THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 6

Spécialité :
Systèmes Informatiques

présentée

par Monsieur **Alexis PLATONOFF**

pour obtenir le grade de DOCTEUR de l'UNIVERSITÉ PARIS 6

Sujet de la thèse :

Contribution à la Distribution Automatique des Données pour Machines Massivement Parallèles

soutenue le 9 mars 1995 devant le jury composé de :

M.	**Claude**	**GIRAULT**	Président
M.	**Paul**	**FEAUTRIER**	Directeur de thèse
M.	**Serge**	**PETITON**	Rapporteurs
M.	**Yves**	**ROBERT**	
M.	**Benoît**	**de DINECHIN**	Examinateurs
M.	**François**	**IRIGOIN**	
M.	**Gérard**	**MEURANT**	

ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE PARIS

A/268/CRI

Résumé

La mémoire physiquement distribuée et le nombre souvent important de processeurs dans les machines "dites" massivement parallèles obligent les utilisateurs à gérer avec une très grande attention la distribution des données. En effet, sans cela les performances risquent d'être fortement dégradées à cause des délais dus aux mouvements de données entre les processeurs.

Pour cela, les constructeurs de ces machines ont défini des communications de données particulières qui sont effectuées plus rapidement que les communications générales et fourni des primitives de programmation de haut niveau pour les utiliser explicitement. Néanmoins, la détermination de la meilleur distribution, prenant en compte ce type particulier de communication, est un problème complexe.

Pour le résoudre, P. Feautrier propose au sein de sa méthode de parallélisation automatique de programme Fortran de déterminer automatiquement une fonction de placement qui donne le numéro du processeur virtuel qui doit exécuter chaque opération du programme source. Ce calcul est fondé sur la minimisation du volume des communications, sans tenir compte du type de communication.

Nous proposons une extension du calcul de cette fonction de placement qui prenne en compte le type des communications. Cette extension est fondée sur la détection préalable des communications bien optimisées. A partir de ces résultats, nous proposons une génération de code qui utilise les primitives de communication explicite.

Notre méthode a été intégrée au paralléliseur PIPS développé à l'Ecole des Mines de Paris, et nous avons effectué des expérimentations sur la CM-5 et le Cray-T3D.

Abstract

The distributed memory and the important number of processors in the so called massively parallel computers force the users to deal with great care the data distribution. Indeed, without such care performances may be highly degraded because of the delays induced by data movements between the processors.

For this, the constructors of these machines define particular data communications which are done faster than the general ones and give high level programming primitives to use them explicitly. However, the determination of the best distribution, taking into account these kind of data mouvements, is a very complex problem.

To solve it, P. Feautrier proposes in his automatic parallelization method of Fortran programs to determine automatically a placement function that gives the virtual processor executing each operation of the source program. This computation is based on the minimisation of the communication volume, without knowing the type of the communication.

We propose an extension of this computation of the placement function that take into account the type of communication. This extension is based on the detection before hand of the well optimized communications. From the results of this extension, we propose a code generation that uses the explicit communication primitives.

Our method has been implemented in the PIPS parallelizer developped at the Ecole des Mines de Paris, and we have done experimentations on the CM-5 and the Cray-T3D.

Remerciements

Je remercie vivement Claude Girault, Professeur à l'Université P. et M. Curie (Paris VI), pour m'avoir fait l'honneur et le plaisir de présider ce jury. J'ai, grâce à lui, été amené à découvrir l'informatique puis le parallélisme.

J'exprime ma plus sincère gratitude à Paul Feautrier, Professeur à l'Université de Versailles et Saint-Quentin-en-Yvelines, pour avoir dirigé mes travaux. Malgré son emploi du temps très chargé, il a toujours pris le temps de me recevoir et ainsi de suivre l'avancement de mes recherches tout en me suggérant de nouvelles idées. Je lui suis également très reconnaissant d'avoir assuré la liaison avec l'administration de l'université.

Je remercie Yves Robert, Professeur à l'Ecole Normale de Lyon, et Serge Petiton, Professeur à l'Université des Sciences et Techniques de Lille (Lille I), d'avoir accepté d'être rapporteurs de cette thèse. Je leur suis particulièrement reconnaissant pour les remarques et conseils qu'ils m'ont prodigués lors de la lecture de mon rapport de thèse.

Je remercie Benoît de Dinechin, François Irigoin et Gérard Meurant pour avoir accepté d'être membre du jury de cette thèse.

Je remercie profondément Patrick Lascaux, Directeur du Centre d'Etudes de Limeil-Valenton, pour m'avoir fait entrer en contact avec les responsables du Projet Calcul Parallèle (PCP), au sein duquel j'ai passé ces trois années.

Je ne dirais jamais assez ma reconnaissance à Benoît de Dinechin, mon responsable de thèse au CEA, pour tous ses bons conseils et remarques qui m'ont fait progresser dans mon travail, et aussi pour toutes les propositions qu'il m'a faites lors des relectures des versions intermédiaires de ma rédaction.

Je tiens à exprimer tous mes remerciements à Arnauld Leservot, thèsard

au PCP, pour sa bonne humeur et sa curiosité. Tous les développements qu'il a réalisés au sein de PIPS m'ont permis de faire progresser les miens. Je lui suis également reconnaissant pour toutes les heures (jours ?) qu'il a bien voulu consacrer à la relecture de ma rédaction.

Je suis particulièrement reconnaissant envers Monique Patron, ancienne responsable du PCP, pour toute son aide lors de mes expérimentations sur le Cray-T3D.

Je tiens à remercier tous les stagiaires qui ont passé quelques mois (parfois même une année) au PCP et particulièrement Michel Gastaldo pour son soutien dans mon travail et pour sa bonne humeur qui est le meilleur encouragement, Antoine Cloué et François Dumontet pour leur aide au développement au sein du Projet PIPS.

Je remercie l'équipe PIPS du CRI (Ecole des Mines de Paris) pour sa collaboration efficace et amicale. Plus particulièrement, j'aimerais remercier François Irigoin pour m'avoir accepté au sein de son équipe avant la fin de mon travail.

Je remercie aussi Stéphane Woillez, stagiaire de P. Feautrier à l'Université de Versailles, et Tanguy Risset, chercheur à l'IRISA (Rennes) pour leur réussite dans l'installation de PIPS augmenté des extensions faites au PCP sur le site de leur laboratoire de recherche. Cet intérêt pour l'ensemble des travaux accomplis au PCP prouvent que ceux-ci ne resteront pas sans suite.

Enfin, je remercie ma famille, et surtout mon épouse Valérie pour son soutien discret et inquiet, ses encouragements, sa gentillesse et sa compréhension pendant ces trois années.

Introduction

Depuis plusieurs années, un nouveau type d'architecture d'ordinateur, dit **massivement parallèle**, est apparu. Avec les limites intrinsèques des ordinateurs actuels, il semble très probable que l'architecture des futures machines à très grande puissance de calcul comportera un aspect massivement parallèle. Ce nouveau type de machine présente donc un important intérêt économique. De nombreux constructeurs proposent aujourd'hui des systèmes de ce type, par exemple, Cray Research Incorporated avec le Cray-T3D, Thinking Machine Corporation avec la Connection Machine CM-5, ou encore Intel avec le Paragon.

Sur ces machines la mémoire est physiquement distribuée, les communications de données entre les mémoires locales des processeurs doivent donc être gérées avec soin.

L'architecture de ces machines permet l'utilisation de différents types de communication de données : les **communications uniformes** (translations), les **aggrégations** (diffusions, réductions) et les **communications générales**. Ces deux premiers types de communication sont en général bien optimisés sur ces machines, il est donc intéressant de savoir et de pouvoir les utiliser.

Ainsi, les programmeurs sont confrontés à de nouveaux problèmes notamment le placement des données et des tâches sur les processeurs de ces machines, la communication et la synchronisation entre les tâches. Pour les résoudre, les constructeurs ont été amenés à développer des extensions à Fortran qui ont des caractéristiques communes : spécification du placement des données, boucles parallèles, nouvelles fonctions intrinsèques, etc. Néanmoins, des différences existent et elles sont trop importantes pour que les utilisateurs

puissent passer de manière indifférente d'un langage à l'autre. L'utilisation de HPF pourrait résoudre ce problème, mais le résultat de la première version est loin d'être accepté par l'ensemble des constructeurs, et les discussions pour la mise au point d'une seconde version viennent juste de commencer.

Face à ces difficultés, le recours à la parallélisation automatique est donc indispensable. Or, à l'heure actuelle, les problèmes de parallélisation automatique des algorithmes numériques sur les machines massivement parallèles sont loin d'être résolus, ne serait-ce que partiellement.

Notamment, sur ce type de machine, les mauvaises performances sont dues généralement à un nombre très élevé de communications coûteuses entre les processeurs. De plus, la détermination d'une bonne distribution des données et l'utilisation des communications "rapides" sur ces architectures étaient jusqu'à maintenant laissées à l'utilisateur. De nombreux travaux commencent donc à étudier ce problème de la distribution automatique des données.

Il existe des solutions théoriques aux problèmes de parallélisation automatique pour certaines classes de programmes. Par exemple, Feautrier a obtenu des résultats théoriques extrêmement prometteurs, dans le cadre d'une classe de programmes numériques dits "à contrôle statique", et la thèse de Raji-Werth ([RW92]) a étudié leur ciblage sur la Connection Machine CM-2 (ordinateur vectoriel massivement parallèle).

Un des points fondamentaux de la résolution de ces problèmes est l'optimisation au niveau des communications inter-processeurs, ce qui fait intervenir à la fois la distribution des données sur les processeurs et la détection des opérations "intéressantes".

Or, ces optimisations sur les communications dépendent fortement du placement des données sur les processeurs. Pour cela, Feautrier propose le calcul d'une **fonction de placement** qui caractérise complètement la distribution du code et des données d'un programme à contrôle statique sur les processeurs d'une machine à mémoire distribuée. Elle donne l'identité du processeur qui exécute chaque opération élémentaire. Le but de la première version de cette fonction de placement est de minimiser le volume des communications sans tenir compte du type (et donc du coût) de ces communications.

Nous avons donc étudié ces problèmes en prenant en compte les spécificités de l'architecture de ces machines massivement parallèles, notamment la répartition des données sur la mémoire distribuée. La difficulté réside dans le fait qu'il faut gérer cette répartition des données sur tous les processeurs de manière automatique.

Nous proposons une nouvelle méthode de calcul de la fonction de placement qui tienne compte des différents types de communications. Cette méthode nécessite une détection préalable des différents types de communications que nous sommes susceptibles d'engendrer, et une définition des conditions qu'elles imposent à ce calcul.

Nos travaux s'insèrent dans une collaboration entre l'équipe du Professeur Feautrier du laboratoire PRiSM (Université de Versailles et Saint-Quentin), le Centre de Recherche en Informatique (CRI) de l'Ecole des Mines de Paris et le Projet Calcul Parallèle (PCP) du Centre d'Etudes de Limeil-Valenton du Commissariat à l'Energie Atomique (CEA) sous la coordination de la DRET. Cette collaboration vise à adapter et étendre la méthode générale de parallélisation automatique de Feautrier (développée au sein du paralléliseur PAF) au paralléliseur PIPS développé au CRI.

Ainsi, nous avons été amené à étudier les adaptations à effectuer sur la phase de transformation de programme proposée par Feautrier en vue d'une intégration qui tienne compte de cette nouvelle fonction de placement. De plus, nous avons réalisé une génération de code en Fortran parallèle pour la Connection Machine CM-5 et le Cray-T3D. Les deux langages utilisés n'étant pas les mêmes, nous avons dû adapter notre génération de code à chacun d'eux.

A partir des résultats obtenus, nous avons effectué des mesures de performances sur l'exécution de nos programmes et nous les avons comparées avec les programmes non optimisés.

Dans le premier chapitre, nous présentons les différentes architectures et modèles de programmation des machines massivement parallèles.

Le second chapitre présente les bases et les principales méthodes de parallélisation automatique et donne une vue d'ensemble des travaux portant plus précisément sur la distribution automatique des données.

La méthode de parallélisation proposée par Feautrier et son équipe est exposée en détail dans le troisième chapitre ; elle constitue le cadre théorique de notre travail.

Dans le quatrième chapitre, nous présentons notre nouvelle méthode de calcul de la fonction de placement.

Les extensions à la génération de code que nous proposons pour la génération du programme parallèle résultat, ainsi que les résultats obtenus lors d'expérimentations sur CM-5 et Cray-T3D, sont exposées au cinquième chapitre.

Finalement, dans le dernier chapitre, nous concluons sur l'ensemble de notre étude.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Parallélisme massif

1.1 Architectures massivement parallèles

La dénomination *ordinateur massivement parallèle* est assez floue, puisqu'elle s'applique aussi bien à la CM-2 avec ses 65536 PE 1 bit qu'à des machines dont le nombre de processeurs dépasse à peine la centaine. Une mesure objective du degré de parallélisme, appelée $n_{1/2}$, a été introduite par Hockney & Jesshope ([HJ88]) ; elle s'applique dès que la performance (en nombre d'opérations flottantes par seconde) en fonction de la taille n des données traitées peut être représentée approximativement par une courbe d'équation[1] :

$$R = R_\infty \frac{1}{1 + \frac{n_{1/2}}{n}} \tag{1.1}$$

où R_∞ représente la performance asymptotique de la machine et $n_{1/2}$ la taille des données nécessaire pour obtenir la moitié de la performance asymptotique. Une machine sera dite massivement parallèle si son $n_{1/2}$ est supérieur à un millier.

1. La figure 1.1 donne une représentation graphique de cette équation.

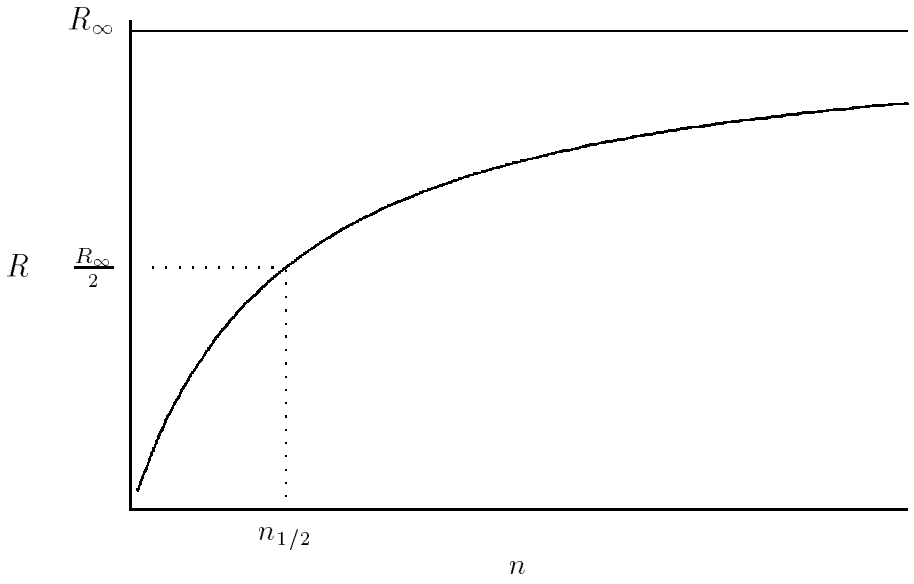


FIG. 1.1 - *Représentation graphique de l'équation 1.1*

Néanmoins, ces machines massivement parallèles ont en commun une architecture possédant les trois caractéristiques suivantes :

1. Un ensemble de nœuds de calcul identiques possédant chacun sa propre mémoire locale (ce sont des machines à mémoire distribuées). L'architecture interne et les fonctionnalités de ces nœuds varient beaucoup suivant les machines. En effet, ces caractéristiques vont de l'élément de calcul (qui exécute des instructions envoyées par un séquenceur) au processeur (qui gère lui-même la mémoire et les instructions). Malgré ces différences, ces nœuds de calcul seront appelés PE (pour "Processing Element") dans la suite de ce document.

2. Un processeur *hôte* qui compile les programmes, exécute le code séquentiel et distribue les tâches aux nœuds de calcul. Ce processeur est soit l'un des nœuds de calcul (PE) pour les machines dites *symétriques* soit un nœud particulier (appelé **frontal**) pour les machines dites *asymétriques.*

3. Un réseau de communications reliant les PE entre eux et assurant la liaison avec le processeur hôte. La topologie du réseau varie d'une ma-

chine à une autre (anneau, grille 2D, grille 3D, hypercube, etc.). Dans la plupart des cas, il existe des mécanismes hardware pour traiter des mouvements de données particuliers, tels que les diffusions et les réductions.

Dans la suite de cette section nous présentons quelques exemples de machines massivement parallèles.

1.1.1 Connection Machine

Le produit le plus récent de la société Thinking Machine Corporation (TMC) est la Connection Machine CM-5. C'est une machine qui allie la simplicité et l'efficacité du SIMD avec la flexibilité du MIMD[2], ce que TMC appelle du *MIMD synchronisé*.

La CM-5 possède de 32 à 16384 nœuds de calculs (noté PN, pour "Processing Node")), chacun composé d'un processeur SPARC (32 MHz), de 32 Mo de mémoire et de quatre unités vectorielles (128 Mflops en 64 bits). L'administration des tâches systèmes et l'exécution des tâches séquentielles sont gérées par les processeurs de contrôle (notés CP, pour "Control Processor"), également des processeurs SPARC. Enfin, les entrées/sorties sont effectuées par les nœuds d'Entrées/Sorties (notés ION, pour "I/O Nodes"). Le système opère sur une ou plusieurs partitions, chacune composée d'un certain nombre de PN et d'un unique CP (il sert de frontal). Chaque utilisateur travaille sur une partition, soit en accès exclusif, soit en accès partagé.

Tous ces composants sont interconnectés par trois réseaux distincts (pour plus de détail voir [LAD+92]) : le réseau de données (DN) (voir figure 1.2) pour des communications point à point à très haute performance, le réseau de contrôle (CN) pour les opérations globales (diffusion, synchronisation et *scan*) et les opérations de gestion du système, et enfin le réseau de diagnostic (DIN) pour tester l'intégrité du système et détecter les erreurs. Ce dernier réseau est invisible à l'utilisateur. Les réseaux de contrôle (CN) et de diagnostique (DIN) ont une architecture en arbre binaire.

2. "Multiple Instruction Multiple Data".

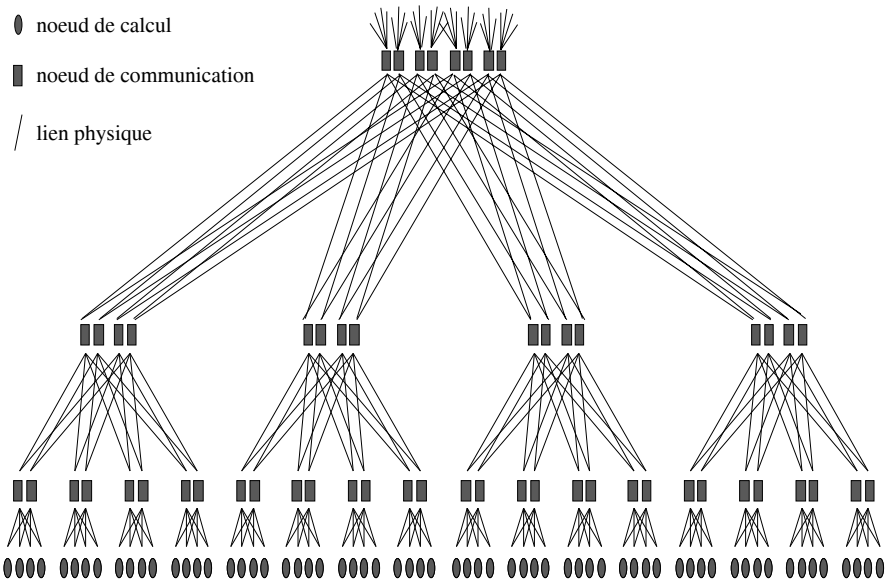


FIG. 1.2 - *Le "fat-tree" quaternaire du réseau de données de la CM-5*

La CM-5 possède une interface réseau qui permet (1) aux processeurs d'avoir une vision simple et uniforme du réseau et vice versa, (2) de gérer le "time-sharing" et le "space-sharing" (lors d'un changement de contexte, si l'utilisateur est en cours de communication, le système suspend les communications tout en garantissant qu'elles reprendront là où elles ont été arrêtées ; ce mécanisme est appelé *All-fall-down*), (3) de découpler les processeurs du réseau, (4) de réaliser des communications sans appels systèmes.

Le réseau de données (DN) a une architecture en *fat-tree* quaternaire (figure 1.2), *i.e.* en arbre quaternaire dont le nombre de canaux de communication croît avec la distance par rapport aux feuilles. Les feuilles sont les processeurs (PN, CP, ION), les nœuds internes sont des *routeurs* (crossbar à 8 liens, à 20 Mo/s en "full duplex" sur chaque canal). Chaque processeur a deux connexions sur le DN. Sur les deux premiers niveaux chaque routeur n'a que deux parents, ensuite quatre. Ainsi, à chaque niveau, le nombre de routeurs par nœud double puis quadruple. Pour une machine de 1024 PN, la racine de l'arbre contient 128 routeurs, ce qui donne un débit croisé de 10 Go/s. L'algorithme de routage remonte les messages jusqu'au premier ancêtre commun en choisissant les liens de manière pseudo-aléatoire, puis les

redescend suivant un algorithme déterministe.

Les buts de cette conception des réseaux d'interconnexion sont (1) d'obtenir des hautes performances, (2) de faire une économie de mécanisme, (3) de permettre des extensions sans limite théorique (*scalability*), (4) de supporter le modèle parallèle sur les données.

Afin d'avoir un ordre de grandeur du gain de performance lorsque des communications optimisées sont utilisées à la place de communications générales, nous avons comparé ces différentes types de mouvement de données sur la CM-5.

Nous nous sommes bornés à comparer les diffusions, les réductions, les translations et les communications générales. A chaque type de mouvement de données nous avons associé un "bout" de programme, tous ces bouts de programme effectuant, en théorie, le même nombre de communications.

Nous donnons dans le tableau 1.1 les résultats de nos mesures sous la forme d'une comparaison des vitesses d'exécution[3] entre quatre types de mouvement de données. Ces mesures ont été effectuées sur une CM-5 possédant 32 nœuds de calcul.

Réduction	Diffusion	Translation	Communication générale
1	1.5	2.5	90

TAB. 1.1 - *Comparaisons entre les différents mouvements de données*

Nous en déduisons que l'utilisation des trois premiers types de communication est beaucoup plus avantageux en terme de performance que le dernier.

De plus, la moins bonne vitesse d'exécution de la translation s'explique par la forme du réseau particulièrement bien adapté pour les réductions et les diffusions.

1.1.2 Kendall Square Research

Cette société propose la KSR-1, un multiprocesseurs à mémoire distribuée. Son architecture varie de 32 à 1088 processeurs RISC 64-bit à 20 MHz,

3. Nous avons fixé la vitesse d'exécution d'une réduction à 1 unité de temps.

ce qui lui donne une performance crête théorique entre 0.32 et 43.5 GFlops. Chacun d'eux est couplé avec une unité de calcul flottant, une unité arithmétique et logique et un processeur de communication.

Le réseau de connexion est une hiérarchie d'anneaux. Au niveau le plus bas, chaque anneau est constitué de 32 processeurs et est relié à un contrôleur. Les niveaux supérieurs sont constitués d'anneaux de 32 contrôleurs eux même reliés aux contrôleurs du niveau supérieur.

Chaque processeurs possède un cache pour les instructions et pour les données ainsi qu'une mémoire locale de 32 Mo gérée comme un cache par page de 16 Ko accédée par blocs de 128 octets. Ainsi, le modèle de programmation utilisé pour cette machine est le modèle à mémoire virtuellement partagée (voir section 1.2).

1.1.3 Intel Paragon

La société Intel propose la machine Paragon XP/S, une réalisation industrielle qui reprend les options du projet de machine massivement parallèle Delta.

Le système de Paragon XP/S est constitué d'un réseau d'interconnexion en forme de grille 2-D. Il existe deux types de nœud de calcul, tous deux constitués à base de processeurs i860 XP :

- "general-purpose" (GP), avec un processeur dédié aux applications et un port d'entrée/sortie,
- "multi-processor" (MP), avec quatre processeurs dédiés aux applications.

Dans les deux cas, chaque nœud de calcul possède en plus un processeur (i860 XP) dédié à la gestion des messages.

Les GP sont utilisés pour les calculs, les entrées/sorties (nœud d'E/S) et l'utilisation interactive (nœud de service). Les MP sont uniquement utilisés pour les calculs.

Le routage des messages est effectué par un système indépendant constitué de MRC ("Mesh Routing Chip") répartis sur la grille 2-D (un par nœud). Chaque MRC possède cinq liens et le débit d'un lien est 200 Mo/s en "full duplex".

Sur le réseau, la première et la dernière colonnes sont constituées de nœuds d'E/S. De même, une colonne est réservée aux nœuds de service. L'existence de ces derniers permet d'éliminer le besoin d'une machine frontale, chaque utilisateur se connectant directement sur la machine.

Le Paragon peut contenir de 71 nœuds (dont 66 de type GP dédiés aux calculs) à 1096 nœuds (dont 1024 de type MP, d'où un nombre total de processeurs i860 XP dédiés aux calculs égal à 4096), pour une puissance crête allant de 5 à 300 Gflops (en 64 bits).

Deux modes de programmation sont possibles :

- Echange explicite de messages avec la bibliothèque standard PVM ([BDG+91]) ou la bibliothèque NX pour la compatibilité avec la famille des iPSC d'Intel.
- Echange implicite de messages avec parallélisme sur les données ou mémoire virtuellement partagée.

1.1.4 IBM SP

Depuis quelques années, IBM a lancé une famille de machines parallèles à hautes performances (appelée "IBM 9076 Scalable POWERParallel Systems"), le SP1 et le SP2 (seconde version qui améliore sensiblement les caractéristiques de la première). Ce sont des machines à mémoire distribuée.

Basée sur la technologie du microprocesseur IBM RISC System/6000, les nœuds de calcul (PE) sont formés de processeurs POWER ou POWER2. Deux types de PE sont disponibles : le "thin" PE et le "wide" PE. Le "thin" PE est un noeud bien adapté pour effectuer des calculs alors que le "wide" PE est généralement configuré pour agir comme un serveur pour le système. Ce système peut posséder de quatre à 128 PE qui sont combinés dans des "frames" (ou grappes de PEs). Chaque "frame" peut compter jusqu'à 16 "thin" PE, 8 "wide" PE, ou un mélange des deux.

Pour la SP2, la configuration la plus grande (128 PE) possède une performance crête de 34 Gflops, une mémoire locale de 256 Go et une mémoire disque de 1024 Go. Ainsi, cette nouvelle version offre aux utilisateurs une puissance de calcul deux fois plus grande que celle du SP1, une mémoire huit fois plus grande et une bande passante (capacité de communication) huit fois plus élevée.

Un routeur haute performance ("High Performance Switch", HPS) est également disponible dans le SP2 au niveau des interconnexions du réseau. Ce HPS peut être relié jusqu'à 16 PE et chaque "frame" en contient un.

Sur ces machines, la programmation avec envois de messages explicites est réalisée soit avec PVM ([BDG+91]) soit avec une bibliothèque de bas-niveau appelée EUI/EUIH.

1.1.5 Cray T3D

La dernière née de Cray Research Inc. est le Cray-T-3D, un système massivement parallèle. Son évolution comporte trois phases, la seconde est prévue pour 1995 avec une performance crête de 1 Tflops ; la troisième phase, prévue pour 1997, vise une performance *soutenue* de 1 Tflops.

Le Cray-T3D est une machine dans laquelle la mémoire est physiquement distribuée. Néanmoins, la programmation avec CRAFT Fortran utilise une mémoire logiquement partagée, *i.e.* chaque PE peut accéder directement à la mémoire locale des autres PEs. Dans ce cas, les accès mémoires sont non uniformes car les PEs accèdent plus rapidement à leur mémoire locale qu'à celle des autres. Sinon, il est possible de programmer avec échange explicite de message en utilisant la bibliothèque de communication PVM ([BDG+91]), ou SHMEM.

Son architecture est constituée de nœuds de calcul connectés par un réseau en forme de tore 3-D. Les nœuds sont constitués de deux processeurs DEC Alpha 21064, chacun ayant une performance maximale de 150 Mflops. La première version de cette machine est prévue pour recevoir jusqu'à 2048 processeurs, ce qui représente une performance crête de 300 Gflops.

Cette machine fonctionne couplée avec un Cray Y-MP qui sert uniquement pour la compilation, le chargement du code et la récupération des résultats.

1.2 Programmation parallèle

Pour les machines massivement parallèles, nous distinguons trois modèles de programmation : le modèle *parallèle sur les données*, le modèle *à échanges de messages* et, dérivant de ce dernier, le modèle *à mémoire virtuellement partagée*. Dans tous ces modèles, au niveau le plus bas de la machine[4], il y a communications des données entre les PE, par échanges de messages sur le réseau de connexion. La différence entre ces modèles provient, non pas du mode d'exploitation du réseau, mais des contraintes de synchronisation.

Le modèle parallèle sur les données : l'ensemble des PEs effectuent simultanément soit un calcul sur des données locales soit une communication avec un autre PE. Tous les PEs sont synchronisés. A un instant donné, ils effectuent tous la même instruction (certains pouvant ne rien faire). Ce modèle correspond au modèle SIMD[5].

Son avantage est la simplicité. Son inconvénient est qu'il ne permet pas d'utiliser toutes les possibilités d'une machine MIMD[6], comme par exemple l'exécution asynchrone d'instructions différentes en parallèle.

Le modèle à échange de messages : chaque PE effectue ses calculs et ses communications (si asynchrones) indépendamment des autres PEs. Dans le cas de communications synchrones, les PEs impliqués se synchronisent entre eux. Ce modèle correspond au modèle SPMD[7]/MIMD.

A l'inverse du modèle précédent, celui-ci permet la pleine utilisation des capacités des machines MIMD. Par contre il est beaucoup plus difficile conceptuellement pour un utilisateur.

4. Rappelons que nous ne considérons que les machines à mémoire distribuée.
5. "Single Instruction Multiple Data".
6. "Multiple Instruction Multiple Data".
7. "Single Program Multiple Data".

Le modèle à mémoire virtuellement partagée : chaque donnée réside physiquement dans la mémoire d'un PE qui en assure la mise à jour et la propagation pour les autres PEs. Ce modèle nécessite du matériel et logiciel système hautement sophistiqués et suppose des communications à très hauts débits. Il correspond au modèle SPMD/MIMD.

Avantage de ce dernier modèle : il permet d'utiliser les outils et les codes pour supercalculateurs à mémoire partagée. Par contre, il n'est pas sûr que ce modèle soit efficace en termes de performances ; en effet, il accroît considérablement le volume des communications.

Pour tous ces modèles se pose un problème très important, l'optimisation des communications. En effet, même asynchrones, le nombre et le type des communications sont deux facteurs qui influencent considérablement les performances, surtout sur les machines possédant des processeurs de calcul très puissants.

Un deuxième problème, fortement couplé au précédent, est le partage du travail sur les PEs. Pour obtenir de bonnes performances sur la plupart des algorithmes, il est crucial que l'utilisateur puisse définir comment l'exécution parallèle de son programme est réalisée.

Pour résoudre ces deux difficultés, les langages de programmation parallèle ont dû être adaptés aux machines massivement parallèles afin que leurs caractéristiques propres soient prises en compte. En particulier, les constructeurs de ces machines ont été amenés à développer leurs propres extensions à Fortran, considéré, depuis sa création dans les années 50, comme le langage privilégié pour la programmation scientifique, que ce soit sur les micro-ordinateurs ou sur les supercalculateurs. L'avenir de ces diverses extensions est un grand sujet d'intérêt comme le montre l'organisation du Forum HPF (pour "High Performance Fortran").

Sur la base du langage séquentiel Fortran 77, les constructeurs définissent des directives de compilation, des fonctions intrinsèques, des structures de contrôle et même des nouvelles syntaxes pour les expressions. Les extensions les plus courantes sont :

– les notations matricielles de Fortran 90 ;

- une instruction "boucle parallèle" (souvent notée "FORALL") ;
- un masque de contrôle parallèle (souvent notée "WHERE") ;
- des directives de placement et de distribution des données ;
- des fonctions intrinsèques pour calculer des réductions (et des diffusions ou des *scans*[8]) et manipuler des tableaux (décalage, transposition, multiplication) ;
- des primitives de synchronisation et de communication entre les PEs.

Toutefois, beaucoup de différences subsistent entre les diverses extensions. Ainsi, le *principal* inconvénient de ces langages est que chaque constructeur est tenté de développer ses propres extensions, d'où un manque de portabilité évident. Pour Fortran, ce problème sera peut-être résolu avec HPF.

Dans la suite de cette section, nous présentons quelques versions de Fortran parallèle pour machines massivement parallèles (pour plus de détails, voir [Pla93]).

1.2.1 CM Fortran

Le langage CM Fortran (CMF, voir [TMC91]) est une extension SIMD de Fortran 77 avec parallélisme sur les données (notations matricielles de Fortran 90). Bien que les deux dernières machines construites par Thinking Machine Corporation (la CM-2 et la CM-5, [TMC90, TMC92]) possèdent des architectures très différentes, CMF est directement utilisable sur chacune.

La CM-5 est une machine asymétrique. Chaque application est associée à une partition de la machine, les données sont donc soit distribuées sur les nœuds de calcul (PNs, *i.e.* les PEs) de la partition, soit allouées sur la mémoire du processeur de contrôle (CP) de la partition (voir section 1.1.1 pour une description de l'architecture de la CM-5).

8. Opérations à préfixe parallèle, récurrences et réductions associatives.

Processeurs virtuels

Un mécanisme de *processeurs virtuels* (VP, pour "Virtual Processors") permet de simuler un nombre arbitrairement grand de processeurs en allouant plus d'un processeur virtuel par PE.

Les VP sont configurés en grilles rectangulaires multidimensionnelles. Pour un tableau donné, le compilateur crée, par défaut, un *ensemble de processeurs virtuels* (VPS, pour "VP Set") de même forme (même nombre de dimensions et même longueur pour chaque dimension)[9] que ce tableau. A l'aide de directives de distribution des données, l'utilisateur peut guider le compilateur pour la construction des VPS et associer deux tableaux (ou plus) au même VPS. Bien entendu, un VPS a une taille (nombre d'éléments) supérieure ou égale à celle de la machine physique et les VP sont distribués uniformément sur les PE.

Distribution

La directive LAYOUT spécifie la forme du VPS sur lequel les tableaux sont placés. Cette spécification est réalisée pour chaque dimension.

Cette directive permet de définir des dimensions séries (`:SERIAL`), dites également "écrasées", pour lesquelles tous les éléments sont placés sur le même VP, ou des dimensions parallèles (`:NEWS`) qui sont distribuées sur les dimensions du VPS.

> Dans cet exemple, certaines dimensions du tableau A sont déclarées séries ce qui permet de ne pas engendrer de communications. En effet, tous les éléments d'une même colonne de A sont contenus dans le même VP (la figure 1.3 montre le placement de ce tableau sur le VPS associé) :

```
      DIMENSION A(4,N)
CMF$  LAYOUT A(:SERIAL, :NEWS)
      A(1,:) = A(2,:) + A(3,:)**2 + SIN(A(4,:))
```

9. Nous appelons **dimension** d'un tableau le nombre d'indices nécessaire pour le décrire.

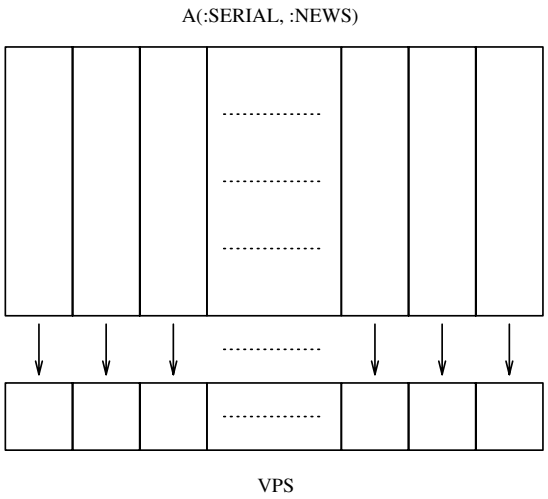


FIG. 1.3 - *Distribution d'un tableau sur les processeurs virtuels*

Alignement

La directive ALIGN permet d'aligner des tableaux de formes distinctes dans le même VPS. L'alignement d'un tableau sur un autre n'est autorisé que si le premier est entièrement contenu dans le second. Ceci implique donc que le plus "petit" des deux soit ajusté sur un VPS de plus grande taille, d'où une perte de mémoire. Cet alignement se fait à l'intérieur ou entre les dimensions :

- Alignement de dimensions de longueurs différentes : il est possible d'aligner une dimension sur une autre en décalant les indices :

```
      DIMENSION C(5), D(20)
CMF$  ALIGN C(I) with D(I+5)
```

  Par contre, le compilateur n'accepte pas l'exemple précédent si $D(I+5)$ est remplacé par $D(2*I+5)$. En effet, cela revient à aligner les éléments de C sur des éléments non contigus de D.

- Alignement de tableaux de dimensions différentes : cet alignement de tableaux ne permet pas les permutations, *i.e.* les indices associés à

chaque dimension doivent apparaître dans le même ordre dans les deux tableaux :

```
      DIMENSION B(N,M), C(N,M,10)
CMF$ ALIGN B(I,J) with C(I,J,5)
      C(:,:,5) = C(:,:,5) * B(:,:)
```

Parallélisme SIMD

Le CMF permet une programmation parallèle sur les données en mode SIMD. Ainsi, l'exécution en parallèle d'une instruction consiste à travailler en parallèle sur les éléments d'un ou de plusieurs tableaux.

Nous définissons une opération sur un tableau comme étant une référence à un objet de type tableau dans une expression, une assignation, une fonction intrinsèque ou un sous-programme. Pour cela, CMF utilise l'extension de Fortran 90 qui permet aux opérateurs et aux fonctions de prendre en argument des objets tableaux et d'opérer sur leurs éléments. Fortran 90 définit une syntaxe, appelée **notation triplet**, pour spécifier une **section de tableau**, *i.e.* une partie ou la totalité des éléments de celui-ci. Cette notation triplet ressemble à la spécification de la boucle **DO** :

borne-inférieure : borne-supérieure : incrément

Lorsque une opération implique deux tableaux (ou plus), les sections de tableau que chaque référence représente doivent être **conformes**, *i.e.* de même dimension et de même longueur sur chaque dimension (si une partie d'un tableau est utilisée, la conformité est nécessaire sur cette partie seulement).

Dans cet exemple, le vecteur A est lu sur ses valeurs d'indices paires comprises entre 10 et 30 (11 valeurs au total), le tableau B (de dimension 2) est écrit sur sa deuxième colonne aux valeurs d'indices comprises entre 90 et 100 (11 valeurs également) ; les sections de A et B sont donc conformes :

```
DIMENSION A(30)
DIMENSION B(100,N)
B(90:100,2) = A(10:30:2)
```

Boucles parallèles

CMF définit une boucle parallèle FORALL. Elle décrit une collection d'assignations sur des éléments de tableaux. Les instructions sont exécutées dans un ordre indéfini, elles sont considérées comme simultanées. Cette instruction n'est pas imbricable et ne peut contenir qu'une seule assignation.

Ainsi, pour réaliser une transposition, le FORALL ne nécessite pas l'utilisation explicite de variables temporaires (le compilateur les génère automatiquement) :

```
FORALL (I=1:N, J=1:N) A(I,J) = A(J,I)*B(I,J)
```

La distribution des itérations d'une boucle parallèle sur les PEs se fait suivant la règle "owner compute", *i.e.* chaque itération est exécutée par le PE possédant la valeur à modifier. Ainsi, dans l'exemple précédent, le PE qui détient la valeur $A(I,J)$ exécute le calcul $A(J,I) * B(I,J)$.

Fonctions intrinsèques

CMF propose toute une famille de fonctions intrinsèques de Fortran 90 pour réaliser les opérations parallèles sur les tableaux. Citons par exemple la transposition, le décalage, la réduction, la diffusion, ou encore les opérations à préfixe parallèle (*scans*[10]). Pour chacune d'elles, le compilateur utilise au mieux les capacités de la machine pour qu'elle soit le plus efficace possible. Ainsi, pour une réduction, les communications sont gérées par un mécanisme global qui utilise la grille multidimensionnelle.

Diffusion : Fortran 90 définie la fonction intrinsèque `SPREAD`, qui copie les valeurs du tableau donné en argument selon une dimension particulière. Pour réaliser une diffusion sur plusieurs dimensions il suffit de faire plusieurs appels à cette fonction.

Une particularité de CMF est que le compilateur génère automatiquement une diffusion sur tous les processeurs lorsque une opération met en jeu une section de tableau et une valeur (un élément de tableau ou une variable

10. Récurrences et réductions associatives.

scalaire). Dans ce cas, qui correspond à ce que nous appelons une **diffusion globale**, aucun appel à la fonction `SPREAD` n'est nécessaire.

La syntaxe de cette fonction est la suivante :

```
SPREAD(SOURCE, DIM, NCOPIES)
```

Cette fonction retourne un tableau de même type que celui passé en argument, et ayant une dimension de plus. Le premier argument est le tableau (ou le scalaire) à copier, les second et troisième arguments sont des entiers.

`DIM` donne le numéro de la dimension du tableau résultat sur laquelle la copie doit être effectuée.

`NCOPIES` donne le nombre de copies à réaliser. Pour que les contraintes de formes soient vérifiées, la longueur de la dimension `DIM` du tableau résultat doit être égale à `NCOPIES`, les autres dimensions étant conformes aux dimensions du tableau `SOURCE`.

> Soit par exemple un tableau A de dimension 1 sur lequel nous souhaitons réaliser la diffusion suivante :
>
> ```
> D = SPREAD(A, DIM=2, NCOPIES=4)
> ```
>
> Les effets d'une telle diffusion sont les suivants :
>
> $$A = \begin{pmatrix} 1 & 5 & -3 \end{pmatrix} \Rightarrow D = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 5 & 5 & 5 & 5 \\ -3 & -3 & -3 & -3 \end{pmatrix}$$

Réduction : Fortran 90 propose plusieurs fonctions intrinsèques de réductions, qui ont un principe d'utilisation identique. Par exemple, la fonction `SUM` réalise une somme, `PRODUCT` un produit, `ALL` un "ET" booléen.

Etudions la syntaxe de la somme : `SUM(ARRAY[, DIM][, MASK])`. Cette fonction retourne la somme des éléments d'un tableau `ARRAY`. Le second argument est un entier, le troisième argument est un tableau booléen de même forme que `ARRAY`. Ils sont tous les deux optionnels.

Si l'argument `DIM` est spécifié alors la somme est effectuée sur la dimension `DIM` du tableau `ARRAY`, le résultat est dans ce cas donné dans un tableau ayant une dimension de moins que `ARRAY`. Sinon le résultat est un scalaire.

Si l'argument `MASK` est spécifié alors le calcul de la somme n'utilise que les éléments pour lesquels ce `MASK` à la valeur "VRAI".

Soit par exemple un tableau A de dimension 2 sur lequel nous souhaitons réaliser la réduction suivante :

```
S = SUM(A, DIM=1, MASK=(A>0))
```

Les effets d'une telle réduction sont les suivants :

$$A = \begin{pmatrix} 1 & 5 & -3 & 7 \\ 4 & 2 & 6 & 3 \\ 9 & 2 & 1 & -5 \end{pmatrix} \Rightarrow S = \begin{pmatrix} 14 & 9 & 7 & 10 \end{pmatrix}$$

Translation : il existe deux fonctions de décalage dimensionel (ou translation). `CSHIFT` réalise un décalage circulaire alors que `EOSHIFT` réalise un décalage simple non circulaire.

Leur syntaxe est similaire, étudions par exemple celle du décalage simple : `EOSHIFT(ARRAY, SHIFT[, BOUNDARY][, DIM])`. Cette fonction retourne un tableau de même forme que `ARRAY`. Le premier argument est le tableau à décaler. Le second argument est de type entier, et est un scalaire ou un tableau ; il donne le ou les pas de décalage. Le troisième argument est un scalaire ou un tableau de même type que `ARRAY` ; il donne la ou les valeurs frontières. Le quatrième argument est un scalaire entier ; il donne la dimension sur laquelle doit être effectué le décalage. Les deux derniers arguments sont optionnels.

Si l'argument `DIM` n'est pas spécifié, la valeur 1 lui est donné par défaut. Le nombre de décalage à effectuer est donné par le nombre de composantes qu'il y aurait dans un tableau de même forme que `ARRAY` avec la dimension `DIM` omise.

Si l'argument `SHIFT` est scalaire, le résultat est obtenu en décalant `SHIFT` fois les éléments de `ARRAY` parallèlement à `DIM`. L'argument `SHIFT` doit être scalaire si la dimension de `ARRAY` est égale à 1.

Si `SHIFT` est un tableau, il doit être de même forme que `ARRAY` avec la dimension `DIM` omise. Dans ce cas, le décalage d'une composante est effectué selon le pas donné par la valeur de `SHIFT` pour cette composante.

Ce type de décalage n'étant pas circulaire, il introduit des "trous" à la frontière. Si l'argument `BOUNDARY` est spécifié et est scalaire, il donne la valeur à mettre dans ces trous. Si c'est un tableau, il doit être de même forme que `ARRAY` avec la dimension `DIM` omise, et il fournit une valeur pour chaque décalage. Si l'argument `BOUNDARY` n'est pas spécifié, la valeur mise par défaut est zéro.

Soit par exemple un tableau A de dimension 2 sur lequel nous souhaitons réaliser le décalage suivant :

```
A = EOSHIFT(A, SHIFT=(1,2,3), BOUNDARY=(10,20,30), DIM=2)
```

Les effets d'un tel décalage sont les suivants :

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \Rightarrow A = \begin{pmatrix} 2 & 3 & 4 & 10 \\ 3 & 4 & 20 & 20 \\ 4 & 30 & 30 & 30 \end{pmatrix}$$

1.2.2 CRAFT Fortran

Le CRAFT Fortran[11] ([PMM93]) est développé par Cray Research Incorporated. C'est une extension MIMD de Fortran 77 pour la nouvelle machine massivement parallèle de Cray Research, le Cray-T3D ([Mel94]). Sur cette machine, il est prévu que soient accessibles deux modes de programmation : le modèle à échanges de messages qui utilise la bibliothèque de fonctions de PVM (voir [BDG+91]) pour exprimer explicitement les communications et synchronisations, et le modèle à mémoire virtuellement partagée, pour lequel un schéma d'adressage efficace a été développé (voir [MPM92]).

Dans toute la suite nous nous intéressons à la distribution des données et des tâches du modèle à mémoire virtuellement partagée. En effet, ce dernier permet une programmation de haut niveau, caractéristique nécessaire pour une utilisation industrielle.

11. Appelé précédemment MPP Fortran, pour "Massively Parallel Processor Fortran".

Le Cray-T3D est une machine symétrique, toutes les données sont placées sur les PE. CRAFT distingue deux types de données :

- les données *privées*, qui sont répliquées sur chaque PE[12],
- les données *partagées*, qui sont distribuées sur tous les PEs.

Par défaut, une variable est déclarée privée. Les données déclarées SHARED sont partagées (voir exemple plus loin).

Distribution

CRAFT permet à l'utilisateur de spécifier le placement des données partagées sur les processeurs physiques (il n'y pas de mécanisme de processeurs virtuels[13]).

La distribution par dimension permet de spécifier le placement de chaque dimension indépendamment des autres. La spécification ":BLOCK" divise la dimension en blocs d'éléments placés sur les PEs (la dimension est dite *parallèle*) ; la spécification ":" écrase tous les points de la dimension sur le même PE (la dimension est dite également *série* ; cette spécification est équivalente à la spécification ":SERIAL" de CMF). Par exemple, avec une distribution A(:,:BLOCK) et pour J fixé, tous les éléments du vecteur A(:,J) (une colonne de A) sont alloués sur le même PE.

Pour chaque tableau de données, le compilateur définit un tableau logique de PEs (de dimension `m`) sur lequel il distribue le tableau de données : `m` est égal au nombre de dimensions parallèles du tableau (*i.e.* le nombre de dimensions qui ne sont pas écrasées).

Ensuite, à chaque dimension est associé un facteur du nombre total de PEs (appelé *facteur de PEs*[14], ce facteur est égal à 1 pour les dimensions écrasées).

En notant N la longueur d'une dimension et P le facteur de PEs qui lui est attribué, nous pouvons distinguer deux types de distribution parallèle pour

12. Avec des valeurs qui peuvent être différentes.
13. Il existe néanmoins une numérotation "virtuelle" des processeurs lorsque la machine est divisée en partitions.
14. Le produit des facteurs est égal au nombre total de PEs.

une dimension :

:BLOCK la dimension est divisée en blocs. La taille M de chaque bloc dépend du nombre de PE alloué à la dimension : $M = \lceil N/P \rceil$. Chaque PE reçoit un bloc.

:BLOCK(M) la dimension est divisée en blocs de taille M. Pour une taille de bloc assez petite, ce type de distribution permet d'allouer plus d'un bloc par PE.

Prenons comme exemple la déclaration des tableaux suivants :

```
      REAL A(1024,1024), B(1024,1024)
CDIR$ SHARED A(:,:BLOCK(128))
CDIR$ SHARED B(:BLOCK(256),:BLOCK)
```

La figure 1.4 donne la distribution par dimension de ces tableaux sur une machine à 4 PEs.

Pour le tableau A, les lignes sont écrasées et les colonnes distribuées par blocs de 128 éléments (les facteurs de PEs sont donc respectivement 1 et 4) ; ainsi, chaque PE reçoit 2 blocs contenant chacun 128 colonnes du tableau.

Pour le tableau B, les deux dimensions sont distribuées par blocs sur les PEs (le facteur de PEs est le même pour les deux dimensions et égal à 2) ; ainsi, chaque PE reçoit 2 blocs contenant chacun un sous-tableau de taille 256×512.

Alignement

La distribution des données est toujours faite à partir du PE 0. Ainsi, l'alignement est limité aux tableaux ayant une distribution parallèle conforme, *i.e.* la longueur des dimensions parallèles et leur nombre sont identiques et la distribution par bloc est la même.

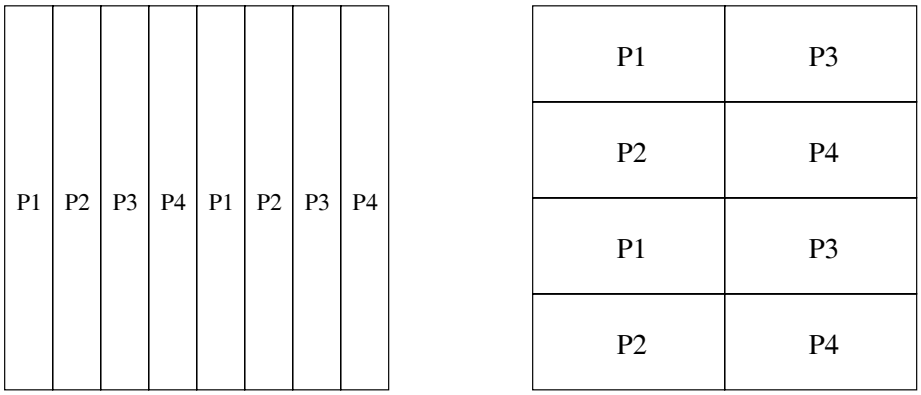


FIG. 1.4 - *Distributions par dimension sur une machine à 4 PEs*

Exécution parallèle

CRAFT est construit principalement autour de la notion du partage du travail. Ainsi, les constructions de ce modèle permettent l'accès à des mécanismes qui distribuent le travail sur différentes tâches. Toutes les tâches sont créées au démarrage du programme et chacune est attachée à un PE spécifique. Pour cela, le code est formé de *sections séquentielles* et de *sections parallèles*.

Initialement, le programme s'exécute dans une section parallèle. Le programme reste dans ce mode d'exécution jusqu'à ce qu'il rencontre la directive qui indique le début d'une section séquentielle. Le programme exécute alors tous les calculs séquentiellement jusqu'à ce qu'il rencontre la directive de fin de section séquentielle.

La seule tâche qui s'exécute dans une section séquentielle est celle du PE 0. Toutes les autres tâches sont arrêtées sur une barrière de synchronisation jusqu'à la fin de la section séquentielle. A cause du nombre de PEs, les sections séquentielles contenant beaucoup de travail s'exécuteront avec des performances très en dessous du potentiel de la machine.

A l'intérieur d'une section parallèle il est possible de définir des boucles *privées* et des boucles *partagées*. Une boucle privée est entièrement exécutée

par la tâche qui l'invoque. La syntaxe et la sémantique des boucles privées sont les mêmes que dans Fortran 77. Par contre, les boucles partagées permettent de distribuer l'exécution des itérations sur tous les PEs.

Boucles parallèles

Définir une boucle comme "partagée" permet de spécifier le comportement de toutes les tâches (contenues dans cette boucle) collectivement et de définir implicitement leur comportement individuel. Pour une telle boucle, l'ordre d'exécution des itérations est quelconque.

Il est possible de mettre des boucles (partagées ou non) à l'intérieur d'une boucle partagée, mais seuls les nids de boucles partagées parfaitement imbriqués seront considérés comme parallèles. Dans le cas contraire, seule la boucle partagée la plus externe est parallèles.

La distribution des itérations sur les tâches (*i.e.* les PEs) est spécifiée par l'utilisateur à l'aide de la clause `ON` qui place les itérations sur le processeur qui possède la référence donnée en argument.

Prenons par exemple le calcul de la transposition d'une matrice. En voici une version en CRAFT avec l'utilisation de boucles partagées:

```
CDIR$ DOSHARED (I,J) ON A(I,J)
      DO I = 1,N
        DO J = 1,M.
          A(I,J) = B(J,I)
        END DO
      END DO
```

Fonctions intrinsèques

CRAFT offre une variété de fonctions intrinsèques de manipulations et calculs sur les tableaux, notamment les réductions, les scans et les scans segmentés.

Les fonctions intrinsèques `SPREAD`, `SUM`, `CSHIFT`, etc., définies par Fortran 90, sont disponibles pour CRAFT (voir section 1.2.1 pour leur description détaillée).

1.2.3 High Performance Fortran

Avec l'apparition de nombreuses extensions à Fortran 77 et Fortran 90 pour la programmation de la nouvelle génération de machines parallèles, la commercialisation de ces machines est actuellement limitée par l'absence d'un langage standard.

A l'initiative de DEC, coordonné par Rice University et dirigé par Ken Kennedy, le forum "High Performance Fortran" (HPFF, voir [WNC92]) est une collaboration entre des représentants des industries, des universités et des laboratoires de recherches du monde entier qui travaillent à définir des extensions à Fortran 90 dans le but de donner aux utilisateurs la possibilité d'utiliser pleinement les caractéristiques des architectures hautes performances tout en maintenant une certaine portabilité entre elles ([Koe94]).

Le résultat de ce projet est supposé produire un standard Fortran portable depuis les stations de travail jusqu'aux machines massivement parallèles. Commencé en 1992, ce forum a édité, par l'intermédiaire de l'Université de Rice, un rapport ([HPF94]) sur les propositions de spécifications d'une première version de ce langage en mai 1993 (version 1.0) et d'une version corrigée et clarifiée en novembre 1994 (version 1.1).

Placement

Pour le placement des données, HPF distingue deux niveaux qui reflètent la distinction entre le parallélisme à grain fin, *i.e.* le placement des tableaux les uns par rapport aux autres et le parallélisme à gros grain, *i.e.* le placement des tableaux sur les processeurs physiques. Pour cela, HPF définit deux domaines de placement (TEMPLATE et PROCESSORS) et deux directives de placement sur ces domaines (ALIGN et DISTRIBUTE). L'utilisateur doit spécifier le placement des données sur ces deux niveaux de parallélisme. La figure 1.5 résume ce schéma de placement des données.

Domaines de placement

Voici la spécification des deux domaines de placement avec les directives TEMPLATE et PROCESSORS.

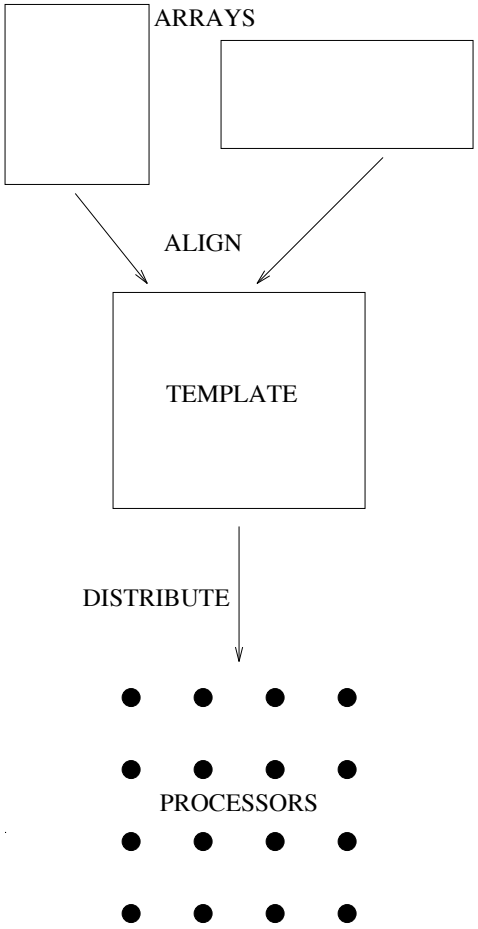


FIG. 1.5 - *Niveaux de distribution des données en HPF*

– TEMPLATE : spécifie le nom et la forme d'un tableau (abstrait) sur lequel des tableaux de données peuvent être alignés.

  Dans l'exemple qui suit, `A` est un template monodimensionnel de taille 100 et `B` est un template bidimensionnel 100×100 (nous utilisons sans les redéfinir ces deux templates dans les exemples de la suite de cette section) :

```
!HPF$ TEMPLATE A(100), B(100,100)
```

– PROCESSORS: déclare un arrangement des PEs, en spécifiant son nom, sa dimension et la longueur de chaque dimension. Le compilateur HPF accepte toutes les déclarations dans lesquelles le produit des dimensions est égal au nombre de processeurs physiques, valeur donnée par une fonction intrinsèque.

  Dans l'exemple suivant, P est un arrangement de `NOP` PEs alignés, Q est un arrangement sur une matrice bi-dimensionnelle `(NS` $\times$ `NOP)` (si `NS` ne divise pas `NOP` alors la déclaration n'est pas valable, cf. ci-desus). Si la forme de l'arrangement n'est pas précisée, elle est par défaut scalaire. Ainsi, l'arrangement SCALARPROC est un vecteur de PEs de longueur égale au nombre de processeurs physiques disponibles (les arrangements SCALARPROC et P sont dit similaires) :

```
      PARAMETER (NOP = NUMBER_OF_PROCESSORS())
      PARAMETER (NS = INT(SQRT(REAL(NOP))))
!HPF$ PROCESSORS P(NOP), Q(NS, NOP/NS), SCALARPROC
```

Directives de placement

Voici la spécification de la distribution des données sur les domaines de placement avec les directives ALIGN et DISTRIBUTE.

– ALIGN: aligne un tableau sur un template. Tous les tableaux placés sur un même template sont automatiquement alignés. Cette proposition

autorise également un alignement directement sur un tableau de taille supérieure ou égale.

L'exemple suivant fait correspondre exactement le tableau sur le template (les templates A et B sont définis plus haut) :

```
      REAL X1(100), X2(100, 100)
!HPF$ ALIGN X1(I) with A(I)
!HPF$ ALIGN X2(I,J) with B(I,J)
```

– DISTRIBUTE: spécifie le placement d'un template sur les processeurs physiques, *i.e.* le placement des tableaux alignés sur ce template. Une distribution explicite d'un template sur un arrangement de PEs se fait à l'aide de la clause ONTO (voir exemple ci-dessous).

Chaque dimension d'un template est distribuée indépendemment des autres. Soit la dimension est divisée en blocs distribués sur tous les PEs associés à cette dimension (clauses BLOCK et CYCLIC), soit la dimension est entièrement placée sur le même PE (clause "*").

Notons N la longueur d'une dimension d'un template et P la longueur de la dimension de l'arrangement de PEs qui lui est associée.

BLOCK(M) divise la dimension en blocs de taille M en assurant que le nombre de blocs n'est pas supérieur au nombre de PEs. Ainsi, un bloc, au plus, est alloué à chaque PE.

CYCLIC(M) divise la dimension en blocs de taille M, mais cette fois le nombre de blocs peut être plus grand que le nombre de PEs. Dans ce cas, il y a un repliage cyclique.

Boucles parallèles

La proposition retenue définit 2 types de boucle FORALL :

– Instruction FORALL équivalente à celle de CMF, *i.e.* une seule assignation et optionnellement une condition (masque).

– Construction FORALL qui permet plusieurs assignations, l'imbrication avec d'autres constructions FORALL et des instructions WHERE. Dans ce cas, le compilateur considère que cette boucle n'est pas parallèle mais **vectorielle**, *i.e.* il peut exister des dépendances entre des instructions différentes du corps de la boucle. Ainsi, l'exécution d'une construction FORALL contenant deux assignations est équivalente à l'exécution successive de deux instructions FORALL contenant chacune une assignation.

Exécution parallèle

Pour permettre davantage de parallélisme MIMD explicite, HPF propose de donner aux utilisateurs la possibilité de signaler au compilateur les boucles dans lesquelles n'apparaissent pas de dépendances ; cette directive, `INDEPENDENT`, peut être composée avec la boucle DO ou la boucle FORALL (cette dernière devient alors une vraie boucle **parallèle**). Cette directive permet à l'utilisateur d'indiquer au compilateur que les instructions de la boucle sont indépendantes.

Dans un programme, cette directive doit immédiatement précéder la boucle `DO` ou `FORALL`. L'exemple suivant (qui est un produit de matrices), dans lequel seule la boucle interne n'est pas indépendante, donne une illustration de l'utilisation de cette directive :

```
!HPF$ INDEPENDENT (I,J)
      DO I=1,L
         DO J=1,N
            DO K=1,M
               C(I,J) = C(I,J) + A(I,K)*B(K ,J)
            ENDDO
         ENDDO
      ENDDO
```

Fonctions intrinsèques

En ce qui concerne les fonctions intrinsèques, HPF retient celles définies par Fortran 90 et en ajoute quelques unes : des nouvelles fonctions de ré-

duction, des fonctions de combinaison, des fonctions à préfixes et suffixes parallèles et des fonctions de tri en parallèle.

1.3 Conclusion

Les trois langages présentés dans ce chapitre possèdent tous les caractéristiques communes données en introduction. Cependant nous avons pu remarquer qu'il y a de nombreuses différences qui ne sont pas toutes justifiées par le modèle d'exécution de la machine qui compile le langage. Nous présentons dans le tableau 1.2 une comparaison des caractéristiques principales de ces langages pour le placement des données et l'exécution parallèle.

	CMF	CRAFT	HPF
Processeurs virtuels	oui	non	possible
Distribution par blocs de données	non	oui	oui
Directive d'alignement	oui	non	oui
Directive de distribution des boucles	non	oui	non fixé

TAB. 1.2 - *Quelques caractéristiques de CMF, CRAFT et HPF*

Les différences entre ces quatre caractéristiques sur les trois langages peuvent s'expliquer de la manière suivante :

1. CMF gère explicitement un mécanisme de processeurs virtuels. Pour sa part, HPF permet de définir un arrangement de processeurs plus grand que le nombre de processeurs physiques, ce qui revient implicitement à définir des processeurs virtuels (la spécification HPF donne le choix au compilateur de permettre ou interdire un tel arrangement). Cette possibilité est nécessaire pour que HPF soit le standard Fortran

pour machines massivement parallèles, notamment pour les Connection Machines CM-2 et CM-5.

2. nous avons pu remarquer que seuls CRAFT et HPF permettent aux utilisateurs de définir une distribution des données par blocs d'éléments. Cette possibilité est peut-être moins utile pour des machines SIMD dont les communications régulières de voisinage sont réalisées efficacement par le réseau (c'est le cas de la CM-2).

3. afin d'optimiser d'avantage le placement des tableaux les uns par rapport aux autres, CMF et HPF proposent une directive d'alignement. L'intérêt de cette directive est de permettre à l'utilisateur de définir explicitement l'alignement des tableaux. En CRAFT, l'alignement est possible mais très simplifié et n'offre pas beaucoup de possibilités aux utilisateurs.

4. la distribution des itérations des boucles est la plupart du temps faite en utilisant la règle "owner compute" (une itération donnée s'exécute sur le proceseur qui possède la donnée à modifier), ce qui est spécifique au modèle SIMD. Néanmoins, laisser à l'utilisateur le soin de choisir la meilleure (en terme de performance) distribution est un problème non trivial. Ainsi, CRAFT propose la clause `ON` qui est utilisée en général pour appliquer une distribution suivant la règle "owner compute". Par contre, cette distribution des boucles n'est pas encore clairement définie en HPF ; une proposition est de retenir le principe de la règle "owner compute".

Le problème pour les concepteurs de HPF est ainsi de définir une norme qui s'adapte à toutes les machines et qui exploite au maximum leurs possibilités. HPF se doit donc de permettre aux utilisateurs de spécifier un grand nombre d'informations parmi lesquelles le compilateur fait le tri en fonction de la machine cible. Par exemple, des trois langages présentés ci-dessus, HPF est celui qui donne le plus de possibilités en ce qui concerne l'alignement et la distribution des tableaux. D'un autre côté, toutes ces spécifications ne sont pas compatibles et peuvent embrouiller l'utilisateur par leur nombre et leur complexité.

En l'état actuel de la recherche sur la parallélisation automatique, HPF est donc promis à une utilisation très facilement portable d'une machine à

une autre. Cependant son efficacité est pour le moment liée aux optimisations propres à chaque machine que les utilisateurs devront spécifier.

Tous ces modèles montrent que les concepteurs de langages parallèles ont la volonté de fournir aux utilisateurs des spécifications pour le placement de données et l'exécution parallèle des boucles. Si les techniques qui sont envisagées ne sont pas concordantes sur la forme, elles visent les mêmes objectifs : distribuer les données afin de pouvoir optimiser les commmunications et expliciter le parallélisme.

Ainsi, notre but final étant de générer un programme parallèle source dans un de ces langages pour une exécution em mode SIMD, nous pouvons en déduire que toutes les étapes intermédiaires de notre étude sont indépendantes du langage cible final. Seule l'étape ultime de génération de code devra être accordée sur le langage cible.

Chapitre 2

Parallélisation automatique

2.1 Dépendances et transformations de programme

Le but de la parallélisation/vectorisation automatique est de pouvoir réutiliser la base de programmes séquentiels existants pour les nouveaux supercalculateurs à architecture parallèle. Le moyen utilisé est de générer, à partir d'un programme source séquentiel, un programme source parallèle qui pourra être compilé sur ces machines. Cette parallélisation peut être réalisée automatiquement, *i.e.* sans intervention extérieure, ou bien de manière interactive, *i.e.* comme une "aide" à la parallélisation manuelle.

Dans tous les cas, cela constitue un problème difficile et les résultats ne sont pas toujours satisfaisants.

2.1.1 Test de dépendance

Un programme est constitué d'instructions qui lisent et écrivent dans des tableaux. Une même instruction peut être exécutée plusieurs fois lorsqu'elle est contenue dans une ou plusieurs boucles. Dans ce cas, chaque exécution de l'instruction sera appelée instance d'instruction ou **opération**. Pour une instruction donnée, chaque opération peut être repérée par les valeurs des indices des boucles qui englobent cette instruction, ce que nous appelons le **vecteur d'itération**. Ainsi, une opération est représentée par son instruction

et son vecteur d'itération.

Etant donné un programme séquentiel, *i.e.* une liste d'opérations bien ordonnée, le problème fondamental est de trouver un programme parallèle équivalent, *i.e* qui donne le même résultat (il est alors dit *déterministe*) et dans lequel il y a le plus grand nombre possible *d'opérations parallèles.*

La solution générale au problème de la construction d'un programme parallèle déterministe est donnée par les conditions de Bernstein ([Ber66]) :

Théorème 1 *Pour qu'un programme parallèle soit déterministe, il suffit que toute paire a et b d'opérations, dont l'ordre d'exécution n'est pas fixé (elles sont alors dites* parallèles*), vérifie les trois conditions suivantes :*

$$\begin{cases} \mathcal{M}(a) \cap \mathcal{M}(b) = \emptyset \\ \mathcal{M}(a) \cap \mathcal{L}(b) = \emptyset \\ \mathcal{L}(a) \cap \mathcal{M}(b) = \emptyset \end{cases}$$

où $\mathcal{M}(x)$ et $\mathcal{L}(x)$ sont les ensembles des cellules mémoires respectivement modifiées et lues par l'opération x.

Définition 1 *Deux opérations sont en* dépendance *si elles ne vérifient pas les conditions de Bernstein.*

Soient a et b deux opérations en dépendance telles que a s'exécute avant b dans le programme séquentiel ; nous avons la classification suivante[1] :

- si $\mathcal{M}(a) \cap \mathcal{L}(b) \neq \emptyset$ alors la dépendance est une **flow-dependence** (ou producteur-consommateur) ;
- si $\mathcal{L}(a) \cap \mathcal{M}(b) \neq \emptyset$ la dépendance est une **anti-dependence** (ou consommateur-producteur) ;
- si $\mathcal{M}(a) \cap \mathcal{M}(b) \neq \emptyset$ alors la dépendance est une **output-dependence** (ou producteur-producteur).

Notre but est de déterminer pour chaque opération de notre programme les opérations avec lesquelles elle est en dépendance.

1. Le cas $\mathcal{L}(a) \cap \mathcal{L}(b) \neq \emptyset$ appelé **input-dependence** (ou consommateur-consommateur) n'est pas considéré comme une dépendance car inverser l'ordre d'exécution de deux opérations différentes lisant la même cellule mémoire ne change pas le résultat du programme.

La complexité de ce calcul apparaît dans les boucles. En effet, pour qu'une boucle soit parallélisable il faut qu'elle vérifie les conditions de Bernstein, *i.e.* qu'aucune itération (d'indice i) ne soit en dépendance avec une autre (d'indice i').

Soient deux références $A(X(i))$ et $B(Y(i'))$ lues ou modifiées dans deux itérations distinctes ($i \neq i'$), où A et B sont des variables (scalaires ou tableaux) et X et Y des vecteurs d'indices (de longueur nulle si la variable est scalaire) ; elles correspondent à la même cellule mémoire si et seulement si[2] :

$$\begin{cases} A \equiv B \\ X(i) = Y(i') \end{cases}$$

L'égalité $X(i) = Y(i')$ est équivalente à un système ayant autant d'équations que les tableaux ont de dimensions, ce qui revient donc à déterminer si un système d'équations à coefficients entiers (appelé *équations aux indices*) a des solutions entières.

En plus des équations aux indices, il faut ajouter les contraintes de bornes de boucle ($lb_i \leq i \leq ub_i$ et $lb_{i'} \leq i' \leq ub_{i'}$) et la contrainte de différenciation d'itération ($i \neq i'$, en fait $i < i'$, par symétrie) ; cela donne donc un système d'équations et d'inéquations à coefficients entiers.

Le problème se généralise au traitement d'un nid de boucles (de dimension n)[3] pour lequel le test se fait indépendamment sur chacune des boucles. Pour tester le parallélisme de la k-ième boucle ($k \leq n$)[4], dans le système à résoudre i et i' deviennent des vecteurs d'itérations et l'équation $i < i'$ est remplacée par :

$$\begin{cases} i[k, \ldots, n] \ll i'[k, \ldots, n] \\ i[1, \ldots, k-1] = i'[1, \ldots, k-1] \end{cases}$$

2. Cette définition ne prend pas en compte l'*aliasing* car $A \equiv B$ est vrai si et seulement si A et B correspondent à la même variable. R. Triolet ([Tri84]) donne une méthode pour traiter l'aliasing dans les programmes Fortran 77.

3. Nous appelons dimension, ou profondeur, d'un nid de boucles le nombre de boucles imbriquées qu'il contient.

4. La numérotation est croissante de la boucle la plus externe vers la boucle la plus interne.

où $\ll$ désigne l'ordre lexicographique.

En développant cet ordre lexicographique et si les fonctions d'indices sont linéaires, cela revient à tester l'existence de solutions entières dans des systèmes d'équations et d'inéquations **linéaires à coefficients entiers**.

La complexité de ce type de problème est NP-complet et les méthodes de résolution classiques telles que la méthode du *simplexe* ([Sch86]) ou la méthode des *coupes de Gomory* ([Gre71]) ne sont pas très bien adaptées.

En fait la complexité au pire cas est exponentielle, alors qu'en moyenne elle est polynomiale (voir les travaux de K.H. Borgwardt [Bor87]). Le problème vient surtout du fait que ces tests sont appelés un très grand nombre de fois[5], d'où la nécessité d'avoir des résolutions rapides.

Ainsi, de nombreuses études traitent le problème spécifique du test de dépendance, dont voici quelques exemples (pour plus de détails voir la thèse de Yi-Quing Yang, [Yan93]) :

Test du GCD (pour "Greatest Common Divisor") ([Ban76, Ban88]) : il s'agit d'un test approximatif qui vérifie l'existence de solutions entières dans un système d'équations linéaires. Ce test est de complexité polynômiale, ce qui fait qu'il est utilisé dans de nombreux paralléliseurs. Néanmoins, il est peu sélectif.

Test de Banerjee ([Ban88]) : ce test est également approximatif, mais il permet de vérifier l'existence de solutions réelles dans un systèmes d'inégalités. Pour cela, il utilise le théorème des valeurs intermédiaires.

Test de Fourier-Motzkin ([DE73, Duf74]) : à l'origine, ce test vérifiait l'existence d'une solution réelle dans un système d'inégalités par éliminations successives des variables. Il a été adapté ([Kuh80, TIF86]) pour tester l'existence de solution entières.

Test PIP ([Fea88]) : ce test utilise un logiciel de programmation linéaire paramétré en nombres entiers appelé PIP (pour "Parametric Integer Programming"). Ce logiciel calcule la solution entière ou rationnelle

5. Pour le programme `gauss` de la figure 3.1 (page 71) trente résolutions de ce type sont effectuées pour calculer le graphe de dépendance exact (appelé DFG) par la méthode exposée section 3.2.

d'un problème de minimum lexicographique d'un système paramétré. Il permet un calcul exact des dépendances. Son algorithme utilise une extension de la méthode des coupes de Gomory basée sur le simplexe.

Test Omega ([Pug90]) : ce test permet également un calcul exact des dépendances. Il se base sur l'élimination de Fourier-Motzkin, mais effectue une résolution en entiers sur le système à traiter. Sa complexité au pire cas est exponentielle, mais W. Pugh montre que dans la plupart des cas elle est polynômiale.

Une grande partie de l'efficacité de la parallélisation automatique repose sur la justesse avec laquelle ce calcul est réalisé ; le choix du ou des tests à utiliser est donc primordial.

2.1.2 Graphe de dépendance

Pour mémoriser l'ensemble des dépendances d'un programme, la structure appropriée est le graphe.

Définition 2 *Le* graphe de dépendance détaillé *(GDD) d'un programme séquentiel est un graphe orienté dont les sommets sont les opérations et les arcs sont des couples* (a, b) *tels que a et b sont en dépendance et a précède b dans l'ordre séquentiel*[6].

Les GDD sont des graphes sans cycles. L'information qu'ils contiennent est la plus complète, mais ce type de graphe prend très vite des proportions gigantesques pour peu que le nombre d'itérations des boucles soit élevé. Dans la pratique, nous utilisons un graphe résumé (ou réduit).

Définition 3 *Le* graphe de dépendance résumé *(GDR) d'un programme séquentiel est un graphe orienté dont les sommets sont les instructions et les arcs représentent les relations de dépendance entre les instructions.*

Ce type de graphe peut contenir des cycles qui indiquent les parties séquentielles du programme. Les méthodes de parallélisation doivent donc impérativement être capables de détecter ces cycles, en utilisant par exemple

6. C'est un ordre total.

l'algorithme de Tarjan qui donne une méthode de calcul des composantes fortement connexes d'un graphe ([Tar72]).

De plus, les arcs sont étiquetés par des informations sur la nature de la dépendance qu'ils représentent. De nombreuses abstractions plus ou moins précises de ces informations sont proposées dans la littérature, en voici quelques unes (pour plus de détail voir [Yan93]) :

Itération de dépendance : cette abstraction donne une énumération exhaustive et exacte de toutes les paires d'itérations entre toutes les paires d'instructions qui sont en dépendance. Le graphe de dépendance proposé par Feautrier (voir section 3.2) utilise cette abstraction.

Vecteur de Distance de Dépendance : cette abstraction représente une dépendance par un vecteur qui donne la distance (*i.e.* la différence) entre les itérations. Lorsque cette distance est constante, la dépendance est dite *uniforme*.

Vecteur de Direction de Dépendance : l'information donnée par cette abstraction est un vecteur composé des signes des composantes du vecteur de distance. Une composante positive sera notée "<", une négative ">" et une nulle "=". Cette abstraction est parmi celles les plus utilisées et est couramment appelée **DDV** (pour "Dependence Direction Vector")..

Profondeur de Dépendance : cette abstraction résume le vecteur de direction de dépendance. Une dépendance est dite de profondeur p si le DDV a ses $p-1$ premières composantes égales à "=" et la p-ième est égale à "<".

Ainsi, toutes les méthodes de parallélisation automatique doivent mettre en place un calcul de dépendance, mémorisé sous la forme d'un graphe. Il reste alors à générer le programme parallèle. Cette génération de code s'accompagne d'une ou plusieurs transformations du programme.

2.1.3 Transformations de programme

Toutes les méthodes de parallélisation/vectorisation utilisent de manière implicite ou explicite des transformations de programme. Celles-ci vont per-

mettre de faciliter l'analyse du programme (normalisation de programme), d'augmenter le parallélisme (suppression de dépendance, adaptation à la machine cible), et de générer le code parallèle (construction des boucles parallèles et des instructions vectorielles).

Etant donné que la majorité du temps d'exécution d'un programme est passée dans les nids de boucle, le but principal est donc d'opérer des transformations sur ces nids de boucle afin d'obtenir le code le plus efficace possible pour la machine ciblée.

Pour réaliser cela, deux problèmes se posent. Le premier est de déterminer à partir du graphe de dépendance quelles transformations sont applicables. Le deuxième est de savoir quelles transformations il est "intéressant" d'appliquer et dans quel ordre.

Le premier point dépend entièrement du calcul de dépendance utilisé par la méthode de parallélisation. Ainsi, certaines méthodes utilisent les mêmes transformations mais leur résultats sont différents car ils n'utilisent pas les mêmes abstractions de dépendance. Par exemple, la méthode de partitionnement *supernœud* de Irigoin et Triolet ([IT88b]) utilise les transformations proposées dans la méthode *hyperplane* ([Lam74]) mais elle utilise le *cône de dépendance* ([IT88a]) qui est plus précis que le *vecteur de direction de dépendance* (voir [Yan93]).

Le second point constitue la partie la plus complexe et la plus fondamentale. En effet, il faut non seulement choisir les transformations à appliquer, mais également l'ordre dans lequel cela doit être fait, car ce ne sont pas des opérations commutatives[7]. Ce choix est nécessaire car essayer "naïvement" toutes les possibilités conduirait à une explosion combinatoire.

7. Cette non commutativité s'explique par le fait que, comme nous le verrons loin, appliquer une transformation revient à effectuer une multiplication de matrices, ce qui n'est pas commutatif.

Transformations classiques

De nombreuses transformations sont proposées dans la littérature. Voici une description des transformations les plus courament utilisées (pour plus de détail voir [Wol89, Yan93, ZC90]) :

décalage de boucle [8] : cette transformation opère un décalage de l'espace d'itération d'un nid de boucle par rapport à l'un des indices. Associée à d'autres transformations, elle donne des méthodes de parallélisation (voir l'exemple donné ci-dessous).

distribution de boucle : cette transformation distribue le contrôle d'une boucle sur des groupes d'instructions de son corps. Une distribution de boucle est légale (*i.e.* conserve les dépendances) si ces groupes d'instructions correspondent aux composantes fortement connexes du graphe de dépendance. Ce type de transformation permet la vectorisation et la parallélisation partielle d'un nid de boucle.

échange de boucle [9] : cette transformation intervertit deux boucles dans un nid. Elle permet de vectoriser la boucle la plus interne (pour les machines ayant des instructions vectorielles), de placer les boucles parallèles à l'extérieur (ce que préfèrent certaines machines), d'augmenter le nombre de tableaux accédés par pas de 1 (pour les machines ayant de meilleures performances dans ce cas), et d'augmenter le nombre d'opérandes invariants dans la boucle la plus interne (*i.e.* augmente la localité).

inversion de boucle [10] : cette transformation inverse l'ordre d'exécution des itérations d'une boucle. Associée à d'autres transformations, elle donne des méthodes de parallélisation.

"strip-mining" : cette transformation divise une boucle en bande en la remplaçant par deux autres boucles. Elle permet de s'accorder à l'architecture de la machine cible (*e.g.*, longueur des opérations vectorielles).

8. Loop skewing.
9. Loop interchange.
10. Loop reversal.

fusion de boucle : cette transformation rassemble plusieurs boucles en une seule. C'est l'inverse de la distribution de boucle. Elle ne permet pas d'obtenir davantage de parallélisme. Par contre, elle réduit l'overhead de boucle (argument traditionnel), le trafic des pages dans les mémoires hiérarchiques (pour les machines à mémoire virtuelle), le trafic des registres mémoire (pour les machines à registre vectoriel), et les besoins totaux de mémoire (dans certains cas).

A chaque transformation est associée une méthode systématique pour son application ; les paralléliseurs qui souhaitent appliquer des transformations sur leurs programmes doivent donc implanter chaque méthode.

Etudions l'application de deux transformations sur le nid de boucle de la figure 2.1(a). Chaque boucle porte une dépendance (les DDVs sont $(=,<)$ et $(<,=)$), aucune ne peut donc être parallélisée.

Nous pouvons décaler d'un facteur 1 la boucle sur j par rapport à i : $j' = j + i$, figure 2.1(b).

Ensuite, nous échangeons les deux boucles, ce qui permet de paralléliser la boucle sur i, figure 2.1(c).

```
do i = 2, n
  do j = 2, n
    a(i,j) = a(i-1,j) + a(i,j-1)
  enddo
enddo
```

(a)

```
do i = 2, n
  do j' = i+2, i+n
    a(i,j'-i) = a(i-1,j'-i) + a(i,j'-i-1)
  enddo
enddo
```

(b)

```
do j' = 4, 2n
  doall i = max(2, j-n), min(n, j-2)
    a(i,j'-i) = a(i-1,j'-i) + a(i,j'-i-1)
  enddoall
enddo
```

(c)

FIG. 2.1 - *Transformation d'un nid de boucle*

Transformation unimodulaire

Pour formaliser simplement la transformation d'un nid de boucle, le concept de transformation unimodulaire a été introduit ([Iri87, Ban91]). Une telle transformation est représentée sous la forme d'une matrice unimodulaire qui donne l'expression des nouveaux indices de boucle en fonction des anciens :

Définition 4 *Une matrice carrée est* unimodulaire *si et seulement si tous ses coefficients sont entiers et son déterminant est égal à* ± 1.

Parmis les transformations "classiques" de nid de boucle, il s'avère que la plupart sont des transformations unimodulaires. Ainsi, l'échange de boucles, l'inversion de boucles ou encore le décalage de boucles sont des transformations unimodulaires. Pour appliquer successivement plusieurs de ces transformations, il suffit de multiplier les matrices correspondantes puis de faire la transformation proprement dite.

Dans le cas d'une transformation unimodulaire quelconque, le problème majeur est de déterminer les nouvelles bornes de boucles ; les travaux de Ancourt et Irigoin ([AI91]) proposent une méthode pour reconstruire le nid de boucle. Utiliser un tel formalisme permet donc au paralléliseur de posséder une méthode générale d'application d'une transformation.

> Revenons à notre exemple précédent, figure 2.1. Ce décalage de boucle et cet échange de boucle peuvent s'exprimer sous la forme de deux matrices U_d et U_e unimodulaires telles que :
>
> $$U_d = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, U_e = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
>
> Appliquer successivement ces deux transformations consiste à appliquer une transformation unimodulaire de matrice $U = U_d.U_e$, *i.e.* construire un nouveau nid de boucle d'indices (i', j') tels que :
>
> $$\begin{pmatrix} i' \\ j' \end{pmatrix} = U.\begin{pmatrix} i \\ j \end{pmatrix} \Leftrightarrow \begin{cases} i' = j \\ j' = i + j \end{cases}$$

Se restreindre au cas unimodulaire n'est pas forcément optimal et des travaux relâchant cette restriction commencent à apparaître, comme par exemple ceux de Barnett & Lengauer ([BL92]). En effet, dans le cas général, toutes les transformations légales ne sont pas forcément unimodulaires ; se restreindre à une transformation unimodulaire peut alors être pénalisant.

De plus, J. Xue ([Xue94]) vient de démontrer que la reconstruction d'un nid de boucles à partir d'une transformation non unimodulaire n'est pas aussi complexe que cela pouvait paraître. En effet, dans son algorithme, la seule étape de complexité non polynômiale est celle qui calcule les nouvelles bornes de boucle, étape également nécessaire dans le cas unimodulaire.

2.1.4 Parallélisation d'un nid de boucles

Voici quelques exemples de méthodes de parallélisation d'un nid de boucles.

Une des méthodes les plus populaires est celle de Kennedy & Allen (voir [AK87]). Initialement, elle est utilisée pour vectoriser un nid de boucles, puis elle a été étendue pour la parallélisation. Cette méthode est fondée sur trois transformations de base : réordonnancement d'instruction (échange l'ordre d'exécution de deux instructions), distribution de boucles et vectorisation d'instruction. Elle utilise un graphe de dépendance (GDR) étiqueté avec des DDVs et a été implantée dans un grand nombre de paralléliseurs.

> Etudions l'exemple du nid de boucles de la figure 2.2(a). Ce nid contient deux dépendances dont les DDVs sont $(=,=)$ et $(=,<)$. La boucle externe ne porte donc pas dépendance, elle est parallèle. L'application de l'algorithme de K&A parallélise la boucle interne en la distribuant, figure 2.2(b).

Citée précédemment, la méthode hyperplane ([Lam74]) est utilisée pour paralléliser un nid de boucles parfaitement imbriquées. Elle découpe l'espace d'itération en hyperplans parallèles, exécutés séquentiellement. Elle est équivalente à trois transformations unimodulaires : permutation de boucles,

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

(a)

```
doall i = 1, n
  doall j = 1, n
    a(i,j) = b(i,j)
  enddoall
  doall j = 1, n
    b(i,j) = b(i,j) + a(i,j-1)
  enddoall
enddoall
```

(b)

FIG. 2.2 - *Parallélisation d'un nid de boucle*

inversion de boucles et décalage de boucles. Cette méthode permet de paralléliser des nids de boucles qui ne le sont pas par l'algorithme de K&A.

> La transformation appliquée sur l'exemple de la figure 2.1 est équivalente à l'application de la méthode hyperplane pour paralléliser ce même nid de boucles. La méthode de K&A ne parallélise aucune des deux boucles.

La méthode *cycle shrinking* ([Pol88]) propose de partitionner l'espace d'itération en blocs rectangulaires, chaque boucle étant divisée en deux. Cette méthode utilise deux transformations de base : "strip-mining" et échange de boucles (cf. ci-dessus, section 2.1.3).

Kumar, Kulkarni, Basu & Paulraj ([KKBP91, KKB92]) proposent des transformations de boucles ayant des dépendances uniformes afin de les appliquer à des machines parallèles hiérarchiques à mémoire distribuée. Ces transformations sont unimodulaires et leurs choix dépend de trois facteurs : le parallélisme (le nombre d'itérations dans la boucle la plus externe après la transformation), l'équilibre des charges (variation du nombre d'itérations des boucles internes), le volume de communication.

Dans le domaine du systolique, nous pouvons citer les travaux de Clauss, Mongenet & Perrin ([MCP91, CMP92]) qui définissent un ordonnancement explicite du programme. Ils proposent de résoudre le problème de la minimisation du nombre de processeurs utilisés et leur allocation pour les algorithmes systoliques. Ce problème est formalisé avec des SAREs (pour "Systems of

Affine Recurrence Equations") définis sur des polyèdres convexes. Ils déterminent une *fonction de temps*, affine et monodimensionnelle, et une *fonction d'allocation*, projection selon une direction.

Enfin, inspirée de la méthode hyperplane et du systolique, la méthode de P. Feautrier ([RWF91]) propose une transformation de programme fondée sur le calcul d'un graphe de dépendance exact et d'une base de temps, affine par morceaux et multidimensionnelle, qui donne la date d'exécution de chaque opération. Cette transformation n'est pas forcément unimodulaire, J.-F. Collard propose une méthode pour reconstruire le programme ([Col93]).

Dans ces deux dernières méthodes apparaissent des études sur la distribution des données et des calculs. Cela ne fait que quelques années que de nombreux travaux ont été entrepris dans ce domaine pour les machines à mémoire distribuée. En effet, la plupart des projets de parallélisation automatique ne pouvaient pas traiter ce problème au niveau de la compilation car il est reconnu comme très complexe.

2.2 Distribution automatique

La phase de distribution des données d'un paralléliseur automatique est une étape cruciale. En effet, c'est elle qui détermine le nombre, le volume et le type de communication que notre programme aura à effectuer lors de son exécution, et (dans la plupart des cas) la distribution des calculs et donc l'équilibre des charges ("load balancing").

Ce problème se définit comme le placement des données et des opérations sur les processeurs physiques afin que l'exécution de notre programme soit la plus efficace possible en terme de performance. Ceci signifie que l'on doit satisfaire deux problèmes contradictoires : distribuer au maximum les calculs et réduire au minimum les communications.

Nous distinguons deux types de problème dans la distribution des données : l'alignement et le partitionnement.

Mace ([Mac87]) a montré que trouver le schéma optimal de stockage des données pour le calcul parallèle est NP-complet, même pour des tableaux de une ou deux dimensions. Dans le cas d'un nid de deux boucles, Li & Chen ([LC90]) ont démontré que le problème de trouver l'alignement du domaine d'indice des tableaux optimal est NP-complet. Darte & Robert ([DR93a]) ont généralisé ce résultat au cas d'un nid de boucles parfaitement imbriquées de profondeur quelconque ; même sous la condition que toutes les fonctions d'accès aux tableaux soient des translations, ils démontrent que le problème reste NP-complet. Ainsi, pour résoudre ces problèmes de nombreuses heuristiques ont été proposées dans la littérature.

2.2.1 Alignement

Ce problème consiste à déterminer la distribution relative des tableaux les uns par rapport aux autres. En pratique, ceci est réalisé par un alignement des tableaux sur une *architecture commune*[11].

Les extensions parallèles de Fortran telles que CM Fortran et HPF proposent une directive d'alignement appelée `ALIGN` (voir les sections 1.2.1 et 1.2.3).

Li & Chen ([LC90]) proposent de construire un *graphe d'affinité* (CAG, pour "Component Affinity Graph") dans lequel les nœuds représentent les différentes dimensions des tableaux, puis réaliser une partition de ce graphe en minimisant le poids des arcs reliant les différentes parties. Trouver un partitionnement du graphe consiste en fait à résoudre le problème d'alignement des tableaux.

Gupta & Banerjee ([GB92]) étendent cette méthode en proposant une distribution des données qui définit des contraintes à satisfaire. La solution de l'algorithme de Li & Chen est que chaque partition contient les dimensions des tableaux qui doivent être alignées, *i.e.*, chaque partition est associée à une dimension de la grille de processeurs virtuels.

11. Cette architecture commune est une grille de processeurs virtuels ou physiques. Dans le premier cas, le problème de la distribution de l'architecture virtuelle sur la machine physique est un problème de partitionnement.

Ils utilisent deux types de contrainte (ce ne sont pas des exigences impératives, il faut seulement essayer de les satisfaire) : des contraintes de parallélisme sur les tableaux contenus dans les boucles parallèles (répartition équitable des charges), et des contraintes de communication pour les tableaux utilisés dans la même instruction (minimisation du nombre de communications).

A chaque contrainte est associée une mesure de qualité ([GB91]) qui correspond soit à une pénalité sur le temps d'exécution quand la contrainte peut ne pas être satisfaite, soit au temps réel d'exécution lorsque la contrainte spécifie la distribution de la dimension d'un tableau sur un certain nombre de processeurs.

Leurs travaux sont implantés dans le projet PARAFRASE-2 et des expérimentations sont effectuées sur iPSC/2.

Avec une méthodologie différente, la méthode de Knobe, Lukas & Steele ([KLS90]) recherche également à aligner les tableaux. Ils proposent un algorithme qui propose de réduire les mouvements de données dans les programmes parallèles sur les données. Pour cela, ils définissent le *graphe de préférence* dont les nœuds sont les références aux tableaux et les arcs sont les alignements souhaités entre ces références.

Deux types d'alignement sont considérés, la *préférence d'identité* qui met en jeu deux références au même tableau, et la *préférence de conformité* qui met en jeu deux tableaux différents lus ou écrits dans la même instruction. Un cycle dans ce graphe indique un conflit d'alignement, qui est traité soit en écrasant la dimension sur laquelle porte ce conflit, soit en effectuant une redistribution des données, *i.e.* ils définissent le mouvement de données à engendrer.

Un aspect intéressant de leur travail est qu'ils n'utilisent pas la règle "owner compute", ils proposent au contraire de changer le propriétaire des données au cours de l'exécution. Ceci permet donc distinguer le placement des données de celui des calculs ; en revanche, le problème devient plus complexe puisque cela introduit de nouvelles inconnues. Leur but est de maximiser le parallélisme, de minimiser le volume de communications et d'optimiser l'utilisation mémoire.

Cette méthode vise la compilation pour machine SIMD de programmes

écrit en Fortran 77 dans lesquels apparaissent des notations Fortran 90 sur les tableaux.

Chapman & Fahringer ([CF93]) proposent de générer automatiquement la distribution des données d'un programme écrit en Vienna Fortran. Pour cela, ils utilisent un outil de "profiling" ("Weight Finder") qui donne des mesures de performances (des poids) des différentes parties d'un programme (boucles, procédures, etc.) : temps d'exécution de chaque instruction, nombre d'itérations de chaque boucle, nombre d'exécutions de chaque instruction, et ratios sur la valeur des tests.

Leur méthode applique tout d'abord une phase intraprocédurale qui donne des informations sur l'alignement des tableaux locaux à une procédure et qui effectue une reconnaissance de forme des accès aux tableaux. Puis, une étape interprocédurale permet de déterminer la distribution de chaque tableau du programme. Cette dernière étape est "bottom-up", elle associe à chaque boucle une fonction de coût qui donne le temps d'exécution en fonction de la distribution et combine ces fonctions de coût et les poids des appels de procédure pour obtenir la fonction de coût d'une unité de programme.

Les travaux de Darte & Robert ([DR93b, DR93a]) proposent de traiter le problème de la distribution des données (et du code) en le modélisant par des fonctions affines des indices de boucle. Ils construisent pour cela un *graphe de communication* qui résume toutes les lectures et écritures d'un programme ; chaque arc du graphe représente une communication potentielle, à laquelle est associée une *distance*. Leurs but est de rendre nul le plus grand nombre de distances, *i.e.* pas de communication, ou tout au moins les rendre constantes, *i.e.* indépendantes des indices de boucle. Sinon, ils proposent de détecter les diffusions de données et les vectorisations de message.

Ce graphe de communication a deux types de nœuds : les instructions et les variables. Une variable $v1$ est lue par une instruction $s1$ si et seulement si il y a un arc du nœud $v1$ vers le nœud $s1$. Une variable $v2$ est écrite par $s1$ si et seulement si il y a un arc depuis $s1$ vers $v2$.

Leur transformation s'inspire directement du systolique en utilisant la méthode de projection : l'ordonnancement est donné par une forme linéaire et la fonction d'allocation est une projection du domaine d'itération de dimension

n sur une architecture de processeurs de dimension $n-1$.

De même, les travaux de Anderson & Lam ([AL93]) proposent de calculer automatiquement la distribution des données et des calculs en la représentant par des fonctions affines appelées *décompositions*. Leur décomposition des données et des boucles se fait en trois étapes. D'abord la *partition* qui détermine les dimensions des tableaux et des nids à distribuer sur les mêmes processeurs ; ensuite, l'*orientation* détermine l'alignement entre les tableaux et les nids, et les processeurs ; enfin, le *déplacement* détermine les décalages d'alignement. Il s'agit donc encore de rendre les distances indépendantes des indices de boucle (partition et orientation), puis de les annuler (déplacement).

2.2.2 Partitionnement

Ce problème consiste à déterminer le nombre et la taille des blocs des dimensions des tableaux associés à chaque processeur. En pratique, ceci est réalisé en spécifiant le partitionnement des tableaux sur l'architecture commune.

En CM Fortran, la directive `LAYOUT` permet de déterminer le partitionnement des tableaux. En HPF, le partitionnement est spécifié par la directive `DISTRIBUTE` (voir les sections 1.2.1 et 1.2.3).

A partir de leur méthode de recherche de l'alignement des tableaux (voir section 2.2.1) et de programmes écrits dans un langage parallèle pour machine à mémoire partagée, Li & Chen ([LC91]) proposent de générer des programmes à passage de message pour multiprocesseurs à mémoire distribuée. Pour cela, ils utilisent des techniques de reconnaissance de forme ("pattern matching") pour détecter différents types de communications (permutation, communication uniforme, diffusion, réduction, communication générale). Le partitionnement de chaque tableau est paramétré, avec au départ toutes les dimensions distribuées cycliquement par bloc. Ensuite, les primitives de communications sont sélectionnées et ordonnancées dans le programme. Le coût global des communications est alors formulé en fonction des paramètres de distribution des tableaux. Ces paramètres sont évalués de manière à minimiser ce coût.

Ils ont appliqué cette méthode au compilateur expérimental du langage fonctionnel Crystal de l'Université de Yale. Le projet global propose de construire un compilateur de programmes Fortran pour iPSC/2 et NCUBE qui utilise Crystal comme langage intermédiaire.

De même, Crooks & Perrott ([CP94]) proposent de déterminer le schéma de partitionnement des données d'un programme par reconnaissance de forme sur les références aux tableaux. A chaque tableau est associée une série de préférences. La meilleure d'entre elles est choisie par un estimateur de performance qui donne les temps d'exécution et de redistribution.

Ils ont appliqué leur méthode au développement d'un traducteur automatique source-à-source d'un sous-ensemble de Fortran90 (appelé *Fort*), le "Fort Compilation System".

Dans un tout autre esprit, Ramanujan & Sadayapan ([RS91]) proposent de partitionner les tableaux par une famille d'hyperplans, afin de trouver des hyperplans parallèles n'engendrant aucune communication. Ce type de partition est avantageux dans le sens où la restructuration du programme ne nécessite que des transformations simples : décalage et échange de boucle ("skewing" et "interchange"). Ils donnent une formulation mathématique de ce problème qui utilise les notations matricielles.

Ils proposent une extension de leur méthode dans le cas où il n'est pas possible de trouver une partition qui n'engendre pas de communication. Dans ce cas, ils essayent de minimiser le coût des communications.

Ces travaux peuvent être rapprochés de ceux de Abraham & Hudak (voir [AH91]) qui proposent de partitionner les boucles parallèles pour réduire les communications et les synchronisations. Les partionnements rectangulaires et hexagonaux sont particulièrement visés. Ils ne considèrent que des nids de boucles parallèles (*i.e.*, sans dépendance entre les itérations). Leurs cibles sont les multiprocesseurs à mémoire partagée avec cache ("ADT system"), ils tentent de minimiser le volume de communication.

Sheu & Tai ([ST91]) proposent d'utiliser les vecteurs de dépendance sur des dépendances constantes portées par les boucles, ceci pour la méthode

hyperplane ([Lam74]) qui donne une transformation temporelle d'un nid de boucles représentée par une projection selon un vecteur (*i.e.*, sur un hyperplan perpendiculaire à ce vecteur).

Ils définissent alors un partitionnement par blocs de l'espace d'itération qui minimise les communications inter-blocs et qui ne soit pas contraire à l'ordre donné par la transformation temporelle.

F. Hassaine ([Has93]) propose une méthode pour déterminer automatiquement le partitionnement par blocs des tableaux d'un programme parallèle pour multiprocesseur à mémoire distribuée. Il découpe les tableaux selon une seule dimension afin que le schéma de partitionnement soit simple. De plus, il choisit de construire des blocs de taille maximale pour minimiser les frontières, génératrices de communications. Un tel partitionnement vise les architectures en anneau de processeurs.

Citons encore les travaux de M. O'Boyle ([O'B93]) qui propose un algorithme de partitionnement automatique des données fondé sur quatre analyses différentes (parallélisme, équilibre des charges, alignement, volume de données accédé). Ces quatre analyses proposent chacune une série de partitionnements, le meilleur d'entre eux étant choisi de manière à minimiser le nombre de communications et le volume des données à transférer, et également à équilibrer la charge de travail de chaque processeur.

Ces travaux ont pour but de générer des programmes parallèles pour machines à mémoire distribuée. Des essais sur des vrais programmes ont été effectués sur une KSR-1 (voir section 1.1.2).

En plus de l'alignement (voir section 2.2.1), Anderson & Lam calculent le partitionnement des données dans le but principal d'éviter de faire des communications. S'il n'est pas possible de faire en sorte qu'il n'y ai qucune communication, il faut trouver des dimensions qui, au lieu d'être écrasées, sont distribuées par blocs. Dans ce cas, leur algorithme essaie d'éviter les coûteuses réorganisations des données en générant des communications internes aux boucles parallèles (*.e.g.*, les "nearest-neighbor shifts"). Pour cela, il utilise une transformation de programme appelée *tiling* qui partitionne l'espace d'itération en blocs. En dernier recours, les communications générées par la

réorganisation des données sont nécessaires, elles sont effectuées entre deux nids de boucles distincts.

Ils utilisent un *graphe d'interférence* qui a pour nœuds les tableaux et les nids de boucles, et il existe un arc non orienté entre un tableau et un nid si et seulement si le tableau est référencé dans le nid. Une contrainte existe pour chaque cycle et toutes les contraintes sont additionnées.

Pour résoudre le problème général de décomposition, ils utilisent un graphe de communication, dont les nœuds sont les nids de boucles et les arcs représentent tous les cas de réorganisations de données entre nids de boucles. Chaque arc de ce graphe est valué (poids) par une approximation au pire cas du coût de communication. L'algorithme général essaye d'éliminer les arcs successivement par ordre décroissant de poids.

2.2.3 Comparaisons

Comme nous l'avons déjà dit à la section précédente, les travaux de Feautrier et de son équipe proposent une méthode systématique de construction automatique de programme parallèle. Il calcule notamment une *fonction de placement* qui distribue explicitement les instructions sur une grille de processeurs virtuels. Le cadre théorique de cette méthode est présenté au chapitre suivant (chapitre 3).

Les méthodes de Li & Chen, et Gupta & Banerjee peuvent être comparées à celle de Feautrier (voir section 3.4) dans le sens où elles cherchent à résoudre le problème d'alignement des tableaux en minimisant le volume de communications, et avec également une heuristique pour réaliser un partitionnement qui cherche à minimiser le coût global des communications.

Ces deux méthodes, ainsi que celles de Knobe & Lukas & Steele, et Chapman & Fahringer, s'appliquent lors de la compilation de programmes parallèles dans lesquels il manque la distribution des données. Ainsi, contrairement à ce que propose Feautrier (voir section 3.4), la détermination de la distribution des données n'engendre pas de transformation de programmes.

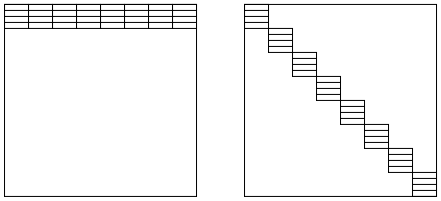
Ceci constitue une restriction qui peut s'avérer pénalisante. En effet, l'alignement s'effectue "parallèlement" aux dimensions des tableaux, *i.e.* il n'est

pas possible de faire des alignements biaisés.

Prenons par exemple le nid de boucles de la figure 2.3(a). Les deux boucles sont parallèles, mais pour ne pas engendrer de communication il faudrait pouvoir aligner les diagonales de A sur les lignes de B (voir figure 2.3(b)). Ceci n'est pas possible sans faire une transformation de programme.

```
do i = 1, n
  do j = 1, n-i+1
    B(i,j) = A(i+j-1,j)
  enddo
enddo
```

(a)



(b)

FIG. 2.3 - *Alignement biaisé*

En ce qui concerne son calcul de dépendance, la méthode proposée par M. Lam et son équipe ([MAL93]) s'inspire directement des travaux de Feautrier (voir section 3.2), et propose juste un algorithme qui traite des cas simples plus efficacement. Elle ne propose pas le calcul d'un ordonnancement mais utilise des transformations unimodulaires pour trouver le parallélisme de plus gros grain dans les nids de boucle.

De plus, de même que Darte & Robert et contrairement à Feautrier, Anderson & Lam peuvent relâcher la contrainte "owner compute" en séparant les fonctions de distribution des tableaux et des instructions.

Par contre, l'heuristique proposée par Darte & Robert ne traite que le cas d'un nid de boucle parfaitement imbriqué dans lequel il n'y a que des dépendances uniformes[12], alors que Feautrier traite le cas plus général d'un nid quelconque dans lequel les fonctions d'accès sont affines (voir section 3.1).

Enfin, Darte & Robert proposent également d'optimiser certaines communications lorsqu'elles ne peuvent pas être évitées. Nous verrons (voir cha-

12. Toutes les fonctions d'accès aux tableaux sont des translations.

pitre 4) que notre étude se propose d'étendre les travaux de Feautrier afin de réaliser de telles optimisations.

Enfin, les méthodes de Ramanujan & Sadayapan, et Abraham & Hudak se proposent de résoudre le problème d'alignement des tableaux dans un cas particulier de partitionnement et appliqué à des nids de boucles parfaitement parallèles, *i.e.* sans aucune dépendance. Comme Feautrier (voir section 3.5), elles proposent des transformations de programmes qui prennent en compte la distribution des données, néanmoins leurs restrictions sont très fortes, aussi bien au niveau du programme traité que sur le type de distribution proposé.

Chapitre 3

Cadre de travail et outils utilisés

3.1 Restrictions

Dans le prolongement des travaux de Feautrier et de son équipe du laboratoire PRiSM (Université de Versailles et Saint-Quentin), notre étude se restreint à une certaine classe de programmes dits à "contrôle statique". A la base de ce type de programme nous avons ce que nous appellerons les **paramètres de structure** et les **expressions à contrôle statique**.

Définition 5 *Un* ***paramètre de structure*** *est une variable scalaire entière dont la définition (*i.e. *l'assignation d'une valeur) a les deux propriétés suivantes :*

- *cette définition se trouve en dehors de tout corps de boucle et de tout corps de test dans l'ensemble du programme ;*
- *elle est réalisée soit par une instruction d'entrée/sortie, soit par une instruction d'affectation dont le membre droit ne fait intervenir que des paramètres de structure déjà évalués.*

Définition 6 *Pour une instruction donnée d'un programme, une* ***expression à contrôle statique*** *est une expression* quasi-linéaire entière *(*i.e.,

une expression linéaire entière dans laquelle peuvent apparaître des divisions entières) fonction uniquement des indices des boucles englobant l'instruction et des paramètres de structure.

Définition 7 *Un* ***programme à contrôle statique*** *est un programme structuré dans lequel il n'y a ni appel de procédure (**CALL**) ni instruction de branchement (**GOTO**). Les instructions admises sont :*

- *la* ***boucle DO*** *normalisée, si les bornes sont des* expressions à contrôle statique *;*
- *le* ***test IF****, si la conditionnelle est une formule de la logique de Presburger ([Gra88]),* i.e. *une combinaison booléenne sur des* expressions à contrôle statique *;*
- *l'****assignation****, si les accès aux tableaux sont des* expressions à contrôle statique.

Cette condition de quasi-linéarité est nécessaire pour les bornes de boucle afin que les espaces d'itération des boucles définissent des polyèdres convexes. De même, cette condition sur les fonctions d'accès aux tableaux est la plus faible qui rende encore possible une analyse statique exacte (*i.e.* l'utilisation de la programmation linéaire entière paramétrique).

La définition d'un programme à contrôle statique donnée par Feautrier (cf. [Fea91]) est plus restrictive. Néanmoins, à partir de la définition donnée ci-dessus, X. Redon ([Red91]) et A. Leservot ont montré ([Les93]) qu'il est possible de transformer le programme pour qu'il soit conforme à la définition de Feautrier. Ces transformations sont *l'expansion de scalaire*, *la substitution avant de variables scalaires*, *la normalisation des boucles* et *la mise sous forme normale disjonctive des tests.*

La figure 3.1 donne un exemple de programme à contrôle statique (élimination de Gauss), avec un premier nid de boucle qui réalise la décomposition `L-U` puis un second nid de boucle qui effectue la remontée triangulaire (également appelée substitution arrière, [GL83]).

```
      program gauss

      integer i, j, k, n
      real a(n,n), x(n), s, f

      do i = 1,n-1
        do j = i+1,n
s1          f = a(j,i)/a(i,i)
           do k = i+1,n
s2            a(j,k)=a(j,k) - f*a(i,k)
           end do
        end do
      end do
      do i = 1,n
s3       s = 0.
         do j = 1,i-1
s4          s = s + a(n-i+1,n-j+1)*x(n-j+1)
         end do
s5       x(n-i+1) = (a(n-i+1, n+1) - s)
     &         /a(n-i+1, n-i+1)
      end do
      end
```

FIG. 3.1 - *Programme* `gauss`

En prenant la tableau a comme référence spatiale, la figure 3.2 montre les trois principaux mouvements de données nécessaires à l'exécution de ce programme : pour i fixé $(s1)$, $a(i,i)$ doit être diffusé parallèlement aux colonnes ; pour (i,k) fixés $(s2)$, $a(i,k)$ doit être diffusé parallèlement aux colonnes ; pour (i,j) fixés $(s2)$, f doit être diffusée parallèlement aux lignes.

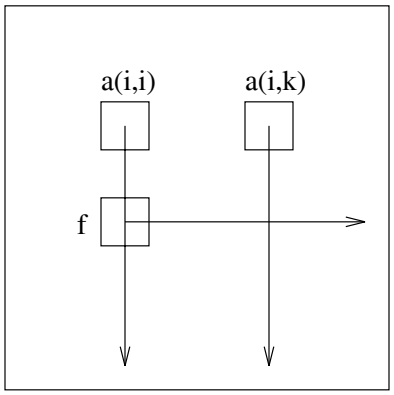


FIG. 3.2 - *Mouvements de données dans le programme* `gauss`

3.2 Graphe du flot de données

A partir d'un programme à contrôle statique, Feautrier calcule un *graphe du flot des données* (appelé DFG, pour "Data Flow Graph", voir [Fea91]). Ce DFG est un graphe de dépendances exact dans lequel n'apparaissent que des arcs producteur-consommateur (ou "flow-dependence"). Ces arcs portent sur des opérations dont le vecteur d'itération de la source est déterminé de manière exacte en fonction du vecteur d'itération de la destination ; cette abstraction de dépendance est appelée *itération de dépendance* (voir section 2.1.2).

Ce graphe résume très précisément la circulation des données dans un programme; en effet, il établit pour chaque valeur lue par une opération la dernière opération l'ayant modifiée.

3.2.1 Définitions

A chaque **nœud** du DFG est associé :

1. une **instruction** du programme ;
2. un **domaine d'exécution** qui est un système de contraintes spécifiant l'espace de variation des indices de boucle englobant cette instruction.

Les nœuds sont reliés entre eux par des arcs orientés qui représentent chacun une dépendance de donnée de type producteur-consommateur. Ces arcs sont orientés depuis la source (*i.e.* l'opération de définition) vers la destination (*i.e.* l'opération d'utilisation). A chaque **arc** sont associés les arguments suivants :

1. une **référence** qui donne le nom et la fonction d'accès en lecture de la variable sur laquelle porte la dépendance ;
2. une **transformation** qui est une expression linéaire exprimant les indices de boucle englobant l'instruction source en fonction des indices de boucle englobant l'instruction destination et des paramètres de structure ;
3. un **prédicat d'existence** de cet arc qui est un système de contraintes sur les indices de boucle englobant l'instruction destination et les paramètres de structures. Il spécifie la restriction au domaine d'exécution de l'instruction destination pour lequel cet arc existe.

Le domaine définit par ce prédicat correspond à l'intersection entre le domaine d'exécution de l'instruction destination et l'image par la transformation du domaine d'exécution de l'instruction source.

La figure 3.3 donne le graphe correspondant au DFG du programme `gauss`. Une description détaillée de chaque nœud et chaque arc de ce DFG est donnée dans les tableaux 3.1 et 3.2.

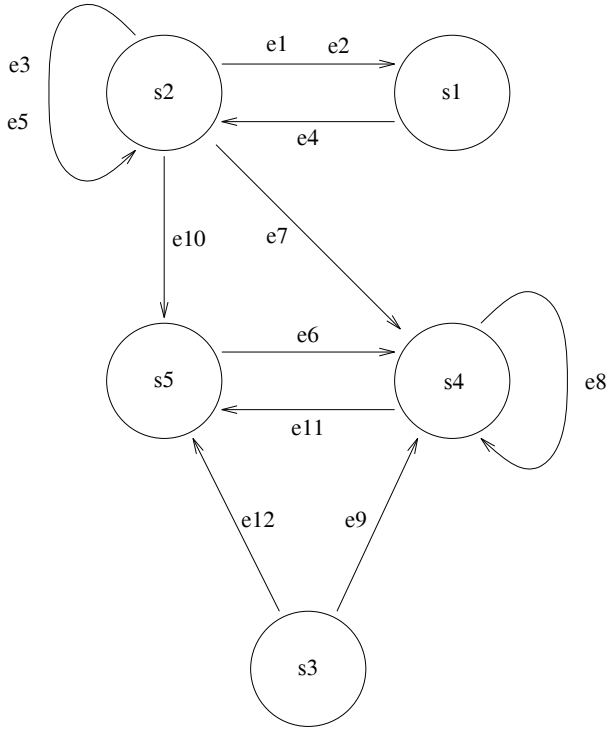


FIG. 3.3 - *DFG du programme* `gauss`

Instructions	Indices de boucle	Domaines
$s1$	i, j	$\begin{pmatrix} n-i-1 \\ i-1 \\ n-j \\ j-i-1 \end{pmatrix} \geq 0$
$s2$	i, j, k	$\begin{pmatrix} n-i-1 \\ i-1 \\ n-j \\ j-i-1 \\ n-k \\ k-i-1 \end{pmatrix} \geq 0$
$s3$	i	$\begin{pmatrix} n-i \\ i-1 \end{pmatrix} \geq 0$
$s4$	i, j	$\begin{pmatrix} n-i \\ i-1 \\ i-j-1 \\ j-1 \end{pmatrix} \geq 0$
$s5$	i	$\begin{pmatrix} n-i \\ i-1 \end{pmatrix} \geq 0$

TAB. 3.1 - *Nœuds du DFG du programme* `gauss`

Arc	Source	Puits	Référence	Prédicat
e1	$\langle s2, i-1, i, i\rangle$	$\langle s1, i, j\rangle$	a(i,i)	$i-2 \geq 0$
e2	$\langle s2, i-1, j, i\rangle$	$\langle s1, i, j\rangle$	a(j,i)	$i-2 \geq 0$
e3	$\langle s2, i-1, i, k\rangle$	$\langle s2, i, j, k\rangle$	a(i,k)	$i-2 \geq 0$
e4	$\langle s1, i, j\rangle$	$\langle s2, i, j, k\rangle$	f	
e5	$\langle s2, i-1, j, k\rangle$	$\langle s2, i, j, k\rangle$	a(j,k)	$i-2 \geq 0$
e6	$\langle s5, j\rangle$	$\langle s4, i, j\rangle$	x(n-j+1)	
e7	$\langle s2, n-i, n+1-i, n+1-j\rangle$	$\langle s4, i, j\rangle$	a(n-i+1,n-j+1)	$n-i-1 \geq 0$
e8	$\langle s4, i, j-1\rangle$	$\langle s4, i, j\rangle$	s	$j-2 \geq 0$
e9	$\langle s3, i\rangle$	$\langle s4, i, j\rangle$	s	$1-j \geq 0$
e10	$\langle s2, n-i, n+1-i, n+1-i\rangle$	$\langle s5, i\rangle$	a(n-i+1,n-i+1)	$n-i-1 \geq 0$
e11	$\langle s4, i, i-1\rangle$	$\langle s5, i\rangle$	s	$i-2 \geq 0$
e12	$\langle s3, i\rangle$	$\langle s5, i\rangle$	s	$1-i \geq 0$

TAB. 3.2 - *Arcs du DFG du programme* `gauss`

3.2.2 Principes théoriques

Le calcul de ce graphe s'effectue en déterminant pour chaque référence en lecture la source la plus tardive dans l'ordre d'exécution du programme. Cette source est calculée à partir de l'ensemble des sources possibles, *i.e.* l'ensemble des références en écriture susceptibles d'avoir produit la valeur considérée.

Dans un programme quelconque, nous notons (s, x) l'opération associée à l'instruction s et de vecteur d'itération x. Soit $A[g(x)]$ une référence à droite de cette instruction, nous notons $(s_k, y_k)_{1 \leq k \leq I}$ les opérations qui écrivent dans le tableau A. Elles ont un membre de gauche de la forme $A[f_k(y_k)]$.

> Prenons par exemple le programme `gauss`, voir figure 3.1. Le tableau a est lue par l'instruction $s4$ avec un vecteur d'itération $x = (i, j)$; la fonction d'accès g à pour valeur :
>
> $$g(i,j) = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} n \\ 1 \end{pmatrix}$$

Pour une instruction s_k, l'ensemble des sources possibles à la profondeur[1] p est donné par le polyèdre suivant :

$$Q_k^p = \{u \mid f_k(u) = g(x), (s_k, u) \prec_p (s, x), e_{s_k}(u) \geq 0\}$$

1. Voir section 2.1.2 pour la définition d'une profondeur de dépendance.

Nous notons $N_{r,t}$ le nombre de boucles englobant à la fois l'instruction r et l'instruction t, et $T_{r,t}$ le booléen de valeur `VRAI` si l'instruction r précède l'instruction t dans l'ordre textuel du programme. Avec ces notations, la condition $(s_k, u) \prec_p (s, x)$ désigne le prédicat de séquencement à la profondeur p qui peut s'écrire sous la forme :

$$(s_k, u) \prec_p (s, x) \equiv \begin{cases} (u[1 \cdot p] = x[1 \cdot p]) \wedge (u[p+1] < x[p+1]), si\ p < N_{s_k,s} \\ (u[1 \cdot p] = x[1 \cdot p]) \wedge T_{s_k,s}, si\ p = N_{s_k,s} \end{cases}$$

A chacun de ces polyèdres Q_k^p correspond une *dépendance directe*, *i.e.* une dépendance qui ne peut pas être déterminée par transitivité. Cette dépendance directe peut être représentée par une *forme quasi-affine* (ou *quast*), dont la syntaxe est donnée au tableau 3.3. Le signe $\perp$ indique que pour certaines valeurs des indices de boucles la référence lue par l'instruction s n'est pas définie par l'instruction s_k.

form	::=	integer
		| parameter
		| integer $*$ *form*
		| *form* $\div$ integer
		| *form* $+$ *form*
vector	::=	(*form* [, *form*] ...)
quast	::=	$\perp$
		| *vector*
		| **IF** *form* ≥ 0 **THEN** *quast* **ELSE** *quast*

TAB. 3.3 - *Syntaxe du* `quast`

Prenons par exemple les arcs $e8$ et $e9$ du DFG de notre programme `gauss`, qui donnent la source de la valeur de s utilisée dans l'instruction $s4$ (voir le tableau 3.2). Ils sont définis par le `quast` suivant :

`IF` $(j \geq 2)$`THEN`

$(s4, i, j-1)$

ELSE

$(s3, i)$

Cette dépendance directe est calculée comme le maximum lexicographique sur chacun des polyèdres, ce qui est résolu grâce au logiciel de programmation linéaire paramétrique en nombre entier PIP (voir section 2.1.1) :

$$K_k^p(x) = max_{\ll} Q_k^p$$

et la source recherchée est la plus tardive des dépendances directes, suivant le prédicat de séquencement :

$$S = max_{\prec}\{(s_k, K_k^p(x)) \mid 1 \leq k \leq I, 0 \leq p \leq N_{s_k,s}\}$$

En renumérotant toutes les sources possibles avec un seul indice l, et en notant L le nombre total de sources possibles, nous avons :

$$S = max_{\prec}\{(s_l, K_l(x)) \mid 1 \leq l \leq L\}$$

Notons $\{S_l\}_{1 \leq l \leq L}$ la famille comme suit :

$$\begin{cases} S_0 = \perp \\ \forall l \in \{1, \ldots, L\}, S_l = max_{\prec}(S_{l-1}, (s_l, K_l(b))) \end{cases}$$

Pour déterminer S, il suffit de déterminer par récurrence tous les éléments de cette suite : nous avons $S = S_L$.

3.3 Base de temps

La *base de temps* détermine pour chaque opération d'un programme sa date logique d'exécution ([Fea92a]). Pour cela il faut calculer pour chaque instruction s une base de temps θ_s exprimée en fonction des indices de boucles englobantes et des paramètres de structure. Elle permet alors de définir des ensembles d'opérations appelés *fronts*, constitués d'opérations indépendantes :

$$F(t) = \{u \mid \forall s, \theta_s(u) = t\}$$

L'exécution des opérations d'un même front peut être faite simultanément. Par contre l'exécution de deux fronts successifs doit être séquentielle, *i.e.* l'exécution des opérations du front $F(t+1)$ ne peut se faire que lorsque toutes les opérations du front $F(t)$ ont été exécutées.

Les hypothèses faites sur cette base de temps est qu'elle est affine par morceaux, positive, multidimensionnelle ([Fea92b]) et qu'elle respecte la condition de causalité qui impose à toute opération de ne débuter que lorsque toutes celles dont elle dépend sont achevées. Cette base de temps est calculée à partir des connaissances portant sur la circulation des données dans le programme Fortran à traîter qui sont résumées dans le DFG du programme.

Le calcul de cette base de temps fait en sorte que son expression soit la plus simple possible. Néanmoins, l'utilisation du logiciel PIP peut conditionner certaines valeurs (d'où l'affinité par morceaux). De plus, pour certains programmes il n'existe pas d'ordonnancement linéaire, dans ce cas plusieurs dimensions sont calculées.

Principe du calcul

La méthode de résolution repose sur une application d'un résultat théorique appelé *lemme de Farkas*. Ce lemme exprime le fait que si une forme affine est positive ou nulle sur un polyèdre (convexe) alors elle peut s'écrire sous la forme d'une combinaison linéaire positive des inéquations qui définissent ce polyèdre :

Lemme 8 *Soit $\mathcal{D}$ un polyèdre non vide défini par les inégalités suivantes :*

$$i \in \{1, \cdots, n\}, a_i x + b_i \geq 0$$

Une forme affine ϕ est positive ou nulle sur $\mathcal{D}$ si et seulement si elle peut s'écrire sous la forme :

$$\phi(x) = \mu_0 + \sum_{i=1}^{n} \mu_i (a_i x + b_i), \mu_i \geq 0$$

Pour calculer la base de temps nous posons que l'ordonnancement découle du DFG de la manière suivante :

Pour chaque sommet (s, x) du DFG, où s est l'instruction et x le vecteur d'itération, nous pouvons déduire du lemme de Farkas une fonction de la forme :

$$t_s(x) = \mu_0^s + \sum_{i=1}^{n} \mu_i^s f_i^s(x)$$

où les $f_i^s(x) \geq 0$ sont les contraintes définissant le domaine du sommet (s, x).

Chaque arc du DFG représente une dépendance, nous pouvons donc leur associer à chacun une condition de causalité qui assure qu'une instruction ne s'exécute que si celles dont elle dépend ont terminé leur exécution.

Avec ces notations et en supposant que la durée d'exécution des instructions est la même pour toutes (nous fixons notre unité de temps sur cette durée), la condition de causalité entre la source et la destination de chaque arc e du DFG, de transformation h_e, s'exprime sous la forme suivante :

$$t_{\delta(e)}(h_e(x)) - t_{\sigma(e)}(x) \geq 1$$

En appliquant le lemme de Farkas, nous obtenons l'équation suivante :

$$t_{\delta(e)}(h_e(x)) - t_{\sigma(e)}(x) - 1 = \lambda_0^e + \sum_{k=1}^{m} \lambda_k^e g_k^e(x)$$

où les $g_k^e(x) \geq 0$ sont les contraintes définissant le domaine de l'arc e.

Le but du calcul de la base de temps est de déterminer les coefficients μ_i qui minimisent cette fonction. Cette résolution est effectuée sur le graphe quotient par la relation d'équivalence de la connexité forte du DFG, qui est par définition acyclique (voir [Ber83]). La méthode consiste alors à parcourir ce graphe quotient et, pour chaque composante fortement connexe (CFC), à calculer les bases de temps associées à chacun des noeuds de celle-ci.

Dans une CFC et pour un nœud donné du DFG, les conditions de causalité de tous les arcs qui entrent en jeu sont prises en compte. Il en résulte un système linéaire d'inéquations à résoudre. Cette résolution est faite par PIP, et il y a au moins autant d'appel à PIP qu'il y a de noeuds dans le DFG (*i.e.* d'instructions dans le programme).

Lors de l'analyse du résultat donné par un appel à PIP, si toutes les conditions ne sont pas satisfaites alors une dimension de la base de temps a été trouvée mais il faut en calculer une autre à partir des conditions non satisfaites. Dans ce cas, nous sommes en présence d'une base de temps *multidimensionnelle*.

Considérons le programme `gauss`. Son DFG (voir figure 3.3) possède trois composantes fortement connexes $(s1, s2)$, $(s3)$ et $(s4, s5)$. De plus, ce programme a un ordonnancement linéaire. Le calcul de sa base de temps donne le résultat suivant :

$$\begin{array}{ll} \theta_{s1}(i,j) & = 2i - 2 \\ \theta_{s2}(i,j,k) & = 2i - 1 \\ \theta_{s3}(i) & = 0 \\ \theta_{s4}(i,j) & = 2j + 2n - 2 \\ \theta_{s5}(i) & = 2i + 2n - 3 \end{array}$$

Considérons maintenant l'extrait de programme suivant :

```
do i = 0, n
  do j = 0, i
    s = s + a(i,j)
  enddo
enddo
```

Il n'a pas d'ordonnancement linéaire, le calcul de sa base de temps introduit une deuxième dimension :

$$\theta_{s1}(i,j) \quad = (i \quad j)$$

3.4 Fonction de placement

A partir du DFG d'un programme, Feautrier propose le calcul d'une fonction de placement qui distribue explicitement les instructions sur une grille de processeurs virtuels ([Fea93]). Le programme est finalement transformé

en programme à assignation unique, ainsi en suivant la règle "owner compute" [2] la distribution des données et la distribution des instructions sur les processeurs ne forment plus qu'un unique problème.

Cette fonction de placement associe à chaque instruction une fonction affine π_s multidimensionnelle sur les indices des boucles englobantes (de cette instruction) et les paramètres de structure du programme. Pour déterminer le nombre maximal de dimensions à calculer pour une instruction donnée, nous nous limitons à la différence entre la dimension de son espace d'itération (*i.e.* la profondeur du nid de boucles dans lequel elle se trouve) et la dimension de sa base de temps.

En effet, le but est d'utiliser la base de temps et la fonction de placement d'une instruction pour déterminer la transformation des boucles qui l'englobent et sa réindexation. Or, notre méthode de génération de code nécessite que la transformation soit inversible (voir section 3.5.3) ; ainsi, la base de temps définit une partie de cette transformation et la fonction de placement détermine le complément. Si la somme de leurs dimensions respectives pouvait être plus grande que la dimension de l'espace d'itération alors au moins une composante de la fonction de placement serait combinaison linéaire des autres et de celles de la base de temps ; il est donc inutile de la calculer. Par contre, cette somme peut être inférieure à la dimension de l'espace d'itération, car dans ce cas il est toujours possible de trouver des expressions linéairement indépendantes pour compléter la transformation.

Pour chaque instruction, cette fonction définie donc une projection de l'espace d'itération (ou l'espace d'adressage du tableau accédée en écriture dans cette instruction) sur l'espace des processeurs virtuels.

Nous pouvons modéliser ces deux espaces comme des espaces vectoriels. Ainsi, l'espace d'itération a une dimension égale à la profondeur de son nid de boucle, les indices de boucle donnent la base canonique. De plus, l'espace de distribution (ou espace des processeurs), qui est un sous espace de l'espace d'itération, a sa base définit par les différentes composantes de la fonction de placement en fonction de la base canonique de l'espace d'itération.

2. Cette règle dit qu'une opération donnée sera exécutée par le processeur qui possède la donnée à modifier.

Principe du calcul

Le principe initial du calcul de la fonction de placement est de réduire au maximum le nombre de communications. Ainsi, pour un arc e du DFG de source $\sigma(e)$, de destination $\delta(e)$, de transformation $h_e()$ et de vecteur d'itération x (fonction des indices de la destination), nous associons une distance :

$$d_e(x) = \pi_{\delta(e)}(x) - \pi_{\sigma(e)}(h_e(x))$$

Chaque arc représente une communication potentielle, le but de ce calcul est donc de rendre nulles le plus grand nombre possible de distances. Néanmoins, elles ne peuvent en général pas être toutes annulées sans que cela entraîne une distribution sur un seul processeur. Le choix des distances à annuler est donc dirigé par une heuristique qui propose de les annuler une par une dans l'ordre décroissant du poids des arcs. Le poids d'un arc est égal au volume de données qui sont échangées. Cette méthode s'applique donc à satisfaire ce que nous appellerons des *conditions de coupure*.

Sachant que sur les machines parallèles le coût d'une communication varie selon le mouvement de données, le problème de cette heuristique est qu'elle ne sait pas différencier les différents types de communication qu'un arc est susceptible d'engendrer. Nous présentons au chapitre 4 notre extension de ce calcul qui optimise les communications.

Considérons le programme `gauss`. Sa fonction de placement est de dimension 2, et a pour valeur :

$$\begin{array}{ll} \pi_{s1}(i,j) & = (j-1 \quad 0) \\ \pi_{s2}(i,j,k) & = (j-1 \quad k) \\ \pi_{s3}(i) & = (-i+n \quad n+1) \\ \pi_{s4}(i,j) & = (-i+n \quad n+1) \\ \pi_{s5}(i) & = (-i+n \quad n+1) \end{array}$$

3.5 Génération de code et optimisation

La génération de code est destinée à construire le programme parallèle en tenant compte de toutes les informations apportées par les phases précé-

dentes, à savoir le calcul du graphe de flot de données, le calcul de la base de temps et le calcul de la fonction de placement.

A une date donnée, ce programme parallèle doit exécuter le front correspondant (voir section 3.3) en parallèle, synchroniser puis passer à la date suivante. Ceci donne la forme générale de notre programme :

```
do t = 1, n
  executer en parallele F(t)
  synchroniser
end do
```

Pour cela, chaque instruction du programme est dupliquée et décomposée autant de fois que nécessaire pour tenir compte des éléments suivants:

- les domaines de base de temps de l'instruction[3] ;
- les flots de données sur chaque référence à droite de l'instruction[4] ;

Pour chacune de ces nouvelles instructions du programme, nous effectuons la réindexation, c'est-à-dire que nous remplaçons tous les compteurs de boucles du programme par de nouvelles variables dépendant de l'expression de la base de temps (donc du temps global) et de l'expression de la fonction de placement (si nécessaire).

La génération de code est fondée sur trois transformations :

- expansion totale
- réordonnancement
- réindexation

Pour produire du code parallèle compilable sur une machine cible donnée, nous avons été amenés à ajouter quelques extensions à cette génération de code. Par exemple, nous ajoutons la déclaration de la distribution des tableaux. Toutes ces extensions sont données au chapitre 5.

3. La base de temps est affine par morceaux, chaque "morceaux" constitue un domaine.

4. Chaque référence à droite peut avoir plusieurs sources possibles, chacune constitue un flot

3.5.1 Expansion totale

Cette transformation élimine toutes les dépendances du type *anti-dependence* et *output-dependence* pour construire un programme à assignation unique. Dans un tel programme chaque cellule mémoire n'est assignée qu'une seule fois.

L'expansion totale est une transformation bien connue ([PW86]) et ne pose aucun problème théorique. Voici la méthode d'expansion totale donnée par M. Raji-Werth et P. Feautrier ([RWF90]) :

L'expansion totale d'un programme est réalisée en appliquant à chaque instruction s d'un programme les deux étapes suivantes :

1. remplacer la *lhs* ("left hand side", *i.e.* référence à gauche) de s par un tableau ins_s indicé par les compte-tours des boucles englobantes ;
2. remplacer chaque *rhs* ("right hand side", *i.e.* référence à droite) de s par sa source donnée par le DFG. S'il y a plusieurs sources possibles, une instruction conditionnelle est créée et des variables temporaires sont utilisées.

Cette méthode propose d'introduire des temporaires qui sont affectés de la source de la dépendance de la référence qu'ils représentent. Il est également possible de recopier dans chaque test l'instruction en entier, dans ce cas les temporaires sont devenus inutiles. Par contre cela demande d'imbriquer tous les tests introduits par chaque référence.

> Revenons à notre programme `gauss`, voir figure 3.1. Le code suivant est le résultat de l'expansion totale (sans temporaire) sur l'instruction $s2$ de ce programme (le tableau définit dans l'instruction `s#` a été renommé `ins#`) :
>
> ```
> do i = 1, n-1
>   do j = i+1, n
>     ...
>     ...
>     do k = i+1, n
>       if (i >= 2) then
> ```

```
          ins2(i,j,k) = ins2(i-1,j,k)-ins1(i,j)*ins2(i-1,i,k)
        else
          ins2(i,j,k) = a(j,k)-ins1(i,j)*a(i,k)
        endif
      enddo
    enddo
  enddo
  ...
  ...
```

Ce programme présente le cas particulier où le prédicat associé à chaque source est le même pour les deux `rhs` qui ont plusieurs sources possibles.

3.5.2 Réordonnancement

Cette transformation consiste à opérer un changement de base dans l'espace d'itération des boucles pour faire apparaître explicitement ce que nous avons appelé les fronts (voir section 3.3). Le code généré doit énumérer tous les points entiers du polyèdre formé par le domaine d'itération ([AI91]). Cette transformation est représentée par une matrice inversible mais non nécessairement unimodulaire, qui est construite à partir de la base de temps et de la fonction de placement.

Le principe du réordonnancement de boucle est donc de générer un nouveau nid de boucles. Voici les principaux problèmes et concepts qui y sont attachés ([Col93]) :

Matrices de transformation

Soit une instruction s de vecteur d'itération $\vec{i}_s$, de base de temps θ_s et de fonction de placement π_s. La transformation associée à cette instruction est caractérisée par la formule suivante ([RW92]) :

$$\vec{j}_s = T_s(\vec{i}_s) = \begin{pmatrix} \theta_s \\ \pi_s \\ A_s \end{pmatrix} = M_s.\vec{i}_s + C_s$$

Pour chaque instruction, les matrices qui caractérisent cette transformation T_s sont donc construites à partir de la base de temps et de la fonction de placement. La matrice M_s doit être inversible (voir section 3.5.3), donc de taille $n_s \times n_s$, où n_s est la dimension du nid de boucles qui englobe s. Ainsi, lorsque le nombre de composantes linéairement indépendantes de la base de temps et de la fonction de placement n'est pas suffisant, il est nécessaire d'introduire le complément A_s, dont les composantes doivent être convenablement choisies afin que M_s soit unimodulaire (si cela est possible).

Bien entendu, seules les composantes non constantes (*i.e.* qui sont fonction des indices de boucle) sont prises en compte dans cette formule.

> Prenons par exemple l'instruction $s2$ de notre programme `gauss`, voir figure 3.1 :
>
> $$s2 : a(j,k) = a(j,k) - f * a(i,k)$$
>
> L'espace d'itération de cette instruction $s2$ est de dimension 3. Sa base de temps est de dimension 1 et sa fonction de placement est de dimension 2 :
>
> $$\begin{cases} \theta_{s2}(i,j,k) = & 2i - 1 \\ \pi_{s2}(i,j,k) = & (j - 1 \quad k) \end{cases}$$
>
> Ces trois expressions sont linéairement indépendantes, il n'y a donc pas besoin de matrice de complément (A_{s2}). La correspondance en coordonnées temps-espace est la suivante (t_{s2} est une variable **temporelle**, p_0 et p_1 sont des variables de **placement**):
>
> $$T_{s2} : \begin{pmatrix} t_{s2} \\ p_0 \\ p_1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 & -1 \\ 0 & -1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix}$$

Non unimodularité

J.-F. Collard propose une méthode de réindexation qui prend en compte le cas où l'expression de la base de temps entraîne la non-unimodularité de M_s ([Col93]).

Dans ce cas, il propose de déterminer la période de chaque dimension de la base de temps qui correspond au coefficient de la diagonale de la forme de Hermite ([Sch86]) de la matrice de base de temps.

Il montre qu'en divisant chaque ligne de cette matrice par la période correspondante, nous obtenons alors une matrice qu'il est possible de compléter pour obtenir une matrice carrée unimodulaire. Il suffit alors de revenir à la véritable matrice de base de temps en gardant la partie complétée intacte.

Nous obtenons donc une matrice non-unimodulaire qui transforme un polyèdre convexe en un polyèdre qui n'a pas de "trous" sur les parties qui correspondent à la complétion, *i.e.* à la fonction de placement. Ceci permet donc de pouvoir générer des boucles parallèles ayant des pas de un.

> Reprenons notre instruction $s2$ du programme `gauss`. La matrice M_{s2} a la valeur suivante :
>
> $$M_{s2} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
>
> Cette matrice a pour déterminant 2, elle n'est donc pas unimodulaire. Néanmoins, la base de temps est monodimensionnelle et de période 2. Ainsi, en divisant par 2 la première ligne de cette matrice nous obtenons une matrice unimodulaire :
>
> $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Cette "élimination" de la non unimodularité est réalisée en introduisant ce que Collard appelle un **ordonnancement mineur**, voir plus loin.

Bornes de boucles

Le problème est de déterminer à partir d'un système de contraintes les bornes du nid de boucles qui va permettre de parcourir une fois et une seule chaque point entier du polyèdre correspondant.

Une première méthode ([CFR93]) consiste à utiliser le logiciel de programmation paramétrique en nombres entiers, appelé PIP (cf. la section 2.1.1).

Cette méthode démontre que le résultat donné par PIP (un *quast*, voir section 3.2.2) peut être simplifié par un maximum (ou un minimum) sur les feuilles du quast. Cette méthode est utilisée par Collard ([Col93]).

Une seconde méthode ([AI91]) propose d'utiliser l'algorithme d'élimination de variable de Fourier-Motzkin (voir section 2.1.1). Le problème de cette résolution est qu'elle génère un grand nombre de contraintes redondantes, il est donc nécessaire de posséder un test de redondance assez fin pour en éliminer le plus possible. Cette méthode est utilisée par Raji-Werth ([RW92]).

Revenons à notre programme `gauss`. Pour l'instruction $s2$, il s'agit de déterminer les intervalles dans lesquelles les trois variables t_{s2}, p_0 et p_1 sont définies :

$$\begin{cases} 1 \leq t_{s2} \leq 2n-3 \\ (t_{s2}+1)/2 \leq p_0 \leq n-1 \\ (t_{s2}+3)/2 \leq p_1 \leq n \end{cases}$$

Ordonnancement local

Chaque instruction possède son ordonnancement propre. Ainsi, à chaque instruction peut être associé un *ordonnancement local* qui est une fonction multidimensionnelle et qui correspond à la base de temps de l'instruction. Pour chaque ordonnancement local, et sur chacune de ses dimensions, nous devons calculer les bornes inférieures et supérieures de la fonction associée.

L'ordonnancement local est matérialisé par une variable de **temps local**. L'exécution d'une instruction est alors gardée par un test qui vérifie que la variable d'ordonnancement local est égale à la variable d'ordonnancement global ; cette variable est initialisée avec sa borne inférieure, elle est incrémentée de sa période et un test de dépassement de sa borne supérieure est introduit.

Etant donné que les bornes d'un ordonnancement local peuvent dépendre de paramètres de structure, il faut s'assurer que la borne inférieure est bien *inférieure* à la borne supérieure[5].

Prenons, par exemple, un ordonnancement local t ayant les bornes

5. En pratique cela consiste à mettre une garde sur les instructions d'initialisation des variables d'ordonnancements locaux

suivantes :

$$2n + 1 \leq t \leq 4n - 5$$

Nous voyons bien que si $n \leq 2$, la borne inférieure est strictement plus grande que la borne supérieure. Dans ce cas, il est nécessaire de mettre un test à l'initialisation de cette variable.

Ordonnancement global

Pour déterminer l'ordonnancement global du programme, il faut en déterminer la dimension et les bornes. Sa dimension est égale à la plus grande dimension parmi les dimensions des ordonnancements locaux, sa borne inférieure est égale à la plus petite valeur parmi les bornes inférieures de ces ordonnancements, et sa borne supérieure est égale à la plus grande valeur parmi les bornes supérieures de ces ordonnancements.

Concrètement, il faut introduire autant de variables d'ordonnancement global qu'il y a de dimensions.

Revenons à notre programme `gauss`. La base de temps de chaque instruction est de dimension 1, donc la dimension de l'ordonnancement global est de dimension 1. De plus, les bornes inférieures des bases de temps sont $(2n, 1, 0, 2n-1, 0)$, donc, en supposant $n \geq 1$, la borne inférieure de l'ordonnancement global est 0. Enfin, les bornes supérieures des bases de temps sont $(2n-4, 4n-3, 2n-3, 0, 4n-4)$, donc, en supposant $n \geq 1$, la borne supérieure de l'ordonnancement global est $4n - 3$.

Le squelette de notre programme parallèle a donc la forme suivante (la variable T_0 correspond à la première et seule dimension de l'ordonnancement global) :

```
PROGRAM gauss
DO T0 = 0, 4n - 3
   exécuter le front F(T0) en parallèle
ENDDO
END
```

Ordonnancement mineur

Comme nous le signalions ci-dessus, l'ordonnancement mineur permet de se ramener à une transformation unimodulaire. En outre, en remplaçant dans les fonctions d'accès aux tableaux les variables d'ordonnancement locaux par des variables d'ordonnancement mineurs, nous économisons de la mémoire.

Ainsi, le passage d'un ordonnancement *majeur* (t_x) à un ordonnancement mineur (τ_x) revient à faire le changement de variables suivant :

$$\begin{cases} t_x = \theta_x(\vec{\imath}) = \omega_x \times \tau_x + l_x \\ l_x = min_{\vec{\imath}}(\theta_x(\vec{\imath})) \end{cases}$$

Les constantes ω_x et l_x correspondent respectivement à la période de la base de temps de l'instruction x et à la valeur minimale de cette base de temps.

Ces variables d'ordonnancement mineur sont initialisées à zéro et elles sont incrémentées de 1. Les bornes des boucles parallèles peuvent également être exprimées en fonction de cette nouvelle variable.

Reprenons l'instruction $s2$ de notre programme `gauss`. En appelant τ_{s2} notre variable d'ordonnancement mineur pour cette instruction, nous avons :

$$t_{s2} = 2\tau_{s2} + 1$$

L'introduction de cette variable d'ordonnancement mineur nous donne le squelette de code suivant pour l'instruction $s2$ (`S0t0` est la variable d'ordonnancement global, `S2t0` le temps local pour cette instruction, `S2q0` le temps mineur, `S2p0` et `S2p1` les deux variables de placement) :

```
IF(S0t0 .eq. S2t0) THEN
  FORALL(S2p0 = S2q0+1:n-1, S2p1 = S2q0+2:n)
    ins2(S2q0, S2p0, S2p1) = ...
  ENDFORALL
  S2t0 = S2t0 + 2
  S2q0 = S2q0 + 1
  IF(S2t0 .gt. 2n-3) S2t0 = -1
ENDIF
```

3.5.3 Réindexation

Cette transformation consiste à répercuter sur les fonctions d'accès aux tableaux le passage des anciens aux nouveaux indices de boucle. Cette transformation est une conséquence directe du réordonnancement.

Soient deux instructions quelconques S_b et S_a dans notre programme initial, tels que la seconde lit une donnée définie dans la première :

$$\begin{aligned} &S_b : b[\vec{\imath_b}] = \dots \\ &\dots \\ &S_a : a[\vec{\imath_a}] = \dots b[\mathcal{L}_e(\vec{\imath_a})] \dots \end{aligned}$$

Le vecteur d'indices $\vec{\imath_a}$ appartient à l'espace d'itération de S_a, et $\mathcal{L}_e$ est la fonction d'accès en lecture au tableau $\mathtt{b}$ en fonction des indices $\vec{\imath_a}$ (*i.e.* la transformation de l'arc e qui donne la source de cette lecture). Soient T_a et T_b les transformations associées à nos instructions, elles définissent les nouveaux indices en fonction des anciens :

$$(t_b, p_b) = T_b(\vec{\imath_b}), (t_a, p_a) = T_a(\vec{\imath_a})$$

L'expansion totale est effectuée à ce niveau, il est donc nécessaire d'introduire de nouveaux tableaux. Ainsi, les fonctions d'accès aux tableaux des membres de gauche sont exprimées en fonction des nouveaux indices :

$$\begin{aligned} &S_b : b'(t_b, p_b) = \dots \\ &\dots \\ &S_a : a'(t_a, p_a) = \dots b'(T_b \circ \mathcal{L} \circ T_a^{-1}(t_a, p_a)) \dots \end{aligned}$$

Avec ce type de réindexation, Collard ([Col93]) a montré qu'il est alors possible de réduire le "trop plein" d'allocation mémoire engendré par l'expansion totale.

Pour cela, il introduit l'ordonnancement mineur (voir ci-dessus). Le passage en "mineur" peut être caractérisé par une fonction :

$$\Theta_x : (\tau_x, p_x) \mapsto (\frac{t_x - l_x}{\omega_x}, p_x)$$

La formule de réindexation est alors plus complexe :

$$S_b : b'(t_b, p_b) = \ldots$$
$$\ldots$$
$$S_a : a'(t_a, p_a) = \ldots b'(\Theta_b \circ T_b \circ \mathcal{L} \circ {T_a}^{-1} \circ {\Theta_a}^{-1}(\tau_a, p_a)) \ldots$$

Prenons par exemple l'arc $e4$ du DFG de notre programme `gauss`, qui porte sur la variable `f`. L'instruction $s2$ est la destination de cet arc. L'instruction $s1$ est la source de cet arc, elle définie cette variable qui, après expansion totale, est devenue un tableau à deux dimensions `ins1`. La réindexation de l'accès en lecture de ce tableau dans l'instruction $s2$ est définie comme suit :

$$\Theta_{s1} \circ T_{s1} \circ \mathcal{L}_{e4} \circ {T_{s2}}^{-1} \circ {\Theta_{s2}}^{-1}(\tau_{s2}, p_{s2})$$

Les valeurs de T_{s2} et Θ_{s2} ont été données à la section précédente :

$$T_{s2} : \begin{pmatrix} t_{s2} \\ p_0 \\ p_1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i_{s2} \\ j_{s2} \\ k_{s2} \end{pmatrix} + \begin{pmatrix} 0 & -1 \\ 0 & -1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix}$$
$$\Theta_{s2} : \begin{pmatrix} \tau_{s2} \\ p_0 \\ p_1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} t_{s2} \\ p_0 \\ p_1 \end{pmatrix} + \begin{pmatrix} 0 & -\frac{1}{2} \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix}$$

Par un calcul similaire, nous avons les valeurs suivantes pour T_{s1} et Θ_{s1} :

$$T_{s1} : \begin{pmatrix} t_{s1} \\ p_0 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i_{s1} \\ j_{s1} \end{pmatrix} + \begin{pmatrix} 0 & -2 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix}$$
$$\Theta_{s2} : \begin{pmatrix} \tau_{s1} \\ p_0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} t_{s1} \\ p_0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix}$$

La valeur de $\mathcal{L}_e$ est donnée par le tableau 3.2, page 75 :

$$\mathcal{L}_e : \begin{pmatrix} i_{s1} \\ j_{s1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i_{s2} \\ j_{s2} \\ k_{s2} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix}$$

Après la combinaison de toutes ces expressions, nous obtenons la fonction d'accès suivante : (τ_{s2}, p_0)

Arbre de réindexation

Le but de cette étape est d'effectuer la réindexation dans le code, *i.e.* remplacer les anciens indices de boucle par les nouveaux dans les fonctions d'accès aux tableaux.

Pour chaque instruction, il faut en générer une nouvelle qui correspond à la construction d'un arbre binaire, dans lequel les nœuds intermédaires sont des tests (avec une branche `VRAI` et une branche `FAUX`) et les feuilles sont des assignations. Nous dirons qu'un nœud appartient au niveau n de cet arbre si le chemin entre la racine de l'arbre et ce nœud contient exactement $n-1$ branches `VRAI`. Le premier niveau de test correspond aux différents domaines de base de temps de l'instruction[6] ; les niveaux suivants correspondent aux différentes sources possibles des références à droite de l'instruction.

Ainsi, chaque instruction peut être dupliquée de nombreuses fois, et à chaque copie est associée un prédicat.

Dans l'instruction S_a, supposons que la référence $b(\mathcal{L}(\vec{\imath}_a))$ ait deux sources possibles b' et b''. Alors, l'instruction S_a doit être dupliquée une fois, le prédicat du test étant noté $C(t_a, p_a)$. Le squelette du code de l'instruction S_a est le suivant (le polyèdre définissant la boucle parallèle est noté P_a) :

```
IF t == t_a THEN
   FORALL p_a ∈ P_a
      a'(τ_a,.p_a) = if C(t_a, p_a) then b'(f_1(τ_a, p_a)) else b''(f_2(τ_a, p_a))
   ENDFORALL
   t_a = t_a + ω_a
   τ_a = τ_a + 1
   IF (t_a > ub_a) t_a = −1
ENDIF
```

Code Fortran

La syntaxe Fortran n'autorise pas les expressions conditionnelles dans les assignations (contrairement au langage C). Ainsi, lorsqu'une instruction est dupliquée, il est nécessaire de découper l'ensemble du nid de boucles qui

6. Il y a autant de domaine que de "morceaux".

lui correspond. Pour cela, il faut introduire un ordonnancement local pour chaque duplication, avec ses propres bornes (elles sont calculées en tenant compte du prédicat associé à la copie qui lui correspond). Par contre, l'ordonnancement mineur reste unique (les accès en écriture sont toujours fait sur le même tableau).

Reprenons l'instruction $s2$ de notre programme `gauss`. La base de temps de cette instruction n'est pas divisée en plusieurs domaines. Par contre, il y a trois références à droite dont deux (`a(i,k)` et `a(j,k)`) ont deux sources possibles (de même prédicat, $i \geq 2$). Le code de cette instruction doit donc être découpé en deux morceaux possédant chacun leur propre ordonnancement local, `S2t0` et `S2t1`; pour l'incrémentation du temps mineur nous utilisons un `flag`, pour éviter plusieurs incrémentations :

```
 S2t0 = 3
 S2t1 = 1
 S2flag0 = 0
 ...
   IF(T0 .eq. S2t0) THEN
     FORALL(S2p0 = S2q0+1:n-1, S2p1 = S2q0+2:n)
       ins2(S2q0, S2p0, S2p1) = ins2(S2q0-1, S2p0, S2p1) -
&                             ins1(S2q0, S2p0)*ins2(S2q0-1, S2q0, S2p1)
     ENDFORALL
   S2t0 = S2t0 + 2
   S2flag0 = 1
   IF(S2t0 .gt. 2n-3) S2t0 = -1
   ENDIF
   IF(T0 .eq. S2t1) THEN
     FORALL(S2p0 = 1:n-1, S2p1 = 2:n)
       ins2(S2q0, S2p0, S2p1) = a(S2p0+1, S2p1) -
&                             ins1(S2q0, S2p0)*a(S2q0+1, S2p1)
     ENDFORALL
   S2t1 = S2t1 + 2
   S2flag0 = 1
   IF(S2t1 .gt. 1) S2t1 = -1
   ENDIF
   IF(S2flag0 .eq. 1) S2q0 = S2q0 + 1
   S2flag0 = 0
 ...
```

Le réordonnancement et la réindexation du code complet du programme `gauss` sont donnés à l'annexe B.1.

3.5.4 Optimisation

Voici deux optimisations qui permettent de réduire respectivement l'allocation mémoire et l'*overhead* de contrôle.

Délais

Dans une instruction S_a, supposons qu'une référence à droite (un tableau a) est lue avec une fonction d'indice sur la première dimension (*i.e.*, la première dimension temporelle) de la forme $(t_a - d)$, où d est une constante numérique positive appelée **délai**. Nous pouvons en déduire que, pour des indices de boucle fixés, l'élément lu a été produit il y a d unités de temps, et il ne sera plus jamais lu par cette référence. Si aucune autre référence ne le lit, alors il peut être écrasé.

Si toutes les références à droite de ce tableau sont de cette forme, alors, en notant D le plus grand délai de ces références, seulement $D+1$ éléments peuvent être alloués à ce tableau sur sa première dimension. Dans ce cas, il est nécessaire d'effectuer les accès à cette dimension modulo $D+1$.

Avec un ordonnancement mineur, une autre condition doit être remplie : la période de l'instruction source et la période de l'instruction destination doivent être égales.

De plus, notre fonction d'accès est de la forme $(\tau - d', p)$, où d' est une constante numérique (d' peut être négative). Dans ce cas, le délai est égal à $\mid d' + 1 \mid$.

> Revenons à notre programme `gauss`. Le tableau défini par l'instruction $s1$ a un délai d'accès égal à 2. Toutes les références au tableau `ins1` ont donc leur fonction d'accès calculée modulo 2, notamment la référence en écriture :
>
> $$\texttt{ins1}(mod(\tau_{s1}, 2), p_0, p_1) = \ldots$$
>
> Pour les autres instructions, le délai est égal à 0 pour $s1$, $s3$ et $s5$, et il n'y a pas de délai pour $s2$.

Découpage de boucle

M. Raji-Werth propose une optimisation qui découpe la boucle sur le temps (*i.e.* l'intervalle de définition de l'ordonnancement global). Cette optimisation est reprise par Collard ([Col93]) qui propose d'utiliser les Composantes Fortement Connexes (CFC) du DFG pour déterminer le découpage à réaliser.

Néanmoins, les ordonnancements de deux CFC reliées par un chemin (que nous appellerons des CFC connectées) peuvent se chevaucher, comme le montre l'exemple qui suit.

Etudions le programme `doacross` donné figure 3.4.

```
      program doacross
      integer i,j,n
      real a(n), b(n), c(n)

      do i = 1, n
s1      a(i) = a(i-1) + b(i)
      end do
      do i = 1, n
s2      c(i) = c(i-1) + a(i)
      end do
      end
```

FIG. 3.4 - *Programme* `doacross`

La figure 3.5 donne le DFG correspondant à ce programme. Nous en déduisons que ce graphe a deux CFC connectées, une pour chacun des nœuds. De plus, la base de temps de ce programme est la suivante :

$$\begin{cases} \theta_{s1}(i) = i \\ \theta_{s2}(i) = i + 1 \end{cases}$$

Nous avons donc bien deux CFC connectées dont les ordonnancements se chevauchent.

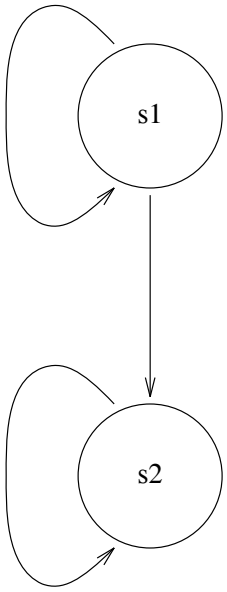


FIG. 3.5 - *DFG du programme* `doacross`

Le point important vient du fait qu'à partir du moment où les ordonnancements se chevauchent alors le parallélisme existant entre deux CFC est du type `DOACROSS`, ce qui est presque impossible à exploiter sur les machines actuelles.

Pour réaliser un découpage sur deux CFC connectées il est donc nécessaire de vérifier que leur ordonnancements ne se chevauchent pas. Dans ce cas, la boucle d'ordonnancement global peut être découpée en deux. Chacune de ces boucles ne contient alors que les instructions qui appartiennent à la CFC correspondante. Cela permet d'éliminer des instructions conditionnelles.

> Sur notre programme `gauss`, nous avons trois CFC (voir figure 3.3, page 74) ; la figure 3.6 donne le graphe quotient. Nous voyons donc que ces CFC ne sont pas toutes connectées : il n'y a pas de chemin reliant `cfc1` et `cfc2`. Par contre, `cfc3` est connectée aux deux autres et son ordonnancement ne chevauche pas les leurs.
>
> Nous pouvons donc couper la boucle d'ordonnancement global en deux : la première contient l'exécution des instructions $s1, s2$ et $s3$; la seconde, l'exécution des instructions $s4$ et $s5$. Les intervalles d'ordonnancement global sont respectivement $[0, 2n-3]$ et $[2n-1, 4n-2]$.

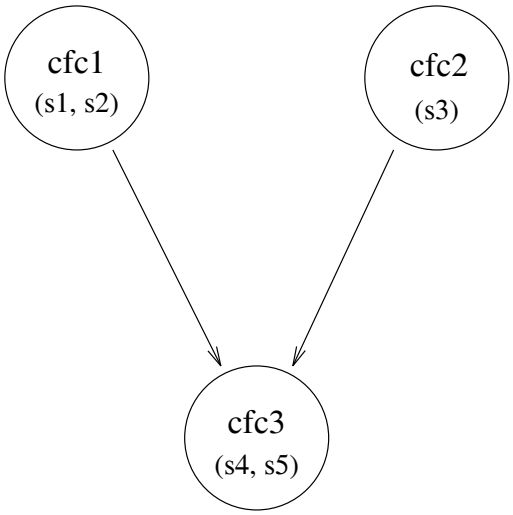


FIG. 3.6 - *Graphe quotient du DFG du programme* `gauss`

Chapitre 4

Placement avec optimisation des communications

Le but du calcul de la fonction de placement est de déterminer pour chaque opération le numéro du processeur virtuel qui l'exécute.

Pour cela, nous associons à chaque instruction une fonction affine $\pi_s(x_s)$ qui donne le numéro du processeur virtuel qui doit exécuter l'itération x_s de l'instruction s. Cette fonction de placement est donc une projection de l'espace d'itération sur l'espace des processeurs virtuels, que nous appellerons **espace de distribution**.

Pour une instruction s donnée et deux matrices d'entiers A_s et B_s, nous pouvons donc décrire notre fonction de placement comme :

$$\pi_s(x_s) = A_s.x_s + B_s$$

Nous représentons chaque dimension de notre fonction de placement par un prototype (noté π_s^p), sous la forme :

$$\pi_s^p(x_s) = \sum_{i=1}^{m_s} (\lambda_i^s.x_i^s) + \lambda_0^s$$

où $x_s = (x_0^s, ..., x_{m_s}^s)$ est l'ensemble des indices de boucle et paramètres de structure de l'instruction s, et $\lambda^s = \{\lambda_i^s\}_{0 \leq i \leq m_s}$ est une famille de coefficients inconnus entiers. Le but du calcul de la fonction de placement est de déterminer l'espace de distribution de chaque instruction qui minimise le coût de

communication entre les processeurs, *i.e.* déterminer pour chaque dimension les valeurs des coefficients de λ^s. A présent, nous notons λ la famille de tous les coefficients inconnus ainsi créés pour les instructions d'un programme donné.

Revenons à notre programme `gauss`. Pour l'instruction $s1$, nous avons :

$$x_{s1} = (i, j, n)$$
$$\pi^p_{s1}(i, j, n) = \lambda_2.i + \lambda_3.j + \lambda_4.n + \lambda_1$$

Comme nous l'avons déjà dit, la méthode classique de calcul de la fonction de placement (voir section 3.4) ne sait pas différencier les différents types de communication qu'un arc est susceptible d'engendrer.

En effet, comme nous l'avons montré section 1.1.1 pour la CM-5, sur la plupart des machines massivement parallèles ont été implantés de manière efficace les mouvements de données des types diffusion et réduction avec un coût dépendant du nombre de processeurs. Egalement ont été implantés les mouvements de données du type translation avec un coût indépendant du nombre de processeurs. Ainsi, ces architectures proposent aux programmeurs des primitives de haut niveau qui utilisent ces types de communication (par exemple, la fonction intrinsèque `SPREAD`, qui diffuse parallèlement aux axes de distribution).

En détectant dans un programme les instructions qui sont susceptibles de produire ce type de communication, nous pouvons utiliser cette information pour calculer une fonction de placement qui permet ce type de communication. En outre, elle permet de générer du code en utilisant les primitives fournies par le langage pour obtenir une exécution plus efficace.

Nous savons que rares sont les programmes, dont les données sont distribuées, où il n'y a aucune communication. Ainsi, il est préférable que les mouvements de données qui sont nécessaires puissent être optimisés. Pour cette raison, nous devons étudier les conditions qui permettront d'utiliser ces différents mouvements de données bien optimisés.

Dans ce chapitre nous n'étudions que les conditions qu'il faut satisfaire pour que les communications à engendrer soient des diffusions, des réductions ou des translations. La génération proprement dite des primitives entrainant ce type de communication est étudiée au chapitre suivant, section 5.2.

4.1 Optimisation des communications

J. Li et M. Cheng ([LC91]) distinguent cinq formes de mouvement de données sur les machines à mémoire distribuée : la permutation, la translation (ou communication uniforme) qui est un cas particulier de permutation, l'aggrégation (ou diffusion et réduction), la communication générale et la communication asynchrone.

Nous présentons dans cette section les trois mouvements de données que sont la translation, la diffusion et la réduction.

Il est important de noter que ces trois mouvements de données sont toujours effectués parallèlement aux dimensions des tableaux qu'ils utilisent. Ainsi, lorsque l'ensemble d'un tableau prend part à un tel mouvement de données, nous parlerons d'une communication **globale**. Par contre, lorsqu'une partie seulement du tableau intervient, elle doit être spécifiée par les dimensions utilisées. Nous parlons dans ce cas d'une communication **partielle**.

4.1.1 La diffusion

Une diffusion correspond au fait qu'une même cellule mémoire est lue par différentes instances (ou opérations) d'une même instruction. Si ces différentes instances sont exécutées sur des processeurs différents, il est nécessaire de communiquer à tous cette même valeur.

Soit une machine abstraite ayant une architecture en forme de grille de dimension D, chaque dimension étant de longueur N. Sur une telle machine, le coût théorique d'une diffusion d'une donnée de taille B parallèlement à une dimension de la grille est en $\mathcal{O}(B\log(N))$ (voir [GB91]). Par contre, en utilisant le mécanisme général de communication, ce coût théorique est en $\mathcal{O}(N.B)$, car l'énumération est nécessairement séquentielle au niveau du processeur qui émet la donnée.

Ainsi, lorsqu'une diffusion nécessite des données qui sont distribuées sur un certain nombre de processeurs, cela peut entraîner des communications importantes et coûteuses. La détection de ce type de communications peut alors s'avérer très profitable.

En fait, nous avons vu que Fortran 90 a défini une fonction intrinsèque SPREAD qui exécute ce type d'opération (voir section 1.2). Cette fonction ne réalise que des diffusions parallèlement à l'une des dimensions du tableau passé en paramètre.

La figure 4.1 donne les schémas de communication qui correspondent aux diffusions. Dans les sections suivantes, nous nous attacherons donc, quand cela est possible, à faire en sorte que toutes les diffusions détectées soient rendues globales ou partielles.

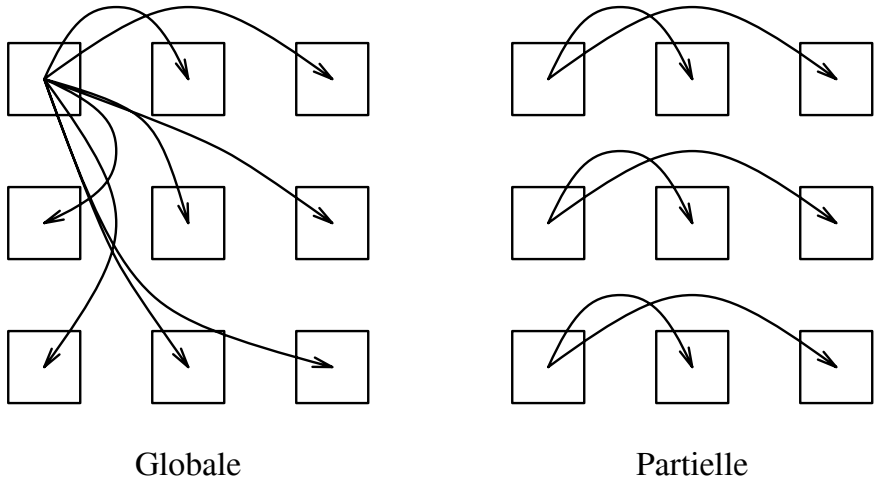


FIG. 4.1 - *Schéma de communication des diffusions*

L'extrait de programme suivant donne un exemple dans lequel il y a une diffusion :

```
do i=1,n
  do j=1,n
     a(i,j) = b(i)
  enddo
enddo
```

Cette diffusion pourrait être exprimée en Fortran 90 par l'instruction suivante :

```
a = SPREAD(b, DIM=2, NCOPIES=n)
```

4.1.2 La réduction

Une réduction est l'opération inverse de la diffusion ; elle correspond au fait que le calcul de la valeur d'une même cellule mémoire utilise les valeurs calculées par différentes instances d'instructions dans une opération associative (cette propriété est importante afin d'assurer que l'ordre dans lequel est réalisé la combinaison des valeurs reçues n'influence pas le résultat).

De même que pour une diffusion, le coût d'une communication de ce type est logarithmique.

Formellement, une réduction correspond à un système de récurrence à une équation d'ordre 1. Quelques méthodes de détection des réductions existent dans la littérature, notamment celle de P. Jouvelot & Dehbonei ([JD89]), ou celle de X. Redon ([Red92]). Cette dernière utilise notamment les informations très précises du DFG, et réalise en fait une détection des récurrences d'ordre quelconque.

En pratique, Fortran 90 a défini des fonctions intrinsèques pour réaliser des réductions avec addition (`SUM`), multiplication (`PRODUCT`), ou encore conjonction de booléen (`ALL`) (voir section 1.2). De même que pour le `SPREAD`, ces fonctions ne réalisent que des réductions parallèlement à l'une des dimensions du tableau passé en paramètre, *i.e.* parallélement à un axe de l'espace des processeurs.

La figure 4.2 donne les schémas de communication qui correspondent aux réductions.

L'extrait de programme suivant donne un exemple de réduction :

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

Cette réduction pourrait être exprimée en Fortran 90 sous la forme suivante :

```
s = SUM(a)
```

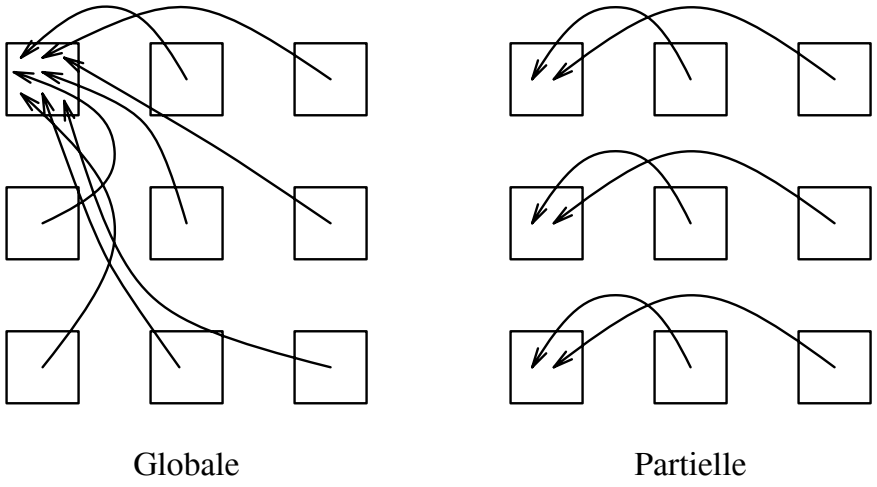


FIG. 4.2 - *Schéma de communication des réductions*

4.1.3 La translation

Une translation est un mouvement de données dans lequel tous les processeurs envoient une donnée sur la même direction de la grille et à la même distance.

Ce type de mouvement de données est implanté très efficacement dans la plupart des machines massivement parallèles car dans la pratique il correspond à une communication de voisin à voisin. Ce terme *voisin* vient des réseaux de type grille ou hypercube, mais peut également s'employer pour d'autre type de réseau comme le **fat-tree** de la CM-5 (voir section 5.3.1)[1].

La figure 4.3 définit les schémas de communication qui correspondent aux translations.

Dans l'algorithme de calcul de la fonction de placement (voir la section 4.5.5), nous prenons en compte cette propriété en essayant dans un premier temps de rendre constantes toutes les distances, *i.e.* indépendantes des compte-tours des boucles. Cette heuristique nous permet, dans les cas les

1. Dans ce cas, deux processeurs sont voisins si leur *routeur* de niveau 1 est le même. De plus, si les processeurs virtuels sont distribués par *bloc*, une majorité des communications voisin à voisin seront des opérations internes.

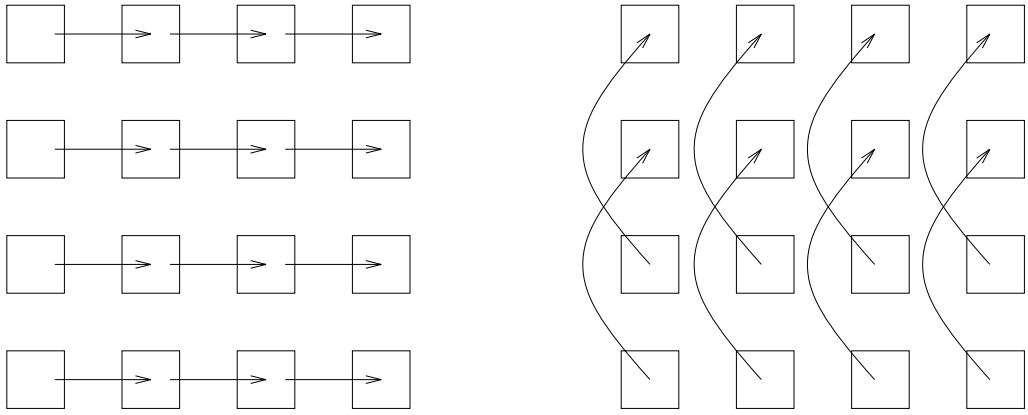


FIG. 4.3 - *Schéma de communication des translations*

plus favorables, de rendre constantes la plupart des distances.

Voici un exemple de programme dans lequel toutes les communications peuvent être mises sous la forme d'une translation :

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

En exprimant explicitement les translations, le code Fortran 90 suivant est équivalent :

```
 d = EOSHIFT(a,DIM=3,SHIFT=1) +
&    EOSHIFT(EOSHIFT(b,DIM=2,SHIFT=-1),DIM=3,SHIFT=-1) *
&    EOSHIFT(EOSHIFT(c,DIM=1,SHIFT=1),DIM=3,SHIFT=-1)
```

4.2 Détection et condition de diffusion

Nous avons vu section 4.1.1 qu'une diffusion correspond au fait qu'une même cellule mémoire est lue par différentes instances (ou opérations) d'une même instruction. Ainsi, un arc du DFG représente une diffusion si et seulement si pour un indice fixé de la source il existe plusieurs valeurs possibles pour les indices de la destination. Nous appellerons **espace de diffusion**, l'espace défini par toutes ces valeurs des indices de la destination, à indice de la source fixé.

A partir d'une modélisation sous forme d'espaces vectoriels (comme nous l'avons déjà fait section 3.4, page 80), nous disposons au départ d'un espace d'itération et d'un espace de diffusion (ses vecteurs de base, appelés **directions de diffusions**, sont exprimés en fonction de la base canonique de l'espace d'itération, c'est donc un sous-espace de ce dernier). Notre but est de déterminer un espace de distribution qui autorise le plus grand nombre de diffusion.

Nous devons donc faire en sorte que l'intersection entre l'espace de diffusion d'un arc de destination s et l'espace de distribution de s soit la plus grande possible (en terme de dimension).

Pour cela, nous proposons une résolution en deux étapes :

1. La première associe à chaque arc de diffusion une condition qui garantit pour celui-ci que l'espace de distribution de son instruction destination soit entièrement inclus dans la projection de son espace de diffusion (ce qui revient à dire que la projection de son espace de diffusion dans son espace de destination est de dimension pleine). Cette condition sera appelée **condition de diffusion globale**.

2. La seconde associe à chaque instruction une condition assurant que certaines directions de son espace de distribution soient parallèles à la projection des directions de diffusions des arcs dont elle est la destination et pour lesquels la condition de diffusion globale n'a pas été remplie. Cette condition sera appelée **condition de diffusion partielle**.

Prenons comme exemple une instruction fictive ayant un espace d'itération de dimension trois (I, J, K) et pour laquelle nous avons déterminé un espace de distribution à deux dimensions (P, Q) tel que :

$$P = I + J \ , \ Q = I - J$$

Notons $\vec{e_I}$, $\vec{e_J}$ et $\vec{e_K}$ les trois vecteurs de la base canonique de l'espace d'itération. Supposons, maintenant, que notre instruction fictive est la destination de quatre diffusions différentes : la première d'espace de diffusion définit par $(\vec{e_J})$, la seconde d'espace $(\vec{e_I}, \vec{e_J})$, la troisième d'espace $(\vec{e_I} - \vec{e_J})$ et la quatrième d'espace $(\vec{e_I} + \vec{e_J}, \vec{e_K})$.

La figure 4.4 donne pour chacune d'elle le type de diffusion qu'il sera possible d'engendrer sur l'espace de distribution.

Nous voyons donc que la première de ces quatre diffusions n'est pas réalisable en raison du fait qu'elle est partielle et non colinéaire à l'un des axes de l'espace de distribution. Des communications générales seront donc nécessaires pour réaliser ce mouvement de données.

De plus, la quatrième diffusion n'est réalisée que sur une seule dimension de l'espace de distribution car l'autre dimension correspond à une dimension écrasée (tous les buts de la diffusion selon cette dimension sont contenus dans le même processeur).

4.2.1 Notations

Comme nous venons de le voir, le traitement des diffusions ne s'intéresse qu'aux directions de distribution de la fonction de placement. Dans toute cette section, nous ne considérerons donc que la partie homogène de la fonction de placement.

Ainsi, pour une instruction s, nous noterons i_s l'ensemble des indices de boucle englobante, n_s la dimension de son espace d'itération (*i.e.* le nombre de boucles englobantes) et d_s la dimension de l'espace de distribution associé. Nous cherchons à déterminer la partie homogène du prototype de sa fonction de placement, qui est de la forme :

$$\pi_s^p(i_s) = \lambda^T . Q_s . i_s$$

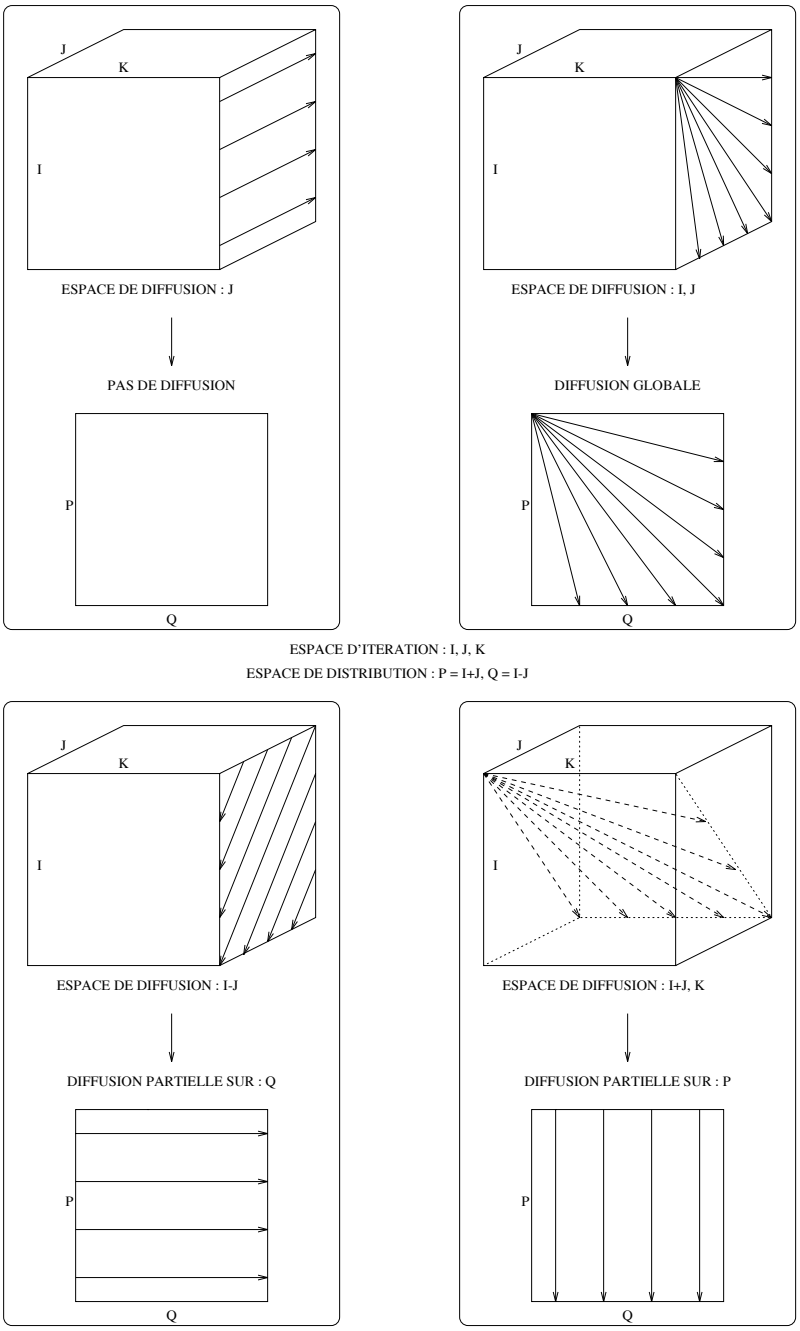


FIG. 4.4 - *Exemple de diffusions globales et partielles*

où λ est un vecteur de coefficients inconnus (de longueur l) et Q_s une matrice $(l \times n_s)$ à coefficients entiers. L'espace défini par ce prototype est engendré par les vecteurs lignes de Q_s ; ainsi, pour déterminer les directions de distribution, il faut calculer cette matrice Q_s.

Cette matrice Q_s va ainsi permetre de sélectionner les coefficients de λ qui apparaîtront dans l'expression du prototype de fonction de placement de l'instruction s [2].

Pour un programme donné, nous associons un tel prototype à chaque instruction et construisons notre liste λ. Comme nous avons vu au début de ce chapitre, la partie non homogène de notre prototype contient également des coefficients. En notant r le nombre de paramètres de structure (constant pour un programme donné) et S le nombre d'instruction de notre programme, nous pouvons en déduire la relation donnant le nombre total (l) de coefficients inconnus :

$$l = \sum_s n_s + S(r+1)$$

Notre algorithme procède par étapes successives, et, à chaque étape, certains coefficients sont déterminés en fonction des autres, ce qui a pour effet de modifier la taille et la valeur de Q_s. Au départ, Q_s est de rang [3] $p_s = n_s$; à la fin, elle est de rang $p_s = d_s$. Dans toute la suite, nous noterons b_s une famille de p_s vecteurs libres [4] pris parmi les n_s vecteurs lignes de Q_s. Cette famille b_s forme une base de l'espace de distribution engendré par le prototype, que nous appellerons **espace du prototype**.

> Pour notre programme `gauss`, nous avons à considérer cinq prototypes, qui sont de la forme :

2. La valeur de Q_s change à chaque étape de notre traitement ; par contre, la liste de coefficients λ n'est modifiée (sa longueur restant la même) que lors du traitement des diffusions partielles.

3. Le rang d'une matrice est égal à la dimension de son espace image, ou encore au nombre de vecteurs colonnes (ou lignes) linéairement indépendants.

4. Une famille de vecteurs $\{a_1, \ldots, a_n\}$ est dite libre, ou linéairement indépendante, si et seulement si : $\sum_{j=1}^{n} \alpha_j a_j = 0 \Leftrightarrow \alpha_1 = \ldots = \alpha_n = 0$.

$$
\begin{array}{ll}
\pi^p_{s1}(i,j,n) & = \lambda_2.i + \lambda_3.j + \lambda_4.n + \lambda_1 \\
\pi^p_{s2}(i,j,k,n) & = \lambda_6.i + \lambda_7.j + \lambda_8.k + \lambda_9.n + \lambda_5 \\
\pi^p_{s3}(i,n) & = \lambda_{11}.i + \lambda_{12}.n + \lambda_{10} \\
\pi^p_{s4}(i,j,n) & = \lambda_{14}.i + \lambda_{15}.j + \lambda_{16}.n + \lambda_{13} \\
\pi^p_{s5}(i,n) & = \lambda_{18}.i + \lambda_{19}.n + \lambda_{17}
\end{array}
$$

Notre ensemble λ est donc de taille 19. Notre matrice Q_{s2} est de taille (19×3) et sa valeur initiale est la suivante :

$$
Q^T_{s2} = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

4.2.2 Détection des diffusions

Le but de cette détection est de déterminer une base de l'espace de diffusion pour chaque arc du DFG.

Considérons un arc e du DFG entre une instruction source de vecteur d'itération i', et une instruction destination de vecteur d'itération i. La partie homogène de la transformation associée à cet arc peut être représentée par une matrice T de taille $(n' \times n)$ qui exprime la valeur de i' en fonction de i :

$$T.i = i' \tag{4.1}$$

Pour déterminer l'espace de diffusion de cet arc, nous fixons i' et nous cherchons toutes les valeurs possibles de i. Ce problème est donc équivalent au problème de la résolution d'un système d'équations linéaires en nombres entiers. Or les solutions d'un tel système (voir [Min83]) appartiennent à l'espace défini par le noyau[5] de T, $Ker(T)$. Nous appellerons β_e, une base de ce noyau, et **directions de diffusion** les vecteurs de cette base.

Donc, **l'espace de diffusion d'un arc est un translaté du noyau de la transformation associée**. Par conséquent, un arc définit une diffusion si et seulement si le noyau de sa transformation n'est pas nul.

5. Le noyau d'une matrice A est définie par : $Ker(A) = \{x \in R^n \mid Ax = 0\}$.

Nous appliquons une petite restriction lorsque certaines directions de diffusion sont incluses dans l'espace défini par la base de temps de l'instruction destination de e. Cet espace est défini par les expressions de la base de temps de l'instruction (ces directions sont les orthogonales aux fronts). L'espace défini par la base de temps donne donc les directions de l'espace d'itération sur lesquelles les calculs sont séquentiels.

Sur un front donné, toutes les instructions peuvent être exécutées en parallèle. Afin d'obtenir le plus de parallélisme, nous devons distribuer ces instructions sur le plus grand nombre possible de processeurs. Nous en déduisons qu'il est nécessaire que notre espace de distribution (il représente les directions sur lesquelles les calculs sont parallèles) ait une intersection vide avec l'espace définie par les bases de temps.

En outre, nous souhaitons avoir des directions de distribution qui correspondent à des directions de diffusion. Il ne faut donc pas tenir compte des directions de diffusion incluses dans l'espace défini par la base de temps, *i.e.* ces directions sont retirées de β_e.

> Prenons en exemple le programme `sequent` donné figure 4.5. La dépendance sur la variable x correspond à une diffusion selon i et j. De plus, la boucle sur i est séquentielle ($\theta_{s2}(i,j) = i - 1$), nous sommes donc dans le cas où une des direction de diffusion est parallèlement à une dimension séquentielle.
>
> Pour éviter de faire des communications il ne faut pas distribuer le tableau s selon sa première dimension, donc la diffusion selon i devient inutile car ses buts sont tous contenus dans le même processeur. Par contre, la diffusion selon j est conservée.

Dans la pratique, calculer le noyau de T revient à faire la résolution d'un système d'équations linéaires de la forme :

$$T.i = 0$$

Pour ce faire, nous pouvons par exemple utiliser la méthode d'élimination de Gauss (voir, par exemple, [GL83]) pour résoudre ce système.

En résumé, pour détecter si un arc e contient une diffusion, il suffit de considérer la matrice de transformation T, de calculer son noyau et d'en

```
      program sequent
      integer i,j,n
      real s(n,n), x

s1    x = s(1,1)
      do i = 1, n
        do j = 1, n
s2        s(i,j) = s(i-1, j) + x
        end do
      end do
      end
```

FIG. 4.5 - *Programme* `sequent`

déduire les vecteurs de base de l'espace de diffusion. Ces vecteurs s'expriment comme une combinaison linéaire des vecteurs de la base canonique de l'espace d'itération.

Pour notre programme `gauss`, prenons par exemple l'arc $e5$. Sa transformation est égale à la matrice identité, de noyau nul. L'espace de diffusion de $e5$ est donc nul.

Prenons maintenant l'arc $e3$. Sa transformation est :

$$T_{e3} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Dans ce cas précis, l'application de l'élimination de Gauss est triviale :

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} = 0 \Leftrightarrow \begin{cases} a = 0 \\ c = 0 \end{cases}$$

Ce qui signifie que tous les éléments du noyau de T_{e3} sont de la forme :

$$\begin{pmatrix} 0 \\ b \\ 0 \end{pmatrix}$$

Son noyau est donc de dimension 1, et le vecteur $(0,1,0)$ en est une base.

Finalement, nous pouvons déterminer que les arcs $e1, e3, e4$ et $e6$ ont un espace de diffusion non nul. Voici leur base respective (en fonction des vecteurs de la base canonique de l'espace d'itération de l'instruction destination de l'arc) :

$$\begin{aligned} \beta_{e1} &= \{(0,1)\} \\ \beta_{e3} &= \{(0,1,0)\} \\ \beta_{e4} &= \{(0,0,1)\} \\ \beta_{e6} &= \{(1,0)\} \end{aligned}$$

4.2.3 Diffusions globales

Soit un arc de diffusion e, d'instruction destination s. La diffusion qu'il représente caractérise le fait qu'un point de l'espace de distribution de la source de cet arc doit être diffusé sur un sous-espace (noté K_e) de l'espace de distribution de s. Ce sous-espace est le noyau de la transformation T_e de l'arc e et est de dimension k_e. Dire que cette diffusion est *globale* consiste donc à caractériser une distribution de s sur un tel sous-espace.

Or, nous avons vu que le prototype de s est de la forme (voir section 4.2.1) :

$$\pi_s^p = \lambda^T . Q_s . i_s$$

Nous pouvons trouver une nouvelle base ϵ_s de l'espace du prototype dans laquelle apparaissent tous les vecteurs de la base de K_e et complétée, si besoin est, par des vecteurs de b_s (théorème de la base incomplète, théorie sur les espaces vectoriels de dimension finie). En notant μ_s un ensemble de coefficients inconnus (de taille p_s), nous pouvons exprimer π_s^p en fonction de μ_s et ϵ_s :

$$\pi_s^p = \mu_s^T . \epsilon_s$$

La condition pour que la destination soit entièrement distribuée sur K_e est donc que les coefficients de μ_s, appliqués aux vecteurs de ϵ_s qui ne sont pas des directions de diffusion, soient nuls. En construisant ϵ_s de manière à placer en premier les vecteurs de base de K_e, nous pouvons exprimer la condition qui garantit que l'espace de distribution soit inclus dans l'espace

de diffusion $Ker(T_e)$ par l'annulation de la valeur du coefficient des $p_s - k_e$ derniers vecteurs de ϵ_s[6], *i.e.*, en notant $\mu_s = (\mu_1^s, \ldots, \mu_{p_s}^s)$:

$$\forall j \in \{k+1, \cdots, p_s\}, \mu_j^s = 0 \tag{4.2}$$

Il ne reste plus qu'à exprimer cette condition en fonction de λ. Or, en notant P_s la matrice de changement de base de b_s vers ϵ_s[7] et B_s la matrice $(p_s \times n_s)$ de rang p_s à coefficients entiers qui exprime les vecteurs de b_s en fonction de la base canonique[8], nous avons :

$$\left.\begin{array}{l} \pi_s^p = \mu_s^T.P_s.B_s.i_s \\ \pi_s^p = \lambda_s^T.Q_s.i_s \end{array}\right\} \Rightarrow \mu_s^T.P_s.B_s = \lambda^T.Q_s$$

Comme P_s et B_s sont de rang p_s, nous pouvons exprimer les p_s coefficients de μ_s en fonction des l coefficients de λ. En les remplaçant dans notre équation 4.2, nous obtenons $p_s - k_e$ conditions, pour l'arc e, en fonction de λ :

$$G_e.\lambda = 0$$

où G_e est une matrice $((p_s - k_e) \times n)$ à coefficients entiers.

Dans la pratique, nous pouvons déterminer la matrice G_e en utilisant par exemple la méthode d'élimination de Gauss.

Si toutes les équations d'un tel système sont satisfaites, alors nous pouvons garantir que l'espace de distribution de l'instruction destination de cet arc e est entièrement contenu dans l'espace de diffusion de ce dernier ; nous pourrons alors générer une diffusion sur tout l'espace de distribution (*i.e.* une diffusion globale).

En revanche, si une de ces équations ne peut pas être satisfaite, nous pouvons, au mieux, obtenir une distribution de l'instruction destination de l'arc sur un espace incluant l'espace de diffusion de ce dernier. De telles diffusions sont dites *partielles*. De plus, dans ce deuxième cas, l'arc doit être pris en compte dans le traitement des équations de coupure. En effet, nous devons essayer d'annuler la distance de tout arc de diffusion pour lequel

6. Si $p_s = k_e$, alors l'espace de distribution est égal à l'espace de diffusion.
7. $\epsilon_s = P_s.b_s$
8. $b_s = B_s.i_s$

l'instruction destination n'est pas à coup sûr distribuée entièrement sur un sous-espace de l'espace de diffusion.

Reprenons notre programme `gauss`, et calculons la condition de diffusion globale de l'arc $e4$, qui a pour espace de diffusion l'espace engendré par la famille β_{e4}.

Cet arc a pour instruction destination $s2$. Nous avons donné la valeur de Q_{s2} à la section 4.2.1. Nous avons donc :

$$\lambda^T.Q_{s2} = (\lambda_6\ \lambda_7\ \lambda_8)$$

Supposons que Q_{s2} n'a pas encore changée de valeur, dans ce cas b_{s2} est égale à la base canonique. Nous pouvons en déduire que la matrice B_{s2} est égale à la matrice identité.

Nous construisons ensuite la base ϵ_{s2}, en complétant la famille β_{e4} par des vecteurs de la base canonique de l'espace d'itération de $(s2)$:

$$\epsilon_{s2} = ((0,0,1),(1,0,0),(0,1,0))$$

Avec P_{s2}, la matrice de changement de base de la base canonique vers ϵ_{s2}, et $\mu_{s2} = (\mu_1, \mu_2, \mu_3)$, nous avons :

$$\mu_{s2}^T.P_{s2} = (\lambda_6\ \lambda_7\ \lambda_8) \Leftrightarrow \begin{pmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{pmatrix} = (P_{s2}^{-1})^T . \begin{pmatrix} \lambda_6 \\ \lambda_7 \\ \lambda_8 \end{pmatrix}$$

Il reste donc à calculer la transposée de l'inverse de P_{s2} :

$$(P_{s2}^{-1})^T = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Nous obtenons donc les relations suivantes :

$$\begin{cases} \mu_1 = \lambda_8 \\ \mu_2 = \lambda_6 \\ \mu_3 = \lambda_7 \end{cases}$$

La condition de diffusion globale de $e4$ s'exprime donc en annulant les $p_{s2} - k_{e4}$ $(3 - 1 = 2)$ derniers coefficients de μ_{s2}. Nous annulons

donc les deux derniers coefficients de μ_{s2}, (μ_2 et μ_3), ce qui donne la condition de diffusion globale de $e4$:

$$G_{e4}.\lambda = 0 \Leftrightarrow \begin{cases} \lambda_6 = 0 \\ \lambda_7 = 0 \end{cases}$$

Le même calcul pour les autres arcs donnent les systèmes suivants :

$$\begin{aligned} G_{e1}.\lambda = 0 &\Leftrightarrow \{ \lambda_2 = 0 \\ G_{e3}.\lambda = 0 &\Leftrightarrow \begin{cases} \lambda_6 = 0 \\ \lambda_8 = 0 \end{cases} \\ G_{e4}.\lambda = 0 &\Leftrightarrow \begin{cases} \lambda_6 = 0 \\ \lambda_7 = 0 \end{cases} \\ G_{e6}.\lambda = 0 &\Leftrightarrow \{ \lambda_{15} = 0 \end{aligned}$$

4.2.4 Diffusions partielles

Dans le cas de diffusions partielles, notre but est d'obtenir que certaines directions de l'espace de distribution soient parallèles à des directions de diffusion. L'idée est donc de conserver toutes les directions de diffusion partielle de chaque instruction, ce qui forme une famille de vecteurs (notée β_s).

Néanmoins, plusieurs arcs de diffusion peuvent avoir la même instruction de destination. Dans un tel cas, la famille de vecteurs n'est pas forcément libre et son nombre d'éléments n'est pas nécessairement inférieure à la dimension de l'espace de distribution de l'instruction. Il faut dans ce cas conserver un nombre limité de vecteurs, tels qu'ils forment une famille libre.

Se pose un premier problème pouvant entrainer une explosion combinatoire : comment choisir les directions de diffusion partielle de chaque instruction ?

Une heuristique simple est de construire cette famille au fur et à mesure que les conditions de diffusion sur les arcs sont traitées. Une direction donnée n'est alors acceptée que si elle répond au critère indiqué ci-dessus, *i.e.* le nombre maximal de directions n'a pas encore été atteint et elle n'est pas égale à une combinaison linéaire de celles déjà présentes.

Une fois cette famille de vecteurs construite pour chaque instruction, il reste à construire la condition de diffusion partielle qui en résulte. Bien en-

tendu, les instructions ayant un ensemble de directions de diffusion partielle vide ne sont pas traitées.

Notre but est de faire en sorte que certaines des directions de distribution soient parallèles aux directions de diffusion partielle. En effet, le principe de valuation des coefficients associés aux indices de boucle est de successivement tous les annuler sauf un. Si certains coefficients sont facteurs d'une direction de diffusion partielle, alors la valuation déterminera automatiquement certaines dimensions comme parallèles à une direction de diffusion partielle.

Le principe est proche de celui utilisé précédemment pour déterminer la condition de diffusion de chaque arc. Pour chaque instruction s, nous devons exprimer son prototype en fonction d'une nouvelle liste de coefficients inconnus μ_s, de taille p_s, chacun d'eux correspondant à une direction de diffusion partielle de β_s. Cette famille de vecteurs ne forme pas forcément une base de l'espace du prototype. Nous devons donc la compléter en une base ϵ_s, puis exprimer le prototype en fonction de celle-ci. Il reste alors à déterminer p_s équations qui expriment p_s coefficients de λ en fonction des autres et des coefficients de μ_s.

En reprenant les notations utilisées précédemment, nous avons donc :

$$\lambda^T.Q_s = \mu_s^T.P_s.B_s$$

Ce système est de rang p_s. Nous pouvons construire pour chaque instruction s un système de la forme :

$$L_s.\lambda = M_s.\mu_s$$

où L_s et M_s sont deux matrices à coefficients entiers de tailles respectives $(n_s \times l)$ et $(n_s \times p_s)$.

Un tel système permet donc d'exprimer p_s coefficients de λ en fonction des autres et des coefficients de μ_s.

> Reprenons notre programme `gauss` et supposons que notre traitement des conditions de diffusion ait valué à zéro trois coefficients, λ_2, λ_6 et λ_{15}, et ait construit la base de direction de diffusion partielle suivante :
>
> $$\beta_{s2} = \{(0,1,0),(0,0,1)\}$$

Il reste donc 16 coefficients inconnus et la partie homogène des prototypes associés à nos instructions ont maintenant la forme suivante :

$$\begin{array}{ll} \pi^p_{s1}(i,j) & = \lambda_3.j \\ \pi^p_{s2}(i,j,k) & = \lambda_7.j + \lambda_8.k \\ \pi^p_{s3}(i) & = \lambda_{11}.i \\ \pi^p_{s4}(i,j) & = \lambda_{14}.i \\ \pi^p_{s5}(i) & = \lambda_{18}.i \end{array}$$

La valeur de Q_{s2} a donc changé, nous avons :

$$\lambda^T.Q_{s2} = \begin{pmatrix} 0 & \lambda_7 & \lambda_8 \end{pmatrix}$$

Cette matrice est de rang 2, donc la matrice P_{s2} est égale à l'identité. Nous introduisons un nouveau couple de coefficients (μ_1, μ_2).

De plus, la matrice B_{s2} a la valeur suivante (les lignes correspondent aux vecteurs de β_{s2}) :

$$B_{s2} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Nous pouvons donc construire le système qui correspond aux conditions de diffusion partielle de cette instruction :

$$L_{s2}.\lambda = M_{s2}.\mu_{s2} \Leftrightarrow \begin{pmatrix} \lambda_7 \\ \lambda_8 \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}$$

4.3 Condition de réduction

Le schéma de communication engendré par une réduction est exactement l'inverse du schéma de communication engendré par une diffusion équivalente. Ainsi, en théorie, la condition de réduction peut être calculée d'une manière similaire à la condition de diffusion.

Par contre, en ce qui concerne la détection de ce type d'opération, nous avons vu que le problème est tout autre (voir section 4.1.2). Dans cette section nous considérons que le résultat d'une phase préalable de détection de réduction est disponible.

Dans ce cas, un arc de réduction est un arc dans lequel l'instruction source produit des valeurs qui sont utilisées pour une réduction. Une telle réduction se fait à partir d'un *espace de réduction*, qui est défini par des directions de réductions ; comme pour la diffusion, ces directions s'expriment comme une combinaison linéaire des vecteurs de la base canonique de l'espace d'itération.

Le but est donc de faire en sorte que l'intersection entre la projection par π_s de l'espace de réduction d'un arc de source s et l'espace de distribution de cette instruction soit la plus grande possible.

Nous pouvons donc reprendre la méthode de calcul des conditions de diffusion, en traitant cette fois l'instruction **source** de l'arc de réduction.

De plus, en pratique, tenir compte d'une réduction entraîne une modification de l'ordonnancement du programme. En effet, pour qu'une réduction puisse être exécutée en parallèle il est nécessaire que le calcul de la base de temps prenne en compte le résultat d'une phase de détection préalable, sinon les opérations effectuant cette réduction sont séquentialisées.

Etudions l'extrait de programme suivant :

```
s1   r = 0.
     DO i = 1, n
s2     r = r + a(i)
     ENDDO
```

Sans tenir compte de cette réduction, l'ordonnancement de ces deux instructions est le suivant :

$$\begin{aligned} \theta_{s1} &= 0 \\ \theta_{s2}(i) &= i \end{aligned}$$

Par contre, en considérant cette réduction comme une opération parallèle, cet ordonnancement devient :

$$\begin{aligned} \theta_{s1} &= 0 \\ \theta_{s2}(i) &= 1 \end{aligned}$$

Cette prise en compte des réductions pour le calcul de la base de temps a été traitée par X. Redon ([RF94]), et leur exploitation sur le calcul de la fonction de placement et la génération de code fait l'objet d'une étude menée par M. Barreteau ([BF95])[9].

4.4 Condition de translation

Une translation correspond à une communication parallèle aux axes et à une distance fixe. Or, nous avons vu (voir section 3.4) que chaque arc du DFG correspond à une communication potentielle. De plus, la distance de la communication associée à chaque arc peut être formalisée par une expression affine fonction des indices de boucles et des paramètres de structure.

Ainsi, la condition pour que la communication engendrée par un arc du DFG soit une translation est que la distance associée soit indépendante des indices de boucle.

Le but du calcul initial de la fonction de placement est d'annuler le plus possible de distances, ce que nous avons appelé le traitement des conditions de coupure.

A chaque arc peut être associée une condition de coupure qui est un système d'égalités à satisfaire pour que la distance de cet arc soit nulle.

Soit d_e la distance associée à l'arc e. En notant i l'ensemble des indices de boucle englobant l'instruction destination de cet arc, c l'ensemble des paramètres de structure, et, $\{f_i\}_{1\leq i\leq n}$ et $\{g_i\}_{0\leq i\leq r}$ des fonctions linéaires entières, nous avons :

$$d_e(i,c) = \sum_{j=1}^{n}(f_j(\lambda).i_j) + \sum_{j=1}^{r}(g_j(\lambda).c_j) + g_0(\lambda) \qquad (4.3)$$

La condition de coupure peut être exprimée par deux systèmes qui annulent respectivement les facteurs appliqués aux indices de boucle, et les facteurs appliqués aux paramètres de structure et au terme constant :

9. Cette étude vise également la prise en compte des "scans" (récurrences et réductions associative, opérations à préfixe parallèle).

$$M_e^i.\lambda = 0 \Leftrightarrow \{f_j(\lambda) = 0\}_{1 \leq j \leq n}$$

$$M_e^c.\lambda = 0 \Leftrightarrow \{g_j(\lambda) = 0\}_{0 \leq j \leq r}$$

Ainsi, si le premier système est satisfait, *i.e.* $M_e^i = 0$, alors la distance est constante, *i.e.* indépendante des indices de boucle.

De plus, si le second système est satisfait, *i.e.* $M_e^c = 0$, alors la distance est nulle.

La satisfaction de l'ensemble des équations formant la condition de coupures d'un arc peut donc s'avérer très restrictive. La méthode de résolution (voir section 4.5.5) propose donc d'essayer de satisfaire les équations une par une ; la distance est rendue constante dès que l'ensemble des équations du premier système ont été satisfaites.

4.5 Algorithme de calcul de la fonction de placement

Ce calcul se propose de déterminer pour chaque instruction d'un programme donné une fonction de placement multidimensionnelle linéaire entière.

Notre algorithme général se décompose en quatre grandes phases (mise à part l'initialisation) :

1. La première phase s'attache à détecter les arcs de diffusion. Le résultat associe à chaque arc les directions sur lesquelles se font les diffusions. La détection des diffusions est décrite en détail à la section 4.2.2.

2. La seconde phase traite les diffusions. Elle est divisé en deux étapes, décrites en détail à la section 4.2.3 :

   (a) La première étape consiste à faire en sorte que les diffusions qui ont été détectées soient globales (*i.e.* sur tout l'espace de distribution).

(b) La seconde étape traite le cas des diffusions partielles. Comme nous le verrons plus loin, cette deuxième étape impose des conditions sur les coefficients de λ qui sont facteurs des indices de boucle dans le prototype et introduit de nouveaux coefficients.

3. La troisième phase est le traitement des équations de coupure. Son principe est identique à celui présenté par Feautrier dans [Fea93]. Contrairement aux phases précédentes, celle-ci impose des conditions sur tous les coefficients des prototypes. En effet, pour rendre nulle la distance d'un arc (voir ci-dessus), peuvent intervenir des conditions sur les coefficients de λ qui sont facteurs des paramètres de structure. C'est dans cette phase qu'intervient la condition de translation.

4. La quatrième et dernière phase, appelée *valuation*, consiste à déterminer la valeur finale de chaque dimension de la fonction de placement pour chaque instruction. C'est donc elle qui détermine les différentes directions de distribution des instructions (par rapport aux indices de boucle) ainsi que leur décalage les unes par rapport aux autres. Les directions de distribution sont les combinaisons linéaires d'indices de boucle associées à chaque coefficient.

Au départ, ce calcul suppose connu un graphe du flot des données du programme, ainsi qu'une base de temps. De plus, pour chaque instruction doit être associé un prototype de fonction de placement (voir la section 3.4) qui représente une dimension de celle-ci. Ces prototypes initiaux associent à chaque indice de boucle englobante et paramètre de structure un coefficient inconnu.

Les deux phases intermédiaires s'attachent à restreindre l'espace des prototypes et à déterminer des relations entre eux. Pour cela, nous utilisons une liste de coefficients dit "à déterminer", notée λ, et une substitution qui exprime les coefficients dits "déterminés" en fonction des autres (*i.e.*, des coefficients non déterminés) et de constantes entières, notée σ.

Algorithme 9

1. *Initialisation : calcul des dimensions de la fonction de placement pour chaque instruction ; détection des diffusions par l'application de l'algo-*

rithme 10 (page 126) ; calcul du poids de chaque arc ; construction des prototypes initiaux et de λ ; initialisation de σ à $\emptyset$;

2. *Conditions de diffusion globale : traitement sur les arcs ayant un espace de diffusion non nul par application de l'algorithme 11 (page 127) ;*

3. *Conditions de diffusion partielle : traitement sur les instructions ayant un espace de diffusion partielle non nul avec l'application de l'algorithme 13 (page 131) ;*

4. *Conditions de coupure : traitement sur les arcs dont toutes les conditions de diffusion globale n'ont pas été satisfaites avec l'application de l'algorithme 12 (page 128) ;*

5. *Valuation : détermination de la valeur de chaque dimension de la fonction de placement avec l'application de l'algorithme 14 (page 137).*

4.5.1 Initialisation

L'initialisation est décomposée en trois étapes, le calcul de la dimension de la fonction de placement, le calcul du poids de chaque arc et la construction des prototypes initiaux.

Dimension

La dimension de la fonction de placement donne la dimension de la grille de processeurs virtuels sur laquelle nous distribuons les données et les opérations d'un programme donné. Cette dimension est déterminée en fonction de la machine cible, car toutes les architectures n'acceptent pas une grille de processeurs virtuels de dimension quelconque. De plus un autre problème provient du fait que certains fronts du programme peuvent ne pas avoir assez de dimensions pour remplir la grille. Dans [Fea93], Feautrier propose deux solutions pour résoudre ce dernier problème.

La première, si l'architecture le permet, est de réarranger la grille selon les instructions afin de ne pas avoir de processeurs à l'état de repos (*i.e.*, oisif). Le désavantage de cette première solution est qu'elle va entraîner des changements non linéaires et donc la minimisation des communications sera difficile.

L'autre solution est d'utiliser la même géométrie pour toutes les instructions, certains processeurs devenant oisifs si besoin est.

Dans tous les cas, il faut calculer pour chaque instruction s la dimension maximale (notée d_s) de l'espace de distribution de cette instruction. Cette dimension maximale est égale à la différence entre la dimension de l'espace d'itération et la dimension de la base de temps de cette instruction.

Ainsi, si nous choisissons d'utiliser la même géométrie pour toutes les instructions, notre fonction de placement aura pour dimension la valeur maximale des dimensions des espaces de distribution associées à chaque instruction ($d_{max} = MAX_s(d_s)$). Mais, parmi ces d_{max} dimensions, une instruction s n'aura, au plus, que d_s dimensions indépendantes (par rapport aux indices de boucle).

> Pour notre programme `gauss`, avec la base de temps que nous avons donnée plus haut (voir section 3.3), nous pouvons déterminer la valeur de la dimension pour chaque instruction :
>
> $$\begin{cases} d_{s1} = & 1 \\ d_{s2} = & 2 \\ d_{s3} = & 1 \\ d_{s4} = & 1 \\ d_{s5} = & 0 \end{cases}$$

Poids

Le poids d'un arc est utilisé pour ordonner les arcs. Défini dans [Fea93], il est égal à la dimension du polyèdre des transferts, *i.e.* la dimension de l'image par la transformation de l'arc de son domaine d'itération.

Dans le cas général, l'algorithme de calcul du poids d'un arc est assez lourd, nous proposons donc une méthode simple pour calculer le poids d'un arc lorsque celui-ci définit une diffusion :

le poids d'un arc de diffusion est calculé simplement puisqu'il est égal à la dimension de l'espace sur lequel se fait la diffusion ;

> Dans notre programme Gauss, nous avons détecté quatre arcs de diffusion ($e1, e3, e4, e6$) : le poids de ces arcs est égal à 1.

le poids d'un arc par la méthode générale est égal à la dimension de l'ensemble E_e qui est l'image par la transformation h_e du domaine d'itération P_e :

$$E_e = \{y \mid \exists y = h_e(x), x \in P_e\}$$

où y est le vecteur donnant la valeur des indices de boucle englobante de l'instruction source, et x est le vecteur des indices de boucle englobante de l'instruction destination.

Dans le programme `gauss`, huit arcs ont un espace de diffusion nul, $(e2, e5, e7, e8, e9, e10, e11, e12)$. Calculons, par exemple, le poids de $e5$.

Pour cela, nous devons calculer la dimension de l'ensemble E_{e5} défini par :

$$\exists(i', j', k')/ \begin{cases} i' = i - 1 \\ j' = j \\ k' = k \end{cases} et \begin{cases} 1 \leq i \leq n - 1 \\ i + 1 \leq j \leq n \\ i + 1 \leq k \leq n \\ i - 2 \geq 0 \end{cases}$$

Nous pouvons alors en déduire que E_{e1} est défini par :

$$\exists(i', j', k')/ \begin{cases} 0 \leq i' \leq n - 2 \\ i' + 2 \leq j' \leq n \\ i' + 2 \leq k' \leq n \\ i' - 1 \geq 0 \end{cases}$$

Comme ce système ne contient aucune équation implicite, nous pouvons en déduire que sa dimension est égale à 3. Donc le poids de $e5$ est 3.

De la même manière, nous pouvons calculer que le poids de $e2, e7$ et $e8$ est égal à 2, le poids de $e9, e10$ et $e11$ est égal à 1, et le poids de $e12$ est égal à 0.

Cette méthode de calcul du poids d'un arc a le mérite d'être assez simple, mais elle n'est pas très précise. Il existe d'autres méthodes plus complexes qui permettraient de réaliser cette classification des arcs de manière plus fine.

Ainsi, Thomas Fahringer propose un outil de prédiction de performance paramétré ("Parameter based Performance Prediction Tool", [Fah93])qui, afin d'examiner par exemple si un schéma donné de distribution améliore les performances, calcule à la compilation un certain nombre de paramètres du programme parallèle ; parmis ces paramètres figurent, notamment, le nombre de transferts, le volume de données transféré, les temps de transfert ou encore la charge du réseau.

Une autre méhode consisterait à faire un comptage symbolique du nombre d'éléments contenu dans un polyèdre, ce qui se ramène à un problème de somme symbolique de polynômes. Ce type de problème fait l'objet d'études très poussées depuis quelques années (voir [Taw90, PW92, Pug94, Zho94]).

D'après quelques expériences, les modifications dans l'ordre des arcs que l'utilisation de ces méthodes pourraient entraîner auraient un impact assez faible sur le résultat final. En résumé, une classification plus fine n'améliorerait pas le résultat tout en augmentant la complexité de notre méthode.

Prototypes

A chaque instruction s est associé un prototype π_s^p (voir section 3.4, page 80).

En notant i l'ensemble des indices de boucle englobant l'instruction s, c l'ensemble des paramètres de structure, et $\lambda^s = \{\lambda_j^s\}_{0 \leq j \leq n_s+r}$ une liste de coefficients inconnus, nous avons :

$$\pi_s^p(i) = \sum_{j=1}^{n_s} (\lambda_j^s . i_j) + \sum_{j=1}^{r} (\lambda_{j+n_s}^s . c_j) + \lambda_0^s \tag{4.4}$$

Notre liste de coefficients λ correspond donc à la réunion des λ^s.

4.5.2 Détection des diffusions

A chaque arc e du DFG nous devons associer un espace de diffusion. Nous caractérisons cet espace par une base notée β_e.

Algorithme 10 *Cet algorithme prend en entrée la liste des arcs du DFG et associe à chacun un espace de diffusion défini par des vecteurs.*

1. *Extraire la matrice de transformation T_e de l'arc courant e ;*
2. *Calculer le noyau $Ker(T_e)$ de cette matrice ;*
3. *Déduire de $Ker(T_e)$ la valeur d'une de ses bases β_e contenant les vecteurs de base de l'espace de diffusion de e ;*
4. *Eliminer tous les vecteurs de β_e qui sont inclus dans l'espace défini par la base de temps.*

Nous partitionnons la liste des arcs en deux, avec d'une part les arcs de diffusions D, et d'autre part les autres arcs C (ceux pour lesquels nous avons : $\beta_e = \emptyset$).

4.5.3 Diffusions globales

A chaque arc e de D, nous pouvons associer une condition de diffusion globale exprimée sous la forme d'un système d'équations (voir section 4.2.3) :

$$G_e.\lambda = 0$$

Le but de notre traitement est de "satisfaire" le plus grand nombre possible de ces conditions. L'algorithme 12 qui réalise le test de satisfaction d'une condition est donné plus loin[10]. Voici notre algorithme de traitement des arcs de diffusion, triés par poids décroissant :

Algorithme 11 *Cet algorithme prend en entrée λ et σ et modifie leur valeur. A chaque instruction est associée une famille de vecteurs β_s, vide initialement, qui correspond aux directions de diffusion partielle. Nous effectuons une boucle sur la liste D :*

1. *extraire la matrice G_e associée à l'arc courant e ;*

10. Cet algorithme prend en entrée un système $M_E.\lambda_E = 0$.

2. *lui appliquer l'**algorithme 12** avec* $M_E = G_e$ *et* $\lambda_E = \lambda$ *;*

3. *si au moins une équation de* $G_e.\lambda = 0$ *n'a pas été satisfaite alors :*

   - *ajouter* e *à* C *;*
   - *si* β_s *(*s *étant l'instruction destination de* e*) possède strictement moins de* d_s *vecteurs, alors lui ajouter les directions de diffusion partielle de* e *qui ne sont pas incluses dans l'espace défini par celles déjà présentes.*

4. *enlever de* λ *tous les coefficients qui sont substitués par* σ_E *;*

5. *la nouvelle valeur de* σ *est* $\sigma \circ \sigma_E$*.*

Cet algorithme terminé, toute instruction s pour laquelle la famille β_s n'est pas vide est une instruction ayant des diffusions partielles.

Satisfaction d'une condition

Nous définissons la satisfaction d'une condition (*i.e.*, d'un système d'équations) comme équivalente à la satisfaction de toutes ses équations. Pour cela, nous dirons qu'une équation est satisfaite si et seulement si elle ne rend pas triviale la fonction de placement, *i.e.* si aucun prototype n'est rendu trivial.

Dans [Fea93] est présenté un algorithme pour le calcul de la fonction de placement dont le cœur (appelé E) consiste à essayer de satisfaire une par une les équations contenues dans une liste ordonnée. Voici cet algorithme :

Algorithme 12 *Il s'agit du traitement d'une liste ordonnée d'équations à satisfaire :* $M_E.\lambda_E = 0$*. La solution de cet algorithme est construite au fur et à mesure sous la forme d'une substitution* σ_E*, initialisée à* $\emptyset$*, qui exprime certaines variables de* λ_E *en fonction des autres :*

1. *appliquer* σ_E *à l'équation courante. Nous obtenons une équation de la forme :* $f(\lambda_E) = \sum_{j=1}^{l} c_j.\lambda_j = 0$ *;*

2. *trouver un entier* j_p *tel que* $c_{j_p} \neq 0$ *;*

3. *construire la substitution élémentaire :* $\tau = [\lambda_{j_p} \leftarrow \frac{f(\lambda_E) - c_{j_p}.\lambda_{j_p}}{c_{j_p}}]$

4. *en déduire une nouvelle substitution :* $\sigma'_E = \sigma_E \circ \tau$;

5. *Test de trivialité : appliquer* σ'_E *à tous les prototypes et si aucun n'est devenu trivial, alors faire :* $\sigma_E = \sigma'_E$

Test de trivialité

Nous dirons qu'un prototype (associé à une instruction s) n'est pas trivial si et seulement si il dépend d'un nombre suffisant d'indices de boucle de telle sorte que l'on puisse construire le nombre requis de solutions linéairement indépendantes, *i.e.* si et seulement si la *dimension du prototype* est supérieure ou égale à d_s.

En ne considérant que sa partie homogène (voir section 4.2.1), notre prototype peut être représenté comme suit :

$$\pi_s^p = \lambda^T.Q_s$$

où Q_s est une matrice $(l \times n_s)$ à coefficients entiers.

Avec cette représentation, nous avons trivialement que le nombre d'indices de boucles dont dépend π_s^p est égal au nombre de vecteurs lignes libres de Q_s, *i.e.* est égal au rang de la matrice Q_s. Nous avons donc le résultat suivant : **un prototype $\lambda^T.Q_s$ est trivial si et seulement si** $rang(Q_s) < d_s$.

> Appliquons l'algorithme 11 aux arcs de diffusion de notre programme `gauss`, triés par poids décroissant : $D = (e6, e1, e4, e3)$. La liste des autres arcs a la valeur suivante (triée par poids décroissant) : $C = (e5, e8, e2, e7, e11, e9, e10, e12)$. Au départ, $\sigma = \emptyset$. De plus, ces quatre arcs ont pour instruction destination soit $s1$ soit $s2$ soit $s4$, nous devons donc initialiser à vide trois familles de vecteurs β_{s1}, β_{s2} et β_{s4}. Nous traitons un par un chacun des arcs de D :
>
> – Le premier arc à traiter est $e6$, le système associé est : $\{\lambda_{15} = 0\}$. Nous lui appliquons l'algorithme 12, *i.e.* nous traitons une par une les équations du système.

La seule équation est $\lambda_{15} = 0$. Nous avons $\sigma' = [\lambda_{15} \leftarrow 0]$. Seul π^p_{s4} est touché par cette σ', et étant donné que $d_{s4} = 1$ il n'est pas rendu trivial :

$$\pi^p_{s4}(i,j) = \lambda_{14}.i$$

Notre substitution est donc mise à jour : $\sigma = [\lambda_{15} \leftarrow 0]$, et λ_{15} est retiré de λ.

L'algorithme 12 est terminé, la substitution qui en résulte est :

$$\sigma = [\lambda_{15} \leftarrow 0]$$

Toutes les équations ont été satisfaites, nous considérons donc que la condition de diffusion globale de l'arc $e6$ a été satisfaite. Cet arc n'est donc pas ajouté à la liste C, et β_{s4} reste inchangée.

- Le deuxième arc à traiter est $e1$, le système associé est : $\{\lambda_2 = 0\}$. L'application de l'algorithme 12 satisfait cette équation, la substitution a changé : $\sigma = [\lambda_{15} \leftarrow 0, \lambda_2 \leftarrow 0]$ et λ_2 est retirée de λ. Cet arc n'est donc pas ajouté à la liste C, et β_{s1} reste inchangée.
- le troisième arc à traiter est $e4$, le système associé est : $\{\lambda_6 = 0, \lambda_7 = 0\}$. L'application de l'algorithme 12 satisfait une seule de ces équations $\{\lambda_6 = 0\}$, car seul π^p_{s2} est touché par ce système et $d_{s2} = 2$. La substitution a donc changé : $\sigma = [\lambda_{15} \leftarrow 0, \lambda_2 \leftarrow 0, \lambda_6 \leftarrow 0]$, et λ_6 est retirée de λ. De plus, cet arc est rajouté à la liste C et β_{s2} est modifiée : $\beta_{s2} = \{(0,0,1)\}$.
- Le quatrième et dernier arc est $e3$, le système associé est : $\{\lambda_6 = 0, \lambda_8 = 0\}$. De même que précédemment, l'application de l'algorithme 12 satisfait une seule de ces équations $\{\lambda_6 = 0\}$. Cette équation est déjà présente dans σ donc cette substitution reste inchangée. Par contre, cet arc est rajouté à la liste C, et β_{s2} est modifiée : $\beta_{s2} = \{(0,1,0),(0,0,1)\}$.

Finalement, le traitement des conditions de diffusion globale a permis de donner une valeur à trois variables :

$$\sigma = [\lambda_2 \leftarrow 0, \lambda_6 \leftarrow 0, \lambda_{15} \leftarrow 0]$$

De plus, la liste C a changé :

$$C = (e5, e8, e2, e7, e4, e3, e11, e9, e10, e12)$$

Enfin, nous avons une famille de directions de diffusion qui n'est pas vide :

$$\beta_{s2} = \{(0,1,0),(0,0,1)\}$$

Il ne nous reste donc plus que 16 coefficients inconnus et les prototypes sont donc maintenant les suivants :

$$\begin{array}{ll} \pi^p_{s1}(i,j) & = \lambda_3.j + \lambda_4.n + \lambda_1 \\ \pi^p_{s2}(i,j,k) & = \lambda_7.j + \lambda_8.k + \lambda_9.n + \lambda_5 \\ \pi^p_{s3}(i) & = \lambda_{11}.i + \lambda_{12}.n + \lambda_{10} \\ \pi^p_{s4}(i,j) & = \lambda_{14}.i + \lambda_{16}.n + \lambda_{13} \\ \pi^p_{s5}(i) & = \lambda_{18}.i + \lambda_{19}.n + \lambda_{17} \end{array}$$

4.5.4 Diffusions partielles

Nous avons vu à la section précédente qu'à chaque instruction est associée une famille de direction de diffusion partielle. Soit s une instruction et β_s sa famille de direction de diffusion partielle, nous pouvons alors exprimer la condition de diffusion partielle de cette instruction sous la forme d'un système d'équations (voir la section 4.2.4) :

$$L_s.\lambda = M_s.\mu_s$$

Algorithme 13 *Nous effectuons une boucle sur la liste des instructions pour lesquelles nous avons détecté des diffusions partielles :*

- *construire le système associé à l'instruction courante s :*

$$L_s.\lambda = M_s.\mu_s$$

- *à partir de ce système, éliminer des coefficients de λ, en déduire la substitution σ' qui en résulte ;*
- *appliquer σ' à tous les prototypes ;*
- *en déduire la nouvelle valeur de σ : $\sigma \circ \sigma'$.*

Cet algorithme doit être appliqué à chaque instruction ayant une liste de direction de diffusion partielle non vide. Pour chacune d'elles, cela introduit de nouveaux coefficients (notés μ_i) qui remplacent certains coefficients de λ. Il reste alors à traiter les conditions de coupure.

Revenons à notre programme `gauss`. Seule l' instruction $s2$ est traitée puisque seule la famille β_{s2} n'est pas vide. Appliquons lui l'algorithme 13 :

Nous avons donné (voir section 4.2.4) la valeur du système :

$$L_{s2}.\lambda = M_{s2}.\mu_{s2}$$

Il a la valeur suivante :

$$\begin{pmatrix} \lambda_7 \\ \lambda_8 \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}$$

Notre substitution σ' a donc pour valeur :

$$[\lambda_7 \leftarrow \mu_1,\ \lambda_8 \leftarrow \mu_2]$$

Notre famille λ contient donc toujours 16 coefficients inconnus, deux d'entre eux (μ_1, μ_2) ayant été substitués à (λ_7, λ_8).

Les prototypes ont donc maintenant les formes suivantes :

$$\begin{array}{ll}
\pi^p_{s1}(i,j) & = \lambda_3.j + \lambda_4.n + \lambda_1 \\
\pi^p_{s2}(i,j,k) & = \mu_1.j + \mu_2.k + \lambda_9.n + \lambda_5 \\
\pi^p_{s3}(i) & = \lambda_{11}.i + \lambda_{12}.n + \lambda_{10} \\
\pi^p_{s4}(i,j) & = \lambda_{14}.i + \lambda_{16}.n + \lambda_{13} \\
\pi^p_{s5}(i) & = \lambda_{18}.i + \lambda_{19}.n + \lambda_{17}
\end{array}$$

4.5.5 Conditions de coupure

La liste C, éventuellement augmentée des arcs de D dont la condition de diffusion n'a pas été satisfaite, est triée par poids décroissant. Après le traitement des diffusions, de nouveaux coefficients ont pu être introduits. Néanmoins, nous gardons la même notation, *i.e.*, l'ensemble des coefficients apparaissant dans les prototypes est appelé λ.

A chaque arc e de C est associée une distance (voir section 3.4), qui a la forme suivante :

$$d_e(x) = \pi_{\delta(e)}(x) - \pi_{\sigma(e)}(h_e(x))$$

Reprenons notre exemple sur le programme `gauss`. La liste C a été augmentée de deux arcs $(e4, e3)$, qui sont tous de poids 1. Nous trions cette liste par poids décroissants, ce qui nous donne (par exemple) :

$$C = (e5, e8, e2, e7, e4, e3, e11, e9, e10, e12)$$

Calculons maintenant pour chaque arc de C la distance qui lui est associée.

Prenons, par exemple, l'arc $e5$. La formule générale est :

$$d_{e5}(i,j,k) = \pi^p_{s2}(i,j,k) - \pi^p_{s2}(h_{e5}(i,j,k))$$

Le tableau 3.2 donne :

$$h_{e5}(i,j,k) = (i-1,j,k)$$

Nous en déduisons donc :

$$d_{e5}(i,j,k) = \pi^p_{s2}(i,j,k) - \pi^p_{s2}(i-1,j,k)$$

En remplaçant les prototypes par leur valeur, nous avons donc :

$$d_{e5}(i,j,k) = 0$$

Toutes les autres distances se calculent de la même manière. Voici leur valeurs (nous les donnons dans le même ordre que C, les distances de $(e1, e6)$ n'apparaissent pas car ces arcs ne sont pas dans C) :

$$\begin{array}{ll}
d_{e5}(i,j,k) = & 0 \\
d_{e8}(i,j) = & 0 \\
d_{e2}(i,j) = & -\mu_1.i + (\lambda_3 - \mu_2).j + (\lambda_4 - \lambda_9).n + \lambda_1 - \lambda_5 \\
d_{e7}(i,j) = & (\mu_2 + \lambda_{14}).i + \mu_1.j + (\lambda_{16} - \lambda_9 - \mu_1 - \mu_2).n + \lambda_{13} - \lambda_5 - \mu_1 - \mu_2 \\
d_{e4}(i,j,k) = & (\mu_2 - \lambda_3).j + \mu_1.k + (\lambda_9 - \lambda_4).n + \lambda_5 - \lambda_1 \\
d_{e3}(i,j,k) = & -\mu_2.i + \mu_2.j \\
d_{e11}(i) = & (\lambda_{18} - \lambda_{14}).i + (\lambda_{19} - \lambda_{16}).n + \lambda_{17} - \lambda_{13} \\
d_{e9}(i,j) = & (\lambda_{14} - \lambda_{11}).i + (\lambda_{16} - \lambda_{12}).n + \lambda_{13} - \lambda_{10} \\
d_{e10}(i) = & (\mu_2 + \mu_1 + \lambda_{18}).i + (\lambda_{19} - \lambda_9 - \mu_1 - \mu_2).n + \lambda_{17} - \lambda_5 - \mu_1 - \mu_2 \\
d_{e12}(i) = & (\lambda_{18} - \lambda_{11}).i + (\lambda_{19} - \lambda_{12}).n + \lambda_{17} - \lambda_{10}
\end{array}$$

Annuler une distance est équivalent à annuler les facteurs des indices de boucle, des paramètres de structure et le terme constant, s'il n'existe pas de relation entre eux. Pour en être sûr, il nous faut remplacer tous les indices et paramètres qui s'expriment en fonction des autres. Pour cela, chaque distance d_e a pour ensemble de définition un polyèdre P_e qui est l'intersection du prédicat d'existence de l'arc e avec le domaine d'exécution de l'instruction destination de cet arc. Nous devons rechercher les équations implicites des P_e, qui nous permettront d'éliminer certains indices ou paramètres.

> Pour notre programme `gauss`, le tableau 3.2 nous permet de voir que les ensembles P_{e9} et P_{e12} contiennent chacun une équation implicite, respectivement $j = 1$ et $i = 1$. Dans les distances qui leurs correspondent, nous remplaçons donc la variable par la valeur donnée par l'équation implicite. Seule d_{e12} change, sa nouvelle valeur est la suivante :
>
> $$d_{e12}(i) = (\lambda_{19} - \lambda_{12}).n + \lambda_{18} - \lambda_{11} - \lambda_{10} + \lambda_{17}$$

De plus, afin de satisfaire la condition de translation (voir section 4.4), nous proposons dans un premier temps de rendre constantes le plus grand nombre de distances. Puis dans un deuxième temps nous proposons de les annuler. Pour cela, il nous faut considérer les deux systèmes qui annulent respectivement les facteurs appliqués aux indices de boucle, et les facteurs appliqués aux paramètres de structure et au terme constant :

$$M_e^i.\lambda = 0, M_e^c.\lambda = 0$$

Nous construisons donc un système $M.\lambda = 0$ ordonné en faisant l'union de tous ces systèmes de la manière suivante[11] :

1. tout d'abord l'ensemble des matrices $\{M_e^i\}_{e \in C}$ ordonné par poids décroissants ;
2. puis l'ensemble des matrices $\{M_e^c\}_{e \in C}$ également ordonné par poids décroissants.

11. Le poids d'une matrice M_e^i ou M_e^c est égal au poids de l'arc e.

Le traitement des équations de coupure est alors réalisé à l'aide de **l'algorithme 12**, avec $M_E = M$ et $\lambda_E = \lambda$. Cet algorithme terminé, les coefficients substitués par σ_E sont enlevés de λ. La nouvelle valeur de σ est : $\sigma \circ \sigma_E$.

Il reste alors à construire les différentes dimensions de la fonction de placement.

Construisons les conditions de coupure pour notre programme `gauss`. Pour chaque arc e de C, il faut construire les systèmes M_e^i et M_e^c, qui constituent notre système $M.\lambda = 0$:

$$M_{e5}^i = \emptyset \qquad M_{e5}^c = \emptyset$$

$$M_{e8}^i = \emptyset \qquad M_{e8}^c = \emptyset$$

$$M_{e2}^i = \begin{cases} \lambda_3 - \mu_2 = 0 \\ \mu_1 = 0 \end{cases} \qquad M_{e2}^c = \begin{cases} \lambda_1 - \lambda_5 = 0 \\ \lambda_4 - \lambda_9 = 0 \end{cases}$$

$$M_{e7}^i \begin{cases} \mu_2 + \lambda_{14} = 0 \\ \mu_1 = 0 \end{cases} \qquad M_{e7}^c \begin{cases} \lambda_{13} - \lambda_5 - \mu_1 - \mu_2 = 0 \\ \lambda_{16} - \lambda_9 - \mu_1 - \mu_2 = 0 \end{cases}$$

$$M_{e4}^i = \begin{cases} \mu_2 - \lambda_3 = 0 \\ \mu_1 = 0 \end{cases} \qquad M_{e4}^c = \begin{cases} \lambda_5 - \lambda_1 = 0 \\ \lambda_9 - \lambda_4 = 0 \end{cases}$$

$$M_{e3}^i = \begin{cases} \mu_2 = 0 \end{cases} \qquad M_{e3}^c = \emptyset$$

$$M_{e11}^i = \begin{cases} \lambda_{18} - \lambda_{14} = 0 \end{cases} \qquad M_{e11}^c = \begin{cases} \lambda_{17} - \lambda_{13} = 0 \\ \lambda_{19} - \lambda_{16} = 0 \end{cases}$$

$$M_{e9}^i \begin{cases} \lambda_{14} - \lambda_{11} = 0 \end{cases} \qquad M_{e9}^c \begin{cases} \lambda_{13} - \lambda_{10} = 0 \\ \lambda_{16} - \lambda_{12} = 0 \end{cases}$$

$$M_{e10}^i = \begin{cases} \mu_2 + \mu_1 + \lambda_{18} = 0 \end{cases} \qquad M_{e10}^c = \begin{cases} \lambda_{17} - \lambda_5 - \mu_1 - \mu_2 = 0 \\ \lambda_{19} - \lambda_9 - \mu_1 - \mu_2 = 0 \end{cases}$$

$$M_{e12}^i = \emptyset \qquad M_{e12}^c = \begin{cases} \lambda_{18} - \lambda_{11} - \lambda_{10} + \lambda_{17} = 0 \\ \lambda_{19} - \lambda_{12} = 0 \end{cases}$$

Appliquons maintenant l'algorithme 12. Seules trois équations ne sont pas satisfaites par cet algorithme :

$$\begin{cases} \mu_1 = 0 \\ \mu_2 = 0 \\ \mu_1 + \mu_2 + \lambda_{18} = 0 \end{cases}$$

Ces trois équations viennent des distances de $e2, e3, e4, e7, e10$, qui sont donc les seules à ne pas être annulées.

L'annulation de toutes les autres équations produit donc la substitution suivante :

$$\sigma = \left[\begin{array}{lll} \lambda_{19} \leftarrow \lambda_{12} \ , & \lambda_4 \leftarrow \lambda_{12} - \mu_1 - \mu_2 \ , & \lambda_{17} \leftarrow \lambda_{10} \\ \lambda_{13} \leftarrow \lambda_{10} \ , & \lambda_1 \leftarrow -\mu_2 - \mu_1 + \lambda_{10} \ , & \lambda_{16} \leftarrow \lambda_{12} \\ \lambda_{11} \leftarrow -\mu_2 \ , & \lambda_{18} \leftarrow -\mu_2 \ , & \lambda_9 \leftarrow \lambda_{12} - \mu_2 - \mu_1 \\ \lambda_{14} \leftarrow -\mu_2 \ , & \lambda_3 \leftarrow \mu_2 \ , & \lambda_5 \leftarrow -\mu_1 - \mu_2 + \lambda_{10} \end{array} \right]$$

En appliquant cette substitution sur les prototypes, nous obtenons :

$$\begin{array}{ll} \pi^p_{s1}(i,j) & = \mu_2.j + (\lambda_{12} - \mu_1 - \mu_2).n + \lambda_{10} - \mu_1 - \mu_2 \\ \pi^p_{s2}(i,j,k) & = \mu_2.j + \mu_1.k + (\lambda_{12} - \mu_1 - \mu_2).n + \lambda_{10} - \mu_1 - \mu_2 \\ \pi^p_{s3}(i) & = -\mu_2.i + \lambda_{12}.n + \lambda_{10} \\ \pi^p_{s4}(i,j) & = -\mu_2.i + \lambda_{12}.n + \lambda_{10} \\ \pi^p_{s5}(i) & = -\mu_2.i + \lambda_{12}.n + \lambda_{10} \end{array}$$

4.5.6 Valuation

Cette dernière étape consiste à valuer les coefficients de λ. Cette valuation se fait en construisant une par une les différentes dimensions de la fonction de placement. Le nombre de dimensions à construire pour chaque instruction dépend du choix de géométrie pour chaque instruction (voir section 4.5.1) ; si c'est la même pour toutes, ce nombre est égal à d_{max}.

Nous partageons la liste λ en deux partitions, avec d'une part une liste λ^x contenant les coefficients qui sont facteurs dans les prototypes d'indices de boucle, et d'autre part une liste λ^p contenant tous les autres coefficients. Pour chaque dimension, la valuation est donc divisée en deux étapes, la valuation des coefficients de λ^x, et la valuation des coefficients de λ^p.

Valuation des facteurs des indices

C'est la valuation de ces coefficients qui détermine la direction de distribution de la dimension. Deux cas se présentent :

1. si le nombre de coefficients libres de λ^x est plus petit ou égal au nombre de dimensions à calculer (d_{max}) alors, pour une dimension donnée (la $i^{ème}$), il s'agit d'annuler tous ses coefficients sauf un (valué à 1), le $i^{ème}$ dans la liste ;

2. sinon, nous proposons d'introduire des relations entre ces coefficients pour réduire le nombre de ceux qui sont libres.

   Une première idée est qu'il serait intéressant de valuer à 1 pour chaque prototype et pour chaque dimension un seul coefficient facteur d'indices de boucle. Ainsi, si deux coefficients (facteurs d'indices de boucle dans deux prototypes distincts) n'apparaissent jamais dans le même prototype, nous pouvons les valuer ensemble[12] à 1.

   S'il reste encore trop de coefficients libres, une deuxième idée est de les classer dans l'ordre décroissant du nombre d'occurences dans les prototypes. Les premiers de la liste sont alors mis en relation avec les derniers, ce qui crée de nouvelles relations.

   Nous proposons donc de construire une liste ordonnée L (de longueur d_{max}) de listes (l_i) de coefficients de λ^x. L'algorithme 14 donne une méthode pour la construire. Pour une liste l_i donnée, il suffit ensuite de considérer tous ses coefficients comme égaux ; il ne reste plus alors que d_{max} coefficients libres, nous sommes revenus au premier cas.

Algorithme 14 *Construction de L :*

1. *au départ, $L = \{l_i\}_{i \in I}$, avec $I = \emptyset$;*
2. *boucle sur la liste des instructions du programme :*
   (a) *extraire le prototype π_s^p de l'instruction courante s ;*
   (b) *construire une sous-liste λ^s de λ^x contenant les coefficients qui apparaissent dans π_s^p ;*
   (c) *pour chaque élément c de λ^s, si $\forall i \in I, c \notin l_i$, alors :*
      - *si $\exists i_0 \in I \mid \lambda^s \cap l_{i_0} = \emptyset$ alors ajouter c dans l_{i_0}.*
      - *sinon créer $l_{i_0} = \{c\}$ et l'ajouter à L ;*
3. *trie des l_i par ordre décroissant d'occurences : le nombre d'occurences de l_{i_0} est égal à la somme des nombres d'occurences dans les prototypes des coefficients contenu dans cette liste.*

12. Ceci revient à générer une relation de plus : égalité de ces deux coefficients.

4. si $I = \{1, \ldots, d_{max}, \ldots, D\}$ *alors pour chaque valeur* i *de* $\{d_{max} + 1, \ldots, D\}$ *:*

- *remplacer* $l_{i-d_{max}}$ *par* $l_{i-d_{max}} \cup l_i$
- *enlever* l_i *de* L*.*

Valuation des facteurs des paramètres de structure

En ce qui concerne les coefficients λ^p, la seule contrainte est que notre fonction de placement soit positive. Ainsi, pour chaque dimension, une fois que les valeurs des coefficients de λ^p sont déterminées, nous les valuons en appliquant une méthode décrite par Feautrier dans [Fea92a]. Cette méthode utilise un résultat de la théorie sur les inégalités linéaires, le lemme de Farkas ([Sch86]) et un outil de résolution de systèmes linéaires paramétrés en nombre entier, le logiciel PIP (pour "Parametric Integer Programming", [Fea88]).

Revenons encore à notre programme `gauss`. Il reste quatre variables à valuer $(\mu_1, \mu_2, \lambda_{10}, \lambda_{12})$. Parmi ces variables, deux seulement (μ_1, μ_2) sont facteurs d'indice de boucle ; de plus, $d_{max} = 2$, donc nous nous trouvons dans le premier cas de la section 4.5.6. Ainsi, notre traitement se déroule en deux étapes :

- Calcul de la valeur de la première dimension des prototypes. Nous valuons μ_2 à 1 et μ_1 à 0. Pour le calcul des valeurs de λ_{10} et λ_{12}, la seule contrainte est que la fonction de placement doit être positive. En appliquant le lemme de Farkas et en résolvant avec PIP, nous trouvons les valeurs suivantes :

$$\lambda_{10} = 0 \ , \lambda_{12} = 1$$

- Calcul de la valeur de la deuxième dimension des prototypes. Nous valuons μ_2 à 0 et μ_1 à 1. Il reste à calculer λ_{10} et λ_{12}. De la même manière que précédemment, nous trouvons les valeurs suivantes :

$$\lambda_{10} = 1 \ , \lambda_{12} = 1$$

Notre fonction de placement pour chaque instruction est donc finalement :

$$\begin{array}{ll} \pi_{s1}(i,j) & = (j-1 \quad 0) \\ \pi_{s2}(i,j,k) & = (j-1 \quad k) \\ \pi_{s3}(i) & = (-i+n \quad n+1) \\ \pi_{s4}(i,j) & = (-i+n \quad n+1) \\ \pi_{s5}(i) & = (-i+n \quad n+1) \end{array}$$

En ce qui concerne les distances des arcs, nous en déduisons leur valeur finale (sur chaque dimension de la fonction de placement) :

$$\begin{array}{lll} d_{e1}(i,j) = & (j-i & -i) \\ d_{e2}(i,j) = & (0, & -i) \\ d_{e3}(i,j,k) = & (-i+j & 0) \\ d_{e4}(i,j,k) = & (0, & k) \\ d_{e5}(i,j,k) = & (0, & 0) \\ d_{e6}(i,j) = & (j-i & 0) \\ d_{e7}(i,j) = & (0, & j) \\ d_{e8}(i,j) = & (0, & 0) \\ d_{e9}(i,j) = & (0, & 0) \\ d_{e10}(i) = & (0, & i) \\ d_{e11}(i) = & (0, & 0) \\ d_{e12}(i) = & (0, & 0) \end{array}$$

Analysons instruction par instruction les communications qu'il sera nécessaire d'engendrer :

- l'instruction $s1$ lit deux valeurs. La première, $a(j,i)$, est portée par l'arc $e2$. Ce n'est pas une diffusion et sa distance n'est pas nulle. Une communication générale est donc nécessaire.

  Par contre, la deuxième, $a(i,i)$, portée par $e1$ correspond à une diffusion. Cette diffusion est réalisable, la distance de cet arc n'est donc pas nulle.

- l'instruction $s2$ lit trois valeurs. La première, $a(j,k)$, portée par $e5$ ne nécessite pas de communication puisque la distance est nulle.

  Par contre, la deuxième, $a(i,k)$, et la troisième, f, portées par $e3$ et $e4$ sont des diffusions. Comme ci-dessus ces diffusions sont réalisables et donc les distances ne sont pas nulles.

– l'instruction $s3$ ne lit aucune valeur. Elle n'engendre donc aucune communication.

– l'instruction $s4$ lit trois valeurs. La première, $x(n-j+1$, portée par $e6$ est une diffusion réalisable. La distance n'est donc pas nulle.

  La deuxième, $a(n-i+1, n-j+1)$, est portée par $e7$ dont la distance n'est pas nulle et qui n'est pas une diffusion. Une commmunication générale est donc engendrée.

  Enfin, la dernières, s, portée par 8 et $e9$ ne nécessite pas de communication puisque les distances de ces deux arcs sont nulles.

– l'instruction $s5$ lit trois valeurs. La première, $a(n-i+1, n+1)$, vient de la valeur initiale de a.

  La seconde, $a(n-i+1, n-i+1)$, est portée par $e10$ dont la distance n'est pas nulle et qui n'est pas une diffusion. Une communication générale est engendrée.

  Enfin, la troisième, s, portée par $e11$ et $e12$ n'engendre aucune communication puisque ces deux arcs ont des distances nulles.

Les arcs $e1$, $e3$ et $e4$ correspondent en fait aux trois diffusions que nous avons représentées figure 3.2. La quatrième diffusion (sur $e6$) est effectuée lors de la remontée triangulaire et elle correspond dans cet algorithme à la réutilisation des valeurs déjà calculées. Nous verrons au chapitre suivant pourquoi ces quatre diffusions sont réalisables.

4.6 Comparaison avec la méthode directe

Afin de mesurer l'efficacité de cette nouvelle méthode, nous avons fait une étude comparative entre notre méthode de calcul de la fonction de placement avec optimisation des communications et la méthode directe (voir section 3.4) sur huit programmes pris dans différents domaines d'application : `fmm` (multiplication de matrices) ; `lampif` (relaxation) ; `choles`, `gauss` et `seidel` (résolution de systèmes) ; `lczos` (calcul de valeurs propres) ; `burg2` et `thom` (traitement du signal). Nous donnons le code de ces programmes en annexe C.

Cette étude se borne à comparer le nombre et le type de communications que chaque méthode rendra nécessaire lors de l'exécution du programme

parallèle. Pour cela, nous utilisons la valeur des distances des arcs du DFG calculée à partir de la fonction de placement obtenue.

Comme nous le verrons au chapitre suivant, connaissant les arcs de diffusion, la valeur de la distance d'un arc nous permet de déterminer le type de communication qu'il représente. Nous classons donc les arcs dans quatre catégories suivant la valeur de leur distance :

1. Distance nulle (notée `DN`) : pas de communication
2. Distance constante non nulle (notée `DC`) : translation
3. Distance variable sur un arc de diffusion (notée `DD`) : diffusion
4. Distance variable (notée `DG`) : communication générale

Le tableau 4.1 donne pour tous ces programmes le taux de chacun de ces types d'arc suivant la méthode utilisée pour calculer la fonction de placement.

Programmes	Nombre d'arcs	Taux des types d'arc						
		Placement avec optimisation des communications				Placement : méthode directe		
		DN	DC	DD	DG	DN	DC	DG
`fmm`	4	75%	0%	25%	0%	50%	0%	50%
`lampif`	6	78%	22%	0%	0%	72%	22%	6%
`choles`	12	67%	8%	17%	8%	58%	0%	42%
`gauss`	12	67%	0%	21%	12%	54%	0%	46%
`seidel`	10	55%	25%	5%	15%	83%	4%	13%
`lczos`	47	79%	6%	11%	4%	58%	19%	23%
`burg2`	28	32%	14%	14%	40%	43%	27%	30%
`thom`	47	67%	5%	20%	8%	67%	0%	33%

TAB. 4.1 - *Taux de chaque type d'arc pour huit programmes*

Pour le programme `burg2`, la prise en compte des diffusions va empêcher l'annulation d'un certain nombre de distances. Dans ce cas, notre méthode d'optimisation n'est pas performante puisqu'elle augmente le nombre de communications générales par rapport à la méthode directe.

De même, pour le programme `seidel`, avec l'optimisation des communications, nous faisons d'abord en sorte que les distance soient rendues constantes avant d'essayer de les annuler (voir section 4.4). Dans ce cas également, cette optimisation n'est pas performante puisqu'elle va empêcher l'annulation d'un grand nombre de distance, sans pour autant réduire sensiblement le nombre de communications générales.

Néanmoins, pour tous les autres programmes, notre optimisation donne des résultats très satisfaisants.

En effet, étudions maintenant le tableau 4.2 qui combine tous ces résultats et donne le taux moyen de chacun de ces types d'arc.

Taux des types d'arc						
Placement avec optimisation des communications				Placement : méthode directe		
DN	DC	DD	DG	DN	DC	DG
67%	8%	15%	10%	60%	12%	28%

TAB. 4.2 - *Taux moyen de chaque type d'arc*

Nous pouvons déduire de ces deux tableaux que, en règle générale, la nouvelle méthode de calcul de la fonction de placement permet de générer un taux légèrement plus élevé d'arcs ayant une distance nulle (`DN`), *i.e.* il y a moins de communications.

Plus significatif encore est le nombre d'arc ayant une distance variable, *i.e.* représentant des communications générales, qui se trouve considérablement réduit par notre nouvelle méthode. En fait, ces communications générales sont remplacées, le plus souvent, par des diffusions.

Enfin, le nombre d'arc ayant une distance constante non nulle reste en moyenne équivalente.

Afin d'avoir une idée du coût moyen de communication, nous avons pondéré ces résultats par le tableau 1.1 qui donne le coût des différents mouvements de données sur la CM-5 : un arc `DN` a un coût nul, `DC` a un coût de 2.5, `DD` de 1.5 et `DG` de 90.

Avec un placement qui optimise les communications nous obtenons un coût moyen de 943 ; avec un placement avec la méthode directe nous obtenons un coût moyen d'environ 2550.

Cette différence signifie que notre méthode de calcul de fonction de placement permet de gagner, en moyenne, plus d'un facteur 2 sur le coût des communications sur la CM-5.

A partir de cet échantillon de huit programmes, nous constatons donc que cette nouvelle méthode de calcul de la fonction de placement permet, dans la grande majorité des cas, une bonne optimisation des communications et, ainsi, une réduction du coût global des communications.

Chapitre 5

Génération de code et expérimentations

La génération de code est réalisée à partir de la méthode donnée section 3.5. Cette méthode reconstruit juste le programme parallèle sans s'occuper des communications ni de la distribution des tableaux.

Prenons par exemple le programme de multiplication de matrice de la figure 5.1). En appliquant cette phase de réindexation, nous obtenons le code donné figure 5.2 (ceci constitue le corps de notre programme parallèle).

Nous voyons donc que pour obtenir le code final, il est nécessaire de procéder à deux autres phases.

La première introduit les déclarations de distribution des tableaux et les appels aux primitives engendrant des communications bien optimisées. Il est important de noter que cette phase est indépendante de la machine cible.

La seconde produit une version du programme parallèle qui respecte la syntaxe du langage parallèle correspondant, ceci afin de pouvoir compiler et exécuter ce code sur la machine cible. Cette phase est donc entièrement dépendante de la machine cible, ce qui nous a obligés à rechercher les particularités de chaque machine cible (la CM-5 et le Cray-T3D) pour la production du programme parallèle.

```
      program fmm
      integer i, j, k, n
      real a(n,n), b(n,n), c(n,n), a'(n,n), b'(n,n)
      do i=1,n
         do j=1,n
s1          a(i,j) = a'(i,j)
s2          b(i,j) = b'(i,j)
         end do
      end do
      do i=1,n
         do j=1,n
s3          c(i,j) = 0.
            do k=1,n
s4             c(i,j) = c(i,j) + a(i,k) * b(k,j)
            end do
         end do
      end do
      end
```

FIG. 5.1 - *Programme fmm (multiplication de matrices)*

5.1 Déclaration des tableaux

Pour chaque tableau de notre programme nous devons établir la longueur de ses dimensions, leur distribution et leur alignement. Nous distinguons les dimensions **temporelles** et les dimensions **spatiales**. Les premières correspondent aux dimensions du tableau qui sont accédées en fonction d'une variable temporelle lors de son écriture, les secondes sont celles accédées par une variable de placement.

5.1.1 Longueur des dimensions

La longueur de chaque dimension est donnée par les valeurs maximales que peuvent prendre les variables temporelles et de placement qui les accèdent lors de l'écriture.

Pour les dimensions temporelle, il faut calculer le nombre de valeurs prises par l'ordonnancement mineur correspondant. Ce nombre dépend des plus

```
 s4t0 = 2
 s4t1 = 1
 s4q0 = 0
 s4flag0 = 0
 DO S0t0 = 0, MAX(1, N)
    IF (S0t0.EQ.0) THEN
       DOALL s1p0 = 1, N
          DOALL s1p1 = 1, N
             ins_1(s1p0,s1p1) = A'(s1p0,s1p1)
          ENDDOALL
       ENDDOALL
    ENDIF
    IF (S0t0.EQ.0) THEN
       DOALL s2p0 = 1, N
          DOALL s2p1 = 1, N
             ins_2(s2p0,s2p1) = B'(s2p0,s2p1)
          ENDDOALL
       ENDDOALL
    ENDIF
    IF (S0t0.EQ.0) THEN
       DOALL s3p0 = 1, N
          DOALL s3p1 = 1, N
             ins_3(s3p0,s3p1) = 0.
          ENDDOALL
       ENDDOALL
    ENDIF
    IF (S0t0.EQ.s4t1) THEN
       DOALL s4p0 = 1, N
          DOALL s4p1 = 1, N
             ins_4(HOD(s4q0, 2),s4p0,s4p1) = ins_3(s4p0,s4p1)+
&            ins_1(s4p0,1+s4q0)*ins_2(1+s4q0,s4p1)
          ENDDOALL
       ENDDOALL
       s4t1 = s4t1+1
       IF (s4t1.GT.1) THEN
          s4t1 = -1
       ENDIF
       s4flag0 = 1
    ENDIF
    IF (S0t0.EQ.s4t0) THEN
       DOALL s4p0 = 1, N
          DOALL s4p1 = 1, N
             ins_4(HOD(s4q0, 2),s4p0,s4p1) = ins_4(HOD(-1+s4q0,
&            2),s4p0,s4p1)+ins_1(s4p0,1+s4q0)*ins_2(1+s4q0,s4p1)
          ENDDOALL
       ENDDOALL
       s4t0 = s4t0+1
       IF (s4t0.GT.N) THEN
          s4t0 = -1
       ENDIF
       s4flag0 = 1
    ENDIF
    IF (s4flag0.EQ.1) THEN
       s4q0 = s4q0+1
    ENDIF
    s4flag0 = 0
 ENDDO
```

FIG. 5.2 - *Programme* `fmm` *parallélisé*

grandes valeurs (M_t) prises par les variables de temps locaux, des plus petites valeurs (m_t), et également de la période (ω) : $\frac{M_t - m_t}{\omega} + 1$. Or, la borne inférieure est toujours mise à zéro, car toutes les variables de temps mineur sont initialisées à zéro. Donc, la borne supérieure est égale à : $\frac{M_t - m_t}{\omega}$.

Pour les dimensions spatiales, il faut calculer pour chacune d'elles la plus petite valeur et la plus grande valeur prises par la variable de placement correspondante dans les différentes boucles parallèles générées. Ces deux valeurs donnent respectivement la borne inférieures et la borne supérieures de la dimension.

> Revenons à l'instruction $s2$ de notre programme `gauss`. Il faut calculer les longueurs des trois dimensions du nouveau tableau ins_2. La première dimension est temporelle, la plus grande valeur prise par le temps local est $2n-3$, la plus petite valeur est 1 et la période est 2. Les bornes de cette dimension sont donc $0 : n-2$. Les deux autres dimensions sont spatiales, elles ont respectivement pour bornes $(1 : n-1)$ et $(2 : n)$.
>
> Prenons maintenant l'instruction $s1$. Il faut calculer les longueurs des trois dimensions du nouveau tableau ins_1. La première dimension est temporelle, la plus grande valeur prise par le temps local est $2n-4$, la plus petite valeur est 0 et la période est 2. La longueur de cette dimension devrait donc être égale à $n-1$. Or, nous avons vu section 3.5.4 qu'il y a un délai d'accès de 0 sur cette dimension, sa longueur est donc réduite à 1 (*i.e.* ses bornes sont $0 : 0$). L'autre dimension est spatiale, elle a pour bornes $(1 : n-1)$.
>
> En utilisant la notation *triplet* de Fortran 90 (voir section 1.2.1, page 30), les déclarations des dimensions des tableaux ins_1 et ins_2 sont les suivantes :
>
> ```
> REAL ins1(0:0, 1:n-1)
> REAL ins2(0:n-2, 1:n-1, 2:n)
> ```

Nous verrons plus bas que la longueur de certaines dimensions des tableaux que nous générons doit être légèrement modifiée afin de satisfaire à deux types de contraintes : les contraintes d'alignement des données en mémoire (voir section 5.1.3) et les contraintes imposées par le compilateur de la machine cible (voir section 5.3).

5.1.2 Partitionnement des dimensions

La distribution d'un tableau sur une grille de processeurs virtuels consiste à distribuer chacune de ses dimensions. La distribution des dimensions est effectuée en fonction de la nature de la dimension. L'important est en fait de déterminer les dimensions qui doivent être effectivement distribuées de celles qui doivent être écrasées, *i.e.* placées sur le même processeur.

Les dimensions temporelles sont accédées séquentiellement, elles ne doivent pas être distribuées : elles sont donc écrasées (voir section 1.2.1).

Par contre, les dimensions spatiales correspondent exactement aux dimensions accédées en parallèle : elles sont donc distribuées sur l'ensemble des processeurs.

Par exemple, pour notre tableau ins_2 du programme gauss, nous avons donc la distribution suivante (en CM Fortran) :

```
LAYOUT ins2(:SERIAL, :NEWS, :NEWS)
```

5.1.3 Alignement des dimensions

L'alignement des tableaux est effectué par rapport aux dimensions distribuées et en fonction des valeurs des fonctions de placement.

Cet alignement est implicite lorsque deux tableaux ont le même nombre de dimensions distribuées et qu'elles sont conformes (*i.e.* leurs dimensions distribuées ont la même longueur).

Ainsi, pour réaliser cet alignement implicite, deux instructions ayant la même valeur sur une de leur dimension de la fonction de placement (à une constante près) doivent déclarées la dimension correspondante de leur tableau avec la même longueur.

Si les longueurs calculées par la phase de réindexation ne sont pas identiques il faut ajuster par excès la plus petite.

Par contre, l'alignement doit être explicite si les tableaux n'ont pas le même nombre de dimensions distribuées. Dans ce cas nous alignons les dimensions

qui correspondent à la même direction de distribution, *i.e.* pour lesquelles la différence entre les composantes correspondantes de la fonction de placement est constante (*i.e.* ne dépend pas des indices de boucles).

Là encore, si les dimensions à aligner ne sont pas de même longueur il faut ajuster la plus petite par excès.

> Reprenons notre exemple sur le programme `gauss`. Nous devons explicitement aligner la seconde dimension de ins_1 avec la seconde dimension de ins_2. Cet alignement est fait sur le premier élément de la troisième dimension de ins_2, ce qui donne la déclaration d'alignement suivante (en CM Fortran):
>
> ```
> ALIGN ins1(I,J) WITH ins2(I,J,1)
> ```
>
> De plus, pour réaliser l'alignement implicite de ins_1 et ins_2 il faut ajuster (*i.e.* lui ajouter 1) la longueur de la deuxième dimension de ins_1 pour qu'elle devienne égale à celle de ins_2.

Comme nous l'avons vu section 1.2, certains langages de programmation des machines massivement parallèles ne permettent pas de faire de l'alignement explicite de dimensions (par exemple CRAFT Fortran). Dans ce cas, nous ne pouvons rien faire et seul l'alignement implicite est réalisé.

5.2 Génération des communications optimisées

Lors de la génération de code en Fortran parallèle nous souhaitons utiliser des appels aux fonctions intrinsèques définissant des communications optimisées, ou tout au moins faire en sorte que ces communications soient réalisables. Cette section donne les conditions pour lesquelles cela est possible.

5.2.1 Diffusions

Les diffusions correspondent à la fonction intrinsèque `SPREAD` (voir section 1.2.1), qui diffuse les données parallèlement à un axe. Nous devons in-

troduire un appel à cette fonction partout où cela est possible afin que la communication soit rendue explicite.

La phase de calcul de la fonction de placement a déterminé toutes les diffusions potentielles du programme. Parmis elles, il faut déterminer celles qui sont effectivement réalisables. Ensuite, il reste à générer l'appel à la fonction `SPREAD`.

Dimension de diffusion

La première étape est de savoir quelles sont les diffusions parmi celles détectées à l'étape précédente (voir section 4.2.2) qui sont faisables. En effet, il est possible que certaines d'entre elles soient inutiles dans le cas où la distance de l'arc a été annulée ; ou d'autres encore peuvent être non parallèles aux axes, auxquel cas il n'est pas possible de les programmer.

Supposons que dans une phase précédente nous avons détecté une diffusion sur un arc e du DFG. La valeur de la fonction de placement étant maintenant connue, il est possible d'un déduire la distance de cet arc. Si elle est nulle alors le code généré ne produit pas de communication due à cet arc. Par contre, si elle n'est pas nulle alors il y a une communication due à cet arc. Pour que celle-ci soit exprimable sous la forme d'une diffusion, il faut s'assurer que les directions de distribution de l'instruction destination (*i.e* les différentes composantes de sa fonction de placement) de cet arc font partie de l'espace de diffusion (voir section 4.1.1 pour la définition d'un espace de diffusion). Ces directions sont alors appelées les **dimensions de diffusion**, puisqu'elles correspondent chacune à une dimension du tableau à diffuser.

> Revenons à notre programme `gauss`. Quatre arcs ont été détectés comme diffusion (voir section 4.2.2), $e1, e3, e4$ et $e6$, sur les directions suivantes :
>
> $$\begin{aligned} \beta_{e1} &= \{(0,1,0)\} \\ \beta_{e3} &= \{(0,1,0)\} \\ \beta_{e4} &= \{(0,0,1)\} \\ \beta_{e6} &= \{(1,0,0)\} \end{aligned}$$
>
> La destination de l'arc $e1$ est l'instruction $s1$; l'espace de distribution de cette instruction est de dimension 1 et a pour direction le vecteur

$(0, 1, 0)$. Cette direction est donc une dimension de diffusion sur laquelle sera diffusée la donnée produite par l'instruction source de l'arc $e1$.

De même, pour les trois autres arcs $e3, e4$ et $e6$, à chaque direction de diffusion correspond une dimension de l'espace de distribution de l'instruction source. Pour résumer :

- $e1$ correspond à une diffusion sur la dimension de distribution de l'instruction $s1$ (il n'y en a qu'une seule) ;
- $e3$ correspond à une diffusion sur la première dimension de distribution de l'instruction $s2$;
- $e4$ correspond à une diffusion sur la seconde dimension de distribution de l'instruction $s2$;
- $e6$ correspond à une diffusion sur la dimension de distribution de l'instruction $s4$ (il n'y en a qu'une seule) ;

Génération d'une diffusion

La première étape nous a permis de déterminer pour chaque instruction l'ensemble des variables qu'elle doit diffuser et sur quelles dimensions. La deuxième étape est la génération des diffusions à partir de ces dimensions de diffusion.

La fonction intrinsèque `SPREAD` réalise une diffusion d'un tableau source (ou une section d'un tableau source) parallèlement à l'une des dimensions du tableau destination (qui a par conséquent une dimension de plus), nous devons donc engendrer autant d'appels qu'il y a de dimensions de diffusion.

Néanmoins, ces appels ne peuvent pas être effectués à l'intérieur des boucles parallèles car cette fonction travaille sur des sections de tableau (voir section 1.2, page 30). Nous proposons de résoudre ce problème de deux manières différentes :

1. la première méthode est d'introduire pour chaque diffusion un tableau temporaire, résultat de la ou des diffusions de notre tableau. Ce tableau temporaire doit être de même taille que le tableau destination de notre diffusion et distribué de la même manière.

Cette solution est pénalisante du fait de l'utilisation d'une variable supplémentaire qui peut requérir beaucoup de place mémoire.

2. la seconde méthode est de transformer le nid de boucles parallèles en une instruction Fortran 90 qui utilise la notation triplet et les opérations sur les sections de tableaux. Dans ce cas, aucune variable temporaire n'est nécessaire car l'appel à la fonction `SPREAD` est inséré directement dans l'instruction.

   Pour pouvoir appliquer cette deuxième méthode, il est donc nécessaire que le langage cible accepte ce type d'instruction.

Dans les deux cas, il reste à déterminer les arguments à passer à notre fonction `SPREAD` afin de réaliser notre diffusion convenablement.

Le premier argument a donné à la fonction `SPREAD` est la section du tableau source à diffuser. Celle-ci correspond à la section du tableau accédée par le nid de boucles parallèles qui englobe l'instruction. Elle est donc obtenue est transformant la référence en lecture de ce tableau par la section de tableau Fortran 90 qui lui est équivalente et que l'on obtient facilement à l'aide de l'espace d'itération de ce nid de boucles parallèles.

Etant donné que la fonction `SPREAD` crée une dimension de plus, la dimension de la section du tableau source de la diffusion est nécessairement strictement inférieure à la profondeur de ce nid de boucles parallèles. Ceci implique que l'un au moins des indices de ce nid de boucles parallèles ne soit pas utilisé dans la fonction d'accès à ce tableau source.

Dans notre cas précis, seules des sections des dimensions distribuées sont diffusées. Ainsi, le résultat de cette fonction est un tableau qui doit être conforme aux dimensions distribuées du tableau accédé en écriture par l'instruction dans laquelle est réalisée la diffusion.

Pour cela, soit le tableau à diffuser a déjà autant de dimensions distribuées que celui accédé en écriture, et alors l'une d'elles est accédée avec une variable d'ordonnancement (elle ne varie pas dans l'espace d'itération des boucles parallèles), soit il en a une de moins et alors la diffusion en crée une supplémentaire qui correspond à la dimension de placement non représentée dans les fonctions d'accès au tableau à diffuser.

Le second argument demandé par la fonction `SPREAD` (voir section 1.2.1) est la dimension selon laquelle la diffusion est réalisée.

Nous avons vu dans un chapitre précédent que ces directions de diffusions sont exprimées comme une combinaison linéaire des anciens indices de boucles. En appliquant le même changement de variable que lors de la réindexation, nous obtenons une combinaison linéaire ne contenant plus que l'un des nouveau indices de boucles, qui en outre doit être une variable de placement. En fait, il suffit de repérer dans l'expression des fonctions d'accès au tableau source laquelle des variables de placement n'est pas utilisée ; la diffusion doit alors être réalisée sur la dimension correspondante du tableau destination.

Le troisième argument demandé par la fonction `SPREAD` est le nombre de copies à diffuser. Ce nombre doit être égal à la longueur de la dimension distribuée du tableau destination parallèlement à laquelle la diffusion est réalisée.

> Pour l'instruction $s2$ de notre programme `gauss`, il y a deux arcs qui définissent une dimension de diffusion : $e3$ et $e4$.
>
> Pour $e4$, il faut diffuser la valeur calculée à l'instruction $s1$ parallèlement à la seconde dimension de distribution de l'instruction $s2$. Le tableau à diffuser (`ins1`) a une seule dimension distribuée, elle est diffusée autant de fois que la longueur de la troisième dimension[1] de `ins2`.
>
> Pour $e3$, il faut diffuser la valeur calculée à l'instruction $s2$ parallèlement à la première dimension de distribution de $s2$. Le tableau à diffuser (`ins2`) a déjà deux dimensions distribuées, l'une d'elle n'est pas diffusée.
>
> L'utilisation de la notation Fortran 90 nous permet ainsi d'éviter l'utiliser de deux tableaux temporaires.
>
> Le code parallèle pour l'instruction $s2$ donnée section 3.5.3 devient alors le code suivant (en CM Fortran) :

1. La première dimension étant écrasée, elle correspond donc à la deuxième dimension distribuée.

```
      REAL ins1(0:0, 1:n-1)
      REAL ins2(0:n-2, 1:n-1, 2:n)
      ...
CMF$  LAYOUT ins2(:SERIAL, :NEWS, :NEWS)
      ...
CMF$  ALIGN ins1(I,J) WITH ins2(I,J,1)
      ...
      S2t0 = 3
      S2t1 = 1
      S2flag0 = 0
      ...
       IF(T0 .eq. S2t0) THEN
        ins2(S2q0,S2q0+1:n-1,S2q0+2:n) =
     &   ins2(S2q0-1,S2q0+1:n-1,S2q0+2:n) -
     &   SPREAD(ins1(S2q0,S2q0+1:n-1),DIM=2,NCOPIES=n-1-S2q0)*
     &   SPREAD(ins2(S2q0-1,S2q0,S2q0+2:n),DIM=1,NCOPIES=n-1-S2q0)
       S2t0 = S2t0 + 2
       S2flag0 = 1
       IF(S2t0 .gt. 2n-3) S2t0 = -1
       ENDIF
       IF(T0 .eq. S2t1) THEN
        ins2(S2q0,S2q0+1:n-1,S2q0+2:n) =
     &   a(S2q0+2:n,S2q0+2:n) -
     &   SPREAD(ins1(S2q0,S2q0+1:n-1),DIM=2,NCOPIES=n-1-S2q0)*
     &   SPREAD(a(S2q0+1,S2q0+2:n),DIM=1,NCOPIES=n-1-S2q0)
       S2t1 = S2t1 + 2
       S2flag0 = 1
       IF(S2t1 .gt. 1) S2t1 = -1
       ENDIF
       IF(S2flag0 .eq. 1) S2q0 = S2q0 + 1
       S2flag0 = 0
      ...
```

5.2.2 Translations

Les translations correspondent aux deux fonctions intrinsèques `CSHIFT` et `EOSHIFT` (voir section 1.2.1), qui réalisent des décalages dimensionnels.

De même que pour les diffusions, la première étape consiste à déterminer toutes les instructions pour lesquelles nous pouvons générer une translation. Ces instructions sont données par la destination des arcs ayant une distance constante (voir section 4.4) sur chaque dimension de la fonction de placement. Ensuite, il reste à générer les appels à la fonction `EOSHIFT`[2].

2. Nous pouvons utiliser indifféremment l'une ou l'autre de ces fonctions car les éléments qui correspondent aux "trous" (voir section 1.2.1) ne sont pas utilisés.

Dimension de translation

La première étape est de savoir quelles sont les translations qui sont faisables.

Supposons que la distance d'un arc e du DFG soit constante sur toutes les dimensions de la fonction de placement. Toutes les dimensions sur lesquelles elle est nulle ne produisent aucune communication. Par contre, pour chaque dimension sur laquelle elle n'est pas nulle la communication à engendrer est une translation parallèlement à cette dimension. Cette dernière est alors appelée **dimension de translation**.

Reprenons notre exemple sur le programme `gauss` (voir figure 3.1). La fonction de placement est de dimension 2. Voici les distances des arcs sur chacune de ces deux dimensions :

$$\begin{cases} d_{e1}(i,j) = & j-i \\ d_{e2}(i,j) = & 0 \\ d_{e3}(i,j,k) = & -i+j \\ d_{e4}(i,j,k) = & 0 \\ d_{e5}(i,j,k) = & 0 \\ d_{e6}(i,j) = & j-i \\ d_{e7}(i,j) = & 0 \\ d_{e8}(i,j) = & 0 \\ d_{e9}(i,j) = & 0 \\ d_{e11}(i) = & 0 \\ d_{e10}(i) = & 0 \\ d_{e12}(i) = & 0 \end{cases} \qquad \begin{cases} d_{e1}(i,j) = & -i \\ d_{e2}(i,j) = & -i \\ d_{e3}(i,j,k) = & 0 \\ d_{e4}(i,j,k) = & k \\ d_{e5}(i,j,k) = & 0 \\ d_{e6}(i,j) = & 0 \\ d_{e7}(i,j) = & j \\ d_{e8}(i,j) = & 0 \\ d_{e9}(i,j) = & 0 \\ d_{e10}(i) = & i \\ d_{e11}(i) = & 0 \\ d_{e12}(i) = & 0 \end{cases}$$

Nous remarquons que les distances sont soit nulles soit non constantes. Il n'y a donc aucune dimension de translation.

Prenons maintenant le programme `voisin` (voir figure 5.3) qui remplace chaque élément de la matrice A par la somme de ses quatre voisins.

Ce programme a un DFG de quatre arcs (voir tableaux 5.1 et 5.2), tous de l'instruction $s1$ vers l'instruction $s2$ (un pour chaque référence à droite de $s2$).

```
      program voisin
      integer i, j, n
      parameter(n = 1024)
      real a(n,n), b(n,n)
      do i = 1,n
        do j = 1,n
s1        b(i,j) = a(i,j)
        enddo
      enddo
      do i = 2,n-1
        do j = 2,n-1
s2        a(i,j) = b(i,j-1) + b(i,j+1) + b(i-1,j) + b(i+1,j)
        enddo
      enddo
      end
```

FIG. 5.3 - *Programme* `voisin`

Instructions	Indices de boucle	Domaines
$s1$	i, j	$\begin{pmatrix} n-i \\ i-1 \\ n-j \\ j-1 \end{pmatrix} \geq 0$
$s2$	i, j	$\begin{pmatrix} n-i-1 \\ i-2 \\ n-j \\ j-2 \end{pmatrix} \geq 0$

TAB. 5.1 - *Nœuds du DFG du programme* `voisin`

Arc	Source	Puits	Référence	Prédicat
e1	$\langle s1, i, j-1 \rangle$	$\langle s2, i, j \rangle$	b(i,j-1)	
e2	$\langle s1, i, j+1 \rangle$	$\langle s2, i, j \rangle$	b(i,j+1)	
e3	$\langle s1, i-1, j \rangle$	$\langle s2, i, j \rangle$	b(i-1,j)	
e4	$\langle s1, i+1, j \rangle$	$\langle s2, i, j \rangle$	b(i+1,j)	

TAB. 5.2 - *Arcs du DFG du programme* `voisin`

Voici la base de temps et la fonction de placement de ce programme :

$$\begin{aligned} \theta_{s1}(i,j) &= 0 \\ \theta_{s2}(i,j) &= 1 \end{aligned}$$

$$\begin{aligned} \pi_{s1}(i,j) &= (j+1 \quad i) \\ \pi_{s2}(i,j) &= (j \quad\quad i) \end{aligned}$$

Voici les distances des arcs sur chacune des dimensions de la fonction de placement :

$$\left\{ \begin{array}{ll} d_{e1}(i,j) = & 0 \\ d_{e2}(i,j) = & -2 \\ d_{e3}(i,j) = & -1 \\ d_{e4}(i,j) = & -1 \end{array} \right. \quad \left\{ \begin{array}{ll} d_{e1}(i,j) = & 0 \\ d_{e2}(i,j) = & 0 \\ d_{e3}(i,j) = & 1 \\ d_{e4}(i,j) = & -1 \end{array} \right.$$

Nous remarquons cette fois que trois distances sont nulles et cinq sont constantes. Il y a donc cinq dimensions de translation.

Génération d'une translation

Comme pour la génération d'une diffusion (voir section précédente), la première étape nous a permis de déterminer pour chaque instruction l'ensemble des variables qui doivent être décalées et sur quelles dimensions. La deuxième étape est la génération de la translation, à partir de ces dimensions de translation.

La fonction intrinsèque `EOSHIFT` réalise une translation d'un tableau source (ou une section d'un tableau source) parallèlement à l'une des dimensions du tableau destination (qui est par conséquent de même dimension), nous devons donc engendrer autant d'appels qu'il y a de dimensions de translation.

De plus, comme pour la fonction `SPREAD`, ces appels ne peuvent pas être effectués à l'intérieur des boucles parallèles car cette fonction travaille sur des sections de tableau. Les deux méthodes proposées pour la génération d'une diffusion sont donc applicables, avec une préférence pour celle qui n'utilise pas de tableau temporaire afin de minimiser la mémoire utilisée.

Dans les deux cas, il reste à déterminer les arguments à passer à notre fonction `EOSHIFT` afin de réaliser notre translation convenablement.

Le premier argument a donné à la fonction `EOSHIFT` est la section du tableau source sur laquelle nous devons réaliser la translation. Celle-ci correspond à la section du tableau accédée par le nid de boucles parallèles qui englobe l'instruction. Elle est donc obtenue est transformant la référence en lecture de ce tableau par la section de tableau Fortran 90 qui lui est équivalente et que l'on obtient facilement à l'aide de l'espace d'itération de ce nid de boucles parallèles.

Etant donné que la fonction `EOSHIFT` conserve les dimensions, la dimension de la section du tableau source est nécessairement égale à la profondeur de ce nid de boucles parallèles. Ceci implique que tous les indices de ce nid de boucles parallèles soient utilisés dans la fonction d'accès à ce tableau source.

Dans notre cas précis, seules des sections des dimensions distribuées sont décalées. Ainsi, le résultat de cette fonction est un tableau qui doit être conforme aux dimensions distribuées du tableau accédé en écriture par l'instruction dans laquelle est réalisée la diffusion.

Le second argument demandé par la fonction `EOSHIFT` (voir section 1.2.1) est le pas du décalage. Celui-ci est égal à la distance trouvée à l'étape précédente.

Le troisième argument est optionnel, il donne les valeurs à mettre à la frontière. Nous ne le donnons pas, pas défaut les valeurs frontières sont mises à zéro.

Enfin, le quatrième argument demandé par la fonction `EOSHIFT` (voir section 1.2.1) est également optionnel (par défaut sa valeur est 1), il donne la dimension selon laquelle la translation doit être réalisée. Nous ne pouvons pas être sûr que notre translation soit toujours sur la première dimension ; la valeur de cet argument correspond en fait au numéro de la dimension de translation dans les dimensions distribuées du tableau source.

> Reprenons notre exemple précédent sur le programme `voisin` (voir ci-dessus). Pour l'instruction $s2$, il y a trois arcs sur lesquels nous avons des dimensions de translations : $e2$, $e3$ et $e4$.
>
> Pour $e2$, il faut faire un décalage sur la première dimension d'un pas de 2. Pour $e3$, il faut faire un décalage sur la première dimension d'un pas de 1 et un décalage sur la seconde dimension d'un pas de -1. Pour

$e4$, il faut faire un décalage sur la première dimension d'un pas de 1 et un décalage sur la seconde dimension d'un pas de 1.

Comme pour les diffusions, l'utilisation de la notation Fortran 90 nous permet ainsi d'éviter d'utiliser de deux tableaux temporaires. Le code parallèle généré pour ce programme est alors le suivant (en CM Fortran) :

```
      PROGRAM VOISIN
      INTEGER N,S1p0,S1p1,S2p0,S2p1,S0t0
      PARAMETER(N = 1024)
      REAL A(N,N),INS_1(2:N+1,1:N),INS_2(2:N+1,2:N+1)
CMF$  LAYOUT INS_1(:NEWS, :NEWS)
CMF$  LAYOUT INS_2(:NEWS, :NEWS)
      DO S0t0 = 0, 1
        IF (S0t0.EQ.0) THEN
          INS_1(2:N+1,1:N) = A(1:N,1:N)
        ENDIF
        IF (S0t0.EQ.1) THEN
          INS_2(2:N-1,2:N-1) = INS_1(2:N-1,2:N-1)+
     &      EOSHIFT(INS_1(4:N+1,2:N-1),DIM=1,SHIFT=2)+
     &      EOSHIFT(EOSHIFT(INS_1(3:N,1:N-2),DIM=1,SHIFT=1),DIM=2,SHIFT=-1)+
     &      EOSHIFT(EOSHIFT(INS_1(3:N,3:N),DIM=1,SHIFT=1),DIM=2,SHIFT=1)
        ENDIF
       ENDDO
       END
```

5.3 Génération de code sur machines cibles

Dans cette section nous détaillons les usages et problèmes que nous avons rencontré pour la génération de code sur les deux machines cibles auxquelles nous avons eu directement accès : le CM Fortran pour la Connection Machine CM-5 (voir section 1.1.1, page 19) et le CRAFT Fortran pour le Cray-T3D (voir section 1.1.5, page 24).

Comme nous l'avons dit auparavant (section 1.3), notre méthode est indépendante de la machine cible sauf en ce qui concerne la phase ultime de génération de code en langage source parallèle. Il serait donc assez simple de produire du code pour une autre machine massivement parallèle.

5.3.1 CM Fortran

Pour les déclarations des tailles des tableaux, le compilateur CM Fortran pour la CM-5 accepte n'importe quelle longueur pour chaque dimension.

Néanmoins, le guide utilisateur ([TMC94]) indique que ce paramètre peut influencer les performances.

En effet, l'allocation d'un tableau distribué sur l'ensemble des processeurs est réalisée en le divisant en blocs contigus appelés **sous-grilles** ("subgrids" en anglais) allouées chacune sur un PE. Chaque sous-grille a la même dimension que le tableau initial et elles sont toutes de même taille.

Cela n'est pas toujours possible de diviser un tableau donné en sous-grilles de mêmes tailles. Dans ce cas, des éléments dits **de remplissage** ("padding" en anglais) sont ajoutés à certaines dimensions du tableau afin que la nouvelle taille soit divisible équitablement. Pour qu'un tableau de génère pas d'éléments de remplissage il faut donc que son nombre d'éléments soit un multiple du nombre de PE.

Notre génération code ne se préoccupe pas de ce problème, néanmoins nous avons fait quelques expérimentations pour évaluer son influence (voir chapitre suivant).

Comme nous l'avons vu auparavant (voir section 1.2.1), CMF propose deux type de partitionnement des dimensions des tableaux : `:NEWS` et `:SERIAL`. Le premier spécifie que la dimension doit être distribuée sur l'ensemble des processeurs alors que le second spécifie qu'elle doit être écrasée.

Pour notre génération de code en CMF, nous mettons `:SERIAL` pour les dimensions temporelles et `:NEWS` pour les dimensions spatiales.

Pour l'alignement explicite, CMF propose la directive `ALIGN`. Néanmoins, le guide utilisateur ([TMC94]) met en garde contre une utilisation abusive de cette directive qui peut, par exemple, requérir une place mémoire très importante lorsque un petit tableau est aligné sur gros.

Nous avons donc choisi de ne pas l'utiliser pour notre génération de code en CM Fortran (nous avons quelques expérimentations pour évaluer son influence sur les performances, voir chapitre suivant).

Pour la génération des nid de boucles paralléles, nous avons généré des boucles `FORALL`, chacune décrivant l'espace d'itération du nid auquel elle correspond.

Comme nous l'avons dit plus haut (section 5.2), ce type de génération ne permet pas l'emploi direct des fonctions intrinsèques de diffusion et de translation. Au chapitre suivant, nous donnons les résultats de notre étude des performances de ce type de code par rapport à celui contenant les appels explicites à ces fonctions (lorsque cela est nécessaire).

Nous donnons la version parallèlisée du programme `gauss` en CM Fortran à l'annexe B.2.

5.3.2 CRAFT Fortran

A la différence du compilateur CMF, le compilateur CRAFT (tout au moins sa version actuelle) n'accepte pas les déclarations de tableau dans lesquelles la longueur des dimensions qui sont distribuées sur les processeurs n'est pas une puissance de deux.

Pour la génération de code en CRAFT, il est donc nécessaire d'ajuster la longueur de toutes les dimensions distribuées des tableaux à la puissance de deux supérieure.

Nous avons vu (voir section 1.2.2) que CRAFT en propose deux types de partitionnement par dimension, similaires à ceux de CMF. Seule la syntaxe difffère : ":" pour les dimensions écrasées, `:BLOCK` pour les dimensions distribuées.

CRAFT donne la possibilité aux utilisateurs de définir la taille des blocs de partitionnement. Pour simplifier le problème, nous ne nous en servons pas mais il pourrait être intéressant de faire une études sur son influence sur les performances.

Donc, pour notre génération de code en CMF, nous mettons ":" pour les dimensions temporelles et `:BLOCK` pour les dimensions spatiales.

En ce qui concerne l'alignement des tableaux, seul l'alignement implicite est réalisée du fait de l'absence de directive pour la spécification de l'alignement explicite en CRAFT.

Les boucles parallèles sont définies par la directive `DOSHARED` pour laquelle doivent être donnés l'ensemble des indices des boucles à paralléliser. De plus, comme nous l'avons auparavant (voir section 1.2.2), la version actuelle du compilateur n'accepte que la clause `ON` qui aligne la distribution des itérations sur les processeurs contenant la référence spécifiée.

Cette clause demande la donnée de cette référence (un tableau "partagé") dans la laquelle chaque fonction d'indices est de la forme $aI + b$, où l'indice de boucle I est utilisé au plus une fois dans cette référence, et, a et b sont des valeurs entières (des expressions, des constantes ou des variables).

Pour appliquer la règle "owner compute", nous devons donner comme argument à cette clause `ON` la référence à gauche de l'instruction englobée par le nid de boucles correspondant. Or, après l'application de la phase de réindexation, nous avons vu qu'il est possible que la fonction `MOD` apparaissent dans les fonctions d'accès des dimensions temporelles des tableaux (voir section 3.5.4). Ces dimensions étant écrasées, le numéro du processeur qui contient la référence ne dépend pas de la valeur de la fonction d'indice sur celle-ci.

Donc, en résumé, l'argument que nous donnons à la clause `ON` est la référence à gauche de l'instruction contenue dans notre nid de boucles, dans laquelle nous avons supprimé les appels à la fonction `MOD`.

Enfin, dans un nid de boucles donné, toutes les boucles parallèles (`i.e.` déclarées `DOSHARED`) doivent être parfaitement imbriquées et leurs indices ne doivent pas être utilisés dans les expressions des bornes des boucles plus internes.

Avec cette dernière restriction, un parcours triangulaire en parallèle nécessite donc l'utilisation de tests explicites.

Nous donnons la version parallèlisée du programme `gauss` en CRAFT Fortran à l'annexe B.3.

5.4 Expérimentations

A partir des travaux que nous décrivons aux chapitres précédents nous avons pu entammer une série d'expérimentations sur CM-5 et Cray-T3D. Le but final de ces expérimentations est de valider en pratique l'efficacité de notre nouvelle méthode de calcul de la fonction de placement.

Avant d'en venir à cela, nous avons néanmoins du tester les différentes possibilités, évoquées au chapitre précédent, pour la génération de code de chaque machine.

5.4.1 CM-5

Toutes nos expérimentations sur CM-5 ont été effectuées sur la machine située au Site Expérimental en Hyperparallélisme (SEH) de l'Etablissement Technique Central de l'Armement (ETCA) à Arcueil. Cette machine posséde 32 nœuds de calcul, chacun contenant quatre unités vectorielles. Sauf lorsque cela est spécialement indiqué, ces unités sont toujours utilisées.

Dans une première série d'expérimentations, nous avons cherché à déterminer l'influence de la taille des dimensions, de l'utilisation de la directive `ALIGN` et de l'utilisation de la primitive `SPREAD` sur les performances.

Toutes ces expérimentations ont été fondées sur les performances du programme `gauss` (la version séquentielle est donnée figure 3.1, page 71) parallélisé, avec $n = 256$.

A partir de ces trois optimisations possibles, nous avons comparé les temps d'exécution de dix versions parallèles possibles du programme `gauss` : le programme parallélisé "brut" (voir annexe B.2) avec uniquement les directives de distribution `LAYOUT` et sans appel à la fonction `SPREAD` (programme noté `B`) ; le programme brut avec ajustement de la taille des dimensions à une puissance de deux (noté `B2`) ; utilisation de la directive `ALIGN` quand la taille des dimensions le permet[3] (noté `Ba`) ; appel explicite à la fonction `SPREAD` (noté `Bs`) ; en mettant plusieurs optimisations ensembles nous obtenons les

3. Concrêtement, un seul appel à cette directive est effectué, c'est l'alignement du tableau `INS_5` sur le tableau `INS_3`.

programmes notés `B2s`, `Ba2`, `Bas` et `Ba2s` ; enfin, lorsque les dimensions sont ajustés sur les puissances de deux, il est possible d'utiliser la directive `ALIGN` sur deux tableaux[4], nous obtenons alors les deux derniers programmes notés `Baa2` et `Baa2s`.

Versions parallèles du programme `gauss`	Temps Ecoulé (secondes)	Temps Occupé (secondes)
B	3.47	2.33
Ba2	3.42	2.40
Baa2s	3.61	2.40
Baa2	3.73	2.41
B2	3.78	2.41
Ba2s	4.75	2.41
Bs	4.38	2.42
B2s	4.17	2.48
Bas	4.11	2.60
Ba	4.58	2.72

TAB. 5.3 - *Temps d'exécution des différentes versions parallèles du programme `gauss`*

Le tableau 5.3 donne les résultats obtenus pour ces dix versions du programme `gauss` parallélisé. Pour chaque version du programme parallèle nous donnons deux temps d'exécution, le premier appelé `temps écoulé` ("elapsed time") représente le temps d'exécution total du programme parallèle (il cumule le temps passé sur les processeurs parallèles et le temps passé sur le frontal), le second appelé `temps occupé` ("busy time") représente le temps d'exécution passé réellement sur les processeurs parallèles.

En ce qui concerne le temps `occupé`, nous apercevons qu'il est le même pour tous, sauf pour les cas où nous avons de l'alignement explicite. Nous avons déjà expliqué (voir section 5.3.1) que ceci est du au problème de mémoire que l'utilisation de la directive `ALIGN` entraîne.

Par contre, pour le temps `écoulé`, les différences sont plus sensibles et s'expliquent en partie par le fait que cette mesure peut être influencée par

4. Le deuxième alignement est `INS_2` sur `INS_1`.

les autres programme purement séquentiels s'exécutant sur le frontal.

Nous pouvons donc retenir qu'il ne faut pas utiliser la directive `ALIGN` systématiquement, que les gains obtenus en ajustant les longueurs des dimensions sont compensés par les pertes que cela engendre, et enfin, qu'un appel explicite à la fonction `SPREAD` n'est pas nécessaire lorsque la diffusion est évidente (*i.e.* facilement détectable par le compilateur)[5].

Dans une seconde expérimentations, nous avons effectué une comparaison des performances obtenues suivant la version du calcul de la fonction de placement utilisée (*i.e.*, avec ou sans optimisation des communications).

Nous avons effectué ces expérimentations sur un programme de multiplication matrices carrées de taille $n \times n$ (le programme séquentiel initial est donné figure 5.1, page 146). Etant donné que pour ce type de calcul il existe une fonction intrinsèque `MATMUT`, nous avons également comparé nos performances avec celles de cette fonction intrinsèque.

Avec la version initiale de la fonction de placement, nous obtenons le placement suivant :

$$\begin{array}{ll} \pi_{s1}(i,j) & = (i+j \quad 0) \\ \pi_{s2}(i,j) & = (0 \qquad i+j) \\ \pi_{s3}(i,j) & = (i \qquad j) \\ \pi_{s4}(i,j,k) & = (i \qquad j) \end{array}$$

Avec la nouvelle version, nous obtenons le placement suivant :

$$\begin{array}{ll} \pi_{s1}(i,j) & = (i \quad j) \\ \pi_{s2}(i,j) & = (i \quad j) \\ \pi_{s3}(i,j) & = (i \quad j) \\ \pi_{s4}(i,j,k) & = (i \quad j) \end{array}$$

Nous pouvons donc noter que l'instruction $s4$ sur laquelle nous avons détecté des diffusions a le même placement dans les deux versions. Néanmoins, avec la nouvelle version, les placements des instructions d'initialisation ($s1$ et $s2$) sont alignés sur celui de $s4$ alors que ce n'est pas le cas avec la version initiale. C'est cette différence d'alignement qui va expliquer les écarts des temps d'exécutuion obtenus.

5. C'est la fonction de placement qui met en évidence les diffusions réalisables.

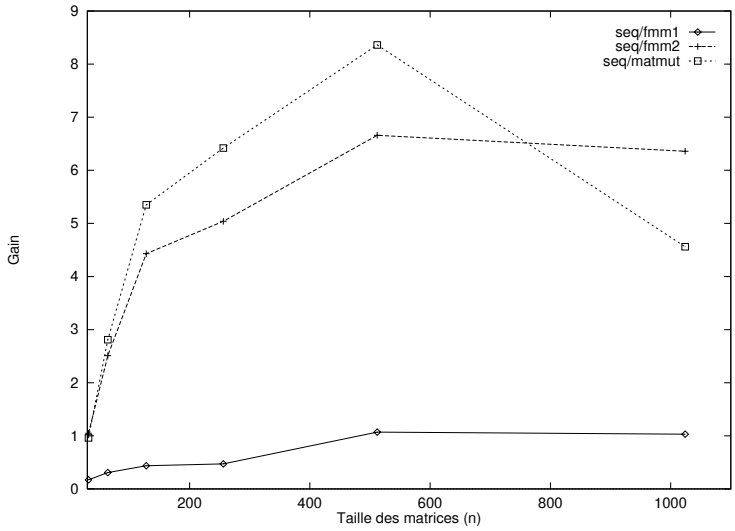


FIG. 5.4 - *Gain de performance du programme* `fmm` *sur la CM-5, sans VU*

Nous appellerons `fmm1` la version parallélisée à partir de l'ancienne fonction de placement et `fmm2` celle parallélisé à partir de notre nouvelle fonction de placement. Nous avons effectué des mesures pour six valeurs différentes de n : 32, 64, 128, 256, 512 et 1024. La figure 5.4 donne le gain du temps d'exécution des deux versions parallèles (`fmm1` et `fmm2`), ainsi que de la fonction intrinsèque `MATMUT`, par rapport à la version séquentielle (`seq`) en fonction de la taille des matrices.

Les performances du programme séquentiel sont calculées avec un programme Fortran 77 exécuté sur le frontal. Afin de pouvoir les comparées avec celles obtenues par les programmes parallèles, nous avons fait en sorte que les unités vectorielles (VU) ne soient pas utilisées, de sorte que chaque nœud de la machine est un processeur Sparc seul. Le processeur du frontal n'étant pas identiques à ceux-ci, nous avons également ajusté les temps d'exécution du programme séquentiel d'un facteur 1.15 [6].

Sur cet exemple simple, nous voyons donc que la nouvelle version de la fonction de placement nous permet d'obtenir des gains beaucoup plus importants par rapport à la version intiale, à condition bien sûr que les

6. Nous avons mesuré expérimentalement ce facteur en exécutant le même programme séquentiel sur chacun des nœuds et sur le frontal.

communications optimisées soient réalisables. De plus, ces performances sont du même ordre que celles obtenues en faisant appel à la fonction `MATMUT`.

La baisse du gain pour des grandes valeurs de N est du à un problème de mémoire-cache (antémémoire). En effet, dans ce mode d'exécution (sans VU), seul le cache du processeur SPARC est utilisé (pas ceux des VU). Lorsque la taille des données dépasse une certaine limite, elles ne tiennent plus dans le cache et doivent être stockées dans la mémoire principale qui a un accès beaucoup plus lent ([FWPS92]).

A titre indicatif, la figure 5.5 donne le gain de performance[7] pour ce même exemple, mais cette fois en utilisant les unités vectorielles.

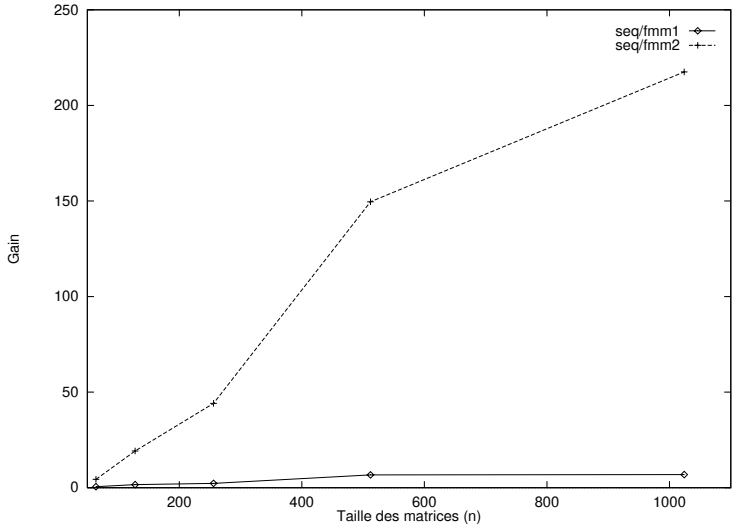


FIG. 5.5 - *Gain de performance du programme `fmm` sur la CM-5, avec VU*

La différence entre `FMM1` et `FMM2` est donc accentué, et nous pouvons également noté que les unités vectorielles sont bien utilisées puisque le gain de `FMM2` pour $N = 1024$ passe de 6 à 200.

7. Le temps séquentiel de comparaison utilisé est le même que précédemment.

5.4.2 Cray-T3D

Nous n'avons malheureusement pas pu faire les expérimentations que nous aurions souhaité sur le Cray-T3D en raison du retard dans sa mise en service, et notamment de son compilateur CRAFT.

Toutes nos expérimentations sur Cray-T3D ont été effectuées sur la machine du Centre d'Etudes de Grenoble du CEA. Cette machine posséde 128 nœuds de calcul mais pour des raisons techniques, nous avons utilisé une partition de 32 nœuds.

Les quelques exprérimentations que nous avons réalisées nous ont seulement permis de tester la justesse du code généré (les restrictions imposées par la version actuelle du compilateur sont données au chapitre précédent) et, également, d'avoir une première idée de l'efficacité de notre méthode.

Nous avons repris notre programme de multiplication de matrice utilisé précédemment et avons comparé les temps d'exécution des différentes versions en fonction du nombre de processeurs. La figure 5.6 donne les résultats pour $N = 256$.

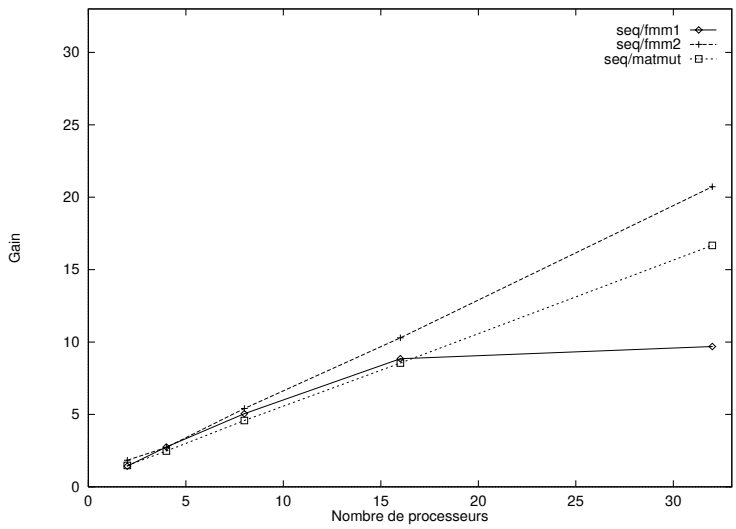


FIG. 5.6 - *Gain de performance du programme* `fmm` *sur le T3D*

Nous remarquons que l'efficacité de nos trois programmes équivalente

lorsque peu de processeurs sont utilisés. Par contre, sur 32 processeurs, notre programme `FMM1` n'est plus aussi efficace, ceci étant du aux nombreuses (et coûteuses) communications qu'il engendre. Nous voyons donc que, comme sur la CM-5, notre méthode permet d'améliorer sensiblement les performances par rapport à sa version initiale.

Chapitre 6

Conclusions

Comme nous l'avons déjà dit, ce travail de thèse s'intègre dans un projet de recherche visant à la génération et l'optimisation de code pour machines massivement parallèles, et reposant sur une collaboration tripartite : le CEA (Centres de Limeil-Valenton, Projet Calcul Parallèle), l'ENSMP (Ecole des Mines de Paris, Centre de Recherche en Informatique) et l'UVSQ (Université de Versailles et Saint-Quentin, laboratoire PRISM).

Le but de ce projet est d'intégrer et d'étendre la méthode de parallélisation étudiée à l'UVSQ (développée dans le paralléliseur PAF et fondée sur le *graphe du flot de données*, la *base de temps* et la *fonction de placement*) au sein du paralléliseur PIPS de l'ENSMP ; la contribution du CEA est de réaliser la base initiale de ce travail.

6.1 Contribution

Nos travaux ont ainsi contribué à étendre la méthode de calcul de la fonction de placement dans le but de pouvoir utiliser au maximum les communications les plus efficaces des machines massivement parallèles. En effet, comme nous avons pu le vérifier, les diffusions, les réductions et les translations sont effectuées beaucoup plus rapidement par rapport aux communications générales.

Nous proposons un algorithme original de calcul de la fonction de placement qui traite le cas particulier des diffusions avant de traiter le cas général.

Il faut noter que cet nouvel algorithme à la même complexité que l'algorithme initial proposé par Feautrier ([Fea93]).

De plus, le but final de notre étude étant de générer du code parallèle dans un langage source d'une machine massivement parallèle, nous garantissons la portabilité de cette méthode.

En effet, il est important de noter que tout notre travail reste indépendant de la machine cible jusqu'au stade ultime de production de code parallèle source.

Ceci nous a donc permis de ne pas être attaché à une architecture particulière et ainsi de générer facilement du code pour la CM-5 et le Cray-T3D qui ont des architectures assez différentes.

En ce qui concerne la génération de code, il nous a fallu adapter la méthode proposée par Collard ([Col93]) pour prendre en compte les communications optimisées. Nous proposons une méthode pour générer les instructions dans lesquelles apparaissent des primitives engendrant des diffusions ou des translations.

En parallèle, nous avons contribué à l'intégration dans PIPS des techniques de parallélisation mises en œuvre dans le paralléliseur PAF. Pour notre part, nous avons développé le calcul de la fonction de placement avec les extensions présentées dans cette thèse, ainsi que la phase de réindexation et de génération de code.

Cette intégration et extension continue au CEA et tous ces développement ont, bien entendu, été intégrés dans la version de PIPS de l'Ecole des Mines de Paris. Mais, de plus, l'Université de Versailles a également récupéré une version de PIPS contenant toutes ces extensions, dans le but de continuer ses développement en langage `C` plutôt qu'en `le_lisp` (pour PAF). Enfin, le laboratoire de l'IRISA (à Rennes) spécialisé dans la parallélisation automatique a fait savoir qu'il serait intéressé d'obtenir lui aussi cette nouvelle version de PIPS afin de profiter de ces extensions.

En ce qui concerne l'efficacité de notre méthode, nous n'avons malheureusement pas pu la tester aussi intensément sur le T3D que sur la CM-5, en

raison de la mise en service tardive de la première.

Néanmoins, les résultats que nous avons obtenus sont encourageants. Il ressort de nos expérimentations que, lors du calcul de la fonction de placement, la prise en compte du type de communication à engendrer permet d'améliorer considérablement les performances.

Il serait nécessaire de poursuivre plus avant ces expérimentations afin d'évaluer plus généralement cette efficacité.

6.2 Travaux futurs

Afin d'améliorer davantage le code parallèle généré, voici les différentes optimisations qu'il serait intéressant d'étudier.

Tout d'abord, grâce à sa portabilité, il serait aisé de tester notre méthode sur une gamme plus étendue de machines puisqu'il suffirait de rajouter une phase de production de code parallèle source comme nous l'avons fait pour CM Fortran et Craft Fortran.

De plus, pour tenir encore plus compte des spécificités relatives aux coûts des communications de chaque machine cible, il serait assez facile de modifier l'ordre de priorité des communications en changeant l'ordre de traitement des arcs.

Une optimisation importante est qu'il serait nécessaire de réduire la taille mémoire des tableaux utilisés dans le programme parallélisé. En effet, dans notre exemple sur le programme `gauss` parallélisé, nous avons un tableau de taille n^3 alors que dans la version séquentielle il est de taille n^2.

De plus, la taille du code parallèle généré pourrait être réduite en éliminant de nombreuses conditionnelles (et les instructions contenues à l'intérieure) dues au traitement particulier de la valeur initiale d'un tableau.

Par exemple, l'instruction $s2$ du programme `gauss` est découpée en deux afin de séparer l'initialisation du tableau avec le traitement générale. En augmentant de un la taille de la première dimension du tableau ins_2 et en l'initialisant convenablement avant le début de la boule d'ordonnancement

global, nous pourrions alors supprimer le test sur `S2t0` ainsi que les instructions qu'il contient.

Les conditionnelles mettant en jeu uniquement des paramètres de structure pourraient également être supprimées en faisant intervenir l'utilisateur.

Par exemple, dans notre programme `gauss`, en supposant que l'inégalité $n \geq 4$ est toujours vraie nous pouvons supprimer tous les tests mettant uniquement en jeu ce paramètre.

Avec ce type de question simple, nous proposerions alors une parallélisation intéractive intelligente.

Une autre manière de simplifier et d'optimiser le code parallèle serait d'utiliser des techniques d'éclatement de boucles (par exemple avec les composantes fortement connexes, voir section 3.5.4). Néanmoins, ce type de méthode doit être utiliser avec prudence car elle peut faire exploser la taille du code.

Au niveau de la distribution des données sur les processeurs physiques, de nombreux modèles de programmation proposent un mécanisme de pondération des axes. Un tel outil est utilisé pour privilégier certains axes par rapport à d'autres lorsque, par exemple, de nombreuses communications se font parallèlement à ceux-là.

Nous pourrions donc générer automatiquement une pondération des axes à partir de notre connaissance de l'ensemble des communications.

Comme nous l'avons vu section 4.6, notre calcul de la fonction de placement avec optimisation des communications nous donne, dans quelques cas particuliers, des résulats moins satisfaisants que la méthode directe.

Une étude sur la prévision d'un tel comportement et la possibilité de faire un choix de la méthode pourrait donc être intéressante.

Enfin, pour améliorer encore plus les performances, il serait nécessaire d'intégrer les travaux sur la prise en compte des réductions (et des scans, voir section 4.3, page 118) afin d'expliciter leurs usages dans le code source généré.

D'autres types moins connus de commmunications bien optimisées (particuliers à certaines machines) pourraient également être pris en compte comme le "spread-with-add" (une réduction suivie d'une diffusion) ou encore les "stencils" (schémas particuliers de communication).

Le gain obtenu avec seulement la prise en compte des diffusions laisse présager que de telles optimisations seraient très bénéfiques.

Annexe A

Implantation dans PIPS

L'ensemble des techniques décrites dans cette thèse ont été implantées dans le paralléliseur PIPS. Cet annexe a pour but de présenter cette implantation. Il se découpe en cinq sections.

La première section présente les grandes lignes du projet PIPS. Dans la seconde section, nous décrivons un peu plus en détail l'outil de génie logiciel Newgen. La troisième section présente la bibliothèque d'algèbre linéaire. L'implantation de toutes ces nouvelles techniques est décrite dans la troisième section. Enfin, la représentation intermédiaire de PIPS est donnée dans la dernière section.

A.1 Le projet PIPS

Le projet PIPS (Paralléliseur Interprocédural de Programmes Scientifiques) a pour but de réaliser une compilateur paralléliseur qui transforme un programme écrit en Fortran 77 en un programme écrit dans un des dialectes de Fortran pour machine parallèle.

Ce projet a été à l'origine financé par la DRET (Direction des Recherches Etudes et Techniques, Ministère de la Défense) dans un contrat d'une durée de deux ans (de mai 1988 à septembre 1990, [IJ90]). Pour la réalisation de ce paralléliseur, ce contrat prévoyait trois services :

1. élaboration du sous-langage de programmation et constitution du jeu

de tests ;

2. réalisation d'un analyseur syntaxique et sémantique interprocédural ;

3. détection de parallélisme et production de programmes parallèles.

PIPS se distingue des autres paralléliseurs ou vectoriseurs en plusieurs points particuliers.

Tout d'abord, il intègre les travaux de recherches du Centre de Recherche en Informatique (CRI) de l'Ecole des Mines de Paris en matière de détection du parallélisme interprocédural.

Ensuite, il peut traiter des "gros" programmes. En effet, d'une part il n'impose pratiquement pas de restrictions par rapport à la norme ANSI Fortran 77, et d'autre part sa vitesse de traitement est élevée.

Enfin, il bénéficie de structures de données extrêmement compactes grâce à l'utilisation d'un outil de génie logiciel appelé NewGen (voir ci-dessous).

Ce paralléliseur est composé de **phases** qui utilisent et produisent des **ressources**. Ces ressources sont les informations ou les résultats démandés par l'utilisateurs (voir la figure A.1 qui donne la structure du paralléliseur PIPS).

Un gestionnaire de ressource `pipsmake` exécute les phases nécessaires à la requête de l'utilisateur, tout en gardant la cohérence des enchaînements.

De plus, un gestionnaire de base de données `pipsdbm` gère les accès aux ressources, et les échanges entre les fichiers et la mémoire.

Les phases se classent en deux catégories :

Phases d'analyse : elles produisent les ressources attachées à la Représentation Intermédiaire (voir ci-dessous) à partir du texte séquentiel, et construisent le graphe de dépendance.

Elles font appel à `pipsdbm` pour les lectures des ressources qu'elles utilisent et les écritures de celles qu'elles produisent

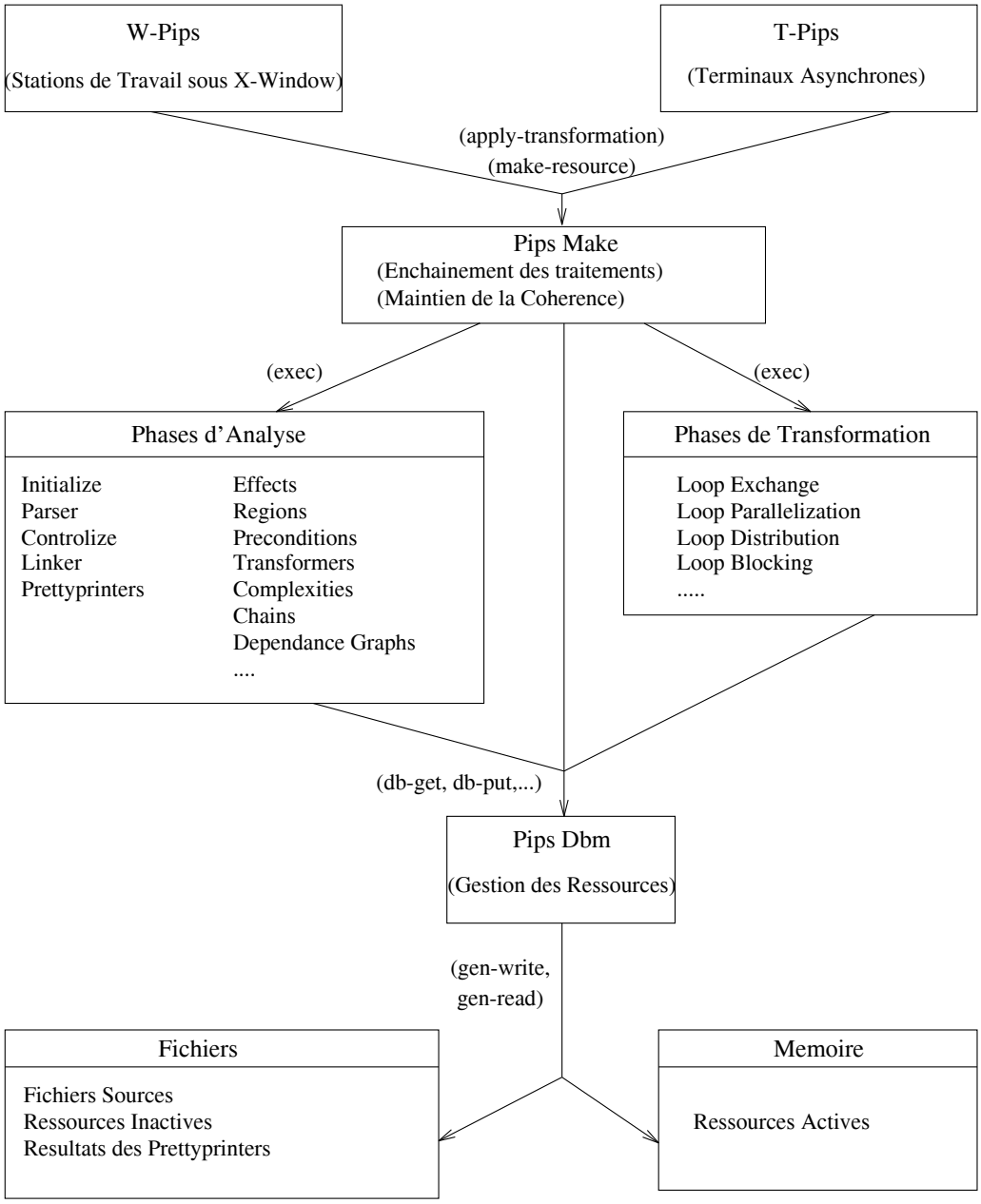


FIG. A.1 - *Structure du paralléliseur PIPS*

Phases de transformation : elles produisent les ressources qui résultent d'une modification du programme source de façon à réduire le nombre de dépendances et générer le code parallèle.

Elles utilisent également `pipsdbm` pour les ressources qu'elles utilisent et produisent.

Toutes ces phases ne sont pas tout à fait indépendantes les unes des autres. Ainsi, la plupart des transformations nécessitent que les résultats d'une ou plusieurs analyses soient disponibles ; par exemple, la parallélisation ne peut être faite que si les dépendances ont été préalablement calculées.

Quand aux analyses, il existe un ordre partiel entre elles ; par exemple, le calcul des dépendances ne peut être fait que si les effets sont disponibles. Cet ordre d'exécution est géré par `pipsmake`.

La figure A.2 donne l'ordre d'enchaînement des phases principales de PIPS.

Un dernier outil mis à la disposition des utilisateurs est la définition de variables de *propriété*, appelées `properties`. Ces variables sont booléenes ; elles peuvent être utilisée dans certaines phases pour conditionner le traitement à faire.

Par exemple, pour notre phase de production de code parallèle, nous avons conditionné le langage cible avec des `properties` :

```
PRETTYPRINT_CMFORTRAN
PRETTYPRINT_CRAFT
```

Leur valeur par défaut est `FAUX`. Lorsque nous souhaitons générer du code pour la CM-5, la valeur de variable `PRETTYPRINT_CMFORTRAN` doit alors être mise à `VRAI`.

A.2 Structures de donnée

L'ensemble des phases de PIPS sont construites autour d'une structure de donnée, appelée Représentation Intermédiaire (RI), qui représente l'arbre

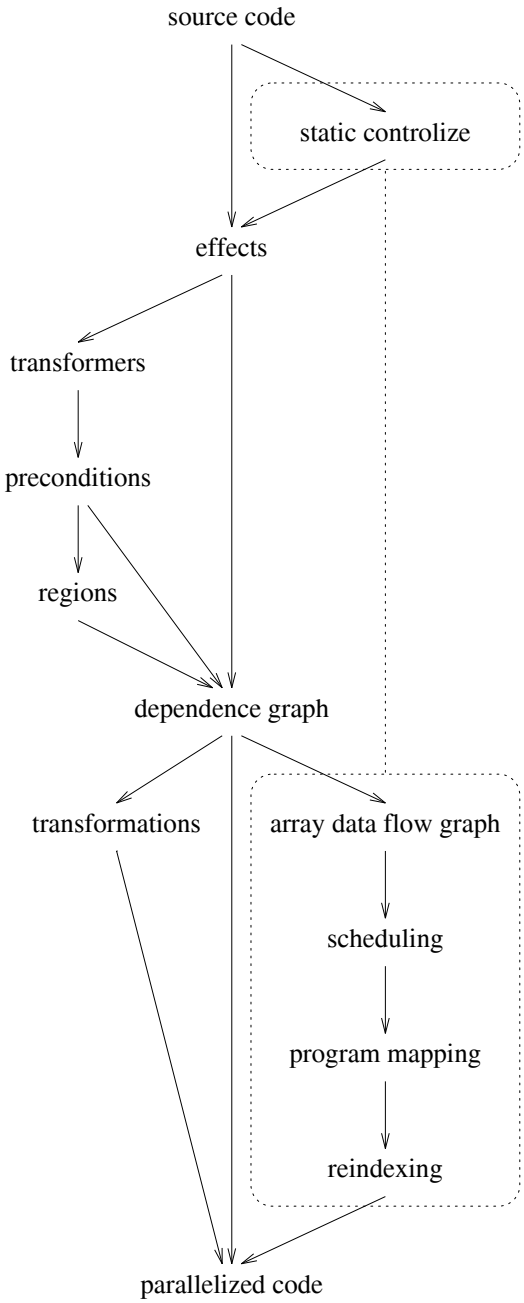


FIG. A.2 - *Enchaînement des phases de PIPS (En pointillés : apports réalisés dans le contexte de cette thèse)*

syntaxique d'un programme Fortran 77. La RI (voir annexe A.5) à été implantée à l'aide d'un outil de génie logiciel développé à l'Ecole des Mines appelé **Newgen** ([JT89]).

Cet outil permet de faciliter la définition et l'implantation de structures de données complexes. Son utilisation se fait de la manière suivante :

1. les types de données sont définis à l'aide d'un langage de haut niveau appelé DDL (pour "Domain Definition Language") ;
2. la définition des domaines est compilée pour générer :
   - une implantation de ces types de données en C ou en Common-Lisp ;
   - des fonctions de création, destruction, mise à jour, écriture et lecture sur fichier d'objets de ce type.

A titre d'exemple, voici comment la structure de **graphe** est définie dans PIPS :

```
graph = vertices:vertex*
vertex = vertex_label x successors:successor*
successor = arc_label x vertex
```

Ceci définit en fait trois domaines (ou structures): `graph`, `vertex` et `successor`.

Le domaine `graph` définit un graphe comme une liste de nœuds (`vertex`) ; chaque nœud contient une information (`vertex_label`) et la liste des arcs (`successor`) sortant ; chaque arc contient une information (`arc_label`) et le nœud vers lequel il pointe.

Pour ces trois domaines, sont définies automatiquement une série de fonctions. Par exemple, pour `graph`, voici un extrait de ce que newgen génère :

```
typedef chunk *graph ;
#define graph_undefined ((graph)chunk_undefined)
#define graph_undefined_p(x) ((x)==graph_undefined)
```

```
#define copy_graph(x) ((graph)gen_copy_tree((chunk *)x))
#define write_graph(fd,obj) (gen_write(fd,(chunk *)obj))
#define read_graph(fd) ((graph)gen_read(fd))
#define free_graph(o) (gen_free((chunk *)o))
#define graph_vertices(li) ((((li)+1)->l))
```

Nous distinguons, notamment, la fonction (`graph_vertices`) permettant d'accéder au champs `vertices` d'un objet de ce type, ou encore la fonction (`copy_graph`) effectuant une copie physique d'un objet de ce type.

Le type `chunk` est un type générique sur lequel sont construits tous les autres. Tous les objets déclarés avec des types définis en Newgen sont des pointeurs vers des objets de type `chunk`:

```
typedef union chunk {
        unit u ;
        bool b ;
        char c ;
        int i ;
        float f ;
        string s ;
        struct cons *l ;
        set t ;
        hash_table h ;
        union chunk *p ;
} chunk ;
```

A.3 Bibliothèque linéaire

PIPS dispose d'une bibliothèque d'algèbre linéaire et de calcul en nombre entier, appelée **bibliothèque C3**. Cette bibliothèque, qui est structurellement indépendante de PIPS, permet de définir et de manipuler des objets du type *vecteur entiers*, *matrice rationnelle* ou *polyèdre convexe à bornes rationnelles*.

Comme PIPS, cette bibliothèque est composée de modules séparés, chacun traitant un certain type de données (voir [Lam93]).

D'après Halbwachs (voir [Hal79]), un polyèdre convexe peut être défini de deux manières équivalentes soit par un système d'égalités et d'inégalités linéaires (appelé *système de contrainte*), soit par un système de sommets, de rayons et de droites (appelé *système générateur*).

La bibliothèque C3 utilise ces deux types de représentation et fournit des fonctions de passage de l'une vers l'autre.

Ces deux représentations utilisent des objets du type vecteur définis dans C3 qui est en fait une structure de vecteur *creux*. Par contre, le type matrice définit une structure de matrice *pleine*. La bibliothèque C3 fournit des fonctions de passage d'une structure creuse vers une structure pleine et vice-versa.

> Voici, par exemple, la définition de la structure de vecteur creux : Le domaine `Pvecteur` est utilisé pour représenter les expressions linéaires telles que `3I+2` ou des contraintes linéaires telles que `3I + J <= 2` ou `3I == J`
>
> Un objet de type `Pvecteur` est une suite de monômes, un monôme étant un couple (coefficient,variable). Le coefficient d'un tel couple est un entier, positif ou négatif. La variable est une entité, sauf dans le cas du terme constant qui est représenté par la variable prédéfinie de nom `TCST`.

Les fonctions proposées vont de la simple manipulation de ces objets pour leur appliquer des transformations usuelles telles que la combinaison linéaire de deux vecteurs ou l'inversion d'une matrice, aux calculs plus complexes tels que la résolution d'un problème de programmation linéaire ou la détermination de l'enveloppe convexe de deux polyèdres.

Enfin, en réutilisant de manière différente la structure de type vecteur, un module de cette bibliothèque propose des fonctions de définition et de manipulation de polynômes à coefficients réels[1].

1. Ce type de calculs ne fait pas partie de l'algèbre linéaire ou du calcul en nombre entier, mais pour des raisons pratiques il a été intégré à cette bibliothèque.

A.4 Implantation de la méthode

L'environnement de développement de PIPS offre à l'utilisateur les moyens de développer de manière modulaire. Chaque module définit des fonctions dont les appels peuvent être gérés par `pipsmake`, auquel cas chacune d'elles correspond à une phase de PIPS.

Sept nouveaux modules ont été ajoutés à PIPS pour l'implantation de l'ensemble de cette méthode. L'ensemble de ces modules ont été intégrés au schéma d'enchaînement des phases, comme le montre la figure A.2.

A.4.1 `static_controlize`

Ce module[2] est à la base de tous les autres calculs de cette méthode. Il définit deux phases (`static_controlize` et `print_code_static_control`) et une ressource (`static_control`) qui déterminent les parties du programme qui sont "à contrôle statique" et associent à chaque instruction son domaine d'exécution.

De plus, ce module modifie le code en normalisant toutes les boucles de telle manière que leur pas d'incrémentation soit égal à un. Cette normalisation introduit des nouveau comptes-tours de boucle appelé "NLC". Toutes les occurences des anciens comptes-tours sont remplacées par les nouveaux.

Enfin, il collecte les paramètres de structure et cherche les variables qui sont susceptibles d'être considérées comme tels.

Ce module utilise directement l'information produite par l'analyse lexicale et syntaxique de PIPS[3].

Voici comment ce module est définit au gestionnaire des ressources :

```
static_controlize               > MODULE.static_control
        < PROGRAM.entities
        < MODULE.code

print_code_static_control       > MODULE.printed_file
```

2. Environ 3000 lignes de code implantées par A. Leservot.
3. Le fait qu'il modifie le code n'est pas spécifé à cause d'une erreur dans `pipsmake`.

```
< PROGRAM.entities
< MODULE.code
< MODULE.static_control
```

La ressource `static_control` associe à chaque instruction un objet du type `static_control` définit en Newgen comme suit :

```
static_control = yes:bool x params:entity* x loops:loop* x
                 tests:expression*
```

Le domaine `static_control` définit la structure de l'information qui est associée à une instruction. Le sous-domaine `yes` indique si ce statement se trouve dans une zone à contrôle statique ; si ce n'est pas le cas, les trois sous-domaines suivant sont vides. Le sous-domaine `params` donne la liste des paramètres de structure de la zone à contrôle statique dans laquelle se trouve ce statement. Le sous-domaine `loops` donne la liste des boucles englobantes de ce statement. Le sous-domaine `tests` donne la liste des conditions trouvées dans les tests englobants de ce statement.

Le domaine `expression` permet de stocker les expressions sous deux formes. La première est une description de l'expression telle qu'elle apparaît dans le texte source du programme et la deuxième est une description compilée des expressions linéaires sous forme de Pvecteur (voir section A.3).

Le domaine `loop` permet de représenter les boucles du type DO Fortran ou FOR Pascal.

Tout objet ayant un nom dans un programme Fortran est représenté par une `entity`. Un tel objet peut être un module, une variable, un common, un opérateur, une constante, un label, etc.

Ces trois derniers domaines sont définis dans la RI (voir annexe A.5).

A.4.2 array_dfg

Ce module[4] définit deux phases (`array_dfg` et `print_array_dfg`) et deux ressources (`adfg` et `adfg_file`) pour le calcul du graphe du flot de données

4. Environ 5500 lignes de code implantées par A. Leservot.

(DFG). Il reprend directement le graphe de dépendance produit par PIPS et utilise le résultat produit par la phase `static_controlize`.

Bien entendu, la méthode de calcul ne peut s'appliquer que sur des programmes à contrôle statique, sinon l'erreur est signalée.

Voici comment ce module est définit au gestionnaire des ressources :

```
array_dfg                       > MODULE.adfg
        < PROGRAM.entities
        < MODULE.code
        < MODULE.static_control
        < MODULE.dg
        < MODULE.preconditions

print_array_dfg                 > MODULE.adfg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.adfg
```

La ressource `adfg` est un objet de type `dataflow` définit en Newgen comme suit :

```
dataflow = reference x transformation:expression* x
           governing_pred:predicate x communication
dfg_vertex_label = statement:int x exec_domain:predicate x
                     sccflags
dfg_arc_label = dataflows:dataflow*
communication = broadcast:predicate x reduction:predicate x
                shift:predicate
```

Le domaine `dataflow` permet de décrire les informations nécessaires à la description du flot d'une donnée. Le sous-domaine `reference` donne la référence sur laquelle porte la dépendance. Le sous-domain `transformation` donne dans l'ordre la liste d'expressions qui donnent la valeur des indices des boucles englobantes de l'instruction source en fonction des indices des boucles

englobantes de l'instruction destination. Le sous-domaine `governing_pred` donne le prédicat de contrôle de la dépendance.

Le sous-domaine `communication` spécifie le type de communication que ce flot est susceptible d'engendrer. Ce champ n'est utilisé que dans la phase de calcul de la fonction de placement.

Le domaine `dfg_vertex_label` définit la structure des informations qui seront attachées à un nœud du DFG. Le sous-domaine `statement` spécifie l'instruction sur laquelle porte la dépendance. Le sous-domaine `exec_domain` est le domaine d'exécution de cette instruction ; c'est un prédicats, qui correspond aux bornes des boucles englobantes ainsi qu'aux tests englobants. Le sous-domaine `sccflags` contient diverses informations utiles pour le calcul des composantes fortement connexes.

Le domaine `dfg_arc_label` définit la structure des informations qui seront attachées à un arc du DFG, *i.e.* le flot de données passant par cet arc. La structure de graphe elle-même (domaine `graph`) est donnée plus haut (voir section A.2).

Le domaine `communication` permet de spécifier le type de communication qui peut être engendrée par le flot auquelle il est associé. Le sous-domaine `broadcast` correspond à une diffusion. Le sous-domaine `reduction` correspond à une récurrence. Le sous-domaine `shift` correspond à une translation.

Le domaine `reference` est utilisé pour représenter une référence à un élément de tableau. Les variables scalaires étant représentées par des tableaux de dimension zéro, les références à des scalaires sont aussi prises en compte. Elles contiennent une liste vide d'expressions d'indices.

Le domaine `predicate` définit une relation entre valeurs de variables scalaires entières (*i.e.* un système d'équations et d'inéquations linéaires etières.).

Ces deux derniers domaines sont définis dans la RI (voir annexe A.5).

A.4.3 `scheduling`

Ce module[5] définit deux phases (`scheduling` et `print_bdt`) et deux ressources (`bdt` et `bdt_file`) pour le calcul de la base de temps (BDT). Celle-ci associe à chaque instruction une fonction d'ordonnancement.

5. Environ 4500 lignes de codes implantées par A. Cloué.

Pour réaliser ce calcul, ce module reprend directement le DFG produit par la phase précédente.

Voici comment ce module est définit au gestionnaire des ressources :

```
scheduling              > MODULE.bdt
        < PROGRAM.entities
        < MODULE.code
        < MODULE.static_control
        < MODULE.adfg

print_bdt              > MODULE.bdt_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.bdt
```

La ressource `bdt` est un objet de type `bdt` définit en Newgen comme suit :

```
bdt = schedules:schedule*
schedule = statement:int x predicate x dims:expression*
```

Le domaine `bdt` définit la base de temps pour l'ensemble des instructions du programme. Elle se compose tout simplement d'une liste de fonctions d'ordonnancement, chacune associée à une instruction et un prédicat.

Le domaine `schedule` est utilisé pour décrire la fonction d'ordonnancement associée à une instruction et un prédicat. C'est en fait une liste de fonctions, car la base de temps peut être multidimensionnelle. Le sous-domaine `statement` spécifie l'instruction à laquelle correspond cette fonction. Le sous-domaine `predicate` donne le prédicat d'existence de cette fonction. Enfin, le sous-domaine `dims` donne les différentes expressions de la fonction multidimensionnelle.

A.4.4 prgm_mapping

Ce module[6] définit deux phases (`prgm_mapping` et `print_plc`) et deux ressources (`plc` et `plc_file`) pour le calcul de la fonction de placement (PLC).

6. Environ 6000 lignes de code implantées par A. Platonoff.

Celle-ci associe une fonction multidimensionnelle de placement à chaque instruction.

Pour réaliser ce calcul, ce module reprend directement le DFG et la BDT produits par les phases précédentes.

Voici comment ce module est définit au gestionnaire des ressources :

```
prgm_mapping            > MODULE.plc
        < PROGRAM.entities
        < MODULE.code
        < MODULE.static_control
        < MODULE.adfg
        < MODULE.bdt


print_plc               > MODULE.plc_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.plc
```

La ressource `plc` est un objet de type `plc` définit en Newgen comme suit:

```
plc = placements:placement*
placement = statement:int x dims:expression*
```

Le domaine `plc` définit la fonction de placement pour l'ensemble des instructions du programme. Elle se compose tout simplement d'une liste de de placements, chacun associé à une instruction.

Le domaine `placement` est utilisé pour décrire le placement associée à une instruction. C'est en fait une liste d'expressions, car la fonction de placement peut être multidimensionnelle. Le sous-domaine `statement` spécifie l'instruction à laquelle correspond ce placement. Enfin, le sous-domaine `dims` donne les différentes expressions de la fonction multi-dimensionnelle.

A.4.5 reindexing

Ce module[7] définit trois phases (`reindexing` pour le réindexation du code, `print_parallelizedCMF_code` pour la production de CM Fortran et `print_parallelizedCRAFT_code` pour la production de CRAFT Fortran) et une ressource (`reindexed_code`) pour la transformation du code. Il reprend directement le DFG, la BDT et la PLC produits par les phases précédentes.

Pour la génération de code proprement dite, deux versions sont pour le moment disponibles. La première produit du CM Fortran et le résultat est stocké dans un fichier suffixé par `.fcm`. La seconde version produit du CRAFT Fortran et le résultat est stocké dans un fichier suffixé par `.craft`.

Voici comment ce module est définit au gestionnaire des ressources :

```
reindexing              > MODULE.reindexed_code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.static_control
        < MODULE.adfg
        < MODULE.bdt
        < MODULE.plc

print_parallelizedCMF_code       > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.adfg
        < MODULE.bdt
        < MODULE.plc
        < MODULE.reindexed_code

print_parallelizedCRAFT_code     > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.adfg
        < MODULE.bdt
        < MODULE.plc
        < MODULE.reindexed_code
```

7. Environ 6000 lignes de code implantées par A. Cloué et A. Platonoff.

A.4.6 pip

Ce module[8] définit une bibliothèque pour l'intégration du logiciel PIP à l'environnement de PIPS.

Le logiciel PIP a été développé en langage `C` par Feautrier. Ce module définit les fonctions pour l'interface en entrée et en sortie de PIP.

Pour des raisons techniques, il n'a pu encore être intégré à la bibliothèque linéaire C3.

Pour réaliser l'interface en sortie, il nous a fallu décrire la structure de `quast` utilisé par PIP. Nous l'avons fait en Newgen[9] :

```
quast = quast_value x newparms:var_val*
quast_value = quast_leaf + conditional
conditional = predicate x true_quast:quast x false_quast:quast
quast_leaf = solution:expression* x leaf_label
leaf_label = statement:int x depth:int
var_val = variable:entity x value:expression
```

Le domaine `quast` est utilisé pour contenir les informations renvoyées par PIP. Ces informations sont l'expression (dépendant de certaines conditions) de la valeur de chaque variable du système en fonction des paramètres de structure. Le sous-domaine `quast_value` donne la valeur du quast.

Le sous-domaine `newparms` donne la liste des nouveaux paramètres de structure introduit par le quast. Pour chaque nouveau paramètre, l'information est composée d'une `variable` et d'une `value`. La `variable` est une entité représentant le nouveau paramètre de structure et la `value` est l'expression de sa valeur en fonction des autres paramètres de structure. La création de nouveau paramètre de structure est nécessaire lorsqu'apparaissent dans les expressions des divisions par des constantes, comme par exemple : $n/2$.

Le domaine `quast_value` contient la valeur du quast dans une structure directement inspirée de la structure "quast" utilisée dans PAF. Cette valeur est soit une feuille `quast_leaf`, soit un `conditional`.

8. Environ 3500 lignes de code dont 2500 implantées par F. Dumontet et A. Leservot ; les 1000 lignes restantes correspondent à celles implantées à l'origine par P. Feautrier.

9. C'est la raison de la non intégration de PIP dans `C3` : il aurait fallu définir une structure de `quast` sans utiliser Newgen.

Le domaine `conditional` est utilisé pour contenir les informations d'un `quast` qui dépendent d'un prédicat. Si le `predicate` est vrai, alors l'information utilisée est celle contenue dans `true_quast`, sinon l'information utilisée est celle contenue dans `false_quast`.

Le domaine `quast_leaf` contient une `solution` rendue par PIP et le champ `leaf_label` est utilisé lors du calcul d'une source dans le data flow graph. `Solution` est une liste d'expressions, dont chacune donne la valeur de la solution trouvée par PIP pour la variable de même rang dans la liste des variables du système (l'ordre de ces deux listes est donc **très** important).

Le domaine `leaf_label` est utilisé lors du calcul d'une source : il informe sur l'origine de cette source (`statement`) ainsi que sur la profondeur de la dependance (`depth`). Le quast rendu par PIP n'utilise pas ce champ.

Le domaine `var_val` est utilisé pour spécifier l'association d'une variable et de sa valeur exprimée en fonction d'autres variables.

A.5 Représentation Intermédiaire de PIPS

```
-- External domains
-- ----------------
external Psysteme ;
external Pvecteur ;

-- Domains
-- -------
action = read:unit + write:unit ;
approximation = may:unit + must:unit ;
area = size:int x layout:entity* ;
basic = int:int + float:int + logical:int + overloaded:unit + complex:int + string:value ;
call = function:entity x arguments:expression* ;
callees = callees:string* ;
code = declarations:entity* x decls_text:string ;
constant = int + litteral:unit ;
control = statement x predecessors:control* x successors:control* ;
dimension = lower:expression x upper:expression ;
effect =  persistant reference x action x approximation x context:transformer ;
effects =  effects:effect* ;
execution = sequential:unit + parallel:unit ;
expression = syntax x normalized ;
formal = function:entity x offset:int ;
functional = parameters:parameter* x result:type ;
instruction = block:statement* + test + loop + goto:statement + call + unstructured ;
loop = index:entity x range x body:statement x label:entity x execution x locals:entity* ;
mode = value:unit + reference:unit ;
normalized = linear:Pvecteur + complex:unit ;
parameter = type x mode ;
```

```
predicate = system:Psysteme ;
ram = function:entity x section:entity x offset:int x shared:entity* ;
range = lower:expression x upper:expression x increment:expression ;
reference = variable:entity x indices:expression* ;
statement = label:entity x number:int x ordering:int x comments:string x instruction ;
storage = return:entity + ram + formal + rom:unit ;
symbolic = expression x constant ;
syntax = reference + range + call ;
tabulated entity = name:string x type x storage x initial:value ;
test = condition:expression x true:statement x false:statement ;
transformer = arguments:entity* x relation:predicate ;
type = statement:unit + area + variable + functional + unknown:unit + void:unit ;
unstructured = control x exit:control ;
value = code + symbolic + constant + intrinsic:unit + unknown:unit ;
variable = basic x dimensions:dimension* ;
```

Annexe B

Parallélisation du programme gauss

B.1 Programme gauss après réindexation

```
PROGRAM GAUSS
INTEGER N
INTEGER s1q0,s1t0,s1p1,s1p0,s1flag0,s1t1
INTEGER s2q0,s2t0,s2p1,s2p0,s2flag0,s2t1
INTEGER s3t0,s3p1,s3p0,s3q0
INTEGER s4flag0,s4t3,s4t2,s4t1,s4q0,s4t0,s4p1,s4p0
INTEGER s5q0,s5t0,s5p1,s5p0,s5flag0,s5t3,s5t2,s5t1
INTEGER S0t0

REAL*4 A(1:N,1:N)

IF (3.LE.N) THEN
   s4t0 = 2+2*N
ELSE
   s4t0 = -1
ENDIF
IF (4.LE.N) THEN
   s4t1 = 2+2*N
ELSE
   s4t1 = -1
ENDIF
IF (2.LE.N) THEN
   s4t2 = 2*N
ELSE
   s4t2 = -1
ENDIF
IF (3.LE.N) THEN
   s4t3 = 2*N
ELSE
```

```
   s4t3 = -1
ENDIF
s4q0 = 0
s4flag0 = 0
IF (2.LE.N) THEN
   s2t0 = 1
ELSE
   s2t0 = -1
ENDIF
IF (3.LE.N) THEN
   s2t1 = 3
ELSE
   s2t1 = -1
ENDIF
s2q0 = 0
s2flag0 = 0
IF (2.LE.N) THEN
   s5t0 = -3+4*N
ELSE
   s5t0 = -1
ENDIF
IF (3.LE.N) THEN
   s5t1 = 1+2*N
ELSE
   s5t1 = -1
ENDIF
IF (1.LE.N.AND.N.LE.1) THEN
   s5t2 = -3+4*N
ELSE
   s5t2 = -1
ENDIF
IF (2.LE.N) THEN
   s5t3 = -1+2*N
ELSE
   s5t3 = -1
ENDIF
s5q0 = 0
s5flag0 = 0
IF (2.LE.N) THEN
   s1t0 = 0
ELSE
   s1t0 = -1
ENDIF
IF (3.LE.N) THEN
   s1t1 = 2
ELSE
   s1t1 = -1
ENDIF
s1q0 = 0
s1flag0 = 0
s3q0 = 0
DO S0t0 = 0, MAX(-3+4*N, 1, 2*N)
   IF (S0t0.EQ.s4t3) THEN
      DOALL s4p0 = 1, -2+N-s4q0
         ins4(MOD(s4q0, 2),s4p0) = ins3(s4p0)+ins2(-1+s4p0,
```

```
&         s4p0,-s4q0+N)*ins5(0)
       ENDDOALL
       s4t3 = s4t3+2
       IF (s4t3.GT.2*N) THEN
          s4t3 = -1
       ENDIF
       s4flag0 = 1
    ENDIF
    IF (S0t0.EQ.s4t2) THEN
       DOALL s4p0 = 0, 0
          ins4(MOD(s4q0, 2),s4p0) = ins3(s4p0)+A(s4p0+1,-s4q0+
&         N)*ins5(0)
       ENDDOALL
       s4t2 = s4t2+2
       IF (s4t2.GT.2*N) THEN
          s4t2 = -1
       ENDIF
       s4flag0 = 1
    ENDIF
    IF (S0t0.EQ.s4t1) THEN
       DOALL s4p0 = 1, -2+N-s4q0
          ins4(MOD(s4q0, 2),s4p0) = ins4(MOD(-1+s4q0, 2),s4p0)
&         +ins2(-1+s4p0,s4p0,-s4q0+N)*ins5(0)
       ENDDOALL
       s4t1 = s4t1+2
       IF (s4t1.GT.-6+4*N) THEN
          s4t1 = -1
       ENDIF
       s4flag0 = 1
    ENDIF
    IF (S0t0.EQ.s4t0) THEN
       DOALL s4p0 = 0, 0
          ins4(MOD(s4q0, 2),s4p0) = ins4(MOD(-1+s4q0, 2),s4p0)
&         +A(s4p0+1,-s4q0+N)*ins5(0)
       ENDDOALL
       s4t0 = s4t0+2
       IF (s4t0.GT.-4+4*N) THEN
          s4t0 = -1
       ENDIF
       s4flag0 = 1
    ENDIF
    IF (s4flag0.EQ.1) THEN
       s4q0 = s4q0+1
    ENDIF
    s4flag0 = 0
    IF (S0t0.EQ.s2t1) THEN
       DOALL s2p0 = 1+s2q0, -1+N
          DOALL s2p1 = 2+s2q0, N
             ins2(s2q0,s2p0,s2p1) = ins2(-1+s2q0,s2p0,s2p1)-
&            ins1(0,s2p0)*ins2(-1+s2q0,s2q0,s2p1)
          ENDDOALL
       ENDDOALL
       s2t1 = s2t1+2
       IF (s2t1.GT.-3+2*N) THEN
          s2t1 = -1
```

```
         ENDIF
         s2flag0 = 1
      ENDIF
      IF (S0t0.EQ.s2t0) THEN
         DOALL s2p0 = 1+s2q0, -1+N
            DOALL s2p1 = 2+s2q0, N
               ins2(s2q0,s2p0,s2p1) = A(1+s2p0,s2p1)-ins1(0,s2p0
     &         )*A(1+s2q0,s2p1)
            ENDDOALL
         ENDDOALL
         s2t0 = s2t0+2
         IF (s2t0.GT.1) THEN
            s2t0 = -1
         ENDIF
         s2flag0 = 1
      ENDIF
      IF (s2flag0.EQ.1) THEN
         s2q0 = s2q0+1
      ENDIF
      s2flag0 = 0
      IF (S0t0.EQ.s5t2) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins3(-1-s5q0+N))/A(-s5q0+N,-
     &   s5q0+N)
         s5t2 = s5t2+2
         IF (s5t2.GT.-1+2*N) THEN
            s5t2 = -1
         ENDIF
         s5flag0 = 1
      ENDIF
      IF (S0t0.EQ.s5t3) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins3(-1-s5q0+N))/ins2(-2-
     &   s5q0+N,-1-s5q0+N,-s5q0+N)
         s5t3 = s5t3+2
         IF (s5t3.GT.-1+2*N) THEN
            s5t3 = -1
         ENDIF
         s5flag0 = 1
      ENDIF
      IF (S0t0.EQ.s5t0) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins4(MOD(-1+s5q0, 2),-1-s5q0+
     &   N))/A(-s5q0+N,-s5q0+N)
         s5t0 = s5t0+2
         IF (s5t0.GT.-3+4*N) THEN
            s5t0 = -1
         ENDIF
         s5flag0 = 1
      ENDIF
      IF (S0t0.EQ.s5t1) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins4(MOD(-1+s5q0, 2),-1-s5q0+
     &   N))/ins2(-2-s5q0+N,-1-s5q0+N,-s5q0+N)
         s5t1 = s5t1+2
         IF (s5t1.GT.-5+4*N) THEN
            s5t1 = -1
         ENDIF
         s5flag0 = 1
```

```
         ENDIF
         IF (s5flag0.EQ.1) THEN
            s5q0 = s5q0+1
         ENDIF
         s5flag0 = 0
         IF (S0t0.EQ.s1t1) THEN
            DOALL s1p0 = 1+s1q0, -1+N
               ins1(0,s1p0) = ins2(-1+s1q0,s1p0,1+s1q0)/ins2(-1+
     &         s1q0,s1q0,1+s1q0)
            ENDDOALL
            s1t1 = s1t1+2
            IF (s1t1.GT.-4+2*N) THEN
               s1t1 = -1
            ENDIF
            s1flag0 = 1
         ENDIF
         IF (S0t0.EQ.s1t0) THEN
            DOALL s1p0 = 1+s1q0, -1+N
               ins1(0,s1p0) = A(1+s1p0,1+s1q0)/A(1+s1q0,1+s1q0)
            ENDDOALL
            s1t0 = s1t0+2
            IF (s1t0.GT.0) THEN
               s1t0 = -1
            ENDIF
            s1flag0 = 1
         ENDIF
         IF (s1flag0.EQ.1) THEN
            s1q0 = s1q0+1
         ENDIF
         s1flag0 = 0
         IF (S0t0.EQ.0) THEN
            DOALL s3p0 = 0, -1+N
               ins3(s3p0) = 0.
            ENDDOALL
         ENDIF
      ENDDO
      END
```

B.2 Programme gauss parallélisé pour la CM-5

```
      PROGRAM GAUSS
      INTEGER s3q0,s1flag0,s1t1,s5flag0,s5t3,s5t2,s5t1,s2flag0,s2t1,
     &s4flag0,s4t3,s4t2,s4t1,S0t0,s3t0,s3p1,s3p0,s1q0,s1t0,s1p1,s1p0,
     &s5q0,s5t0,s5p1,s5p0,s2q0,s2t0,s2p1,s2p0,s4q0,s4t0,s4p1,s4p0,N
      REAL*4 ins1(0:0,1:-1+N),ins5(0:0),ins2(0:-2+N,1:-1+N,2:N),
     &ins3(0:-1+N),ins4(0:1,0:-2+N),A(1:N,1:N)
CMF$  LAYOUT ins4(:SERIAL, :NEWS)
CMF$  LAYOUT ins5(:SERIAL)
CMF$  LAYOUT ins3(:NEWS)
CMF$  LAYOUT ins2(:SERIAL, :NEWS, :NEWS)
```

```
CMF$  LAYOUT ins1(:SERIAL, :NEWS)
      IF (3.LE.N) THEN
         s4t0 = 2+2*N
      ELSE
         s4t0 = -1
      ENDIF
      IF (4.LE.N) THEN
         s4t1 = 2+2*N
      ELSE
         s4t1 = -1
      ENDIF
      IF (2.LE.N) THEN
         s4t2 = 2*N
      ELSE
         s4t2 = -1
      ENDIF
      IF (3.LE.N) THEN
         s4t3 = 2*N
      ELSE
         s4t3 = -1
      ENDIF
      s4q0 = 0
      s4flag0 = 0
      IF (2.LE.N) THEN
         s2t0 = 1
      ELSE
         s2t0 = -1
      ENDIF
      IF (3.LE.N) THEN
         s2t1 = 3
      ELSE
         s2t1 = -1
      ENDIF
      s2q0 = 0
      s2flag0 = 0
      IF (2.LE.N) THEN
         s5t0 = -3+4*N
      ELSE
         s5t0 = -1
      ENDIF
      IF (3.LE.N) THEN
         s5t1 = 1+2*N
      ELSE
         s5t1 = -1
      ENDIF
      IF (1.LE.N.AND.N.LE.1) THEN
         s5t2 = -3+4*N
      ELSE
         s5t2 = -1
      ENDIF
      IF (2.LE.N) THEN
         s5t3 = -1+2*N
      ELSE
         s5t3 = -1
      ENDIF
```

```
      s5q0 = 0
      s5flag0 = 0
      IF (2.LE.N) THEN
         s1t0 = 0
      ELSE
         s1t0 = -1
      ENDIF
      IF (3.LE.N) THEN
         s1t1 = 2
      ELSE
         s1t1 = -1
      ENDIF
      s1q0 = 0
      s1flag0 = 0
      s3q0 = 0
      DO S0t0 = 0, MAX(-3+4*N, 1, 2*N)
         IF (S0t0.EQ.s4t3) THEN
            FORALL(s4p0 = 1:-2+N-s4q0) ins4(MOD(s4q0, 2),s4p0) =
     &      ins3(s4p0)+ins2(-1+s4p0,s4p0,-s4q0+N)*ins5(0)
            s4t3 = s4t3+2
            IF (s4t3.GT.2*N) THEN
               s4t3 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (S0t0.EQ.s4t2) THEN
            FORALL(s4p0 = 0:0) ins4(MOD(s4q0, 2),s4p0) = ins3(s4p0)
     &      +A(s4p0+1,-s4q0+N)*ins5(0)
            s4t2 = s4t2+2
            IF (s4t2.GT.2*N) THEN
               s4t2 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (S0t0.EQ.s4t1) THEN
            FORALL(s4p0 = 1:-2+N-s4q0) ins4(MOD(s4q0, 2),s4p0) =
     &      ins4(MOD(-1+s4q0, 2),s4p0)+ins2(-1+s4p0,s4p0,-s4q0+N)*
     &      ins5(0)
            s4t1 = s4t1+2
            IF (s4t1.GT.-6+4*N) THEN
               s4t1 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (S0t0.EQ.s4t0) THEN
            FORALL(s4p0 = 0:0) ins4(MOD(s4q0, 2),s4p0) = ins4(MOD(
     &      -1+s4q0, 2),s4p0)+A(s4p0+1,-s4q0+N)*ins5(0)
            s4t0 = s4t0+2
            IF (s4t0.GT.-4+4*N) THEN
               s4t0 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (s4flag0.EQ.1) THEN
            s4q0 = s4q0+1
```

```
      ENDIF
      s4flag0 = 0
      IF (S0t0.EQ.s2t1) THEN
         FORALL(s2p0 = 1+s2q0:-1+N, s2p1 = 2+s2q0:N) ins2(s2q0,
     &      s2p0,s2p1) = ins2(-1+s2q0,s2p0,s2p1)-ins1(0,s2p0)*ins2
     &      (-1+s2q0,s2q0,s2p1)
         s2t1 = s2t1+2
         IF (s2t1.GT.-3+2*N) THEN
            s2t1 = -1
         ENDIF
         s2flag0 = 1
      ENDIF
      IF (S0t0.EQ.s2t0) THEN
         FORALL(s2p0 = 1+s2q0:-1+N, s2p1 = 2+s2q0:N) ins2(s2q0,
     &      s2p0,s2p1) = A(1+s2p0,s2p1)-ins1(0,s2p0)*A(1+s2q0,s2p1)
         s2t0 = s2t0+2
         IF (s2t0.GT.1) THEN
            s2t0 = -1
         ENDIF
         s2flag0 = 1
      ENDIF
      IF (s2flag0.EQ.1) THEN
         s2q0 = s2q0+1
      ENDIF
      s2flag0 = 0
      IF (S0t0.EQ.s5t2) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins3(-1-s5q0+N))/A(-s5q0+N,-
     &      s5q0+N)
         s5t2 = s5t2+2
         IF (s5t2.GT.-1+2*N) THEN
            s5t2 = -1
         ENDIF
         s5flag0 = 1
      ENDIF
      IF (S0t0.EQ.s5t3) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins3(-1-s5q0+N))/ins2(-2-
     &      s5q0+N,-1-s5q0+N,-s5q0+N)
         s5t3 = s5t3+2
         IF (s5t3.GT.-1+2*N) THEN
            s5t3 = -1
         ENDIF
         s5flag0 = 1
      ENDIF
      IF (S0t0.EQ.s5t0) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins4(MOD(-1+s5q0, 2),-1-s5q0+
     &      N))/A(-s5q0+N,-s5q0+N)
         s5t0 = s5t0+2
         IF (s5t0.GT.-3+4*N) THEN
            s5t0 = -1
         ENDIF
         s5flag0 = 1
      ENDIF
      IF (S0t0.EQ.s5t1) THEN
         ins5(0) = (A(-s5q0+N,1+N)-ins4(MOD(-1+s5q0, 2),-1-s5q0+
     &      N))/ins2(-2-s5q0+N,-1-s5q0+N,-s5q0+N)
```

```
            s5t1 = s5t1+2
            IF (s5t1.GT.-5+4*N) THEN
               s5t1 = -1
            ENDIF
            s5flag0 = 1
         ENDIF
         IF (s5flag0.EQ.1) THEN
            s5q0 = s5q0+1
         ENDIF
         s5flag0 = 0
         IF (S0t0.EQ.s1t1) THEN
            FORALL(s1p0 = 1+s1q0:-1+N) ins1(0,s1p0) = ins2(-1+s1q0,
     &      s1p0,1+s1q0)/ins2(-1+s1q0,s1q0,1+s1q0)
            s1t1 = s1t1+2
            IF (s1t1.GT.-4+2*N) THEN
               s1t1 = -1
            ENDIF
            s1flag0 = 1
         ENDIF
         IF (S0t0.EQ.s1t0) THEN
            FORALL(s1p0 = 1+s1q0:-1+N) ins1(0,s1p0) = A(1+s1p0,1+
     &      s1q0)/A(1+s1q0,1+s1q0)
            s1t0 = s1t0+2
            IF (s1t0.GT.0) THEN
               s1t0 = -1
            ENDIF
            s1flag0 = 1
         ENDIF
         IF (s1flag0.EQ.1) THEN
            s1q0 = s1q0+1
         ENDIF
         s1flag0 = 0
         IF (S0t0.EQ.0) THEN
            FORALL(s3p0 = 0:-1+N) ins3(s3p0) = 0.
         ENDIF
       ENDDO
       END
```

B.3 Programme gauss parallélisé pour le T3D

```
       PROGRAM GAUSS
       INTEGER s3q0,s1flag0,s1t1,s5flag0,s5t3,s5t2,s5t1,s2flag0,s2t1,
      &s4flag0,s4t3,s4t2,s4t1,S0t0,s3t0,s3p1,s3p0,s1q0,s1t0,s1p1,s1p0,
      &s5q0,s5t0,s5p1,s5p0,s2q0,s2t0,s2p1,s2p0,s4q0,s4t0,s4p1,s4p0,N
       REAL*4 ins1(0:0,1:-1+N),ins5(0:0),ins2(0:-2+N,1:-1+N,2:N),
      &ins3(0:-1+N),ins4(0:1,0:-2+N),A(1:N,1:N)
CDIR$ SHARED ins4(:, :BLOCK)
CDIR$ SHARED ins5(:)
CDIR$ SHARED ins3(:BLOCK)
CDIR$ SHARED ins2(:, :BLOCK, :BLOCK)
CDIR$ SHARED ins1(:, :BLOCK)
       IF (3.LE.N) THEN
```

```
   s4t0 = 2+2*N
ELSE
   s4t0 = -1
ENDIF
IF (4.LE.N) THEN
   s4t1 = 2+2*N
ELSE
   s4t1 = -1
ENDIF
IF (2.LE.N) THEN
   s4t2 = 2*N
ELSE
   s4t2 = -1
ENDIF
IF (3.LE.N) THEN
   s4t3 = 2*N
ELSE
   s4t3 = -1
ENDIF
s4q0 = 0
s4flag0 = 0
IF (2.LE.N) THEN
   s2t0 = 1
ELSE
   s2t0 = -1
ENDIF
IF (3.LE.N) THEN
   s2t1 = 3
ELSE
   s2t1 = -1
ENDIF
s2q0 = 0
s2flag0 = 0
IF (2.LE.N) THEN
   s5t0 = -3+4*N
ELSE
   s5t0 = -1
ENDIF
IF (3.LE.N) THEN
   s5t1 = 1+2*N
ELSE
   s5t1 = -1
ENDIF
IF (1.LE.N.AND.N.LE.1) THEN
   s5t2 = -3+4*N
ELSE
   s5t2 = -1
ENDIF
IF (2.LE.N) THEN
   s5t3 = -1+2*N
ELSE
   s5t3 = -1
ENDIF
s5q0 = 0
s5flag0 = 0
```

```
      IF (2.LE.N) THEN
         s1t0 = 0
      ELSE
         s1t0 = -1
      ENDIF
      IF (3.LE.N) THEN
         s1t1 = 2
      ELSE
         s1t1 = -1
      ENDIF
      s1q0 = 0
      s1flag0 = 0
      s3q0 = 0
      DO S0t0 = 0, MAX(-3+4*N, 1, 2*N)
         IF (S0t0.EQ.s4t3) THEN
CDIR$ DOSHARED(s4P0) ON ins4(s4q0,s4p0)
            DO s4p0 = 1, -2+N-s4q0
               ins4(MOD(s4q0, 2),s4p0) = ins3(s4p0)+ins2(-1+s4p0,
     &         s4p0,-s4q0+N)*ins5(0)
            ENDDO
            s4t3 = s4t3+2
            IF (s4t3.GT.2*N) THEN
               s4t3 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (S0t0.EQ.s4t2) THEN
CDIR$ DOSHARED(s4P0) ON ins4(s4q0,s4p0)
            DO s4p0 = 0, 0
               ins4(MOD(s4q0, 2),s4p0) = ins3(s4p0)+A(s4p0+1,-s4q0+
     &         N)*ins5(0)
            ENDDO
            s4t2 = s4t2+2
            IF (s4t2.GT.2*N) THEN
               s4t2 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (S0t0.EQ.s4t1) THEN
CDIR$ DOSHARED(s4P0) ON ins4(s4q0,s4p0)
            DO s4p0 = 1, -2+N-s4q0
               ins4(MOD(s4q0, 2),s4p0) = ins4(MOD(-1+s4q0, 2),s4p0)
     &         +ins2(-1+s4p0,s4p0,-s4q0+N)*ins5(0)
            ENDDO
            s4t1 = s4t1+2
            IF (s4t1.GT.-6+4*N) THEN
               s4t1 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (S0t0.EQ.s4t0) THEN
CDIR$ DOSHARED(s4P0) ON ins4(s4q0,s4p0)
            DO s4p0 = 0, 0
               ins4(MOD(s4q0, 2),s4p0) = ins4(MOD(-1+s4q0, 2),s4p0)
     &         +A(s4p0+1,-s4q0+N)*ins5(0)
```

```
            ENDDO
            s4t0 = s4t0+2
            IF (s4t0.GT.-4+4*N) THEN
               s4t0 = -1
            ENDIF
            s4flag0 = 1
         ENDIF
         IF (s4flag0.EQ.1) THEN
            s4q0 = s4q0+1
         ENDIF
         s4flag0 = 0
         IF (S0t0.EQ.s2t1) THEN
CDIR$ DOSHARED(s2P0, s2P1) ON ins2(s2q0,s2p0,s2p1)
            DO s2p0 = 1+s2q0, -1+N
               DO s2p1 = 2+s2q0, N
                  ins2(s2q0,s2p0,s2p1) = ins2(-1+s2q0,s2p0,s2p1)-
     &            ins1(0,s2p0)*ins2(-1+s2q0,s2q0,s2p1)
               ENDDO
            ENDDO
            s2t1 = s2t1+2
            IF (s2t1.GT.-3+2*N) THEN
               s2t1 = -1
            ENDIF
            s2flag0 = 1
         ENDIF
         IF (S0t0.EQ.s2t0) THEN
CDIR$ DOSHARED(s2P0, s2P1) ON ins2(s2q0,s2p0,s2p1)
            DO s2p0 = 1+s2q0, -1+N
               DO s2p1 = 2+s2q0, N
                  ins2(s2q0,s2p0,s2p1) = A(1+s2p0,s2p1)-ins1(0,s2p0
     &            )*A(1+s2q0,s2p1)
               ENDDO
            ENDDO
            s2t0 = s2t0+2
            IF (s2t0.GT.1) THEN
               s2t0 = -1
            ENDIF
            s2flag0 = 1
         ENDIF
         IF (s2flag0.EQ.1) THEN
            s2q0 = s2q0+1
         ENDIF
         s2flag0 = 0
         IF (S0t0.EQ.s5t2) THEN
            ins5(0) = (A(-s5q0+N,1+N)-ins3(-1-s5q0+N))/A(-s5q0+N,-
     &      s5q0+N)
            s5t2 = s5t2+2
            IF (s5t2.GT.-1+2*N) THEN
               s5t2 = -1
            ENDIF
            s5flag0 = 1
         ENDIF
         IF (S0t0.EQ.s5t3) THEN
            ins5(0) = (A(-s5q0+N,1+N)-ins3(-1-s5q0+N))/ins2(-2-
     &      s5q0+N,-1-s5q0+N,-s5q0+N)
```

```
            s5t3 = s5t3+2
            IF (s5t3.GT.-1+2*N) THEN
               s5t3 = -1
            ENDIF
            s5flag0 = 1
         ENDIF
         IF (S0t0.EQ.s5t0) THEN
            ins5(0) = (A(-s5q0+N,1+N)-ins4(MOD(-1+s5q0, 2),-1-s5q0+
     &      N))/A(-s5q0+N,-s5q0+N)
            s5t0 = s5t0+2
            IF (s5t0.GT.-3+4*N) THEN
               s5t0 = -1
            ENDIF
            s5flag0 = 1
         ENDIF
         IF (S0t0.EQ.s5t1) THEN
            ins5(0) = (A(-s5q0+N,1+N)-ins4(MOD(-1+s5q0, 2),-1-s5q0+
     &      N))/ins2(-2-s5q0+N,-1-s5q0+N,-s5q0+N)
            s5t1 = s5t1+2
            IF (s5t1.GT.-5+4*N) THEN
               s5t1 = -1
            ENDIF
            s5flag0 = 1
         ENDIF
         IF (s5flag0.EQ.1) THEN
            s5q0 = s5q0+1
         ENDIF
         s5flag0 = 0
         IF (S0t0.EQ.s1t1) THEN
CDIR$ DOSHARED(s1P0) ON ins1(0,s1p0)
            DO s1p0 = 1+s1q0, -1+N
               ins1(0,s1p0) = ins2(-1+s1q0,s1p0,1+s1q0)/ins2(-1+
     &         s1q0,s1q0,1+s1q0)
            ENDDO
            s1t1 = s1t1+2
            IF (s1t1.GT.-4+2*N) THEN
               s1t1 = -1
            ENDIF
            s1flag0 = 1
         ENDIF
         IF (S0t0.EQ.s1t0) THEN
CDIR$ DOSHARED(s1P0) ON ins1(0,s1p0)
            DO s1p0 = 1+s1q0, -1+N
               ins1(0,s1p0) = A(1+s1p0,1+s1q0)/A(1+s1q0,1+s1q0)
            ENDDO
            s1t0 = s1t0+2
            IF (s1t0.GT.0) THEN
               s1t0 = -1
            ENDIF
            s1flag0 = 1
         ENDIF
         IF (s1flag0.EQ.1) THEN
            s1q0 = s1q0+1
         ENDIF
         s1flag0 = 0
```

```
         IF (S0t0.EQ.0) THEN
CDIR$ DOSHARED(s3P0) ON ins3(s3p0)
            DO s3p0 = 0, -1+N
               ins3(s3p0) = 0.
            ENDDO
         ENDIF
      ENDDO
      END
```

Annexe C

Codes divers

C.1 Programme `fmm`

```
        program fmm

        integer i, j, k, n
        real a(n,n), b(n,n), c(n,n), a'(n,n), b'(n,n)

        do i=1,n
          do j=1,n
s1          a(i,j) = a'(i,j)
s2          b(i,j) = b'(i,j)
          end do
        end do
        do i=1,n
          do j=1,n
s3          c(i,j) = 0.
            do k=1,n
s4            c(i,j) = c(i,j) + a(i,k) * b(k,j)
            end do
          end do
        end do
        end
```

C.2 Programme `lampif`

```
      program lampif

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

      do i = 1, l
        do j = 1, n-1
         do k = 1, n-1
           if (j.eq.1 .or. k.eq.1) then
             a(j,k) = 0.0
           else
             a(j,k) = 0.25*(a(j,k-1)+a(j,k+1)+a(j-1,k)+a(j+1,k))
           end if
         end do
       end do
      end do
      end
```

C.3 Programme choles

```
        program choles

        integer i, j, k, n
        real x, a(n,n), p(n)

        do i=1,n
1         x = a(i,i)
          do k = 1, i-1
2           x = x - a(i,k)**2
          end do
3         p(i) = 1.0/sqrt(x)
          do j = i+1, n
4           x = a(i,j)
            do k=1,i-1
5             x = x - a(j,k) * a(i,k)
            end do
6           a(j,i) = x * p(i)
          end do
        end do
        end
```

C.4 Programme gauss

```
      program gauss

      integer i, j, k, n
      real a(n,n), x(n), s, f

      do i = 1,n-1
        do j = i+1,n
s1         f = a(j,i)/a(i,i)
           do k = i+1,n
s2            a(j,k)=a(j,k) - f*a(i,k)
           end do
        end do
      end do
      do i = 1,n
s3       s = 0.
         do j = 1,i-1
s4          s = s + a(n-i+1,n-j+1)*x(n-j+1)
         end do
s5       x(n-i+1) = (a(n-i+1, n+1) - s)
     &         /a(n-i+1, n-i+1)
      end do
      end
```

C.5 Programme seidel

```
      program seidel

      integer count, iter, i, n, j
      real a(n,n), b(n), x(n)

      do count = 1,iter
         do i = 1,n
            s = b(i)
            do j = 1,i-1
               s = s-a(i,j)*x(j)
            end do
            do j = i+1,n
               s = s-a(i,j)*x(j)
            end do
            x(i) = s/a(i,i)
```

```
   end do
end do
end
```

C.6 Programme `lczos`

```
program lczos

integer n, m
real alpha(n), a(n,n), beta(n)
real y(n,m), x(n)
real Norm1, Norm2(n), gamma(n,m), yp(n,m)

do i = 1, n
   alpha(i) = 0.0
   beta(i) = 0.0
end do
do i = 1, n
   y(i, 1) = 0.0
end do
Norm1 = 0.0
do i = 1, n
   Norm1 = Norm1 + x(i)*x(i)
end do
Norm1 = 2*Norm1
do i = 1, n
   y(i,1) = x(i)/Norm1
end do
do i = 1, n
   gamma(i,1) = 0.0
   do k = 1, n
      gamma(i,1) = gamma(i,1) + a(i,k)*y(k,1)
   end do
end do
do i = 1, n
   alpha(1) = alpha(1)+ gamma(i,1)*y(i,1)
end do
do i = 1, n
   yp(i,2) = gamma(i,1) - alpha(1)*y(i,1)
end do
Norm2(1) = 0.0
do i = 1, n
```

```
      Norm2(1) = Norm2(1) + yp(i,2)*yp(i,2)
   end do
   beta(1) = sqrt(Norm2(1))
   do i = 1, n
      y(i,2) = yp(i,2)/ beta(1)
   end do
   do j = 2, m-1
      do i = 1, n
         gamma(i,j) = 0.0
         do  k = 1,n
            gamma(i,j) = gamma(i,j) + a(i,k)*y(k,j)
         end do
      end do
      do i = 1, n
         alpha(j) = alpha(j)+ gamma(i,j)*y(i,j)
      end do
      do i = 1,n
         yp(i, j+1) = gamma(i,j)-alpha(j)*y(i,j)- beta(j-1)*y(i,j-1)
      end do
      Norm2(j) = 0.0
      do i = 1, n
         Norm2(j) = Norm2(j) + yp(i,j+1)*yp(i,j+1)
      end do
      beta(j) = sqrt(Norm2(j))
      do i = 1, n
         y(i,j+1) = yp(i,j+1)/ beta(j)
      end do
   end do
   end
```

C.7 Programme burg2

```
  program burg2

  integer i, j, m, max
  real e(m), b(m), c(m)
  real a(m), a1(m)
  real s1, s2, temp

  do j=2,max
    s1=0.0
    s2=0.0
```

```
          do i=j+1,m
            s1=s1+e(i)*b(i-j)
            s2=s2+e(i)**2+b(i-j)**2
          end do
          c(j)=-2.0*s1/s2
          do i=1,j-1
            a1(i)=a(i)+c(j)*a(j-i)
          end do
          do i=1,j-1
            a(i)=a1(i)
          end do
          a(j)=c(j)
          do i=j+1,m
            temp=e(i)+c(j)*b(i-j)
            b(i-j)=b(i-j)+c(j)*e(i)
            e(i)=temp
          end do
        end do
        end
```

C.8 Programme thom

```
      program thom

      integer n, m
      real x(m,n),r(m,m),c(m,m),w(m,m),y(m,n)

c     .......... etape 1  ........
      do j=1,m
         do i=1,m
            s=0
            do k=1,n
               s=s+x(i,k)*x(j,k)
            end do
            r(i,j)=s/n
         end do
      end do

c     .......... etape 2  ........
      do j=1,m
         do i=1,m
            w(i,j)=c(i,j)
```

```
         end do
      end do
      do k=1,m
         do i=1,k-1
            aux=-r(i,k)/r(k,k)
            do j=k+1,m
               r(i,j)=r(i,j)+r(k,j)*aux
            end do
            do  jl=1,m
               c(i,jl)=c(i,jl)+c(k,jl)*aux
            end do
         end do
         do i=k+1,m
            aux=-r(i,k)/r(k,k)
            do j=k+1,m
               r(i,j)=r(i,j)+r(k,j)*aux
            end do
             do jl=1,m
               c(i,jl)=c(i,jl)+c(k,jl)*aux
            end do
         end do
      end do
      do j=1,m
         do i=1,m
            c(i,j)=c(i,j)/r(i,i)
         end do
      end do

c     .......... etape 3  ........
      do j=1,m
         s=0.0
         do i=1,m
            s=s+c(i,j)*w(i,j)
         end do
         do k=1,m
            w(k,j)=c(k,j)/s
         end do
      end do

c     .......... etape 4  ........
      do j=1,n
         do i=1,m
            s=0.0
            do k=1,m
```

```
            s=s+w(i,k)*x(k,j)
         end do
         y(i,j)=s
      end do
   end do
   end
```

Bibliographie

[AH91] S.G. Abraham and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *PPOPP'91*, 1991.

[AK87] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AL93] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, June 1993.

[Ban76] U. Banerjee. Data dependence in ordinary programs. M.S. Thesis, Rpt. UIUCDCS-76-837, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, 1976.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publisher, 1988.

[Ban91] U. Banerjee. Unimodular transformation of double loops. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compiler for Parallel Processing*, pages 192–219. MIT Press, Cambridge, MA, 1991.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users guide to PVM: Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[Ber66] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5), October 1966.

[Ber83] C. Berge. *Graphes*. Gauthier-Villars, 1983.

[BF95] M. Barreteau and P. Feautrier. Placement automatique de réductions. Exposé aux Journées SEH'95, à paraître, 1995.

[BL92] M. Barnett and C. Lengauer. Unimodularity considered not essential. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Parallel Processing CONPAR92-VAPP V, Lecture Notes in Computer Science 634*, pages 659–664. Elsevier, Amsterdam, 1992.

[Bor87] K.H. Borgwardt. *The Simplex Method: A Probalistic Analysis*. Springer-Verlag, Berlin, 1987.

[CF93] J.-F. Collard and P. Feautrier. Automatic generation of data parallel code. In *Fourth International Workshop on Compilers for Parallel Computers*, Delft University of Technology, The Netherlands, December 1993.

[CFR93] J.-F. Collard, P. Feautrier, and T. Risset. Construction of do loops from systems of affine constraints. Technical Report 93-15, LIP-IMAG, May 1993.

[CMP92] P. Clauss, C. Mongenet, and G.-R. Perrin. Synthesis of size-optimal toroïdal arrays for the algebraic path problem: A new contribution. *Parallel Computing, North-Holand*, 18:185–194, 1992.

[Col93] J.-F. Collard. Code generation in automatic parallelizers. Technical Report 93-21, LIP-IMAG, July 1993.

[CP94] P. Crooks and R.H. Perrott. An automatic data distribution generator for distributed memory MIMD machines. In *4th Int. Workshop on Compilers for Parallel Computers*, Delft University, The Netherlands, january 1994.

[DE73] G.B. Dantzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory*, A(14), 1973.

[DR93a] Alain Darte and Yves Robert. A graph-theoretic approach to the alignment problem. Technical Report 93-20, LIP-IMAG, July 1993.

[DR93b] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. Technical Report 93-03, LIP-IMAG, January 1993.

[Duf74] R.J. Duffin. On fourier's analysis of linear inequality systems. In *Mathematical Programming Study*, number 1, pages 71–95. North-Holland Publishing Company, 1974.

[Fah93] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Paralllel Computers*. PhD thesis, Institute for Statistics and Computer Science, University of Vienna, 1993.

[Fea88] P. Feautrier. Parametric Integer Programming. In *RAIRO Recherche Opérationnelle*, volume 22, pages 243–268, September 1988.

[Fea91] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part I : One-dimensional Time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part II : Multidimensional Time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[Fea93] P. Feautrier. Toward Automatic Partitioning of Arrays on Distributed Memory Computers. In *ACM ICS'93*, pages 175–184, Tokyo, July 1993.

[FWPS92] W. Ferng, K. Wu, S. Petiton, and Y. Saad. Basic Sparse Matrix Computations on Massively Parallel Computers. Preprint 92-084, Army High Performnace Computing Research Center, University of Minnesota, Minneapolis, 1992.

[GB91] M. Gupta and P. Banerjee. Compile-time estimation of communication costs in multicomputers. Technical Report UILU-ENG-91-2226 CRHC-91-16, Coordinate Science Laboratory, College of Engineering, University of Illinois at Urbana-Champaign, May 1991.

[GB92] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[GL83] G.H. Golub and C.F. van Loan. *Matrix Computations.* North Oxford Academic, 1983.

[Gra88] Gradel. Subclasses of presburger arithmetic and the polynomial-time hierarchy. *Theoretical Computer Science*, 56, 1988.

[Gre71] H. Greenberg. *Integer programming.* Academic Press, 1971.

[Hal79] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme.* Thèse de 3ème cycle, INP, Université de Grenoble, Grenoble, 1979.

[Has93] F. Hassaine. *Placement de données et Génération de Code Automatique pour les Multiprocesseurs à Mémoire Distribuée.* Thése de doctorat, Université Pierre et Marie Curie Paris VI, 1993.

[HJ88] R. Hockney and C. Jesshope. *Parallel Computers 2.* Adam Hilger, 1988.

[HPF94] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Version 1.1. Rice University, November 1994.

[IJ90] F. Irigoin and P. Jouvelot. Projet PIPS – Rapport de synthèse finale. Technical Report EMP-CAI-I E134, Centre de Recherche en Informatique de l'Ecole des Mines de Paris, December 1990.

[Iri87] F. Irigoin. *Partitionnement des Boucles Imbriquées – Une technique d'optimisation pour les programmes scientifiques.* Thèse de doctorat, Université Pierre et Marie Curie (Paris VI), 1987.

[IT88a] F. Irigoin and R. Triolet. Dependence approximation and global parallel code generation for nested loops. In *Int. Workshop Parallel and Distributed Algorithms*, Bonas, France, oct 1988.

[IT88b] F. Irigoin and R. Triolet. Supernode partitioning. In *Conf. Record of 15th ACN Symp. on Principles of Programming Languages*, 1988.

[JD89] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Procs. of the 3rd Int. Conf. on Supercomputing*, pages 186–194. ACM Press, 1989.

[JT89] P. Jouvelot and R. Triolet. Newgen: A Language-Independant Program Generator. Technical Report EMP-CAI-I A/191, Centre d'Automatique et d'Informatique de l'Ecole des Mines de Paris, Fontainebleau, July 1989.

[KKB92] K.G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines. In *International Conference on Supercomputing*, pages 82–92, July 1992.

[KKBP91] D. Kulkarni, K.G. Kumar, A. Basu, and A. Paulraj. Loop partitioning unimodular transformations for distributed memory multiprocessors. In *International Conference on Supercomputing*, pages 599–604, 1991.

[KLS90] K. Knobe, J.D. Lukas, and G.L. Steele, Jr. Data Optimization: Allocation of Arrays to Reduce Communications on SIMD-Machines. *The Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[Koe94] C. Koelbel. *The HPF Handbook*. The MIT Press, Cambridge, Massachusetts, 1994.

[Kuh80] R.H. Kuhn. Optimization and interconnection complexity for: Parallel processors, single-stage networks, and decision trees. Ph.D. Thesis, Rpt. UIUCDCS-R-80-1009, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, feb 1980.

[LAD+92] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings 1992 ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, 1992.

[Lam74] L. Lamport. The parallel execution of do loops. *Communication of the ACM*, 17(2), February 1974.

[Lam93] F. Lamour. Comment utiliser la bibliothèque d'algèbre linéaire, en nombres entiers, C3? De la théorie à la pratique. Note CEA 2740, CEA CEL-V, September 1993.

[LC90] J. Li and M. Chen. Index Domain Alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90, 3rd Symp. Frontiers Massively Parallel Computation*, College Park, MD, October 1990.

[LC91] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[Les93] A. Leservot. *Le Calcul de l'Array Data Flow Graph dans Pips, Partie I: Détection du code à contrôle statique*. Rapport de dea, systèmes informatiques, Université P. et M. Curie, August 1993.

[Mac87] M. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic Publishers, 1987.

[MAL93] D.E. Maydan, S.P. Amarasinghe, and M.S. Lam. Array data-flow analysis and its use in array privatization. In *ACM SIGACT-SIGPLAN Symposium on Principles of programming Languages (POPL'93)*, January 1993.

[MCP91] C. Mongenet, P. Clauss, and G.-R. Perrin. A geometrical coding to compile affine recurrence equations on regular arrays. In *5th Int. Parallel Processing Symposium (IPPS'91)*, pages 582–590, Los Angeles, 1991. IEEE.

[Mel94] A. Meltzer. Programming for Performance in CRAFT on the T3D. Technical report, Cray Research Inc., Eagan, Minnesota, July 1994.

[Min83] M. Minoux. *Programmation Linéaire. Théorie et Algorithmes*, volume 2. Dunod, Paris, 1983.

[MPM92] T. MacDonald, D.M. Pase, and A. Meltzer. Addressing in Cray Research's MPP Fortran. Technical report, Cray Research Inc., Eagan, Minnesota, July 1992. presented in the Third Workshop on Compilers for Parallel Computers.

[O'B93] M. O'Boyle. A data partitioning algorithm for distributed memory compilation. Technical Report UMCS-93-7-1, Department Of Computer Science, University of Manchester, 1993.

[Pla93] A. Platonoff. Quel Fortran pour la Programmation Massivement Parallèle? Note CEA 2713, CEA CEL-V, January 1993.

[PMM93] D. Pase, T. MacDonald, and A. Meltzer. MPP Fortran Programming Model. Technical report, Cray Research Inc., Eagan, Minnesota, November 1993.

[Pol88] C.D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[Pug90] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing'91*, pages 4–13, November 1990.

[Pug94] W. Pugh. Counting solutions to presburger formulas: How and why. Technical Report UMIACS-TR-94-27, Dept of Computer Science, University of Maryland, March 1994.

[PW86] D.A. Padua and M.J. Wolf. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12), 1986.

[PW92] W. Pugh and D. Wonnacott. Static analysis of upper bounds on parallelism. Technical Report UMIACS-TR-92-125, CS-TR-2994, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, nov 1992.

[Red91] X. Redon. Traitement des tests structuraux : le programme. Technical report, université Pierre et Marie-Curie. Paris 6, October 1991.

[Red92] X. Redon. Détection des réductions. Rapport MASI 92.52, Institut Blaise Pascal, September 1992.

[RF94] X. Redon and P. Feautrier. Scheduling reductions. In ACM Press, editor, *Procs of the 8th ACM International Conference on Supercomputing*, pages 117–125, July 1994.

[RS91] J. Ramanujam and P. Sadayappan. Compile-time techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–481, October 1991.

[RW92] Mourad Raji-Werth. *Construction systématique de programmes pour ordinateurs à mémoire distribuée.* Thèse de doctorat, Université P. et M. Curie, 1992.

[RWF90] M. Raji-Werth and P. Feautrier. A Systematic Approach to Program Transformations. In *International Workshop on Compilers for Parallel Computers*, pages 117–129, Paris, 1990.

[RWF91] M. Raji-Werth and P. Feautrier. Systematic Construction of Programs for Distributed Memory Computers. Rapport MASI 91.36, Institut Blaise Pascal, June 1991.

[Sch86] A. Schrijver. *Theory of linear and integer programming.* Wiley, New York, 1986.

[ST91] J.-P. Sheu and T.-H. Tai. Partitioning and mapping nested loops on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):430–439, October 1991.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), June 1972.

[Taw90] N. Tawbi. *Parallélisation Automatique : Estimation des Durées d'Exécution et Allocation Statique de Processeurs.* PhD thesis, Université de Paris VI, September 1990.

[TIF86] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. *Proceedings of the ACM SIGPLAN'86 Symp. on Compiler Construction*, 21(6), jun 1986.

[TMC90] Thinking Machines Corporation, Cambridge, Massachussetts. *Connection Machine Model CM-2 Technical Summary*, November 1990.

[TMC91] Thinking Machines Corporation, Cambridge, Massachussetts. *CM Fortran Programming Guide*, January 1991.

[TMC92] Thinking Machines Corporation, Cambridge, Massachussetts. *Connection Machine CM-5 Technical Summary*, January 1992.

[TMC94] Thinking Machines Corporation, Cambridge, Massachussetts. *CM-5 CM Fortran Performnace Guide*, January 1994.

[Tri84] R. Triolet. *Contribution à la Parallélisation Automatique de Programmes Fortran comportant des Appels de Procédures.* Thèse de Docteur-Ingénieur, Université P. et M. Curie, Paris, December 1984.

[WNC92] B.J.N. Wylie, M.G. Norman, and L.J. Clarke. High Performance Fortran: A Perspective. Technical Note EPCC-TN92-05.04, Edinburgh Parallel Computing Centre, University of Edinburgh, May 1992.

[Wol89] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[Xue94] J. Xue. Automating non-unimodular transformation for massive parallelism. *Parallel Computing*, 20:711–728, 1994.

[Yan93] Y.-Q. Yang. *Tests des Dépendances et Transformations de Programme*. PhD thesis, Université Pierre et Marie Curie Paris VI, 1993.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

[Zho94] L. Zhou. *Analyse Statique et Dynamique de la Complexité des Programmes Scientifiques*. PhD thesis, Université de Paris VI, September 1994.