrire le nid initial en ne prenant en compte que la partie linéaire de l'ordonnancement. Un tel nid réécrit ressemble au programme 4.6.

PROG. **4.6** Un nid après transformation par un ordonnancement linéaire

```
     do t=t^l, t^u
$HPF! INDEPENDENT
        do pr_1=pr_1^l(t), pr_1^u(t)
           ...
$HPF! INDEPENDENT
           do pr_d=pr_d^l(t), pr_d^u(t)
              S_1(t, pr_1, ..., pr_d)
              ...
              S_k(t, pr_1, ..., pr_d)
           enddo
        ...
        enddo
     enddo
```

Considérons l'ordonnancement suivant :

$$\mathrm{ord}(I) = \left\lfloor \frac{1}{q}(p\pi I + c) \right\rfloor \tag{4.1}$$

pour une instruction donnée.

La transformation précédente utilise $t = \pi I$. Donc le temps d'exécution donné par l'ordonnancement linéaire décalé (équation 4.1) peut être réécrit comme

$$\mathrm{exe}(t) = \left\lfloor \frac{1}{q}(pt + c) \right\rfloor . \tag{4.2}$$

Dans le code transformé du programme 4.6, les processeurs (virtuels) sur lesquels les calculs sont exécutés ne dépendent que du temps d'exécution t (et des paramètres du programme). Donc, si nous « inversons » la fonction exe nous pourrons réécrire le nid de boucles avec une nouvelle variable de temps correspondant au temps donné par l'ordonnancement ord.

Montrons maintenant que cet « inverse » est :

$$\mathrm{lin}(T) = \left\lceil \frac{1}{p}(qT - c) \right\rceil . \tag{4.3}$$

Nous allons prouver la proposition suivante :

Proposition 1

$$\forall t \in \mathbf{Z}, (t \in [\mathrm{lin}(T), \mathrm{lin}(T+1)[\Leftrightarrow \mathrm{exe}(t) = T)$$

Preuve Soit $t \in \mathbf{Z}$ tel que :

$$\text{lin}(T) \leq t < \text{lin}(T+1)$$

Nous pouvons déduire successivement :

$$\left\lceil \frac{1}{p}(qT - c) \right\rceil \leq t < \left\lceil \frac{1}{p}(q(T+1) - c) \right\rceil \tag{4.4}$$

et

$$\frac{1}{q}\left(p\left\lceil \frac{1}{p}(qT-c)\right\rceil + c\right) \leq \frac{1}{q}(pt+c) < \frac{1}{q}\left(p\left\lceil \frac{1}{p}(q(T+1)-c)\right\rceil + c\right).$$

Soit $f(T) = \frac{1}{q}\left(p\left\lceil \frac{1}{p}(qT-c)\right\rceil + c\right)$, nous avons alors :

$$f(T) \leq \frac{1}{q}(pt+c) < f(T+1).$$

Il existe toujours α et β tels que

$$qT - c = p\alpha + \beta, 0 \leq \beta < p, \alpha \in \mathbf{Z} \tag{4.5}$$

Calculons $f(T)$:

$$\begin{aligned} f(T) &= \frac{1}{q}\left(p\left\lceil \frac{1}{p}(qT-c)\right\rceil + c\right) \\ &= \frac{1}{q}\left(p\left\lceil \alpha + \frac{\beta}{p}\right\rceil + c\right) \\ &= \frac{1}{q}\left((p\alpha + c) + p\left\lceil \frac{\beta}{p}\right\rceil\right) \\ &= \frac{1}{q}\left((qT - \beta) + p\left\lceil \frac{\beta}{p}\right\rceil\right) \\ &= T + \frac{1}{q}\left(p\left\lceil \frac{\beta}{p}\right\rceil - \beta\right). \end{aligned}$$

Si nous prouvions que $\lfloor f(T) \rfloor = T$, nous obtiendrions alors

$$T \leq \left\lfloor \frac{1}{q}(pt+c)\right\rfloor < T+1$$

qui est équivalent à $\text{exe}(t) = T$, ce qui est ce que nous voulons prouver.

Discutons maintenant la valeur de

$$\lfloor f(T) \rfloor = \left\lfloor T + \frac{1}{q}\left(p\left\lceil \frac{\beta}{p}\right\rceil - \beta\right)\right\rfloor$$

en fonction de β.

- Si $\beta = 0$ alors $f(T) = T$ et $\lfloor f(T) \rfloor = T$.
- Sinon $0 < \beta < p$ et alors $\left\lceil \frac{\beta}{p}\right\rceil = 1$. Nous voudrions avoir $\frac{1}{q}(p-\beta) < 1$. Ce qui est équivalent à prouver

$$1 < \frac{q+\beta}{p}. \tag{4.6}$$

  - Si $p < q$ alors l'équation 4.6 est vérifiée.

– Sinon, comme $p \wedge q = 1$, $p > q$. Rappelons nous l'hypothèse (équation 4.4):

$$\left\lceil \frac{1}{p}(qT - c) \right\rceil \leq t < \left\lceil \frac{1}{p}(q(T+1) - c) \right\rceil .$$

Cela implique que

$$\left\lceil \frac{1}{p}(qT - c) \right\rceil < \left\lceil \frac{1}{p}(q(T+1) - c) \right\rceil .$$

Introduire α et β (équation 4.5) dans cette équation conduit à

$$\left\lceil \alpha + \frac{\beta}{p} \right\rceil < \left\lceil \alpha + \frac{\beta + q}{p} \right\rceil .$$

Comme α est un entier, nous avons

$$\left\lceil \frac{\beta}{p} \right\rceil < \left\lceil \frac{\beta + q}{p} \right\rceil .$$

Comme $\left\lceil \frac{\beta}{p} \right\rceil = 1$, nous avons

$$1 < \left\lceil \frac{\beta + q}{p} \right\rceil$$

qui implique l'équation 4.6.

Nous avons prouvé que dans tous les cas, $\lfloor f(T) \rfloor = T$, ce qui implique $\mathrm{exe}(t) = T$ et

$$\forall t \in \mathbf{Z}, (t \in [\mathrm{lin}(T), \mathrm{lin}(T+1)[\Rightarrow \mathrm{exe}(t) = T) .$$

Nous devons maintenant monter la réciproque.
Il est immédiat que

$$\bigcup_{T \in \mathbf{Z}} [\mathrm{lin}(T), \mathrm{lin}(T+1)[= \mathbf{Z}.$$

Donc, soit $t \in \mathbf{Z}$ et $T = \mathrm{exe}(t)$. L'équation précédente implique que

$$\exists T', \mathrm{lin}(T') \leq t < \mathrm{lin}(T'+1).$$

Nous venons de prouver que cela implique que $\mathrm{exe}(t) = T'$. Nous pouvons maintenant conclure que $T' = T$ et que

$$\forall t \in \mathbf{Z}, (T = \mathrm{exe}(t) \Rightarrow t \in [\mathrm{lin}(T), \mathrm{lin}(T+1)[).$$

□

Transformation formelle

Bornes du temps Après la transformation linéaire, le temps t varie de t^l à t^u. Nous devons trouver les bornes du temps final T qui correspond à l'intervalle $[t^l, t^u]$. Nous prenons comme temps d'exécution :

$$\mathrm{exe}_i(t^l) \leq T \leq \mathrm{exe}_i(t^u) \tag{4.7}$$

où $\mathrm{exe}_i(t) = \left\lfloor \frac{1}{q}(pt + c_i) \right\rfloor$.

PROG. **4.7** Code pour l'instruction S_i avec un ordonnancement linéaire décalé

```
     do T=exe_i(t^l), exe_i(t^u)
$HPF! INDEPENDENT
        do t=lin_i(T), lin_i(T+1)-1
$HPF! INDEPENDENT
           do pr_1=pr_1^l(t), pr_1^u(t)
           ...
$HPF! INDEPENDENT
              do pr_d=pr_d^l(t), pr_d^u(t)
                 S_i(t,pr_1,...,pr_d)
              enddo
           ...
           enddo
        enddo
     enddo
```

En utilisant la proposition 1, pour l'instruction S_i, en considérant $\text{lin}_i(t) = \left\lceil \frac{1}{p}(qt - c_i) \right\rceil$, nous obtenons le code du programme 4.7.

Vérifions que cette transformation est bien celle que nous cherchions : l'index de boucle `T` définit dans le programme 4.7 correspond-it à l'ordonnancement ? Considérons le point d'itération I. L'instance $S_i(I)$ de l'instruction S_i devrait être exécutée à l'instant $T = \text{exe}_i(\pi I)$. La proposition 1 prouve que πI est dans l'intervalle de définition de t. Donc l'instruction $S_i(I)$ est en fait bien exécutée au moment voulu sur les processeurs assignés par la phase de placement.

Quand il y a plusieurs instructions Il serait intéressant ici d'avoir une construction pour exprimer le parallélisme de contrôle en HPF. Pour le moment [4] une telle construction n'existe pas, mais au moment de l'écriture de cette thèse, elle est en cours de discussion au sein du Forum HPF pour inclusion dans la spécification d'HPF 2.

En effet, nous voudrions exécuter en parallèle toutes les instances des instructions qui ont la même valeur d'ordonnancement. Comme ce n'est pas possible actuellement, nous les exécutons simplement séquentiellement, il est vraissemblable qu'elles seraient séquentialisées de toute manière par le compilateur.

Le dernier problème que nous devons résoudre est la prise en compte de différentes constantes de décalage pour différentes instructions. Ce n'est pas très difficile :

- premièrement, faisons varier le temps séquentiel T de la borne inférieure du temps donnée par la plus petite constante à la borne supérieure donnée par la plus grande : T varie de T^l à T^u définies par :

$$T^l = \left\lfloor \frac{1}{q} \left(pt^l + \min_i c_i \right) \right\rfloor$$
$$T^u = \left\lfloor \frac{1}{q} \left(pt^u + \max_i c_i \right) \right\rfloor$$

- et ensuite vérifions pour chaque instruction que le temps linéaire t correspondant ne sort pas de son intervalle de définition $[t^l, t^u]$. Cela peut être fait en faisant varier t de $\max(t^l, \text{lin}_i(T))$ à $\min(t^u, \text{lin}_i(T+1) - 1)$.

4. la spécification courante est HPF 1 et est décrite dans [KLS+94]

Attention : *ce style d'écriture de programmes est correct car, en Fortran, si une boucle* `do` *a sa borne inférieure plus grande que sa borne supérieure, alors le corps de la boucle n'est jamais exécuté.*

Quelques optimisations Pour éviter le plus souvent possible le calcul des fonctions min et max dans les bornes de boucles, on peut distinguer trois étapes dans l'exécution du programme parallélisé :

1. L'étape initiale quand certaines instructions ne sont pas encore exécutées : on calcule la borne inférieure comme $\max(t^l, \mathrm{lin}_i(T))$ mais on enlève le min dans la borne supérieure qui devient alors $\mathrm{lin}_i(T+1) - 1$. Ceci est valable pour les valeurs de T variant entre

$$\left\lfloor \frac{1}{q}(pt^l + \min_i c_i) \right\rfloor$$

et

$$\left\lceil \frac{1}{q}(pt^l + \max_i c_i) \right\rceil - 1.$$

2. L'étape de régime permanent quand toutes les instructions sont toujours exécutées : il n'y a plus de min ni de max. Ceci est valable pour les valeurs de T variant entre

$$\left\lceil \frac{1}{q}(pt^l + \max_i c_i) \right\rceil$$

et

$$\left\lfloor \frac{1}{q}(pt^u + \min_i c_i) \right\rfloor - 1.$$

3. L'étape finale quand certaines instructions ont fini leur exécution : on enlève le max de la borne inférieure : $\mathrm{lin}_i(T)$ et on calcule la borne supérieure comme $\min(\mathrm{lin}_i(T+1) - 1)$. Ceci est valable pour les valeurs de T variant entre

$$\left\lfloor \frac{1}{q}(pt^u + \min_i c_i) \right\rfloor$$

et

$$\left\lfloor \frac{1}{q}(pt^u + \max_i c_i) \right\rfloor.$$

Nous supposons ici que n, le paramètre de taille du programme est suffisamment grand.

La simplification symbolique des bornes de boucles a aussi été abordée. Le module qui travaille sur les expressions symboliques met les expressions affines sous une forme « normalisée » (où chaque variable n'apparaît qu'une fois), ensuite les fonctions partie entières inférieure et supérieure sont enlevées si elles s'appliquent à des expressions entières et le nombre de divisions (et donc d'erreurs d'arrondi) est réduit en factorisant les dénominateurs communs des fractions. Cela conduit à un code plus lisible et à des bornes de boucles moins coûteuses en calcul.

Une troisième optimisation s'occupe du cas des boucles non exécutées parce que leur borne inférieure est toujours plus grande que leur borne supérieure. Dans ce cas, elle est éliminée du code généré. Si une boucle a ses deux bornes égales, alors elle est remplacée par une affectation dans une variable à la valeur commune des deux bornes. Ces simplifications donnent un code plus lisible par l'utilisateur humain et aussi plus facile à analyser par un compilateur.

4.4.3 Commentaires

Nous avons présenté dans cette section les problèmes à résoudre lors de la génération du code et les solutions implémentées dans Bouclettes. Le choix d'HPF comme langage cible s'est révélé critique sur plusieurs points. En effet, l'utilisation d'HPF nous a permis de ne pas nous occuper de la génération des communications de bas niveau dans le programme parallèle. D'un autre côté, des complications sont nées des limitations d'HPF :

- Le fait qu'HPF respecte la règle des écritures locales nous a forcé à générer des tableaux temporaires quand le placement n'est pas compatible avec cette règle. Il faut noter que l'utilisateur dispose de l'option de forcer Bouclettes à respecter cette règle des écritures locales.

- Les stratégies de répartition de données autorisées par HPF ne sont pas toujours assez performantes pour exprimer le placement calculé. Nous avons donc développé une méthode de redistribution pour pouvoir traiter tous nos placements.

- Le data-parallélisme pur d'HPF n'autorise aucun parallélisme de contrôle qui serait nécessaire pour exprimer tout le parallélisme exposé par certains ordonnancements.

4.5 Exemple détaillé

Dans cette section nous étudions la parallélisation complète d'un exemple depuis l'analyse de dépendance jusqu'à la génération de code. Les deux méthodes d'ordonnancement sont présentées et pour l'ordonnancement linéaire, on a deux versions : avec ou sans redistribution.

4.5.1 Le programme d'entrée

L'exemple que nous considérons (voir programme 4.8) est un nid de profondeur $d = 2$ avec deux instructions internes. C'est un exemple d'école destiné à montrer la difficulté de la réécriture (et aussi de l'extraction du parallélisme).

PROG. **4.8** Exemple : programme d'entrée

```
      parameter (n=100)

      integer i,j
      real a(n,n)
      real b(n,n)

      do i= 2, n-5
         do j= 6, n-3
c   Instruction 1
            a(i,j)=a(i,j-5)+b(i-1,j+3)-a(i+5,j-4)
c   Instruction 2
            b(i+1,j-2)=a(i,j-1)
         enddo
      enddo
```

Bouclettes trouve les quatre dépendances de données :

Depuis l'instruction	Vers l'instruction	Vecteur de dépendance
2	1	$\binom{2}{-5}$
1	2	$\binom{0}{1}$
1	1	$\binom{5}{-4}$
1	1	$\binom{0}{5}$

La première ligne, par exemple, veut dire que l'élément de tableau produit par l'instance (i, j) de l'instruction 2 est utilisé par l'instance $(i + 2, j - 5)$ de l'instruction 1.

4.5.2 Ordonnancement linéaire sans redistribution

Le vecteur d'ordonnancement linéaire optimal est $(3 \quad 1)$. La matrice de projection construite par Bouclettes est $(1 \quad 0)$ et les constantes d'alignement sont :

tableau	décalage	instruction	décalage
a	0	1	0
b	1	2	0

On peut facilement vérifier que la règle des écritures locales est vérifiée ici.[5] Ainsi, le code ne comprendra pas de temporaires qui auraient été créés si cela n'avait pas été le cas. Comme la matrice de projection est $(1, 0)$, la projection des tableaux bidimensionnels est faite sur la première dimension suivant la deuxième. La redistribution n'est donc pas nécessaire.[6]

Le code résultant est le programme 4.9.[7]

Après les déclarations, on a les directives de répartition et d'alignement des données. Elles sont générées à partir de la projection et des constantes de décalage du placement comme expliqué dans la section 4.4.1. La directive `DISTRIBUTE` utilise une stratégie par `BLOCK`s pour projeter le « template » sur les processeurs. Comme cela a déjà été mentionné, n'importe quelle stratégie pourrait être utilisée ici et une approche bloc-cyclique avec une taille de blocs dépendant de la machine cible serait probablement une meilleure solution.

Le nid de boucles transformé consiste en une boucle séquentielle (indice `T`) encadrant une boucle parallèle (indice `P1`). `T` et `P1` sont obtenus à partir des indices de boucles initiaux par :

$$\begin{pmatrix} \texttt{T} \\ \texttt{P1} \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \texttt{i} \\ \texttt{j} \end{pmatrix}.$$

4.5.3 Ordonnancement linéaire avec redistribution

L'utilisateur peut choisir de forcer une redistribution des données, Dans ce cas, les tableaux de données sont copiés temporairement dans d'autres tableaux avec lesquels ont réécrit un nid qui est en général plus simple à analyser pour un compilateur HPF (voir section 4.6). Cela vient du

5. Elle est en fait vérifiée très souvent en pratique, même quand on ne la force pas. Cela se comprend bien en constatant que son application élimine toutes les communications dûes aux stockage des résultats des calculs.

6. Il est assez rare que la redistribution soit nécessaire vu l'algorithme utilisé pour calculer la projection. En effet, il suffit qu'il y ait un 1 dans le vecteur de temps pour que cette matrice corresponde à une projection parallèle à un axe.

7. Dans tous les programmes les variables ont été renommées pour une lisibilité accrue.

PROG. **4.9** Exemple : programme HPF avec ordonnancement linéaire et sans redistribution

```
      PROGRAM boucle

      INTEGER P1
      INTEGER T
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)

!HPF$ TEMPLATE BCLT_template(n+1,n+3)
!HPF$ DISTRIBUTE BCLT_template(BLOCK,*)
!HPF$ ALIGN a(i1,i2) WITH BCLT_template(i1,i2+3)
!HPF$ ALIGN b(i1,i2) WITH BCLT_template(i1+1,i2)
      DO T = 12, 4*n-18
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+T)/3.0+1,2)), floor(min((T)/3.0-2,n-5))
            a(P1,T-3*P1) = (a(P1,T-3*P1-5)+(b(P1-1,T-3*P1+3)-a(P1+5,T-3*P1-4)))
            b(P1+1,T-3*P1-2) = a(P1,T-3*P1-1)
         END DO
      END DO
      END
```

fait qu'en faisant cette redistribution, les fonctions d'accès affines aux tableaux sont remplacées par des translations qui sont beaucoup mieux optimisées par les compilateurs HPF actuels. Le programme 4.10 présente le code résultant de cette redistribution.

Les instructions `FORALL` expriment la redistribution avant et après le calcul du nid transformé utilisant les tableaux temporaires. Remarquons que les fonctions d'accès dans le nid central sont bien des translations et étaient plus compliquées sans la redistribution.

4.5.4 Ordonnancement linéaire décalé

Nous étudions ici l'ordonnancement linéaire décalé sans redistribution. Il est bien sûr possible de redistribuer les tableaux comme dans la section précédente mais cela ne montrerait rien de nouveau si ce n'est un code encore plus compliqué.

Les fonctions[8] d'ordonnancement linéaire décalé sont :

$$\begin{cases} \mathrm{ord}_1(I) = \left\lfloor \frac{1}{5}([7,1].I) \right\rfloor \\ \mathrm{ord}_2(I) = \left\lfloor \frac{1}{5}([7,1].I + 4) \right\rfloor \end{cases}$$

Le placement des tableaux (et des calculs) est le même que celui que nous avions obtenu pour l'ordonnancement linéaire précédent. Le code produit par Bouclettes peut se découper en trois étapes (voir section 4.4.2) :

1. L'étape initiale (voir le programme 4.11) est limitée à une unité de temps (`T = 4`) et nous pouvons remarquer la fonction `max` dans la borne inférieure de la boucle sur le temps virtuel `VT`. Cette fonction assure que les différentes instructions ne sont calculées que pendant le bon intervalle de temps dû aux constantes de temps. Chaque instruction est à l'intérieur d'un nid

8. $\mathrm{ord}_i(I)$ est la fonction d'ordonnancement de l'instruction i

PROG. **4.10** Exemple : programme HPF avec ordonnancement linéaire et redistribution

```
      PROGRAM boucle

      INTEGER I1
      INTEGER I0
      INTEGER P1
      INTEGER T
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)
      REAL ROTa(4*n-3,n)
      REAL ROTb(4*n-3,n)

!HPF$ TEMPLATE BCLT_template(4*n,n+10)
!HPF$ DISTRIBUTE BCLT_template(*,BLOCK)
!HPF$ ALIGN ROTa(i1,i2) WITH BCLT_template(i1+3,i2)
!HPF$ ALIGN ROTb(i1,i2) WITH BCLT_template(i1,i2+10)
      FORALL (I0 = 1:4*n-3,I1 = 1:n)
         ROTa(I0,I1) = 0
      END FORALL
      FORALL (I0 = 1:4*n-3,I1 = 1:n)
         ROTb(I0,I1) = 0
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         ROTa(3*I0+I1-3,I0) = a(I0,I1)
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         ROTb(3*I0+I1-3,I0) = b(I0,I1)
      END FORALL
      DO T = 12, 4*n-18
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+T)/3.0+1,2)), floor(min((T)/3.0-2,n-5))
            ROTa(T-3,P1) = (ROTa(T-8,P1)+(ROTb(T-3,P1-1)-ROTa(T+8,P1+5)))
            ROTb(T-2,P1+1) = ROTa(T-4,P1)
         END DO
      END DO
      FORALL (I0 = 1:n,I1 = 1:n)
         a(I0,I1) = ROTa(3*I0+I1-3,I0)
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         b(I0,I1) = ROTb(3*I0+I1-3,I0)
      END FORALL
      END
```

de profondeur 2 : l'indice `VT` énumère les instances de l'instruction qui sont ordonnancées au même moment (ici `T = 4`), et l'indice `P1` énumère les processeurs (virtuels) concernés par ce calcul.

PROG. **4.11** Exemple : ordonnancement linéaire décalé, étape d'initialisation

```
      PROGRAM boucle

      INTEGER T
      INTEGER VT
      INTEGER P1
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)

!HPF$ TEMPLATE BCLT_template(n+1,n+7)
!HPF$ DISTRIBUTE BCLT_template(BLOCK,*)
!HPF$ ALIGN a(i1,i2) WITH BCLT_template(i1,i2+7)
!HPF$ ALIGN b(i1,i2) WITH BCLT_template(i1+1,i2)
      T = 4
!HPF$ INDEPENDENT
      DO VT = max(20,5*T), 5*T+4
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
            a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
         END DO
      END DO
!HPF$ INDEPENDENT
      DO VT = max(20,5*T-4), 5*T
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
            b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
         END DO
      END DO
```

2. L'étape de régime permanent (voir programme 4.12) est l'étape principale pendant laquelle il n'y a pas de problème causé par les constantes de décalage (toutes les instructions sont évaluées à chaque instant), tout est régulier. Une fois de plus nous avons les deux nids de boucles parallèles à l'intérieur de la boucle séquentielle sur le temps.

3. L'étape finale (voir le programme 4.13) est le pendant de l'étape initiale pour traiter la fin des calculs avec les problèmes venant des constantes de décalage temporel.

4.6 Évaluation expérimentale du code produit par Bouclettes

Nous évaluons dans cette section le code produit par Bouclettes sur différents compilateurs HPF, et plus particulièrement ADAPTOR [BZ94]. Des compilateurs de recherche pour du Fortran data-parallèle ont été développés dans les projets Superb [ZBG88], Kali [KM91], Fortran 77D [HKT94], ADAPT [Mer91], et Fortran 90D/HPF [BCF+93]. Les premiers compilateurs HPF commerciaux

PROG. **4.12** Exemple : ordonnancement linéaire décalé, étape de régime permanent

```
      DO T = 5, (floor((8*n-38)/5.0)-1)
!HPF$ INDEPENDENT
         DO VT = 5*T, 5*T+4
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
            END DO
         END DO
!HPF$ INDEPENDENT
         DO VT = 5*T-4, 5*T
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
            END DO
         END DO
      END DO
```

PROG. **4.13** Exemple : ordonnancement linéaire décalé, étape finale

```
      DO T = floor((8*n-38)/5.0), floor((8*n-34)/5.0)
!HPF$ INDEPENDENT
         DO VT = 5*T, min(8*n-38,5*T+4)
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
            END DO
         END DO
!HPF$ INDEPENDENT
         DO VT = 5*T-4, min(8*n-38,5*T)
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
            END DO
         END DO
      END DO
      END
```

disponibles ont été le compilateur xHPF d'Applied Parallel Research [For93], le compilateur pghpf de Portland Group [PGH94] et le compilateur Fortran 90 de DEC. D'autres compilateurs ont été annoncés depuis et sont maintenant disponibles. Bien que certains de ces compilateurs soient capables d'identifier des boucles parallèles, aucun d'entre eux ne parallélise le code comme c'est fait dans Bouclettes.

Nous avons couplé pour cette évaluation Bouclettes et ADAPTOR de telle façon que Bouclettes génère du code qui pourrait être compilé efficacement avec ADAPTOR qui met en œuvre les techniques d'optimisation les plus avancées à l'heure actuelle.

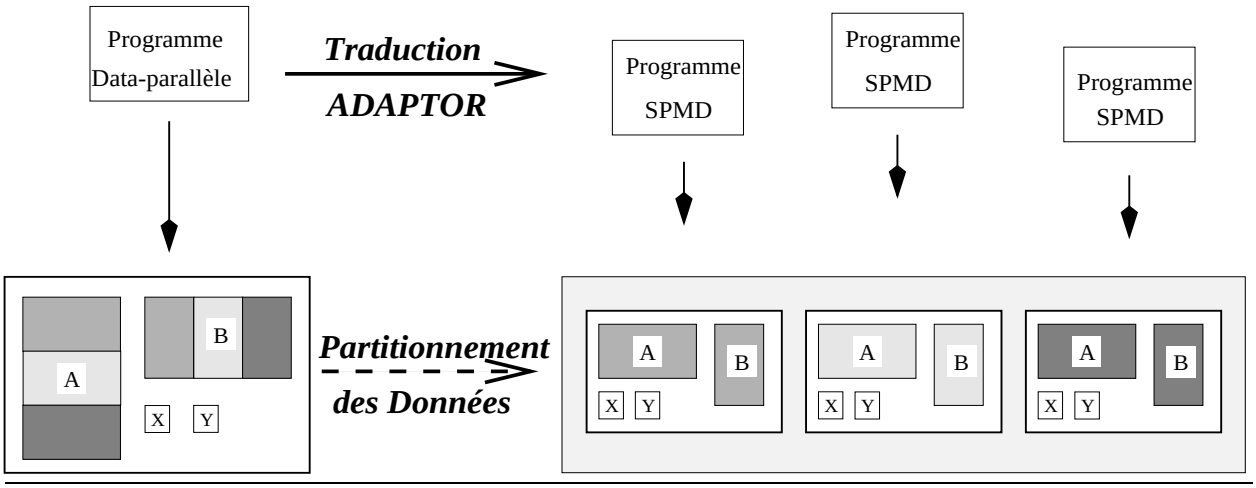
4.6.1 Description rapide d'ADAPTOR

ADAPTOR (Automatic Data Parallelism Translator) est un système développé à GMD pour compiler des programmes data-parallèles en des programmes équivalents avec communications explicites par envoi de messages. Ce système supporte la plupart des caractéristiques des langages data-parallèles utilisés dans le contexte de Fortran : Connection Machine Fortran (CMF) et High Performance Fortran (HPF).

ADAPTOR permet l'utilisation du paradigme de programmation data-parallélisme pour des machines MIMD[9] depuis plus de deux ans alors que les compilateurs commerciaux arrivent juste sur le marché. Pendant le développement du système, les idées directrices ont plus été la fiabilité du système et la justesse et l'efficacité des programmes générés que la complétude du langage.

ADAPTOR a été distribué comme programme du domaine public et les commentaires de nombreux utilisateurs ont aidé à améliorer la fonctionnalité et la stabilité de cet outil de traduction.

FIG. **4.2** - Partitionnement MIMD avec ADAPTOR



Au moyen d'une transformation source à source, ADAPTOR traduit le programme data-parallèle en un programme SPMD[10] qui fonctionne sur tous les nœuds disponibles. L'idée principale de la traduction est de répartir les tableaux du programme source sur les nœuds (processeurs) où les boucles parallèles et les opérations de tableaux sont restreintes à la partie locale possédée par le processeur (voir figure 4.2). Les instructions de communication pour échanger des données non locales sont générées automatiquement. Le flux de contrôle et les instructions scalaires sont répliqués sur tous les nœuds.

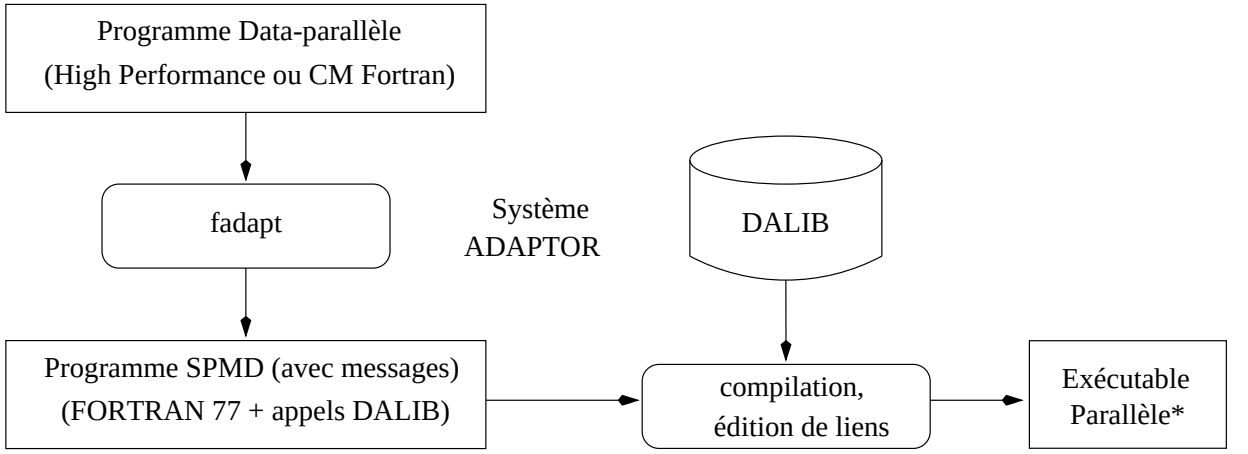
9. *Multiple Instruction flow, Multiple Data flow*
10. *Single Program Multiple Data*

Beaucoup d'optimisations ont été implémentées dans la dernière version (3.1) d'ADAPTOR.

Au regard du langage cible du programme SPMD généré, ADAPTOR est très flexible. Il peut non seulement générer des programmes Fortran 77 ou Fortran 90 avec des communications par messages, mais aussi du Fortran 77 avec des constructions supplémentaires (par exemple des tableaux dynamiques ou des opérations de tableaux).

À côté du système de traduction, un environnement d'exécution appelé « DALIB » (*distributed array library*) a été développé. Il est lié avec le programme avec messages (voir figure 4.3). Cette bibliothèque possède des fonctions pour des réductions globales, des transpositions, des opérations de regroupement et d'éclatement[11], des décalages circulaires, des redistributions et réplications de tableaux locaux ou répartis. Des possibilités de chronométrage et de traçage ainsi qu'un générateur de nombres aléatoires sont aussi disponibles dans cette bibliothèque. Comme l'environnement d'exécution est disponible sur la plupart des systèmes parallèles, les programmes générés fonctionneront sur toutes ces machines.

FIG. **4.3 -** Aperçu d'ADAPTOR



4.6.2 Résultats

Nous présentons ici quelle efficacité peut être obtenue avec les techniques de compilations actuelles. Tous les résultats sont mesurés sur un IBM SP 2 à nœuds fins, AIX version 3.2, le compilateur pghpf version 1.3, ADAPTOR version 3.1, et le compilateur XL Fortran version 3.2.

Nous présentons deux exemples : la relaxation de Gauss-Seidel et un exemple artificiel que nous avons appelé « Mtest ».

Relaxation de Gauss-Seidel

Code séquentiel Le programme 4.14 montre le code séquentiel utilisé.

En partant de ce code, le compilateur HPF de PGI et ADAPTOR n'ont été capables ni l'un ni l'autre de détecter du parallélisme. Les deux compilateurs génèrent du code SPMD comme celui du programme 4.15.

Alors que le compilateur PGI diffuse les valeurs non locales, ADAPTOR échange les valeurs entre processeurs voisins mais au prix d'un temps de calcul des possesseurs plus élevé (voir la table 4.1).

Le surcoût spectaculaire dû aux envois et réception de valeurs non locales une à une et au calcul des propriétaires domine totalement le calcul dans le programme SPMD.

11. *Gather* et *scatter* en anglais.

PROG. **4.14** Code séquentiel de la relaxation de Gauss-Seidel

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A
      ...
      DO I = 2, N-1
        DO J = 2, N-1
          A(I,J) = (A(I,J-1)+A(I-1,J)+A(I,J+1)+A(I+1,J))*0.25
        END DO
      END DO
```

PROG. **4.15** Code SPMD donné par la compilation de la version séquentielle de la relaxation de Gauss-Seidel

```
      DO J = 2, N-1
        DO I = 2, N-1
c         récupère des copies locales des valeurs non locales A(I,J-1) et A(I,J+1)
          IF (HAVE I A(I,J)) then
             A(I,J) = (A(I,J-1)+A(I-1,J)+A(I,J+1)+A(I+1,J))*0.25
          END IF
        END DO
      END DO
```

TAB. **4.1 -** Résultats pour le code séquentiel de la relaxation de Gauss-Seidel

N = 256	1 nœud	2 nœuds	4 nœuds	8 nœuds
PGI HPF	7,6 s	4,8 s	3,8 s	2,5 s
ADAPTOR	4,8 s	4,9 s	4,8 s	4,6 s

Code généré par Bouclettes Pour le nid étudié, Bouclettes produit le code du programme 4.16

PROG. **4.16** Code généré par Bouclettes pour la relaxation de Gauss-Seidel

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A
!HPF$ DISTRIBUTE A(*,BLOCK)
      ...
      DO T = 4, 2*N-2
         FORALL (J=MAX(2,T-N+1):MIN(N-1,T-2))
     &      A(T-J,J) = (A(T-J,J-1)+A(T-J-1,J)+A(T-J,J+1)+A(T-J+1,J))*0.25
      END DO
```

Le compilateur PGI HPF n'a pas su tirer avantage du data-parallélisme dans la boucle `FORALL` et c'est pourquoi le programme SPMD qu'il produit utilise encore des boucles séquentielles avec à peu près les mêmes temps d'exécution que précédemment. Le système ADAPTOR tire avantage du parallélisme de l'instruction `FORALL`. La table 4.2 montre les résultats.

TAB. **4.2 -** Résultats pour le code généré par Bouclettes, compilé par ADAPTOR pour la relaxation de Gauss-Seidel

ADAPTOR	1 nœud	2 nœuds	4 nœuds	8 nœuds
N = 256	0,06 s	0,14 s	0,15 s	0,16 s
N = 512	0,11 s	0,35 s	0,40 s	0,42 s
N = 1024	0,78 s	1,36 s	1,37 s	1,35 s
N = 2048	3,55 s	3,66 s	4,06 s	4,84 s
N = 4096	13,42 s	20,05 s	18,34 s	15,56 s

Les fonctions d'accès aux tableaux affines dans le code généré ont la conséquence que le compilateur HPF ne sait pas optimiser le code et échange trop de données à chaque étape (à cause de l'utilisation de zones de recouvrement qui vectorisent les messages mais peut générer des communications inutiles).

Code généré par Bouclettes avec redistribution Grâce à la redistribution que génère Bouclettes (voir le programme 4.17), il est garanti que les fonctions d'accès aux tableaux deviennent des translations. Dans ce cas, les compilateurs HPF peuvent optimiser les communications.

Avec le système ADAPTOR, l'exécution parallèle du nid de boucles sur le tableau redistribué donne les résultats de la table 4.3. Si on les compare aux résultats sans la redistribution, on note une meilleurs extensibilité. Le compilateur PGI HPF génère maintenant aussi un code efficace en tirant avantage de la boucle parallèle et en utilisant des communications efficaces.

Néanmoins, la redistribution en elle-même et la mémoire additionnelle pour les nouveaux tableaux doivent être prises en compte. Le compilateur PGI HPF a complètement séquentialisé la redistribution et les temps d'exécution totaux sont similaires aux temps obtenus avec le programme séquentiel. Avec ADAPTOR, la redistribution est parallélisée et ne domine pas le temps de calcul, mais elle n'est pas encore très efficace. Ceci est principalement dû aux fonctions affines pour les accès aux tableaux qui apparaissent maintenant dans la redistribution.

PROG. **4.17** Code généré par Bouclettes avec redistribution pour la relaxation de Gauss-Seidel

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A
      REAL, DIMENSION (N,2:2*N)  :: A1
!HPF$ DISTRIBUTE A1(*,BLOCK)
      ...
      FORALL (I=1:N,J=1:N) A1(I,I+J) = A(I,J)
      ...
      DO T = 4, 2*N-2
         FORALL (2=MAX(2,T-N+1):MIN(N-1,T-2))
     &      A1(I,T) = (A1(I,T-1)+A1(I-1,T-1)+A1(I,T+1)+A1(I+1,T+1))*0.25
      END DO
      ...
      FORALL (I=1:N,J=1:N) A(I,J) = A1(I,I+J)
```

TAB. **4.3 -** Résultats pour le code généré par Bouclettes avec redistribution, compilé par ADAPTOR pour la relaxation de Gauss-Seidel

ADAPTOR	1 nœud	2 nœuds	4 nœuds	8 nœuds
N = 256	0,05 s	0,11 s	0,12 s	0,12 s
N = 512	0,15 s	0,24 s	0,26 s	0,28 s
N = 1024	0,91 s	0,80 s	0,63 s	0,60 s
N = 2048	3,48 s	2,71 s	1,79 s	1,43 s
N = 4096	mem !	9,42 s	5,31 s	3,67 s

Résultats pour Mtest

Code séquentiel Nous utilisons ici le code tridimensionnel donné par le programme 4.18. Les temps de calcul et la taille des tableaux sont de l'ordre de $O(N^3)$.

La table 4.4 montre les temps d'exécution pour ce code séquentiel (en utilisant le compilateur XL Fortran).

TAB. **4.4 -** Résultats pour le code séquentiel de Mtest

xlf	1 nœud
N = 50	0,21 s
N = 75	0,85 s
N = 100	2,76 s
N = 150	14,12 s

Ordonnancement linéaire (avec redistribution) Pour le nid de boucles considéré, Bouclettes donne comme vecteur d'ordonnancement linéaire, le vecteur $\begin{pmatrix} \frac{3}{4} & \frac{1}{2} & \frac{-9}{4} \end{pmatrix}$. La matrice de projection déduire par complétion unimodulaire de ce vecteur est :

$$M = \begin{pmatrix} 1 & 1 & -5 \\ 0 & 0 & 1 \end{pmatrix}.$$

Cette matrice ne correspond pas à une projection selon un axe, la redistribution est donc nécessaire.

PROG. **4.18** Code séquentiel de Mtest

```
PARAMETER (N=...)
REAL, DIMENSION (N,N,N) :: A, B, C, D
...
DO i = 2, n-1
  DO j = 7, n-5
    DO k = 3, n-6
      a(i,j,k) = (b(i,j-6,k-1)+d(i-1,j+3,k+1))
      b(i+1,j-1,k) = c(i+2,j+5,k+2)
      c(i+3,j-1,k-2) = a(i,j-2,k)
      d(i,j-1,k) = a(i,j-1,k+6)
    END DO
  END DO
END DO
```

Le programme 4.19 montre le code généré par Bouclettes avec cet ordonnancement.

ADAPTOR et le compilateur PGI HPF n'ont pas su tirer avantage du parallélisme. La table 4.5 montre les résultats pour un problème de petite taille (N= 30).

Les problèmes suivants ont pu être identifiés :

- Le code généré a besoin de tableaux très gros qui consomment à peu près 100 fois (7×14) plus de mémoire que ceux de la version séquentielle. Pour cette raison, le code ne tourne que pour des problèmes de très petite taille.

- La redistribution est très inefficace à cause des accès affines aux tableaux (par exemple `3*J0+2*J1-9*J2`) qui empêchent le compilateur de générer un code de communication efficace.

- Dans le nid de calcul, seule la boucle la plus interne des deux boucles parallèles a été réellement parallélisée. Comme l'indice de cette boucle interne, `J2` dépend de `J1`, le domaine d'itération n'est pas rectangulaire et le compilateur n'a pas généré de code efficace.

- Le compilateur n'a pas tiré avantage des zones de recouvrement.

TAB. **4.5 -** Résultats pour le code généré par Bouclettes avec l'ordonnancement linéaire pour Mtest

ADAPTOR	1 nœud	2 nœuds	4 nœuds	8 nœuds
pré-copie	13,9 s	11,6 s	12,4 s	11,5 s
post-copie	13,5 s	11,3 s	12,1 s	11,1 s
calcul	3,1 s	4,3 s	4,5 s	5,1 s
total	30,5 s	27,2 s	29,0 s	27,7 s
PGI HPF	**1 nœud**	**2 nœuds**	**4 nœuds**	**8 nœuds**
pré-copie	7,4 s	5,6 s	7,7 s	5,5 s
post-copie	7,2 s	5,2 s	7,9 s	6,3 s
calcul	3,8 s	4,4 s	8,3 s	26,2 s
total	18,4 s	15,2 s	23,9 s	38,1 s

PROG. **4.19** Code généré par Bouclettes pour Mtest avec l'ordonnancement linéaire

```
      REAL, DIMENSION (14*n-13,7*n-6,n) :: ROT_1_a, ROT_1_b
      REAL, DIMENSION (14*n-13,7*n-6,n) :: ROT_1_c, ROT_1_d

!HPF$ TEMPLATE BCLT_0_template(14*n+143,7*n+174,n+6)
!HPF$ DISTRIBUTE BCLT_0_template(*,BLOCK,BLOCK)
!HPF$ ALIGN ROT_1_a(i1,i2,i3) WITH BCLT_0_template(i1+156,i2,i3+6)
!HPF$ ALIGN ROT_1_b(i1,i2,i3) WITH BCLT_0_template(i1+76,i2+87,i3+2)
!HPF$ ALIGN ROT_1_c(i1,i2,i3) WITH BCLT_0_template(i1+72,i2+104,i3+4)
!HPF$ ALIGN ROT_1_d(i1,i2,i3) WITH BCLT_0_template(i1,i2+180,i3)
      ...
      FORALL (J0 = 1:n,J1 = 1:n,J2 = 1:n)
         ROT_1_a(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = a(J0,J1,J2)
         ROT_1_b(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = b(J0,J1,J2)
         ROT_1_c(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = c(J0,J1,J2)
         ROT_1_d(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = d(J0,J1,J2)
      END FORALL

     DO J0 = -9*n+74, 5*n-46

        FORALL (J1 = ceiling(max(-2*n+(J0+43)/3.0,
     &                           -n+(J0+9)/2.0,
     &                           (-7*n+5*J0+23)/9.0)):
     &               floor(min((n+J0-23)/3.0,
     &                         (J0-5)/2.0,
     &                         (5*J0-19)/9.0)),
     &          J2 = ceiling(max(3.0,(J0+7)/6.0-J1/2.0,
     &                               -n+J0-2*J1+3.0)):
     &              floor(min(n-6.0,(n+J0-5)/6.0-J1/2.0,
     &                                J0-2*J1-2.0)))

           ROT_1_a(9*n+J0-4,5*n+J1-1,J2) =
     &        ROT_1_b(9*n+J0-7,5*n+J1-2,J2-1)+
     &        ROT_1_d(9*n+J0-10,5*n+J1-4,J2+1)

           ROT_1_b(9*n+J0-3,5*n+J1-1,J2) =
     &         ROT_1_c(9*n+J0-6,5*n+J1-4,J2+2)

           ROT_1_c(9*n+J0+21,5*n+J1+11,J2-2) =
     &        ROT_1_a(9*n+J0-8,5*n+J1-3,J2)

           ROT_1_d(9*n+J0-6,5*n+J1-2,J2) =
     &        ROT_1_a(9*n+J0-60,5*n+J1-32,J2+6)
        END FORALL
      END DO

      FORALL (J0 = 1:n,J1 = 1:n,J2 = 1:n)
         a(J0,J1,J2) = ROT_1_a(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
         b(J0,J1,J2) = ROT_1_b(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
         c(J0,J1,J2) = ROT_1_c(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
         d(J0,J1,J2) = ROT_1_d(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
      END FORALL
```

Si une distribution de données unidimensionnelle est utilisée à la place de la distribution bidimensionnelle,

```
!HPF$ DISTRIBUTE BCLT_0_template(*,*,BLOCK)
```

le code généré est plus efficace car des zones de recouvrement peuvent être utilisées et un nombre inférieur de communications est généré.

Ordonnancement linéaire décalé Le programme 4.20 donne le code généré par Bouclettes avec l'ordonnancement linéaire décalé sans la redistribution.

ADAPTOR a pu générer du code très efficace (voir la table 4.6). Ce code est aussi bien extensible quand on fait varier le nombre de processeurs. Le compilateur PGI HPF n'a pas pu quant à lui générer un code efficace.

Un résultat remarquable est que le code tournant sur un nœud est plus rapide que sa contrepartie séquentielle. Cela vient du fait que les boucles les plus profondément imbriquées itèrent sur le premier indice et donc que le cache est mieux utilisé (pas[12] de 1 pour les accès mémoire aux tableaux). Il faut bien noter que c'est presque complètement un hasard. Cela vient du fait que la partie linéaire de l'ordonnancement est $\begin{pmatrix} 0 & 0 & -1 \end{pmatrix}$ et que la matrice de projection est

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

TAB. **4.6 -** Résultats pour le code généré par Bouclettes avec l'ordonnancement linéaire décalé, sans redistribution, pour Mtest

ADAPTOR	1 nœud	2 nœuds	4 nœuds	8 nœuds
50	0,07 s	0,13 s	0,11 s	0,16 s
75	0,23 s	0,32 s	0,20 s	0,24 s
100	0,63 s	0,65 s	0,43 s	0,40 s
150	2,24 s	2,23 s	1,17 s	0,87 s
200	mem !	5,27 s	2,84 s	1,69 s
250	mem !	mem !	5,41 s	3,05 s
PGI HPF	**1 nœud**	**2 nœuds**	**4 nœuds**	**8 nœuds**
50	2,83 s	4,63 s	4,39 s	4,14 s
75	8,85 s	6,23 s	5,79 s	4,66 s
100	21,38 s	13,48 s	12,36 s	9,31 s
150	88,70 s	46,18 s	42,73 s	29,70 s
200	mem !	118,82 s	123,67 s	64,24 s

Dans ce cas, l'avantage de l'ordonnancement linéaire décalé vient du fait qu'aucune fonction d'accès affine n'est générée. Ce n'est pas généralisable. Mais, comme l'ordonnancement linéaire décalé est plus général que l'ordonnancement linéaire (quand toutes les constantes sont nulles, l'ordonnancement linéaire décalé devient un ordonnancement linéaire), la probabilité d'avoir un ordonnancement « simple » est plus grande avec l'ordonnancement linéaire décalé. En effet, la partie linéaire la plus rapide qu'on peut avoir est un vecteur avec un 1 (ou -1) et les autres composantes nulles. Un autre avantage de ce genre de vecteurs est qu'ils donnent une matrice de projection qui projette le long d'un axe, ne générant pas de redistribution ni d'accès affines aux tableaux.

12. *stride* en anglais

PROG. **4.20** Code généré par Bouclettes avec l'ordonnancement linéaire décalé sans redistribution pour Mtest

```
!HPF$ TEMPLATE BCLT_0_template(n+3,n+6,n)
!HPF$ DISTRIBUTE BCLT_0_template(BLOCK,BLOCK,*)
!HPF$ ALIGN a(i1,i2,i3) WITH BCLT_0_template(i1,i2+5,i3)
!HPF$ ALIGN b(i1,i2,i3) WITH BCLT_0_template(i1+2,i2,i3)
!HPF$ ALIGN c(i1,i2,i3) WITH BCLT_0_template(i1+3,i2+6,i3)
!HPF$ ALIGN d(i1,i2,i3) WITH BCLT_0_template(i1,i2+5,i3)
      ....
      DO NT = -n+6, -n+8
        FORALL (KT = max(-n+6,NT-2):NT-2,J1 = 7:n-5,J2 = 2:n-3)
     &     a(J2,J1,-KT) = b(J2,J1-6,-KT-1)+d(J2-1,J1+3,-KT+1)
        FORALL (KT = max(-n+6,NT):NT,J1 = 7:n-5,J2 = 2:n-3)
     &     b(J2+1,J1-1,-KT) = c(J2+2,J1+5,-KT+2)
        FORALL (KT = max(-n+6,NT-3):NT-3,J1 = 7:n-5,J2 = 2:n-3)
     &     c(J2+3,J1-1,-KT-2) = a(J2,J1-2,-KT)
        FORALL (KT = max(-n+6,NT-2):NT-2,J1 = 7:n-5,J2 = 2:n-3)
     &     d(J2,J1-1,-KT) = a(J2,J1-1,-KT+6)
      END DO

      DO NT = -n+9, -4
        KT = NT-2
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     a(J2,J1,-KT) = b(J2,J1-6,-KT-1)+d(J2-1,J1+3,-KT+1)
        KT = NT
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     b(J2+1,J1-1,-KT) = c(J2+2,J1+5,-KT+2)
        KT = NT-3
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     c(J2+3,J1-1,-KT-2) = a(J2,J1-2,-KT)
        KT = NT-2
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     d(J2,J1-1,-KT) = a(J2,J1-1,-KT+6)
      END DO

      DO NT = -3, 0
        FORALL (KT = NT-2:min(-3,NT-2),J1 = 7:n-5,J2 = 2:n-3)
     &     a(J2,J1,-KT) = b(J2,J1-6,-KT-1)+d(J2-1,J1+3,-KT+1)
        FORALL (KT = NT:min(-3,NT),J1 = 7:n-5,J2 = 2:n-3)
     &     b(J2+1,J1-1,-KT) = c(J2+2,J1+5,-KT+2)
        FORALL (KT = NT-3:min(-3,NT-3),J1 = 7:n-5,J2 = 2:n-3)
     &     c(J2+3,J1-1,-KT-2) = a(J2,J1-2,-KT)
        FORALL (KT = NT-2:min(-3,NT-2),J1 = 7:n-5,J2 = 2:n-3)
     &     d(J2,J1-1,-KT) = a(J2,J1-1,-KT+6)
      END DO
```

Ordonnancement linéaire décalé et redistribution La redistribution générée ici ne fait que permuter deux indices (voir le programme 4.21).

PROG. **4.21** Redistribution pour l'ordonnancement linéaire décalé pour Mtest

```
FORALL (J0 = 1:n,J1 = 1:n,J2 = 1:n)
    ROT_1_a(n-J2+1,J1,J0) = a(J0,J1,J2)
    ROT_1_b(n-J2+1,J1,J0) = b(J0,J1,J2)
    ROT_1_c(n-J2+1,J1,J0) = c(J0,J1,J2)
    ROT_1_d(n-J2+1,J1,J0) = d(J0,J1,J2)
END FORALL
```

En fait, les boucles générées ne font que compliquer la tâche du compilateur (voir le programme 4.22).

PROG. **4.22** Boucles de calcul pour l'ordonnancement linéaire décalé pour Mtest

```
DO NT = -n+9, -4
  KT = NT-2
  FORALL (J1 = 7:n-5,J2 = 2:n-3)
&    ROT_1_a(n+KT+1,J1,J2) =
&        ROT_1_b(n+KT+2,J1-6,J2)+ROT_1_d(n+KT,J1+3,J2-1)
  KT = NT
  FORALL (J1 = 7:n-5,J2 = 2:n-3)
&    ROT_1_b(n+KT+1,J1-1,J2+1) = ROT_1_c(n+KT-1,J1+5,J2+2)
  KT = NT-3
  FORALL (J1 = 7:n-5,J2 = 2:n-3)
&    ROT_1_c(n+KT+3,J1-1,J2+3) = ROT_1_a(n+KT+1,J1-2,J2)
  KT = NT-2
  FORALL (J1 = 7:n-5,J2 = 2:n-3)
&    ROT_1_d(n+KT+1,J1-1,J2) = ROT_1_a(n+KT-5,J1-1,J2)
 END DO
```

ADAPTOR ne reconnaît pas qu'il peut utiliser des zones de recouvrement (une technique d'optimisation des communications). Il crée donc des copies entières des tableaux qui requièrent plus de copies de données. Les temps de calcul sont à peu près trois fois plus grands que dans le cas précédent. De plus, la redistribution elle-même nécessite du temps.

TAB. **4.7 -** Résultats pour l'ordonnancement linéaire décalé avec redistribution pour Mtest avec N= 100

ADAPTOR	1 nœud	2 nœuds	4 nœuds	8 nœuds
pré-copie	1,80 s	1,32 s	0,80 s	0,40 s
post-copie	1,45 s	1,56 s	1,08 s	0,59 s
calcul	3,04 s	1,59 s	1,11 s	0,47 s

Il faut noter que la redistribution est, dans ce cas, bien plus rapide que dans le cas de l'ordonnancement linéaire.

Nous voyons sur cet exemple que la supériorité théorique des ordonnancements linéaires décalés peut être vérifiée en pratique. Mais il faut aussi mentionner qu'il n'y a pas de règle générale ici. En effet, comme le compilateur peut optimiser mieux l'une ou l'autre des versions, la situation

contraire peut avoir lieu sur un autre exemple. Cependant, nous pensons que les ordonnancements linéaires décalés sont le plus souvent meilleurs que les linéaires.

4.6.3 Commentaires

Les résultats montrent les bénéfices de la parallélisation automatique de boucles. La technologie actuelle permet de tirer avantage de cette parallélisation dans des compilateurs HPF.

On a pu aussi identifier où les compilateurs HPF devaient être améliorés. Le problème principal est d'identifier la communication entre deux étapes de calcul et de trouver une formule close pour leur détermination. En particulier le support des fonctions affines d'accès aux tableaux pourrait améliorer spectaculairement le code généré par Bouclettes.

La redistribution optionnelle de Bouclettes simplifie la tâche du compilateur mais il reste quand même un surcoût important et d'autres problèmes surviennent comme la mémoire supplémentaire utilisée. De plus, dans le cadre d'un programme complet, des redistributions seraient nécessaires pour tous les tableaux alignés, sinon d'autres problèmes pourraient apparaître à d'autres endroits du programme parallèle.

4.7 Conclusion

Nous avons présenté dans ce chapitre un paralléliseur automatique de boucles, Bouclettes. Les différentes méthodes utilisées ont été présentées. Le choix du langage de sortie, High Performance Fortran, s'est révélé être un choix critique qui a guidé toute la dernière phase de la parallélisation : la génération du code. Simplifiant la gestion des communications, HPF a aussi eu des contraintes assez fortes sur la forme du code généré et n'a pas permis l'exploitation totale du parallélisme potentiel extrait par les premières phases de la parallélisation (analyse de dépendance, ordonnancement et allocation).

Nous avons ensuite testé la sortie de Bouclettes sur des compilateurs HPF et plus particulièrement ADAPTOR. Une première constatation importante est que le code produit est juste, dans le sens que le programme parallèle donne le même résultat que le programme séquentiel dont il est issu. Les tests montrent que le code généré peut être efficace dans certains cas, mais que le plus souvent, le fait que les fonctions d'accès aux tableaux deviennent des fonctions affines plutôt que des translations rend le code beaucoup plus difficile à optimiser pour le compilateur HPF. On peut espérer qu'avec les progrès des techniques de compilation, le code généré par Bouclettes sera de mieux en mieux optimisé.

Chapitre 5

Le pavage ultime ?

5.1 Introduction

Dans le chapitre précédent, nous avons vu que Bouclettes laisse le soin au compilateur HPF de faire le partitionnement des processeurs virtuels sur les processeurs physiques de la machine cible. Nous présentons ici une méthode destinée à augmenter la granularité du calcul et ainsi réduire le surcoût dû aux communications. Le *pavage par parallélépipèdes* s'applique dans le cadre des nids de boucles uniformes et parfaits. Nous passons en revue des approches existantes dans la littérature, ainsi que des critères d'optimisation utilisés pour déterminer un pavage « bon » ou « optimal ». Ensuite, nous expliquons pourquoi il faut introduire un nouveau critère d'optimisation dans un environnement extensible. Bien que notre critère soit plus complexe que ceux qui étaient utilisés précédemment, nous avons pu prouver un théorème d'optimalité et donner une méthode constructive pour construire un pavage « optimal ».

5.1.1 Pavage : motivation

Le *pavage par parallélépipèdes* (qu'on appellera simplement « pavage » dans la suite) est une technique de groupement de points de calcul qui augmente la granularité et ainsi diminue le temps de communication. Cette technique est restreinte aux nids de boucles parfaits avec des dépendances uniformes, comme définis par Banerjee dans [Ban88] (voir aussi les exemples qui suivent).

Le *pavage* a été introduit par Irigoin et Triolet comme un « partitionnement en super-nœuds » [IT88], et a été étudié depuis par plusieurs auteurs [SD90, SHS93, Ram92a, LW92, Ram92b, SS93] dans des contextes différents. Pour tirer un profit maximum des architectures des super-ordinateurs, il est nécessaire d'utiliser des transformations de programmes qui produisent des instructions vectorielles et des accès aux données locales. L'idée de regrouper plusieurs itérations de boucles pour augmenter la granularité, et ainsi obtenir une meilleure vectorisation et améliorer l'utilisation du cache, remonte à des transformations telles que le « découpage de boucles en bandes [1] » [Wol89], éventuellement combiné avec l'échange de boucles.

Cependant, considérons l'exemple suivant (programme 5.1) emprunté à Ramanujam et Sadayappan [RS90]. Dans cet exemple, le découpage en bandes n'est pas une transformation légale car il violerait la sémantique du code séquentiel. La méthode de l'hyperplan de Lamport [Lam74] (avec par exemple le vecteur d'ordonnancement $\pi = (1 \quad 1 \quad 1)$) conduirait à des instructions vectorielles, mais n'améliorerait pas la localité des données.

1. *strip-mining* en anglais

PROG. **5.1** Exemple de Ramanujam et Sadayappan

```
do i = 2, N
   do j = 2, N
      do k = 2, N − 1
         a(i, j, k) =a(i − 1, j, k) + a(i, j − 1, k)
                    +a(i, j, k − 1) + a(i − 1, j − 1, k + 1)
      enddo
   enddo
enddo
```

Dans le contexte de machines vectorielles avec une mémoire hiérarchique à deux niveaux, Irigoin et Triolet [IT88] ont introduit le pavage en parallélépipèdes comme une méthode bien plus puissante que le découpage en bandes ou le partitionnement rectangulaire [Pei86]. Ce pavage consiste à agréger des points de calcul élémentaires en *tuiles* (ou super-nœuds) définies comme des parallélépipèdes multi-dimensionels de telle sorte que chaque tuile est exécutée de façon atomique, sans synchronisation ni communication. Cette approche a des objectifs très proches de ceux de Schreiber et Dongarra [SD90] dont le but est de construire automatiquement des versions par blocs d'algorithmes élémentaires sous forme de boucles imbriquées.

Une autre motivation pour étudier le pavage est lié à la programmation des ordinateurs parallèles à mémoire répartie. Minimiser le surcoût dû aux communications est une condition *sine qua non* d'obtention de bonnes performances. Cependant, les possibilités de communication des ordinateurs parallèles restent inférieures à leurs possibilités de calcul. Même si des stratégies de routage sophistiquées sont utilisées par le matériel, le temps d'établissement de la communication reste très long comparé au temps de transmission d'une donnée élémentaire (c'est encore pire si on compare aux temps de calcul). Des échanges fréquents de petits messages doivent donc être remplacés par des envois moins fréquents de messages plus gros. Il est aussi très important de recouvrir dans le temps les communications et les calculs quand c'est possible. Le pavage est une technique qui permet d'atteindre ces buts [RS90] : les communications entre les tuiles sont regroupées en un message unique et des calculs à l'intérieur des tuiles peuvent être faits pendant que les données des bords sont échangées avec les tuiles voisines.

Nous revenons sur les avantages du pavage dans la section 5.2.

5.1.2 Pavage: définition

Considérons de nouveau l'exemple du programme 5.1 pour fixer le vocabulaire :

Cet exemple est un *nid de boucles parfait* de profondeur $n = 3$. Le *domaine de calcul* est un cube $Dom = \{(i, j), 2 \leq i \leq N, 2 \leq j \leq N, 2 \leq k \leq N - 1\}$. Nous ne nous intéressons pas aux calculs effectués par les instructions du corps du nid de boucles. Ce qui est intéressant ici, ce sont les quatre vecteurs de dépendance qui sont capturés par la *matrice de dépendance* D suivante :

$$D = (d_1, d_2, d_3, d_4) = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix}.$$

En général le domaine de calcul est un polyèdre convexe de $\mathbf{Z}^n$ où n est la profondeur, c'est-à-dire le nombre de boucles imbriquées et la matrice de dépendance D est une matrice $n \times m$ où m est le nombre de vecteurs de dépendance. Nous écrirons $Dom = \{p \in \mathbf{Z}^n, Ap \leq b\}$, où $p = (i_1, \cdots, i_n)$ est un point de l'espace d'itération appelé un nœud.

Une *tuile* (ou un super-nœud) est un ensemble de nœuds défini par une boîte parallélépipèdique de dimension n. Il y a deux moyens équivalents de définir une tuile :

En coupant par des hyperplans. Considérons n vecteurs linéairement indépendants h_i. Une tuile est une copie translatée de la tuile canonique définie par : un nœud appartient à la tuile si et seulement si

$$\forall i \in [1, n] \qquad 0 \leq h_i.p < \beta_i$$

Une bonne représentation des vecteurs h_i est celle de Dongarra et Schreiber [SD90] : « imaginez un couteau, aligné de telle sorte que h_i soit normal à sa face plate, coupant le domaine en tranches de même épaisseur ». L'épaisseur des tranches (β_i) peut être choisie soit en fixant $\beta_i = 1$ et en faisant varier les h_i proportionnellement (c'est le choix que nous faisons ici), soit en fixant la norme euclidienne des h_i et en faisant varier les β_i (comme dans [SD90]). Ainsi, dans la suite, nous posons $\beta_i = 1$ de telle sorte qu'une tuile est l'ensemble des nœuds p vérifiant pour tout i, $0 \leq h_i.p < 1$. Nous notons H la matrice dont les lignes sont les h_i :

$$H = \begin{pmatrix} h_1 \\ \vdots \\ h_n \end{pmatrix}$$

Par un changement de base. Considérons n vecteurs linéairement indépendants p_i qui définissent les bords d'une tuile. La tuile est alors l'ensemble des nœuds dont les coordonnées sont positives et strictement inférieures à 1 dans la base définie par les p_i. Si P dénote la matrice dont les colonnes sont les p_i, alors cette définition est équivalente à la précédente avec $P^{-1} = H$.

Paver est simplement recouvrir le domaine de calcul avec des copies translatées de la tuile canonique définie ci-dessus. Les contraintes classiques sur les tuiles sont les suivantes :

Les tuiles sont bornées. Pour des raisons d'extensibilité, nous voulons que le nombre de points dans une tuile soit borné par une constante indépendante de la taille du domaine. Si la matrice H n'était pas inversible, toute une « tranche » du domaine serait incluse dans une seule tuile. C'est pourquoi nous prenons H *inversible*.

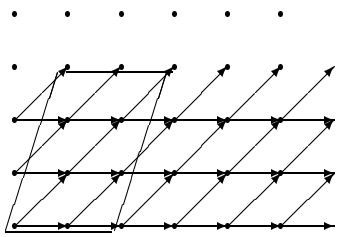
Les tuiles sont identiques par translation. Cette contrainte est imposée pour permettre la génération automatique du code : chaque tuile doit être l'image par une translation de toute autre sauf si elle coupe les frontières du domaine. Pour garantir cette contrainte, nous imposons que les n arêtes de la tuile aient des coordonnées entières, i.e. P *est une matrice entière*.

Les tuiles sont « atomiques ». Chaque tuile est une unité de calcul : tous les points de synchronisation sont au début et à la fin des tuiles. L'ordre sur les tuiles doit être compatible avec l'ordre sur les nœuds : on doit donc éviter que deux tuiles soient interdépendantes. Ceci est vérifié si tous les vecteurs de dépendance appartiennent au cône défini par les n arêtes de la tuile (cette condition n'est que suffisante en général). Cette condition peut être exprimée mathématiquement comme suit : toutes les composantes d'un vecteur de dépendance doivent être positives dans la base P, i.e. $Hd \geq 0$. En considérant tous les vecteurs de dépendance, on obtient : $HD \geq 0$.

PROG. **5.2** Exemple de nid pour illustrer la forme des tuiles

```
do i = 0, N
   do j = 0, N
      a(i,j) = f(a(i-1,j), a(i-1,j-1))
   enddo
enddo
```

FIG. **5.1 -** Graphe de dépendance et tuile possible pour l'exemple du programme 5.2



Illustrons cela sur un exemple très simple (voir le programme 5.2) : Il correspond au graphe représenté sur la figure 5.1 (pour plus de clarté, seul un sous-ensemble des vecteurs de dépendance sont représentés).

La matrice de dépendances est $D = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$. La tuile correspondant $P = \left(\begin{smallmatrix} 2 & 1 \\ 0 & 3 \end{smallmatrix}\right)$ est dessinée sur la figure 5.1 (ses colonnes sont les arêtes de la tuile). La matrice H correspondant est $H = \frac{1}{6}\left(\begin{smallmatrix} 3 & -1 \\ 0 & 2 \end{smallmatrix}\right)$. Ainsi $HD = \frac{1}{6}\left(\begin{smallmatrix} 3 & 2 \\ 0 & 2 \end{smallmatrix}\right)$ et la condition $HD \geq 0$ est vérifiée.

5.1.3 Organisation du chapitre

Quelles sont la taille et la forme « optimales » d'une tuile? Définir l'optimalité n'est pas facile et plusieurs auteurs ont proposé des critères contradictoires. Minimiser une quantité liée au rapport communications/calcul semble le plus naturel, mais quelle quantité?

Nous discutons les critères d'optimalité pour le pavage dans la section 5.2. Tout d'abord (dans la section 5.2.1), deux notions importantes sont introduites : le volume de communication et le volume de calcul d'une tuile. Ensuite (dans la section 5.2.2), en passant en revue trois approches connues (par Irigoin et Triolet [IT88], Schreiber et Dongarra [SD90], et Ramanujam et Sadayappan [RS90], respectivement) nous comparons les critères d'optimisation utilisés dans la littérature. Cette étude nous conduit à proposer un nouveau critère lié au rapport communications/calcul mais qui ne dépend que de la forme de la tuile, et pas de sa taille. Minimiser un tel critère semble être le but le plus utile pour plusieurs classes d'architectures cibles.

Nous montrons comment construire le pavage optimal dans la section 5.3. La preuve est simple quand la matrice de dépendances D est carrée et inversible, mais devient plus compliquée quand D est une matrice $n \times m$ de rang n, où $m > n$.[2] Nous illustrons la construction par plusieurs exemples pour une meilleure compréhension.

Enfin, nous concluons par quelques remarques dans la section 5.4.

2. Le cas où D n'est pas de rang plein peut être facilement capturé par une projection sur un espace de dimension inférieure, voir [Iri87]

5.2 Critère pour un pavage « optimal »

5.2.1 Que voulons-nous optimiser ?

L'impact du choix d'un pavage sur le code résultant est lié à deux paramètres importants : le *volume de calcul* et le *volume de communication*. Soit T une tuile définie par une matrice P ou H comme précédemment, avec $P = H^{-1}$, P entière et $HD \geq 0$. Soit $h_1, ..., h_n$ les vecteurs lignes de H et $p_1, ..., p_n$ les vecteurs colonnes de P.

Volume de calcul $V_{calc}(T)$ de la tuile : c'est le nombre de nœuds dans une tuile et il représente le nombre de calculs effectués par chaque processeur (contrainte d'atomicité). Nous avons

$$V_{calc}(T) = |\det P| = \frac{1}{|\det H|}$$

Cette expression est le volume de calcul exact. Si chaque sommet de la tuile est un point entier de $\mathbf{Z}^n$ alors P est une matrice entière et $|\det P|$ est le nombre de points entiers dans la tuile.

Volume de communication $V_{com}(T)$ de la tuile : c'est le nombre de vecteurs de dépendance allant d'un nœud interne à la tuile vers un nœud externe à la tuile. Ce nombre représente la quantité de données qui sera transférée depuis et jusqu'à chaque processeur à chaque synchronisation. Il peut être approché comme suit : si d est un vecteur de dépendance, alors $|\det(d, p_2, ..., p_n)|$ représente le volume du parallélépipède soutenu par $d, p_2, ..., p_n$. C'est une approximation du nombre de nœuds de la tuile dont le vecteur d sort de la face soutenue par $p_2, ..., p_n$ (ce n'est qu'une approximation comme le montre la figure 5.2). De plus,

$$|\det(d, p_2, ..., p_n)| = |(p_2 \wedge ... \wedge p_n).d| = |(\det P)h_1.d| = |\det P|(h_1.d)$$

(car $HD \geq 0$). Le volume de communication induit par le vecteur d à travers toutes les faces est ainsi à peu près égal à $|\det P|(\sum_{i=1}^{n} h_i.d)$. Considérons maintenant chaque vecteur de dépendance, nous obtenons que le volume de communication induit par toute la tuile est approché par :

$$V_{com}(T) = \frac{1}{|\det H|} \sum_{i=1}^{n} \sum_{j=1}^{m} (h_i.d_j) = \frac{1}{|\det H|} \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{n} h_{i,k} d_{k,j} \tag{5.1}$$

Dans la suite, nous identifions cette expression avec le volume de communication réel (remarquons que l'équation 5.1 est l'expression utilisées dans [RS90]).

Il y a plusieurs raisons pour fixer les valeurs de ces deux paramètres V_{calc} et V_{com} :

- V_{calc} est quasi-proportionnel à la quantité de mémoire locale utilisée par chaque processeur, il ne doit donc pas dépasser une certaine valeur.
- Quand V_{calc} augmente, on perd du parallélisme potentiel (les calculs sont séquentiels au sein d'une même tuile).
- D'autre part, quand V_{calc} augmente, il y a moins de pas de synchronisation pendant l'exécution parallèle.
- V_{com} ne doit pas être trop petit à cause du temps perdu lors de l'initiation des communications.

FIG. **5.2 -** Erreur sur le volume de communication

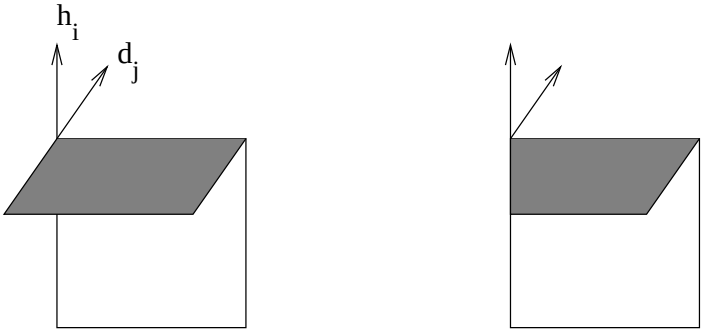
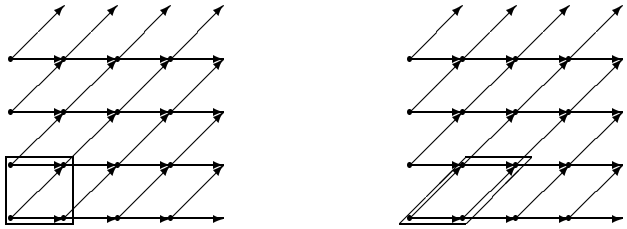


FIG. **5.3 -** Deux formes de tuiles de même volume de calcul avec des volumes de communication différents



– La valeur de V_{com} doit être ajustée avec celle de V_{calc} afin d'essayer de recouvrir un maximum de communications par des calculs.

Les contraintes sur V_{calc} sont dépendantes de la machine, et nous voudrions garder cette quantité comme paramètre lors de la construction des tuiles. D'autre part, on ne sait pas vraiment fixer V_{com} et V_{calc} indépendamment. Nous allons donc essayer de minimiser V_{com} pour une valeur fixée de V_{calc}. Ceci est une approximation valide car les communications sont souvent le facteur limitant de la vitesse d'exécution des programmes parallèles. Si le volume est fixé, c'est la *forme* de la tuile qui influencera le volume de communication.

Pour l'exemple du programme 5.2, deux formes de tuiles sont proposées sur la figure 5.3, avec le même volume (4 points dans chaque tuile). La tuile de gauche donne 5 communications alors que celle de droite n'en donne que 4.

Remarquons comment V_{calc} and V_{com} varient quand on transforme la tuile par une homothétie de facteur λ :

$$V_{calc}(\lambda T) = \lambda^n V_{calc}(T) \qquad V_{com}(\lambda T) = \lambda^{n-1} V_{com}(T).$$

Ainsi,

$$\lim_{\lambda \to 0} V_{com} = 0 \qquad \lim_{\lambda \to +\infty} \frac{V_{com}}{V_{calc}} = 0.$$

Donc, minimiser le nombre de communications et minimiser le volume de communication par rapport au volume de calcul sont des optimisations contradictoires.

5.2.2 Travaux liés

Nous commencerons par présenter le papier *Supernode Partitioning* d'Irigoin et Triolet [IT88], l'un des tous premiers papiers à parler de pavage. Nous examinerons ensuite *Automatic Blocking*

of Nested Loops de Schreiber et Dongarra [SD90], où le pavage est vu comme une technique de transformation de boucles. Enfin, nous présenterons le travail de Ramanujam et Sadayappan à travers leur papier *Tiling Multidimensional Iteration Spaces for Multicomputers* [RS90].

Supernode Partitioning

C'est dans ce papier qu'est défini pour la première fois avec précision le problème du pavage (par parallélépipèdes). La plupart de nos notations et hypothèses viennent de cette étude. La notion de *super-nœud* (ou tuile) y est introduite, et les contraintes que doivent satisfaire les tuiles y sont discutées. Irigoin et Triolet montrent l'importance de considérer les tuiles atomiques, identiques par translation, bornées et qui pavent le domaine de calcul.

Ils étudient pour commencer une technique appelée « partitionnement par hyperplan », qui consiste en la partition du domaine de calcul avec des hyperplans parallèles (tous normaux à un vecteur h) et régulièrement espacés. Ensuite, ils généralisent aux partitionnements définis grâce à plusieurs vecteurs h_i. Ils introduisent la matrice H dont les lignes sont les h_i et montrent comment les contraintes sur les tuiles se transposent sur H.

Quand H est carrée et inversible, ils montrent que cette technique de partitionnement par hyperplans n'est autre qu'une méthode de *regroupement défini par une base* [3] définie par la matrice $P = H^{-1}$.

Le problème du pavage étant posé, Irigoin et Triolet montrent comment construire une matrice H valide et comment les super-nœuds (et les nœuds à l'intérieur des super-nœuds) peuvent être ordonnancés.

Pour finir, Irigoin et Triolet remarquent que le choix de la taille et de la forme des super-nœuds (et ainsi le choix de H) dépend de plusieurs objectifs :

- la génération d'instructions vectorielles avec des vecteurs de longueur constante,
- l'équilibrage de charge,
- la minimisation du coût de communication.

Cependant, ils ne proposent pas de manière automatique de construction de super-nœuds optimisant l'un de ces objectifs.

Automatic Blocking of Nested Loops

Le sujet de ce papier est de dériver automatiquement des *programmes par blocs* à partir de programmes ordinaires. La technique utilisée est en fait très proche de celle de *Supernode Partitioning*.

Les tuiles sont définies par une matrice H et un vecteur β comme expliqué section 5.1.2. Chaque ligne de H est normalisée pour avoir une norme Euclidienne unitaire. Un nœud p appartient à une tuile si et seulement si :

$$\forall i \in [1, n], 0 \leq h_i.p < \beta_i$$

Ainsi, la forme (la matrice H) et la taille (le vecteur $b = (\beta_1, ..., \beta_n)$) d'une tuile sont étudiés séparément.

Dongarra et Schreiber remarquent que le volume de la tuile mesure le nombre de points d'itérations qu'elle contient, alors que l'aire de sa surface est liée à la quantité de communications nécessaires pour échanger des données avec les tuiles voisines. Leur but est d'étudier comment le choix de H et β détermine le rapport volume/surface ρ des tuiles induites et comment il peut être

3. *basis clustering* en anglais

maximisé soumis à une limite μ (un paramètre dépendant de la machine). Leur approche est une heuristique en deux étapes :

- D'abord, étudier le cas $b = (\beta, ..., \beta)$ en choisissant H de façon adéquate.
- Ensuite, ajuster b en fonction d'un critère lié aux contraintes de taille de mémoire locale.

Ils montrent que dans le premier cas, le meilleur choix est une matrice H de déterminant maximum, satisfaisant aux contraintes habituelles ($HD \geq 0$) et en plus à la contrainte de normalisation (chaque ligne a une norme euclidienne de un). Ils proposent un algorithme heuristique pour résoudre ce problème. Ensuite, ils fixent le facteur b en dérivant la meilleure valeur de β_n quand β_1, ..., β_{n-1} sont supposés fixés.

Cette approche essaye de fixer les deux paramètres V_{com} et V_{calc} séparément. La fonction de coût utilisée (le rapport ρ) n'est pas indépendante de V_{calc}, c'est pourquoi le problème est si difficile à résoudre. Aucun résultat d'optimalité n'est donné. La méthode proposée par Dongarra et Schreiber est un algorithme heuristique.

Tiling Multidimensional Iteration Spaces for Multicomputers

Ramanujam et Sadayappan [RS90, Ram92b] proposent de minimiser le volume de communication V_{com}. L'espace de recherche est limité aux matrices triangulaires inférieures avec diagonale unitaire. L'argument pour considérer cette restriction est qu'il existe toujours une telle matrice H, vérifiant $HD \geq 0$ dans le cas de nids de boucles uniformes. Un schéma de preuve de ce résultat est donné. Les tuiles générées par cette méthode auront un volume de 1 nœud et devront être homothétisées pour augmenter la granularité.

Le problème est posé sous la forme d'un problème de programmation linéaire, à savoir trouver une transformation H telle que :

$$\sum_{k=1}^{n} H_{i,k} D_{k,j} \geq 0, \forall\ i, j,\ 1 \leq i \leq n,\ 1 \leq j \leq m,$$

avec les contraintes supplémentaires :

$$H_{i,k} = 0 \text{ si } k > i \text{ et } H_{i,i} = 1,$$

qui minimise le volume de communication.

Comme dans *Supernode Partitioning*, les tuiles sont ordonnancées par le vecteur

$$\pi = (1 \quad 1 \quad ... \quad 1)$$

qui est toujours valide si les tuiles sont suffisamment grandes. Le graphe ordonnancé est alors alloué par une projection parallèle à un axe pour internaliser les communications selon cette direction.

Enfin, les auteurs discutent rapidement le choix de la taille de la tuile en fonction de plusieurs paramètres liés à la machine.

La principale faiblesse de ce papier (et c'est un problème majeur) est que les auteurs restreignent leur choix aux matrices unimodulaires triangulaires inférieures (pour éviter la contrainte non linéaire sur le déterminant). Ils peuvent ainsi passer à côté de la matrice optimale (exactement ce que nous voulons éviter dans la suite !).

Prenons l'exemple du programme 5.3. La matrice de dépendances est

PROG. **5.3** Exemple pour illustrer la méthode de Ramanujam et Sadayappan

```
do i = 1, N
   do j = 1, N
      do k = 1, N
         a(i,j,k) = f(a(i-1,j-1,k), a(i,j-1,k-1), a(i-1,j,k-1))
      enddo
   enddo
enddo
```

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

Le problème est de trouver une matrice triangulaire inférieure

$$H = \begin{pmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & c & 1 \end{pmatrix},$$

minimisant le volume de communication et vérifiant $HD \geq 0$. Cela conduit à minimiser

$$6 + 2a + 2b + 2c \text{ vérifiant } HD = \begin{pmatrix} 1 & 0 & 1 \\ a+1 & 1 & a \\ b+c & c+1 & b+1 \end{pmatrix} \geq 0$$

La solution optimale est pour $a = b = c = 0$ avec $V_{com} = 6$. La tuile résultante T est telle que $V_{calc}(T) = 1$ et $V_{com}(T) = 6$. Si nous appliquons une homothétie de rapport λ à T de façon à obtenir un volume de λ^3 nœuds, nous obtenons $V_{calc}(\lambda) = \lambda^3$ et $V_{com}(\lambda) = 6\lambda^2$. Il aurait été meilleur de choisir la matrice :

$$H' = D^{-1} = \frac{1}{2}\begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & -1 \\ -1 & 1 & 1 \end{pmatrix}.$$

Cela donne une famille de tuiles homothétiques $T'(\lambda)$ telles que $V'_{calc}(\lambda) = 2\lambda^3$ et $V'_{com}(\lambda) = 6\lambda^2$, qui a un meilleur rapport calcul/communications.

En général, minimiser V_{com} conduira toujours à la plus petite tuile : ainsi ce simple critère ne répond pas à nos objectifs.

5.2.3 Un critère extensible

Dans cette section, nous discutons les buts de l'optimisation afin de définir le « pavage optimal ». Nous expliquons pourquoi les critères précédemment introduits dans la littérature ne sont pas totalement satisfaisants, et nous en proposons un nouveau dont la principale caractéristique est d'être extensible : notre critère, lié au rapport calcul/communication d'une tuile, ne dépend que de sa forme, pas de sa taille.

Il semble que minimiser V_{com} ou $\frac{V_{com}}{V_{calc}}$ conduise à des problèmes très difficiles qui ne peuvent pas être résolus exactement. Une procédure « idéale » serait la suivante :

– Pour chaque valeur fixée de V_{calc}, trouver H qui minimise la valeur de V_{com}.

– Fixer la valeur de V_{calc} en relation avec les contraintes de mémoire locales.

Considérons maintenant l'expression suivante :

$$V(H) = \frac{\sum_{i=1}^{n}\sum_{i=1}^{m}\sum_{k=1}^{n} h_{i,k}d_{k,j}}{|\det(H)|^{\frac{1}{n}}} \tag{5.2}$$

Homothétiser H d'un facteur λ ($\lambda \geq 0$) donne :

$$V(\lambda H) = \lambda \left|\frac{\det(H)}{\det(\lambda H)}\right|^{\frac{1}{n}} V(H) = V(H)$$

Nous avons ainsi l'important lemme suivant :

Lemme 1 *Minimiser $V(H)$ et homothétiser la solution pour obtenir une valeur fixée du volume de calcul $V_{calc} = V_0$ minimise aussi V_{com} pour la valeur V_0 de V_{calc}.*

Preuve En effet, considérons $H_1(V_0)$ et H_2 telles que :

$$V_{com}(H_1) = \min_{V_{calc}(H)=V_0}(V_{com}(H)), H \text{ inversible}, HD \geq 0$$

$$V(H_2) = \min V(H), H \text{ inversible}, HD \geq 0$$

Soit

$$H_2' = \sqrt[n]{\frac{1}{V_0|\det(H_2)|}} H_2.$$

$|\det(H_2')| = 1/V_0$, i.e. $V_{calc}(H_2') = V_0$, ainsi

$$V_{com}(H_2') \geq V_{com}(H_1).$$

Comme H_2' et H_1 ont le même déterminant (en valeur absolue) et que $V(H) = |\det(H)|^{\frac{n-1}{n}} V_{com}(H)$, nous avons

$$V(H_2') \geq V(H_1).$$

Mais $V(H_2') = V(H_2)$. Ainsi,

$$V(H_2') = V(H_1) = V(H_2)$$

(par définition de H_2).

Cela signifie que, si la valeur minimale de $V(H)$ est obtenue pour une matrice H_0 donnée, alors nous avons la certitude que

$$H_1 = \sqrt[n]{\frac{1}{V_0|\det(H_0)|}} H_0$$

donnera la valeur minimale de V_{com} parmi toutes les matrices valides qui définissent une tuile de volume V_0. Ainsi nous pouvons appliquer la procédure « idéale » mentionnée précédemment et V est la bonne fonction à minimiser. La seule restriction est qu'en homothétisant la matrice H_0, nous oublions la contrainte d'intégralité sur P. Ainsi, le résultat ne sera juste que pour les valeurs de V_0 pour lesquelles $V_0|\det(H_0)|$ est un entier, i.e. pour les tuiles dont le volume est un multiple du volume d'une tuile définie par H_0. □

5.3 Minimiser $V(H)$ est un problème combinatoire

Comme exposé dans la section précédente, le problème d'optimisation que nous allons étudier est le suivant :

minimiser :

$$V(H) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{|\det H|^{\frac{1}{n}}}$$

sous les contraintes :

$$\det H \neq 0$$
$$HD \geq 0$$

D est une matrice $n \times m$. Nous supposons que $m \geq n$ et que D est de rang plein n. Notons V_{min} ce minimum.

L'espace de recherche est infini, inclus dans l'ensemble $\Gamma = \{H|HD \geq 0\}$. On pourrait penser qu'il n'y a pas d'espoir de trouver la matrice H optimale par un algorithme. Cependant, nous allons prouver que ce problème d'optimisation est en fait un problème combinatoire dont la solution est un élément de certaines faces du cône Γ.

Étudions d'abord la structure de Γ.

5.3.1 Le cône $\Gamma = \{H|HD \geq 0\}$

Il y a une dualité entre les cônes avec un nombre fini de générateurs et les cônes polyédraux, comme entre les cônes et les cônes polaires. Cela est parfaitement expliqué dans [Sch86, Corollary 7.1a], par exemple.

Soit C le cône généré par les vecteurs colonnes de $D : d_1, ..., d_m$. C est aussi un cône polyédral : $C = \{x|b_1^T x \geq 0, ..., b_t^T x \geq 0\}$ où les b_i sont construits comme suit :

1. Sélectionner $n-1$ vecteurs linéairement indépendants parmi les vecteurs colonnes de D, notons les $x_1, ..., x_{n-1}$.
2. Calculer c, un vecteur normal à l'hyperplan décrit par les $x_1, ..., x_{n-1}$, par exemple $x_1 \wedge ... \wedge x_{n-1}$.
3. Si, pour tout i, $c^T d_i \geq 0$, alors sélectionner c comme un b_i. Si, pour tout i, $c^T d_i \leq 0$, alors sélectionner $-c$ comme un b_i. Sinon, l'ignorer, l'hyperplan décrit par les $x_1, ..., x_{n-1}$ n'est pas une vraie face de C.

Maintenant, considérons le cône $C^* = \{z|z^T x \geq 0 \text{ pour tout } x \in C\}$ qui est un cône polyédral : $C^* = \{z|z^T d_1 \geq 0, ..., z^T d_m \geq 0\}$. Avec les mêmes arguments que dans la preuve du Corollaire 7.1a dans [Sch86], C^* est aussi un cône généré, le cône généré par $b_1, ..., b_t$.

Lemme 2 *Quand les vecteurs colonnes de D sont lexicographiquement positifs, le cône C^* est de dimension pleine.*

Preuve Soit D_i l'ensemble des vecteurs colonnes de D dont la première coordonnée non nulle est celle d'indice i. Si les vecteurs colonnes de D sont lexicographiquement positifs (ce qui est le cas pour les nids de boucles uniformes), alors pour tout $x \in D_i$, $x_1 = ... = x_{i-1} = 0$ et $x_i > 0$. Construisons maintenant les vecteurs c^j, $1 \leq j \leq n$ comme suit :

- $c^j_{j+1} = ... = c^j_n = 0$. Ainsi, pour tout $i > j$, pour tout $x \in D_i$, $c^{jT} x = 0$.

- $c_j^j = 1$. Ainsi, pour tout $x \in D_j$, $c^{jT}x > 0$.
- Construisons par induction décroissante c_i^j pour $1 \leq i < j$. Supposons que c_{i+1}^j, ..., c_n^j sont construits et que pour tout $k > i$, pour tout $x \in D_k$, $c^{jT}x \geq 0$. Si D_i est vide, soit $c_i^j = 0$, soit $c_i^j = -\min\{\frac{1}{x_i}\sum_{k=i+1}^n c_k^j x_k | x \in D_i\}$. Alors, pour tout $x \in D_i$, $c^{jT}x = x_i(c_i^j + \frac{1}{x_i}\sum_{k=i+1}^n c_k^j x_k) \geq 0$.

Ainsi, pour tout $1 \leq j \leq n$, c^j est dans C^* et les c^j sont linéairement independents. Cette preuve a déjà été esquissée dans [RS90]. □

Nous sommes maintenant prêts à étudier Γ. Notons que le lemme 2 implique qu'il existe toujours une matrice inversible dans Γ dont les lignes sont n vecteurs linéairement independents de C^*. Ainsi $V_{min} < +\infty$.

Γ est manifestement un cône polyédral dans un espace de dimension n^2. On pourrait construire des générateurs de Γ comme nous l'avons fait pour C^*, mais en considérant H comme un vecteur de longueur n^2. Cependant, pour garder la structure de matrices, il est bien plus simple de visualiser Γ comme suit : étant donné un H dans Γ, chaque vecteur ligne de H est dans C^*. Ainsi, c'est une combinaison linéaire positive des b_i et Γ est généré par les nt matrices dont les composants sont nuls sauf une ligne égale à un b_i.

Voyons un exemple simple avec

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

C^* est un cône dans un espace de dimension 3, généré par $b_1 = (1,-1,1)$, $b_2 = (1,1,-1)$ et $b_3 = (-1,1,1)$ tandis que Γ est un cône dans un espace de dimension 9 généré par les matrices :

$$\begin{pmatrix} 1 & -1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} -1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ -1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & -1 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & -1 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 1 & 1 \end{pmatrix}$$

5.3.2 Cas particulier : D est une matrice carrée

Quand D est une matrice $n \times n$, les choses sont plus simples. C^* est le cône généré par les vecteurs lignes de D^{-1} et la valeur minimale peut être immédiatement identifiée en utilisant le lemme suivant :

Lemme 3 *Pour toute matrice carrée A, à éléments positifs :*

$$|\det A|^{\frac{1}{n}} \leq \frac{1}{n}\sum_{i=1}^n \sum_{j=1}^n A_{i,j}$$

Preuve Le résultat est évident si A n'est pas inversible ($\det A = 0$).

– Supposons pour commencer que A est une matrice symétrique, définie positive. Alors, les valeurs propres de A, $\lambda_1, ..., \lambda_n$ sont des nombres réels positifs. nous avons $|\det A| = \det A = \prod_{i=1}^n \lambda_i$ et $\operatorname{trace} A = \sum_{i=1}^n \lambda_i$. De plus, comme la fonction logarithmique est concave :

$$\frac{1}{n}\sum_{i=1}^{n}\log\lambda_i \leq \log(\frac{1}{n}\sum_{i=1}^{n}\lambda_i) \Leftrightarrow \log(\prod_{i=1}^{n}\lambda_i)^{\frac{1}{n}} \leq \log(\frac{1}{n}\sum_{i=1}^{n}\lambda_i)$$
$$\Leftrightarrow (\prod_{i=1}^{n}\lambda_i)^{\frac{1}{n}} \leq \frac{1}{n}\sum_{i=1}^{n}\lambda_i$$

Ainsi, $|\det A|^{\frac{1}{n}} \leq \frac{1}{n}\operatorname{trace} A$.

– Si A est inversible, alors il existe Q orthogonale et S symétrique définie positive telle que $QA = S$. Notons $Q_{i.}$ la i^{e} ligne de Q et $A_{.j}$ la j^{e} colonne de A. $\langle x|y\rangle$ est le produit scalaire et $\|.\|$ la norme qui lui est associée. Alors :

$$\operatorname{trace} QA = \sum_{i=1}^{n}\sum_{j=1}^{n} Q_{i,j}A_{j,i} = \sum_{i=1}^{n}\langle Q_{i.}|A_{.i}\rangle \leq \sum_{i=1}^{n}\underbrace{\|Q_{i.}\|}_{1}\|A_{.i}\| = \sum_{i=1}^{n}\|A_{.i}\|$$

Ainsi:

$$|\det A|^{\frac{1}{n}} = (|detQ^{-1}S|^{\frac{1}{n}} = |\det S|^{\frac{1}{n}} \leq \frac{1}{n}\operatorname{trace} S = \frac{1}{n}\operatorname{trace} QA$$
$$\leq \frac{1}{n}\sum_{i=1}^{n}\|A_{.i}\| = \frac{1}{n}\sum_{i=1}^{n}\sqrt{\sum_{j=1}^{n}(A_{j,i})^2} \leq \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}|A_{j,i}|$$

Finalement, si pour tout $1 \leq i,j \leq n$, $A_{i,j} \geq 0$, nous obtenons le résultat en permuttant i et j: $|\det A|^{\frac{1}{n}} \leq \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n} A_{i,j}$. □

Théorème 1 *Si D est une matrice carrée inversible, alors $V_{min} = n|\det D|^{\frac{1}{n}}$ et ce minimum est atteint avec $H_0 = D^{-1}$.*

Preuve Pour toute matrice inversible H dans Γ:

$$V(H) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{|\det H|^{\frac{1}{n}}} = (n|\det D|^{\frac{1}{n}})\frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{n|\det HD|^{\frac{1}{n}}}$$
$$V(H) = V(D^{-1})\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{n|\det HD|^{\frac{1}{n}}}\right) \geq V(D^{-1})$$

d'après le lemme 3 appliqué à $A = HD$. □

Dans ce cas particulier d'une matrice D, $n \times n$, le minimum est ainsi atteint pour une matrice dont les lignes sont n vecteurs linéairement indépendants de C^*. C'est en fait aussi vrai dans le cas général comme c'est démontré dans la section suivante.

Retour sur l'exemple du programme 5.3 La matrice de dépendances est

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

Le théorème précédent montre que la matrice optimale est $H = D^{-1}$ (voir section 5.2.2).

5.3.3 Cas général : D est une matrice $n \times m$ de rang plein

Faisons d'abord deux remarques :

- V a été choisi pour avoir une bonne propriété homothétique : si $H \in \Gamma$ alors pour tout $\lambda \geq 0$, $\lambda H \in \Gamma$ et $V(\lambda H) = V(H)$.
- Si $H \in \Gamma$ et $\sum_{i=1}^{n} \sum_{j=1}^{m} (HD)_{i,j} = 0$ alors $HD = 0$ et ainsi $H = 0$ car les lignes de D sont de rang plein.

Nous avons besoin du lemme suivant :

Lemme 4 *Soit $V_{min} = \min\{V(H) | H \in \Gamma, \det H \neq 0\}$. Alors, il existe $H_0 \in \Gamma$, H_0 inversible, telle que $V(H_0) = V_{min}$. C'est-à-dire que le minimum est atteint.*

Preuve Pour une matrice H, soit $\|H\|_\infty = \max_{1 \leq i,j \leq n} |H_{i,j}|$. Par définition de V_{min}, il y a une suite de matrices inversibles F_n dans Γ telles que $V(F_n) \to V_{min}$. Soit $G_n = \frac{1}{\|F_n\|_\infty} F_n$. Pour tout n, nous avons par construction $\|G_n\|_\infty = 1$ et $V(G_n) = V(F_n) \to V_{min}$.

G_n est une suite dans $\{M | MD \geq 0, \|M\|_\infty = 1\}$ qui est un espace compact. Ainsi, il y a une sous-suite H_n de G_n telle que $H_n \to H_0$ avec $H_0 D \geq 0$, $\|H_0\|_\infty = 1$ et $V(H_n) \to V_{min}$. De plus,

$$\sum_{i=1}^{n} \sum_{j=1}^{m} (H_n D)_{i,j} \to \sum_{i=1}^{n} \sum_{j=1}^{m} (H_0 D)_{i,j}$$

et $\det H_n \to \det H_0$ par continuité. Comme $H_0 \neq 0$,

$$\sum_{i=1}^{n} \sum_{j=1}^{m} (H_0 D)_{i,j} \neq 0.$$

Ainsi, si $\det H_0 = 0$ alors $V(H_n) \to +\infty$, ce qui est impossible. Ainsi, H_0 est inversible et

$$V(H_0) = V_{min}.$$

□

Donnons maintenant quelques propriétés plus précises de la fonction V.

Considérons une matrice inversible H dans Γ telle qu'une ligne de H est une combinaison linéaire positive d'au moins deux b_i (les générateurs de C^* construits précédemment). Sans perte de généralité :

$$H = \lambda_1 \begin{pmatrix} b_1 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} b_2 \\ 0 \end{pmatrix} + G \qquad \text{avec } G \in \Gamma$$

Pour simplifier le reste de la preuve, soit :

$$\alpha_1 = \sum_{j=1}^{m} (b_1 D)_j, \quad \alpha_2 = \sum_{j=1}^{m} (b_2 D)_j, \quad K = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} (HD)_{i,j},$$

$$M = \alpha_2 \begin{pmatrix} b_1 \\ 0 \end{pmatrix} - \alpha_1 \begin{pmatrix} b_2 \\ 0 \end{pmatrix}, \text{ et } h = H_{\cdot 1}^{-1} \text{ (première colonne de } H^{-1}).$$

Remarquons que b_1 ne peut pas être orthogonal à tous les vecteurs colonnes de D (D est de rang n) ainsi $\alpha_1 \neq 0$ (c'est la même chose pour α_2).

Alors, nous avons le lemme suivant :

Lemme 5

- *Si $\alpha_i \neq K\langle b_i | h \rangle$ pour $i = 1$ ou $i = 2$ alors $V(H)$ n'est pas minimum.*
- *Sinon, l'application $\lambda \to V(H + \lambda M)$ est constante.*

Preuve Supposons que $\alpha_1 \neq K\langle b_1 | h \rangle$. Étudions V dans un voisinage de H. Soit ε, $-1 < \varepsilon < 1$:

$$\begin{aligned} V\left(H + \varepsilon \begin{pmatrix} b_1 \\ 0 \end{pmatrix}\right) &= \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} (HD)_{i,j} + \varepsilon \sum_{j=1}^{m} (b_1 D)_j}{\left| \det \left(H + \varepsilon \begin{pmatrix} b_1 \\ 0 \end{pmatrix} \right) \right|^{\frac{1}{n}}} \\ &= \frac{nK + \varepsilon \alpha_1}{|\det H|^{\frac{1}{n}} \left| \det \left(I_d + \varepsilon \underbrace{\begin{pmatrix} b_1 \\ 0 \end{pmatrix} H^{-1}}_{A} \right) \right|^{\frac{1}{n}}} \end{aligned}$$

Il existe une matrice inversible P telle que $A = PTP^{-1}$ avec T triangulaire. Comme A est une matrice de rang 1, tout les éléments diagonaux de T sont nuls sauf un (qui est égal à $\operatorname{trace} T = \operatorname{trace} A = \langle b_1 | h \rangle$). Ainsi :

$$\det(I_d + \varepsilon A) = \det(P^{-1}(I_d + \varepsilon A)P) = \det(I_d + \varepsilon T) = 1 + \varepsilon \langle b_1 | h \rangle$$

et

$$\begin{aligned} V\left(H + \varepsilon \begin{pmatrix} b_1 \\ 0 \end{pmatrix}\right) &= \frac{nK + \varepsilon \alpha_1}{|\det H|^{\frac{1}{n}} |1 + \varepsilon \langle b_1 | h \rangle|^{\frac{1}{n}}} \\ &= \frac{(nK + \varepsilon \alpha_1)(1 - \frac{1}{n} \varepsilon \langle b_1 | h \rangle + o(\varepsilon))}{|\det H|^{\frac{1}{n}}} \\ &= \frac{nK + \varepsilon(\alpha_1 - K\langle b_1 | h \rangle) + o(\varepsilon))}{|\det H|^{\frac{1}{n}}} \\ &= V(H) + \varepsilon \left(\frac{\alpha_1 - K\langle b_1 | h \rangle}{|\det H|^{\frac{1}{n}}} \right) + o(\varepsilon) \end{aligned}$$

Si $\alpha_1 - K\langle b_1 | h \rangle > 0$, alors avec $\varepsilon < 0$ suffisamment petit, on peut trouver un élément H_ε de Γ pour lequel $V(H) > V(H_\varepsilon)$. Si $\alpha_1 - K\langle b_1 | h \rangle < 0$, prendre $\varepsilon > 0$. Ainsi, $V(H)$ n'est pas minimum.

Supposons maintenant que $\alpha_1 = K\langle b_1|h\rangle$ et $\alpha_2 = K\langle b_2|h\rangle$. Alors :

$$V(H+\lambda M) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j} + \lambda(\alpha_2\sum_{j=1}^{m}(b_1D)_j - \alpha_1\sum_{j=1}^{m}(b_2D)_j)}{|\det H|^{\frac{1}{n}}|\det(I_d+\lambda MH^{-1})|^{\frac{1}{n}}}$$

et pour la même raison que précédemment :

$$\det(I_d+\lambda MH^{-1}) = 1+\lambda\,\mathrm{trace}(MH^{-1}) = 1+\lambda(\alpha_2\langle b_1|h\rangle - \alpha_1\langle b_2|h\rangle)$$

Enfin :

$$\alpha_2\sum_{j=1}^{m}(b_1D)_j - \alpha_1\sum_{j=1}^{m}(b_2D)_j = \alpha_2\alpha_1 - \alpha_1\alpha_2 = 0$$

et

$$\alpha_2\langle b_1|h\rangle - \alpha_1\langle b_2|h\rangle = \alpha_2(\frac{\alpha_1}{K}) - \alpha_1(\frac{\alpha_2}{K}) = 0$$

Ainsi, $V(H+\lambda M) = V(H)$. □

Pour mettre en évidence le minimum de V, nous appliquons une homothétie aux vecteurs b_i de telle sorte que $\sum_{j=1}^{m}(b_iD)_j = 1$ pour tout i. Remarquons que ceci est légèrement différent de la procédure de Dongarra et Schreiber dans laquelle ils homothétisent les générateurs de C^* pour qu'ils soient unitaires pour la norme euclidienne.

Nous sommes maintenant prêts à énoncer le résultat principal :

Théorème 2 *La valeur minimale de V est atteinte pour une matrice dont les lignes sont n vecteurs b_i linéairement indépendants.*

Preuve Selon le lemme 4, il existe une matrice inversible H_0 dans Γ telle que $V(H_0) = V_{min}$. Supposons qu'une ligne de H_0 soit une combinaison linéaire positive d'au moins deux b_i. Avec les mêmes notations que précédemment, nous avons sans perte de généralité :

$$H_0 = \lambda_1\begin{pmatrix} b_1 \\ 0 \end{pmatrix} + \lambda_2\begin{pmatrix} b_2 \\ 0 \end{pmatrix} + G \qquad \text{avec } G\in\Gamma$$

En appliquant le lemme 5, pour tout λ, $V(H+\lambda M) = V(H)$ car $V(H_0)$ est minimum.

$$H+\lambda M = (\lambda_1+\lambda\alpha_2)\begin{pmatrix} b_1 \\ 0 \end{pmatrix} + (\lambda_2-\lambda\alpha_1)\begin{pmatrix} b_2 \\ 0 \end{pmatrix} + G$$

Prenons $\lambda = \frac{\lambda_2}{\alpha_1}$ tel que

$$H+\lambda M = (\lambda_1+\lambda\alpha_2)\begin{pmatrix} b_1 \\ 0 \end{pmatrix} + G$$

$H+\lambda M$ est une matrice inversible de Γ, $V(H+\lambda M) = V_{min}$ et la première ligne de $H+\lambda M$ est une combinaison linéaire positive d'un b_i de moins. Par induction, avec les mêmes arguments que ci-dessus, on peut construire une matrice inversible H_{opt} dans Γ dont chaque ligne est un multiple d'un b_i et pour laquelle $V(H_{opt})$ est minimum. Soit B la matrice dont les lignes sont les b_i impliqués

dans H_{opt}. $H_{opt} = \Lambda B$ où Λ est une matrice diagonale positive.

$$V(H_{opt}) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(\Lambda BD)_{i,j}}{|\det \Lambda B|^{\frac{1}{n}}} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{n}\Lambda_{i,k}(BD)_{k,j}}{|\det \Lambda B|^{\frac{1}{n}}}$$

$$= \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\Lambda_{i,i}(BD)_{i,j}}{(\prod_{i=1}^{n}\Lambda_{i,i})^{\frac{1}{n}}|\det B|^{\frac{1}{n}}} = \frac{\sum_{i=1}^{n}\Lambda_{i,i}\overbrace{\sum_{j=1}^{m}(BD)_{i,j}}^{1}}{(\prod_{i=1}^{n}\Lambda_{i,i})^{\frac{1}{n}}|\det B|^{\frac{1}{n}}}$$

$$= \frac{\sum_{i=1}^{n}\Lambda_{i,i}}{(\prod_{i=1}^{n}\Lambda_{i,i})^{\frac{1}{n}}|\det B|^{\frac{1}{n}}} \geq \frac{n}{|\det B|^{\frac{1}{n}}}$$

d'après le lemme 3 appliqué à Λ. Ainsi, le minimum est atteint pour $\Lambda = I_d$. □

Ceci prouve que minimiser $V(H)$ pour toute matrice inversible H dans Γ est un problème combinatoire. Le minimum doit être recherché parmi les $\binom{t}{n}$ matrices dont les lignes sont n vecteurs b_i linéairement indépendants.

Remarquons que la contrainte sur les b_i ($\sum_{j=1}^{m}(b_iD)_j = 1$ pour tout i) peut être relachée à la condition que $\sum_{j=1}^{m}(b_iD)_j$ soit constant (ceci est dû à la propriété homothétique de V).

Retour sur l'exemple du programme 5.1 page 76 La matrice de dépendances est

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

Le cône C^* est construit comme suit. Il y a six vecteurs pouvant être des générateurs de C^* : $(0,0,1)$, $(0,1,0)$, $(0,1,1)$, $(1,0,0)$, $(1,0,1)$ et $(1,-1,0)$, mais le premier et le dernier n'appartiennent pas à C^*. On doit maintenant homothétiser les vecteurs $(0,1,0)$, $(0,1,1)$, $(1,0,0)$, $(1,0,1)$ pour que les quatre sommes $\sum_{k=1}^{n}(xD)_k$ soient égales. Elles sont déjà égales à 2, nous prenons donc : $b_1 = (0,1,0)$, $b_2 = (0,1,1)$, $b_3 = (1,0,0)$ et $b_4 = (1,0,1)$. Chaque triplet de vecteurs b_i definit une matrice H telle que $|\det(H)| = 1$. Ainsi, il y a quatre matrices minimales à une permutation près :

$$H_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad H_2 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

$$H_3 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad H_4 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

La valeur optimale de $V(H)$ est alors égale à $2+2+2=6$.

5.4 Conclusion

Nous avons étudié dans ce chapitre des techniques de pavage qui servent à adapter la granularité des nids de boucles uniformes à l'exécution sur machines parallèles à mémoire distribuée.

Après avoir passé en revue et discuté les travaux antécédants, nous avons pu identifier un nouveau critère pour définir l' « optimalité » de la forme et de la taille des tuiles. La principale

caractéristique de ce critère est son extensibilité, et ainsi sa flexibilité pour une utilisation efficace sur une grande classe de machines.

Nous avons présenté une méthode constructive pour construire le pavage optimal. Nous pensons que cette caractérisation est à la fois importante et complexe techniquement. Nous avons utilisé plusieurs exemples pour expliquer la construction et démontré comment obtenir de meilleurs résultats que ceux qui ont été publiés précédemment.

Il reste cependant beaucoup de travail à faire dans le domaine des techniques de pavage (en dépit du titre de ce chapitre !). Pour ne citer que deux directions de recherche, il est nécessaire d'élargir le champ d'application de notre technique à des nids de boucles plus généraux avec des dépendances affines ; et d'autre part, la relation entre le pavage, l'ordonnancement et l'allocation des données n'est pas encore bien comprise et nécessite une exploration plus profonde.

Chapitre 6

Évaluation d'expressions de tableaux

6.1 Introduction

Il est intéressant de paralléliser automatiquement du code, mais encore faut-il que ce code soit optimisé par le compilateur parallèle. L'objet de ce chapitre est l'étude d'une des optimisations qui peuvent être faites par le compilateur du langage parallèle utilisé, ici High Performance Fortran.

Nous étudions ici l'évaluation d'expressions de tableaux d'HPF sur des ordinateurs massivement parallèles à mémoire répartie. La difficulté d'une telle évaluation réside dans la détermination de l'ordre dévaluation et du placement des résultats intermédiaires. Chatterjee et al. [GS91, CGST92, CGST93] se sont intéressés à ce problème sous l'hypothèse stricte que les calculs et les communications ne se recouvraient pas dans le temps. Cependant, les machines actuelles peuvent recouvrir les calculs et les communications ; nous avons donc supprimé cette restriction. Cette simple modification a eu un effet important sur la complexité de l'évaluation optimale des expressions de tableaux. Nous montrons d'abord que le problème est NP-complet ; et ensuite nous présentons des heuristiques pour résoudre ce problème. Nous garantissons ces heuristiques dans des cas très importants en pratique : les calculs de faible granularité et les calculs de grosse granularité, les calculs de granularité moyenne étant moins fréquents.

6.1.1 Les règles du jeu

Nous fixons ici les définitions et les règles de base pour notre travail et nous les illustrons sur un exemple.

Notre problème est l'évaluation d'une expression binaire T. Nous supposons que T est donnée par un arbre binaire : les arrangements par commutativité ou associativité sont interdits et il n'y a pas de sous-expressions communes. De plus, sans perte de généralité, nous supposons qu'il n'y a pas d'opérateurs unaires, ce qui ne change pas beaucoup le problème. Ainsi T peut être décrite par un arbre binaire : tous les nœuds internes ont un degré entrant de 2 ; tous les nœuds sauf la racine ont un degré sortant de 1. Voir la figure 6.1 pour l'arbre correspondant à l'expression $T = f_0(f_1(f_2(A, B), f_3(C, D)), f_4(E, F))$.

Oublions un moment les tableaux répartis et les expressions HPF, i.e. ne relions pas les nœuds de l'arbre avec les données. Donnons plutôt une interprétation abstraite de notre problème : les feuilles représentent des positions et les nœuds internes des calculs (on dit aussi que ces nœuds sont évalués). Un nœud interne peut être évalué si et seulement si

– ses deux fils ont été évalués (les feuilles ne nécessitent aucune évaluation),

FIG. **6.1 -** Arbre d'une expression simple

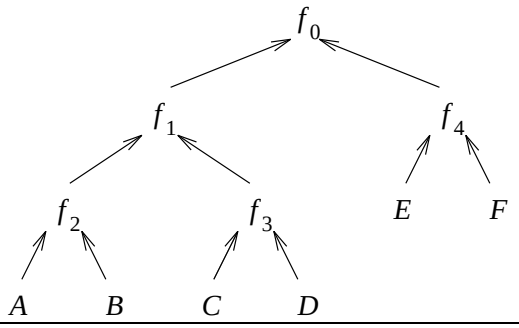
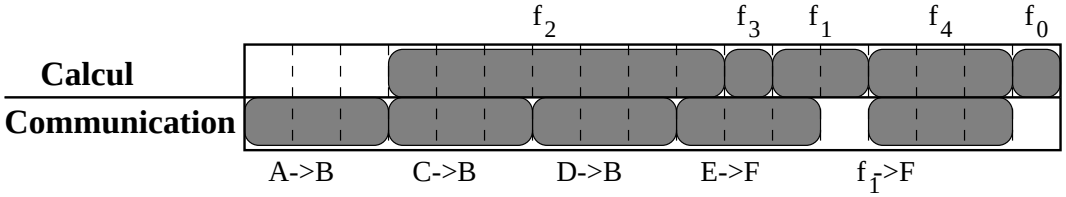


FIG. **6.2 -** Une exécution simple

Temps de calcul :

f_0	f_1	f_2	f_3	f_4
1	2	7	1	3

Diagramme d'exécution :



– ses deux fils sont placés de la même façon.

Dans l'exemple précédent, nous avons six feuilles, les nœuds A, B, C, D, E et F, et cinq nœuds internes, f_i, $0 \leq i \leq 4$.

Si les deux fils d'un nœud interne ne sont pas placés dans la même position, une communication au moins est nécessaire. Par exemple pour évaluer le nœud f_2, on peut décider de transférer les données de la position A vers la position B et évaluer le nœud f_2 dans la position B, ou vice-versa. Mais on peut aussi décider d'avoir deux communications, par exemple depuis la position A vers la position C, et de B vers C ; le nœud f_2 sera alors évalué dans la position C. Cela permettrait le calcul du nœud f_1 sans communication en position C, pourvu que le nœud f_3 ait été évalué à la position C, utilisant une communication de la feuille D vers la feuille C.

Quelles sont les règles du jeu pour ce problème abstrait ? Nous devons évaluer l'expression T le plus vite possible tout en respectant l'ordre partiel induit par l'arbre. Calculs et communications peuvent avoir lieu en parallèle. Plus précisément, nous supposons qu'un calcul et une communication peuvent avoir lieu simultanément. De plus, les communications sont séquentielles entre elles et les calcul sont aussi séquentiels entre eux. D'un certain sens, c'est comme un problème d'ordonnancement à deux processeurs, mais avec une machine dédiée aux calculs et une autre aux communications.

Voir la figure 6.2 pour un exemple. Nous supposons ici que toute les communications coûtent 3 unités de temps alors que les temps de calcul sont donnés dans la table. Bien sûr, il n'y a a priori aucune raison pour que les coûts de communication soient les mêmes, mais il se trouve que c'est un cas important en pratique (voir section 6.1.2).

Mettons en avant quelques spécifications du problème :

– La position où le résultat final doit être disponible n'est pas spécifiée dans le problème. Si

T est une affectation mettant en jeu une expression binaire, on pourrait imposer la position du résultat. Cela ne change pas beaucoup le problème car on peut toujours ajouter une communication à la fin.

- Comme pour le modèle d'évaluation, nous faisons une double supposition :
  - Il ne peut y avoir qu'un calcul au plus à un moment donné. Ceci est une conséquence naturelle de la formulation originelle du problème en termes d'expressions de tableaux d'HPF (voir section 6.1.2).
  - Il ne peut y avoir qu'une communication au plus à un moment donné. Cette hypothèse est plus discutable, car elle vient de la modélisation des machines parallèles actuelles. Nous pourrions supposer que plus d'une communication peuvent avoir lieu en parallèle mais cela induirait des conflits dans le réseau de communication et les temps de communication seraient encore plus difficiles à modéliser (cependant, la plus grande partie de notre travail peut être étendue directement pour tenir compte de plusieurs communications simultanées)

6.1.2 Expressions de tableaux d'HPF

La motivation originelle pour le problème ci-dessus vient de l'évaluation des expressions de tableaux d'HPF. Considérons une nouvelle fois l'expression

$$T = f_0(f_1(f_2(A, B), f_3(C, D)), f_4(E, F))$$

Supposons que nous avons un tore de processeurs de dimension 2 et de taille $P \times P$ (comme le Paragon d'Intel), et les tableaux a, b, c, d, e, f et res de taille $N \times N$.

Considérons le nid de boucles du programme 6.1. Il correspond à l'expression T précédente, sauf qu'on a une affectation (on pourrait modifier facilement l'arbre d'expression pour tenir compte de cette affectation).

PROG. **6.1** Un nid de boucles correspondant à l'expression T

```
do i = 3, N-2
   do j = 7, N-5
      res(i,j) = ((a(i-2,j+3)+b(i,j-6))*(c(i-1,j+3)+d(i+2,j+5)))
                 +(e(i-1,j+2)/f(i,j))
   enddo
enddo
```

Supposons que nous avons le schéma de placement des données, statique, en HPF, donné par le programme 6.2. Ce nid de boucles aurait pu être écrit en utilisant la notation de sections de tableaux (voir le programme 6.3).

Cette répartition des données indique que tous les tableaux sont alignés de la même façon (sur le même « `template` ») qui est réparti dans le deux dimensions de manière cyclique. Chaque processeur du tore contient ainsi une section de chaque tableau. Nous pouvons maintenant faire le lien avec les positions A à F qui apparaissent dans l'expression T. Considérons par exemple le nœud f_2 :

$$f_2 : temp(i, j) = a(i - 2, j + 3) + b(i, j - 6), 3 \leq i \leq N - 2, 7 \leq j \leq N - 5.$$

L'élément de tableau $a(i - 2, j + 3)$ est stocké dans le processeur $proc_a(i, j) = (i - 2 \bmod P, j + 3 \bmod P)$ alors que l'élément $b(i, j - 6)$ est stocké dans le processeur $proc_b(i \bmod P, j - 6 \bmod P)$.

PROG. **6.2** Un nid de boucles correspondant à l'expression T avec une répartition des données en HPF

```
      PARAMETER (N = 100, P = 10)
!HPF$ PROCESSORS PROC(P,P)
!HPF$ TEMPLATE T(N,N)
!HPF$ ALIGN WITH T:: a, b, c, d, e, f, res
!HPF$ DISTRIBUTE T(CYCLIC,CYCLIC) ONTO PROC
      FORALL(i=3:N-2, j=7:N-5)
            res(i,j) = ((a(i-2,j+3)+b(i,j-6))*(c(i-1,j+3)+d(i+2,j+5)))
                       +(e(i-1,j+2)/f(i,j))
```

PROG. **6.3** Un nid de boucles correspondant à l'expression T écrit avec la notation de sections de tableaux

```
      PARAMETER (N = 100, P = 10)
!HPF$ PROCESSORS PROC(P,P)
!HPF$ TEMPLATE T(N,N)
!HPF$ ALIGN WITH T:: a, b, c, d, e, f, res
!HPF$ DISTRIBUTE T(CYCLIC,CYCLIC) ONTO PROC
      res(3:N-2,7:N-5) = ((a(1:N-4,10:N-2)+b(3:N-2,1:N-11))
                *(c(2:N-3,10:N-2)+d(5:N,12:N)))
                +(e(2:N-3,9:N-3)/f(3:N-2,7:N-5))
```

Ainsi chaque élément $b(i, j-6)$ doit être envoyé selon un vecteur de distance $\binom{-2}{9}$ si on décide de calculer le résultat temporaire $temp(i,j)$ du nœud f_2 dans $proc_a(i,j)$. Notons $\tau(u,v)$ la translation de vecteur $\binom{u}{v}$. Cette communication correspond à un *décalage global* du tableau b pour *aligner* l'origine du tableau a translaté de $\tau(-2,3)$ avec l'origine du tableau b translaté de $\tau(0,-6)$ (car a et b sont alignés sur le même « `template` »).

Comme cela a déja été dit, rien ne nous empêche d'évaluer le nœud f_2 à une autre position. Nous pouvons choisir une autre « origine » et communiquer les tableaux a et b en fonction de ce choix.

Nous comprenons maintenant pourquoi on ne peut faire qu'un calcul à un moment donné : c'est parce que tous les processeurs opèrent en parallèle sur différentes sections des tableaux répartis impliqués dans l'expression. Cependant, on pourrait avoir plusieurs décalages en parallèle selon les possibilités de la machine cible.

La plupart des machines parallèles sont capables de faire au moins une communication en parallèle avec les calculs, d'où notre hypothèse clé : les calculs et les communications se recouvrent.

Évaluer les coûts de communication est bien plus difficile que les coûts de calcul, et ceci pour plusieurs raisons :

- Les machines parallèles actuelles ont des moyens dédiés au routage, de telle sorte que la longueur d'une communication peut s'avérer moins importante que le nombre de conflits sur les liens de communication [CD94, EK93, PS92].

- Même dans un cas simple comme l'exemple de l'arbre, la taille des communications dépend de la répartition des données et de leur alignement.

C'est pourquoi nous supposons que les coûts de communication sont des paramètres fixés qui peuvent être calculés avec des formules complexes où, parmi les paramètres les plus importants, on

trouve la taille du message, le coût d'initiation, la distance, et surtout le coût des conflits[1].

Cependant, en pratique, on peut souvent supposer que les coûts de communication sont constants car il serait très compliqué d'en calculer une approximation à chaque fois. En effet, nous ne connaissons pas les algorithmes utilisés pour le routage des messages et donc nous ne pouvons pas deviner quand il y aura des conflits. Nous considérons donc ici le cas où les temps de communications sont constants (le maximum ou la moyenne des coûts réels). Cela modélise particulièrement bien les motifs de communications qui viennent de dépendances de données uniformes. Un autre cas important en pratique est quand tous les coûts de communications sont inférieurs à tous les coûts de calcul (grosse granularité). Nous étudions ce cas dans la section 6.4.4.

6.1.3 Organisation du chapitre

Ce chapitre est organisé comme suit : pour commencer, dans la section 6.2, nous passons en revue des travaux précédents liés au sujet traité ici. Ensuite, dans la section 6.3, nous montrons qu'une instance simple du problème est NP-complète. Nous proposons enfin dans la section 6.4 une heuristique qui est optimale dans un cas restreint mais utile, c'est-à-dire les calculs de grosse granularité et aussi dans un sous cas des calculs de faible granularité. Pour finir, nous donnons nos conclusions dans la section 6.5.

6.2 Revue de travaux précédents

Ce travail s'insère bien dans le domaine des compilateurs paralléliseurs s'occupant de placement de données. Ce problème a été étudié par plusieurs chercheurs : Anderson et Lam [AL93], Lukas et Knobe [LK92, KLS90, KN93], Li et Chen [LC91], Feautrier [Fea94], O'Boyle et Hedayat [O'B92, OH92], Ramanujam et Sadayappan [RS91], Huan et Sadayappan [HS91] et Darte et Robert [DR93b], entre autres.

Dans le champ de l'évaluation parallèle d'expressions de tableaux, Gilbert et Schreiber [GS91] ont proposé un algorithme optimal pour aligner les temporaires dans des arbres d'expressions en caractérisant les réseaux d'interconnexion à travers des espaces métriques. Leur algorithme s'applique à une classe de métriques appelées « robustes ».

Chatterjee *et al.* [CGST92] ont étendu le travail de Gilbert et Schreiber dans deux directions. Ils ont reformulé le problème en termes de programmation dynamique et ont proposé une famille d'algorithmes pour traiter le cas de nombreuses métriques robustes et non-robustes. Ils autorisent aussi des tableaux non superposables en étudiant le cas d'arbres d'expressions avec arêtes pondérées.

Chatterjee *et al.* [CGST93] ont étendu leur travail précédent en étudiant le cas d'un programme complet et plus seulement d'une expression. Ils proposent un alignement pour tous les objets présents dans le programme, aussi bien pour les variables nommées que pour les temporaires générés par le compilateur. Ils ne se limitent pas à la règle des *écritures locales*.

Nous concentrons notre travail sur les expressions de tableaux mais à la différence de Chatterjee *et al.*, nous ne nous focalisons pas sur la forme du réseau d'interconnexion et la minimisation du temps de communication, mais plutôt sur la minimisation du temps d'exécution total en autorisant le recouvrement des calculs et des communications. En effet, nous considérons une autre modélisation des machines massivement parallèles à mémoire répartie modernes qui sont pour la plupart capables de faire des calculs et des communications simultanées. De plus, leur réseau d'interconnexion est souvent basé sur des routeurs, et donc les temps de communications ne peuvent pas

1. Des expériences dépendant de la machine seraient nécessaires pour déterminer une formule approximative qui donnerait le coût de communication en fonction de la taille des données et du motif de communication.

être déduits facilement du placement des processeurs mais dépend principalement des conflits de communication.

De nombreux résultats de NP-complétude sont apparus dans le contexte du placement automatique des données. Li et Chen [LC90] ont montré que le problème de la détermination d'un alignement statique optimal entre les dimensions de tableaux distincts est NP-complet. Anderson et Lam [AL93] ont montré que le problème du placement dynamique des données est NP-dur en présence d'un flux de contrôle entre des nids de boucles. Mace [Mac87] a confronté trois formulations du problème du placement dynamique des données pour des machines à mémoire entrelacée. Kremer [Kre93] a prouvé que le problème du placement des données entre phases est NP-complet. Gilbert et Schreiber [GS91] ont prouvé qu'aligner les temporaires dans les expressions de tableaux avec sous-expressions communes est NP-complet suivant l'hypothèse de différentes stratégies.

Nous montrons ici que, avec des hypothèses différentes de celles de Gilbert et Schreiber, aligner les temporaires dans les expressions de tableaux est NP-complet.

6.3 NP-complétude du problème

Dans cette section, nous définissons formellement une version simplifiée du problème de l'évaluation d'expressions de tableaux HPF, appelée *EXP-DAG*. Nous prouvons que ce problème simplifié est NP-complet par réduction au problème de satisfaisabilité *m3-SAT*. Insistons sur le fait que pour *EXP-DAG*, les coûts de communication sont supposés constants : même avec cette hypothèse, le problème présente encore une complexité importante.

6.3.1 Définition du problème simplifié

Nous considérons ici le problème de l'évaluation d'une expression de tableau qui est modélisée par ce que nous appelons un « DAG-expression » et que nous définissons comme suit :

1. un DAG-expression E est un graphe direct et acyclique[2] avec
   - f feuilles f_i, $1 \leq i \leq f$
   - n nœuds internes n_i, $1 \leq i \leq n$ étiquetés par des coûts de communications entiers et positifs, $\delta_{calc}(i)$, $1 \leq i \leq n$
   - un coût de communication δ_{com} positif supposé indépendant de la distance entre les localisations et de la longueur du message
   - tous les degrés rentrants sont égaux à 1 ou 2 (ceci est une simplification, pas une restriction; le problème serait encore plus compliqué sans cela).
2. Les règles du jeu :

   séquentialité des communications

   séquentialité des calculs

   recouvrement : une communication peut avoir lieu en même temps que des calculs

   pas de réutilisation de la mémoire : une valeur ne peut être utilisée qu'une seule fois. Si elle doit être réutilisée, elle doit être recommuniquée.

2. *Direct Acyclic Graph* en anglais, d'où le nom DAG

ordre des calculs et des communications : pour calculer un nœud, tous ses prédécesseurs doivent avoir été évalués et tous les opérandes doivent être dans la même localisation. Sinon, ils doivent être communiqués dans une localisation commune, celle de l'un d'entre eux ou n'importe quelle autre. Supposons par exemple que le nœud n a deux prédécesseurs n_1 et n_2 qui ont été évalués aux localisations loc_1 et loc_2 :

- si $loc_1 = loc_2$ alors le nœud n peut être calculé dans la localisation loc_1 sans communication,
- si $loc_1 \neq loc_2$, alors le nœud n ne peut être calculé dans la localisation loc_1 qu'après que le résultat du nœud n_2 a été communiqué de loc_2 à loc_1. Une telle communication peut avoir lieu dès que le nœud n_2 a été calculé, sans attendre la fin du calcul de n_1. Cette situation est bien sûr symétrique pour le calcul en loc_2 en échangeant le rôle de n_1 et n_2
- dans tous les cas, le calcul du nœud n dans une localisation distincte de loc_1 et loc_2 est possible, mais au prix de deux communications

les feuilles sont directement disponibles (pas de calcul).

3. une constante K positive qui représente la borne de l'ordonnancement.

Le problème de décision est : « est-il possible de construire un ordonnancement pour l'évaluation d'un DAG-expression E qui mène à un temps d'exécution total inférieur à K ? » (on dit $E \in$ *EXP-DAG*(K)).

Théorème 3 *EXP-DAG(K) est NP-complet.*

6.3.2 Preuve du théorème

***EXP-DAG(K)* est dans la classe NP**

C'est la partie facile. Clairement, étant donné un ordonnancement, c'est-à-dire deux listes :

1. la liste des calculs : quand et dans quelle localisation chaque nœud interne doit être calculé,
2. la liste des communications : pour quel résultat, quand, de où et vers où chaque communication a lieu,

nous pouvons vérifier en temps polynômial si les règles du jeu sont respectées et si le temps d'exécution est inférieur à K.

Réduction à *monotone 3SAT*

Une instance du problème *monotone 3SAT* (*m3SAT* [GJ91, p. 259]) consiste en une expression booléenne B sous forme normale conjonctive,

$$B = \bigwedge_{i=1}^{t} C_i$$

- où $C_i = x_i^1 \vee x_i^2 \vee x_i^3$, $1 \leq i \leq t$, est une clause
- où chaque littéral x_i^j est une variable ou sa négation dans l'ensemble des variables $\mathcal{V} = \{v_1, \cdots, v_r\}$

– avec la restriction supplémentaire que chaque clause C_i ne contient que des variables ou que des négations de variables (d'où le nom « monotone »). Une clause qui ne contient que des variables est appelée une « clause vraie », sinon, on a une « clause fausse ».

Le problème de décision associé est représenté comme suit : existe-t-il une assignation de valeurs $w : \mathcal{V} \to \{vrai, faux\}$ telle que B s'évalue à *vrai* sous w? (on dit $B \in m3SAT$).
Voici un exemple que nous allons utiliser tout au long de la preuve : $t = 3$, $r = 4$, et

$$B = (v_1 \vee v_2 \vee v_4) \wedge (\neg v_1 \vee \neg v_3 \vee \neg v_4) \wedge (v_2 \vee v_3 \vee v_4).$$

Étant donnée B, nous construisons le DAG-expression $g(B)$ comme suit. Soit $c = \delta_{com}$ le coût constant de communication. Pour clarifier la construction, nous donnons des noms intuitifs aux nœuds de $g(B)$.

Premiers nœuds : nous commençons avec $2r + 3$ feuilles, ne partageant que deux localisations distinctes nommées V (pour *Vrai*) et F (pour *Faux*).

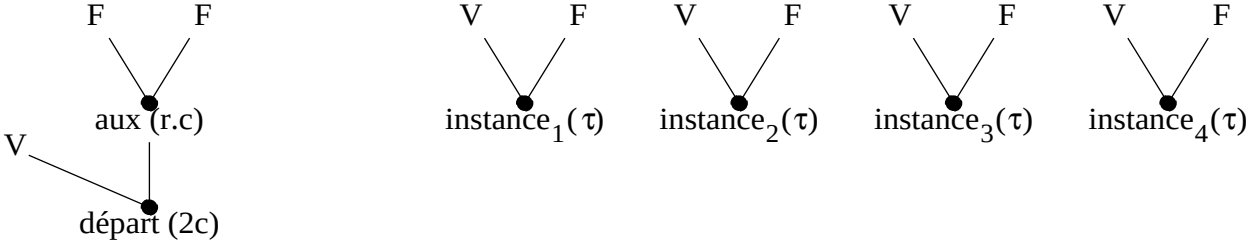
– Pour tout i, $1 \leq i \leq r$, deux feuilles, l'une avec la localisation V et l'autre avec la localisation F, sont connectées à un nœud interne $instance_i$: intuitivement, le nœud $instance_i$ peut être calculé dans la localisation V ou F, ce qui correspond à l'instanciation du littéral v_i, i.e. assignation d'une valeur à v_i. Le temps de calcul du nœud $instance_i$ est $\delta_{calc}(instance_i) = \tau$, où $r\tau = c$.

– Deux feuilles partageant la même localisation V ou F, dépendant du type de la première clause, *fausse* ou *vraie* respectivement, et sont connectées au nœud interne *aux* dont le temps de calcul est $\delta_{calc}(aux) = rc$. r est le nombre de variables et t le nombre de clauses.

– La dernière feuille est à la localisation V or F, dépendant du type de la première clause, *vraie* ou *fausse* respectivement, au contraire des deux feuilles précédentes. Cette dernière feuille est connectée au nœud interne *départ* ainsi que le nœud *aux*.Le temps de calcul du nœud *départ* est $\delta_{calc}(départ) = 2c$.

dans notre exemple, comme la première clause est vraie, *départ* est à la localisation V, comme illustré par la figure 6.3.

FIG. **6.3 -** Premières feuilles du DAG-expression (les temps de calcul sont entre parenthèses)



Réseau : Pour chaque clause C_i on utilise une copie R_i du même réseau R. Le réseau R est constitué de 6 feuilles et 10 nœuds internes, comme décrit par la figure 6.4. Tout nœud interne a un temps de calcul égal à 0.

Pour chaque clause $C_i = x_i^1 \vee x_i^2 \vee x_i^3$, soit $\mathcal{V}_i$ l'ensemble des littéraux de C_i. Alors le réseau R_i est connecté aux nœuds *instance* correspondant aux littéraux de $\mathcal{V}_i$. Nous illustrons ceci

FIG. **6.4 -** Le réseau R

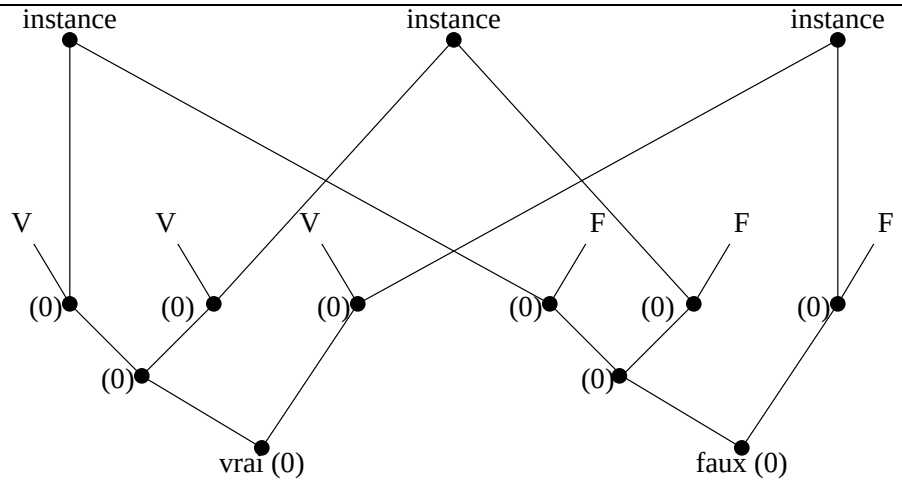
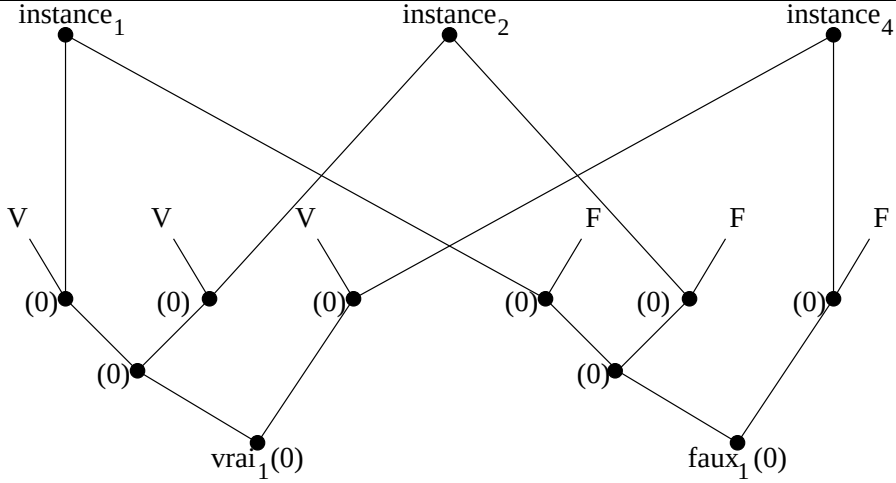


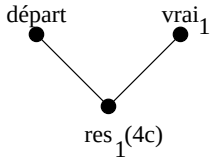
FIG. **6.5 -** Le réseau R_1



avec le réseau R_1 sur la figure 6.5.

Assemblage : Nous connectons les copies des réseaux en mode « série ». Le nœud *départ* est connecté au nœud *vrai* si la première clause est une clause *vraie* et au nœud *faux* dans le cas contraire (voir figure 6.6).

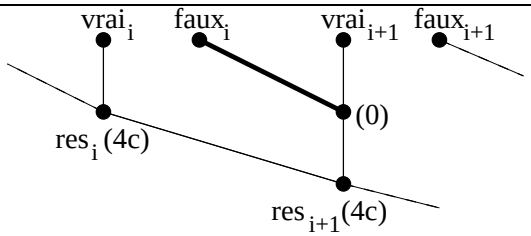
FIG. **6.6 -** La connexion entre *départ* et R_1.



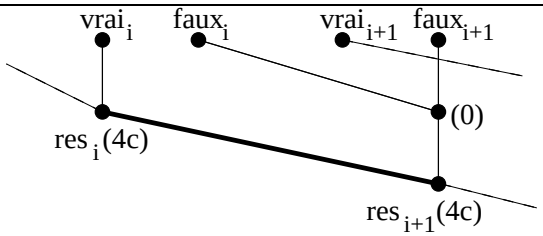
Ce nœud de connexion, res_1, a un temps de calcul de $4c$. La connexion entre une clause et la suivante dépend de leur type. Voir la figure 6.7 pour les quatre cas. Tous les nouveaux nœuds ont un temps de calcul nul, sauf un nœud spécial par réseau, étiqueté res_i pour R_i. Le temps de calcul du nœud res_i est $4c$, sauf pour le dernier, le nœud res_t:

$$\delta_{calc}(res_i) = 4c, 1 \leq i \leq t-1 \text{ et } \delta_{calc}(res_t) = 2c.$$

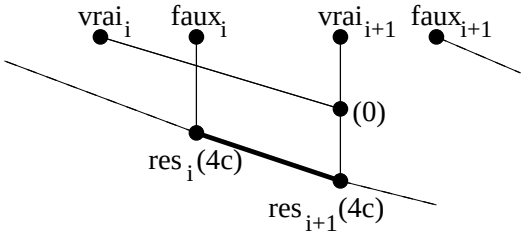
FIG. **6.7 -** Connexion entre deux clauses



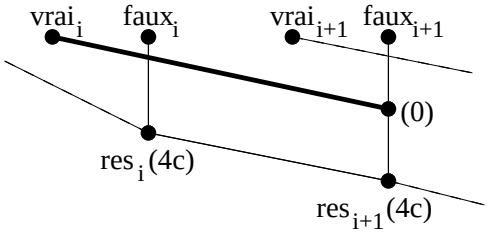
(a) clause vraie suivie d'une clause vraie
communication: faux$_i$ -> V



(b) clause vraie suivie d'une clause fausse
communication: res$_i$ -> F



(c) clause fausse suivie d'une clause vraie
communication: res$_i$ -> V



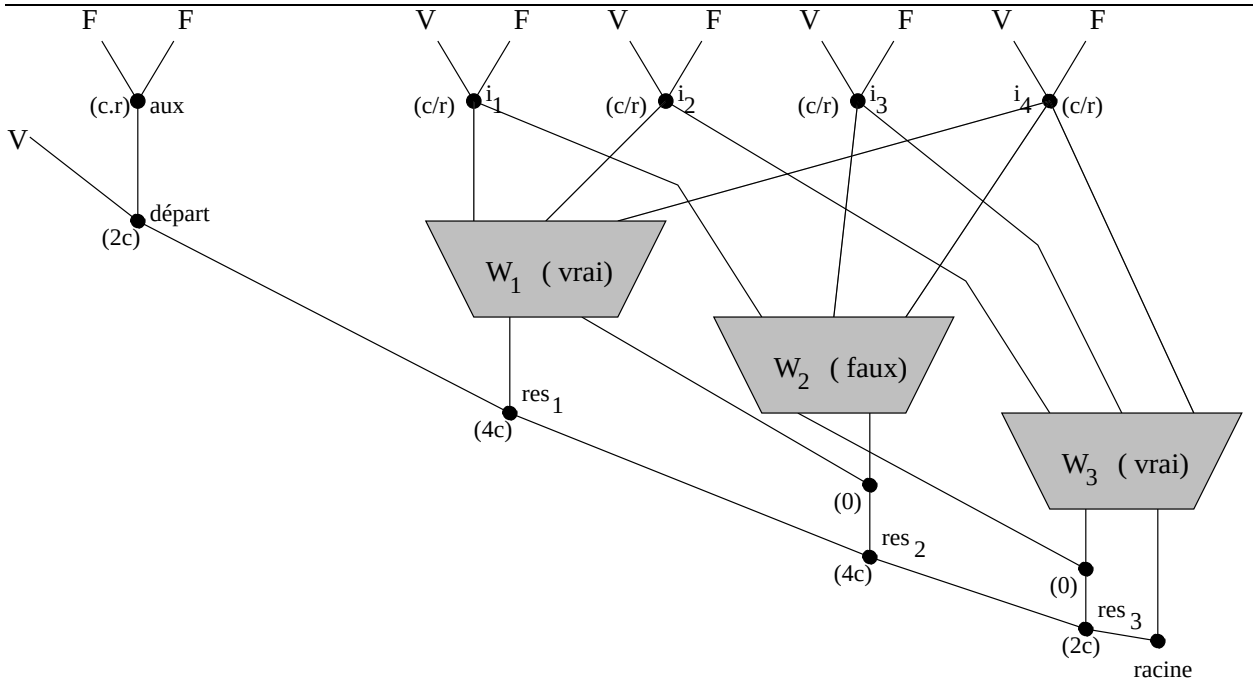
(d) clause fausse suivie d'une clause fausse
communication: vrai$_i$ -> F

Enfin, voir la figure 6.8 pour le DAG-expression complet de notre exemple, $g(B)$. Les nœuds res_t et $faux_t$ sont connectés au nœud racine, *racine*, si la dernière clause est *vraie*. Sinon, ce sont res_t et $vrai_t$ qui sont connectés à *racine*. Nous prenons $\delta_{calc}(racine) = 0$.

Nous vérifions aisément le lemme suivant.

Lemme 6 *La construction du DAG-expression $g(B)$ est polynômiale en la taille de B.*

FIG. **6.8 -** Le DAG-expression complet



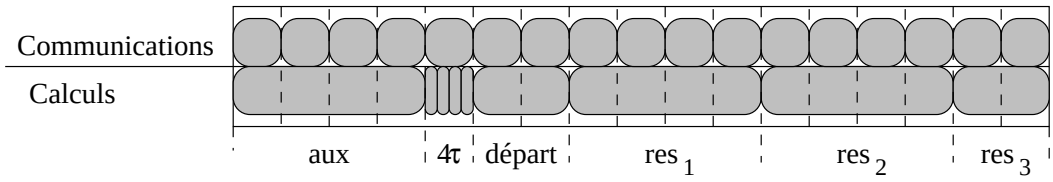
Ordonnancement de $g(B)$

L'idée directrice est la construction d'un ordonnancement « sans trous » pour $g(B)$ si et seulement s'il existe une assignation de valeurs de vérité pour B.

Lemme 7 *Soit $w : \mathcal{V} \to \{vrai, faux\}$ une assignation de valeurs telle que B s'évalue en vrai sous w. Nous pouvons alors ordonnancer le DAG-expression $g(B)$ en temps $(4t + r + 1) \times c$.*

Preuve Dans notre exemple, B peut être satisfaite avec l'assignation $v_1 = v_2 = vrai$, $v_3 = v_4 = faux$. Voir figure 6.9 pour suivre l'ordonnancement.

FIG. **6.9 -** L'ordonnancement



Pendant les $r \times c$ premiers pas de temps, on évalue le nœud aux dans la localisation X, sans communication. Pendant ce temps, on fait r communications: si $w(v_i) = vrai$ alors on évalue le nœud $instance_i$ dans la localisation V, ainsi on communique sa feuille de la localisation F vers la localisation V. On fait le contraire si $w(v_i) = faux$.

Pendant les c pas de temps suivants, on évalue les r nœuds $instance_i$, $1 \leq i \leq r$ (en effet $r \times \tau = c$). Pendant ce temps, on communique le résultat de aux à la localisation de l'autre feuille du nœud *départ* (V dans notre exemple, car la première clause est une clause *vraie*).

Pendant les $2c$ pas de temps suivants, on évalue le nœud *départ*. On peut faire deux communications pendant ce temps. C'est ici que la partie « astucieuse » de la preuve arrive. Considérons

la première copie du réseau pour la clause C_1, qui est une clause *vraie*. Il doit y avoir au moins un littéral de $\mathcal{V}_1$ qui a été instancié à *vrai* (car B est satisfaite), et le nœud *instance* a été calculé dans la localisation V. Ainsi, comme on a le temps pour deux communications, on est assuré de pouvoir déplacer les données des autres nœuds *instance* V_i si elles ne sont pas déjà dans la localisation V. Dans notre exemple, on ne doit communiquer que (le résultat) du nœud $instance_4$ vers la localisation V.

Notons qu'on a toujours exactement trois communications avant l'évaluation des nœuds *vrai* et *faux* d'un réseau, quelle que soit l'instanciation des littéraux : comme les nœuds *instance* sont déjà dans la localisation V ou F, il reste trois communications pour évaluer le mot *vrai* dans la localisation V et le nœud *faux* dans la localisation F. Mais l'idée clé est la suivante : on ne pourra évaluer le nœud *res* après trois unités de temps que si la clause est satisfaisable. Dans le cas contraire, on devrait faire trois communications. On pourrait évaluer le nœud *faux* immédiatement, mais cela ne donnerait pas de calcul supplémentaire plus tard.

Dans l'exemple, en plus de communiquer le nœud $instance_4$, on communique le nœud $instance_1$ vers la localisation F.

Comme la clause est satisfaisable, on peut évaluer le nœud res_1 pendant les $4c$ pas de temps suivants. On fait la dernière communication pour le nœud *faux* (depuis le nœud $instance_2$ vers la localisation F dans l'exemple), et on transmet l'information au réseau suivant. Cela coûte toujours une communication, comme expliqué sur la figure 6.7. On envoie ici le nœud *faux* vers la localisation V. Il reste $2c$ unités de temps, d'où la possibilité de faire deux communications.

On gère le second réseau exactement comme le premier. Après deux communications, comme la seconde clause est satisfaisable, on pourra évaluer le nœud res_2. On aura $4c$ unités de temps de calcul qu'on dépensera aussi, en deux communications pour finir le réseau R_2 et deux communications pour le réseau suivant.

Le dernier réseau est spécial en ce qu'on n'a besoin que de finir son évaluation et de faire une communication avant d'évaluer le nœud *racine*. D'où le choix $\delta_{calc}(res_t) = 2c$.

Finalement, le temps d'exécution total de $g(B)$ est $(4t + r + 1)c$ ($17c$ pour notre exemple). □

Lemme 8 *Soit B une formule avec t clauses et r variables, et soit $K = (4t + r + 1)c$, où c est un entier positif fixé, soit $E = g(B)$ le DAG-expression construit ci-dessus, on a alors*

$$B \in m3SAT \Leftrightarrow g(B) \in EXP\text{-}DAG(K).$$

Preuve Nous avons déjà prouvé

$$B \in m3SAT \Rightarrow g(B) \in EXP\text{-}DAG(K).$$

Réciproquement, supposons que $g(B) \in EXP\text{-}DAG(K)$. Alors nous pouvons affirmer que tous les nœuds $instance_i$ sont évalués soit dans la localisation V soit dans la localisation F. On pose :

$$\begin{aligned} w : \mathcal{V} &\to \{vrai, faux\} \\ v_i &\mapsto \text{ localisation du nœud } instance_i. \end{aligned}$$

Nous affirmons alors que w est une assignation de valeurs de vérité pour B, d'où $B \in m3SAT$. Nous ne détaillons pas la preuve de ces affirmations mais l'idée sous-jacente suit la construction du lemme précédent. Si ces affirmations ne tenaient pas, il y aurait au moins un « trou » dans l'ordonnancement. Comme la somme des temps de calcul de tout les nœuds du DAG-expression $g(B)$ est exactement égale à la longueur de l'ordonnancement, aucun trou ne peut apparaître.

La preuve du théorème est maintenant complète. □

Conjecture 1 *Le problème est NP-complet même dans le cas simple de l'arbre binaire dont les feuilles sont à des localisations toutes différentes.*

6.4 Heuristiques

6.4.1 Introduction

Maintenant que nous avons prouvé que le problème général est NP-complet, nous proposons dans cette section des heuristiques pour le résoudre. Heureusement ces heuristiques donnent un temps optimal dans des cas utiles en pratique.

Voici quelques notations utilisées dans cette section :

arbre-expression : c'est un DAG-expression[3] dont les nœuds internes forment un arbre,

$\boldsymbol{n}$ représente le nombre de nœuds internes de l'arbre-expression,

$\boldsymbol{\delta_{calc}(i)}$ est le temps nécessaire pour évaluer l'opération du nœud i,

$\boldsymbol{\delta_{com}}$ est le temps nécessaire pour communiquer les données d'une localisation à une autre,

racine représente le nœud racine de l'arbre-expression, i.e. le seul nœud de degré sortant 0,

$\boldsymbol{\mathcal{N}}$ est l'ensemble des nœuds de l'arbre-expression,

$\boldsymbol{\mathcal{F}}$ est l'ensemble des feuilles de l'arbre-expression,

$\boldsymbol{\mathcal{I}}$ est l'ensemble des nœuds internes de l'arbre-expression.

6.4.2 Quand toutes les feuilles sont dans des positions différentes

Hypothèses

Nous supposons ici que toutes les feuilles sont distinctes. Nous sommes alors dans le cas où l'arbre-expression est en fait un arbre binaire localement complet.

Propriété 1 *Pour un arbre binaire localement complet avec $n+1$ feuilles, il y a au moins n communications nécessaires.*

Preuve La démonstration se fait par analogie avec le problème de connexité d'un graphe. Pour évaluer un nœud interne, les deux opérandes (ses deux fils) doivent partager la même localisation. Un sommet du graphe représente une localisation (une feuille dans notre cas). Une arête représente une communication entre deux localisations. Le problème original se transforme alors en trouver le nombre minimal d'arêtes pour que le graphe soit connexe. C'est le problème bien connu de la connexité par arêtes dont la réponse pour $n+1$ sommets est n arêtes. □

3. Voir page 98 pour la définition d'un DAG-expression.

Borne inférieure

Soit

$$B = \delta_{com} + \max\left(\sum_{i \in \mathcal{I} \setminus racine} \delta_{calc}(i), (n-1)\delta_{com}\right) + \delta_{calc}(racine) \tag{6.1}$$

$$= \max\left(\delta_{com} + \sum_{i \in \mathcal{I}} \delta_{calc}(i), n\delta_{com} + \delta_{calc}(racine)\right) \tag{6.2}$$

B est une borne inférieure du temps nécessaire pour évaluer l'expression. En effet, la première étape de l'exécution est une communication pour amener l'un des opérandes du premier calcul à la même localisation que l'autre opérande, d'où le premier δ_{com}. Ensuite, la dernière étape est le calcul de la racine de l'arbre, d'où le $\delta_{calc}(racine)$. Entre deux, comme la machine ne peut faire qu'un calcul à la fois, tous les calculs sont séquentialisés, d'où le terme $\sum_{i \in \mathcal{I} \setminus racine} \delta_{calc}(i)$. De même, les communications sont aussi séquentialisées, d'où les $(n-1)\delta_{com}$. Comme les calculs et les communications peuvent avoir lieu simultanément, B est une borne inférieure du temps d'exécution de l'arbre-expression.

Heuristique

Évaluation d'un nœud Pour minimiser le nombre de communications, nous calculons chaque résultat intermédiaire à la localisation de l'un des deux opérandes dont il dépend. Nous avons donc un calcul et une communication par nœud.

Stratégie générale Nous considérons un ordre total sur les nœuds internes de l'arbre. Nous évaluons ensuite les nœuds suivant cet ordre au plus tôt. Voici l'ordre $\prec$ que nous considérons :

1. L'évaluation est faite par niveau de profondeur, en partant des feuilles les plus profondes et, quand un niveau a été complété, en remontant au niveau supérieur. Chaque niveau est pris du nœud le plus à gauche vers le nœud le plus à droite.

2. La direction de communication est toujours depuis le fils gauche, sauf si le fils droit est une feuille, auquel cas, la communication se fait à partir du fils droit.

3. Les communications sont faites dans l'ordre $\prec$ d'exécution des nœuds, au plus tôt.

Les cas optimaux Nous supposons que : soit tous les temps de calcul sont inférieurs au temps de communication, soit ils lui sont tous supérieurs. Dans ces cas particuliers mais réalistes (granularité fine ou grosse), nous pouvons décrire un ensemble de stratégies qui donnent un temps d'exécution égal à la borne inférieure B.

Théorème 4 *Si on suppose que toutes les feuilles sont distinctes, l'évaluation proposée précédemment donne un temps d'exécution optimal.*

Preuve Notons $t_{calc}(i)$ l'instant du début du calcul de l'opération du nœud i. De même, $t_{com}(i)$ représente l'instant du début de la communication nécessaire pour le nœud i. Notons aussi prev(i) le prédécesseur du nœud i dans l'ordre $\prec$ et filsC(i) le fils du nœud i d'où part la communication. Notons index(i) l'indice du nœud interne i dans l'ordre $\prec$.

La stratégie décrite ci-dessus se traduit en :
Soit $i_0 \in \mathcal{I} \mid \text{index}(i_0) = 1$,

$$t_{com}(i_0) = 0 \tag{6.3}$$
$$t_{calc}(i_0) = \delta_{com} \tag{6.4}$$

$\forall i \in \mathcal{I} \setminus \{i_0\}$,

$$t_{com}(i) = \max(t_{com}(\text{prev}(i)) + \delta_{com}, t_{calc}(\text{filsC}(i)) + \delta_{calc}(\text{filsC}(i))) \tag{6.5}$$
$$t_{calc}(i) = \max(t_{com}(i) + \delta_{com}, t_{calc}(\text{prev}(i)) + \delta_{calc}(\text{prev}(i))) \tag{6.6}$$

où

$$\forall i \in \mathcal{I}, \text{filsC}(i) \in \mathcal{F} \Rightarrow \delta_{calc}(\text{filsC}(i)) = t_{calc}(\text{filsC}(i)) = 0.$$

Nous pouvons déduire de la description de l'ordre d'évaluation la relation :

$$\forall i \in \mathcal{I}, \text{filsC}(i) \prec \text{prev}(i) \text{ ou } \text{filsC}(i) \in \mathcal{F}. \tag{6.7}$$

Il y a alors deux cas :

1. $\forall i \in \mathcal{I}, \delta_{calc}(i) \leq \delta_{com}$

   Montrons par induction sur les équations (6.3), (6.4), (6.5) et (6.6) que $\forall i \in \mathcal{I}$,

   $$t_{com}(i) = (\text{index}(i) - 1)\delta_{com} \tag{6.8}$$
   $$t_{calc}(i) = \text{index}(i).\delta_{com}. \tag{6.9}$$

   Ces équations traduisent le fait que le chemin critique dans le graphe de tâches de l'exécution est la séquence de communications suivie par le dernier calcul.

   (a) Ces équations sont vraies pour i_0.

   (b) Montrons que si les équations (6.8) et (6.9) sont vraies pour tout $j \prec i$, alors, elles le sont aussi pour i :

   $t_{com}(i)$: Comme $\text{index}(\text{prev}(i)) = \text{index}(i) - 1$, nous avons par induction

   $$t_{com}(\text{prev}(i)) + \delta_{com} = (\text{index}(i) - 1)\delta_{com}$$

   et l'équation (6.7) implique que

   $$t_{calc}(\text{filsC}(i)) \leq (\text{index}(i) - 2)\delta_{com},$$

   donc, comme $\delta_{calc}(\text{filsC}(i)) \leq \delta_{com}$, remplacer dans l'équation (6.5) conduit à

   $$t_{com}(i) = (\text{index}(i) - 1)\delta_{com}.$$

   $t_{calc}(i)$: Nous avons immédiatement par la relation précédente et l'équation (6.6) que

   $$t_{calc}(i) = \text{index}(i).\delta_{com}.$$

   Donc, les équations (6.8) et (6.9) sont vérifiées pour tout $i \in \mathcal{I}$.

   Nous avons ainsi le temps d'exécution total

   $$t_{calc}(racine) + \delta_{calc}(racine) = n\delta_{com} + \delta_{calc}(racine) = B.$$

2. $\forall i \in \mathcal{I}, \delta_{calc}(i) \geq \delta_{com}$

   Montrons par induction sur les équations (6.3), (6.4), (6.5) et (6.6) que $\forall i \in \mathcal{I}$,

$$t_{com}(i) \leq \sum_{j \prec i} \delta_{calc}(j) \tag{6.10}$$

$$t_{calc}(i) = \delta_{com} + \sum_{j \prec i} \delta_{calc}(j). \tag{6.11}$$

   Ces équations traduisent le fait que le chemin critique dans le graphe de tâches de l'exécution est la séquence de calculs précédée par la première communication.

   (a) Ces équations sont vraies pour i_0.

   (b) Montrons que si les équations (6.10) et (6.11) sont vraies pour tout $j \prec i$, alors, elles le sont aussi pour i :

   $\boldsymbol{t_{com}(i)}$ **:** À partir de l'équation (6.7) nous avons deux cas :
   - si $\text{filsC}(i) \in \mathcal{F}$ alors $t_{calc}(\text{filsC}(i)) + \delta_{calc}(\text{filsC}(i)) = 0$
   - sinon $\text{filsC}(i) \prec \text{prev}(i)$, donc

$$\begin{aligned} t_{calc}(\text{filsC}(i)) + \delta_{calc}(\text{filsC}(i)) &\leq t_{calc}(\text{prev}(i)) \\ &= \sum_{j \prec prev(i)} \delta_{calc}(j) + \delta_{com} \\ &\leq \sum_{j \prec i} \delta_{calc}(j). \end{aligned}$$

   Comme nous avons aussi par induction

$$\begin{aligned} t_{com}(\text{prev}(i)) + \delta_{com} &\leq \sum_{j \prec prev(i)} \delta_{calc}(j) + \delta_{calc}(i) \\ &\leq \sum_{j \prec i} \delta_{calc}(j), \end{aligned}$$

   nous pouvons conclure en remplaçant dans l'équation (6.5) que

$$t_{com}(i) \leq \sum_{j \prec i} \delta_{calc}(j).$$

   $\boldsymbol{t_{calc}(i)}$ **:** Le résultat précédent donne

$$t_{com}(i) + \delta_{com} \leq \sum_{j \prec i} \delta_{calc}(j) + \delta_{com},$$

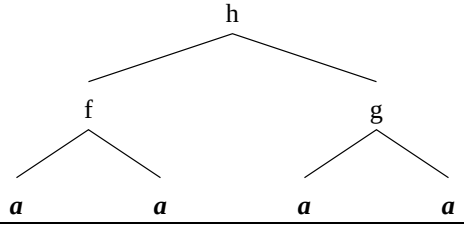
   et par induction

$$t_{calc}(\text{prev}(i)) + \delta_{calc}(\text{prev}(i)) = \delta_{com} + \sum_{j \prec i} \delta_{calc}(j).$$

   Donc,

$$t_{calc}(i) = \delta_{com} + \sum_{j \prec i} \delta_{calc}(j).$$

FIG. **6.10 -** Arbre d'expression avec données partageant une même localisation



Donc, les équations (6.10) et (6.11) sont vérifiées pour tout $i \in \mathcal{I}$.

Nous avons ainsi le temps d'exécution total

$$t_{calc}(racine) + \delta_{calc}(racine) = \delta_{com} + \sum_{i \in \mathcal{I}} \delta_{calc}(i) = B.$$

Donc, dans les deux cas, la borne inférieure B est atteinte, l'ordonnancement est alors optimal. □

En fait, n'importe quel ordre $\prec$ qui vérifie les équations (6.3), (6.4), (6.5), (6.6) et (6.7) donne un temps d'exécution optimal.

Complexité Comme l'heuristique est un parcours d'arbre, sa complexité est linéaire en fonction du nombre de nœuds internes de l'arbre (qui correspond au nombre d'opérateurs de l'expression à évaluer).

6.4.3 Quand des feuilles peuvent partager la même localisation

Cas étudié

Nous nous intéressons ici au cas où les données ne sont pas nécessairement toutes stockées dans des localisations différentes (en termes de processeurs). Mais nous imposons dans ce cas qu'il n'y ait pas de stockage temporaire. C'est-à-dire qu'un tableau stocké en position a peut être déplacé à la localisation b pour un calcul, mais qu'il ne sera pas disponible en b pour un autre calcul. Nous pouvons ainsi représenter l'expression par un arbre-expression [4].

La figure 6.10 présente un exemple d'expression avec des données différentes stockées au même endroit et qui est mal traitée par l'heuristique de la section précédente.

Cet arbre peut correspondre à l'expression $h(f(d_1, d_2), g(d_3, d_4))$ où tous les tableaux de données, d_1, d_2, d_3 et d_4, sont alignés et stockés à la même position a. En prenant tous les temps de calcul égaux à 1 et tous les temps de communication égaux à 10, l'heuristique précédente donne un temps d'exécution total de 31, faisant trois communications inutiles. L'heuristique suivante détecte le nombre minimal de communications et donne ainsi une exécution sans communication d'un temps total égal à 3.

Algorithme pour déterminer le nombre minimal de communications

Nous présentons ici un algorithme qui calcule le nombre minimal de communications nécessaires pour évaluer un arbre-expression. Cet algorithme est légèrement inspiré de celui qui est présenté dans [GS91] mais utilisé dans un contexte totalement différent.

L'algorithme (voir programme 6.4) est construit suivant le paradigme de la programmation dynamique : il est basé sur l'étiquetage de chaque nœud de l'arbre-expression par le nombre minimal

4. voir la section 6.4.1 pour la définition d'un arbre-expression

de communications nécessaires pour calculer le sous-arbre dont le nœud en question est la racine et par la liste des localisations où ce nombre minimal est atteint. Nous noterons cette étiquette (n_c, s_c) où $s_c = \{l_1, l_2, ..., l_k\}$.

PROG. **6.4** Algorithme pour déterminer le nombre minimal de communications nécessaires pour évaluer un arbre-expression

---

```
/* Initialisation */
pour tout l dans feuilles(DAG) faire
    étiquette(l)=(0, {l})
fin pour
pour tout n dans nœuds_internes(DAG) faire
    étiquette(n)=(0, ∅)
fin pour
/* Première Phase */
parcourir le DAG depuis les feuilles jusqu'à la racine
    soit c le nœud courant
    soit (n1, s1)=étiquette(fils_gauche(c))
    soit (n2, s2)=étiquette(fils_droit(c))
    si s1 ∩ s2 ≠ ∅ alors
        étiquette(c)=(n1 + n2, {s1 ∩ s2})
    sinon
        étiquette(c)=(n1 + n2 + 1, {s1 ∪ s2})
    fin si
fin parcourir
/* Deuxième Phase */
soit (nr, sr)=étiquette(racine) choisir une localisation l pour la racine dans sr
étiquette(racine)=(nr, {l})
parcourir les nœuds internes du DAG
        depuis la racine jusqu'aux feuilles
    soit c le nœud courant
    soit (nc, sc)=étiquette(c)
    soit (np, {lp})=étiquette(père(c))
    si lp ∈ sc alors
        choisir une localisation l dans sc
        étiquette(c)=(nc, {l})
    sinon
        étiquette(c)=(nc, sp)
    fin si
fin parcourir
```

Théorème 5 *L'algorithme décrit par le programme 6.4 donne le nombre minimal de communications nécessaires pour calculer l'expression donnée. Il donne aussi une stratégie de placement des résultats intermédiaires qui utilise ce nombre minimal de communications.*

Preuve

1. Après la première phase, le nombre de communications de chaque étiquette représente le nombre minimal de communications pour chaque nœud.

En effet, montrons par induction que l'étiquette représente :

(a) le nombre minimal, m, de communications nécessaires pour calculer le sous-arbre du nœud,

(b) l'ensemble des localisations pour lesquelles ce minimum, m, est atteint

(c) et pour toutes les autres localisations, le nombre minimal de communications est : $m+1$.

- Les trois propriétés, (1a), (1b) et (1c) sont vérifiées pour toutes les feuilles.
- Si elles sont vraies pour le sous-arbre de racine N (N exclu), alors elles sont vraies pour le nœud N. Notons (n_1, s_1) et (n_2, s_2) les étiquettes des fils de N.
  - Si $s_1 \cap s_2 \neq \emptyset$ alors calculer l'opération du nœud N est fait avec le nombre minimal de communications à une localisation dans l'ensemble $s_1 \cap s_2$, ne causant pas plus de communications que $n_1 + n_2$. Pour les localisations dans $(s_1 \cup s_2) \setminus (s_1 \cap s_2)$, une communication supplémentaire est nécessaire et pour les autres localisations, la meilleure stratégie de communication est de calculer à une localisation dans $s_1 \cap s_2$ et de déplacer ensuite le résultat à la localisation considérée. N'importe quelle autre stratégie augmenterait le nombre de communications de 2 au lieu de 1.
  - Si $s_1 \cap s_2 = \emptyset$ alors calculer l'opération du nœud N est fait avec le nombre minimal de communications à une localisation dans l'ensemble $s_1 \cup s_2$, ne causant qu'une seule communication supplémentaire par rapport à $n_1 + n_2$. Toutes les autres stratégies conduiraient à au moins 2 communications de plus.

  Donc les trois propriétés, (1a), (1b) et (1c) sont aussi vraies pour le nœud N.

Donc, par induction, après la première phase, le nombre de communications de chaque étiquette représente le nombre minimal de communications pour chaque nœud.

2. Après la seconde phase, l'étiquette représente une allocation des résultats intermédiaires qui utilise le nombre minimal de communications.

   En effet, par construction, l'étiquette de chaque nœud représente une allocation des résultats intermédiaires de ce nœud qui utilise le nombre minimal de communications pour calculer son sous-DAG.

□

Complexité Comme l'algorithme est une séquence de deux parcours de DAG, son temps d'exécution est linéaire en fonction du nombre de nœuds du DAG.

Heuristique

Nous considérons ici un ensemble d'algorithmes tous basés sur le même schéma. La première supposition est que pour calculer vite, on doit utiliser le moins de communications possible (ce qui n'est pas toujours vrai). C'est pourquoi nous utilisons une allocation des résultats intermédiaires donnée par l'algorithme précédent.

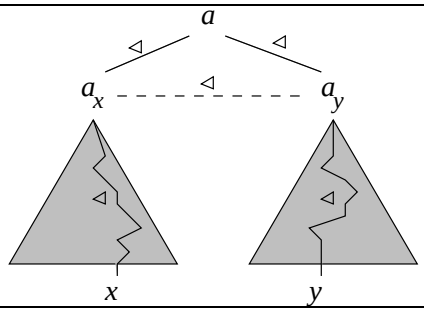
Cette allocation et la structure de l'arbre-expression induisent un ordre partiel $\lhd$ sur les nœuds :

$$i \text{ est un fils de } j \Rightarrow i \lhd j \tag{6.12}$$

$$i \text{ est un frère de } j, i \text{ est communiqué et pas } j \Rightarrow i \lhd j \tag{6.13}$$

$$\lhd \text{ est transitif} \tag{6.14}$$

FIG. **6.11 -** L'ordre partiel $\lhd$



Propriété 2 $\lhd$ *est un ordre partiel.*

Preuve

$$\lhd \text{ est un ordre partiel } \Leftrightarrow \forall x, y \in \mathcal{N}, \neg(x \lhd y \text{ et } y \lhd x)$$

Supposons que nous ayons x et y tels que $x \lhd y$ et $y \lhd x$.

Comme ils sont dans le même arbre-expression, ils ont un ancêtre commun, a. Notons a_x et a_y les fils de a qui sont sur la branche $x \to a$ et $y \to a$ respectivement (voir la figure 6.11).

Il y a alors plusieurs cas :

$\boldsymbol{a = x}$ **ou** $\boldsymbol{a = y}$ alors l'un est ancêtre de l'autre et seule une des relations $x \lhd y$ ou $y \lhd x$ peut être vraie (d'après les équations (6.12) et (6.14)).

$\boldsymbol{a_x = x}$ **et** $\boldsymbol{a_y = y}$ alors seule une des relations $x \lhd y$ et $y \lhd x$ peut être vraie (d'après l'équation (6.13)).

$\boldsymbol{a_x = x}$ **ou** $\boldsymbol{a_y = y}$, supposons $a_x = x$, alors $a_x \lhd y$ et $y \lhd a_x$.

Comme $y \lhd a_y$ (par définition de a_y), par transitivité, $a_x \lhd a_y$.

Nous avons $y \lhd a_x$, donc il existe une liste $y_1, ..., y_k$ de nœuds tels que

$$y \lhd y_1 \lhd ... \lhd y_k \lhd a_x$$

avec toutes les relations dérivées de (6.12) et (6.13). Comme y et a_x sont dans des sous-arbres différents, la seule possibilité de comparer deux nœud de ces sous-arbres avec une relation dérivée seulement de (6.12) ou (6.13) est de comparer a_x et a_y. Donc, $y_k = a_y$ et $a_y \lhd a_x$. Nous sommes dans le deuxième cas présenté ci-dessus avec $x \leftarrow a_x$ et $y \leftarrow a_y$.

$\boldsymbol{a_x \neq x}$ **et** $\boldsymbol{a_y \neq y}$ Comme $y \lhd x$ et $x \lhd a_x$, par (6.14) nous avons $y \lhd a_x$. Le même raisonnement que ci-dessus donne $a_y \lhd a_x$. De manière symétrique, nous avons $a_x \lhd a_y$. Nous sommes dans le deuxième cas présenté ci-dessus avec $x \leftarrow a_x$ et $y \leftarrow a_y$.[5]

Donc, on ne peut pas avoir $x \lhd y$ et $y \lhd x$ et $\lhd$ est bien un ordre partiel.

□

À partir de cet ordre partiel $\lhd$, nous construisons un ordre total $\prec$ sur les nœuds du DAG. Cet ordre total est l'ordre d'exécution des nœuds. L'évaluation d'un nœud est faite en deux étapes : d'abord, communication des données nécessaires au calcul et ensuite évaluation de l'opération du

5. $x \leftarrow a_x$ se lit « x *remplacé* par a_x ».

nœud considéré. Nous considérons les feuilles comme des nœuds internes qui ne nécessitent aucune communication et ont un temps de calcul nul.

Un raffinement de cette stratégie consiste à choisir l'ordre total $\prec$ parmi ceux qui évaluent comme premier nœud interne un nœud qui ne nécessite pas de communication, si possible. Ce qui permet le recouvrement de la première communication.

Il y a beaucoup de liberté dans cet algorithme : on peut construire plusieurs allocations qui donnent le nombre minimal de communications et la complétion linéaire de l'ordre partiel $\lhd$ peut conduire à plusieurs ordres totaux.

Complexité Comme on peut construire un ordre total à partir d'un ordre partiel en un temps linéaire en fonction du nombre d'éléments (en utilisant la complétion linéaire), la complexité de l'heuristique est linéaire en fonction du nombre de nœuds internes du DAG.

Cas optimal Dans le cas des calculs à gros grain, où pour chaque nœud, le temps de calcul est plus grand que le temps de communication, l'heuristique précédente est optimale et donne un temps d'exécution égal à la somme de tous les temps de calcul, $\sum_{i\in\mathcal{I}} \delta_{calc}(i)$, quand la première communication peut être recouverte ; et quand ce n'est pas possible, le temps d'exécution devient $\delta_{com} + \sum_{i\in\mathcal{I}} \delta_{calc}(i)$.

En effet, l'algorithme assure qu'au plus une communication arrive à chaque nœud. La démonstration du théorème 4 est encore valide.

Cependant, dans le cas du grain fin, le fait qu'il puisse n'y avoir aucune communication pour un nœud invalide la démonstration. Voici un exemple : considérons le DAG de l'expression

$$G = f_0(f_1(A, B), f_2(f_3(C, C), C)).$$

Les coûts de communication sont tous égaux à 10 et les coûts de calcul sont :

f_0	f_1	f_2	f_3
5	1	10	10

Voir la figure 6.12 pour différentes allocations possibles des résultats intermédiaires données par l'algorithme. Alors que les exécutions dérivées des allocations (3) et (4) sont optimales, celles qui sont dérivées des allocations (1) et (2) ne le sont pas. Ce contre-exemple prouve que notre heuristique n'est pas toujours optimale quand la granularité est fine.

6.4.4 Extension

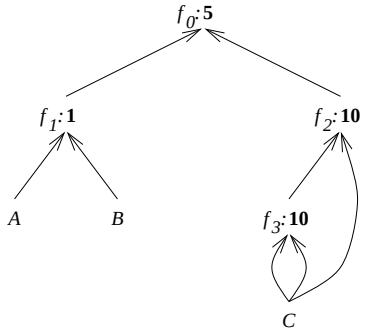
L'heuristique est encore optimale dans un cas un petit peu plus général, avec des coûts de communication variables. En effet quand on fait l'hypothèse que le coût de communication doit être plus faible que tous les coûts de calcul, on n'utilise que le fait que la communication la plus lente est plus rapide que le calcul le plus rapide.

Effectivement, toutes les preuves présentées précédemment sont encore valides dans ce cas légèrement plus étendu.

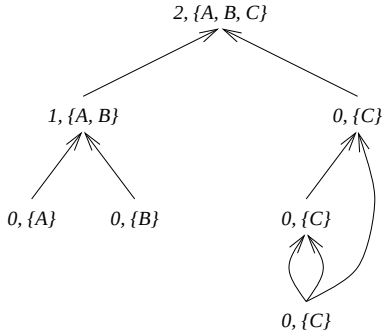
6.5 Conclusion

Nous nous sommes intéressés dans ce chapitre au problème de l'évaluation d'expressions de tableaux du style de celles d'HPF sur des ordinateurs parallèles à mémoire répartie. Nous avons considéré que les machines cibles étaient capables de recouvrir les calculs et les communications. Nous avons d'abord montré qu'une version simple du problème était NP-complète. Ensuite, nous

FIG. **6.12 -** Différentes allocations pour l'expression G



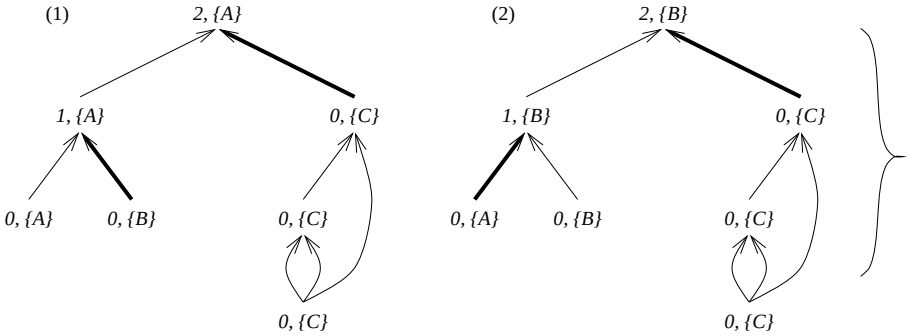
le DAG-expression



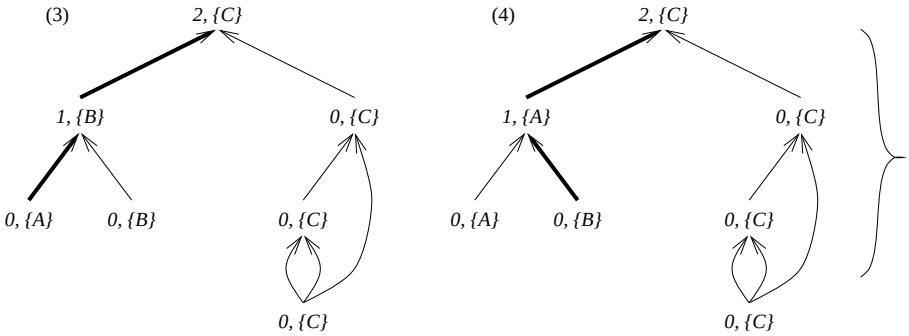
les étiquettes après la première phase

Les différentes allocations :

Diagrammes d'exécution :



deux exécutions non optimales



deux exécutions optimales

avons présenté une famille d'heuristiques qui donnent des résultats optimaux dans des cas particuliers importants en pratique : les calculs à grain fin pour des arbres-expressions avec toutes les feuilles dans des localisations différentes et les calculs à gros grain pour tous les arbres-expressions.

Des travaux futurs pourraient inclure des expérimentations sur des machines réelles afin de raffiner ces heuristiques et une généralisation de ces techniques pour des blocs d'instructions et des programmes complets

Chapitre 7

Conclusions et perspectives

Un des buts de la parallélisation automatique est la réutilisation de programmes séquentiels sur des machines parallèles. Il faut pour ce faire les transformer par un programme, appelé paralléliseur, qui détecte le parallélisme intrinsèque de ces programmes et qui les réécrit dans un langage de programmation parallèle après une suite de transformations.

Bilan

J'ai présenté dans cette thèse un tel paralléliseur : Bouclettes. Bouclettes valide des méthodes théoriques d'analyse et de parallélisation et génère à partir d'un nid de boucles parfait dont les fonctions d'accès aux tableaux sont des translations, un programme parallèle écrit en HPF et qui contient non seulement des boucles parallèles, mais aussi une répartition des données sur les processeurs de la machine cible. Bouclettes sert à la fois de validation, mais aussi de plate-forme d'expérimentation. En effet, les techniques de réécritures en HPF sont complètement originales et Bouclettes est facilement extensible à d'autres méthodes d'analyse de dépendance, d'ordonnancement ou d'allocation.

J'ai aussi présenté deux méthodes, pour l'instant non implémentées, d'optimisation du code parallèle. La première est un type de partitionnement des données applicable à une classe de nids de boucles un peu plus générale que ceux traités par Bouclettes : le pavage par parallélépipèdes. Cette méthode, dont l'optimalité a été démontrée et qui est susceptible d'automatisation, permet l'adaptation de la granularité des calculs à la machine cible, et réduit ainsi le surcoût dû aux communications dans le programme parallèle. La deuxième méthode présentée est une technique d'optimisation de code HPF qui permet d'optimiser le calcul des expressions de tableaux. Après la preuve de la difficulté du problème (il est en effet NP-complet), des heuristiques ont été proposées pour le résoudre le mieux possible.

Perspectives

Bouclettes n'est qu'un premier pas vers un compilateur-paralléliseur général et performant. Il reste encore beaucoup de travail à faire à tous les niveaux. La réalisation de cet outil a soulevé un grand nombre de questions dont entre autres le choix du langage de sortie. HPF a été choisi pour des raisons de simplification du code généré et de disponibilité sur de nombreuses plate-formes. Cependant, il y a un certain nombre de limitations dans ce langage comme l'impossibilité de gérer du parallélisme de contrôle ou une limitation des possibilités de répartition des données sur les processeurs. D'autre part, l'évaluation du code HPF avec ADAPTOR a montré que la parallélisation automatique était une alternative intéressante pour écrire certains algorithmes mais que le compilateur avait des difficultés à optimiser le code produit automatiquement par le paralléliseur.

Une réponse à ces problèmes serait un outil amalgamant le paralléliseur et le compilateur. Cela permettrait d'une part au compilateur de disposer de la phase d'analyse de dépendance du paralléliseur, et ainsi d'avoir plus d'informations sur le motif des communications pour leur optimisation, et d'autre part, le paralléliseur pourrait générer du code de bas niveau quand c'est nécessaire et être moins limité par les contraintes du langage de plus haut niveau.

Je pense qu'un paralléliseur automatique capable de paralléliser une application complète est encore un peu loin, mais qu'un paralléliseur intégré à un compilateur parallèle pour traiter les cas que le programmeur ne veut pas (ou ne sait pas) écrire directement en parallèle est une aide efficace à la programmation des machines parallèles. Il reste encore à construire un langage de haut niveau agréable à utiliser. High Performance Fortran est un bon point de départ. Le système des directives qui sont des conseils que l'utilisateur donne au compilateur est intéressant car un bon compilateur-paralléliseur qui pense qu'il peut faire mieux que ce que lui a indiqué l'utilisateur a la liberté de ne pas tenir compte de ces conseils (en le prévenant éventuellement). Un tel outil pourrait profiter du meilleur des deux mondes : la connaissance de l'algorithme par le programmeur et des méthodes automatiques performantes, pour obtenir le programme parallèle le plus performant possible. L'utilisation de bibliothèques optimisées et standardisées pour les communications et les noyaux de calcul scientifique serait bien entendu une aide considérable.

Un tel outil n'est plus une utopie à l'heure actuelle mais il reste encore beaucoup de travail avant de disposer des compilateurs-paralléliseurs d'un langage de haut niveau qui puissent permettre de tirer un avantage maximum de la puissance potentielle considérable des machines parallèles actuelles.

Bibliographie

[AL93] Anderson (Jennifer M.) et Lam (Monica S.). – Global optimizations for parallelism and locality on scalable parallel machines. *ACM Sigplan Notices*, vol. 28, n° 6, juin 1993, pp. 112–125.

[Ban88] Banerjee (Utpal). – An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, vol. 2, 1988, pp. 133–149.

[Ban+95] Bannerjee (P.) et al. – The paradigme compiler for distributed-memory multicomputers. *IEEE Computers*, Oct 1995, pp. 37–47.

[BCF+93] Bozkus (Z.), Choudhary (A.), Fox (G.), Haupt (T.) et Ranka (S.). – *Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results.* – Technical Report, Syracuse Center for Computational Science, avril 1993.

[Bla90] Blank (T.). – The MasPar MP-1 architecture. *In : Compcon Spring.* pp. 20–24. – IEEE Press.

[Bou95] Boulet (Pierre). – *The Bouclettes loop parallelizer.* – Research Report n° 95-40, Laboratoire de l'Informatique de Parallélisme, Nov 1995.

[BZ94] Brandes (Th.) et Zimmermann (F.). – ADAPTOR - A Transformation Tool for HPF Programs. *In : Programming Environments for Massively Parallel Distributed Systems*, éd. par Decker (K.M.) et Rehmann (R.M.). pp. 91–96. – Birkhäuser.

[Can69] Cannon (L.E.). – *A cellular computer to implement the Kalman filter algorithm.* – Thèse de PhD, Montana State University, 1969.

[CD94] Cosnard (M.) et Desprez (F.). – Machines parallèles actuelles. *In : Algorithmes Parallèles: Analyse et Conception.* – Hermès, 1994.

[CFR93] Collard (Jean-François), Feautrier (Paul) et Risset (Tanguy). – *Construction of do loops from systems of affine constraints.* – Rapport technique n° 93-15, Laboratoire de l'Informatique du Parallélisme, may 1993.

[CGST92] Chatterjee (S.), Gilbert (J. R.), Schreiber (R. S.) et Tseng (S.-H.). – Optimal evaluation of array expressions on massively parallel machines. *In : Proceedings of the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors.* – Boulder, CO, octobre 1992.

[CGST93] Chatterjee (S.), Gilbert (J. R.), Schreiber (R. S.) et Tseng (S.-H.). – Automatic array alignment in data-parallel programs. *In : Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, éd. par Press (ACM), pp. 16–28. – Charleston, South Carolina, janvier 1993.

[Chr89] Chretienne (Ph.). – Task scheduling over distributed memory machines. *In : Parallel and Distributed Algorithms*, éd. par Cosnard (M.), Quinton (P.), Raynal (M.) et Robert (Y.). pp. 165–176. – North Holland.

[Chr90] Christy (P.). – Software to support massively parallel computing on the MasPar MP-1. *In : Compcon Spring*. pp. 29–33. – IEEE Press.

[Col95] Collard (Jean-François). – *Parallélisation automatique des programmes à contrôle dynamique*. – Thèse de PhD, Ecole Normale Supérieure de Lyon, 1995.

[Cri] projet Cristal. – The caml language. – World Wide Web document, URL: `http://pauillac.inria.fr/caml/`.

[DKR91] Darte (Alain), Khachiyan (Leonid) et Robert (Yves). – Linear scheduling is nearly optimal. *Parallel Processing Letters*, vol. 1, n° 2, 1991, pp. 73–81.

[DR93a] Darte (Alain) et Robert (Yves). – Communication-minimal mapping of uniform loop nests onto distributed memory architectures. *In : Application Specific Array Processors 93*, éd. par Dadda (L.) et Wah (B.). pp. 1–14. – IEEE Computer Society Press.

[DR93b] Darte (Alain) et Robert (Yves). – *A Graph-Theoretic Approach to the Alignment Problem*. – Rapport technique n° 93-20, Ecole Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, juillet 1993.

[DR94a] Darte (Alain) et Robert (Yves). – The alignment problem for perfect uniform loop nest: Np-completeness and heuristics. *In : Environments and Tools for Parallel Scientific Computing II*, éd. par Dongarra (J.J.) et eds (B. Tourancheau), pp. 33–42.

[DR94b] Darte (Alain) et Robert (Yves). – Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, vol. 5, n° 8, 1994, pp. 814–822.

[DR94c] Darte (Alain) et Robert (Yves). – Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, vol. 20, 1994, pp. 679–710.

[DRR93] Darte (Alain), Risset (Tanguy) et Robert (Yves). – Loop nest scheduling and transformations. *In : Environments and Tools for Parallel Scientific Computing*, éd. par Dongarra (J.J.) et Tourancheau (B.). pp. 309–332. – North Holland.

[EK93] Esser (R.) et Knetcht (R.). – *Intel Paragon XP/S - Architecture and Software Environment*. – Rapport technique n° KFA-ZAM-IB-9305, Zentralinstitut fur Angewandte Mathematik - Forschungszentrum Julich, avril 1993.

[Fea88] Feautrier (Paul). – Parametric integer programming. *RAIRO Recherche Opérationnelle*, vol. 22, septembre 1988, pp. 243–268.

[Fea92a] Feautrier (Paul). – Some efficient solutions to the affine scheduling problem, part I, one-dimensional time. *Int. J. Parallel Programming*, vol. 21, n° 5, octobre 1992, pp. 313–348.

[Fea92b] Feautrier (Paul). – Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *Int. J. Parallel Programming*, vol. 21, n° 6, décembre 1992, pp. 389–420.

[Fea94] Feautrier (Paul). – Towards automatic distribution. *Parallel Processing Letters*, vol. 4, n° 3, 1994, pp. 233–244.

[For93] FORGE 90. – *xHPF 1.0 Automatic Parallelizer for High Performance Fortran on Distributed Memory Systems - User's Guide.* – Rapport technique, Applied Parallel Research, Inc., avril 1993.

[Fos95] Foster (Ian). – *Designing and building parallel programs.* – Addison-Wesley, 1995.

[FT90] Feautrier (Paul) et Tawbi (Nadia). – *Résolution de Systèmes d'Inéquations Linéaires; mode d'emploi du logiciel PIP.* – Rapport technique n° 90-2, Laboratoire MASI (Paris), Institut Blaise Pascal, janvier 1990.

[GJ91] Garey (Michael R.) et Johnson (Davis S.). – *Computers and Intractability, a Guide to the Theory of NP-Completeness.* – W. H. Freeman and Company, 1991.

[GL89] Golub (Gene H.) et Loan (Charles F. Van). – *Matrix computations.* – Johns Hopkins, 1989, 2 édition.

[GS91] Gilbert (J. R.) et Schreiber (R. S.). – Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, vol. 13, n° 1, septembre 1991, pp. 58–64.

[HKT94] Hiranandani (S.), Kennedy (K.) et Tseng (C.-W.). – Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, vol. 21, avril 1994, pp. 27–45.

[HPF93] High Performance Fortran Forum. – *High Performance Fortran Language Specification.* – Rapport technique, Rice University, janvier 1993.

[HS91] Huang (C.H.) et Sadayappan (P.). – Communication-free hyperplane partitioning of nested loops. *In : Languages and Compilers for Parallel Computing*, éd. par Banerjee, Gelernter, Nicolau et Padua, pp. 186–200. – Springer Verlag, 1991.

[Iri87] Irigoin (François). – *Partitionnement des boucles imbriquées, une technique d'optimisation pour les programmes scientifiques.* – Thèse de PhD, Ecole Nationale Supérieure des Mines de Paris, juin 1987.

[IT88] Irigoin (F.) et Triolet (R.). – Supernode partitioning. *In : Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pp. 319–329. – San Diego, CA, janvier 1988.

[KLS90] Knobe (Kathleen), Lukas (Joan D.) et Steele (Guy L.). – Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, vol. 8, 1990, pp. 102–118.

[KLS+94] Koelbel (Charles H.), Loveman (David B.), Schreiber (Robert S.), Steele (Guy L. Jr.) et Zosel (Mary E.). – *The High Performance Fortran Handbook.* – The MIT Press, 1994.

[KM91] Koelbel (C.) et Mehrotra (P.). – Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, octobre 1991.

[KN93] Knobe (Kathleen) et Natarajan (Venkataraman). – Automatic data allocation to minimize data motion on SIMD machines. *Journal of Supercomputing*, vol. 7, n° 4, décembre 1993, pp. 387–415.

[Kre93] Kremer (Ulrich). – NP-completeness of dynamic remapping. *In : Fourth International Workshop on Compilers for Parallel Computers*, éd. par Sips (H.J.). TU Delft.

[Kun88] Kung (S.Y.). – *VLSI array processors.* – Prentice-Hall, 1988.

[Lam74] Lamport (Leslie). – The parallel execution of DO loops. *Communications of the ACM*, vol. 17, n° 2, février 1974, pp. 83–93.

[LC90] Li (Jingke) et Chen (Marina). – Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. *In : Frontiers 90: The 3rd Symposium on the Frontiers of Massively Parallel Computation.* – College Park, MD, octobre 1990.

[LC91] Li (Jingke) et Chen (Marina). – The data alignment phase in compiling programs for distributed memory machines. *Journal of Parallel and Distributed Computing*, vol. 13, 1991, pp. 213–221.

[Len+] groupe du Pr. Lengauer. – The loopo project. – World Wide Web document, URL: `http://brahms.fmi.uni-passau.de/cl/loopo/index.html`.

[LER92] Lewis (Ted G.) et El-Rewini (Hesham). – *Introduction to Parallel Computing.* – Prentice-Hall, 1992.

[LHS90] Liu (L.S.), Ho (C.W.) et Sheu (J.P.). – On the parallelism of nested for-loops using index shift method. *In : Proceedings of International Conference on Parallel Processing*, pp. 119–123.

[Lis90] Lisper (Bjorn). – Linear programming methods for minimizing execution time of indexed computations. *In : International Worshop on Compilers for Parallel Computers*, éd. par Feautrier (P.) et Irigoin (F.), pp. 131–142. – MASI, PARIS, décembre 1990.

[LK92] Lukas (Joan D.) et Knobe (Kathleen). – Data optimization and its effect on communication costs in MIMD fortran code. *In : Fifth SIAM Conf. Parallel Processing for Scientific Computing*, éd. par Dongarra, Kennedy, Messina, Sorensen et Voigt. pp. 478–483. – SIAM Press.

[LW92] Lam (M.S.) et Wolf (M. E.). – Automatic blocking by a compiler. *In : Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing*, éd. par Dongarra (J.), Kennedy (K.), Messina (P.), Sorensen (D. C.) et Voigt (R. G.), pp. 537–542.

[LW94] Leroy (Xavier) et Weis (Pierre). – *Manuel de Référence du Langage Caml.* – Inter Editions, 1994.

[Mac87] Mace (M. E.). – *Memory Storage Patterns in Parallel Processing.* – Boston, MA, Kluwer Academic Publishers, 1987.

[Mer91] Merlin (J.). – ADAPTing Fortran 90 Array Programs for Distributed Memory Architectures. *In : Proc. 1st International Conference of the Austrian Center for Parallel Computation*. – Salzburg, octobre 1991.

[Nic90] Nickolls (J. R.). – The design of the maspar mp-1: a cost effective massively parallel computer. *In : Compcon Spring*. – IEEE Press.

[O'B92] O'Boyle (Michael). – *Program and Data Transformations for Efficient Execution on Distributed Memory Architectures*. – Thèse de PhD, University of Manchester, janvier 1992.

[OH92] O'Boyle (Michael) et Hedayat (G.A.). – Data alignment: Transformations to reduce communications on distributed memory architectures. *In : Scalable High-performance Computing Conference SHPCC-92*. pp. 366–371. – IEEE Computer Society Press.

[PC89] Peir (J.K.) et Cytron (R.). – Minimum distance: a method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, vol. 38, n° 8, août 1989, pp. 1203–1211.

[Pei86] Peir (J. K.). – *Program partitioning and synchronization on multiprocessor systems*. – Thèse de PhD, University of Illinois at Urbana-Champaign, mars 1986. Report UIUC-DCS-R-86-1259.

[PGH94] PGHPF. – *Reference Manual, User's Guide*. – Rapport technique, The Portland Group, Inc., novembre 1994.

[Pip] équipe PIPS. – Pips (interprocedural parallelizer for scientific programs). – World Wide Web document, URL: `http://www.cri.ensmp.fr/~pips/index.html`.

[Pol88] Polychronopoulos (C.D.). – Compiler optimization for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, vol. 37, n° 8, août 1988, pp. 991–1004.

[Pri] équipe PRiSM SCPDP. – Systematic construction of parallel and distributed programs. – World Wide Web document, URL: `http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html`.

[PS92] Palmer (J.) et Steele (G. L.). – Connection Machine Model CM-5 Overview. *In : The Fourth Symposium on the Frontiers of Massively Parallel Computation*, éd. par Siegel (H.J.). pp. 474–483. – IEEE Computer Society Press.

[Pug+] Pugh (William) et l'équipe Omega. – The omega project. – World Wide Web document, URL: `http://www.cs.umd.edu/projects/omega/index.html`.

[Ram92a] Ramanujam (J.). – A linear algebraic view of loop transformations and their interaction. *In : Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing*, éd. par Dongarra (J.), Kennedy (K.), Messina (P.), Sorensen (D. C.) et Voigt (R. G.), pp. 543–548.

[Ram92b] Ramanujam (J.). – Non-unimodular transformations of nested loops. *In : Proc. Supercomputing'92*. pp. 214–223. – IEEE Computer Society Press.

[RS90] Ramanujam (J.) et Sadayappan (P.). – Tiling of iteration spaces for multicomputers. *In : Proc. Internal Conference on Parallel Processing*, pp. 179–186.

[RS91] Ramanujam (J.) et Sadayappan (P.). – Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distributed Systems*, vol. 2, n° 4, 1991, pp. 472–482.

[SCG] Stanford Compiler Group. – Suif compiler system. – World Wide Web document, URL: `http://suif.stanford.edu/suif/suif.html`.

[Sch86] Schrijver (Alexander). – *Theory of Linear and Integer Programming.* – New York, John Wiley and Sons, 1986.

[SD90] Schreiber (R.) et Dongarra (Jack J.). – *Automatic Blocking of Nested Loops.* – Rapport technique n° 90-38, Knoxville, TN, The University of Tennessee, août 1990.

[SF91] Shang (Weijia) et Fortes (A.B.). – Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, vol. 40, n° 6, juin 1991, pp. 723–742.

[SHS93] Sharma (S.), Huang (C.-H.) et Sadayappan (P.). – On data dependence analysis for compiling programs on distributed-memory machines. *ACM Sigplan Notices*, vol. 28, n° 1, janvier 1993. – Extended Abstract.

[SS93] Sinharoy (B.) et Szymanski (N.). – Finding optimum wavefront of parallel computation. *In : Proc. of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, éd. par El-Rewini (H.), Lewis (T.) et Shriver (B. D.). pp. 225–234. – IEEE Computer Society Press.

[ST91] Sheu (Jang-Ping) et Tai (Tsu-Huei). – Partitioning and mapping nested loops on multiprocessor systms. *IEEE Trans. Parallel Distributed Systems*, vol. 2, n° 4, 1991, pp. 430–439.

[Wol89] Wolfe (M.). – *Optimizing Supercompilers for Supercomputers.* – Cambridge MA, MIT Press, 1989.

[ZBG88] Zima (H.), Bast (H.) et Gerndt (M.). – Superb: A Tool for Semi-Automatic SIMD/MIMD Parallelizatin. *Parallel Computing*, janvier 1988.

[ZC90] Zima (Hans) et Chapman (Barbara). – *Supercompilers for Parallel and Vector Computers.* – ACM Press, 1990.

Publications personnelles

Rapports de recherche

- *Experimental Evaluation of Affine Schedules for Matrix Multiplication on the MasPar Architecture*, rapport de recherche 93-34, en collaboration avec José Fortes.
- *(Pen)-ultimate tiling?*, rapport de recherche 93-36, en collaboration avec Alain Darte, Tanguy Risset et Yves Robert.
- *Evaluating Array Expressions on Massively Parallel Machines with Communication/Computation Overlap*, rapport de recherche 94-10, en collaboration avec Vincent Bouchitte, Alain Darte et Yves Robert.
- *The Bouclettes loop parallelizer*, rapport de recherche 95-40.
- *Code Generation in Bouclettes* , rapport de recherche 95-43, en collaboration avec Michèle Dion.
- *Evaluation of Automatic Parallelization Strategies for HPF Compilers*, rapport de recherche 95-44, en collaboration avec Thomas Brandes

Rapports techniques

- *Reference manual of the Bouclettes parallelizer*, rapport technique, en collaboration avec Michèle Dion, Eric Lequiniou et Tanguy Risset.

Conférences internationales

- *Experimental Evaluation of Affine Schedules for Matrix Multiplication on the MasPar Architecture*, dans Proceedings MPCS'94, pages 452–459, 1994, en collaboration avec José Fortes.
- *(Pen)-ultimate tiling?*, dans SHPCC'94, IEEE Computer Society Press, pages 568–576, 1994, en collaboration avec Alain Darte, Tanguy Risset et Yves Robert.
- *Evaluating Array Expressions on Massively Parallel Machines with Communication/Computation Overlap*, dans Parallel Processing: CONPAR 94-VAPP VI, Springer Verlag LNCS 854, pages 713–724, en collaboration avec Vincent Bouchitte, Alain Darte et Yves Robert.

Revues internationales

- *(Pen)-ultimate tiling?*, dans Integration, the VLSI Journal, volume 17, pages 33–51, 1994, en collaboration avec Alain Darte, Tanguy Risset et Yves Robert.

– *Evaluating Array Expressions on Massively Parallel Machines with Communication/Computation Overlap*, dans The International Journal of Supercomputing Applications and High Performance, volume 9, numéro 3, pages 205–219, 1995, en collaboration avec Vincent Bouchitte, Alain Darte et Yves Robert.

Soumis à publication

– *The Bouclettes loop parallelizer*, soumis à publication à HPCN'96

– *Evaluation of Automatic Parallelization Strategies for HPF Compilers*, soumis à publication à HPCN'96, en collaboration avec Thomas Brandes.

Outils pour la parallélisation automatique

Pierre BOULET

Janvier 1996

Abstract

Automatic parallelization is one of the approaches aimed at a better and easier use of parallel computers. It consists in taking a program written for a *sequential* computer (which has only one processor) and to adapt it to a parallel computer. The interest to have a program, named a *parallelizer*, do this parallelization automatically is that we could reuse all the code already written in Fortran for sequential machines, after parallelization, on parallel machines. We are not there yet, but we are getting closer.

My work fits in this framework. Approximately half of my thesis focuses on the realization of a software that automatically parallelizes a reduced class of programs (the uniform loop nests that use translations as data array accesses) into HPF (High Performance Fortran). I mainly discuss the HPF code generation which is the most innovative part of this program. Besides the realization of Bouclettes, my contribution to the domain is also theoretical with a study of a data partitioning technique called *tiling* and a study of the optimization of the evaluation of array expressions in High Performance Fortran. Tiling is a technique aimed at optimizing the size of the distributed tasks to reduce the communication time overhead. Array expression evaluation is an optimization step of the parallel compiler (the program that translates the parallel code written in a high level language as HPF into directly executable machine code for the parallel computer).

Key-words: automatic parallelization, data parallelism, High Performance Fortran, loop nests, compilation, optimization

Résumé

La parallélisation automatique est une des approches visant à une plus grande facilité d'utilisation des ordinateurs parallèles. La parallélisation consiste à prendre un programme écrit pour une machine *séquentielle* (qui n'a qu'un processeur) et de l'adapter à une machine parallèle. L'intérêt de faire faire cette parallélisation automatiquement par un programme appelé *paralléliseur* est qu'on pourrait alors réutiliser tout le code déjà écrit en Fortran pour machine séquentielles, après parallélisation, sur des machines parallèles. Nous n'y sommes pas encore, mais on s'en approche.

C'est dans ce cadre que se situe mon travail. Une moitié approximativement de ma thèse est consacré à la réalisation d'un logiciel qui parallélise automatiquement une classe réduite de programmes (les nids de boucles uniformes qui utilisent des translations comme accès aux tableaux de données) en HPF (High Performance Fortran). J'insiste surtout sur la partie génération de code HPF, qui est la partie la plus novatrice de ce programme. Outre la réalisation de Bouclettes, ma contribution au domaine est aussi théorique avec une étude sur un partitionnement des données appelé *pavage par des parallélépipèdes* et une étude de l'optimisation des calculs d' « expressions de tableaux » dans le langage High Performance Fortran. Le pavage est une technique permettant d'optimiser la taille des tâches qu'on répartit sur les processeurs pour diminuer le temps passé en communications. L'évaluation d'expressions de tableaux est une étape d'optimisation du compilateur parallèle (le programme qui traduit le code parallèle écrit dans un langage de haut niveau comme HPF en code machine directement exécutable par l'ordinateur parallèle).

Mots-clés : parallélisation automatique, data-parallélisme, High Performance Fortran, nids de boucles, compilation, optimisation