No d'ordre

THESE de DOCTORAT de L'UNIVERSITÉ PARIS VI

Spécialité :

Systèmes Informatiques

présentée

par Madame *Martine ANCOURT*

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

Sujet de la Thèse :

Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales

soutenue le *18 Mars 1991*

devant le jury composé de:

Monsieur Chretienne	Président
Monsieur Feautrier	Examinateur
Monsieur Irigoin	Examinateur
Monsieur Jalby	Rapporteur
Monsieur Quinton	Rapporteur

ECOLE NATIONALE SUPERIEURE DES MINES DE PARIS

Résumé

Les paralléliseurs automatiques génèrent du code pour les multiprocesseurs à mémoire partagée sans tenir compte de leur hiérarchie mémoire. Le but de ce travail est d'ajouter à un paralléliseur une phase lui permettant de transformer un ensemble de tâches parallèles en un ensemble de tâches équivalentes utilisant les mémoires locales du multiprocesseur. Les algorithmes présentés permettent de générer automatiquement les codes de transfert des données entre deux niveaux de mémoire du multiprocesseur. Ces codes de transfert doivent copier l'ensemble des éléments de tableaux référencés au cours de l'exécution d'une tâche sur un processeur. Caractériser cet ensemble de données conduit à modéliser des ensembles de points entiers, non nécessairement convexes.

Pour les tranferts de la mémoire globale vers la mémoire locale, il est possible de copier plus d'éléments que ceux réellement utilisés par la tâche, et par exemple de transférer leur enveloppe convexe. Un compromis est alors fait entre l'espace alloué en mémoire locale, le volume des transferts et la complexité des bornes de boucle des codes de transfert.

Pour le code de recopie de la mémoire locale vers la mémoire globale il n'est pas possible de transférer plus d'éléments que ceux réellement modifiés par la tâche, car il faut conserver une mémoire globale cohérente. Chaque processeur doit recopier en mémoire globale uniquement les résultats qu'il a calculés. Les expressions des bornes de boucles du code de transfert contiendront des divisions entières pour traduire la non-convexité des domaines.

Pour des programmes réels, les codes de transfert générés par nos algorithmes sont toujours optimaux: chaque élément n'est transféré qu'une seule fois. Associés à une phase d'analyse des dépendances, ils peuvent aussi être utilisés pour générer les codes de transfert de données pour des multiprocesseurs à mémoire distribuée. Lorsque le partitionnement des données dans les mémoires locales est connu, ils permettent l'élimination du masque d'exécution utilisé pour spécifier l'ensemble des données locales à calculer sur un processeur.

MOTS CLES: PARALLELISATION, MULTIPROCESSEUR, HIERARCHIE MEMOIRE, COMPILATION, TRANSFERT DE DONNEES, POLYEDRES, EQUATIONS DIOPHANTIENNES.

Abstract

Parallel tasks generated by automatic parallelizers do not take advantage of supercomputer memory hierarchies. This thesis presents algorithms to transform a parallel task into an equivalent one that uses data with fast access memory. Algorithms to automatically generate code to move data between two different memory levels of (super)computer are presented. These copy codes should move back and forth array elements that are accessed when an elementary processor execute an array reference located in a set of loops. This set of array elements is characterized by a set of integer points in $\mathcal{Z}^p$ that is not necessarily a convex polyhedron.

In the case of data transfers from global memory to local memory, it is possible to copy a superset of accessed elements, for instance its convex hull. A trade-off has to be made between local memory space, transfer volume and loop bound complexity.

To copy data back from local memory to global memory is more difficult because global memory consistency must be preserved. Each processor (or processus) should only copy its own results to avoid errors and, secondarily, to decrease global memory traffic.

The input of our main algorithm is an integer convex polyhedron defining the computation space and an affine function representing the index expressions. Its output is set of nested loops containing a new array reference whose execution copies exactly accessed elements. Each element is copied only once. Loop bound expressions use integer divisions to generate non-convex sets.

For most practical programs this algorithm provides optimal code in number of data movements and control overhead. Associated with a dependence analysis phase, it can be used to generate data movements in distributed memory multiprocessors. When data partitionning in local memories is specified, it eliminates most of execution guards used to compute only local values on each processor.

KEY WORDS: PARALLELIZATION, MULTIPROCESSOR, MEMORY HIERARCHY, COMPILATION, DATA MOVEMENTS, POLYHEDRON, DIOPHANTINE EQUATIONS.

Je tiens à exprimer toute ma gratitude au Professeur Michel Lenci, Directeur du Centre de Recherche en Informatique de l'Ecole des Mines de Paris, qui m'a accueillie dans son équipe et permis de réaliser ce travail dans les meilleures conditions. Ses annotations au cours de la lecture de cette thèse ont été précieuses et m'ont permis de clarifier certains points importants.

Je remercie vivement le Professeur Chretienne qui s'est intéressé à mes travaux et a accepté de présider mon jury.

Je veux témoigner au Professeur Paul Feautrier toute ma reconnaissance pour m'avoir initiée à la recherche dans le domaine de la parallélisation automatique et avoir consacré une partie de son temps aux lectures de mes rédactions intermédiaires. Ses remarques pertinentes ont permis d'améliorer et d'appronfondir cette étude.

Je remercie tout particulièrement François Irigoin, Directeur adjoint du centre, qui a encadré cette thèse et a accepté de faire partie des membres du jury. Ses suggestions, ses critiques ainsi que les discussions toujours très constructives que nous avons eues ont grandement contribué à l'aboutissement de ce travail.

Je tiens à exprimer toute ma gratitude à Patrice Quinton pour son soutien à la direction du PRC C^3 dans le cadre du projet Parallélisation, de l'attention qu'il a toujours portée à mes travaux et d'avoir accepté d'être rapporteur de cette thèse.

Je tiens à remercier vivement le Professeur William Jalby, dont les travaux de grand intérêt sur les transferts de données inter-mémoires des multiprocesseurs possédant une hiérarchie mémoire m'ont permis de débuter rapidemment cette étude, et pour sa présence parmi les membres du jury en tant que rapporteur.

Je remercie également Pierre Jouvelot pour ses fructueuses remarques et sa disponibilité.

Mes derniers remerciements s'adressent à toutes les personnes qui ont contribué de près ou de loin au bon déroulement de ce travail, et en particulier à l'ensemble des membres de l'équipe du CRI de l'Ecole des Mines pour leurs encouragements et conseils, à J. Altimira qui a participé à la mise en page de ce document et à A. Pech qui m'a aidé lors des recherches bibliographiques.

Table des matières

" Toute chose contient un peu d'inconnu. Trouvons le."
Guy de Maupassant

Introduction

Les performances des multiprocesseurs utilisant des mémoires partagées sont souvent limitées par les temps d'accès à la mémoire. L'introduction des mémoires locales et des mémoires caches plus rapides a permis de réduire considérablement ces délais d'accès. Cette hiérarchie mémoire introduit cependant deux problèmes: la gestion du transfert des données inter-mémoires et le maintien de la cohérence des données, dont les différentes copies se trouvent en mémoire globale et en mémoire locale.

La gestion des mémoires caches est en général assurée par le matériel, certaines méthodes de programmation des mémoires locales s'en sont inspirées. Pour conserver la cohérence des données en mémoire globale, les données partagées par plusieurs tâches parallèles ne sont jamais transférées dans les mémoires privées des processeurs. Seules les données locales peuvent être transférées dans les mémoires privées. Les données peuvent être déclarées comme partagées soit par le programmeur, soit par le compilateur.

Les données de type tableaux, souvent partagées par plusieurs tâches parallèles, ne pourront donc que rarement être transférées en mémoires locales. Or les accès aux tableaux se trouvent généralement dans une structure de boucles. Leurs éléments sont très souvent référencés et il est dommage de ne pas les copier dans une mémoire à accès rapide.

Le but de cette étude est d'ajouter à un paralléliseur une phase lui permettant de mieux utiliser les ressources mémoire du multiprocesseur et de le rendre capable de transformer un ensemble de tâches parallèles synchronisées en un ensemble de tâches équivalentes utilisant les mémoires locales. Nous transformons le code des tâches en un code de sous-programme contenant des déclarations de variables locales, le code de transfert des données utilisées de la mémoire globale vers la mémoire locale du processeur où est exécutée la tâche, le code de calcul, et le code de transfert des résultats de la mémoire locale vers la mémoire globale.

Caractériser les données utilisées ou modifiées par une tâche soulève des problèmes essentiellement quand il s'agit de tableaux, car c'est l'ensemble des éléments du tableaux référencés par la tâche qu'il faut évaluer. Leurs éléments étant souvent utilisés, leur transfert en mémoire locale est important. C'est donc principalement vers la génération automatique du code de transfert de ces éléments que cette étude est orientée.

Caractériser l'ensemble des données référencées par un tableau T dans un corps de boucles conduit à modéliser des ensembles de points entiers. Nous avons choisi de caractériser ces éléments par des *polyèdres*. Cette particularité nous a permis d'étendre simplement nos premiers résultats obtenus pour une référence au tableau dans un corps de boucles au cas où nous avons plusieurs références au même tableau.

Pour simplifier, nous supposons qu'il est possible de distinguer les phases de calculs, qui s'effectuent uniquement avec la mémoire locale, et les phases de transferts entre la mémoire globale et les mémoires locales. Les tâches sont supposées ne pas comporter de synchronisations internes, car elles nécessiteraient la remise en cohérence de la mémoire globale au cours de la phase de calcul.

La gestion des mémoires caches est assurée en général par le matériel. Le premier chapitre de cette étude introduit les méthodes qui ont été proposées pour conserver des données cohérentes dans une mémoire globale associée à un ensemble de mémoires caches. Des exemples d'algorithmes de gestion de mémoire privée sont ensuite donnés, dans les multiprocesseurs actuels possédant des caches locaux (IBM 3090, ENCORE, Alliant) et/ou des mémoires locales (CEDAR, CRAY2, RP3).

Le deuxième chapitre décrit quelques langages "parallèles" et les techniques de vectorisation et parallélisation utilisées dans les vectoriseurs et paralléliseurs actuels. Notre méthode peut être appliquée à des tâches parallèles dont le parallélisme est implicite ou explicite, mais s'adapte mieux à un code de tâche parallèle généré par un paralléliseur. Dans ce dernier cas, la tâche parallèle satisfait naturellement les hypothèses nécessaires à l'utilisation de nos algorithmes.

Ces hypothèses et leur fréquence de validité dans les algorithmes numériques sont présentées dans la première partie du chapitre trois. La dernière partie expose les différents problèmes rencontrés pour générer automatiquement le code de transfert: l'évaluation des dimensions des tableaux locaux, la caractérisation de l'ensemble exact des éléments référencés par un tableau dans un corps de boucles, l'évaluation du volume des objets que l'on va transférer, la définition d'une fonction de coût des transferts en mémoire locale.

Nous définissons dans le chapitre quatre tous les outils nécessaires au développement de nos algorithmes. Les éléments référencés par un tableau dans un corps de boucles sont caractérisés par des polyèdres représentés par des systèmes linéaires en nombres entiers. Nous avons défini et développé un ensemble d'opérations de manipulation des systèmes linéaires nécessaires à la génération automatique du code de transfert.

Les trois chapitres suivants traitent l'ensemble des problèmes rencontrés pour générer ce code de transfert.

Le chapitre 5 traite successivement de l'optimisation de la base de parcours du transfert des éléments, de la génération des déclarations des tableaux locaux, et de la modification des références aux tableaux en des références aux tableaux locaux.

Le chapitre 6 présente les algorithmes que nous avons développés pour générer automatiquement le code de transfert des éléments référencés par une, puis deux, et enfin plusieurs fonctions d'accès aux éléments d'un tableau dans le corps de boucles.

Enfin, le chapitre 7 présente les extensions des hypothèses d'application de nos algorithmes, décrites au chapitre 3, et reprend la liste des principaux algorithmes développés.

Le dernier chapitre de cette étude situe notre approche parmi les autres méthodes développées pour transférer des données dans les différentes mémoires des multiprocesseurs possédant une mémoire partagée, à laquelle les processeurs ont accès via des mémoires privées, ou distribuée.

Chapitre 1

Les multiprocesseurs à mémoire hiérarchisée

Les performances des multiprocesseurs utilisant des mémoires partagées sont souvent limitées par les temps d'accès à la mémoire. Les processeurs accédent aux différents modules ou bancs mémoire par l'intermédiaire d'un bus ou d'un réseau d'interconnexion. Lorsque tous les accès passent par ce réseau, il est rapidement saturé et les performances commencent à se dégrader au delà de quatre processeurs. On note des conflits de bancs mémoire importants pour certaines applications sur le CRAY 2 [Call88], qui possède quatre processeurs et une large mémoire globale.

L'introduction des mémoires locales (programmables) et des mémoires caches ou antémémoires (non programmables) plus rapides a permis de réduire les délais d'accès à ces mémoires. Elles sont soit privées, accessibles uniquement par un processeur, (cache de l'IBM 3090, mémoires locales du CRAY2), soit partiellement partagées par un groupe de processeurs (cache du MULTIMAX, cache et mémoires locales des *clusters* du CEDAR), soit totalement partagées par l'ensemble des processeurs (cache de l'Alliant FX/8). Un projet de multiprocesseur RP3 a été conçu pour tester ces différentes architectures. Il permet de partitionner la mémoire pour qu'elle soit vue comme entièrement partagée, ou comme des modules totalement indépendants (les hypercubes), ou encore mixte (mémoire globale + mémoires locales).

Cependant l'introduction de ces mémoires à accès rapide soulève deux importants problèmes: celui des transferts des données entre les deux niveaux mémoire et celui du maintien de la cohérence des différentes copies à plusieurs endroits.

La première partie de ce chapitre introduit les méthodes, matérielles et logicielles, qui ont été développées pour accélérer les délais d'accès aux mémoires et les solutions proposées pour conserver des données cohérentes en mémoire globale.

La programmation des mémoires caches n'est en général pas assurée par le programmeur. Dans la seconde partie, nous présentons donc les méthodes utilisées pour résoudre les problèmes exposés dans les multiprocesseurs actuels possédant des caches locaux (IBM 3090, ENCORE, Alliant) et/ou des mémoires locales (CEDAR, CRAY2, RP3).

1.1 Problèmes architecturaux

Les mémoires locales et mémoires caches permettent de réduire considérablement les temps d'accès aux mémoires globales, mais introduisent d'autres difficultés:

- la gestion des transferts entre les deux niveaux de mémoire,
- le maintien de la cohérence des données dans ces deux niveaux.

Nous commencerons par présenter les méthodes qui ont été développées pour augmenter le débit mémoire et permettre l'accès en *parallèle* de données se trouvant dans la "même mémoire".

Puis nous détaillerons les différents problèmes liés à l'existence de deux niveaux de mémoire en donnant les solutions qui ont été proposées dans les multiprocesseurs actuels.

1.1.1 Débit mémoire - transferts contigus

Les délais d'accès à la mémoire sont les premiers coupables des baisses de performance enregistrées sur les programmes scientifiques s'exécutant sur les multiprocesseurs.

Pour améliorer les temps d'accès, les mémoires sont souvent découpées en *bancs*. Ils permettent l'accès en parallèle de données appartenant à la même mémoire. Parmi les techniques de distribution des données sur ces bancs, deux sont couramment utilisées: l'entrelacement et le non-entrelacement.

- Si les bancs sont non-entrelacés, les éléments sont rangés successivement sur un banc, puis sur le suivant, et ainsi de suite ...
- Si les bancs sont entrelacés (mémoire globale du CRAY 2, Alliant), deux données possédant des adresses consécutives se trouvent sur des bancs différents, tandis que deux données dont les adresses sont distantes d'un pas multiple du nombre de bancs sont sur le même.

Ces deux distributions améliorent les délais d'applications différentes.

La première est intéressante lorsque un processeur accède plus particulièrement aux données se trouvant sur un même banc mémoire.

La seconde permet d'accéder en parallèle aux adresses successives d'un même bloc mémoire. Elle est souvent utilisée car elle est basée sur le fait que l'une des premières optimisations des compilateurs et des programmeurs est de transformer des opérations scalaires en opérations vectorielles, possédant dans la majorité des cas un pas de "1". Cependant elle ne convient pas aux accès non contigus. Le cas le plus défavorable se produit lorsque les opérations exécutées sur un processeur sont effectuées sur des données dont les adresses sont distantes d'un pas multiple du nombre de bancs; les requêtes arrivent sans arrêt sur le même banc mémoire et augmentent à nouveau les délais d'accès à la mémoire.

Certains ordinateurs regroupent les deux techniques. C'est le cas du calculateur Univac 1108 dont les 4 bancs mémoire sont adressés de manière consécutive, chacun des bancs étant composé de 2 modules affectés respectivement aux adresses paires et impaires.

L'un des autres facteurs de chute des performances des multiprocesseurs est le volume des données locales nécessaires à l'exécution d'une tâche. Ces données sont transférées dans les mémoires locales le temps des calculs et recopiées en mémoire globale quand ils sont terminés. Lorsque le volume des données locales utiles aux calculs est supérieur à la taille de la mémoire locale, des algorithmes de remplacement des blocs mémoire doivent être utilisés. De nombreuses méthodes de remplacement des blocs mémoire existent: LRU, FIFO, LFU, RANDOM. Ces méthodes sont relativement simples, car elles doivent être exécutées par le matériel en moins d'un temps d'accès mémoire. Il est donc difficile de choisir une méthode universelle, l'architecture du multiprocesseur, l'emplacement des données dans les différents niveaux de mémoire et l'application elle-même devant entrer en ligne de compte dans ce choix.

C'est pour ces raisons que des techniques de découpage des tâches ont été étudiées [ASKL79], [MuNe80],[LiSt88] pour diminuer la taille des tâches parallèles de telle sorte que les données utilisées tiennent dans la mémoire privée du processeur.

Les communications extérieures étant aussi responsables du surplus de trafic, de nombreux multiprocesseurs disposent de plusieurs processeurs de contrôle qui gèrent les entrées-sorties et le système.

Les avantages et inconvénients de nombreux algorithmes développés pour gérer les caches sont présentés dans l'étude de Smith [Smit82].

1.1.2 Cohérence des données

Rappelons qu'une mémoire est cohérente si la valeur de chaque variable en lecture vaut la dernière valeur que l'on lui a affectée en écriture.

La cohérence des données entre la mémoire globale et la (les) mémoire(s) locale(s) doit être assurée par le programmeur. Pour les mémoires caches, deux politiques de mise à jour des données sont couramment utilisées:

- le mode **store-through** ou **write-through**: toute modification d'une des variables est immédiatement transmise à la fois à la mémoire globale et à la mémoire cache si la variable y est présente.

- le mode **store-in** ou **write-back**: les lectures et les écritures sont traitées lors des défauts de cache. Si une donnée est absente du cache, on transfère le bloc correspondant à la variable de la mémoire globale dans le cache, les dernières mises à jour étant faites à cette occasion.

Cependant ces deux modes de mise à jour des données ne suffisent pas à assurer la cohérence si les tâches, qui s'exécutent en parallèle, travaillent sur le même ensemble de données [CeFe78]. En effet, considérons deux tâches T1 et T2 s'exécutant en parallèle sur deux [1] processeurs différents P1 et P2 possédant des caches locaux C1 et C2 contenant chacun une copie d'une variable a. Si au cours du traitement, la tâche T1 modifie la variable a, la mise à jour de cette valeur ne sera pas reportée dans le cache C2, et toute lecture de a par la tâche T2 conduit à une incohérence.

[1] Les mêmes problèmes d'incohérence existent sur les monoprocesseurs multitâches possédant une mémoire locale, lorsque chaque tâche posséde un espace d'adressage des données séparé.

Plusieurs techniques (Snoopy cache [OwAg89], [ChVe89]) ont été proposées pour résoudre ces problèmes. La méthode la plus couramment utilisée en mode *store-through* (write-through) est la suivante (Snoopy cache): les adresses des blocs modifiés par un processeur sont communiquées à l'ensemble des processeurs qui, s'ils possèdent dans leur cache privé une partie du même bloc, marquent ce bloc comme modifié. Tout accès à l'un de ces blocs entraîne la recherche de la dernière copie du bloc (qui peut se trouver dans la mémoire centrale ou dans l'une des autres mémoires caches) et son transfert.

D'autres techniques plus sophistiquées ont été proposées dans [CeFe78],[BrDu83] .

Dans le mode *store-in* (aussi connu sous le nom de write-back), la responsabilité des problèmes de maintenance de la cohérence mémoire est souvent rejetée au niveau du compilateur. Il devra être capable de reconnaître les données qui sont strictement locales et celles qui sont partagées. Les données locales pourront être rangées dans les caches, tandis que les données partagées devront rester dans la mémoire globale. Cette méthode est aussi utilisée dans le mode *store-through.*

Le surplus de communications engendré par les deux méthodes est important. Dans le mode *store-through*, cela impose un trafic supplémentaire constant, tandis que le mode *store-in* impose un trafic plus ponctuel, mais plus lourd lors des mises à jour.

Le MULTIMAX d'ENCORE et le multiprocesseur du projet RP3 ont choisi le mode *store-through*, leur topologie leur permettant plus facilement d'espionner les échanges mémoires que celles de l'IBM 3090 ou du projet CEDAR.

1.2 Exemples de multiprocesseurs

Dans cette partie, nous décrivons des multiprocesseurs qui représentent quelques unes des topologies variées que l'on peut rencontrer dans les multiprocesseurs possédant deux niveaux de hiérarchie mémoire.

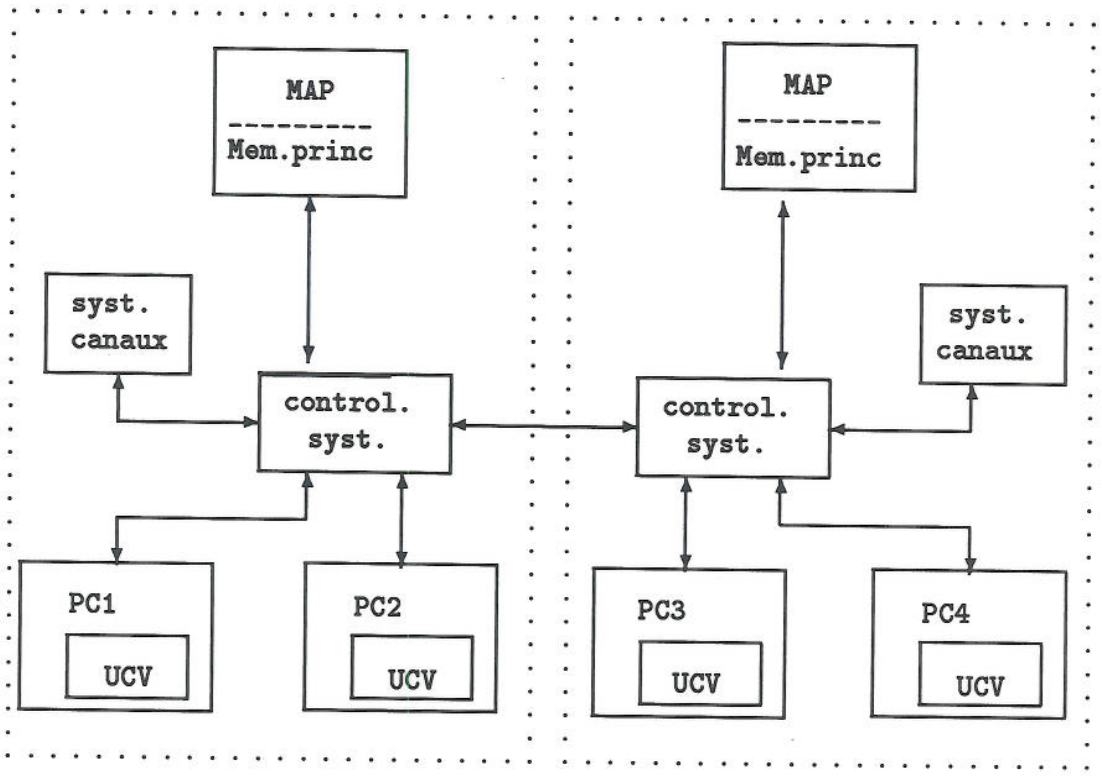
Nous nous sommes attachés essentiellement à l'aspect "hiérarchie mémoire" et les solutions qui ont été proposées pour résoudre les problèmes présentés précédemment.

1.2.1 IBM 3090

L'IBM 3090 [Tuck86] est équipé d'une ou deux unités centrales et de 0 à 6 *unités vectorielles.*

Chaque unité centrale (au nombre de 2 dans le modèle M400) se compose:

- de deux processeurs possédant chacun une **mémoire cache** (PC),
- d'une **mémoire principale** accessible par les processeurs,
- d'une mémoire d'arrière plan (MAP),
- d'un contrôleur système,
- et d'un système de canaux qui gère les entrées-sorties.



IBM 3090

La mémoire principale accessible par les quatres processeurs est de 128 M-octets.

Les unités de calcul vectoriel (UCV) améliorent les performances des calculs pouvant être vectorisés. Elles exécutent des opérations vectorielles arithmétiques et logiques sur des ensembles comportant jusqu'à 128 éléments (taille des registres vectoriels) avec une seule instruction. Elles sont organisées en pipe-line et peuvent fournir 1 résultat d'opération à chaque cycle. Les chargements et déchargements des registres vectoriels passent par le cache.

La mémoire cache de chaque processeur est de 64K-octets avec un temps d'accès de deux cycles (37 nanosecondes). La politique de mise à jour de cette mémoire est le **store-in** (write-back), pour laquelle chaque donnée modifiée est immédiatement mise à jour dans la mémoire cache, mais pas obligatoirement dans la mémoire centrale. Les transferts peuvent s'effectuer d'un cache C1 à un autre C2 sans que l'on soit obligé de passer par la mémoire centrale. Les temps de transfert sont ainsi réduits car il n'est pas nécessaire de recopier le bloc de C1 dans la mémoire globale avant de le copier dans l'antémémoire C2 qui en a fait la requête.

Dans l'IBM 3090, c'est le contrôleur système qui est chargé de maintenir la cohérence des données en mémoire. Il doit s'assurer qu'il n'existe toujours qu'une unique copie d'un bloc mémoire dans l'une des antémémoires, être capable de le localiser (soit dans la mémoire globale, soit dans l'un des caches privés des processeurs) et doit pouvoir le transférer au processeur qui a fait une requête en lecture ou en écriture à ce bloc.

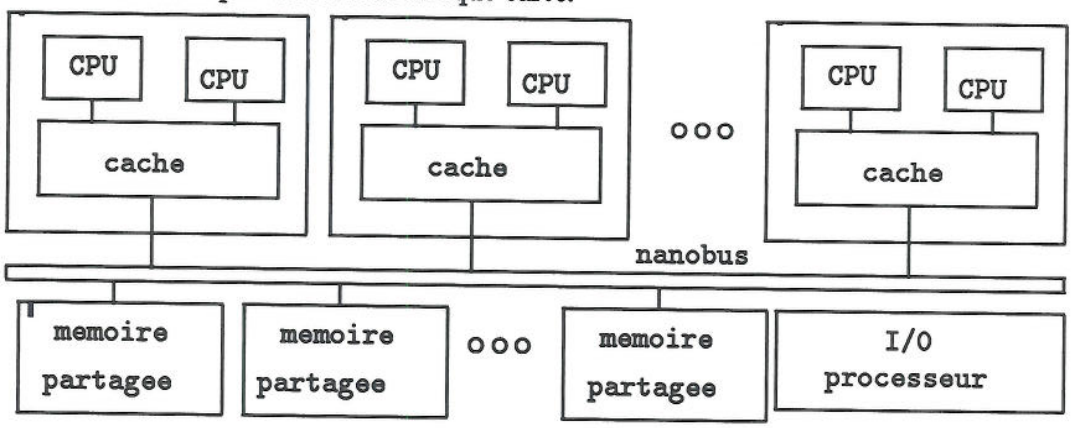
La mémoire d'arrière plan va de 128 à 256 M-octets. Cette dernière réduit la charge de

pagination des canaux et des unités périphériques. Les transferts entre cette mémoire et la mémoire principale sont gérés par le contrôleur système.

1.2.2 MULTIMAX d'ENCORE

Le MULTIMAX [Enc85] peut possèder jusqu'à 10 cartes, contenant chacune 2 processeurs, interconnectées par un *Nanobus* (100M-octets/s).

Les processeurs peuvent accéder aux données se trouvant dans une **mémoire globale** (4 à 32 M-octets) par l'intermédiaire du nanobus via une **mémoire cache** de 32 k-octets commune aux deux processeurs de chaque carte.



Encore Multimax

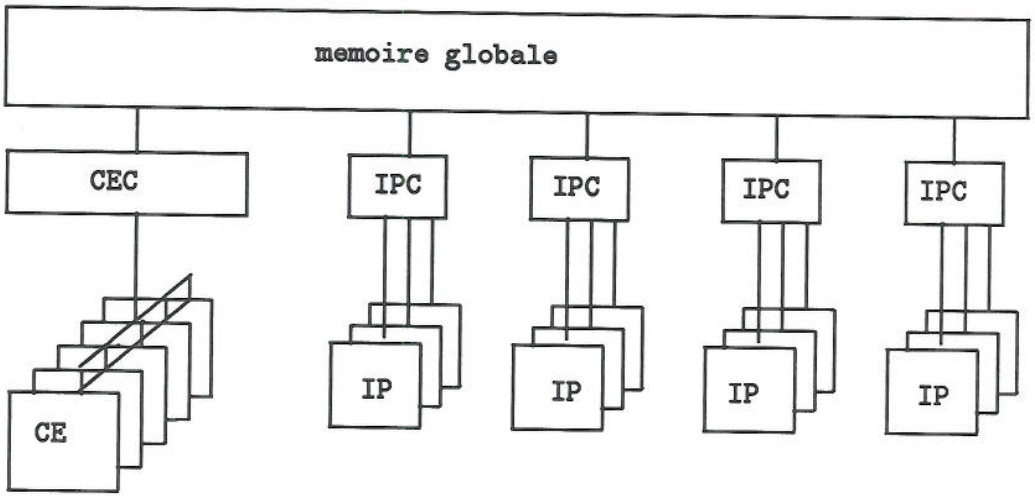
La politique de mise à jour des données est le **store-through** (write-through).

Pour conserver la cohérence des données, les caches sont tenus au courant des changements effectués en mémoire globale. Tous ces changements passent obligatoirement par le *nanobus*. Il est donc "espionné " en permanence, et lors d'accès en écriture à la mémoire faisant référence à des adresses locales, le bit de validité du bloc mémoire correspondant est modifié. Toute requête en lecture à ce bloc entraîne le transfert des données correctes se trouvant dans la mémoire globale vers la mémoire cache.

Un contrôleur d'entrées-sorties gère les communications extérieures (disques,..) pour diminuer le trafic interne.

1.2.3 Alliant FX/8

L'Alliant FX/8 [PeMu86],[Babb88] possède **huit processeurs** (Computational Element), qui accèdent à la **mémoire principale** (32 M-octets) via une **mémoire cache** (Computational Element Cache).

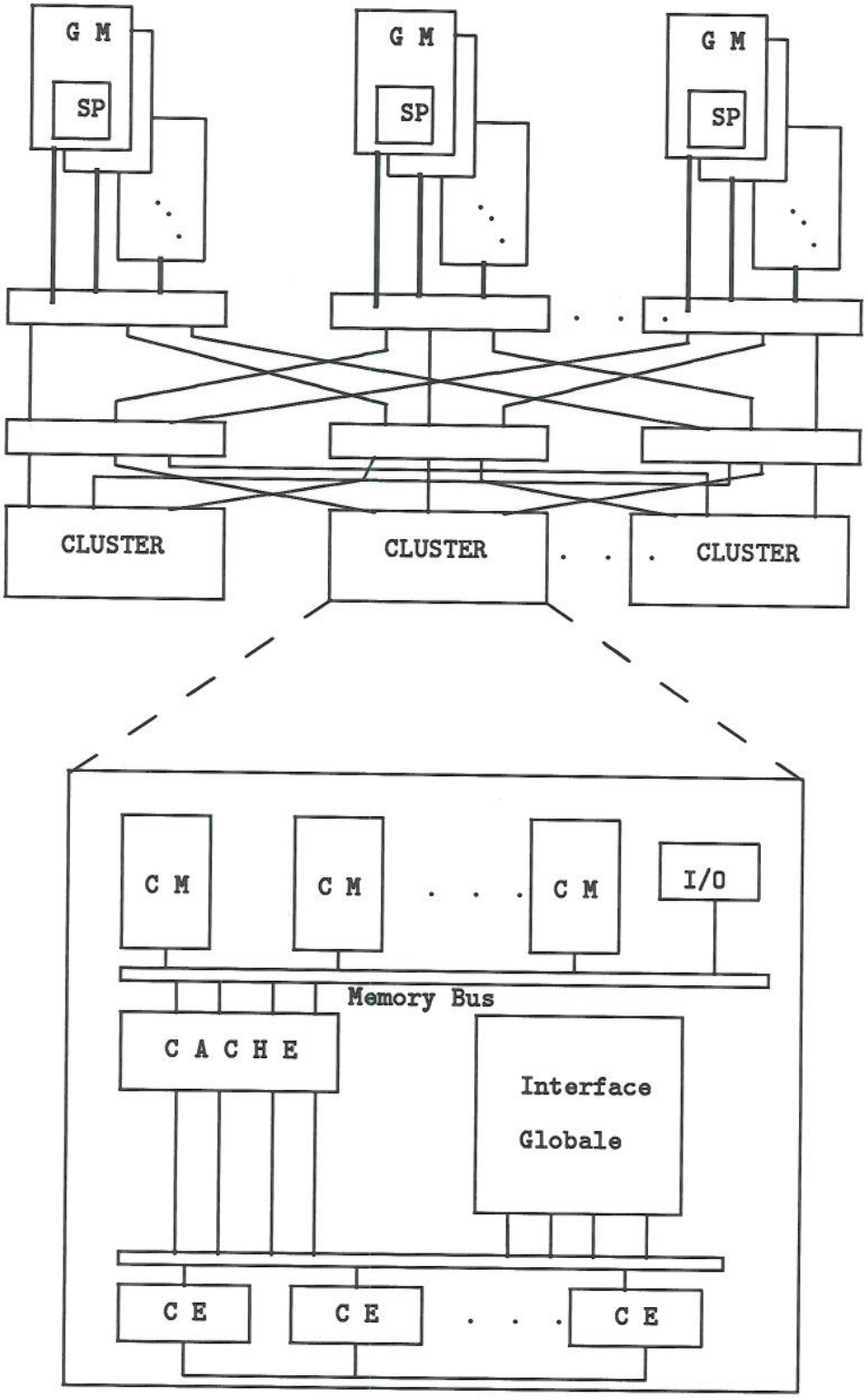


ALLIANT FX/8

Les processeurs (CE) peuvent directement communiquer par l'intermédiaire du bus, mais ne peuvent accéder à la mémoire globale ou aux processeurs de contrôle que par l'intermédiaire de la mémoire cache. Chaque processeur peut effectuer des calculs scalaires ou vectoriels. Les expériences effectuées sur l'Alliant FX/8 [ASMa86] ont montré que les performances augmentent en même tant que la taille des vecteurs utilisés augmente, jusqu'à un seuil relativement grand. Des speed-up supérieurs à 7 ont été calculés pour des vecteurs de taille 1000 environ. Le speed-up descend à 4 lorsque la taille des vecteurs est petite ($O(100)$) ou très grande ($O(10000)$) et jusqu'à 3, si toutes les références sont effectuées directement en mémoire partagée. Elles montrent bien l'importance de la génération d'opérations manipulant des vecteurs de taille moyenne, et du maintien de la localité des données, de telle sorte que toutes les données utiles puissent tenir dans la mémoire cache.

La mémoire cache fait au total 512K-octets et est divisée en quatre quadrants. Les transferts de données s'effectuant tous par l'intermédiaire de la mémoire cache, les problèmes de cohérence sont évités.

Les entrées-sorties et le contrôle du système sont effectués par les 4 unités "processeur-cache" se composant chacune de 1 à 3 processeurs (Interactive Processor) et d'une antémémoire (Interactive Processor Cache).


G M
SP
G M
SP
G M
SP
CLUSTER
CLUSTER
CLUSTER
C M
C M
C M
I/O
Memory Bus
C A C H E
Interface
Globale
C E
C E
C E

C E D A R

1.2.4 Projet CEDAR

Les processeurs du multiprocesseur du projet CEDAR [Yew86],[EmPY89] sont rassemblés en *clusters* et sont reliés à la mémoire partagée par un graphe complet.

Chaque *cluster* est une Alliant FX/8 un peu modifiée et contient:

- huit processeurs,
- une **mémoire cache partagée** par les huit processeurs (CE), de 128K-octets, et dont le mode de mise à jour est **store-in** (write-back).
- des **registres vectoriels et scalaires.**

Chaque *cluster* dispose d'une unité (logicielle) de préchargement des données en mémoire cache. Ces dernières sont transférées en mémoire cache avant que les processeurs en aient besoin, et les préchargements peuvent être effectués concurremment avec d'autres opérations. Ces unités ainsi que le réseau d'interconnexion global, qui permet des requêtes *pipelinées* entre chaque niveau de la hiérarchie mémoire, contribuent à diminuer les temps d'accès à la mémoire globale.

La technique utilisée pour traiter les problèmes d'incohérence mémoire est la suivante; les données *non-locales* (partagées par plusieurs *clusters* en lecture ou écriture) ne sont jamais transférées dans les mémoires cache des *clusters*. Seules les données *partiellement partagées* par un ensemble de processeurs appartenant au même *cluster* peuvent être copiées dans le cache correspondant,ainsi que toutes les données *locales*. Les données peuvent être déclarées comme partagées soit par le programmeur, soit par le compilateur.

Le délai d'accès à la mémoire centrale est non négligeable. Il faut donc, dans la mesure du possible, conserver une bonne localité des données et utiliser toutes les mémoires locales pour éviter les accès en mémoire globale.

Les informations nécessaires au compilateur FORTRAN du CEDAR sont exposées dans [Guzz87].

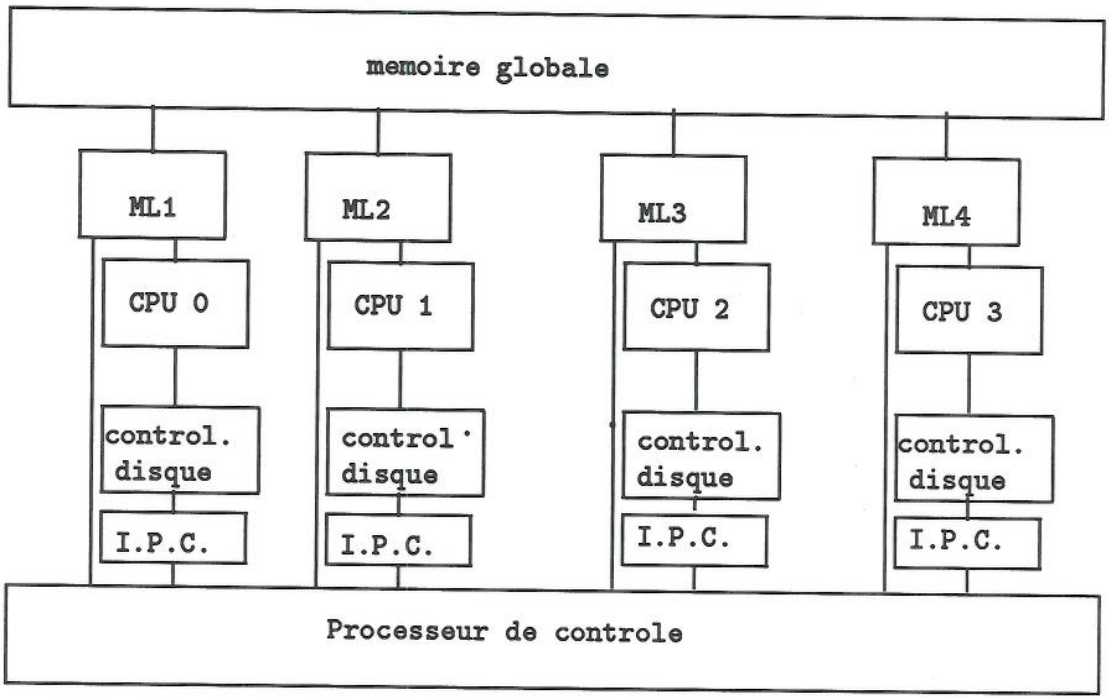
1.2.5 CRAY 2

Le CRAY 2 [CRAY 2],[Scho87],[HoJe81] dispose:

- de quatre processeurs indépendants pouvant effectuer des calculs vectoriels,
- d'une **mémoire principale** de 256M-mots,
- et d'un processeur de contrôle.

Chaque processeur possède:

- une **mémoire locale** de 16 M-mots,
- 8 **registres vectoriels** de 64 mots (organisés en quatre bancs de 16 mots) utilisés lors des calculs vectoriels,
- et 8 **registres scalaires** de 64 bits.



CRAY 2

La mémoire principale (256 Millions-mots de 64 bits) est divisée en 128 bancs regroupés pour former quatre quadrants de 32 bancs. Les bancs et les quadrants sont entrelacés, c'est à dire que deux éléments consécutifs se trouvent dans deux bancs différents, et deux bancs consécutifs de trouvent dans deux quadrants différents. En mode vectoriel, la mémoire peut délivrer jusqu'à un mot mémoire par période d'horloge (4 nanosecondes) soit jusqu'à 64 Gbits par seconde.

Parmi les sources de conflits en mémoire globale rencontrées sur le CRAY 2 [Scho87], les principales sont:

- des conflits dus à des requêtes au même quadrant (regroupement de 32 bancs mémoire non consécutifs) dans le cas où un processeur effectue des accès vectoriels avec un pas multiple de 4, par exemple.
- des conflits dus à des requêtes au même banc mémoire; dans le cas où le processeur fait des opérations sur des éléments avec un pas de 128.

La mémoire locale de chaque processeur est divisée en quatres bancs mémoire. Pour éviter les conflits, ces 4 bancs partagent un même registre d'écriture et un registre de lecture. Les données sont transférées de la mémoire globale dans la mémoire locale le temps des calculs et y retournent juste après. Les références aux tableaux sont faites directement en mémoire globale.

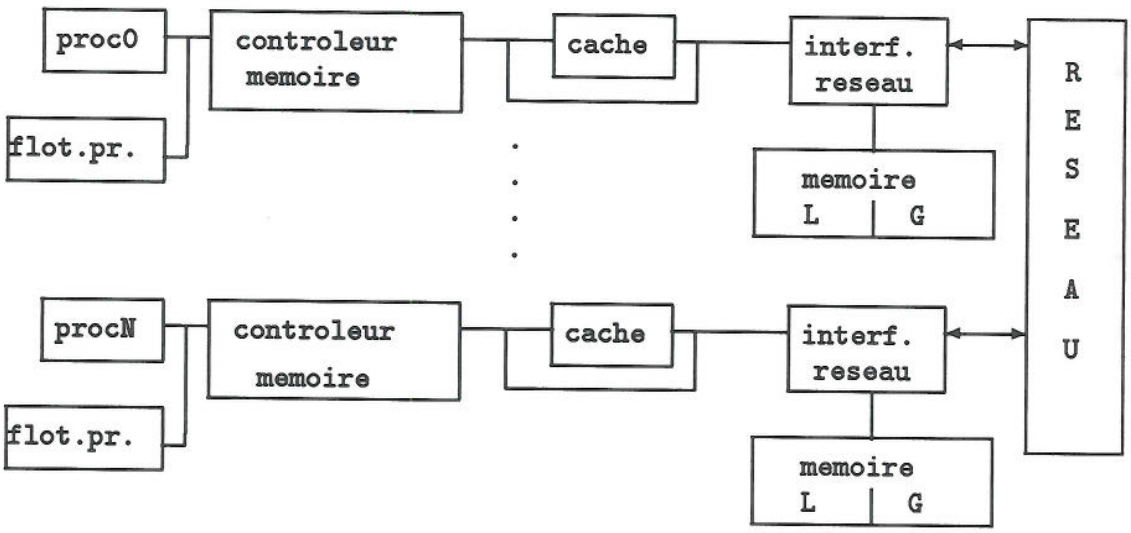
La mémoire principale est partagée à la fois par les quatre processeurs, le processeur de contrôle et les contrôleurs périphériques.

Le processeur de contrôle possède une mémoire locale de 4 M-mots. Il est chargé de superviser les opérations systèmes, de répondre aux requêtes des processeurs centraux, et de contrôler les entrées-sorties.

1.2.6 Le projet RP3

Le projet RP3 [PBGH85],[BMAW85] est un réseau de 512 unités "processeur-mémoire". Chaque unité dispose:

- d'un processeur,
- d'une unité vectorielle en calcul flottant (FP),
- d'une **mémoire** de 4M-octets,
- d'une **mémoire cache** de 32K-octets,
- d'une unité de contrôle mémoire,
- et d'interfaces pour gérer les entrées-sorties.



Projet RP3

Le projet RP3 a été conçu pour tester plusieurs architectures. Son organisation mémoire est souple. Le système peut être configuré de telle sorte que la mémoire soit vue comme soit entièrement partagée (comme le CRAY2), soit comme des mémoires locales indépendantes accessibles par les processeurs par "messages" (comme les hypercubes), soit mixte.

Tous les transferts passent par le réseau. Il n'y a donc pas de possibiblité de communication directe processeur-processeur.

La politique de mise à jour utilisée pour la mémoire cache est le **store-through** (write-through).

RP3 a traité les problèmes d'incohérence mémoire au niveau du compilateur. En s'appuyant sur le fait que tout compilateur qui veut faire des optimisations doit connaître les données qui sont partagées par le programme, la méthode utilisée est la suivante:

> Rappelons qu'une donnée est *non-cachable* (ne peut pas être transférée dans la mémoire cache) si elle peut être lue **et** écrite par plusieurs tâches parallèles et est *cachable* si elle ne peut qu'être lue par plusieurs tâches ou lue **et** écrite par une seule. Pour conserver la cohérence des données, le compilateur va marquer toutes les variables partagées comme *non-cachable*. Cela n'empêche pas que l'on puisse les transférer dans les caches à condition que le processeur possédant cette copie soit bien le seul à l'utiliser. Toutes les données *cachables* peuvent être transférées dans des mémoires caches sans créer de conflits mémoire.

Cette gestion des données cachables ou non cachables est effectuée au niveau de la page.

1.3 Conclusion

Pour conserver des temps d'accès à la mémoire globale acceptables, tous les multiprocesseurs possèdent une hiérarchie mémoire. Dans la majorité des cas, cette hiérarchie est à trois niveaux: une mémoire globale et des mémoires locales, des registres scalaires et vectoriels.

Les gains de performance enregistrés sur un multiprocesseur dépendent essentiellement des applications, de leur partitionnement en tâches parallèles, de la distribution des données en mémoire et, en général, de la vectorisation et de la parallélisation pouvant être effectuées sur ces applications.

Les problèmes d'incohérence mémoire introduits par la présence des antémémoires ont été résolus par des méthodes matérielles et logicielles. Les algorithmes de traitement des blocs mémoire manquants et la répartition des données dans les antémémoires sont préprogrammés. Il est donc souvent impossible de les influencer afin de diminuer les échanges inter-mémoires. Pourtant plusieurs améliorations seraient intéressantes; il faudrait avoir la possibilité:

- de préciser les données que l'on veut transférer dans les mémoires privées (comme dans le RP3),
- d'influencer l'algorithme de remplacement des blocs manquants, en renvoyant dans la mémoire globale un bloc plutôt qu'un autre,
- de ne transférer dans la mémoire globale que les données qui ont été modifiées par l'application et non l'ensemble des éléments qui appartiennent au bloc mémoire,[2]
- et surtout de transférer des éléments d'un tableau dans les mémoires caches. Dans la majorité des cas, pour éviter les problèmes d'incohérence, ces derniers sont maintenus en mémoire globale. Or les accès aux tableaux se trouvent généralement dans

[2] Certaines méthodes ont été proposées mais n'ont pas encore été implantées [Smit82]

une structure de boucles. Leurs éléments sont donc très souvent référencés et il est dommage de ne pas les copier dans une mémoire à accès rapide.

Les mémoires locales, quant à elles, ont l'avantage d'être programmables, et permettent aux utilisateurs d'améliorer ces techniques. En revanche, la cohérence des données et la gestion du transfert de ces données d'une mémoire à l'autre doivent être gérées par le programmeur.

Le but de cette étude est de fournir un algorithme qui génère automatiquement le code de transfert de données entre deux niveaux de hiérarchie mémoire programmables: une mémoire globale et plusieurs mémoires locales. Les programmes scientifiques faisant très souvent référence à des structures de tableaux, figurant dans des corps de boucles, le transfert de leurs éléments dans les mémoires locales à accès rapide est important; c'est donc dans ce sens que cette étude s'est orientée.

Chapitre 2

La parallélisation automatique

Un maximum de parallélisme doit être détecté dans les programmes pour utiliser tous les atouts des multiprocesseurs. Les opérations vectorielles et parallèles sont parfois explicitement mentionnées dans les programmes grâce à des instructions du langage de programmation (OCCAM) ou à des directives de compilation (Compilateur Fortran du CRAY2). Lorsque le parallélisme n'est pas explicite (ou pas totalement) ce sont les vectoriseurs et paralléliseurs qui sont chargés d'analyser les programmes et de le détecter.

Dans la première partie de ce chapitre, nous présentons quelques uns des langages de programmation "parallèles". Dans la deuxième, nous décrivons quelques techniques de vectorisation et parallélisation utilisées dans les vectoriseurs et paralléliseurs actuels, ainsi que les méthodes qui permettent de les améliorer.

Nos techniques peuvent être appliquées dans les deux cas, mais s'adaptent mieux à un code de tâche parallèle généré par un paralléliseur. Dans ce cas, la tâche parallèle satisfait en général les hypothèses nécessaires à l'utilisation de nos algorithmes.

2.1 Parallélisme explicite

Nous présentons ici trois langages de programmation qui caractérisent trois types de parallélisme explicite: le langage OCCAM (pour ses communications par envoi de messages), le FORTRAN 90 (parallélisme des données: tableaux), et le FORTRAN du Cray-2 (parallélisme MIMD).

2.1.1 OCCAM

Le langage Occam [Hoar87], [Inmo82], [Kerr87] est le fruit de la collaboration entre la société INMOS et le professeur Hoare de l'université d'Oxford. Il a été conçu pour faciliter la programmation des "transputers".

Le concept fondamental du langage est le processus (séquence d'opérations).

C'est un langage de programmation très simple qui ne contient que 22 mots clés, et qui permet l'exécution en parallèle de processus. Les communications inter-processus se font par **envoi de messages** sur des **canaux**.

Il y a, entre autres, des commandes

- d'affectation(":="),
- d'envoi de message sur un canal("!"),
- et de réception d'un message sur un canal("?");

ainsi que des structures de contrôle:

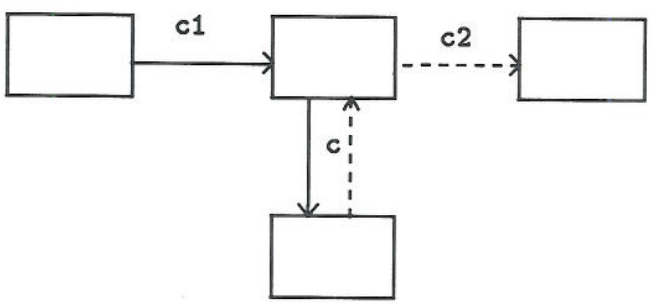
- séquentielles (SEQ),
- parallèles (PAR),
- et d'attente (WAIT).

Chaque processus peut envoyer ou recevoir un message par chacun des canaux le reliant à un autre processus.

La définition de canaux logiques permet de ne pas se soucier de la topologie physique du multiprocesseur.

C'est un outil simple pour la programmation parallèle, dont voici un exemple:

```
CHAN c:
PAR
  VAR x:
  SEQ
    c1 ?  x
    c !  x
  VAR x:
  SEQ
     c ?  x
    c2 !  x
```



Un premier processus reçoit la valeur d'une variable par le canal c1 et la renvoie par le canal c, tandis qu'en parallèle un deuxième reçoit la valeur d'une variable par le canal c et la renvoie au canal c2.

Le parallélisme est totalement explicite et notifié par la directive *PAR*.

2.1.2 FORTRAN 90

Le dernier projet de standardisation du FORTRAN proposé par L'*American National Standards Institute* et l'*International Standards Organization* est le FORTRAN 90 [Metc85], [Gent87], [MeRe89], [XAcm89].

Tout en restant compatible avec le FORTRAN 77, il introduit les concepts de paquetage à l'aide de modules, d'imbrication et de récursion des procédures. Mais l'aspect le plus populaire du FORTRAN 90 est sa partie traitement des "tableaux en parallèle". Elle permet d'optimiser et de simplifier les opérations sur les tableaux en étendant aux tableaux les opérations scalaires, les affectations et les fonctions scalaires.

Le FORTRAN 90 dispose de fonctions telles que: la somme de deux vecteurs SUM, le produit des éléments de deux vecteurs DOTPRODUCT, ainsi que de fonctions optimisées pour la manipulation des matrices (multiplication de deux matrices MATMUL, calcul de la transposée d'une matrice TRANSPOSE,..).

On pourra effectuer en mode vectoriel des calculs du type:

```
 ARRAY(20,10) ::  A,B,C
...
 C=3 * A *SQRT(B)
```

ou encore utiliser une condition au sein d'une affectation en employant l'instruction WHERE:

```
WHERE (B.GT.0.)
 C=A/B
 ELSEWHERE
 C=A
 END WHERE
```

Ici le parallélisme explicite est contenu dans les expressions du langage lui même.

2.1.3 FORTRAN du CRAY 2

Le FORTRAN disponible sur le CRAY2 propose une autre forme de parallélisme explicite: sous forme de directives de compilation, de vectorisation et multitasking, et de fonctions vectorielles appartenant à une bibliothèque de modules préprogrammés.

Le compilateur FORTRAN du Cray 2 (CFT2) [CRAY2] analyse les boucles DO les plus internes des programmes pour déterminer quelles sont les méthodes de vectorisation applicables pour améliorer l'efficacité du programme. Si des améliorations sont possibles, CFT2 produit une séquence de code contenant les instructions vectorielles qui permettent de guider les unités vectorielles, de calculs flottants, et les 8 registres vectoriels pour ces opérations. Ces opérations sont effectuées automatiquement sans que le programmeur ait besoin de le spécifier.

Cependant le programmeur peut aussi donner des ordres de vectorisation au compilateur à l'aide de directives, dont:

- VECTOR qui force la vectorisation des boucles les plus internes (le compilateur ne tiendra pas compte des conflits de dépendance éventuels, et vectorisera),
- NOVECTOR qui supprime toute vectorisation (utile par exemple lorsque la taille des vecteurs spécifiée par une constante symbolique est petite),
- VFUNCTION qui indique l'existence d'une fonction vectorielle programmée par l'utilisateur (qui peut être écrite en assembleur),
- REGFILE qui permet de transférer le contenu d'un bloc en mémoire locale (par exemple d'un tableau),...

Le programmeur peut aussi faire appel à des fonctions vectorielles préprogrammées telles que la multiplication de deux matrices, la somme d'un tableau et d'un scalaire: "S = S +A(i)",..., appartenant à la librairie.

Pour générer automatiquement le transfert des données utilisées par des tâches "explicitement parallèles", il faudra adapter nos algorithmes pour qu'ils utilisent les primitives de ces langages parallèles, telle que la directive "REGFILE" du Fortran CRAY2 qui permet le transfert d'un bloc mémoire en mémoire locale.

2.2 Parallélisme implicite

Les vectoriseurs et les paralléliseurs sont chargés de trouver les boucles du programme qui peuvent être vectorisées ou parallélisées. Les tests de dépendances proposés par Banerjee et Wolfe [Wolf82] sont utilisés dans la plupart de ces vectoriseurs et paralléliseurs pour construire le graphe des dépendances des instructions contenues dans le corps de boucles. Les méthodes d'Allen et Kennedy [AlKe87] déduisent de ce dernier les boucles qui seront effectivement vectorisées ou parallélisées.

Toutefois, la détection du parallélisme est totalement inhérente à la manière dont a été écrit le corps de boucles. Des techniques de transformation des nids de boucles et de partitionnement ont donc été proposées pour en améliorer la détection.

2.2.1 Vectorisation

Dans cette partie, nous commençons par présenter les conditions que doivent respecter les instructions *vectorisables*, et les méthodes nécessaires à leur évaluation. Puis nous exposons quelques transformations qui permettent d'augmenter le nombre de boucles vectorielles.

Boucles vectorielles

Rappelons, en simplifiant, qu'une instruction *S2* dépend de l'instruction *S1* (S1 et S2 adressent le même emplacement mémoire) si l'un des termes de *S2* a été créé par l'instruction *S1* au cours d'une itération précédente [Bern66].

Considérons le corps de boucle:

```
      DO I = N,M
S           TAB(f(I)) = Φ(TAB(g(I)))
      ENDDO
```

Par définition [Wolf88], l'instruction S dépend d'elle-même si et seulement si il existe des entiers i_1 et i_2 tels que $N \leq i_1 < i_2 \leq M$ et $f(i_1) = g(i_2)$. Toute instruction simple peut être *vectorisée*, si elle ne dépend pas d'elle-même.

Pour savoir si l'on peut vectoriser, il faut rechercher une solution entière au système constitué de cette série d'inéquations et d'équation.

Les algorithmes qui permettent de rechercher une solution entière dans un tel système (méthode des congruences décroissantes utilisant la méthode des coupes de Gomory, par exemple) sont plus coûteux que les algorithmes de recherche de solution rationnelle.

C'est pour cette raison que des tests moins *onéreux*, mais pas toujours exacts, sont aussi utilisés. Le test du PGCD permet de savoir si une équation admet une solution entière, il est souvent associé au test de Banerjee-Wolfe [Wolf82] qui teste l'existence d'une solution réelle dans le système d'inégalités défini par le domaine d'itération.

Lorsque l'instruction à vectoriser se trouve au milieu de plusieurs boucles, ces calculs de dépendance sont un peu plus compliqués (voir le calcul des dépendances présenté pour la parallélisation des boucles) et doivent prendre en compte la profondeur de ces dépendances (i.e. l'indice de boucle à partir duquel une instruction devient dépendante d'une autre).

Le vectoriseur VELOUR et les paralléliseurs PAF [FDT87] et PIPS [IJTr90] ont adopté un algorithme exact de recherche de solution entière leur permettant de mieux trouver l'ensemble des dépendances.

Les vectoriseurs VAST, et VATIL [LiTh85] utilisent les deux tests associés: PGCD et Banerjee.

Transformations de programme

Un grand nombre de transformations, applicables aux corps de boucles, a été proposé [AlKe87], [Wolf88] pour améliorer la détection des instructions vectorisables. Toutes ces transformations ne sont effectuées que si elles conservent les dépendances existantes dans le nid de boucles.

Le but de cette présentation n'est pas de donner une liste exhaustive des techniques employées pour augmenter la vectorisation des instructions, mais d'introduire les concepts qu'il faut respecter pour conserver de bonnes propriétés vectorielles, et, entre autre, dans le cadre de notre étude, de donner une idée des règles de réécriture des boucles, que nous allons essayer de respecter, pour conserver des possibilités de transferts vectoriels.

- Echange de boucles

L'échange de boucles fait partie des techniques couramment utilisées pour améliorer la vectorisation d'un corps de boucles.

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

Dans cet exemple, la boucle la plus interne n'est pas vectorisable puisque à chaque itération de la boucle d'indice J l'instruction S fait référence à un élément de tableau *A(I,J-1)* calculé à l'itération précédente. L'instruction S dépend d'elle-même.
En échangeant les deux boucles, la boucle d'indice I est vectorisable et l'instruction S peut s'exécuter en mode vectoriel.

```
DO   J = 2,M
(S)          A(1:N,J) = A(1:N,J-1) × C(1:N) + B(1:N,J)
ENDDO
```

- Vectorisation des tests IF

Les opérations dépendant d'une condition au sein d'une boucle ne sont en général pas vectorisables. Seule l'utilisation des masques vectoriels, quand ils existent, le permet. L'une des transformations qui peut être appliquée, sur le CRAY 2 par exemple, est la suivante:

```
DO   I = 1,N
  IF   A(I) ≥ 0    THEN
           B(I) = A(I) × A(I) + B(I)
ENDDO
```

qui donne:

```
C(1:N) = A(1:N) ≥ 0
WHERE  (C(1:N))
           B(1:N) = A(1:N) × A(1:N) + B(1:N)
```

- Incrément de boucle

Certains ordinateurs ne peuvent pas exécuter de manière vectorielle les boucles dont l'indice varie avec un pas différent de 1. Pour cette raison, lorsque l'incrément de

boucle est une constante symbolique, dont on n'a pas pu déterminer la valeur à la compilation, il est préférable de découper la boucle en deux parties de la manière suivante:

La boucle initiale:

```
DO  I = 1,N,INC
          A(I) = A(I) × C(I) + B(I)
ENDDO
```

Après transformation:

```
IF  (INC = 1)   THEN
   DO  I = 1,N
          A(I) = A(I) × C(I) + B(I)
   ENDDO
ELSE
   DO  I = 1,N,INC
          A(I) = A(I) × C(I) + B(I)
   ENDDO
```

Si INC vaut 1 la boucle s'exécutera en mode vectoriel.

- Réduction des Boucles.

  Pour certains ordinateurs pouvant exécuter des opérations sur des vecteurs très longs, il est intéressant de regrouper des boucles pour diminuer le nombre de démarrage des unités vectorielles, coûteux en temps.
  Le nid de boucles suivant en est un exemple:

```
REAL A(N,N),B(N,N),C(N,N)

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

  après transformation, on peut effectuer le calcul vectoriel:

$$A(1,1:N \times N) = B(1,1:N \times N) + C(1,1:N \times N)$$

Ces différentes transformations sont appliquées par la plupart des vectoriseurs. On voit que des boucles bien imbriquées, ne contenant pas si possible, de condition d'affection, avec un incrément de "un" facilitent grandement la génération de boucles vectorielles.

2.2.2 Parallélisation

Nous introduisons, dans cette partie, les conditions que doivent respecter les instructions d'une boucle pour qu'elle soit parallélisable. Leur nombre est nettement plus important que dans le cas de la vectorisation, car il faut conserver l'ensemble des dépendances entre toutes les instructions de la boucle, et non seulement les autodépendances.

Avec l'analyse des dépendances, l'analyse sémantique et les transformations de programmes constituent deux autres phases importantes des paralléliseurs. Elles permettent d'obtenir de précieux renseignements sur les variables des programmes, de détecter et d'améliorer la parallélisation. Cependant, la détection des boucles parallèles et leur parallélisation est totalement dépendante de la manière dont a été écrite l'application. Nous présentons dans la dernière partie de cette section des techniques de réordonnancement des itérations d'un corps de boucles qui permettent de détecter les instructions pouvant s'exécuter en parallèle après réordonnancement.

La présentation de ces différentes méthodes n'est pas exhaustive et se réfère à de nombreux articles. Le but ici n'est pas de rappeler l'ensemble des méthodes utilisées pour le calcul des dépendances, la parallélisation et le partitionnement automatique mais de les introduire afin de montrer que ces différentes méthodes génèrent des tâches parallèles dont le code vérifie les conditions d'application de nos algorithmes.

Quand peut-on paralléliser ?

Soit le nid de boucles:

```
DO   x_1 = min_1, max_1
 DO   x_2 = min_2, max_2
   ...
    DO    x_n = min_n, max_n

                  DO   x_{n1+1} = min_{n1+1}, max_{n1+1}
                       ...
                           DO    x_{n1+m1} = min_{n1+m1}, max_{n1+m1}
S1                                 TAB(f(x_1, x_2, ..., x_{n+m1})) = Φ()
                           ENDDO
                   ENDDO
                    DO    x_{n2+1} = min_{n2+1}, max_{n2+1}
                         ...
                             DO    x_{n2+m2} = min_{n2+m2}, max_{n2+m2}
S2                                 A = G(TAB(g(x_1, x_2, ..., x_{n+m2})))
                             ENDDO
                   ENDDO
   ENDDO
  ...
 ENDDO
ENDDO
```

où *S1* et *S2* ont les mêmes n premières boucles englobantes, et où la fonction Φ est une fonction quelconque.

Wolf [Wolf82] a montré que l'instruction *S2* **dépend de l'instruction** *S1*, si et seulement si il existe les entiers $(i_1, ...i_{k-1})$, $(j_{k+1}, ..., j_{n+m_1})$ et $(l_{k+1}, ..., l_{n+m_2})$ et deux entiers ξ_1 et ξ_2 tels que:

$$
\begin{array}{lll}
min_q \leq i_q \leq max_q & pour & 1 \leq q < k \\
min_q \leq j_q \leq max_q & pour & k < q \leq n \\
min_q \leq l_q \leq max_q & pour & k < q \leq n \\
min_{n_1+q} \leq j_q \leq max_{n_1+q} & pour & 1 \leq q \leq m_1 \\
min_{n_2+q} \leq l_q \leq max_{n_2+q} & pour & 1 \leq q \leq m_2 \\
min_k \leq \xi_1 < \xi_2 \leq max_k & &
\end{array}
$$

et $f(i_1, i_2, ..., i_{k-1}, \xi_1, j_{k+1}, ..., j_{n+m_1}) = g(i_1, i_2, ..., i_{k-1}, \xi_2, l_{k+1}, ..., l_{n+m_2})$

Une boucle est *parallélisable* s'il n'y pas de dépendances (pour cette boucle) entre les instructions qu'elle contient.

Il faut donc rechercher l'existence d'une solution entière au système constitué des inéquations et de l'équation donné ci-dessus. Pour la déterminer les paralléliseurs utilisent les mêmes tests que les vectoriseurs: tests du PGCD et de Banerjee ou tests exacts de Gondran [Gond73] ou de Feautrier [FeTa90].

Ensuite, l'ensemble des dépendances du corps de boucles sont regroupées dans un *graphe des dépendances* à partir duquel la méthode proposée dans [AlKe87] génère de nouvelles boucles parallèles. Ces méthodes sont utilisées dans la plupart des paralléliseurs (PARAFRASE [PGHL89], PTRAN [ABCC88], PIPS [IJTr90] , PAF [FDT87]).

Certaines transformations citées pour améliorer la vectorisation sont aussi utilisées pour la parallélisation des boucles. Cependant ce type de parallélisation est totalement dépendant de la manière dont a été écrit le corps de boucles.

Méthodes de partitionnement

La méthode de parallélisation de Kennedy et Allen ne recherche pas les itérations non dépendantes qui pourraient être exécutées en parallèle. Prenons l'exemple:

```
DO  100 I = 1, L
  DO  100  J = 2, M
    DO 100 K  =  2, N
      T(J,K)  =  T(J+1,K)  +  T(J,K+1)  +  T(J-1,K)  +  T(J,K-1)
100  CONTINUE
```

Après calcul, le graphe de dépendances de ce corps de boucles ne permet d'effectuer aucune parallélisation, et le nombre d'opérations séquentielles qui seront effectuées est: L*(M-1)*(N-1). Or l'ensemble des références au tableau T pour les itérations telles que

" $2I + J + K = constante$" sont indépendantes. Ces itérations pourraient donc s'exécuter en parallèle.

Plusieurs méthodes de réordonnancement des itérations d'un corps de boucles ont été développées, et entre autres: la méthode hyperplane proposée par L. Lamport [Lamp74], et les méthodes de partitionnement des boucles de F.Irigoin [Irig87] et de Peir et Cytron [PeCy89]. Elles réordonnent les itérations du corps de boucles, tout en conservant l'ensemble des dépendances existantes, pour améliorer la parallélisation.

Le but de la méthode hyperplane est de trouver un hyperplan dans lequel chaque itération peut être exécutée en parallèle. Elle permet d'obtenir, pour la boucle précédente, le résultat suivant [Lamp74]:

```
DO  100  I = 6,2*L+M+N
  DOCONC   100    pour tout (J,K) ϵ{(j,k):
        1 ≤ j ≤ L, 2 ≤ I-2j-k ≤ M et 2 ≤ k ≤ N}
    T(I-2×J-K,K) = T(I-2×J-K+1,K) + T(I-2×J-K,K+1) +
                 + T(I-2×J-K-1,K) + T(I-2×J-K,K-1)
100  CONTINUE
```

pour lequel le nombre d'exécutions séquentielles est réduit à: $2L + M + N - 5$

La méthode de partitionnement de F.Irigoin est basée sur les mêmes principes. Toutefois, elle utilise le *cône de dépendance* [IrTr87] et partitionne l'espace d'itération par des hyperplans pour fabriquer des blocs d'instructions parallèles, appelés *supernoeuds*, pratiquement tous identiques en volume. Elle permet ainsi d'obtenir une meilleure efficacité du taux d'occupation des processeurs. De plus, la taille des *supernoeuds* peut être ajustée, ce qui permet de limiter l'ensemble des itérations de telle sorte que l'ensemble des données, référencées par un supernoeud, tienne en mémoire locale.

La méthode de partitionnement de Peir et Cytron [PeCy89] utilise les vecteurs de dépendance et la "distance minimale" entre deux itérations dépendantes. Son but est d'obtenir un partitionnement maximal conservant pour des partitions distinctes des itérations indépendantes.

Les boucles parallèles générées par ces méthodes vérifient les conditions nécessaires [§[3.1.1]] à l'utilisation de nos algorithmes. En effet, les bornes définissant les espaces d'itération sont des fonctions linéaires des indices du corps de boucles, elles contiennent parfois des divisions entières [§Annexe] et/ou des fonctions MODULO [§[7.1.2]] que nous savons traiter. De plus, les conditions d'application de ces méthodes pour les fonctions d'accès aux éléments des tableaux sont les mêmes que les nôtres, elles doivent être affines.

Le code d'une tâche généré par l'une de ces méthodes vérifie donc toutes les hypothèses nécessaires à l'utilisation de nos algorithmes.

2.3 Conclusion

La parallélisation des instructions d'un programme n'est pas facile, et les informations données par le programmeur pour vectoriser ou paralléliser certaines parties d'un programme sont toujours très intéressantes. Qu'elles soient des instructions parallèles du langage de programmation ou des directives de compilation, elles permettent d'exécuter des instructions en parallèle qui n'auraient pas forcément été parallélisées par les vectoriseurs et paralléliseurs, faute d'informations suffisantes à la compilation.

Cependant, la parallélisation automatique des programmes simplifie énormément la tâche du programmeur, les calculs des dépendances d'un grand corps de boucles pouvant être très fastidieux, et donne un très bon pourcentage de boucles correctement parallélisées lorsque les instructions sont simples (affectations, références linéaires aux tableaux) et ont été astucieusement partitionnées.

L'utilisation de nos algorithmes dans le cadre de tâches **explicitement parallèles** demande une phase d'adaptation supplémentaire au langage parallèle employé. Nous ne la décrivons pas dans cette étude. Nous supposons donc que la tâche parallèle a été générée par un paralléliseur.

Si le paralléliseur utilise une méthode de réordonnancement ou de partitionnement des itérations du corps de boucles, les conditions d'application de nos algorithmes sont vérifiées, car ces méthodes génèrent du code pseudo-linéaire et leurs conditions d'application sont les mêmes que les nôtres (ou plus restrictives [§[7.2]]).

Si le paralléliseur détecte les boucles qui sont parallèles sans méthode de réordonnancement des itérations, les hypothèses sur les tâches sont moins strictes. Le code de la tâche parallèle peut alors contenir des tests, ou des appels de fonctions. Dans ces deux cas, une adaptation de nos algorithmes est nécessaire [§[7.4]]. Dans les autres cas, les conditions de linéarité des expressions nécessaires au calcul des dépendances sont suffisantes pour que nos algorithmes soient applicables.

Chapitre 3

Présentation du problème et propositions

Les paralléliseurs automatiques génèrent, en général, du code pour des multiprocesseurs à mémoire globale sans tenir compte de leur hiérarchie mémoire, formée de mémoires caches ou de mémoires locales à accès rapides et d'une mémoire globale plus lente.

Le but de ce travail est d'ajouter à un paralléliseur une phase lui permettant d'utiliser au mieux ces ressources mémoire et de le rendre capable de transformer un ensemble de tâches parallèles synchronisées en un ensemble de tâches équivalentes utilisant les mémoires locales.

Cette phase doit transformer le code des tâches en un code de sous-programme contenant: le code de transfert des données utilisées par la tâche de la mémoire globale vers la mémoire locale du processeur où elle est exécutée, le code de calcul, et le code de transfert des données modifiées par la tâche de la mémoire locale vers la mémoire globale.

Caractériser les données utilisées ou modifiées par une tâche soulève des difficultés essentiellement quand il s'agit de tableaux. Leurs éléments sont souvent utilisés dans le corps de boucles et leur transfert en mémoire locale est important. Nous nous sommes donc attaché à l'étude des caractéristiques des éléments référencés par un tableau dans un corps de boucles.

Les indices de tableau référencés dans un corps de boucles sont des composantes entières. L'ensemble des points que l'on doit caractériser est donc un **ensemble de points entiers**. C'est la cause de la majorité des problèmes que nous avons rencontrés.

La caractérisation d'un tel ensemble soulève des problèmes lors de l'évaluation des dimensions des tableaux locaux, du calcul des codes de transfert, du calcul du nombre des éléments référencés par le tableau ainsi que de l'efficacité du code généré.

Dans la première partie de ce chapitre, nous présentons les conditions d'application de nos algorithmes, ainsi que la fréquence de validité de ces conditions dans les algorithmes usuels.

Dans la seconde partie, nous détaillons les problèmes rencontrés et introduisons quelques solutions.

3.1 Hypothèses

Pour conserver de bonnes performances, de nombreuses méthodes de parallélisation s'attachent à garder une bonne localité des données utilisées. Les méthodes de partitionnement en tâches parallèles génèrent des tâches qui limitent le nombre de références afin qu'elles puissent tenir dans les mémoires locales. Nous supposons, dans cette étude, que la tâche vérifie ces conditions.

Pour simplifier, nous supposons aussi qu'il est possible de distinguer les phases de calculs, qui s'effectuent uniquement avec la mémoire locale, et les phases de transferts entre la mémoire globale et les mémoires locales. Les tâches sont supposées ne pas comporter de synchronisation interne.

Nous supposons de plus que l'on transfère en mémoire locale l'ensemble des éléments référencés par le tableau dans le corps de boucles et non seulement ceux qui seraient, par exemple, référencés au moins deux fois au cours de l'exécution (comme le proposent Jalby & co. dans [GJGa88b] et [EJWB90], voir §[8.2.2]).

Après avoir introduit les hypothèses générales sur les tâches, nous présentons maintenant les hypothèses concernant le code même de la tâche et leur fréquence de validité dans les algorithmes et méthodes *parallèles.*

3.1.1 Hypothèses de linéarité

Le traitement automatique des expressions contenues dans une application (bornes de boucle, fonction d'accès aux références d'un tableau,..) pour générer du code nécessite certaines hypothèses de linéarité. Le traitement d'informations non linéaires imposerait l'utilisation de méthodes de calcul trop particulières à chaque type d'information.

Dans un premier temps on se restreint donc au cas où:

- le domaine d'itération du corps de boucles est un polyèdre convexe borné [§4.1.1].

  exemple de domaine traité:

  ```
  DO  I = 1, N
    DO J = I + N, MIN(2 × I + 3, 3 × I − 5)
         ....
  ```

  exemple de domaine non traité:

  ```
  DO  I = 1, 10
    DO  J = I × N, I × I
         ....
  ```

- les fonctions d'accès aux références sont des fonctions affines de variables scalaires entières.

  exemples de cas traités: $T(3 \times I + 4 \times J), T(3 \times I, 5 \times J + N)$
  exemples de cas non traités: $T(3 \times I \times J), T(N \times J), T(S(2 \times I))$
  où I,J sont des indices de boucle, N une constante symbolique, et S un tableau

- le corps de boucles n'est formé que d'une suite d'affectations:

  Pas de tests, pas d'appels de fonction, et pas d'entrées/sorties.

Ces hypothèses vont nous permettre d'utiliser les *polyèdres convexes*[1] pour caractériser l'ensemble des domaines que l'on a besoin de manipuler: le domaine d'itération et le domaine image (l'ensemble des indices référencés par un tableau dans un corps de boucles).

La caractérisation de ce dernier domaine par un polyèdre convexe est importante car il est beaucoup plus simple de générer un corps de boucles définissant les éléments appartenant à un polyèdre que d'en générer un pour un ensemble quelconque de points entiers, et donc plus simple de générer son code de transfert.

3.1.2 Fréquence de validité

L'étude effectuée par Shen, Li et Yew [SLYe88] permet de constater que 53% des fonctions d'accès aux références d'un tableau sont totalement linéaires. C'est-à-dire qu'elles sont de la forme:

$T(Exp_1, Exp_2, ..., Exp_m)$ où les m expressions Exp_i sont de la forme:
$a_1.I_1 + a_2.I_2 + ... + a_n.I_n + b$ où:

les I_j sont des indices de boucle pour $1 \leq j \leq n$
les a_j sont des entiers pour $1 \leq j \leq n$
b est un terme constant
Les valeurs des coefficients des indices de boucle et des termes constants doivent être connues numériquement.

Les expressions non linéaires sont celles qui comportent dans la majorité des cas des constantes symboliques (96%) dont on ne connaît pas la valeur (ex: T(I +N)), ou des références à d'autres tableaux (4%) (ex: T(A(I))).

Certaines méthodes de calcul traitent les constantes (symboliques ou non) comme des variables particulières du programme. Elles permettent ainsi de considérer les expressions possédant une constante symbolique dans leur terme constant comme linéaire. Toutefois ces constantes symboliques ne doivent pas apparaître ailleurs que dans les termes constants des expressions, puisqu'elles ne doivent pas dépendre des itérations. L'utilisation de telles méthodes permet donc de traiter efficacement environ 90%[2] des références aux tableaux usuelles.

[1] La définition des *polyèdres convexes* est rappelée en 4.1
[2] pourcentage évalué d'après l'étude de Shen, Li et Yew [SLYe88]

3.1.3 Relation avec la parallélisation automatique

Les méthodes utilisées en parallélisation automatique imposent et utilisent les mêmes hypothèses de linéarité.

Les méthodes utilisées en analyse sémantique, telle que la propagation des constantes [Halb79], [CaKe86] et la détection des invariants de boucle, se basent sur les transformations linéaires appliquées aux variables dans le programme pour calculer et propager les informations qu'elles ont recueillies sur ces variables. Lorsque les transformations ne sont pas linéaires, elles ne peuvent pas les appliquer aux domaines qui caractérisent les variables et perdent donc toute information les concernant.

Les méthodes de calcul des dépendances entre les instructions d'un programme s'appuient aussi sur les mêmes hypothèses de linéarité. En effet, les algorithmes de résolution des systèmes en nombres entiers dépendent du type de la fonction pour laquelle on cherche une solution (linéaire, polynomiale,...). Les méthodes de calcul des dépendances devant rester générales pour des raisons de rapidité, d'efficacité et de décidabilité ne doivent pas dépendre du type de relation existant entre les instructions. Elles traitent donc uniquement les fonctions linéaires, soit environ 90% [3] des références usuelles.

Les méthodes de partitionnement des boucles [Irig87], [PeCy89] en tâches parallèles réordonnent les itérations, et restructurent l'application pour la découper en tâches parallèles. L'estimation de la taille de l'application et des dépendances entre instructions, utiles à un bon partitionnement, nécessite les mêmes hypothèses. De plus, le code généré par ces méthodes conserve toutes les propriétés linéaires de l'application puisque le partitionnement est souvent fait à base d'hyperplans.

Enfin, les méthodes d'analyse interprocédurale [Trio84], [CaKe86], [CaKe87], [BaKe89] utilisent les propriétés linéaires des transformations sur les variables pour calculer les effets de l'exécution d'une procédure sur ses variables.

3.2 Problèmes à résoudre

Le but de cette étude est de transformer le code des tâches en un code de sous-programme contenant: le code de transfert des données utilisées par la tâche de la mémoire globale vers la mémoire locale du processeur où elle est exécutée, le code de calcul, et le code de transfert des données modifiées par la tâche de la mémoire locale vers la mémoire globale.

La génération de ces deux codes de transfert doit si possible conserver des possibilités de transferts vectoriels avec accès contigus en mémoire.

Outre la caractérisation des ensembles d'indices de tableaux utilisés et modifiés par la tâche, cette phase de transformation nécessite l'évaluation des dimensions des tableaux locaux, le calcul du nombre des données référencées, et la définition d'une fonction permettant d'évaluer le coût des transferts en mémoire locale.

Nous présentons donc successivement ces phases d'élaboration du nouveau code. Nous détaillons les problèmes que nous avons recontrés lors des différentes étapes et introduisons les solutions, proposées dans les chapitres suivants.

[3] pourcentage évalué d'après l'étude de Shen, Li et Yew [SLYe88]

3.2.1 Déclarations

Le nombre d'éléments référencés par un tableau T dans un corps de boucles, qui appartient à une tâche parallèle, est souvent inférieur au nombre total d'éléments du tableau. L'espace occupé par le tableau temporaire servant à stocker ces données en mémoire locale doit le traduire.

La plage des valeurs référencées par le tableau dépend du domaine d'itération du corps de boucles. Il permet de calculer les bornes inférieure et supérieure des éléments du tableau qui sont utilisés dans le corps de boucles.

Le langage Fortran 77 ne permet pas d'allocation dynamique d'espace mémoire, et accepte comme type de déclaration des dimensions des tableaux, uniquement des entiers ou des constantes ou des expressions linéaires d'entiers et de constantes.

Pour générer les déclarations des tableaux locaux, il faut donc calculer le plus précisément possible les bornes inférieure et supérieure des éléments du tableau qui sont utilisés. Si ce calcul dépend d'une constante symbolique, parce que le domaine d'itération du corps de boucles contient une constante symbolique par exemple, nous utiliserons l'algorithme paramétrique de résolution de systèmes d'inéquations linéaires en nombres entiers proposé par Feautrier [Feau88b]. Dans les autres cas, nous utiliserons des algorithmes de projection.

Si r est la dimension de l'ensemble des éléments référencés par un tableau T, un tableau local de dimension r est nécessaire et suffisant pour stocker l'ensemble de ces éléments. Dans tous les cas où cette dimension est inférieure à celle du tableau T, nous choisirons une fonction d'accès aux données du tableau local différente de celle utilisée pour T, afin de minimiser l'espace alloué en mémoire locale.

Lorsque la norme des vecteurs directeurs du Z-module (base permettant de parcourir les éléments référencés) défini par la fonction d'accès aux éléments du tableau est supérieure à 1, on divise d'autant la dimension de la déclaration de tableau lui correspondant. On réduit encore ainsi l'espace mémoire alloué pour les tableaux temporaires.

3.2.2 Copies multiples

La solution la plus simple pour générer automatiquement le code de transfert consiste à utiliser comme code de transfert les boucles du programme ayant servi aux calculs des éléments du tableau [GJGa88]. On est ainsi certain de ne copier que des éléments utilisés par l'application. Cependant dans de nombreux cas, elle conduit à transférer plusieurs fois le même élément. Comme nous pouvons le voir dans l'exemple proposé également dans [GJGa88]:

```
DO   I = 1,10
  DO   J = 1,20
    DO   K = 1,30
      T(3 × I + K − 5,J + K) = ...
ENDDO
```

Exemple 3.2.2

pour lequel l'utilisation du code de calcul comme code de transfert conduit à transférer 6000 références au lieu de 1858 réellement calculées (figure 3.2.2).

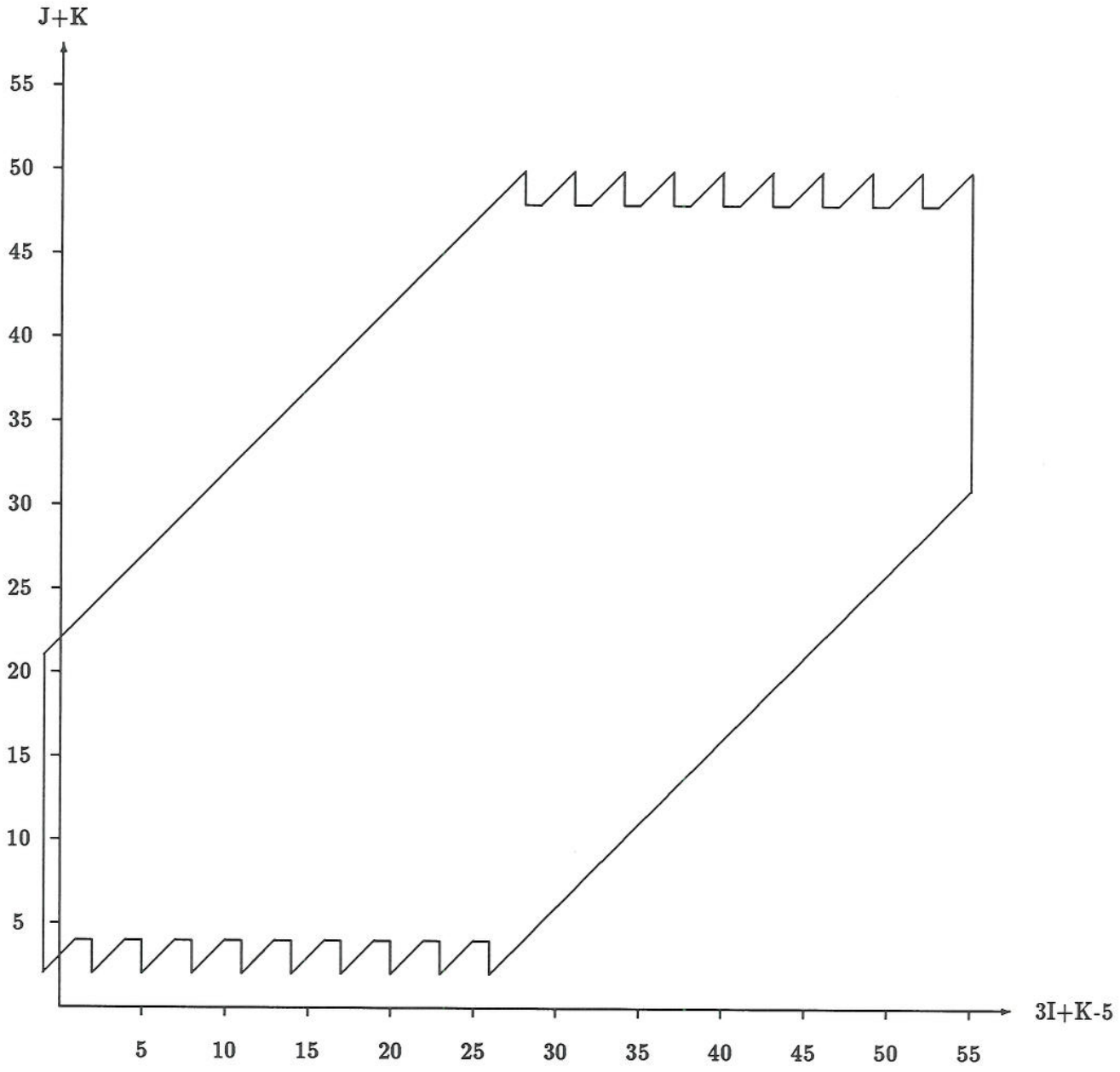


figure 3.2.2 Volume de stockage de l'exemple 3.2.2

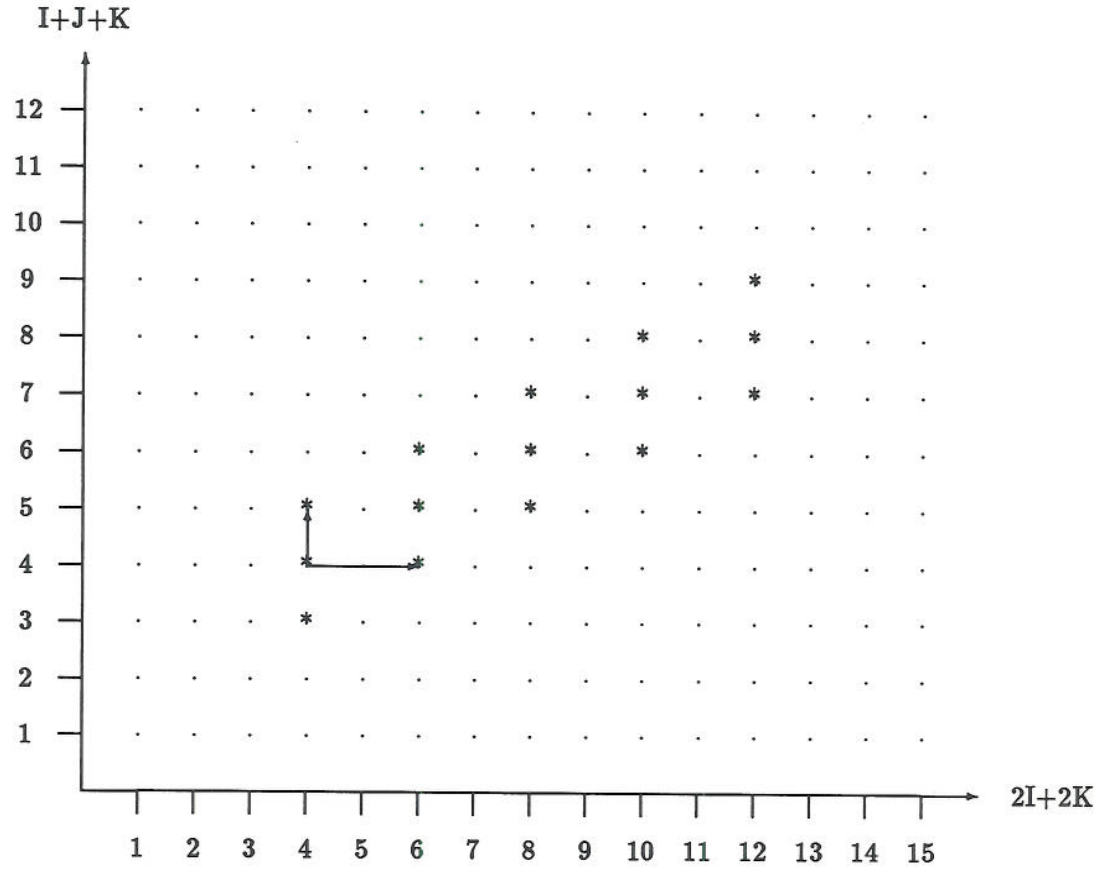


figure 3.2.2 bis - $DO\ I,J,K = 1,3 \qquad T(2 \times I + 2 \times K, I + J + K) = ... \qquad ENDDO$

Les temps de transfert entre mémoire globale et mémoire locale étant non négligeables, il faut minimiser le nombre des transferts. Utiliser le code de calcul comme code de transfert n'est donc pas une bonne solution.

Le problème de copies multiples intervient, en particulier, lorsque la dimension de la fonction d'accès aux références du tableau est inférieure à la dimension du domaine d'itération (voir figure 3.2.2 et figure 3.2.2 bis). Dans les deux exemples présentés, la fonction d'accès à une référence du tableau T est de dimension 2, alors que 3 boucles génèrent le code de calcul.

Pour minimiser le nombre de transferts, il faut donc trouver une base de parcours des éléments référencés par le tableau. Pour l'exemple 3.2.2 bis, nous pouvons choisir comme base de parcours: la base de vecteurs directeurs (2I+2K,0) et (0,I+J+K).

Notons que l'ordre des transferts est indépendant de la tâche, les données peuvent être transférées dans un ordre tout à fait différent de celui dans lequel elles sont utilisées par la tâche.

Une fois cette base choisie, il faut exprimer tous les éléments référencés par T dans cette nouvelle base. Ces éléments s'exprimaient en fonction de la base du domaine d'itération, il faut maintenant les exprimer en fonction de la base du domaine image. Cette opération

se traduit par un changement de base, et une projection sur la nouvelle base de dimension parfois inféreure à la première.

Cependant l'ensemble des indices référencés par un tableau dans un corps de boucles n'est pas nécessairement convexe (exemple 3.2.2). La génération d'un nid de boucles caractérisant les éléments appartenant à un polyèdre étant beaucoup plus simple que celle d'un ensemble quelconque de points entiers, nous allons approximer les ensembles non convexes par des ensembles convexes (dans tous les cas où c'est possible).

Pour les transferts de la mémoire globale vers la mémoire locale, il est possible de copier plus d'éléments de tableaux que ceux réellement utilisés par la tâche. En effet, il ne peut pas y avoir de conflits de dépendance entre deux ensembles de données référencées. Si l'ensemble des données utilisées par la tâche est non convexe, nous transférons alors le polyèdre convexe englobant le plus proche.

Pour le code de recopie de la mémoire locale vers la mémoire globale, il n'est pas possible de transférer des éléments de tableaux qui n'auraient pas été modifiés par la tâche. En effet, sinon, un processeur pourrait transférer en mémoire globale une donnée qu'il n'a pas lui même modifiée, mais qui l'a été par un autre processeur. La mémoire globale deviendrait alors incohérente. Si l'ensemble des éléments modifiés par la tâche est non convexe, on introduit des divisions entières dans les expressions des bornes de boucle du code de recopie. Elles traduiront la non-convexité au bord du domaine.

Ces solutions, présentées dans les chapitres suivants, ont l'avantage d'optimiser le volume de transfert, qui sera en général égal au volume de stockage. Mais elles compliquent parfois le code de transfert généré. Lorsque les conditions le permettent, certains compromis coût/efficacité seront faits.

3.2.3 Evaluation de volume

Le calcul automatique du nombre de points **entiers** contenus dans un ensemble est très complexe. Il est utilisé pour des problèmes d'évaluation exacte du volume des données utilisées par l'application, à stocker en mémoire locale, et du volume des données à transférer.

Prenons un exemple:

```
DO  I = 1,3
    DO  J = 1,3
        DO K = 1,3
            T(2 × I + 2 × J, K + I) = .....
```

les données référencées par la fonction $(I, J, K) \longrightarrow (2 \times I + 2 \times J, I + K)$ sont représentées sur la figure 3.2.3.

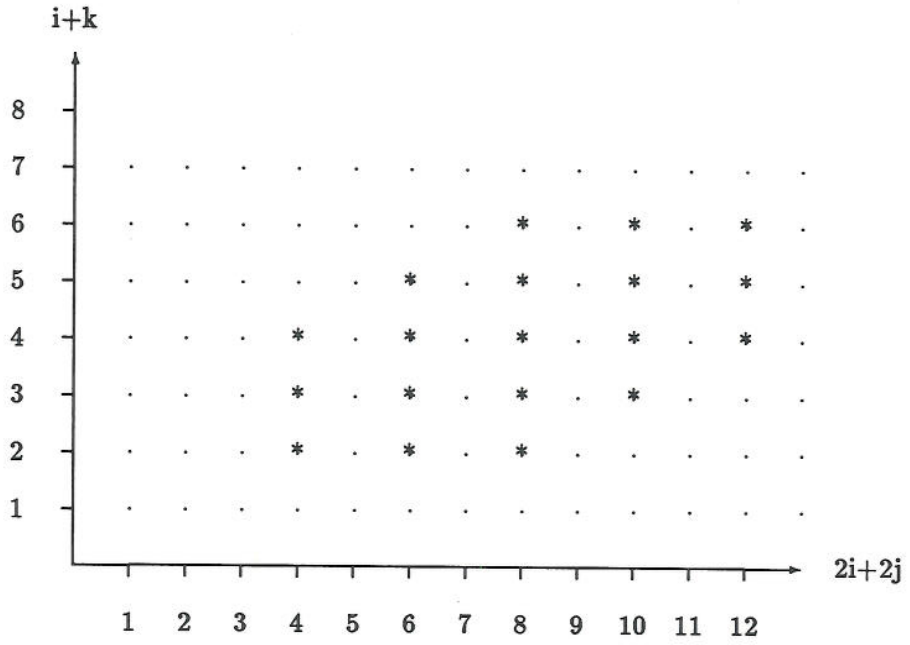


figure 3.2.3

- **volume des références**

  On appelle le volume des références, le nombre total de références faites au tableau dans le corps de boucles.

  Il est utile au calcul d'une fonction de coût des transferts en mémoire locale.

  Il correspond au volume de l'espace d'itération, dans notre exemple: 27(=3x3x3) références au tableau T.

- **volume de stockage**

  Le volume de stockage est le nombre de données réellement utilisées par l'application.

  Dans notre exemple, l'ensemble de ces éléments est représenté sur la figure 3.2.3. Soit au total: 19 éléments.

- **volume de transfert**

  Le volume de transfert est égal au nombre d'éléments transférés. Il peut être plus important que le volume de stockage puisqu'il est possible de transférer plusieurs fois le même élément.

  Il est utile à l'évaluation de la fonction de coût des transferts en mémoire locale, ainsi qu'à la génération des déclarations des tableaux locaux utilisés pour le stockage des données en mémoire locale.

  Dans notre exemple, le nombre des éléments transférés peut varier de 19 à 27 suivant que l'on transfère une seule fois les éléments réellement référencés ou que l'on utilise comme algorithme de transfert, par exemple, le code de calcul.

3.2.4 Fonction de coût

Dans le cas où les processeurs peuvent accéder aux deux niveaux de mémoire, il est intéressant de savoir s'il est préférable de laisser les données en mémoire globale ou de les transférer.

Plusieurs fonctions de coût des transferts de données en mémoire locale sont envisageables. K. GAllivan, W. Jalby et D. Gannon en proposent une dans [GJGa88]. Elle suppose que le temps d'accès en lecture ou en écriture à un mot appartenant à la mémoire locale est t_l, et que le temps d'accès à k mots consécutifs en mémoire globale est $k\ t_g + d_g$ (t_g est le temps d'accès à la mémoire globale et d_g le temps de démarrage du transfert, avec $t_l \ll t_g$). On compare avant d'effectuer tout transfert en mémoire locale, les temps que devrait passer le processeur en accès mémoire, si les données restaient en mémoire globale ou si elles étaient transférées en mémoire locale. Quand le second est plus court, les copies sont effectuées.

Si un processeur exécute le corps de boucles:

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

les éléments du tableau T seront donc transférés si:

$M'\ t_g + d_g + N\ M\ t_l \leq N\ (\ M\ t_g + d_g\ )$ où M' est le nombre d'éléments référencés transférés.

Cette fonction de coût est basée sur l'évaluation du volume des données transférées en mémoire locale, et du nombre total de références faites au tableau dans le corps de boucles.

Seule une bonne approximation de ces deux volumes est nécessaire, sachant que la fonction ne pourra donner qu'une approximation des temps réellement passés par un processeur pour effectuer les transferts.

3.3 Exemple

En entrée, on dispose du code d'une tâche parallèle que l'on veut transformer en un code de tâche utilisant la mémoire locale du processeur P où elle s'exécute. Prenons pour exemple la figure 3.3.

La première partie du code contient les déclarations des tableaux locaux TLA et TLB. Tous les calculs s'effectueront en mémoire locale et feront donc uniquement référence à ces tableaux.

La seconde correspond au code de transfert des éléments utilisés par la tâche de la mémoire globale vers la mémoire locale. On copie les éléments du tableau TB référencés dans le corps de boucles dans un tableau TLB local à la mémoire locale.

La troisième correspond au code de calcul. Les références au éléments des tableaux TA et TB on été transformées en des références aux tableaux locaux TLA et TLB.

```
C Code de calcul
C
DO A=1+IT*P,(IT+1)*P
  DO B=1,N
    DO C=1,M
        DO D=1,L
            TA(2*B+D,C+D,A) = TB (B+D,C+D,A)
           Ref.1                    Ref.2
        ENDDO
    ENDDO
  ENDDO
```

Voici le type de code que l'on souhaite générer:

```
TASK  T(IT)
C
REAL TLA(3:2*N+L,2:M+L,1+IT*P:(IT+1)*P)
C
REAL TLB(2:N+L,2:M+L,1+IT*P:(IT+1)*P)
C
C Copie de la memoire globale vers la memoire locale (Ref.2)
C
DO I=1+IT*P,(IT+1)*P
   DO J=2,M+L
      DO K=MAX (2,J-M+1),MIN(L+N, J-1+N)
          TLB(K,J,I) ←—  TB(K,J,I)
      ENDDO
   ENDDO
ENDDO
C
C Code de calcul
C
DO A=1+IT*P,(IT+1)*P
  DO B=1,N
    DO C=1,M
        DO D=1,L
            TLA(2*B+D,C+D,A) = TLB (B+D,C+D,A)
           Ref.1                    Ref.2
        ENDDO
    ENDDO
  ENDDO
ENDDO
C
C Copie de la memoire locale vers la memoire globale (Ref.1)
C
DO I=1+IT*P,(IT+1)*P
  DO J=1-2*N,M-2
    DO K=MAX(3,1+2*((2-J)/2)),MIN(L+2*N,L+2*((M-J)/2))
        TA(K,J+K,I) ←— TLA(K,J+K,I)
    ENDDO
  ENDDO
ENDDO
```

figure 3.3

Enfin, la dernière partie correspond au code de transfert de la mémoire locale vers la mémoire globale des éléments qui ont été modifiés au cours de l'exécution. On recopie ces éléments du tableau local TLA dans le tableau TA. Les bornes de boucle de ce code de transfert contiennent ici des divisions entières, elles traduisent la non convexité de l'ensemble des éléments modifiés par la tâche.

3.4 Conclusion

La génération des codes de transfert pour une référence au tableau revient à calculer la projection d'un polyèdre (domaine d'itération) dans une nouvelle base (une base du domaine image). L'ensemble résultant de cette opération n'est malheureusement pas forcément un polyèdre [GJGa88].

Pour le code de copie de la mémoire globale vers la mémoire locale, nous utilisons le polyèdre le plus approchant de cette projection, l'enveloppe convexe des éléments référencés.

Cependant, pour le code de transfert de la mémoire locale vers la mémoire globale, il faut recopier l'ensemble exact des éléments modifiés par la tâche pour maintenir la cohérence des données. Lorsque l'ensemble des éléments modifiés est non convexe, nous utilisons des divisions entières dans les expressions des bornes de boucle du code de transfert pour permettre de traduire la non-convexité du polyèdre sans avoir à ajouter d'overhead de contrôle.

Reprenons l'exemple proposé dans [GJGa88] :

```
DO I=1,10
  DO J=1,20
    DO K=1,30
       T(3*I+K-5,J+K)  = ...
    ENDDO
  ENDDO
ENDDO
```

le code de transfert de la mémoire locale vers la mémoire globale généré automatiquement par nos algorithmes est le suivant:

```
DO A=-29,17
   DO B=MAX(4,1+3*((-A+3)/3)),MIN(17,28-3*((A-18)/3))
          T(B-5,A+B)  = TL(B-5,A+B)
   ENDDO
ENDDO
```

Il contient des divisions entières dans les expressions des bornes de boucle car le domaine image est non convexe (voir figure3.2.2).

Dans le cas où les processeurs peuvent accéder aux deux niveaux de mémoire, il est intéressant de savoir s'il est préférable de laisser les données en mémoire globale ou de les transférer. On détermine alors à l'aide d'une fonction de coût quelles variables doivent être

dupliquées en mémoire locale. Cette fonction de coût est basée sur l'évaluation du volume des données transférées en mémoire locale et du nombre total de références faites au tableau dans le corps de boucles.

Chapitre 4

Utilisation de la théorie des polyèdres

La caractérisation automatique de l'ensemble des données référencées par un tableau T dans un corps de boucles conduit à modéliser des ensembles de points entiers.

Nous avons exposé dans le chapitre précédent, les hypothèses de linéarité qui garantissent un domaine d'itération du corps de boucles convexe. Elles nous permettent d'utiliser les *polyèdres* pour représenter ce domaine et attribuent au polyèdre une propriété très intéressante: ses sommets ont des coordonnées entières.

Pour modéliser les éléments référencés par T, il faut projeter le domaine d'itération du corps de boucles sur le Z-module défini par la (ou les) fonction d'accès au tableau. Le résultat de cette opération de projection est parfois non convexe.

Effectuer des opérations sur des ensembles non convexes étant souvent trés compliqué, nous définissons donc, dans ce chapitre, les fonctions qui permettent de découper les ensembles non convexes en ensembles convexes et/ou de trouver des ensembles convexes approchants.

Après quelques rappels et définitions, nous décrirons les algorithmes qui permettent de trouver:

- l'intersection de deux polyèdres,
- l'enveloppe convexe de deux polyèdres,
- la différence de deux polyèdres,
- la projection d'un polyèdre,
- le plus petit polyèdre convexe englobant un "ensemble de points entiers non convexe",
- le plus grand polyèdre convexe contenu dans un "ensemble de points entiers non convexe",
- le découpage d'un "ensemble de points entiers non convexe" en polyèdres convexes.

Enfin, nous présenterons des méthodes de calcul du volume de ces polyèdres.

Note: dans toutes les démonstrations, les divisions sont des divisions entières, et les opérations de transformation des expressions entières utilisées sont celles exposées en Annexe [§Annexe].

4.1 Définitions

Dans cette section nous rappelons [Schr87], [NeWo88] les propriétés des objets que nous utilisons tout au long de cette étude.

4.1.1 Polyèdres

Un **polyèdre** convexe $P \subseteq R^n$ est un ensemble de points qui satisfont un nombre fini d'inéquations linéaires; c'est à dire, P peut s'écrire sous la forme :
$P = \{ x \in R^n \mid A \cdot x \leq b \}$
où A est une matrice et b un vecteur. P est donc une intersection finie de demi-espaces affines.

Un polyèdre convexe peut être caractérisé soit par un **système d'inéquations linéaires** soit par un **système générateur** [CaSz89].

Définissons les systèmes générateurs. Un **système générateur** est constitué de trois ensembles. Un ensemble de sommets, un ensemble de rayons et un ensemble de droites.

Soit P un polyèdre, tout sommet de P est un point X qui ne peut pas être obtenu par combinaison linéaire d'autres points de P:

$$\begin{cases} X = \sum_{i=1}^{p} \lambda_i X_i \ , \ \sum_{i=1}^{p} \lambda_i = 1, \ p \geq 1, \\ \forall i \in [1..p], \lambda_i \in [0,1] \ \ et \ \ \vec{X_i} \in P \end{cases} \implies \begin{cases} \forall i \in [1..p] \\ \lambda_i = 0 \ \ ou \ \ \vec{X_i} = X \end{cases}$$

Un vecteur $\vec{r}$ de R^n est un rayon de P si et seulement si il existe une demi-droite de vecteur directeur $\vec{r}$, entièrement contenue dans P:

$$\forall \mu \geq 0 \ \ (\vec{X} \in P) \implies (\vec{X} + \mu \vec{r} \in P)$$

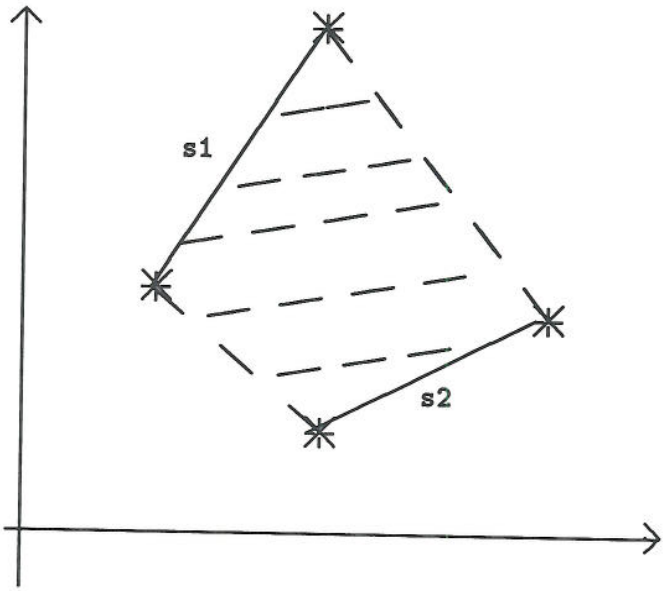
Deux rayons parallèles et de même sens sont considérés comme égaux.

Un vecteur $\vec{d}$ de R^n est une droite de P si et seulement si $\vec{d}$ est un vecteur directeur d'une droite entièrement contenue dans P:

$$\forall \nu \in R, \ (\vec{X} \in P) \implies (\vec{X} + \nu \vec{d} \in P)$$

Halbwachs propose dans sa thèse [Halb79] les algorithmes permettant de passer d'une représentation à l'autre. La conversion d'un système générateur en système linéaire s'effectue par recherches successives de l'enveloppe convexe d'un polyèdre avec l'un des représentants du système générateur. L'opération inverse se traduit par intersections successives d'un polyèdre avec un demi-espace (une contrainte du système linéaire). Une autre méthode a été proposée par F.Fernandez et P.Quinton [FeQu88]. Elle utilise l'algorithme de programmation linéaire de Chernikova.

4.1.2 Enveloppe convexe



\- - enveloppe convexe de deux segments de droite

figure 4.1.2

L'**enveloppe convexe** d'un ensemble X de points est égale au plus petit ensemble convexe contenant X, on la note env.conv(X):

$$env.conv(\ X\ ) = \{\ \lambda_1 x_1 + \lambda_2 x_2 + \cdots \lambda_n x_n \ /\ n \geq 1\ ; x_1, \cdots, x_n \in X; \lambda_1, \cdots \lambda_n \geq 0;$$
$$\lambda_1 + \cdots + \lambda_n = 1\}$$

Un ensemble est dit **convexe** s'il est égal à son enveloppe convexe.

4.1.3 Z-module

On appelle **Z-module**[1] dans Z^n, l'ensemble des combinaisons entières d'un ensemble de vecteurs linéairement indépendants dans Z^n.

Ces vecteurs indépendants forment la base du Z-module.

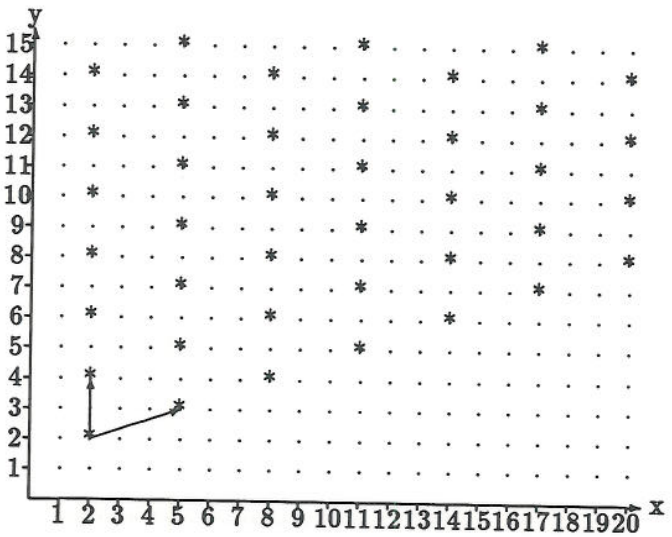


figure 4.1.3

On note le Z-module:
$L(A) = \{y \in Z^n : y = Ax, x \in Z^m\}$ où A est une matrice entière $n \times m$.

On a le **théorème** suivant [Herm51]:

Si A est une matrice $n \times m$ de rang r, alors il existe deux matrices unimodulaires P et Q telles que:

$$P \,.\, A \,.\, Q = \begin{pmatrix} L & 0 \\ S & 0 \end{pmatrix} = H$$

où H est la **forme réduite de Hermite** *associée à A de rang r, $L(r \times r)$ est une matrice triangulaire inférieure, $S((n-r) \times r)$ une matrice, et $P(n \times n)$ une matrice de permutation.*

et les propositions suivantes, extraites de [NeWo88](p.190).

Proposition 1:

$L(A) = L(H)$

Proposition 2:

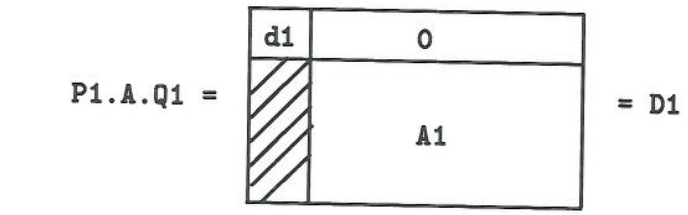
Si $L(A) = L(H)$ alors H est une base du Z-module $L(A)$.

[1] La notion de *treillis* étant plus couramment utilisée pour définir un ensemble ordonné, nous avons repris la notion de Z-module utilisée par Jalby & Co. Les références bibliographiques référencées utilisent la notion de *lattice*

Calcul de la forme réduite de Hermite

Pour toute matrice $A(m \times n)$ de rang r le calcul de la forme réduite de Hermite s'effectue de façon constructive de la manière suivante [Min83]:

1. Déterminons d'abord deux matrices unimodulaires P_1 (matrice de permutation) et Q_1 telles que:



(les termes peuvent être non nuls dans les parties hachurées).

Par permutation de lignes, on amène une ligne non identiquement nulle en première ligne (prémultiplication par une matrice de permutation P_1).

Puis par des permutations de colonnes (postmultiplication par des matrices de permutation) on amène le plus petit terme en valeur absolue des termes non nuls de cette première ligne en position (1,1). Soit a_1^1 ce terme.

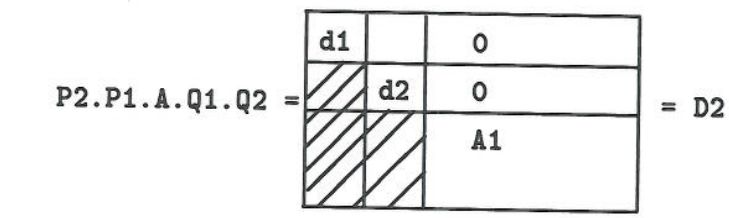
En postmultipliant par des matrices élémentaires appropriées, on peut remplacer chaque terme de la première ligne (autre que a_1^1) par son reste de la division euclidienne par a_1^1.

Si tous ces restes sont nuls, on obtient la forme D_1 avec $d_1 = a_1^1$.

Sinon on recommence sur la matrice obtenue ce que l'on a fait précédemment sur A. Nécessairement au bout d'un nombre fini de telles opérations, on aboutira à la forme D_1 puisque les termes non nuls de la ligne 1 autre que a_1^1 sont remplacés à chaque fois par des termes strictement plus petits en valeur absolue. Les seules transformations effectuées sur les lignes sont des permutations (prémultiplication par la matrice de permutation P_1).

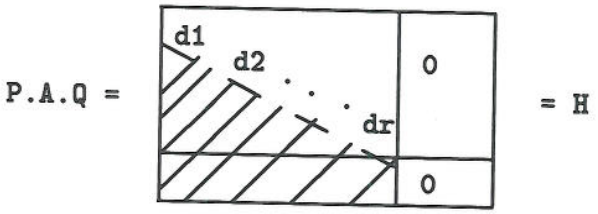
2. La forme D_1 ayant été obtenue, si $A_1 = 0$ alors la matrice A est de rang $r = 1$ et $H = D_1$ est une forme réduite de Hermite de A.

Si A_1 n'est pas nulle, alors on peut appliquer à A_1 les transformations définies en (1) pour obtenir la forme:



où P_2 est une matrice de permutation.

On obtient ainsi une suite de matrices $A_1, A_2, ...$ dont les dimensions diminuent d'une unité à chaque pas. Si $r = rang(A)$, on obtient ainsi, nécessairement, en r étapes $A_r = 0$ et une forme réduite de Hermite du type:



avec $P = P_r\ P_{r-1}\ ...\ P_2\ P_1$ (matrice de permutation) et $Q = Q_1\ Q_2\ ...Q_r$.

La forme réduite de Hermite se calcule en temps polylomial [KaBa79], [Min83], [NeWo88]. De manière générale, il n'est pas possible de calculer la forme de Hermite associée à une matrice comportant des constantes symboliques

4.1.4 Z-polyèdre

On appelle **Z-polyèdre** l'ensemble des points entiers résultant de l'intersection d'un polyèdre convexe (un système linéaire en nombres entiers) et d'un Z-module.

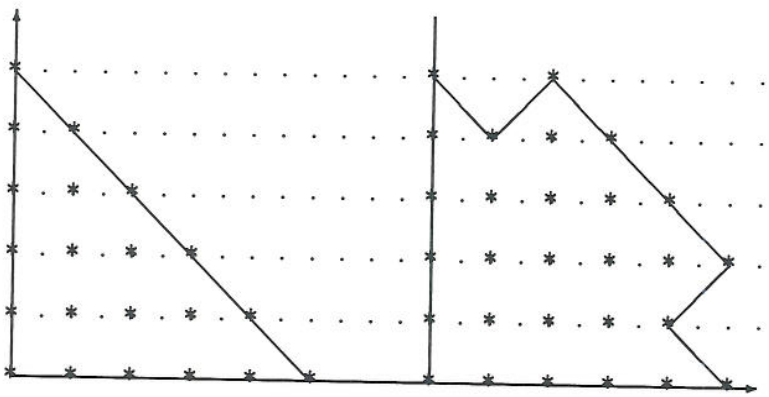
Tout point entier du Z-polyèdre doit appartenir au système linéaire qui le définit et être une combinaison linéaire des vecteurs de base qui génèrent le Z-module.

Toutes les opérations définies sur les Z-polyèdres peuvent être utilisées pour manipuler les ensembles de points entiers contenus dans un polyèdre, puisque ces ensembles constituent une famille particulière de Z-polyèdres dont les vecteurs de base sont les vecteurs unitaires.

4.1.5 Z-polyèdre non convexe

Tout **Z-polyèdre** est par définition **convexe.**

Par abus de langage nous parlerons de **Z-polyèdre non convexe**, pour désigner un ensemble non convexe de points entiers, déduit d'un polyèdre par transformations linéaires ou par projections.



Z-polyedre convexe

Z-polyedre non convexe

4.2 Algorithmes usuels

Nous décrivons dans cette section les algorithmes souvent utilisés lorsque l'on manipule les polyèdres et a fortiori des Z-polyèdres: l'étude de la faisabilité d'un système d'inéquations linéaires, l'intersection et l'union de deux polyèdres.

Les algorithmes traitant des Z-polyèdres et de la différence de deux (Z-)polyèdres ont été développés dans le cadre de cette étude.

4.2.1 Faisabilité d'un système d'inéquations linéaires

Le système linéaire $Ax \leq b$ est **faisable** si les variables de x peuvent prendre des valeurs telles que toutes les inéquations sont satisfaites. Dans les autres cas il est infaisable.

De même nous dirons qu'un système linéaire est **faisable en entiers** s'il existe une valeur entière de x satisfaisant les inéquations du système.

Les algorithmes permettant de rechercher une solution à un système linéaire et donc de tester la faisabilité de ce système sont relativement nombreux. Tous les algorithmes de programmation linéaire tels que la *méthode du simplexe* en font partie et de nombreux ouvrages leurs ont été consacrés [Min83], [Schr87], [NeWo88].

La méthode de Fourier-Motzkin [Four24] permet de tester uniquement l'existence d'une solution. Elle ne la calcule pas. Elle procède par éliminations successives des variables du système, par combinaisons linéaires de paires d'inégalités, jusqu'à:

- trouver une inéquation absurde dans le système (*exemple* : $1 \leq 0$). Le système est alors déclaré comme non faisable.
- élimination totale de toutes les variables du système. Le système est alors faisable.

Ces algorithmes sont en général de complexité exponentielle. Cependant, sur des cas pratiques, ils se révèlent d'une bonne efficacité [Min83].

Il n'existe pas de méthode équivalente permettant de tester l'existence d'une solution entière. Il faut, dans ce cas, recourir aux algorithmes de programmation linéaire en nombres entiers du type: algorithme des congruences décroissantes (utilisant les coupes de Gomory) [Min83] ou à l'algorithme paramétrique de résolution de systèmes d'inéquations linéaires en nombres entiers proposé par P.Feautrier [Feau88b]. Ces algorithmes sont tous de complexité exponentielle.

4.2.2 Intersection de deux Z-polyèdres

Il est plus facile de calculer l'intersection de deux polyèdres lorsqu'ils sont représentés sous la forme de systèmes linéaires que lorsqu'ils sont sous la forme de systèmes générateurs.

L'**intersection** de deux Z-polyèdres est un Z-polyèdre dont le système linéaire est l'union des deux systèmes linéaires initiaux.

Le Z-module du nouveau Z-polyèdre est différent des deux Z-modules initiaux s'ils sont distincts, comme le montre la figure suivante.

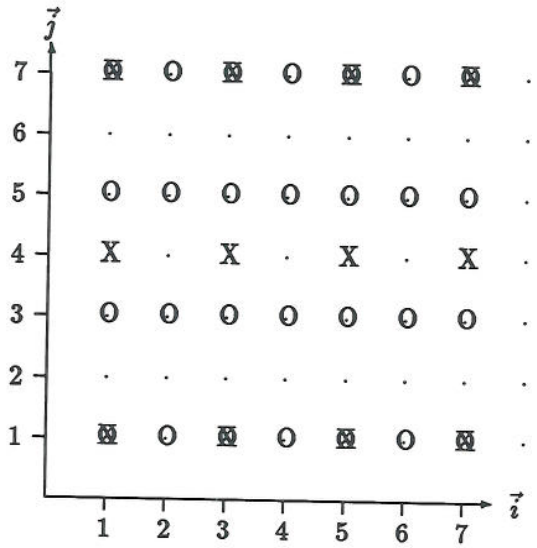


figure 4.2.2

Le premier Z-module dont les points sont représentés par des "o" a pour vecteurs de base (1,0) et (0,2). Le second dont les points sont représentés par des "X" a pour vecteurs de base (2,0) et (2,3). Le Z-module intersection a pour vecteurs de base (2,0) et (0,6).

La base du Z-module intersection se calcule de la manière suivante:

Soit T_1 un Z-module de base H_1 et T_2 le Z-module de base H_2.

Soit X un point du Z-module T_1, X est combinaison linéaire des vecteurs de base du Z-module et vérifie: $X = \vec{\alpha}^t \vec{H}_1 \vec{t_1}$ où $\vec{\alpha}$ est un vecteur à composantes entières et $\vec{t_1}$ la base dans lequelle s'exprime le Z-module.

Si X appartient au Z-module intersection $T_1 \bigcap T_2$ alors il est aussi combinaison linéaire des vecteurs de base du Z-module T_2 et vérifie: $X = \vec{\beta}^t \vec{H}_2 \vec{t_2}$ où $\vec{\beta}$ est un vecteur à composantes entières et $\vec{t_2}$ la base dans lequelle s'exprime le Z-module T_2.

L'ensemble des points appartenant au Z-module intersection vérifient:
$\vec{\alpha}^t \vec{H}_1 \vec{t_1} = \vec{\beta}^t \vec{H}_2 \vec{t_2}$. Ce système d'équations linéaires en nombres entiers peut être résolu en utilisant la méthode de McLane et Birkhoff sur Z [MaBi71], [Anco87] qui utilise les formes réduite ou normale de Smith [Min83].

Soit $\vec{\alpha_i}$ et $\vec{\beta_i}$ les solutions du système d'égalités $\vec{\alpha_i}^t \vec{H}_1 \vec{t_1} = \vec{\beta_i}^t \vec{H}_2 \vec{t_2}$ et $\vec{\alpha_0}$ le vecteur dont les composantes sont les pgcd des composantes des vecteurs $\vec{\alpha_i}$. $\vec{\alpha_0}$ est le plus petit vecteur vérifiant: $X = \vec{\alpha_0}^t \vec{H}_1 \vec{t_1}$.

La base du Z-module intersection vaut donc: $\vec{\alpha_0}^t \vec{H}_1 \vec{t_1}$.

Dans l'exemple précédent, nous avons:

$$H_1 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \quad t_1 = \begin{pmatrix} i \\ j \end{pmatrix} \qquad H_2 = \begin{pmatrix} 2 & 0 \\ 2 & 3 \end{pmatrix} \quad t_2 = \begin{pmatrix} i \\ j \end{pmatrix}$$

Tout point appartenant à l'intersection des deux Z-modules doit vérifier:

$$\vec{\alpha^t} \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \vec{\beta^t} \begin{pmatrix} 2 & 0 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

c'est-à-dire le système d'inéquations linéaires suivant: $\begin{cases} \alpha_1 \, i = 2\beta_1 i \\ 2\alpha_2 j = 2\beta_2 i + 3\beta_2 j \end{cases}$

Ce système admet une solution entière si α_1 est un multiple de deux et α_2 un multiple de trois. Nous en déduisons la base du Z-module intersection:

$$\vec{\alpha_0^t} \vec{H_1} \vec{t_1} = \begin{pmatrix} 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 2i & 6j \end{pmatrix}$$

et les vecteurs de base (2,0) et (0,6).

4.2.3 Union de deux polyèdres

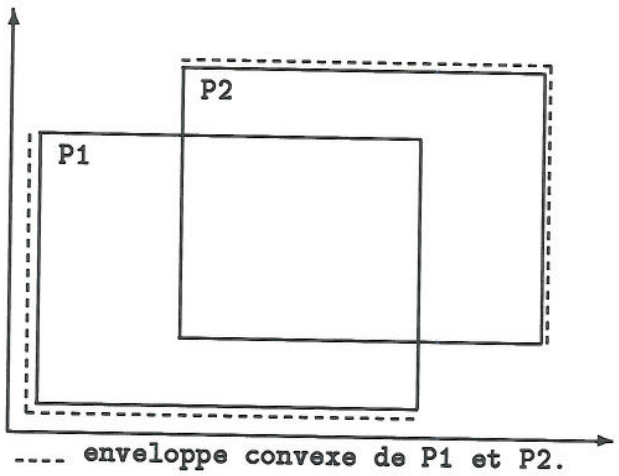


figure 4.2.3

L'union de deux polyèdres n'est pas en général un polyèdre, comme le montre la figure 4.2.3.

Nous approximerons donc l'union de deux polyèdres par l'enveloppe convexe de ces deux polyèdres. Ceci nous permet de rester dans le domaine des polyèdres.

L'enveloppe convexe de deux polyèdres est facile à calculer lorsque les deux polyèdres sont sous la forme de systèmes générateurs: c'est l'union des deux systèmes générateurs.

Nous ne parlerons ni de l'union, ni de l'enveloppe convexe de deux Z-polyèdres car l'union de deux Z-modules ne peut pas, en général, être caractérisée par un Z-module. L'exemple 4.2.2 en témoigne bien. Il est effectivement impossible de trouver une base de vecteurs dont les combinaisons linéaires caractériseraient l'ensemble des points ("X" et "o") représentés sur la figure. Il est donc impossible de représenter l'union ou l'enveloppe convexe de deux Z-polyèdres par un Z-polyèdre.

4.3 Différence de deux Z-polyèdres

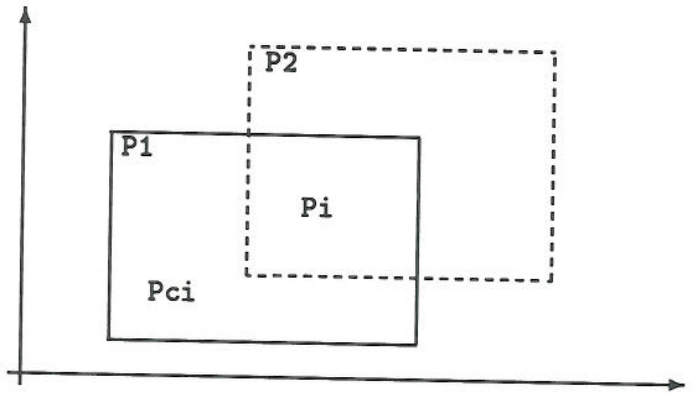


figure 4.3.1

Il est souvent utile de connaître le complémentaire de l'intersection de deux polyèdres dans l'un d'eux.

L'algorithme que nous présentons calcule un ensemble de polyèdres convexes P_{C_j} dont l'union forme ce complémentaire.

Notons P_i le polyèdre résultant de l'intersection des deux polyèdres P_1 et P_2, et P_{C_j} les polyèdres dont l'union est égale au polyèdre complémentaire recherché P_{COMP}.

Algorithme 4.3:

Si nous cherchons le complémentaire de P_i dans P_1, nous posons $P_C = P_1$
Pour chaque inégalité $A_i.X \leq b_i$ de P_i appartenant à P_2, faire:

- **ajouter la contrainte $A_i.X > b_i$ à P_C**
- **tester la faisabilité du système P_C**
  - **Si le système est non faisable, retirer l'inégalité $A_i.X > b_i$ de P_C.**
  - **Si le système est faisable , P_C est l'une des partitions de P_{COMP}. Ajouter P_C à l'ensemble des polyèdres formant le complémentaire: $P_{C_i} = P_C$, $P_{COMP} = P_{COMP} + P_{C_i}$, et réinverser le sens de l'inégalité $A_i.X > b_i$ dans P_C.**

Preuve:

Chaque inégalité constituant P_i appartient soit à P_1 soit à P_2. $P_C = P_1$ au départ.

A chaque étape de l'algorithme on ajoute une contrainte $\overline{C_i}$ à P_C , où $C_i \in P_i \bigcap P_2$. $P_C \bigcap \overline{C_i}$ et $P_C \bigcap C_i$ forment une partition de P_C. $P_{C_i} = P_C \bigcap \overline{C_i} \in P_{COMP}$ car $C_i \in P_i$ et $P_C \in P_1$.

A l'itération suivante on a $P_C = P_C \bigcap C_i$ et l'on obtient une partition de P_C donnant un nouveau polyèdre $P_{C_{i+1}} = P_C \bigcap \overline{C_{i+1}} \in P_{COMP}$.

Après n itérations, on obtient n polyèdres P_{C_i} constituant une partition de P_{COMP}, où n est le nombre d'inégalités de P_2 contenues dans P_i.

Pour traiter la différence de deux Z-polyèdres, il faut introduire en plus la notion de Z-module. Il faut aussi distinguer plusieurs cas: les cas où les Z-modules des deux Z-polyèdres sont identiques ou inclus, des autres cas.
Voici un exemple de Z-modules ni identiques, ni inclus:

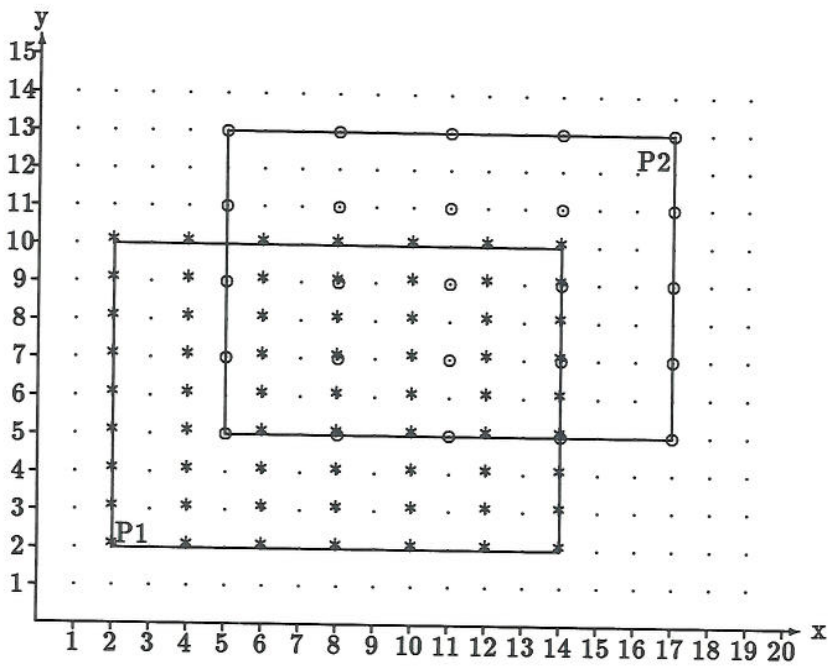


figure 4.3.2 - Z-modules différents, non inclus.

Lorsque les Z-modules de P_1 et P_2 ne sont ni identiques, ni inclus, l'ensemble des éléments de P_1 qui n'appartiennent pas à l'intersection de P_1 et P_2 est constitué non seulement de tous les éléments extérieurs au domaine "intersection", mais aussi des éléments qui appartiennent à ce domaine et qui sont générés par le Z-module de P_1 moins le Z-module "intersection".

La différence de deux Z-modules n'étant en général pas un Z-module, la différence des deux Z-polyèdres P_1 et P_2 ne peut pas s'exprimer directement en fonction de Z-polyèdres.

Nous ne rechercherons donc pas, dans ce cas, à calculer la différence des deux Z-polyèdres.

La figure suivante montre un exemple de Z-modules inclus.

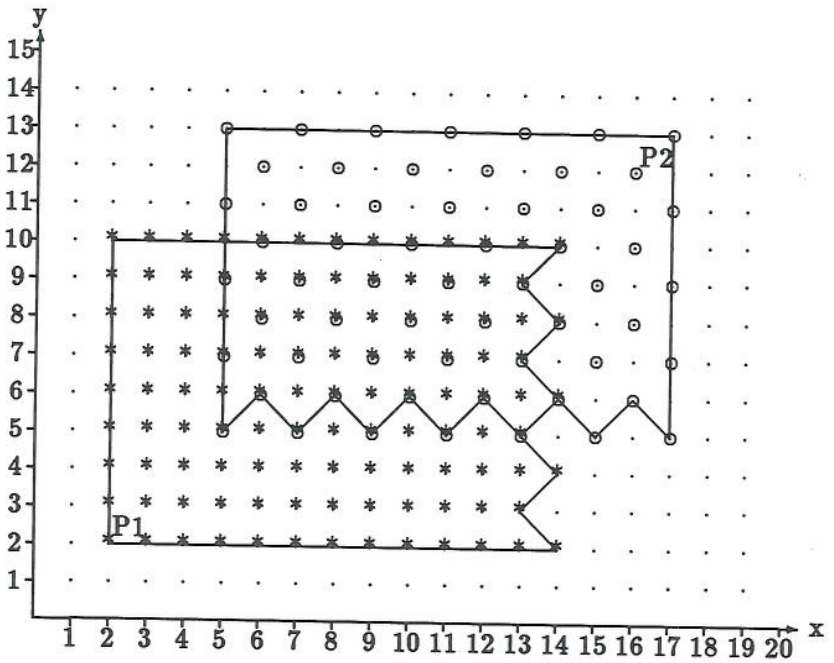


figure 4.3.3 - Z-modules inclus

Soit T_1 le Z-module générant les éléments de P_1 et T_2 le Z-module générant les éléments de P_2. On note $H1$ la base caractérisant le Z-module T_1 selon la base $\vec{t_1}$ de P_1 et $H2$ la base caractérisant le Z-module T_2 selon la base $\vec{t_2}$ de P_2.

On dira que T_1 et T_2 sont *identiques* si: $H_1 = H_2$. De même, on dira que T_1 est *inclus* dans T_2 s'il existe un vecteur entier $\vec{k}$ tel que chaque composante du vecteur vérifie $|k_i| \geq 1$, *et* $H_1 = \vec{k}^t.H_2$

Tous les éléments appartenant à l'intersection de P_1 et P_2 appartiennent ici au Z-module "intersection": le Z-module T_1. De plus, l'ensemble des éléments de P_2 qui n'appartiennent pas à l'intersection de P_1 et P_2 est constitué uniquement des éléments extérieurs au domaine "intersection".

Le calcul du complémentaire du Z-polyèdre intersection P_i dans le Z-polyèdre P_2 peut

donc être effectué en utilisant l'algorithme 4.3 décrit précédemment. Les éléments appartenant à ce complémentaire appartiennent au Z-module de P_2.

4.4 Etude de la projection d'un polyèdre

Nous avons développé dans le chapitre précédent l'un des problèmes les plus importants pour générer automatiquement le code de transfert: la caractérisation des éléments référencés par un tableau T dans un corps de boucles.

Soit F la fonction d'accès aux éléments du tableau T. L'ensemble des indices de tableau recherchés correspond à l'image par F du domaine d'itération. Cette image s'obtient par un changement de base, de la base du domaine d'itération dans une base du domaine image, et une projection sur cette nouvelle base, de dimension parfois inférieure à la première.

L'opération de projection est donc l'une des opérations clé de nos algorithmes: elle permet de caractériser l'ensemble exact des points référencés par F.

4.4.1 Définitions

Définition de la projection:

Si P est un polyèdre convexe de R^p , on appelle **projection** de P selon la i-ème variable, le polyèdre $proj(P, X_i)$ défini par :

$$proj(P, X_i) = \{(X_1..X_{i-1}X_{i+1}..X_p) \in R^{p-1} \mid \exists\, z \in R \text{ tel que } (X_1...X_{i-1}\ z\ X_{i+1}...X_p\ ) \in P\ \}$$

Soit $A.X \leq B$ le système de contraintes définissant P. Alors un système de contraintes $A'.X \leq B'$ de $proj(P, X_i)$ est défini comme suit :

$$\forall\ l \in 1..m \quad \text{tel que} \quad A^i_l = 0,$$
$$\exists\ l' \quad \text{tel que} \quad A'_{l'} = A_l \text{ et } B'_{l'} = B_l$$
$$\forall\ l_1\ ,\ l_2 \in 1...m \quad \text{tels que} \quad A^i{}_{l_1} > 0 \text{ et } A^i{}_{l_2} < 0\ ,$$
$$\exists\, l' \quad \text{tel que} \quad A'_{l'} = A^i{}_{l_1} \cdot A_{l_2} - A^i{}_{l_2} \cdot A_{l_1} \quad \text{et} \quad B'_{l'} = A^i{}_{l_1} \cdot B_{l_2} - A^i{}_{l_2} \cdot B_{l_1}$$

toutes les contraintes du système $A'.X \leq B'$ sont obtenues par les règles précédentes (i.e. des combinaisons linéaires positives de lignes de A de manière à annuler la i-ème colonne).

Calculer la projection d'un polyèdre de R^n selon une variable revient à éliminer la variable du système de contraintes qui le définit. L'algorithme d'élimination d'une variable a été proposé par Fourier-Motzkin [Four24].

4.4.2 Exemples et problèmes

Prenons le système

$$P1 \left\{ \begin{array}{rcrl} x - 5\,y & \leq & -8 & (1) \\ -2\,x + 5\,y & \leq & 6 & (2) \\ x & \leq & 7 & (3) \end{array} \right\} \quad x - 5y - 2x + 5y \leq -8 + 6 \iff -x \leq -2$$

la projection selon Y donne

$$(DP) \left\{ \begin{array}{rcl} -x & \leq & -2 \\ x & \leq & 7 \end{array} \right.$$

Le polyèdre correspondant au système P1 est représenté sur la figure suivante:

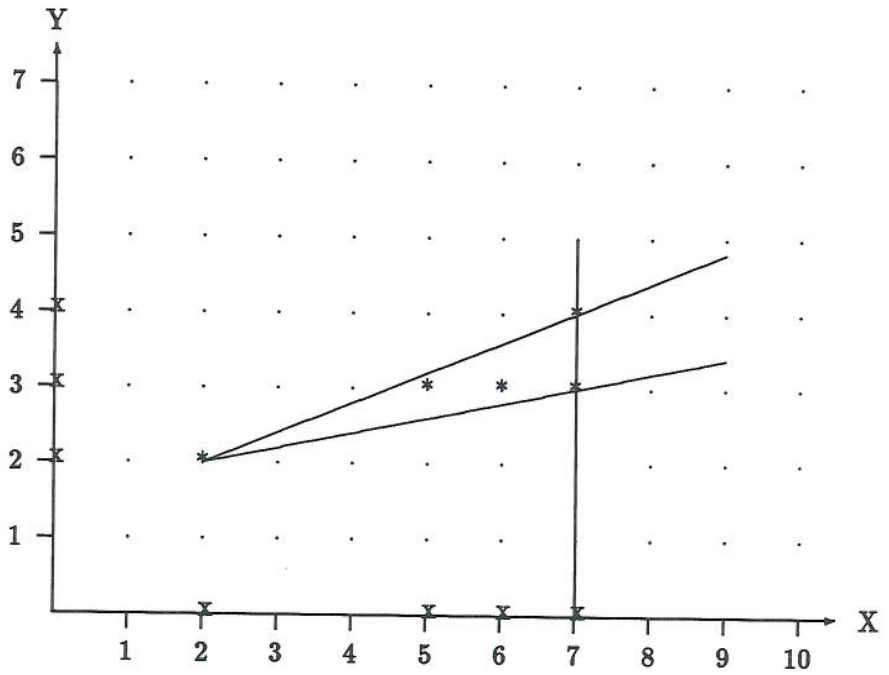


figure 4.4.2

Les projections des points entiers du polyèdre sont matérialisées par des "x" sur les axes. La projection des points entiers du polyèdre selon la variable y est égale à l'ensemble: {2, 5, 6 , 7}.

Nous remarquons que le domaine de projection (DP) sur **R** contient les valeurs $X = 3$ et $X = 4$, auxquelles ne correspondent aucun point entier du polyèdre initial.

Nous définissons donc la projection entière d'un polyèdre ou projection d'un Z-polyèdre:

On appelle **projection entière** d'un polyèdre P selon une variable X_i l'ensemble des projections des points entiers du polyèdre P. Elle se définit par:

$$proj\text{-}entiere(P, X_i) = \{(X_1..X_{i-1}X_{i+1}..X_p) \in Z^{p-1} \mid \exists\, z \in Z \text{ tel que } (X_1...X_{i-1}\ z\ X_{i+1}...X_p\ ) \in P\ \}$$

La projection entière d'un polyèdre selon une variable s'obtient en éliminant la variable du système linéaire tout en respectant les règles de transformation des expressions entières, rappelées au [§Annexe]. Ces règles introduisent des divisions entières dans les expressions.

Cette projection entière ne correspond pas forcément à un polyèdre convexe. Nous dirons par abus de langage que nous obtenons un Z-polyèdre non-convexe.

Donnons un exemple:

$$\begin{cases} -z & \leq & -1 \\ z & \leq & 10 \\ -x + 3z & \leq & -1 \\ -y - 3z & \leq & -1 \end{cases} \iff \begin{array}{l} z \leq \frac{x-1}{3} \\ \frac{-y+3}{3} \leq z \end{array}$$

La projection **entière** selon la variable **z** donne:

$$\begin{cases} \frac{-y+3}{3} & \leq & \frac{x-1}{3} \\ -x & \leq & -4 \\ -y & \leq & 29 \end{cases}$$

Ces éléments sont représentés sur la figure 4.4.2 bis.

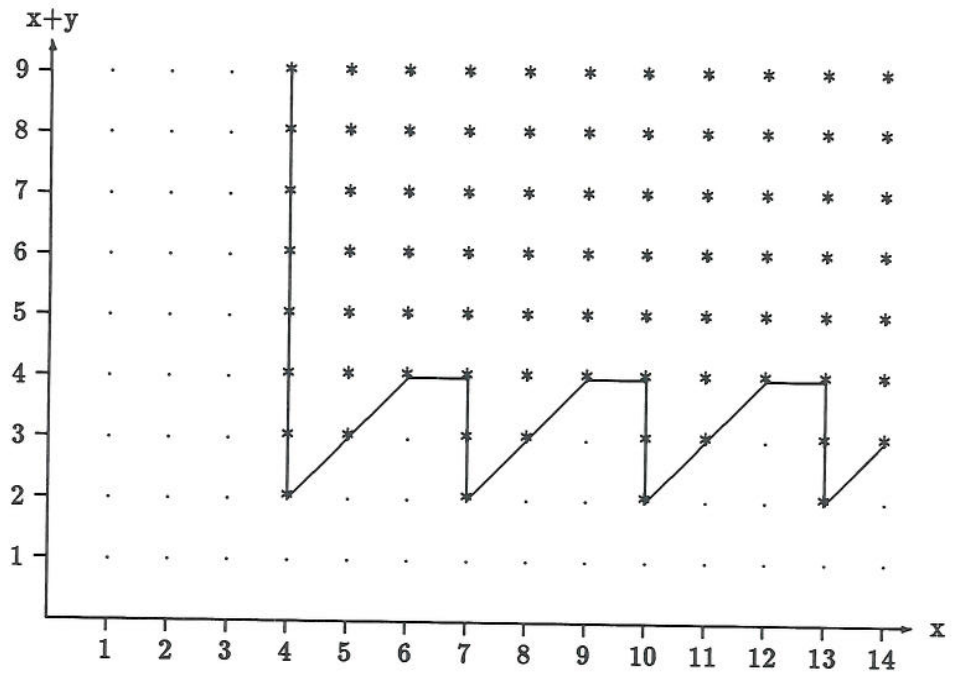


figure 4.4.2 bis

En fait, la "projection continue " est différente de la "projection entière " uniquement lorsque les coefficients de la variable selon laquelle on veut projeter ne vérifient aucune des trois conditions exposées dans la section suivante. Dans les autres cas l'élimination de la variable, par l'algorithme de Fourier, n'ajoute au domaine projeté aucun point entier.

4.4.3 Conditions suffisantes

Nous avons trouvé des conditions sur les variables du système qui permettent de savoir si la projection des points entiers du polyèdre est convexe et, donc, équivaut à l'ensemble des points entiers contenus dans la projection "continue" du polyèdre. Ces conditions nous ont permis de développer les algorithmes permettant de caractériser exactement l'ensemble des points entiers recherchés.

Nous avons le **théorème** suivant:

Soit

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

le système linéaire caractérisant un ensemble points entiers IP_s.
(les expressions A_j sont des expressions linéaires en nombres entiers
$A_j = \sum_{l=1, l \neq k}^{n} a_{jl}\, v_l$ *, les a_{jl} des entiers, v_l des variables du système,*
d_1 et d_2 des entiers positifs, k une variable du système)
et

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

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

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

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

La *projection entière* d'un polyèdre (comportant plusieurs paires d'inéquations contraignant k) selon la variable k est équivalente à la *projection continue* du polyèdre si toutes les paires d'inéquations contenant la variable k vérifient au moins l'une de ces trois conditions.

Démonstration:
Nous commençons par démontrer que tout point de IP_{sp} correspond bien à la projection entière d'un point de IP_s si d_1 ou d_2 vaut 1.

Tout point de IP_{sp} vérifie la contrainte:

$$d_1\ (A_2 - c_2) \ \leq \ \ d_2\ (-A_1 + c_1) \qquad (I)$$

- Pour $d_1 = 1$, nous avons:

$$A_2 - c_2 \ \leq \ \ d_2\ (-A_1 + c_1) \qquad (I)$$

A_1 et c_1 étant des expressions entières, il existe donc un entier $z = -A_1 + c_1$ tel que $A_2 - c_2 \ \leq \ \ d_2\ z$

Cet entier z vérifie les deux conditions:

$$\begin{cases} \ \ z \ \leq \ c_1 - A_1 \\ (A_2 - c_2) \ \leq \ d_2 z \end{cases}$$

Tout point de IP_{sp} correspond donc à la projection selon k d'au moins un point entier z de IP_s si d_1 vaut 1.

- Pour $d_2 = 1$, tout point de IP_{sp} doit vérifier:

$$d_1(A_2 - c_2) \ \leq \ \ -A_1 + c_1 \qquad (II)$$

A_2 et c_2 étant des expressions entières, il existe donc un entier $z = A_2 - c_2$ tel que $d_1 z \ \leq \ \ c_1 - A_1$

Cet entier z vérifie les deux conditions: $\begin{cases} d_1 z \ \leq \ \ c_1 - A_1 \\ (A_2 - c_2) \ \leq \ z \end{cases}$

Tout point de IP_{sp} correspond donc à la projection selon k d'au moins un point entier z de IP_s si d_2 vaut 1.

Montrons maintenant que si la troisième condition est vérifiée, tout point entier de IP_{sp} correspond à la projection entière d'un point de IP_s.

Si la troisième condition est vérifiée, la contrainte

$$d_1\ (A_2 - c_2) \ + \ d_1\ d_2 \ - \ d_1 \leq -d_2\ (A_1 - c_1) \quad (III)$$

l'est toujours. L'inégalité suivante l'est donc aussi (d_1 et d_2 sont positifs):

$$\frac{d_1\ (A_2 - c_2) \ + \ d_1\ d_2 \ - \ d_1}{d_1 d_2} \leq \frac{-d_2\ (A_1 - c_1)}{d_1 d_2}\ ^1 \quad (IV)$$

[1] Rappelons que dans toutes les démonstrations, les divisions sont des divisions entières, et que les opérations de transformation des expressions entières utilisées sont celles exposées en Annexe [§Annexe].

Il existe alors une valeur entière z telle que:

$$\frac{d_1\ (A_2 - c_2)\ +\ d_1\ d_2\ -\ d_1}{d_1 d_2} \leq z \leq \frac{-d_2\ (A_1 - c_1)}{d_1 d_2}$$

En utilisant les règles de réécriture des expressions entières décrites en [§Annexe], les inégalités sur la variable z se traduisent:

$$\begin{cases} d_1\ (A_2 - c_2)\ +\ d_1\ d_2\ -\ d_1 \leq d_1 d_2 z + d_1 d_2 - 1 \\ d_1 d_2 z \leq -d_2\ (A_1 - c_1) \end{cases} \Longrightarrow \begin{cases} d_1\ (A_2 - c_2)\ -\ d_1 + 1 \leq d_1 d_2 z \\ d_1 d_2 z \leq -d_2\ (A_1 - c_1) \end{cases}$$

En utilisant une nouvelle fois les règles de réécriture des expressions entières, on obtient:

$$\begin{cases} \frac{(d_1\ (A_2 - c_2) - d_1 + 1) + d_1 - 1}{d_1} \leq z d_2 \\ d_1 z \leq -(A_1 - c_1) \end{cases}$$

d'où

$$\begin{cases} (A_2 - c_2) \leq z d_2 \\ d_1 z \leq -(A_1 - c_1) \end{cases}$$

c.q.f.d.

4.4.4 Introduction des divisions entières

Si les trois conditions précédentes ne sont pas vérifiées, il est aussi parfois possible d'éliminer la variable du système, sans modifier la projection de IP_s, en introduisant des divisions entières.

Pour éliminer une variable entière k d'un système linéaire, sans modifier le résultat de la projection entière du système selon cette variable, il faut éliminer cette variable tout en respectant les règles de transformation des expressions entières. Ces règles introduisent des divisions entières dans les expressions.

Soit S un système linéaire:

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

(les expressions A_j sont des expressions linéaires en nombres entiers $A_j\ =\ \sum_{l=1, l \neq k}^{n} a_{jl}\ v_l$, les a_{jl} des entiers, v_l des variables du système, d_1 et d_2 des entiers positifs, k une variable du système)

L'utilisation des règles de réécriture des expressions entières nous permet de le mettre sous la forme:

$$(S)\ \begin{cases} \frac{A_2 - c_2 + d_2 - 1}{d_2} \leq k \\ k \leq \frac{(-A_1 + c_1)}{d_1} \end{cases} \Longleftrightarrow (S)\ \begin{cases} \frac{d_1\ (A_2 - c_2)\ +\ d_1\ d_2\ -\ d_1}{d_1 d_2} \leq k \\ k \leq \frac{d_2\ (-A_1 + c_1)}{d_1 d_2} \end{cases}$$

Après élimination de la variable entière k, on obtient l'inégalité:

$$\frac{d_1\ (A_2 - c_2)\ +\ d_1\ d_2\ -\ d_1}{d_1 d_2} \leq \frac{-d_2\ (A_1 - c_1)}{d_1 d_2}$$

Cette inégalité équivaut à la projection entière du système S selon la variable entière k.

Il est possible de déduire de cette inégalité, une contrainte sur une variable l du système, si cette variable n'apparaît que d'un côté de l'inégalité.

On obtient dans ce cas une inégalité:

(III) $A_l l \leq E_l$ où E_l est une fonction de A_1, A_2, c_1, c_2, d_1, *et* d_2 contenant une division entière.

Exemples:

Soit le système $S1 = \left\{ \begin{array}{r} -i + 3\ k\ \leq\ 3 \\ j\ -\ 3\ k\ \leq\ -2 \\ -j\ \leq\ 0 \end{array} \right. \iff S1 = \left\{ \begin{array}{r} j + 2 \leq\ 3\ k\ \leq\ 3 + i \quad (I) \\ -j\ \leq\ 0 \end{array} \right.$

L'inégalité (I) est aussi équivalente à (II) $\frac{j+4}{3} \leq \frac{3+i}{3}$ qui peut se réécrire sous la forme:

$j + 4 \leq\ 3(\frac{3+i}{3}) + 2$, soit $j \leq\ 3(\frac{3+i}{3}) - 2$

Prenons un autre système

$$S2 = \left\{ \begin{array}{r} -j\ +\ 3\ k\ \leq\ 3 \\ j\ -\ 3\ k\ \leq\ -2 \\ -j\ \leq\ 0 \end{array} \right. \iff S2 = \left\{ \begin{array}{r} j + 2 \leq\ 3\ k\ \leq\ 3 + j \quad (I) \\ -j\ \leq\ 0 \end{array} \right.$$

L'inégalité (I) est équivalente à (II) $\frac{j+4}{3} \leq \frac{3+j}{3}$ qui ne peut pas se traduire automatiquement sous la forme d'une contrainte sur j

Cette possibilité d'élimination est importante car elle va permettre d'éliminer certaines variables que l'on ne pouvait pas éliminer par projection simple, sans modifier l'ensemble de points entiers recherchés. Par contre, elle introduit des divisions entières dans les expressions des contraintes du système.

Ces divisions entières ne facilitent pas l'élimination des autres variables du système (ce n'est pas une opération interne). Il faut donc retarder leur introduction dans le système.

4.4.5 Algorithme de calcul de la projection entière d'un polyèdre

Nous présentons maintenant l'algorithme permettant d'obtenir le polyèdre résultant de la projection entière d'un système sur un espace de dimension inférieure. C'est à partir de ce dernier que l'on générera le code correspondant à l'ensemble des points entiers recherchés, lorsque la dimension du domaine image est inférieure à celle du domaine d'itération.

On suppose que l'on recherche la projection entière d'un système linéaire à m variables sur n variables. On appelle ces n variables, les variables de base et les $m - n$ autres, les variables hors-base.

On associe à chaque variable un indice N_i, en supposant que les indices associés aux variables hors-base à éliminer sont les plus forts. On associera le numéro N_1 à la variable de base la plus externe, dans le corps de boucles que l'on veut générer, N_2 à la suivante,... N_n à la variable de base la plus interne, N_{n+1} à la variable hors base la plus externe, ..., et N_m à la variable hors base la plus interne.

On note $S_{/k}$ le système linéaire obtenu après projection de la variable k.

1. **Pour chaque variable hors base k:**

   (a) **Séparer les contraintes de S contenant la variable k des autres. Soit R l'ensemble des contraintes ne contenant pas la variable k.**

   (b) **Poser** $S_{/k} := R$,

   (c) **Soit POS l'ensemble des contraintes** $\{(A_1 - c_1) + d_1 k \leq 0\}$ **où le coefficient de k est positif,**
   et NEG l'ensemble des contraintes $\{(A_2 - c_2) - d_2 k \leq 0\}$ **où il est négatif.**
   Combiner chaque paire d'inégalités $(pos \in POS, neg \in NEG)$**:**

      i. **Pour chaque paire d'inégalités telles que** $d_1 = 1$ **ou** $d_2 = 1$ **ou** $(d_1\ (A_2 - c_2) + d_1 d_2 - d_1 \leq -d_2(A_1 - c_1)$ **redondante pour** S**) faire:**

      $$S_{/k} := S_{/k} + \{d_1(A_2 - c_2) \leq -d_2(A_1 - c_1)\}$$

      ii. **Pour chaque paire d'inégalités telles que** $d_1 > 1$ **et** $d_2 > 1$ **et** $(d_1(A_2 - c_2) + d_1 d_2 - d_1 \leq -d_2(A_1 - c_1)$ **non redondante pour** S**) faire:**

      $$S_{/k} := S_{/k} + \{d_1(A_2 - c_2) \leq -d_2(A_1 - c_1)\} \ + \ \{pos, neg\}$$

   (d) **Poser** $S := S_{/k}$

2. **Associer à chaque contrainte du système un numéro de variable N_i:**

   Associer à chaque contrainte le numero de la variable, appartenant à la contrainte, et d'indice N_i le plus fort.

3. **Eliminer les contraintes redondantes du système:**

   (a) **Eliminer les contraintes redondantes en cherchant à éliminer en premier les contraintes associées à un indice N_i fort.**
   Conserver au moins 2 contraintes pour chacune des variables N_i du système (une contrainte où le coefficient de la variable N_i est positif pour la borne supérieure, une contrainte où il est négatif pour la borne inférieure).
   Eliminer toutes les contraintes redondantes associées à une variable hors base N_k si la variable n'apparaît pas dans les contraintes associées à un indice N_j, $j > k$

4. **Simplifier les contraintes contenant encore des variables hors base**

   (a) **Pour chaque paire d'inégalités** $((A_1 - c_1) + d_1 k \leq 0\ , (A_2 - c_2) - d_2 k \leq 0)$ **où** k **est une variable hors base n'apparaissant dans aucune des contraintes associées à un indice** $N_j, j > k$

      i. **Eliminer la variable** k **en combinant les 2 inégalités et en introduisant des divisions entières.**

      ii. **Associer à la nouvelle contrainte la variable, appartenant à la contrainte, d'indice** N_i **le plus fort.**

   (b) **Eliminer les contraintes redondantes du dernier système en exécutant la phase 3.**

L'introduction des divisions entières et la conservation de variables suplémentaires (selon lesquelles on voulait projeter) dans le système, traduisent la non convexité du domaine projeté. En effet, c'est lorsque les contraintes sur ces variables supplémentaires ne sont pas vérifiées, pour des valeurs possibles des autres variables du système linéaire, que le domaine projeté est non convexe.

Cet algorithme a la même complexité que l'algorithme de Fourier-Motzkin c'est-à-dire exponentielle.

4.4.6 Exemple

Soit le système de contraintes:

$$S = \begin{cases} -t_3 \leq -1 \\ t_3 \leq 10 \\ -t_1 - 3t_3 \leq -1 \\ t_1 + 3t_3 \leq 20 \\ -t_2 + 3t_3 \leq -1 \\ t_2 - 3t_3 \leq 30 \end{cases}$$

Voici les étapes de l'algorithme de projection entière du système S selon la variable t_3. Au cours de l'étape *1* les systèmes R, POS et NEG sont calculés:

$$R = \emptyset, \quad POS = \begin{cases} t_3 \leq 10 \\ t_1 + 3t_3 \leq 20 \\ -t_2 + 3t_3 \leq -1 \end{cases} \quad et\ NEG = \begin{cases} -t_3 \leq -1 \\ -t_1 - 3t_3 \leq -1 \\ t_2 - 3t_3 \leq 30 \end{cases}$$

Après l'étape *1.c.i* le système $S_{/t_3}$ vaut:

$$S_{/t_3} = \begin{cases} t_1 \leq 17 \\ -t_1 \leq 29 \\ -t_2 \leq -4 \\ t_2 \leq 60 \end{cases}$$

Après l'étape *1.c.ii* il vaut:

$$S_{/t_3} = \begin{cases} t_1 \leq 17 \\ -t_1 \leq 29 \\ -t_2 \leq -4 \\ t_2 \leq 60 \\ -t_1 - 3t_3 \leq -1 \\ t_1 + 3t_3 \leq 20 \\ -t_2 + 3t_3 \leq -1 \\ t_2 - 3t_3 \leq 30 \end{cases}$$

Au cours de l'étape *2* on associe à chaque contrainte le numéro N_i se trouvant à la droite des contraintes:

$$S_{/t_3} = \begin{cases} t_1 \leq 17 & 1 \\ -t_1 \leq 29 & 1 \\ -t_2 \leq -4 & 2 \\ t_2 \leq 60 & 2 \\ -t_1 - 3t_3 \leq -1 & 3 \\ t_1 + 3t_3 \leq 20 & 3 \\ -t_2 + 3t_3 \leq -1 & 3 \\ t_2 - 3t_3 \leq 30 & 3 \end{cases}$$

La phase *3* élimine les contraintes redondantes du système, on obtient ici le même système:

$$S_{/t_3} = \begin{cases} t_1 \leq 17 & 1 \\ -t_1 \leq 29 & 1 \\ -t_2 \leq -4 & 2 \\ t_2 \leq 60 & 2 \\ -t_1 - 3t_3 \leq -1 & 3 \\ t_1 + 3t_3 \leq 20 & 3 \\ -t_2 + 3t_3 \leq -1 & 3 \\ t_2 - 3t_3 \leq 30 & 3 \end{cases}$$

L'étape *4-a* permet de simplifier les contraintes contenant encore la variable t_3 de la manière suivante:

$$\begin{cases} -t_1 - 3t_3 \leq -1 \\ t_1 + 3t_3 \leq 20 \\ -t_2 + 3t_3 \leq -1 \\ t_2 - 3t_3 \leq 30 \end{cases} \iff \begin{cases} -t_1 + 1 \leq 3t_3 \leq t_2 - 1 \\ t_2 - 30 \leq 3t_3 \leq 20 - t_1 \end{cases}$$

$$\iff \begin{cases} 1 + 3(\frac{3-t_1}{3}) \leq t_2 \\ t_2 \leq 30 + 3(\frac{20-t_1}{3}) \end{cases}$$

La phase *4.b* élimine les contraintes redondantes du nouveau système (phase *3* de l'algorithme). Après calculs on obtient:

$$S_{/t_3} = \begin{cases} t_1 \leq 17 & 1 \\ -t_1 \leq 29 & 1 \\ -t_2 \leq -4 & 2 \\ t_2 \leq 60 & 2 \\ 1+3(\frac{3-t_1}{3}) \leq t_2 & 2 \\ t_2 \leq 30+3(\frac{20-t_1}{3}) & 2 \end{cases}$$

A partir de ce polyèdre nous déduirons un corps de boucles du type:

```
DO  t1 = -29,17
  DO  t2 = MAX(4,1+3×(3-t1)/3)) , MIN(60, 30+3×(20-t1)/3))
         .....
  ENDDO
ENDDO
```

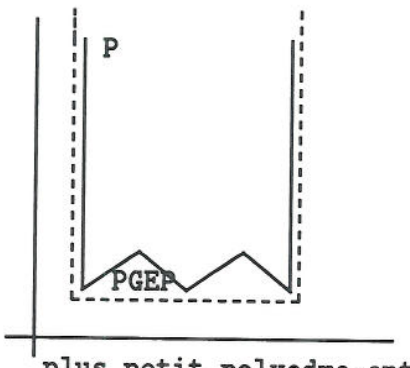
4.5 Décomposition des Z-polyèdres

Nous présentons dans cette section trois algorithmes qui permettent d'approximer les "Z-polyèdres non convexes" par des "Z-polyèdres convexes":

- le plus petit polyèdre convexe englobant un Z-polyèdre non convexe,
- le plus grand polyèdre convexe contenu dans un Z-polyèdre non convexe,
- le découpage d'un Z-polyèdre non convexe en polyèdres convexes.

4.5.1 Le plus petit polyèdre convexe englobant un Z-polyèdre non convexe.

Les ensembles de points entiers obtenus par projections entières successives d'un polyèdre selon un ensemble de variables peuvent être des Z-polyèdres non convexes.



plus petit polyedre-entier
contenant P.

Pour caractériser le plus petit polyèdre englobant cet ensemble de points nous utilisons l'algorithme de Fourier.

Proposition :

L'algorithme d'élimination d'une variable dans un système linéaire proposé par Fourier [Four24] (§4.4.1) permet de calculer l'enveloppe convexe de l'ensemble des projections des points entiers appartenant au polyèdre, à condition que ce polyèdre posséde des sommets aux coordonnées entières.

Les polyèdres dont nous recherchons la projection ont des sommets entiers par définition du domaine d'itération.

La démonstration de cette proposition a été présentée dans [LaMa88] par Lassez et Maher.

4.5.2 Le plus grand polyèdre convexe contenu dans un Z-polyèdre non convexe.

Les transformations affines appliquées aux polyèdres convexes peuvent générer des polyèdres non convexes. Cette non convexité est de deux types (voir §3.2.2). La première (polyèdre à *trous*) est caractérisée par une base non-unitaire du Z-module du Z-polyèdre. La seconde ne peut pas être caractérisée par le Z-polyèdre. C'est une non-convexité périodique au "bord du domaine".

Nous définissons le plus grand polyèdre contenu dans un Z-polyèdre non convexe comme étant le Z-polyèdre non-convexe auquel on ôte la partie non convexe au "bord du domaine". Dans la majorité des cas, ce polyèdre sera le plus grand polyèdre dont la projection entière est convexe à l'intérieur du Z-polyèdre non convexe.

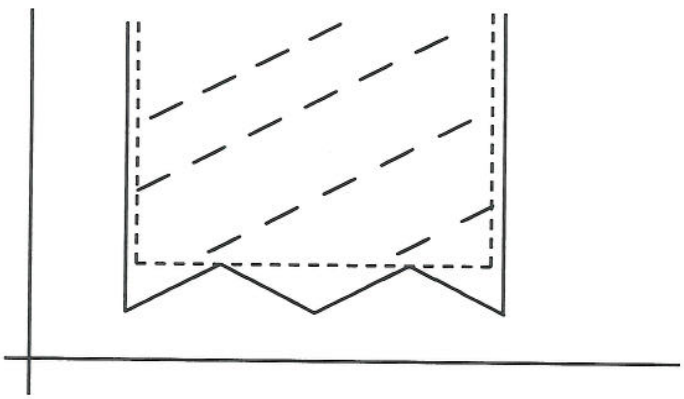


figure 4.5.2

Propositions

Soit S1 le système linéaire en nombres entiers définissant un polyèdre **P1**:

$$(S1) \quad \left\{ \begin{array}{rcll} A & \leq & kd_1 & (I) \\ kd_2 & \leq & B & (II) \end{array} \right.$$

où **A** et **B** sont des expressions linéaires en nombres entiers, **k** une variable entière du système, et d_1 et d_2 des entiers positifs supérieurs à 1.

Proposition 1:

Soit **P2** *le polyèdre défini par* **S2**:

$$(S2) \quad \left\{ \begin{array}{rcll} A & \leq & kd_1 & (I) \\ kd_2 & \leq & B & (II) \\ A.d_2 + d_1.d_2 - d_2 & \leq & B.d_1 & (III) \end{array} \right.$$

Le polyèdre **P2** *défini par* **S2** *est un polyèdre convexe, contenu dans* **P1** *défini par* **S1**, *dont la projection entière selon la variable* **k** *est convexe.*

Démonstration:

Le polyèdre **P2** est contenu dans **P1** par définition, puisqu'on ajoute une contrainte au système **S1**, pour définir **S2**.

Soit **P3** le polyèdre défini par **S3**:

$$(S3) \quad \left\{ A.d_2 + d_1.d_2 - d_2 \leq B.d_1 \quad (III) \right\}$$

et **PP2** le polyèdre résultant de la projection entière de **P2** selon la variable **k**:

$$(SS2) \quad \left\{ \begin{array}{rcll} \frac{A.d_2 + d_1.d_2 - d_2}{d_1 d_2} & \leq & \frac{B.d_1}{d_1 d_2} & (IV) \\ A.d_2 + d_1.d_2 - d_2 & \leq & B.d_1 & (III) \end{array} \right\}$$

P3 est le polyèdre correspondant à l'enveloppe convexe de **PP2**, obtenu par projection selon la variable **k**.

Montrons que **PP2** est bien un polyèdre convexe:

P3 étant l'enveloppe convexe de **PP2**, montrons que tous les points de **P3** appartiennent aussi à **PP2**. Montrons que (III) est vérifiée pour tout point de **PP2**.

$$(III) \quad A.d_2 + d_1.d_2 - d_2 \leq B.d_1 \quad \Longrightarrow \quad \frac{A.d_2 + d_1.d_2 - d_2}{d_1.d_2} \leq \frac{B.d_1}{d_1.d_2}$$
$$\Longleftrightarrow \quad (IV)$$

c.q.f.d.

Proposition 2:

Soit $\overline{P2}$ le polyèdre défini par $\overline{S2}$.

$$(\overline{S2}) \quad \left\{ \begin{array}{lcll} A & \leq & kd_1 & (I) \\ kd_2 & \leq & B & (II) \\ A.d_2 & \leq & B.d_1 \leq A.d_2 + d_1.d_2 - d_2 - 1 & (V) \end{array} \right.$$

L'ensemble $\overline{P2}$ défini par $\overline{S2}$ est un polyèdre, contenu dans **P1**, *dont la projection entière est non convexe.*

Démonstration:

Le polyèdre $\overline{P2}$ est contenu dans **P1** par définition, puisqu'on a ajouté une contrainte au système **S1**, pour définir $\overline{S2}$.

Soit **P4** le polyèdre défini par **S4**:

$$(S4) \quad \left\{ A.d_2 \leq B.d_1 \leq A.d_2 + d_1.d_2 - d_2 - 1 \quad (V) \right\}$$

et $\overline{PP2}$ le polyèdre résultant de la projection entière de $\overline{P2}$ selon la variable **k**:

$$(\overline{SS2}) \quad \left\{ \begin{array}{lcll} \frac{A.d_2+d_1.d_2-d_2}{d_1d_2} & \leq & \frac{B.d_1}{d_1d_2} & (VI) \\ A.d_2 & \leq & B.d_1 \leq A.d_2 + d_1.d_2 - d_2 - 1 & (V) \end{array} \right\}$$

P4 est le polyèdre correspondant à l'enveloppe convexe de $\overline{PP2}$, obtenue par projection selon la variable **k**.

Montrons que la projection entière de $\overline{P2}$ selon la variable **k** est non convexe; c'est-à-dire qu'il existe au moins un point **x** de **P4** qui n'appartient pas à $\overline{PP2}$.

La contrainte (V) est vérifiée par tous les points appartenant à **P4**.

$$(V) \quad \Longrightarrow \quad \frac{A.d_2}{d_1d_2} \leq \frac{B.d_1}{d_1d_2} \leq \frac{A.d_2+d_1.d_2-d_2-1}{d_1d_2} \qquad (VII)$$

A est une expression entière, posons $A = \alpha.d_1 + r \quad avec \quad 0 \leq r \leq d_1 - 1$.

Montrons qu'il existe des valeurs de A pour lesquelles (VI) n'est pas vérifiée lorsque (VII) l'est, i.e.:

$$\exists\, A \;/\; \frac{A.d_2}{d_1d_2} < \frac{A.d_2+d_1.d_2-d_2}{d_1d_2}$$

$$\Longleftrightarrow \quad \exists\ \alpha,\ r \;/\; \frac{(\alpha.d_1+r).d_2}{d_1d_2} < \frac{(\alpha.d_1+r).d_2+d_1.d_2-d_2}{d_1d_2}$$

$$\Longleftrightarrow \quad \exists\ \alpha,\ r \;/\; \frac{\alpha d_1d_2+rd_2}{d_1d_2} < \frac{(\alpha+1)d_1d_2+rd_2-d_2}{d_1d_2}$$

Cette contrainte est toujours vérifiée pour la valeur de $r = 1$. Il existe donc des points qui vérifient (VII) mais pas (VI). Nous en déduisons l'existence de points de **P4** qui n'appartiennent pas à la projection entière de $\overline{P2}$. La projection entière de $\overline{P2}$ est non convexe.

c.q.f.d.

Théorème:

P2 *est le plus grand polyèdre contenu dans le polyèdre non convexe, résultant de la projection de* **P1** *selon la variable* **k**.

Démonstration:

Rappelons **P1**, **P2** et $\overline{P2}$:

$$(S1) \quad \begin{cases} A & \leq & kd_1 & (I) \\ kd_2 & \leq & B & (II) \end{cases}$$

$$(S2) \quad \begin{cases} A & \leq & kd_1 & (I) \\ kd_2 & \leq & B & (II) \\ A.d_2 + d_1.d_2 - d_2 & \leq & B.d_1 & (III) \end{cases}$$

$$(\overline{S2}) \quad \begin{cases} A & \leq & kd_1 & (I) \\ kd_2 & \leq & B & (II) \\ A.d_2 & \leq & B.d_1 & \leq \quad A.d_2 + d_1.d_2 - d_2 - 1 \quad (V) \end{cases}$$

P1 est l'union des deux polyèdres **P2** et $\overline{P2}$ par définition.

Nous avons démontré pour la proposition 2 qu'il existait des points de **P4** qui n'appartenaient pas à la projection entière de $\overline{P2}$, donc pas à la projection entière de **P1**.

L'ensemble des points entiers de **P1** dont la projection entière est convexe est donc égal à l'ensemble (**P1** - $\overline{P2}$), soit **P2**.

Généralisation à un polyèdre à n contraintes:

Pour calculer le plus grand polyèdre contenu dans un Z-polyèdre non convexe, il faut conserver uniquement l'ensemble des points du Z-polyèdre dont la projection entière est convexe.

Algorithme de calcul du plus grand polyèdre convexe contenu dans le Z-polyèdre non convexe résultant de projections entières successives sur un polyèdre

On note PEC le polyèdre initial, et PGPEC le plus grand polyèdre contenu dans le Z-polyèdre PEP (Z-polyèdre " projeté ").

Pour chaque variable k selon laquelle on veut projeter en entier:

1. Séparer les contraintes de PEC contenant la variable k des autres. Soit R l'ensemble des contraintes ne contenant pas la variable k.
2. Poser $PEP = PEC$ et $PGPEC = R$,
3. Soit POS l'ensemble des contraintes $\{(A_i - c_i) + d_i k \leq 0\}$ où le coefficient de k est positif,

   et NEG l'ensemble des contraintes $\{(A_j - c_j) - d_j k \leq 0\}$ où il est négatif.

   Combiner chaque paire d'inégalités $(pos \in POS, neg \in NEG)$:

   (a) pour chaque paire d'inégalités telles que $|d_i| > 1$ et $|d_j| > 1$ et $(d_i\ (A_j - c_j) + d_i d_j - d_i \leq -d_j(A_i - c_i)$ non redondante pour $PEP)$ faire:
   PGPEC = PGPEC + $\{(A_j - c_j).d_i + d_j.d_i - d_i \leq (c_i - A_i).d_j\}$

   (b) pour chaque paire d'inégalités telles que $|d_i| = 1$ ou $|d_j| = 1$ ou $(d_i\ (A_j - c_j) + d_i d_j - d_i \leq -d_j(A_i - c_i)$ redondante pour $PEP)$ faire:
   PGPEC = PGPEC +$\{(A_j - c_j).d_i \leq (c_i - A_i).d_j\}$

PEP = PGPEC;

4.5.3 Décomposition d'un Z-polyèdre non convexe en polyèdres convexes.

Nous appelons, ici, découpage d'un Z-polyèdre non convexe en polyèdres, l'algorithme qui permet d'obtenir le résultat de la projection entière d'un polyèdre sous la forme de plusieurs polyèdres.

Remarquons qu'il n'existe pas de découpage unique d'un polyèdre non convexe en polyèdres convexes.

Exemples

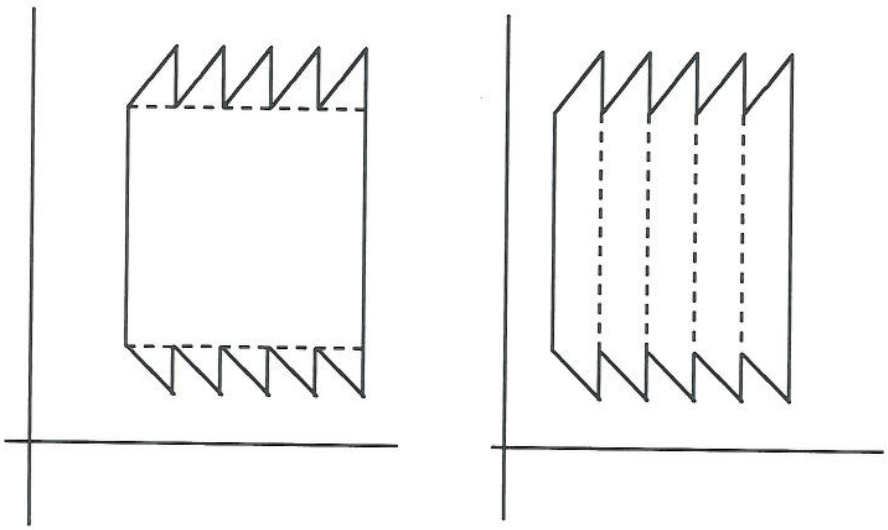


figure 4.5.3

La solution que nous proposons permet d'obtenir, dans la majorité des cas, un polyèdre relativement grand et plusieurs petits polyèdres (exemple de gauche).

Nous avons choisi de découper chaque projection de polyèdre de la manière suivante:

- un grand polyèdre : le plus grand polyèdre contenu dans le Z-polyèdre non convexe. Les détails de l'algorithme permettant de calculer cet ensemble ont été présentés dans la partie [§4.5.2].
- plusieurs groupes de polyèdres dont l'union forme un Z-polyèdre non convexe, résultant de la projection d'un polyèdre selon une variable.

Partitionnement d'un Z-polyèdre non convexe

Soit

$$(\overline{S2}) \quad \left\{ \begin{array}{lcll} A & \leq & kd_1 & (I) \\ kd_2 & \leq & B & (II) \\ A.d_2 & \leq & B.d_1 \leq & A.d_2 + d_1.d_2 - d_2 - 1 \quad (V) \end{array} \right.$$

un polyèdre dont la projection entière selon `k` est non convexe. Rappelons que d_1, d_2 sont des entiers positifs dont la valeur absolue est supérieure à 1,`k` est une variable du système, `A` et `B` des expressions entières.

Soit

$$P_\alpha = \left\{ A.d_2 \leq (k+\alpha).d_1.d_2 \leq B.d_1 \leq A.d_2 + d_1.d_2 - d_2 - 1 \right\}$$

Proposition 1:

Les P_α sont disjoints.

Démonstration:

Montrons que: $\forall\, x \in P_\alpha\ ,\ x \notin P_{\alpha'},\ \alpha \neq \alpha'$,

$$P_\alpha = \left\{ A.d_2 \leq (k+\alpha).d_1.d_2 \leq B.d_1 \leq A.d_2 + d_1.d_2 - d_2 - 1 \right\} \ (I)$$

$$P_{\alpha'} = \left\{ A.d_2 \leq (k+\alpha').d_1.d_2 \leq B.d_1 \leq A.d_2 + d_1.d_2 - d_2 - 1 \right\} \ (II)$$

Traitons le cas $\alpha' \geq \alpha + 1$:

$$(I) \implies A.d_2 \leq (k+\alpha).d_1.d_2 \quad (III)$$
$$(III)+(II) \iff (k+\alpha').d_1.d_2 \leq (k+\alpha).d_1.d_2 + d_1.d_2 - d_2 - 1 \ (VI)$$
$$(IV) \iff (\alpha' - \alpha - 1).d_1.d_2 \leq -d_2 - 1$$

Comme d_1, d_2, α et α' sont des entiers positifs ($\alpha' \geq \alpha + 1$), (IV) n'est jamais vérifiée. Pour $\alpha' \geq \alpha + 1$ les ensemble $P_{\alpha'}$ et P_α sont donc disjoints.

Traitons le cas $\alpha' \leq \alpha - 1$:

$$(II) \implies A.d_2 \leq (k+\alpha').d_1.d_2 \quad (V)$$
$$(I)+(V) \iff (k+\alpha).d_1.d_2 \leq (k+\alpha').d_1.d_2 + d_1.d_1 - d_2 - 1 \quad (VI)$$
$$(VI) \iff d_2 + 1 \leq d_1.d_2(\alpha' - \alpha + 1)$$

Comme $d_1, et\ d_2$ sont des entiers positifs, et α' un entier inférieur à $\alpha - 1$,(VI) n'est jamais vérifiée. Pour $\alpha' \leq \alpha - 1$ les ensemble $P_{\alpha'}$ et P_α sont aussi disjoints.

c.q.f.d.

Proposition 2:

si k est une constante égale à c, le polyèdre défini par:

$$P_c = \left\{ A.d_2 \leq c.d_1.d_2 \leq B.d_1 \leq A.d_2 + d_1.d_2 - d_2 - 1 \right\} \ (I)$$

est un polyèdre convexe.

Démonstration:

La projection entière de P_c selon c est égale à P_c, puisque c est une constante. Elle est donc convexe par définition de $\overline{S2}$.

Théorème:

Soit k_{min} et k_{max} les bornes inférieures et supérieures des valeurs prises par la variable k *dans le système linéaire en nombres entiers $\overline{S2}$.*

Les polyèdres P_c / $k_{min} \leq c \leq k_{max}$ forment une partition de $\overline{P2}$.

Démonstration:

Le polyèdre défini par $\overline{S2}$ est par définition égal à l'union des polyèdres P_c pour tout ($k_{min} \leq c \leq k_{max}$). Les P_c étant disjoints entre eux, ils forment une partition de $\overline{P2}$.

La projection entière du polyèdre $\overline{P2}$ défini par $\overline{S2}$ peut donc être découpée en $k_{max} - k_{min} + 1$ polyèdres P_c.

Algorithme de découpage d'un Z-polyèdre non convexe en polyèdres

Soit PECS l'ensemble des polyèdres $PECS_l$ constituant le Z-polyèdre non convexe, résultant de projections entières successives d'un polyèdre P selon un ensemble de variable k_l.

Avant toute projection entière, nous avons $PECS = P$.

Pour chaque variable k_l selon laquelle on veut projeter en entier, et pour chaque polyèdre $PECS_l$ appartenant à PECS, faire:

1. Séparer les contraintes de $PECS_l$ contenant la variable k_l des autres. Soit R l'ensemble des contraintes ne contenant pas la variable k_l.
2. Poser $PEP_l = R$,
3. Soit POS l'ensemble des contraintes $\{(A_i - c_i) + d_i k_l \leq 0\}$ où le coefficient de k_l est positif,

   et NEG l'ensemble des contraintes $\{(A_j - c_j) - d_j k_l \leq 0\}$ où il est négatif.

   Combiner chaque paire d'inégalités ($pos \in POS, neg \in NEG$):
4. pour chaque paire d'inégalités telles que $|d_i| = 1$ ou $|d_j| = 1$ ou $(d_i\ (A_j - c_j) + d_i d_j - d_i \leq -d_j(A_i - c_i)$ redondante pour $PECS_l$) calculer PEP_l:

   $PEP_l \quad = \quad PEP_l \quad + \quad \{(A_j - c_j).d_i \leq (c_i - A_i).d_j\}$
5. pour chaque paire d'inégalités telles que $|d_i| > 1$ et $|d_j| > 1$ et $(d_i\ (A_j - c_j) + d_i d_j - d_i \leq -d_j(A_i - c_i)$ non redondante pour $PECS_l$):

   (a) calculer le plus grand polyèdre convexe contenu dans $PECS_l$ après projection selon la variable k_l:

   $$PEP_{l0} \quad = \quad PEP_l + \{(A_j - c_j).d_i + d_j.d_i - d_i \quad \leq \quad (c_i - A_i).d_j\}$$

   (b) calculer le polyèdre non convexe $\overline{PEP_{l_{i,j}}}$ résultant du complémentaire de PEP_{l0} dans PEP_l après élimination de k_l dans les deux inégalités.

$$\overline{PEP_{li,j}} = PEP_l + \{(A_j - c_j).d_i \leq (c_i - A_i).d_j \leq (A_j - c_j).d_i + d_j.d_i - d_i - 1\}$$

Calcul des polyèdres convexes contituant $\overline{PEP_{li,j}}$:

i. **Calculer k_{lmin} et k_{lmax} par projection '' continue'' du polyèdre $\overline{PEP_{li,j}}$ selon toutes les autres variables du système.**

$$\overline{PEP_{li,j}} = \bigcup_{m=1}^{k_{lmax} - k_{lmin} + 1} PEP_{li,j,m}$$

où $PEP_{li,j,m}$ est convexe et défini par:

$$PEP_{li,j,m} = PEP_l + \left\{ \begin{array}{r} (A_j - c_j).d_i \leq (m + k_{lmin} - 1).d_j.d_i \leq \\ (c_i - A_i).d_j \leq (A_j - c_j).d_i + d_j.d_i - d_i - 1 \end{array} \right\}$$

4.6 Calcul du nombre de points entiers contenus dans un polyèdre.

Pour connaître le nombre des éléments référencés par un tableau dans un corps de boucles et le nombre des éléments que l'on a choisi de transférer de la mémoire globale vers les mémoires locales, nous avons besoin de calculer le nombre de points entiers contenus dans un polyèdre.

Le calcul du volume d'un polyèdre convexe quelconque étant très complexe [DyFr88], nous chercherons seulement une bonne approximation de ces deux volumes.

K. Gallivan, W. Jalby et D. Gannon proposent dans [GJGa88] une méthode qui permet de calculer le volume de l'image d'un polyèdre par une transformation linéaire.

Cette méthode est inspirée du calcul du volume d'un polyèdre $P' = B.P$ de même dimension que P pour lequel $vol(P') = |det(B)|.vol(P)$.

Elle permet de calculer le volume d'un polyèdre $P' = B.P$ lorsque la dimension du polyèdre image est inférieure ou égale à celle du polyèdre initial P, et que l'on sait évaluer le volume du polyèdre initial.

Soit P' un polyèdre tel que $P' = B.P$. Si P' est de dimension s, on exprime les vecteurs de base p_i de chaque face de P de dimension s en fonction des vecteurs de base p'_i de P'.

$$p_i = \sum_{i=1}^{s} \alpha_{is} p'_i$$

le volume de l'image de chacune de ces faces peut être approximé par:

$$|det(A)|.vol(face_i)$$

où A est la matrice dont les coefficients sont les α_{is}, et $vol(face_i)$ le volume de la face de P de dimension s considérée.

Le volume total de l'image de P peut être approximé par:

$$\sum_{i \in F} |det(A_i)|.vol(face_i)$$

où F est l'ensemble des $\begin{pmatrix} k \\ s \end{pmatrix}$ faces de dimension s de P.

Cette méthode suppose que l'on sache calculer facilement le volume du polyèdre initial (multiplication des intervalles des valeurs prises par les vecteurs de base). Or le calcul de ce volume n'est pas souvent simple. En effet, le polyèdre initial est défini par un ensemble de contraintes sur ces vecteurs de base, chaque vecteur de base peut être contraint par plusieurs (> 2) contraintes, chacune de ces contraintes peut être une combinaison linéaire des autres vecteurs de base, etc...

Nous utiliserons donc cette méthode, dans le même cadre que [GJGa88], pour calculer un approximation du nombre des éléments référencés par un tableau dans un corps de boucles et lorsque l'on peut facilement calculer le volume du domaine d'itération (polyèdre initial), c'est-à-dire lorsque ce domaine est un parallélépipède. Dans les autres cas nous utiliserons la méthode proposée par N. Tawbi dans [Tawb90].

Cette dernière découpe les polyèdres en un ensemble de polyèdres pour lesquels le calcul du volume est relativement simple: chaque vecteur de base est contraint par uniquement deux contraintes, les contraintes sont linéaires, ... Enfin, le calcul du nombre des éléments appartenant aux polyèdres résultant de la décomposition se traduit sous forme de sommations et utilise les nombres de Bernouilli.

Nous utiliserons cet algorithme pour calculer le volume des éléments référencés par un tableau dans un corps de boucles lorsque l'on ne sait pas calculer directement le volume du domaine d'itération (polyèdre initial) ainsi que pour évaluer le volume des éléments à transférer, qui ne dépend pas directement du polyèdre initial.

4.7 Conclusion

Tous les algorithmes que nous venons de présenter sont utilisés pour caractériser les ensembles de données référencées par une (ou des) fonction d'accès aux éléments d'un tableau dans un corps de boucles.

Ces ensembles n'étant pas toujours convexes, nous avons proposé un algorithme pour chacune des opérations permettant de les approximer par des ensembles convexes et/ ou de les décomposer en plusieurs ensembles convexes.

Ces ensembles pourront ainsi toujours être représentés par des polyèdres, principalement des systèmes linéaires en nombres entiers.

La non convexité des ensembles référencés résulte de projections successives de polyèdres. Nous avons trouvé des conditions suffisantes pour savoir si l'opération de projection des points entiers du polyèdre est exacte. Ces conditions sont à la base de nos algorithmes.

Dans la majorité des cas, on construit le polyèdre recherché en ajoutant des contraintes au système linéaire initial pour le diviser ou conserver certaines propriétés (convexité ou non-convexité) du polyèdre. Ces opérations conduisent rapidement à des systèmes linéaires de taille très importante. Nous utiliserons donc des algorithmes de simplification des

systèmes linéaires: la normalisation et l'élimination des contraintes redondantes qui permettent de réduire considérablement leur taille. Ces algorithmes doivent être utilisés après chaque élimination d'une des variables du système, de manière à conserver des systèmes de dimension acceptable.

L'ensemble des programmes servant à la génération du code de transfert des données a été écrit en langage C. Ils appartiennent à la bibliothèque C3 "polylib" regroupant des algorithmes de manipulation des matrices et des polyèdres développés à l'IRISA par l'équipe de P.Quinton, à l'Université Pierre et Marie Curie par F.Feautrier, et à l'Ecole des Mines de Paris par l'équipe du projet PIPS.

Chapitre 5

Génération de code: Base de parcours, Déclarations, Modifications des références

Nous présentons dans ce chapitre et dans les deux suivants les algorithmes que nous avons développés pour générer le code de transfert des éléments référencés par un (ou des) tableau dans une tâche parallèle.

Rappelons que nous supposons qu'il est possible de distinguer les phases de calculs, qui s'effectuent uniquement avec la mémoire locale, et les phases de transferts entre la mémoire globale et les mémoires locales. Les tâches sont supposées ne pas comporter de synchronisation interne.

Dans un premier temps, nous supposons que chaque tâche ne contient que des affectations et des boucles DO à bornes affines (i.e. ni CALLs, ni IFs). Les références sont aussi supposées affines.

La passe que nous avons étudiée doit transformer le code des tâches en un code de sous-programme contenant: des déclarations de variables locales (chapitre 5), le code de transfert des données de la mémoire globale vers la mémoire locale du processeur où est exécutée la tâche (chapitre 6), le code de calcul, et le code de transfert des résultats de la mémoire locale vers la mémoire globale (chapitre 6).

Nous présentons dans le dernier chapitre consacré à la génération de code (chapitre 7) les extensions des hypothèses faites auparavant.

Dans ce chapitre nous développons les premières étapes de la génération de code:

- le choix d'une base de parcours des éléments référencés par T,
- la déclaration des tableaux locaux,
- l'algorithme de modification des références aux tableaux en des références aux tableaux locaux.

Nous exposons dans la première partie les notations utilisées tout au long de cette étude. Le choix d'une nouvelle base de parcours des éléments référencés par le tableau T est importante puisqu'elle va permettre de minimiser le nombre des transferts. C'est l'objet

de la deuxième partie de ce chapitre. Les troisième et quatrième parties définissent les fonctions d'accès aux éléments du tableau T, référencés en mémoire globale, et aux éléments du tableau local TL en fonction de la base de parcours précédemment calculée. Dans la cinquième partie, nous exposons les techniques envisagées pour calculer les dimensions des tableaux locaux. Enfin, dans la dernière, nous présentons la méthode de transformation des références aux tableaux en des références aux tableaux locaux.

5.1 Quelques notations:

On exprime l'espace d'itération d'un ensemble de boucles "DO" par un polyèdre convexe $A.\vec{j} <= \vec{\alpha}$ où A est une matrice, $\vec{j}$ une des itérations du corps de boucles, et $\vec{\alpha}$ un vecteur.

On note φ la fonction d'accès d'une référence à un tableau T dans le corps de boucles. On a $\vec{\varphi} = F . \vec{j} + \vec{f_0}$ où F est la matrice représentant la fonction d'accès φ dans $\vec{j}$ et $\vec{f_0}$ la partie constante de la fonction d'accès à la référence.

Par exemple, si on considére le corps de boucles:

```
DO I = 1,10
  DO J = 1,20
    T(2 × I + J − 5,3 × J + 1) = ...
  ENDDO
ENDDO
```

la matrice A, le vecteur $\vec{\alpha}$ et les matrices associées à F et f_0 s'écrivent:

$$A = \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \quad , \quad \vec{\alpha} = \begin{pmatrix} -1 \\ 10 \\ -2 \\ 20 \end{pmatrix}$$

et

$$F = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \quad , \quad \vec{f_0} = \begin{pmatrix} -5 \\ 1 \end{pmatrix}$$

On note F_i le i-ième indice du tableau T, c'est à dire celui qui correspond à la i-ième ligne de la matrice F et du vecteur $\vec{f_0}$.

Rappelons le type de code transfert que l'on veut générer et les relations qu'il implique entre les tableaux, adressés en mémoire globale, et les tableaux locaux.

On veut obtenir, pour le code de transfert de la mémoire globale vers la mémoire locale, un nid de boucles permettant de copier l'ensemble des éléments référencés par un tableau T dans un tableau TL local à la mémoire locale.

Ce nid de boucles sera de la forme:

```
DO t1 = N, M
  DO t2 = f21(t1), f22(t1)
    ...
     DO tn = fn1(t1, t2, ..., tn-1), fn2(t1, t2, ..., tn-1)
          TL(φ'(t1, t2, ..., tn)) = T (φ(t1, t2, ..., tn))
     ENDDO
    ...
  ENDDO
ENDDO
```

où $\vec{t}$ représente une base de vecteurs permettant de parcourir l'ensemble des éléments référencés par le tableau T et les fonctions $f_{ij}(t_1, t_2, ..., t_n)$ des fonctions affines des indices de boucle $t_1, t_2, ..., t_n$. φ est la fonction d'accès aux éléments du tableau T, accessibles en mémoire globale, et φ' la fonction d'accès aux éléments du tableau local TL.

De même, pour le code de recopie nous aurons un code du type:

```
DO t1 = N, M
  DO t2 = f21(t1), f22(t1)
    ...
     DO tn = fn1(t1, t2, ..., tn-1), fn2(t1, t2, ..., tn-1)
          T(φ(t1, t2, ..., tn)) = TL (φ'(t1, t2, ..., tn))
     ENDDO
    ...
  ENDDO
ENDDO
```

où $\vec{t}$ représente une base de vecteurs permettant de parcourir l'ensemble des éléments modifiés au cours de l'exécution.

5.2 Choix d'une base de parcours des données

On appelle **domaine image** l'ensemble des éléments référencés par un tableau T dans un corps de boucles. Rappelons que pour référencer de façon minimale (sans références multiples [§3.2.2]) l'ensemble des éléments de ce domaine, il est important de trouver une base les définissant. Ses éléments appartiennent tous au Z-module qui caractérise le domaine. Pour trouver une base permettant de parcourir tous ces éléments, il suffit donc de trouver une base caractérisant ce Z-module.

La partie constante de la fonction φ d'accès aux éléments du tableau est indépendante du Z-module, on recherche donc une base définissant le Z-module $L(F)$. Les propositions 4.1.3.1 et 4.1.3.2 montrent que la forme réduite de Hermite associée à la matrice F permet de trouver une telle base. C'est cette dernière que nous utiliserons, dans la majorité des cas.

Rappelons [§4.1.3] que pour toute matrice F, il existe deux matrices unimodulaires P et Q telles que:

$$P \,.\, F \,.\, Q \;=\; \begin{pmatrix} L & 0 \\ S & 0 \end{pmatrix} \;=\; H$$

où H est la forme réduite de Hermite associée à F de rang r, L une matrice triangulaire inférieure de dimension $r \times r$, S une matrice de dimension $n - r \times r$, et P une matrice de permutation de dimension $n \times n$.

La fonction d'accès aux éléments du tableau s'écrit dans cette base:
$\varphi = P^{-1} H Q^{-1} \vec{j} + \vec{f}_0$.

5.2.1 Optimisation de la base pour des transferts contigus

En Fortran, les éléments des tableaux sont stockés d'abord selon le premier indice du tableau. Pour privilégier, lors des transferts, des accès contigus aux éléments du tableau [§1.1.1], il faut que l'un des vecteurs de base du domaine image soit colinéaire au premier indice F_1 du tableau et que les autres vecteurs soient linéairement indépendants de ce dernier. On utilisera ce vecteur comme indice de la boucle la plus interne dans le code transfert.

Exemple

Prenons pour exemple le code de calcul suivant:

```
DO I = 1,10
  DO J = 1,20
    T(2 × I + J ,3 × J + 1) = ...
  ENDDO
ENDDO
```

Pour privilégier des accès contigus aux éléments du tableau T, il faut trouver un vecteur de base colinéaire au vecteur $2\vec{i} + \vec{j}$. Soit $\vec{t}$ la nouvelle base du domaine image et $\vec{t_2}$ ce vecteur vérifiant $\alpha\ \vec{t_2} = 2\vec{i} + \vec{j}$. La nouvelle fonction d'accès aux éléments du tableau est de la forme:
$\vec{\varphi} = (\alpha \times t_2\ , \varphi_2(t_1, t_2))$ et le code de transfert se traduira sous la forme suivante:

```
DO t1= ...
  DO t2= ...
    TRANSFERER   T (α × t2, φ2(t1, t2))
  ENDDO
ENDDO
```

Si $\varphi_2(t_1, t_2)$ est linéairement indépendant de t_2 et si α vaut 1 tous les transferts se feront de manière contiguë, si $\alpha \neq 1$ ils se feront avec un pas de α unités (il est alors aussi possible les vectoriser sur certains multiprocesseurs).

La forme réduite de Hermite associée à φ permet d'obtenir une base caractérisant les éléments référencés par le tableau, cette base ne favorise pas nécessairement les possibilités de transferts contigus. En fait, elle les favorise lorsque la matrice $P^{-1}H$ est diagonale. Soient α_i les coefficients de cette diagonale, la fonction d'accès aux éléments du tableau se traduit alors: $\varphi = \begin{pmatrix} \alpha_1 t_1 + f_1 \\ \alpha_2 t_2 + f_2 \\ \dots \\ \alpha_n t_n + f_n \end{pmatrix}$ et le code de transfert correspondant aura la forme:

```
DO t_n = N, M
  DO t_{n-1} = f_21(t_n), f_22(t_n)
    ...
        DO t_1 = f_n1(t_n, t_{n-1}, ..., t_2), f_n2(t_n, t_{n-1}, ..., t_2)
          T(α_1 × t_1 + f_1, α_2 × t_2 + f_2, ..., α_n × t_n + f_n)
        ENDDO
    ...
  ENDDO
ENDDO
```

Les autres formes canoniques telles que la *forme normale de Hermite* et les *formes réduite et normale de Smith* permettent d'obtenir aussi la matrice F en fonction d'une matrice de base et de deux matrices P et Q unimodulaires. La matrice P n'est pas forcément une matrice de permutation, les indices F_i peuvent donc être des combinaisons linéaires des vecteurs de bases. Ces autres formes canoniques n'apportent donc pas plus de possibilités de transferts contigus.

Les solutions que nous proposons maintenant permettent de calculer une base favorisant les transferts contigus lorsque la matrice $P^{-1}H$ est non diagonale et lorsque les éléments référencés sont contigus. Nous distinguons deux cas: le cas où la dimension du domaine image est égale à celle du tableau et le cas où elle est inférieure.

1. Lorsque $rang(F) = dim(T) = n$ il faut trouver n vecteurs de base. La base constituée de l'ensemble des indices F_i est valide (les n vecteurs sont linéairement indépendants et on conserve F_1 comme vecteur de base).

   La base caractérisant le Z-module des éléments référencés se traduit dans cette nouvelle base par une matrice H_m diagonale dont les éléments diagonaux h_{ii} valent: $h_{ii} = pgcd(F_{i1}, F_{i2}, ..., F_{in})$. La fonction φ d'accès aux éléments du tableau s'écrit:

$$\varphi = F\vec{j} + \vec{f_0} = P_m^{-1} H_m Q_m^{-1} \vec{j} + \vec{f_0} = Id \begin{pmatrix} h_{11} & 0 & 0 & 0 & 0 \\ 0 & h_{22} & 0 & 0 & 0 \\ \dots & & & & \\ 0 & \dots & h_{ii} & \dots & 0 \\ \dots & & & & \\ 0 & 0 & 0 & 0 & h_{nn} \end{pmatrix} \begin{pmatrix} \frac{F_1}{h_{11}} \\ \frac{F_2}{h_{22}} \\ \dots \\ \frac{F_i}{h_{ii}} \\ \dots \\ \frac{F_n}{h_{nn}} \end{pmatrix} \vec{j} + \vec{f}$$

Exemple

Voici un exemple où il est préférable de choisir une autre base que celle de Hermite associée à F et où $dim(\varphi) = dim(T)$.

Considérons la référence au tableau $T(6i + 3k, j + k)$. En prenant la forme de Hermite H associée à F on obtiendrait:

$$\begin{pmatrix} 6i & + & 3k \\ j & + & k \end{pmatrix} = P^{-1}HQ^{-1}\vec{j}+\vec{f_0} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 3 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}\begin{pmatrix} 2 & 0 & 1 \\ -2 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}\vec{j} + \vec{f_0}$$

le deuxième vecteur de base du Z-module $t_1 + t_2$ n'est pas indépendant du premier.

Il est préférable de prendre la base $\vec{t} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}\vec{j}$

Soit: $\begin{pmatrix} 6i & + & 3k \\ j & + & k \end{pmatrix} = H_m Q_m^{-1}\vec{j} + \vec{f_0} = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}\vec{j} + \vec{f_0}$

où les deux vecteurs de base du Z-module ($3 \times t_1$ et t_2) sont indépendants.

2. Lorsque $r = rang(F) < dim(T)$ il faut trouver r vecteurs de base linéairement indépendants et conserver le premier indice F_1 comme vecteur de base afin de garder des possibilités de transferts contigus.

   La nouvelle base peut être construite par ajouts successifs des indices F_i dans la base, si l'indice F_i n'est pas combinaison linéaire des vecteurs de base déjà choisis. Soit $\vec{t}$ cette base.

   La fonction d'accès aux éléments se décompose alors sous la forme suivante:

   $\varphi = P_m^{-1}H_m Q_m^{-1}\vec{j} + \vec{f_0}$ où les r premières lignes de $Q_m^{-1}\vec{j}$ correspondent à la base $\vec{t}$ choisie, les r premières lignes de H_m à la base du Z-module (selon la base $\vec{t} = Q_m^{-1}\vec{j}$) et les $n - r$ dernières lignes de H_m aux coefficients de multiplicité des indices en fonction des vecteurs de base. Notons que H_m est une matrice de Hermite particulière associée à φ et que la matrice P_m est une matrice de permutation.

   Exemple

   Voici un exemple où il est aussi préférable de choisir une autre base que la forme réduite de Hermite lorsque $dim(\varphi) < dim(T)$.

   Considérons la référence au tableau $T(i + j, 2i + 2j, 3i + 3k)$. En prenant la base suivante déduite de H on obtient un vecteur de base $3 \times t_1 - 3 \times t_2$ dépendant du premier.

$$\begin{pmatrix} i+j \\ 2i+2j \\ 3i+3k \end{pmatrix} = P^{-1}HQ^{-1}\vec{j}+\vec{f_0} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 3 & -3 & 0 \\ 2 & 0 & 0 \end{pmatrix}\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}\vec{j} + \vec{f_0}$$

   En prenant la base $\vec{t} = Q_m^{-1}\vec{j} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}\vec{j}$

on obtient des vecteurs indépendants:

$$\begin{pmatrix} i+j \\ 2i+2j \\ 3i+3k \end{pmatrix}_{\vec{f_0}} = P_m^{-1} H_m Q_m^{-1} \vec{j} + \vec{f_0} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \vec{j} +$$

Pour faciliter la compréhension et la discussion des méthodes présentées dans les sections et chapitres suivants, liées à la base de parcours dont nous venons d'exposer les principes de calcul, nous considérons que la nouvelle fonction d'accès aux éléments du tableau est de la forme: $\varphi = P^{-1}HQ^{-1}\vec{j} + \vec{f_0}$ où H est la forme réduite de Hermite associée à F. Lorsque la matrice $P^{-1}H$ n'est pas diagonale, il faudra remplacer les matrices P, H, Q par les matrices particulières P_m, H_m, Q_m présentées précédemment, qui correspondent à des formes de Hermite particulières.

5.3 Nouvelle fonction d'accès aux éléments du tableau global

Pour référencer les éléments des tableaux se trouvant en mémoire globale, il est important de minimiser les accès à cette mémoire. Nous avons donc choisi d'exprimer ces éléments en fonction de l'une des bases du Z-module qui les caractérise. Soit $\varphi = P^{-1}HQ^{-1}\vec{j} + \vec{f_0} = P^{-1}H\vec{t} + \vec{f_0}$

Les r premières colonnes de la matrice H représentent une base du Z-module $L(F)$ [§4.1.3.1 et 4.1.3.2].

Notons B cette base et $\vec{t}$ le vecteur correspondant au changement de base $\vec{j} = Q\ \vec{t}$

La matrice Q étant unimodulaire, à tout point entier du domaine d'itération $\vec{j}$ correspond bien un point entier dans la nouvelle base $\vec{t} = Q^{-1} \cdot \vec{j}$

La fonction d'accès aux références du tableau T s'écrit dans cette base:

$\vec{\varphi} = F \cdot \vec{j} + \vec{f_0}$

$\vec{\varphi} = P^{-1} \cdot H \cdot Q^{-1}\vec{j} + \vec{f_0} = P^{-1} \cdot H \cdot \vec{t} + \vec{f_0}$

$\vec{\varphi} = P^{-1} \cdot [B\ 0] \cdot \vec{t} + \vec{f_0}$.

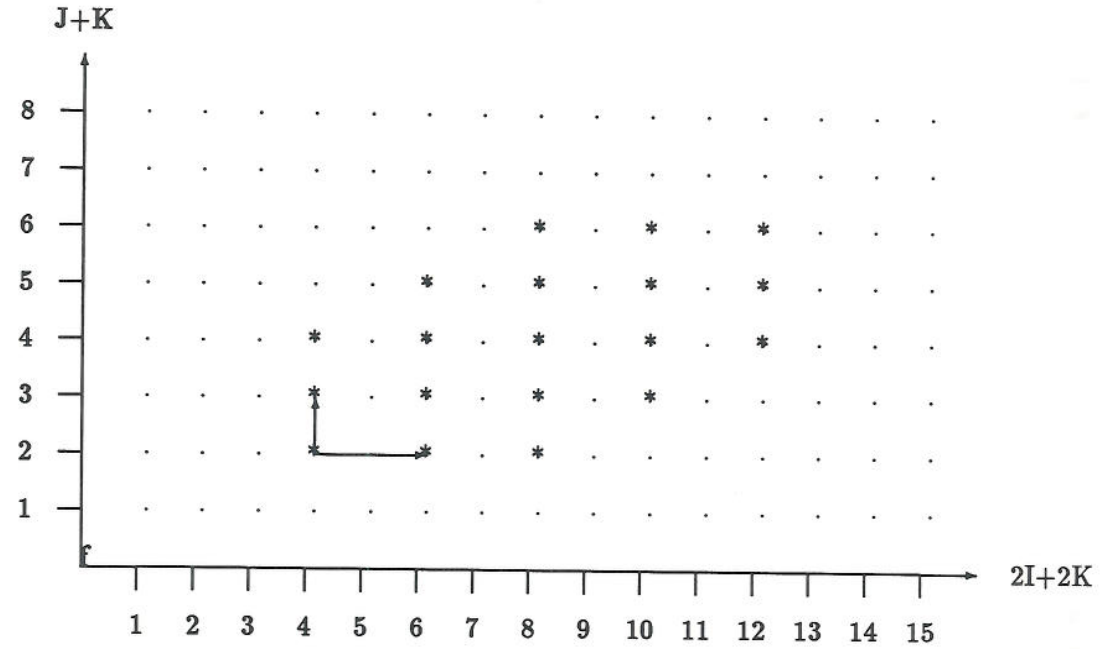
La matrice B étant de rang r, seuls les r premiers indices du vecteur $\vec{t}$ servent à exprimer la fonction d'accès aux éléments de T.

5.3.1 Exemple

Prenons le nouvel exemple:

```
DO I = 1,3
  DO J = 1,3
    DO K = 1,3
    T(2 × I + 2 × K, J + K) = ...
  ENDDO
ENDDO
```

les éléments référencés par T dans le corps de boucles sont représentés sur la figure suivante:



$$F = \begin{pmatrix} 2 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix} = P^{-1} H Q^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

La fonction d'accès s'exprime:

$$\vec{\varphi} = P^{-1} H \vec{t} + \vec{f}_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2t_1 \\ t_2 \\ 0 \end{pmatrix}$$

avec $\vec{t} = \begin{pmatrix} i+k \\ j+k \\ k \end{pmatrix}$

La base $H \vec{t}$ caractérise bien le Z-module des éléments référencés par le tableau. Les codes de transfert seront du type:

```
DO t2 = N, M
  DO t1 = f21(t1), f22(t1)
    TL(φ'(t1, t2)) = T(2 × t1, t2)
  ENDDO
ENDDO
```

Ici la fonction $\vec{\varphi}$ d'accès aux éléments du tableau T accède à tous les éléments du tableau avec un pas de 2 unités.

5.4 Nouvelle fonction d'accès aux éléments du tableau local

Pour référencer les éléments d'un tableau TL local, la solution la plus simple est de prendre la même fonction que celle choisie pour adresser le tableau T en mémoire globale. Cette solution conduit à déclarer un tableau local TL de même dimension que le tableau T et à générer un code de transfert de la mémoire globale vers la mémoire locale du type:

```
DO t1 = N, M
  DO t2 = f21(t1), f22(t1)
    ...
     DO tn = fn1(t1, t2, ..., tn-1), fn2(t1, t2, ..., tn-1)
          TL(φ(t1, t2, ..., tn)) = T (φ(t1, t2, ..., tn))
     ENDDO
    ...
  ENDDO
ENDDO
```

et le code inverse pour le code de recopie.

Cependant, lorsque la dimension du tableau T est supérieure à la dimension du domaine image, il est préférable d'utiliser une autre fonction d'accès aux éléments du tableau TL afin de minimiser l'espace alloué en mémoire locale. En effet, si le domaine image est de dimension r inférieure à celle du tableau T, un tableau local de dimension r suffit pour contenir tous les éléments de ce domaine.

On le constate sur l'exemple simple suivant:

```
DO I = 1,10
  DO J = 1,20
    T(I + J ,I + J ) = ...
  ENDDO
ENDDO
```

où la dimension du tableau T est 2, alors que le rang de la fonction F est 1.

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = P^{-1} H Q^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

La fonction d'accès aux éléments de T en mémoire globale est après calculs [§5.3] de la forme:

$\vec{\varphi} = (t_1, t_1)$ avec $t_1 = i + j$

En prenant la même fonction pour le tableau local, on serait obligé de déclarer un tableau local de dimension 2 alors qu'un tableau de dimension 1 suffit, puisque seuls les éléments diagonaux du tableau T sont référencés.

5.4.1 Fonction d'accès aux éléments du tableau local lorsque $dim(\varphi) < dim(T)$

La seule transformation bijective permettant de passer d'une fonction de dimension n caractérisant un ensemble de points entiers de dimension r à une fonction de dimension r caractérisant ces même points est (quand il est possible) un changement de base unimodulaire [Irig87].

Nous proposons le changement de base suivant:

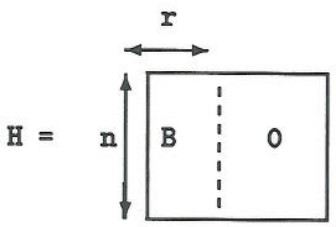
Proposition

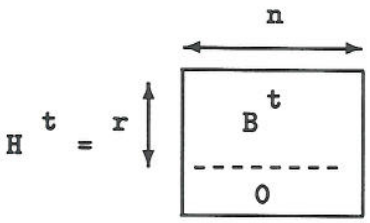
Soit $\vec{\varphi} = P^{-1}H\vec{t} + \vec{f_0}$ la fonction d'accès aux éléments du tableau T de dimension n, et $r < n$ la dimension du domaine image référencé par T. En prémultipliant la fonction φ par une matrice Q'^t unimodulaire on obtient une nouvelle fonction d'accès à ces éléments de dimension r. Cette matrice Q' est définie par $(P^{-1}H)^t = P'^{-1}H'Q'^{-1}$ où H' est la forme réduite de Hermite associée à la matrice $(P^{-1}H)^t$.

Démonstration

Le tableau T étant de dimension n et le domaine image de dimension r, la fonction d'accès aux éléments du tableau s'exprime sous la forme:

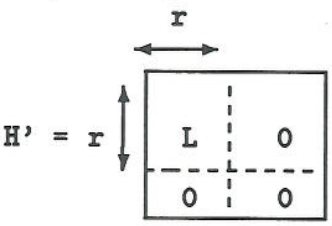
$\vec{\varphi} = P^{-1}H\vec{t} + \vec{f_0}$ où la matrice H a la forme:

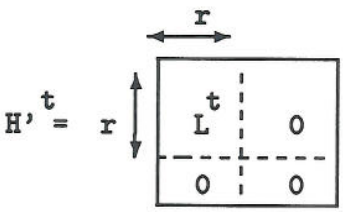




La matrice P étant une matrice de permutation, la matrice $(P^{-1}H)^t$ est une matrice à coefficients entiers et il est possible de l'exprimer en fonction de matrices P', H' *et* Q' telles que:

$(P^{-1}H)^t = P'^{-1}H'Q'^{-1}$ où H' est la forme réduite de Hermite associée à la matrice $(P^{-1}H)^t$ de rang r et de la forme:





La fonction d'accès aux éléments du tableau s'écrit alors:

$\vec{\varphi} = F\vec{j} + \vec{f_0} = P^{-1}H\vec{t} + \vec{f_0}$

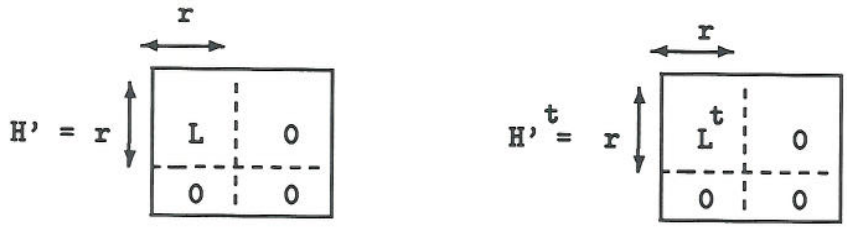
$= ((P^{-1}H)^t)^t\vec{t} + \vec{f_0} = (P'^{-1}H'Q'^{-1})^t\vec{t} + \vec{f_0} = Q'^{-1t}H'^tP'^{-1t}\vec{t} + \vec{f_0}$

$= Q'^{-1t}H'^tP'\vec{t} + \vec{f_0}$ car P' est une matrice de permutation.

En prémultipliant cette fonction par $(Q'^{-1t})^{-1}$, soit Q'^{t} on obtient une nouvelle fonction d'accès linéaire de la forme:

$\vec{\varphi'} = Q'^{t}\vec{\varphi} = H'\ {}^{t}P'\vec{t} + Q'^{t}\vec{f_0}$

Cette fonction est de dimension r, puisque H' est de la forme:



et que P' est une matrice de permutation.

La matrice Q' étant unimodulaire, à tout point entier du domaine défini par φ correspond bien un point entier dans la nouvelle base et inversement. Il existe donc une bijection entre les éléments référencés par la fonction $\vec{\varphi}$ et ceux référencés par $\vec{\varphi'}$.

En Résumé:

Lorsque la dimension du domaine image est égale à la dimension du tableau T, la nouvelle fonction d'accès aux éléments du tableau local est identique à celle du tableau T adressé en mémoire globale:

$\vec{\varphi'} = \vec{\varphi} = F\vec{j} + \vec{f_0} = P^{-1}H\vec{t} + \vec{f_0}$ avec $\vec{t} = Q^{-1} \cdot \vec{j}$ et $F = P^{-1}HQ^{-1}$ où H est la forme réduite de Hermite associée à F.

Lorsque la dimension du domaine image est inférieure à celle du tableau, elle est définie par:

$\vec{\varphi'} = Q'^{t}\vec{\varphi} = H'\ {}^{t}P'\vec{t} + Q'^{t}\vec{f_0}$ avec $(P^{-1}\ H)^t = P'^{-1}\ H'\ Q'^{-1}$ où H' est la forme réduite de Hermite associée à la matrice $(P^{-1}\ H)^t$.

5.4.2 Exemple

Reprenons l'exemple précédent:

```
DO I = 1,10
  DO J = 1,20
    T(I + J ,I + J ) = ...
  ENDDO
ENDDO
```

$$\text{où} \quad F = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = P^{-1}\ H\ Q^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}\ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

La matrice $(P^{-1}H)^t$ se réécrit en fonction de P', H', Q':

$$(P^{-1}\ H)^t = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} = P'^{-1}\ H'\ Q'^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}\ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

d'où

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = (P'^{-1} H' Q'^{-1})^t Q^{-1} = (\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix})^t \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$= Q'^{-1t} H'^t P'^t Q^{-1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^t \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}^t \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}^t \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$= Q'^{-1t} H'^t P'^t Q^{-1} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

En prémultipliant φ par la matrice $Q'^t = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$ on obtient:

$$\vec{\varphi'} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\vec{j} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\vec{f_0}$$

$$= \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}\vec{t} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad avec \quad \vec{t} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\vec{j}$$

d'où $\vec{\varphi'} = (t_1, 0)$

5.4.3 Présence de plusieurs références au tableau

Lorsque le code de calcul contient plusieurs références au même tableau, il faut déterminer les fonctions d'accès aux éléments du tableau local commun aux différentes références. Ces fonctions dépendent des domaines image définis par les différentes références. Nous distinguons plusieurs cas:

- l'un des domaines image est de dimension égale à celle du tableau,
- les domaines image ont des Z-modules identiques,
- les domaines images ont des Z-modules inclus,
- les domaines image ont des Z-modules différents

A tout tableau T on fait correspondre un seul tableau local TL. Même si les domaines référencés par les différentes références sont disjoints, on utilise le même tableau local pour stocker ces ensembles disjoints . Cette hypothèse est nécessaire pour pouvoir effectuer des transferts regroupés d'ensembles contigus et disjoints.

5.4.3.1 Une des références au tableau T définit un domaine image de dimension égale à celle du tableau

Dans ce cas, la dimension du tableau local doit être de dimension égale à celle du tableau T afin de pouvoir enregistrer l'ensemble des données référencées. Toutes les fonctions d'accès φ'_i au tableau local sont donc identiques aux fonctions d'accès φ_i du tableau T.

Pour chaque référence φ_i au tableau T, nous aurons:

$\vec{\varphi'_i} = \vec{\varphi_i} = F_i\vec{j_i} + \vec{f_{0i}} = P_i^{-1}H_i\vec{t_i} + \vec{f_{0i}}$

5.4.3.2 Les domaines images ont des Z-modules identiques de dimension inférieure à celle du tableau

Dans ce cas, l'ensemble des éléments référencés par chacune des fonctions φ_i d'accès au tableau T est de même dimension $r < dim(T)$. Les fonctions φ_i possédent un Z-module TI identique de dimension r et de base $P^{-1}H$.

Chaque fonction φ_i s'écrit: $\varphi_i = F_i\vec{j_i} + \vec{f_{0i}} = P^{-1}H\vec{t_i} + \vec{f_{0i}}$ où $P^{-1}H$ est la base du Z-module caractérisant φ_i selon la base $\vec{t_i}$.

La fonction d'accès linéaire de dimension r permettant de référencer l'ensemble des éléments référencés par φ_i est: $\vec{\varphi'_i} = Q'^t\vec{\varphi_i} = H'\,{}^tP'\vec{t_i} + Q'^t\,\vec{f_{0i}}$ où Q' est définie par $(P^{-1}\,H)^t = P'^{-1}\,H'\,Q'^{-1}$ et H' la forme réduite de Hermite associée à la matrice $(P^{-1}\,H)^t$.

Comme les bases $P^{-1}H$ des Z-modules sont identiques, à tout élément du tableau T correspond bien un seul élément du tableau local: $T(\varphi_1) = T(\varphi 2) \Longleftrightarrow TL(\varphi'_1) = TL(\varphi' 2)$.

Car $\varphi_1 = \varphi 2 \iff P^{-1}HQ_1^{-1}\vec{j_1} + \vec{f_{01}} = P^{-1}HQ_2^{-1}\vec{j_2} + \vec{f_{02}}$

$\iff Q'^tP^{-1}HQ_1^{-1}\vec{j_1} + Q'^t\vec{f_{01}} = Q'^tP^{-1}HQ_2^{-1}\vec{j_2} + Q'^t\vec{f_{02}}$

$\iff H'\,{}^tP'\vec{t_1} + Q'^t\,\vec{f_{01}} = H'\,{}^tP'\vec{t_2} + Q'^t\,\vec{f_{02}}$

$\iff \varphi'_1 = \varphi' 2$

La matrice $H'\,{}^tP'$ étant de rang r, les $n-r$ composantes de la fonction φ'_i correspondent aux $n-r$ dernières constantes de $Q'^t\,\vec{f_{0i}}$.

Lorsque ces constantes sont égales pour toutes les références au tableau, on utilisera un tableau local de dimension r pour stocker l'ensemble des données référencées.

Dans les autres cas, le tableau devra être de dimension supérieure afin de pouvoir contenir l'ensemble des données référencées par les différentes fonctions d'accès au tableau. Ces cas correspondent à des domaines image de même dimension et de même Z-module, mais disjoints.

Exemples

Prenons les deux références au tableau T:

$$T(i+j+2, i+j+1) \;\; et \;\; T(i+j+1, i+j)$$

On note $\varphi_1 = (i+j+2, i+j+1)$ et $\varphi_2 = (i+j+1, i+j)$

$$\vec{\varphi_1} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = P_1^{-1}\,H_1\,Q_1^{-1}\,\vec{j_1} + \vec{f_{01}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\vec{j_1} + \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

$$\vec{\varphi_2} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = P_2^{-1} H_2 Q_2^{-1} \vec{j_2} + \vec{f_{02}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \vec{j_2} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

φ_1 et φ_2 ont le même Z-module de même base $P^{-1}H$. Les domaines référencés sont tous deux de dimension 1 inférieure à celle du tableau.

Après calculs des fonctions d'accès aux éléments du tableau local, on obtient

$$Q'^t \varphi_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \vec{t_1} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \quad et \quad Q'^t \varphi_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \vec{t_2} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

avec $\vec{t_2} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \vec{j_2}$ *et* $\vec{t_1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \vec{j_1}$

d'où $\vec{\varphi_1'} = (t_1 + 2, -1)$ et $\vec{\varphi_2'} = (t_1 + 1, -1)$

Les constantes étant identiques, le tableau local pourra être de dimension 1.

Le code de transfert aura la forme:

```
C
C     Transfert en memoire locale des elements references par φ1
C
      DO t1 = N, M
         TL(t1 + 2) = T(t1 + 2, t1 + 1)
      ENDDO
C
C     Transfert en memoire locale des elements references par φ2
C
      DO t1 = L, K
         TL(t1 + 1) = T(t1 + 1, t1)
      ENDDO
```

En fait, comme ces deux ensembles de données à transférer ne sont pas disjoints, nos algorithmes généreront un seul code de transfert pour l'ensemble de ces éléments référencés. C'est l'objet de la partie [§6.2.1] de cette étude.

Prenons maintenant les deux références au tableau T:

$$T(i+j+4, i+j+1) \quad et \quad T(i+j+1, i+j+1)$$

On note $\varphi_1 = (i+j+4, i+j+1)$ et $\varphi_2 = (i+j+1, i+j+1)$

$$\vec{\varphi_1} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = P_1^{-1} H_1 Q_1^{-1} \vec{j_1} + \vec{f_{01}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \vec{j_1} + \begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

$$\vec{\varphi_2} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = P_2^{-1} H_2 Q_2^{-1} \vec{j_2} + \vec{f_{02}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \vec{j_2} + \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

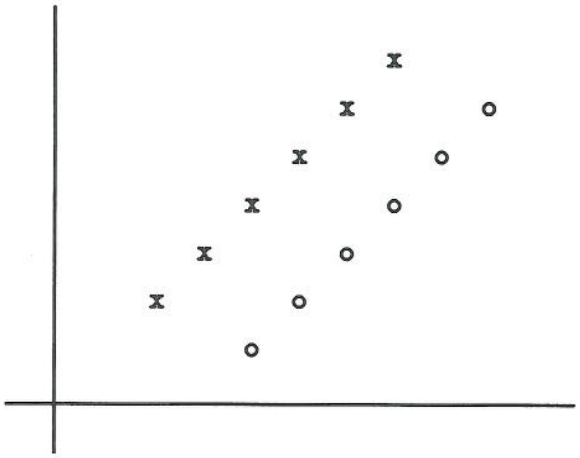
φ_1 et φ_2 ont le même Z-module. Les domaines référencés sont tous deux de dimension 1 inférieure à celle du tableau. On recherche donc les fonctions d'accès aux éléments de TL:

$$Q'^t \varphi_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \vec{t_1} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 4 \\ 1 \end{pmatrix} \quad et \quad Q'^t \varphi_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \vec{t_2} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

avec $\vec{t_2} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \vec{j_2}$ et $\vec{t_1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \vec{j_1}$

d'où $\vec{\varphi_1'} = (t_1 + 4, -3)$ et $\vec{\varphi_2'} = (t_1 + 1, 0)$

Ici on ne pourra pas utiliser un tableau local de dimension $r = 1$ car les $2 - 1$ dernières composantes des fonctions sont différentes. Elles attestent de la disjonction des deux ensembles référencés par φ_1 et φ_2, de même dimension et de même Z-module. Nous le constatons sur la figure suivante:



Les points entiers référencés par φ_1 sont représentés par "o" et ceux de φ_2 par des "x".

Le code de transfert aura la forme:

```
C
C     Transfert en memoire locale des elements references par φ1
C
      DO t1 = N, M
          TL(t1 + 4, -3) = T(t1 + 4, t1 + 1)
      ENDDO
C
C     Transfert en memoire locale des elements references par φ2
C
      DO t1 = L, K
          TL(t1 + 1, 0) = T(t1 + 1, t1 + 1)
      ENDDO
```

5.4.3.3 Les domaines image ont des Z-modules inclus de dimensions inférieures à celle du tableau

On suppose ici que tous les domaines image référencés par les différentes fonctions φ_i d'accès au tableau T ont un Z-module commun: le Z-module dans lequel ils sont inclus. Soit TI ce Z-module de base $P^{-1}H$ et de dimension r. Les Z-modules T_i de base $P_i^{-1}H_i$ inclus dans TI vérifient: $P_i^{-1}H_i = k_i^t P^{-1}H$ où $\vec{k_i}$ est un vecteur dont les composantes sont entières et supérieures ou égales à 1.

Chacune des fonctions φ_i s'écrit:

$$\varphi_i = F\vec{j_i} + \vec{f_{0i}} \;=\; k_i^t \; P^{-1}H\vec{t_i} + \vec{f_{0i}}$$

Soit $Q\,'^t$ la matrice de changement de base permettant de passer d'une fonction de dimension n caractérisant les éléments du Z-module TI à une fonction de dimension r caractérisant les même éléments.

Tous les éléments des Z-modules T_i appartenant aussi au Z-module TI, le changement de base permettant de référencer les éléments de T_i à l'aide d'une fonction de dimension r est identique à celui de TI, soit $Q\,'^t$.

Les fonctions d'accès aux éléments du tableau local se traduiront:

$\vec{\varphi_i'} = Q'^t\vec{\varphi_i} \;=\; Q\,'^t k_i^t P^{-1}H\vec{t_i} + Q'^t\,\vec{f_{0i}}$

Les $n - r$ composantes de la fonction φ_i' correspondent aux $n - r$ dernières constantes de $Q'^t\,\vec{f_{0i}}$.

Lorsque ces constantes sont égales pour toutes les références au tableau, on utilisera un tableau local de dimension r pour stocker l'ensemble des données référencées.

Dans les autres cas, le tableau devra être de dimension supérieure afin de pouvoir contenir l'ensemble des données référencées par les différentes fonctions d'accès au tableau. Ces cas correspondent à des domaines image de même dimension dont les Z-modules sont inclus, mais disjoints.

5.4.3.4. Les domaines ont des Z-modules différents, non inclus

Dans ce cas, il faut trouver un tableau de dimension minimale permettant de stocker ces ensembles dont les Z-modules sont différents. Il faut donc calculer le Z-module de dimension minimale caractérisant tous les éléments référencés. Ce Z-module aura pour dimension au minimum la dimension du plus grand Z-module *plus* 1 (sinon il serait le Z-module incluant les autres). Dans la majorité des cas, on trouvera alors un Z-module de dimension égale à celle du tableau. Le calcul exact de ce Z-module étant complexe, nous traitons ce cas de façon pessimiste et nous prenons les fonctions d'accès au tableau local identiques à celles du tableau T.

5.5 Génération des déclarations des tableaux locaux

Le nombre d'éléments référencés par un tableau T dans un corps de boucles, qui appartiennent à une tâche parallèle, est souvent inférieur au nombre total d'éléments du tableau. L'espace occupé par le tableau local TL servant à stocker ces données en mémoire locale doit le traduire.

Le langage Fortran accepte comme type de déclaration des dimensions des tableaux, uniquement des entiers ou des constantes ou encore des expressions linéaires d'entiers et de constantes.

Le calcul des bornes inférieure ou supérieure des éléments référencés du tableau T peut dépendre de constantes symboliques, contenues dans les expressions des bornes du domaine d'itération. Nous proposons de calculer ces bornes par la méthode présentée au §5.5.1.

Une fois ces bornes calculées, il nous suffit de définir deux constantes par indice du tableau T, une pour la borne inférieure et une pour la borne supérieure. On attribue à chaque constante les valeurs symboliques calculées. Puis on définit les tableaux locaux avec ces dimensions de type CONSTANTE.

5.5.1 Calcul des bornes minimales et maximales des indices d'un tableau

Reprenons les notations de [§5.1], nous disposons d'un domaine d'itération: $A.\vec{j} \leq \vec{\alpha}$, d'une fonction φ caractérisant les indices du tableau T: $\vec{\varphi} = P^{-1}H\vec{t} + \vec{f_0}$ et d'une fonction φ' caractérisant les indices du tableau TL identique à la précédente ou égale à: $Q'^t\vec{\varphi} = Q'^tP^{-1}H\vec{t} + Q'^t\vec{f_0}$. Les fonctions d'accès aux éléments du tableau local dépendent essentiellement des valeurs prises par les vecteurs de base $\vec{t}$ puisqu'elles s'expriment sous la forme de combinaisons linéaires de ces vecteurs.

Nous présentons donc maintenant les méthodes permettant de calculer les bornes inférieure et supérieure des valeurs référencées par le vecteur de base $\vec{t}$. Nous distinguons le cas où la dimension du domaine image est égale à celle du domaine d'itération de celui où elle est inférieure.

cas où $Dim(\varphi) = Dim(\vec{j})$

Lorsque la dimension du domaine image est égale à la dimension de $\vec{j}$, il est possible de calculer la matrice F^{-1} et donc d'effectuer le changement de base $\vec{j} = F^{-1}.\vec{t}$ sur le domaine d'itération.

Cette transformation nous donne les contraintes du domaine en fonction des vecteurs de base générant les éléments du tableau. Soit, $A.F^{-1}.\vec{t} \leq \vec{\alpha}$.

Nous calculons, dans ce cas, les bornes minimale et maximale de chaque indice du tableau t_i par projections successives de ce nouveau domaine sur l'ensemble des variables du système exceptée t_i.

cas où $Dim(\varphi) < Dim(\vec{j})$

Lorsque la dimension du domaine image est inférieure à celle de $\vec{j}$, on ne peut pas calculer F^{-1}. Nous pouvons trouver une approximation majorant ces bornes en:

- calculant les bornes minimale et maximale de chacune des variables de base j_i par projections successives du domaine sur toutes les autres variables du système,
- en déduisant une valeur approximative des bornes des indices du tableau qui sont des combinaisons linéaires de ces variables.

Mais il est préférable de trouver une évaluation exacte de ces bornes.

Les algorithmes classiques de programmation linéaire en nombres entiers, permettant de calculer le minimum ou le maximum d'une fonction linéaire dans un système (tel que l'algorithme des congruences décroissantes de Gondran [Min83] utilisant les coupes de Gomory) ne permettent pas de résoudre les programmes linéaires comportant des constantes symboliques. Or les expressions des bornes des boucles de nombreux codes de calcul contiennent des constantes symboliques. Nous utiliserons donc, pour un calcul exact des bornes minimale et maximale des indices du tableau T, l'algorithme paramétrique de résolution de systèmes d'inéquations linéaires en nombres entiers proposé par P.Feautrier [Feau88b].

Optimisation pour une seule référence

Lorsque le code de calcul ne possède qu'une seule référence au tableau T, il est parfois possible de réduire encore la taille des tableaux locaux. Il faut alors tenir compte de la fonction d'accès aux éléments du tableau local choisie.

Lorsqu'elle est *identique* à celle du tableau T: $\vec{\varphi'} = P^{-1}H\vec{t} + \vec{f_0}$, nous pouvons réduire la taille du tableau local en divisant les valeurs de chacune de ces bornes par le facteur de multiplicité des vecteurs directeurs de $P^{-1}H$.

En effet, soit $H' = P^{-1} \,.\, H$. Chaque indice F_i du tableau est un multiple du p.g.c.d. des coefficients $H'_{i,1}$, $H'_{i,2}$, ..., $H'_{i,r}$, que nous noterons p_i. Nous pouvons donc diviser les bornes calculées précédemment par chaque coefficient p_i correspondant.

De même, lorsque la fonction d'accès aux éléments de TL vaut: $\vec{\varphi'} = Q'^t\vec{\varphi} = H'\,{}^tP'\vec{t} + Q'^t\vec{f_0}$. Il est possible de réduire la taille des tableaux locaux en divisant les valeurs de chacune de ces bornes par le facteur de multiplicité de H'^t.

Présence de plusieurs références au tableau

Le calcul des bornes inférieure et supérieure des éléments du tableau local dépend, dans ce cas, de l'ensemble des plages de valeurs référencées par les différentes fonctions d'accès à T. La première étape du calcul consiste donc à calculer ces bornes pour chacune des références au tableau. La seconde, à calculer ces bornes pour l'ensemble des éléments référencés, c'est à dire à calculer le minimum des bornes inférieures et le maximum des bornes supérieures.

5.5.2 Exemples

Soit le code de calcul suivant:

```
DO I = 1,N
  DO J = 1,M
    DO K = 1,L
         T(2*I+2*J,3*K)  ..
    ENDDO
  ENDDO
ENDDO
```

Nous avons:

$$F \,.\, \vec{j} + \vec{f_0} = \begin{pmatrix} 2 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \,.\, \vec{j} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \,.\, \vec{t} = \begin{pmatrix} 2t_1 \\ 3t_3 \end{pmatrix}$$

avec $$\vec{t} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \,.\, \vec{j}$$

Sur cet exemple, nous constatons bien que le premier indice du tableau F_1 est multiple de 2, et que le second est multiple de 3.

Les bornes inférieure et supérieure des indices du vecteur $\vec{t}$ sont:

$$2 \leq t_1 \leq N+M \quad et \quad 1 \leq t_3 \leq L \quad \Longrightarrow \quad 4 \leq 2t_1 \leq 2N+2M \quad ,3 \leq 3t_3 \leq 3L$$

Ces bornes peuvent être réduites par un facteur 2 pour les bornes du premier indice du tableau et par 3 pour celles du second. Nous générerons les déclarations suivantes:

```
PARAMETER(C11 = 4/2,C12 = (2*N+2*M)/2,C21 = 3/3,C22 = (3*L)/3)

DIMENSION TL(C11:C12,C21:C22)
```

Prenons un autre code de calcul:

```
DO I = 1,10
  DO J = 1,20
    T(3 × I + 3 × J +1,3 × I + 3 × J +1) = ...
  ENDDO
ENDDO
```

où $F = \begin{pmatrix} 3 & 3 \\ 1 & 1 \end{pmatrix} = P^{-1} H Q^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 3 & 0 \\ 3 & 0 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$

$$F = \begin{pmatrix} 3 & 3 \\ 1 & 1 \end{pmatrix} = Q'^{-1t} H'^{t} P' Q^{-1} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 3 & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

En prémultipliant φ par la matrice $Q'^{t} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$ on obtient:

$$\vec{\varphi'} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 3 & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\vec{j} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\vec{f}_0$$

$$= \begin{pmatrix} 3 & 0 \\ 0 & 0 \end{pmatrix}\vec{t} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad avec \quad \vec{t} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\vec{j}$$

d'où $\vec{\varphi'} = (3t_1 + 1, 0)$

Les bornes inférieure et supérieure de t_1 référencées dans le corps de boucles se définissent par:

$2 \leq t_1 \leq 30 \implies 7 \leq 3t_1 + 1 \leq 91$

Ici encore elles peuvent être réduites par le facteur 3 de H'. On générera les déclarations suivantes:

```
PARAMETER(C11=7/3,C12=91/3)

DIMENSION TL(C11:C12)
```

5.6 Modification des références

La transformation des références au tableau T en des références locales au tableau TL est assez simple. Ces transformations dépendent de la fonction d'accès au tableau local choisie précédemment [§5.4] et des déclarations des tableaux locaux.

La fonction d'accès aux éléments du tableau T est définie par:

$$\vec{\varphi} = F\vec{j} + \vec{f_0} \;=\; P^{-1}H\vec{i} + \vec{f_0}$$

La fonction φ' d'accès aux éléments du tableau local TL est celle déterminée au paragraphe [§5.4] .

Le type des codes de transfert sera en général:

```
DO t1 = N,M
  DO t2 = f21(t1), f22(t1)
    ...
      DO tn = fn1(t1,t2,...,tn-1), fn2(t1,t2,...,tn-1)
          TL(φ'(t1,t2,...,tn)) = T (φ(t1,t2,...,tn))
      ENDDO
    ...
  ENDDO
ENDDO
```

pour le code de transfert de la mémoire globale vers la mémoire locale, et le code inverse pour le code de recopie.

Les références au tableau T dans le code de calcul seront modifiées de la même manière en références au tableau local TL. .

Cependant si les bornes inférieure et supérieure des éléments référencés ont été divisées par un coefficient p_i lors de la génération des déclarations des tableaux locaux, il faut réduire les références aux éléments locaux de la même manière.

Nous faisons donc correspondre à chaque élément référencé par la fonction d'accès φ au tableau T l'élément correspondant dans le tableau local dont la fonction d'accès est $(\varphi'_1/p_1, \varphi'_2/p_2, ..., \varphi'_n/p_n)$ où les p_i sont les coefficients calculés lors de la génération des déclarations du tableau local TL.

5.6.1 Exemples

Reprenons le code de calcul de l'exemple donné au [§6.1.7].:

```
DO I = 1,N
  DO J = 1,M
     DO K = 1,L
          T(2 × I + 2 × J, 3 × K)   ..
     ENDDO
  ENDDO
ENDDO
```

Chapitre 6

Génération du code de transfert

Nous commençons, dans ce chapitre, par présenter les algorithmes qui permettent de générer automatiquement le code de transfert des données modifiées par la tâche de la mémoire locale vers la mémoire globale.

Nous présentons ces algorithmes avant ceux utilisés pour le code de copie de la mémoire globale vers la locale pour les raisons suivantes:

- pour générer le code de recopie, il faut caractériser l'ensemble **exact** des données **modifiées** par la tâche s'exécutant sur le processeur, pour conserver la cohérence des données en mémoire globale.
- pour générer le code de copie, il n'est pas utile de copier l'ensemble exact de ces données, on peut en copier davantage.
- il nous semble plus facile de commencer par spécifier un ensemble **exact** de points et d'exposer les contraintes engendrées par cette spécification avant de chercher à l'approximer afin d'éliminer ces contraintes.
- sur une machine où les communications sont très coûteuses, un code de copie **exact** peut être préférable à un code de copie par **excès**.

Pour minimiser le nombre de transferts, on a trouvé une base de parcours de l'ensemble des éléments référencés par la tâche [§5.2]. Il faut maintenant rechercher toutes les caractéristiques de cet ensemble dans cette nouvelle base.

Nous présentons les algorithmes que nous avons développés pour générer automatiquement le code de transfert des éléments référencés par une, puis deux, et enfin plusieurs fonctions d'accès aux éléments du tableau dans le corps de boucles.

Nous avons divisé l'étude des cas où le code de la tâche contient plusieurs références au même tableau en quatre sous-cas bien spécifiques. Cette distinction permet de trouver pour chacun d'eux un meilleur compromis entre le nombre d'éléments à transférer et la complexité des codes de transfert réalisant les copies inter-mémoire.

Nous dissocions:

- le cas où les fonctions d'accès aux éléments du tableau sont dépendantes de manière uniforme, car il peut être traité en utilisant les algorithmes de génération de code pour *une* seule référence au tableau,

- le cas où les domaines référencés par les différentes fonctions d'accès aux éléments du tableau sont des Z-polyèdres de même Z-module, car il est alors possible d'optimiser le nombre d'éléments à transférer plus simplement que lorsque les Z-modules sont différents,
- le cas où les domaines référencés par les différentes fonctions d'accès aux éléments du tableau sont des Z-polyèdres dont l'un des Z-modules inclus les autres,
- et enfin, le cas où les domaines référencés par les différentes fonctions d'accès aux éléments du tableau sont des Z-polyèdres dont les Z-modules sont très différents.

Les algorithmes proposés pour générer ces codes de transfert sont basés autour des quatres algorithmes suivant:

- l'algorithme de génération des codes de transfert pour une seule référence au tableau comprenant: le calcul d'une base du *domaine image* [§5.2] et le calcul de la projection entière d'un polyèdre [§4.4.5] dans cette nouvelle base,
- l'algorithme de recherche du plus petit polyèdre englobant un Z-polyèdre non convexe [§4.5.1],
- l'algorithme de recherche du plus grand polyèdre contenu dans un Z-polyèdre non convexe [§4.5.2],
- l'algorithme de calcul de la différence de deux Z-polyèdres [§4.3].

Le tableau suivant rappelle les différentes options étudiées et les sections qu'il est conseillé de lire pour chacune d'elle.

<table>
<tr><td colspan="5">Déclaration du tableau local</td><td>5.1 → 5.5</td></tr>
<tr><td colspan="5">Choix d'une fonction d'accès aux éléments du tableau global</td><td>5.1 → 5.3</td></tr>
<tr><td colspan="5">Choix d'une fonction d'accès aux éléments du tableau local</td><td>5.1 → 5.4</td></tr>
<tr><td></td><td>Une référence au tableau</td><td>plus. ref. en translation</td><td>plus. ref. ayant le même Z-module</td><td>plus. ref. ayant des Z-modules inclus</td><td>plus. ref. ayant des Z-modules différents</td></tr>
<tr><td>Génération du code de transfert de la MG[1] vers la ML[2]</td><td>6.1
6.4</td><td>6.1 - 6.2.1
6.3.1 - 6.5
6.5.1 - 6.6.1</td><td>6.1 - 6.2.2
6.3.2 - 6.5
6.5.2 - 6.6.2</td><td>6.1 - 6.2.4
6.3.3 - 6.5
6.5.3 - 6.6.3</td><td>6.1 - 6.2.3
6.3.4 - 6.5
6.5.4 - 6.6.4</td></tr>
<tr><td>Génération du code de transfert de la ML vers la MG</td><td>6.1</td><td>6.1
6.2.1
6.3.1</td><td>6.1
6.2.2
6.3.2</td><td>6.1
6.2.4
6.3.3</td><td>6.1
6.2.3
6.3.4</td></tr>
<tr><td colspan="5">cas des tests IF</td><td>7.1</td></tr>
<tr><td colspan="5">Fonctions d'accès aux éléments du tableau non linéaires</td><td>7.2</td></tr>
<tr><td colspan="5">Boucles non imbriquées</td><td>7.3</td></tr>
<tr><td colspan="5">Appel de fonction</td><td>7.4</td></tr>
</table>

[1] Mémoire Globale
[2] Mémoire Locale

6.1 Génération du code de transfert de la mémoire locale vers la mémoire globale pour une seule référence à un tableau dans le corps de boucles

Nous avons trouvé en [§4.1.3] que la forme réduite de Hermite H associée à la fonction φ d'accès aux éléments du tableau T permettait d'obtenir une base du Z-module caractérisant ces éléments: $\varphi = P^{-1}HQ^{-1}\vec{j} + \vec{f_0} = P^{-1}H\vec{t} + \vec{f_0}$. Pour générer le code de transfert de la mémoire locale vers la mémoire globale des éléments référencés par le tableau T, il faut générer le (ou les) nid de boucles permettant de copier l'ensemble de ces éléments. Le nouvel espace d'itération des vecteurs de base $\vec{t}$ s'obtient par le changement de base $\vec{j} = Q\vec{t}$, et le nouveau domaine vaut: $A\vec{j} \leq \vec{\alpha} \Longrightarrow A\,Q\,\vec{t} \leq \vec{\alpha}$. Les nouvelles bornes du code de transfert dépendent de ce dernier, il aura la forme suivante:

```
DO t1 = N, M
  DO t2 = f21(t1), f22(t1)
    ...
     DO tn = fn1(t1, t2, ..., tn-1), fn2(t1, t2, ..., tn-1)
          TL(φ'(t1, t2, ..., tn)) = T (φ(t1, t2, ..., tn))
     ENDDO
    ...
  ENDDO
ENDDO
```

Toutefois, lorsque la dimension r du domaine image est inférieure à la dimension n du domaine d'itération, la matrice H est de rang r et la nouvelle fonction d'accès aux éléments du tableau ne s'exprime qu'en fonction des r premiers vecteurs de base. Afin de minimiser l'overhead de contrôle et les copies multiples engendrées par le code de transfert précédent, il faut alors effectuer une projection entière du domaine d'itération sur les r premiers vecteurs de base. Nous en déduirons, dans tous les cas où c'est possible, un code de transfert de la forme:

```
DO t1 = N, M
  DO t2 = f21(t1), f22(t1)
    ...
     DO tr = fn1(t1, t2, ..., tr-1), fn2(t1, t2, ..., tr-1)
          TL(φ'(t1, t2, ..., tr)) = T (φ(t1, t2, ..., tr))
     ENDDO
    ...
  ENDDO
ENDDO
```

La dernière étape (étape 4) de l'algorithme de projection entière [§4.4.5] consiste à simplifier le système linéaire en éliminant les variables supplémentaires des contraintes du système par introduction de divisions entières. La génération du code de transfert dépend de cette derniere étape.

6.1.1 Cas où l'ensemble des éléments à transférer est un polyèdre convexe.

Lorsque l'ensemble des éléments à transférer est convexe, le résultat P des trois premières phases de l'algorithme de projection entière selon les n-r variables t_{r+1}, t_{r+2}, ..., t_n ne contient plus de contraintes sur l'une de ces variables.

Pour générer le code de transfert des éléments référencés par T, il suffit donc de calculer les nouvelles bornes de boucle de ce polyèdre "projeté P " et d'en déduire le code de transfert correspondant.

On utilise l'algorithme de F.Irigoin présenté dans [Irig88]:

6.1.2 Algorithme de calcul des bornes de boucle d'un polyèdre convexe

Si le polyèdre définissant le domaine d'itération dans la nouvelle base s'exprime $A \,.\, Q \,.\, \vec{t} \leq \vec{\alpha}$.

1. **On commence par rechercher les bornes de la boucle la plus interne d'indice t_1. C'est le système initial défini par $A \,.\, Q \,.\, \vec{t} \leq \vec{\alpha}$ qui donne directement les bornes de t_1 en fonction de tous les autres indices des boucles englobantes.**

   Exemple:

   $$\begin{cases} -t_1 + t_2 \leq -1 \\ t_1 - t_2 \leq 10 \\ -t_2 \leq -1 \\ t_2 \leq 10 \end{cases}$$ *les bornes de t_1 en fonction des autres indices sont :* $$\begin{cases} -t_1 \leq -1 - t_2 \\ t_1 \leq 10 + t_2 \end{cases}$$

2. **Pour calculer les bornes de la (r-1)-ième boucle, on projette le système selon la variable t_1 ce qui nous permet d'exprimer t_2 en fonction des r-2 premiers indices de boucle.**

3. **Pour calculer les bornes de chaque (r-i)-ième boucle, on projette le système selon la variable t_i ce qui nous permet d'exprimer t_{i+1} en fonction des r-i-1 premiers indices de boucle.**

4. **Enfin, ce sont les projections successives selon les r-1 premières variables du système qui nous donneront les bornes numériques de boucle la plus englobante d'indice t_r.**

   A chaque étape de l'algorithme on élimine les contraintes redondantes dans le sytème, en respectant les règles suivantes:

   (a) **On conserve au moins deux contraintes sur chacune des variables de manière à pouvoir générer les bornes inférieure et supérieure de la boucle lui correspondant.**

   (b) **Les contraintes redondantes portant sur les indices de boucle les plus internes sont éliminées en premier de façon à conserver des expressions de bornes de boucle les plus simples possibles.**

6.1.3 Cas où l'ensemble des éléments à transférer est un Z-polyèdre non convexe.

Lorsque l'ensemble des éléments à transférer est un Z-polyèdre non convexe, le système linéaire résultant des trois premières phases de l'algorithme de projection entière selon les variables t_{r+1}, t_{r+2}, ..., t_n a conservé des inégalités sur l'une au moins de ces variables supplémentaires. Ces contraintes traduisent la non-convexité du domaine.

La phase 4 de l'algorithme permet d'éliminer ces variables par combinaisons de paires d'inégalités et par introduction de divisions entières.

Dans un premier temps, on élimine la variable k des deux contraintes de la façon suivante [§4.4.4]:

$$(S) \begin{cases} -d_2\ k + A_2 \leq c_2 \\ d_1\ k + A_1 \leq c_1 \end{cases} \implies (S) \left\{ \frac{d_1\ (A_2 - c_2) + d_1\ d_2 - d_1}{d_1 d_2} \leq \frac{-d_2\ (A_1 - c_1)}{d_1 d_2} \right. \qquad (C)$$

Nous avons alors deux cas de figure:

1. le cas où après élimination de la variable supplémentaire, il est possible de déduire de la contrainte C une nouvelle contrainte C' sur la variable de boucle t_i la plus interne (d'indice N_i le plus fort) parce qu'elle n'apparaît que d'un côté de l'inégalité C.

   Dans ce cas, la nouvelle contrainte C' est ajoutée à la borne inférieure ou supérieure de la boucle correspondant à la variable de base t_i dans le code de transfert.

   Prenons, pour exemple, les deux contraintes suivantes:

$$\begin{cases} t_2 - 30 \leq 3\ t_4 \\ 3\ t_4 \leq -\ t_1 + 20 \end{cases} \Leftrightarrow \begin{cases} \frac{t_2 - 28}{3} \leq t_4 \\ t_4 \leq \frac{-t_1 + 20}{3} \end{cases} \Leftrightarrow \frac{t_2 - 28}{3} \leq \frac{-t_1 + 20}{3}$$

   on peut ici déduire des deux premières inégalités, la contrainte sur t_1:

$$t_1 \leq 20\ - 3\ \frac{t_2 - 28}{3}$$

   Exemple [1]

   Considérons le nouvel espace d'itération des vecteurs de base d'un domaine image de dimension 2 défini par S. On recherche la projection entière du système sur les deux premiers vecteurs de base t_1 et t_2.

$$S = \begin{cases} -t_3 \leq -1 \\ t_3 \leq 10 \\ -t_2 - 3t_3 \leq -1 \\ t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

[1] Cet exemple est entièrement détaillé à la section [§6.1.7] (après la présentation complète de l'algorithme de génération automatique du code de transfert pour une référence au tableau)

La projection entière du domaine d'itération selon la dernière variable donne:

$$SP = \begin{cases} t_2 \leq 17 \\ -t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \\ -t_2 - 3t_3 \leq -1 \\ t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

Le système résultant de cette "projection entière" contient encore des contraintes sur la variable t_3.

L'on peut déduire de chaque contrainte du système SP des inégalités sur les variables de base les plus internes qu'elles contiennent:

$$\begin{cases} -t_2 - 3t_3 \leq -1 \\ t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases} \iff \begin{cases} -t_2 + 1 \leq 3t_3 \leq t_1 - 1 \\ t_1 - 30 \leq 3t_3 \leq 20 - t_2 \end{cases}$$

$$\iff \begin{cases} 1 + 3(\frac{3-t_2}{3}) \leq t_1 \\ t_1 \leq 30 + 3(\frac{20-t_2}{3}) \end{cases}$$

On ajoutera donc la première contrainte à la borne inférieure de la boucle de t_1 et la seconde à la borne supérieure.

2. le cas où après élimination de la variable supplémentaire, il n'est pas possible de déduire de la contrainte C une nouvelle contrainte C' sur la variable de boucle t_i la plus interne (d'indice N_i le plus fort), parce qu'elle apparaît des deux côtés de l'inégalité C.

   Ces contraintes attestent de l'appartenance ou de la non appartenance du point d'itération calculé dans le domaine d'itération "projeté ". Ces contraintes sont donc introduites dans la partie IF du code de transfert.

   Par exemple,

$$\begin{cases} t_1 - 30 \leq 3\,t_4 \\ 3\,t_4 \leq t_1 - 29 \end{cases} \Leftrightarrow \begin{cases} \frac{t_1 - 28}{3} \leq t_4 \\ t_4 \leq \frac{t_1 - 29}{3} \end{cases} \Leftrightarrow \frac{t_1 - 28}{3} \leq \frac{t_1 - 29}{3}$$

   On ne peut déduire aucune contrainte sur t_1 de cette dernière inégalité. Ici on a une condition périodique sur la variable de base. On ajoutera donc cette inégalité dans la partie test IF du code de transfert.

Remarque:

Le système:

$$\left.\begin{array}{l} t_1 + c \leq d.k \\ d.k \leq t_1 + c \end{array}\right\} \iff \frac{t_1 + c + d - 1}{d} \leq \frac{t_1 + c}{d}$$

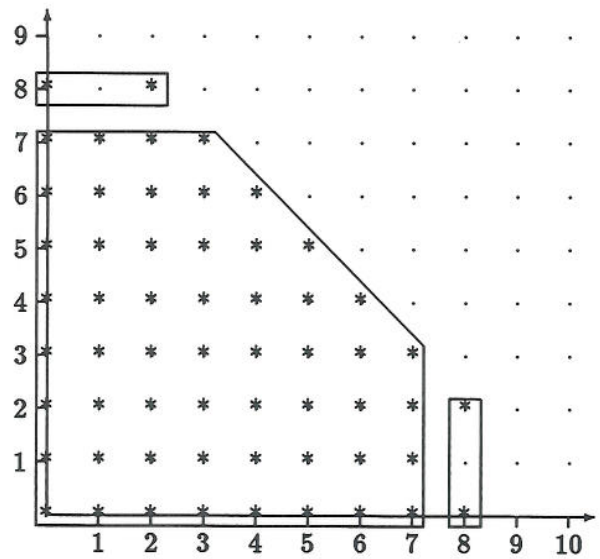
est un cas particulier, car il est équivalent à $t_1 + c = kd$. Nous n'utiliserons pas dans ce cas de test IF contenant l'expression: $\frac{t_1+c+d-1}{d} \leq \frac{t_1+c}{d}$, mais plutôt une boucle sur la variable t_1 avec un pas de d unités [voir §6.1.7 exemple 4].

Exemple [2]

Prenons le polyèdre

$$P = \left\{\begin{array}{rcl} t_1 + t_3 + t_4 - t_5 - t_6 & \leq & 7 \\ -t_1 - t_3 - t_4 + t_5 + t_6 & \leq & -1 \\ t_1 + t_2 + 2.t_3 - 2.t_6 & \leq & 8 \\ -t_2 - t_3 + t_4 - t_5 + t_6 & \leq & -1 \\ t_3 & \leq & 1 \\ -t_3 & \leq & 0 \\ t_3 + t_4 & \leq & 1 \\ -t_4 & \leq & 0 \\ t_3 + t_4 + t_5 & \leq & 1 \\ -t_5 & \leq & 0 \\ t_3 + t_4 + t_5 + t_6 & \leq & 1 \\ -t_6 & \leq & 0 \end{array}\right.$$

dont la projection entière sur les deux variables de base t_1 et t_2 donne l'ensemble de points référencés recherchés suivant:



[2] Cet exemple est entièrement détaillé à la section [§6.1.7] (après la présentation complète de l'algorithme de génération automatique du code de transfert pour une référence au tableau)

Le système résultant de la projection entière contient encore des contraintes sur les variables t_3 *et* t_6:

$$PP = \left\{ \begin{array}{rcl} t_2 + t_1 + 4t_3 & \leq & 10 \\ t_2 + t_1 & \leq & 10 \\ -2t_6 + 2t_3 + t_2 + t_1 & \leq & 8 \\ -2t_6 + t_2 + t_1 & \leq & 8 \\ -t_1 - 2t_3 - t_2 & \leq & -2 \\ 2t_6 - t_2 & \leq & 0 \\ -t_2 & \leq & 0 \\ 2t_6 - t_1 & \leq & 0 \\ -t_1 & \leq & 0 \end{array} \right.$$

L'on ne peut pas déduire de chaque contrainte du système PP des inégalités sur les variables de base les plus internes:

$$\left\{ \begin{array}{rcl} -2t_6 + t_2 + t_1 & \leq & 8 \\ 2t_6 - t_1 & \leq & 0 \end{array} \right. \iff \frac{t_2+t_1-7}{2} \leq \frac{t_1}{2}$$

Cette inégalité permet de masquer le point: $(t_1 = 1, t_2 = 8)$ qui n'appartient pas au domaine image.

Le type de code de transfert généré devrait donc être de la forme:

```
DO  t1 = N, M
 DO  t2 = f21(t1), f22(t1)
     IF  ((t1+t2-7)/2 ≤ t1/2)      THEN      T(t1,t2)  =  TL(t1,t2)
ENDDO
```

En observant bien la figure précédente, nous remarquons que le domaine est *non-convexe* dans les deux dimensions. Il est donc impossible de copier l'ensemble des éléments référencés en utilisant un seul corps de boucles sans test. En effet, les bornes du premier indice de boucle sont numériques, les bornes du second traduisent la non convexité du domaine selon la deuxième dimension en employant des bornes pseudo-linéaires, elles ne peuvent pas traduire la non convexité selon la première dimension. Il faut donc un test pour masquer ces éléments.

L'ensemble des problèmes que nous avons traités, jusqu'à maintenant, montre que la phase 4 de l'algorithme de projection entière permet d'éliminer la totalité des variables supplémentaires. Le code de transfert en résultant, aura alors un nombre de boucles minimal. Cependant nous n'avons pas prouvé cette propriété.

Dans le cas où il resterait des variables supplémentaires dans le système projeté il faudrait alors conserver cette variable supplémentaire, au même titre qu'une variable de base, et génèrer une boucle lui correspondant. Cette boucle supplémentaire peut cependant conduire à des transferts multiples d'une donnée puisqu'on ajoute une dimension (dim(H) +1) au domaine projeté. Comme l'introduction des divisions entières, ces boucles supplémentaires servent à retraduire la non-convexité du domaine "projeté ".

Lorsque les transferts vectoriels sont possibles, il faut éviter de générer des tests dans le code de transfert. La deuxième solution que nous proposons permet de conserver des possibilités de transferts vectoriels de l'ensemble des éléments appartenant au plus grand polyèdre convexe contenu dans le polyèdre non convexe. Elle permet de plus de conserver un nombre de transferts optimal.

Elle est la suivante:

1. **Rechercher le plus grand polyèdre convexe contenu dans le polyèdre non convexe résultant de la projection. [§4.5.2]**
2. **Rechercher l'ensemble des polyèdres non convexes dont l'union avec le polyèdre précédant forme le polyèdre initial. [§4.5.3]**
3. **Générer le code de transfert de ces ensembles séparément en utilisant l'algorithme décrit précédemment.**

Cette solution permet de conserver un nombre de transferts optimal, car on ne transfère que des ensembles convexes.

6.1.4 Constantes symboliques dans la fonction d'accès des références

La présence de constantes symboliques dans la partie constante de la fonction d'accès aux références d'un tableau T ne pose aucun problème.

En effet, la partie constante de φ ne modifie pas la base du Z-module, elle correspond juste à une translation. Toute constante symbolique sera donc placée dans f_0.

6.1.5 Constantes symboliques dans la partie constante des bornes de boucle

La présence de constantes symboliques dans la partie constante des bornes de boucle ne pose pas de problème, car les algorithmes que nous utilisons traitent indépendamment les constantes et les variables. Les constantes symboliques doivent donc être traitées comme des variables particulières du système, selon lesquelles on n'a pas à projeter.

6.1.6 Algorithme de génération du code de transfert pour une référence dans un corps de boucles

L'algorithme de génération du code de transfert des éléments référencés par une fonction d'accès au tableau T est le suivant:

1. Calcul de la matrice de Hermite H [§4.1.3] de dimension r associée à F et des matrices P et Q telles que: $F = P^{-1}.H.Q^{-1}$
2. Calcul du domaine d'itération dans la nouvelle base $\vec{t} = Q^{-1}.\vec{j} : A.Q.\vec{j} \leq \vec{\alpha}$
3. Projection entière (phases 1-2-3) [§4.4.5] du domaine d'itération selon les n-r dernières variables t_{r+1}, t_{r+2}, ..., *et* t_n de $\vec{t}$ pour obtenir le domaine projeté PP.
4. Si le système résultant de la projection entière ne contient plus de variable supplémentaire:
   (a) Calculer les nouvelles bornes [§6.1.2] du polyèdre projeté par projections successives de la première variable t_1 jusqu'à la dernière t_r.
   (b) Calculer la nouvelle fonction d'accès aux éléments dans la nouvelle base: $\vec{F} = P^{-1}.H.\vec{t} + \vec{f_0}$.
   (c) Générer le code de transfert correspondant.
5. Si le système résultant de la projection entière contient encore des contraintes sur l'une au moins des variables supplémentaires:
   (a) Séparer le système en deux sous-systèmes: un premier S_1 constitué des contraintes dans lesquelles les variables supplémentaires n'apparaissent pas, et un second S_2.
   (b) Si l'on peut déduire de chaque contrainte du système S_2 des inégalités sur les variables de base les plus internes qu'elles contiennent:
      i. Calculer les nouvelles bornes du polyèdre défini par le sous-système S_1 par projections successives de la première variable t_1 jusqu'à la dernière t_r.
      ii. Ajouter les contraintes supplémentaires imposées par S_2 (contenant des divisions entières) aux bornes de boucle correspondantes.
      iii. Calculer la nouvelle fonction d'accès aux éléments dans la nouvelle base: $\vec{\varphi} = P^{-1}.H.\vec{t} + \vec{f_0}$.
      iv. Générer le code de transfert correspondant.
   (c) Si l'on ne peut pas déduire de chaque contrainte du système S_2 des inégalités sur les variables de base les plus internes qu'elles contiennent:
      i. calculer le plus grand polyèdre PGPC contenu dans le polyèdre projeté, et l'ensemble des polyèdres constituant le complémentaire PC de PGPC dans PP.

ii. Générer le code de transfert du polyèdre PGPC.

A. Calculer les nouvelles bornes du polyèdre projeté par projections successives de la première variable t_1 jusqu'à la dernière t_r.

B. Calculer la nouvelle fonction d'accès aux éléments dans la nouvelle base: $\vec{\varphi} = P^{-1}.H.\vec{t} + \vec{f_0}$.

C. Générer le code de transfert correspondant.

iii. Générer le code de transfert du polyèdre PC, en introduisant une partie de TEST IF, et des boucles supplémentaires selon les contraintes.

A. Simplifier chaque contrainte comportant une variable supplémentaire.

B. Calculer les nouvelles bornes du polyèdre projeté par projections successives selon l'ensemble des variables restantes $t_1, ...t_m$.

C. Ajouter les contraintes simplifiées aux bornes de boucle correspondantes ou dans la partie TEST IF du corps de boucles

D. Générer le code de transfert correspondant.

6.1.7 Exemples

Nous présentons ici cinq exemples permettant d'observer les différentes étapes de notre algorithme:

Exemple 1:

Voici le premier exemple et les différentes étapes de notre algorithme pour le corps de boucles:

```
DO  I = 1,10
  DO  J = 1,10
      TL(I+J,I+J) = ..
ENDDO
```

La matrice A, le vecteur $\vec{\alpha}$ et les matrices associées à F et f_0 s'écrivent:

$$A = \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} , \quad \vec{\alpha} = \begin{pmatrix} -1 \\ 10 \\ -1 \\ 10 \end{pmatrix}$$

et

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} , \quad f_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Calcul de la matrice de Hermite H associée à F (étape 1):

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Calcul du domaine d'itération dans la nouvelle base (étape 2)

$$A.Q.\vec{t} \leq \vec{\alpha} \iff \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \vec{t} \leq \begin{pmatrix} -1 \\ 10 \\ -1 \\ 10 \end{pmatrix}$$

qui définit le système de contraintes:

$$P = \begin{cases} -t_1 + t_2 \leq -1 \\ t_1 - t_2 \leq 10 \\ -t_2 \leq -1 \\ t_2 \leq 10 \end{cases}$$

Projection entière du domaine d'itération selon les n-r dernières variables, soit ici uniquement t_2 (étape 3):
Le polyèdre projeté est:

$$PP = \begin{cases} t_1 \leq 20 \\ -t_1 \leq -2 \end{cases}$$

Le système résultant de cette "projection entière" ne contient plus de contraintes sur la variable t_2:

- Calcul des nouvelles bornes du polyèdre (ici explicites) (étape 4a):

$$\begin{cases} t_1 \leq 20 \\ -t_1 \leq -2 \end{cases}$$

- Calcul de la nouvelle fonction d'accès aux éléments dans la nouvelle base(étape 4b):

$$\vec{\varphi} = \begin{pmatrix} t_1 \\ t_1 \end{pmatrix}$$

- Génération du code de transfert (étape 4c):

```
DO  t1 = 2,20
  T(t1,t1)  =  TL(t1)
ENDDO
```

Exemple 2:

Le deuxième exemple présente les étapes de notre algorithme pour le code de calcul proposé dans [GJGa88]. Nous traitons successivement le cas où l'on choisit la forme de Hermite comme base du domaine image et le cas où l'on choisit une base favorisant les transferts contigus.

```
DO  I = 1,10
  DO  J = 1,20
    DO  K = 1,30
        TL(3 × I + K − 5, J + K) = ....
  ENDDO
```

La matrice A, le vecteur $\vec{\alpha}$ et les matrices associées à F et f_0 s'écrivent:

$$A = \begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \quad , \quad \vec{\alpha} = \begin{pmatrix} -1 \\ 10 \\ -1 \\ 20 \\ -1 \\ 30 \end{pmatrix}$$

et

$$F = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \quad , \quad f_0 = \begin{pmatrix} -5 \\ 0 \end{pmatrix}$$

Calcul de la matrice de Hermite H associée à F (étape 1):

$$F = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 & 1 \\ -3 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Calcul du domaine d'itération dans la nouvelle base (étape 2)

$$A.Q.\vec{t} \leq \vec{\alpha} \iff \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 1 \\ 0 & -1 & -3 \\ 0 & 1 & 3 \\ -1 & 0 & 3 \\ 1 & 0 & -3 \end{pmatrix} \vec{t} \leq \begin{pmatrix} -1 \\ 10 \\ -1 \\ 20 \\ -1 \\ 30 \end{pmatrix}$$

qui définit le système de contraintes:

$$P = \begin{cases} -t_3 \leq -1 \\ t_3 \leq 10 \\ -t_2 - 3t_3 \leq -1 \\ t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

Projection entière du domaine d'itération selon les n-r dernières variables, soit ici uniquement t_3 (étape 3):

Le polyèdre projeté est:

$$PP = \begin{cases} t_2 \leq 17 \\ -t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \\ -t_2 - 3t_3 \leq -1 \\ t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

Le système résultant de cette "projection entière" contient encore des contraintes sur la variable t_3:

- Séparation du système en deux sous-systèmes (étape 5a):

$$S_1 = \begin{cases} t_2 \leq 17 \\ -t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \end{cases} \quad et \quad S_2 = \begin{cases} -t_2 - 3t_3 \leq -1 \\ t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

L'on peut déduire de chaque contrainte du système S_2 des inégalités sur les variables de base les plus internes qu'elles contiennent:

$$S_2 = \begin{cases} -t_2 - 3t_3 \leq -1 \\ t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases} \iff \begin{cases} -t_2 + 1 \leq 3t_3 \leq t_1 - 1 \\ t_1 - 30 \leq 3t_3 \leq 20 - t_2 \end{cases}$$

$$\iff \begin{cases} 1 + 3(\frac{3-t_2}{3}) \leq t_1 \\ t_1 \leq 30 + 3(\frac{20-t_2}{3}) \end{cases}$$

- Calcul des nouvelles bornes du polyèdre défini par S_1 (ici explicites) (étape 5bi):

$$\begin{cases} t_2 \leq 17 \\ -t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \end{cases}$$

- Ajout des contraintes supplémentaires imposées par S_2 aux bornes de boucle correspondantes (étape 5bii).

$$\begin{cases} t_2 \leq 17 \\ -t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \\ 1 + 3(\frac{3-t_2}{3}) \leq t_1 \\ t_1 \leq 30 + 3(\frac{20-t_2}{3}) \end{cases}$$

- Calcul de la nouvelle fonction d'accès aux éléments dans la nouvelle base(étape 5biii):

$$\vec{\varphi} = \begin{pmatrix} t_1 - 5 \\ t_1 + t_2 \end{pmatrix}$$

- Génération du code de transfert (étape 5biv):

```
DO  t2 = -29,17
  DO  t1 = MAX(4,1 + 3 × ((3-t2)/3)) , MIN(60, 30 + 3 × ((20-t2)/3))
      T(t1 - 5, t1 + t2)  =  TL(t1 - 5, t1 + t2)
ENDDO
```

Reprenons maintenant le même exemple mais en prenant une base qui garantit les possibilités de transferts contigus.

On choisit la matrice de changement de base [§5.2.1] $Q = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 1 & 3 \\ 1 & 0 & -3 \end{pmatrix}$

Calcul de la matrice H_m associée à F [§5.2.1] (étape 1):

$$F = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} = P^{-1}HQ^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Calcul du domaine d'itération dans la nouvelle base (étape 2)

$$A.Q.\vec{t} \leq \vec{\alpha} \iff \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 1 \\ 1 & -1 & -3 \\ -1 & 1 & 3 \\ -1 & 0 & 3 \\ 1 & 0 & -3 \end{pmatrix} \vec{t} \leq \begin{pmatrix} -1 \\ 10 \\ -1 \\ 20 \\ -1 \\ 30 \end{pmatrix}$$

qui définit le système de contraintes:

$$P = \begin{cases} -t_3 \leq -1 \\ t_3 \leq 10 \\ t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

Projection entière du domaine d'itération selon les n-r dernières variables, soit ici uniquement t_3 (étape 3):

Le polyèdre projeté est:

$$PP = \begin{cases} -t_1 + t_2 \leq 17 \\ t_1 - t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \\ t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

Le système résultant de cette "projection entière" contient encore des contraintes sur la variable t_3:

- Séparation du système en deux sous-systèmes (étape 5a):

$$S_1 = \begin{cases} -t_1 + t_2 \leq 17 \\ t_1 - t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \end{cases} \quad et \quad S_2 = \begin{cases} t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \end{cases}$$

L'on ne peut pas déduire de chaque contrainte du système S_2 des inégalités sur les variables de base les plus internes (ici t_1):

$$\begin{cases} t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + 3t_3 \leq -1 \end{cases} \iff \frac{t_1 - t_2 + 3}{3} \leq \frac{t_1 - 1}{3}$$

dont on ne peut rien déduire sur t_1 $\Longrightarrow$:

- calcul du plus grand polyèdre PGPC contenu dans le polyèdre projeté, et de l'ensemble des polyèdres constituant le complémentaire PC de PGPC dans PP (étape 5ci).

$$PP = \begin{cases} -t_1 + t_2 \leq 17 \\ t_1 - t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \\ -t_2 \leq -4 \\ t_2 \leq 48 \end{cases}$$

$$PC = PC_1 \begin{cases} -t_1 + t_2 \leq 17 \\ t_1 - t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \\ t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \\ t_2 \leq 3 \end{cases} + PC_2 \begin{cases} -t_1 + t_2 \leq 17 \\ t_1 - t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 \leq 60 \\ t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + t_2 + 3t_3 \leq 20 \\ -t_1 + 3t_3 \leq -1 \\ t_1 - 3t_3 \leq 30 \\ -t_2 \leq -49 \end{cases}$$

qui après élimination des contraintes redondantes s'écrivent:

$$PC = PC_1 \begin{cases} t_1 - t_2 \leq 29 \\ -t_1 \leq -4 \\ t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + 3t_3 \leq -1 \\ t_2 \leq 3 \end{cases} + PC_2 \begin{cases} -t_1 + t_2 \leq 17 \\ t_1 \leq 60 \\ -t_1 + t_2 + 3t_3 \leq 20 \\ t_1 - 3t_3 \leq 30 \\ -t_2 \leq -49 \end{cases}$$

- Génération du code de transfert du polyèdre PGPC (étape 5cii).
  - Calcul de la nouvelle fonction d'accès aux éléments dans la nouvelle base:
    $\vec{\varphi} = \begin{pmatrix} t_1 - 5 \\ t_2 \end{pmatrix}$
  - Génération du code de transfert correspondant.

```
DO  t2 =  4,48
  DO  t1  =  MAX(4,t2 - 17) , MIN(60,29 - t2)
      T(t1 - 5,t2)   =   TL(t1 - 5,t2)
ENDDO
```

- Génération du code de transfert du polyèdre PC (étape 5ciii)
  - Simplification de chaque contrainte conportant une variable supplémentaire.

$$\begin{cases} t_1 - t_2 - 3t_3 \leq -1 \\ -t_1 + 3t_3 \leq -1 \end{cases} \iff \frac{t_1 - t_2 + 3}{3} \leq \frac{t_1 - 1}{3}$$

$$\begin{cases} -t_1 + t_2 + 3t_3 \leq 20 \\ t_1 - 3t_3 \leq 30 \end{cases} \iff \frac{t_1 - 28}{3} \leq \frac{t_1 - t_2 + 20}{3}$$

  - Génération du code de transfert correspondant.
    Pour PC_1

```
DO  t2  =  2,3
  DO  t1 = 4,29 + t2
  IF  ((t1-t2+3)/3  ≤  (t1-1)/3)  THEN    T(t1 - 5,t2)   =   TL(t1 - 5,t2)
ENDDO
```

    Pour PC_2

```
DO  t2  =  49,50
  DO  t1 = t2 - 17,60
   IF  ((t1-28)/3  ≤  (t1-t2+20)/3) THEN    T(t1 - 5,t2)   =   TL(t1 - 5,t2)
ENDDO
```

La totalité du code de tranfert est:

```
DO  t2 = 4,48
DO  t1 = MAX(4,t2 - 17) , MIN(60,29 - t2)
    T(t1 - 5,t2)  =  TL(t1 - 5,t2)
ENDDO

DO  t2 = 2,3
  DO  t1 = 4,29 + t2
  IF  ((t1-t2+3)/3  ≤  (t1-1)/3)  THEN   T(t1 - 5,t2)  =  TL(t1 - 5,t2)
ENDDO

DO  t2 = 49,50
  DO  t1 = t2 - 17,60
   IF ((t1-28)/3  ≤  (t1-t2+20)/3)  THEN   T(t1 - 5,t2)  =  TL(t1 - 5,t2)
ENDDO
```

Les éléments référencés lors des premiers transferts pourront être copiés de manière contiguë.

Exemple 3:

Voici le troisième exemple et les différentes étapes de notre algorithme pour le corps de boucles:

```
DO  I = 1,N
 DO  J = 1,M
  DO  K = 1,L
      TL(2 × I + K,J + K) = ...
ENDDO
```

La matrice A, le vecteur $\vec{\alpha}$ et les matrices associées à F et f_0 s'écrivent:

$$A = \begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \quad , \quad \vec{\alpha} = \begin{pmatrix} -1 \\ N \\ -1 \\ M \\ -1 \\ L \end{pmatrix}$$

et

$$F = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \quad , \quad f_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Calcul de la matrice de Hermite H associée à F (étape 1):

$$F = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 1 \\ -2 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Calcul du domaine d'itération dans la nouvelle base (étape 2)

$$A.Q.\vec{t} \leq \vec{\alpha} \iff \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 1 \\ 0 & -1 & -2 \\ 0 & 1 & 2 \\ -1 & 0 & 2 \\ 1 & 0 & -2 \end{pmatrix} \vec{t} \leq \begin{pmatrix} -1 \\ N \\ -1 \\ M \\ -1 \\ L \end{pmatrix}$$

qui définit le système de contraintes:

$$P = \begin{cases} -t_3 \leq -1 \\ t_3 \leq N \\ -t_2 - 2.t_3 \leq -1 \\ t_2 + 2.t_3 \leq M \\ -t_1 + 2.t_3 \leq -1 \\ t_1 - 2.t_3 \leq L \end{cases}$$

Projection entière du domaine d'itération selon les n-r dernières variables, soit ici uniquement t_3 (étape 3):

Le polyèdre projeté est:

$$PP = \begin{cases} t_2 \leq M - 2 \\ -t_2 \leq 2.N - 1 \\ -t_1 \leq -3 \\ t_1 \leq L + 2.N \\ -t_2 - 2.t_3 \leq -1 \\ t_2 + 2.t_3 \leq M \\ -t_1 + 2.t_3 \leq -1 \\ t_1 - 2.t_3 \leq L \end{cases}$$

Le système résultant de cette "projection entière" contient encore des contraintes sur la variable t_3:

- Séparation du système en deux sous-systèmes (étape 5a):

$$S_1 = \begin{cases} t_2 \leq M - 2 \\ -t_2 \leq 2.N - 1 \\ -t_1 \leq -3 \\ t_1 \leq L + 2.N \end{cases} \quad et \quad S_2 = \begin{cases} -t_2 - 2.t_3 \leq -1 \\ t_2 + 2.t_3 \leq M \\ -t_1 + 2.t_3 \leq -1 \\ t_1 - 2.t_3 \leq L \end{cases}$$

L'on peut déduire de chaque contrainte du système S_2 des inégalités sur les variables de base les plus internes qu'elles contiennent:

$$S_2 = \begin{cases} -t_2 - 2.t_3 \leq -1 \\ t_2 + 2.t_3 \leq M \\ -t_1 + 2.t_3 \leq -1 \\ t_1 - 2.t_3 \leq L \end{cases} \iff \begin{cases} -t_2 + 1 \leq 2.t_3 \leq t_1 - 1 \\ t_1 - L \leq 2.t_3 \leq M - t_2 \end{cases}$$

$$\iff \begin{cases} 1 + 2.(\frac{2-t_2}{2}) \leq t_1 \\ t_1 \leq L + 2.(\frac{M-t_2}{2}) \end{cases}$$

* Calcul des nouvelles bornes du polyèdre défini par S_1 (ici explicites) (étape 5bi):

$$\begin{cases} t_2 \leq M - 2 \\ -t_2 \leq 2.N - 1 \\ -t_1 \leq -3 \\ t_1 \leq L + 2.N \end{cases}$$

* Ajout des contraintes supplémentaires imposées par S_2 aux bornes de boucle correspondantes (étape 5bii).

$$\begin{cases} t_2 \leq M - 2 \\ -t_2 \leq 2.N - 1 \\ -t_1 \leq -3 \\ t_1 \leq L + 2.N \\ -t_1 \leq -1 - 2.(\frac{t_2-2}{2}) \\ t_1 \leq L + 2.(\frac{M-t_2}{2}) \end{cases}$$

* Calcul de la nouvelle fonction d'accès aux éléments dans la nouvelle base(étape 5biii):

$$\vec{\varphi} = \begin{pmatrix} t_1 \\ t_1 + t_2 \end{pmatrix}$$

* Génération du code de transfert (étape 5biv):

```
DO  t2 = 2 × N − 1, M − 2
  DO  t1 = MAX(3, 1 + 2 × ((t2−2)/2)) , MIN(L + 2 × N, L + 2 × ((M−t2)/2))
      T(t1, t1 + t2) = TL(t1, t1 + t2)
ENDDO
```

Exemple 4:

Voici un autre exemple, nous permettant d'observer d'autres étapes de l'algorithme. Le code de calcul est le suivant:

```
DO   a = 1,7
  DO   b = 1 , 8 - a
   DO   c = 0,1
    DO     d = 0 ,1 - c
    DO     e = 0,1 - c - d
     DO     f = 0,1 - c - d - e
       T(a - c - d + e + f , b - c + d - e + f) = ...
ENDDO
```

Ce code de calcul correspond au code de calcul de plusieurs références au tableau T que l'on a transformé [§6.2.1] pour pouvoir utiliser les algorithmes de génération du code de transfert pour une seule référence.

La matrice A, le vecteur $\vec{\alpha}$ et les matrices associées à F et f_0 s'écrivent:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} , \quad \vec{\alpha} = \begin{pmatrix} 7 \\ -1 \\ 8 \\ -1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

et

$$F = \begin{pmatrix} 1 & 0 & -1 & -1 & 1 & 1 \\ 0 & 1 & -1 & 1 & -1 & 1 \end{pmatrix} , \quad f_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Calcul de la matrice de Hermite H associée à F (étape 1):

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

Calcul du domaine d'itération dans la nouvelle base (étape 2)

$$A.Q.\vec{t} \leq \vec{\alpha} \iff \begin{pmatrix} 1 & 0 & 1 & 1 & -1 & -1 \\ -1 & 0 & -1 & -1 & 1 & 1 \\ 1 & 1 & 2 & 0 & 0 & -2 \\ 0 & -1 & -1 & 1 & -1 & 1 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \vec{t} \leq \begin{pmatrix} 7 \\ -1 \\ 8 \\ -1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

qui définit le système de contraintes:

$$P = \begin{cases} t_1 + t_3 + t_4 - t_5 - t_6 \leq 7 \\ -t_1 - t_3 - t_4 + t_5 + t_6 \leq -1 \\ t_1 + t_2 + 2.t_3 - 2.t_6 \leq 8 \\ -t_2 - t_3 + t_4 - t_5 + t_6 \leq -1 \\ t_3 \leq 1 \\ -t_3 \leq 0 \\ t_3 + t_4 \leq 1 \\ -t_4 \leq 0 \\ t_3 + t_4 + t_5 \leq 1 \\ -t_5 \leq 0 \\ t_3 + t_4 + t_5 + t_6 \leq 1 \\ -t_6 \leq 0 \end{cases}$$

Projection entière du domaine d'itération selon les n-r dernières variables, soit ici t_3, t_4, t_5, t_6 (étape 3):

Le polyèdre projeté est:

$$PP = \begin{cases} t_2 + t_1 + 4t_3 \leq 10 \\ t_2 + t_1 \leq 10 \\ -2t_6 + 2t_3 + t_2 + t_1 \leq 8 \\ -2t_6 + t_2 + t_1 \leq 8 \\ -t_1 - 2t_3 - t_2 \leq -2 \\ 2t_6 - t_2 \leq 0 \\ -t_2 \leq 0 \\ 2t_6 - t_1 \leq 0 \\ -t_1 \leq 0 \end{cases}$$

Le système résultant de cette projection entière contient encore des contraintes sur les variables t_3 *et* t_6:

– Séparation du système en deux sous-systèmes (étape 5a):

$$S_1 = \begin{cases} t_2 + t_1 \leq 10 \\ -t_2 \leq 0 \\ -t_1 \leq 0 \end{cases} \quad et \quad S_2 = \begin{cases} t_2 + t_1 + 4t_3 \leq 10 \\ -2t_6 + 2t_3 + t_2 + t_1 \leq 8 \\ -2t_6 + t_2 + t_1 \leq 8 \\ -t_1 - 2t_3 - t_2 \leq -2 \\ 2t_6 - t_2 \leq 0 \\ 2t_6 - t_1 \leq 0 \end{cases}$$

L'on ne peut pas déduire de chaque contrainte du système S_2 des inégalités sur les variables de base les plus internes:

$$\begin{cases} -2t_6 + t_2 + t_1 \leq 8 \\ 2t_6 - t_1 \leq 0 \end{cases} \iff \frac{t_2 + t_1 - 7}{2} \leq \frac{t_1}{2}$$

dont on ne peut rien déduire sur t_1 $\Longrightarrow$:

* calcul du plus grand polyèdre PGPC contenu dans le polyèdre projeté, et de l'ensemble des polyèdres constituant le complémentaire PC de PGPC dans PP (étape 5ci).

$$PP = \begin{cases} t_2 + t_1 \leq 10 \\ -t_2 \leq 0 \\ -t_1 \leq 0 \\ t_2 \leq 7 \\ t_1 \leq 7 \end{cases}$$

$$PC = PC_1 \begin{cases} -t_2 \leq -8 \\ t_2 + t_1 + 4t_3 \leq 10 \\ t_2 + t_1 \leq 10 \\ -2t_6 + 2t_3 + t_2 + t_1 \leq 8 \\ -2t_6 + t_2 + t_1 \leq 8 \\ -t_1 - 2t_3 - t_2 \leq -2 \\ 2t_6 - t_2 \leq 0 \\ -t_2 \leq 0 \\ 2t_6 - t_1 \leq 0 \\ -t_1 \leq 0 \end{cases} + PC_2 \begin{cases} -t_1 \leq -8 \\ t_2 + t_1 + 4t_3 \leq 10 \\ t_2 + t_1 \leq 10 \\ -2t_6 + 2t_3 + t_2 + t_1 \leq 8 \\ -2t_6 + t_2 + t_1 \leq 8 \\ -t_1 - 2t_3 - t_2 \leq -2 \\ 2t_6 - t_2 \leq 0 \\ -t_2 \leq 0 \\ 2t_6 - t_1 \leq 0 \\ -t_1 \leq 0 \end{cases}$$

qui après élimination des contraintes redondantes s'écrivent:

$$PC = PC_1 \begin{cases} t_2 \leq 8 \\ -t_2 \leq -8 \\ t_1 \leq 2 \\ -t_1 \leq 0 \\ -t_1 + 2.t_6 \leq 0 \\ t_1 - 2.t_6 \leq 0 \end{cases} + PC_2 \begin{cases} t_1 \leq 8 \\ -t_1 \leq -8 \\ t_2 \leq 2 \\ -t_2 \leq 0 \\ -t_2 + 2.t_6 \leq 0 \\ t_2 - 2.t_6 \leq 0 \end{cases}$$

* Génération du code de transfert du polyèdre PGPC (étape 5cii).
  - Calcul de la nouvelle fonction d'accès aux éléments dans la nouvelle base:

    $\vec{\varphi} = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$

  - Génération du code de transfert correspondant.

```
DO  t2 = 0,7
  DO  t1 = 0 , MIN(7,10 - t2)
    T(t1,t2)  =  TL(t1,t2)
ENDDO
```

* Génération du code de transfert du polyèdre PC (étape 5ciii)
  - Simplification de chaque contrainte conportant une variable supplémentaire.

$$\begin{cases} -t_1 + 2.t_6 \leq 0 \\ t_1 - 2.t_6 \leq 0 \end{cases} \iff \frac{t_1+1}{2} \leq \frac{t_1}{2}$$

  - Génération du code de transfert correspondant.

Pour PC_1

```
DO  t1 = 0,2,2
    T(t1,8)  =  TL(t1,8)
ENDDO
```

Pour PC_2

```
DO  t2 = 0,2,2
  T(8,t2)  =  TL(8,t2)
ENDDO
```

La totalité du code de tranfert pour le quatrième exemple dont voici la représentation est:

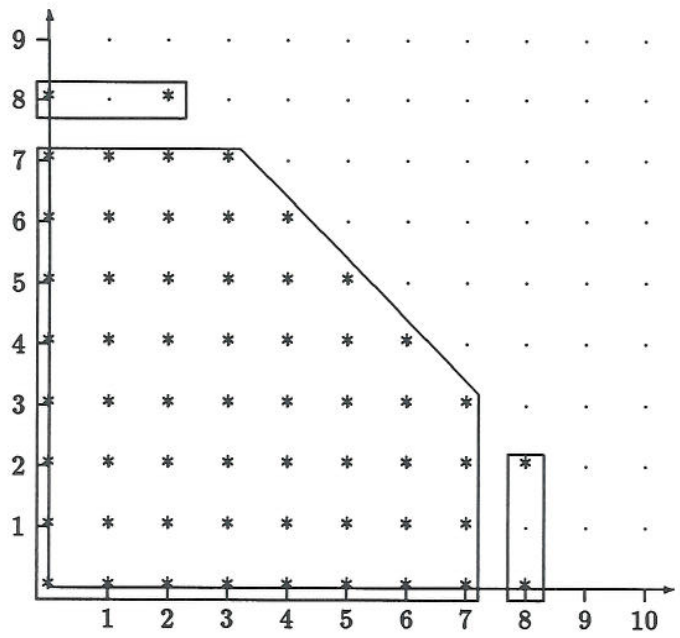


figure 6.1.7

```
DO  t2 = 0,7
  DO  t1 = 0 , MIN(7,10 - t2)
      T(t1,t2)  =  TL(t1,t2)
ENDDO


DO  t1 = 0,2,2
    T(t1,8)  =  TL(t1,8)
ENDDO


DO  t2 = 0,2,2
   T(8,t2)  =  TL(8,t2)
ENDDO
```

Exemple 5:

Voici le cinquième exemple et les différentes étapes de notre algorithme pour le corps de boucles:

```
DO  I = 1,N
  DO  J = 1,M
    DO  K = 1,L
      TL(2 × I + 2 × K, J + K) = ...
ENDDO
```

La matrice A, le vecteur $\vec{\alpha}$ et les matrices associées à F et f_0 s'écrivent:

$$A = \begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \quad , \quad \vec{\alpha} = \begin{pmatrix} -1 \\ N \\ -1 \\ M \\ -1 \\ L \end{pmatrix}$$

et

$$F = \begin{pmatrix} 2 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix} \quad , \quad f_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Calcul de la matrice de Hermite H associée à F (étape 1):

$$F = \begin{pmatrix} 2 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Calcul du domaine d'itération dans la nouvelle base (étape 2)

$$A.Q.\vec{t} \leq \vec{\alpha} \quad \Longleftrightarrow \quad \begin{pmatrix} -1 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \vec{t} \leq \begin{pmatrix} -1 \\ N \\ -1 \\ M \\ -1 \\ L \end{pmatrix}$$

qui définit le système de contraintes:

$$P = \begin{cases} -t_1 + t_3 \leq -1 \\ t_1 - t_3 \leq N \\ -t_2 + t_3 \leq -1 \\ t_2 - t_3 \leq M \\ -t_3 \leq -1 \\ t_3 \leq L \end{cases}$$

Projection entière du domaine d'itération selon les n-r dernières variables, soit ici uniquement t_3 (étape 3):
Le polyèdre projeté est:

$$PP = \begin{cases} -t_1 + t_2 \leq M - 1 \\ -t_2 \leq -2 \\ -t_1 \leq -2 \\ t_1 \leq N + L \\ t_1 - t_2 \leq N - 1 \\ t_2 \leq M + L \end{cases}$$

Le système résultant de cette "projection entière" ne contient plus de contraintes sur la variable t_3.

- Génération du code de transfert:

```
DO  t2 = MAX(-N+3,2), MIN(M+L, M+L+N-1)
  DO  t1 = MAX(2, t2-M+1) , MIN(N+L, t2+N-1)
      T(2×t1, t2) = TL(2×t1, t2)
ENDDO
```

6.2 Génération du code de transfert de la mémoire locale vers la mémoire globale pour deux références au même tableau dans un corps de boucles

Nous présentons dans cette section des algorithmes permettant de générer le code de transfert de la mémoire locale vers la mémoire globale des éléments référencés par deux fonctions d'accès au tableau T.

Nous étudions successivement les cas où:

- les fonctions d'accès au tableau ont le même Z-module et sont en translation (ex: $T(2 \times I)$ et $T(2 \times I + 4)$),
- les fonctions d'accès ont le même Z-module mais ne sont pas en translation (ex: $T(2 \times I, J)$ et $T(2 \times I + 2 \times J, J)$),
- les fonctions d'accès ont des Z-modules différents (ex:$T(3 \times I, J)$ et $T(I, J)$).

6.2.1 Fonctions d'accès ayant le même Z-module et en translation

En calcul numérique, les fonctions d'accès φ_i aux références d'un tableau ne diffèrent souvent que par leur terme constant, il est alors possible et même préférable de ne générer qu'un seul code de transfert.

Le code de calcul a la forme suivante:

```
DO a1 = l1,1 , l2,1
 .....
   DO  an = l1,n(a1 ,.., an-1) , l2,n(a1 ,.., an-1)
     T (φ ( a1 ,.., an ) ) = ...
      T (φ ( a1 ,.., an )  + C1 ) = ...
ENDDO
```

Prenons l'exemple simple:

```
DO  a = 0,N
  DO  b = 0,N
    T(a, b) = ...
    T(a + 2 , b + 2) = ...
ENDDO
```

Si l'on traitait chaque référence séparément, en utilisant l'algorithme de génération automatique du code de transfert décrit précédement, on obtiendrait les deux codes suivants:

```
DO  a = 0,N
 DO  b = 0,N
 TRANSFERER   T(a, b)
```

Puis,

```
DO  a = 0,N
  DO  b = 0,N
  TRANSFERER   T(a + 2 , b + 2)
ENDDO
```

ce qui conduirait à transférer 2 fois les éléments du tableau $T(i,j)$ pour les valeurs de i et j allant de 2 à N.

Pour remédier à ce problème, il faut tenir compte des parties constantes des fonctions φ_i, et rechercher une base pour le Z-module correspondant à l'union des Z-modules générés par chacune de ces fonctions.

La fonction $\varphi' = (\varphi\ (\ a_1\ ,..,\ a_n\ )\ +\ C_1.X_1)$ génère un tel Z-module. En effet, il est facile de voir que le corps de boucles:

```
DO a = 0,N
 DO  b = 0,N
  DO X = 0,1
  TRANSFERER     T(a + 2×X, b + 2×X)
ENDDO
```

fait référence aux même données que le corps de boucles initial.

Lorsque les ensembles référencés par les deux fonctions d'accès ne sont pas disjoints, on utilisera donc le nid de boucles:

```
DO  a1 = f1,1 , f2,1
 .....
   DO  an = f1,n(a1 ,.., an-1) , f2,n(a1 ,.., an-1)
    DO   X1 = 0 , 1
    TRANSFERER      T (φ ( a1 ,.., an )  +  C1 × X1)
ENDDO
```

pour générer le code de transfert du corps de boucles initial.

Ceci nous permet non seulement de générer un code de transfert commun à l'ensemble des fonctions φ_i, mais aussi de pouvoir utiliser l'algorithme que nous avons décrit précédement s'appliquant à un corps de boucles ne contenant qu'une seule référence à un tableau, et donc de générer un code de transfert optimal.

6.2.2 Fonctions d'accès ayant le même Z-module et non en translation

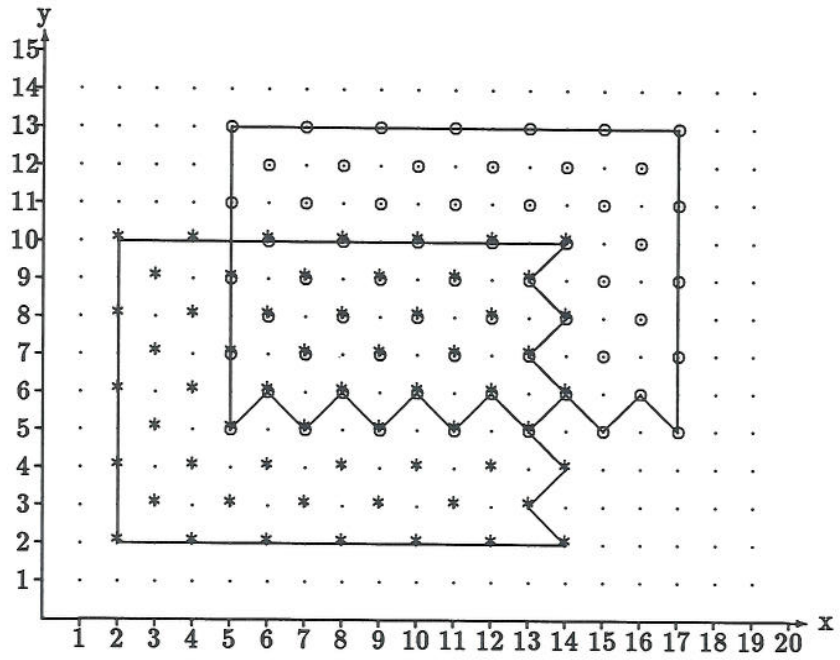


figure 6.2.2

Nous avons un code de calcul qui a la forme suivante:

DO $a_1 = l_{1,1}$, $l_{2,1}$
.....
DO $a_n = l_{1,n}(a_1, .., a_{n-1})$, $l_{2,n}(a_1, .., a_{n-1})$
$T(\varphi_1(a_1, a_2, ..., a_n)) = ...$
$T(\varphi_2(a_1, a_2, ..., a_n)) = ...$
ENDDO

Nous notons P_1 le polyèdre définissant l'ensemble des éléments référencés par la fonction φ_1 et P_2 le polyèdre définissant les éléments référencés par φ_2. Les matrices H_1 et H_2 sont les matrices de Hermite associées respectivement à F_1 et F_2 et P_1, Q_1, P_2, Q_2 les matrices telles que:

$$\varphi_1 = P_1^{-1}.H_1.Q_1^{-1} + f_1 \qquad \text{et} \qquad \varphi_2 = P_2^{-1}.H_2.Q_2^{-1} + f_2.$$

L'espace d'itération est défini par les contraintes $A.\vec{j} \leq \vec{\alpha}$. Les éléments référencés à la fois par la fonction φ_1 et par la fonction φ_2 sont caractérisés par le système:

$$(S) \quad \begin{cases} A.\vec{j_1} \leq \vec{\alpha} \\ A.\vec{j_2} \leq \vec{\alpha} \\ F_1.\vec{j_1} + f_1 = F_2.\vec{j_2} + f_2 \end{cases}$$

Nous dirons que les deux fonctions d'accès définissent le même Z-module si:

$$P_1^{-1}.H_1 \quad = \quad P_2^{-1}.H_2$$

Nous allons distinguer deux cas:

- Le cas où l'intersection des deux polyèdres P_1 et P_2 est vide ((S) n'est pas faisable).

  Dans ce cas, on génère le code de transfert des éléments appartenant aux deux polyèdres P_1 et P_2 séparément, en utilisant l'algorithme présenté au [§6.1.6]

- Le cas où l'intersection des deux polyèdres est non vide ((S) admet au moins une solution)

  Pour éviter de transférer deux fois les mêmes éléments, on copie séparément l'ensemble des éléments communs aux deux polyèdres, puis les éléments restants, non transférés.

  Plus précisément, on limite l'intersection P_i à l'ensemble des éléments appartenant aux plus grands polyèdres convexes contenus dans P_1 et P_2. Cet ensemble est toujours convexe, et ses éléments appartiennent au même Z-module que P_1 et P_2. Cette propriété permet de trouver une projection entière de cet ensemble sur les bases $\vec{j_1}$ et $\vec{j_2}$ sans variables supplémentaires.

  Puis on recherche les éléments appartenant aux complémentaires de l'intersection dans chacun des polyèdres P_1 et P_2, pour transférer les éléments qui n'appartiennent pas à l'intersection. Ces ensembles peuvent être calculés en utilisant l'algorithme présenté au [§4.3].

  L'algorithme de génération du code de transfert des données référencées par deux fonctions définissant le même Z-module est le suivant:

  1. **Calcul du plus grand polyèdre convexe contenu dans P_1 : $PGPC_1$**
  2. **Calcul du plus grand polyèdre convexe contenu dans P_2 : $PGPC_2$**
  3. **Calcul de l'intersection de $PGPC_1$ et$PGPC_2$: $PGPC_i$.**
  4. **Génération du code transfert du polyèdre convexe $PGPC_i$ (dans l'une des bases) en utilisant l'algorithme présenté au [§6.1.6].**
  5. **Projection entière de l'intersection $PGPC_i$ sur la base $\vec{j_1}$: PR_1.**
  6. **Calcul du complémentaire de PR_1 dans P_1:PC_1**
  7. **Génération du code de transfert des éléments de l'ensemble des polyèdres constituant PC_1.**
  8. **Projection entière de l'intersection $PGPC_i$ sur la base $\vec{j_2}$: PR_2.**
  9. **Calcul du complémentaire de PR_2 dans P_2:PC_2**
  10. **Génération du code de transfert des éléments de l'ensemble des polyèdres constituant PC_2.**

6.2.3 Fonctions d'accès possédant deux Z-modules différents

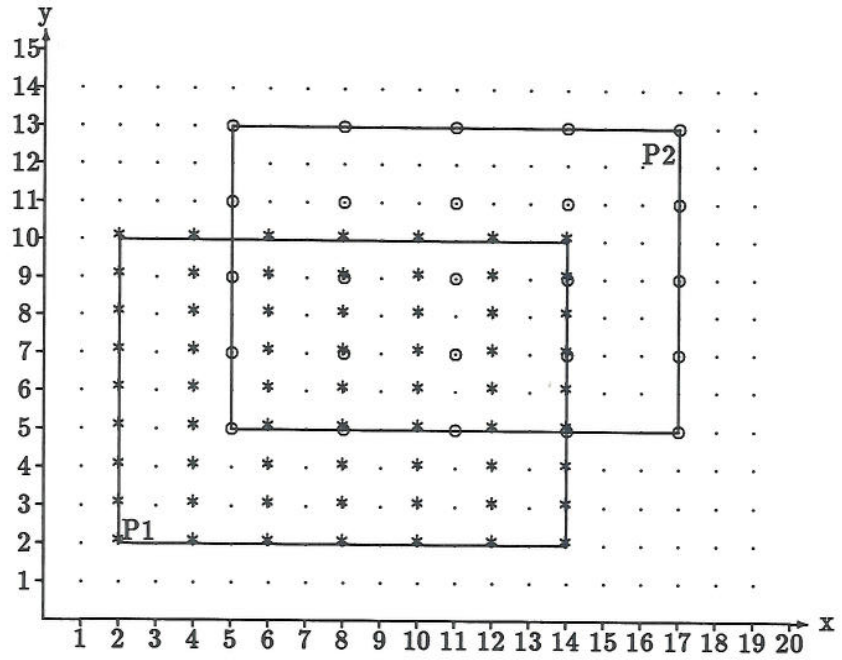


figure 6.2.3

Le code de calcul a la même forme que précédemment.

DO $a_1 = l_{1,1}$, $l_{2,1}$

.....

DO $a_n = l_{1,n}(a_1, .., a_{n-1})$, $l_{2,n}(a_1, .., a_{n-1})$

$T(\varphi_1(a_1, a_2, ..., a_n)) = ...$

$T(\varphi_2(a_1, a_2, ..., a_n)) = ...$

ENDDO

Cependant les éléments référencés par φ_1 et φ_2 n'appartiennent pas au même Z-module. Deux choses importantes en découlent:

- Les éléments appartenant à l'intersection de P_1 et P_2 appartiennent au Z-module "intersection" des deux Z-modules.
- L'ensemble des éléments de P_1 qui n'appartiennent pas à l'intersection de P_1 et P_2 est constitué non seulement de tous les éléments extérieurs au domaine "intersection", mais aussi des éléments qui appartiennent à ce domaine et qui sont générés par le Z-module de P_1 moins le Z-module "intersection".

La différence de deux Z-modules n'étant en général pas un Z-module, il est difficile de générer automatiquement cet ensemble. Nous transférerons donc, dans ce cas, chaque référence au tableau T séparément, en utilisant le code de transfert pour une référence présenté au [§6.1.6].

6.2.4 Fonctions d'accès possédant des Z-modules inclus

Cependant nous devons distinguer le cas particulier où l'un des Z-modules est inclus dans l'autre.

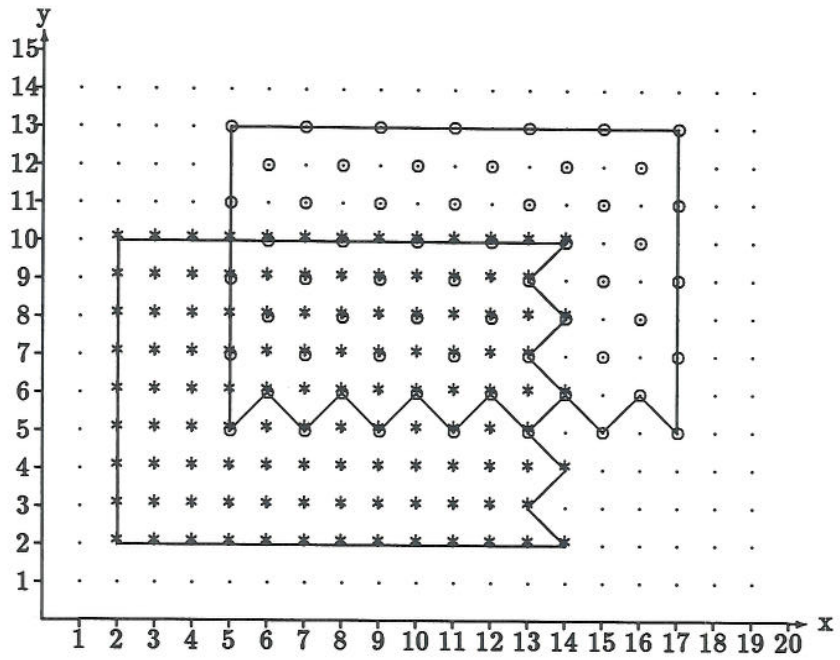


figure 6.2.4

Soit T_1 le Z-module de base $P_1^{-1}H_1$ générant les éléments de P_1 et T_2 le Z-module de base $P_2^{-1}H_2$ générant les éléments de P_2. Rappelons que T_1 est inclus dans T_2 s'il existe un vecteur entier $\vec{k}$ tel que: $|k_i| \geq 1$, *et* $P_1^{-1}H_1 = \vec{k}^t.P_2^{-1}H_2$

Dans ce cas, dans le domaine "intersection", tous les éléments de P_1 appartiennent aussi à P_2. Le complémentaire de l'intersection de P_1 et P_2 dans P_1 se calcule donc en utilisant le même algorithme que précédemment présenté en [§[4.3]].

L'algorithme de génération du transfert des éléments de P_1 et P_2 est alors le suivant, si T_1 est inclus dans T_2:

1. **Génération du code de transfert de l'ensemble des éléments appartenant au polyèdre P_2.**
2. **Calcul du plus grand polyèdre convexe contenu dans P_1 : $PGPC_1$**
3. **Calcul du plus grand polyèdre convexe contenu dans P_2 : $PGPC_2$**
4. **Calcul de l'intersection de $PGPC_1$ et de $PGPC_2$:P_i**
5. **Projection entière de l'intersection $PGPC_i$ sur la base $\vec{j_1}$: PR_1.**
6. **Calcul du complémentaire de PR_1 dans P_1:PC_1**
7. **Génération du code de transfert des éléments de l'ensemble des polyèdres constituant PC_1.**

6.3 Génération du code de transfert de la mémoire locale vers la mémoire globale pour plusieurs références au même tableau dans un corps de boucle

Nous présentons dans cette section des algorithmes permettant de générer le code de transfert, de la mémoire locale vers la mémoire globale, des éléments référencés par plusieurs fonctions d'accès au tableau T.

Nous distinguons ici encore les cas où:

- les fonctions possèdent le même Z-module et sont en translation,
- les fonctions possèdent le même Z-module et ne sont pas en translation,
- les fonctions ont des Z-modules inclus,
- les fonctions ont des Z-modules différents

6.3.1 Fonctions d'accès ayant le même Z-module et en translation

Le code de calcul a la forme suivante:

```
DO  a1 = f1,1 , f2,1
    .....
   DO   an = f1,n(a1 ,.., an-1) , f2,n(a1 ,.., an-1)
     T (φ ( a1 ,.., an ) )= ...
     T (φ ( a1 ,.., an )  + C1 )= ...
     T (φ ( a1 ,.., an )  + C2 )= ...
      ...
     T (φ ( a1 ,.., an )  + Cm )= ...
```

Rappelons que nous transférons séparément les ensembles disjoints d'éléments référencés par des fonctions d'accès au tableau T.

Nous déduisons de l'algorithme de génération automatique du code de transfert pour deux références en translation, le code suivant:

```
DO  a1 = f1,1 , f2,1
      .....
  DO   an = f1,n(a1 ,.., an-1) , f2,n(a1 ,.., an-1)
    DO   X1 = 0 , 1
      DO  X2 = 0 ,1 - X1
      ...
        DO  Xm = 0,1 - X1 - X2 - Xm-1
      TRANSFERER   T (φ ( a1 ,.., an )  + C1 × X1  + C2 × X2  + ... + Cm × Xm )
ENDDO
```

qui permet de référencer exactement les mêmes éléments que le code initial.

Pour générer le code de tranfert des éléments référencés par plusieurs fonctions d'accès ne différant que par leur terme constant, nous appliquons l'algorithme de génération du code de transfert pour une référence dans un corps de boucle au code de calcul précédent.

6.3.2 Fonctions d'accès possédant le même Z-module et non en translation

Pour minimiser au mieux les transferts des données dans le cas de plusieurs références dans un même corps de boucles, il faudrait:

Effectuer le transfert des données des deux premiers ensembles référençant les éléments du tableau en utilisant les algorithmes présentés précédemment.

Puis pour chaque référence supplémentaire, caractérisant un nouveau Z-polyèdre P_j:

- Calculer l'intersection P_i de P_j avec le premier Z-polyèdre transféré P_1,
- Calculer le complémentaire P_C de P_i dans P_1,
- puis pour ne pas transférer des éléments communs à P_2, déjà transféré:

  Pour chacun des Z-polyèdres P_{C_i} constituant P_C

  - Calculer l'intersection de P_{C_i} avec P_2,
  - Calculer le complémentaire $P_{C'}$ de P_{C_i} dans P_2,
  - puis pour ne pas transférer des éléments communs avec P_3, déjà transféré:
    * Calculer l'intersection de $P_{C'_i}$ avec P_3,
    * Calculer le complémentaire de $P_{C'_i}$ dans P_3,
    * ...

      Enfin, transférer les ensembles constituant les $P_{C'}$.

Cependant cet algorithme est très coûteux, nous proposons donc l'algorithme suivant qui est moins efficace dans certain cas, mais plus rapide.

Algorithme:

Effectuer le transfert des données des deux premiers ensembles référençant les éléments du tableau en utilisant les algorithmes présentés précédemment.

Puis pour chaque référence supplémentaire, caractérisant un nouveau Z-polyèdre P_j:

1. **Calculer l'intersection P_i de P_j avec chaque Z-polyèdre P_k déjà transféré.**
2. **Calculer le volume de chaque P_i**
3. **Conserver le Z-polyèdre P_g dont l'intersection avec P_j est la plus importante (s'il existe),**
4. **Calculer le complémentaire P_C de P_{i_g} dans P_j,**
5. **Générer le code de transfert des éléments de chacun des ensembles constituant P_C.**

6.3.3 Fonctions d'accès ayant des Z-modules inclus

Nous traiterons à part le cas des fonctions d'accès aux éléments d'un tableau dont les Z-modules sont inclus et utiliserons l'algorithme présenté en [§6.2.4].

6.3.4 Fonctions d'accès ayant des Z-modules différents

Dans ce cas, nous traitons chaque référence contenue dans le corps de boucles séparément, en utilisant le code de transfert pour une référence à un tableau T présenté en [§6.1.6].

6.4 Génération du code de transfert de la mémoire globale vers la mémoire locale pour une seule référence à un tableau dans le corps de boucles

Pour transférer l'ensemble des éléments **utilisés** par une tâche de la mémoire globale vers la mémoire locale (du processeur où est exécutée la tâche) plusieurs codes de transfert sont envisageables. Il n'est pas nécessaire de copier dans la mémoire locale l'ensemble exact des éléments référencés par le processeur, on peut en copier davantage. De plus, il est souvent plus facile de transférer un ensemble convexe approchant un ensemble de points entiers, que l'ensemble lui-même. Un ensemble convexe se caractérise aisément par un corps de boucles, et le calcul de son volume est plus simple.

Soit P l'ensemble des points référencés par une fonction d'accès au tableau T, et PPPC le plus petit polyèdre convexe englobant cet ensemble. Le nombre d'éléments qui appartiennent au complémentaire de P dans PPPC est généralement très inférieur au nombre total des éléments de P.

Nous avons donc choisi de transférer de la mémoire globale vers la mémoire locale le plus petit polyèdre convexe englobant l'ensemble des éléments référencés par la fonction d'accès à T. Il permet de transfèrer un ensemble convexe de points entiers, contenant l'ensemble des points P qui nous intéressent, sans ajouter trop de points supplémentaires.

6.4.1 Algorithme de génération du code de transfert pour une référence dans un corps de boucles

L'algorithme de génération du code de transfert des éléments référencés par une fonction d'accès au tableau T est le suivant:

1. **Calcul de la matrice de Hermite H de dimension r associée à F et des matrices P et Q telles que:** $F = P^{-1}.H.Q^{-1}$
2. **Calcul du domaine d'itération dans la nouvelle base** $\vec{t} = Q^{-1} . \vec{j}$: $A.Q.\vec{j} \leq \vec{\alpha}$
3. **Projection selon Fourier** [1] **du domaine d'itération selon les n-r dernières variables** t_{r+1}, t_{r+2}, ..., *et* t_n **de** $\vec{t}$ **pour obtenir le domaine projeté PP.**
4. **Calculer les nouvelles bornes du** Z**-polyèdre projeté, par projections successives de la première variable** t_1 **jusqu'à la dernière** t_r.
5. **Calculer la nouvelle fonction d'accès aux éléments dans la nouvelle base:** $\vec{\varphi} = P^{-1} .H . \vec{t} + \vec{f}_0$.
6. **Générer le code de transfert correspondant.**

[1] nous soulignons les différences avec la génération du code de recopie présentée précédemment

6.5 Génération du code de transfert de la mémoire globale vers la mémoire locale pour deux références au même tableau dans un corps de boucles

Pour le transfert de la mémoire globale vers la mémoire locale des éléments référencés par plusieurs fonctions, on peut envisager une solution simple: le transfert des éléments appartenant à l'enveloppe convexe de l'ensemble des données référencées par les différentes fonctions. Cependant le nombre des éléments de cette enveloppe convexe peut être très vite supérieur au nombre des données réellement utilisées (surtout lorsque la dimension de l'ensemble des éléments référencés est supérieure ou égale à 3).

La taille des mémoires locales étant à l'heure actuelle relativement petite, nous avons choisi une solution permettant de transférer moins d'éléments, mais qui génére parfois un peu plus d'overhead de contrôle.

Les algorithmes présentés dans cette partie sont tous inspirés des algorithmes de génération du code de transfert de la mémoire locale vers la mémoire globale, nous reprenons donc tous les résultats présentés dans le chapitre précédent.

Nous distinguons les cas où:

- les fonctions possèdent le même Z-module et sont en translation,
- les fonctions possèdent le même Z-module et ne sont pas en translation,
- les fonctions ont des Z-modules inclus,
- les fonctions ont des Z-modules différents

Comme dans le cas des recopies de la mémoire locale vers la mémoire globale, lorsque les ensembles d'éléments référencés par les deux fonctions sont disjoints, on transfère séparément les deux ensembles.

6.5.1 Fonctions d'accès ayant le même Z-module et en translation

Lorsque les fonctions sont en translation, le code de calcul a la forme suivante:

```
DO  a1 = f1,1 , f2,1
 .....
   DO  an = f1,n(a1 ,.., an-1) , f2,n(a1 ,.., an-1)
  ...   = F'( T (φ ( a1 ,.., an ) ) )
  ...   = F''( T (φ ( a1 ,.., an ) + C1 ))
ENDDO
```

Nous reprenons ici les résultats trouvés en [§6.2.1] et déduisons le code de calcul suivant:

```
DO  a_1 = f_{1,1} , f_{2,1}
 .....
    DO   a_n = f_{1,n}(a_1 ,.., a_{n-1}) , f_{2,n}(a_1 ,.., a_{n-1})
     DO  X_1 = 0 , 1
    ...  = F'(  T  (φ ( a_1 ,.., a_n )  +  C_1 × X_1))
ENDDO
```

qui fait référence aux mêmes éléments que le code initial.

Nous utilisons, dans le cas de deux références en translation, ce nouveau code de calcul et l'algorithme de génération du code de transfert pour une référence décrit en [§6.4.1] pour générer le code de transfert des données utilisées.

6.5.2 Fonctions d'accès ayant le même Z-module et non en translation

Losque les fonctions ont le même Z-module et ne sont pas en translation, le code de calcul a la forme suivante:

```
DO  a_1 = f_{1,1} , f_{2,1}
 .....
   DO   a_n = f_{1,n}(a_1 ,.., a_{n-1}) , f_{2,n}(a_1 ,.., a_{n-1})
   ... =  F' ( T(φ_1(a_1, a_2, ..., a_n))
   ... =  F'' ( T(φ_2(a_1, a_2, ..., a_n))
ENDDO
```

Pour éviter de transférer trop de fois les mêmes éléments, on copie séparément l'ensemble des éléments communs aux deux Z-polyèdres, puis les éléments restants, non copiés.

Plus précisément, on étend l'intersection P_i à l'ensemble des éléments appartenant au plus petit polyèdre convexe englobant P_1 et P_2. Cet ensemble est toujours convexe, et ces éléments appartiennent au même Z-module que P_1 et P_2.

Puis on recherche les éléments appartenant aux complémentaires de l'intersection P_i dans chacun des Z-polyèdres P_1 et P_2, pour transférer les éléments qui n'appartiennent pas à l'intersection. Ces ensembles peuvent être calculés en utilisant l'algorithme présenté en [§4.3].

L'algorithme de génération du code de transfert des données référencées par deux fonctions définissant le même Z-module est le suivant:

Algorithme:

1. **Calcul du plus petit polyèdre convexe englobant P_1 : $PPPC_1$**
2. **Calcul du plus petit polyèdre convexe englobant P_2 : $PPPC_2$**
3. **Calcul de l'intersection de $PPPC_1$ *et* $PPPC_2$: $PPPC_i$.**
4. **Génération du code transfert du Z-polyèdre convexe $PPPC_i$ (dans l'une des bases) en utilisant l'algorithme présenté en [§6.4.1]**
5. **Projection selon Fourier de l'intersection $PPPC_i$ sur la base $\vec{j_1}$: PR_1.**
6. **Calcul du complémentaire de PR_1 dans P_1:PC_1**
7. **Génération du code de transfert des éléments de l'ensemble des Z-polyèdres constituant PC_1.**
8. **Projection selon Fourier de l'intersection $PPPC_i$ sur la base $\vec{j_2}$: PR_2.**
9. **Calcul du complémentaire de PR_2 dans P_2:PC_2**
10. **Génération du code de transfert des éléments de l'ensemble des Z-polyèdres constituant PC_2.**

6.5.3 Fonctions d'accès ayant deux Z-modules inclus

Nous utilisons le même type d'algorithme que celui utilisé pour le code de recopie.
L'algorithme de génération du transfert des éléments de P_1 et P_2 est :
Si T_1 est inclus dans T_2.

1. **Génération du code de transfert de l'ensemble des éléments appartenant au Z-polyèdre P_2.**
2. **Calcul du plus petit polyèdre convexe englobant P_1 : $PPPC_1$**
3. **Calcul du plus petit polyèdre convexe englobant P_2 : $PPPC_2$**
4. **Calcul de l'intersection de $PPPC_1$ et de $PPPC_2$:P_i**
5. **Projection selon Fourier de l'intersection $PPPC_i$ sur la base $\vec{j_1}$: PR_1.**
6. **Calcul du complémentaire de PR_1 dans P_1:PC_1**
7. **Génération du code de transfert des éléments de l'ensemble des Z-polyèdres constituant PC_1.**

6.5.4 Fonctions d'accès ayant des Z-modules différents

Rappelons les deux choses importantes qui découlent du fait que les Z-modules sont différents:

- Les éléments appartenant à lintersection de P_1 et P_2 appartiennent au Z-module "intersection" des deux Z-modules.
- L'ensemble des éléments de P_1 qui n'appartiennent pas à l'intersection de P_1 et P_2 est constitué non seulement de tous les éléments extérieurs au domaine "intersection", mais aussi des éléments qui appartiennent à ce domaine et qui sont générés par le Z-module de P_1 moins le Z-module "intersection".

Soit PU_i le Z-polyèdre définissant les points entiers dont les frontières sont celles de l'intersection des Z-polyèdres P_1 et P_2 et dont le Z-module est le Z-module "union" des deux Z-modules de P_1 et P_2. Les coefficients des vecteurs du Z-module qui permettent de génèrer, entre autres, les éléments de PU_i sont les p.g.c.d. des coefficients des vecteurs des Z-modules de P_1 et P_2.

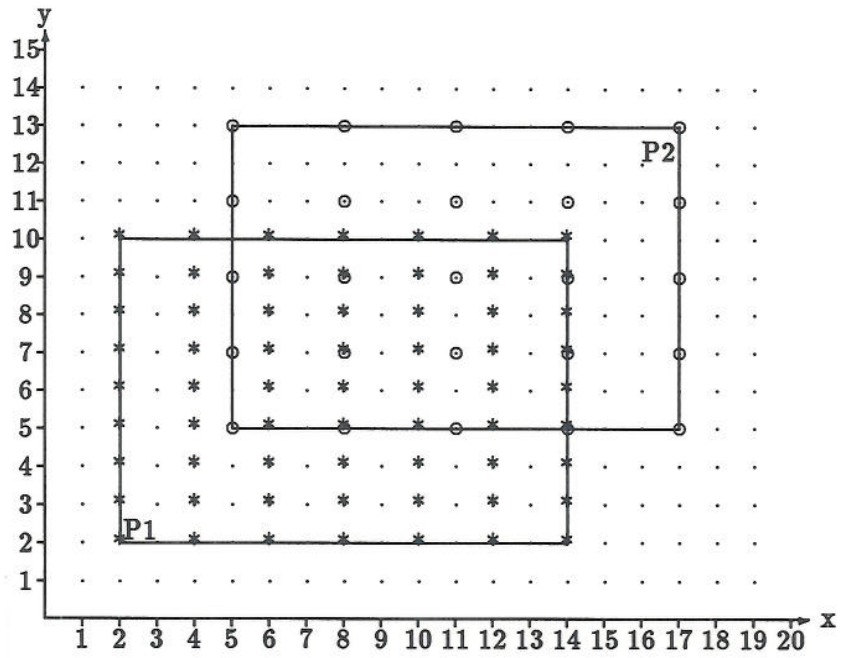


figure 6.5.4

Sur notre figure, le Z-module qui permet de génèrer les éléments de PU_i a pour vecteurs de base: (1,0) et (0,1).

Lorsque la dimension de l'ensemble des éléments référencés par T est supérieure à 3, le nombre des éléments appartenant à PU_i est souvent très supérieur au nombre des éléments appartenant à l'intersection de P_1 et P_2.

Pour cette raison, nous avons choisi de distinguer deux cas:

- Le cas où la dimension de l'ensemble des points référencés (Dim(H)) est supérieure ou égale à 3.

  Nous transférons, dans ce cas, chaque référence au tableau T séparément, en utilisant le code de transfert pour une référence présenté en [§6.4.1]

- Le cas où la dimension de l'ensemble des points référencés (Dim(H)) est inféreure à 3.

  Nous proposons, dans ce cas, de transférer l'ensemble des points se trouvant dans le domaine d'intersection, puis les autres points de P_1 et P_2 appartenant aux complémentaires de cette intersection dans les deux Z-polyèdres.

  L'algorithme est le suivant:

  1. Calcul du plus petit polyèdre convexe englobant P_1 : $PPPC_1$
  2. Calcul du plus petit polyèdre convexe englobant P_2 : $PPPC_2$
  3. Génération du code de transfert de l'ensemble des points appartenant à PU_i:
     (a) Calcul des bornes inférieures et supérieures des indices du tableau pour le domaine intersection.
     (b) Calcul du Z-module ''union'' T_u des Z-modules de P_1 et P_2.
     (c) Génération du code de transfert de l'ensemble des points contenus dans ces nouvelles frontières avec le nouveau Z-module T_u.
  4. Génération du code de transfert des points appartenant aux complémentaires de l'intersection dans les Z-polyèdres P_1 et P_2:
     (a) Calcul de l'intersection de $PPPC_1$ et$PPPC_2$: $PPPC_i$.
     (b) Projection selon Fourier de l'intersection $PPPC_i$ sur la base $\vec{j_1}$: PR_1.
     (c) Calcul du complémentaire de PR_1 dans P_1:PC_1
     (d) Génération du code de transfert des éléments de l'ensemble des Z-polyèdres constituant PC_1.
     (e) Projection selon Fourier de l'intersection PPPsur la base $\vec{j_2}$: PR_2.
     (f) Calcul du complémentaire de PR_2 dans P_2:PC_2
     (g) Génération du code de transfert des éléments de l'ensemble des Z-polyèdres constituant PC_2.

6.6 Génération du code de transfert de la mémoire globale vers la mémoire locale des éléments référencés par plusieurs fonctions d'accès dans le corps de boucles

Nous présentons dans cette section des algorithmes permettant de générer le code de transfert de la mémoire globale vers la mémoire locale des éléments référencés par plusieurs fonctions d'accès au tableau T.

Nous distinguons ici encore les cas où:

- les fonctions possèdent le même Z-module et sont en translation,
- les fonctions possèdent le même Z-module et ne sont pas en translation,
- les fonctions ont des Z-modules inclus,
- les fonctions ont des Z-modules différents

6.6.1 Fonctions d'accès ayant le même Z-module et en translation

Le code de calcul a la forme suivante:

```
DO  a1 = f1,1 , f2,1
 .....
  DO   an = f1,n(a1 ,.., an-1) , f2,n(a1 ,.., an-1)
       T (φ ( a1 ,.., an ) ) = ...
       T (φ ( a1 ,.., an )  + C1 ) = ...
       T (φ ( a1 ,.., an )  + C2 ) = ...
          ...
       T (φ ( a1 ,.., an )  + Cm ) = ...
ENDDO
```

Pour générer le code de tranfert des éléments référencés par plusieurs fonctions d'accès non en translation, nous appliquons l'algorithme de génération du code de transfert pour une référence dans un corps de boucles au code de calcul suivant:

```
DO  a1 = f1,1 , f2,1
 .....
 DO   an = f1,n(a1 ,.., an-1) , f2,n(a1 ,.., an-1)
  DO  X1 = 0 , 1
   DO   X2 = 0 ,1 - X1
  ...
    DO   Xm = 0,1 - X1 - X2 - Xm-1
     TRANSFERER   T (φ ( a1 ,.., an )  + C1 × X1  + C2 × X2  +...+  Cm × Xm )
ENDDO
```

6.6.2 Fonctions d'accès possédant le même Z-module et non en translation

L'algorithme que nous proposons pour effecter le transfert des données référencés par deux fonctions d'accès possédant le même Z-module est identique à celui proposé pour le code de recopie.

6.6.3 Fonctions d'accès ayant des Z-modules inclus

Nous traitons à part le cas des fonctions d'accès aux éléments d'un tableau dont les Z-modules sont inclus et utiliserons l'algorithme présenté en [§6.5.3].

6.6.4 Fonctions d'accès ayant des Z-modules différents

Dans ce cas, nous traitons chaque référence contenue dans le corps de boucles séparément, en utilisant le code de transfert pour une référence à un tableau T présenté en [§6.4.1].

6.7 Conclusion

L'étude effectuée par Shen, Li et Yew [SLYe88] a montré que 53% des fonctions d'accès aux références d'un tableau sont totalement linéaires, les expressions non linéaires étant à 96% des expressions comportant des constantes symboliques. L'utilisation des méthodes qui permettent de manipuler ces constantes symboliques comme des variables particulières du polyèdre permet de traiter efficacement environ 90% des références aux tableaux usuelles.

Que l'ensemble des éléments référencés par la tâche soit convexe ou non, le code de transfert généré de la mémoire locale vers la mémoire globale pour une référence à un tableau dans un corps de boucles est toujours optimal: on transfère exactement une seule fois les éléments qui ont été modifiés par le processeur.

La généralisation de cet algorithme au transfert des éléments référencés par plusieurs fonctions d'accès au même tableau conduit parfois à un code très complexe, on recherche donc dans ce cas un compromis entre la minimisation du nombre d'éléments à transférer, et l'overhead de contrôle.

Cependant, dans de nombreux cas le nombre de références que nous avons choisi de transférer est pratiquement optimal.

Voici un tableau récapitulatif des différents cas étudiés pour le code de transfert de la mémoire locale vers la mémoire globale:

Z-modules	Z-polyèdres projetés	1 référence	2 références	plusieurs références
même Z-module	en translation	-	**** [1]	****
	convexes	****	****	** [2]
	non convexes	****	*** [3]	**
Z-modules différents	convexes	-	* [4]	*
	non convexes	-	*	*
Z-modules inclus	convexes	-	****	**
	non convexes	-	***	**
quelconques	disjoints	-	****	****

Les algorithmes présentés, pour transférer les données utilisées par une tâche de la mémoire globale à la mémoire locale sont basés sur les même principes que ceux utilisés pour la génération du code de recopie. On approxime, dans les deux cas, les Z-polyèdres non convexes par des polyèdres pour faciliter la génération du code.

L'avantage important du code de copie est de pouvoir approximer l'ensemble des éléments par un seul polyèdre, englobant l'ensemble des éléments référencés. Le code de transfert généré est simple car il ne comporte ni divisions entières, ni boucles sur des variables supplémentaires.

Les tailles des mémoires locales étant encore relativement petites, nous avons choisi de ne pas approximer l'ensemble de tous les éléments référencés par plusieurs fonctions d'accès au tableau par leur enveloppe convexe. Nos algorithmes permettent de transférer un nombre plus réduit d'éléments mais augmentent un peu l'overhead de contrôle.

Si la taille des mémoires locales est trop réduite, il est toujours possible d'utiliser les algorithmes de génération du code de transfert de la mémoire locale vers la mémoire globale pour transférer les éléments utilisés par la tâche.

[1] ****** optimal:** on transfère une seule fois chaque élément référencé par la tâche.

[3] ***** presque optimal:** le volume des éléments appartenant au complémentaire du plus grand polyèdre convexe contenu dans le Z-polyèdre non convexe est transféré deux fois. Ce volume est en général très faible par rapport au volume total des éléments à transférer.

[2] **** quelconque:** le code de transfert dépend du code de calcul. Au mieux les ensembles référencés par les fonctions d'accès ont tous la même intersection et on génère un code optimal, au pire les intersections des ensembles référencés par chacune des fonctions d'accès au tableau sont transférées plusieurs fois.

[4] *** séparés :** chaque transfert s'effectue séparément. L'intersection des ensembles référencés par les différentes fonctions n'étant pas vide, le nombre total des éléments transférés n'est pas optimal.

Chapitre 7

Généralisation à un module, extension des hypothèses d'application des algorithmes et développement

Ce chapitre présente les extensions des hypothèses d'application de nos algorithmes présentées au chapitre 3.

Nous développons successivement les cas où nous avons:

- un test dans le code de la tâche,
- un domaine d'itération pseudo-linéaire,
- une fonction d'accès aux éléments du tableau non linéaire,
- des accès au tableau dans des corps de boucles différents,
- un appel de sous programme dans la tâche,

7.1 Extension à certains domaines d'itération non linéaires

L'extension à certains domaines d'itération non linéaires n'est possible que si les expressions non linéaires peuvent être remplaçées par une suite de contraintes linéaires.

7.1.1 Tests IF

Pour détecter si une tâche contenant un test peut être exécutée en parallèle, les paralléliseurs ont besoin de connaître les effets de ce test sur les instructions de la

tâche, et principalement de connaître les domaines des tableaux référencés par ce test. Pour calculer ces effets, les paralléliseurs supposent que les fonctions de test sont linéaires.

Nous ne traitons donc que les tests linéaires. Ces derniers se traduisent directement sous la forme de contraintes linéaires que nous ajoutons au système de contraintes du domaine d'itération.

Si les références au tableau en écriture dépendent d'un test, les transferts de la mémoire locale vers la mémoire globale en dépendent aussi.

Nous générerons, dans ce cas, un code de transfert des éléments modifiés par la tâche de la mémoire locale vers la mémoire globale pour chacune des parties du test.

Les transferts de la mémoire globale vers la mémoire locale n'ont pas besoin d'être multipliés (puisque l'on peut copier plus de données que celles réellement utilisées). Nous ne générerons donc qu'un seul code de transfert de la mémoire globale vers la mémoire locale, au début de la tâche.

Exemple

```
TASK  T
REAL TL(0:11,1:20)     C
C Copie de la memoire globale vers la memoire locale (Ref.2)
C
DO J= 1,20
   DO I=0,11
         TL(I,J) =  T(I,J)
   ENDDO
ENDDO
C
C Code de calcul
C
DO A=1,10
  DO B=1,20
     IF (A <= N)
         TL(A+1,B) = TL (A,B)+1
     ELSE
         TL(A-1,B) = TL (A,B)-1
  ENDDO
ENDDO
C
C Copie de la memoire locale vers la memoire globale (partie 1 du test IF)
C
DO J=1,20
  DO I=1,MIN(10,N)
       T(I+1,J) = TL(I+1,J)
  ENDDO
ENDDO
```

```
C
C Copie de la memoire locale vers la memoire globale (partie 2 du test IF)
C
DO J=1,20
  DO I=MAX(1,N+1),10
        T(I-1,J) = TL(I-1,J)
  ENDDO
ENDDO
C
```

7.1.2 Fonction MODULO dans un test IF

Proposition:

On peut remplacer une expression comportant une fonction MODULO du type:
$IF\ (MODULO(X,d) \leq A)$, *par les deux contraintes:*

$$\begin{cases} X - A \leq k\,d \\ k\,d \leq X \end{cases}$$

où k est une entière variable supplémentaire que l'on introduit dans le système auquel on devait ajouter le test MODULO.

Démonstration:

$$MODULO(X,d) \leq A \Rightarrow X - (\frac{X}{d})\,d \leq A$$

$$\Rightarrow \quad X - A \leq d\,(\frac{X}{d}) \Leftrightarrow \frac{X - A + d - 1}{d} \leq \frac{X}{d}$$

Nous introduisons une nouvelle variable entière **k**,

$$IF\ (MODULO(X,d) \leq A) \Leftrightarrow \exists\ k/ \begin{cases} X - A \leq k\,d \\ k\,d \leq X \end{cases}$$

c.q.f.d.

De même on remplacera une expression comportant une fonction MODULO du type:
$IF\ (\ A \leq MODULO(X,d)\ )$ par les deux contraintes:

$$\begin{cases} k\,d \leq X - A \\ X \leq k\,d + d - 1 \end{cases}$$

7.2 Extension à certaines fonctions d'accès non linéaires

Il est possible d'étendre les algorithmes de génération de code présentés précédemment à certaines fonctions φ pseudo-linéaires d'accès au tableau. Afin de permettre la réutilisation des algorithmes, ces fonctions doivent posséder une fonction d'accès linéaire équivalente ayant les même caractéristiques (même Z-module, même base de parcours).

Rappelons que la fonction φ d'accès aux éléments du tableau s'écrit: $\vec{\varphi} = F \,.\, \vec{j} + \vec{f_0}$

Les fonctions dont la matrice associée F se décompose en un produit de matrices F_1 et F_2, où F_1 est une matrice diagonale contenant sur la diagonale les constantes symboliques responsables de la non linéarité de la fonction, font partie de ces fonctions pseudo-linéaires que nous savons traiter.

En effet, dans ce cas, la fonction φ d'accès aux éléments du tableau se décompose de la manière suivante:

$$\varphi = F\vec{j} + \vec{f_0} = F_1F_2\vec{j} + \vec{f_0} = \begin{pmatrix} \alpha_{11} & 0 & 0 \\ 0 & \alpha_{22} & 0 \\ \cdots & \cdots & \cdots \\ 0 & 0 & \alpha_{nn} \end{pmatrix} F_2\vec{j} + \vec{f}$$

$$= \begin{pmatrix} \alpha_{11} & 0 & 0 \\ 0 & \alpha_{22} & 0 \\ \cdots & \cdots & \cdots \\ 0 & 0 & \alpha_{nn} \end{pmatrix} P_2^{-1}H_2Q_2^{-1}\vec{j} + \vec{f_0} = \begin{pmatrix} \alpha_{11} & 0 & 0 \\ 0 & \alpha_{22} & 0 \\ \cdots & \cdots & \cdots \\ 0 & 0 & \alpha_{nn} \end{pmatrix} P_2^{-1}H_2\vec{t_2} + \vec{f_0}$$

La matrice F_1 sert à caractériser la base du Z-module qui vaut selon la base $\vec{t_2}$: $F_1P_2^{-1}H_2$

La base de parcours $\vec{t_2}$ des éléments référencés par cette fonction ne dépend pas de F_1 mais uniquement de F_2. Il en est de même du calcul des bornes du Z-polyèdre caractérisant les éléments référencés dans la nouvelle base $\vec{t_2}$.

Nous étendons donc nos hypothèses de linéarité, à la génération automatique du code de transfert des éléments référencés par une référence, aux fonctions d'accès dont les indices sont multiples d'une constante symbolique entière et d'une combinaison linéaire des indices de boucle.

Si la matrice F_1 n'est pas une matrice diagonale, elle intervient alors dans le choix de la base de parcours des éléments référencés par la fonction et dans le calcul des bornes de l'espace d'itération de ces vecteurs de base. Comme nous ne savons pas traiter automatiquement et efficacement les opérations de base sur les matrices symboliques, il n'est pas possible d'étendre nos algorithmes à ce type de fonctions.

Exemple

Si le code de calcul est:

```
DO I = 1, N
  DO J = 1, M

    T(K × I + 2 × K, 3 × L J + 5, I + J) = ...
ENDDO
```

On associera à la fonction d'accès les matrices suivantes:

$$F = \begin{pmatrix} K & 0 & 0 \\ 0 & L & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} 1 & 0 \\ 0 & 3 \\ 1 & 1 \end{pmatrix} \quad et \quad f_0 = \begin{pmatrix} 2\,K \\ 5 \\ 0 \end{pmatrix}$$

7.3 Boucles non imbriquées

Si l'on veut traiter le cas de plusieurs références au même tableau se trouvant dans des corps de boucles différents, on peut utiliser dans les deux types de transfert les mêmes algorithmes.

Le seul cas qui devra être traité différemment est le cas de plusieurs références en translation. En effet, ce cas est le seul qui suppose que les différents espaces de données référencés sont identiques à une translation près. Or dans le cas de plusieurs corps de boucles, donc domaines d'itération, cette propriété sera rarement vérifiée. Nous utiliserons donc pour traiter le cas de plusieurs (2 ou plus) références en translation l'algorithme utilisé pour générer le code de transfert de plusieurs (respectivement 2 ou plus) fonctions d'accès au même tableau de même Z-module.

7.4 Appels de fonctions

Nous supposons maintenant que le code de la tâche contient un appel à une fonction faisant référence au tableau T.

Les méthodes d'analyse interprocédurale calculent les effets de la procédure sur les différents paramètres du sous-programme pour détecter les zones d'aliasing. Ces informations sont souvent représentées par des domaines convexes: des *polyèdres* [Trio84], des *sections régulières* (lignes, colonnes,..) [CaKe87], ou des sous ensembles de polyèdres [BaKe89]. Nous pourrions donc utiliser ces informations (en supposant qu'elles soient disponibles) pour calculer les domaines du tableau utilisés par le sous programme.

Cependant, pour générer le code de transfert des éléments modifiés par la tâche après l'exécution d'un appel de procédure possédant en paramètre un tableau local,

il faudrait connaître très exactement les éléments modifiés par la procédure. Les méthodes d'analyse interprocédurale comparent, en général, uniquement les éléments du tableau utilisés dans le sous programme à ceux utilisés dans la tâche elle-même. Ces informations ne sont donc pas assez précises pour notre problème.

Nous traitons donc ici le cas de tâches comportant des appels de procédure de manière pessimiste i.e. tous les appels de procédure se feront sur les tableaux directement en mémoire globale.

Nous générerons un code de transfert des données de la mémoire locale vers la mémoire globale avant chaque appel de procédure. Puis un code de transfert de la mémoire globale vers la mémoire locale après chaque appel de procédure, car on ne connaît pas l'ensemble des données modifiées par le sous programme en mémoire globale.

7.5 Code ne vérifiant pas les hypothèses

Si le domaine d'itération n'est pas un polyèdre convexe, ou que les fonctions d'accès ne sont ni linéaires ni pseudo-linéaires (7.2,7.1) le code de transfert généré sera identique au code de calcul.

Pour minimiser les codes de transfert, une autre solution consiste à masquer les transferts par un tableau.

7.6 Conclusion

Nous avons présenté dans ce chapitre les méthodes permettant d'adapter nos algorithmes, développés dans les trois chapitres précédents, à un code de tâche qui ne vérifie pas les hypothèses décrites au chapitre 3.

Avec ces derniers résultats nous pensons pouvoir traiter la majorité des codes de tâches parallèles usuelles.

Chapitre 8

Comparaison avec d'autres méthodes

De nombreuses méthodes ont été proposées pour transférer et maintenir la cohérence des données dans les multiprocesseurs possédant des mémoires privées (caches ou locales).

Nous comparons dans ce chapitre quelques unes de ces méthodes avec la nôtre.

Nous commençons par les méthodes de gestion des mémoires caches. Deux types de méthodes sont couramment employées: celles qui vérifient en permanence la cohérence des blocs mémoire et celles qui associent une phase d'analyse des dépendances à l'ajout de directives dans les programmes, pour influencer les transferts.

Les solutions proposées pour les mémoires locales sont moins nombreuses, elles constituent une phase supplémentaire des compilateurs. Elles améliorent les performances des programmes s'exécutant sur ce type de multiprocesseurs en admettant le transfert dans les mémoires locales de certains types de données, tels que les tableaux, partagées par plusieurs tâches s'exécutant en parallèle.

Enfin, nous terminerons ce chapitre avec la présentation de quelques méthodes chargées d'assurer le transfert des données entre les différentes mémoires des multiprocesseurs à mémoire répartie. Même si ce type d'architecture est différent de celui à mémoire partagée, les méthodes présentées comportent certaines similitudes avec la nôtre.

8.1 Mémoire globale et mémoires caches

La gestion des mémoires caches est en général assurée par le système, le programmeur n'a à se soucier ni de l'emplacement des données utilisées par une tâche, ni de leur transfert inter-mémoires. Le code des tâches parallèles s'exécutant sur ces processeurs peut donc comporter des boucles faisant référence à un ensemble de données beaucoup

plus important que le nombre de données adressables en mémoire cache et/ou des synchronisations internes.

De nombreuses méthodes de gestion des mémoires caches ont été proposées. Il y a principalement deux catégories de solution au problème d'incohérence mémoire: les méthodes qui vérifient en permanence la validité des blocs mémoire et celles qui associent une phase d'analyse des dépendances à l'ajout de directives dans le code de la tâche parallèle.

On distingue pour chacune de ces catégories deux types de solution. Pour la première, il y a les *Snoopy cache* ou l'utilisation des tables de validité:

- Snoopy cache [OwAg89]: les adresses des blocs modifiés par un processeur sont communiquées à l'ensemble des processeurs qui, s'ils possèdent dans leur cache privé une partie du même bloc, marquent ce bloc comme modifié. Tout accès à l'un de ces blocs entraîne la recherche de la dernière copie du bloc (qui peut se trouver dans la mémoire centrale ou dans l'une des autres mémoires caches) et son transfert.
- table de validité [CeFe78], [ChVe88]
  Chaque bloc mémoire (ligne, mot) possède un bit de validité permettant de savoir si la copie du bloc mémoire se trouvant dans un cache précis a été modifiée ou pas. Il permet de savoir si cette copie est valide pour l'opération (lecture, écriture) en cours.

Dans la seconde catégorie , certaines méthodes utilisent des directives autorisant (ou pas) le transfert des données en mémoire cache, d'autres des directives d'invalidation des zones du cache:

- les variables *locales* (cachables) et *non-locales* (non-cachables) :
  les données *non-locales* (partagées par plusieurs groupes en lecture ou écriture) ne sont jamais transférées dans les mémoires cache des groupes. Seules les données *partiellement partagées* par un ensemble de processeurs appartenant au même groupe peuvent être copiées dans le cache correspondant, ainsi que toutes les données *locales*. Les données peuvent être déclarées comme partagées soit par le programmeur, soit par le compilateur.
- directives d'invalidation de zone du cache [Veid86], [ChVe89], [CKMA88], [Karl89]:
  Des directives sont ajoutées au programme, soit pour forcer la recopie de l'ensemble des données (ou d'une donnée [Karl89]) se trouvant dans le cache dans la mémoire globale, soit pour forcer la mise à jour d'une variable à la fois dans la mémoire cache et dans la mémoire globale, ou seulement dans la mémoire globale.

Dans le cas des mémoires locales, c'est le programmeur qui doit gérer la cohérence des données et tous les transferts inter-mémoires. Une phase de partitionnement des tâches parallèles est donc souvent effectuée pour obtenir des tâches dont la totalité des données peut être contenue en mémoire locale. Elle limite les échanges inter-mémoires et augmente la localité des données. Notre étude ne traite que ce type de tâche parallèle "partitionnée".

Nos algorithmes ne s'adaptent donc pas directement aux transferts des données entre une mémoire globale et des mémoires caches.

De plus, dans le cas des mémoires locales, la phase de parallélisation suffit à partitionner une tâche de telle sorte qu'il n'existe plus de conflits de dépendance entre deux exécutions en parallèle de la tâche. Dans le cas des mémoires caches, ce n'est pas toujours le cas. En effet, pour des raisons de performance [LiHu89] (les tâches accèdent souvent aux données de manière contiguë) la taille de la ligne transférée pour une donnée manquante dans le cache est souvent supérieure à la taille de la donnée. Or les calculs de dépendance servant à paralléliser les tâches ne tiennent pas compte de ces "débordements".

Les problèmes de cohérence des données entre une mémoire globale et des mémoires caches sont donc plus difficiles à résoudre que pour les mémoires locales, même si l'on se restreint aux hypothèses fixées pour ces dernières. Toutefois l'objectif reste commun: conserver la cohérence des données en mémoire et minimiser le trafic inter-mémoires.

Comparons les algorithmes que nous avons proposés et les algorithmes développés pour conserver la cohérence des données dans les mémoires caches, appliqués aux mémoires locales, avec les hypothèses exposées précédemment.

Nos algorithmes appropriés aux mémoires locales diminuent les transferts inter-mémoires et favorisent toujours les transferts contigus.

En effet, si la taille des lignes cachables est supérieure à 1 (taille d'un mot), des éléments non référencés sont aussi transférés. Le nombre de transferts généré par la stratégie "cache" est donc supérieur à la nôtre. De plus, elle crée des possibilités de conflits de dépendance inter-mémoires locales et augmente ainsi les transferts.

De plus, si le mode de mise à jour des données dans le cache est *store-through*, toutes les données modifiées sont non seulement copiées dans les mémoires caches mais aussi dans la mémoire globale. Si certaines des données sont référencées plusieurs fois par la tâche dans le corps de boucles, elles seront recopiées autant de fois en mémoire globale. Le trafic est donc plus important que si l'on recopie l'ensemble des données modifiées en une seule étape à la fin de la tâche.

Nos algorithmes favorisent toujours les transferts contigus, car le code de transfert qu'ils génèrent permet de transférer contiguëment l'ensemble des données référencées (si elles sont contiguës) même si cette contiguïté n'est pas explicite dans le code de calcul.

8.2 Mémoire globale et mémoires locales

Nous présentons ici deux autres méthodes de transfert des données entre une mémoire globale et des mémoires locales. La première utilise l'algorithme paramétrique en nombres entiers de P.Feautrier. La seconde, présentée dans [GJGa88], a été proposée par K. Gallivan, W. Jalby et D. Gannon.

8.2.1 Algorithme paramétrique en nombres entiers de P. Feautrier

L'algorithme paramétrique de résolution de systèmes d'inéquations linéaires en nombres entiers proposé par P.Feautrier [Feau88b] peut être utilisé pour calculer les bornes inférieure et supérieure des indices du tableau référencés dans un corps de boucles.

Il peut donc être utilisé pour générer le code de transfert de la mémoire globale vers les mémoires locales et transférer tous les éléments appartenant au polyèdre défini par ces bornes.

Il peut aussi être utilisé pour le code de recopie de la mémoire locale vers la mémoire globale, mais uniquement dans les cas où tous les éléments modifiés par la tâche sont référencés avec contiguïté.

8.2.2 Méthode de génération automatique des transferts proposée par K. Gallivan, W. Jalby et D. Gannon.

K. Gallivan, W. Jalby et D. Gannon proposent dans [GJGa88] une méthode de génération automatique des transferts entre une mémoire globale et des mémoires locales. Cet article a mis en évidence les différents problèmes rencontrés pour caractériser l'ensemble des éléments référencés par un tableau dans un corps de boucles. Il propose plusieurs solutions de transferts des données référencées par une référence au tableau T. Outre la solution consistant à utiliser le code de calcul comme code de transfert, il propose deux autres solutions.

La première permet de caractériser l'ensemble des points recherchés par un ensemble de points englobant approchant: l'enveloppe convexe des points entiers appartenant au Z-module du domaine image. Cette solution est équivalente à celle proposée pour copier les données utilisées par la tâche de la mémoire globale vers la mémoire locale.

La seconde solution permet de caractériser exactement un ensemble non convexe de points entiers. Elle associe cet ensemble à l'union des images d'un certain nombre de faces du polyèdre définissant le domaine d'itération. Les codes de transfert déduits de ces images sont optimaux et les éléments référencés ne sont copiés qu'une seule fois en mémoire.

Les caractéristiques de notre étude par rapport à ces deux méthodes sont les suivantes:

> Nous utilisons les polyèdres pour caractériser l'ensemble des éléments référencés par une fonction. La représentation des polyèdres par des systèmes linéaires en nombres entiers et l'existence d'algorithmes simples de manipulation de tels systèmes nous a permis:

- de réduire les hypothèses classiques sur les domaines d'itération
  - les contraintes caractérisant l'ensemble des données référencées par le corps de boucles sont des expressions affines des indices de boucle (pas uniquement des constantes),
  - chaque indice peut être contraint par plusieurs contraintes (expressions MIN et MAX dans les expressions des bornes des boucles),
  - les domaines d'itération peuvent être disjoints (boucles non imbriquées [§7.3]).
- de réduire les hypothèses de linéarité des fonctions d'accès aux éléments du tableau, en autorisant certaines fonctions pseudo-linéaires:
  - la fonction MODULO [§7.1.2] ,
  - certaines fonctions dont les composantes sont des expressions linéaires de constantes symboliques et des indices de boucle [§7.2] .
- de réduire les hypothèses sur le code de calcul, en autorisant dans le corps de boucles des instructions autres que les instructions d'affectations:
  - les fonctions de test IF[§7.1.1],
  - les appels de procédures CALL [§7.4].

mais surtout,

- d'étendre simplement les algorithmes proposés pour une seule référence au tableau dans le corps de boucles au cas où plusieurs fonctions référencent le même tableau dans le corps de boucles.

La fonction de coût des transferts en mémoires locales et la fonction d'évaluation du volume d'un polyèdre, résultant d'une transformation linéaire d'un polyèdre convexe, que nous proposons d'utiliser sont celles proposées dans l'article [GJGa88].

Les différentes méthodes proposées pour transférer les éléments référencés par 1 référence au tableau de la mémoire globale vers la mémoire locale sont équivalentes.

Pour les multiprocesseurs dont les processeurs ont accès non seulement aux mémoires locales mais aussi à la mémoire globale, Gallivan, Jalby, Gannon, Eisenbeis, Windheiser et Bodin proposent dans [GJGa88b] et [EJWB90] de ne transférer dans les mémoires locales que les éléments qui sont référencés plusieurs fois. Ces éléments sont caractérisés par des "reference window" et déterminés à l'aide d'une phase d'analyse des dépendances. Cette méthode n'est proposée que pour le cas de plusieurs références en translation. Elle permet de réduire le volume de données à stocker et à transférer en mémoire locale; les processeurs accédent aux données référencées une seule fois directement en mémoire globale.

Cette possibilité de ne transférer que l'ensemble des points référencés plusieurs fois au cours de l'exécution est importante. Comme ces ensembles de points plusieurs fois référencés se traduisent simplement sous la forme de polyèdres convexes, il serait intéressant d'étudier l'influence de cette hypothèse supplémentaire sur nos algorithmes.

8.3 Mémoires distribuées

La gestion des mémoires distribuées est plus complexe que celle des multiprocesseurs qui possèdent une mémoire globale. En effet, un facteur important doit être pris en compte: le partitionnement et la distribution des données dans les différentes mémoires privées des processeurs. Pour assurer une bonne gestion de ces données, deux techniques sont envisageables. La première consiste à trouver un placement possible des données utiles à l'exécution des tâches sur les mémoires privées, et d'en déduire un partitionnement approprié des instructions de la tâche sur les différents processeurs. La deuxième consiste à choisir en premier le partitionnement des tâches en tâches parallèles et de distribuer au mieux les données sur les différents processeurs afin de conserver le parallélisme obtenu par la première opération.

La deuxième solution semble plus difficile à automatiser que la première. En effet, il faut gérer plusieurs ensembles de données référencées par les tâches: les données utilisées par l'ensemble des tâches parallèles. Ces ensembles sont rarement disjoints, et le choix d'un bon partitionnement est complexe. Une distribution compliquée très liée aux tâches parallèles serait trop délicate à gérer et en particulier pour les transferts inter-mémoires. Une distribution trop simple (par colonnes ou par lignes pour les tableaux par exemple) donnerait des temps d'exécution supérieurs pour les communications que pour les calculs eux-mêmes, car elle ne conserverait plus tout le parallélisme obtenu par la première opération.

8.3.1 Partitionnement des données

La majorité des méthodes [CaKe88], [GJGa88], [RoPi89], [Tsen89], [APTh90], [Gern90], [KoMe90] optent pour la première solution. Elles supposent que le partitionnement des données utiles à l'exécution des tâches sur les processeurs est effectué par le programmeur et qu'il est "simple". Ce partitionnement doit pouvoir être exploité facilement par les étapes de génération des codes de transfert (l'endroit où se trouve une donnée non locale doit être facilement calculable) et de calcul du masque des itérations exécutables sur un processeur (on n'a pas de distribution des tâches parallèles sur les processeurs, elle est implicite). Le partitionnement des données correspond donc, dans la majorité des cas, à une union de polyèdres convexes.

8.3.2 Masque de calcul

Une fois le partitionnement des données effectué sur les processeurs, il faut déterminer les instructions qu'ils vont exécuter, et générer les codes de transfert des données utiles aux tâches. Toutes les études se sont intéressées principalement à la distribution, sur les processeurs, des itérations d'un corps de boucles faisant référence aux éléments d'un tableau.

De nombreuses méthodes [CaKe88], [RoPi89], [APTh90], [Gern90] ont choisi d'exécuter sur un processeur les itérations auxquelles correspondent des mises à jour de données locales au processeur. Cette stratégie permet de garantir une certaine cohérence des données et de limiter les transferts inter-mémoires. En effet, les données locales à un processeur sont toujours correctes, et il est simple de savoir où elles sont localisées. De plus, on n'a besoin

de transférer que des données utilisées par les tâches, qui n'ont besoin d'être copiées qu'une seule fois (pas besoin de les recopier).

Pour caractériser les itérations exécutables sur un processeur, un masque d'exécution est utilisé. Il est défini par de nouvelles instructions du langage. Il teste à l'exécution et à chaque itération des boucles si la donnée devant être modifiée est locale, si elle l'est, le processeur exécute l'instruction. Ce test est donc relativement coûteux. [Gern90] propose une optimisation du code de calcul.

8.3.3 Optimisation du masque de calcul

Dans [Gern90], chaque instruction possède un masque. L'optimisation consiste à reporter ce masque dans les expressions des bornes de boucle du code de calcul. Des fonctions MIN et MAX sont introduites dans ces bornes. Elles servent à caractériser les itérations auxquelles correspondent des éléments de tableaux susceptibles d'appartenir à la mémoire locale et d'être modifiés. Le masque sera encore fréquemment conservé, car pour l'éliminer, il faut que les nouvelles bornes caractérisent très exactement ces données **modifiées et locales.**

Nos algorithmes peuvent être utilisés pour calculer cet ensemble de données modifiées et locales et générer le nouveau code de calcul correspondant qui ne contiendra pas, dans de nombreux cas, de masque d'exécution.

Le partitionnement des données sur les différents processeurs se traduit par des contraintes sur la fonction d'accès aux éléments du tableau et non sur les indices de boucle. En effet, pour une référence en écriture au tableau `T(2i+j)`, elles se traduiront par des contraintes du type: $L \leq 2i + j \leq R$, où L et R sont les bornes inférieure et supérieure des indices du tableau appartenant à la mémoire locale du processeur. L'ensemble des itérations à exécuter peut donc être non convexe. Nos algorithmes utilisent, dans ce cas, des divisions entières dans les expressions des bornes de boucle du code pour traduire la non-convexité et n'ont pas besoin de masque d'exécution. Les fonctions MIN et MAX ne permettent pas de traduire des domaines d'itération non-convexes. La base caractérisant les éléments référencés doit être identique à celle du code de calcul initial afin de conserver les dépendances internes à la tâche. Les contraintes de partitionnement des données sur un processeur sont ajoutées au polyèdre caractérisant les éléments modifiés par le corps de boucles (projection du domaine d'itération sur la base de calcul). A partir de ce nouveau système, on génère automatiquement le nouveau code.

Lorsque le code de calcul possède plusieurs instructions ayant des masques différents, il faut calculer les blocs d'itérations devant être exécutés pour chacune des instructions. Il faut, ensuite, distribuer ces blocs afin d'éliminer totalement les masques des instructions du code de calcul.

8.3.4 Transferts des données

L'ensemble des données utiles à la tâche, c'est-à-dire celles qui sont utilisées par la tâche et n'appartiennent pas à la mémoire locale du processeur où elle est exécutée, est complexe à calculer.

Certaines méthodes proposent de le calculer à *l'exécution* [KoMe90], d'autres à la *compilation* [GJGa88], [RoPi89], [Gern90]. Ces dernières introduisent des primitives de réception et d'envoi de donnée "individuelle", utile à la tâche et détenue par un autre processeur ou utile à une autre tâche. [RoPi89] et [Gern90] proposent de les vectoriser (à la compilation), [Gern90] propose aussi de les fusionner (à l'exécution).

Dans les cas où on peut effectuer tous les transferts au début de la tâche (tableaux en lecture et en écriture différents, vecteur de direction des dépendances négatif, ...), nous proposons d'utiliser, pour effectuer ces transferts, nos algorithmes de génération de code de la mémoire globale vers la mémoire locale auxquels on ajoute une phase d'*exclusion*. Cette dernière consiste à éliminer des éléments utilisés par la tâche, ceux qui sont déjà disponibles dans la mémoire locale du processeur, en utilisant l'algorithme d'évaluation de la différence de deux ensembles de points entiers.

Cependant, comme les tâches ne sont pas supposées parallèles, des dépendances peuvent exister entre les instructions du code de calcul et on ne peut pas toujours transférer les données avant le début de la tâche.

Dans ce cas, il faut trouver une politique de placement des transferts pour savoir quand on peut copier, au plus tôt, une partie des éléments utilisés par la tâche. Nous n'avons pas envisagé dans nos algorithmes de telles politiques. Nous proposons donc d'utiliser soit la méthode proposée par [Gern90], soit une méthode de prefetching [Gorn89] pour déterminer ces placements.

Nos algorithmes permettent alors de fusionner, à la compilation, les transferts d'une partie des données utiles aux tâches. Ils ne garantissent plus, par contre, l'unicité des transferts: certaines données pourront être transférées plusieurs fois lors de transferts différents.

Dans tous les cas où il n'est pas possible de faire du prefectching, il ne sera pas possible non plus de regrouper des transferts de données. Et il faudra alors utiliser les primitives d'envoi de donnée "individuelle".

8.4 Conclusion

Conserver la cohérence des données dans les différentes mémoires locales (ou caches) et globale est l'objectif commun de l'ensemble des méthodes associées à la gestion de ces mémoires privées.

Cependant la gestion des mémoires caches étant assurée par le système et celle des mémoires locales par le programmeur, deux stratégies différentes sont adoptées.

Les méthodes de gestion des mémoires caches cherchent surtout à améliorer les performances des algorithmes de remplacement des blocs mémoire manquants. Les méthodes de transfert en mémoires locales supposent que les tâches sont partitionnées, de telle sorte que l'ensemble des données utilisées par tâche tiennent en mémoire locale, et élimine ainsi les remplacements de blocs mémoire.

Il est donc difficile de comparer ces deux stratégies, chacune étant spécifique aux mémoires privées des processeurs.

Pour les méthodes appropriées aux mémoires locales, la méthode proposée par K. Gallivan, W. Jalby et D. Gannon pour transférer les éléments référencés par une seule fonction d'accès génère, comme la nôtre, un code optimal i.e. on ne transfère qu'une seule fois les éléments référencés par le tableau entre les mémoires.

L'algorithme paramétrique de résolution de systèmes d'inéquations linéaires en nombres entiers proposé par P.Feautrier permet aussi, si les éléments référencés sont contigus, de générer un code de transfert optimal.

Seule notre méthode propose, à l'heure actuelle, une extension des résultats obtenus pour une référence au tableau dans un corps de boucles au cas où nous avons plusieurs références au même tableau. C'est la caractérisation des éléments référencés par un ensemble de polyèdres, représentés par des systèmes linéaires, qui nous a permis d'étendre simplement nos premiers résultats.

Les méthodes de gestion des données en mémoires réparties sont plus complexes que les précédentes, car elles doivent prendre en compte la répartition des données sur l'ensemble des processeurs.

Les ensembles de données que l'on cherche à caractériser sont particuliers à chaque type d'architecture. Pour les architectures à mémoire partagée, on cherche en général à transférer l'ensemble des éléments utilisés par la tâche. Dans le cas des architectures à mémoires réparties, on cherche à transférer uniquement l'ensemble des données utiles à la tâche qui n'appartiennent pas à la mémoire privée du processeur où la tâche s'exécute.

Cependant, toutes les méthodes cherchent à optimiser les codes de transfert en fusionnant les transferts de données "individuelles".

Lorsque le partitionnement des données est donné, et que le partitionnement d'un tableau sur un processeur peut se modéliser par un polyèdre convexe, il est possible d'utiliser nos algorithmes pour: optimiser le masque d'exécution des instructions du code de calcul et calculer l'ensemble des données utiles à l'application (non locales à la mémoire locale du processeur où elle s'exécute).

Conclusion

Nous avons développé dans cette étude toutes les phases de transformation permettant de transformer une tâche parallèle en un ensemble de tâches équivalentes utilisant les mémoires locales. Nous transformons le code des tâches en un code de sous-programme contenant: des déclarations de variables locales, le code de transfert des données utilisées de la mémoire globale vers la mémoire locale du processeur où est exécutée la tâche, le code de calcul, et le code de transfert des résultats de la mémoire locale vers la mémoire globale.

Les phases de génération automatique des codes de transfert des données entre la mémoire globale et les mémoires locales sont les plus complexes. Elles doivent générer les corps de boucles caractérisant les éléments référencés par les tableaux dans le but de les transférer. Ces éléments forment des ensembles de points entiers convexes ou non convexes.

La génération d'un corps de boucles parcourant des éléments contenus dans un polyèdre étant beaucoup plus simple que celle d'un ensemble quelconque de points entiers, nous avons cherché à caractériser les ensembles de points recherchés par des polyèdres. Nous proposons donc un ensemble d'algorithmes qui permettent de définir par des polyèdres l'image par une (ou des) transformation affine d'un polyèdre (l'image par la fonction d'accès aux éléments du tableau de l'espace d'itération).

L'ensemble des éléments référencés par une seule fonction d'accès au tableau, contenue dans une tâche parallèle, est dans la majorité des cas (au moins 82 %) un ensemble convexe. Il se caractérise donc simplement par un polyèdre.

Toutefois, les algorithmes numériques classiques contiennent souvent plusieurs références au même tableau en translation dans le même corps de boucles. Lorsque leur vecteur de dépendance uniforme est de norme supérieure à 1, l'ensemble des éléments référencés par ces fonctions est non convexe. Pour éviter les transferts multiples d'une même donnée, qu'engendreraient les transferts séparés des éléments référencés par chacune des fonctions, nous avons montré l'importance de fusionner ces ensembles. Nous avons donc proposé plusieurs solutions permettant de caractériser les ensembles de points entiers non convexes. Certaines calculent une approximation convexe de cet ensemble, d'autres en donnent une formulation exacte en utilisant plusieurs polyèdres ou un seul pseudo-polyèdre (polyèdre dont certaines contraintes contiennent des divisions entières).

L'exactitude des algorithmes proposés est fondée sur des conditions qui permettent de savoir si l'élimination d'une variable dans un système linéaire en nombres entiers est *correcte*, c'est-a-dire ne modifie pas l'ensemble des points entiers *solution* du système. Ces conditions sont à la base de la majorité de nos algorithmes. Associées à la méthode de Fourier-Motzkin, elles permettent de confirmer l'existence d'une solution entière si le système admet une solution rationnelle (autorisation d'éliminer toutes les variables du système).

Les opérations d'intersection et d'union (intersection + complémentaire) de deux do-

maines image ont pu être définies. Des méthodes permettant de regrouper, lors des transferts, des ensembles non disjoints d'éléments référencés par plusieurs fonctions d'accès à un même tableau dans un corps de boucles ont donc pu aussi être dévéloppées. Ces méthodes s'avèrent très utiles puisqu'environ 60 % des références à un tableau sont *dépendantes* d'une autre référence, les ensembles de points référencés sont alors non disjoints.

Les codes de transfert générés par nos algorithmes sont pratiquement toujours optimaux, c'est à dire qu'on ne transfère qu'une seule fois les éléments référencés par le tableau. Le nombre de ces transferts est minimal lorsque l'on cherche à copier les éléments référencés par une seule fonction d'accès au tableau, ou lorsque l'ensemble des données à transférer est convexe, ou encore lorsque ces éléments représentent la fusion de plusieurs fonctions d'accès en translation au même tableau. Dans les cas les plus défavorables, nous générons un code qui effectue plusieurs fois la copie des données appartenant à l'intersection des éléments référencés par les différentes fonctions. Mais ce cas ne se présente que lorsque les fonctions ont des Z-modules différents et donc ont une intersection restreinte.

Nous avons supposé que l'on pouvait effectuer l'ensemble des transferts des données utilisées par la tâche au début de l'exécution de cette dernière. Cependant, il est aussi possible d'utiliser nos algorithmes pour calculer l'ensemble de ces données pour des itérations particulières de certaines boucles. Ils peuvent donc être employés pour générer les codes de transferts des données utilisées au cours de l'exécution de la tâche.

Cette solution est utile, entre autres, s'il faut limiter le trafic des données de la mémoire globale vers les mémoires locales au début de l'exécution de la tâche. Cependant, il n'est plus possible dans ce cas de garantir l'unicité des transferts, certaines données pourront être copiées plusieurs fois lors de transferts différents.

L'ensemble des méthodes présentées nous a permis d'améliorer l'utilisation des mémoires locales. Elles permettent de préciser les données que l'on veut transférer en mémoire locale, et notamment les tableaux, ainsi que les données à recopier en mémoire globale.

Les méthodes de parallélisation des tâches sur des multiprocesseurs à mémoires totalement distribuées ont aussi besoin de caractériser les données utilisées ou devant être calculées par les tâches. Lorsque le partitionnement des données sur les mémoires réparties est connu et que le partitionnement peut se modéliser par des polyèdres, il est possible d'utiliser nos algorithmes pour optimiser le code de calcul approprié à chaque processeur et calculer l'ensemble des données utiles à la tâche (non disponibles dans la mémoire privée du processeur où elle s'exécute).

Des expériences permettant d'évaluer les améliorations réelles apportées à l'utilisation des mémoires locales manquent toutefois à cette étude. C'est l'objectif de nos futurs projets.

Annexe

Règles de transformation des expressions entières.

Certaines règles de transformation doivent être respectées lorsque l'on manipule des inégalités en nombres entiers. Ces règles introduisent des divisions entières.

La division entière pouvant avoir un comportement différent au voisinage de zéro selon la définition choisie, voici celle que nous avons adoptée:

On note $\frac{a}{b}$ la division entière de a par b.

$\frac{a}{b} = q \implies a = b \times q + r$ où $0 \leq r \leq b - 1$.

La divison entière de -1 par 2 vaudra par conséquent -1.

Règles de transformation des expressions entières

Soit E une expression quelconque et α un nombre entier positif. On a les quatre relations suivantes:

règle 1: $\alpha\ V \leq E \iff V \leq \frac{E}{\alpha}$

règle 2: $-\alpha\ V \leq E \iff V \geq \frac{(-E + \alpha - 1)}{\alpha}$

règle 3: $\frac{V}{\alpha} \leq E \iff V \leq \alpha\ E + \alpha - 1$

règle 4: $\frac{-V}{\alpha} \leq E \iff V \geq -\alpha\ E$

Démonstration:

Notons tout d'abord que si V est un entier, il peut s'écrire sous la forme $V = \alpha q + r$ où r $(0 \leq r \leq \alpha - 1)$ est le reste de division entière de V par l'entier α.

Nous avons l'égalité suivante $V = \alpha\frac{V}{\alpha} + r$.

- règle 1:

  1. montrons l'implication $\implies$:

$$\alpha\ V \leq E \implies \frac{\alpha V}{\alpha} \leq \frac{E}{\alpha} \implies V \leq \frac{E}{\alpha}$$

2. montrons l'implication $\Longleftarrow$:

$$V \leq \frac{E}{\alpha} \implies \alpha V \leq \alpha\frac{E}{\alpha} \implies \alpha V \leq E - r \implies \alpha V \leq E$$

- règle 2:

1. montrons l'implication $\Longrightarrow$:

$$-\alpha V \leq E \implies -\alpha V - \alpha + 1 \leq E - \alpha + 1 \implies \alpha V + \alpha - 1 \geq -E + \alpha - 1$$
$$\implies \frac{\alpha V + \alpha - 1}{\alpha} \geq \frac{-E+\alpha-1}{\alpha} \implies V \geq \frac{-E+\alpha-1}{\alpha}$$

2. montrons l'implication $\Longleftarrow$:

$$V \geq \frac{-E+\alpha-1}{\alpha} \implies \alpha V \geq \alpha\left(\frac{-E+\alpha-1}{\alpha}\right) \implies \alpha V \geq (-E + \alpha - 1) - r'$$
$$ou\ \ r' = (-E + \alpha - 1) - \alpha\left(\frac{-E+\alpha-1}{\alpha}\right)$$
$$comme\ \ \alpha - 1 + r' \geq 0 \implies \alpha V \geq -E \implies -\alpha V \leq E$$

- règle 3:

1. montrons l'implication $\Longrightarrow$:

$$\frac{V}{\alpha} \leq E \implies \alpha\frac{V}{\alpha} \leq \alpha E \implies V - r \leq \alpha E$$
$$\implies V \leq \alpha E + r \implies V \leq \alpha E + \alpha - 1$$

2. montrons l'implication $\Longleftarrow$:

$$V \leq \alpha E + \alpha - 1 \implies \frac{V}{\alpha} \leq \frac{\alpha E + \alpha - 1}{\alpha} \implies \frac{V}{\alpha} \leq E$$

- règle 4:

1. montrons l'implication $\Longrightarrow$:

$$\frac{-V}{\alpha} \leq E \implies \frac{V}{\alpha} \geq -E \implies \alpha\frac{V}{\alpha} \geq -\alpha E$$
$$\implies \alpha\frac{V}{\alpha} + r \geq -\alpha E + r \implies V \geq -\alpha E + r \implies V \geq -\alpha E$$

2. montrons l'implication $\Longleftarrow$:

$$V \geq -\alpha\ E \implies \frac{-V}{\alpha} \leq \frac{\alpha\ E}{\alpha} \implies \frac{-V}{\alpha} \leq E$$

Développement

L'ensemble des programmes servant à la génération du code de transfert des données a été écrit en langage C. Ils appartiennent à la bibliothèque C3 "polylib" regroupant des algorithmes de manipulation des matrices et des polyèdres développés à l'IRISA par l'équipe de P.Quinton, à l'Université Pierre et Marie Curie par F.Feautrier, et à l'Ecole des Mines de Paris par l'équipe du projet PIPS, à laquelle on a ajouté des modules servant à la génération du code proprement dit.

Les modules principaux sont:

- mise sous forme réduite de Hermite d'une matrice,
- faisabilité d'un système linéaire,
- élimination des contraintes redondantes d'un système linéaire,
- projection selon une variable d'un système linéaire,
- projection entière selon une variable d'un système linéaire,
- génération des bornes de boucle d'un polyèdre,
- intersection de deux polyèdres,
- différence de deux Z-polyèdres,
- calcul du plus petit polyèdre convexe englobant un Z-polyèdre non convexe,
- calcul du plus grand polyèdre convexe contenu dans un Z-polyèdre non convexe,
- décomposition d'un Z-polyèdre non convexe en polyèdres convexes,
- algorithme de génération de code proprement dit.

Mise sous forme réduite de Hermite d'une matrice

Nous utilisons la forme réduite de Hermite associée à une matrice [§4.1.3], [Herm51] car elle est plus rapide à calculer que la forme normale de Hermite [KaBa79], [Ilio89].

Faisabilité d'un système linéaire.

L'algorithme de faisabilité d'un système linéaire est utilisé principalement pour éliminer les contraintes redondantes dans le système. Cette opération est donc effectuée de nombreuses fois. Pour éviter des temps de calcul trop importants nous avons choisi d'utiliser un test de faisabilité d'un système linéaire en réels et non entiers. Cette perte d'information peut conduire, dans les cas défavorables, à conserver une inégalité redondante dans le système. Le fait que les polyèdres que nous manipulons possèdent des sommets entiers minimise le nombre de ces cas.

Elimination des contraintes redondantes d'un système linéaire

De nombreux algorithmes de manipulation des polyèdres construisent un polyèdre à partir d'un autre en ajoutant des contraintes au système linéaire initial, pour le diviser ou conserver certaines propriétés (convexité ou non-convexité) du polyèdre. Ces opérations conduisent rapidement à des systèmes linéaires de taille trés importante. Nous utilisons donc régulièrement une phase d'élimination des contraintes redondantes dans le système.

L'algorithme que nous utilisons est présenté dans [Chen87].

Projection selon une variable d'un système linéaire.

Nous utilisons pour calculer la projection continue d'un système linéaire selon une variable, l'algorithme d'élimination d'une variable dans un système de Fourier-Motzkin [Four24].

Projection entière selon une variable d'un système linéaire.

Pour calculer la projection entière d'un système linéaire selon une variable, nous employons l'algorithme présenté au [§4.4.5].

Génération des bornes de boucle d'un polyèdre

Nous utilisons la méthode présentée par F.Irigoin dans [Irig88] pour générer les bornes de boucle d'un polyèdre sous la forme d'un système linéaire.

Intersection et différence de deux polyèdres

Nous employons pour calculer respectivement l'intersection et la différence de deux Z-polyèdres les algorithmes présentés au [§4.2.2] et au [§4.3].

Calcul du plus petit polyèdre convexe englobant un Z-polyèdre non convexe

L'algorithme proposé au [§4.5.1] permet de calculer le plus petit polyèdre convexe englobant un Z-polyèdre non convexe. Il utilise l'algorithme de projection continue d'un polyèdre selon une variable.

Calcul du plus grand polyèdre convexe contenu dans un Z-polyèdre non convexe

Pour calculer le plus grand polyèdre convexe contenu dans un Z-polyèdre non convexe, on utilise l'algorithme proposé au [§4.5.2]. Ce dernier se sert de l'algorithme de projection entière d'un polyèdre selon une variable.

Décomposition d'un Z-polyèdre non convexe en polyèdres convexes.

Nous utilisons pour calculer la décomposition d'un Z-polyèdre non convexe en polyèdre convexe l'algorithme proposé au [§4.5.3].

Génération de code proprement dit

La partie génération de code comprend, entre autres, tous les algorithmes proposés pour transférer les éléments référencés par une ou plusieurs fonctions d'accès à un tableau dans un corps de boucles entre la mémoire globale et les mémoires locales, ainsi que les algorithmes de génération des déclarations des tableaux locaux et de transformation des références aux tableaux en des références aux tableaux locaux.

Bibliographie

[ABCC88] Allen F., Burke M., Charles P., Cytron R., Ferrante J., "An overview of the PTRAN analysis system for multiprocessing," *Journal of Parallel and distributed computing*, Vol. 5, No 5, Oct. 1988.

[AlKe87] Allen R., Kennedy K., "Automatic translation of FORTRAN programs to vector form," *ACM Transactions on Programming Langages and Systems*, Vol. 9, No 4, Oct. 1987.

[AlGo89] Almasi G.S., Gottlieb A., "Highly parallel computing," *Benjamin Cummings*, 1989.

[Anco87] Ancourt M., " Utilisation de systèmes linéaires sur Z pour la parallélisation de programmes ", *Rapport Interne Ecole des Mines de Paris, no CAI-87-E87*, 1987

[APTh90] Andre F., Pazat J-L., Thomas H., " PANDORE: a system to manage data distribution ", *International Conference on Supercomputing*, pp 380-388, June 1990

[ASKL79] Abu-sufah W., Kuck D., Lawrie D., "Automatic program transformations for virtual memory computers," *National Computer Conference*, 1979.

[ASMa86] Abu-Sufah W., Malony A.D., "Experimental results for vector processing on the Alliant FX/8," *Tech. Report C.S.R.D. Univ. of Illinois at Urbana-Champaign*, Feb. 1986.

[Babb88] Babb R.G.II, "Programming parallel processors," *Addison Wesley*, 1988.

[BaFu86] Barany I., Furedi Z., " Computing the Volume is difficult ", *Proceedings of the 18th annual ACM symposium on theory of computing* , 1986, pp. 442-447

[BaKe89] Balasundaram V., Kennedy K., "A technique for summarizing data access and its use in parallelism enhancing transformations," *ACM SIGPLAN*, 1989.

[BePa87] Bell J.L., Paterson G.S., "Data organization in large numerical computations," *Journal of Supercomputing*, Vol. 1, 1987, pp. 105-136.

[Bern66] Bernstein A.J., " Analysis of programs for parallel processing," *IEEE Transactions on Electronic Computers*, Vol. 15, No 5, pp.757-763, October 1966.

[BrDu83] Briggs F.A., Dubois M., "Effectiveness of private caches in multiprocessor systems with parallel-pipelined memories," *IEEE Transactions on computers*, Vol. 32, No 1, Jan. 1983.

[BMAW85] Brantley W.C., McAuliffe K.P., Weiss J., "RP3 Processor-Memory Element," *Report IBM T.J. WATSON Research, Yorktown Heights, N.Y.10598*, July 1985.

[Call88] Calahan D.A., "Performance evaluation of static and dynamic memory systems on the CRAY 2," *International Conference on Supercomputing*, 1988.

[CaKe86] Callahan D., Cooper K.D., Kennedy K., Torczon L., "Interprocedural constant propagation," *SIGPLAN*, June 1986.

[CaKe87] Callahan D., Kennedy K., "Analysis of interprocedural side effects in a parallel programming environment," *First International Conference on Supercomputing*, Greece, 1987

[CaKe88] Callahan D., Kennedy K., " Compiling programs for distributed-memory multiprocessors ", *Journal of supercomputing*, pp 151-169, 1988

[CaSz89] Carlier T., Szafran N., " Structure de données pour la représentation et la manipulation des polyèdres " *Rapport de recherche Informatique et Mathématiques Appliquées de Grenoble*, No RR772-M, 1989

[CeFe78] Censier L., Feautrier P., "A new solution to coherence problems in multicache systems," *IEEE transactions on computers*, Vol. c-27, No 12, December 1978.

[Chen87] Cheng M. C., "General criteria for redundant and nonredundant linear inequalities," *Journal of optimization theory and applications* Vol. 53, No 1, April 1987.

[ChDi89] Chi C.H., Dietz H., " Unified Management of Registers and Cache Using Liveness and Cache Bypass", *SIGPLAN 89, Conference on Programming Langage Design and Implementation*, 1989

[ChVe87] Cheong H., Veidenbaum A., " The performance of software-managed multiprocessor caches on parallel numerical programs " *International Conference on Supercomputing*, June 87

[ChVe88] Cheong H., Veidenbaum A., " A cache coherence scheme with fast selective invalidation " *The 15th Annual international symposium on computer architecture* June 1988.

[ChVe89] Cheong H., Veidenbaum A., "Compiler-directed cache management for multiprocessors" *Tech. Report C.S.R.D. Univ. of Illinois at Urbana-Champaign*, No 937, 1989

[CKMA88] Cytron R., Karlovsky S., McAuliffe, " Automatic management of programmable caches " *International Conference on Parallel Processing*, 1988

[CRAY 2] "Documentation CRAY 2," *Hardware reference manuel*

[CRAY2] "CRAY-2 computer systems," *Cray2 Fortran (CFT2) reference manual*

[Dong87] Dongarra J.J., "Experimental parallel computing architectures," *Northholland*, 1987.

[DyFr88] Dyer M.E., Frieze A.M., "On the complexity of computing the volume of a polyhedron," *Siam Journal of computing*, Vol.17, No 5, Oct. 1988.

[DMVL] Dehbonei B., Memmi G., Verleye-Laurent C., "VELOUR: An experience towards language independent vectorization,"

[Enc85] "Multiprocessor makes parallelism work," *Electronics*, September 2, 1985, pp. 46.

[EmPY89] Emrath P.A., Padua D.A., Yew P.C., "CEDAR architecture and its software," *Proceedings of the twenty second Annuka Hawai. International Conf. in System Sciences*, 1989.

[EJWB90] Eisenbeis C., Jalby W., Windheiser D., Bodin F., " A Strategy for array management in local memory," *3rd Workshop on Languages and Compilers for Parallel Computing*, Irvine, 1990.

[FDT87] Feautrier P., Dumay A., Tawbi N., "PAF: un paralléliseur pour FORTRAN," *rapport technique du laboratoire MASI de l'Université de Paris VI*, No 185, Mai 1987

[Feau88] Feautrier P., "Semantical analysis and mathematical programming," *Int. Workshop on Parallel and Distributed Algorithms*, Bonas, 3-6 Octobre 1988.

[Feau88b] Feautrier P., "Parametric integer programming," *RAIRO Recherche opérationnelle* pp:243-268, Sept. 1988.

[FeTa90] Feautrier P., Tawbi N., "Résolution de systèmes d'inéquations linéaires," *Rapport MASI - Univ. Pierre et Marie Curie*, Janvier 1990.

[FeQu88] Fernandez F., Quinton P., "Extension of Chernikova's algorithm for solving general mixed linear programming problems," *Publication interne IRISA-INRIA* No 437

[Four24] Fourier J.B.J., "Analyse de travaux de l'Académie Royale des Sciences, pendant l'année 1824, partie mathématique," *Histoire de l'Académie Royale des Sciences de l'Institut de France*, 1827.

[Gent87] Genty H., "La norme FORTRAN 8X: Description générale," *Minis et Micros*, No 277, Avril 1987.

[Gern90] Gerndt H. M., " Automatic parallelization for distributed-memory multiprocessing systems" *Dissertation zur Erlangung der Doktorwurde der Mathematisch-Naturwissenschaftlichen Fakultat der Rheinischen Friedrich-Wilhelms-Universitat* Bonn 1989.

[GJGa88] Gallivan K., Jalby W., and Gannon D., "On the problem of optimizing data transfers for complex memory systems," *Proceeding of the ACM Int. Conf. on supercomputing*, St-Malo, 1988.

[GJGa88b] Gannon D., Jalby W., Gallivan K., " Strategies for cache and local memory management by global program transformation ", *Journal of Parallel and Distributed Computing*, Vol. 5, pp 587-616, 1988.

[Gond73] Gondran, "Algorithme des congruences décroissantes," *Programmation mathématique - théorie et algorithmes (tome 2). Dunod*, 1983, pp. 22.

[Gorn89] Gornish E.H., " Compile time analysis for data prefetching ", *Tech. Report C.S.R.D. Univ. of Illinois at Urbana-Champaign*, No 949, December 1989.

[Guzz87] Guzzi M.D., "CEDAR FORTRAN Programmer's handbook," *Tech. Report C.S.R.D. Univ. of Illinois at Urbana-Champaign*, June 1987.

[Halb79] Halbwachs N., "Détermination automatique de relations linéaires par les variables d'un programme," *Thèse de 3ème cycle, INP Grenoble*, 1979.

[Herm51] Hermite C., "Sur l'introduction des variables continues dans la théorie des nombres," *Journal fur dir reine und angewandte Mathematik Math.,*, Vol. 41, 1851.

[Hoar87] Hoare A.A.R., "Processus sequentiels communicants," *Masson*, 1987.

[Holl89] D'Hollander E., "Partitioning and labeling of index sets in DO loops with constant dependence" *International Conference on Parallel Processing*, 1989

[HoJe81] Hockney R. W., Jesshope C.R., "Parallel Computers2," *Adam Hilger*

[Irig87] Irigoin F., "Partitionnement des boucles imbriquées - Une technique d'optimisation pour les programmes scientifiques," *Thèse de l'Université Pierre et Marie Curie*, 1987.

[Irig88] Irigoin F., "Code generation for the hyperplane method and for loop interchange," *Rapport ENSMP-CAI-88-E/102/CAI/I*, 1988

[IrTr87] Irigoin F., Triolet R., " Computing Dependence Direction Vectors and Dependence Cones with Linear Systems ", *Rapport Interne Ecole des Mines de Paris no CAI-87-E94*, 1987

[IJTr90] F. Irigoin, P. Jouvelot, R. Triolet, " Overview of the PIPS project", International Workshop on Compilers for Parallel Computers, Paris, December 3-5, 1990.

[Karl89] Karlovsky S.R., "Automatic management of programmable caches: Algorithms and experience." *Thesis Univ. of Illinois at Urbana-Champain*, 1989

[Ilio89] Iliopoulos C. S., "Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix," *Siam Journal of computing*, Vol.18, No 4, August 1989.

[Inmo82] Inmos, "The Occam programming Manual," *Inmos 82*

[Jouv88] Jouvelot P., "Pragmatics in Parallel Functional Programming," *Journées FIRTECH Paris*, November 1988.

[JoDo89] Jouvelot P., Dornic V., "FX-87, or what comes after Scheme," *BIGRE 65*, Juillet 1989, pp. 55-66.

[KaBa79] Kannan R., Bachem A., "Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix," *Siam Journal of computing*, Vol.8, No 4, Nov. 1979.

[Kerr87] Kerridge J., "Occam programming: a pratical approach," *Blackwell Scientific Publications*

[KoMe90] Koelbel C., Mehrotra P, " Supporting shared data structures on distributed memory architectures " *Second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, Vol. 25, No 3, March 1990

[Lamb87] Lambert J.L., "Trois algorithmes de résolution d'une équation linéaire en nombres entiers positifs pour l'intelligence artificielle et la parallélisation," *Rapport de recherche L.R.I. No 394* , Décembre 1987.

[Lamp74] Lamport L., "The parallel execution of DO loops," *Communications of the ACM*, Vol 17, No 2, Feb. 1984.

[LaMa88] Lassez J.L., Maher M., "On Fourier's algorihtm for linear arithmetic constraints," *IBM research report, T.J. Watson Research Center*

[LiHu89] Li K., Hubak P., " Memory coherence in shared virtual memory systems " *ACM Transactions on computer systems*, 89

[LiSt88] Liu B., Strother N., "Programming in VS Fortran on the IBM 3090 for Maximum Vector Performance," *IEEE Computer journal*, June 1988.

[LiTh85] Lichnewsky A., Thomasset F., "Techniques de base sur l'exploitation automatique du parallélisme dans les programmes," *INRIA, Rapport de recherche No 460*

[MaBi71] MacLane S., Birkhoff G., " Algègre : Les grands théorèmes ", *Ed. Gauthiers-Villars*, Paris, 1971.

[Metc85] Metcalf M., "Fortran Optimization," *Academic Press*, 1985.

[Min83] Minoux M., "Programmation mathématique - Théorie et algorithmes," *Dunod*, 1983.

[MeRe89] Metcalf M., Reid J., "Fortran 8x Explained," *Oxford science publications*, 1989.

[MuNe80] Muramatsu H., Negishi H., "Page replacement algorithm for large-array manipulation," *Software-pratice and experience*, Vol. 10, 1980, pp. 575-587.

[NeWo88] Nemhauser G.L., Wolsey L.A., "Integer and combinatorial optimization," *Ed. Wiley*, 1988.

[OwAg89] Owcki S., Agarwal A., " Evaluating the performance of sotfware cache coherence " *Tech. Report Digital*, No 41, 1989

[PeCy89] Peir J.-K., Cytron R., "Minimum distance: A method for partitioning recurrences for multiprocessors," *IEEE transactions on computers,* Vol. 38, No 8, august 1989.

[PBGH85] Pfister G.H., Brantley W.C., George D.A., Harvey S.L., Kleifelder W.J., McAuliffe K.P., Melton E.A., Norton V.A., Weiss J., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Report, IBM T.J. WATSON Research, Yorktown Heights*

[PGHL89] Polychronopoulos C.D., Girkar M., Haghighat M.R., Lee C.L., "Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors," *International journal of high speed computing*, Vol. 1, No 1, May 1989.

[PeMu86] Perron R., Mundie C., "The architecture of the Alliant FX/8 computer," *IEEE Computer Society Press,* Spring, 1986.

[RoPi89] Rogers A., Pingali K., " Process decomposition through locality of reference", *Conference on Programming Langage Design and Implementation*, pp 69-80, June 1989

[Scho87] Schonauer W., " Special topics on Supercomputing - Scientific Computing on vector computers," *North Holand*

[SLYe88] Shen Z., Li Z., Yew P.C., "An empirical study on arrays subscripts and data dependences," *International Conference on Parallel processing*, August 1989, pp. 145-153.

[Schr87] Schrijver A., "Theory of linear and integer programming," *Wiley*

[Smit82] Smith A. J., " Cache memories ", *IEEE Computing Surveys*, Vol. 14, No 3, September 1982

[Tawb90] Tawbi N., " Allocation de processeurs et ordonnancement sur multiprocesseurs." *Thèse de l'Univ. Pierre et Marie Curie*, 1991

[ThSm89] Thompson J.G., Smith A.J., "Efficient (Stack) algorithms for analysis of write-back and sector memories," *ACM Transactions on Computer Systems*, Vol 7, No 1, Feb. 1989.

[Trio84] Triolet R., "Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédure," *Thèse de l'Université Pierre et Marie Curie*, Décembre 1984.

[Tsen89] Tseng P.S., " A parallelizing compiler for distributed-memory parallel computers ", *PhD Thesis - Canergie Mellon University*, 1989

[Tuck86] Tucker S.G., "The IBM 3090 system: An overview," *IBM systems journal*, Vol 25, No 1, 1986.

[Veid86] Veidenbaum A., " A compiler-assisted cache coherence solution for multiprocessors " *International conference on parallel processing*, august, 1986

[Wall88] Wallace D., "Dependence of multi-Dimensional Arrays References," *International Conference on Supercomputing*, 1988, pp. 418-427.

[Wolf82] Wolfe N.J., "Optimizing supercompilers for supercomputers," *Ph.D. thesis University of Illinois, Urbana*, Rep. no UIUCDCS-R-82-1105, 1982.

[Wolf88] Wolfe M., "Vector optimization vs Vectorization," *Journal of Parallel and distributed computing*, Vol. 5, No 5, Oct. 1988.

[Yew86] Yew P.C., "Architecture of the CEDAR parallel supercomputer," *Tech. Report C.S.R.D. Univ. of Illinois at Urbana-Champaign*, Aug. 1986.

[XAcm89] X/Acm, "Fortran 8X Draft," *Fortran Forum*, Vol. 8, No 8, pp. 1-348, 1989.

[ZBGH86] Zima H.P., Bast H-J., Gerndt M., Hoppen P.J., " Superb: The Suprenum parallelizer " *Suprenum Research Report*, No 861203, December 1986