N° d'ordre

THÈSE

présentée

pour obtenir

LE GRADE DE DOCTEUR

DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

par

François IRIGOIN

Spécialité: Mathématiques
Mention: Informatique

Sujet de la thèse

PARTITIONNEMENT DES BOUCLES IMBRIQUÉES

UNE TECHNIQUE D'OPTIMISATION POUR LES PROGRAMMES SCIENTIFIQUES

Soutenue le 18 juin 1987 devant la commission composée de:

Messieurs	G. PUJOLLE	.....	Président
	C. GIRAULT	.....	Rapporteurs
	A. LICHNEWSKY	.....	
	P. FEAUTRIER	.....	Examinateurs
	R. TRIOLET	.....	
	R. CYTRON	.....	Invité

ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE PARIS

A/183/CAI/I

Résumé

L'architecture compliquée des superordinateurs modernes doit être prise en compte lors de la conception des programmes pour assurer à la fois une saturation du parallélisme vectoriel et des processeurs élémentaires, une bonne localité des références de manière à ne pas se trouver limité par la bande passante de la mémoire principale, et un coût de synchronisation acceptable.

Nous proposons une nouvelle transformation de programme, appelée partitionnement en supernoeuds, qui s'applique aux boucles imbriquées, et qui permet d'atteindre les quatre objectifs cités précédemment. Cette transformation est basée sur la connaissance précise des dépendances entre instructions du corps de boucle. Elle consiste à découper l'espace des calculs par des familles d'hyperplans respectant les dépendances pour ne pas créer de cycles entre les partitions. La validité d'une famille peut être testée automatiquement.

Nous montrons comment cette transformation peut être appliquée de manière automatique avec des algorithmes sur les systèmes linéaires d'inégalités dès que les caractéristiques du partitionnement sont connues numériquement.

Le choix d'un partitionnement optimal, qui dépend à la fois des transferts de données, du volume de données que peut contenir la mémoire la plus rapide, des caractéristiques des unités fonctionnelles et des performances des primitives de synchronisation disponibles ne peut pas être fait par dérivation analytique car de nombreuses données de base, relatives à la génération de code ou à l'ensemble des calculs considérés, sont inconnues en général. Nous proposons donc une liste de paramètres et de facteurs qu'il faut prendre en compte pour déterminer manuellement un partitionnement intéressant. Nous avons ainsi réussi à obtenir une augmentation de vitesse de 50 % sur l'Alliant FX-8 en améliorant la localité des références.

Cette méthode est une généralisation de l'échange de boucles, de la méthode hyperplane et de la classe des partitionnements qu'on obtient par blocage et échange de boucles. La méthode de génération de code développée s'applique donc aussi à ces cas.

Mots-clés

Parallélisme, parallélisation, vectorisation, multiprocesseur, transformation de programme, partitionnement.

Abstract

The intricate architectures of today's supercomputers must be taken into account when programming to insure the best use of vector units and/or elementary processors, proper data locality to exploit the memory hierarchy, and small synchronization overhead.

We propose a new restructuring technique, called **supernode partitioning**, that applies to nested loops and provides code that satisfies the four previous goals. This transformation is based on a precise knowledge of dependencies.

We show how this technique can be applied automatically with algorithms on linear systems when the partitioning is numerically defined.

The choice of an optimal partitioning, which depends simultaneously on data transfers, on memory sizes, on vector unit features and on synchronization means, is intractable analytically because detailed information about machine code generation is usually unavailable. We explain in a qualitative way the effects of partitioning parameters like directions, lengths and scheduling. They must be hand chosen to devise a sound transformation. For example we get a 50% speed increase on the Alliant FX-8 thanks to the data locality increase.

Rectangular block partitionings, obtained by a combination of strip-mining and loop interchange, and loop reorderings such as the hyperplane method, are special cases of our approach. Thus, algorithms based on linear systems are also useful to generate code for these well-known methods.

Keywords

Parallelism, Supercomputer, Multiprocessor, Partitioning, Program Restructuring.

REMERCIEMENTS

Je tiens tout d'abord à remercier Monsieur le Professeur G. Pujolle, Directeur du MASI, de l'honneur qu'il m'a fait en acceptant de présider le jury de cette thèse.

Je veux aussi dire le plaisir que m'a fait Monsieur le Professeur C. Girault en acceptant d'être rapporteur et de participer à ce jury. C'est en effet lui qui m'a initié aux secrets du parallélisme par le biais des réseaux de Petri et qui m'a ouvert les portes d'un domaine que je n'ai pas quitté depuis.

Le rôle de Monsieur le Professeur P. Feautrier a été déterminant. Il n'a ménagé ni son temps, ni sa peine, pour guider le choix d'un sujet intéressant, pour lire les rédactions intermédiaires, pour suggérer des solutions et pour assurer les relations avec l'administration de l'Université. Tout cela, sans jamais perdre son amabilité coutumière, qui est si agréable.

Ce jury n'aurait pas eu l'occasion de se réunir si Rémi Triolet ne m'avait pas convaincu d'entamer une thèse juste avant de disparaître aux Etats-Unis et s'il n'avait pas joué le rôle de *sparing partner* toutes les fois qu'une nouvelle idée ou un blocage se présentaient. Cette collaboration s'est poursuivie durant l'année qu'il a passée aux Etats-Unis grâce au réseau EARN. A son retour, de nombreuses lectures d'obscures rédactions intermédiaires l'attendaient...

La présence de Monsieur le Professeur A. Lichnewsky, qui dirige l'équipe de recherche sur la parallélisation automatique à l'INRIA, est le symbole de la coopération qui existe dans ce domaine. Ma gratitude lui est acquise pour le support qu'il a fourni à ces travaux et l'intérêt qu'il leur a toujours montré.

R. Cytron, qui travaille maintenant au T. J. Watson Research Center, aprés être passé par l'Université Rice et l'Université d'Illinois à Urbana-Champaign, a bien voulu s'intéresser à ce travail et prendre la peine d'en lire une version rédigée en français.

W. Jalby et Monsieur le Professeur A. Sameh ont influencé considérablement le cours de mes travaux en me permettant d'avoir accès à un multiprocesseur vectoriel Alliant FX/8. Ceci a permis de voir quels résultats le partitionnement de boucles pouvait apporter et de mesurer la difficulté que représentait la génération de code. W. Jalby a aussi participé à de nombreuses discussions sur le problème des mouvements de données dans l'Alliant. Qu'ils soient tous les deux assurés de ma gratitude.

Il serait injuste de ne pas remercier Monsieur le Professeur M. Lenci, Directeur du Centre d'Automatique et Informatique de l'Ecole des Mines, qui m'a supporté dans son équipe pendant de nombreuses années et m'a laissé le temps nécessaire pour écrire une thèse. Il a de plus joué un rôle ingrat en assurant la lecture intégrale de la dernière rédaction, et ce avec succès puisqu'il a permis d'éliminer de

nombreuses erreurs. Cette relecture a aussi été assurée par Monsieur le Professeur P. Feautrier et Monsieur le Professeur J. Irigoin, mon père, qui m'ont tous les deux donnés de nombreuses indications sur les points à améliorer.

Je voudrais aussi dire merci à mes collègues du C.A.I., pour le plaisir que j'ai à travailler avec eux, et à mes parents, pour le soutien discret, et parfois inquiet, qu'ils m'ont apporté pendant des années. Ce merci s'adresse aussi à Judith, à Caroline, à Steve et à quelques quadripèdes qui ont subi, sans toujours le savoir, le contrecoup de mes travaux.

Enfin, il est d'usage de remercier le personnnel technique qui a assuré la réalisation matériel de cet ouvrage. Je ne dirai jamais assez à quel point j'ai apprécié la patience, la gentillesse et la disponibilité de François Irigoin tout au long de la saisie et de la composition de ce document. Plus sérieusement, il me faut rappeler l'aide que m'ont apportée J. Altimira, A. Le Gallic et A. Pech, des recherches bibliographiques à la mise en page finale. Le travail de DEA de J. Teulon m'a aussi été bien utile puisque la plupart des figures ont été réalisées à l'aide de GRAPHEMINES, le logiciel expérimental qu'il a développé au C.A.I.

TABLE DES MATIERES

1. OBJECTIFS, HYPOTHESES ET NOTATIONS

1.1. Objectifs

1.1.1. Les machines-cibles potentielles et leur modélisation

1.1.2. Décomposition de l'objectif général de minimisation du temps d'exécution

1.1.3. Améliorations espérées

1.2. Exemple

1.2.1. Le problème

1.2.2. Code séquentiel

1.2.3. Application d'un restructureur

1.2.4. Adaptation à un multiprocesseur

1.2.5. Problème de synchronisation posé par un partitionnement bidimensionnel

1.2.6. La solution proposée

1.3. Hypothèses de travail

1.3.1. Portions de code considérées

1.3.2. Application préalable d'un paralléliseur

1.3.3. Architectures considérées

1.3.4. Application automatique de la méthode

1.3.5. Découpage de l'optimisation en étapes indépendantes

1.4. Notations et définitions

1.4.1. Instruction et exécution d'une instruction - Noeud et supernoeud

1.4.2. Vecteur d'itération, espace d'itération, domaine d'itération

1.4.3. Dépendances

1.4.4. Vecteurs, graphe et cône de dépendance

1.4.5. Rappel sur les calculs en nombres entiers

1.5. Conclusion

2. PARTITIONNEMENT

2.1. Contraintes de partitionnement.

2.1.1. Synchronisation externe

2.1.2. Unicité de forme.

2.1.3. Partitionnement exact.

2.1.4. Conclusion

2.2. Graphe et cône de dépendance.

2.2.1. Connexité du graphe de dépendance et dimension du cône

2.2.2. Saturation des cônes de dimension inférieure à N.

2.2.3. Cône de dimension N et graphe non connexe.

2.3. Partitionnement par hyperplans

2.3.1. Définitions

2.3.2. Partitionnement valide

2.3.3. Inconvénient du partitionnement par hyperplans

2.3.4. Hyperplans critiques

2.3.5. Vecteur de direction d'itération

2.4. Base de partitionnement

2.4.1. Agglomération par une base. Définition et validité

2.4.2. Relation avec le partitionnement par hyperplans

2.4.3. Les différentes bases considérées

2.5. Facteur d'échelle λ

2.6. Conclusion

3. GENERATION DE CODE

3.1. Choix d'un ordonnancement global: les fronts

3.1.1. Solutions possibles

3.1.2. Choix de la solution "fronts"

3.1.3. Squelette de code

3.2. Génération du code de contrôle global

3.2.1. Base de parcours G

3.2.2. Les différents types de supernoeuds partiels

3.2.3. Choix de l'origine du partitionnement

3.2.4. Calcul des bornes des boucles

3.2.5. Tests de nature des supernoeuds

3.2.6. Squelette de code résultant

3.3. Code d'un supernoeud complet

3.3.1. Parcours des noeuds d'un supernoeud

3.3.2. Direction d'itération libre

3.3.3. Direction d'itération fixée. Cas favorable

3.3.4. Direction d'itération fixée. Cas général.

3.4. Code de contrôle pour un supernoeud partiel

3.4.1. Identification des types de supernoeuds partiels le long d'une frontière

3.4.2. Génération de code optimal, scalaire ou vectoriel

3.4.3. Génération de code simple

3.4.4. Cas des supernoeuds intersectant plusieurs frontières

3.4.5. Ajout du code mal imbriqué

3.5. Modification des références aux variables

3.5.1. Accès automatique

3.5.2. Problèmes posés par la copie optimale

3.5.3. Copie par excès avec accès linéaire

3.5.4. Copie exacte avec accès pseudo-linéaire

3.5.5. Références multiples au même tableau

3.5.6. Variables locales

3.5.7. Références et variables en entrée ou en sortie

3.5.8. Déclarations des tableaux de copie

3.5.9. Conclusion

3.6. Génération du code de copie

3.6.1. Copie programmée

3.6.2. Transfert par message

3.6.3. Opérateurs spécialisés de transfert

3.6.4. Exécution parallèle des transferts et de données des calculs

3.6.5. Comparaison des différents types de copie

3.7. Conclusion

4. COUTS ET CONTRAINTES DUS AUX MIGRATIONS DES DONNEES

4.1. Types d'architectures étudiés

4.1.1. Cache local

4.1.2. Cache global

4.1.3. Mémoires locales et mémoire globale (Cedar, Cray-2)

4.1.4. Mémoires locales et transferts centralisés (multi-array-processor avec host)

4.1.5. Mémoires locales et communications par messages

4.1.6. Mémoire virtuelle

4.1.7. Conclusion

4.2. Evaluation des volumes référencés. Optimisation de P.

4.2.1. Calcul direct du volume relatif à une référence

4.2.2. Calcul dual du volume relatif à une référence

4.2.3. Calcul exact du volume d'une copie par excès avec accès linéaire

4.2.4. Calcul direct du volume d'une copie d'une famille de références

4.2.5. Calcul de volume par simulation

4.2.6. Récapitulatif des résultats obtenus.

4.3. Volume de stockage

4.3.1. Cas de la dimension 2

4.3.2. Cas des expressions d'index mono-indicées

4.3.3. Cas général: simulation

4.4. Ordonnancement des transferts par rapport aux calculs

4.5. Exemple

4.6. Conclusion

4.6.1. Politique d'assignation

4.6.2. Expression du coût de transfert

4.6.3. Expression de la contrainte sur la mémoire rapide

4.6.4. Influence asymptotique de la taille de la mémoire rapide

5. TEMPS DE CALCUL D'UN SUPERNOEUD

5.1. Influence de l'architecture du processeur élémentaire

5.1.1. Processeur élémentaire scalaire

5.1.2. Processeur vectoriel à faible $n_{1/2}$

5.1.3. Processeur vectoriel à fort $n_{1/2}$

5.2. Coûts et contraintes

5.2.1. Coûts

5.2.2. Contraintes sur λ

5.2.3. Contraintes sur la forme

5.3. Conclusion

6. CHOIX DE LA FORME ET DE LA TAILLE DES SUPERNOEUDS

6.1. Temps d'exécution d'un supernoeud

6.2. Nombre de processeurs illimité

6.2.1. Calcul du temps total

6.2.2. Les étapes de l'optimisation

6.2.3. Construction de l'ensemble des vecteurs critiques

6.2.4. Choix d'une base de partitionnement à directions fixées

6.2.5. Sélection des meilleures directions de partitionnement

6.2.6. Optimisation de la taille des supernoeuds

6.2.7. Calcul du nombre maximum de processeurs utilisables

6.2.8. Conclusion

6.3. Nombre de processeurs limité

6.3.1. Overhead global

6.3.2. Temps de calcul d'un supernoeud partiel

6.3.3. Equilibrage de charge dans un front

6.3.4. Evolution du degré de parallélisme des fronts

6.3.5. Fonctionnement en régime stationnaire

6.3.6. Approximation du temps global d'exécution

6.4. Critères de choix

7. EXEMPLE

7.1. Programme à restructurer

7.2. Partitionnements possibles

7.3. Calcul des transferts de données

7.4. Choix d'une direction d'itération globale

7.5. Choix du facteur d'échelle *ll*

7.6. Génération du code de contrôle global

7.6.1. Principes généraux

7.6.2. Calcul des bornes des boucles DO

7.6.3. Test des supernoeuds vides

7.6.4. Test des supernoeuds complets

7.6.5. Test d'un supernoeud frontière

7.6.6. Tests des supernoeuds coins

7.6.7. Code de contrôle global complet

7.7. Code de contrôle des supernoeuds

7.7.1. Code d'un supernoeud complet

7.7.2. Code de contrôle d'un supernoeud partiel

7.8. Modification des références aux variables

7.9. Génération du code de copie

7.10. Résultats expérimentaux

7.11. Conclusion

8. COMPARAISONS AVEC D'AUTRES METHODES

8.1. Méthode des vagues et méthode hyperplane

8.2. Echange de boucles

8.3. Adaptation des programmes à la mémoire virtuelle

8.4. Partitionnement avec synchronisation de J.-K. Peir

8.5. Systolisation

8.6. Etude manuelle de stratégies de décomposition

8.7. Optimisation du produit de matrice sur l'Alliant FX/8

8.8. Autre travaux d'optimisation manuelle de code ou d'algorithmes

8.9. Conclusion

INTRODUCTION

Le calcul scientifique et ses besoins illimités en puissance de calcul et en volume mémoire ont toujours attiré les concepteurs d'ordinateurs mais ces derniers n'ont pu commencer à mettre en pratique leurs idées que très récemment grâce aux progrès incessants de la technologie des circuits intégrés. Dès que la densité nécessaire à l'intégration d'une unité de calcul flottant a été atteinte et même un peu avant pour ceux qui voulaient obtenir une part de marché, tout le monde a pu espérer concevoir et réaliser une machine d'une puissance comparable à celle du Cray 1 sans avoir à faire de trop gros investissements.

De nombreuses start-up américaines ont donc vu le jour pour exploiter la vieille idée des multiprocesseurs et l'année 1985 a vu entre autres la naisssance de l'Alliant FX-8[1] , un octo-processeur vectoriel ayant la puissance d'un petit Cray pour un dixième du prix comme le disent les ingénieurs technico-commerciaux. Cette machine présente pour nous l'intérêt d'intégrer beaucoup d'idées architecturales nouvelles, d'avoir été choisie comme processeur élémentaire pour le projet CEDAR[7] du Centre pour la Recherche et le Développement des Superordinateurs (*Center for Supercomputing Research and Development*) de l'Université d'Illinois à Urbana-Champaign, et d'avoir fait l'objet de nombreux travaux de la part de l'équipe *Applications* de ce centre. Ces derniers ont mis **plusieurs mois** pour arriver à optimiser le sous-programme pourtant on ne peut plus simple qui effectue le produit de matrice[2] . L'effort en valait la peine puisque la vitesse d'exécution a presque été multipliée par cinq, passant d'un maigre 10 MFlops (MFlops: millions d'opérations flottantes par seconde) à près de 50 ce qui est tout à fait acceptable en regard de la puissance théorique de pointe qui est de moins de 70 MFlops. Mais combien d'utilisateurs sont prêts à payer ce prix pour optimiser leurs programmes d'application? Combien de centaines de lignes de code compte la version optimisée des **cinq** lignes du produit de matrice?

Tout aussi révélatrice d'un problème est la proportion fréquemment citée de 1 pour 20 entre programmes utilisant les possibilités de *multitasking* du Cray X-MP et programmes séquentiels classiques n'exploitant que les unités vectorielles du Cray X-MP. Il suffit de lire la notice d'utilisation des primitives de gestion de tâches pour voir que les contraintes qu'elles impliquent sur les déclarations des données et sur la structure générale d'un programme même très simple ont de quoi décourager de nombreux utilisateurs.

Les architectures performantes mais compliquées des processeurs scientifiques ne permettent donc pas aux usagers de profiter directement de leur puissance potentielle faute d'outils logiciels permettant d'effectuer automatiquement les transformations qu'imposent les deux ou trois niveaux de parallélisme et les multiples couches de hiérarchies mémoire qui vont des registres aux disques en passant par les antémémoires (aussi appelées cache), les mémoires locales, les mémoires principales et les mémoires d'arrière-plan, qui caractérisent ces architectures.

Les solutions proposées auparavant

Plusieurs solutions ont été proposées pour résoudre ce problème. La première, qui est à la fois la plus simple pour le constructeur et la plus ancienne car elle ne nécessite pas de longues études théoriques, consiste à fournir aux utilisateurs une bibliothèque scientifique de base adaptée manuellement à leur machine. Il leur *suffit* de recoder leurs programmes en utilisant les primitives fournies dans la bibliothèque pour obtenir des performances honorables sur la plupart de leurs programmes. Et de recommencer l'opération le jour où ils changent de machine... ce qui est un atout supplémentaire pour le vendeur initial tant que les bibliothèques ne sont pas normalisées. C'est par exemple la première voie qu'a choisie le groupe *Applications* du CSRD précédemment cité. La bibliothèque scientifique BLAS[3] (*Basic Linear Algebra Subprograms*), conçue initialement avec des opérations sur les vecteurs a été réécrite successivement avec des opérations mixtes portant sur un vecteur et une matrice (BLAS 2) puis avec des opérations sur les matrices uniquement (BLAS 3) quand le groupe s'est rendu compte que c'était le type d'opérations qui utilisait au mieux le potentiel de l'Alliant FX-8.

La deuxième solution consiste à reprendre des méthodes anciennes, développées pour les machines vectorielles, et à les adapter rapidement mais sommairement aux problèmes nouveaux. C'est ainsi que le préprocesseur VAST qui est fourni avec le compilateur Fortran de l'Alliant pour découvrir automatiquement le parallélisme des programmes qui lui sont soumis se contente généralement de découper les boucles vectorielles pour fournir des tronçons élémentaires à chacun des huit processeurs ce qui ne permet ni de profiter d'une proportion importante du parallélisme contenu dans le programme, ni d'exploiter pleinement les possibilités de la machine.

Une troisième solution consiste à utiliser la structure en modules du programme pour identifier des tâches susceptibles d'être exécutées en parallèle sur des processeurs élémentaires différents. Aucun produit commercial n'utilise pour le moment cette possibilité dont la faisabilité théorique a été démontrée par Rémi Triolet dans sa thèse[5] et dont la mise en oeuvre a été effectuée sous forme d'une phase supplémentaire de Parafrase[6] par ce dernier au cours de l'année qu'il a passée au CSRD dans une des équipes du projet CEDAR.

Une nouvelle approche

Mais cette dernière solution ne modifie pas les programmes au niveau le plus fin. Elle ne permet donc pas de satisfaire l'ensemble des contraintes imposées par les superordinateurs modernes, c'est-à-dire d'assurer que chaque processeur élémentaire ait du travail en permanence, que ce travail puisse être exécuté le plus efficacement possible avec des unités vectorielles, qu'il s'effectue avec un minimum de données de manière à utiliser au mieux la hiérarchie mémoire, et enfin que le temps passé à synchroniser l'ensemble des processeurs reste faible par rapport au temps passé à effectuer des calculs.

Cet ensemble d'objectifs est beaucoup trop contraignant pour qu'une première étude du problème permette d'en donner une solution générale pour tous les types de programmes. Nous nous sommes donc

restreint à des séquences d'opérations bien particulières, les boucles imbriquées, qui sont particulièrement fréquentes dans les programmes scientifiques où l'on calcule un certain nombre de grandeurs physiques sur un domaine bi- tri- et quelquefois quadri-dimensionnel. Chaque boucle correspond alors à une dimension d'espace ou au temps, et le volume de calculs, qui est proportionnel au produit des longueurs des différentes boucles, est énorme par rapport au volume de code qui lui a donné naissance. L'optimisation de ces séquences doit conduire à un gain absolu en temps important même si le gain relatif est limité et elle est plus facile que pour des séquences de code quelconque car l'espace des calculs et les relations entre les calculs sont réguliers. De plus, le volume des calculs est si important que la taille des tâches va pouvoir être ajustée d'une manière quasi-continue.

La méthode de partitionnement que nous proposons dans ce travail est basée sur l'idée intuitive que les valeurs calculées en un point d'un espace physique quelconque ne sont pas utilisées à court terme, directement ou indirectement, pour évaluer les grandeurs intéressantes en un autre point du domaine, éloigné du premier. Concrètement, l'évolution des conditions climatiques à Paris peut être calculée sans prendre en compte la situation de New-York, du moins pendant un certain temps. La propagation d'influences d'un point à un autre de l'espace physique se traduit dans les calculs par des relations producteurs-consommateurs entre points de l'espace des calculs plus ou moins voisins, relations qui sont appelées **dépendances**. Ces relations entre points de l'espace des calculs sont fréquemment des relations entre voisins, même quand les calculs ne correspondent à aucun phénomène physique. C'est le cas, par exemple, de la multiplication de matrice et de nombreux algorithmes d'analyse numérique qui sont fondés sur des schémas à n points.

L'étude des dépendances va nous permettre d'une part de séparer les calculs qui sont effectués sur des données indépendantes et de les effectuer en parallèle, sous forme de vecteurs de calculs et/ou sous forme de tâches confiées à des processeurs élémentaires, et d'autre part de regrouper les calculs effectués avec les mêmes données de manière à exploiter au mieux la hiérarchie mémoire de la machine. Les deux objectifs, qui consistent l'un à effectuer des opérations en parallèle et l'autre à effectuer des opérations localement se révèlent donc complémentaires.En revanche, des compromis sont nécessaires entre le parallélisme vectoriel et le parallélisme de tâches et entre le degré de parallélisme et le temps perdu à effectuer des synchronisations entre les tâches.

L'adaptation manuelle de programmes à des multiprocesseurs a déjà fait l'objet de nombreuses études qui ont permis de distinguer plusieurs problèmes. Le premier est la constitution des tâches qui jusqu'à présent étaient généralement considérées comme **données** avec le programme. Le second est l'affectation statique ou dynamique des tâches aux processeurs en fonction des relations entre tâches et des liens de communications qui existent entre les processeurs. Le troisième problème est l'ordonnancement de ces tâches qui devient nécessaire dès que le nombre des tâches qui peuvent être effectuées en parallèle est supérieur au nombre de processeurs. Ces deux derniers problèmes ont été étudiés davantage que le premier et, bien que les trois soient en fait intimement liés, nous nous consacrerons aussi exclusivement

que possible au premier, la constitution des tâches, aussi appelé **partitionnement**, ce qui explique le titre de notre travail.

Organisation de la thèse

Nous présentons notre travail en huit chapitres consacrés successivement aux objectifs, hypothèses et notations, au partitionnement à proprement parler, à la génération de code correspondante, à l'évaluation des coûts et des contraintes dus aux données, à l'optimisation du temps de calcul des tâches élémentaires et enfin à l'optimisation globale du partitionnement. Le premier des deux chapitres restants nous permet ensuite de mettre en oeuvre les chapitres précédents de manière continue sur un exemple et le deuxième situe notre travail par rapport aux méthodes précédemment connues. L'ordre des chapitres risque de ne pas être toujours bien compris du lecteur car il peut sembler absurde d'effectuer la génération de code avant d'avoir choisi un partitionnement optimal. C'est vrai lorsqu'il s'agit d'appliquer le partitionnement à un programme particulier, mais l'exposé des procédés d'optimisation ne peut s'effectuer avant d'avoir déterminé l'objet à optimiser qui est le code produit par partitionnement et non le partitionnement en soi.

Le premier chapitre comporte d'abord une introduction rapide aux solutions matérielles qui sont proposées pour résoudre les problèmes propres au calcul scientifique, illustrée de nombreux exemples de superordinateurs. La variété de ces solutions nous conduit à limiter l'ensemble des machines cibles aux machines disposant d'une mémoire globale, ce qui élimine les architectures du type hypercube en dépit de la mode actuelle, et à modéliser les machines en ne conservant que quelques caractéristiques comme le nombre de processeurs, leurs éventuelles capacités vectorielles, la taille et la nature des mémoires (antémémoire ou mémoire locale). Nous ne prenons aussi en compte que deux niveaux de parallélisme et deux niveaux dans la hiérarchie mémoire.

Un exemple est ensuite présenté informellement pour donner une idée des problèmes qui se posent puis nous précisons des hypothèses de travail sur le code qui est accepté en entrée, les informations que nous devons avoir sur lui, les machines cibles, la décomposition des étapes d'optimisation. Le fait qu'il est indispensable de pouvoir à terme appliquer le partitionnement d'une manière automatique ou tout au moins semi-automatique limite aussi l'espace des solutions.

Enfin, nous terminons ce long chapitre introductif par des rappels sur les bases de la parallélisation automatique de programmes et des définitions précises des concepts pour lesquels un sens précis n'est pas encore accepté d'une manière générale.

Le deuxième chapitre est le coeur de la méthode. Nous montrons d'abord que les hypothèses qui sont raisonnablement nécessaires pour permettre une génération de code automatique et une optimisation des choix limitent l'ensemble des solutions. Deux paradigmes duaux permettent de constituer des tâches:

le découpage de l'ensemble global en parties de plus en plus petites et l'agglomération de calculs élémentaires en des ensembles de plus en plus importants. Nous proposons d'abord d'effectuer un découpage de l'espace des calculs par des hyperplans et ensuite de regrouper les calculs qui se trouvent dans les cellules unités du maillage créé par une matrice, que nous appelons matrice de partitionnement. Comme on pouvait s'y attendre, les deux approches donnent le même résultat mais le découpage par hyperplans, qui peut aussi être caractérisé par une matrice, est plus adapté à l'optimisation tandis que l'agglomération par matrice de partitionnement fournit une méthode de génération de code automatique.

Cette génération de code automatique, objet du chapître trois, présente de très nombreux problèmes qui ont longtemps semblé tout à la fois impossibles à résoudre automatiquement et trop délicats pour être traités à la main sans risque de commettre de nombreuses erreurs. La première et seule génération de code que nous avons effectuée uniquement avec du papier et un crayon nous a rapidement convaincu de l'intérêt de la génération automatique.

Pourtant l'ensemble des problèmes - c'est-à-dire la génération du code de contrôle global qui assure l'enchaînement des tâches et l'identification des tâches particulières apparaissant aux frontières du domaine des calculs, la génération de code vectoriel ou parallèle de chacun des types de tâches, quelle que soit leur forme, l'instantiation de copie dans les machines à mémoires locales et enfin la génération du code de copie - est systématiquement résolu par le recours à la théorie des polyèdres convexes.

Cette théorie apparaît régulièrement en parallélisation et en transformation de programmes car les fonctions linéaires marquent fréquemment les limites de ce qu'on sait faire. Il n'est donc pas étonnant que notre méthode, qui présuppose le calcul des informations traditionnelles en parallélisation, y ait aussi recours. Il faut noter que les résultats obtenus sont optimaux, qu'il s'agisse de calculs de bornes aussi précises que possibles ou de simplification de tests, qu'ils sont généralement meilleurs que ceux que nous avions obtenu manuellement auparavant, et enfin qu'ils sont toujours corrects...

Le chapitre quatre a deux objectifs. Il faut voir d'une part si le choix d'un partitionnement et d'un ordonnancement local à une tâche a une influence sur la réutilisation des données contenues dans la mémoire la plus rapide et quelle est cette influence, et d'autre part voir quelles contraintes la taille de cette mémoire peut entraîner sur la taille des tâches, en fonction du partitionnement et de l'ordonnancement local choisis.

L'optimisation du partitionnement et de l'ordonnancement par les accès aux données n'a pas abouti à des résultats aussi complets et exploitables que nous le souhaitions. Les références aux données sont nombreuses même dans un simple corps de boucle et donnent des indications contradictoires. Il n'est pas toujours possible de procéder à des évaluations **symboliques**, seules susceptibles de permettre une optimisation. Enfin, l'optimisation de l'exécution des calculs est un problème primordial par rapport à celui de l'optimisation des accès aux données et cette dernière ne doit donc être tentée que quand le parallélisme

du programme est supérieur à celui de la machine, comme c'est le cas, par exemple, avec la multiplication de matrice.

Heureusement, le calcul de la taille que doivent avoir les tâches et les parties de tableaux qu'il faut copier dans un cache ou une mémoire locale est faisable dès que les directions de partitionnement sont fixées numériquement. On remarquera qu'il faut évaluer non seulement le volume de transfert mais aussi le volume de stockage et que ce dernier dépend de l'ordonnancement des transferts mais le temps a manqué pour étudier ce problème.

Le chapitre suivant est extrêmement bref car l'optimisation de l'exécution d'une tâche sur un processeur élémentaire est un problème qui a été étudié pour les processeurs de la génération précédente comme le Cray 1. On montrera cependant que le choix de la **forme** et, non seulement celui de la taille du partitionnement, dépend du temps d'amorçage des unités vectorielles. Il faut s'arranger pour que les vecteurs aient des longueurs constantes quand ce temps est important.

Deux cas sont envisagés dans le chapitre 6 qui fait une synthèse des chapitres précédents pour répondre à la question: quel partitionnement choisir? Le premier est celui d'une machine théorique disposant d'un nombre infini de processeurs. Cette hypothèse permet d'obtenir une forme symbolique du temps d'exécution qui permet de déterminer les conditions d'existence d'un partitionnement idéal ou la dégénérescence en l'ordonnancement donné par la méthode hyperplane avec réduction des tâches à un calcul unique ou en l'ordonnancement séquentiel par constitution d'une tâche unique. Les coefficients de la fonction et donc les valeurs caractérisant le partitionnement optimal, quand il existe, ne sont pas connus car ils dépendent principalement du générateur de code utilisé et non du code source que nous prenons comme entrée. En dépit du caractère théorique de l'hypothèse, ce calcul nous permet donc de d'obtenir un résultat utile, le nombre maximum de processeurs qui peuvent être utilisés.

Le deuxième cas consiste à ne faire aucune hypothèse sur la machine et à étudier qualitativement les différents termes qui interviennent dans le temps d'exécution total. Cela ne conduit pas à la définition d'un algorithme de choix mais permet de dresser la liste des facteurs qu'il faut prendre en compte pour choisir un partitionnement et de préciser comment il sont sensibles à celui-ci.

Le chapitre sept est une illustration presque complète des chapitres qui le précèdent. Un exemple unique est suivi. On part du code source séquentiel produit habituellement par les programmeurs pour finir avec deux versions, une pour machine à cache ou à un seul niveau mémoire et une pour machine a mémoire locale. Nous présentons aussi les résultats d'une expérience de partitionnement effectuée manuellement pour l'Alliant FX-8 sur ce même exemple qui a permis d'obtenir un gain de vitesse de 50 % uniquement grâce à une meilleure réutilisation des données présentes dans le cache.

La comparaison qui termine ce travail permet d'abord de montrer que le partitionnement est une généralisation de la méthode hyperplane, de l'échange de boucles et de certaines des techniques qui avaient été proposées pour améliorer l'exécution des programmes scientifiques sur les machines à mémoire virtuelle. Comme la méthode hyperplane, elle suppose cependant des informations sur les dépendances que l'école américaine estime trop coûteuses à calculer. C'est pourquoi deux des trois types de partitionnements que propose J.-K. Peir[4] semblent un peu simplistes. Ce ne sont d'ailleurs aussi que des cas particuliers de notre méthode. Mais seules des expériences permettraient de déterminer la fréquence, parmi des programmes réels, des cas qui profitent d'une méthode plus compliquée.

La systolisation automatique d'algorithme est fondée sur les mêmes idées et sur les mêmes concepts que la méthode hyperplane ou le partitionnement décrit dans ce travail, bien qu'elle se prévale de travailler sur des équations plutôt que sur des programmes impératifs procéduraux, i.e. des programmes Fortran. Il est donc intéressant de relever les points communs et les différences qui existent entre deux domaines qui s'ignorent mutuellement.

Enfin, de nombreux travaux ont été effectués manuellement, au coup par coup, pour un algorithme particulier et pour une machine déterminée. C'est le cas du produit de matrice dont nous parlions au début de cette introduction. Son importance fondamentale en analyse numérique lui a valu une étude approfondie et il est important de voir ce qu'une méthode générale et automatisable perd par rapport à une étude spécifique et manuelle. De nombreux autres algorithmes ont été adaptés à des multiprocesseurs à hiérarchie mémoire, mais il n'est pas possible de les étudier tous.

Certains chapitres contiennent plus d'informations utiles ou de résultats exploitables que d'autres. Une lecture rapide peut être limitée à la section 1.4 du premier chapitre, au chapitre 2, et spécialement à ses sections 2.3 et 2.4, au chapitre 3, en portant une attention particulière aux sections 3.2 à 3.4, et enfin à la conclusion du chapitre 6 (6.4). Un dernier coup d'oeil sur le chapitre 7 devrait suffire pour se faire une idée des résultats obtenus: calcul de l'ensemble des partitionnements valides et vérification de la validité d'un partitionnement -automatiques tous les deux-, génération de code automatique, applicable aux méthodes dont le partitionnements en supernoeuds est une généralisation (échange de boucle, méthode hyperplane, partitionnement en blocs rectangulaires), évaluation des volumes de données mis en jeu, critères de sélection d'un partitionnement et, enfin, mise en oeuvre expérimentale permettant d'obtenir une augmentation de vitesse de 50%.

References

1. R. Gottlieb, K. Kimball, T. Jaskiewicz, and R. Swift, "A New Way To Speed Up a Supercomputer," *Electronics* **58**(30), pp.56-58 (July 1985).

2. W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Two-Level Memory Hierarchy," Technical Report CSRD 555, University of Illinois at Urbana-Champaign, Urbana-Champaign (Feb. 1986).

3. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krough, "Basic Linear Algebra Subprograms for FORTRAN Usage," *ACM Transactions on Mathematical Software* **4**(3), pp.308-323 (Sept. 1978).

4. J.-K. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," PhD Thesis, Report UIUCDCS-R-86-1259, University of Illinois at Urbana-Champaign (March 1986).

5. R. Triolet, "Contribution à la parallélisation automatique de programmes," Thèse de Docteur-Ingénieur, Université Paris VI (Déc. 1984).

6. R. Triolet, "Interprocedural Analysis for Program Restructuring with PARAFRASE," Technical Report CSRD-538, University of Illinois, Center for Supercomputing Research & Development, Urbana-Champaign (1985).

7. P.-C. Yew, "Architecture of the Cedar Parallel Supercomputer," CSRD Report 609, University of Illinois at Urbana-Champaign (Aug. 1986).

1. OBJECTIFS, HYPOTHESES ET NOTATIONS

Nous allons tout d'abord voir sur de nombreux exemples que les architectures conçues récemment pour le calcul scientifique présentent toutes au moins deux sources de parallélisme, grâce à la multiplication des processeurs et à l'adoption d'instructions vectorielles, et une hiérarchie mémoire seule susceptible de fournir à la fois le débit, le volume et le temps d'amorçage nécessaires. Nous montrerons que ces caractéristiques nécessitent un découpage des programmes en tâches indépendantes susceptibles de s'exécuter sur les processeurs élémentaires, et un ordonnancement des calculs permettant à la fois une exécution vectorielle et une bonne localité des références.

Nous traiterons ensuite informellement un exemple concret en partant du code séquentiel Fortran que les programmeurs ont l'habitude de produire pour arriver à une reformulation du code permettant d'exploiter pleinement une architecture multiprocesseur vectorielle.

La généralité du problème de l'adaptation d'un code séquentiel à une machine parallèle en rend impossible une solution générale. Nous allons donc détailler les hypothèses de travail choisies pour obtenir les premiers résultats dans un domaine encore largement inexploré. Nous ne considérerons que de petites parties de code, les boucles DO parfaitement imbriquées, nous supposerons que tout l'acquis de la vectorisation est à notre disposition et nous limiterons l'ensemble des machines cibles. Seuls deux niveaux de parallélisme et deux niveaux de hiérarchie mémoire seront pris en compte. L'automatisation complète de la méthode restera un objectif à long terme car de nombreux problèmes sont encore insolubles, ne serait-ce que parce que les données nécessaires, comme le temps d'exécution d'une séquence d'instructions, sont inconnues. Nous nous limiterons donc à une application manuelle ou semi-automatique de la méthode.

Enfin, la dernière partie de ce chapitre sera consacrée aux notations et aux définitions. Le domaine de la parallélisation et de la vectorisation est encore jeune et en plein foisonnement. Aussi de nombreux termes sont-ils utilisés avec des sens différents par telle ou telle école. Nous donnerons donc des définitions pour les notions de base utilisées dans les chapitres suivants.

1.1. Objectifs

Nous allons d'abord présenter de plusieurs architectures de machines conçues pour le calcul numérique et voir qu'elles conduisent à un problème de partitionnement et de réordonnancement. La résolution de ce problème nécessite une modélisation qui conservera le minimum d'information suffisant pour optimiser le temps d'exécution des programmes (§ 1.1.1). Ensuite, nous énumérerons les facteurs sur lesquels on peut agir pour faire varier ce temps d'exécution (§ 1.1.2) et nous essaierons enfin d'estimer a priori les améliorations que nous pouvons apporter (§ 1.1.3).

1.1.1. Les machines-cibles potentielles et leur modélisation

Nous nous intéressons essentiellement aux machines conçues pour le calcul scientifique. Prenons par exemple un modèle sismique tridimensionnel[44] . Chaque dimension est discrétisée en 512 points. Le calcul à effectuer est une transformation de Fourier discrète. Chaque résultat est un nombre complexe qui va être évalué moyennant $O\,(log_2(512 \times 512 \times 512))$ opérations qui nécessitent que le tableau ($512 \times 512 \times 512 \times 2$ mots $= 2048$ Mo) passe en mémoire au moins deux fois. Comme on le voit sur cet exemple, le calcul scientifique se caractérise à la fois par un recours massif à des opérations flottantes sur 64 bits et par la manipulation relativement régulière de très grosses masses de données.

Après avoir vu quels étaient les apports récents de la technologie (§ 1.1.1.1), nous allons passer en revue les différentes solutions architecturales qui ont été proposées sans nous laisser enfermer dans la traditionnelle classification de Flynn[20] ni dans celles, plus précises, que proposent Kuck[46] ou Hockney[35] car aucune d'entre elles ne permet d'en couvrir l'ensemble (§ 1.1.1.2). Nous présenterons ensuite quelques exemples de machines réelles, mettant en oeuvre certaines de ces idées. Ces machines sont regroupées suivant la nature de leur hiérarchie mémoire (§ 1.1.1.3 à 1.1.1.5). Nous expliquons ensuite quelles nous semblent être parmi les caractéristiques de ces machines celles qui jouent un rôle critique dans l'adaptation ou l'écriture de programmes ciblés sur elles (§ 1.1.1.6). La plupart des exemples donnés dans les paragraphes 1.1.1.3 à 1.1.1.5 sont modélisés selon ces critères dans le dernier paragraphe (§ 1.1.1.8).

1.1.1.1. L'avancée technologique

Le progrès de la technologie n'introduit pas directement de contraintes sur la programmation des machines, mais il modifie les coûts relatifs des différentes composants d'une machine (unité arithmétique et logique, unité de contrôle, etc...) et favorise donc des solutions architecturales qui varient avec le temps.

Le temps d'exécution des opérations flottantes a été progressivement réduit grâce à l'augmentation de la densité d'intégration et par l'adoption d'architectures pipelinées. Les sociétés Weitek et Analog Devices, par exemple, vendent des opérateurs flottants en un boîtier qui ont des performances brutes de 5 à 10 MFlops[14] . La première annonce même un ensemble de boitiers appelé *ACCEL*[67] permettant de

construire une machine ayant 25 Mflops de vitesse de pointe. De tels opérateurs permettraient d'envisager de doter même les microordinateurs de puissances de calcul considérables si les mémoires de ces machines étaient capables de fournir les opérandes et de ranger les résultats à une vitesse suffisante. Le débit d'un opérateur flottant tournant à 10 Mflops est de 3×8 octets toutes les 100 ns soit 240 Mo/s. On est loin des capacités des bus mémoire traditionnels.

Le **volume** de la mémoire, contrairement à son débit, ne pose pas à l'heure actuelle de problèmes autres que financier. La fiabilité des composants est suffisante pour proposer des mémoires de 256 millions de mots de 64 bits. Les erreurs *soft* des composants, qui sont dues à des radiations ou à des particules cosmiques, sont corrigées par des codes détecteurs correcteurs d'erreurs qui n'augmentent pas trop le temps d'accès, les pannes matérielles sont suffisamment rares et la puissance consommée reste à un niveau acceptable.

Bien que considérable, le progrès n'a pas été aussi rapide dans le domaine des mémoires que dans le domaine des opérateurs. Dans un article[14] montrant l'influence de l'intégration VLSI sur l'architecture des calculateurs scientifiques, Charlesworth et Gustafson donnent les chiffres suivants sur les surfaces de circuit imprimé et sur les puissances nécessaires aux trois unités fonctionnelles de base:

Unité fonctionnelle	1979	1985	amélioration
Unité arithmétique et logique (10 à 20 MFlops)	2450 sq.in. 760 W	39 sq.in. 18 W	63 42
Unité de contrôle et génération d'adresses	700 sq.in. 240 W	6 sq.in. 4 W	116 60
Mémoire (250 ns, 1 Mégamot de 64 bits)	5600 sq.in. 430 W	350 sq.in. 40 W	16 11

Ces chiffres demanderaient à être confirmés par d'autres sources mais les fabricants les indiquent rarement. Il en ressort cependant que le problème architectural se trouve maintenant plus dans la mémoire de la machine que dans l'unité arithmétique et logique ou dans l'unité de contrôle. Des considérations similaires sont développées brièvement dans [24].

Les technologies magnétiques sont aussi utilisées pour accroître encore le volume de la mémoire. Contrairement à ce que l'on croit souvent, des progrès sont encore possibles et l'évolution a même été aussi rapide que dans le domaine des circuits intégrés avec un doublement de la densité tous les deux ans et demi pendant trente ans[34] . Le progrès est plus limité pour les temps d'accès moyens qui se rapprochent néanmoins de la dizaine de millisecondes et pour les débits quoiqu'on puisse trouver des disques spéciaux assurant un débit **continu** de plus de 10 Mo/sec. C'est le cas des disques DD-550 proposés pour

le supermini SCS 40 qui, avec quatre têtes de lecture, transfèrent des blocs de 16 Ko à 10.2 Mo/s en régime continu et à 12 Mo/s en vitesse de pointe.

1.1.1.2. Les solutions architecturales

Nous laissons de côté les machines à cadencement par les données (Data Flow Machines) ou à réduction[61] qui n'ont pas encore fait leurs preuves en matière de calcul scientifique, pour ne nous occuper que des modifications apportées au modèle de Von Neumann. La plupart de ces modifications consistent en une multiplication d'un ou de plusieurs des trois composants de base: l'unité de contrôle, l'unité arithmétique et logique et la mémoire. L'architecte doit ensuite **interconnecter** les nombreuses unités qui résultent de cette multiplication sans créer un nouveau goulot d'étranglement.

Volume de la mémoire

Malgré les progrès de l'intégration, une mémoire virtuelle de capacité encore supérieure est souvent proposée pour pouvoir traiter des volumes de données qui s'expriment en milliards d'octets dès que l'on veut traiter des problèmes à 3 dimensions. L'intérêt de la mémoire virtuelle pour les calculateurs scientifiques n'est pas démontré car il vaut peut-être mieux laisser le programmeur gérer lui-même ses entrées-sorties efficacement plutôt que de lui laisser croire que la machine peut le faire. Les algorithmes de remplacement de pages, qui fonctionnent correctement dans un environnement de gestion, peuvent donner de très mauvais résultats sur du code scientifique. Le problème est connu depuis longtemps et il nécessite une adaptation des programmes qui a été étudiée[1] mais pas implémentée, à notre connaissance, dans des compilateurs commerciaux. Un réordonnancement des opérations peut conduire à une diminution (ou à une augmentation!) spectaculaire des temps d'exécution.

Vitesse de la mémoire

Trois solutions complémentaires sont utilisées sur la plupart des machines: augmentation du débit par le découpage de la mémoire en bancs, présence de nombreux registres vectoriels et diminution du nombre d'accès à la mémoire à l'aide d'une antémémoire (cache) ultra-rapide. Ces trois solutions entraînent des contraintes de programmation. Les performances d'une mémoire à plusieurs bancs dépendent largement de la succession des adresses accédées. Cela conduit à éviter certaines dimensions de tableaux pour ne pas provoquer de conflits de bancs, à modifier les schémas de stockage traditionnels, en colonnes ou en lignes, ou à échanger des boucles pour faire progresser les adresses de la manière la plus efficace possible.

Les registres et l'antémémoire posent un même problème: étant donné leur vitesse, leur prix est élevé et l'espace de stockage qu'ils offrent est limité. Il faut adapter les tailles des boucles et des tâches à cette capacité. De plus ils ne présentent un intérêt que si les données qui y sont stockées sont utilisées plusieurs fois. Pour y arriver, il faut choisir un bon ordonnancement des opérations.

Le fonctionnement d'un cache est aussi caractérisé par la politique de remplacement choisie. Quelle donnée déjà présente faut-il remettre en mémoire quand le cache est plein et qu'on veut y conserver une nouvelle donnée? Différentes solutions, à base de mémoire associative, sont possibles mais la plus simple, et sans doute la moins coûteuse et la plus rapide, est celle qui est utilisée dans l'Alliant FX/8. A chaque adresse en mémoire correspond une seule position du cache obtenue en négligeant les bits de poids fort de l'adresse physique.

Deux politiques de répercussion des écritures s'opposent: le *write-back* et le *write-through*. La seconde consiste à effectuer les écritures au fur et à mesure qu'elles se présentent ce qui permet de charger la mémoire régulièrement et de ne pas avoir à vider le cache lors d'un changement de contexte. La première consiste à ne faire l'écriture que quand la position occupée dans le cache est réclamée pour une autre donnée. La charge de la mémoire en écriture est réduite mais il faut brutalement vider le cache de temps à autre.

Le problème se complique encore quand il faut traduire les adresses logiques en adresses physiques. Faut-il faire fonctionner le cache avec les adresses logiques avant la traduction ou sur les adresses physiques après traduction?

Enfin, les antémémoires, comme les mécanismes de mémoire virtuelle, gèrent les donnnées présentes par *lignes*, par *blocs* ou par *pages* de mots machine plutôt que par mots pour avoir une capacité suffisante à un coût acceptable. Les défauts de cache entraînent une activité mémoire très supérieure à ce que croit souvent le programmeur. En cas de défaut de cache, il faut réécrire l'ensemble de la *ligne* si elle a été modifiée (cas des caches *write-back*) et, dans tous les cas, charger toute la nouvelle ligne avant de pouvoir reprendre l'exécution des calculs. Une mauvaise utilisation du cache peut donc avoir des conséquences catastrophiques sur les performances globales.

La multiplicité des paramètres à prendre en compte rend difficile l'analyse et la prévision exacte des performances à attendre d'une machine comportant un cache. Nous ne prendrons en compte dans la suite que la taille du cache pour éviter les débordements.

Débit des mémoires magnétiques

Quand le volume des données est aussi important que dans les calculs scientifiques, le temps d'entrée-sortie n'est plus uniquement fixé par le temps d'accès. Le débit du disque importe, surtout quand les fichiers sont stockés de manière contiguë et que des pistes entières sont transférées. La plupart des systèmes d'exploitation des machines de calcul scientifique répartissent (*File Spreading*) donc les fichiers sur plusieurs disques.

Vitesse de calcul

Partant de processeurs dont la vitesse est fixée par la technologie, l'augmentation de la vitesse de calcul globale ne peut venir que de leur segmentation ou de leur multiplication. La multiplication peut s'effectuer de différentes manières: multiplication des unités fonctionnelles dans l'unité arithmétique et logique (CDC 6600), multiplication des unités arithmétiques et logiques toujours sous le contrôle d'une unique unité de contrôle capable d'envoyer simultanément un ordre différent à chaque unité arithmétique et logique (VLIW[19]), multiplication des couples formés de l'unité arithmétique et logique et de la mémoire avec une unique unité de contrôle envoyant la même instruction à chaque unité arithmétique et logique (SIMD[20]) et multiplication des processeurs (UAL+UC+M) eux-mêmes (MIMD[20]). Trois catégories sur cinq n'apparaissent pas dans la classification de Flynn.

1) Les unités arithmétiques pipelinées sont la base des machines vectorielles, machines pour lesquelles un important travail de recherche a conduit à des compilateurs commerciaux aux performances acceptables. Le taux d'accélération peut être très fort quand le programme est presque exclusivement vectoriel. Mais il décroît très fortement quand la proportion de code scalaire augmente (Amdahl's Law[7]). Il n'est pas possible de donner le gain moyen qu'on peut attendre de ces architectures car il dépend simultanément du rapport des vitesses des processeurs scalaires et vectoriels et de la proportion de code scalaire, et donc des programmes exécutés.

2) La multiplication des unités fonctionnelles est couramment utilisée au moins au niveau des fonctions: l'additionneur ne partage pas de circuiterie avec le multiplieur. Cette idée a été utilisée depuis longtemps puisque les CDC 3600, 6600 et 7600, les IBM STRETCH, 7094 et 370/165, le Cray X-MP, de nombreux *array*-processeurs et la plupart des nouveaux minisupercalculateurs (Alliant FX/8, Convex C1, SCS 40) en ont bénéficié. Les gains que cette technique peut apporter ont été étudiés par de nombreux auteurs. Foster et Riseman[21] ont trouvé un gain moyen de 1.72 avec un échantillon de 7 programmes pour un CDC 3600 ayant un nombre infini d'unités et de registres. Tjaden et Flynn[60] arrivent à un gain moyen de 1.86 avec 3 programmes de l'ACM pour l'IBM 7094. Avec le CDC 6600 les gains sont aussi autour de 1.7 mais montent à 2.43 sur un exemple. Kuck conclut dans[45] que le gain doit se trouver entre 1.5 et 2.5. Veidenbaum[66] propose aussi 1.5 et Ellis[17] ne trouve des gains plus forts qu'en déroulant massivement les boucles parallèles des programmes (jusqu'à 32 fois).

3) Ces gains plus forts sont effectifs avec les machines de type VLIW où de très nombreuses unités fonctionnelles sont disponibles. La présence de plusieurs multiplieurs ou de plusieurs additionneurs est difficile à exploiter efficacement mais une nouvelle technique de compilation appelée *trace scheduling*[17] a été récemment développée pour résoudre ce problème. Par rapport aux machines vectorielles, le progrès est sensible quand les programmes contiennent des boucles parallèles non vectorielles, par exemple quand l'adressage est non-linéaire. C'est le cas de la FFT.

4) La classe des machines SIMD, qui, notons le au passage, résoud aussi le problème de la bande passante mémoire, est large, mais, d'une manière générale, ce type de machine impose des programmes si spéciaux que personne n'a encore réussi à y adapter des programmes automatiquement. En effet, les unités arithmétiques et logiques n'accèdent qu'au banc mémoire qui leur correspond exactement, ou bien selon une permutation qu'autorise le réseau d'interconnexion entre les UALs et les bancs mémoire. Les échanges de données entre processeurs sont donc difficiles à coder (tri, transposition de matrice, FFT). Sur des cas particuliers, comme le traitement d'image, ce type d'architecture peut être extrêmement efficace.

5) Il ne nous reste plus à étudier que le cas des machines MIMD où plusieurs processeurs exécutent simultanément des portions de code indépendantes, se synchronisent de temps à autre et échangent des données via un mécanisme de communication. La multiplication du nombre de processeurs aggrave encore le problème du débit mémoire si tous les accès ont lieu dans une mémoire globale et les solutions évoquées précédemment, registre, antémémoire et mémoire à bancs, sont aussi utilisées dans un contexte multiprocesseur au prix d'un réseau d'interconnexion dont la complexité et le prix augmentent plus que linéairement (en $O(n^2)$ ou en $O(n \log n)$) avec le nombre de processeurs. Une autre solution consiste à diminuer le nombre d'accès à la mémoire globale en utilisant des mémoires locales.

6) La solution extrême consiste à supprimer la mémoire globale et à implémenter un mécanisme de communication et de synchronisation par messages entre les processeurs. Dans tous les cas, la présence de mémoires locales impose un travail supplémentaire au programmeur et/ou au compilateur: il faut décider de ce qu'on y met et s'arranger pour ne pas y mettre trop de choses car les mémoires locales sont plus petites que la mémoire globale. L'absence totale de mémoire globale pose aussi le problème du stockage des données globales. Il faut les partitionner, puis les répartir dans les mémoires locales, et ensuite gérer la transmission de ces données entre processeurs. La présence de certaines données sur certains processeurs conduit à essayer d'y exécuter certains processus. La suppression de la mémoire globale conduit donc à un problème d'affectation des processus aux processeurs.

Cette dernière catégorie de machines requiert du parallélisme de haut niveau (*Coarse Grain Parallelism*) alors que les précédentes exploitaient le parallélisme élémentaire (*Fine Grain Parallelism*). Ce parallélisme de haut niveau est beaucoup plus difficile à découvrir dans un programme séquentiel. La recherche de boucles parallèles contenant des appels de procédures est une première possibilité[63] . Le partitionnement des boucles imbriquées que nous proposons en est une autre.

Réseau d'interconnexion

Dans de nombreuses architectures, le réseau d'interconnexion, qu'il soit à liens fixes ou variables, constitue le goulot d'étranglement à la fois pour la vitesse pure et pour la validité économique de la machine car le produit du temps de traversée du réseau et du nombre de ses composants augmente plus vite que le nombre de processeurs. Le problème a déjà reçu une attention considérable[2] et nous n'avons pas l'intention de lui apporter une solution par le logiciel. Nous supposerons qu'il a été résolu d'une manière satisfaisante dans la machine pour laquelle nous devons optimiser un programme.

Moyens de synchronisation

Suivant les architectures, les besoins de synchronisation et de communication sont plus ou moins importants. Ils sont généralement assurés par du matériel mais le programmeur ne peut parfois y accéder que par le biais d'une bibliothèque spécialisée et du système d'exploitation. La sécurité du système et l'exécution simultanée de tâches appartenant à des usagers différents nécessitent cette interface et conduisent à des overheads importants, que l'usager s'efforcera d'atténuer en augmentant la taille de ces tâches ou plus simplement en n'exploitant pas le parallélisme de la machine. D'après le constructeur, 95% des programmes s'exécutant sur Cray X-MP/4 n'utilisent qu'un processeur.

Conclusion

Nous allons maintenant voir comment ces différentes solutions partielles ont été intégrées dans des machines MIMD existantes, en nous intéressant principalement au problème de l'organisation de la mémoire.

1.1.1.3. Multiprocesseurs à mémoire globale

Le cas typique est celui du Cray X-MP/48. Quatre processeurs se partagent une unique mémoire globale dont le débit est à peu près suffisant pour les alimenter. En fait, les conflits mémoires provoquent de 20 à 30 % de baisse de performance[53, 36] . Les synchronisations de tâches se font par appel à une bibliothèque de modules spécialisés[48] , via le système d'exploitation. Ce sont des opérations coûteuses.

L'Alliant FX-8[27] est aussi un multiprocesseur à mémoire globale, mais contrairement à la machine précédente, il possède un cache global de 128 Ko relié aux processeurs élémentaires par un réseau crossbar dont la bande passante est deux fois supérieure à celle de la mémoire. Le débit de la mémoire est insuffisant pour alimenter les 8 processeurs, qui sont tous identiques et vectoriels (8 $\times$ 11.8 Mflops en pointe sur 64 bits), et les données doivent nécessairement se trouver en cache et en registre pour que les unités fonctionnelles atteignent leur vitesse nominale. Cette machine dispose de plus d'une mémoire virtuelle, elle aussi susceptible d'être le goulot d'étranglement. Enfin, de puissantes primitives de synchronisation câblées permettent aux processeurs élémentaires de s'attribuer très rapidement une nouvelle tâche à effectuer.

1.1.1.4. Multiprocesseurs à mémoire locale et globale

Le nouveau modèle de Cray Research, le Cray 2, possède une mémoire locale. La mémoire globale, qui peut atteindre une capacité de 256 MW, fournit un mot de 64 bits par cycle de 4 ns à chacun des quatre processeurs, en l'absence de conflits. Cette mémoire est réalisée avec de la mémoire dynamique NMOS dont le temps d'accès de 220 ns implique un temps de lancement de 55 cycles pour les chargements vectoriels et une attente équivalente pour les accès scalaires. Les mémoires locales de chaque processeur, dont la capacité est limitée à 16 KW, ont le même débit de un mot par cycle mais le temps d'accès est limité à quelques cycles. Les unités fonctionnelles sont uniquement reliées aux huit registres par un crossbar qui permet leur fonctionnement simultané. Pour atteindre un temps de cycle aussi faible et pour diminuer la taille des cartes, l'unité de contrôle a été simplifiée et le chaînage des pipes n'est pas possible. Cela implique un important travail d'ordonnancement à la génération de code plus proche de ce qui se fait en microprogrammation ou en programmation d'array-processeurs bas de gamme que dans les compilateurs traditionnels. Les algorithmes d'ordonnancement travaillent classiquement sur des corps de boucles déroulées et permettent d'atteindre des performances supérieures au Gigaflops.

Le projet NYU[25, 26] , qui est surtout connu à cause de son réseau d'interconnexion opératif *fetch-and-add*, a été conçu avec une mémoire globale. Quoique l'objectif soit de connecter 512 processeurs et d'atteindre des puissances considérables, le prototype ne compte pour le moment que 8 processeurs à base de 68010. Ce projet doit surtout permettre de valider le concept de réseau d'interconnexion opératif, concept qui a été repris par IBM pour son projet RP3. Théoriquement, ce réseau doit permettre aux processeurs élémentaires de gérer collectivement et rapidement des files d'attente, et donc de s'allouer très rapidement des tâches à exécuter. Pour ne pas ralentir excessivement les 512 processeurs élémentaires à cause du temps de latence du réseau, il est prévu de doter chaque processeur d'un cache local. La solution mémoire locale est écartée au profit du cache pour simplifier l'écriture du logiciel.

Le projet de recherche RP3[11] (Research Parallel Processor Project) d'IBM a été conçu pour permettre de tester différentes architectures possibles. Chaque processeur élémentaire dispose d'un cache de 32 Ko et d'une mémoire de 4 Mo qui peut être découpée à volonté entre mémoire locale et mémoire globale par une frontière ajustable gérée par le contrôleur mémoire. On peut donc faire varier continûment la configuration d'une machine sans mémoire globale à une machine sans mémoire locale. De plus, un mécanisme de mémoire virtuelle permet d'offrir une capacité d'adressage suffisante pour n'importe quel programme scientifique. Chaque processeur est vectoriel mais les performances sont limitées: 0.9 Mflops en pointe. Le temps de *start-up* est faible puisqu'on obtient 90 % des performances avec un vecteur de 20 éléments. L'unité vectorielle permet aussi d'effectuer des transferts de données entre les niveaux de la hiérarchie mémoire. Enfin, les processeurs et les mémoires, dont le nombre doit être de 64 dans la première étape et monter à 512 ensuite, sont interconnectés par un réseau opératif emprunté au projet NYU, qui doit permettre de réaliser efficacement les synchronisations.

Le projet CEDAR[22] , pour lequel le Centre de Recherche et de Développement des Superordinateurs de l'Université d'Illinois à Urbana-Champaign (Center for Supercomputing Research and Development at UIUC) doit recevoir environ 30 millions de dollars étalés sur 5 ans, a pour but à construire une machine multiprocesseur à mémoire globale. Chaque processeur élémentaire est lui-même un multiprocesseur vectoriel commercial, un Alliant FX-8, dont le contrôleur mémoire doit être modifié pour mettre la mémoire globale dans l'espace d'adressage des processeurs élémentaires. De nouvelles primitives de synchronisation, exécutées à l'aide d'un processeur spécialisé intégré à la mémoire, ont été développées[70] pour gérer ce multimultiprocesseur vectoriel, qui possède trois niveaux de parallélisme et quatre niveaux de hiérarchie mémoire.

La compagnie IBM commercialise aussi une machine de ce type, le 3090 VF[65] . Cette machine peut comporter de un à quatre processeurs, comme le Cray X-MP. Ces processeurs partagent une mémoire globale pouvant monter théoriquement à 128 Mo mais disposent aussi de caches locaux de 64 Ko et d'une autre mémoire limitée à 256 Mo (l'architecture permettrait de monter à 16 To) qui sert de cache disque pour la mémoire virtuelle (*Expanded Storage ou Mémoire d'arrière-plan*). L'espace virtuel d'un processus est de 2 Go. L'utilité d'un cache pour une machine vectorielle a été longtemps contestée à cause du manque de localité de référence auquel conduit une programmation orientée vecteur. Ce choix et ses conséquences sont largement défendus et expliqués dans [15]. La puissance de pointe théorique d'un processeur vectoriel est de 108 MFlops, mais le gain réel du processeur vectoriel par rapport au processeur scalaire n'est que de 3 à 4, ce qui conduit à des vitesses d'une vingtaine de MFlops. La synchronisation des processeurs se fait par l'intermédiaire du système d'exploitation, ce qui doit conduire à définir des tâches assez grosses. Aucun outil automatique n'aide l'utilisateur à programmer *en parallèle* mais le compilateur Fortran[57] génère du code vectoriel. Ce compilateur a été développé avec l'aide de l'équipe du projet PFC[3] de Rice University.

En plus de RP3, IBM a entrepris un deuxième projet de recherche sur les architectures parallèles, le projet LCAP (*Loosely Coupled Array of Processors*). Il s'agit d'interconnecter une machine haut de gamme, tournant sous un système d'exploitation classique IBM, et plusieurs array-processors. Trois étapes doivent se succéder:

- LCAP1: un 3081 sous VM et 10 FPS 164 avec l'option FPS Max, plus un 4381 en frontal; la puissance attendue est d'environ 500 Mflops;
- LCAP2: identique à LCAP1 mais le 3081 doit être exploité sous MVS;
- LCAP3: le 3081 est remplacé par un 3090 et la puissance attendue passe à 1.33 Gflops.

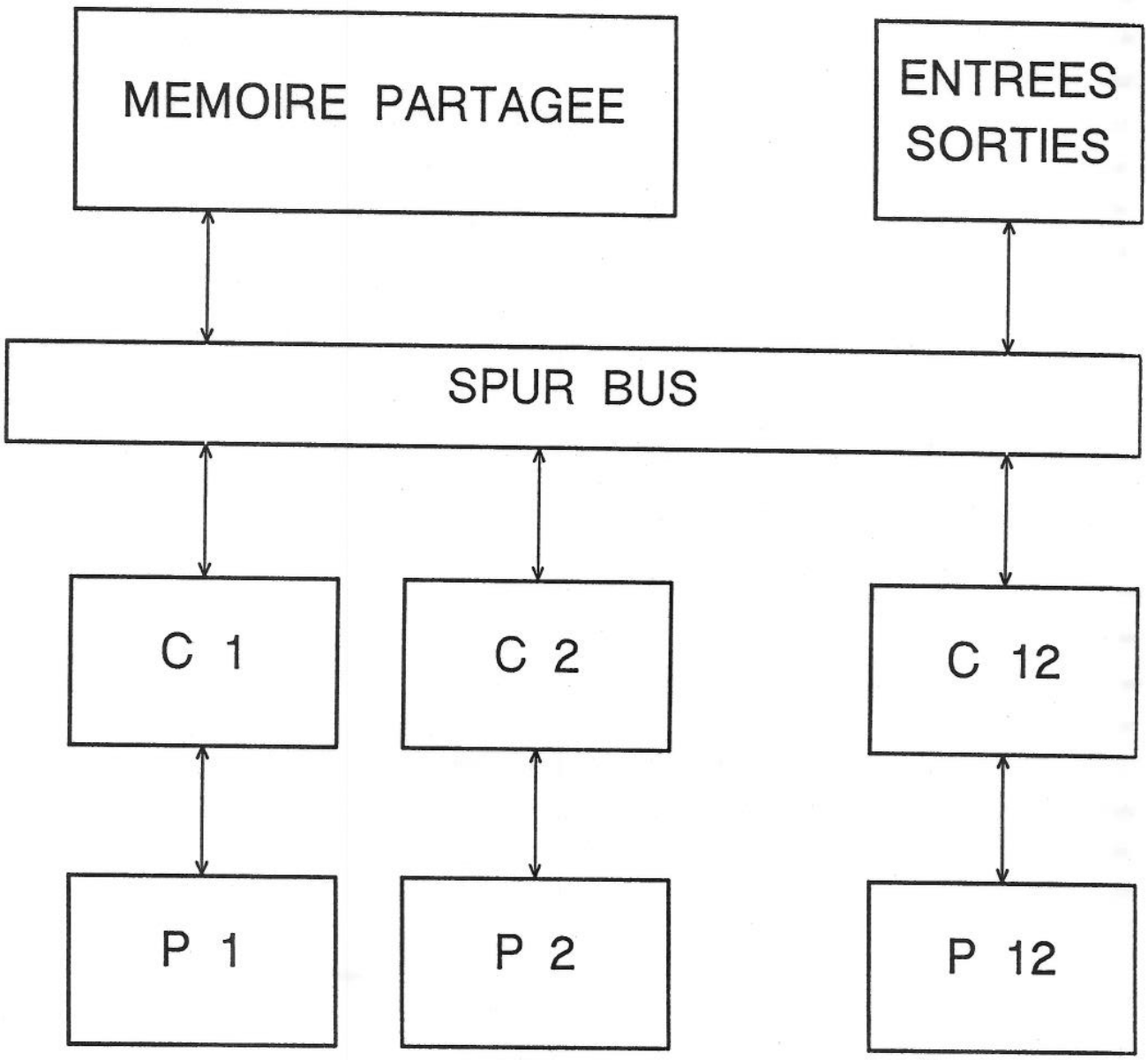
Cette architecture, qu'on retrouve dans la configuration Multi AP de l'ONERA et dans le projet MARIANNE de la CIMSA-SINTRA, se distingue des précédentes car, s'il existe toujours une mémoire globale, cette dernière n'est plus accessible directement par les processeurs élémentaires. Il faut d'abord transférer les données de la mémoire globale, ici la mémoire virtuelle du 3081, à la mémoire locale du FPS 164,

transférer les programmes et après exécution recopier les résultats sur le 3081. Il est donc nécessaire de prévoir un important effort de programmation pour adapter un programme à ce type d'architecture. De plus, le temps de lancement d'une tâche et, dans une moindre mesure, le temps d'exécution des primitives de synchronisation sont assez élevés parce que tout se fait par logiciel. Cela conduit à rechercher du parallélisme à *forte granularité*.

La nouvelle machine de ETA, la filiale de CDC spécialisée dans les superordinateurs, dispose aussi de plusieurs niveaux dans sa hiérarchie mémoire. Les processeurs élémentaires de l'ETA 10^{69} reprennent le jeu d'instructions du Cyber 205. Ils n'ont pas de cache et peu de registres (256 mots) mais une importante mémoire locale (4 MW = 32 Mo). La mémoire globale, qui peut monter à 256 MW (2 Go) semble jouer le rôle de la SSD de Cray. En effet, elle sert à conserver les pages qui ne tiennent plus en mémoire locale et à copier les fichiers dans l'espace d'adressage des processeurs. La taille des pages peut varier dans de larges limites (1024, 2048, 8192, 65536 ou 262144 octets). Les transferts de données entre mémoire globale et mémoires locales semblent gérés par le mécanisme de mémoire virtuelle. Les 8 processeurs élémentaires partagent aussi un *buffer* collectif destiné à assurer des synchronisations rapides. De 1 à 18 unités d'entrée-sortie autonomes assurent les transferts entre les disques et la mémoire globale.

L'Université de Californie à Berkeley développe un projet de superstation de travail pouvant compter de 6 à 12 processeurs élémentaires. Cette machine, appelée SPUR[33] (Symbolic Processing Using RISCs) devra pouvoir aussi bien exécuter du code scientifique que des applications écrites en Common Lisp, mais le traitement symbolique est l'objectif premier. Chaque processeur a un jeu d'instructions de type RISC augmenté des opérations flottantes les plus courantes (elles sont exécutées par un coprocesseur) et une puissance comparable à un VAX 8600 ou à une Symbolics 3600, mais l'architecture globale est similaire à celles du Sequent Balance 8000 et du Elxsi 6400 (cf. figure 1.1.1.4). Chaque processeur élémentaire dispose d'un cache instruction de 512 octets intégré dans la même puce et d'un cache local de 128 Ko, organisé en blocs de 32 octets, dont la cohérence est assurée par le matériel. Il peut accéder à la mémoire globale par un bus unique en une quinzaine de cycles, contre un pour le cache. Ce temps est acceptable, d'après les concepteurs de SPUR, parce que plus de 98 % des accès mémoires peuvent être satisfaits par le cache, dans le cas de leurs programmes étalons. Nous pensons que le comportement du cache sera moins satisfaisant avec des programmes de calcul scientifique. Le temps de cycle projeté et utilisé dans les simulations est de 100 à 150 ns, ce qui conduit à des performances de 1 à 2 MFlops suivant la proportion d'additions, de soustractions, de multiplications et de divisions qui sont effectuées.

Enfin, la société BBN propose le Butterfly qui peut compter jusqu'à 128 processeurs. Chaque processeur est constitué d'un 68000, d'un 68881 et de 4 Mo de mémoire (maximum) et coûte de 6000 à 9000 dollars. Sa puissance est de 0.5 Mips, ce qui donnne environ 60 Mips pour 128 processeurs. Les mémoires locales sont accessibles par tous les autres processeurs via un réseau à commutation de paquets dont la bande passante croît avec le nombre de processeurs.



Le projet SPUR

- Figure 1.1.1.4 -

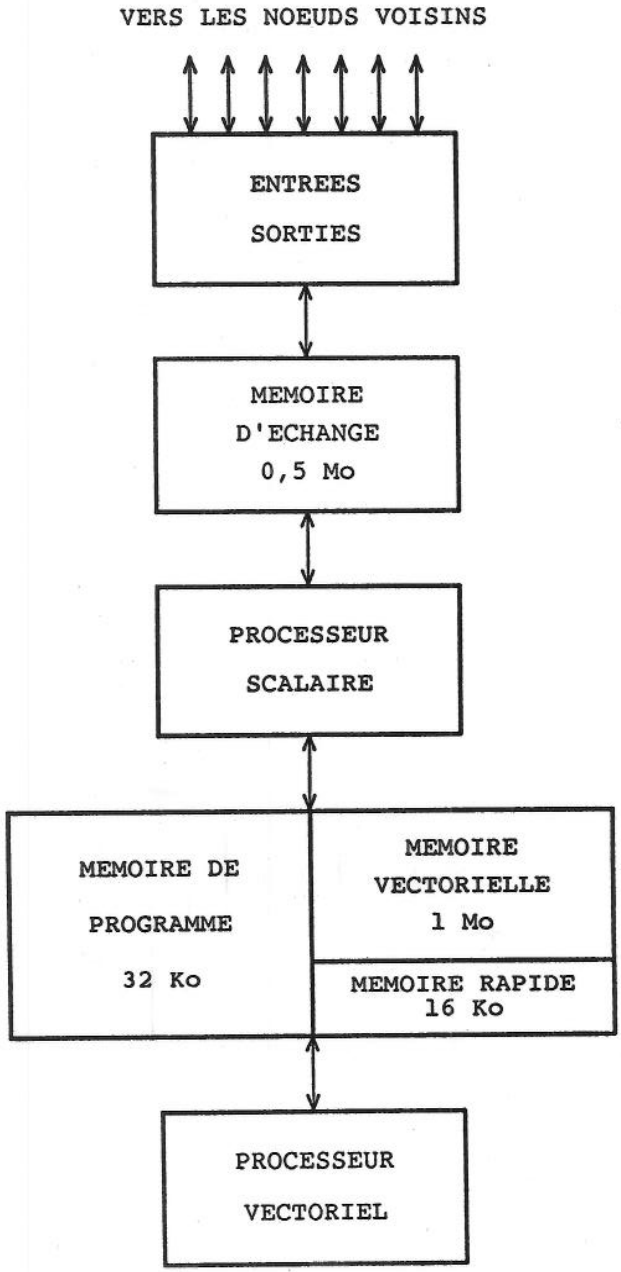
1.1.1.5. Multiprocesseurs à mémoire locale

Suite au succès du projet Cosmic Cube[59] du CalTech, les architectures sans mémoire globale ont commencé à se répandre, spécialement avec un réseau de connexion hypercubique. Après avoir acheté la licence du CalTech, la société Intel a développé un produit commercial, l'iPSC (Intel Personal Supercomputer), en plusieurs versions comptant 32, 64 ou 128 noeuds. Chaque noeud contient de 512 Ko à 4.5 Mo et le processeur de calcul flottant est un 80287. Cette première version a été assez décevante car les performances en calcul flottant d'un processeur élémentaire étaient tout à fait insuffisantes (20 KFlops!). Dans le meilleur des cas, on pouvait obtenir 1 MFlops avec 128 processeurs. Ce défaut a été corrigé dans la dernière version[38] où chaque processeur élémentaire se voit adjoindre un coprocesseur vectoriel à base de circuits intégrés Weitek (iPSC-VX) et atteint une vitesse de pointe de 6.67 MFlops. Pour les opérations de base, les vitesses effectives vont de 0.28 à 4.4 MFlops pour des opérandes de 64 bits résidant dans la partie "rapide" de la mémoire vectorielle. Ce coprocesseur n'est malheureusement pas intégré au processeur principal (80286) dont il ne partage même pas l'espace d'adressage. Dans ces conditions, il est difficile de comprendre comment le préprocesseur vectoriseur VXP, qui est censé transformer les séquences de code parallèle en appels à des routines d'une librairie résidant en PROM, pourrait produire un code efficace. L'usager est fortement encouragé à programmer en utilisant directement les modules de cette bibliothèque.

L'espace d'adressage limité du coprocesseur vectoriel nécessite la copie des données pour pouvoir les traiter vectoriellement, puis la recopie des résultats pour pouvoir les échanger avec les processeurs voisins (voir figure 1.1.1.5). D'autre part, le *start-up* d'un envoi de message entre deux processeurs est très long (quelques millisecondes: 4.5 à 6 ms suivant les sources), même si le débit point à point n'est pas négligeable (de 250 Ko/s à 700 Ko/s suivant les sources). Enfin, la synchronisation des processeurs peut s'effectuer en recourant directement au processeur hôte via une liaison Ethernet. Comme dans toutes les architectures sans mémoire globale, le code doit être dupliqué sur chaque processeur, ce qui ne laissait souvent pas beaucoup de place pour les données dans la première version. L'absence de mémoire globale entraînait aussi une perte de temps considérable à l'initialisation pour diffuser le code et les données initiales dans chaque noeud et à la fin pour regrouper les résultats sur le frontal, un microordinateur Intel. Ceci dit, la société Intel a tout de même eu le mérite de prendre le risque de proposer le premier multiprocesseur commercial de type hypercube.

D'autres sociétés proposent des produits basés sur l'architecture hypercube. Ametek propose le System 14 qui est un hypercube, comptant de 16 à 256 processeurs élémentaires (80286 + 80287) et atteignant des performances de 0.8 à 12 MFlops, sur lequel nous n'avons pas trouvé d'informations précises, Ncube, le NCUBE/ten, qui en compte de 64 à 1024, et Floating Point Systems, la série T.

Le multiprocesseur NCUBE/ten[31, 32] est basé sur un microprocesseur conçu directement pour être utilisé dans une architecture hypercube. Il intègre un cache instruction de 32 octets et toutes les fonctionnalités nécessaires pour communiquer avec les noeuds voisins (11 canaux bidirectionnels) ou pour



Un noeud de l'iPSC-VX

- Figure 1.1.1.5 -

gérer la mémoire dynamique locale. Cela permet de réaliser un noeud disposant de 128 Ko avec seulement 7 boîtiers car les boîtiers mémoire dynamique sont organisés en 64K $\times$ 4, mais les échanges se font par demi-mots de 16 bits. Cette machine étant destinée au calcul scientifique, le jeu d'instructions est inspiré de celui du VAX et comprend les instructions flottantes. Les 0.5 MFlops (sur 32 bits, 0.3 MFlops sur 64 bits) de chaque processeur lui donnent une puissance équivalente à celle d'un VAX 11/780 et permettent d'afficher une vitesse de pointe de 500 MFlops pour la configuration maximale de 1024 processeurs. Le onzième canal permet à chaque noeud de communiquer directement avec un processeur d'entrée-sortie, qui gère 128 canaux simultanément. La machine est livrée avec un dérivé d'UNIX comme système d'exploitation, étendu pour gérer des cubes de processeurs par des primitives comme WHOAMI qui permet au programme de savoir sur quel processeur il s'exécute ou comme NREAD et NWRITE qui permettent d'échanger des données entre noeuds voisins ou entre un noeud et la machine hôte.

La série T de Floating Point Systems[29] , qui est basée sur le Transputer d'INMOS et sur des chips Weitek, comprend des modèles allant de 16 noeuds à 16384 noeuds. La configuration la plus importante atteint une vitesse théorique de 262 GFlops et nécessite un immeuble entier pour abriter la machine. Chaque noeud contient 1 Mo de mémoire organisée en 2 bancs dissymétriques dont la bande passante en mode vectoriel contigu atteint 2560 Mo/s pour le chargement et le déchargement des registres grâce à l'utilisation de circuit video RAM. Le débit entre les registres et l'unité vectorielle est de 3 $\times$ 64 Mo/s, deux lectures et une écriture. L'unité vectorielle a une vitesse de pointe de 16 MFlops. Chacun des canaux du Transputer offre un débit de 0.5 Mo/s et leur temps d'initialisation n'est que de 4 μs . Pour que les transferts entre noeuds soient masqués par les calculs, il faut effectuer 100 opérations sur chaque mot de 64 bits...

D'autres types de réseau d'interconnexion ont été proposés, comme les grilles ou les arbres, mais ils ne permettent pas de simuler un aussi grand nombre de topologies que l'hypercube. Le multiprocesseur[23] Cm* en est un exemple. Les processeurs sont reliés d'une manière hiérarchique, d'abord en groupe (*clusters*), puis en ensemble de groupes. Les échanges entre processeurs se font via les mémoires grâce à des contrôleurs mémoire intelligents qui gèrent les communications et augmentent l'espace d'adressage des processeurs (LSI 11, 64 Ko). Suivant la programmation des contrôleurs, la machine peut fonctionner en mode mémoire partagée ou communication par messages. Les processeurs élémentaires lisent ou écrivent leur mémoire locale en 3 μs, la mémoire d'un de leurs compagnons de cluster en 8.6 μs et la mémoire d'un processeur d'un autre cluster en 35.2 μs. Le message le plus court est transmis en 85 μs. Cette machine est d'une conception assez ancienne et ne supporte pas les calculs flottants. Les expériences qui ont été menées pour en tester l'architecture ont donc été limitées à des calculs en nombres entiers[9, 56] . Il ne semble pas prévu à l'heure actuelle de remettre à niveau les processeurs élémentaires mais les travaux sur l'écriture de programmes parallèles et le problème du mapping des tâches sur les processeurs [58] continuent.

1.1.1.6. Modélisation

La description précise de chaque architecture contient beaucoup trop d'informations pour pouvoir être utilisée comme base d'optimisation de programmes. Il faut ne retenir que les caractéristiques qui conditionnent le plus les performances et donc modéliser les machines.

Le nombre de processeurs élémentaires (appelés processeurs par la suite), supposés tous identiques, et la présence d'unités vectorielles restent des éléments importants, mais comme nous l'avons montré en 1.1.1.1, les mouvements de données conditionnent de plus en plus les performances. Nous nous intéresserons donc à l'organisation hiérarchique de la mémoire mais en ne considérant que **deux** niveaux au plus tout au long de ce travail. Nous espérons pouvoir montrer que l'application itérative de notre méthode résoud aussi le problème à N niveaux. La puissance de calcul de chaque processeur est un facteur important, au moins relativement à la vitesse de transfert.

Chacun de ces deux niveaux, le niveau lent et le niveau rapide (mémoire locale ou cache), peut être distribué sur chaque processeur comme dans les architectures hypercubiques ou bien global comme sur l'Alliant. Les tailles des différentes mémoires sont connues.

Les échanges entre les mémoires peuvent être gérés automatiquement, par un mécanisme d'antémémoire ou de mémoire virtuelle (mécanisme qui n'est pas représenté sur les figures), ou programmés et exécutés soit par les processeurs élémentaires (P) soit par des processeurs d'échange (E). Dans le cas où il y a des processeurs spécialisés, les processeurs élémentaires ne peuvent généralement pas avoir accès à la mémoire la plus lente. Le débit et le temps de démarrage des transferts entre les deux niveaux sont intéressants.

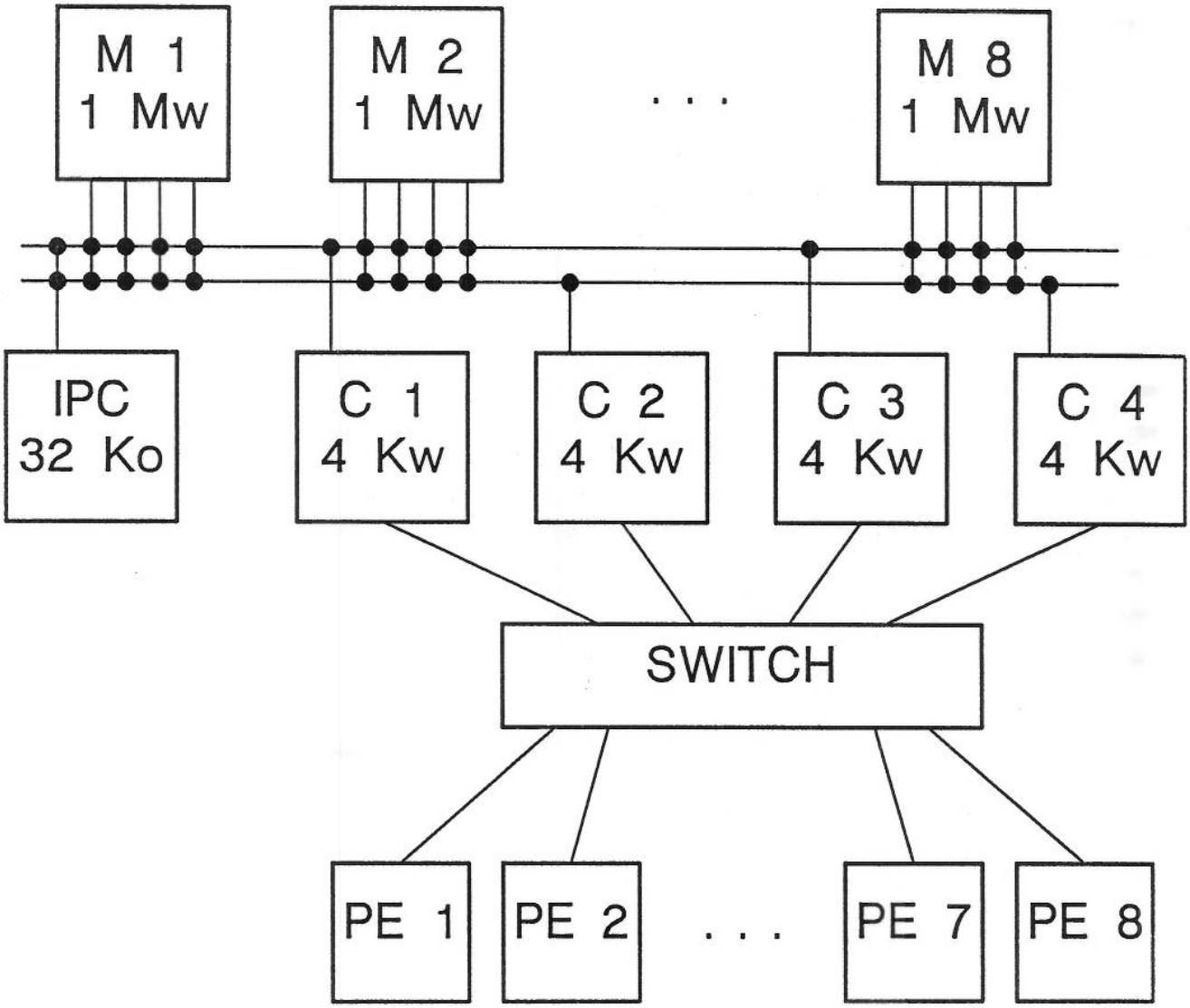
Le réseau d'interconnexion des processeurs élémentaires et des mémoires locales ou globales n'est pas pris en compte car le problème du *mapping* des tâches sur les processeurs ne sera pas étudié. Nous faisons l'hypothèse que le réseau est symétrique vis-à-vis de tous les processeurs et de toutes les mémoires et que les concepteurs de la machine lui ont donné une bande passante suffisante pour que les performances d'un processeur ne soient pas perturbées par l'activité d'un autre.

Les moyens de synchronisation disponibles sur la machine peuvent être directement câblés ou accessibles via le système d'exploitation. Leur nature ne nous intéresse pas. Seul compte le temps d'exécution des primitives que nous allons utiliser. Cet aspect de la machine sera donc caractérisé par un simple coefficient, relatif à la puissance de calcul et au débit des transferts.

1.1.1.7. Exemples de modélisation

Nous allons voir sur quelques exemples l'effet simplificateur de la modélisation. Tout d'abord, l'Alliant FX-8 (figure 1.1.1.7.a) avec ses quatre caches globaux (C1 à C4), ses huit bancs mémoires (M1 à M8), ses huit processeurs élémentaires (PE1 à PE8), ses deux bus mémoire et ses processeurs d'entrée-sortie (IPC). La manière dont est obtenue la bande passante de la mémoire, des caches ou du réseau

d'interconnexion ne nous intéresse pas. Les bancs mémoire sont donc regroupés comme les caches partiels et le réseau cross-bar disparaît. Ceci conduit à la première modélisation (figure 1.1.1.7.b). Comme les tâches que nous allons générer par partitionnement sont toutes identiques, elles occuperont le même espace dans le cache global. Il est donc équivalent de le remplacer par 8 caches locaux d'une capacité égale à 1/8ème de la capacité originale.



Alliant FX/8

- Figure 1.1.1.7.a -

Le projet RP3 (figure 1.1.1.7.c) a été conçu pour tester différentes architectures. Plusieurs modélisations sont donc possibles, mais comme nous ne prenons en compte que deux niveaux de la hiérarchie mémoire il faut choisir entre le cache et la mémoire locale (figure 1.1.1.7.d). La gestion de la mémoire virtuelle, les processeurs d'entrée-sortie et de mesure de performances disparaissent. L'unité vectorielle est fondue avec le reste du processeur.

Le projet NYU est un sous-ensemble du projet RP3. Comme dans les autres cas, les composants d'interconnexion disparaissent. Seuls subsistent les caches que contiennent les interfaces processeur élémentaire / réseau d'interconnexion (figure 1.1.1.7.e).

L'IBM 3090-400 avec l'option VF (figure 1.1.1.7.f) a une structure assez compliquée, qui comporte cinq niveaux de hiérarchie mémoire si les registres et les disques sont pris en compte, mais qui est, elle aussi, ramenée au schéma général (figure 1.1.1.7.g.).

Enfin, l'ETA 10 (figure 1.1.1.7.h et 1.1.1.7.i) pose un problème. En effet, la mémoire globale est en fait une mémoire d'arrière-plan et les transferts se font donc par page. L'occupation de la mémoire locale par une partie de tableau n'est pas une fonction directe du nombre d'éléments. Le stockage d'une ligne ou une colonne peut nécessiter un espace de N pages dans un cas et de N mots dans un autre. Les transferts entre niveaux mémoire par un mécanisme de type mémoire virtuelle paginée nécessiteraient donc un traitement spécifique. Pour le moment, faute d'informations suffisamment précises sur l'ETA 10, nous faisons une approximation par des mémoires caches.

Le Cray 2 (figure 1.1.1.7.j) est directement représenté au niveau modélisation car c'est une machine assez simple quant aux principes mis en oeuvre. Il faut noter les mémoires locales qui ne peuvent être chargées que par l'intermédiaire des registres scalaires ou vectoriels du processeur. Le compilateur Fortran semble l'utiliser pour stocker des scalaires et, sans doute, quelques copies des registres vectoriels.

Les machines modélisées sont assez semblables, à part RP3 et le Cray 2. Il serait intéressant de voir ce que donneraient des machines possédant des opérateurs de transfert programmables, comme le démonstrateur de MARIANNE, pour lesquelles nous n'avons malheureusement pas trouvé de documentation.

1.1.1.8. Conclusion

A l'image de ce qui se passe dans les industries manufacturières, les machines de pointe nécessitent une meilleure organisation du travail. Le programmeur (et, à terme, le compilateur) doit gérer l'implantation des données dans les différentes mémoires, ordonnancer correctement les opérations élémentaires, partitionner ses calculs en tâches, organiser les communications et les synchronisations entre les processus et affecter les processus aux processeurs. Faute d'outils automatiques, il ne sera possible d'exploiter efficacement ce type de machines qu'avec des bibliothèques spécialisées dont les modules auront été adaptés manuellement, et à grand frais.

1.1.2. Décomposition de l'objectif général de minimisation du temps d'exécution

Le temps total d'exécution d'un programme est, approximativement, la longueur du chemin critique du graphe d'ordonnancement correspondant, graphe qui est construit sur les tâches de calcul, sur les tâches de transfert et sur les tâches de synchronisation. L'optimisation du temps d'exécution du programme passe donc par l'optimisation des tâches élémentaires et par leur mise en parallèle.

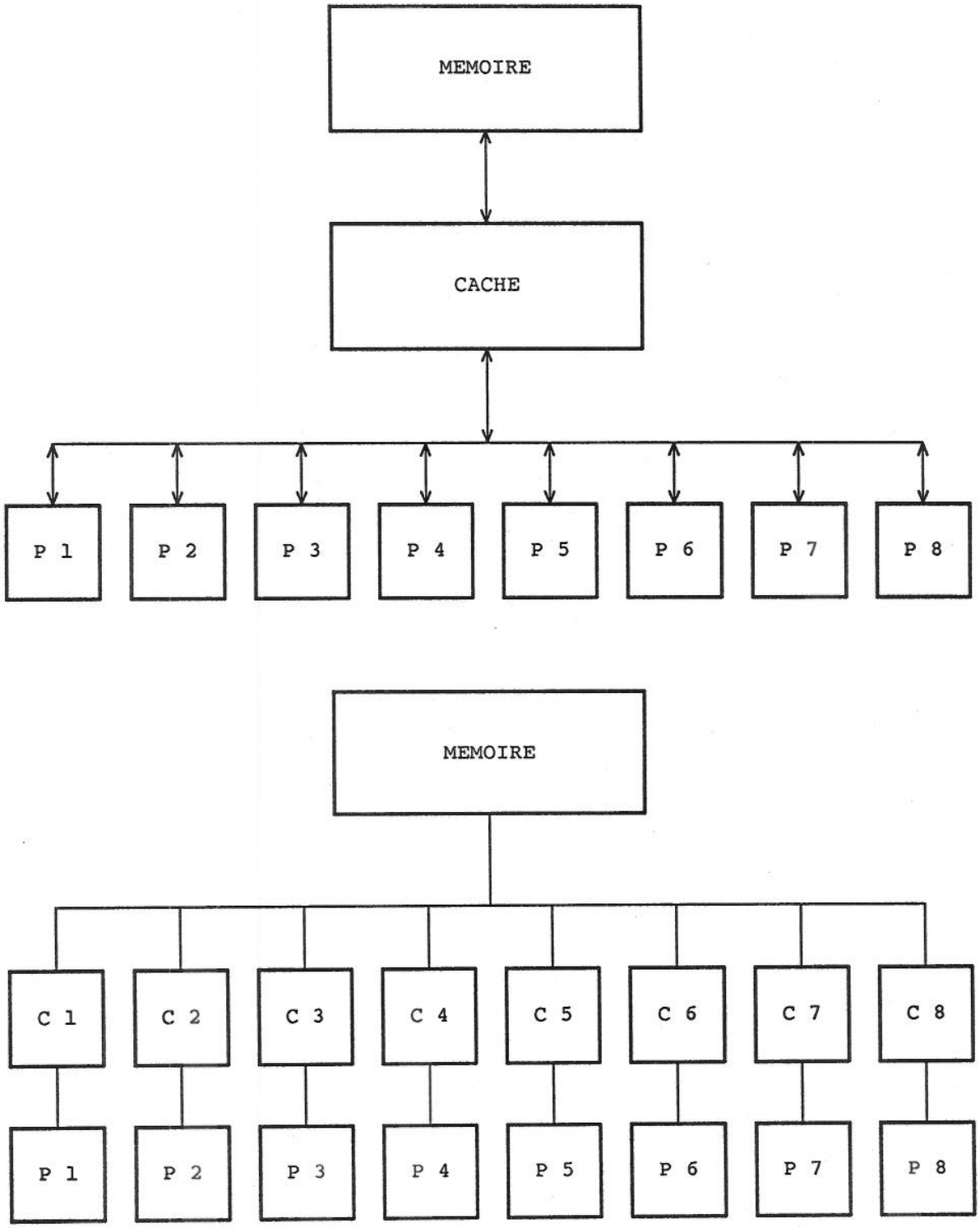
La réduction du temps de calcul est d'abord obtenue en utilisant les possibilités des processeurs élémentaires, en particulier leurs unités vectorielles. Il est inutile d'essayer d'optimiser le temps d'exécution sans commencer par le niveau vectoriel, qui apporte un gain de 3 à 10 suivant les machines, alors que les gains espérés pour d'autres optimisations sont de l'ordre de 50 %. En effet notre partitionnement repose sur les mêmes conditions que la méthode hyperplane et il sera donc souvent possible d'obtenir une exécution vectorielle comme base de comparaison pour évaluer l'amélioration due au partitionnement. Il faudra donc toujours commencer par exploiter cette possibilité, si les processeurs élémentaires sont vectoriels et si les instructions du corps de boucle peuvent se mettre sous forme vectorielle.

Le temps de calcul dépend aussi, dans un processeur ayant plusieurs niveaux de hiérarchie mémoire, de la localisation des données. Il faut essayer de garder les données nécessaires à un calcul dans le niveau le plus rapide, ou bien diminuer la taille du calcul pour que les données correspondantes puissent y tenir. Dans le cas des mémoires caches, le gain attendu est souvent de l'ordre de 30 %, mais il peut être plus fort du fait des gestions de cache par ligne[†] . Pour les mémoires virtuelles, le gain peut être très fort puisque dans le cas le plus défavorable chaque instruction, ou au moins chaque tour de boucle, provoque un défaut de page et donc un temps d'exécution de l'ordre d'une dizaine de millisecondes. Enfin, si les accès à la mémoire globale s'effectuent à travers un réseau d'interconnexion, son temps de latence va conditionner le gain. Il n'y a rien à faire quand il n'y a qu'un niveau de hiérarchie mémoire comme sur le Cray X-MP, quoique la bonne utilisation des registres soit un problème voisin.

Garder les données dans le niveau mémoire le plus rapide aussi longtemps que possible n'est pas un objectif facile à quantifier, mais on est sûr de l'avoir atteint si, en minimisant les transferts entre les deux niveaux, on ne transfère chaque donnée qu'une seule fois.

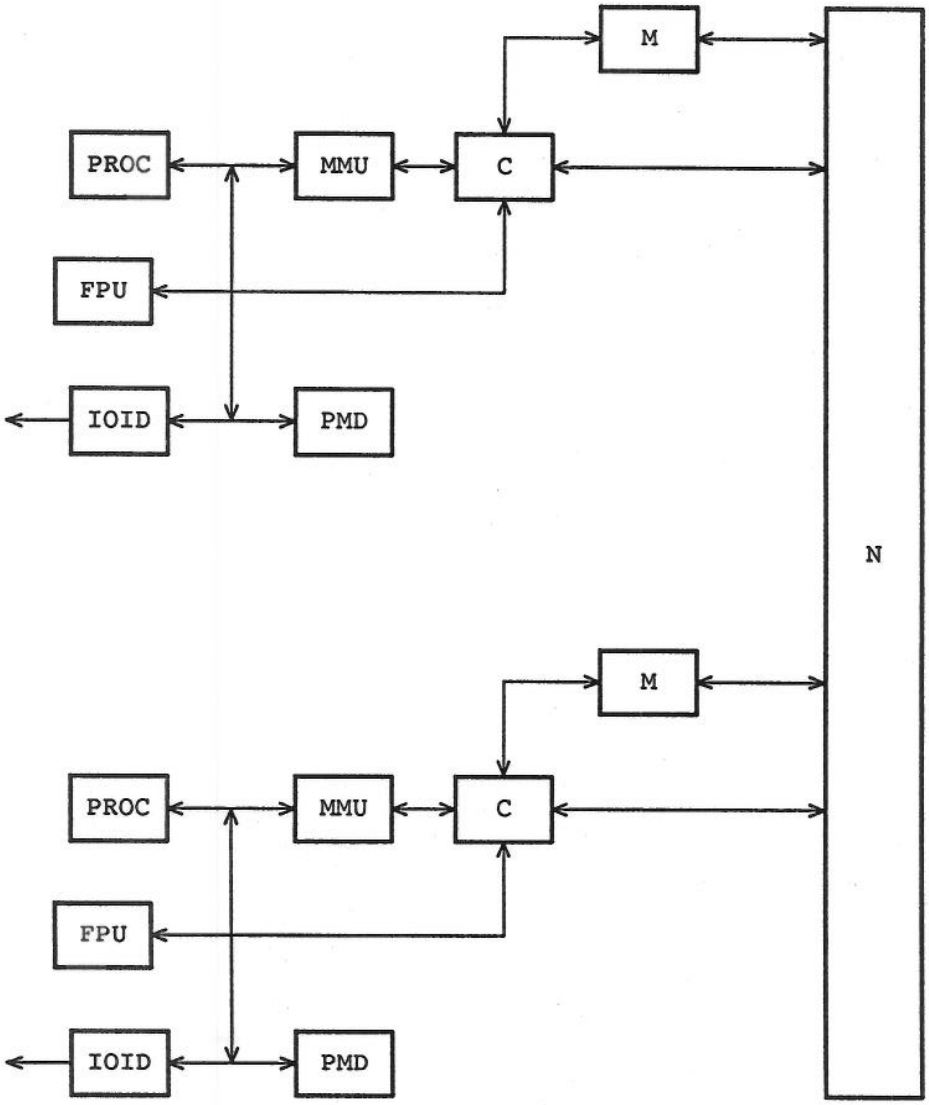
Il reste ensuite à paralléliser les tâches élémentaires. Pour les tâches de calcul, cela revient à équilibrer la charge entre les processeurs. On sait qu'une solution optimale est obtenue quand ils sont tous actifs en permanence. Cet objectif est d'autant plus facile à réaliser que les tâches élémentaires sont plus petites. Quand le processeur utilisé comporte des processeurs élémentaires de transferts, on peut

[†]Quand un cache est géré *par ligne*, ce n'est pas la présence d'un octet ou d'un mot qui est mémorisée dans le cache mais la présence d'un groupe d'octets. En cas de défaut de cache, il faut ranger la ligne puis recharger la nouvelle, ce qui prend généralement plus de temps qu'une écriture-lecture simple bien que dans certaines machines la longueur de la ligne soit prise égale à la somme des largeurs de tous les bancs mémoire. L'IBM 3090[65] et l'Alliant FX-8[27] utilisent cette technique qui permet d'augmenter le volume du cache à faible coût et qui donne de bons résultats avec du code scientifique.



Modélisation de l'Alliant FX/8

- Figure 1.1.1.7.b -



N : NETWORK PROC : PROCESSEUR

M : MEMORY FPU : FLOATING POINT UNIT

C : CACHE IOID : I/O INTERFACE DEVICE

MMU : MEMORY MANAGEMENT UNIT PMD : PERFORMANCE MONITORING DEVICE

Le projet RP3

- Figure 1.1.1.7.c -

aussi essayer de paralléliser transferts et calculs en utilisant des buffers supplémentaires.

La durée d'exécution des arcs du graphe des tâches dont nous avons parlé au début n'est pas nulle ou négligeable sur la plupart des machines. Ces arcs correspondent aux opérations de synchronisation qui sont souvent implémentées dans le système d'exploitation de la machine et non directement par matériel, comme dans l'Alliant. Ceci pousse à diminuer le nombre d'arcs, donc le nombre de tâches et par voie de conséquence à augmenter la taille des tâches élémentaires. Ce problème a déjà été largement étudié pour le Cray X-MP [12] car les temps d'exécution des modules de la première bibliothèque de multitasking étaient de l'ordre de la milliseconde. La situation a été améliorée depuis avec l'introduction du *micro-tasking*.

On voit donc que ces sous-objectifs sont contradictoires, puisqu'il faut dans certains cas augmenter la taille des tâches et dans d'autres la diminuer.

1.1.3. Améliorations espérées

Les améliorations espérées sont de deux natures différentes: performance et facilité d'utilisation.

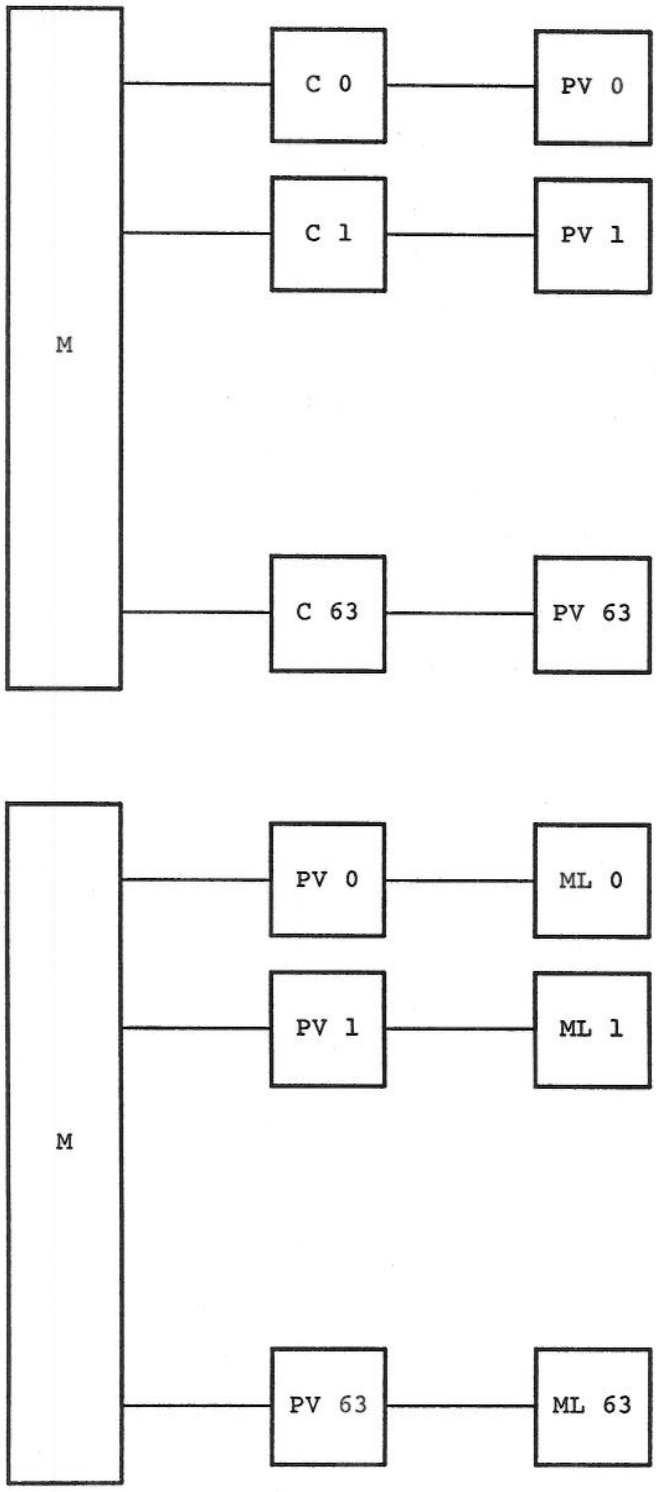
Le développement d'une méthode de partitionnement automatique peut faciliter considérablement la programmation de machines à plusieurs espaces d'adressage comme les multi-array-processors qui obligent le programmeur à partitionner son programme pour tenir dans les mémoires locales et à générer des ordres de transfert.

La simple optimisation d'un programme prend un temps considérable. W. Jalby et U. Meier ont passé des mois pour obtenir des routines de multiplication de matrices efficaces sur l'Alliant FX-8[41] . Il est clair que l'on ne peut pas passer autant de temps sur des procédures moins cruciales, mais l'application automatique des idées qu'ils ont appliquées manuellement est potentiellement intéressante.

Ensuite, à supposer que les programmeurs soient disposés à effectuer manuellement les transformations nécessaires, l'application automatique d'un partitionnement permet de conserver des programmes portables et corrects.

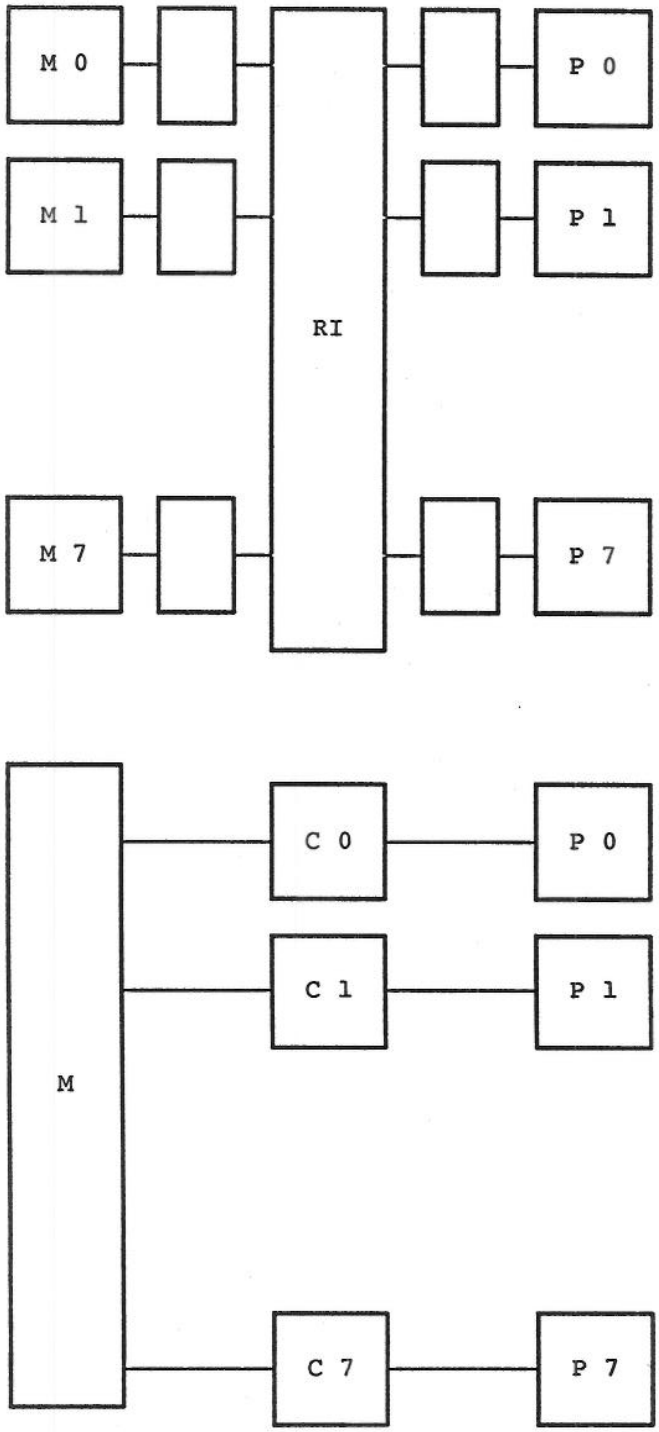
En ce qui concerne le gain en performance, tout dépend de la structure de la machine cible et des outils logiciels disponibles. Si la machine dispose d'un vectoriseur capable d'appliquer la méthode des vagues et si les temps de synchronisation sont suffisamment courts pour qu'une instruction vectorielle puisse constituer une tâche élémentaire, le gain va être entièrement dû à l'amélioration des transferts, et donc relativement limité dans le cas des caches.

Si les tâches doivent être plus considérables ou si le vectoriseur n'applique pas la méthode des vagues, le gain sera de l'ordre de grandeur du nombre de processeurs ou de l'ordre du rapport vectoriel / scalaire. Dans le cas le plus favorable et, nous semble-t-il, le plus fréquent, il faudra multiplier ces deux gains ensemble puis appliquer le gain dû à l'optimisation des transferts.



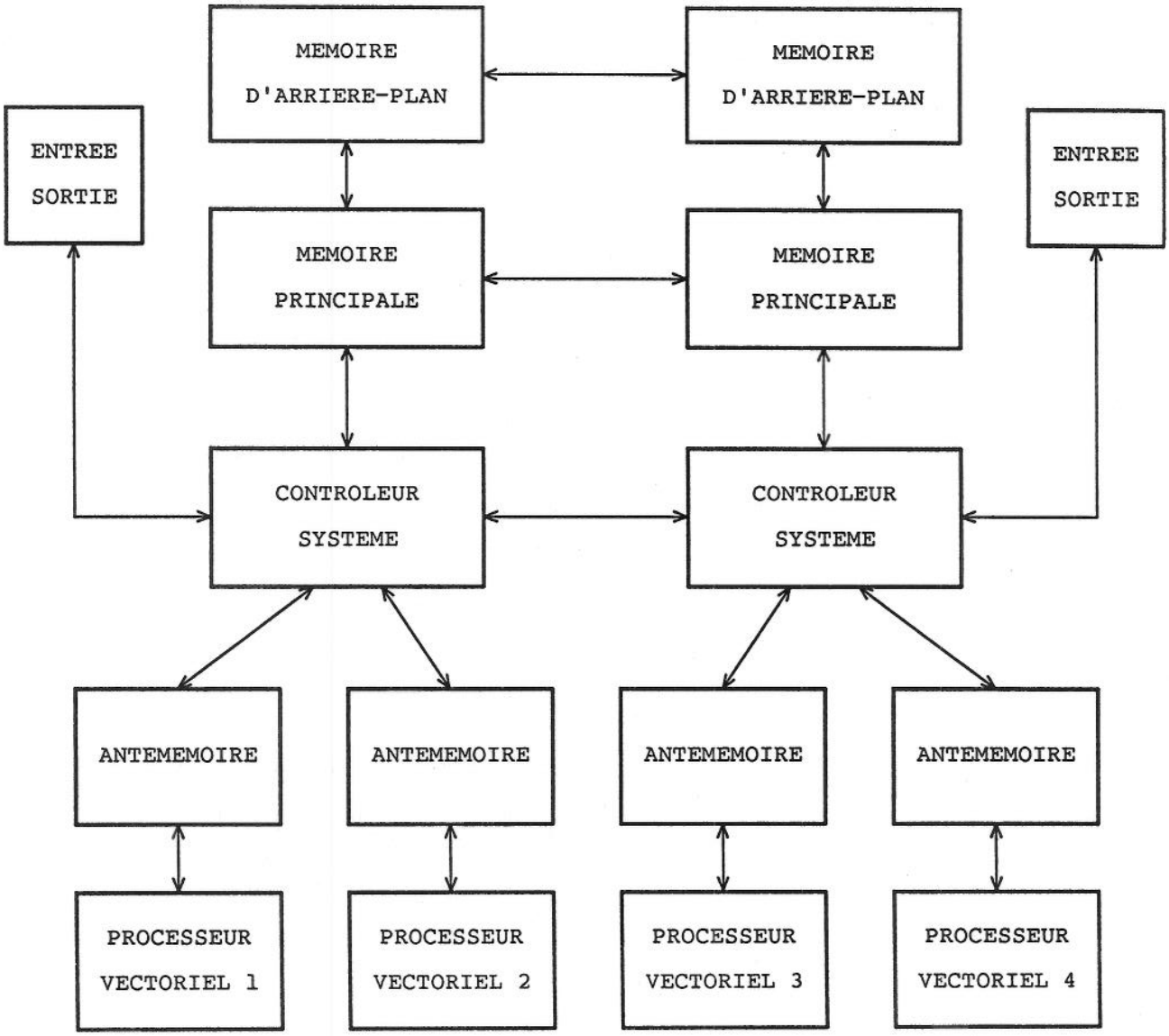
Deux modélisations possibles de RP3

- figure 1.1.1.7.d -



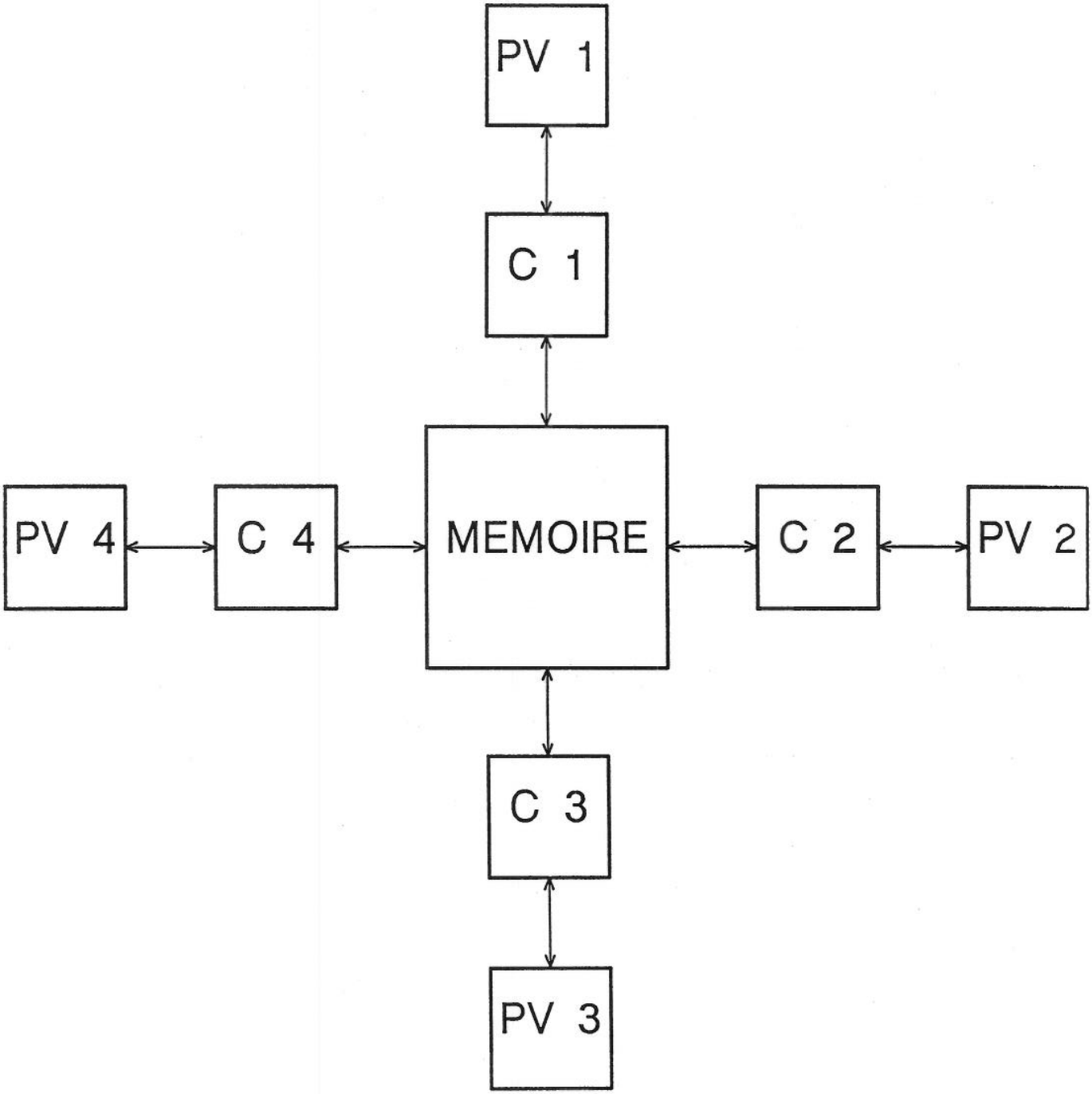
Le NYU Ultracomputer et sa modélisation

- Figure 1.1.1.7.e -



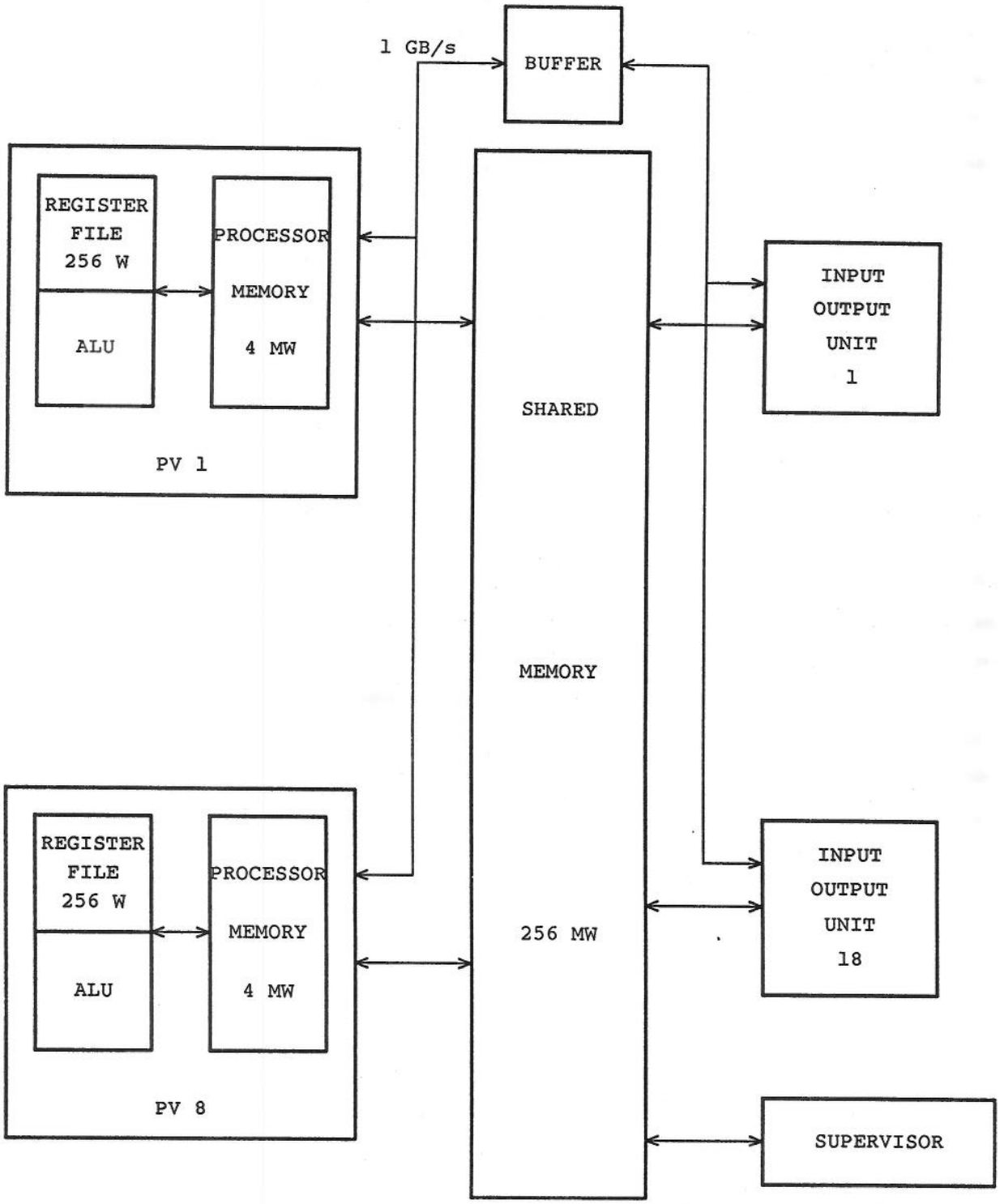
Ordinateur IBM 3090-400 VF

- Figure 1.1.1.7.f -



Une modélisation du 3090-400 VF

- Figure 1.1.1.7.g -



Multiprocesseur ETA 10

- Figure 1.1.1.7.h -

1.2. Exemple

1.2.1. Le problème

Nous avons choisi un exemple représentatif de calculs numériques où la valeur en un point est calculée en fonction des valeurs des points voisins.

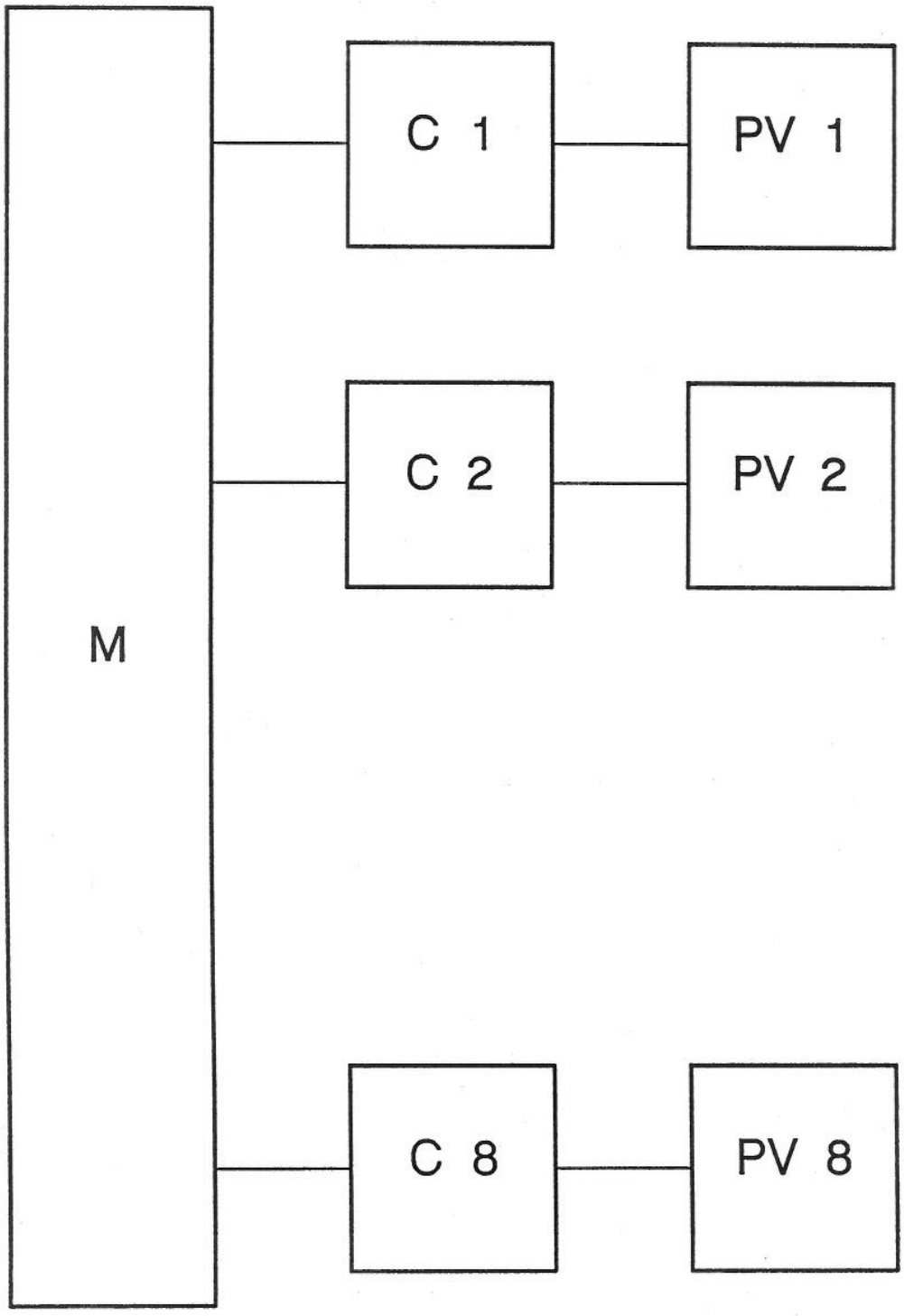
Il s'agit d'une résolution de l'équation de la chaleur sur une barre métallique chauffée aux deux extrémités. Le problème a deux dimensions, une spatiale et une temporelle. La température à l'instant t en un point l de la barre va dépendre de la température des points voisins et du point l à l'instant t–1. En négligeant les coefficients introduits par les unités et par la conductivité de la barre, et en appliquant une méthode de différences finies, on arrive à une équation de récurrence de la forme:

$$B_{t,l} = \frac{B_{t-1,l-1} + B_{t-1,l} + B_{t-1,l+1}}{3.}$$

Les conditions aux limites sont définies en $B_{0,*}$ qui représente l'état initial de la barre, en $B_{*,0}$ et en $B_{*,N+1}$ qui représentent l'état des deux bouts de la barre au cours du temps.

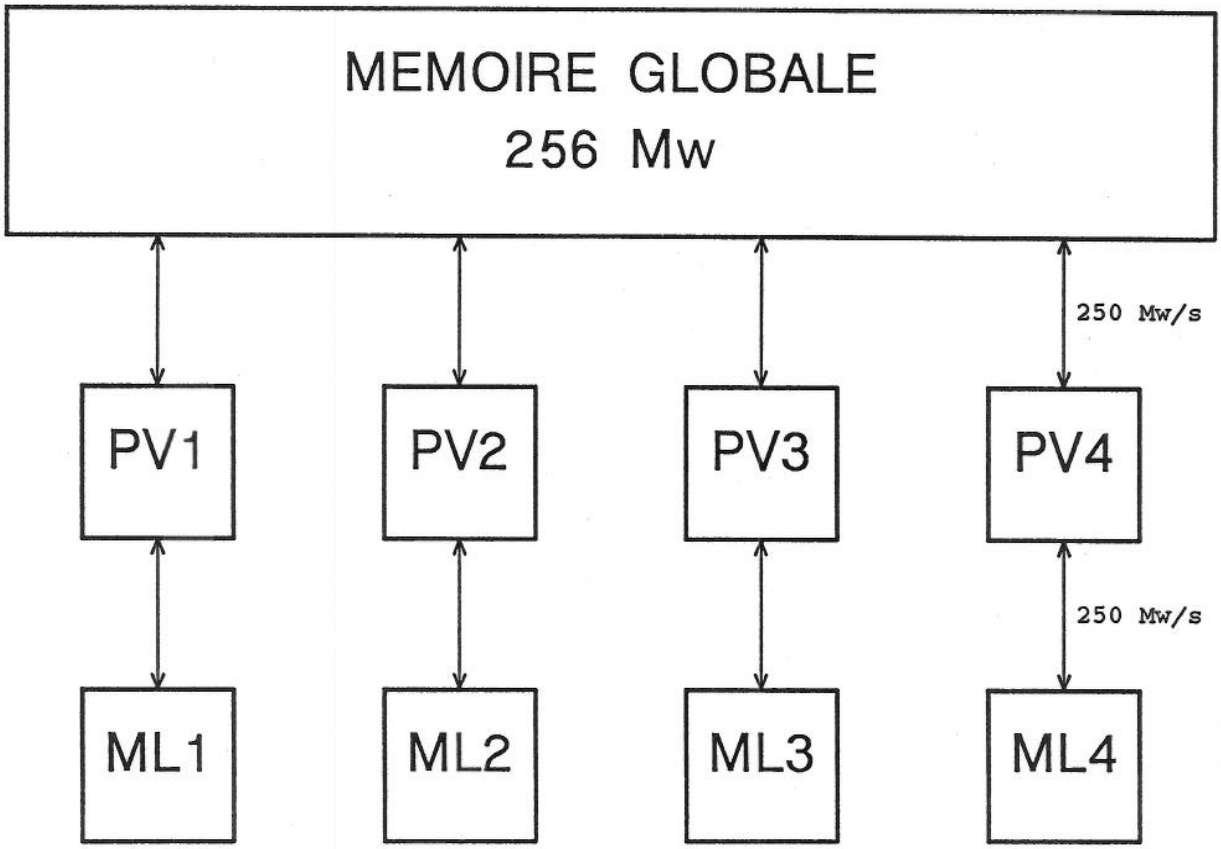
1.2.2. Code séquentiel

Aucun programmeur ne laissera l'équation précédente sous cette forme si seul l'état de la barre à l'instant final importe. Il essaiera de réduire l'espace mémoire nécessaire au calcul. De plus, les identificateurs seront choisis pour être un peu plus significatifs. Pour simplifier les conditions initiales, on se contente de chauffer symétriquement les deux bouts de la barre de manière à en augmenter la température de INCRTEMP à chaque pas de temps. L'instant final est TFIN. Le programme typique ressemble donc au suivant:



Modélisation de l'ETA 10

- Figure 1.1.1.7.i -



Modélisation du Cray 2

- Figure 1.1.1.7.j -

```
      REAL BARRE(0:LNGBARRE+1)

C     On suppose que le tableau barre est initialisé pour T=0
      DO T = 1, TFIN
        BARRE(0) = BARRE(0) + INCRTEMP
        BARRE(LNGBARRE+1) = BARRE(0)
        T1 = BARRE(0)
        DO L = 1, LNGBARRE
          T2 = T1
          T1 = BARRE(L)
```

```
        BARRE(L) = (BARRE(L+1) + T1 + T2)/3.
      ENDDO L
    ENDDO T
```

Calcul de la température d'une barre

On remarque que ce programme est nettement moins lisible que l'équation du paragraphe précédent.

1.2.3. Application d'un restructureur

La boucle interne ressemble fort à un exemple utilisé par Allen et Kennedy[6] pour montrer comment leur vectoriseur PFC modifiait les programmes.

```
      T1 = X(1)
      T2 = X(2)
      DO 100 I=2, N-1
        X(I) = (X(I+1) + T1 + T2)/3.
        T1 = T2
        T2 = X(I+1)
100   CONTINUE
```

Version originale du programme d'Allen et Kennedy

Le programme original est d'abord vectorisé en utilisant l'expansion de scalaire sur T1 et T2:

```
T1 = X(1)
T2 = X(2)
TMP1(0) = T1
TMP2(0) = T2
TMP2(1:N-2) = X(3:N)
TMP1(1:N-1) = TMP2(0,N-2)
X(2:N-1) = (X(2:N-1) + TMP2(0,N-3) + TMP1(0:N-3))/3.
```

Programme vectorisé

Le programme est bien vectoriel, mais pas très astucieux puisqu'un seul temporaire aurait suffit. Mais PFC va plus loin et élimine les deux temporaires en exploitant la sémantique de Fortran 8X pour obtenir une instruction vectorielle unique équivalant à l'équation mathématique initiale:

```
X(2:N-1) = (X(1:N-2) + X(2:N-1) + X(3:N)) / 3.0
```

Instruction vectorielle

La manière dont ce résultat est obtenu n'est pas expliquée dans le rapport que nous avons cité. C'est une illustration de *Forward Substitution* appliquée à des **tableaux**. Il montre cependant que partir d'un programme FORTRAN ou d'une équation de récurrence peut être équivalent.

Ce genre d'instruction vectorielle montre aussi l'amélioration de lisibilité qu'un FORTRAN vectoriel apporte, au prix il est vrai d'un effort supplémentaire pour le compilateur qui doit générer proprement les temporaires nécessaires. Notre programme devient maintenant:

```
      DO T = 1, TFIN
        BARRE(0) = BARRE(0) + INCRTEMP
        BARRE(LNGBARRE+1) = BARRE(0)
        BARRE(1:LNGBARRE) = (BARRE(0:LNGBARRE) + BARRE(1:LNGBARRE)
     &          + BARRE(2:LNGBARRE))/3.
      ENDDO T
```

Version vectorielle du calcul de température

Le calcul itératif et le schéma à trois points sont maintenant mis en évidence.

1.2.4. Adaptation à un multiprocesseur

La première idée consiste à découper l'instruction vectorielle en tronçons, de manière à avoir du travail pour chaque processeur. La longueur des tronçons peut être choisie en fonction du temps de démarrage des unités vectorielles, de la longueur des registres vectoriels, qui est en principe corrélée au paramètre précédent, ou du nombre de processeurs. La première solution est plus intéressante quand la machine peut fonctionner en mode dégradé, avec moins de 8 processeurs en cas de panne), ou bien répartir aléatoirement ses processeurs à des tâches totalement indépendantes (multitasking). Dans ces deux cas (non exclusifs), le nombre de processeurs utiles est inconnu lors de la compilation. Le programme résultant est le suivant:

```
      DO T = 1, TFIN
        BARRE(0) = BARRE(0) + INCRTEMP
        BARRE(LNGBARRE+1) = BARRE(0)
        DO L = 1, LNGBARRE - L1, L1
          BARRE(L:L+L1-1) = (BARRE(L-1:L+L1-2) + BARRE(L:L+L1-1)
     &                    + BARRE(L+1:L+L1))/3.
        ENDDO L
        IF ( MOD(LNGBARRE,L1).NE.1) THEN finir ce qui reste à calculer
      ENDDO T
```

Version vectorielle (mal compilée) du calcul de température

Cette version est malheureusement fausse car la sémantique du programme initial n'est pas respectée. La définition de BARRE(L+L1-1) à l'itération L écrase la valeur recherchée par la référence BARRE(L-1) à l'itération L+L1. Le compilateur de l'Alliant FX/8, qui procède par tronçonnage pour alimenter les processeurs élémentaires de la machine, nous a semblé commettre cette erreur[39] .

Quand la machine cible dispose d'un cache ou d'une mémoire locale, on s'aperçoit que cette solution nécessite un nombre de transferts de données égal au nombre d'utilisations. Il est un effet peu probable que le tableau BARRE soit suffisamment petit pour tenir intégralement en mémoire rapide et chaque calcul d'un nouveau tronçon va nécessiter l'évacuation des nouvelles valeurs du tronçon précédent et le chargement des anciennes valeurs du nouveau. De plus, le nombre de synchronisations est élevé, ce qui peut être gênant quand elles sont gérées par logiciel.

C'est la solution qui a été retenue par le préprocesseur VAST utilisé sur l'Alliant FX/8, machine qui effectue les synchronisations entre processeurs d'une manière extrêmement efficace.

1.2.5. Problème de synchronisation posé par un partitionnement bidimensionnel

Pour résoudre les deux problèmes qui viennent d'être exposés, il faut réutiliser les données quand elles se trouvent en mémoire rapide et augmenter la taille des tâches. Le premier objectif est atteint en itérant plusieurs fois selon T avant de passer aux valeurs voisines en L. Le second, en définissant des tâches bidimensionnelles.

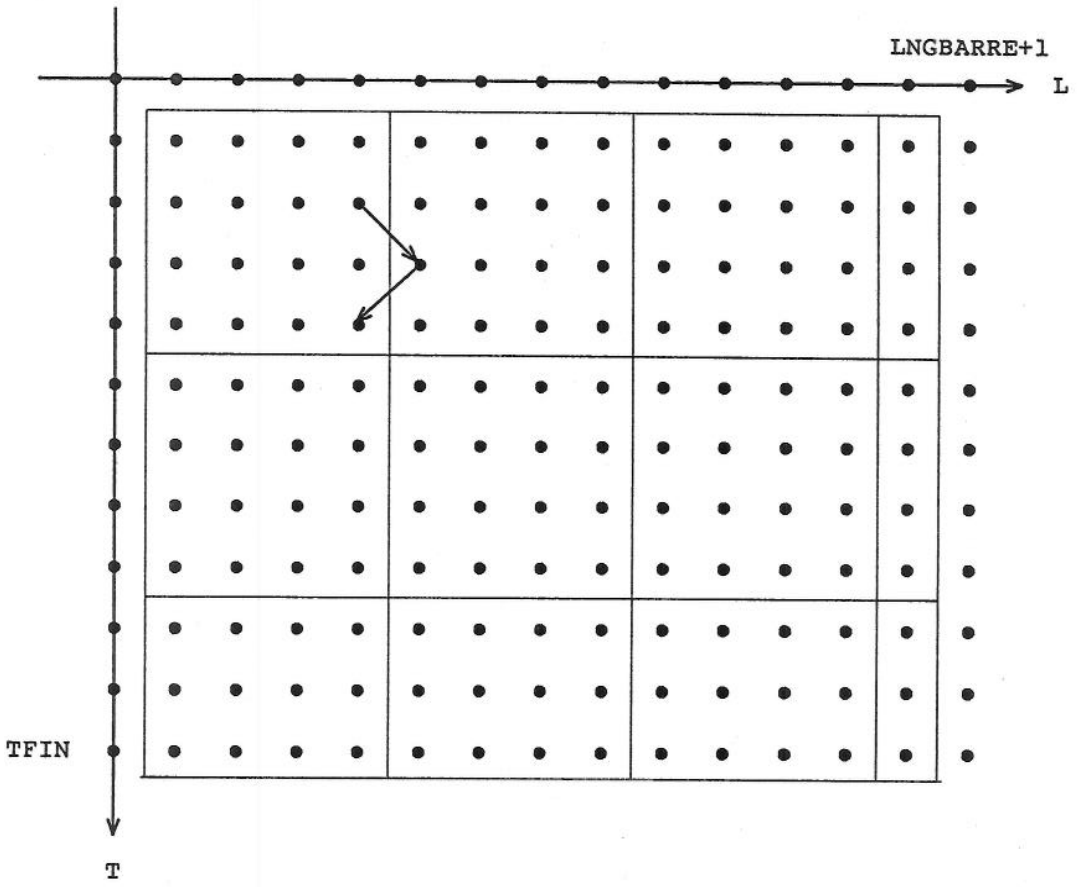
Pour ce faire, on peut tout d'abord penser utiliser le blocage de boucle (*Strip-Mining*[68]) qui est utilisé classiquement pour adapter la quantité de parallélisme disponible à la machine cible ou à ses

registres. En bloquant les deux boucles sur T et L par des facteurs de T1 et L1, et en permutant les deux boucles internes, on obtient bien des tâches rectangulaires dans l'espace d'itération. Elles sont représentées figure 1.2.5. pour T1=L1=4 et voici le programme résultant:

```
      DO T = 1, TFIN, T1
        DO L = 1, LNGBARRE, L1
C         Code d'une tache
          DO T' = T, MIN(T+T1-1,TFIN)
            IF(L.EQ.1) THEN
              BARRE(0) = BARRE(0) + INCRTEMP
              BARRE(LNGBARRE+1) = BARRE(0)
            ENDIF
            L' = MIN(L+L1-1,LNGBARRE)
            BARRE(L:L') = (BARRE(L-1:L'-1) + BARRE(L:L') + BARRE(L+1:L'+1))/3.
          ENDDO T'
C         Fin du code de la tache
        ENDDO L
      ENDDO T
```

Version *bloquée* du calcul de température

Mais si le blocage de boucle est toujours valide, l'échange de boucles n'est possible que si les dépendances vérifient certaines conditions[68, 5] et le programme partitionné que nous venons d'obtenir ne fournira des résultats exacts que si des synchronisations sont ajoutées à chaque itération T' de chaque tâche (cf. figure 1.2.5). Autrement dit, il faudrait se contenter de tâches monodimensionnelles, qui seraient constituées d'un tronçon de ligne comme dans le paragraphe 1.2.4. Le gain qu'on pouvait attendre en augmentation de localité de référence serait alors réduit à néant. Chercher à n'avoir qu'une synchronisation par tâche conduit à une étreinte fatale: la tâche t_1 ne peut être exécutée ni avant, ni après, ni parallèlement à la tâche t_2.



Partitionnement par blocage et permutation de boucles

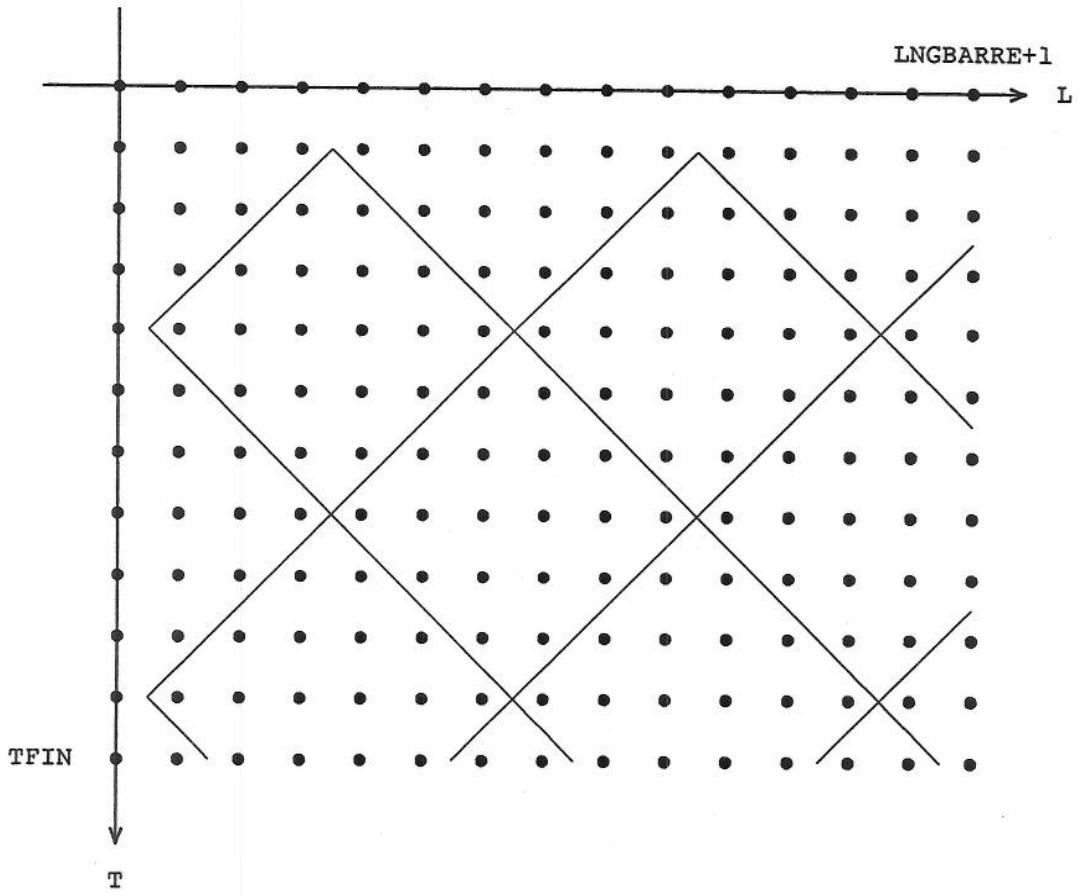
- Figure 1.2.5 -

1.2.6. La solution proposée

La solution précédente se ramène à découper le domaine d'itération en pavés. C'est ce découpage que nous appelons partitionnement.

La solution proposée ne définit pas les partitionnements à partir de structures de contrôle comme les boucles DO mais utilise l'ordre partiel sur les calculs que génère le flots des données. Dans le cas précédent, l'ensemble des calculs pourrait par exemple être découpé par des droites de pente 1 et -1. Ces deux pentes sont déterminées en étudiant la manière dont les calculs échangent de l'information. Le nouveau partitionnement obtenu est dessiné figure 1.2.6. et le lecteur pourra se convaincre qu'il est possible d'en synchroniser correctement l'exécution avec un point de synchronisation par tâche élémentaire.

Du fait de la dualité qui existe entre les synchronisations et les communications, cette solution minimise simultanément les deux. En prenant pour facteur de blocage n pour les deux directions de



Un partitionnement admissible

- Figure 1.2.6 -

l'espace d'itération, on obtient $3/n$ communications par point calculé avec le découpage carré et $2/n + 1/2$ avec le découpage en losange. Le nombre de synchronisations passe de $1/n$ à $1/n^2$.

Appliquée au cas de l'Alliant FX-8, ce partitionnement a permis d'obtenir avec ce calcul de température une augmentation de 50 % de la vitesse de calcul uniquement grâce à la bonne utilisation du cache qui en résulte.

1.3. Hypothèses de travail

Nous ne prétendons pas apporter une solution universelle au problème du partitionnement des programmes. C'est un vaste sujet de recherche qui est en train de prendre de l'importance avec le développement du marché des multiprocesseurs et de nombreuses personnes l'étudient à l'heure actuelle.

Nous allons donc nous intéresser à un sous-problème en restreignant à la fois les portions de code considérées (§ 1.3.1) et l'ensemble des machines cibles (§ 1.3.2) et en profitant du travail qui a déjà été effectué pour les machines vectorielles (*vectoriseur*: § 1.3.3). Certaines des hypothèses faites ici seront affaiblies dans les chapitres suivants toutes les fois que ce sera possible.

Nous allons aussi restreindre l'espace des solutions en prévoyant l'application automatique (§ 1.3.4) de notre méthode de partitionnement et le simplifier en décomposant le problème (§ 1.3.5).

1.3.1. Portions de code considérées

Nous nous intéressons exclusivement aux boucles DO imbriquées des langages impératifs procéduraux comme Fortran. Rappelons qu'un ensemble de boucles est dit *imbriqué* si pour toute paire de boucles (B, B'), ou bien B fait partie du corps de B' ou bien B' fait partie du corps de B. Une structure de contrôle comme:

```
      DO 100 I = ...
        DO 200 J= ...
          . . .
200     CONTINUE
        DO 300 K = ...
          . . .
300     CONTINUE
100   CONTINUE
```

Boucles non-imbriquées

n'est pas appelée un ensemble de boucles imbriquées.

Du fait de leur imbrication, l'exécution de ces boucles consomme une part considérable du temps d'exécution et leur optimisation est donc intéressante. D'autre part, la régularité du domaine d'itération qui est parcouru à l'exécution simplifie le partitionnement. Etant donné la nouveauté du problème, il

vaut mieux commencer par étudier le cas régulier avant d'étudier le partitionnement d'un graphe de tâches quelconques. Enfin, dans beaucoup de cas, la durée d'exécution d'une tâche sera proportionnelle au nombre d'itérations de boucle qu'elle contiendra, ce qui simplifie le problème de l'ordonnancement.

Pour que le domaine d'itération soit parfaitement défini statiquement, nous ne traiterons que les boucles dont les expressions de bornes sont des expressions affines des indices englobants. Cette restriction peut sembler draconienne mais nous espérons que les méthodes de calcul de prédicats[16] et de propagation interprocédurale[62, 13] de prédicats permettront de ramener la plupart des boucles imbriquées à ce cas particulier.

Ces boucles doivent de plus être **parfaitement** imbriquées, c'est-à-dire que toutes les instructions autres que les ordres DO doivent se trouver dans le corps de la boucle la plus imbriquée. Cette restriction est trop forte et il faudra la lever pour rendre applicable la méthode de partitionnement présentée ici. Nous l'avons néanmoins jugée indispensable pour simplifier la génération de code qui est déjà suffisamment complexe avec cette restriction. Dans le chapitre 3 consacré à la génération de code, nous signalons toutes les utilisations de cette hypothèse.

Sur le plan théorique, les boucles non parfaitement imbriquées ne posent pas de problème car Abu-Sufah a montré dans son PhD[1] que tout ensemble de boucles imbriquées pouvait être transformé en un ensemble de boucles parfaitement imbriquées. Par exemple, le programme

---

```
      DO 100 I=1,N
         T(I)=0.
         DO 100 J=1,M
            T(I)=T(I)+A(I,J)
100   CONTINUE
```

Boucles non parfaitement imbriquées

---

devient:

```
      DO 100 I=1,N
        T(I)=0.
100   CONTINUE

      DO 101 I=1,N
        DO 101 J=1,M
          T(I)=T(I)+A(I,J)
101   CONTINUE
```

Boucles parfaitement imbriquées

Malheureusement cette transformation nécessite parfois l'ajout de tests dans le corps de boucle, tests qui vont, au pire, empêcher la vectorisation de la boucle la plus interne et, au mieux, ralentir considérablement l'exécution vectorielle en nécessitant la génération d'instructions vectorielles conditionnelles ou d'instructions de *scatter-gather*. Or ces tests seront faux la plupart du temps.

On peut envisager d'ajouter ces tests dans une première phase de normalisation des boucles imbriquées, puis d'appliquer notre méthode de partitionnement et enfin d'essayer de retirer la plupart de ces tests en appliquant une méthode d'élimination de code mort basée sur des prédicats calculés interprocéduralement. Les instructions non parfaitement imbriquées ne devraient alors être conservées que dans les tâches qui se trouvent à la surface du domaine d'itération.

Une autre méthode, plus efficace, consiste à prendre en compte ces instructions pendant le partitionnement, et à profiter du fait que les tâches se trouvant à la surface du domaine d'itération ont de toute façon un code particulier pour les y réintroduire. C'est cette voie que nous avons jugée trop délicate pour être mise en oeuvre dès la première présentation du partitionnement.

Les boucles considérées devront aussi avoir un incrément unitaire pour simplifier l'évaluation du nombre de calculs et le remplacement des indices initiaux par certaines de leurs combinaisons linéaires. Les méthodes traditionnelles de vectorisation nécessitent la même *normalisation* des boucles.

Nous avons déjà envisagé la présence de tests dans les corps de boucle pour résoudre le problème de la non parfaite imbrication. Les tests peuvent poser deux problèmes. S'ils conduisent à la sortie d'une ou de plusieurs boucles (EXIT), ils empêcheront vraisemblablement toute parallélisation et indirectement tout partitionnement intéressant. S'ils ne provoquent que des branchements internes au corps de boucle, la situation est moins mauvaise mais nous ne pouvons plus faire l'hypothèse de proportionalité du temps d'exécutionet du nombre d'exécutions du corps de boucle. Reste la possibilité d'utiliser une valeur

moyenne, évaluable à partir du ratio *true/false* du test, comme temps d'exécution. Ce ratio serait donné par l'utilisateur.

La présence d'appels de sous-programmes ou de fonctions pose aussi plusieurs problèmes. Tout d'abord, il faudra être capable de calculer les dépendances propres au corps de boucle et recourir à une méthode complexe[63] car les méthodes traditionnelles ne peuvent pas traiter ce cas. Une fois cet obstacle surmonté, il reste à être capable d'évaluer le temps d'exécution de la routine appelée et à supposer qu'il est indépendant de la valeur des paramètres d'appel puis, au moment de la génération de code, à estimer le volume mémoire qui va être nécessaire pour exécuter cette routine. En effet, qu'il s'agisse d'un cache dont on veut éviter le débordement ou d'une mémoire locale limitée, il faut disposer d'une formule explicite donnant l'espace mémoire requis en fonction des paramètres. Cette fonction peut être une constante.

L'élimination des boucles contenant des appels de procédures est plus gênante qu'il n'y paraît au premier abord. En effet, les boucles DO imbriquées sur beaucoup de niveaux sont rares ou, plus exactement, cachées par des CALLs dès que le programmeur structure son programmme ou fait appel à des modules de bibliothèques scientifiques. L'expansion de procédure[37] pourrait constituer une solution pour les procédures les plus simples.

Enfin, pour le calcul des dépendances, pour le calcul du volume de données manipulées par une tâche et pour la génération de code, nous ne pourrons traiter que les corps de boucle qui ne contiennent que des références affines aux tableaux. Une référence affine est une référence dans laquelle toutes les expressions d'indices sont des fonctions affines des indices de boucle. Cette restriction, classique dans le domaine de la parallélisation automatique, élimine les codes qui font appel à l'indirection comme ceux qui utilisent des matrices creuses. P. Jouvelot[42] vient de mener une étude préliminaire de ce problème délicat mais aucune méthode satisfaisante n'est encore disponible. On verra dans le chapitre 3 que le problème du maintien de la cohérence des copies ne permet pas de se satisfaire d'une hypothèse pessimiste de dépendance par défaut dans le cas des machines à mémoire locale.

1.3.2. Application préalable d'un paralléliseur

L'application préalable d'un paralléliseur permet de garantir le respect de certaines des restrictions que nous venons d'énumérer, et de diminuer le nombre des dépendances qui contraignent le partitionnement et l'ordonnancement des calculs.

Il n'est pas possible de passer en revue toutes les transformations de programme qui peuvent être appliquées par un paralléliseur. Nous ne présentons rapidement ici que celles qui peuvent améliorer l'application de notre méthode de partitionnement. Le lecteur intéressé pourra consulter [68, 3] ou bien [8] pour trouver des présentations plus complètes.

1.3.2.1. Distribution et fusion de boucle

La distribution et la fusion de boucle (*Loop Distribution* ou *Loop Fission* et *Loop Fusion*[68]) sont deux transformations inverses qui permettent toutes les deux de diminuer les échanges entre les deux niveaux mémoire, comme l'a montré Abu-Sufah [1] en étudiant le comportement des programmes exécutés dans un environnement de mémoire virtuelle. Sous certaines conditions, il est possible de distribuer les instructions du corps de boucle dans deux nouvelles boucles pour diminuer les ressources mémoires nécessaires au traitement de chaque boucle. Réciproquement, si la même variable apparaît dans deux boucles similaires, il faut essayer de fusionner les boucles pour utiliser au maximum les variables une fois qu'elles sont chargées dans le niveau mémoire le plus rapide.

```
      DO 100 I = 1, N
        DO 100 J = 1, M
          A(I,J) = A(I,J) + A(I,J-1)
          B(I,J) = B(I,J) + B(I-1,J)
  100 CONTINUE
```

```
      DO 100 I = 1, N
        DO 100 J = 1, M
          A(I,J) = A(I,J) + A(I,J-1)
  100 CONTINUE
      DO 101 I = 1, N
        DO 101 J = 1,M
          B(I,J) = B(I,J) + B(I-1,J)
  101 CONTINUE
```

Distribution de boucles

Dans l'exemple précédent, la distribution de boucle permet en plus de dissocier la dépendance sur J introduite par les références au tableau A et la dépendance sur I introduite par B. Aucune des deux boucles n'était parallèle avant la distribution. Après transformation, chacune des deux boucles imbriquées peut être exécutée vectoriellement, moyennant un échange des boucles I et J qui contrôlent le calcul de A(I,J) (cf. § 1.3.2.4).

1.3.2.2. Alignement de boucle

L'alignement de boucle (*Loop Alignment*[6]) permet aussi d'augmenter le parallélisme potentiel. Cette transformation consiste à translater les itérations d'une instruction d'un corps de boucle par rapport aux itérations du reste. Cela a pour effet d'appliquer la même translation (ou son opposée) aux vecteurs de dépendances existant entre des références de l'instruction décalée et des références des autres instructions. Voici un exemple tiré de l'algorithme de QD-décomposition:[54]

```
      DO I = 2, N
         DO J = 0, N-1
(S1)        Q(I,J+1) = Q(I-1,J+1) + E(I-1,J+1) - E(I,J)
(S2)        E(I,J) = Q(I-1,J+1) * E(I-1,J) / Q(I,J)
         ENDDO
      ENDDO
```

Version initiale

Ces deux boucles imbriquées génèrent 6 conflits entre les définitions de E et Q et leurs utilisations (figure 1.3.2.2.a). Le graphe de dépendance, qui est représenté en figure 1.3.2.2.b (chaque point représente une exécution d'une instruction et chaque flèche une occurrence de conflit), ne comporte cependant que les quatre vecteurs de dépendance suivants:

$$\vec{d}_o = \begin{pmatrix}0\\0\end{pmatrix} \quad \vec{d}_1 = \begin{pmatrix}1\\0\end{pmatrix} \quad \vec{d}_2 = \begin{pmatrix}0\\1\end{pmatrix} \quad \vec{d}_3 = \begin{pmatrix}1\\-1\end{pmatrix}$$

car plusieurs conflits conduisent au même vecteur de dépendance. Le vecteur $\vec{d}_o$ correspond à une antidépendance; nous l'éliminons en renommant la seconde référence à E en une référence à E' dans l'instruction S1.

Les vecteurs de dépendances $\vec{d}_1$, $\vec{d}_2$ et $\vec{d}_3$ empêchent l'exécution vectorielle des boucles sur I et sur J. Il est cependant possible de translater les itérations de S1 de 1 selon l'indice J, c'est-à-dire de retarder les calculs des Q jusqu'au moment où ils sont effectivement utilisés dans les calculs des E. A repère constant, cela se traduit par l'addition de –1 à toutes les expressions d'indices contenant J:

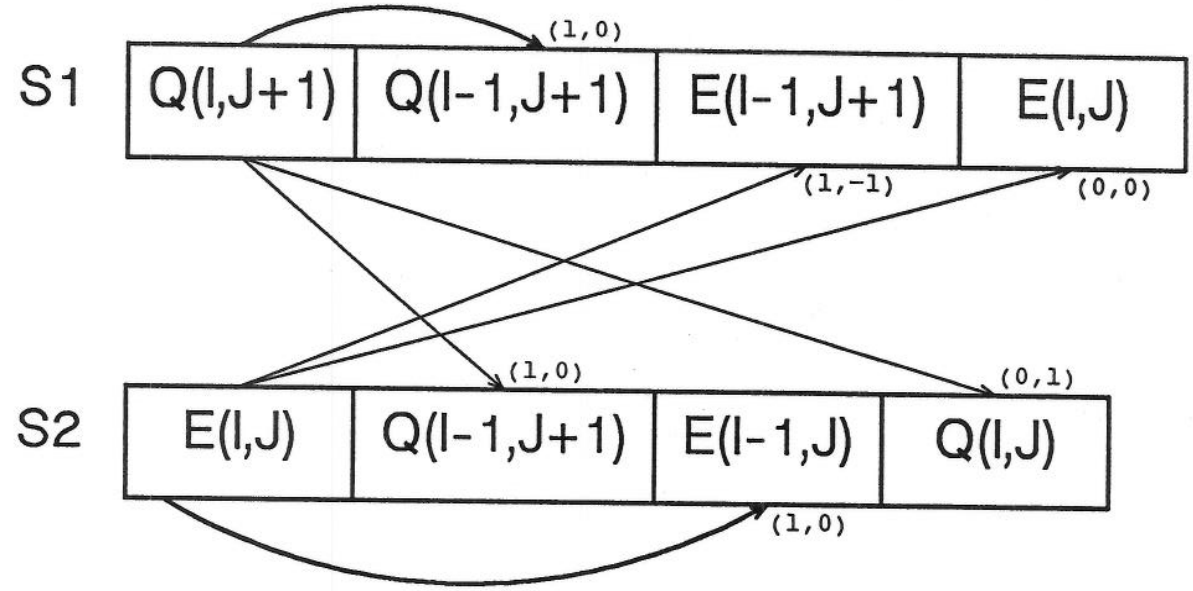
```
      DO I=2, N
          E(I,0) = Q(I-1,1) * E(I-1,0) / Q(I,0)
      ENDDO
      DO I=2, N
          DO // J=1, N-1
(S1')         Q(I,J) = Q(I-1,J) + E(I-1,J) - E'(I,J-1)
(S2')         E(I,J) = Q(I-1,J+1) * E(I-1,J) / Q(I,J)
          ENDDO
      ENDDO
      DO I=2, N
          Q(I,N) = Q(I-1,N) + E(I-1,N) - E'(I,N-1)
      ENDDO
```

Version alignée

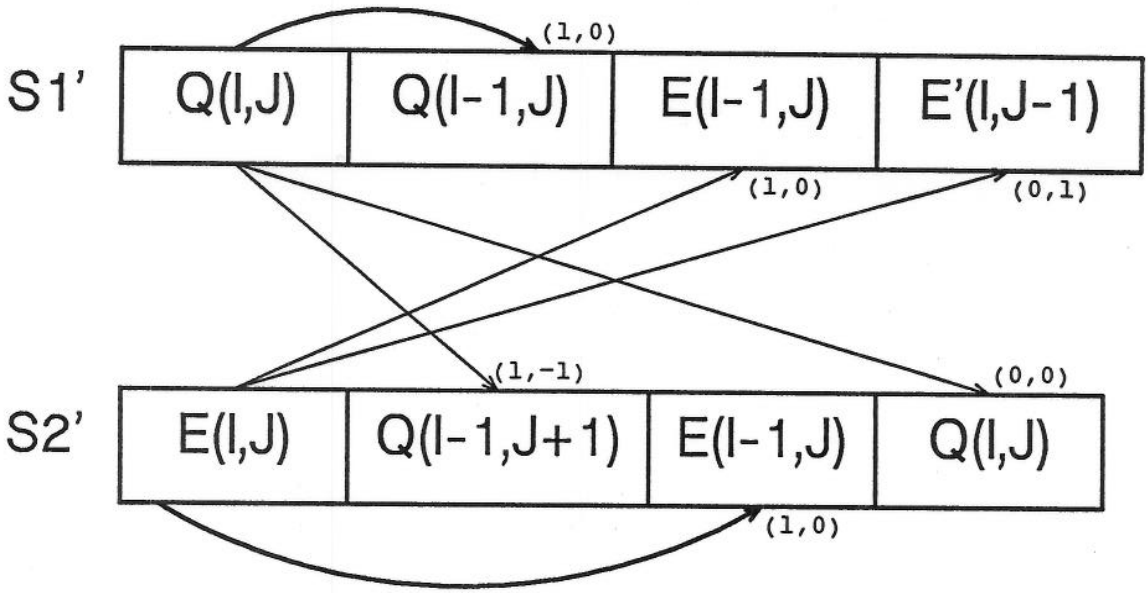
```
      DO I=2, N
          E(I,0) = Q(I-1,1) * E(I-1,0) / Q(I,0)
      ENDDO
      DO I=2, N
(S1')     Q(I,1:N-1) = Q(I-1,1:N-1) + E(I-1,1:N-1) - E'(I,0:N-2)
(S2')     E(I,1:N-1) = Q(I-1,2:N) * E(I-1,1:N-1) / Q(I,1:N-1)
      ENDDO
      DO I=2, N
          Q(I,N) = Q(I-1,N) + E(I-1,N) - E'(I,N-1)
      ENDDO
```

Version vectorielle alignée

Le code transformé ne génère plus que deux vecteurs de dépendance. Les conflits internes à S1 et à S2 donnent toujours les mêmes vecteurs de dépendances mais les trois autres sont modifiés de $\pm\vec{j}$ suivant



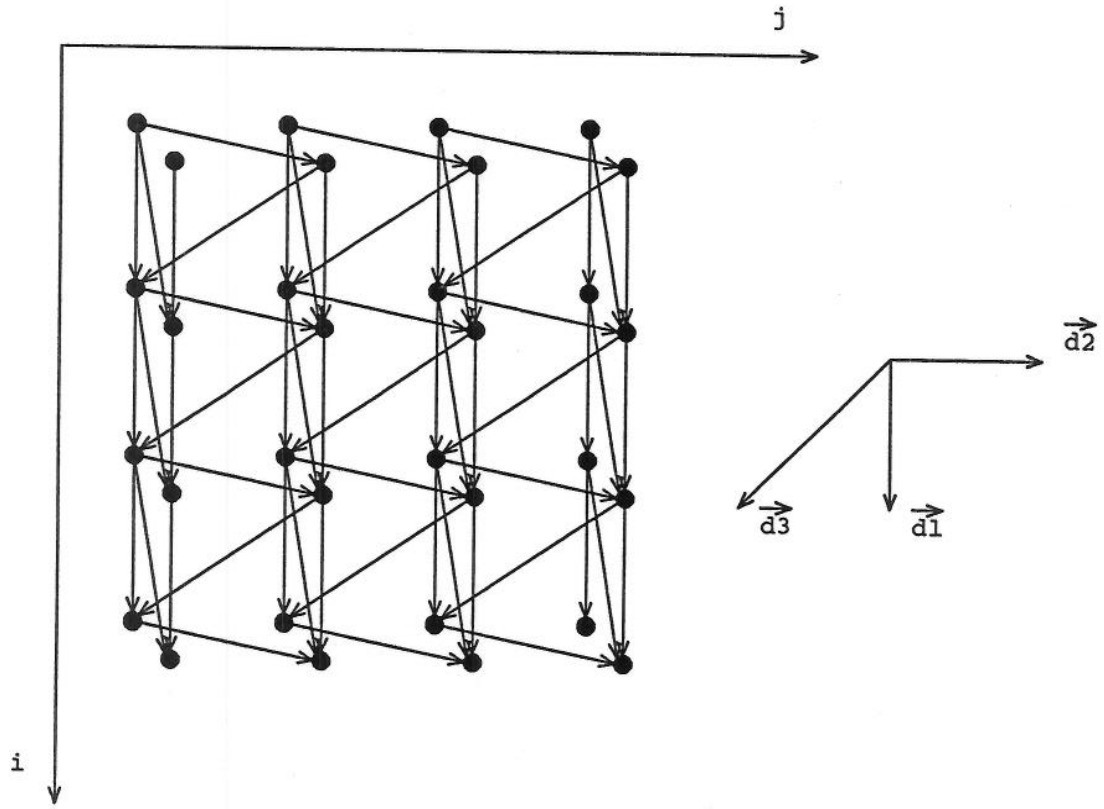
Avant alignement



Apres alignement

Conflits dans l'algorithme de QD Décomposition

- Figure 1.3.2.2.a -



Graphe de dépendance initial de la décomposition QD

- Figure 1.3.2.2.b -

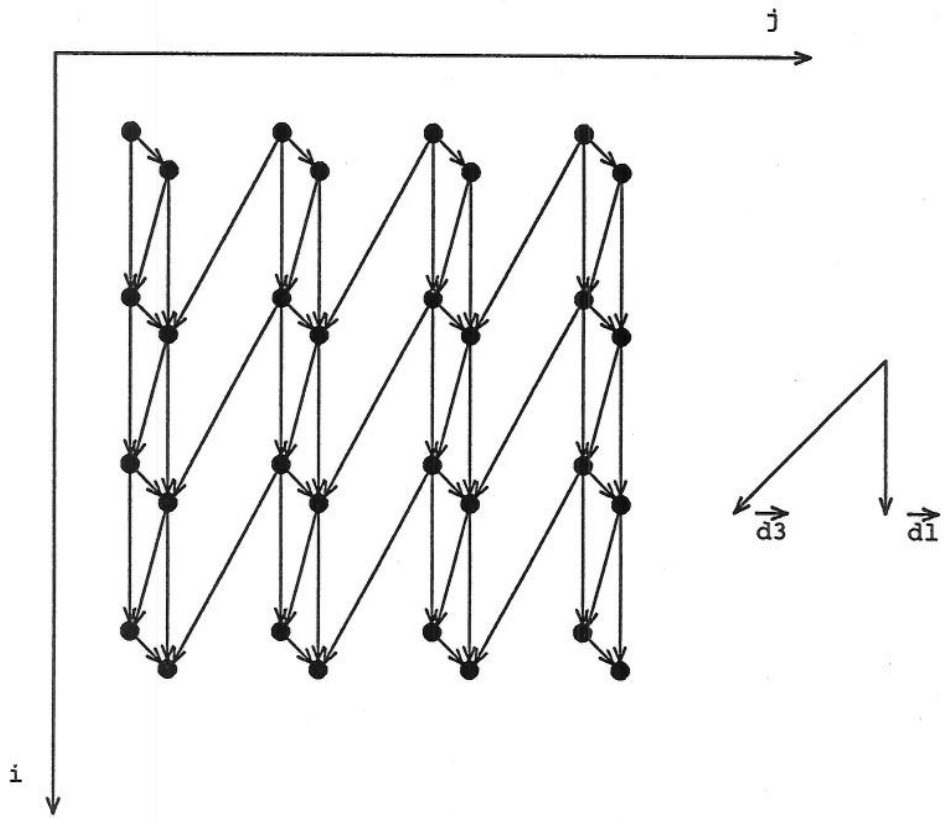
qu'ils sont orientés de S2 vers S1 ou de S1 vers S2. Le vecteur $\vec{d}_o$ aurait empêché cette transformation car la translation aurait rendu $\vec{d}_o$ lexiconégatif. Les vecteurs de dépendance restant sont:

$$\vec{d}_1' = \vec{d}_1 = \begin{pmatrix}1\\0\end{pmatrix} \qquad \vec{d}_2' = \vec{d}_3 = \begin{pmatrix}1\\-1\end{pmatrix}$$

et la boucle sur J est maintenant parallèle (figure 1.3.2.2.c).

1.3.2.3. Transformations sur les scalaires

Les transformations sur les scalaires - expansion, renommage et susbstitution - sont utiles pour supprimer des antidépendances et, dans le cas de la substitution, des datadépendances. Mais l'expansion a l'inconvénient de consommer inutilement de l'espace mémoire car le parallélisme généré sera très supérieur au parallélisme effectif de la machine cible. Il vaudrait mieux effectuer l'expansion au moment de la génération de code en se contentant de marquer la variable comme devant être expansée et en



Graphe de dépendance aligné de la décomposition QD

- Figure 1.3.2.2.c -

oubliant les antidépendances créées par les scalaires.

Le remplacement du calcul itératif des variables inductives par un calcul direct permet de supprimer une datadépendence.

1.3.2.4. Echange de boucles

L'échange de boucles n'augmente pas le parallélisme. Cette transformation, qui ne modifie que l'ordonnancement, est destinée à mettre le parallélisme au niveau où la machine peut l'utiliser: sur la boucle la plus interne s'il s'agit d'une machine vectorielle, et sur la boucle la plus externe s'il s'agit d'une machine parallèle dont les processeurs ne sont pas vectoriels. Il s'agit donc d'une technique d'adaptation à la machine, dont nous montrerons qu'elle ne constitue qu'un cas dégénéré de notre méthode (§ 8.2).

Il faudrait donc supprimer l'échange de boucles du catalogue des transformations d'un paralléliseur appliquant le partitionnement de boucles pour éviter un double emploi. L'application d'un échange de

boucles avant un partitionnement ne provoquerait qu'une perte de temps mais l'ordre inverse donnerait des résultats imprévisibles. Deux raisons pourrait justifier un maintien de l'échange de boucles: des conditions d'applications moins strictes ou un temps de compilation beaucoup plus faible.

1.3.2.5. Analyse sémantique du programme

La propagation interprocédurale de constantes peut permettre de simplifier les expressions de bornes de boucle et les expressions d'indices et donc les rendre compatibles avec les critères que nous avons énoncés en 1.2.1.

La propagation intraprocédurale de prédicats, telle que nous l'avons proposée dans [63] , permet aussi d'améliorer le calcul du graphe de dépendance et donc d'augmenter le parallélisme découvert automatiquement.

Accessoirement, elle permet d'améliorer d'autres optimisations, comme l'élimination de code mort, qui peuvent faire disparaître des tests ou des appels de procédure. Un plus grand nombre de boucles satisferont donc nos critères.

1.3.3. Architectures considérées

Notre objectif est d'abord d'apporter une solution, très partielle mais efficace quand elle peut s'appliquer, au problème du partitionnement. Un certain nombre d'architectures nécessitent la résolution d'un second problème, qui interfère avec le partitionnement, le problème de l'affectation des processus aux processeurs et, par voie de conséquence, de la localisation des données relatives à un processus sur une mémoire accessible par le processeur qui l'exécute.

Les architectures avec lesquelles le temps de transfert des données ne dépend pas uniquement de la **nature** de l'émetteur et du récepteur mais aussi de leur position introduisent ce problème de mapping. L'absence de mémoire globale conduit aussi à mettre les données dans telle ou telle mémoire locale et donc à favoriser certains processeurs pour l'exécution de certains processus. Pour éviter d'avoir à résoudre deux problèmes simultanément, nous nous sommes donc limités aux architectures:

- dans lesquelles l'ensemble des processeurs, tous identiques, n'est pas structuré et est donc défini par son cardinal.
- dans lesquelles il existe une mémoire globale susceptible de contenir **toutes** les données nécessaires à l'exécution du programme.

Ces deux conditions ne sont pas suffisantes pour supprimer totalement le problème de localisation. Tout d'abord, les conflits dans le réseau d'interconnexion et les conflits d'accès aux bancs mémoires sont eux aussi sensibles à la localisation des données dans les bancs et des processus sur les processeurs. Ces deux problèmes seront ignorés dans ce travail. D'autre part, les résultats produits par l'exécution d'un processus pourraient être exploités sans le moindre transfert si le processus les utilisant comme données était

exécuté sur le même processeur. Cette source d'optimisation, appelée *problème de l'héritage* dans la suite, est tout simplement négligée.

Ces conditions éliminent les machines MIMD fondées sur un réseau d'interconnexion à liens fixes en forme d'hypercube ou d'arbre, bien que le partitionnement que nous proposons soit une première étape dans l'adaptation de programmmes à ces architectures.

L'adaptation des programmes à une exécution parallèle et à la taille restreinte des mémoires locales est heureusement non contradictoire car la multiplication des tâches conduit tout naturellement à une diminution de leur taille.

Il n'en est pas forcément de même avec les unités vectorielles, auxquelles il va falloir assigner un minimum de parallélisme aux dépens du niveau multiprocesseur. D'autre part, le découpage en tâches peut conduire à définir des vecteurs inexploitables parce que trop courts ou de longueur variable. Il n'est donc ni possible de découpler le problème de la génération de code vectoriel du problème du partitionnement, ni possible d'ignorer le niveau vectoriel à cause du gain qu'il peut apporter.

Enfin, nous ne prendrons en compte que deux niveaux de la hiérarchie mémoire simultanément. Une application itérative de la méthode devrait permettre de gérer trois niveaux et plus. Remarquons que les registres sont utilisés d'une manière trop spécifique pour se voir appliquer directement les résultats valables pour d'autres types de mémoires.

Il faut noter que, du fait du respect des dépendances, aucun problème de cohérence de caches ou de mémoires locales ne devrait théoriquement se poser au cours de l'exécution d'une tâche. Deux tâches qui peuvent s'exécuter en parallèle ne peuvent pas partager une cellule mémoire en écriture. Le problème de cohérence ne resurgit donc qu'aux points de synchronisation, en début et en fin de tâche, tout au moins si la réécriture d'une ligne de cache partiellement modifiée est effectuée sélectivement. Dans la pratique, le maintien de la cohérence entre caches locaux est assuré par un dispositif matériel.

1.3.4. Application automatique de la méthode

Quoique la méthode que nous proposons puisse être appliquée manuellement, il est souhaitable qu'elle débouche sur un algorithme de partitionnement automatique. Nous ne chercherons donc pas à obtenir en priorité des résultats optimaux mais plutôt à développer une technique aussi simple que possible.

C'est aussi pourquoi nous chercherons à obtenir une formule explicite du temps d'exécution global pour dériver de bons paramètres de partitionnement et de restructuration de code. Cette formule est basée sur le modèle extrêmement simplifié de machine, qui a été présenté en 1.1.1.6 et se compose essentiellement de coefficients, comme le nombre de processeurs ou la taille des mémoires, et de drapeux booléens caractérisant, par exemple, un processeur élémentaire comme vectoriel ou une mémoire comme étant un cache. Nous faisons par exemple l'hypothèse que les machines ne présentent pas de conflits

aléatoires sur l'accès à la mémoire globale ou au bus, quoique le problème ait déjà été étudié en général dans[36] et, pour le Cray X-MP, dans[53]. On utilise une pénalisation moyenne constante, correspondant à l'utilisation simultanée de tous les processeurs, qui est appliquée à tous les transferts entre la mémoire lente et la mémoire rapide.

Pour calculer le temps d'exécution, nous supposerons aussi qu'aucun processeur ne reste inactif quand il existe une ou plusieurs tâches dont les données sont disponibles, et qu'il y a toujours de telles tâches quand le degré de parallélisme est suffisant, quoique les primitives de synchronisation réelles, utilisées lors de la génération de code, ne le permettent pas toujours. En effet, ces primitives ne sont généralement pas assez souples pour pouvoir implémenter n'importe quel graphe de tâches. Leur utilisation entraîne l'apparition d'arcs supplémentaires: il y a sursynchronisation. Cela ne peut être une hypothèse acceptable que si les tâches sont assignées dynamiquement aux processeurs et si les pertes dues aux sursynchronisations sont faibles relativement au temps d'exécution total. Mais du coup le temps d'exécution en parallèle est égal au temps d'exécution séquentiel divisé par le nombre de processeurs.

Le temps d'exécution d'une tâche est considéré comme indépendant du contexte. L'exécution d'une tâche ne pourra pas être interrompue par une synchronisation avec une autre tâche. Sa durée sera considérée comme fixe même si un autre processeur utilise les mêmes ressources de la machine, par exemple, un banc mémoire ou un chemin du réseau d'interconnexion. Ces ressources n'apparaissent d'ailleurs pas dans la modélisation. Enfin, le coût d'une synchronisation est indépendant du nombre total de synchronisations à effectuer. Ceci nous permet de rechercher un ordonnancement sur des tâches dont les durées sont **fixées.**

L'automatisation complète n'est pas encore possible, car on ne sait toujours pas évaluer le temps d'exécution d'une séquence de code linéaire et du code de contrôle d'une boucle DO. Ce temps est bien sûr fonction des performances de la machine mais il dépend aussi du compilateur et de l'optimiseur de ce dernier. Par conséquent, ce renseignement devrait d'abord être fourni interactivement ou calculé par *profiling*[64] , quoique le nombre d'opérations flottantes puisse donner une estimation suffisante, vu les autres approximations qui ont été faites, si le générateur de code est efficace. Pendant l'expérience[39] que nous avons menée sur l'Alliant il a cependant été impossible d'utiliser ce critère du nombre d'opérations flottantes vu l'écart qui existait entre les instructions Fortran et le code généré.

1.3.5. Découpage de l'optimisation en étapes indépendantes

Le problème complet est trop complexe du fait du couplage entre la génération de code vectoriel dans une tâche et le découpage en tâche. Dans le cas particulier du produit de matrice, traité manuellement, W. Jalby et U. Meier[41] proposent de découpler les deux problèmes et de chercher une solution qui appartienne à l'intersection du voisinage des deux optima. Pour pouvoir traiter le cas général, nous résoudrons aussi les deux problèmes successivement mais, au lieu de chercher indépendamment une solution valable pour les deux problèmes, nous utiliserons la solution du premier problème comme contrainte

pour la recherche d'une solution du deuxième. Il est alors difficile de vérifier que le résultat obtenu donne un temps d'exécution proche de l'optimum, sauf s'il conduit à la saturation de toutes les ressources de la machine.

La première phase consiste à ordonnancer les calculs d'une tâche pour exploiter l'unité vectorielle ou/et pour minimiser le volume nécessaire pour éviter les défauts de cache. Les particularités de l'unité vectorielle peuvent conduire à favoriser une forme de tâche ou une autre.

Pendant la deuxième phase, la forme des tâches doit être déterminée, ainsi qu'un facteur d'échelle qui va donner le volume des tâches et donc le nombre de tâches. Le volume d'une tâche va être calculé en utilisant **l'heuristique** suivante:

> toutes les données nécessaires à l'exécution d'une tâche se trouvent dans le niveau mémoire le plus rapide dès qu'elles sont utilisées plus d'une fois.

Cette stratégie est facile à implémenter dynamiquement avec un cache. Avec une mémoire locale, gérée statiquement à la compilation, il faut s'appuyer sur un calcul de dépendance pour l'approximer. Quand la mémoire globale n'est pas accessible directement par les processeurs, toutes les données sont en mémoire locale.

Une troisième étape consisterait à optimiser l'ordonnancement des tâches obtenues. Cette étape n'est pas efffectuée car le graphe d'ordonnancement des tâches est suffisamment simple et régulier pour qu'on puisse choisir a priori un ordonnancement raisonnablement efficace pour n'importe quel corps de boucle.

Il n'y a pas à affecter statiquement les tâches aux processeurs, puisque nous avons éliminé les architectures sur lesquelles cette affectation permettrait des gains significatifs, et que nous laissons de côté le problème de l'*héritage* des données utilisées ou produites par un processus par celui qui suit sur le même processeur. Rappelons que nous avons choisi une allocation dynamique des tâches aux processeurs et qu'une tâche s'exécute à la même vitesse sur n'importe quel processeur.

1.4. Notations et définitions

Il n'est pas possible ici de réintroduire en détail, avec des exemples, les bases de la parallélisation et de la vectorisation automatique (et manuelle) des programmes procéduraux. Les bases sont connues depuis longtemps[10] et les travaux de recherche menés dans les années 70 ont conduit à des cours et à des réalisations commerciales dans les années 1980, au moins pour la *vectorisation* des programmes Fortran. En cas de besoin, les cours de A. Lichnewsky[49] et de P. Feautrier[18] contiennent les explications nécessaires en français. En anglais, Kuck[45, 46] et Kennedy[3] ont aussi écrit des textes introductifs au domaine. Néanmoins, le vocabulaire et les définitions ne sont pas encore normalisés et il est utile de préciser dans quels sens ils seront utilisés dans ce travail.

1.4.1. Instruction et exécution d'une instruction - Noeud et supernoeud

La notion d'instruction est familière à tous les programmeurs qui utilisent des langages procéduraux. Vu les restrictions que nous faisons, les instructions d'entrée-sortie, de test ou d'appel de sous-programme ne sont pas utilisées et seules les affectations seront prises en compte.

Nous ne considérerons pas les instructions individuellement car seuls les corps de boucle, séquences d'affectations, nous intéressent.

Les instructions ou les blocs d'instructions sont des objets statiques alors que nous allons nous intéresser au problème de l'exécution dynamique de ces objets. Une instruction située dans une boucle est exécutée plusieurs fois et c'est chaque occurrence de l'instruction qui doit être considérée pour le partitionnement. Dans la suite, une **exécution** d'une instruction ou d'un bloc d'instructions est appelé un *noeud*. Les noeuds sont représentés par les symboles: n , n_1, n_2, ... L'utilisation des occurrences d'instructions et non des instructions est inhabituel dans le domaine de la parallélisation automatique, mais courant dans le domaine des algorithmes et architectures systoliques.

Un ensemble de noeuds est appelé *supernoeud* et noté s , s_1, s_2, ... Ces deux termes sont traduits de la terminologie utilisée au CSRD pour les problèmes de partitionnement. Ils remplacent avantageusement *exécution d'une instruction ou d'une séquence d'instructions* et *ensemble d'exécutions d'une*.... Le supernoeud constitue donc ce que nous avons appelé *tâche* jusqu'ici.

1.4.2. Vecteur d'itération, espace d'itération, domaine d'itération

Chaque noeud généré par un ensemble de boucles parfaitement imbriquées peut être repéré par les valeurs des indices de boucle. Autrement dit, un noeud peut être mis en correspondance avec un élément de l'espace vectoriel $\mathbf{R}^N$, où N est le nombre de boucles englobantes. Seuls les points de coordonnées entières peuvent être mis en correspondance avec un noeud. L'élément de $\mathbf{R}^N$ relatif à un noeud est appelé son *vecteur d'itération*[47, 68] . Les vecteurs d'itération sont notés $\vec{j}$, $\vec{j}_1$, $\cdots$. $\mathbf{R}^N$ est appelé

espace d'itération. Cet espace est totalement ordonné par l'ordre lexicographique qui traduit l'exécution séquentielle des boucles DO du programme initial. Ces boucles sont notées DOSEQ. L'introduction de boucles DO parallèles, appelées DOALL, va réduire l'ordre total à un ordre partiel en interdisant la prise en compte des coordonnées correspondantes lors des comparaisons.

Les bornes inférieures et supérieures des boucles définissent le *domaine d'itération* (Index Set). C'est un polyèdre convexe car nous avons fait l'hypothèse que ces bornes étaient des expressions affines des indices englobants. On parlera de ses sommets, de ses arêtes et de ses faces. Il est représenté par un système d'inégalités triangulaire[47] . Ce type de système ne donne pas des polyèdres dont la forme serait triangulaire mais les inégalités peuvent être mises sous la forme de deux matrices de contraintes, respectivement à coefficients positifs et à coefficients négatifs sur la diagonale, qui sont triangulaires. Les coefficients diagonaux peuvent être ramenés à 1 et à -1.

Dans le cas des boucles DO, la triangularité est due à l'hypothèse sur les expressions des bornes. Réciproquement, il sera facile de générer du code pour parcourir un polyèdre si on peut le mettre sous une forme triangulaire.

Par exemple, les deux boucles DO suivantes:

```
DO I = 1, 100
  DO J = I-1, 2*I-5
```

ont un espace d'itération de dimension 2 et un domaine d'itération défini par les inégalités:

$$\begin{cases} 1 \leq I \leq 100 \\ I-1 \leq J \leq 2*I-5 \end{cases}$$

En regroupant les variables du même côté du signe d'inégalité et en n'utilisant que des inégalités du type inférieur ou égal, les inégalités peuvent être mises sous une forme matricielle:

$$\begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 100 \\ -5 \end{pmatrix} \qquad \begin{pmatrix} -1 & 0 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

Dans la suite, nous appelerons A la matrice des coefficients et $\vec{\alpha}$ le vecteur des constantes. Dans le cas que nous venons de voir, A serait de dimension 4×2 et $\vec{\alpha}$ aurait quatre composantes.

La forme géométrique du polyèdre correspondant à un système triangulaire n'a pas de relation directe avec les triangles. Ces systèmes définissent des segments en dimension 1 et des trapèzes en dimension 2. Pour passer de la dimension N à la dimension $N+1$, il faut construire un cylindre dont la base est définie en dimension N et le tronquer par deux hyperplans quelconques.

On parlera aussi de systèmes *pseudo-triangulaires* quand les deux matrices de contraintes ne sont plus triangulaires mais que les variables et les contraintes peuvent être **totalement** ordonnées de telle sorte que:

- une contrainte est plus *grande* qu'une autre si elle contient toutes les variables de l'autre (une contrainte contient une variable si le coefficient de la variable est non nul); cet ordre total sur les contraintes induit un ordre total sur les variables;
- pour toute variable, il existe au moins une contrainte dont elle est la plus grande variable et où elle est affectée d'un coefficient positif, et une autre contrainte où elle est affectée d'un coefficient négatif; chaque variable est donc bornée inférieurement et supérieurement en fonction des variables plus *grandes*.

Il est aussi possible de générer du code à partir d'un système pseudo-triangulaire en utilisant les opérateurs min et max.

Par exemple, le système:

$$\begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 100 \\ -5 \end{pmatrix} \quad \begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 2 & -1 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} -1 \\ 1 \\ 3 \end{pmatrix}$$

peut être parcouru par les deux boucles:

```
DO I = 1, 100
  DO J = MAX(I-1,2*I-3), 2*I-5
```

tandis que le système

$$\begin{pmatrix} 1 & 1 \\ -2 & 1 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 100 \\ -5 \end{pmatrix} \quad \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ 2 & -1 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} -1 \\ 1 \\ 3 \end{pmatrix}$$

n'est pas pseudo triangulaire et ne permet pas de générer directement du code de contrôle.

Les faces du domaine d'itération sont aussi appelées ses frontières. Ce terme ne sera utilisé que pour la génération de code, en dimension 2.

Les vecteurs d'itération seront ordonnés par un *vecteur de direction d'itération*, forme linéaire appartenant au dual de $\mathbf{R}^N$. Le vecteur de direction d'itération est généralement noté $\vec{h}$, comme toutes les formes linéaires que nous utiliserons. La valeur prise par la forme linéaire induit un préordre total sur les noeuds. Deux noeuds qui prennent la même valeur peuvent être exécutés simultanément. L'ordre lexicographique ne peut pas être représenté par une forme linéaire sur l'ensemble de l'espace d'itération mais il peut l'être sur le domaine d'itération de la même manière qu'un tableau multidimensionnel peut être linéarisé. La famille des ordonnancements définis par un vecteur de direction d'itération ne contient pas tous les ordonnancements générés par des imbrications de boucles DOALL et DOSEQ, mais seulement ceux pour lesquels les DOALL sont imbriqués dans les DOSEQ.

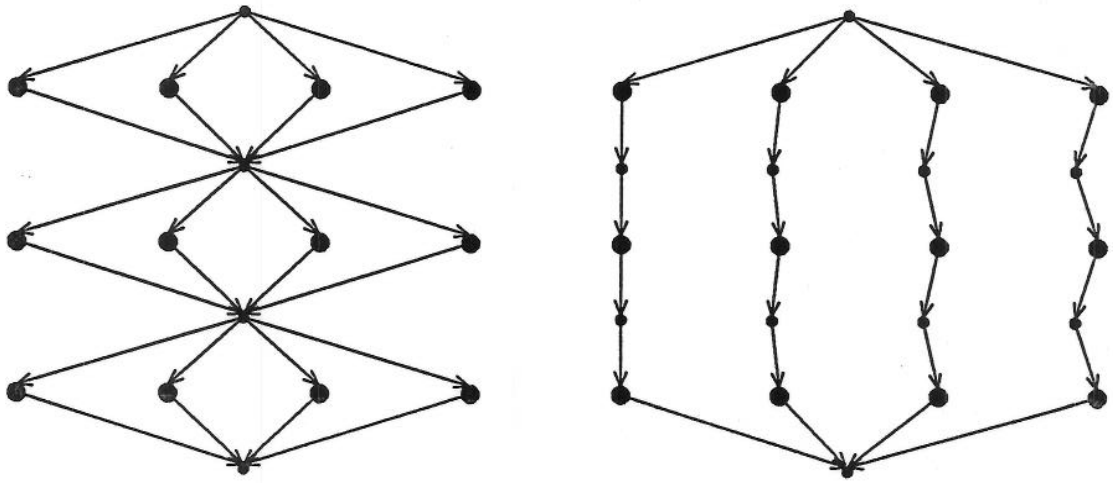
La séquence DOALL i / DOSEQ j génère l'ordre défini par un prédicat moins fort:

$$t_{i,j} < t_{i',j'} \quad <==> \quad j < j' \text{ et } i = i'$$

que la séquence DOSEQ j / DOALL i:

$$t_{i,j} < t_{i',j'} \quad <==> \quad j < j'$$

Le deuxième ordre est plus fort que le premier, comme on le voit aussi figure 1.4.2 où les graphes des deux relations d'ordre sont partiellement développés. La restriction aux ordonnancements de type DOSEQ / DOALL entre supernoeuds est expliquée dans le paragraphe 3.1. Cette même restriction est justifiée entre noeuds, à l'intérieur d'un supernoeud, parce que le seul type de parallélisme que nous prenons en compte dans un processeur élémentaire est le vectoriel. Le parallélisme supplémentaire exhibé par la séquence DOALL / DOSEQ ne pourrait pas être utilisé.



Les ordonnancements DOSEQ/DOALL et DOALL/DOSEQ

- Figure 1.4.2 -

Les indices de boucle successifs sont conventionnellement notés I, J, K, etc... en partant de la boucle externe pour aller vers la boucle la plus interne. La base initiale de l'espace d'itération (de dimension N) est la base $B = (\vec{e}_1, \vec{e}_2, \vec{e}_3, ... \vec{e}_N)$, où le vecteur $\vec{e}_1$ correspond à l'indice I, $\vec{e}_2$ à l'indice J, etc...

Le produit scalaire est noté simplement $\vec{e}_1 \, \vec{j}$ s'il n'y a pas d'ambiguité et $< \vec{e}_1 + \vec{e}_2 \, . \, \vec{j} >$ dans le cas contraire.

1.4.3. Dépendances

Le calcul des dépendances existant entre deux instructions est basé sur l'ensemble des cellules mémoires qu'elles lisent ou écrivent[10] . Ces cellules mémoires sont définies par les *références mémoires* qui apparaissent dans les instructions. L'affectation:

$$A(I,J) = A(I,J) + B(I,K)*C(K,J)$$

contient quatre références mémoires, deux relatives au tableau A, une au tableau B et une au tableau C. La première est une référence de **définition**, les trois autres des références **d'utilisation**. Les différentes expressions d'indice sont par hypothèse (§ 1.2.1) des expressions affines des indices des boucles englobantes, basées sur des formes linéaires de l'espace d'itération notées $\vec{\alpha}_1,\vec{\alpha}_2$, ... La référence:

$$A(I+2*J,K+1)$$

sera souvent étudiée par le biais de ses deux formes linéaires:

$$\vec{\alpha}_1 = (1, 2, 0) \qquad \vec{\alpha}_2 = (0, 0, 1)$$

Les dépendances entre deux instructions S1 et S2 sont provoquées par deux références, l'une dans S1 et l'autre dans S2, qui adressent la même case mémoire. En l'absence de pointeurs, d'instructions EQUIVALENCE, et d'appels de procédures créant un aliasing entre deux symboles, deux références ne peuvent induire une dépendance que si elles portent sur le même tableau.

On distingue différents types de dépendance suivant le type des références. Si la référence R1 dans S1 est une référence de définition et la référence R2 dans S2, une référence d'utilisation, il existe une datadépendance entre S1 et S2 si S1 vient avant S2, et une antidépendance si S2 précède S1. Si les deux références sont des définitions, elles créent une outputdépendance. La situation est plus compliquée quand les deux instructions S1 et S2 se trouvent dans un corps de boucle.

Dans la suite, nous ne nous intéresserons pas à ces dépendances entre instructions. Comme nous sommes passé des instructions aux noeuds, nous allons passer des dépendances classiques à des dépendances entre noeuds.

Pour étudier la réutilisation des données présentes dans le niveau mémoire le plus rapide, nous avons besoin d'une nouvelle sorte de dépendance que nous avons appelée *autodépendance*. Une autodépendance est créée par une référence unique qui va être utilisée plusieurs fois parce que l'instruction qui la contient se trouve dans une boucle. Par exemple, le produit de matrices présente des autodépendances sur les tableaux A, B et C:

```
DO I=1,N
  DO J=1,N
    A(I,J)=0.
    DO K=1,N
        A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

Ces autodépendances peuvent être matérialisées en transformant le programme initial qui devient:

```
DO I=1,N
  DO J=1,N
    A(I,J,0)=0.
    DO K=1,N
        A(I,J,K)=A(I,J,K-1)+B(I,K,J)*C(K,J,I)
        B(I,K,J)=B(I,K,J-1)
        C(K,J,I)=C(K,J,I-1)
```

Les autodépendances sont matérialisées par des dépendances portant sur des affectations identités. Toutes les instantiations de valeurs sont nommées, comme dans un algorithme systolique où il faut prévoir le passage des valeurs d'un processeur à ses voisins[50, 51] .

Les autodépendances vont nous permettre de nous rendre compte que, si la boucle interne est la boucle J, B(I,K) peut être stocké en registre, ou bien que, si I est la boucle externe, il suffit de garder une ligne de B en cache.

Les autodépendances sont un sous-ensemble des *Input Dependence* de Kuck[46] qui sont définies par ce dernier comme un conflit entre **deux** références contenues dans la partie droite d'une assignation. Quand il y a uniquement inputdépendance, les données peuvent n'être réutilisées que deux ou trois fois. Avec une autodépendance, le nombre de réutilisation est nécessairement donné par les bornes d'une boucle et est donc potentiellement intéressant.

Enfin, les controldépendances sont tout simplement ignorées car nous allons dans un premier temps supposer que les corps de boucle ne contiennent pas de tests. De toute façon, Allen et al. ont montré dans [4] que les dépendances de ce type pouvaient être transformées en dépendances sur les données.

1.4.4. Vecteurs, graphe et cône de dépendance

Un vecteur de dépendance $\vec{d}$ est défini à partir des deux vecteurs d'itération $\vec{j_u}$ et $\vec{j_d}$ des noeuds n_u et n_d qui sont en dépendance. Le vecteur $\vec{j_u}$ (u comme utilisation) est associé à une référence d'utilisation, et $\vec{j_d}$ (d comme définition) à une référence de définition. Un vecteur de datadépendance est défini par:

$$\vec{d} = \vec{j}_u - \vec{j}_d$$

et un vecteur d'antidépendance l'est par:

$$\vec{d} = \vec{j}_d - \vec{j}_u$$

La nature de la dépendance est fixée par la condition "$\vec{d}$ lexico-positif" à cause de la sémantique des boucles DO. En général les vecteurs d'outputdépendance sont sans intérêt car l'information qu'ils apportent peut être déduite d'une combinaison de datadépendance et d'antidépendance quand les valeurs produites sont bien utilisées et non seulement écrasées.

A chaque noeud est associé un ensemble de vecteurs de dépendance, ensemble fini quand on ne s'intéresse pas aux dépendances composites, formées d'une combinaison de dépendances plus élémentaires. Dans certains cas, cet ensemble est constant sur l'espace d'itération et ne dépend donc pas du noeud considéré. Il s'agit alors de *récurrences uniformes*[43] dont l'intérêt est connu depuis longtemps pour la méthode des vagues et pour les algorithmes systoliques. Les vecteurs de dépendance, qui sont appelés vecteurs de distance dans la littérature classique sur la parallélisation, sont notés $\vec{d}$, $\vec{d}_1$, $\vec{d}_2$, ... et l'ensemble de ces vecteurs D . D est vu comme une matrice quand cela simplifie les notations.

Quand l'ensemble des vecteurs de dépendance n'est pas constant, on peut l'approximer par un polyèdre qui est l'enveloppe convexe des vecteurs de dépendance générés en chaque point du domaine d'itération. Le test de dépendance proposé dans[63] permet de la calculer automatiquement, moyennant l'introduction de variables supplémentaires. L'ordre induit entre les noeuds par les dépendances peut alors être approximé par la fermeture transitive des dépendances, ce qui transforme le polyèdre des dépendances en un cône dont les rayons sont les rayons et sommets du polyèdre précédent et dont le sommet est à l'origine. On peut calculer rayons et sommets en réécrivant les équations et inéquations qui définissent l'ensemble des dépendances sous forme d'un système générateur. Plusieurs algorithmes sont décrits dans la thèse de 3ème cycle d'Halbwachs[30] . Un cône de dépendance sera noté R où R est l'ensemble des rayons. Le sommet du cône, toujours à l'origine, n'est pas rappelé. Quand cela simplifie les notations, R sera vu comme une matrice formée par les coordonnées des rayons dans la base courante (usuellement B).

Ceci permet, par exemple, de remplacer l'ensemble infini D par l'ensemble fini R :

$$D = \left\{ \binom{1}{0}, \binom{2}{0}, ..., \binom{n}{0}, \cdots \right\} \qquad R = \left\{ \binom{1}{0} \right\}$$

et l'ensemble doublement infini D' , que l'on trouve dans l'algorithme de programmation dynamique[28] , par R' :

$$D' = \left\{ \binom{1}{0}, \binom{0}{1}, \binom{1}{1}, \binom{2}{0}, ..., \binom{n}{m}, \cdots \right\} \qquad R' = \left\{ \binom{1}{0}, \binom{0}{1} \right\}$$

Cette notion de cône de dépendance est une notion nouvelle qui permet de ne pas imposer la contrainte de récurrence uniforme. Ce résultat est obtenu au prix d'un renforcement des dépendances, dû à leur approximation par des rayons d'un polyèdre dont la **norme** est choisie aussi petite que possible, sous la condition que les coordonnées soient entières. Prenons le cas artificiel suivant:

```
DO I = 1, N
  DO J = 1, M
    A(I,J) = A(I,J) + A(I/2,J/2)
```

Récurrence non uniforme

Les vecteurs de dépendance prennent l'une des formes suivantes:

$$\begin{pmatrix} \frac{i}{2} \\ \frac{j}{2} \end{pmatrix} \quad \begin{pmatrix} \frac{i}{2}+1 \\ \frac{j}{2} \end{pmatrix} \quad \begin{pmatrix} \frac{i}{2} \\ \frac{j}{2}+1 \end{pmatrix} \quad \begin{pmatrix} \frac{i}{2}+1 \\ \frac{j}{2}+1 \end{pmatrix}$$

Comme i et j sont positifs, ces vecteurs sont toujours des combinaisons linéaires positives des deux vecteurs de base. Autrement dit, ils se trouvent tous dans le cône que constitue le premier quadrant de B et D peut être réduit à R :

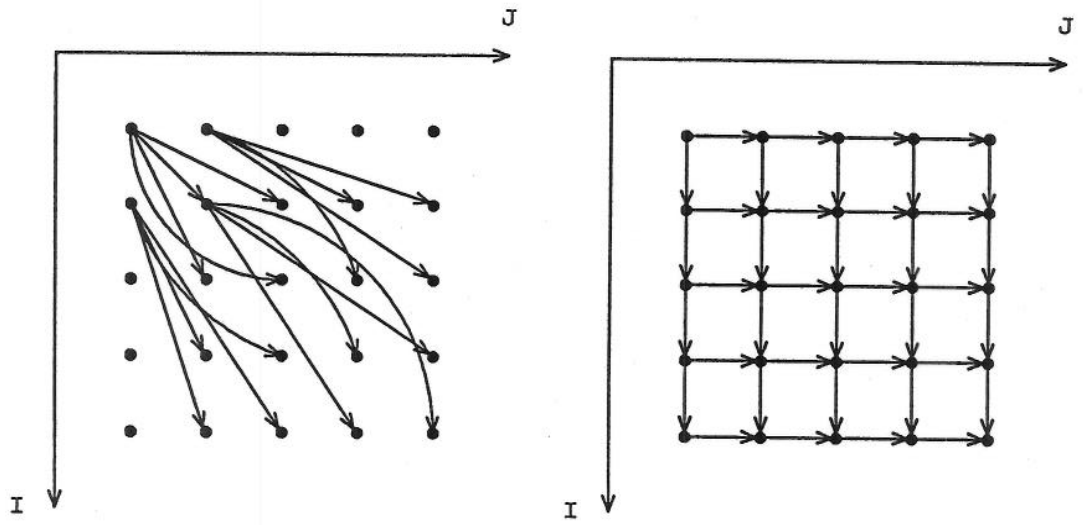
$$R = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

L'influence de cette réduction sur le graphe de dépendance et sur l'ordre partiel des noeuds est montrée figure 1.4.4.

Ceci permet, entre autres, d'appliquer la méthode hyperplane à cet exemple, mais, en prenant en compte le sommet du polyèdre des dépendances, qui est ici (1,1), on peut aussi montrer que la boucle sur I ou la boucle sur J peut être exécutée en parallèle[40] .

Dans le cas où D existe, les algorithmes de passage au système générateur permettent aussi d'éliminer les vecteurs de dépendance redondants. Par exemple, l'ensemble de vecteurs de dépendance D est réduit à D' :

$$D = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\} \qquad D' = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$



Ordre induit par les dépendances et son approximation par le cône

- Figure 1.4.4 -

Dans la suite, il n'y aura pas de raisons† de distinguer rayons et vecteurs de dépendance car ils ne servent tous les deux qu'à définir un ordre partiel sur les noeuds du domaine d'itération et par extension sur tout l'espace d'itération (l'ordre induit par le cône de dépendance contient l'ordre induit par les dépendances dont on a calculé le cône). C'est aussi la raison pour laquelle nous ne faisons pas de distinction entre les différents types de dépendances. Toutes traduisent une contrainte d'ordonnancement et ce n'est que si nous voulions éliminer des dépendances que leur nature pourrait importer.

Les vecteurs de dépendance définissent aussi un graphe de dépendance sur l'espace et le domaine d'itération, vus comme des ensembles discrets, qui est donc différent du graphe de dépendance usuel qui est construit, lui, sur l'ensemble des instructions. Notre graphe est acyclique car il existe au moins une exécution possible, celle qui est définie par le programme Fortran dont nous voulons optimiser l'exécution. C'est une différence majeure par rapport aux méthodes qui s'appuient directement sur des équations récurrentes uniformes[55, 52] .

†Sauf dans le paragraphe 2.2.3 sur la connexité du graphe de dépendance.

1.4.5. Rappel sur les calculs en nombres entiers

La division entière et le modulo sont bien définis sur les entiers positifs. Malheureusement, les définitions se multiplient quand il faut étendre ces deux opérateurs aux entiers relatifs. Comme nous allons utiliser la division et le modulo pour partitionner l'espace d'itération, il faut commencer par préciser le comportement que nous souhaitons au voisinage de zéro. Soit K le facteur de partitionnement, qui est heureusement toujours positif:

Définition: i et j appartiennent à la même classe si $i/K = j/K$.

Il faut donc que $-1/K = -1$ et que $mod(-1,K) = K-1$. Ce ne sont pas apparemment les conventions usuelles, mais c'est nécessaire pour que la classe 0 ne soit pas plus grande que les autres.

Il nous faudra aussi manipuler des inégalités en nombres entiers et réduire des coefficients à 1 pour faire la génération de code. Quatre cas sont possibles (α est un entier positif non nul et E une expression quelconque):

$$\begin{cases} \alpha I \leq E & \rightarrow \quad I \leq \dfrac{E}{\alpha} \\ -\alpha I \leq E & \rightarrow \quad I \geq \dfrac{-E+\alpha-1}{\alpha} \\ \dfrac{I}{\alpha} \leq E & \rightarrow \quad I \leq \alpha E + \alpha - 1 \\ -\dfrac{I}{\alpha} \leq E & \rightarrow \quad I \geq -\alpha E \end{cases}$$

Ces règles de calculs inhabituelles doivent être appliquées dans tous les algorithmes sur les polyèdres convexes (projection, élimination de redondance, inclusion, faisabilité, enveloppe convexe) que nous utiliserons. Le lecteur qui souhaite vérifier la validité des calculs des chapitres 3 et 7, consacrés à la génération de code et à un exemple, devra s'en souvenir.

1.5. Conclusion

Ce chapitre initial nous a d'abord permis de faire assez longuement le tour des solutions architecturales qui sont actuellement proposées pour résoudre le défi permanent du calcul scientifique et de ses masses toujours croissantes de calculs et de données. Une modélisation sommaire de ces architectures, prenant en compte l'organisation de la hiérarchie mémoire et les différents niveaux de parallélisme offerts par le processeur, permet d'estimer l'influence qu'auront l'ordonnancement des calculs et la localité des références aux données.

Nous avons ensuite montré sur un exemple comment on pouvait passer d'un programme Fortran séquentiel à un programme à la fois doublement parallèle et susceptible d'utiliser correctement une hiérarchie mémoire.

Il n'était pas possible de fournir une solution valable pour n'importe quel programme et n'importe quelle machine, susceptible d'être automatisée et, qui plus est, optimale. Nous avons donc dû simultanément restreindre les portions de code considérées aux boucles DO parfaitement imbriquées, ne prendre en compte que deux niveaux de parallélisme et deux niveaux de la hiérarchie mémoire des machines cibles, envisager une application semi-automatique, ou même manuelle, de notre méthode de partitionnement et enfin découper le problème d'optimisation globale en étapes successives optimisées indépendamment.

Une fois notre objectif fixé, nous avons explicité les notations utilisées dans les chapitres suivants et donné des définitions précises aux termes propres au domaine de la parallélisation de programmes car la relative jeunesse de ce dernier n'a pas encore permis de définir un vocabulaire standard.

Nous disposons donc maintenant des éléments nécessaires pour passer à l'étape suivante, l'étude des partitionnements.

References

1. W. A. Abu-Sufah, "Improving the Performance of Virtual Memory Computers," Ph.D. Thesis, DCS Report No. UIUCDCS-R-78-945, University of Illinois at Urbana-Champaign (Jan. 1979).

2. D. P. Agrawal, V. K. Janakiram, and G. C. Pathak, "Evaluating the Performance of Multicomputer Configurations," *IEEE Computer* **19**(5), pp.23-37 (May 1986).

3. J. R. Allen and K. Kennedy, "PFC: A Program To Convert Fortran To Parallel Fortran," *Elaboratori paralleli e calcolo scientifico*, pp.1-63 (March 1982).

4. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," pp. 177-189 in *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages*, Austin (Jan. 1983).

5. J. R. Allen and K. Kennedy, "Automatic Loop Interchange," *SIGPLAN'84 Symposium on Compiler Construction*, pp.233-246 (June 1984).

6. R. Allen and K. Kennedy, "Programming Environments for Supercomputers," COMP TR85-18, Rice Univ., Houston (1985).

7. G. Amdahl, "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conf. Proc. 30* (1967).

8. Z. Ammarguellat, "Les transformations de programme," rapport de DEA, Ecole des Mines de Paris C.A.I., Fontainebleau (1985).

9. G. M. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," PhD dissertation, Carnegie Mellon Univ. (Apr. 1978).

10. A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers* **15**(5), pp.757-763 (Oct. 1966).

11. W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," Report, IBM T.J. Watson Research Center, Yorktown Heights, N.Y. 10598 (July 1985).

12. D.A. Calahan, "Task Granularity Studies on a Many-Processor CRAY X-MP," *Parallel Computing* **2**(2), pp.109-118, North-Holland (1985).

13. D. Callahan, K. D. Cooper, K. Kennedy, and L.Torczon, "Interprocedural Constant Propagation," *Sigplan Notices* **21**(7), pp.152-161, Proceedings of the SIGPLAN '86 Symposium on Compiler Construction (1986).

14. A. E. Charlesworth and J. L. Gustafson, "Introducing Replicated VLSI to Supercomputing: the FPS-164/MAX Scientific Computer," *IEEE Computer* **19**(3), pp.10-23 (March 1986).

15. R. S. Clark and T. L. Wilson, "Vector System Performance of the IBM 3090," *IBM Systems Journal* **25**(1), pp.63-82 (1986).

16. P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages* (1978).

17. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge (1985).

18. P. Feautrier, "Parallélisation et vectorisation automatique," Cours de DEA, Univ. Paris VI - MASI (Nov. 1985).

19. J. A. Fisher and J. J. O'Donnell, "VLIW Machines: Multiprocessors We Can Actually Program," pp. 299-305 in *Spring Compcon 84*, Computer Society Press, San Francisco (1984).

20. M. J. Flynn, "Some Computer Organizations And Their Effectiveness," *IEEE Transactions on Computers* **21**(9), pp.948-960 (Sept. 1972).

21. C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," *IEEE Transactions on Computers* **21**(12), pp.1411-1415 (Dec. 1972).

22. D. D. Gajski, D. H. Lawrie, D. J. Kuck, and A. H. Sameh, "CEDAR," pp. 306-309 in *Spring Compcon 84*, Computer Society Press, San Francisco (1984).

23. E. F. Gehringer, A. K. Jones, and Z. Z. Segall, "The Cm* Testbed," *IEEE Computer* **15**(10), pp.40-53 (Oct. 1982).

24. J. R. Goodman and H. C. Young, "Comments on a Massive Memory Machine," *IEEE Transactions on Computers* **35**(10), pp.907-910 (Oct. 1986).

25. A. Gottlieb, R. Grishman, C. Kruskal, K. MacAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer. Designing a MIMD, Shared-Memory Parallel Machine," *Proc. of the 9th Symp. on Computer Architecture* (1982).

26. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers* **32**(2), pp.175-189 (Feb. 1983).

27. R. Gottlieb, K. Kimball, T. Jaskiewicz, and R. Swift, "A New Way To Speed Up a Supercomputer," *Electronics* **58**(30), pp.56-58 (July 1985).

28. C. Guerra and R. Melhem, "Synthesizing Non-Uniform Systolic Design," *Int'l Conference on Parallel Processing*, pp.765-772 (Aug. 1986).

29. J. Gustafson, S. Hawkinson, and K. Scott, "The Architecture of a Homogeneous Vector Supercomputer," *Int'l Conference on Parallel Processing*, pp.653-660 (Aug. 1986).

30. N. Halbwachs, "Détermination automatique de relations linéaires par les variables d'un programme," Thèse 3ème cycle, I.N.P., Grenoble (1979).

31. J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "A Microprocessor-based Hypercube Supercomputer," *IEEE Micro* **6**(5), pp.6-17 (Oct. 1986).

32. J. P. Hayes, T. N. Mudge, and Q. F. Stout, "Architecture of a Hypercube Supercomputer," *Int'l Conference on Parallel Processing*, pp.653-660 (Aug. 1986).

33. M. Hill, "Design Decisions in SPUR," *IEEE Computer* **19**(10), pp.8-22 (Nov. 1986).

34. A. S. Hoagland, "Information Storage Technology. A Look at the Future," *IEEE Computer* **18**(7), pp.60-67 (July 1985).

35. R. W. Hockney and C. R. Jesshope, *Parallel Computers - Architecture, Programming and Algorithms*, Adam Hilger Ltd., Bristol (1981).

36. M.A. Holliday and M. K. Vernon, "Exact Performance Estimates for Multiprocessor Memory and Bus Interference," Technical Report 594, Univ. of Wisconsin-Madison (May 1985).

37. C.A. Huson, "An In-Line Subroutine Expander for Parafrase," Report No UIUCDCS-R-82-1118, University of Illinois (1982). M.S. Thesis

38. Intel, "Vector Processing Boosts Hypercube's Performance," *Electronics* **59**(15), pp.30-31 (April 1986).

39. F. Irigoin and R. Triolet, "Supernodes and Alliant FX/8 Minisupercomputer," Tech. Rep. ENSMP-CAI-86-081, Ecole des Mines de Paris (Aug. 1986).

40. F. Irigoin and R. Triolet, "Direction Vector Versus Dependence Cone," ENSMP-CAI-E8x, Ecole des Mines de Paris, Fontainebleau (1987).

41. W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy," Technical Report CSRD, University of Illinois at Urbana-Champaign, Urbana-Champaign (Feb. 1986).

42. P. Jouvelot, "Parallélisation sémantique: une approche dénotationnelle non-standard pour la parallélisation de programmes impératifs," Thèse de Doctorat d'Université, rapport n° 174, Univ. Pierre et Marie Curie, Paris (Déc. 1986).

43. R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM* **14**(3), pp.563-590 (July 1967).

44. M. Y. Kim, "Synchronized Disk Interleaving," *IEEE Transactions on Computers* **35**(11), pp.978-988 (Nov. 1986).

45. D. J. Kuck, *The Structure of Computers and Computations,* John Wiley & Sons, New York (1978).

46. D. J. Kuck, "High-Speed Multiprocessors and Their Compilers," pp. 193-224 in *Parallel Processing Systems,* ed. D. J. Evans, Cambridge University Press (1982).

47. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

48. J. L. Larson, "Multitasking on the Cray X-MP-2 Multiprocessor," *IEEE Computer* **17**(7), pp.62-69 (July 1984).

49. A. Lichnewsky and F. Thomasset, "Techniques de Base sur l'Exploitation Automatique du Parallélisme dans les Programmes," *Calcul Parallèle à usage scientifique* (Oct. 1985).

50. W. L. Miranker and A. Winkler, "Spacetime Representations of Systolic Computational Structures," Research Report RC 9775, IBM (Dec. 1982).

51. D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays.," *Proceedings of the IEEE* **71**(1), pp.113-120 (Jan. 1983).

52. C. Mongenet, "Une méthode de conception d'algorithmes systoliques. Résultats théoriques et réalisation.," Thèse de Doctorat, INPL, Nancy (1985).

53. W. Oed and O. Lange, "Modelling, Measurement, and Simulation of Memory Interference in the CRAY X-MP," *Parallel Computing* **3**(4), pp.343-358 (Oct. 1986).

54. J.-K. Peir and D. D. Gajski, "Toward Computer-Aided Programming for Multiprocessors," *Int. Workshop on Parallel Algorithms and Architectures,* pp.191-202 (April 1986).

55. P. Quinton, "The Systematic Design of Systolic Arrays," Rapport 193, IRISA, Rennes (Avril 1983).

56. L. Raskin, "Performance Evaluation of Multiple Processor Systems," PhD dissertation, Carnegie Mellon (Aug. 1978).

57. R. G. Scarborough and H. G. Kolsky, "A Vectorizing Fortran Compiler," *IBM Journal of Research and Development* **30**(2), pp.163-171 (March 1986).

58. K. Schwan and A. K. Jones, "Specifying Resource Allocation for the Cm* Multiprocessor," *IEEE Software* **3**(3) (May 1986).

59. C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* **28**(1), pp.22-33 (1985).

60. G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers* **19**(10), pp.889-895 (Oct. 1970).

61. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys* **14**(1), pp.93-143 (March 1982).

62. R. Triolet, "Contribution à la parallélisation automatique de programmes," Thèse de Docteur-Ingénieur, Université Paris VI (Déc. 1984).

63. R. Triolet, F. Irigoin, and P. Feautrier, "Direct Parallelization of CALL Statements," *SIGPLAN'86 Symposium on Compiler Construction* (June 1986).

64. R. Triolet, "Programming Environments for Parallel Machines," *Séminaire International sur les Grands Calculateurs Scientifiques*, pp.109-130 (1987).

65. S. G. Tucker, "The IBM 3090 System: An Overview," *IBM Systems Journal* **25**(1), pp.4-19 (1986).

66. A. Veidenbaum, "Compiler Optimizations and Architecture Design Issues for Multiprocessors," PhD Thesis, University of Illinois at Urbana-Champaign (1985).

67. F. Ware and G. Lee, "How Weitek's chips run FORTRAN at 25 Megaflops," *Electronics*, pp.69-71 (Oct. 30, 1986).

68. M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD. Thesis, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign (1982).

69. X, "ETA 10 System Overview," PUB-1006 A, ETA Systems (Feb. 1986).

70. C.-Q. Zhu and P.-C. Yew, "A Synchronization Scheme and Its Applications for Large Multiprocessor Systems," *Fourth Int'l. Conf. on Distributed Computing Systems* (1985).

2. PARTITIONNEMENT

Il faut tout d'abord définir la notion de tâche (i.e. de supernoeuds) en fonctions des objectifs que nous voulons atteindre. Premièrement, nous souhaitons séparer les synchronisations des calculs pour rester dans le cadre usuel de la recherche opérationnelle ou de la modélisation par réseau de Petri. Nos tâches élémentaires ne contiennent donc pas d'ordres de synchronisation et toute tâche prête peut s'exécuter sans interruption jusqu'à sa terminaison. Cette approche ne limite pas la généralité du schéma de synchronisation et si nos supernoeuds s'avéraient trop petits pour constituer des tâches au sens du système d'exploitation de la machine cible, leur agglomération avec quelques primitives de synchronisation relèverait de la génération de code.

Le deuxième objectif est d'assurer à terme une génération de code automatique. Cela a été, et reste pour beaucoup, un des points faibles de la méthode: la génération est trop compliquée et elle nécessite plus d'information que n'en fournissent les graphes de dépendance usuels. Nous avons donc cherché à la simplifier au maximum en n'ayant qu'un seul type de supernoeud, éventuellement intersecté par une frontière du domaine d'itération.

Enfin, la troisième contrainte est celle de partitionnement exact. Bien que cela ait été souvent proposé sur les machines où les communications sont coûteuses, nous n'envisagerons pas de recalculer des valeurs plutôt que des les transférer. Cela revient à demander que les supernoeuds constituent un pavage du domaine, ou tout au moins de l'espace, d'itération.

Partant de ces conditions, nous avons tout d'abord essayé de définir l'ensemble des partitionnements valides sur un graphe d'exécution quelconque. Mais aucune condition satisfaisante n'a pu être trouvée pour garantir l'absence d'étreinte fatale (connexité, plus grand et plus petit élément). De plus les pavages des espaces de dimension supérieure à 2 sont rares[8] .

Après avoir développé ces trois contraintes et en avoir étudié les conséquences, nous présenterons quelques propriétés comme la connexité du graphe ou la non-dégénerescence du cône de dépendance qu'il faut vérifier avant d'appliquer notre méthode, et des moyens de les garantir quand elles ne sont pas vérifiées naturellement.

Ensuite nous étudierons le partitionnement par hyperplans qui est au coeur de notre travail, ses conditions de validité et ses inconvénients. Il faudra en particulier définir l'ensemble des hyperplans pouvant servir à partitionner, et caractériser l'ensemble des partitionnements valides.

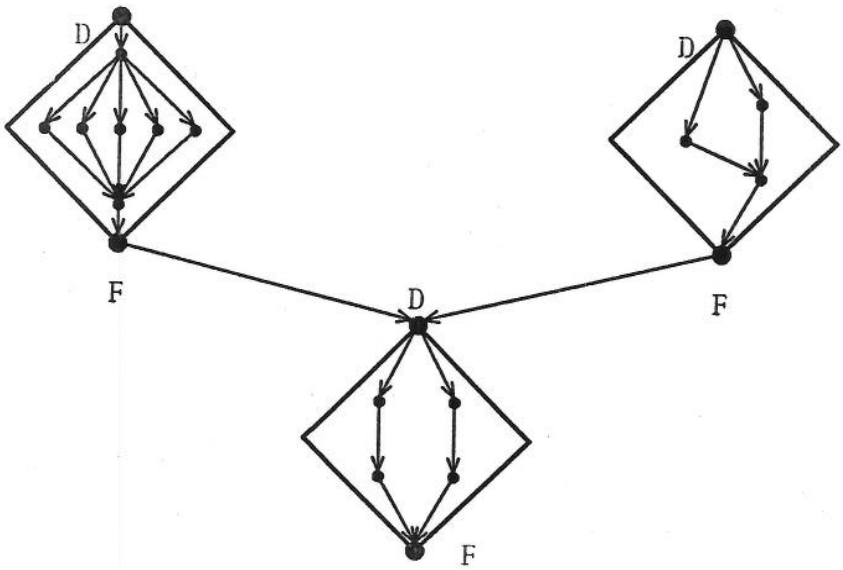
Historiquement, le découpage par hyperplan s'est imposé comme la forme duale d'un partitionnement par agglomération consistant à considérer que tous les noeuds appartenant à une maille de base sont membres du même supernoeud. On utilise alors une base de partitionnement qui permet d'effectuer une génération de code assez simple alors que l'approche hyperplane est plus pratique pour l'optimisation.

2.1. Contraintes de partitionnement.

Le choix d'une technique de partitionnement a été guidé par trois contraintes: synchronisation externe, unicité de forme et partitionnement exact.

2.1.1. Synchronisation externe

La première contrainte, et la plus importante, est l'absence de synchronisation entre supernoeuds pendant l'exécution d'un supernoeud. Les supernoeuds se comportent comme des tâches ininterruptibles, avec un début D et une fin F, et leur graphe de synchronisation est un simple PERT. Les synchronisations internes d'un supernoeud sont totalement encapsulées (cf. figure 2.1.1).



Encapsulation des synchronisations internes

- Figure 2.1.1 -

Toutes les valeurs nécessaires à l'exécution complète doivent être déjà disponibles au démarrage et avoir été produites par des supernoeuds terminés. Les valeurs produites par un supernoeud ne sont mises à la disposition des autres supernoeuds qu'une fois son exécution terminée.

Ce choix fondamental découle de plusieurs observations:

- en dernière analyse, on trouve toujours des séquences d'instructions avec un seul point d'entrée et un seul point de sortie; aucune généralité n'est perdue au niveau du modèle;

- le calcul du temps d'exécution d'un supernoeud est indépendant, moyennant des hypothèses sur la structure de la machine, de l'activité environnante; l'instant terminal est égal à l'instant initial plus le temps d'exécution;
- la localité de la séquence des accès effectués par les processeurs n'est pas détruite par les synchronisations comme dans l'exemple donné en 1.2.5.
- enfin, rien n'empêche de regrouper à la génération de code un ensemble de supernoeuds en une tâche unique avec synchronisation si les overheads de lancement ne permettent pas de faire correspondre les notions de tâche et de supernoeud.

En conséquence les supernoeuds ne peuvent pas être constitués de n'importe quels ensembles de noeuds car des étreintes fatales se produiraient nécessairement. Dans l'exemple 1.2.5 les deux supernoeuds devraient attendre leur terminaison réciproque avant de démarrer.

L'absence de possiblité d'étreinte fatale est garantie s'il existe un ordre partiel sur les supernoeuds respectant l'ordre partiel sur les noeuds qu'induisent les dépendances. Plus précisément l'ordre partiel induit sur les noeuds par l'ordre partiel sur les supernoeuds doit être compatible avec l'ordre partiel induit par les dépendances.

Le problème peut être vu en partant des noeuds et d'une relation d'équivalence Φ caractérisant les supernoeuds:

$$\Phi(n_1,n_2) \equiv n_1 \; et \; n_2 \; appartiennent \; au \; meme \; supernoeud$$

Il faut alors que Φ induise un ordre partiel sur les classes d'équivalence.

D'un point de vue pratique, la condition précédente s'énonce de la manière suivante:

$$\forall n_1, n_1' \in s_1 \quad \forall n_2, n_2' \in s_2 \qquad (1)$$

$$n_1 < n_2 \implies n_1' < n_2' \text{ ou } n_1' <> n_2'$$

Définition: Tout partitionnement vérifiant cette condition est appelé partitionnement valide.

Dans la suite, les mêmes symboles $<$ et $>$ seront utilisés pour les deux ordres partiels. Le groupement $<>$ signifie que les deux arguments ne sont pas comparables.

2.1.2. Unicité de forme.

La deuxième contrainte que nous nous sommes imposée est l'unicité de la forme des supernoeuds: deux supernoeuds peuvent être mis en correspondance totale par translation. Ce sont surtout des raisons pratiques qui ont guidé ce choix:

- il n'y a qu'un seul temps de calcul de supernoeud à estimer;
- les supernoeuds ont des temps d'exécution égaux, moyennant les hypothèses faites sur le contenu du corps de boucle; le calcul du temps d'exécution séquentielle de plusieurs supernoeuds se fait par simple multiplication;
- la génération automatique de code est déjà assez délicate.

Il est facile de montrer que cette contrainte élimine de meilleurs partitionnements. Deux exemples seront donnés ultérieurement: un pour l'unicité de type (partitionnement tridimensionnel de § 2.3.1), l'autre pour l'unicité de taille (cf. § 2.3.3), puisque la contrainte choisie peut se décomposer en deux sous-contraintes.

Cette contrainte d'unicité de forme ne peut généralement pas être imposée sur l'ensemble du domaine d'itération. En effet, les supernoeuds *du bord* sont souvent tronqués par les frontières.

2.1.3. Partitionnement exact.

Enfin la troisième contrainte dérive aussi d'un désir de limiter l'ampleur des problèmes à résoudre. Le partitionnement doit être un partitionnement au sens mathématique. Chaque noeud appartient à un et un seul supernoeud. L'histoire des valeurs calculées n'est pas modifiée, seul l'ordonnancement des calculs l'est. Ceci élimine l'étude de la duplication des calculs pour des machines dont les temps d'échanges de données sont très importants par rapport aux temps de calculs élémentaires.

Ce n'est pas un problème puisque nous ne considérons ni les machines réparties sur un réseau ni les micropartitionnements au niveau de l'instruction machine. Le recalcul ne doit jamais être intéressant.

2.1.4. Conclusion

Les deuxième et troisième contraintes impliquent la recherche de pavages d'espaces de dimension N avec un motif unique. L'hyperrhomboèdre, qui comprend toutes les déformations longitudinales et angulaires de l'hypercube, est le candidat n°1. En dimension 2, l'hexagone s'ajoute a priori au parallélogramme mais le triangle est déjà éliminé par la condition d'identité par translation.

2.2. Graphe et cône de dépendance.

Ces deux notions ont déjà été introduites dans la section 1.4.4, mais la méthode de partitionnement que nous proposons suppose un graphe de dépendance connexe et un cône de dimension N.

2.2.1. Connexité du graphe de dépendance et dimension du cône

Si la dimension du cône de dépendance est inférieure à N, alors le graphe de dépendance n'est pas connexe. Il suffit de considérer deux noeuds séparés par un vecteur orthogonal à l'espace engendré par les rayons du cône. Un tel vecteur à coefficients entiers existe toujours car les rayons du cône ont des coefficients rationnels.

Cette situation se produit toutes les fois qu'il y a du parallélisme de type DO ALL parmi les boucles imbriquées. Le produit matriciel en est un exemple.

2.2.2. Saturation des cônes de dimension inférieure à N.

Le cas de la multiplication de matrices est à lui seul trop important pour que cette situation soit rédhibitoire pour notre méthode.

Les raisons pour lesquelles nous avons choisi de saturer le cône de dépendance plutôt que d'essayer de traiter des supernoeuds cylindriques sont les suivantes:

- le temps d'exécution d'un supernoeud aurait alors dépendu du **domaine** d'itération et de la position du supernoeud; il fallait donc développer une nouvelle technique d'optimisation avec des hypothèses différentes;
- le partitionnement n'aurait plus produit à coup sûr des supernoeuds suffisamment petits pour pouvoir être exécutés sur un processeur élémentaire; cet argument n'est valable que pour les architectures à cache ou à mémoire locale;
- l'augmentation de la localité par réutilisation des données présentes que permet l'utilisation des vecteurs d'autodépendance n'aurait pas été recherchée.

Il faut donc ajouter des vecteurs supplémentaires à l'ensemble des rayons du cône. Ces vecteurs vont être des pseudo-dépendances tronçonnant les supernoeuds cylindriques. Deux solutions nous semblent possibles: prendre des vecteurs de la base initiale B ou prendre des vecteurs d'autodépendance.

Les vecteurs de la base initiale B, $\vec{e}_1$, $\vec{e}_2$,... sont des candidats possibles. On se limite bien sûr aux vecteurs libres par rapport au rayons du cône de dépendance et on les marque pour qu'ils ne soient pas pris en compte lors du choix de la direction d'itération puisqu'ils n'introduisent aucune contrainte d'ordonnancement.

Une autre possibilité est fournie par les vecteurs d'autodépendance. Ils ont l'avantage de conduire à une réutilisation des données mais on ne peut pas être sûr ni qu'ils permettront de générer un cône de

dimension suffisante ni qu'ils ne seront pas trop nombreux et qu'un problème de choix ne se posera pas.

Une étude expérimentale s'avère nécessaire pour définir la bonne solution. Nous n'avons abordé ce problème que pour la multiplication de matrices qui a l'avantage de présenter deux vecteurs de base orthogonaux au cône de dépendance (réduit à une demi-droite) comme vecteurs d'autodépendance...

Pour avancer davantage dans ce domaine, il faudrait étudier manuellement plusieurs cas pratiques ne présentant pas une situation aussi favorable (mais naturelle) que la multiplication de matrices.

Montrons sur un exemple très simple que la prise en compte des autodépendances permet d'optimiser l'ordonnancement des calculs, y compris sur une machine usuelle:

```
DO I=1,N
  DO J=1,M
    A(I,J)=B(J)
```

La recherche d'une direction d'itération aussi normale que possible à l'autodépendance† pour maximiser la réutilisation de B(J) conduit à échanger les deux boucles sur I et J. Cela permet de garder B(J) dans un registre. Accessoirement, l'échange des deux boucles améliore aussi les accès à A si une machine à mémoire virtuelle est utilisée, mais il y aurait pu y avoir conflit entre les accès à A et les accès à B...

Pour finir de compléter le cône de dépendance, qui était initialement de dimension 0 sur cet exemple, il ne reste plus qu'à ajouter un vecteur de base, $\vec{e}_2$ en l'occurrence.

2.2.3. Cône de dimension N et graphe non connexe.

Comme l'espace d'itération est un espace discret, un cône de dépendance de dimension N n'implique pas la connexité du graphe de dépendance.

Deux ou plusieurs composantes connexes sont alors entremêlées. Un exemple de dimension 3 est donné dans [7] avec 3 vecteurs de dépendance constants (donc $\vec{c} = \vec{0}$ et $D = R$) et libres:

$$\vec{d}_1 = \begin{pmatrix} 4 \\ 2 \\ 4 \end{pmatrix} \quad \vec{d}_2 = \begin{pmatrix} 2 \\ 5 \\ 2 \end{pmatrix} \quad \vec{d}_3 = \begin{pmatrix} 1 \\ 4 \\ 3 \end{pmatrix}$$

Au total 32 composantes connexes peuvent être calculées indépendamment. Si une telle situation n'était pas rarissime, il serait intéressant de considérer une transformation englobant les 3 boucles imbriquées par une nouvelle boucle parallèle sur les composantes‡.

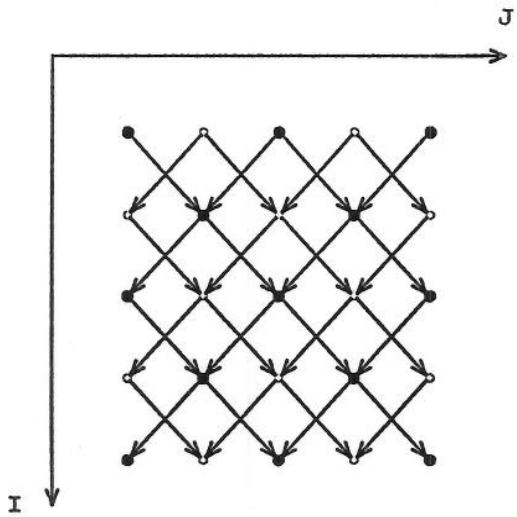
† Le critère est différent pour une dépendance usuelle: on cherche à itérer dans son sens, et non normalement, pour avoir du parallélisme. Voir 8.2 pour une discussion du problème.

‡ les vecteurs d'offset, utilisés au chapitre 3 pour la génération de code, donnent une première idée de ce qu'il faudrait faire.

Mais de tels cas sont très rares bien qu'on en trouve un exemple artificiel dans [2] pour introduire les DOACROSS à délai fractionnaire. Les méthodes *colorées* comme le black-and-red SOR[3] qui sont utilisées en analyse numérique en sont un exemple.

Voici un exemple simple mais non trivial, dont le graphe de dépendance est donné figure 2.2.3:

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



Graphe de dépendance non connexe

- Figure 2.2.3 -

Minoration du nombre de composantes de connexes

Le nombre de composantes connexes est minoré par le plus grand déterminant des bases dans lesquelles les vecteurs de dépendance ont des coordonnées entières. Dans ce cas, il ne peut y avoir de dépendance ni entre l'origine et les points se trouvant dans la cellule de base, ni entre ces points eux-mêmes, ni entre l'origine et des points se trouvant à l'intérieur strict d'une cellule de base:

- le vecteur joignant deux points en dépendance est une combinaison linéaire entière de vecteurs de dépendance entiers;
- ce vecteur est donc entier alors que les points situés dans la cellule de base ont des coordonnées fractionnaires.

Le nombre de ces points, y compris l'origine, est donné par le volume de la cellule de base, c'est-à-dire la valeur absolue du déterminant de la base. J.-K. Peir propose dans son PhD[6] une transformation de programme pour utiliser ce parallélisme.

Dans l'exemple précédent, D forme une base dont le déterminant est égal à deux, qui est justement le nombre de composantes connexes comme on peut le voir sur la figure 2.2.3.:

$$D = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \qquad det(D) = 2$$

Mais si on remplace le corps de boucle par:

```
T(I,J)=(T(I-1,J-1)+T(I-1,J)+T(I-1,J+1))/3.
```

le nouveau vecteur de dépendance rend le graphe connexe, mais n'a pas des coordonnées entières dans la base D. Il faut donc passer à une nouvelle base B' dont le déterminant ne vaut plus que un demi:

$$B' = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix} \qquad det(B') = \frac{1}{2} \qquad \vec{d}_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \qquad \vec{d}_2 = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \qquad \vec{d}_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

En fait on sait que la base initiale B vérifie la condition que les coordonnées des vecteurs de dépendance sont entières. Le minorant ne peut donc être inférieur à un.

Majoration du nombre de composantes connexes

On peut aussi majorer le nombre de composantes connexes en considérant tous les ensembles de N vecteurs de dépendance libres et en cherchant le minimum des déterminants des bases ainsi formées, puisque cela revient à supprimer certains des arcs du graphe de dépendance. Si ce minimum est égal au minorant calculé précédemment, il est aussi égal au nombre de composantes connexes. Dans le cas où il vaut 1, minorant fourni par la base initiale B, il n'y a qu'une composante connexe.

En particulier, quand D forme une base, le déterminant de D est à la fois minorant, car les vecteurs de dépendance ont des coordonnées entières dans D, et majorant. Il est égal au nombre de composantes connexes.

Pour pouvoir appliquer les méthodes de génération de code décrites au chapitre 3, il faut trouver une base dans laquelle tous les noeuds d'une composante connexe ont des coordonnées entières et dans

laquelle tous les points de coordonnées entières sont des noeuds. Quand D est une base, il peut être utilisé moyennant une transformation de programme réexprimant les boucles DO dans la base D.

L'autre solution, que nous retenons, consiste à ne pas profiter de l'existence des composantes connexes.

2.3. Partitionnement par hyperplans

2.3.1. Définitions

Soit un hyperplan défini par un vecteur normal $\vec{h}$ et une constante K. Cet hyperplan découpe l'espace d'itération en deux partitions vérifiant les inégalités linéaires:

$$\vec{h}\ \vec{j} < K \quad \text{et} \quad \vec{h}\ \vec{j} \geq K$$

Comme l'espace d'itération est discret, les deux inégalités pourraient être strictes pourvu que la constante K soit bien choisie. Cela permettrait de tracer des frontières sur lesquelles n'apparaissent aucun noeud, mais ne faciliterait pas les raisonnements. Par exemple, avec l'orientation choisie pour les vecteurs $\vec{h}_1$, $\vec{h}_2$ et $\vec{h}_3$ en figure 2.3.1.a, les noeuds qui appartiennent à un hyperplan appartiennent au supernoeud se trouvant *au dessus*. On remarque donc que $\vec{h}_1$ et $-\vec{h}_1$ ne définissent pas les mêmes sous-espaces.

En ne considérant plus une constante K, mais une famille de constantes entières K_i, on peut découper l'espace d'itération en tranches. Pour obtenir des tranches régulières, il suffit d'avoir:

$$K_i = K \times i$$

Pour simplifier les calculs, on peut reporter la constante K dans le vecteur $\vec{h}$ en le multipliant par K^{-1} et définir l'appartenance à une tranche par la fonction partie entière, E:

$$\vec{j} \in \textit{tranche}\ \ i \quad \equiv \quad E(\vec{j}\ \vec{h}) = i$$

On peut donc définir un partitionnement par un vecteur $\vec{h}$. Sa norme donne l'inverse de la vitesse à laquelle on progresse dans la direction $\vec{h}$. La distance en terme de tranches séparant deux points $\vec{j}_1$ et $\vec{j}_2$ est:

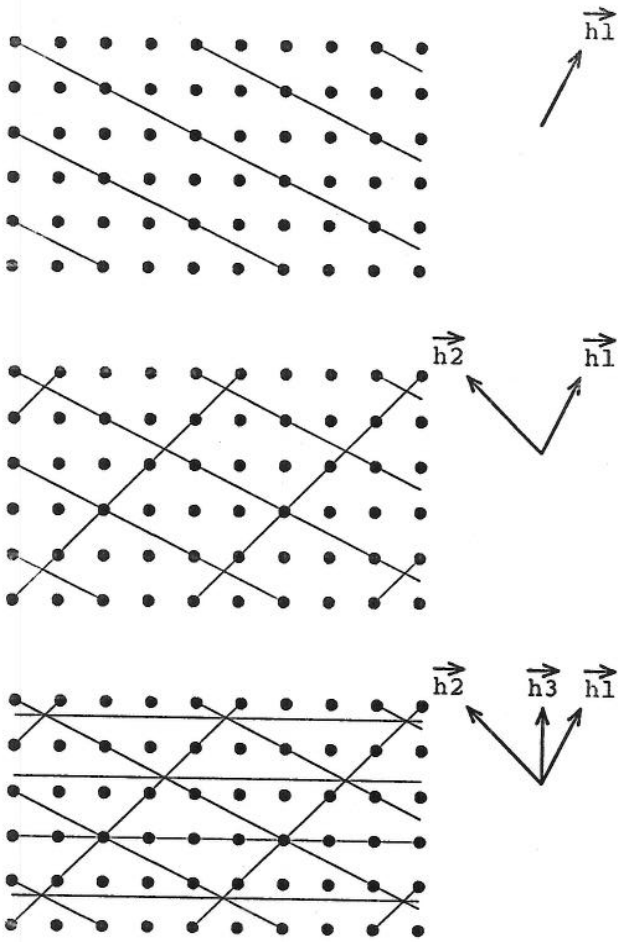
$$<\vec{j}_2 - \vec{j}_1 \,.\, \vec{h}>$$

Après avoir découpé l'espace d'itération en une première série de tranches selon un vecteur $\vec{h}_1$, il est possible de le découper une deuxième fois par un vecteur $\vec{h}_2$, puis une troisième fois et ainsi de suite. Après N découpages (N est la dimension de l'espace d'itération) avec des vecteurs $\vec{h}_i$ libres entre eux, on obtient des partitions égales à une translation près. Chaque partition est repérable par un N-uplet sur **N** et l'appartenance d'un élément $\vec{j}$ à une partition se détermine en calculant:

$$(E(\vec{h}_1\vec{j}),E(\vec{h}_2\vec{j}), \cdots E(\vec{h}_N\ \vec{j}))$$

La mise en correspondance de la partition $(k_1,k_2, \cdots k_N)$ et de la partition $(k_1{}' ,k_2{}' , \cdots k_N{}')$ se fait par une translation $\vec{t}$ telle que:

$$\forall\, i \in [1..N] \quad \vec{t}\ \vec{h}_i = k_i{}' - k_i$$



Partitionnements par hyperplans

- Figure 2.3.1.a -

Cette condition définit parfaitement $\vec{t}$ quand les $\vec{h_i}$'s forment une base. Quand ils ne sont pas générateurs, n'importe quelle translation vérifiant cette condition est valide. Mais quand les $\vec{h_i}$'s sont liés, l'existence d'une telle translation n'est plus garantie car elle est surcontrainte.

Ceci nous conduit à ignorer les partitionnements par un ensemble de $\vec{h_i}$'s non libres afin de respecter le choix fait en 2.1.4 pour simplifier la génération de code. Pour obtenir des partitions de volume fini, nous utilisons N vecteurs $\vec{h_i}$'s. Le volume d'une partition est égal à l'inverse du déterminant de la base H formée par les vecteurs $\vec{h_i}$'s.

La figure 2.3.1.a montre trois exemples de partitionnements par 1, 2 et 3 familles d'hyperplans en dimension 2. En imposant une contrainte sur les distances séparant les hyperplans de la troisième famille, on obtient un découpage plausible, qui permettrait de refabriquer des hexagones en associant plusieurs

triangles, mais qui ne respecte pas la contrainte d'identité par translation.

La figure 2.3.1.b est un exemple de partitionnement en dimension 3 avec 4 plans. Comme avec le partitionnement par trois droites en dimension 2, les itérations successives ne se déduisent plus par translation et aux doubles pyramides à base carrée succèdent des pyramides à base triangulaire. La suppression d'une des familles de plans de partitionnement regroupe les volumes élémentaires pour obtenir des rhomboèdres comme le montre la figure 2.3.1.c. La surface d'une itération devient régulière et le dessin de la figure 2.3.1.b est remplacé par celui de la figure 2.3.1.d.

Ces dessins permettent de bien comprendre la simplification qu'apporte la limitation du partitionnement à N hyperplans dans un espace de dimension N. Mais cette limitation sera payée par la perte d'un sous-ensemble des directions d'itération: le partitionnement à quatre hyperplans montré en figure 2.3.1.b permet d'itérer verticalement, tandis que celui de la figure 2.3.1.d est nécessairement oblique (la direction d'itération représentée est (1,1,1)). Pour progresser verticalement, il faut commencer par progresser horizontalement et construire une espèce d'escalier.

Enfin, le dessin intermédiaire de la figure 2.3.1.a montre que l'existence d'une translation dans un espace réél n'est pas une condition suffisante pour montrer l'existence d'une translation dans l'espace discret: deux supernoeuds tangents ne contiennent pas le même motif de noeuds. Une condition supplémentaire sur les partitionnements sera introduite en 2.4.1.

2.3.2. Partitionnement valide

Comme nous l'avons vu en 2.1.1, nous voulons avoir des synchronisations externes aux supernoeuds et donc respecter la condition (1) de ce même paragraphe.

Théorème: Un partitionnement de l'espace d'itération par une base H est un partitionnement valide si et seulement si pour tout vecteur $\vec{h_i}$ de H et pour tout rayon $\vec{r_k}$ de R $\vec{h_i}\,\vec{r_k}$ est positif.

$$\forall \vec{h_i} \in H \quad \forall \vec{r_k} \in R \qquad \vec{h_i}\ \vec{r_k} \geq 0$$

Preuve:

On remplace les noeuds par les vecteurs d'itération correspondants dans la condition (1) et on fait la démonstration par l'absurde en supposant cette condition fausse:

$$\exists\ \vec{j}_1, \vec{j}_1{}' \in s_1 \quad \exists\ \vec{j}_2, \vec{j}_2{}' \in s_2 \quad \vec{j}_1 < \vec{j}_2 \text{ et } \vec{j}_2{}' < \vec{j}_1{}'$$

En remplaçant le symbole d'inégalité par sa définition, on obtient:

$$\vec{j}_2 = \vec{j}_1 + \sum_k \lambda_k \vec{d}_k \qquad \lambda_k \geq 0$$

$$\vec{j}_1{}' = \vec{j}_2{}' + \sum_k \mu_k \vec{d}_k \qquad \mu_k \geq 0$$

Comme les $\vec{h}_i\,\vec{d}_k$ sont positifs et que $\vec{j}_1$ et $\vec{j}_1'$ appartiennent au même supernoeud s_1:

$$(E(\vec{h}_1\vec{j}_2),E(\vec{h}_2\vec{j}_2),\cdots E(\vec{h}_N\vec{j}_2)) \geq (E(\vec{h}_1\vec{j}_1),E(\vec{h}_2\vec{j}_1),\cdots E(\vec{h}_N\vec{j}_1))$$

$$(E(\vec{h}_1\vec{j}_1),E(\vec{h}_2\vec{j}_1),\cdots E(\vec{h}_N\vec{j}_1)) = (E(\vec{h}_1\vec{j}_1'),E(\vec{h}_2\vec{j}_1'),\cdots E(\vec{h}_N\vec{j}_1'))$$

$$(E(\vec{h}_1\vec{j}_1'),E(\vec{h}_2\vec{j}_1'),\cdots E(\vec{h}_N\vec{j}_1')) \geq (E(\vec{h}_1\vec{j}_2'),E(\vec{h}_2\vec{j}_2'),\cdots E(\vec{h}_N\vec{j}_2'))$$

Comme $\vec{j}_2$ et $\vec{j}_2'$ appartiennent au même supernoeud s_2, les deux termes extrêmes de l'inégalité précédente sont égaux et les deux noeuds sont confondus.

Fin de la preuve

Dans la suite, nous utiliserons aussi l'expression *hyperplan valide* pour tout hyperplan $\vec{h}$ vérifiant:

$$\forall \vec{r}_k \in R \quad \vec{h}\,\vec{r}_k \geq 0$$

Un partitionnement valide est donc obtenu avec N hyperplans valides et libres.

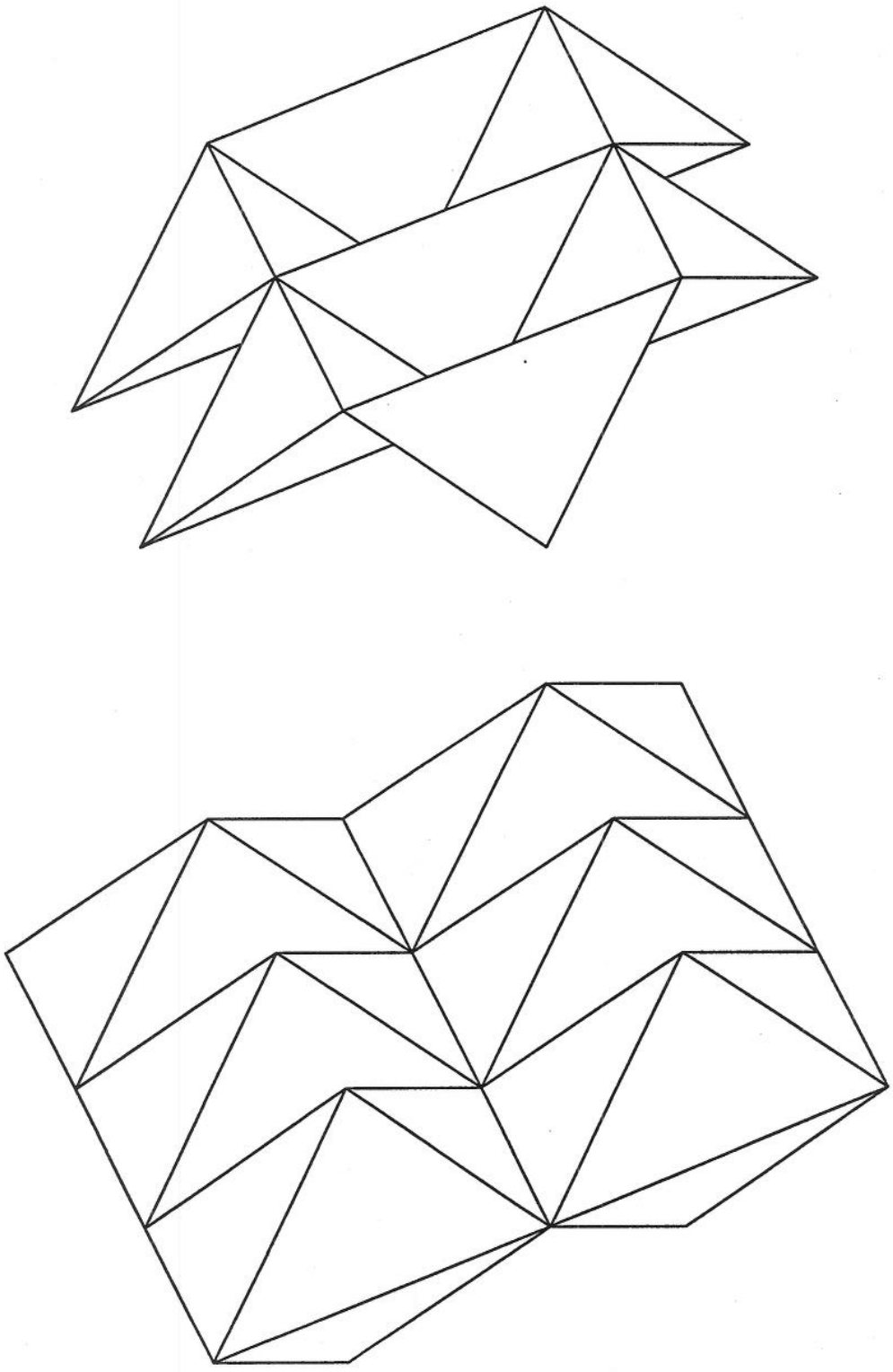
Notons que l'ensemble des hyperplans valides est lui aussi un cône et que le système d'inégalités précédent peut être remplacé par une description équivalente avec un sommet, qui est toujours le vecteur nul, et des rayons. Cette deuxième caractérisation a l'avantage de permettre une génération facile de cet ensemble des hyperplans valides.

2.3.3. Inconvénient du partitionnement par hyperplans

Pour simplifier la génération de code, le partitionnement par hyperplans ignore complètement la forme du **domaine** d'itération et génère toujours des supernoeuds identiques. Par exemple, dans le cas de domaines triangulaires, au sens géométrique, le degré de parallélisme[†] peut varier suivant le point du domaine d'itération considéré. Au voisinage du sommet du domaine triangulaire représenté figure 2.3.3.a, un seul processeur pourra être utilisé.

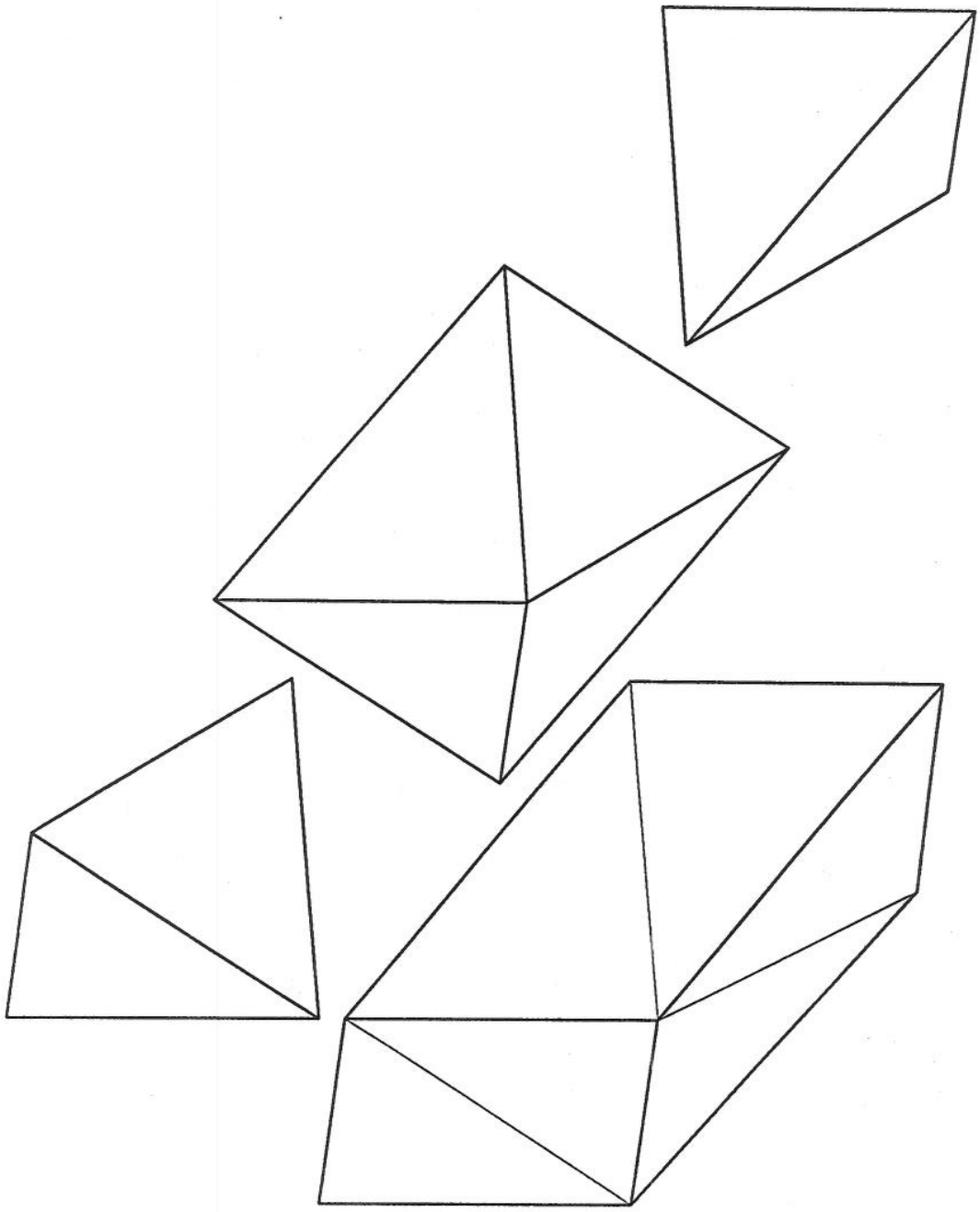
Pour éviter ce problème, on pourrait envisager de partitionner angulairement le domaine d'itération mais les supernoeuds ne vérifieraient pas toujours la condition (1) comme le montrent les noeuds s_1 et s_2 de la figure 2.3.3.b. La génération de code serait aussi assez délicate. On peut aussi envisager d'adapter la taille des supernoeuds au degré de parallélisme disponible comme cela est montré figure 2.3.3.c. Un tel partitionnement pourrait être obtenu en découpant le domaine d'itération en plusieurs zones suivant leur degré de parallélisme et en appliquant indépendamment notre technique de partitionnement à chaque zone.

[†] Le degré de parallélisme est une notion intuitive mal définie, relative à un point de l'espace d'itération. C'est le nombre de processeurs pouvant travailler en parallèle quand un processeur exécute un noeud. Ce nombre n'est pas unique... Il ne s'agit pas du nombre de noeuds pouvant être exécutés en parallèle avec le noeud considéré. Karp et Miller parlent du parallélisme d'un ordonnancement et prennent le max sur l'ensemble des ordonnancements. Pour optimiser l'exécution, on cherchera dans le chapitre 6 un hyperplan à degré de parallélisme constant sur le domaine d'itération.



Partitionnement tridimensionnel par 4 hyperplans. Allure des fronts

- Figure 2.3.1.b -



Composition des supernoeuds réduisant le nombre d'hyperplans à trois

- Figure 2.3.1.c -

M. Cosnard & al.[1] proposent une technique adaptative de ce genre pour effectuer l'élimination de Gauss-Seidel sur un multiprocesseur. Le domaine d'itération est tridimensionnel pyramidal. A chaque étape, le plan de calcul diminue et la taille des supernoeuds est dynamiquement adaptée en fonction du nombre de processeurs et du nombre de calculs à effectuer. L'équivalent des supernoeuds est un ensemble bi-dimensionnel. Le caractère dynamique et quasi-continu de l'adaptation ne pourrait pas être obtenu avec la technique décrite au paragraphe précédent. De plus, nos supernoeuds seraient de dimension 3. Il semble impossible de prévoir quelle technique donnera les meilleurs résultats sans faire des expérimentations.

Ce problème va être négligé par la suite. Dans la mesure où le domaine d'itération sera très grand par rapport au nombre de processeurs, le gain qu'on pourrait obtenir en optimisant les calculs au voisinage des *pointes* sera négligeable. Si le domaine est tel que le temps de calcul des pointes, effectué de manière non optimale, est égal à 1% du temps d'exécution total, la meilleure optimisation n'apportera pas d'amélioration supérieure à ce 1 %.

Le partitionnement par hyperplans ne permet pas non plus d'obtenir des supernoeuds hexagonaux, qui dans certains cas, peuvent être optimaux grâce à leur rapport périmètre / surface. Cependant, ce rapport n'est pas meilleur que pour un carré équivalent sur un espace discret à moins que le programmeur n'utilise une maille de discrétisation hexagonale, car on doit utiliser une distance de "Manhattan" qui n'avantage pas les diagonales. De toute façon ce partitionnement ne se généralise pas aux dimensions supérieures à 2 et ne constitue qu'un cas particulier par rapport aux rhomboèdres que nous allons étudier.

2.3.4. Hyperplans critiques

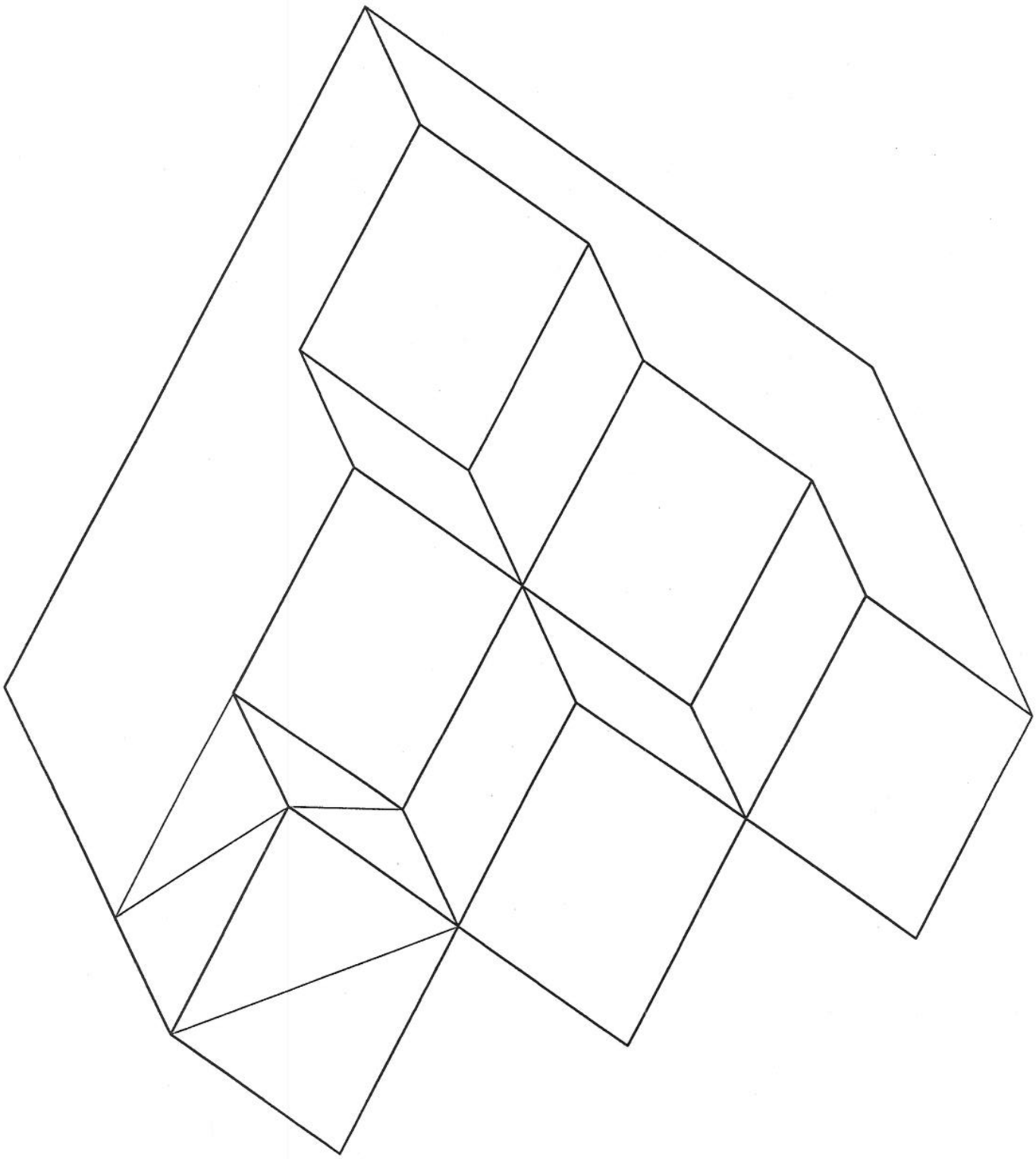
Pour optimiser le partitionnement, il est important de disposer de la définition constructive de l'ensemble des partitionnements valides. Nous allons montrer que ces partitionnements peuvent être définis à partir d'un demi-cône positif. Les normales à ces hyperplans critiques, qui peuvent être vues comme des directions[†] d'itération limites, sont les rayons de ce cône.

2.3.4.1. Définition

Un hyperplan valide $\vec{h}$ est dit critique s'il est un des rayons du cône des hyperplans valides. Il contient alors $N-1$ rayons indépendants du cône de dépendance.

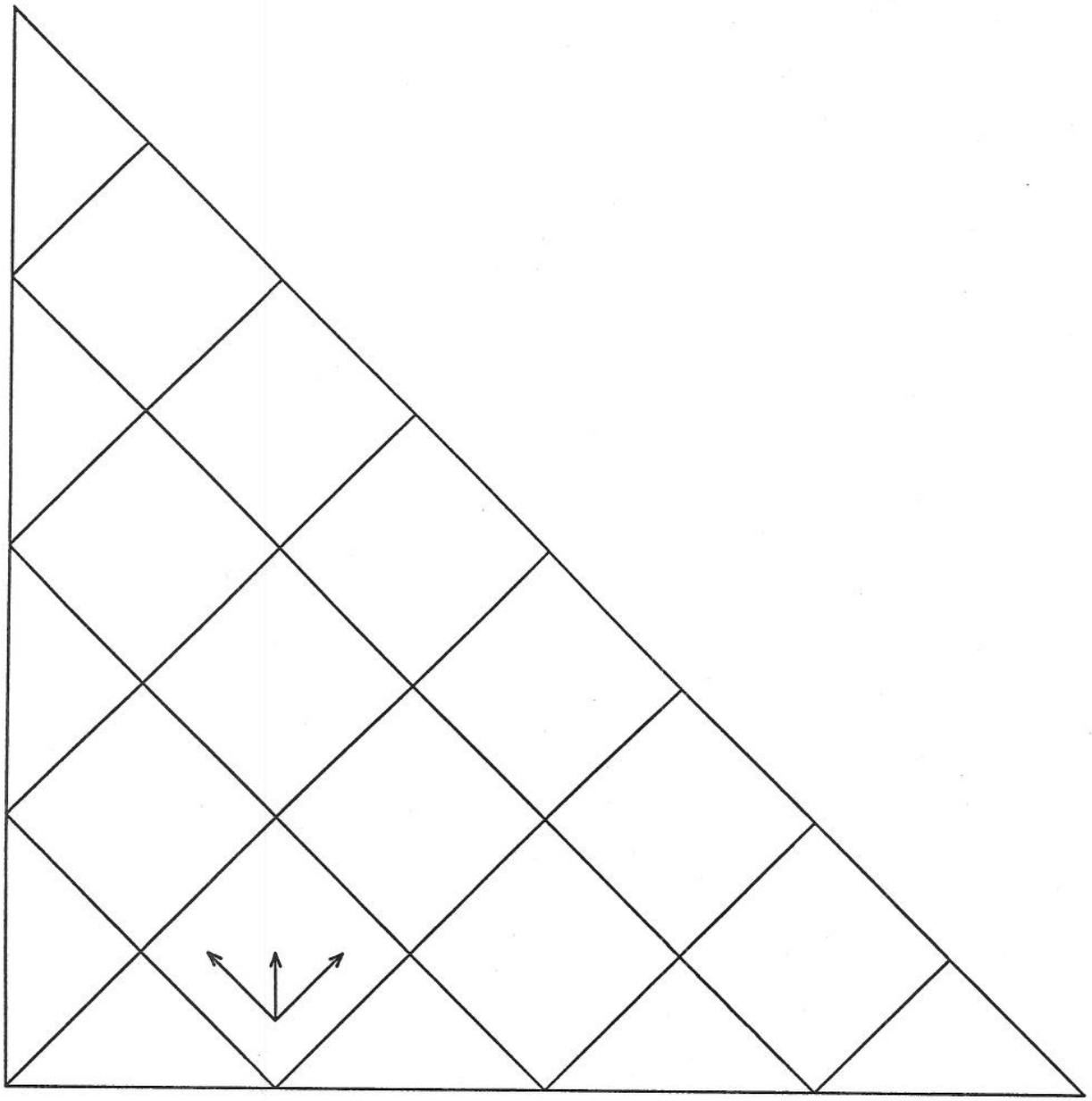
Ces hyperplans critiques sont donc définis par les normales intérieures aux faces du cône de dépendance. Chaque face est un hyperplan défini par $N-1$ vecteurs de dépendance et la condition de validité peut être vérifiée parce que le cône de dépendance ne remplit pas tout l'espace. Il est en effet limité au plus à un demi-espace par l'ordre lexicographique. Toute dépendance $\vec{d}$ doit en effet vérifier

[†] Le terme de *vecteur critique* est utilisé ultérieurement pour définir les chemins critiques de l'ordonnancement.



Allure des fronts du partitionnement à trois hyperplans

- Figure 2.3.1.d -

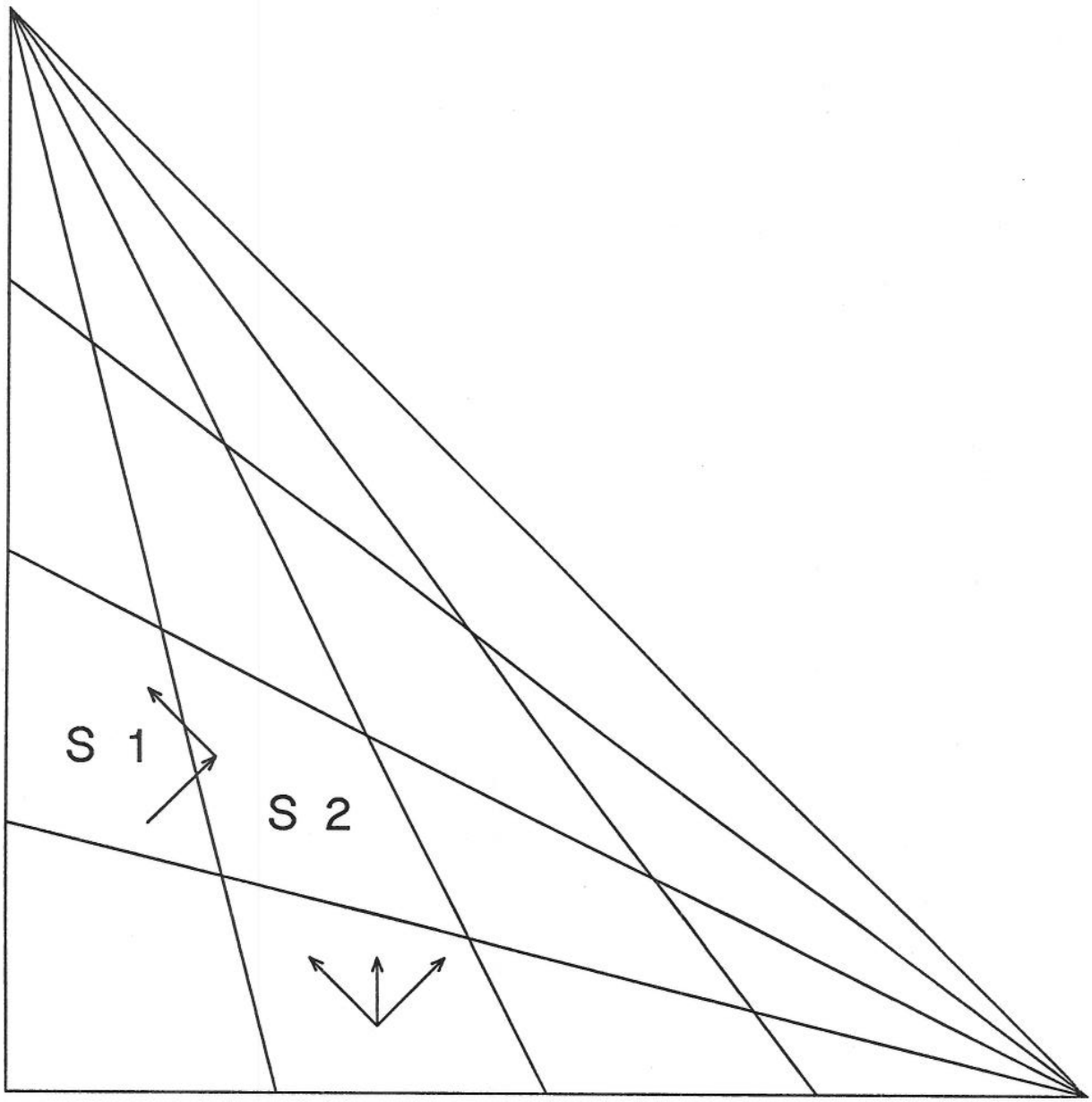


Partitionnement d'un domaine triangulaire

- Figure 2.3.3.a -

$\vec{d}\ \vec{e}_1 \geq 0$.

Comme ce cône est de dimension N, il a au moins N faces et donc N hyperplans critiques au moins sont définis. En dimension 2, il n'existe jamais plus de 2 hyperplans critiques, qui sont d'ailleurs réduits à des droites. Pour les dimensions égales ou supérieures à 3, le nombre d'hyperplans critiques est quelconque. Par exemple, K vecteurs de dépendance du type:

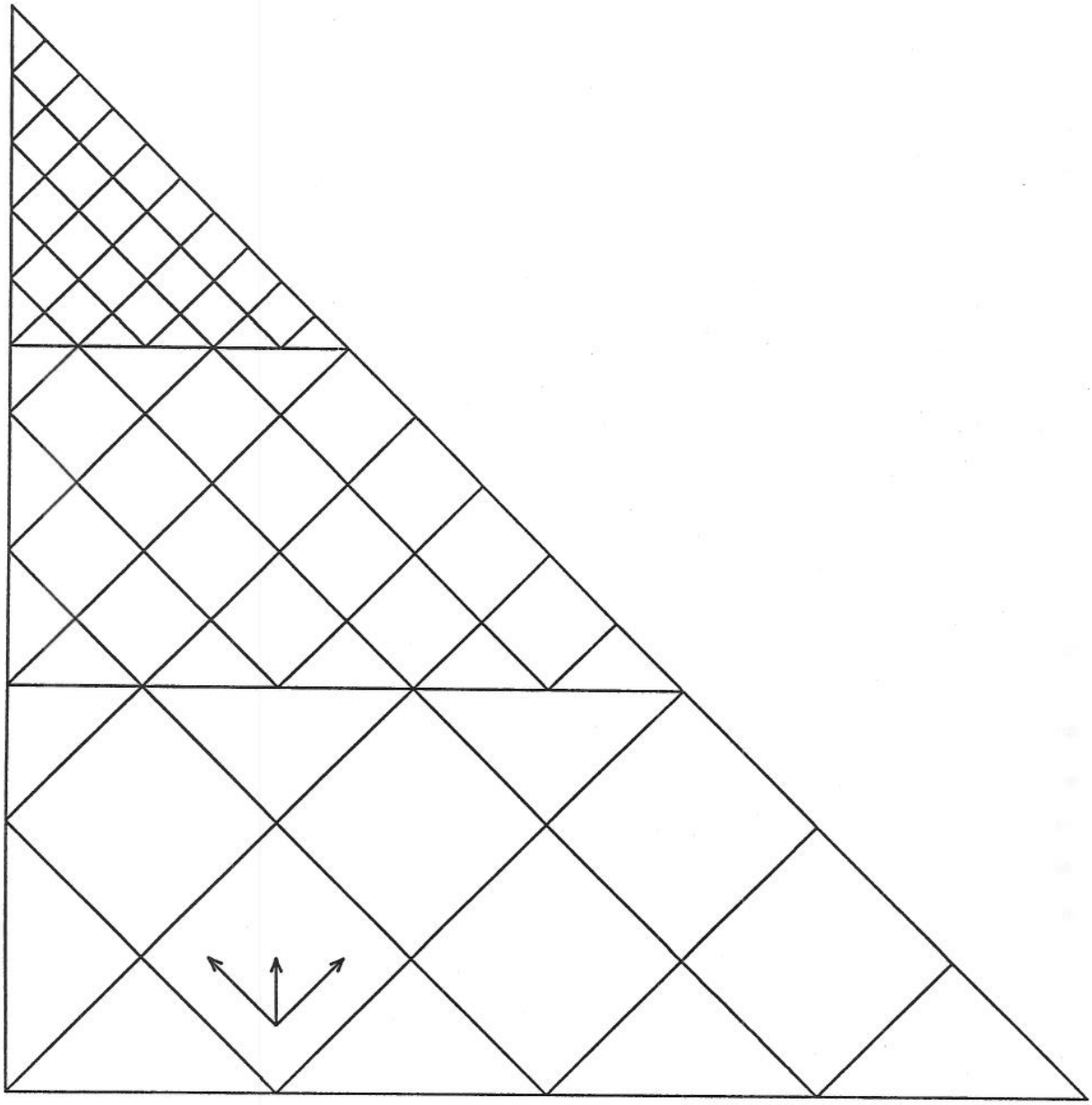


Partitionnement angulaire

- Figure 2.3.3.b -

$$\vec{d}_k = (1, k, k^2)^t \qquad k \in [0..K-1]$$

définissent un cône à K faces et donc K hyperplans critiques.



Partitionnement à densité variable

- Figure 2.3.3.c -

2.3.4.2. Calcul de l'ensemble des hyperplans critiques

L'algorithme général qui permet de passer de la représentation d'un polyèdre par un système d'inégalités à un représentation par sommets, droites et rayons[4] peut être utilisé en dernier ressort. Mais les cônes que nous avons sont des polyèdres bien particuliers et il est possible de trouver des algorithmes plus simples et peut-être plus rapides.

Soit H l'ensemble des hyperplans critiques. La procédure suivante permet de calculer H, même si l'ensemble des vecteurs de dépendance n'a pas été réduit aux rayons du cône de dépendance:

$H := \emptyset$

Pour chaque combinaison de $N-1$ vecteurs indépendants de D :

prendre un vecteur orthogonal unitaire $\vec{h}$

si $\vec{h}^t D \geq 0$ alors $H := H \bigcup \{\vec{h}\}$

sinonsi $\vec{h}^t D \leq 0$ alors $H := H \bigcup \{-\vec{h}\}$

sinon $\vec{h}^t$ ne convient pas

L'ordre utilisé ici est l'ordre partiel *terme à terme* et non l'ordre lexicographique.

Cette procédure semble peu utilisable à première vue, puisqu'elle implique une énumération combinatoire de combinaisons et une inversion de matrice à chaque étape pour trouver le vecteur orthogonal. Nous avons essayé de développer des procédures plus efficaces guidées par une sélection progressive des vecteurs de dépendances de manière à pouvoir couper au plus vite les branches inutiles, mais sans succès.

Dans les cas usuels, la dimension ne devrait pas dépasser 4 et le nombre de vecteurs de dépendance est limité. Une énumération exhaustive est donc sans doute acceptable.

2.3.4.3. Recherche d'hyperplans critiques en dimension 2

Le cas de la dimension 2 est spécial car les faces et les arêtes sont confondues. Il y a toujours exactement 2 hyperplans critiques et leur calcul s'effectue trivialement en sélectionnant les vecteurs de dépendance d'angle extrême, puis en prenant les vecteurs orthogonaux de bon signe. Par exemple, dans le cas décrit en figure 2.3.4.3, les vecteurs de dépendances extrêmes sont $\vec{d}_1$ et $\vec{d}_4$ et ils définissent deux hyperplans critiques $\vec{h}_1$ et $\vec{h}_2$.

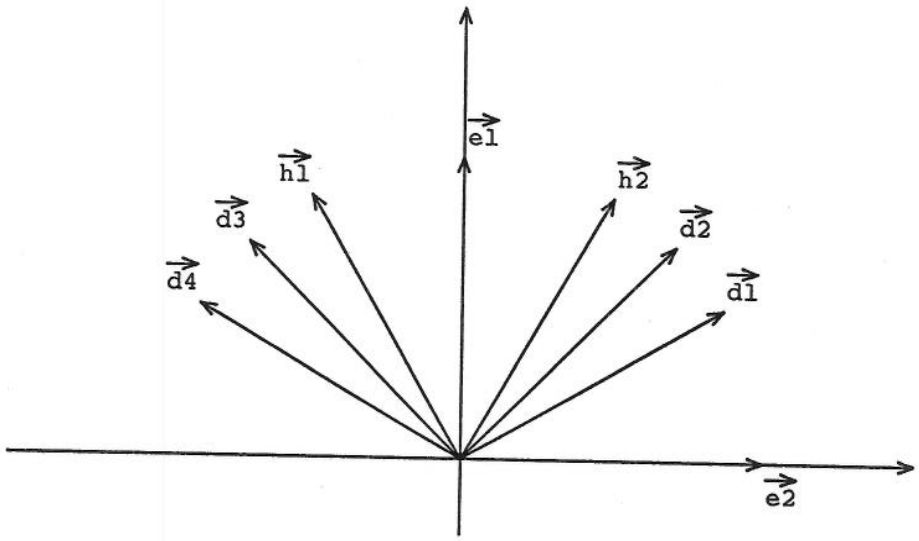
Cette procédure est aussi utilisée dans la méthode hyperplane[5, 9] sous des formes voisines.

2.3.4.4. Cas des cônes à N faces

Si le nombre de rayons du cône de dépendance est égal à N, le nombre de faces est aussi égal à N. Les matrices R et H sont telles que:

$$H^t \; R = I \quad i.e. \quad H^t = R^{-1}$$

Le calcul des hyperplans critiques est équivalent dans ce cas à une inversion de matrice.



Hyperplans critiques en dimension 2

- Figure 2.3.4.3 -

2.3.4.5. Constructions de partitionnements complets

Nous ne cherchons pas un seul hyperplan valide mais N hyperplans dont les normales forment une **base** H. Le théorème suivant permet de résoudre le problème pour les dimensions inférieures à 4, sans garantie d'optimalité (cf. § 6.2.5.):

Théorème: Quand N est strictement inférieur à 4, tout ensemble de N hyperplans critiques forme une base.

Preuve:

- évidente en dimension 1;
- deux rayons sont directement calculés en dimension 2 (cf. 2.3.4.3);
- en dimension 3, il faut utiliser le fait que les rayons sont lexico-positifs;
- en dimension 4, on peut construire un contre-exemple avec les vecteurs $\vec{d}_k$ donnés en § 2.3.4.1 en leur ajoutant une quatrième coordonnée nulle et en leur adjoignant le vecteur $(0,0,0,1)^t$.

Fin de la preuve

2.3.5. Vecteur de direction d'itération

L'ordre des noeuds est fixé du fait de la sémantique des boucles DO comme un ordre lexicographique. Une famille d'ordres plus riche est la famille dérivée des formes linéaires de l'espace d'itération. Ce sont, dans le vocabulaire systolique, les fonctions temporelles linéaires.

Etant donné une forme linéaire $\vec{h}$, le noeud $\vec{j}$ doit être exécuté avant le noeud $\vec{j}'$ si et seulement si $\vec{h}\ \vec{j} < \vec{h}\ \vec{j}'$. Si $\vec{h}\ \vec{j} = \vec{h}\ \vec{j}'$, les deux noeuds peuvent être exécutés en parallèle.

Pour qu'aucune dépendance ne soit violée, il suffit que:

$$\forall\, \vec{d}_k \in D \quad \vec{h}\ \vec{d}_k > 0 \tag{3}$$

Il faut noter que contrairement à la condition (2) de validité des hyperplans, l'inégalité est stricte.

Montrons que cette condition (3) est suffisante. Si $\vec{j}'$ doit être exécuté après $\vec{j}$, on a:

$$\vec{j}' = \vec{j} + \sum_k \lambda_k \vec{d}_k \qquad \forall k\ \ \lambda_k \geq 0 \ \ et \ \ \exists\ k\ \ \lambda_k > 0$$

$$\vec{h}\ \vec{j}' = \vec{h}\ \vec{j} + \sum_k \lambda_k \vec{h}\ \vec{d}_k \qquad \forall k\ \ \lambda_k \geq 0 \ \ et \ \ \exists\ k\ \ \lambda_k > 0$$

$$\vec{h}\ \vec{j}' - \vec{h}\ \vec{j} > 0$$

Un tel vecteur $\vec{h}$ est appelé vecteur de direction d'itération.

L'ordre lexicographique peut être simulé avec un vecteur de direction d'itération dès que le domaine d'itération est borné, de la même manière qu'on linéarise les tableaux multidimensionnels en mémoire.

Caractérisation de l'ensemble des directions d'itération

Théorème: Tout $\vec{h}$, vecteur de direction d'itération, est une combinaison linéaire strictement positive d'hyperplans critiques. Et réciproquement.

Preuve:

- la réciproque est évidente puisque le cône des directions d'itération est inclus dans le cône des partitionnements valides;
- dans l'autre sens, il suffit de remarquer que la différence entre les deux cônes se réduit à la surface du premier: il suffit que tous les coefficients soient strictement positifs.

Fin de la preuve

Ce théorème nous permet de caractériser l'ensemble des directions d'itération pour trouver une valeur optimale.

Recherche d'un partitionnement respectant un vecteur de direction d'itération

La direction d'itération résultant d'un partitionnement H est donnée par le vecteur $\vec{1}_H$, somme de tous les $\vec{h_i}$. En optimisation, nous aurons besoin de faire l'opération inverse et de déterminer le partitionnement à partir de la direction d'itération.

Conjecture: Pour tout $\vec{h}$, vecteur de direction d'itération, il existe N hyperplans critiques tels que $\vec{h}$ soit une combinaison linéaire **positive** (non strictement) de ces N hyperplans critiques.

Justification:

- d'abord un lemme pour montrer qu'on peut décomposer un cône en cônes élémentaires disjoints qui ont chacun exactement N rayons; intuitivement, en dimension 3, il faut prendre trois rayons voisins, ajouter une nouvelle face, éliminer du cône initial le rayon qui n'appartient pas à la nouvelle face et recommencer;
- étudier l'union de ces sous-cônes et se rendre compte qu'on perd les faces ajoutées pour effectuer la décomposition, car la combinaison n'est pas strictement positive sur ces faces.

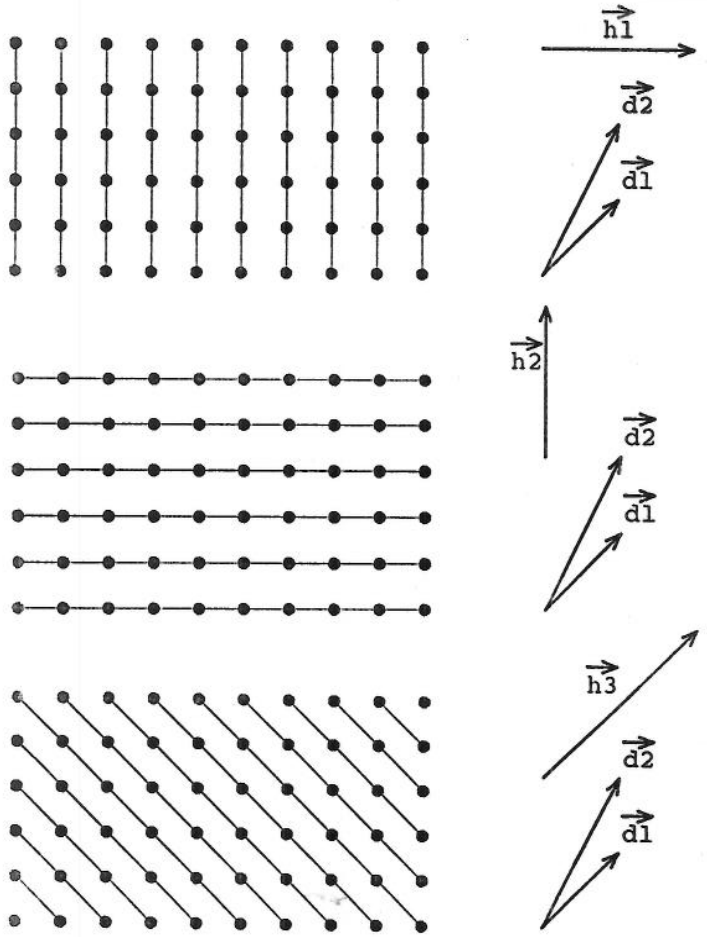
Fin de la justification

La combinaison n'est pas **strictement** positive. On perd un certain sous-espace de directions d'itération éventuellement intéressantes comme on l'a vu en 2.3.1. avec le partitionnement tridimensionnel.

Dans la suite, nous allons nous restreindre à la dimension 2 et la proposition plus forte suivante y est vraie: Tout $\vec{h}$, vecteur de direction d'itération, est une combinaison linéaire strictement positive des deux hyperplans critiques.

Exemple

Trois ordonnancements valides, correspondant à trois $\vec{h}$ différents, sont dessinés en figure 2.3.5. Les noeuds appartenant à la même droite peuvent être exécutés en parallèle, par définition des vecteurs de direction d'itération.



Trois directions d'itération possibles

- Figure 2.3.5 -

2.4. Base de partitionnement

2.4.1. Agglomération par une base. Définition et validité

L'idée consiste à fabriquer des supernoeuds en considérant que deux noeuds $\vec{j}_1$ et $\vec{j}_2$ appartiennent au même supernoeud si les parties entières de leurs coordonnées dans cette base P sont égales.

$$n_1 \text{ et } n_2 \text{ appartiennent au meme supernoeud} \equiv \lfloor \vec{j}_1^{\,P} \rfloor = \lfloor \vec{j}_2^{\,P} \rfloor$$

(la notation synthétique $\lfloor \ldots \rfloor$ représente un n-uplet à base de fonctions E du genre de ceux qu'on avait en 2.3.2)

La cellule unité de la base P constitue le modèle de supernoeud.

Théorème: Le partitionnement est valide si tous les vecteurs de dépendances ont des coordonnées positives dans cette base P.

$$P^{-1} D \geq 0$$

La preuve est donnée dans le paragraphe suivant.

Il suffit que tous les rayons du cône de dépendance vérifient cette propriété, puisque tous les vecteurs de dépendances sont des combinaisons linéaires positives des rayons.

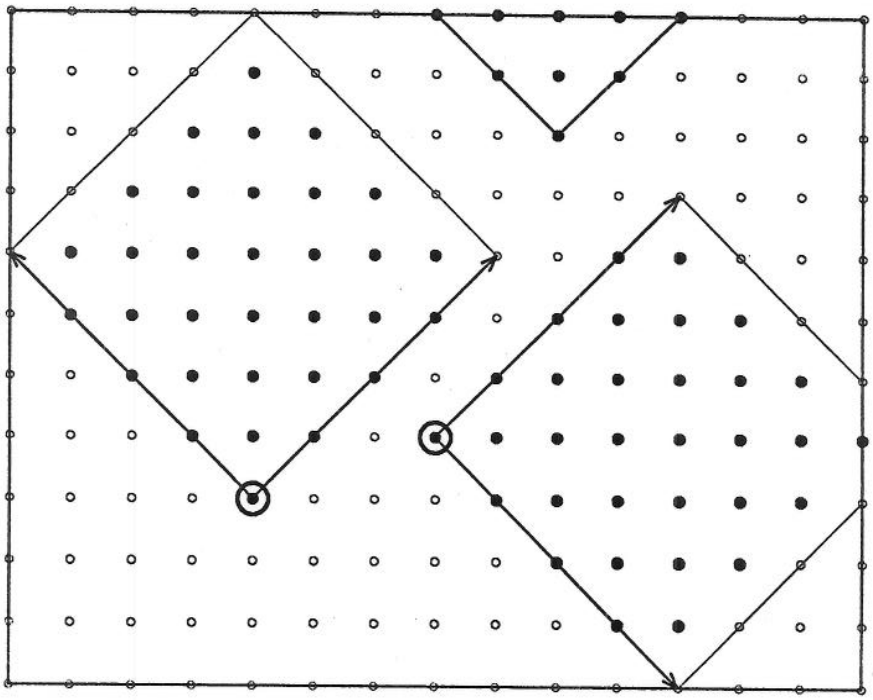
$$P^{-1} R \geq 0$$

Dans le cas où le cône de dépendance R a N rayons, R est une base de partitionnement valide par définition, ainsi que toutes les bases obtenues en remplaçant un $\vec{r}_k$ par $\lambda_k \vec{r}_k$, $\lambda_k > 0$. Les hyperplans critiques s'obtiennent en inversant et en transposant R (cf. 2.3.4.4) ou une de ses dérivées. Cette dernière remarque est donc toujours valide en dimension 2 puisque le cône de dépendance a toujours deux rayons.

De nombreuses définitions et conventions relatives aux supernoeuds définis par une base P vont être utilisées par la suite.

- l'origine d'un supernoeud est le noeud dont le vecteur d'itération (i.e. les coordonnées) est le plus petit dans la base P ; les origines de deux supernoeuds sont représentées par des points cerclés sur la figure 2.4.1;
- en dimension 2, les sommets d'un supernoeud en sont les quatre points extrémes; l'origine du supernoeud est l'un d'eux, le seul qui appartienne au supernoeud, comme on peut le voir sur la figure 2.4.1; cela est dû au caractère strict de deux des quatre inégalités définissant les supernoeuds;
- l'origine est le plus petit noeud du supernoeud pour n'importe quelle direction d'itération valide tandis qu'un des sommets est toujours le plus grand élément;
- la figure 2.4.1 montre l'effet des inégalités strictes et non strictes; seuls les noeuds représentés par des points noirs font partie des supernoeuds alors que les limites de ces derniers passent par des points qui ne leur appartiennent pas; dans la plupart des figures, les frontières des supernoeuds seront ainsi le tracé de la contrainte de définition; l'utilisation de $<$ et $\leq$ dans la définition des supernoeuds pose des problèmes pour les algorithmes sur les polyèdres convexes en nombres réels; heureusement, nous travaillerons toujours avec des entiers relatifs;
- notons la différence qui existe entre un partitionnement par $(\vec{p}_1, \vec{p}_2)$ et $(-\vec{p}_1, \vec{p}_2)$ en discret (figure 2.4.1); elle est duale de la différence existant entre les partitionnements $\vec{h}$ et $-\vec{h}$;

- enfin, un supernoeud limité par le domaine d'itération est appelé supernoeud partiel.



Représentation des supernoeuds

- Figure 2.4.1 -

2.4.2. Relation avec le partitionnement par hyperplans

Le partitionnement par une base P est équivalent à un partitionnement par les hyperplans que définissent $N-1$ vecteurs de base. Réciproquement, un partitionnement par N hyperplans libres est équivalent à un partitionnement par une base dont les vecteurs sont définis par les intersections de $N-1$ hyperplans.

Le critère de validité d'un partitionnement par une base P - la positivité des coordonnées des dépendances dans cette base - permet de trouver des normales $\vec{h_i}$'s aux faces vérifiant $\vec{h_i}\,\vec{d_k} \geq 0$. Le partitionnement par hyperplans équivalent est donc valide.

Le passage de la base P à la base H se fait par inversion et transposition:

$$H^t = P^{-1}$$

Cela conduit en effet aux relations:

$$\vec{h}_i \, \vec{p}_j = \delta_{i,j}$$

qui traduisent le critère de normalité aux faces.

D'autres normes auraient pu être imposées aux vecteurs $\vec{h}_i$, mais la condition choisie permet d'obtenir le résultat suivant, qui sera utilisé en optimisation dans le chapitre 6:

Théorème: La longueur L, exprimée en supernoeuds, du chemin critique joignant deux supernoeuds de coordonnées $\vec{j}_1$ et $\vec{j}_2$ est:

$$L = \vec{1}_H \, \vec{j}_2 - \vec{1}_H \, \vec{j}_1 - N + 1 \qquad \vec{1}_H = \sum_{i=1}^{N} \vec{h}_i$$

où $\vec{1}_H$ est le vecteur diagonal principal de H. Cela vient du fait que tous les chemins allant d'un supernoeud s_1 à un autre supernoeud s_2 dans le graphe d'ordonnancement des supernoeuds sont de longueurs égales (distance de Manhattan). On obtient le résultat précédent en additionnant les longueurs des arêtes dans la base P. Ces longueurs sont les coordonnées de $\vec{j}_2 - \vec{j}_1$, à une unité près pour chaque dimension de l'espace discret. La forme linéaire qui les additionne a toutes ses coordonnées égales à un dans la base duale de P, qui est justement H.

Deux exemples de partitionnements possibles sont illustrés en figure 2.4.2. Le cône de dépendance est défini par deux vecteurs $\vec{d}_1$ et $\vec{d}_2$:

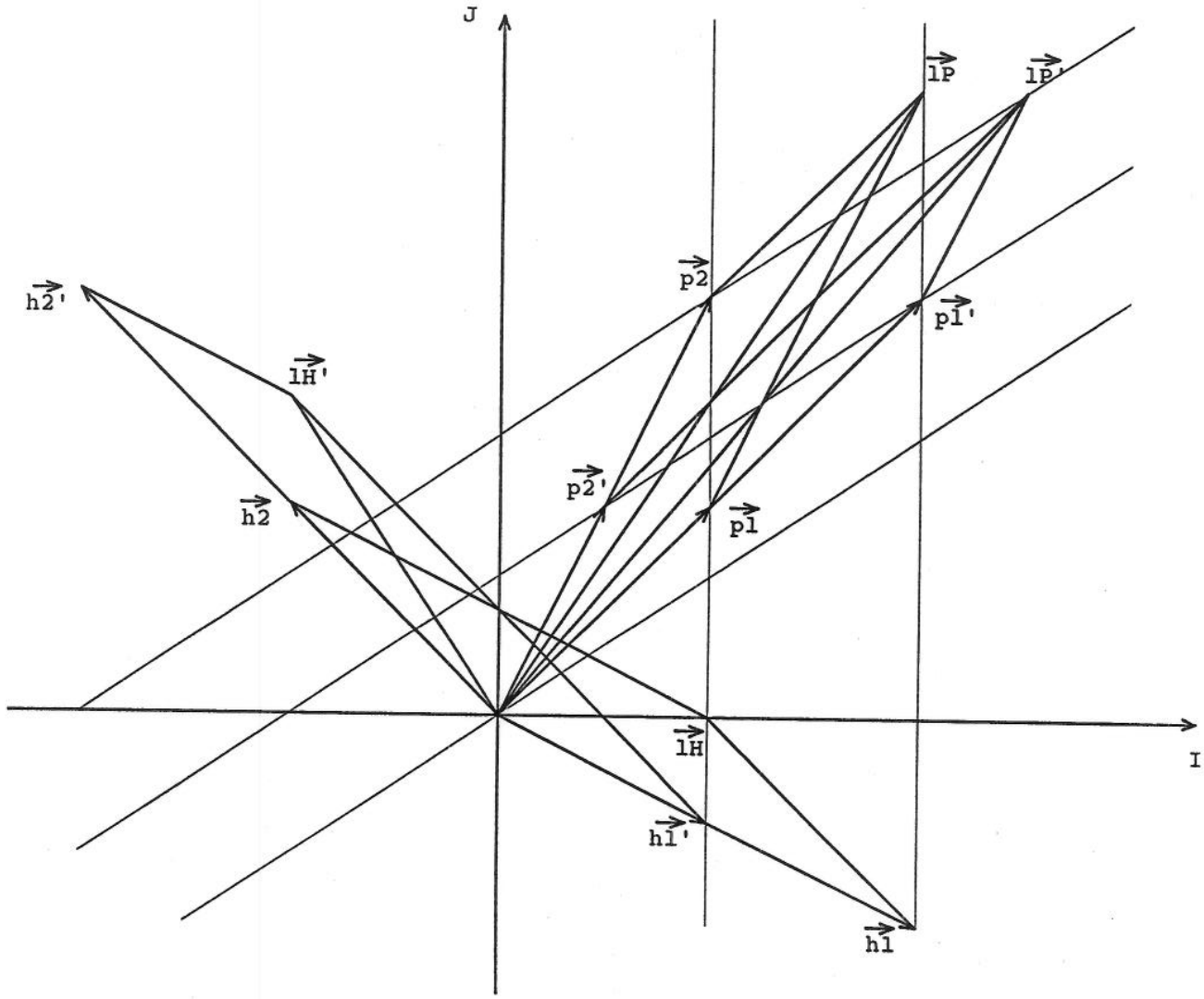
$$\vec{d}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \vec{d}_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Ces deux vecteurs ne sont pas représentés pour ne pas encombrer la figure mais $\vec{d}_1$ (resp. $\vec{d}_2$) est colinéaire à $\vec{p}_1$ et $\vec{p}_1{}'$ (resp. $\vec{p}_2$ et $\vec{p}_2{}'$). En passant de la base $P = (\vec{p}_1, \vec{p}_2)$ à la base $P' = (\vec{p}_1{}', \vec{p}_2{}')$, la forme des supernoeuds résultants est à peine changée tandis que le vecteur d'itération $\vec{1}_H$ est presque complètement retourné. P provoque des itérations selon les I croissants tandis que P' génére un ordonnancement à I décroissants. Les fronts successifs des deux partitionnements sont représentés par les droites les plus minces sur la figure 2.4.2. Elles permettent de vérifier que la formule de longueur donnée par le théorème précédent s'applique bien.

2.4.3. Les différentes bases considérées

A directions fixées, plusieurs bases de partitionnement seront considérées suivant les objectifs:

- P représente une base de volume unité dans les chapitre 4, 5 et 6: $det(P)=1$; en conséquence $det(H)$ est aussi égal à un; ce type de base permet de comparer des partitionnements à volume de supernoeud fixe; on peut en dériver d'autres bases vérifiant la même condition en multipliant les vecteurs de base $\vec{p}_i$ par des coefficients λ_i tels que $\Pi \lambda_i = 1$; dans le chapitre 3, P représente



Relation entre les deux types de partitionnements

- Figure 2.4.2 -

simplement la base de partitionnement, connue numériquement;

- P_1 est la base normée: $\|\vec{p_i}\| = 1$; ce type de base permet d'effectuer une étude de l'influence de l'angle du cône de partitionnement sur les performances;
- $P^I = Min\ P'$ t.q. P' ait tous ses coefficients entiers et respecte les directions de P. P^I s'obtient en partant d'une base formée de vecteurs entiers et en réduisant chacun d'eux par le PGCD de ces composantes: cette base donne le plus petit partitionnement à supernoeuds identiques; partant du noeud origine d'un supernoeud, on aboutit toujours à un noeud au bout d'une ou plusieurs arêtes, qui est donc noeud origine d'un autre supernoeud, identique à une translation

près; tous les partitionnements qui respectent l'identité par translation des supernoeuds dans l'espace discret en dérivent.

2.5. Facteur d'échelle λ

Le facteur d'échelle λ est un facteur d'homothétie qui sera utilisé en optimisation une fois la forme des supernoeuds fixée par une matrice à coefficients réels de type P. Il restera alors à ajuster la taille des supernoeuds aux ressources (taille du cache, nombre de processeurs,...), à l'overhead (lancement de tâche, code de contrôle, etc...) et au domaine d'itération, puis à repasser à une matrice à coefficients entiers, derivée de P^I, pour attaquer la génération de code.

Le facteur d'échelle λ devrait être un entier strictement positif, mais il sera fréquemment considéré comme un réel pour pouvoir appliquer des techniques de dérivation.

2.6. Conclusion

Nous avons montré qu'il était possible de définir des partitionnements non triviaux des espaces d'itération générés par des boucles parfaitement imbriquées, à une profondeur quelconque. Ces partitionnements vérifient les trois conditions de synchronisation externe, d'unicité et de pavage qui sont nécessaires pour permettre une analyse formelle du temps d'exécution et l'automatisation de la génération de code.

Les deux approches opposées du partitionnement par découpage au moyen d'hyperplans et par agglomération se sont révélées duales et complémentaires puisque la première va nous servir dans les chapitres consacrés à l'optimisation tandis que la seconde va être utilisée immédiatement dans le chapitre 3, qui traite de la génération de code.

Quoique cet ordre puisse sembler surprenant à première vue, il faut commencer par montrer comment on peut générer le code partitionné avant d'optimiser le partitionnement. En effet, ce n'est pas à proprement parler le partitionnement qu'il faut optimiser, mais le temps d'exécution du code qu'on saura produire.

References

1. M. Cosnard, J. M. Muller, Y. Robert, and D. Trystram, "Communication Costs Versus Computation Costs in Parallel Gaussian Elimination," *Int. Workshop on Parallel Algorithms and Architectures*, pp.19-29 (April 1986).

2. R. G. Cytron, "Compile-Time Scheduling and Optimization for Asynchronous Machines," PhD Thesis, Report No. UIUCDCS-R-84-1739, University of Illinois at Urbana-Champaign (1984).

3. M. Dubois, J. L. Gaudiot, and N. Tohme, "Analysis of Partitioning and Allocation Techniques in a Cluster-Based Machine," Tech. Report CRI-85-39, USC Computer research Institute, Los Angeles (Nov. 1985).

4. N. Halbwachs, "Détermination automatique de relations linéaires par les variables d'un programme," Thèse 3ème cycle, I.N.P., Grenoble (1979).

5. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

6. J.-K. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," PhD Thesis, Report UIUCDCS-R-86-1259, University of Illinois at Urbana-Champaign (March 1986).

7. C. Renvoise, G. Rouquié, J. C. Cottet, and P. Feautrier, "Détection automatique d'opérations parallèles dans les programmes," Note de programmation, C.I.I. Honeywell Bull, Louveciennes (Nov. 1980).

8. R. Williams, *The Geometrical Foundation of Natural Structure. A Source Book of Design*, Dover, New-York (1979).

9. M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD. Thesis, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign (1982).

3. GENERATION DE CODE

Générer automatiquement du code Fortran efficace suivant la technique de partitionnement présentée dans le chapitre 2 n'est pas chose aisée, surtout quand il faut prévoir l'utilisation d'une unité vectorielle pour l'exécution d'un supernoeud et quand il faut gérer explicitement les mouvements de données à travers la machine. Pouvoir le faire est cependant important, quoique le partitionnement puisse être effectué à la main, pour deux raisons. Tout d'abord cela permettrait de valider (ou d'invalider) l'intérêt du partitionnement sur de nombreux exemples. Ensuite, cela montrerait que la génération automatique de code est possible même pour les multiprocesseurs à hiérarchie mémoire compliquée qui apparaissent de plus en plus nombreux sur le marché.

Pour limiter le nombre de problèmes à résoudre et essayer d'obtenir un résultat exploitable, l'étude de la génération automatique a été limitée à la dimension 2. Traiter la dimension 3 n'aurait pas été simple car géométrie plane et géométrie dans l'espace diffèrent souvent. Par exemple, les arêtes et les faces sont confondues en dimension 2. Et la visualisation des figures est aussi plus difficile.

Dans ce chapitre, on suppose qu'un partitionnement particulier a été choisi. Il est défini par une matrice P qui intègre le facteur d'échelle λ. Mais les difficultés ou possibilités d'optimisation qui vont être rencontrées pendant la génération de code seront exploitées dans les chapitres ultérieurs pour effectuer le choix d'un partitionnement optimal. L'ordre des chapitres ne correspond donc pas à l'ordre de leur utilisation dans un outil automatique: le choix d'un partitionnement optimal y précéderait alors la génération, mais s'effecturait en connaissant les possibilités du générateur de code.

Nous allons d'abord étudier l'ordonnancement des supernoeuds, problème qui ne mérite pas de longs développements étant donné la régularité du graphe de synchronisation et que nous avons délibérément laissé en dehors du champ de ce travail. Il faut néanmoins en choisir un ordonnancement pour donner la structure générale du code que nous allons générer.

Nous pourrons ensuite montrer comment on peut générer les informations nécessaires à un balayage exact, à la fois exhaustif et strict: aucun supernoeud fantôme, dont l'intersection avec le domaine d'itération serait vide, ne doit apparaître, mais aucun supernoeud, ne comptant ne serait-ce qu'un seul noeud du domaine d'itération ne doit être oublié. Ce balayage doit de plus respecter l'ordonnancement parallèle choisi dans la section précédente.

Il ne faut pas oublier de tester la nature de chaque supernoeud car les frontières du domaine d'itération produisent de nombreuses formes de supernoeuds partiels[†]. Il faudra donc générer un test pour chaque forme, y compris celle des supernoeuds complets, après avoir identifié le nombre de formes

[†] Rappelons qu'un supernoeud complet est un supernoeud dont tous les noeuds appartiennent au domaine d'itération et qu'un supernoeud partiel est un supernoeud dont certains noeuds appartiennent au domaine d'itération et d'autres non.

que produit chacune des quatre frontières du domaine.

Ceci fait, il nous faut à nouveau générer un balayage exact et, si possible, vectoriel de l'ensemble des noeuds contenus dans un supernoeud complet ou dans n'importe quel type de supernoeud partiel. Après avoir présenté plusieurs méthodes utilisées dans le passé pour résoudre des problèmes similaires (Index Set Shifting, décomposition en domaine triangulaire, DOACROSS), nous proposerons un nouvel algorithme basé sur la théorie des polyèdres convexes et une variante permettant d'obtenir des résultats équivalents à la *triangularisation* de Kuhn.

A ce moment, nous disposerons de l'ensemble du code de contrôle et il sera possible, à un changement de base près, de réintroduire le corps des boucles initiales dans chacune de nos structures de contrôle, ceci dans le cas des machines à cache, à mémoire virtuelle, ou n'ayant qu'un niveau de mémoire. Pour les autres architectures, nous étudierons et nous présenterons des solutions pour les problèmes qui restent, à savoir: la génération de copies locales, la modification des références contenues dans le corps de boucle et la génération du code de copie.

3.1. Choix d'un ordonnancement global: les fronts

3.1.1. Solutions possibles

L'ensemble des solutions possibles est trop grand pour que nous puissions prétendre l'explorer exhaustivement. Partant des travaux effectués sur les problèmes de synchronisation et des synchronisations disponibles sur les machines vendues à l'heure actuelle, nous allons présenter quatre solutions possibles:

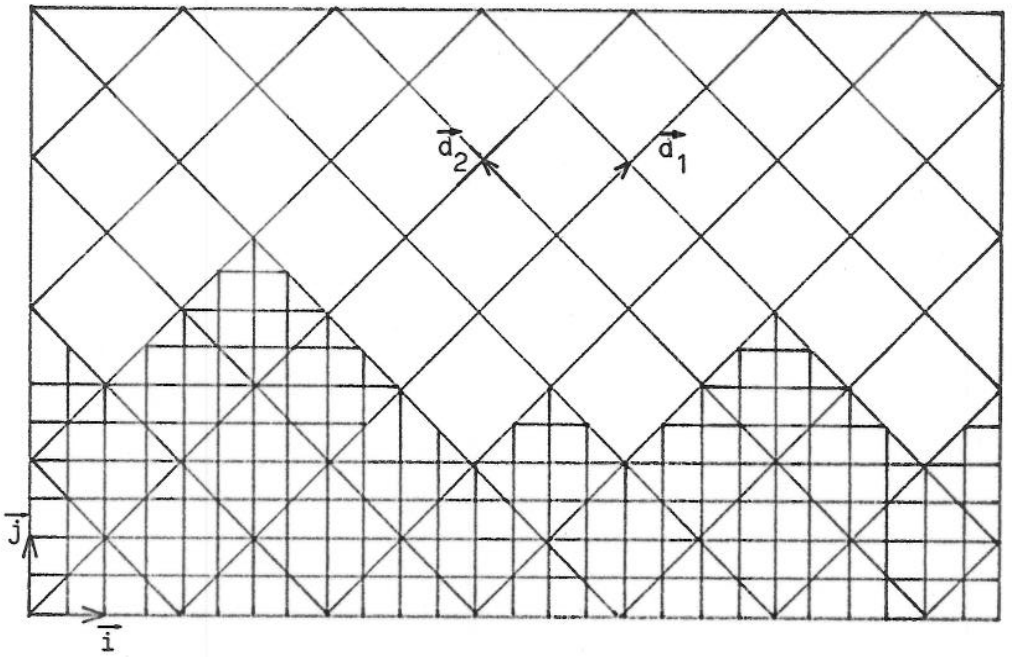
- la synchronisation par flot de données;
- la synchronisation par héritage, qui est une variante de la précédente;
- la synchronisation par fronts, qui découle directement des synchronisations disponibles sur le matériel;
- et la synchronisation par fronts avec recouvrement, qui présente un peu moins de rigidité que la précédente tout en restant peu coûteuse.

La synchronisation par flot de données (data flow ou data driven) est maintenant connue depuis de nombreuses années: une instruction peut être exécutée dès que toutes ses données d'entrée sont disponibles. Les réalisations ou projets de réalisations matérielles basées sur cette méthode, dans les universités[4, 11, 8] ou les entreprises[9] , n'ont pas donné les résultats escomptés et l'activité dans le domaine semble avoir diminué depuis le début des années 80 comme le montre l'ancienneté des références bibliographiques citées par Srini dans un article récent[27] . Ce principe, parfois aussi appelé *assignation unique*[6] , a aussi été appliqué à la définition de nouveaux langages de programmation[2, 21] comme LAU[14] ,Id[3] , SISAL[24] ou VAL[1, 23] . Là aussi, les espoirs des années 70 ne se sont pas concrétisés, partiellement à cause de l'absence de machine à cadencement par les données, et l'attention s'est tournée vers les langages fonctionnels. Cependant, la synchronisation par les données est encore utilisée par les analystes numériciens pour gérer le parallélisme dans les multiprocesseurs comme le Cray X-MP[16, 17] . La lourdeur du mécanisme de synchronisation par les données est en effet acceptable relativement à l'overhead des systèmes de multitasking. Dans le cas des supernoeuds, il faudrait entretenir des listes de supernoeuds prêts[†] à $1/N$, $2/N$,..., 1 (un supernoeud "prêt à 1" est exécutable), vraisemblablement dans une section critique, et cela prendrait un temps non négligeable, sans compter l'espace mémoire nécessaire à leur stockage.

L'avantage essentiel de la synchronisation par les données est le grand volume de parallélisme dégagé: un processeur ne peut être inactif que s'il n'y a réellement rien à faire. Mais le choix aléatoire des supernoeuds par les processeurs, qui peut conduire par exemple à l'état de la figure 3.1.1.a, ne permet pas de réutiliser les données locales, d'exploiter l'héritage. Sur cette figure, comme sur les suivantes, les supernoeuds hachurés sont des supernoeuds terminés. Les supernoeuds en cours d'exécution ne sont pas

[†] Le nombre de prédécesseurs d'un supernoeud est égal à la dimension de l'espace.

distingués. Il pourrait y avoir de un à quatre processeurs actifs.

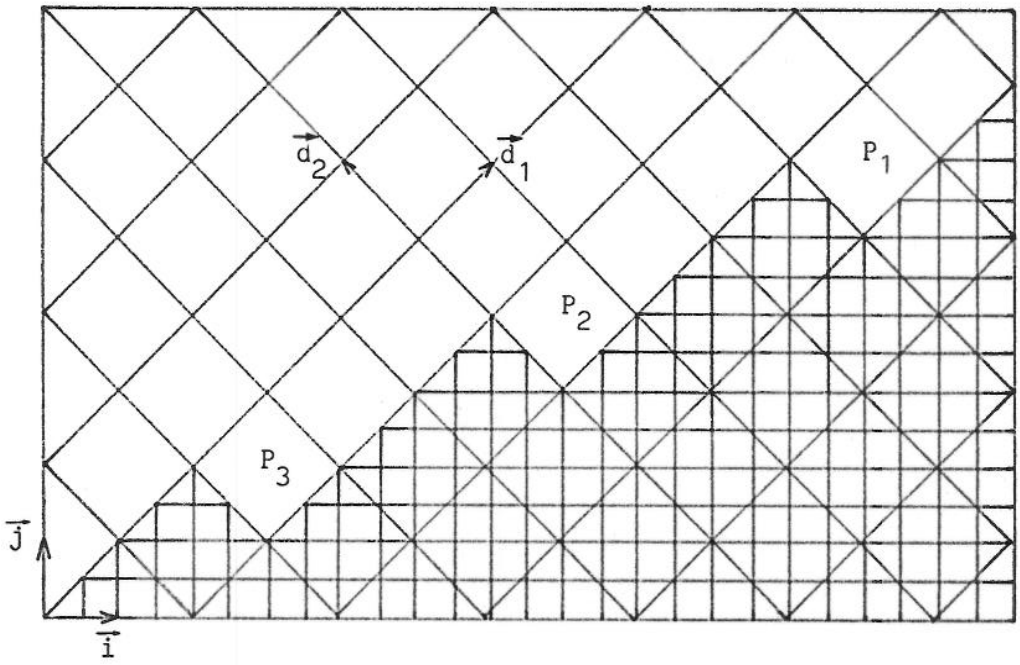


Ordonnancement Data Flow

- Figure 3.1.1.a -

La synchronisation par héritage vise à corriger ce manque de localité en limitant la liberté de choix des processeurs. Ils devront essayer d'abord de sélectionner un supernoeud successeur de leur supernoeud courant s'il en existe un qui soit libre. Si tout va bien, cela conduit au dessin de la figure 3.1.1.b où trois processeurs sont actifs. Pour ne pas bloquer inutilement un processeur, il faut lui permettre d'aller chercher ailleurs du travail s'il ne lui en reste plus, soit qu'il ait atteint une frontière du domaine d'itération, soit qu'il ait rattrapé un autre processeur. Par exemple P2 pourrait rattraper P1 et devoir exécuter l'unique supernoeud exécutable, le triangle vertical situé au coin inférieur gauche. Malgré un parallélisme potentiel important, cette politique conduit à une séquentialisation des exécutions. Suivant la gestion des files de supernoeuds, une synchronisation dataflow pourrait avoir le même effet pervers.

C'est pourquoi nous proposons une solution beaucoup plus simple, qui, si elle est câblée comme dans l'Alliant FX-8[15] , peut avoir un overhead de synchronisation quasi-nul. Il s'agit des classiques DOALL et DOSEQ. A chaque pas d'itération séquentiel (DOSEQ), on libère toutes les itérations du DOALL et on ne passe à l'itération séquentielle suivante que quand toutes les itérations du DOALL sont terminées. Le mécanisme est similaire au classique *fork-join*. Cela produit des fronts d'avancée des itérations comme sur la figure 3.1.1.c et peut provoquer une sous-utilisation des processeurs. Sur cette même figure, deux

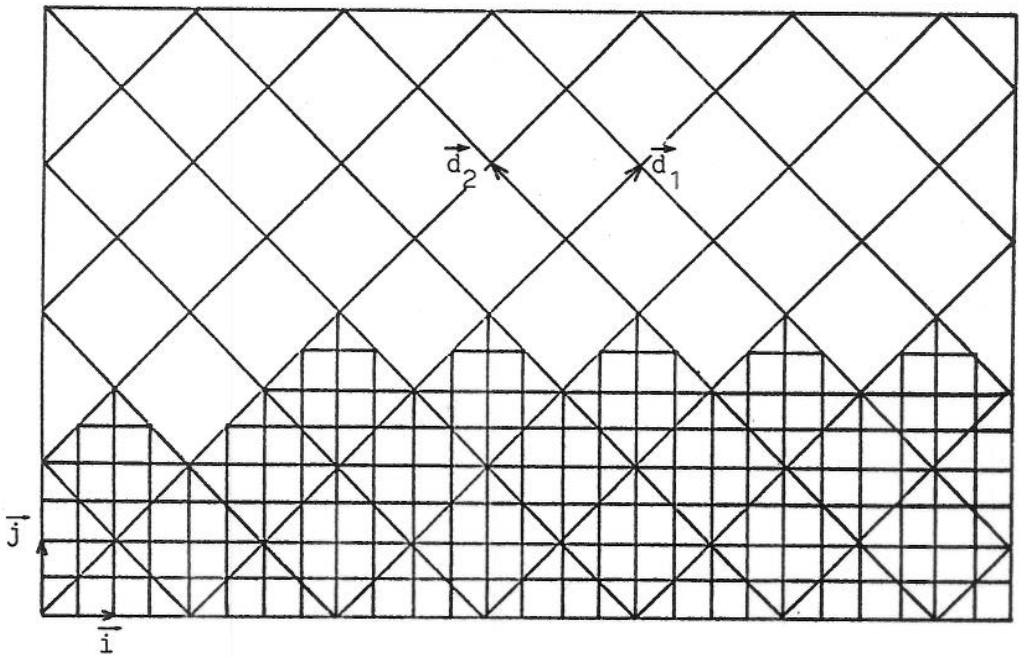


Ordonnancement maximisant l'héritage

- Figure 3.3.1.b -

processeurs au plus peuvent être actifs, exécutant les deux supernoeuds les plus à gauche de la quatrième itération. Cependant, ceci ne se produit que pour le reste des supernoeuds après division par le nombre de processeurs. L'effet sera donc d'autant plus faible que le nombre de supernoeuds par front sera grand devant le nombre de processeurs. C'est un des facteurs qui exerceront une influence sur le choix d'un partitionnement au chapitre 6.

Pour pallier cet inconvénient, il faut pouvoir commencer l'itération $n+1$ avant la fin de l'itération n. Ceci est possible si chaque front est parcouru d'une manière régulière, de gauche à droite par exemple. Les supernoeuds non terminés de l'itération n se trouvent donc sur la droite et n'interdisent pas le lancement de l'exécution des premiers supernoeuds de l'itération $n+1$ qui sont situés sur la gauche puisque les deux prédecesseurs de ces supernoeuds se trouvant sur la gauche sont terminés depuis longtemps. Ceci n'est vrai que pour des fronts suffisamment longs. Nous appelons cet ordonnancement, le quatrième et dernier envisagé, ordonnancement avec recouvrement de fronts (cf. figure 3.1.1.d). Il nécessite un peu plus de synchronisations que le précédent mais ressemble fort à un DOACROSS[10] et doit donc pouvoir être implémenté sans overhead grâce à des synchronisations matérielles comme celles de l'Alliant FX-8.

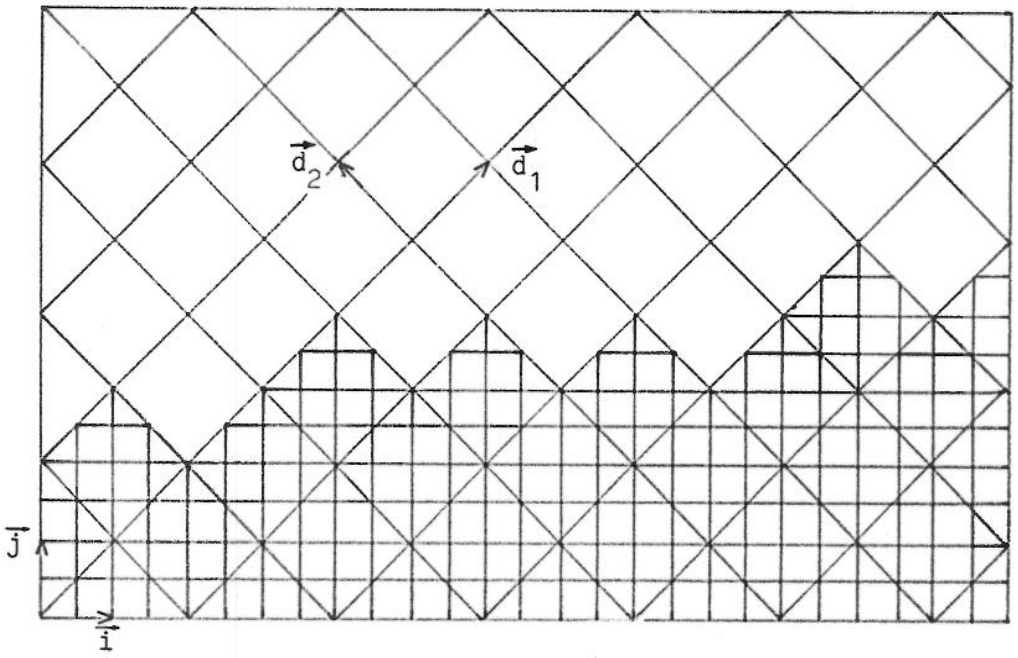


Ordonnancement par fronts

- Figure 3.1.1.c -

Dans le cas où la dimension N est supérieure à 2, on peut essayer de diminuer le nombre de synchronisations générées par les $N-1$ boucles DOALL imbriquées en en faisant un gigantesque mais unique DOALL selon une technique proche de la linéarisation des tableaux. Cette transformation est parfois appelée *loop coalescing*. Malheureusement, elle introduit des divisions dans les calculs d'indices. On peut envisager d'amortir ces calculs compliqués sur l'ensemble des calculs d'un supernoeud en ajoutant des variables temporaires, ce qui n'est pas rentable dans le cas général de loop coalescing mais vraisemblablement acceptable pour des supernoeuds. De toute façon, nous n'insisterons pas sur ce point car ce chapitre 3 sur la génération de code est restreint au cas de la dimension 2.

Le choix entre ces diverses solutions ne se résume pas à un simple compromis entre la bonne charge des processeurs et le coût de l'ordonnancement. D'une part, la synchronisation dataflow peut avoir des effets pervers. D'autre part, on souhaite pouvoir calculer analytiquement le temps d'exécution global.



Ordonnancement avec recouvrement de fronts

- Figure 3.1.1.d -

3.1.2. Choix de la solution "fronts"

Deux raisons militent en faveur de la solution "fronts", éventuellement relaxée en "recouvrement de fronts":

- les synchronisations nécessaires sont simples et, en conséquence, l'overhead est minime (il suffit de partager un compteur) et la génération de code est immédiate;
- le calcul du temps d'exécution d'un front est simple et l'intégration, même sur un domaine triangulaire, est possible.

Les inconvénients de cette solution peuvent être considérés comme négligeables ou correctibles:

- le blocage de processeurs par un front, alors que des supernoeuds de l'itération suivante sont prêts, aura un effet négligeable, comme nous venons de le dire en § 3.1.1, si le nombre de supernoeuds par front est très grand devant le nombre de processeurs ou bien si le nombre de supernoeuds par front est un multiple du nombre de processeurs[†] ; sur les domaines triangulaires ou plus généralement à degré

[†] L'Alliant pouvant fonctionner en mode dégradé avec un nombre quelconque de processeurs entre 1 et 8, il faudrait conserver 8 modules exécutables...

de parallélisme non constant, cette approximation ne sera pas valable sur tout le domaine, mais il reste encore possible de mettre en oeuvre les recouvrements de fronts;

- les supernoeuds partiels, qui ne sont pas inclus dans le domaine d'itération, n'ont pas un temps d'exécution égal à celui des supernoeuds complets; bien que le nombre de noeuds à calculer soit plus faible, ils peuvent avoir des temps d'exécution supérieurs à cause des tests supplémentaires qu'il faut y ajouter pour vérifier que chaque noeud appartient bien au domaine d'itération et à cause du mode de fonctionnement qu'imposent ces tests: le passage d'un mode d'exécution vectoriel pour les supernoeuds complets à un mode d'exécution scalaire pour les supernoeuds partiels peut être catastrophique; il faut donc prévoir une optimisation de leur code et, si elle est réussie, une union de supernoeuds partiels pour obtenir un temps d'exécution égal à celui des supernoeuds complets;

En conclusion, si, comme nous le pensons, les programmes scientifiques qu'il faudra optimiser ont toujours de très larges domaines d'itération, nous estimons que la solution "fronts" fournira des résultats quasi-optimaux car le domaine d'itération pourra être découpé en un nombre de supernoeuds très supérieur au nombre de processeurs.

3.1.3. Squelette de code

Le squelette du code que nous allons générer va donc avoir l'allure suivante:

```
Séquentiellement, pour chaque front faire
  En parallèle, pour chaque supernoeud du front faire
    Si le supernoeud est complet, code pour les supernoeuds complets
    Sinon si le supernoeud a une intersection non vide avec la frontière 1,
      Déterminer le type de forme
      Appeler le code correspondant
    ...
    ...
    Si le supernoeud a une intersection non vide avec la frontière 4,
      Déterminer le type de forme
      Appeler le code correspondant
  Fin
Fin
```

Nous allons donc déterminer dans un premier temps l'ensemble des fronts relatifs à un domaine d'itération et à un partitionnement ainsi que l'ensemble des supernoeuds de chaque front et calculer les tests qu'il faut appliquer pour sélectionner le code approprié pour chaque supernoeud. C'est ce que nous appelons la génération du code de contrôle global.

3.2. Génération du code de contrôle global

3.2.1. Base de parcours G

Il nous faut trouver un moyen d'énumérer l'ensemble des supernoeuds appartenant au domaine d'itération, front par front comme nous venons de le voir. Les boucles DO imbriquées générant des points qu'on peut considérer comme les points à coordonnées entières d'un espace vectoriel, il semble intéressant de voir s'il n'existerait pas une base de génération, que nous appellerons G, qui contienne parmi ses vecteurs de base un vecteur colinéaire à la direction parallèle des fronts et un vecteur assurant l'ordonnancement de l'itération sur les fronts. Cette base devrait de plus associer à chaque point entier de l'espace initial un point entier et réciproquement, de manière que le parcours des supernoeuds soit exact.

Cette base doit être constituée d'une part d'une direction assurant le respect des dépendances, qui va permettre de définir la boucle DOSEQ externe et de balayer les fronts successifs mais qui n'est pas forcément $\vec{1}_H$ puisqu'il ne s'agit que de numéroter les supernoeuds ($\vec{1}_H \ \vec{g}_1 > 0$), et d'autre part d'une direction parallèle, qui va correspondre à la boucle DOALL parcourant chaque front ($\vec{1}_H \ \vec{g}_2 = 0$). Il faut donc qu'un des deux vecteurs de base de G soit orthogonal à $\vec{1}_H$, la direction d'itération. Soit G la matrice de passage de la base G à la base de partitionnement P.

Deux problèmes supplémentaires se posent, qui vont ajouter de nouvelles contraintes sur G. Le premier consiste à trouver le polyèdre convexe ou l'ensemble de polyèdres convexes qui représente le domaine d'itération dans la base G. En effet, il n'est pas possible de générer directement les expressions d'indices de boucles DO qui permettent de parcourir n'importe quel polyèdre convexe. Il faut a priori se restreindre aux polyèdres triangulaires (cf. § 1.4.2) et être prêt à découper le domaine d'itération en plusieurs domaines triangulaires dans la nouvelle base G. Chaque domaine pourra être parcouru par un ensemble différent de boucles imbriquées. Une première possibilité consisterait à choisir une base G conservant la triangularité du domaine d'itération, mais ceci n'est possible que dans des cas particuliers. Une deuxième méthode de calcul de la décomposition est donnée dans le PhD de Kuhn[22] mais elle est assez lourde et repose sur la théorie des polyèdres convexes. Une troisième possibilité, celle que nous adopterons, serait d'accepter un domaine pseudo-triangulaire, plus facile à déterminer que la décomposition en domaines triangulaires.

On peut estimer que ce premier problème est un faux problème et qu'il suffirait d'approximer le domaine d'itération par un hypercube dans la base G. La vérification de l'appartenance complète ou partielle d'un supernoeud au domaine d'itération ne devrait représenter qu'un overhead négligeable dès que les supernoeuds comprennent quelques centaines de noeuds comme c'est toujours le cas quand on veut pouvoir utiliser des instructions vectorielles pour coder un supernoeud. Ce n'est pas notre point de vue car la détermination d'un domaine pseudo-triangulaire est presque aussi simple que la détermination des dimensions de l'hypercube englobant et que nous avons appris expérimentalement à nous méfier de toutes les sources d'overhead déclarées négligeables par un raisonnement fondé sur des O(n).

Le deuxième problème consiste à établir une bijection entre les points à coordonnées entières (appelés *points entiers* par la suite) dans la base de partitionnement P, qui correspondent tous à des origines de supernoeuds, et les points à coordonnées entières de la base G. Les points entiers dans P et non entiers dans G ne seront pas retrouvés par des boucles DO dans G : le supernoeud correspondant ne sera pas exécuté. Et les points entiers dans G et non entiers dans P ne correspondront à aucun supernoeud. La bijection est nécessaire pour ne pas avoir à ajouter un test de filtrage qui apporterait de l'overhead supplémentaire. Ceci impose que $|\,det(G)\,|$ égale un et que tous les coefficients de G soient entiers. Autrement dit, que G soit unimodulaire au sens de la programmation mathématique[25] .

Les considérations précédentes sont valables quelle que soit la dimension N, mais nous allons maintenant construire une solution pour la dimension 2. Les deux vecteurs de base de P sont notés $\vec{p}_1$ et $\vec{p}_2$. Montrons que la direction parallèle (celle du DOALL) est colinéaire à $\vec{p}_2-\vec{p}_1$:

$$< \vec{1}_H \cdot \vec{p}_2-\vec{p}_1 > = \vec{h}_1\,\vec{p}_2 + \vec{h}_2\,\vec{p}_2 - \vec{h}_1\,\vec{p}_1 - \vec{h}_2\,\vec{p}_1 = 0+1-1+0 = 0$$

par définition de H relativement à P (cf. 2.4.2).

La base G définie par:

$$G = (\vec{p}_1,\vec{p}_2-\vec{p}_1)$$

(on pourrait remplacer $\vec{p}_1$ par $\vec{p}_2$ et/ou $\vec{p}_2-\vec{p}_1$ par $\vec{p}_1-\vec{p}_2$) possède bien la direction parallèle et de plus assure la bijection entre origines de supernoeuds et points à coordonnées entières. En effet, G et G^{-1}, les matrices de passage de G à P et de P à G, ont tous leurs coefficients entiers et des déterminants égaux à un:

$$G = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \qquad G^{-1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

La matrice permettant de passer de la base G à la base initiale B est PG. C'est elle qui va permettre de savoir si l'origine (λ_1, λ_2) d'un supernoeud appartient au domaine d'itération. Soit λ_1 le numéro de front, tandis que λ_2 est le numéro du supernoeud dans le front. Ce sont donc nos deux nouveaux indices de boucle. Nous avons caractérisé le domaine d'itération au chapitre 1 par une matrice A et un vecteur de constantes $\vec{\alpha}$. Le test sur λ_1 et λ_2 semble donc pouvoir s'écrire:

$$APG \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} \leq \vec{\alpha} = \begin{pmatrix} BSI \\ BSJ \\ -BII \\ -BIJ \end{pmatrix}$$

où BSI et BSJ (resp. BII et BIJ) sont les bornes supérieures (resp. inférieures) des deux boucles initiales.

Si APG est une matrice triangulaire comme A (A étant triangulaire par définition des boucles

DO), la base G définit un parcours immédiatement encodable du domaine d'itération. Sinon, nous proposons d'utiliser la méthode de pseudo-triangularisation que nous décrirons en détail dans la section 3.2.4 ou bien une décomposition à la Kuhn, mise en oeuvre avec une méthode moins lourde que celle qu'a proposée ce dernier.

Après une normalisation des coefficients diagonaux, le ou les systèmes prennent la forme suivante:

$$\begin{pmatrix} 1 & 0 \\ \alpha & 1 \\ -1 & 0 \\ \beta & -1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} \leq \begin{pmatrix} BS\lambda_1 \\ BS\lambda_2 \\ -BI\lambda_1 \\ -BI\lambda_2 \end{pmatrix}$$

Les bornes $BS\,\lambda_1$, etc... , qui, comme les bornes initiales BSI, etc..., sont des expressions linéaires de variables scalaires du programme, doivent être calculées (cf § 3.2.4).

Voyons ce qui se passe avec l'exemple de la figure 3.2.1, où les vecteurs de dépendance sont:

$$\vec{d}_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \vec{d}_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \vec{d}_3 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

et où le domaine d'itération est rectangulaire, on a:

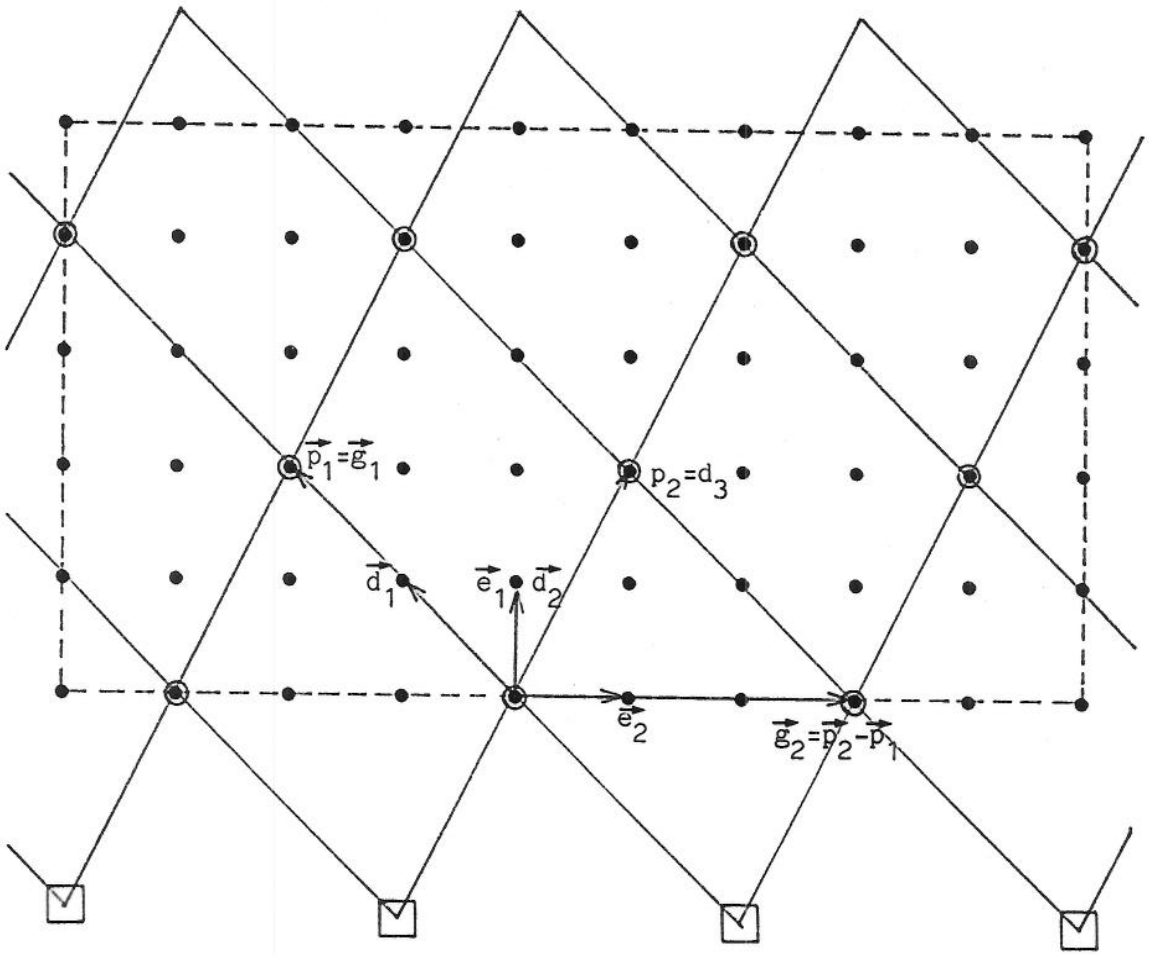
$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \quad P = \begin{pmatrix} 2 & 2 \\ -2 & 1 \end{pmatrix} \quad APG = \begin{pmatrix} 2 & 0 \\ -2 & 3 \\ -2 & 0 \\ 2 & -3 \end{pmatrix}$$

Par chance, APG définit un système triangulaire qui peut être traduit en pseudo-Fortran:

DO $2*\lambda_1 =$ BI1, BS1
  DO $3*\lambda_2 = 2*\lambda_1$+BI2, $2*\lambda_1$+BS2

La présence de coefficients en partie gauche reflète le problème des supernoeuds partiels (leurs origines sont représentées par des carrés sur la figure 3.2.1.). Le calcul en nombres entiers† des expressions de bornes, qui ferait apparaître (BI1+1)/2, BS1/2, $(2*\lambda_1$+BI2+2)/3 et $(2*\lambda_1$+BI2)/3, ne donnerait que l'ensemble des supernoeuds dont l'**origine** est dans le domaine d'itération. Cet ensemble est sans intérêt car l'ensemble des noeuds qu'il contient n'est pas comparable par inclusion avec le domaine d'itération. Nous reviendrons sur ce point en § 3.2.4, où nous verrons aussi comment traiter le cas général des matrices APG non triangulaires.

† Rapellons que DO $\alpha\lambda$ = BI, BS devient DO λ = (BI+ α -1)/α, BS/α et que DO $\alpha^{-1}\lambda$ = BI, BS devient DO λ = αBI, αBS+α−1. Il faut prendre des précautions quand des nombres négatifs apparaissent dans les divisions entières, car la division entière, comme le modulo, a plusieurs définitions[7] . Pour nos expériences sur l'Alliant, nous avons dû réécrire nos propres fonctions *mydiv* et *mymod*.



Recouvrement du domaine d'itération par des supernoeuds

- Figure 3.2.1 -

Notons enfin que lors des essais effectués aux Etats-Unis, décrits rapidement en § 7.10, nous avons utilisé un parcours à base de vecteurs d'offset. Ce deuxième type de parcours, peut-être plus efficace mais surtout plus intuitif et plus simple à mettre en oeuvre, sera décrit en 3.3.1.3.

3.2.2. Les différents types de supernoeuds partiels

Comme on le verra en essayant de générer le code d'un supernoeud partiel (3.4), la part du temps d'exécution dévolue aux supernoeuds partiels n'est pas toujours négligeable. L'optimisation de ce code nécessite la connaissance du domaine d'itération du supernoeud partiel. Un type de supernoeud partiel correspond à un domaine particulier, triangle ou polygone irrégulier à quatre ou cinq côtés. Ces types, dont le nombre ne dépend heureusement pas de la surface du domaine d'itération, doivent être identifiés

dans le code de contrôle général pour appeler le bon sous-programme.

Les pentes des frontières du domaine d'itération sont indépendantes des deux directions de partitionnement $\vec{p}_1$ et $\vec{p}_2$ et l'on conçoit aisément que le nombre de supernoeuds partiels de types différents dépende non seulement de l'origine mais aussi de la pente.

Comme tous les coefficients sont rationnels, la périodicité est certaine et la période est trouvée quand, partant d'une origine de supernoeud sur la frontière, on retombe sur une origine de supernoeud sur la même frontière. Si $\vec{f}$ est le vecteur à coefficients entiers minimum donnant la pente de la frontière, la condition précédente s'exprime sous la forme:

$$\alpha\vec{p}_1 + \beta\vec{p}_2 = \gamma\vec{f}$$

Pour obtenir la période le long de la frontière, il faut minimiser, en nombres entiers, α (ou β, car ils sont liés) et le nombre de supernoeuds partiels NSP à considérer est donné par:

$$NSP(\vec{f}) = \mid \alpha_{\vec{f}} \mid + \mid \beta_{\vec{f}} \mid$$

En effet (cf. figure 3.2.2), le nombre de supernoeuds partiels différents est égal au nombre de bordures de supernoeuds traversées en une période. Si une origine de supernoeud appartient à la frontière, deux bordures sont franchies simultanément par définition de l'origine et on a:

$$NSP(\vec{f}) = \mid \alpha_{\vec{f}} \mid + \mid \beta_{\vec{f}} \mid - 1$$

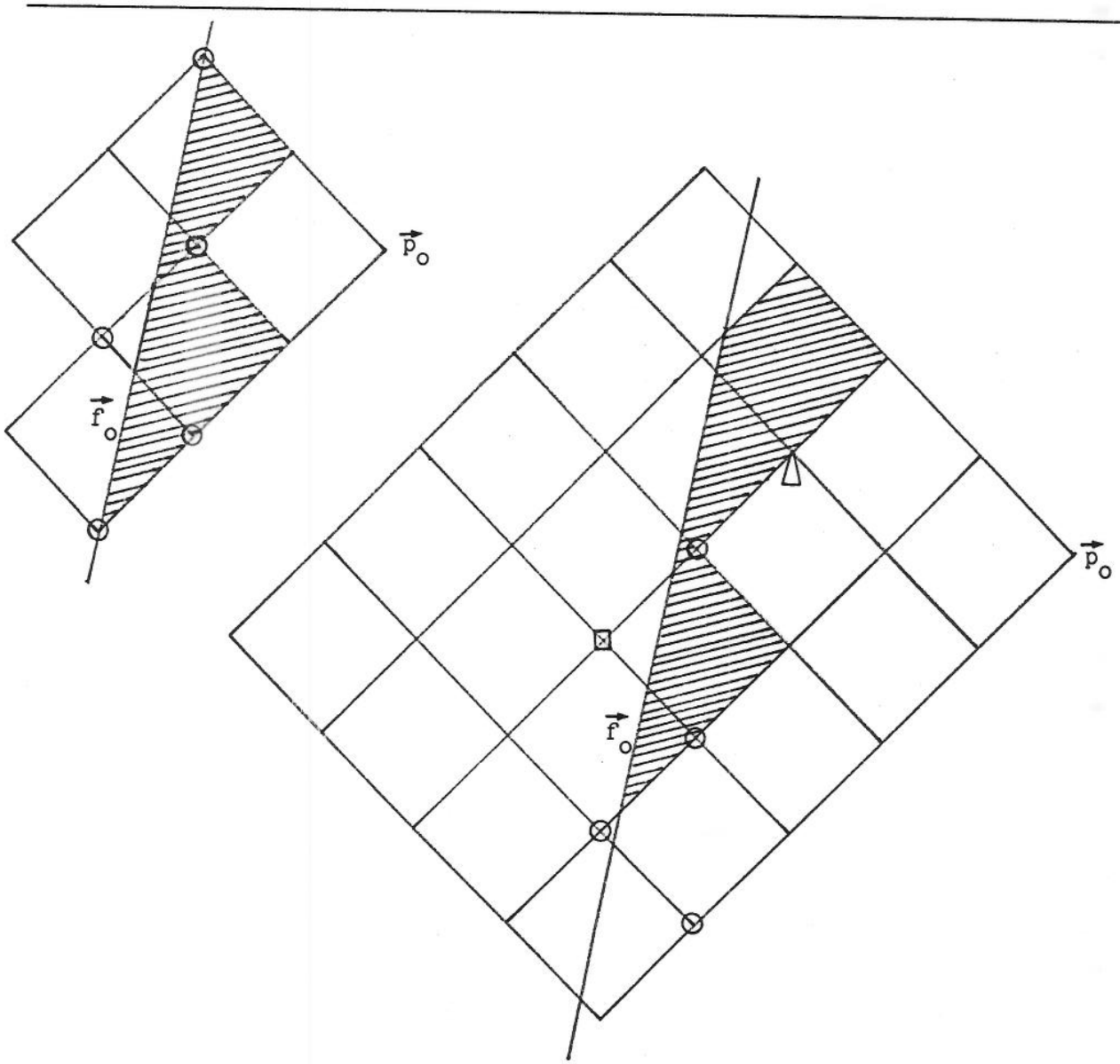
Dans une période, on ne peut avoir qu'une seule origine de supernoeud. S'il y en avait deux, la période serait donnée par leur écart.

Comme $\alpha_{\vec{f}}$ et $\beta_{\vec{f}}$ ressemblent fort aux coordonnées de $\vec{f}$ dans la base P à un coefficient près, la solution se trouve avec la matrice de changement de base associée en posant:

$$\gamma \begin{pmatrix} \alpha_{\vec{f}} \\ \beta_{\vec{f}} \end{pmatrix} = det(P)\, P^{-1} \begin{pmatrix} \vec{e}_1\vec{f} \\ \vec{e}_2\vec{f} \end{pmatrix}$$

Les coefficients γ et $det(P)$ nous permettent de garder tous les termes entiers (sauf γ) et il ne reste plus qu'à réduire $\alpha_{\vec{f}}$ et $\beta_{\vec{f}}$ par leur PGCD.

Les deux expressions donnant le nombre de types de supernoeuds partiels sont un peu simplifiées car, pour tenir compte de la nature discrète des supernoeuds et de la présence d'une inégalité stricte dans leur définition, il faudrait distinguer les frontières appartenant ou non au cône de partitionnement. Le raisonnement précédent vaut pour les frontières qui appartiennent (i.e. dont la direction appartient) au cône de partitionnement, à ceci près que le caractère discret du supernoeud et sa définition par des inégalités strictes permettent d'obtenir le cas le plus favorable même si la frontière est légèrement décalée. La périodicité des domaines continus ne sera plus respectée, mais celle des domaines discrets le sera. Pour l'autre type de frontière, il faut voir si la frontière est *entrante* ou *sortante* vis à vis des



Nombre de types de supernoeuds partiels le long d'une frontière

- Figure 3.2.2 -

vecteurs de partitionnement et dans le deuxième cas modifier la constante de la contrainte pour en faire une contrainte stricte, compatible avec la définition des supernoeuds. En effet, les origines de supernoeud appartenant à la frontière vont correspondre dans le premier cas à des supernoeuds complets et dans le deuxième à des supernoeuds partiels à un noeud. Pour la nouvelle frontière *stricte*, les deux résultats précédents sont exacts.

Un gain de un peut sembler faible, mais il ne faut pas oublier que α et β ont usuellement les valeurs

comme -1, 0 ou 1. Ce gain de un permet alors de diviser le nombre de types de supernoeuds par deux, de simplifier le contrôle et de diminuer le volume de code à générer. Dans les cas les plus favorables (multiplication de matrices), il n'y a même plus de supernoeuds partiels à considérer†.

Enfin, il faudrait pouvoir tester la partialité d'un supernoeud en nombres entiers. Le supernoeud partiel dont l'origine est repérée par un carré sur la figure 3.2.2 est vraisemblablement vide de points entiers, tandis que celui qui est marqué d'un triangle est sans doute complet. La détection de ces cas particuliers permettrait de diminuer l'overhead dû aux noeuds partiels. Nous verrons en § 3.2.4 et en § 3.2.5 qu'il existe des algorithmes pour ce faire.

Dans le cas de la figure 3.2.1, le long de la frontière verticale, on trouve:

$$Det(P)\,P^{-1}\,\vec{f} = \begin{pmatrix} 1 & -2 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$\vec{p}_1 + 2\vec{p}_2 = \begin{pmatrix} 2 \\ -2 \end{pmatrix} + 2 \begin{pmatrix} 2 \\ 1 \end{pmatrix} = 6 \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

donc α=1 et β=2. Le nombre de types de supernoeuds partiels est au moins égal à deux. Il est exactement égal à deux si la frontière contient une origine de supernoeud.

Ceci peut conduire à choisir $\vec{p}_1$ et $\vec{p}_2$ de sorte que:

$$\vec{p}_1 + \vec{p}_2 = \gamma\vec{f} \qquad ou \qquad \vec{p}_1 = \gamma\vec{f} \qquad ou \qquad \vec{p}_2 = \gamma\vec{f}$$

si aucun critère plus important ne conditionne le choix de $\vec{p}_1$ et de $\vec{p}_2$.

3.2.3. Choix de l'origine du partitionnement

Décider de l'existence d'un supernoeud dans une frontière revient à décider de l'existence d'une solution à l'équation diophantienne suivante:

$$\alpha\vec{p}_1 + \beta\vec{p}_2 + \vec{p}_o = \gamma\vec{f} + \vec{f}_o$$

où $\vec{p}_o$ est le vecteur définissant l'origine du partitionnement par rapport à l'origine de la base initiale et où $\vec{f}_o$ est un vecteur désignant un noeud quelconque de la frontière. Chaque frontière compte au moins deux noeuds: les sommets du domaine d'itération qui la définissent. La résolution d'une telle équation est classique dans les tests de dépendance. Banerjee a étudié et résolu le problème dans son PhD[5] .

Mais le problème qui nous intéresse est le **choix** de $\vec{p}_o$. L'équation précédente se ramène à:

$$\vec{p}_o = \vec{f}_o + \alpha'\,\vec{p}_1 + \beta'\,\vec{p}_2$$

† Sauf si les boucles ne sont pas parfaitement imbriquées. Il peut alors y avoir du code à ajouter sur une des deux frontières.

puisqu'on a vu au paragraphe précédent qu'on pouvait exprimer $\vec{f}$ en fonction de $\vec{p}_1$ et de $\vec{p}_2$. Le choix le plus évident pour $\vec{p}_o$ est bien sûr $\vec{f}_o$, qui, s'il correspond à un sommet du domaine d'itération, permet d'obtenir le cas le plus favorable le long de **deux** frontières.

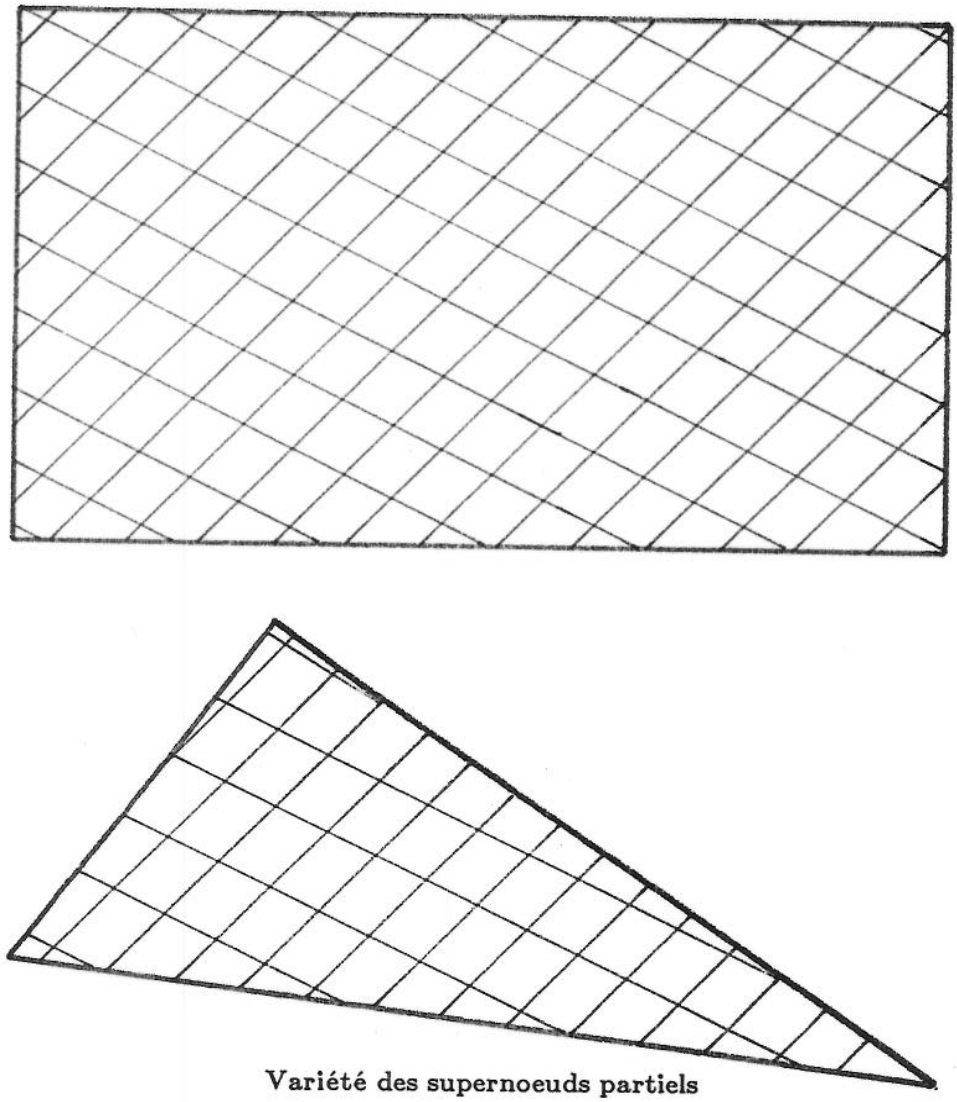
Si de plus le domaine est un parallélogramme (dans le cas général, c'est un trapèze) et si le nombre et la taille des supernoeuds sont bien choisis (comme les calculs s'effectuent en nombres entiers, le bon choix n'est pas toujours possible), le nombre total de types de supernoeuds partiels peut être diminué de deux. Dans le cas de la figure 3.2.1, on profite du parallélisme de la base du domaine d'itération et de $\vec{p}_2$–$\vec{p}_1$, du choix de l'origine du maillage en supernoeuds, et aussi de la hauteur du domaine d'itération pour avoir seulement un type de supernoeuds partiels le long des frontières horizontales et deux types le long des frontières verticales. Avec un vecteur $\vec{p}_1$ deux fois plus court, le nombre de types de supernoeuds partiels le long des verticales n'aurait aussi été que de un. Le faible nombre de supernoeuds complets est dû à la limitation de taille qu'impose la feuille de papier. Un domaine d'itération intéressant pour un superordinateur serait au moins de l'ordre 1024×1024.

La question qui vient à l'esprit est: y-a-t-il autant de supernoeuds partiels que cela? Supposons que les supernoeuds soient des carrés de 64×64 pour pouvoir être calculés efficacement sur un multiprocesseur vectoriel. Le nombre de supernoeuds complets est d'environ 256, pour le domaine 1024×1024 envisagé. En supposant que nous ayons un ou deux types de supernoeuds partiels pour chacune des quatre frontières, nous aurions un total de 32 ou de 64 supernoeuds partiels à calculer. Le gain de un que nous pouvons espérer fait donc chuter la proportion de un pour quatre à un pour huit. Sachant qu'un codage simple des supernoeuds partiels leur donnait un temps d'exécution 15 fois supérieur au temps d'exécution d'un supernoeud complet sur l'Alliant FX/8[18] , on voit l'importance potentielle du problème.

Si la rangée inférieure du domaine d'itération était supprimée (ce qui revient à changer l'origine du partitionnement), le nombre de supernoeuds partiels le long de cette ligne serait doublé et on aurait deux types différents de supernoeuds partiels: un type à un noeud et un type à cinq noeuds.

Cette minimisation peut être importante si le code des supernoeuds partiels n'est pas bien optimisé et si le temps d'exécution d'un supernoeud partiel moyen est très supérieur au temps d'exécution d'un supernoeud complet. Il est envisageable de guider les choix de dimensionnement (i.e. de discrétisation) des usagers. Pour ne pas avoir de problème avec les bancs mémoires du Cray, ils savent déjà qu'il faut éviter les puissances de 2. Sur l'Alliant, les performances du produit de matrices dépendent fortement des dimensions et les utilisateurs savent quelles valeurs éviter (les puissances de 2 à nouveau, et leur voisinage). Il y a donc des précédents. Si dans le cas général, au sens mathématique du terme, le choix de l'origine du partitionnement est presque indifférent (figure 3.2.3), ce n'est pas le cas pour la plupart des programmes réels.

Cette optimisation peut avoir un effet négatif dans le cas où les boucles ne sont pas parfaitement imbriquées. Les supernoeuds complets tangents à la frontière générée par la boucle interne initiale doivent intégrer le bloc de code se trouvant entre les deux DO. Cela donne 3 types de supernoeuds



Variété des supernoeuds partiels

- Figure 3.2.3 -

complets et ajoute le problème de la détection des noeuds auxquels se rattache ce bloc de code. L'existence de supernoeuds complets dont certains noeuds appartiennent à une frontière suppose un test de détection des supernoeuds complets basé sur un calcul en nombres entiers. Si le test est fait avec des polyèdres continus, un tel noeud sera déclaré partiel. Le choix du test de complétude n'est donc pas facile.

3.2.4. Calcul des bornes des boucles

Comme nous avons fait des hypothèses de linéarité sur les expressions de bornes des boucles initiales (matrice A et vecteur $\vec{\alpha}$) et que les supernoeuds sont aussi des polyèdes convexes qu'on peut représenter par un système d'inégalités linéaires, nous allons pouvoir construire un système caractérisant l'ensemble des origines des supernoeuds dont l'intersection avec le domaine d'itération n'est pas vide, puis en dériver des systèmes équivalents à partir desquels il sera possible de générer du code.

Par définition du partitionnement par une matrice P, un noeud $\vec{j}$ appartient à un supernoeud d'origine $\vec{j}_0$ s'ils ont tous les deux les mêmes coordonnées entières dans la base P (cf. 2.4.1). Ceci s'écrit donc:

$$\vec{0} \leq P^{-1} (\vec{j} - \vec{j}_0) < \vec{1}$$

Pour garder des inégalités à coefficients entiers, on multiplie par $det(P)$ cette première inégalité.

La valeur de $\vec{j}_0$ est définie par les deux indices de boucle λ_1 et λ_2 qui sont aussi ses coordonnées dans la base G. En prenant comme matrice de passage de la base G à la base initiale, le produit de matrice PG, on obtient:

$$\vec{j}_0 = PG \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} + \vec{p}_o$$

où $\vec{p}_o$ est l'origine du partitionnement que nous venons de déterminer au paragraphe précédent (cf. 3.2.3).

Il ne reste plus qu'à exprimer le fait que l'intersection du supernoeud et du domaine d'itération n'est pas vide, c'est-à-dire qu'il existe un noeud de vecteur d'itération $\vec{j}$ vérifiant:

$$A \ \vec{j} \leq \vec{\alpha}$$

La conjonction de ces trois systèmes linéaires nous donne le système caractérisant les valeurs possibles de λ_1 et de λ_2 mais ne permet généralement pas de générer directement du code parce qu'il faut d'abord éliminer les variables auxiliaires $\vec{j}$ et $\vec{j}_0$ en projetant le système sur λ_1, λ_2 et toutes les autres variables différentes de $\vec{j}$ et de $\vec{j}_0$, et parce qu'il est peu probable que le système résultant ait une forme triangulaire.

Voyons ce que nous obtenons dans le cas de la figure 3.2.4.a, où nous avons choisi:

$$P = \begin{pmatrix} 3 & -3 \\ 3 & 3 \end{pmatrix} \quad G = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \quad \vec{\alpha} = \begin{pmatrix} 14 \\ 12 \\ -2 \\ 2 \end{pmatrix}$$

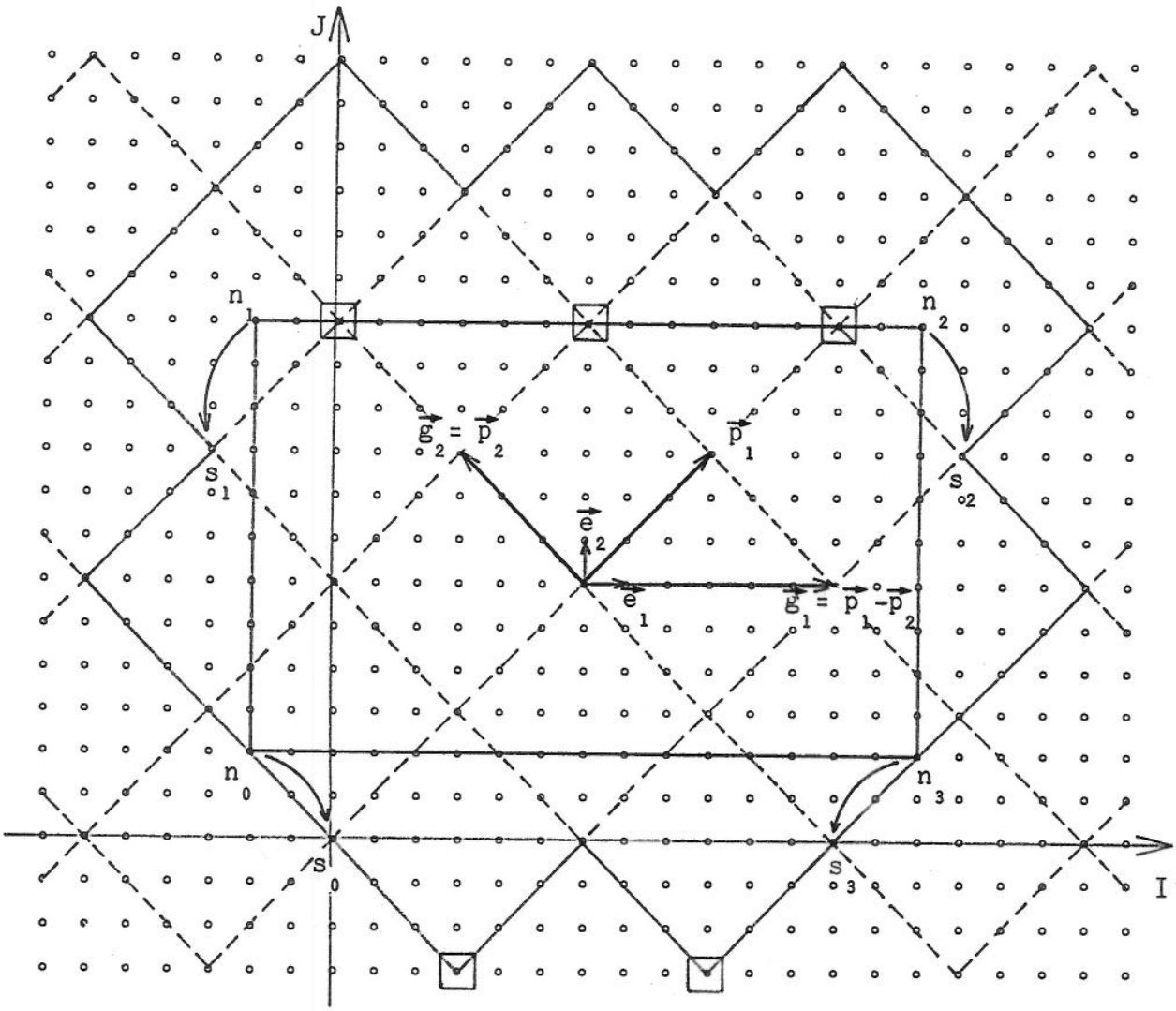
En appelant (I0,J0) les coordonnées de $\vec{j}_0$ dans la base initiale et (I,J) celles de $\vec{j}$, les valeurs précédentes de P, G et A conduisent aux inégalités suivantes:

$$\begin{cases} 0 \leq 3I - 3I0 + 3J - 3J0 < 18 \\ 0 \leq -3I + 3I0 + 3J - 3J0 < 18 \\ I0 = 6\lambda_1 - 3\lambda_2 \\ J0 = 3\lambda_2 \\ -2 \leq I \leq 14 \\ 2 \leq J \leq 12 \end{cases}$$

La projection de ce système sur λ_1 et λ_2 par l'algorithme de Fourier-Motzkin donne, après élimination de redondance en nombres entiers par test de faisabilité, les cinq inégalités suivantes, qui sont représentées sur la figure 3.2.4.b:

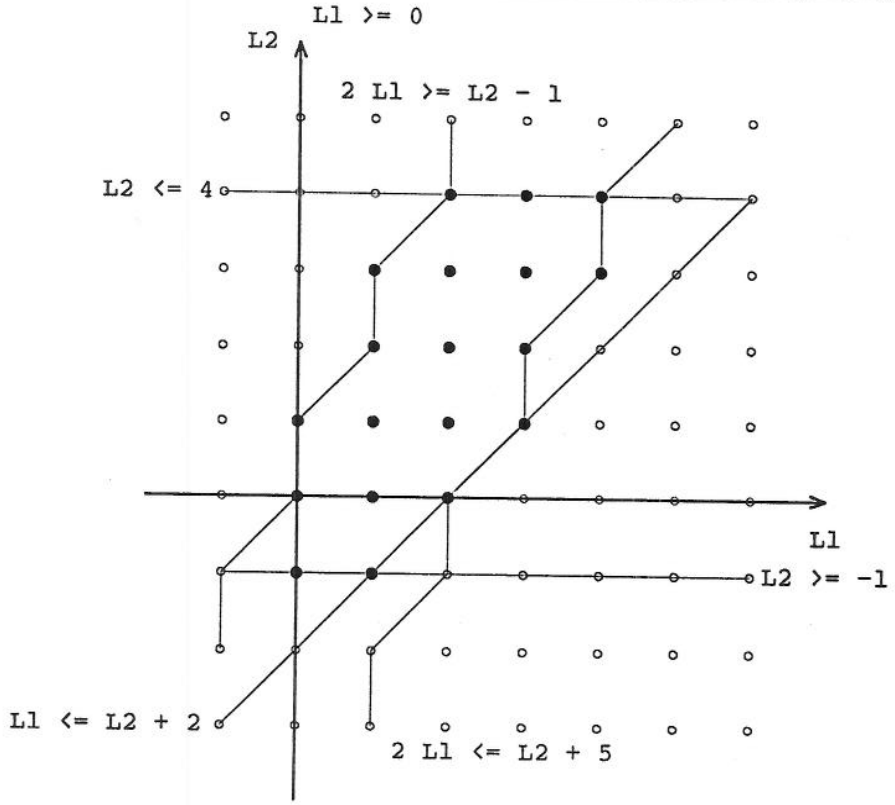
$$\begin{cases} -2\lambda_1 + \lambda_2 \leq 1 \\ \lambda_1 \geq 0 \\ 2\lambda_1 - \lambda_2 \leq 5 \\ \lambda_1 - \lambda_2 \leq 2 \\ -1 \leq \lambda_2 \leq 4 \end{cases}$$

La matrice du domaine dans G n'est pas triangulaire (il y a au moins une contrainte de trop) et il n'est donc pas possible de générer du code directement. Nous verrons en § 3.3.1.4, où un problème identique se pose pour la génération du code d'un supernoeud, comment tirer des expressions de bornes de boucles à partir d'un tel système quand un problème analogue se reposera pour le code de contrôle des supernoeuds complets. Voici déjà le code que nous proposerions:



Couverture d'un domaine d'itération par des supernoeuds

- Figure 3.2.4.a -



L1 ≡ λ_1
L2 ≡ λ_2

Contraintes dans la base G

- Figure 3.2.4.b -

DOSEQ $\lambda_2 = -1, 4$

DOALL λ_1 = MAX0(0, λ_2/2), MIN0((5+λ_2)/2,λ_2+2)

. . .

La variable λ_1 est utilisée comme indice interne car la direction paralléle a été prise comme première direction de la base G.

3.2.5. Tests de nature des supernoeuds

Comme nous l'avons vu en § 3.2.2, il va falloir calculer un test, aussi simple que possible, donnant à l'exécution le type du supernoeud de coordonnées (λ_1, λ_2). Les types potentiellement possibles sont: le supernoeud vide, le supernoeud complet, le supernoeud frontière et le supernoeud coin. Les deux derniers sont deux cas particuliers de supernoeuds partiels.

Supernoeud vide

Ce supernoeud, dont aucun noeud n'appartient au domaine d'itération, ne devrait jamais apparaître si l'algorithme de projection utilisé en 3.2.4 est correct en nombres entiers. Si des approximations sont acceptées, comme avec les algorithmes utilisés en vectorisation et en parallélisation, de tels noeuds risquent d'apparaître.

Supernoeud complet

Tous les noeuds d'un supernoeud complet appartiennent au domaine d'itération. Cela revient à demander que les quatre sommets du supernoeud appartiennent au domaine d'itération. Cette condition fournit un système d'inéquations dont on sait retirer automatiquement les contraintes redondantes et dont on peut donc tirer un test exact et efficace. La justification de cette construction et ses limites sont données en § 7.6.4.

Le code de ce type de supernoeud est simple si les boucles sont parfaitement imbriquées. Sinon, le code ne peut être gardé simple qu'en imposant que tous ses noeuds appartiennent à **l'intérieur** du domaine d'itération. Dans tous les cas le test peut n'être qu'approximatif, car les supernoeuds complets qui seront ignorés seront traités comme des supernoeuds partiels dont il faut exécuter tous les noeuds; seules les performances en souffriront.

Supernoeud partiel

Il faut distinguer deux types de supernoeuds partiels: les supernoeuds frontière et les supernoeuds coin.

Supernoeud frontière

Ce type de supernoeud est un supernoeud partiel auquel on associe la frontière **unique** qui l'intersecte. Le test consiste à faire l'intersection d'un supernoeud et d'un demi-plan, et à dériver des contraintes sur λ_1 et λ_2. Ces contraintes peuvent être optimisées comme celles du test de supernoeud complet par élimination des contraintes redondantes.

Le système linéaire se contruit en considérant deux noeuds $\vec{j}$ et $\vec{j}'$ d'un supernoeud $\vec{j}_0$ et en exprimant les conditions suivantes:

- $\vec{j}$ et $\vec{j}'$ appartiennent à $\vec{j}_0$;
- $\vec{j}_0$ est l'origine d'un supernoeud du partitionnement;
- $\vec{j}$ appartient au domaine d'itération;

- $\vec{j}'$ viole la contrainte de la frontière considérée.

Comme une frontière définit un demi-espace, sa négation est encore un polyèdre convexe. La conjonction de toutes ces conditions forme donc un système linéaire qu'on peut projeter sur λ_1 et λ_2 les coordonnées de $\vec{j}_0$ dans la base G ou sur I et J ses coordonnées dans la base initiale, moyennant l'ajout des deux équations correspondantes.

Supernoeud coin

Ce type de supernoeud intersecte au moins deux frontières; il est inutile d'en optimiser le codage car il y en a au plus quatre à moins que le domaine d'itération ne soit très pointu.

Le système d'inégalités qui permet de calculer ce test se construit comme le précédent, en ajoutant un troisième point qui doit violer une autre frontière. Quand le système résultant est infaisable, c'est tout simplement que de tels supernoeuds n'existent pas.

Il faut utiliser ces tests en **négatif** avant d'appeler le code du supernoeud frontière correspondant. Mais ils sont alors très simples puisque le contexte d'exécution (i.e. l'ensemble des prédicats vrais avant de l'exécuter) contient déjà l'information qu'on a affaire à un supernoeud frontière. L'élimination de redondance devrait donc réduire le système au test d'une égalité.

Conclusion

L'optimisation de l'ensemble de ces tests est importante car le temps d'exécution de ce code crée de l'overhead propre à la méthode de partitionnement proposée. Cet overhead va conduire à augmenter la taille des supernoeuds et à diminuer le domaine d'application de la méthode. Il faut cependant noter que ces tests seront exécutés en parallèle. Seule l'incrémentation du compteur de la boucle interne doit être effectuée en exclusion mutuelle (cf. 3.2.6.).

Les tests peuvent être obtenus dans la base G et porter sur λ_1 et λ_2 ou dans la base initiale et utiliser I et J.

3.2.6. Squelette de code résultant

Le squelette de code résultant a déjà été presque complètement décrit dans les paragraphes précédents. Pour modifier aussi peu que possible le code du corps de boucle, on repasse aussi vite que possible dans la base initiale B. Pour limiter l'overhead, on a pratiqué une réduction de force sur ces calculs de changement de base au prix de l'ajout de deux variables temporaires supplémentaires, I0 et J0. Pour alléger les écritures et pour faire la liaison avec nos notations habituelles, nous utilisons un Fortran étendu. Nous supposons la base G choisie de manière à avoir λ_1 comme indice de la boucle séquentielle: $G = (\vec{p}_1, \vec{p}_2 - \vec{p}_1)$. Les contraintes sur λ_2, qui sont des expressions linéaires sur λ_1, sont représentées par les symboles c_1, c_2, ... car nous ne connaissons pas de formules les donnant symboliquement et

explicitement (mais nous disposons d'algorithmes pour les calculer symboliquement). Il y en a 8 au plus, qui correspondent aux quatre contraintes initiales du domaine d'itération, plus quatre contraintes additionnelles qui règlent des problèmes d'arrondis au voisinage des quatre sommets.

```
I0 = p0 e1 + BIλ1 p1 e1
J0 = p0 e2 + BIλ1 p1 e2
DOSEQ λ1 = BIλ1,BSλ1
  DOALL λ2 = MAX0(c1,c2,...), MIN0(c4, ...,c8)
    I = I0 + λ2< p2-p1 . e1 >
    J = J0 + λ2< p2-p1 . e2 >
      CASE type(supernoeud(I,J))
        complet: CALL SNC(I,J,...)
        /* partiel: CALL SNP(I,J,...) si on n'optimise pas */
        /* détection des cas particuliers si on optimise: */
        frontière 1: CASE si plus d'un sous-type, sinon CALL SNP1(I,J,...)
        frontière 2: CASE si plus d'un sous-type, sinon CALL SNP2(I,J,...)
        frontière 3: CASE si plus d'un sous-type, sinon CALL SNP3(I,J,...)
        frontière 4: CASE si plus d'un sous-type, sinon CALL SNP4(I,J,...)
        coin: CALL SNT(I,J,...)
      ENDCASE
    ENDIF
  ENDDOALL
C   Passage au front suivant
  I0 = I0 + p1 e1
  J0 = J0 + p1 e2
ENDDOSEQ
```

Dans le cas où on a la chance de conserver un domaine triangulaire, les bornes du DOALL ne font intervenir ni MIN0 ni MAX0.

Quelques remarques:

- la variable λ_1 n'apparaît pas dans les calculs d'indices à cause de la réduction de force; la réduction de force n'est pas appliquée aux calculs de I et de J car ils apparaissent dans une boucle parallèle;
- il est possible d'éliminer les MIN et les MAX en appliquant une méthode de décomposition du polyèdre sur λ_1 et λ_2 en polyèdre triangulaire;

- il est aussi possible de supprimer les tests sur la nature des supernoeuds en remarquant que les supernoeuds frontières restent des supernoeuds frontière dans toutes les bases et qu'on peut donc, en séparant le premier et le dernier front des autres fronts et/ou en insérant le code des noeuds frontières entre les deux boucles sur λ_1 et λ_2 (ce qui en fait des boucles non parfaitement imbriquées), ne laisser que des supernoeuds complets au coeur des deux boucles;
- la section § 7.6 est consacrée à la génération du code de contrôle global sur un exemple.

3.3. Code d'un supernoeud complet

La matrice de partitionnement P a été choisie entière pour définir des supernoeuds identiques à une translation près dans l'espace d'itération discret. Il n'y a donc qu'un type de supernoeud complet à coder, paramétré par les coordonnées de son origine.

Nous allons successivement étudier cinq méthodes permettant de parcourir l'ensemble des noeuds d'un supernoeud puis choisir l'une d'entre elles dans chacun des cas suivants: direction d'itération libre, direction d'itération orthogonale à $\vec{p}_1$ ou $\vec{p}_2$ et direction d'itération fixée mais quelconque.

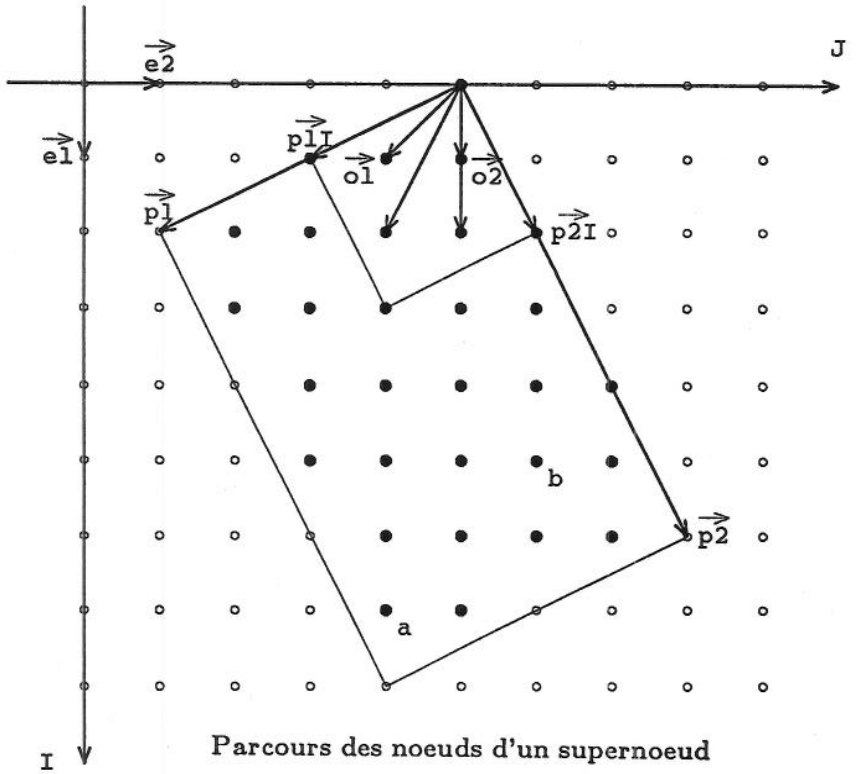
Attention! Nous allons utiliser à nouveau une base de génération $G = (\vec{g}_1, \vec{g}_2)$ et une matrice G. Elles sont relatives à un seul supernoeud et différentes des base et matrice G du paragraphe précédent. Les coordonnées dans cette nouvelle base G seront toujours (λ_1, λ_2).

3.3.1. Parcours des noeuds d'un supernoeud

On retrouve le problème rencontré précédemment en 3.2.1 de l'établissement d'une bijection entre les points entiers d'une base, ici B la base initiale, et les points entiers d'une autre base, G, ou d'un autre repère dans lequel le supernoeud est un domaine triangulaire et peut donc être parcouru par une série de boucles DO imbriquées. Mais cette fois, aucun overhead de test n'est admissible car les calculs élémentaires correspondent à une exécution du corps de boucle et peuvent être très courts, et car les tests empêcheraient une utilisation efficace ou même, sur certaines machines ne permettant pas les tests vectoriels, toute utilisation de l'unité vectorielle.

L'utilisation du deuxième niveau de parallélisme, une unité vectorielle par exemple, n'est possible que si, de plus, le deuxième vecteur de base de G est orthogonal à une direction d'itération valide, comme dans le cas des supernoeuds. Cependant cette direction d'itération, utilisée à **l'intérieur** d'un supernoeud, peut être différente de la direction d'itération pour les fronts de supernoeuds, vue au § 3.2. Ces deux directions d'itération doivent seulement respecter le même cône de dépendance R.

Les différentes solutions sont présentées avec l'exemple de la figure 3.3.1. La deuxième solution est celle que Kuhn propose dans son PhD. Les autres ont été développées pour les supernoeuds, mais la première est en fait équivalente à la transformation de programme que Wolfe appelle *Index Set Shifting* et qu'il présente d'une manière compliquée alors qu'elle revient à effectuer un changement de base. La troisième solution, qui fait appel à un repère non vectoriel et que nous avons utilisée initialement, semble a posteriori d'un intérêt limité, mais son principe peut être aussi utilisé pour résoudre le problème de copie des tableaux en mémoire locale (cf. § 3.5). La solution clé, qui va être utilisée et réutilisée continuellement, est la quatrième. Contrairement aux précédentes, elle n'impose aucune condition sur la forme du polyèdre ou sur la base de balayage. La cinquième en est une version légèrement compliquée qui permet de retrouver l'overhead minimal qui caractérise la solution de Kuhn.



Parcours des noeuds d'un supernoeud

- Figure 3.3.1 -

3.3.1.1. Première solution: transformation du supernoeud en domaine triangulaire

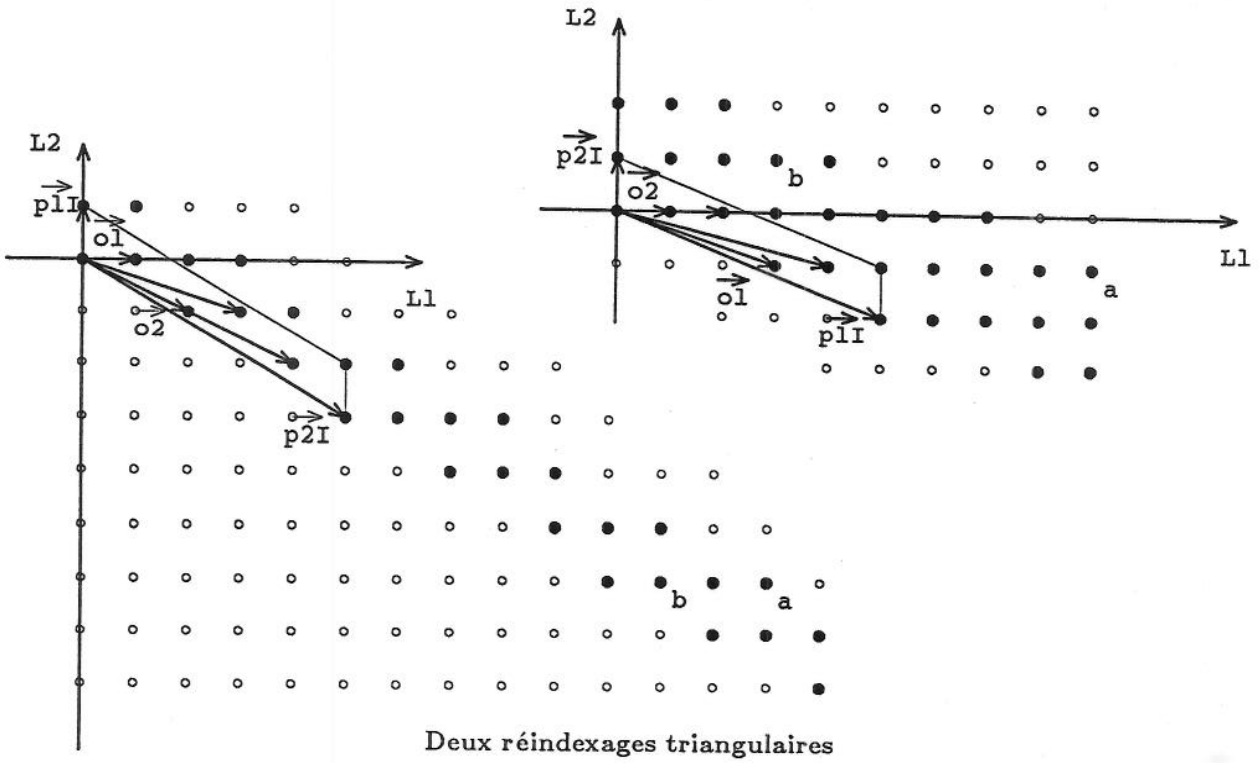
La première solution consiste à chercher une base G dans laquelle le supernoeud soit un domaine triangulaire (i.e. $P^{-1}G$ doit être triangulaire) et dans laquelle tous les points à coordonnées entières soient en bijection avec des points à coordonnées entières de B la base initiale (i.e. $det\,(G)=1$ et G à coefficients entiers, G unimodulaire).

Géométriquement, cela revient à prendre une nouvelle maille parallèle à $\vec{p}_1$ ou $\vec{p}_2$ pour le réseau des points entiers, maille dont tous les sommets sont des noeuds et qui ne contient aucun noeud à l'intérieur. Sur la figure 3.3.1.1.a, les couples de vecteurs $(\vec{o}_1, \vec{p}_1^{\,I})$ et $(\vec{o}_2, \vec{p}_2^{\,I})$ sont deux solutions minimales. Mais $(\vec{p}_1^{\,I}, \lambda\vec{p}_1^{\,I}+\vec{o}_1)$ conviendrait aussi.

Analytiquement, le problème revient à chercher une solution à l'équation:

$$P^{-1}G = \begin{pmatrix} \alpha & 0 \\ \beta & 1 \end{pmatrix}$$

si on intègre immédiatement l'interprétation géométrique en remplaçant temporairement P^I (cf. § 2.4.3) par P pour alléger les notations. $P^{-1}G$ étant la matrice de passage de la base G à la base P, ces équations signifient qu'il faut garder un des deux vecteurs de P, d'où les deux solutions suivant qu'on



Deux réindexages triangulaires

- Figure 3.3.1.1.a -

prend

$$P_1 = (\vec{p}_1^{\,I}, \vec{p}_2^{\,I}) \quad \textit{ou bien} \quad P_2 = (\vec{p}_2^{\,I}, \vec{p}_1^{\,I})$$

Dans le cas de la figure 3.3.1.1.a, on a:

$$P_1 = (\vec{p}_1^{\,I}, \vec{p}_2^{\,I}) = \begin{pmatrix} 1 & 2 \\ -2 & 1 \end{pmatrix} \quad \textit{resp.} \quad P_2 = (\vec{p}_2^{\,I}, \vec{p}_1^{\,I}) = \begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix}$$

On en déduit les matrices inverses:

$$P_1^{-1} = \frac{1}{5}\begin{pmatrix} 1 & -2 \\ 2 & 1 \end{pmatrix} \quad \textit{resp.} \quad P_2^{-1} = \frac{1}{5}\begin{pmatrix} -2 & -1 \\ -1 & 2 \end{pmatrix}$$

On calcule les matrices minimales pour G :

$$G_1 = (\vec{o}_2, \vec{p}_2^{\,I}) = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \quad \textit{ou bien} \quad G_2 = (\vec{o}_1, \vec{p}_1^{\,I}) = \begin{pmatrix} 1 & 1 \\ -1 & -2 \end{pmatrix}$$

et on en déduit les deux domaines d'itération possibles, dans la base G_1:

$$P_1^{-1}G_1 = \begin{pmatrix} 1/5 & 0 \\ 2/5 & 1 \end{pmatrix} \quad P_1^{-1}G_1 \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} < \begin{pmatrix} 2 \\ 3 \end{pmatrix} \quad P_1^{-1}G_1 \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

et dans la base G_2:

$$P_2^{-1}G_2 = \begin{pmatrix} 1/5 & 0 \\ 3/5 & 1 \end{pmatrix} \quad P_2^{-1}G_2 \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} < \begin{pmatrix} 3 \\ 2 \end{pmatrix} \quad P_2^{-1}G_2 \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Ces systèmes correspondent pour G_1 aux inégalités:

$$\frac{\lambda_1}{5} < 2$$

$$2\frac{\lambda_1}{5} + \lambda_2 < 3$$

$$-\frac{\lambda_1}{5} \leq 0$$

$$-2\frac{\lambda_1}{5} - \lambda_2 \leq 0$$

Il faut convertir les inégalités strictes qui proviennent de la définition des supernoeuds en inégalités non strictes et mettre les coefficients sur les indices à un. Tous ces calculs doivent être faits en nombres entiers. Les deux structures de contrôle possibles sont, avec G_1:

DO $\lambda_1 = 0, 9$
  DO $\lambda_2 = -3*\lambda_1/5, -3*\lambda_1/5+2$

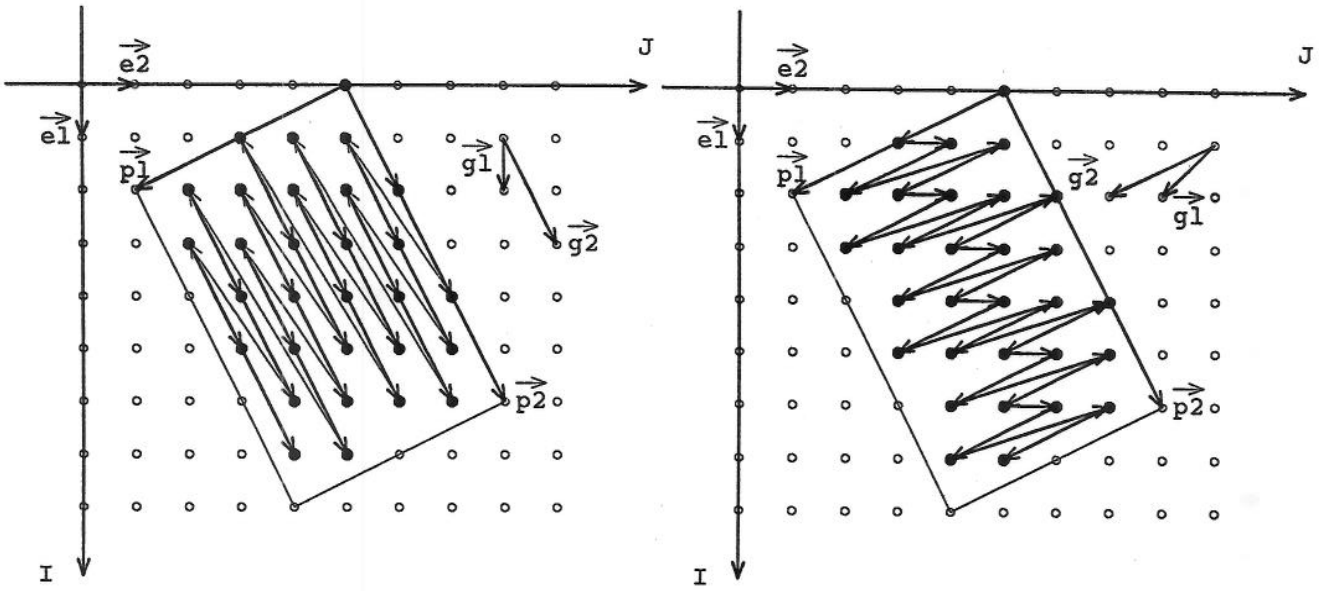
et, avec G_2:

DO $\lambda_1 = 0, 14$
  DO $\lambda_2 = -2*\lambda_1/5, -2*\lambda_1/5+1$

En choisissant la première et en détaillant les calculs de changement de base pour les indices:

```
I0 = j0 e1
J0 = j0 e2
DO λ1=0,9
  I = I0
  J = J0
  DO λ2 = -3*λ1/5, -3*λ1/5+2
    ....
    corps de boucle
    ....
    I = I + g2 e1
    J = J + g2 e2
  ENDDO λ2
  I0 = I0 + g1 e1
  J0 = J0 + g1 e2
ENDDO λ1
```

Deux exemples de tels changements de base sont donnés en figure 3.3.1.1.b.



Une autre représentation des réindexages triangulaires de la figure 3.3.1.1.a

- Figure 3.3.1.1.b -

Cette solution est équivalente à l'*Index Set Shifting* de Wolfe[30] , bien que développée pour un objectif différent et avec un formalisme plus mathématique.

3.3.1.2. Deuxième solution: décomposition en domaines triangulaires

Cette méthode n'impose pas une base particulière, ce qui est utile quand il faut avoir un vecteur orthogonal à $\vec{1}_H$ dans la base de parcours pour générer une boucle DOALL vectorielle. C'est ce pourquoi Kuhn[22] l'a développée. Mais elle est aussi utile pour rendre triangulaire dans la base B un supernoeud car, contrairement à la précédente, elle accepte en entrée n'importe quel polyèdre convexe, à condition que tous ses sommets soient connus numériquement. Cela veut dire que les bornes des boucles DO du code initial doivent être des constantes.

Sa généralité a cependant pour prix la lourdeur de la mise en oeuvre que propose Kuhn et nous proposerons donc une nouvelle méthode, très simple à mettre en oeuvre en dimension 2, comme complément de notre méthode de base (cf. § 3.3.1.4 et § 3.3.1.5).

La méthode de Kuhn me semble malheureusement assez mal expliquée dans son PhD[22] bien que son intérêt soit évident. Les principales étapes sont:

- choix d'une base quelconque; exprimer le système de contraintes dans cette base;
- triangularisation partielle des contraintes: les contraintes sont combinées linéairement pour mettre à zéro autant de coefficients que possible, en commençant par les coefficients associés aux derniers vecteurs de la base;
- séparation de la matrice obtenue en deux matrices, l'une ayant les coefficients pseudo-diagonaux positifs, l'autre les coefficients pseudo-diagonaux négatifs; j'appelle coefficient pseudo-diagonal, le dernier coefficient non-nul de chaque ligne: en triant les inégalités sur le rang de ce coefficient, on obtient un matrice pseudo-triangulaire;
- réduction à 1 ou –1 des coefficients pseudo-diagonaux;
- élimination progressive des contraintes redondantes (algorithme chez Kuhn, p. 40) en choisissant la plus forte; si l'élimination de contraintes est impossible car les deux contraintes ne sont pas comparables, ajout d'une nouvelle contrainte, bissectrice des deux précédentes, et de sa négation pour former deux nouveaux systèmes (Kuhn, p.78); l'élimination est ensuite effectuée sur les deux sous-systèmes;
- quand il n'y a plus qu'une contrainte de chaque rang, on a un ensemble de couples de matrices triangulaires; chaque couple permet de générer un ensemble de boucles imbriquées;

Dans notre cas, on est sûr de ne pas manquer de contraintes car le polyèdre correspondant à un supernoeud est fini. Comme base G il faut choisir un vecteur entier minimum donnant la direction parallèle et un deuxième vecteur entier garantissant $det(G)=1$. Le choix de ce deuxième vecteur n'est pas unique. Pour le déterminer complètement, il suffit d'ajouter la condition que sa norme soit minimale.

Nous ne traiterons pas le cas de la figure 3.3.1, car de nombreux exemples sont donnés dans[22].

L'exemple principal qu'utilise Kuhn pour illustrer sa méthode (p.79), la synthèse de l'algorithme parallèle de Sameh[26] pour la transformation de Givens, peut d'ailleurs être traité par un simple changement de base (cf. figure 3.3.1.2). Kuhn prend comme matrice de changement de base K :

$$K = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix}$$

La matrice K n'est pas triangulaire inférieure et elle détruit la triangularité du domaine d'itération. En conséquence, il faut découper le domaine en deux sous-domaines.

La matrice W que proposerait Wolfe[30] n'est pas très différente:

$$W = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}$$

puisque les deux vecteurs de base sont simplement permutés. Mais W est triangulaire et conserve donc un domaine d'itération triangulaire.

Dans ce cas particulier, un partitionnement suivant les deux vecteurs de dépendance est compatible avec une direction parallèle, celle qui est considérée comme optimale par Kuhn. Le supernoeud résultant serait transformé par le même changement de base en un domaine triangulaire ayant sa direction interne parallèle (cf. figure 3.3.4.1.b). C'est un des cas favorables où le changement de base permet à la fois d'obtenir un domaine d'itération triangulaire et un vecteur de base parallèle.

3.3.1.3. Troisième solution: repère non vectoriel

La troisième solution que nous avons envisagée consiste à ajouter un niveau de boucle supplémentaire en utilisant la décomposition de la cellule de base de la base P en une juxtaposition de cellules de base de la base P^I. Sur la figure 3.3.1, la cellule de base de P est obtenue à partir de 6 cellules de base de P^I. Cette décomposition est toujours possible:

Théorème: Toute base P, associée à un partitionnement définissant des supernoeuds identiques à une translation près, a des vecteurs de base de la forme:

$$\vec{p_i} = \lambda_i \ \vec{p_i^I} \qquad \lambda_i \in \mathbf{N}^*$$

Preuve

S'il existait un $\vec{p_i}$ ne vérifiant pas cette relation, on pourrait trouver un nouveau $\vec{p_i^I}$ inférieur à l'ancien en retirant de $\vec{p_i}$ un certain nombre de l'ancien $\vec{p_i^I}$. Ce nouveau $\vec{p_i^I}$ permettrait de définir une nouvelle base P^I, inférieure à l'ancienne, ce qui est impossible par définition de P^I.

Fin de la preuve

En conséquence, on peut repérer un noeud du supernoeud par trois coordonnées, λ_1, λ_2 et *off*, les deux premières étant les coordonnées de la cellule de type P^I dans laquelle est le noeud (i.e. les parties entières des coordonnées du noeud dans la base P^I) et la troisième un numéro, pour le moment quelconque, permettant de retrouver les points à coordonnées entières dans B qui se trouvent dans une cellule unitaire de P^I. Dans la figure 3.3.1, ces points sont au nombre de cinq, car il ne faut pas oublier l'origine du supernoeud qui nous sert maintenant d'origine de repère.

Pour retrouver les coordonnées dans B d'un point repéré par ces trois coordonnées (λ_1, λ_2, *off*), il suffit d'appliquer un changement de base $P^I \rightarrow B$ et d'ajouter au résultat le **vecteur d'offset** correspondant à la troisième coordonnée. Quatre vecteurs d'offset sont visibles sur la figure 3.3.1., le cinquième est un vecteur nul. Le nombre de ces vecteurs d'offset est bien sûr égal au déterminant de la base P^I. Dans notre exemple:

$$P^I = \begin{pmatrix} 1 & 2 \\ -2 & 1 \end{pmatrix} \qquad det(P^I) = 5$$

Une procédure simple pour trouver ces vecteurs d'offset consiste à balayer circulairement l'espace discret dans la base B à partir de l'origine, à sélectionner les vecteurs se trouvant dans la cellule unitaire de P^I et à s'arrêter quand on en a obtenu $det(P^I)$–1, le vecteur nul étant toujours solution. Pour plus de sûreté, on peut ajouter un deuxième test d'arrêt basé sur la norme de $\vec{p}_1^I + \vec{p}_2^I$.

Deux ordonnancements intéressants sont ceux qui découpent l'espace d'itération en tranches parallèles à $\vec{p}_1$ ou à $\vec{p}_2$. Les deux codes possibles sont:

DO $\lambda_1 = 0$, BS1
  DO $off = 1$, $\mid det(P^I) \mid$
    DO $\lambda_2 = 0$, BS2

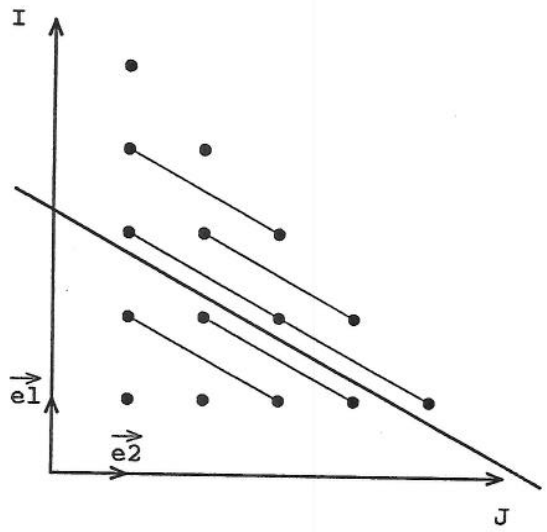
ou bien

DO $\lambda_2 = 0$, BS2
  DO $off = 1$, $\mid det(P^I) \mid$
    DO $\lambda_1 = 0$, BS1

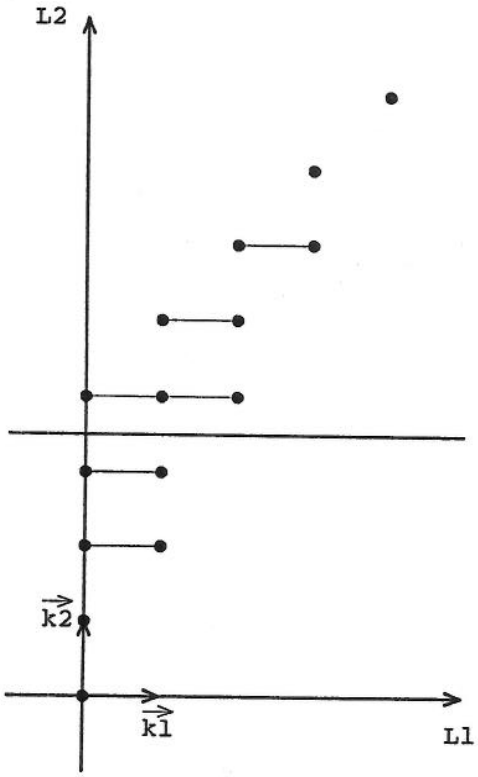
Dans les deux cas, BS1 et BS2 sont définis par les relations:

$$\vec{p}_1 = \text{BS1}\ \vec{p}_1^I \qquad \vec{p}_2 = \text{BS2}\ \vec{p}_2^I$$

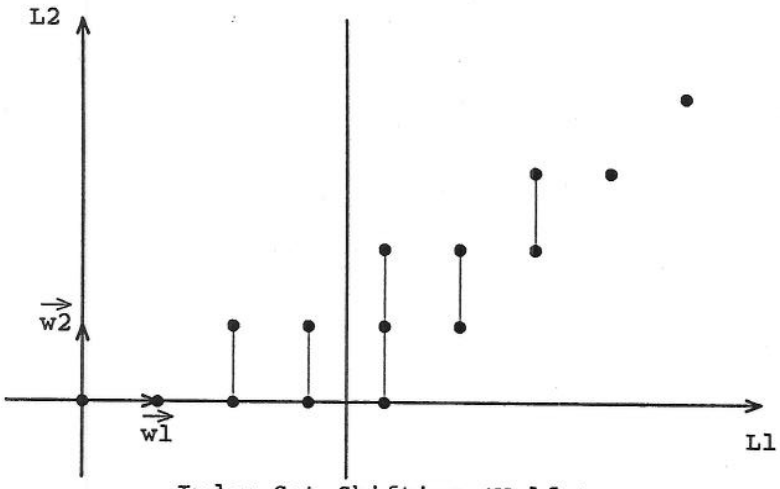
mais les vecteurs d'offset doivent être ordonnés de manière à respecter les dépendances, en commençant par les plus petits suivant l'ordre lexicographique défini par les deux boucles sur λ_1 et λ_2. En supposant que l'on choisisse le premier ordonnancement, avec la boucle sur λ_1 comme boucle externe, la matrice des



Domaine d'iteration initial



Loop Reindexing (Kuhn)



Index Set Shifting (Wolfe)

Comparaison de la méthode de Kuhn et de celle de Wolfe

- Figure 3.3.1.2 -

vecteurs d'offset O vaudrait, pour l'exemple de la figure 3.3.1:

$$O = \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Il ne faut pas chercher d'ordre lexicographique sur ces vecteurs, car ils sont exprimés dans la base B et non dans la base P^I pour pouvoir être utilisés directement dans la génération de code. Le code de contrôle final, avec la réduction de force appliquée systématiquement à tous les calculs de changement de base puisque l'exécution est séquentielle, est le suivant:

```
      I0 = j0 e1
      J0 = j0 e2
      DO λ1 = 0, BS1
        DO off = 1, | det(P^I) |
          I = I0 + < O(off) . e1 >
          J = J0 + < O(off) . e2 >
          DO λ2 = 0, BS2
              ...
            corps de boucle
              ...
C           passage au noeud suivant
            I = I + p2^I e1
            J = J + p2^I e2
          ENDDO λ2
        ENDDO off
        I0 = I0 + p1^I e1
        J0 = J0 + p1^I e2
      ENDDO λ1
```

Dans les pseudo-instructions précédentes, $\vec{j}_0$ définit l'origine du supernoeud et $O(off)$ est le *off*-ième vecteur d'offset.

Comme $\det(P^I)$ sera faible dans la plupart des cas usuels, le déroulement de la boucle sur *off* devrait diminuer l'overhead de contrôle. C'est ce que nous avons fait en écrivant manuellement le programme partitionné qui a été testé sur Alliant FX-8, mais cette solution nous semble maintenant moins intéressante que la suivante.

3.3.1.4. Quatrième solution: domaine non triangulaire

Les domaines triangulaires sont parcourus sans tests par des boucles DO imbriquées dont les bornes sont des expressions linéaires. Mais si cette forme de borne est indispensable pour de nombreux tests de dépendance et calculs de vecteurs de dépendance[†] , elle n'est pas indispensable à la génération de code[‡].

Si les opérateurs min et max sont acceptés, il suffit d'avoir un domaine pseudo-triangulaire pour pouvoir générer du code. Le passage d'un domaine quelconque à un domaine pseudo-triangulaire, dans lequel on admet plusieurs inégalités de rang n, est plus facile que la décomposition en domaines triangulaires, quelle que soit la dimension. En dimension 2, il suffit de projeter le domaine sur la variable correspondant à la boucle externe, puis, facultativement, d'éliminer les inégalités redondantes sans toucher aux deux inégalités relatives à cette variable.

L'expression de borne inférieure (resp. supérieure) de la boucle interne est un max (resp. un min) des contraintes où le coefficient de la deuxième variable est négatif (resp. positif). Dans le cas des supernoeuds, qui sont des parallélogrammes, il y aura au plus quatre expressions à calculer au lieu de deux et deux valeurs à comparer. La simplicité de la génération par rapport à la méthode de décomposition se paie d'un overhead de contrôle un peu plus élevé.

En reprenant l'exemple traité en 3.3.1.2 et en choisissant la base de partitionnement P_1, on obtient le domaine d'itération suivant dans la base initiale B :

$$P_1^{-1} = \frac{1}{5}\begin{pmatrix} 1 & -2 \\ 2 & 1 \end{pmatrix} \qquad P_1^{-1}\begin{pmatrix}\lambda_1 \\ \lambda_2\end{pmatrix} < \begin{pmatrix}2 \\ 3\end{pmatrix} \qquad P_1^{-1}\begin{pmatrix}\lambda_1 \\ \lambda_2\end{pmatrix} \geq \begin{pmatrix}0 \\ 0\end{pmatrix}$$

Ce système est équivalent aux inégalités suivantes:

$$\begin{cases} \lambda_1 - 2\lambda_2 < 10 \\ 2\lambda_1 + \lambda_2 < 15 \\ -\lambda_1 + 2\lambda_2 \leq 0 \\ -2\lambda_1 - \lambda_2 \leq 0 \end{cases}$$

On projette sur $\vec{e}_1$ pour obtenir des bornes fixes sur λ_1 en combinant les inégalités suivant l'algorithme de Fourier-Motzkin[12] :

$$5\lambda_1 < 40 \qquad \lambda_1 \geq 0$$

Puis on calcule les inégalités sur λ_2 en nombres entiers:

[†] Le test de dépendance développé par R. Triolet et présenté dans [29] supporterait des opérateurs comme *min*, *max* et *modulo* moyennant l'ajout de variables auxiliaires.

[‡] Wolfe utilise cette possibilité pour échanger deux boucles dont une seule est parallèle et obtenir une boucle interne parallèle, et donc vectorielle.

$$\begin{cases} \lambda_2 \geq \dfrac{\lambda_1}{2} - 4 \\ \lambda_2 \leq 14 - 2\lambda_1 \\ \lambda_2 \leq \dfrac{\lambda_1}{2} \\ \lambda_2 \geq -2\lambda_1 \end{cases}$$

On obtient la structure de contrôle suivante:

DO $\lambda_1 = 0, 7$
  DO $\lambda_2 =$ MAX(λ_1/2–4, –2λ_1), MIN(14–2λ_1, λ_1/2)
    ...
    corps de boucle
    ...

Comme on est resté dans la base initiale B, I = λ_1 et J = λ_2. La validité du code généré peut être vérifiée sur la figure 3.3.1 en n'oubliant pas que les contraintes ont une certaine *épaisseur* sur un espace discret: il ne faut pas se contenter de tracer des frontières infiniment minces.

Wolfe[30] utilise aussi le fait que deux boucles DO imbriquées peuvent correspondre à un domaine d'itération non triangulaire si les opérateurs Min et Max peuvent apparaître dans les expressions des bornes. Cela lui permet de mettre la boucle parallèle comme boucle interne si la machine cible est vectorielle.

Les contraintes peuvent alors être déduites d'un système d'inéquations linéaires exprimant le fait que le noeud $\vec{j}$ appartient au supernoeud $\vec{j}_0$ et que le vecteur $\vec{j}_0$, étant l'origine d'un supernoeud, a des coordonnées spéciales. Suivant les variables choisies, on peut générer les contraintes sur les coordonnées relatives de $\vec{j}$ par rapport à $\vec{j}_0$ ou sur les coordonnées absolues de $\vec{j}$ dans la base initiale B, ou encore sur les coordonnées de $\vec{j}$ dans n'importe quelle base dont la matrice de changement de base par rapport à la base initiale B, est unimodulaire, G par exemple.

Une fois le système constitué, il suffit de le projeter d'abord sur l'indice λ_1 qu'on souhaite avoir comme indice de boucle externe (la coordonnée selon $\vec{g}_1$ par exemple) pour obtenir les deux bornes de la boucle sur λ_1, puis de le projeter à nouveau sur les deux indices, λ_1 et λ_2, dont on veut connaître les bornes sans jamais oublier d'éliminer la redondance. Les contraintes portant uniquement sur λ_1 sont négligées, puis on transforme celles qui contiennent λ_2 de sorte que le coefficient correspondant soit 1 ou –1. Si plusieurs contraintes à coefficients positifs (resp. négatifs) existent, il faut reporter la détermination de la contrainte dominante à l'exécution au moyen d'un MIN (resp. d'un MAX) qui est utilisé comme borne supérieure (resp. inférieure) de la boucle sur λ_2.

Dans les cas où le nombre de contraintes n'est que de un (il ne peut pas être nul car le polyèdre doit être borné), il n'y a pas de MIN et/ou pas de MAX et les résultats de la première méthode sont

retrouvés.

On remarque que cette méthode ne fait d'hypothèse ni sur la base de balayage du polyèdre, si ce n'est l'unimodularité nécessaire pour visiter tous les noeuds et rien qu'eux, ni sur la forme du polyèdre. Cette méthode va donc pouvoir être appliquée sur les supernoeuds partiels, qui, obtenus par intersection de deux polyèdres convexes (un supernoeud et le domaine d'itération), sont toujours des polyèdres convexes. C'est aussi elle que nous proposons pour générer le code de balayage du domaine d'itération par des supernoeuds. Enfin, on verra au chapitre 7 qu'elle peut aussi servir à générer du code de copie. Accessoirement, elle permet aussi d'appliquer la méthode hyperplane sans avoir un overhead de test insupportable, ou de recalculer les bornes après un échange de boucles.

Bien que nous n'en ayons pas besoin pour le moment, elle semble aussi généralisable aux espaces de dimensions supérieures.

3.3.1.5. Cinquième solution: décomposition en domaines triangulaires par ordonnancement des sommets

La méthode précédente a l'inconvénient de nécessiter plus de tests, et donc d'overhead, que la décomposition en domaines triangulaires proposée par Kuhn. Ceci à cause des MINs et des MAXs. L'importance de cet overhead est difficile à évaluer sans faire d'expériences bien qu'il nous semble devoir être assez limité sur les gros supernoeuds qui vont être nécessaires pour des unités fonctionnelles vectorielles.

Quoi qu'il en soit, il est possible de générer la décomposition du polyèdre en domaines triangulaires par rapport à une base G en ordonnant les sommets (en dimension 2), points frontières des sous-domaines, par rapport à la direction d'itération donnée par $\vec{g}_1$ (on suppose que $\vec{g}_1$ correspond à la boucle externe) puis en découpant le polyèdre en tranches par des droites de vecteur directeur $\vec{g}_2$. Chaque tranche est, au pire, un trapèze, qui dégénère fréquemment en triangle ou en parallélogramme.

Soit $\vec{j}_1$ et $\vec{j}_2$ deux sommets du polyèdre, qui se suivent dans l'ordre précédent, et $\vec{j}$ un noeud dont on veut savoir s'il appartient au sous-domaine contenu entre $\vec{j}_1$ et $\vec{j}_2$ pour l'ordonnancement $\vec{g}_1$. Les deux contraintes qu'il faut ajouter au système initial sont:

$$\vec{j}_1 \, \vec{g}_1 \leq \vec{j} \, \vec{g}_1 < \vec{j}_2 \, \vec{g}_1$$

L'application de la méthode décrite au paragraphe précédent sur le système complété donne un balayage sans MIN ni MAX du sous-domaine.

Cette méthode suppose qu'il soit possible d'ordonner les sommets statiquement et impose donc, comme la méthode de Kuhn, que les bornes des boucles initiales soit connues numériquement. Il est aussi envisageable d'effectuer ce test dynamiquement et de générer les codes correspondant aux divers ordres possibles.

3.3.1.6. Conclusion

Les cinq méthodes que nous venons de décrire peuvent être comparées suivant les critères de facilité de mise en oeuvre, d'overhead de contrôle, d'overhead de calcul des indices dans la base initiale B, de volume de code généré et du nombre de directions d'itération qu'elles peuvent accepter.

	mise en oeuvre	overhead contrôle	overhead chgt de base	volume de code	directions d'itération
décomposition (Kuhn)	− −	+	−	−	∞
chgt de base (Wolfe)	+	+	−	+	2
offset	+	−	− −	+	2
boucles pseudo-triangulaires	++	−	+	+	∞
décomposition (ordre des sommets)	+	+	+	-	∞

Cytron a proposé dans son PhD une méthode plus générale que la méthode hyperplane, appelée DO ACROSS[10] . Cette méthode a l'inconvénient de générer du code parallèle mais non vectoriel. Elle est donc sans intérêt pour les corps de boucle simple, et n'est utilisable que si l'on a un multimultiprocesseur à la Cedar[13] pour pouvoir exploiter le parallélisme interne à un noeud en mode SPMD.

Elle n'a pas été citée parmi les méthodes utilisables car elle ne repose pas sur un réordonnancement par changement de base qui aurait pu servir à trianguláriser les supernoeuds, mais sur la génération de signaux de synchronisation.

Avant de générer le code du supernoeud, il nous faudra choisir une direction d'itération propre au supernoeud lui-même. Elle peut être différente de la direction d'itération choisie pour l'ensemble des supernoeuds. Elle peut aussi être incompatible avec certaines des méthodes de parcours qui viennent d'être décrites.

3.3.2. Direction d'itération libre

C'est le cas a priori quand les processeurs élémentaires sont scalaires. Toutes les directions d'itération vont donner le même temps de calcul, au problème de cache ou plus généralement d'accès aux données près. Comme nous le verrons ultérieurement, le volume de cache nécessaire à l'exécution

d'un ensemble de noeuds dépend de la direction d'itération. Les autodépendances peuvent aussi être prises en compte pour déterminer une direction d'itération. Si l'optimisation est extrême, la direction d'itération sera donc fixée en fonction des accès aux données et on se retrouvera exactement dans une situation similaire à la génération de code vectoriel, la direction d'itération sera fixée (cf. les deux paragraphes suivants).

Si la direction d'itération est libre, n'importe laquelle des méthodes présentées en 3.3.1 peut être utilisée. Il vaut donc mieux appliquer la plus simple, à savoir la première, en posant $\vec{g}_1 = \vec{p}_1$ ou $\vec{g}_1 = \vec{p}_2$. Mais la quatrième méthode donnera le même résultat avec une telle base G car on ne trouvera que deux contraintes par indice. Le problème revient donc à un bon choix de G.

3.3.3. Direction d'itération fixée. Cas favorable

Si la direction d'itération est fixée, soit pour conserver une direction vectorielle, soit pour optimiser l'accès aux données, et si cette direction d'itération est orthogonale à $\vec{p}_1$ ou $\vec{p}_2$, la troisième méthode de génération de code détaillée au paragraphe précédent (repère non vectoriel et offset) est encore valide, moyennant quelques aménagements.

Tout d'abord, le choix de la boucle interne et de la boucle externe est fixé par la direction d'itération. Deuxièmement, le calcul de I et de J doit être dérécursivé (en fait, il suffit de ne pas appliquer la réduction de force).

Si la condition précédente est vérifiée, ou s'il existe un vecteur $\vec{v}$ entier orthogonal à la direction d'itération qui forme une base de déterminant 1 avec $\vec{p}_1$ ou $\vec{p}_2$, la deuxième méthode (changement de base) est valide.

L'orthogonalité de la direction d'itération et d'un des deux vecteurs de P sera rarement le fruit du hasard. En fait, on utilisera plutôt cette propriété comme un critère de choix pour P ou pour la direction d'itération. En fait on ne va pas nécessairement fixer la direction d'itération dans les supernoeuds en phase d'optimisation mais simplement demander la génération de code vectoriel. Dans ce cas, il suffit de vérifier que $\vec{p}_1$ ou $\vec{p}_2$ dans un cas, $\vec{g}_1$ ou $\vec{g}_2$ dans l'autre, ont des normales acceptables comme direction d'itération.

La première méthode est aussi applicable puisqu'il existe une infinité de bases dans lesquelles le supernoeud est représenté par un domaine triangulaire. L'un des deux vecteurs, $\vec{p}_1$ ou $\vec{p}_2$, doit faire partie de la base. L'autre doit être choisi de manière à avoir $det(G) = \pm 1$. Il y a toujours, pour le deuxième vecteur de base, une infinité de solutions qui diffèrent entre elles d'une translation selon $\vec{g}_1$. Tous les vecteurs $\vec{g}_2$ relient deux droites de noeuds de vecteur directeur $\vec{g}_1$ aussi proches que possibles.

3.3.4. Direction d'itération fixée. Cas général.

On se retrouve avec un problème de génération de code similaire à celui de la méthode des vagues en dimension 2 qu'ont successivement étudié Kuhn, sous le nom de *Loop Reindexing*[22] , et Wolfe, sous le nom d'*Index Set Shifting*[30] . Mais contrairement au cas usuel, le domaine d'itération n'est pas triangulaire, ou bien nous avons déjà fait un changement de base pour le rendre triangulaire.

Trois méthodes restent possibles: la décomposition du supernoeud en domaines triangulaires dans une base G contenant un vecteur entier normal à la direction d'itération et un autre vecteur entier assurant que $det(G)=1$ (méthode 2); le passage dans la même base G, mais en remplaçant la décomposition par la pseudo-triangularisation du domaine, et génération de code avec MIN et MAX (méthode 4); le passage dans l'autre base G, celle qui assure la triangularité du supernoeud (méthode 1), et l'ajout de synchronisations de type DOACROSS pour permettre l'exécution parallèle.

3.3.4.1. Changement de base et domaine non triangulaire

Le principe est donc de choisir une base dont le 2ème vecteur est orthogonal à la direction d'itération demandée, puis de pseudo-triangulariser le domaine d'itération dans cette nouvelle base et de générer un code à base de MIN et de MAX.

Nous allons montrer le déroulement des opérations sur un cas particulier qui nous a déjà beaucoup servi, celui de la figure 3.3.1., en faisant l'hypothèse supplémentaire, que, pour une raison inconnue, la direction d'itération $\vec{p}_1^I + \vec{p}_2^I$ donne le temps d'exécution optimal pour le supernoeud.

Avec la direction parallèle $\vec{p}_2^I - \vec{p}_1^I$, le vecteur $2\vec{o}_2 - \vec{p}_1^I$ permet de constituer une base G qui établit une bijection entre les espaces discrets de B et de G :

$$G = \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} \qquad det(G) = 1$$

Le calcul du domaine d'itération dans G va faire intervenir P^{-1} pour passer des coordonnées d'un vecteur d'itération dans G aux coordonnées dans P où le supernoeud est défini par un système simple, en passant par G :

$$P = \begin{pmatrix} 1 & 2 \\ -2 & 1 \end{pmatrix} \qquad P^{-1} = \frac{1}{5}\begin{pmatrix} 3 & -1 \\ -2 & 1 \end{pmatrix}$$

La matrice de changement de base qui nous intéresse est:

$$P^{-1}G = \frac{1}{5}\begin{pmatrix} 3 & -1 \\ -2 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} = \frac{1}{5}\begin{pmatrix} -3 & -5 \\ 4 & 5 \end{pmatrix}$$

Le domaine est donné par:

$$P^{-1}G\begin{pmatrix}\lambda_1\\ \lambda_2\end{pmatrix} < \begin{pmatrix}2\\ 3\end{pmatrix} \quad et \quad P^{-1}G\begin{pmatrix}\lambda_1\\ \lambda_2\end{pmatrix} \geq \begin{pmatrix}0\\ 0\end{pmatrix}$$

Ce qui s'écrit sous forme d'inégalités:

$$\begin{cases} -3\lambda_1 - 5\lambda_2 < 10 \\ 4\lambda_1 + 5\lambda_2 < 15 \\ 3\lambda_1 + 5\lambda_2 \leq 0 \\ -4\lambda_1 - 5\lambda_2 \leq 0 \end{cases}$$

On en tire par projection (cf. 3.3.1.4.):

$$0 \leq \lambda_1 < 25$$

Puis en regroupant les inégalités suivant le signe du coefficient de λ_2 et en changeant les inégalités strictes en inégalités larges:

```
DO λ1 = 0, 24
  DO λ2 = MAX(-4*λ1/5, -1-3*λ1/5), MIN(-3*λ1/5, 2-4*λ1/5)
    changement de base
    ...
    corps de boucle
```

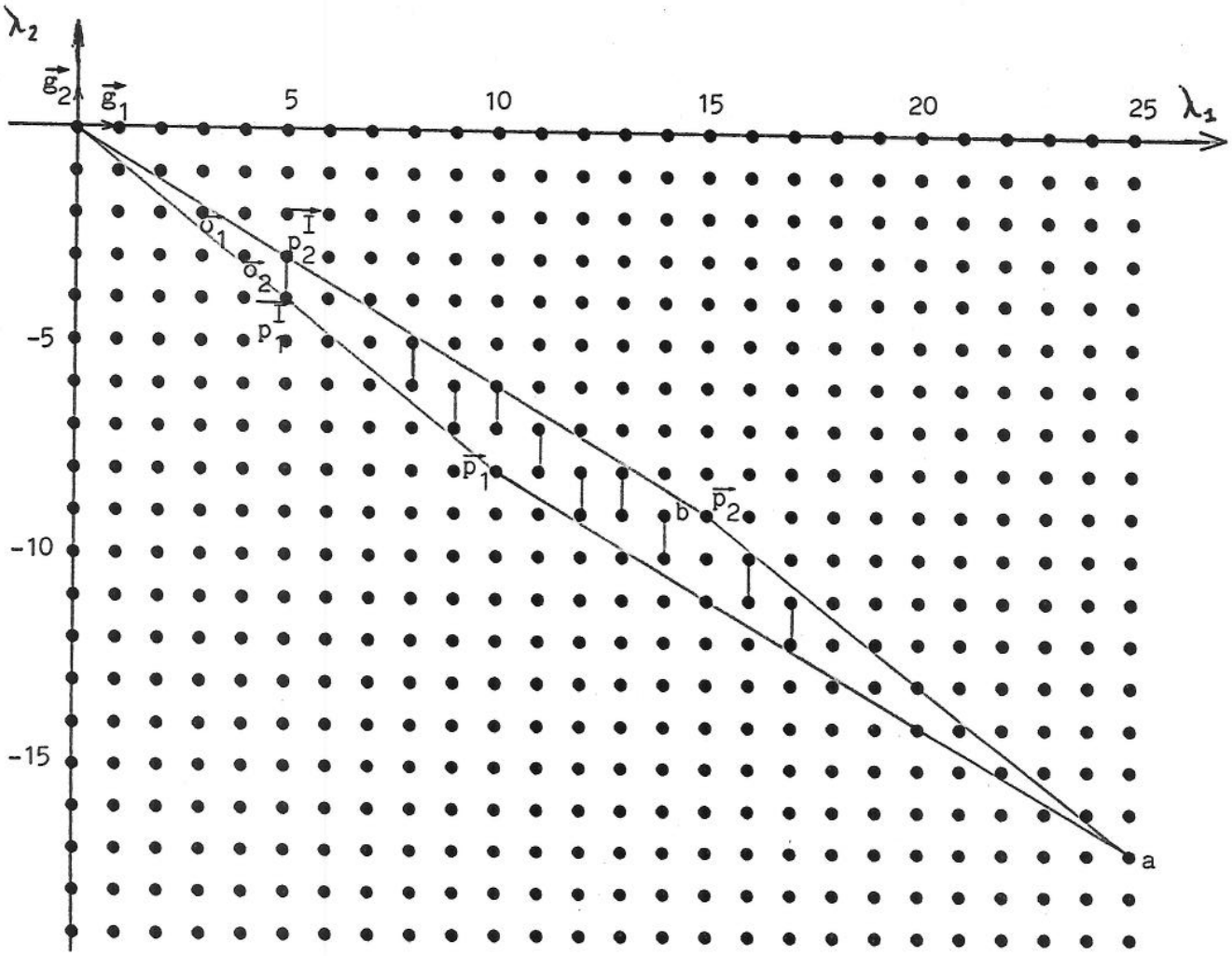
Comme on peut le voir sur la figure 3.3.4.1, le parallélisme vectoriel obtenu n'est pas bon avec le supernoeud de petite taille que nous avons choisi pour faire des figures de taille raisonnable. Il n'est pas bon aussi parce que la direction d'itération choisie n'est pas astucieuse. Une direction optimale† serait vraisemblablement la normale à $\vec{o}_1\vec{p}_1^I$ ou $\vec{o}_2\vec{p}_2^I$. Dans ces deux cas, le supernoeud est un domaine triangulaire dans G car $\vec{p}_1^I$ ou $\vec{p}_2^I$ peuvent servir à former la base.

3.3.4.2. Décomposition du supernoeud en domaines triangulaires

Cette méthode a été présentée au paragraphe 3.3.1.4. Elle est réutilisable telle que, puisqu'aucune hypothèse n'est faite sur la base G dans laquelle on veut trianguláriser le supernoeud, si ce n'est que G est entière et de déterminant unité (i.e. unimodulaire). Cette condition garantit qu'aucun noeud fictif n'est introduit et qu'aucun noeud réel n'est oublié.

---

† La notion d'optimalité est encore floue. Nous reviendrons sur ce point au chapitre 6. Les deux directions citées sont optimales si l'on cherche à avoir des vecteurs de longueur maximum et un minimum d'overhead de contrôle (hypothèse équivalant à celle du nombre de processeurs infini). Le choix de l'une ou de l'autre dépend de la première coordonnée de $\vec{1}_P$ dans la base G.



Parallélisme contenu dans un supernoeud

- Figure 3.3.4.1 -

3.3.4.3. Les DOACROSS

Cytron a proposé dans son PhD une méthode plus générale que la méthode hyperplane, appelée DO ACCROSS[10] . Cette méthode a l'inconvénient de générer du code parallèle mais non vectoriel. Elle est donc sans intérêt pour les corps de boucle simple, et n'est utilisable que si l'on a un multimultiprocesseur à la Cedar[13] pour pouvoir exploiter le parallélisme interne à un noeud.

Elle n'a pas été citée dans le paragraphe car elle ne repose pas sur un réordonnancement par changement de base qui aurait pu servir à trianguläriser les supernoeuds mais sur la génération de signaux de synchronisation, ce qui présente l'avantage de préserver l'aspect du programme initial dans les

restructureurs qui produisent du code source en sortie.

3.3.4.4. Conclusion

Les exécutions générées par ces trois méthodes, quand elles peuvent s'appliquer, sont strictement égales. Seuls diffèrent la génération de code, et donc la lisibilité du résultat, et le type de synchronisation utilisé, et donc les machines utilisables.

Il n'est pas encore possible de faire un choix. Seule l'expérience dira si l'implémentation de la méthode de Kuhn est utile.

3.4. Code de contrôle pour un supernoeud partiel

La solution la plus simple consisterait à n'avoir qu'un seul type de supernoeud partiel. Pour ce faire, le code du supernoeud partiel devrait contenir un test d'appartenance au domaine d'itération pour chaque noeud. Malgré l'overhead dû à ces tests et, sur une machine vectorielle, la scalarisation de l'exécution qui en résulte, nous espérions que le temps de calcul des noeuds partiels n'aurait qu'un effet négligeable sur le temps de calcul global.

Une expérience sur Alliant FX-8 a montré qu'il faudrait de très grands domaines d'itération, ayant une surface évoluant effectivement en $O(n^2)$ où n est la taille d'un côté exprimée en supernoeuds (ce qui élimine les domaines *plats*), pour que le gain dû au partitionnement en supernoeuds soit supérieur au temps perdu dans les supernoeuds partiels. En effet, le nombre de supernoeuds partiels croissait linéairement, en $8 \times n$, si le choix de l'origine du partitionnement et les dimensions du domaine d'itération étaient particulièrement défavorables (en $4 \times n$ dans le cas le plus favorable) et le temps de calcul moyen d'un supernoeud partiel était, sur l'Alliant et dans le cas particulier testé, environ 15 fois supérieur au temps d'exécution d'un supernoeud complet! Le raisonnement à base de $O(n)$ et de $O(n^2)$ s'est avéré dangereux à cause du coefficient de 120 devant le terme de plus bas degré. Et exiger $n >> 120$, revient à réduire considérablement le domaine d'application du partitionnement puisqu'avec des supernoeuds minima de 32×32 et $n = 120$ seulement, il faudrait avoir un domaine d'itération de près de 15 millions de noeuds.

Il faut donc optimiser l'exécution des supernoeuds partiels pour faire passer le nombre de tests de $O(n^2)$ à $O(n)$ et pour pouvoir utiliser l'unité vectorielle lors de l'exécution d'un supernoeud partiel. Mais l'optimisation n'est possible que si la forme exact du supernoeud partiel est connue, ce qui oblige à distinguer un grand nombre de cas particuliers. Le nombre de ces cas dépend, comme nous l'avons vu précédemment en § 3.2., de la pente des frontières et de la position de l'origine. Une généralisation à une dimension supérieure à 2 le ferait aussi croître.

3.4.1. Identification des types de supernoeuds partiels le long d'une frontière

On a vu précédemment comment calculer la périodicité d'apparition d'un type particulier de supernoeud partiel et le nombre de ces types.

Pour calculer le domaine d'itération de chaque type, il suffit de partir d'un point entier quelconque de la frontière (un des sommets du domaine d'itération par exemple), de trouver le supernoeud qui le contient, puis de suivre la frontière suivant l'algorithme de tracé de droite avec des pas élémentaires $\vec{p}_1$ ou $\vec{p}_2$ pendant $|\alpha| + |\beta|$ coups ou $|\alpha| + |\beta| - 1$ si une origine de supernoeud appartient à la frontière (cf. § 3.2.2).

Pour chacun des supernoeuds trouvés, on calcule une distance[†] entière à la frontière en faisant le

[†]Ce n'est pas une distance à proprement parler car elle peut être négative, mais le mot position ne fait pas davantage l'affaire. Il faut que cette quantité, qui exprime la position de l'origine du supernoeud par rapport à la frontière, soit une fonction linéaire des coordonnées.

produit scalaire d'un vecteur normal, à coordonnées entières, quelconques, à la frontière et du vecteur d'itération de l'origine du supernoeud trouvé. Cette distance, qui est linéaire, est ensuite ajoutée comme une contrainte limitant l'aire du supernoeud avant de procéder à la génération de code, exactement comme pour un supernoeud complet.

3.4.2. Génération de code optimal, scalaire ou vectoriel

Il est peu probable qu'on retrouve des cas de génération favorables permettant d'éviter la présence de MIN et de MAX dans les expressions de bornes, mais les deux dernières méthodes que nous avons développées et qui sont totalement indépendantes de la nature du polyèdre qu'il faut parcourir s'appliquent toujours dans ce cas.

Trois types de polyèdres peuvent être générés comme intersection d'une frontière du domaine d'itération et d'un supernoeud: des triangles, des trapèzes et des pentagones irréguliers. Il faut remarquer qu'un changement de base ne modifie pas la nature d'un polyèdre: les nombres de sommets et d'arêtes sont conservés.

Pour les triangles, on se retrouve exactement dans la situation des supernoeuds pleins. On peut faire un changement de base à partir de $\vec{p}_1$ ou de $\vec{p}_2$ et obtenir un domaine triangulaire.

Pour les trapèzes, le changement de base doit obligatoirement s'effectuer en mettant la direction parallèle aux deux côtés parallèles dans la nouvelle base. On a donc moins de chances de pouvoir le faire.

Enfin, dans le cas des pentagones, il est strictement impossible de se ramener à un domaine défini par un système triangulaire qui par définition ne peut avoir que 4 côtés, dont 2 parallèles.

L'unique problème pourrait être la prolifération du code, puisqu'il faut générer quatre bornes de boucles pour chaque type de supernoeud le long de chaque frontière. On a vu en § 3.2.3 que ces types pouvaient être très nombreux.

3.4.3. Génération de code simple

On reprend le code scalaire développé pour un supernoeud complet en 3.3.2, tout en ajoutant au coeur, juste avant le corps de boucle, un test d'appartenance au domaine d'itération.

Pour diminuer un peu l'overhead de test, on peut générer un code simple différent par frontière et ne mettre que le test correspondant à ladite frontière.

Cette solution est facile à mettre en oeuvre mais son intérêt principal est surtout de limiter le volume de code produit.

3.4.4. Cas des supernoeuds intersectant plusieurs frontières

Le problème principal est de les identifier rapidement. Il a été étudié en 3.2.5. Ces supernoeuds étant au nombre de quatre au plus sur la plupart des domaines d'itération, il ne semble pas utile de chercher à les optimiser. L'utilisation du code simple décrit au paragraphe précédent devrait fournir de résultats acceptables.

Cependant, la méthode de pseudo-triangularisation et la méthode de triangularisation peuvent encore être appliquées en partant d'un système d'inégalités renforcé par le test de détection de tels supernoeuds.

Dans le cas général de dimension N, que nous ne traitons pas dans ce chapitre consacré à la génération de code, il y aura aussi des supernoeuds contenant des arêtes. En dimension 2, arêtes et faces sont confondues.

3.4.5. Ajout du code mal imbriqué

Ce problème est mentionné ici pour mémoire, puisque nous n'étudions par hypothèse que les boucles bien imbriquées. Le code mal imbriqué ne doit apparaître que dans des noeuds partiels si le test de noeud complet est bien choisi (cf. 3.2.5). Le code doit être ajouté quand la contrainte de la frontière correspondante est saturée. Cet ajout peut être fait sans test si la saturation de la frontière correspond à une des deux bornes de la boucle interne. Cela devrait être le cas pour les supernoeuds partiels triangulaires. Pour les autres supernoeuds, il faut ajouter un test en début ou/et en fin de l'ensemble des tours de boucle interne pour tester cette saturation.

3.5. Modification des références aux variables

Jusqu'ici, nous ne nous sommes préoccupé que du code de contrôle nécessaire pour induire l'ordre d'exécution correspondant au partitionnement décrit au chapitre 2, et du code nécessaire au retour dans la base initiale. Avec certaines architectures de machines, il suffit de conserver le corps de boucle tel quel. Mais, dans la plupart des cas, il est nécessaire de transformer les références aux tableaux originaux en des références à une ou plusieurs de leurs copies, partielles ou totales.

Plusieurs des solutions proposées seront valables pour des espaces de dimension supérieure à 2.

Dans les paragraphes 3.5 et 3.6, le mot **copie** est ambigu: il s'agit tantôt du tableau contenant la copie des données (en général dans 3.5), tantôt du mouvement de données (3.6).

Après avoir éliminé les architectures de machine qui ne nécessitent aucune modification des références, nous étudierons le cas où un tableau n'est référencé que par une famille de références dans le corps de boucle et proposerons trois copies possibles. Puis nous passerons au cas des familles de références multiples, qui est plus compliqué à cause des problèmes de cohérence entre copies. Enfin nous essaierons de déterminer quelles zones de copies doivent être initialisées ou sauvegardées dans la mémoire globale.

3.5.1. Accès automatique

Avec les machines qui contiennent des dispositifs matériels pour gérer de manière transparente les différents niveaux de la hiérarchie mémoire, il est inutile de changer les références aux tableaux contenues dans le corps de boucle. Il suffit donc de recopier ce dernier tel quel au coeur des structures de contrôle développées dans le paragraphe précédent.

Dans le cas du cache, il faudra quand même vérifier qu'on ne provoque pas de débordement de cache, ce qui conduira dans le chapitre suivant à essayer d'évaluer les volumes des données à mettre simultanément en cache.

3.5.2. Problèmes posés par la copie optimale

Quand la hiérarchie mémoire doit être gérée par l'utilisateur au moyen de copies d'un niveau à un autre, il faut essayer de réserver un minimum de place dans le niveau le plus rapide, de ne copier qu'un minimum d'éléments de tableaux et de ne pas compliquer excessivement les calculs d'indices, pour ne pas risquer de dégrader les performances vectorielles de la machine cible, sinon même imposer une exécution scalaire.

La copie optimale consisterait à ne réserver que l'espace strictement nécessaire, à ne copier que ce qu'il faut et à conserver un adressage linéaire. Il est facile de montrer sur un contre-exemple que ces objectifs sont incompatibles dans le cas général. On se place en dimension 2, avec une référence à un tableau T ayant un vecteur d'autodépendance qui n'est colinéaire avec aucun des vecteurs des bases B et P. La référence est définie par une forme linéaire $\vec{\alpha}$ et elle restera définie par une forme linéaire tant

que les accès aux éléments du tableau seront linéaires: la seule transformation qu'on puisse faire est un changement de base. Il faut que des noeuds qui accédaient le même élément de tableau continuent à accéder le même élément:

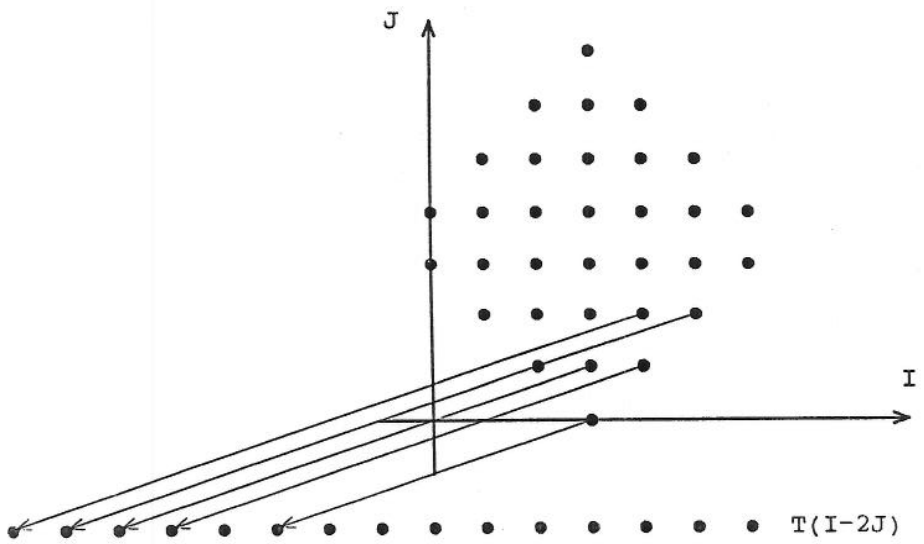
$$\vec{\alpha}\ \vec{j} = \vec{\alpha}\ \vec{j}' \quad => \quad \vec{\alpha}'\ \vec{j} = \vec{\alpha}'\ \vec{j}'$$

Ceci implique que $\vec{\alpha}$ et $\vec{\alpha}'$ sont orthogonaux à $\vec{j}-\vec{j}'$, et, comme nous sommes en dimension 2, que $\vec{\alpha}' = \lambda\vec{\alpha}'$. La seule chose qu'on puisse faire est donc d'appliquer une homothétie à $\vec{\alpha}$ pour supprimer les trous systématiques. Le meilleur coefficient est l'inverse du PGCD de ses coordonnées qui donne une nouvelle forme linéaire $\vec{\alpha}'$ suceptible d'accéder tous les éléments du nouveau tableau.

Le contre-exemple est dessiné figure 3.5.2.a. Le tableau T est référencé par

```
... = ... T(-3*I+J)
```

et le sommet du supernoeud adresse un élément isolé de T, qu'on ne peut pas rapprocher des autres par homothétie sans provoquer des collisions.



Copie linéaire non-optimale

- Figure 3.5.2.a -

Une fois montrée l'absence de solution linéaire, il reste à voir ce qu'on peut faire en levant une des trois conditions définissant la copie optimale dans les trois cas qui se posent en dimension 2, le scalaire, le tableau monodimensionnel et le tableau bidimensionnel.

La dimension dont nous parlons ici n'est pas celle qui est déclarée dans le programme mais la dimension de l'espace généré par l'ensemble des formes linéaires $\vec{\alpha}_i$ qui forment les expressions d'indices.

La constante est négligée. Par exemple, la référence:

T(I,I+1)

est de dimension 1, car elle génère une droite de vecteur directeur:

$$\begin{pmatrix}1\\0\end{pmatrix}$$

Comme l'espace d'itération est de dimension 2, les tableaux ne peuvent pas être plus que bidimensionnels.

Ceci posé, les tableaux 0-dimensionnels, comme T(2,3), et les scalaires peuvent être copiés dans des scalaires et accédés par un nom. Les tableaux monodimensionnels peuvent être accédés linéairement sans perdre beaucoup d'espace. La figure 3.5.2.a montre que l'on ne perdra pratiquement pas d'espace, surtout dans les cas courants où les expressions d'index ne contiennent presque toujours qu'un indice de boucle comme variable: c'est la solution que nous appelons copie par excès. Elle est exposée au paragraphe suivant dans le cas d'une dimension quelconque. Enfin la zone des tableaux bidimensionnels qui est accédée par un supernoeud est aussi un parallélogramme puisqu'il n'y a que changement de base. Ce parallélogramme peut être *tassé* en réduisant chaque $\vec{\alpha}_i$ par le PGCD de ses composantes, et mis sous forme triangulaire comme le supernoeud.

Il est alors possible de le stocker sous une forme continue en utilisant la fonction modulo dans les accès. Géométriquement, cela revient à couper une partie triangulaire du parallélogramme et à la mettre de l'autre côté pour en faire un rectangle. Bien que ce stockage ne soit pas linéaire, les lignes et les colonnes du tableau sont rangées régulièrement. Un exemple est donné figure 3.5.2.b. La matrice de triangularisation est encore appelée G bien qu'elle n'ait pas de rapport avec la matrice G qui a servi à effectuer la génération de code. Le tableau de copie est déclaré par:

DIMENSION T'(0:4,0:5)

et accédé par λ_1 et λ_2 sous la forme

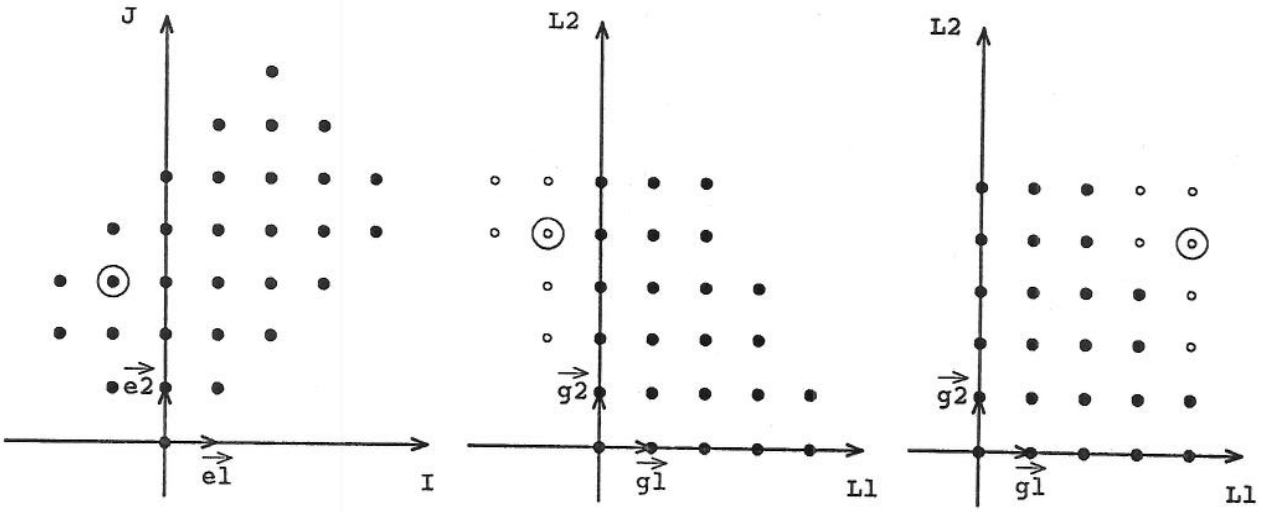
T'(*mymod*(λ_1,5),λ_2)

On peut remplacer λ_1 et λ_2 par I et J en utilisant la matrice de changement de base G^{-1}:

$$G = \begin{pmatrix}1 & 0\\1 & 1\end{pmatrix} \qquad G^{-1} = \begin{pmatrix}1 & 0\\-1 & 1\end{pmatrix} \qquad \begin{pmatrix}\lambda_1\\\lambda_2\end{pmatrix} = G^{-1}\begin{pmatrix}I\\J\end{pmatrix}$$

Ainsi, n'importe quelle référence bidimensionnelle du type T(E1,E2) devient T'(*mymod* (I,5),J–I), quelles que soient les expressions E1 et E2, tant qu'elles permettent de faire un calcul de dépendance pour montrer que la référence est bien bidimensionnelle. La mise en correspondance de T et de T' ne se fait que pendant les opérations de copie. Dans le cas où T n'est référencé qu'en partie droite (utilisation), E1 et E2 peuvent être quelconques et on peut donc profiter de la copie pour supprimer une indirection†.

† Pour les machines dont la mémoire locale, le cache et/ou la mémoire globale favorisent un pas d'accès de 1, la copie permet aussi d'opérer une transposition.



Stockage par modulo

- Figure 3.5.2.b -

Dans les cas favorables, G peut donner un domaine rectangulaire de T qui va pouvoir s'accéder de manière standard sans appel à *mymod* .

Nous n'avons cependant pas approfondi l'étude des stockages non-linéaires car ils sont incompatibles avec le seul mode d'adressage efficace des processeurs vectoriels, le pas constant[†]. Nous proposons donc aussi d'utiliser la copie par excès pour les tableaux bidimensionnels.

Dans le cas de tableaux linéarisés par le programmeur, qui ont des constantes multiplicatives importantes dans leurs références, il faut faire un test de d'autodépendance pour voir s'il ne s'agit pas en fait d'un tableau de dimension 2. S'il n'y a pas autodépendance, on gagnera de la place en traitant un tel tableau comme un tableau bidimensionnel. Malheureusement, le test d'autodépendance ne peut pas être fait avec comme domaine d'itération un supernoeud, car il dépendrait alors d'une taille qui n'a pas encore été fixée: il faut décider du type de stockage avant de calculer la taille du supernoeud dans la phase

[†]Seuls les processeurs de traitement du signal proposent des modes d'adressage à base de modulo pour pouvoir stocker les K derniers points ou *bit-reversal* pour la transformée de Fourier.

d'optimisation qui précède la génération de code. Il faut se contenter du domaine d'itération, qui a moins de chances de donner un résultat de non-autodépendance.

3.5.3. Copie par excès avec accès linéaire

La copie par excès est une méthode compatible avec des dimensions supérieures à 2. Tout d'abord, on détermine la dimensionnalité du parallélépipède relatif à une référence en calculant la dimension du système formé par les formes linéaires $\vec{\alpha}_i$ et non en prenant la dimension déclarée du tableau. On identifie donc bien les lignes, colonnes et diagonales de matrices. Exemple:

$$\text{T(I,I-1)} \rightarrow \vec{\alpha}_1 = \vec{\alpha}_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \rightarrow \text{T'(I)}$$

De plus ces formes sont réduites par le PGCD de leurs composants pour ne pas avoir d'éléments inutiles, jamais accédés, dans le tableau de copie. C'est avec un système libre et générateur de ces formes réduites qu'on évalue la dimensionnalité et les dimensions du plus petit parallélépipède rectangle englobant.

Les dimensions du parallélépipède sont trouvées par un calcul de min et de max sur les quatre sommets du supernoeud puisque les expressions d'indices sont linéaires et donc convexes.

La génération de code est aisée puisqu'il suffit de calculer un offset pour chaque dimension du tableau copie, et l'évaluation du volume est immédiate en fonction de la taille du supernoeud.

L'inconvénient de cette méthode est le gaspillage d'espace mémoire. Ce gaspillage augmente avec la dimension du tableau. Il est nul en dimension 0, quasi nul en dimension 1, car la différence entre l'espace utilisé et l'espace nécessaire est constante, mais proportionnel au volume utile en dimension 2.

Si on reprend l'exemple de la figure 3.5.2.b, on utilise 56 éléments au lieu de 30:

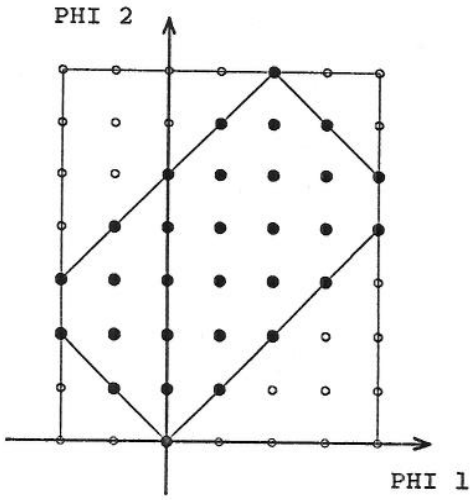
DIMENSION T'(-2:4,0:7)

et n'importe quelle référence bidimensionnelle devient T'(I-I0,J-J0), où I0 et J0 sont les coordonnées de l'origine du supernoeud (figure 3.5.3).

3.5.4. Copie exacte avec accès pseudo-linéaire

La copie exacte avec accès linéaire est impossible comme on l'a vu en 3.5.2. Cependant, il est possible d'avoir des accès linéaires le long de certaines directions et cela peut être suffisant si ce sont justement les directions dans lesquelles se font les opérations vectorielles. C'est ce que nous appelons de l'accès pseudo-linéaire.

Les seuls tableaux pour lesquels la copie par excès n'est pas satisfaisante sont les tableaux bidimensionnels, car l'espace mémoire gaspillé n'est pas négligeable. Mais dans le cas bidimensionnel, il y a bijection entre un noeud du supernoeud et un élément de la copie, et un système de parcours des noeuds



Copie par excès avec accès linéaire

- Figure 3.5.3 -

devrait permettre de parcourir aussi l'ensemble des éléments de la copie. Toutes les méthodes de parcours étudiées en 3.3.1 ne conviennent pas car le stockage des tableaux en Fortran se fait avec des domaines **rectangulaires** et non avec les domaines triangulaires dont s'accomodent les boucles DO. Le seul parcours rectangulaire que nous avons trouvé est le parcours basé sur les vecteurs d'offset.

Cela reviendrait à définir les tableaux de copie comme des tableaux tridimensionnels, ce qui n'est pas gênant, mais aussi à ne pouvoir effectuer les copies que par programme.

La non-linéarité de la technique est maintenant dans le changement de repère de B vers le repère tridimensionnel. Elle ne permettra de générer du code vectoriel que le long de $\vec{p}_1$ et de $\vec{p}_2$.

Une partie du changement de repère est connue. Elle est fondée sur P^I, que nous allons noter P pour alléger les écritures. Dans le cas illustré figure 3.5.4:

$$P = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \quad P^{-1} = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \quad \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = P^{-1}\begin{pmatrix} I \\ J \end{pmatrix}$$

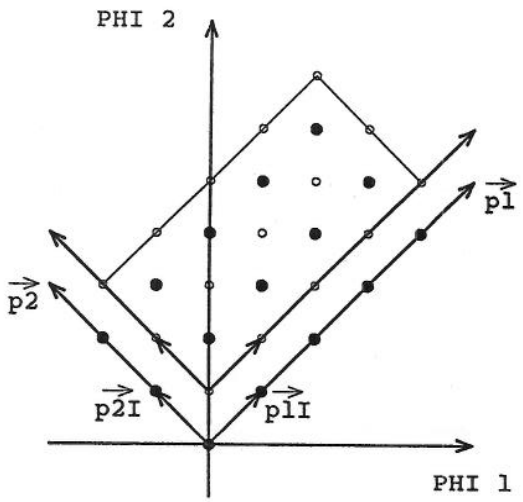
Le calcul de la coordonnée *off* s'appuie sur le fait que tous les vecteurs entiers dans B ont des coordonnées réelles dans P qui sont des multiples de $det(P)^{-1}$. En effet, tous les vecteurs entiers de B ont un représentant dans la cellule unité de P (cf. 3.3.1.1). Pour ne pas obtenir par combinaison linéaire plus de $det(P)$ représentants (groupe à $det(P)$ éléments), la condition sur les coordonnées doit être

vérifiée. On évite les calculs sur R en éliminant l'effet de la division entière par $det(P)$ dans P^{-1} et on choisit arbitrairement de faire le calcul avec la coordonnée relative à $\vec{p}_1$:

$$\begin{pmatrix} \lambda_1' \\ \lambda_2' \end{pmatrix} = P^{-1} \begin{pmatrix} I \times det(P) \\ J \times det(P) \end{pmatrix} \qquad off = mymod(\lambda_1', det(P))$$

Le choix de $\vec{p}_2$ et de λ_2' aurait simplement modifié l'ordre des plans de T'.

Toute référence bidimensionnelle à T devient soit T'(λ_1,λ_2,*off*) si le contrôle est généré par la méthode 1, et T'((I+J)/2,(J–I)/2,*off*) sinon, toujours dans le cas de la figure 3.5.4.



Copie exacte avec accès pseudo-linéaire

- Figure 3.5.4 -

3.5.5. Références multiples au même tableau

Les références multiples au même tableau posent le problème usuel de cohérence des copies. S'il est possible de montrer qu'il n'y a jamais dépendance entre deux références, le problème de cohérence ne se pose pas et un tableau de copie va être généré pour chaque référence. Un exemple typique est:

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

si un calcul de prédicat nous permet de nous rendre compte que K n'est pas modifié dans le corps de boucle. Une constante numérique, à la place de K, rendrait le test de non dépendance encore plus facile.

S'il y a une dépendance constante entre deux références, c'est-à-dire si les formes linéaires d'indices sont les mêmes, une seule copie par excès va être effectuée mais le calcul des dimensions du parallélépipède englobant va prendre en compte les différentes constantes additives qui apparaissent dans les expressions d'indices. Un exemple typique pourrait être:

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

Il faudra évaluer les expressions I-I0 et I-I0+1, J-J0 et J-J0-1 aux quatre sommets du supernoeud. I0 et J0 sont les coordonnées, dans la base initiale, de l'origine du supernoeud. Ils servent aussi à calculer les *offsets* d'accès au tableau copie.

Dans le cas où de nombreuses références forment des dépendances constantes, on parlera par la suite de famille de références.

Enfin, s'il existe une dépendance non constante entre deux références[†] , il faut considérer l'union des deux régions accédées et en faire une copie unique. Il est possible d'approximer cette union par l'enveloppe convexe des deux régions référencées, mais cette enveloppe convexe va varier d'un supernoeud à un autre et, pour avoir un code de supernoeud complet unique, il va falloir prendre l'enveloppe convexe de toutes ces enveloppes convexes. Cette enveloppe globale peut atteindre une taille ridiculement grande. Dans le cas suivant:

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

les supernoeuds affectés aux coins supérieur droit et inférieur gauche se verraient réserver l'ensemble de

[†]Ceci implique que les deux références sont distinctes.

la matrice à transposer et l'intérêt du partitionnement en tant que technique d'adaptation de la taille des tâches aux ressources disponibles serait nul.

Si on sait distinguer dynamiquement les supernoeuds pour lesquels les deux références créent une dépendance et ceux pour lesquels il n'y a pas dépendance, on peut adopter deux stratégies de copie. Pour les supernoeuds sans dépendance, deux copies, une par référence, sont créées et, pour les supernoeuds à dépendance, une copie de l'approximation de l'enveloppe convexe des deux régions référencées est effectuée. Cette fois, l'enveloppe convexe ne pourrait pas prendre des proportions gigantesques du fait de l'existence d'une dépendance: les deux zones à envelopper ont une intersection non vide et un volume limité par la taille du supernoeud; le volume de leur union est inférieur à la somme de leur volume. Pour n'avoir qu'un type de supernoeud à enveloppe convexe, il faudra être capable de prévoir tous les cas possibles et de trouver leur enveloppe. De plus, ceci ne sera viable que si le test de dépendance est rapide à exécuter. Enfin, le nombre de types de supernoeuds complets augmente exponentiellement avec le nombre de couples de références créant des dépendances non constantes.

3.5.6. Variables locales

Les variables locales au corps de boucle n'ont besoin ni d'être copiées de la mémoire globale à la mémoire locale avant l'exécution du supernoeud, ni de la mémoire locale à la mémoire globale à la fin de l'exécution.

La détection de variables locales scalaires se fait en calculant les Use-Definition-Chains[20] sur le corps de boucle. L'analyse est très facile sur de petits corps de boucles. Il ne faudrait pas que ces variables soient expansées par la phase de parallélisation préalable. La détection de la localité serait plus difficile et l'expansion à l'ensemble du domaine d'itération dommageable.

Les tableaux posent un problème plus délicat car il est difficile dans le cas général de savoir si un tableau est entièrement initialisé, ou plutôt si au moins tous les éléments de ce tableau qui sont utilisés dans le corps de boucle sont bien initialisés auparavant. Si le corps de boucle est simple, sans IFs ni CALLs, il suffit de vérifier que les références sont strictement identiques et qu'on commence par une écriture. Sinon, nous ne connaissons pas de méthode générale pour ce problème. Seul Kuhn[22] propose un algorithme de test d'*Array Covering*, à la suite du problème de la couverture des dépendances.

De toute façon, la détection des variables locales peut être imparfaite. Des transferts de données inutiles seront effectués, mais le code généré sera correct.

3.5.7. Références et variables en entrée ou en sortie

Une référence est dite *en entrée* si elle apparaît en partie gauche d'une affectation, dans un ordre de lecture ou comme argument *IN* ou *IN/OUT* d'un sous-programme ou d'une fonction. Elle est dite *en sortie* si elle apparaît en partie droite d'une affectation, dans une expression booléenne de test, dans un ordre

d'écriture ou comme argument *OUT* ou *IN/OUT* d'un sous-programme ou d'une fonction. Ces deux listes sont encore incomplètes (COMMON et variables de contrôles des entrées-sorties par exemple) et une première maquette de *partitionneur automatique* pourrait être limitée aux corps de boucles composés d'une suite d'affectations. Le calcul complet des variables en entrée ou en sortie est du ressort d'un paralléliseur interprocédural[28, 29] et nous n'étudierons pas ce problème davantage ici.

Pour les variables qui ne sont référencées qu'une fois, il suffit de déterminer la nature de la référence. Pour les variables qui sont référencées plusieurs fois, sans qu'apparaissent des dépendances entre les références, on peut traiter chaque référence indépendamment. En ce qui concerne les familles de références et les approximations d'un ensemble de références par des enveloppes convexes, on peut toujours tout copier par défaut sans détruire la correction des calculs. La copie partielle du tableau local peut se faire référence par référence, mais il n'est pas sûr, si l'intersection des références est importante, que le temps de copie sera diminué. Si toutes les références d'une famille sont *IN* ou *OUT*, seule la copie correspondante sera programmée.

L'optimisation des mouvements de données semble assez délicate. Il faudrait avoir plus d'expérience pratique, avec des machines qui nécessitent la programmation de mouvements de données, pour estimer les gains en temps que pourrait apporter une optimisation poussée.

3.5.8. Déclarations des tableaux de copie

Les déclarations des tableaux de copie ne posent pas de problème puisqu'on sait en évaluer les bornes inférieures et supérieures (cf 3.5.3), une fois qu'on a décidé des relations entre références et tableaux de copie (cf 3.5.5).

3.5.9. Conclusion

La génération de variables locales ne semble pas être facile à effectuer d'une manière automatique. Les machines à base de caches simplifient considérablement le travail de programmation, et donc la génération automatique de code. Ceci est vrai même si le cache doit être géré partiellement par le processeur avec des ordres d'invalidation ou de vidage et/ou par les compilateurs qui doivent marquer certaines variables comme *cachable* ou *non-cachable* pour garantir la cohérence dans une architecture multiprocesseur.

Le gain en performance qu'on peut attendre d'une mémoire locale gérée optimalement par rapport à un cache géré selon une politique fixe et indépendante de la nature des traitements ne semble pas devoir être important. En effet, les optimisations de temps d'exécution faites sous l'hypothèse d'un comportement parfait du cache sont valides[19] tant que les données qu'il faudrait maintenir en cache ne sont pas plus volumineuses que celui-ci. Ce sera bien évidemment une contrainte qui sera respectée par les supernoeuds.

Enfin, nous n'avons pas évalué les complications qu'introduirait la présence d'appels de sous-programmes ou de fonctions dans le corps de boucle.

3.6. Génération du code de copie

La génération du code de copie n'est pas toujours nécessaire: le matériel peut décider dynamiquement des copies à effectuer (antémémoire). Le code dépend à la fois de la génération des tableaux partiels (cf. 3.5) et des organes de copie de la machine.

3.6.1. Copie programmée

La copie programmée ne pose pas de problème puisqu'on est libre de faire ce qu'on veut. Elle se fait famille de références par famille de références pour éviter de copier plusieurs fois le même élément de tableau.

Les tableaux locaux bidimensionnels peuvent avoir été déclarés avec des dimensions définies approximativement. Cela crée une petite incertitude sur l'existence d'une fonction associant un élément du tableau copie à un élément du tableau réel, surtout pour les supernoeuds partiels. Pour éviter un accès mémoire invalide ou l'écrasement d'un élément de tableau voisin, il faut se baser sur le code de contrôle du supernoeud correspondant, partiel ou complet, pour effectuer les transferts de tableaux bidimensionnels. Cela permet aussi de ne transférer que les éléments utiles.

Pour les tableaux monodimensionnels, dont l'approximation est meilleure, les trous sont forcément à l'intérieur des bornes fixées par les déclarations et on peut utiliser ces dernières pour générer le contrôle en initialisation, mais pas en mise à jour. Les éléments transférés inutilement seront très peu nombreux avec les formes linéaires d'accès usuelles et, comme leur nombre ne croît pas avec la taille du supernoeud, cette source d'overhead peut être rendue négligeable. Les supernoeuds partiels doivent recevoir un code de copie spécifique, qu'on les optimise ou non. Dans ce dernier cas, les copies non valides doivent être évitées par des tests.

Dans ces deux cas, mono- et bidimensionnel, les formules de changement de base des indices se déduisent aisément du type de copie choisi, du repère de parcours utilisé et de l'origine du supernoeud.

La copie des variables scalaires est une simple affectation.

3.6.2. Transfert par message

Le transfert par message ne permet pas, en général, de transférer des éléments non contigus, même s'ils sont séparés par un écart constant (*constant stride*). Il faut donc constituer le message à transférer exactement comme si on effectuait une copie programmée (cf. 3.6.1), sauf dans le cas des tableaux bidimensionnels de copie qu'il faut remplir complètement d'après leurs dimensions déclarées, en utilisant une valeur par défaut si nécessaire, et non d'après le code de contrôle du supernoeud. Pour les scalaires, il faudrait combattre l'overhead d'initialisation de message (quelques millisecondes sur l'iPSC) en les mettant tous dans un message unique.

A la réception du message, on peut heureusement utiliser la zone d'arrivée pour stocker le message, aux problèmes de contrôle d'erreurs près, puisqu'on ne génère que des zones de copie contiguës. Si les scalaires sont regroupés, il faut ajouter le code de dépackage ou bien remplacer chaque occurrence de scalaire par une référence au tableau des scalaires, que la phase d'optimisation du compilateur transformera en un accès scalaire en principe.

3.6.3. Opérateurs spécialisés de transfert

Les capacités des opérateurs de transfert sont spécifiques de la machine cible. D'une manière générale, leur utilisation efficace suppose qu'ils soient au moins capables de transférer des éléments avec deux pas différents en lecture et en écriture. Cette facilité permet de supporter tous les changements de base ou d'échelle car ce sont des opérations linéaires qui se traduisent par des incréments constants.

Un opérateur de transfert acceptant d'effectuer des indirections serait certainement plus lent et sans utilité tant qu'on n'essaie pas de faire une copie optimale et qu'on se limite aux copies par excès.

Enfin, l'utilisation efficace d'un opérateur de copie transférant uniquement des blocs dont la taille n'est pas ajustable à un mot près, pose d'autres problèmes, trop spécifiques pour être abordés ici.

3.6.4. Exécution parallèle des transferts et de données des calculs

L'exécution parallèle des transferts de données et des calculs est envisageable pour les machines disposant d'opérateurs spécialisés et de mémoires locales ayant une bande passante suffisante pour que calculs et transferts ne se ralentissent pas trop mutuellement.

La génération automatique des synchronisations nécessaires entre transferts et calculs, et la séparation des codes de copie et d'exécution d'un supernoeud pour obtenir de nouveaux supernoeuds associant à un calcul la recopie des résultats du précédent et le chargement des données du suivant ne posent pas en soi de problème. Mais la gestion de la mémoire locale sera perturbée puisqu'il faudra prendre en compte des buffers[†] , et les hypothèses concernant le calcul du temps d'exécution devront être revues car le préchargement introduit une rigidité dans le séquencement des supernoeuds: les ordonnancements dynamiques instantanés, comme l'ordonnancement par les données, ne sont plus possibles. De plus les supernoeuds initiaux et finaux constituent des cas particuliers.

Cette voie ne sera pas poursuivie dans ce travail car les seules machines disposant de tels opérateurs sont pour le moment les hypercubes, que nous avons éliminés au paragraphe 1.3.3 pour cause d'absence de mémoire globale et de temps d'échange fluctuants, et le projet RP3, qui peut prendre des configurations ne nécessitant pas l'utilisation d'opérateurs de copie et sur lequel les détails nous manquent.

[†] Cela ne revient pas à diminuer artificiellement la taille mémoire, car la taille des buffers va varier avec la taille des supernoeuds.

3.6.5. Comparaison des différents types de copie

Il est impossible de choisir un type de copie particulier avant d'avoir mené des expériences sur des programmes réels. Il fallait, dans ce travail, montrer qu'il existait des solutions même si aucune d'entre elles n'est idéale.

	mise en oeuvre	directions à accès linéaire (vectorielles)	utilisation de l'espace	opération de copie
approximation rectangulaire dans B	+ +	∞	−	+ +
rectangularisation par modulo dans G	+	1	+	− −
repère 3 D	-	2	+	-

Avantages et inconvénients des différents types de copie

- Table 3.6.5 -

Ce tableau est incomplet. Ni les interférences entre le code de contrôle et le type de stockage, ni le problème des dépendances variables, ni le fait que le seul cas délicat soit le cas des références bidimensionnelles, n'apparaissent.

Un exemple de génération de code de copie est donné au chapitre 7.

3.7. Conclusion

La génération de code est plus ou moins facile suivant les machines. Le calcul de la structure de contrôle d'un supernoeud, spécialement si on veut préserver la possibilité d'une exécution vectorielle, n'est pas simple, mais conduit à des résultats uniformes.

Une bonne génération de code est importante pour pouvoir faire des approximations valides lors du choix d'une taille de supernoeud et pour que les améliorations espérées ne soient pas détruites par l'overhead de codage quand le seul gain provient d'une meilleure utilisation du cache.

L'optimisation du code des supernoeuds partiels est plus compliquée mais nécessaire pour ne pas reposer trop sur des approximations basées sur le rapport surface/périmètre du domaine d'itération, approximations qui peuvent être invalides dans le cas des domaines très plats, et qui ne sont valables que pour de larges domaines d'itération.

Il nous semble que l'utilisation des polyèdres convexes et des changements de base est très prometteuse puisqu'elle fournit à la fois un cadre unique permettant de traiter l'ensemble des problèmes que nous posait la génération de code avec supernoeuds et des résultats aussi bons (et surtout justes...) que ceux qu'on pourrait obtenir à la main. Elle s'applique de plus à d'autres transformations de programme comme la méthode hyperplane ou l'échange de boucles.

Mais, plus que tout, c'est la gestion des variables locales qui complique la génération de code, bien que les polyèdres convexes résolvent partiellement ce problème subsidiaire. Il n'y a pas une technique unique à raffiner, il faut en développer une par machine. La gestion manuelle, par un programmeur, des variables mises en mémoire locale devrait s'avérer délicate et peut-être décourageante, vu les tailles et les efficacités qu'ont les caches avec des programmes qui tiennent compte de leur présence. La génération automatique de ce code peut sembler provoquer un overhead de copie trop important par rapport au gain espéré, mais il ne faut pas oublier que sur des machines à plusieurs espaces d'adressage indépendants, comme Marianne ou les hypercubes, les copies sont indispensables.

Notons enfin que la différence entre cas usuel et cas général a été souvent faite, vérifiant une fois de plus la loi des 80/20 ou même une loi 99/01: 99 % des problèmes sont posés par 1 % des cas rencontrés dans la pratique. Les cas usuels sont très favorables en ce qui concerne le partitionnement, le parallélisme et l'exécution vectorielle. En revanche, ils ne permettent pas de trouver une solution simple pour l'allocation de tableaux de copie.

References

1. W. B. Ackerman and J. B. Dennis, "VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual," MIT-LCS-TR-128 (1979).

2. W. B. Ackerman, "Dataflow Languages," *1979 Nat. Computer Conf.*, pp.1087-1095 (June 1979).

3. Arvind, K. P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," TR 114a, Dept. of Information and Computer Science, Univ. of California, Irvine (Dec. 1978).

4. Arvind, "A Processing Element For a Large Multiprocessor Dataflow Machine," *Int. Conf. Circuits and Computers* (Oct. 1980).

5. U. Banerjee, "Speedup of Ordinary Programs," Report No UIUCDCS-R-79-989, University of Illinois (1979). PhD Thesis

6. D. D. Chamberlin, "The Single Assignment Approach to Parallel Processing," *Proc. Nat. Computer Conf.* (Nov. 1971).

7. A. P. Chang, "A Note on the Modulo Operation," *SIGPLAN Notices* **20**(4), pp.19-23 (April 1985).

8. D. Comte and N. Hifdi, "LAU Multiprocessor: Microfunctional Description and Technological Choices," *1st. European Conf. Parallel and Distributed Processing*, pp.8-15 (Feb. 1979).

9. M. Cornish, "The TI Dataflow Architectures: The Power of Concurrency For Avionics," *3rd Conf. Digital Avionics Systems*, pp.19-25 (Nov. 1979).

10. R. G. Cytron, "Compile-Time Scheduling and Optimization for Asynchronous Machines," PhD Thesis, Report No. UIUCDCS-R-84-1739, University of Illinois at Urbana-Champaign (1984).

11. J. B. Dennis, "A Highly Parallel Processor Using a Data Flow Machine Language," Tech. Rep. CSG Memo 134-1, MIT (June 1979).

12. R. J. Duffin, "On Fourier's Analysis of Linear Inequality Systems," *Mathematical Programming Study 1*, North-Holland (1974).

13. D. D. Gajski, D. H. Lawrie, D. J. Kuck, and A. H. Sameh, "CEDAR," pp. 306-309 in *Spring Compcon 84*, Computer Society Press, San Francisco (1984).

14. O. Gelly, "LAU Software System: A High Level Data Driven Language for Parallel Programming," *1976 Int. Conf. Parallel Processing*, p.255 (Aug. 1976).

15. R. Gottlieb, K. Kimball, T. Jaskiewicz, and R. Swift, "A New Way To Speed Up a Supercomputer," *Electronics* **58**(30), pp.56-58 (July 1985).

16. H.-C. Hoppe and H. Muehlenbein, "Parallel Adaptative Full-Multigrid Methods on Message-Based Multiprocessors," *Parallel Computing* **3**(4), pp.269-287 (Oct. 1986).

17. R. G. Babb II, "Parallel Processing with Large Grain Data Flow Techniques," *Computer* **17**(7), pp.55-61 (July 1984).

18. F. Irigoin and R. Triolet, "Supernodes and Alliant FX/8 Minisupercomputer," Tech. Rep. ENSMP-CAI-86-081, Ecole des Mines de Paris (Aug. 1986).

19. W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy," Technical Report CSRD, University of Illinois at Urbana-Champaign, Urbana-Champaign (Feb. 1986).

20. K. Kennedy, "A Survey of Data Flow Analysis Techniques," pp. 5-54 in *Program Flow Analysis, Theory and Applications*, ed. N. D. Jones, Prentice Hall (1981).

21. P. R. Kosinski, "A Dataflow Programming Language," Tech. Rep. RC 4264, IBM T. J. Watson Research Center, Yorktown Heights (March 1973).

22. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

23. J. R. McGraw, "Data-Flow Computing: The VAL Language," *ACM TOPLAS*(4), pp.44-82 (1982).

24. J. R. McGraw and S. K. Skedzielewski, "SISAL: Streams and Iterations in a Single-Assignment Language, Language Reference Manual," Tech. Report M-146, Lawrence Livermore National Laboratory (July 1983).

25. M. Minoux, *Programmation Mathématique - Théorie et algorithmes (tome 2)*, Dunod (1983).

26. A. H. Sameh and D. Kuck, "On Stable Parallel Linear System Solvers," *Journal of the ACM* **25**, pp.81-91 (Jan. 1978).

27. V. P. Srini, "An Architectural Comparison of Dataflow Systems," *IEEE Computer* **19**(3), pp.68-88 (March 1986).

28. R. Triolet, "Contribution à la parallélisation automatique de programmes," Thèse de Docteur-Ingénieur, Université Paris VI (Déc. 1984).

29. R. Triolet, F. Irigoin, and P. Feautrier, "Direct Parallelization of CALL Statements," *SIGPLAN'86 Symposium on Compiler Construction* (June 1986).

30. M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD. Thesis, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign (1982).

4. COUTS ET CONTRAINTES DUS AUX MIGRATIONS DES DONNEES

Ce chapitre a pour but de quantifier les mouvements de données qu'il faut prévoir en fonction de l'architecture, que ces mouvements soient effectués automatiquement par la machine dans le cas des caches et de la mémoire virtuelle, ou qu'ils soient explicitement gérés par le programmeur et exécutés soit par un processeur spécialisé en transferts soit par les processeurs élémentaires de calcul.

Un certain nombre d'hypothèses restrictives ont été faites au chapitre 1 sur les machines et les programmes considérés. Nous rappelons brièvement ici les deux qui concernent les transferts. Tout d'abord l'indépendance des performances des processeurs vis-à-vis de leur environnement implique que les transferts de données soient indépendants les uns des autres. Ensuite, les expressions d'indice des références doivent être des fonctions affines des indices des boucles englobantes.

Dans un premier temps, nous allons voir qualitativement quels types de données doivent être transférés en fonction de l'architecture des machines (§ 4.1). Ensuite, nous exposerons quelques méthodes permettant d'obtenir des informations **symboliques** sur le volume de données à transférer à partir des références présentes dans le corps de boucle (§ 4.2). Ces informations permettront d'optimiser ou d'influencer soit la forme des supernoeuds, soit l'ordonnancement des calculs à l'intérieur d'un supernoeud. Puis nous essaierons de calculer le volume de données qu'il faut stocker simultanément dans la mémoire la plus rapide pour en dériver une contrainte sur la taille des supernoeuds (§ 4.3). Ce volume sera d'autant plus faible que les transferts auront lieu juste quand les données transférées seront nécessaires (§ 4.4).

4.1. Types d'architectures étudiés

Nous allons étudier l'influence des architectures présentées au chapitre 1 sur les transferts de données. Quand faut-il les effectuer? Quelles sont les données échangées? Quelle peut être l'influence du partitionnement et de l'ordonnancement des calculs? Telles sont les questions auxquelles nous essaierons de répondre.

Bien entendu, ce paragraphe est sans objet dans le cas des architectures avec un seul niveau mémoire. C'est le cas par exemple avec le Cray X-MP dont la bande passante mémoire a été choisie suffisamment importante pour ne pas pénaliser trop sévèrement les temps d'exécution et se passer d'antémémoire et de mémoire locale, du moins pour les unités vectorielles.

4.1.1. Cache local

Les caches locaux posent deux problèmes: un problème de cohérence, puisque deux copies de la même case mémoire peuvent être modifiées indépendamment et un problème de taille, puisqu'il faut éviter de provoquer le déchargement d'une case qu'on va réutiliser ultérieurement.

Cohérence

La présence de caches locaux dans les multiprocesseurs pose le problème de la cohérence des copies. Si deux processeurs modifient indépendamment une même cellule mémoire dont la valeur a été copiée dans leur cache respectif, le résultat global de l'opération va dépendre de l'ordre des lectures et des écritures. Ce problème a été largement étudié dans les systèmes d'exploitation et dans les bases de données sur le plan logiciel mais il a également été traité dans le cas des caches[4, 5, 3] . R. L. Lee donne de nombreuses références bibliographiques dans [13] mais aucune des solutions proposées ne lui semble satisfaisante, car elles impliquent un ralentissement des caches et un coût non linéaire par rapport au nombre de processeurs. Le projet SPUR[9] a quand même adopté cette solution en implémentant un mécanisme appelé *Berkeley Ownership*.

Le projet RP3[2] a abordé le problème d'une manière différente, en reportant le problème du maintien de la cohérence sur les compilateurs et les utilisateurs. Plusieurs instructions permettent de gérer le cache explicitement, par exemple pour le vider, et les données peuvent être marquées comme *non-cachables*.

La méthode que nous proposons s'insère particulièrement bien dans ce contexte puisque, comme nous l'avons déjà remarqué en 1.2.3, deux supernoeuds qui s'exécutent simultanément ne peuvent pas partager une cellule mémoire en écriture ou risquer de lire une cellule aléatoirement avant ou après une écriture. Ces deux cas impliqueraient des dépendances entre les deux supernoeuds, dépendances incompatibles avec les principes qui ont conduit à leur création.

Le problème de mise à jour des données globales ne se pose qu'à la fin de l'exécution d'un supernoeud. Il faut que les résultats produits soient renvoyés en mémoire globale. Dans le cas du projet RP3[2] ,chaque supernoeud devrait exécuter une instruction *Flush Cache* avant de signaler sa terminaison, tandis que le mécanisme matériel de SPUR[9] nous permet d'ignorer totalement le problème. Enfin, la documentation que nous avons obtenue sur l'IBM 3090/400 VF ne nous permet pas de savoir comment deux processus peuvent communiquer via la mémoire globale et donc si des problèmes de cohérence peuvent ou non se poser. Il faudra attendre l'arrivée d'autres architectures à caches locaux pour voir quelles seront les solutions proposées au problème de cohérence. Il nous semble raisonnable de dire que nous saurons assurer la cohérence à la compilation quelles que soient les primitives de gestion de cache proposées.

Taille

L'objectif est de garder dans le cache toutes les données qui sont utilisées plusieurs fois pendant l'exécution d'un supernoeud. Le volume qu'il faut réserver pour une référence dépend de l'ordre dans lequel les données vont être accédées. Si tous les accès à une donnée qui sont dus à cette référence ont lieu successivement, il suffit de réserver une case du cache pour la référence. Par contre, si l'ensemble des données accédées est parcouru plusieurs fois par la référence au cours de l'exécution il va falloir garder toutes les données référencées en cache. Il y a aussi des cas intermédiaires où il faut garder une ligne ou une colonne d'un tableau.

Dans le cas de la multiplication de matrice exécutée dans l'ordre IJK, pour laquelle nous obtiendrions des supernoeuds ayant le code suivant:

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, P
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
```

Multiplication de matrice

il ne faut compter qu'une case dans le cache pour les deux références A(I,J). Le tableau C doit être contenu intégralement dans le cache car l'ensemble des données qui est référencé par C(K,J) est référencé N fois: il faut associer à cette référence M $\times$ P cases. Enfin la référence B(I,K) constitue un cas intermédiaire puisque chaque colonne B(I,*) est parcourue M fois successivement. Il suffit donc de prévoir P cases de cache[†] pour elle. Il n'y a pas dans cet exemple de références du genre D(I,J,K) qui conduisent à

[†]Il faut tenir compte de la taille des blocs que gère le cache et du mode de rangement des tableaux. Les éléments de B qui sont accédés successivement ne sont pas contigus en mémoire (Fortran). La zone de stockage de B aurait donc une taille de 4 $\times$ P mots sur la machine SPUR et sur l'Alliant FX/8 qui ont des blocs de

marquer D comme étant inutile à ranger en cache.

Comme on le voit informellement sur cet exemple, le calcul du volume à stocker dépend de l'ordonnancement des calculs à l'intérieur d'un supernoeud. De même, le caractère *cachable* ou non d'une variable varie selon les références et leurs environnements.

Ces évaluations supposent que le cache est parfaitement géré. En cas de défaut de cache, la nouvelle adresse cachée remplace une adresse qui ne sera plus utilisée toutes les fois où c'est possible. D'après W. Jalby[10] , cette hypothèse semble acceptable même pour des algorithmes de gestion de cache aussi simples que celui qui est implémenté dans l'Alliant FX/8.

On suppose aussi que les concepteurs ont choisi une politique *write-back* et non une politique *write-through*, c'est-à-dire que les écritures dans la mémoire n'ont lieu que quand il faut faire de la place dans le cache et non quand le processeur exécute une écriture en mémoire. Ce type de cache est plus coûteux à implémenter, mais il charge beaucoup moins la mémoire. Dans le cas de la multiplication de matrice, le nombre d'écritures de A(I,J) est divisé par P et l'optimisation consistant à stocker A(I,J) dans un registre ou dans un scalaire devient même inutile dans le cas de l'Alliant FX/8 car son cache est aussi rapide que ses registres.

Le coût de migration va être égal au nombre de chargements et de déchargements du cache. Ce dernier nombre peut soit être limité au nombre de données modifiées si le cache intègre le drapeau nécessaire, soit être égal au nombre de chargements, soit enfin être égal au volume du cache si l'opération de *flush* est générale.

Si deux supernoeuds sont exécutés successivement sur le même processeur, le second peut trouver dans le cache des données qui le concernent sans avoir à en provoquer le chargement (*héritage*). Nous faisons donc une hypothèse pessimiste en considérant que le cache est vierge au début et à la fin d'une tâche.

Pour minimiser le coût de transfert, on choisit de contraindre la taille de l'espace mémoire utilisé par les supernoeuds par le volume du cache.

4.1.2. Cache global

La gestion d'un cache global est plus simple que celle des caches locaux puisqu'il n'y a plus de problème de cohérence entre plusieurs copies d'une même donnée. La difficulté est simplement reportée sur le matériel puisque le cache global doit avoir une bande passante satisfaisant les besoins de tous les processeurs. C'est le choix qui a été fait sur l'Alliant FX-8[7] .

Ceci dit, les questions qui se posent ensuite sont strictement équivalentes que le cache soit global ou local. Nous supposerons donc à nouveau que le cache est parfaitement géré et que toutes les données qui

---

32 octets.

doivent y résider pendant l'exécution d'un supernoeud peuvent y rester en permanence.

Comme tous les supernoeuds complets ont les mêmes besoins en mémoire et que les supernoeuds partiels sont supposés beaucoup moins nombreux, le volume de cache disponible pour chacun d'eux est égal au volume total divisé par le nombre de processeurs. Ce volume doit être pris en compte à la génération de code. Il ne sera donc pas possible de s'adapter dynamiquement au nombre de processeurs effectivement disponibles, qui peut varier soit à cause de pannes, soit à cause de la multiprogrammation.

L'effet d'héritage signalé en 4.1.1 est renforcé par un effet de partage puisque chaque supernoeud peut profiter occasionnellement des données chargées par des supernoeuds exécutés auparavant ou simultanément. Nous faisons donc une hypothèse pessimiste en considérant que le cache est vierge au début et à la fin d'une tâche et que les parties de cache sont indépendantes.

4.1.3. Mémoires locales et mémoire globale (Cedar, Cray-2)

Le cas étudié ici est celui où chaque processeur peut accéder à volonté sa mémoire locale et la mémoire globale. L'inconvénient par rapport aux caches se situe essentiellement à la génération de code (cf. § 3.5 et § 3.6) puisqu'il faut maintenant réserver des tableaux de copie et programmer les transferts.

Le choix des données à mettre en mémoire locale est équivalent au choix des données à conserver en cache: la copie ne se justifie que si la donnée est utilisée plusieurs fois. Sinon, au lieu de marquer les références correspondantes comme non-cachables, on se contente de les conserver telles quelles en mémoire globale. Les autres références sont transformées en des références aux copies.

Les copies des données utilisées plusieurs fois à la suite doivent être faites au bon moment pour n'utiliser que le nombre minimum de cases mémoires. Le code de la multiplication de matrice devient alors:

```
C'(1:P,1:M) = C(1:P,1:M)
DO I = 1, N
  B'(1:P) = B(I,1:P)
  DO J = 1, M
    A' = 0.
    DO K = 1, P
      A' = A' + B'(K) * C'(K,J)
    ENDDO K
    A(I,J) = A'
  ENDDO J
ENDDO I
```

Multiplication de matrice sur une machine à mémoire locale

Les variables A', B' et C' sont supposées être en mémoire locale tandis que A, B et C sont en mémoire globale.

Le coût des transferts dépend de leurs volumes et des facilités matérielles offertes par la machine. Par exemple, l'unité vectorielle des processeurs du RP3 est capable d'effectuer des transferts mémoire à mémoire. Si les transferts doivent être entièrement exécutés en Fortran, il faudra que les données soient utilisées très souvent ou bien que la mémoire globale soit très lente par rapport à la mémoire locale.

La présence d'un opérateur spécialisé pose aussi le problème du recouvrement des temps de calcul et des temps de transfert, à l'intérieur d'un supernoeud et entre deux supernoeuds successifs.

4.1.4. Mémoires locales et transferts centralisés (multi-array-processor avec host)

La machine dispose toujours de mémoires locales et d'une mémoire globale mais les processeurs élémentaires ne peuvent pas accéder la mémoire globale. Ils doivent demander l'exécution des transferts à un processeur spécialisé.

L'exemple de la multiplication de matrice ne change pas beaucoup a priori puisque toutes les données sont réutilisées: il n'y a pas de références tridimensionnelles du genre D(I,J,K). Cependant, les processeurs de transferts peuvent avoir un temps de start-up non négligeable qui conduirait à regrouper les transferts. Cela ne changerait pas le volume de données transférées mais le volume de mémoire locale nécessaire à l'exécution d'un supernoeud. En prenant comme contrainte la taille de cette mémoire, il faudrait donc réduire la taille des supernoeuds.

```
C'(1:P,1:M) = C(1:P,1:M)
B'(1:N,1:P) = B(1:N,1:P)
DO I = 1, N
  DO J = 1, M
    A'(I,J) = 0.
    DO K = 1, P
      A'(I,J) = A'(I,J) + B'(I,K) * C'(K,J)
ENDDO K, J, I
A(1:N,1:M) = A'(1:N,1:M)
```

Multiplication de matrice sur une machine avec opérateurs de transfert

Le recouvrement des exécutions et des transferts est conditionné par la bande passante de la mémoire locale et par les synchronisations possibles entre les deux types de processeurs. Il vaut mieux ne pas avoir à passer par le système d'exploitation de la machine pour les effectuer. Ce recouvrement nécessite aussi la réservation de buffers pour les transferts.

Si le recouvrement est possible, le coût va intégrer un opérateur non linéaire "MIN". Il va falloir n'utiliser que la moitié de la mémoire locale, et ajouter au code d'un supernoeud le rangement des résultats du supernoeud précédent et le préchargement des données du supernoeud suivant. Cette optimisation induit donc des contraintes sur l'ordonnancement global et le mapping supernoeud-processeur.

4.1.5. Mémoires locales et communications par messages

Nous avons laissé de côté ce type d'architecture pour ne pas avoir à étudier le problème de l'affectation des supernoeuds à des processeurs et le problème de la répartition de la mémoire globale sur les mémoires locales. Cependant, le partitionnement que nous proposons reste une étape nécessaire dans l'adaptation de boucles imbriquées sur une telle machine. Il est donc intéressant de voir comment les choses se passeraient.

Comme dans le cas précédent (§ 4.1.4), les données doivent **toutes** être disponibles en mémoire locale. Rien ne change en ce qui concerne le recouvrement des transferts et des calculs mais le coût n'est plus une fonction affine du volume. Il dépend aussi des positions de l'émetteur et du récepteur, qui peuvent même être confondus si la partie de mémoire globale qu'on cherche à accéder se trouve justement sur le processeur qui en a besoin.

4.1.6. Mémoire virtuelle

La mémoire virtuelle sort un peu du cadre de travail que nous avons choisi mais l'amélioration de localité qu'apporte le partitionnement doit aussi lui être favorable.

L'évaluation des coûts de transfert ne peut plus se faire à partir du volume des données référencées. Il faut prendre en compte la notion de page et le mode de rangement des tableaux multidimensionnels. L'ordonnancement des calculs à l'intérieur d'un supernoeud va non seulement influer sur le scénario (*pattern*) des accès à une même cellule mémoire mais aussi sur le scénario d'accès à des cellules se trouvant dans la même page.

D'autre part, le prix d'un défaut de page est si élevé qu'on peut se demander s'il faut encore faire passer la vectorisation ou la parallélisation avant l'optimisation des transferts. Dans ce cas, il faudrait commencer par repartir du travail d'Abu-Sufah[1] .

Cependant, le partitionnement nous permet d'ajuster le volume de données nécessaire en mémoire locale. Il ne s'agit donc plus de rechercher la courbe du nombre de transferts (i.e. défauts de page) en fonction de la taille de la mémoire réelle disponible mais la relation qui existe entre le nombre de transferts et l'ordonnancement des calculs d'un supernoeud.

Comme dans le cas des caches, il faut faire la même hypothèse de perfection pour l'algorithme de remplacement de pages. Aucune page encore utile n'est déchargée s'il existe une page inutile.

Enfin, il faut remarquer que les caches gérés par bloc posent un problème similaire à celui des pages de la mémoire virtuelle. Dans les deux cas, il faut essayer de parcourir l'espace d'adressage avec un pas de 1.

4.1.7. Conclusion

Quelle que soit l'architecture de la machine, nous avons fait les hypothèses nécessaires en 1.2.3 pour que le coût des transferts soit uniquement fonction du volume transféré et qu'il ne dépende pas de la localisation du processus demandeur ou de la mémoire qui contient les données. Dans la mesure où il n'y a pas recouvrement entre les calculs et les transferts, ce coût est une simple somme de fonctions affines.

A ce coût s'ajoute la contrainte que le volume des données *vivantes* soit conservé inférieur au volume du niveau mémoire le plus rapide. Cette contrainte est nécessaire pour toutes les architectures. Elle permet de garantir que chaque donnée n'est transférée qu'une fois et donc que le volume transféré pour une référence est égal au volume référencé et qu'il est donc indépendant de l'ordonnancement. Ce volume dépend par contre du partitionnement.

La contribution au volume de stockage qu'apporte chaque référence ou famille de références est sensible à la fois à l'architecture de la machine, à l'ordonnancement des noeuds à l'intérieur d'un supernoeud et au partitionnement.

Il en est de même des **instants** auxquels les transferts doivent être exécutés. Ils dépendent aussi bien de l'ordonnancement choisi que de l'architecture de la machine.

4.2. Evaluation des volumes référencés. Optimisation de P.

L'évaluation du volume référencé se fait par référence ou par famille de références. Rappelons que deux références appartiennent à la même famille si elles ont des vecteurs d'autodépendance égaux (cf § 3.5.2). Ce sont les vecteurs d'autodépendance qui traduisent la réutilisation d'une donnée. Nous supposons que les formes linéaires d'accès $\vec{\alpha}_i$ sont libres et réduites par leurs PGCDs.

La notion de **région**[14] n'est pas utilisée parce que les appels de sous-programmes ont été exclus, parce que les bornes des boucles ne contiennent pas de constantes symboliques et parce que nous ne tenons pas compte des tests présents dans le corps de boucle. Dans ces conditions, les régions n'apportent rien par rapport aux simples références.

Même les tests linéaires sur les indices de boucle ne peuvent pas être pris en compte car les évaluations doivent être indépendantes de la position du supernoeud dans le domaine d'itération. Enfin, on n'essaie pas de prendre le maximum sur chacun des chemins du graphe de contrôle ou d'exploiter les EQUIVALENCES.

En conséquence, l'évaluation du volume se fait en ajoutant les volumes partiels relatifs à chaque famille de références. Certaines familles sont réduites à un élément, mais dans tous les cas les familles sont représentées par leurs vecteurs d'autodépendance.

Le calcul du volume n'est pas facile, surtout si on veut obtenir une expression symbolique utilisable pour l'optimisation en forme (P) et en taille (λ). Dans ce paragraphe § 4.2, le partitionnement est effectué avec la matrice $\lambda^N P$ et $det(P)=1$.

Les dimensions du problème interviennent aussi. C'est pourquoi, après avoir présenté les différentes solutions envisagées, nous précisons globalement les contraintes d'applications qu'elles supposent et les résultats qu'elles apportent dans une dernière partie.

4.2.1. Calcul direct du volume relatif à une référence

Si le système formé par les formes linéaires d'accès $\vec{\alpha}_i$ est libre et de dimension N, ce qui signifie aussi que la référence considérée ne présente pas d'autodépendance ou encore que les données ne sont jamais réutilisées, chaque itération de la référence va accéder un élément différent. Le nombre d'éléments référencés est donc le nombre d'itérations, i.e. le volume du supernoeud qui vaut λ^N par définition si les boucles originelles sont normalisées. Pour répondre à ce critère, le tableau référencé doit être de dimension au moins égale à N. Aucune optimisation n'est possible avec cache ou mémoire locale à moins que les transferts puissent être effectués simultanément avec les calculs ou qu'il existe une famille comptant plusieurs références au même tableau.

Dans le cas contraire, quand le système formé par les formes linéaires d'accès est de rang inférieur à N, on forme un système d'égalités et d'inégalités linéaires à partir de la définition d'un supernoeud

formé par une matrice de partitionnement $\lambda^N P$:

$$\begin{cases} \vec{j} = \sum_i \mu_i \ \vec{p}_i \\ 0 \leq \mu_i < \lambda \end{cases}$$

et des égalités définissant les indices de la référence à des constantes de translation près (le volume est indépendant des translations):

$$\phi_k = < \vec{\alpha}_k \ \vec{j} > = \sum_i \mu_i \vec{\alpha}_k \vec{p}_i$$

Il reste alors à calculer le volume du polyèdre Φ défini par les ϕ_k. Ce polyèdre est caractérisé par un système d'inégalités obtenu par élimination des variables auxiliaires μ_i au moyen d'une projection. L'intégration est valable car les $\vec{\alpha}_k$ sont libres et la densité est bien de 1 car les $\vec{\alpha}_k$ ont été réduits à des coefficients premiers entre eux par division par leurs PGCDs.

$$V = \int_\Phi d\,\phi_1 d\,\phi_2 \cdots d\,\phi_k$$

Afin de montrer qu'il n'y a pas de solution simple, un exemple de *volume* V (de dimension 2!) est représenté sur la figure 4.2.1.a pour la référence

A(2*I+3*J–4*K,–I+4*J+5*K)

et pour le partitionnement défini par

$$\lambda P = \begin{pmatrix} 1 & -2 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & -2 \end{pmatrix} \qquad \lambda = 3$$

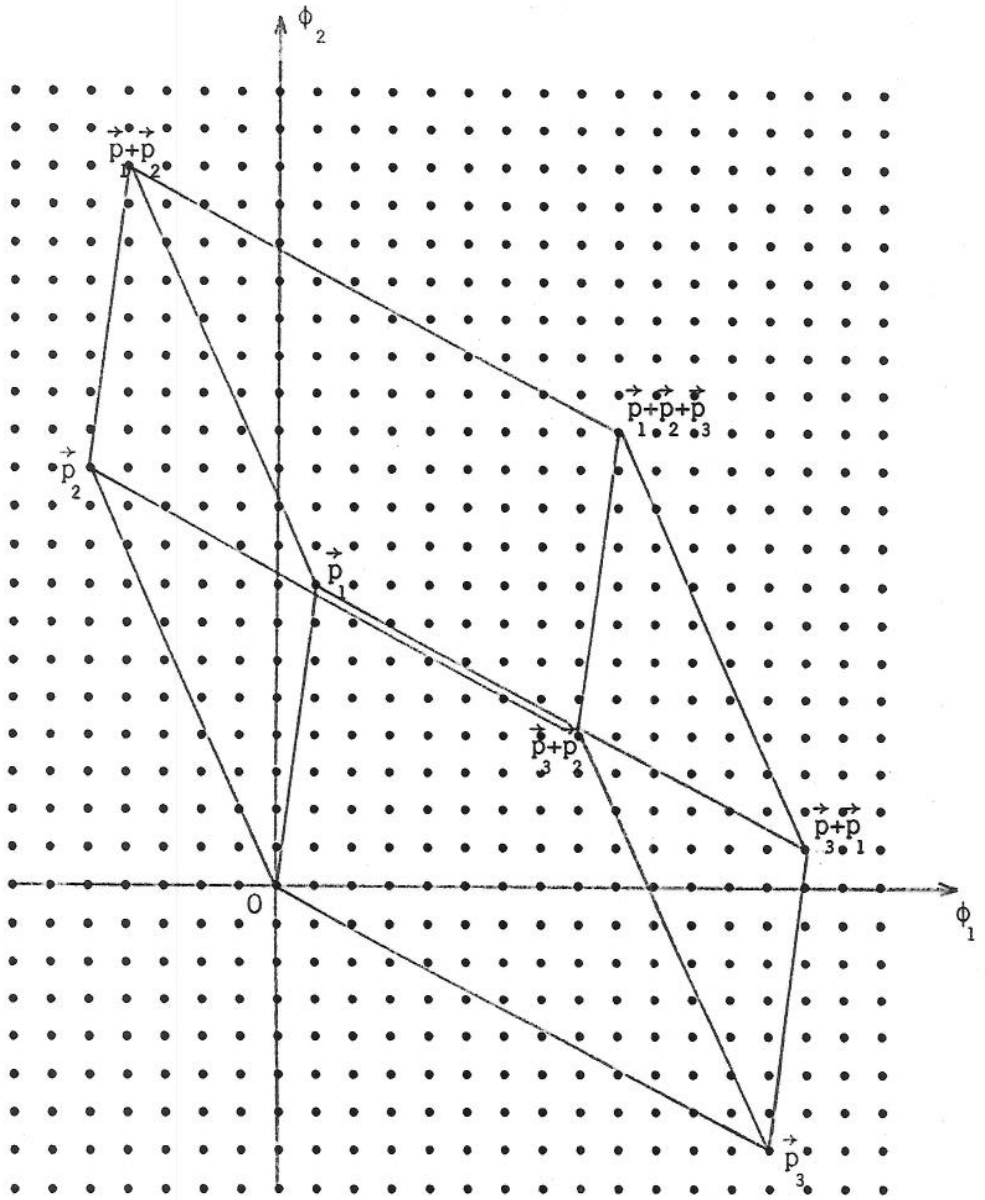
Si le partitionnement P et donc les $\vec{p}_i$ étaient supposés numériquement connus, on pourrait appliquer une procédure générale d'intégration, procédant par décomposition du polyèdre en systèmes triangulaires disjoints que l'on sait intégrer. La décomposition pourrait s'effectuer avec la procédure décrite par Kuhn[11] (cf. § 3.3.1.4) s'il est possible de la transposer en calcul symbolique. Sinon, il ne sera pas possible d'optimiser P et λ.

Cas particulier des références monodimensionnelles

Le problème de l'intégration disparaît en dimension 1 et l'on a:

$$V = \phi_{\max} - \phi_{\min} = \lambda \sum_i \ | \vec{\alpha} \ \vec{p}_i \ |$$

Un exemple est donné sur la figure 4.2.1.b pour la référence A(I) et pour le partitionnement défini par



Eléments accédés de A

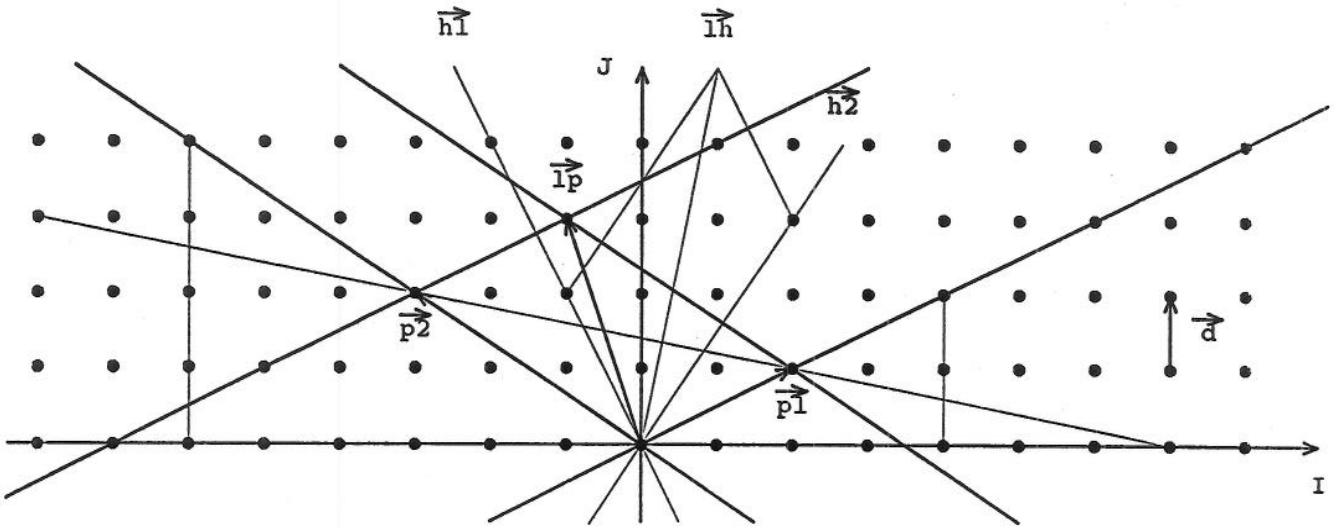
- Figure 4.2.1.a -

$$\lambda P = \begin{pmatrix} 2 & -3 \\ 1 & 2 \end{pmatrix} \quad P = \begin{pmatrix} 2/\sqrt{7} & -3/\sqrt{7} \\ 1\sqrt{7} & 2/\sqrt{7} \end{pmatrix} \quad \lambda = \sqrt{7} \quad \vec{\alpha} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Le volume référencé, calculé avec l'approximation continue, est donc:

$$V = \sqrt{7}\,(\,2/\sqrt{7} + 3/\sqrt{7}\,) = 5$$

Cette valeur diffère de celle, 4, qu'on voit sur la figure 4.2.1.b mais le lecteur peut s'assurer que le coefficient de $5/\sqrt{7}$ est bon en doublant la valeur de λ et en vérifiant sur la figure que le nouveau volume est de 9.



Volume référencé par A(I)

- Figure 4.2.1.b -

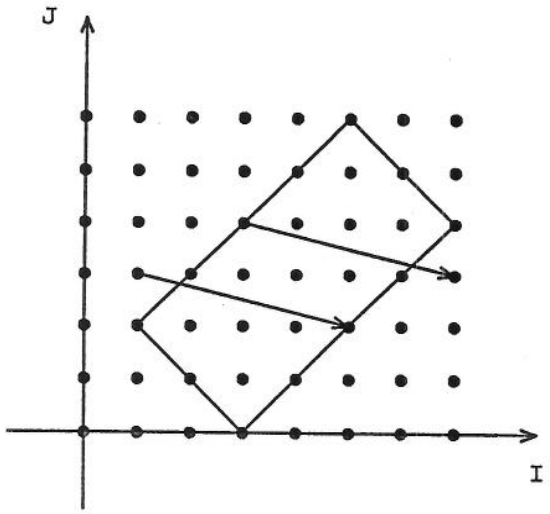
4.2.2. Calcul dual du volume relatif à une référence

On se limite au cas où la référence est de dimension $N-1$ et où elle ne génère donc qu'un seul vecteur d'autodépendance noté $\vec{d}$. Il s'agit donc de projeter un supernoeud sur un hyperplan.

Chaque noeud appartenant à une droite de direction $\vec{d}$ réfère par définition de $\vec{d}$ le même élément de tableau. Dans cette section 4.2.2, nous allons prendre $\vec{d}$ tel que $\| \vec{d} \| = 1$, quoique l'autodépendance définisse des vecteurs à coefficients entiers. Le nombre d'éléments différents référencés par le supernoeud va donc être égal au flux passant à travers ses faces d'entrées ou ses faces de sorties (à un signe près), moyennant une approximation continue au voisinage des sommets et des arêtes du supernoeud. En effet, certaines droites qui traversent le supernoeud à sa périphérie peuvent avoir une intersection **entière** vide avec lui, comme c'est le cas sur la figure 4.2.2.a.

On y voit deux matérialisations du vecteur d'autodépendance généré par la référence A(I+4*J) qui traversent un supernoeud défini par

$$\lambda P = \begin{pmatrix} 4 & -2 \\ 4 & 2 \end{pmatrix} \qquad \vec{d} = \begin{pmatrix} 4 \\ -1 \end{pmatrix}$$



Influence des grands vecteurs $\vec{d}$

- Figure 4.2.2.a -

sans permettre la moindre réutilisation de données.

L'approximation que nous faisons conduit à ne calculer que le terme de degré $N-1$ du polynôme en λ donnant le volume et n'est valable que pour des valeurs suffisamment grandes de λ. Heureusement, les références usuelles génèrent des vecteurs d'autodépendance dont la norme vaut 1 ou, au pire, 2. Les références du type de celle qui a été donnée pour la figure 4.2.1.a:

A(2*I+3*J-4*K,-I+4*J+5*K)

et qui génère un vecteur d'autodépendance de norme 33.436 ne sont pas courantes!

$$\vec{d} = \begin{pmatrix} 31 \\ -6 \\ 11 \end{pmatrix} \qquad \| \vec{d} \|^2 = 1118$$

Nous allons maintenant effectuer les calculs pour un supernoeud élémentaire en prenant $\lambda = 1$.

Les faces du supernoeud sont repérées par les vecteurs $\vec{h_i}$ de la base H, duale de P. Considérons la face i et le vecteur $\vec{h} = \nu \vec{h_i}$ tel que $\|\vec{h}\| = 1$. Le flux par unité de surface est

$$\vec{h}\ \vec{d}$$

La surface de la face i est égale au volume obtenu en la développant selon sa normale sur une longueur de 1. Ce volume est celui de la maille unité de la base P_i qui est déduite de la base P en remplaçant $\vec{p_i}$ par $\vec{h}$. Cette transformation donne une base car $\vec{h_i}$ est libre par rapport aux $\vec{p_j}$ pour

$j \neq i$:

$$\vec{h_i}\ \vec{p_j} = \delta_{ij}$$

où δ_{ij} est l'indice de Kronecker.

L'égalité surface volume est due au fait que la hauteur du volume défini par P_i est de 1.

Mais le volume généré par la base P peut aussi être calculé à partir de la surface définie par les vecteurs $\vec{p_j}$ de P_i et de la hauteur correspondante:

$$det\,(P) = det\,(P_i) \times \vec{p_i}\,\vec{h} = 1$$

Le flux à travers la face i est donc (à un signe près):

$$\phi_i = det\,(P)\,\frac{\vec{h}\ \vec{d}}{\vec{p_i}\,\vec{h}} = \frac{\vec{h}\ \vec{d}}{\vec{p_i}\,\vec{h}}$$

Comme $\vec{h} = \nu\,\vec{h_i}$ et que $\vec{h_i}\ \vec{p_i} = 1$ par définition de P et de H, le flux à travers la face i est:

$$\phi_i = \vec{h_i}\,\vec{d}$$

Le flux total à travers un supernoeud, pour un λ quelconque, est donc:

$$\phi = \lambda^{N-1} \sum_{i=1}^{N} \mid \vec{h_i}\,\vec{d} \mid = V$$

Ce résultat va être utilisé pour calculer à nouveau le volume référencé dans l'exemple 4.2.1.b. qui est caractérisé par:

$$H = (P^{-1})^t = \begin{pmatrix} 2/\sqrt{7} & -1/\sqrt{7} \\ 3/\sqrt{7} & 2\sqrt{7} \end{pmatrix} \quad \vec{1}_H = \begin{pmatrix} 1/\sqrt{7} \\ 5/\sqrt{7} \end{pmatrix} \quad \vec{d} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad N = 2 \quad \lambda = \sqrt{7}$$

On en déduit:

$$V = \sqrt{7}\,(\,\frac{3}{\sqrt{7}} + \frac{2}{\sqrt{7}}\,) = 5$$

ce qui donne bien le même résultat qu'avec la méthode 4.2.1.

Il serait tentant d'appliquer la méthode duale au cas 4.2.1.a qui est de dimension 3, mais le vecteur d'autodépendance est si long que le supernoeud choisi est systématiquement traversé. Le volume du supernoeud est donné par $det\,(P)$ et il vaut 9. Le volume référencé réel vaut aussi 9 et il est bien plus fort que le volume qu'on pourrait calculer:

$$H^{-1} = \begin{pmatrix} 1/3 & -1/3 & 0 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 0 & -1/3 \end{pmatrix} \quad \vec{1}_H = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \lambda = \sqrt{3} \quad \vec{d} = \frac{1}{\sqrt{1118}} \begin{pmatrix} 31 \\ -6 \\ 11 \end{pmatrix}$$

ce qui donnerait:

$$V = \sqrt{3}^2 \left(\frac{36}{3\sqrt{1118}} + \frac{25}{3\sqrt{1118}} + \frac{17}{3\sqrt{1118}}\right) = 3\frac{26}{\sqrt{1118}} = 2.332$$

Le coefficient de $26/\sqrt{1118}$ est difficile à vérifier manuellement mais il est clairement faux pour de petites valeurs de λ comme $\sqrt{3}$.

Dans le cas où l'autodépendance $\vec{d}$ appartient au cône de dépendance ($\forall\, i \;\; \vec{h_i}\ \vec{d} > 0$ par définition de H), les valeurs absolues sont inutiles:

$$V = \phi = \lambda^{N-1}\, \vec{1}_H\, \vec{d}$$

et pour plusieurs familles de références dont les autodépendances vérifient la condition précédente, on peut profiter de l'additivité des volumes

$$V = \lambda^{N-1}\, \vec{1}_H\, \vec{d}_g$$

où $\vec{d}_g$ est la somme de toutes les autodépendances considérées. Géométriquement, la minimisation de V revient à itérer orthogonalement aux diffusions de données, c'est-à-dire à utiliser complètement les données disponibles avant de passer aux suivantes.

Comme on n'obtient que le terme de plus haut degré à cause de l'approximation continue, le résultat reste vrai pour les **familles** de références.

Enfin, l'obtention d'un résultat analytique fonction de la forme du supernoeud par le biais de $\vec{1}_H$ permet d'envisager une optimisation de forme dans le cas où toutes les autodépendances tombent dans le cône de dépendance. Cette optimisation est non négligeable bien qu'il semble à première vue qu'elle ne concerne que le terme de degré $N-1$[†] alors que les calculs, et, éventuellement, d'autres références génèrent des termes de degré N. Mais il faut se méfier de la valeur du coefficient α_{N-1}.

Considérons les deux supernoeuds s_1 et s_2 de la figure 4.2.2.b qui ont tous les deux une épaisseur de 1 ainsi que les deux familles de supernoeuds qui ont des allures similaires. Ces familles sont définies par les matrices suivantes:

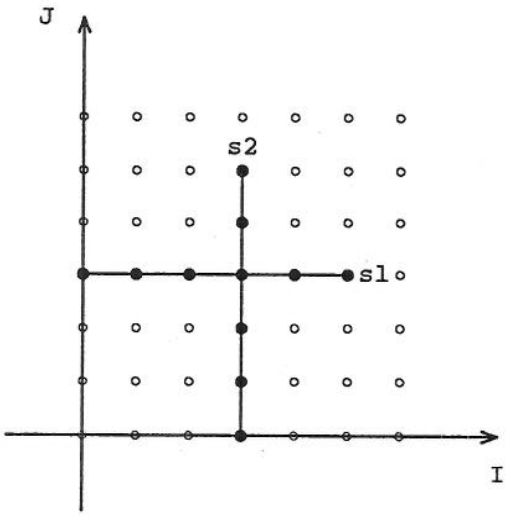
$$P_1 = \begin{pmatrix} \sqrt{K} & 0 \\ 0 & 1/\sqrt{K} \end{pmatrix} \quad H1 = \begin{pmatrix} 1/\sqrt{K} & 0 \\ 0 & \sqrt{K} \end{pmatrix} \quad P_2 = \begin{pmatrix} 1/\sqrt{K} & 0 \\ 0 & \sqrt{K} \end{pmatrix} \quad H2 = \begin{pmatrix} \sqrt{K} & 0 \\ 0 & 1/\sqrt{K} \end{pmatrix}$$

Le coefficient λ vaut $\sqrt{K}$. K est la longueur du supernoeud et vaut 6 sur la figure 4.2.2.b. Pour une référence A(I), on obtient:

$$V_1 = \lambda\, \vec{1}_{H_1}\, \vec{d} = \sqrt{K} \times \sqrt{K} = \lambda^2$$
$$V_2 = \lambda\, \vec{1}_{H_2}\, \vec{d} = 0$$

[†]C'est la constitution des supernoeuds et l'hypothèse de lecture unique de chaque donnée qui permet déjà de faire passer le coût relatif à ce type de référence de λ^N à λ^{N-1}.

Dans le cas 1, le nombre d'éléments référencés reste bien proportionnel au volume, comme on s'y attend intuitivement, car la suite de supernoeuds considérée ne correspond pas à une simple augmentation de λ. Dans le deuxième cas, le nombre d'éléments référencés est constant. La constante 1 n'est pas trouvée puisqu'on ne calcule que le coefficient du terme en λ.



Variations du coefficient $\vec{1}_H \ \vec{d}$

- Figure 4.2.2.b -

Ces supernoeuds très allongés sont typiques des partitionnements qu'il faut effectuer pour obtenir une exécution vectorielle sur une machine à registres. On peut en effet assimiler le supernoeud au registre vectoriel. Le résultat précédent montre qu'il ne pourra pas y avoir réutilisation de données selon les autodépendances.

4.2.3. Calcul exact du volume d'une copie par excès avec accès linéaire

Quand les transferts sont programmés, il peut arriver qu'on soit obligé de transférer des données inutiles (cf. § 3.5.3 et § 3.6.3). Il suffit alors de reprendre la procédure de calcul de volume mais au lieu d'essayer d'intégrer le système définissant les ϕ_k, on le projette sur chaque ϕ_k pour obtenir un Min et un Max sur cette dimension k de la copie du tableau référencé.

La projection d'un système peut être effectuée par une procédure d'élimination de variables bien connue[8] mais coûteuse. Dans le cas d'un système aussi simple que celui qui exprime l'appartenance à un supernoeud et en considérant les ϕ_k comme des fonctions de coûts linéaires et donc convexes, il suffit d'évaluer les ϕ_k en chaque sommet du supernoeud et de sélectionner les *Min* et les *Max* puis de faire la différence.

Pour chaque ϕ_k , l'accroissement est linéaire et le coefficient est:

$$\sum_i \quad | \vec{\alpha}_k \vec{p}_i |$$

comme on peut le voir en raisonnant par récurrence à partir des noeuds correspondant au *Min* et au *Max* pour λ et en évaluant les accroissements pour $\lambda+1$. Le coefficient de plus haut degré pour le volume est donc:

$$\prod_k \sum_i \quad | \vec{\alpha}_k \ \vec{p}_i |$$

Dans le cas où la référence est de dimension 1 et où'il n'y a donc qu'un $\vec{\alpha}_k$, l'approximation du volume (et non seulement du volume approché) est très bonne, puisqu'on ne perd que le terme de degré 0 qui correspond aux variations de densité dues aux bords du supernoeud. Ce terme dépend de la norme de l'autodépendance et il est faible pour les autodépendances usuelles car leurs normes sont petites. Ce terme dépend aussi de la forme du noeud, il est d'autant plus faible que le noeud s'approche de la sphère, i.e. que le rapport surface sur volume est plus faible.

Toujours dans le cas de dimension 1, il est possible d'optimiser la forme pour une référence (plus généralement pour une autodépendance) en choissant des $\vec{p}_i$ aussi orthogonaux que possible à $\vec{\alpha}$.

4.2.4. Calcul direct du volume d'une copie d'une famille de références

Pour une famille de références, il suffit de choisir une référence particulière et de calculer les autodépendances par rapport à elle puis de compliquer le balayage des sommets prévu dans la procédure pour région élémentaire, d'un balayage des vecteurs de dépendances pour chaque sommet.

Mais comme les dépendances sont constantes, le terme de plus haut degré du volume n'est pas modifié et reste:

$$\prod_j \sum_i \quad | \vec{\alpha}_j \ \vec{p}_i |$$

Les autres coefficients nécessaires peuvent être dérivés à partir de plusieurs points d'évaluation en λ_o , λ_{o+1}, etc...

4.2.5. Calcul de volume par simulation

La simulation de l'exécution d'un supernoeud permet d'obtenir des résultats très précis à un coût élevé quand aucune autre méthode plus performante n'est connue. La forme du supernoeud doit être connue et on ne peut pas obtenir de résultats symboliques en P . Elle ne permet donc pas d'optimiser la forme des supernoeuds mais seulement le facteur d'échelle λ.

L'expression symbolique en λ est dérivée de $K+1$ essais car on connaît a priori le degré K du polynôme résultat. Le choix des $\lambda_o, \lambda_o+1, \ldots \lambda_o+K$ correspondant à chaque point de mesure est guidé par le plus grand vecteur d'autodépendance.

En effet, tant que les supernoeuds sont petits par rapport aux vecteurs d'autodépendance, aucune autodépendance n'est matérialisée à l'intérieur d'un supernoeud (cf. figure 4.2.2.a) et les volumes croissent initialement en λ^N où N est la dimension de l'espace d'itération. Apparemment, il suffit qu'il existe un vecteur de dépendance appartenant au supernoeud pour que sa taille puisse être prise comme λ_o. Avec les autodépendances classiques, un λ_o de 2 à 4 suffit, ce qui permet d'envisager l'utilisation d'une méthode aussi coûteuse. Attention cependant aux références linéaires du type 1000*I+J, qui peuvent être produites par une optimisation.

Le résultat n'est valide que pour $\lambda \geq \lambda_o$. Une telle contrainte est valable pour les autres méthodes car elles font l'hypothèse d'une densité constante et négligent les phénomèmes d'arêtes et de sommets. De toute façon, il n'existe pas de solution polynomiale sur N en général.

4.2.6. Récapitulatif des résultats obtenus.

L'ensemble des problèmes, des situations et des solutions proposées et leurs relations sont rappelés dans la figure 4.2.6 où quelques blancs et points d'interrogation subsistent encore[†]. Mais ils disparaissent déjà si on limite la dimension du problème à 3. Il n'y a donc pas de problème en dimension 2.

dimension de l'espace d'itération	dimension des autodépendances	dimension réelle du tableau	nature de la région	calcul du volume asymptotique optimisation P	calcul du volume asymptotique en $f(\lambda)$	calcul exact du volume d'une copie par excès en $f(\lambda)$
N	0	N	simple	X	X	projection
			union	?	simu	projection
N	1	N - 1	simple	dual	dual	projection
			union	dual	dual	projection
N	P	N - P	simple	?	simu	projection
			union	?	simu	projection
N	N - 1	1	simple	direct	direct	projection
			union	direct	direct	projection
N	N	0	simple	X	1	1
			union	X	X	X

Synthèse des résultats obtenus pour les calculs de volumes

- Figure 4.2.6 -

[†]Les cases contenant un X correspondent à des problèmes sans objet.

4.3. Volume de stockage

Le détermination du volume de stockage s'effectue à P fixé et permet de déterminer le facteur d'échelle λ en fonction de la taille de la mémoire rapide. Ce calcul peut être évité quand l'architecture de la machine conduit à exécuter tous les transferts avant le début des calculs ou à la fin des calculs des supernoeuds car le volume de stockage est alors identique au volume transféré.

Mais quand la machine dispose d'un cache ou d'une mémoire locale qu'elle peut charger ou décharger à volonté, le volume de stockage peut être inférieur au volume transféré. Le prendre égal à ce dernier est une solution pessimiste de repli qui permet d'éviter un problème supplémentaire et qui nous semble convenir dans le cas "mémoire locale" pour lequel nous avons prévu au chapitre 3 de générer globalement les transferts.

Néanmoins, les architectures à base de cache et l'éventuelle application manuelle de notre méthode de partitionnement nous poussent à aborder ce nouveau problème que nous avons informellement traité dans le cas de la multiplication de matrice aux paragraphes 4.1.1, 4.1.2 et 4.1.3.

4.3.1. Cas de la dimension 2

Comme nous l'avons vu en 4.2, le cas des espaces de dimension 2 est beaucoup plus facile à traiter que le cas général car il n'y a qu'un type de tableau intéressant, le tableau monodimensionnel. Les scalaires (dimension 0) prennent une case dans le cache. Les tableaux bidimensionnels ne devraient en prendre aucune puisqu'aucune valeur n'est réutilisée.

Les accès à un tableau monodimensionnel sont caractérisés par une seule forme linéaire $\vec{\alpha}$ qui ne génère qu'une seule autodépendance $\vec{d}$, qui est orthogonale à $\vec{\alpha}$. Le volume de cache nécessaire dépend directement de l'ordonnancement des calculs: si toutes les utilisations d'un élément du tableau ont lieu consécutivement, il suffit d'une position dans le cache. L'ordonnancement est défini par un vecteur $\vec{h}$ et le cas limite que nous venons d'évoquer est défini par:

$$\vec{h} = \mu \, \vec{\alpha}$$

Si deux noeuds de vecteurs d'itération $\vec{j}_1$ et $\vec{j}_2$ accèdent le même élément, alors:

$$\vec{\alpha} \, \vec{j}_1 = \vec{\alpha} \, \vec{j}_2$$

On en déduit que $\vec{h} \, \vec{j}_1 = \vec{h} \, \vec{j}_2$ et donc que les deux noeuds sont exécutés en parallèle.

L'autre cas extrême revient à prendre $\vec{h} = \mu \, \vec{d}$ et donc à ne jamais exploiter d'autodépendance au sein d'une itération. En effet, si le même élément est accédé en $\vec{j}_1$ et en $\vec{j}_2$:

$$\vec{j}_1 = \vec{j}_2 + \nu \, \vec{d}$$

et donc:

$$\vec{h}\,\vec{j}_1 = \vec{h}\,\vec{j}_2 + \mu\nu \,\|\vec{h}\|^2$$

Comme μ et ν sont non nuls, ainsi que $\vec{h}$, n_1 et n_2 ne peuvent pas s'exécuter simultanément. Donc deux noeuds qui s'exécutent simultanément ne peuvent accéder que des éléments différents. Par contre, deux itérations successives utilisent les mêmes éléments, à l'effet des frontières de supernoeud près. Si de plus $\vec{h} = \vec{I}_H$, l'itération qui aura besoin du plus grand nombre d'éléments va accéder l'ensemble du volume référencé, qui a été calculé en 4.2:

$$V = \lambda\,\vec{h}\,\vec{d}$$

Il reste à traiter le cas général. En appelant $\vec{w}$ un vecteur orthogonal à $\vec{h}$ qui contient l'ensemble des noeuds relatifs à une itération, le volume de stockage s'exprime sous la forme:

$$V = \vec{w}\,\vec{\alpha}$$

Les valeurs extrêmes possibles de $\vec{w}$ sont de la forme $\lambda(\vec{p}_1 + \nu\,\vec{p}_2)$ ou de la forme $\lambda(\vec{p}_2 + \mu\,\vec{p}_1)$ où $\vec{p}_1$ et $\vec{p}_2$ sont les deux vecteurs de partitionnement et où ν et μ appartiennent à l'intervalle $[-1,1]$ comme on peut le voir sur la figure 4.3.1.a. En effet, il existe toujours une itération de taille maximale passant par un des quatre sommets du supernoeud.

Il faut d'abord chercher, parmi les $\vec{w}$ possibles, un vecteur $\vec{w}$ orthogonal à la direction d'itération locale $\vec{h}$, qui aura peut-être été fixée de manière à avoir des instructions vectorielles. Il y aura soit une solution en μ soit une solution en ν vérifiant l'appartenance à l'intervalle $[-1,1]$ puisque P est une base et que $\mu = \nu^{-1}$.

En prenant $\lambda = 1$, $c = \vec{w}\,\vec{\alpha}$ est le coefficient qui donne le volume en fonction de λ.

Prenons les exemples de la figure 4.3.1.b où le partitionnement est défini par:

$$\vec{\alpha} = \begin{pmatrix}1\\0\end{pmatrix} \quad \vec{d} = \begin{pmatrix}0\\1\end{pmatrix} \quad P = \begin{pmatrix}\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{pmatrix} \quad H = \begin{pmatrix}\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{pmatrix}$$

et les ordonnancements par les cinq vecteurs:

$$\vec{h}_1 = \begin{pmatrix}1\\0\end{pmatrix} \quad \vec{h}_2 = \begin{pmatrix}\frac{2}{\sqrt{5}}\\ \frac{1}{\sqrt{5}}\end{pmatrix} \quad \vec{h}_3 = \begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix} \quad \vec{h}_4 = \begin{pmatrix}\frac{1}{\sqrt{5}}\\ \frac{2}{\sqrt{5}}\end{pmatrix} \quad \vec{h}_5 = \begin{pmatrix}0\\1\end{pmatrix}$$

En prenant la première forme de $\vec{w}$, on obtient:

$$\vec{w} = \vec{p}_1 + \nu\,\vec{p}_2 = \frac{\sqrt{2}}{2}\begin{pmatrix}\nu+1\\ \nu-1\end{pmatrix}$$

$$\vec{w}\,\vec{h}_1 = \nu + 1 = 0 \;\Rightarrow\; \vec{w}_1 = \begin{pmatrix} 0 \\ \sqrt{2} \end{pmatrix} \;\Rightarrow\; c_1 = \vec{w}_1\,\vec{\alpha} = 0$$

$$\vec{w}\,\vec{h}_2 = 3\nu + 1 = 0 \;\Rightarrow\; \vec{w}_2 = \frac{\sqrt{2}}{3}\begin{pmatrix} 1 \\ -2 \end{pmatrix} \;\Rightarrow\; c_2 = \vec{w}_2\,\vec{\alpha} = \frac{\sqrt{2}}{3}$$

$$\vec{w}\,\vec{h}_3 = 2\nu = 0 \;\Rightarrow\; \vec{w}_3 = \frac{\sqrt{2}}{2}\begin{pmatrix} -1 \\ 1 \end{pmatrix} \;\Rightarrow\; c_3 = |\,\vec{w}_3\,\vec{\alpha}\,| = \frac{\sqrt{2}}{2}$$

$$\vec{w}\,\vec{h}_4 = 3\nu - 1 = 0 \;\Rightarrow\; \vec{w}_4 = \frac{\sqrt{2}}{3}\begin{pmatrix} 2 \\ -1 \end{pmatrix} \;\Rightarrow\; c_4 = \vec{w}_4\,\vec{\alpha} = \frac{2\sqrt{2}}{3}$$

$$\vec{w}\,\vec{h}_5 = \nu - 1 = 0 \;\Rightarrow\; \vec{w}_5 = \sqrt{2}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \;\Rightarrow\; c_5 = \vec{w}_5\,\vec{\alpha} = \sqrt{2}$$

Les cinq ordonnancements différents sont représentés en figure 4.3.1.b pour un supernoeud de facteur d'échelle $\lambda = 4\sqrt{2}$. Les résultats discrets obtenus pour plusieurs valeurs de λ (colonnes de gauche) ainsi que les volumes qu'on peut dériver des ordonnancements en faisant une approximation continue (colonnes de droite) sont donnés dans la table suivante où V_s signifie volume de stockage et V_t, volume transféré. Les coefficients sont rappelés à la dernière ligne.

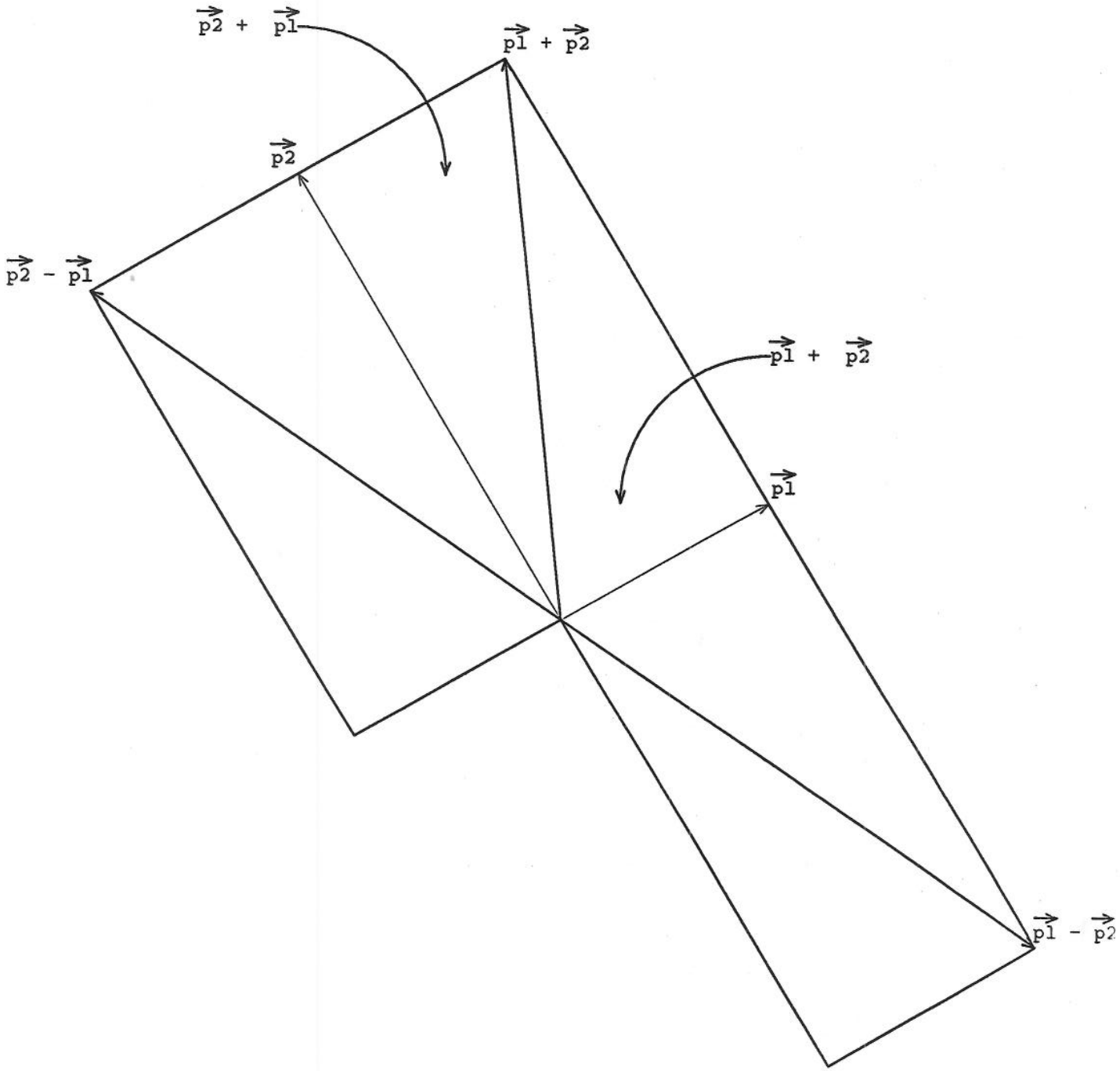
λ	V_s										V_t
	$\vec{h}_1$		$\vec{h}_2$		$\vec{h}_3$		$\vec{h}_4$		$\vec{h}_5$		
c_i	0		$\frac{\sqrt{2}}{3}$		$\frac{\sqrt{2}}{2}$		$\frac{2\sqrt{2}}{3}$		$\sqrt{2}$		$\sqrt{2}$
$\sqrt{2}$	1	0	1	2/3	1	1	1	4/3	1	2	1
$2\sqrt{2}$	1	0	2	4/3	2	2	3	8/3	3	4	3
$3\sqrt{2}$	1	0	2	3	3	3	3	4	5	6	5
$4\sqrt{2}$	1	0	3	8/3	4	4	5	16/3	7	8	7
$5\sqrt{2}$	1	0	4	10/3	5	5	7	20/3	9	10	9
$6\sqrt{2}$	1	0	4	4	6	6	7	8	11	12	11
$7\sqrt{2}$	1	0	5	14/3	7	7	9	28/3	13	14	13

Les volumes qu'on peut prédire en effectuant l'approximation continue et en utilisant les coefficients c_i que nous avons calculés auparavant ne s'écartent jamais des valeurs discrètes de plus de 1.

Nous allons effectuer ces calculs sur l'exemple de la figure 4.3.1.c pour une référence et un partitionnement moins réguliers. Soit la référence:

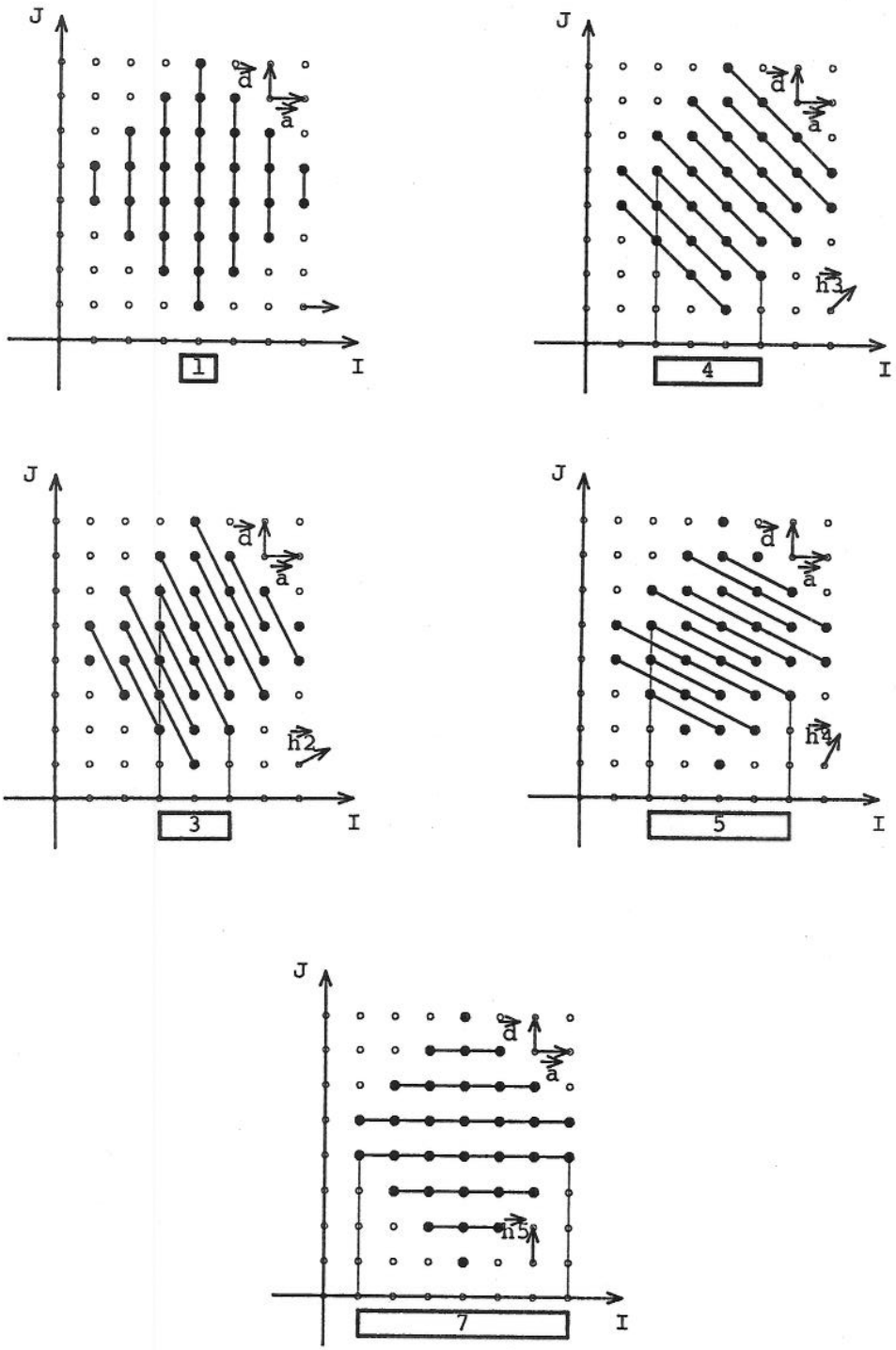
T(2*I+J)

et le partitionnement:



Recherche des fronts extrêmes

- Figure 4.3.1.a -



Variations du volume du cache en fonction de l'ordonnancement

- Figure 4.3.1.b -

$$\vec{\alpha} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \quad \vec{d} = \begin{pmatrix} 1 \\ -2 \end{pmatrix} \quad P = \frac{1}{2\sqrt{6}} \begin{pmatrix} 4 & 3 \\ -4 & 3 \end{pmatrix} \quad H = \frac{1}{\sqrt{6}} \begin{pmatrix} 3 & 4 \\ -3 & 4 \end{pmatrix} \quad \lambda = 2\sqrt{6}$$

et l'ordonnancement par le vecteur:

$$\vec{h} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

On recherche un $\vec{w}$ orthogonal à $\vec{h}$:

$$\vec{w} = \vec{p}_1 + \nu\vec{p}_2 = \frac{1}{2\sqrt{6}} \begin{pmatrix} 3\nu + 4 \\ 3\nu - 4 \end{pmatrix} \quad \vec{h}\,\vec{w} => 3\nu + 12 = 0 => \nu = -4$$

Comme ν n'est pas valide, il faut prendre la formule à base de μ avec $\mu = 1/\nu = -1/4$:

$$\vec{w} = \mu\vec{p}_1 + \vec{p}_2 = \frac{1}{2\sqrt{6}} \begin{pmatrix} 4\mu + 3 \\ -4\mu + 3 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \end{pmatrix}$$

Cela donne:

$$V = \lambda\vec{w}\,\vec{\alpha} = 2\frac{\sqrt{2}}{\sqrt{3}}\,\lambda$$

Pour $\lambda = 2\sqrt{6}$, $V = 8$ ce qui correspond au résultat qu'on obtient graphiquement sur la figure 4.3.1.c.

Optimisation de la direction d'itération locale

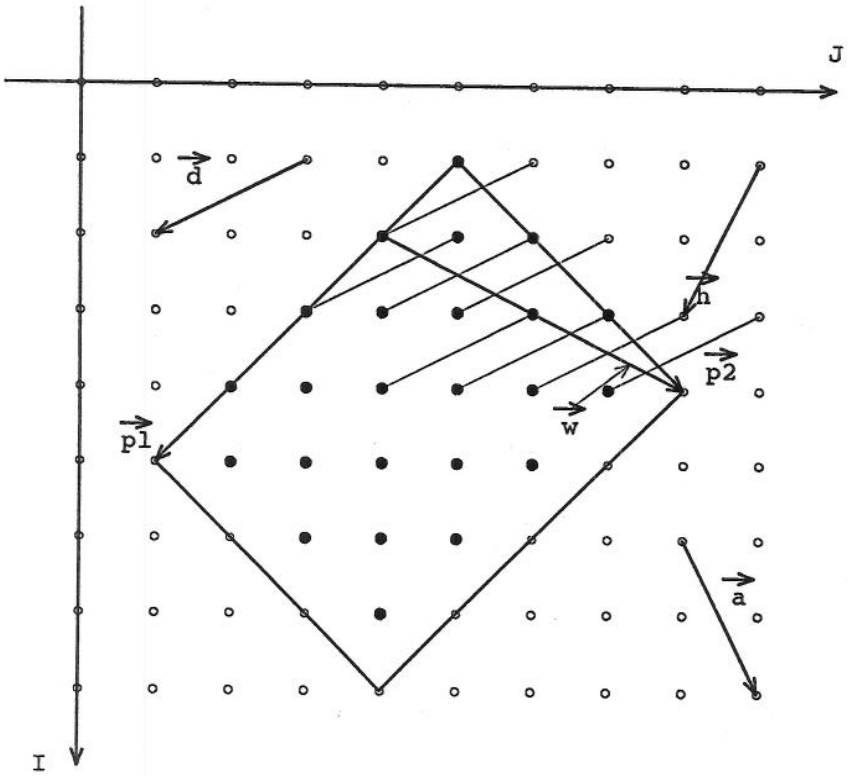
La recherche de la meilleure direction d'itération dans un supernoeud va donc consister à minimiser V_1 et V_2 en respectant les contraintes sur μ et ν:

$$V_1 = \vec{p}_1\,\vec{\alpha} + \nu\,\vec{p}_2\,\vec{\alpha}$$

$$V_2 = \mu\,\vec{p}_1\,\vec{\alpha} + \vec{p}_2\,\vec{\alpha}$$

Les valeurs de ν ou de μ permettent de déduire $\vec{w}$ et donc la direction d'itération $\vec{h}$ optimale pour cette référence. Encore faut-il vérifier qu'elle satisfait les dépendances, à moins qu'on ne préfère ajouter une contrainte sur μ et ν, contrainte dérivée du fait que $\vec{h}$ doit être une combinaison linéaire positive des vecteurs de la base H.

Les formules données jusqu'ici ne concernent que le terme de plus haut degré du volume. Voyons ce que cela donne sur l'exemple suivant:



Calcul d'un volume de cache

- Figure 4.3.1.c -

```
DO I = ...
  DO J = ...
    ... A(I) ...
```

pour lequel on a choisi une matrice de partitionnement P qui vaut:

$$P = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}$$

La minimisation en μ et ν donne $\mu = \nu = 1$:

$$V_1 = \sqrt{2} - \nu\sqrt{2}$$

$$V_2 = \mu\sqrt{2} - \sqrt{2}$$

ce qui conduit bien à

$$\vec{w} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \vec{h} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

comme nous l'avons vu précédemment. L'optimisation est simple si $\vec{\alpha}$ ou $-\vec{\alpha}$ est une direction d'itération autorisée. Sinon, $\vec{h}_1$ ou $\vec{h}_2$ doit être solution. Ils donnent généralement des ordonnancements séquentiels car les $\vec{h}_i$ ne vérifient que $\vec{h}_i \, \vec{d}_k \geq 0$ et non $\vec{h}_i \, \vec{d}_k > 0$.

4.3.2. Cas des expressions d'index mono-indicées

La recherche du volume de cache nécessaire pour effectuer le produit de matrice est particulièrement simple parce que les expressions d'indices des tableaux ne contiennent qu'un indice de boucle.

Un compilateur optimiseur ferait remonter automatiquement les parties de calcul d'adresse qui sont constantes dans les boucles et puis déciderait d'utiliser un registre pour stocker A(I,J). Intuitivement l'idée consiste à regarder quelles expressions d'indices sont modifiées par la boucle englobante. Elle est développée au paragraphe 8.7.

4.3.3. Cas général: simulation

La méthode de simulation qui a été décrite en 4.2.5 peut être adaptée au cas du calcul du volume de stockage. Il suffit de stocker pour chaque élément le numéro de la première et de la dernière itération qui y accède puis de chercher l'itération pendant laquelle le plus grand nombre d'éléments sont vivants.

Après quelques essais sur de petites valeurs de λ, il sera possible de dériver le polynôme donnant le volume de stockage.

Une fois de plus ce type de méthode ne permet pas d'effectuer d'optimisation puisqu'il implique que la direction d'itération $\vec{h}$ soit prédéterminée. La simulation ne peut éventuellement servir qu'à la génération de code pour déterminer précisemment la taille d'un tableau de copie ou le volume de cache en fonction de λ.

4.4. Ordonnancement des transferts par rapport aux calculs

Si tous les transferts ont lieu au début et à la fin et si le volume stocké est égal au volume transféré, le problème de l'ordonnancement des transferts par rapport au calcul ne se pose pas.

Il ne se pose pas non plus quand les transferts sont exécutés automatiquement par le matériel comme c'est le cas avec les caches et la mémoire virtuelle.

Le problème ne subsiste donc que pour les architectures à mémoire locale et globale, où les processeurs élémentaires peuvent accéder à la mémoire globale et effectuer des transferts de petite taille sans avoir à payer un important overhead de lancement. Une des configurations de RP3[2] et le projet CEDAR[6] répondent à ces critères.

Nous n'avons pas trouvé de solution algorithmique permettant de générer le code automatiquement et nous nous sommes contenté de bloquer les transferts en début et en fin d'exécution pour générer le code partitionné au chapitre 3. Le problème reste ouvert.

Des expériences seraient nécessaires pour en mesurer l'intérêt: si l'exécution groupée des transferts conduit à une taille de tâche au voisinage de laquelle le temps d'exécution n'est que peu sensible à ce paramètre, le problème disparaît.

4.5. Exemple

Nous allons illustrer les résultats obtenus dans ce chapitre sur le petit exemple suivant:

```
DO I = 0, N-1
  DO J = 0, M-1
    A(I,J) = B(I) + C(J)
```

Code initial

Sans prendre de précautions particulières, l'exécution de ces boucles va nécessiter $3 \times N \times M$ accès à la mémoire. En sortant le terme constant B(I) de la boucle interne sur J, on obtient une nouvelle version:

```
DO I = 0, N-1
  B' = B(I)
  DO J = 0, M-1
    A(I,J) = B' + C(J)
```

Réduction du nombre des accès à la mémoire

qui ne nécessite plus que $2 \times N \times M + N$ accès. Si C(J) tient en mémoire locale, on tombe au volume transféré minimum, le volume référencé, mais on viole l'hypothèse selon laquelle les données ne tiennent pas en mémoire locale. Cette hypothèse est due au fait que la taille de la mémoire locale est fixée à la construction tandis que les dimensions des tableaux varient avec les problèmes.

En appliquant une technique de partitionnement avec des supernoeuds de côtés N1 et M1, parallèles aux axes I et J (cf. figure 4.5), on obtient:

$$\vec{d}_B = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \vec{d}_C = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad H = \frac{1}{\sqrt{M1 \times N1}} \begin{pmatrix} M1 & 0 \\ 0 & N1 \end{pmatrix} \quad \vec{I}_H = \begin{pmatrix} \sqrt{M1} \\ \sqrt{N1} \\ \sqrt{N1} \\ \sqrt{M1} \end{pmatrix} \quad \lambda = \sqrt{M1 \times N1}$$

Pour les trois références, on a donc un volume de:

$$V = \lambda^2 + \lambda \ |\vec{I}_H \vec{d}_B| \ + \lambda \ |\vec{I}_H \vec{d}_C|$$

$$V = M1\ N1 + N1 + M1$$

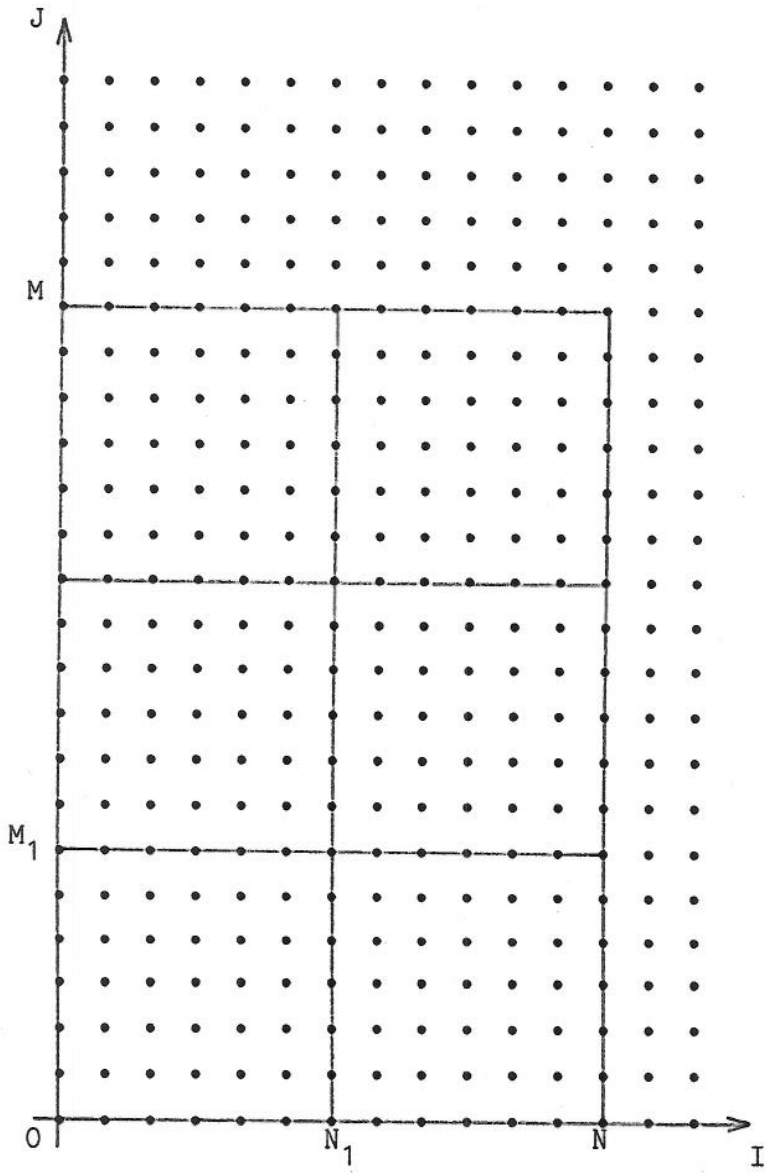
En minimisant d'abord les transferts à volume constant $K = M1\ N1$, on trouve:

$$M1 = N1 = \sqrt{K} \qquad V = M \times N + 2\frac{M \times N}{\sqrt{K}}$$

Il reste ensuite à calculer le volume de cache ou de mémoire locale nécessaire pour avoir des supernoeuds de volume K, puis à en déduire les valeurs maximales possibles pour K en fonction de la taille du cache ou de la mémoire locale.

Si on prenait en compte les possibilités d'*héritage* (cf. § 1.3.3) et si l'on réutilisait des données chargées pour l'exécution du supernoeud précédent lors de l'exécution du noeud suivant, on obtiendrait:

$$V = \frac{M\ N}{2\sqrt{K}} + N + M\ N \quad \text{ou} \quad V = \frac{M\ N}{2\sqrt{K}} + M + M\ N$$



Exemple de minimisation des transferts

- Figure 4.5 -

suivant que l'on découpe des bandes parallèles à J ou I.

4.6. Conclusion

Moyennnant une hypothèse simplificatrice dans l'évaluation du volume de stockage en mémoire locale et dans l'organisation des transferts, nous sommes capable de générer du code automatiquement (calcul de la taille des tableaux de copie, calcul du facteur d'échelle λ à partir de la taille de la mémoire locale).

La minimisation des transferts par un choix optimal de la matrice de partitionnement P est plus délicate. Tout d'abord l'obtention de formules symboliques pour le volume transféré n'est pas toujours possible. Ensuite, la multiplicité des références à l'intérieur d'un corps de boucle peut réduire ces efforts à néant. Il faudrait optimiser relativement à l'ensemble de ces références et non pour une référence particulière. Enfin, la matrice P est déjà conditionnée par les dépendances et il faudrait ajouter cette contrainte dans toute tentative d'optimisation.

De même, le choix d'un *bon* ordonnancement à l'intérieur d'un supernoeud ne doit pas être fait relativement à une seule référence comme nous l'avons fait en 4.3.1 mais à l'ensemble des références. Nos résultats ne valent donc que si le corps de boucle ne contient qu'une seule famille de références. Les ordonnancements sont conditionnés par les dépendances comme les partitionnements et toute optimisation devrait intégrer la contrainte qui en découle.

Finalement, le problème de l'ordonnancement des transferts par rapport aux calculs a été laissé de côté, faute de temps. Il n'est applicable qu'à un type d'architecture, encore peu répandu, et le gain qu'il pourrait apporter sur le temps d'exécution n'est pas encore estimable.

Avant de poursuivre avec l'évaluation du temps de calcul, nous allons récapituler brièvement les résultats obtenus dans ce chapitre.

4.6.1. Politique d'assignation

La politique choisie consiste à adapter la taille du supernoeud à la hiérarchie mémoire pour obtenir un nombre minimum de transferts et non à sélectionner précisément les tableaux qui doivent être copiés suivant le nombre d'accès qui sont effectués vers chaque élément et suivant le coût d'un transfert. On ne se pose pas la question: combien faut-il d'accès rapides pour rentabiliser un transfert? En effet les accès multiples dus à des familles de références non élémentaires comme:

$$A(I,J) = A(I,J+1) + A(I,J-1)$$

ne sont pas pris en compte car le gain maximum est 2. Les transferts sont effectués quand ils sont architecturalement nécessaires ou quand le gain potentiel est d'au moins une dimension et donc fonction non constante du facteur d'échelle λ. Suivant le problème et la taille de la mémoire rapide, des gains de 100, 1000 ou plus sont possibles.

Ce choix permet aussi de calculer très simplement le volume transféré.

Il faut cependant remarquer que les supernoeuds peuvent apporter quelque chose à la génération de code vectoriel. La minimisation des accès relatifs à une famille de références est intéressante quand le transfert est **obligatoire** comme c'est le cas avec les registres. Bien que nous ayons éliminé ce type de mémoire rapide (cf. § 1.3.3), nous allons montrer sur un exemple que l'utilisation d'une dépendance comme direction de découpage et l'assimilation d'un registre vectoriel à un supernoeud conduit à un code intéressant.

Le code:

```
DO I = 1, L
  DO J = 1, M
    A(I,J) = A(I,J) + A(I,J+1) + A(I,J-1)
```

Code initial

peut être partitionné selon la dépendance $\vec{d}$ et sa direction parallèle naturelle comme c'est fait dans la figure 4.6.1 et devenir:

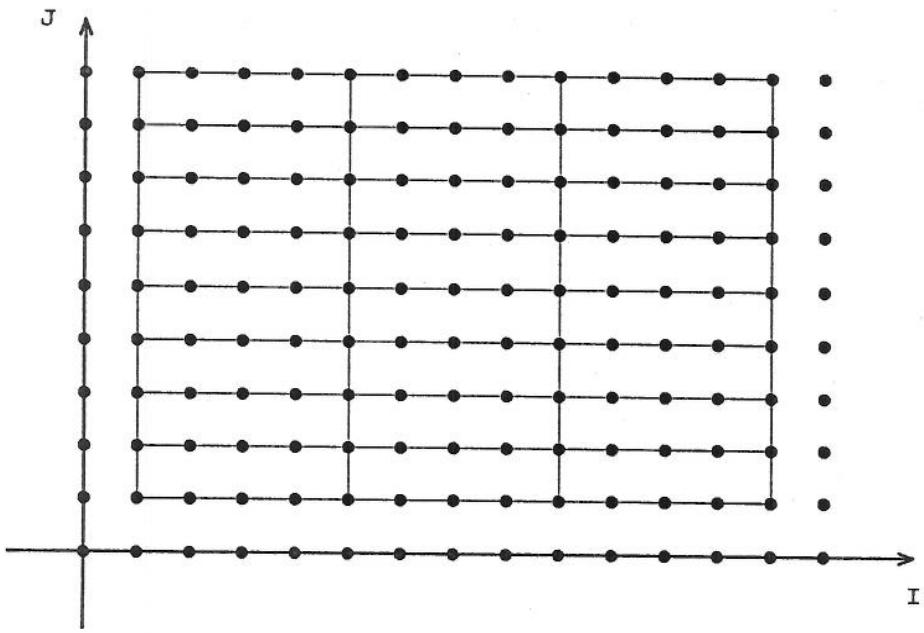
```
DO I = 1, L, L1
  DO J = 1, M
    A(I:I+L1-1,J) = A(I:I+L1-1,J) + A(I:I+L1-1,J+1) + A(I:I+L1-1,J-1)
```

Code partitionné

En prenant pour L1 la longueur des registres vectoriels, un générateur de code optimiseur pourra n'effectuer qu'un chargement au lieu de trois par corps de boucle.

4.6.2. Expression du coût de transfert

Le coût de transfert relatif à un supernoeud est donné par un polynôme en λ, de degré inférieur ou égal à N, dont les coefficients dépendent des directions de partitionnement choisies (P).



Partitionnement pour registres vectoriels

- Figure 4.6.1 -

Le coût de transfert global, obtenu en multipliant le coût précédent par le nombre de supernoeuds, est un polynôme en $1/\lambda$ car le nombre de supernoeuds varie comme λ^{-N}. Dans les cas les plus défavorables, quand une des références a la même dimension que l'espace d'itération, le partitionnement diminue seulement le terme constant de ce polynôme, ce qui n'est pas négligeable. Sinon, l'augmentation progressive de λ diminue le coût relatif des transferts (λ^{N-1}) par rapport aux calculs (λ^N)

4.6.3. Expression de la contrainte sur la mémoire rapide

La contrainte sur la mémoire rapide est aussi donnée par un polynôme en λ mais son degré maximum dépend de l'architecture. Si toutes les données doivent résider simultanément en mémoire rapide (cas des processeurs n'accédant pas la mémoire globale ou bien y accédant par blocs), son degré peut atteindre N. Sinon, il sera limité à N–1 puisque les références de dimension N seront conservées en mémoire globale.

4.6.4. Influence asymptotique de la taille de la mémoire rapide

Les deux résultats précédents peuvent être combinés pour obtenir des résultats sur l'influence de la taille de la mémoire rapide. Quand un programme est dominé par les calculs, c'est-à-dire quand le volume des calculs a une dimension supérieure au volume des données référencées, la présence d'une

mémoire locale permet d'augmenter la vitesse de l'unité arithmétique sans augmenter la bande passante de la mémoire. Kung[12] a obtenu des résultats théoriques montrant les relations qui existaient entre la taille de la mémoire locale et l'augmentation de puissance qu'elle permettait pour le produit et la triangularisation de matrice, les calculs sur des grilles discrètes, la transformée de Fourier rapide et le tri.

Tant que toutes les données ne tiennent pas en mémoire locale, le degré du polynôme donnant le nombre d'échanges en fonction de la taille du problème ne change pas, mais le coefficient du terme de plus haut degré peut être considérablement réduit. Soit C le volume de calcul total et λ_o le facteur d'échelle du supernoeud. Le volume de calcul d'un supernoeud est $C_o = \lambda_o^N$ tandis que le volume transféré est réduit à $V_o = \lambda_o^{N-1}$ quand aucun tableau n'a une dimension (réelle) supérieure à la dimension de l'espace de calcul N. Dans ces conditions, la saturation de la mémoire locale de volume K par le supernoeud conduit à l'égalité $K = \lambda_o^{N-1}$. Le volume transféré total V est égal au nombre de supernoeuds $S = C/C_o$ multiplié par le volume transféré par supernoeud V_o:

$$V = \frac{C}{C_o} V_o = \frac{C\lambda_o^{N-1}}{\lambda_o^N} = \frac{C}{K^{\frac{1}{N-1}}}$$

Dans le cas de la multiplication de matrice, $N = 3$ et les références sont de dimension $2 = N-1$. On retrouve un de résultats de Kung: la réduction de la bande passante mémoire nécessaire au produit de matrice est proportionnelle à la racine carré de la taille de la mémoire locale.

References

1. W. A. Abu-Sufah, "Improving the Performance of Virtual Memory Computers," Ph.D. Thesis, DCS Report No. UIUCDCS-R-78-945, University of Illinois at Urbana-Champaign (Jan. 1979).

2. W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," Report, IBM T.J. Watson Research Center, Yorktown Heights, N.Y. 10598 (July 1985).

3. F. A. Briggs and M. Dubois, "Effectiveness of Private Caches in Multiprocessor Systems with Parallel-Pipelined Memories," *IEEE Transactions on Computers* **C-32**(1), pp.48-59 (Jan. 1983).

4. L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers* **C-27**(12), pp.1112-1118 (Dec. 1978).

5. M. Dubois and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers* **C-31**(11), pp.1083-1099 (Nov. 1982).

6. D. D. Gajski, D. H. Lawrie, D. J. Kuck, and A. H. Sameh, "CEDAR," pp. 306-309 in *Spring Compcon 84*, Computer Society Press, San Francisco (1984).

7. R. Gottlieb, K. Kimball, T. Jaskiewicz, and R. Swift, "A New Way To Speed Up a Supercomputer," *Electronics* **58**(30), pp.56-58 (July 1985).

8. N. Halbwachs, "Détermination automatique de relations linéaires par les variables d'un programme," Thèse 3ème cycle, I.N.P., Grenoble (1979).

9. M. Hill, "Design Decisions in SPUR," *IEEE Computer* **19**(10), pp.8-22 (Nov. 1986).

10. W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy," Technical Report CSRD, University of Illinois at Urbana-Champaign, Urbana-Champaign (Feb. 1986).

11. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

12. H. T. Kung, "Memory Requirements for Balanced Computer Architectures," Report CMU-CS-85-158, Carnegie-Mellon University (June 1985).

13. R. L. Lee, "The Effectiveness of Memory Hierarchies in Large Tightly-Coupled MIMD Computer Systems," Technical Report, Univ. of Illinois at Urbana-Champaign (1985).

14. R. Triolet, "Contribution à la parallélisation automatique de programmes," Thèse de Docteur-Ingénieur, Université Paris VI (Déc. 1984).

5. TEMPS DE CALCUL D'UN SUPERNOEUD

Comme nous venons d'étudier indépendamment le coût du transfert des données de la mémoire la plus lente à la mémoire la plus rapide, nous ferons l'hypothèse que le temps de calcul va pouvoir être évalué en sachant où se trouve chaque donnée.

En fait cette précision a une portée pratique limitée car le temps de calcul va dépendre plus de la qualité du générateur de code et de son optimiseur que de la localisation des données. L'utilisation du nombre d'opérations flottantes ne donne pas de résultats intéressants car les compilateurs s'accommodent mieux d'expressions arithmétiques compliquées que de leur décomposition en une suite d'affectations équivalentes ne nécessitant chacune qu'une seule opération. Le compilateur vectoriseur de Cray, CFT, a même été longtemps sensible aux parenthèses...

Le problème de l'évaluation du temps d'exécution d'instructions de haut niveau n'est pas encore maîtrisé et nous allons nous contenter de distinguer les temps d'exécution obtenus avec des processeurs scalaires ou des processeurs vectoriels efficaces sur les vecteurs courts (20 à 64 éléments) et les processeurs vectoriels comme le Cyber 205 qui ne donnent leur pleine mesure que pour d'énormes vecteurs.

Nous nous restreignons au cas de la dimension 2 dans ce chapitre.

5.1. Influence de l'architecture du processeur élémentaire

5.1.1. Processeur élémentaire scalaire

Chaque noeud va être exécuté dans le même temps et, si l'on fait abstraction de l'overhead de contrôle, le temps total va donc être proportionnel au nombre de noeuds, c'est-à-dire au volume du supernoeud.

Le temps d'exécution ne va pas être sensible à la forme du supernoeud puisque l'exécution d'un noeud est indépendante de celle de ses voisins. Le seul effet que peut avoir la forme est de permettre d'augmenter la taille des supernoeuds grâce à une meilleure utilisation de la mémoire locale (cf. chapitre 4) et l'ordonnancement peut permettre une meilleure utilisation des blocs de cache ou des pages d'une mémoire virtuelle.

Le temps de calcul d'un supernoeud est donc de la forme $\alpha\lambda^N$ et il n'intervient pas directement dans l'optimisation du partitionnement. Mais ce type de multiprocesseur est en voie de disparition car il est plus facile d'ajouter des instructions vectorielles à un processeur que de faire un multiprocesseur et, surtout, les machines vectorielles sont plus faciles à programmer que les machines parallèles.

Cependant, il est envisageable de construire des processeurs élémentaires permettant à deux instructions, ou plus, d'être exécutées simultanément comme dans le CDC 6600. Ce parallélisme peut leur permettre d'atteindre des vitesses similaires à celle des processeurs vectoriels sans en avoir la rigidité,

moyennant une optimisation de l'ordonnancement des instructions. Le temps de calcul d'un supernoeud sur une telle machine pourrait être indépendant de sa forme et redonner de l'intérêt au cas *processeur scalaire*.

5.1.2. Processeur vectoriel à faible $n_{1/2}$

Le coefficient $n_{1/2}$ a été introduit par Hockney et Jesshope[1] comme caractéristique complémentaire de la puissance de pointe que donnent les constructeurs. Cette dernière ne permet pas de prévoir la manière dont une machine vectorielle va se comporter sur un problème donné. Le coefficient $n_{1/2}$ est la longueur de vecteur pour laquelle la machine atteint la moitié de sa puissance de pointe.

Les machines qui ont des $n_{1/2}$ faibles comme l'Alliant FX/8 ou les Cray en général sont capables de traiter efficacement des vecteurs courts, ce qui nous intéresse puisque le partitionnement va conduire à des supernoeuds plus petits que le domaine d'itération initial et même beaucoup plus petits quand on voudra atténuer le problème de *Load Balancing* au chapitre 6.

De plus, cette capacité à traiter les vecteurs courts permet d'envisager de traiter des vecteurs de longueur variable comme on en trouvera dans les supernoeuds la plupart du temps. Il n'y a pas de raison qu'un des côtés ou qu'une des faces du supernoeud puisse être exécuté en mode vectoriel. Dans l'exemple de la barre proposé au chapitre 1, le partitionnement choisi en 1.2.5 donne des vecteurs de longueur croissante puis décroissante puisque l'ordonnancement vectoriel naturel produit des vecteurs horizontaux (figure 5.1.2).

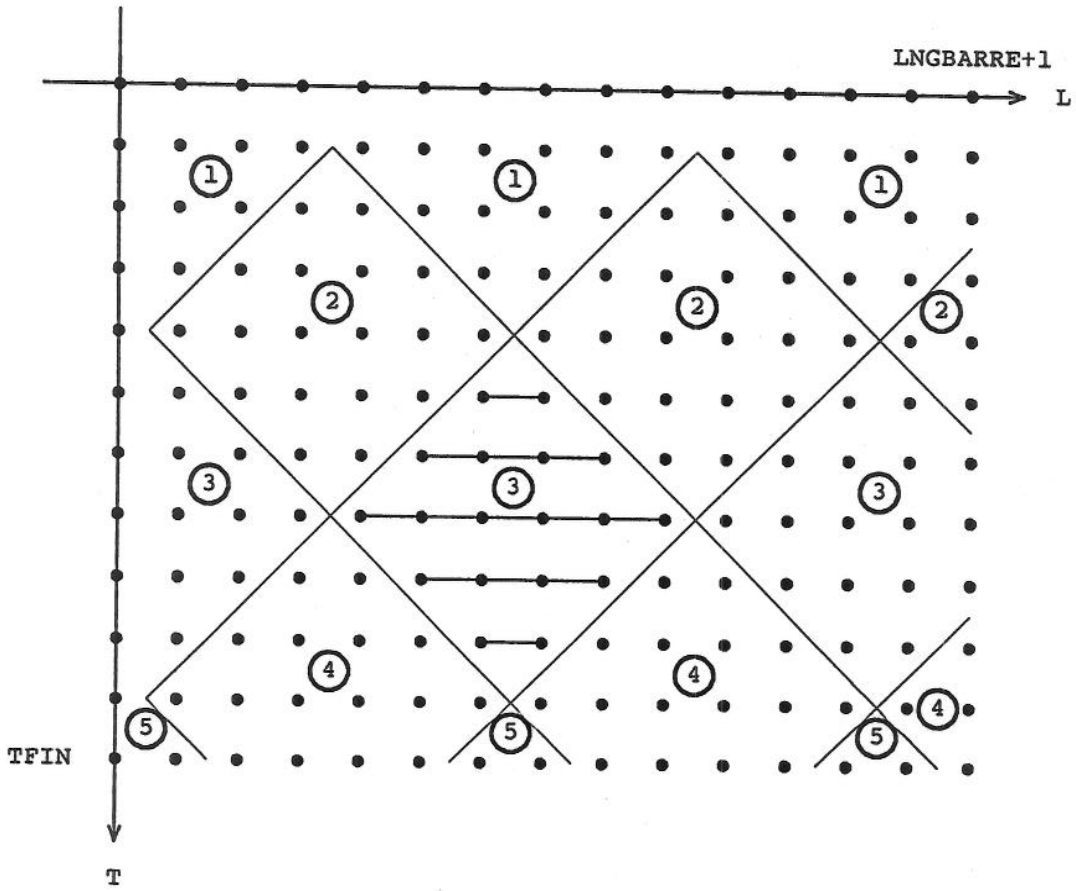
Sous la condition qu'on choisisse un ordonnancement et une taille de supernoeud ne conduisant pas à des vecteurs ridiculement courts, n'importe quel partitionnement va permettre une exécution vectorielle du supernoeud à une vitesse très proche de la vitesse de pointe. Le temps d'exécution d'un supernoeud va encore être proportionnel au volume, $\beta\lambda^N$, mais la constante β est plus faible que la constante α du paragraphe précédent car il faut prendre un temps d'exécution vectoriel pour chaque noeud.

En résumé, ce type de machine n'introduit pas de contrainte forte sur la taille de vecteurs, permet d'avoir des vecteurs de longueur variable et donc accepte n'importe quelle forme de supernoeud[†].

5.1.3. Processeur vectoriel à fort $n_{1/2}$

Ce type de machine nécessite de longs vecteurs pour opérer efficacement. Par exemple, le Cyber 205 de CDC est caractérisé par des $n_{1/2}$ de l'ordre de 100 contre 10 pour le Cray 1. Nous allons donc devoir essayer de n'avoir à calculer que des vecteurs de longueur importante et cela ne sera possible qu'en partitionnant le domaine d'itération de manière à avoir des vecteurs de longueur constante pour éviter

[†]Nous n'avons malheureusement pas eu le temps de voir sur l'Alliant FX/8 si le gain qu'auraient apporté des vecteurs de longueur fixe compensait la dégradation du *load balancing* qu'ils entrainaient. Ce type d'étude nécessite plus d'expérimentations que nous n'avons pu en faire.



Partitionnement à vecteurs de longueur variable

- Figure 5.1.2 -

les vecteurs courts qu'introduisent les partitionnements similaires à celui qui est représenté en figure 5.1.2. Seul un domaine d'itération gigantesque et des supernoeuds très grands permettraient d'amortir le coût des vecteurs courts.

La mise en oeuvre de la méthode hyperplane va permettre de déterminer la direction d'itération optimale à partir des vecteurs de dépendance. Chercher à maximiser la longueur des vecteurs revient à en réduire le nombre ou à minimiser la norme du vecteur de direction d'itération en maintenant ses coordonnées entières, pour avoir un maximum de noeuds par unité de longueur du front d'onde. Un problème équivalent consiste à essayer de minimiser le temps d'exécution sur un multiprocesseur disposant d'un nombre infini de processeurs élémentaires. Kuhn [4] et Wolfe[5] ont proposé des solutions assez simples pour la dimension 2 et Kuhn a montré que l'application itérative de la méthode 2-D pour réordonnancer plus de 2 boucles imbriquées conduisait à la solution optimale dans le cas de programmes

réels bien que cela ne soit pas vrai dans le cas général. La technique de Wolfe, l'*Index Set Shifting*, a été présentée au chapitre 3. Les résultats de Kuhn sont discutés en § 8.1. Quand plusieurs directions vectorielles sont compatibles avec D, on peut effectuer le choix en étudiant le pas d'accès à la mémoire que génèrent les références et en prenant la direction qui maximise le nombre d'accès à pas de un.

La direction vectorielle ainsi obtenue va donc devoir se retrouver dans la matrice de partitionnement P, dont il ne restera plus, en dimension 2, qu'un vecteur à déterminer au chapitre 6. Elle correspond à une direction de partitionnement valide car une direction d'itération vérifie $\vec{h}\ \vec{d}_k > 0$ ce qui implique $\vec{h}\ \vec{d}_k \geq 0$ (cf. § 2.3.2).

En résumé, ce type de machine contraint simultanément la forme des supernoeuds et leur taille minimale (le facteur d'échelle λ). Cela peut conduire avec notre méthode à une impossibilité si la mémoire locale est trop petite. Mais, une fois de plus, le temps d'exécution reste proportionnel au volume du supernoeud, avec un troisième coefficient que nous ignorons autant que les précédents vu les problèmes que posent les générateurs de code.

5.2. Coûts et contraintes

5.2.1. Coûts

Dans tous les cas, le coût, c'est-à-dire le temps de calcul, est asymptotiquement proportionnel au volume du supernoeud mais le coefficient de proportionnalité varie avec l'architecture et le générateur de code utilisé. Des termes de degré inférieur sont générés par les instructions de contrôle et, éventuellement, par les vecteurs de longueur variable.

Il n'est pas possible de savoir a priori si ces termes de plus bas degré, négligeables asymptotiquement, peuvent être ignorés dans la pratique. Cela dépend de la taille du corps de boucle, du rapport des vitesses scalaire et vectorielle, et une fois de plus, du générateur de code: le coefficient du terme en λ^{N-1} peut être très important.

5.2.2. Contraintes sur λ

Trois types de contraintes peuvent donc être imposés à λ par l'architecture de la machine. Tout d'abord on veut être sûr d'avoir des vecteurs assez longs, ce qui donne des contraintes comme $\alpha\lambda > 32$ pour une machine ayant un $n_{1/2} \approx 8$ et des vecteurs dont la longueur moyenne est égale à la moitié du côté d'un supernoeud.

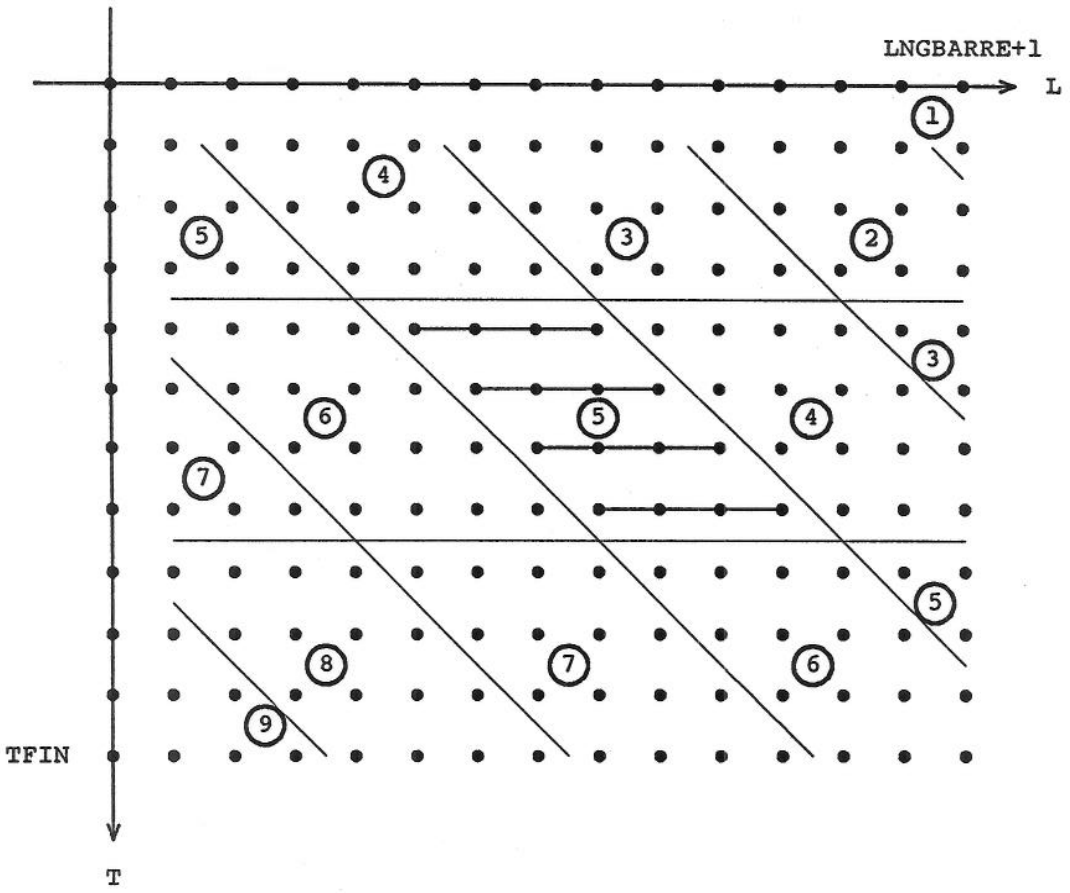
Dans certaines machines, le matériel ne gère que des vecteurs de longueur inférieure à 32 ou 64 et c'est le logiciel qui est chargé de tronçonner le vecteur réel en vecteurs élémentaires. Cette opération entraîne de l'overhead de contrôle qu'il faut limiter en générant a priori des vecteurs de 32 ou 64 éléments. Cette deuxième contrainte peut remplacer la précédente, sauf dans le cas des machines

vectorielles sans registres (Cyber 205, ETA 10).

Enfin, on peut vouloir une grande valeur pour λ pour garantir la validité de l'approximation du temps de calcul par le terme de plus haut degré.

5.2.3. Contraintes sur la forme

Seules les machines à fort $n_{1/2}$ imposent a priori une contrainte sur la forme des supernoeuds. Cette contrainte sur P induit une contrainte sur H et limite donc le nombre de vecteurs de direction d'itération globale $\vec{1}_H$. Certaines directions, qui permettaient d'avoir un degré de parallélisme égal pour chaque front, ne sont plus possibles. C'est le cas avec l'exemple utilisé au paragraphe 1.2, comme on le voit sur la figure 5.2.3 où le degré de parallélisme varie de 1 à 3 suivant les fronts alors qu'il valait toujours 3 sur la figure 5.1.2.



Partitionnement à vecteurs de longueur constante

- Figure 5.2.3 -

Notons que l'obtention de performances *supervectorielles* nécessite dans le cas de la figure 5.2.3 des instructions de copie de registres vectoriels ou la désignation de plusieurs registres comme destinations d'opérations, et la possibilité d'insérer un scalaire en début ou fin de registre vectoriel. Ceci ouvrirait la voie à un partitionnement par bande comme le proposaient Karp et Miller en 1967[3] . En l'absence de telles instructions, il faut choisir un des vecteurs de dépendances pour effectuer le découpage, comme on l'a vu en 4.6.1.

5.3. Conclusion

Il n'est pas possible de déterminer les valeurs des coefficients sans connaître précisément la qualité du restructureur et du générateur de code utilisés. De plus, la courte étude expérimentale que nous avons effectuée sur l'Alliant[2] a montré qu'il fallait se méfier des simplifications asymptotiques. Nous ne proposons donc pas dans ce chapitre un système d'équations permettant de dériver P et λ. Nous proposons juste de contraindre le facteur d'échelle λ et, dans le cas des architectures à fort $n_{1/2}$, la matrice de partitionnement P de manière à toujours avoir un temps d'exécution proportionnel au volume.

References

1. R. W. Hockney and C. R. Jesshope, *Parallel Computers - Architecture, Programming and Algorithms*, Adam Hilger Ltd., Bristol (1981).

2. F. Irigoin and R. Triolet, "Supernodes and Alliant FX/8 Minisupercomputer," Tech. Rep. ENSMP-CAI-86-081, Ecole des Mines de Paris (Aug. 1986).

3. R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM* **14**(3), pp.563-590 (July 1967).

4. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

5. M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD. Thesis, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign (1982).

6. CHOIX DE LA FORME ET DE LA TAILLE DES SUPER-NOEUDS

Nous allons tout d'abord calculer le temps d'exécution d'un supernoeud en combinant les résultats des chapitres 4 et 5. Il ne sera possible d'obtenir que le type de la fonction et non l'ensemble des valeurs numériques, puisque, comme nous venons de le voir au chapitre précédent, les coefficients dépendent du générateur de code utilisé.

Ce premier résultat sera d'abord utilisé pour minimiser le temps d'exécution total sous l'hypothèse théorique du nombre de processeurs illimité. Les différentes étapes de l'optimisation sont la construction de l'ensemble des vecteurs critiques, le choix d'une base de partitionnement à directions de partitionnement fixées, la sélection des meilleures directions de partitionnement et enfin l'optimisation de la taille des supernoeuds. Cette hypothèse, à première vue irréaliste, permet par exemple à Kuhn[5] de maximiser la longueur des vecteurs, ce qui ne nous intéresse pas directement, mais nous permet aussi de diminuer l'overhead de contrôle et de calculer le nombre maximal de processeurs utiles.

Le cas du nombre de processeurs limité est plus délicat. Nous évaluerons quantitativement l'overhead global, l'overhead de balayage des noeuds vides, le temps de calcul des supernoeuds partiels, l'équilibre de la charge des processeurs à l'intérieur d'un front, l'évolution du degré de parallélisme des fronts et le déroulement de l'exécution dans la zone intermédiaire du domaine d'itération où tous les processeurs peuvent être considérés comme actifs en permanence. Ceci nous permettra de déduire une approximation du temps global d'exécution.

La conclusion de ce chapitre sera constituée d'un tableau récapitulatif indiquant d'une part l'ensemble des relations existant entre les divers paramètres qu'il faut déterminer pour fixer un partitionnement et le temps d'exécution, et rappelant d'autre part quelles contraintes pèsent sur leurs choix.

Dans ce chapitre, tous les calculs sont effectués en faisant une approximation continue sur toutes les grandeurs manipulées. Les résultats sont établis pour une dimension N quelconque, avec quelques exceptions.

6.1. Temps d'exécution d'un supernoeud

Comme nous l'avons déjà remarqué en 1.1.2, le temps d'exécution global est la longueur du chemin critique d'un graphe de tâches. Ces tâches sont de trois natures: calcul, transfert de données et contrôle. Le temps d'exécution d'un supernoeud va intégrer des tâches de chaque sorte de telle manière que le chemin critique soit une suite de supernoeuds.

L'overhead de contrôle comprend les appels à des primitives de multitasking comme les lancements de tâches ou les synchronisations entre tâches, etc... et l'exécution du code de contrôle des supernoeuds, qui comprend des boucles DO et des tests (cf. § 3.2). L'exécution de ce code joue un rôle d'autant plus important qu'il est purement séquentiel. En prenant *supernoeud* au sens général, c'est-à-dire supernoeud plein, partiel ou vide, on peut considérer que le temps correspondant est constant pour chaque supernoeud et qu'il ne dépend donc ni de P, ni de λ. L'influence de ce temps sur la taille des tâches a été étudiée sur le Cray X-MP par Calahan[1] car le temps d'exécution très important des premières primitives fournies par Cray a tout de suite attiré l'attention des chercheurs sur le problème de la granularité du parallélisme. Mais, même avec des machines comme l'Alliant FX/8 où ces primitives sont câblées, les versions partitionnées des programmes sont pénalisées par le code de contrôle nécessaire au parcours des supernoeuds.

L'overhead de mouvement de données a été étudié au chapitre 4 où l'on a montré que c'était un polynôme de degré égal au maximum des dimensions réelles des références présentes dans le corps de boucle, dans la mesure où la mémoire locale était assez grande pour éviter de transférer chaque donnée utile à un supernoeud plus d'une fois. Le degré est donc au plus de N. Mais les coefficients dépendent de la matrice de partitionnement P. Ils seront d'autant plus faibles que les autodépendances permettront de réutiliser davantage les données à l'intérieur d'un supernoeud.

Enfin, nous venons de voir au chapitre 5 que le temps de calcul était proportionnel au volume du supernoeud, que les processeurs élémentaires soient vectoriels ou non. S'ils sont vectoriels et inefficaces sur les vecteurs courts (i.e. fort $n_{1/2}$), il faut contraindre P en lui assignant une direction vectorielle. En toute rigueur, ce temps de calcul intègre aussi du code de contrôle et éventuellement des instructions vectorielles de longueurs variables et devrait être donné comme un polynôme en λ de degré N. Seule l'expérience permettra de dire si les termes de plus bas degré peuvent être négligés dans tous les cas. Le coefficient du terme de plus haut degré dépend du mode de fonctionnement des processeurs élémentaires.

Le temps total d'exécution d'un supernoeud est la somme de ces trois composantes tant qu'on n'envisage pas d'effectuer des transferts en parallèle avec des calculs et vaut:

$$\alpha_0 + \alpha_1\lambda + \cdots + \alpha_N\lambda^N$$

Le calcul des coefficients α_0, α_1, etc... semble impossible. Il faudrait les identifier dans chaque cas par des expériences.

6.2. Nombre de processeurs illimité

L'hypothèse irréaliste du nombre de processeurs illimité permet d'obtenir des résultats utiles. Tout d'abord, si le domaine d'itération est petit et le nombre de processeurs élémentaires assez grand, il n'est pas sûr qu'il soit utile de générer des supernoeuds tout petits. L'overhead de synchronisation peut anéantir le gain dû au parallélisme et il faut donc vérifier que la taille de supernoeud qu'on obtient pour la machine à p processeurs est supérieure à la taille de supernoeud qu'on obtient pour une machine à nombre infini de processeurs. Sinon, il vaut mieux ne pas utiliser tous les p processeurs. Le problème, comme on le verra en 6.3, vient de ce que l'on résout le problème de l'optimisation du partitionnement avec p processeurs en considérant p comme une constante et non comme une variable au même titre que λ.

D'autre part, cette hypothèse revient à pénaliser le nombre de fronts par rapport à la largeur des fronts car cette dernière n'a pas d'importance. Dans le cas où les supernoeuds sont réduits à un noeud ($\lambda = 1$), cela revient à maximiser le nombre de noeuds par fronts et à réduire le nombre de fronts comme le fait la méthode hyperplane.

Bien que nous nous soyons fréquemment limité à la dimension 2, nous allons essayer de résoudre ce problème dans le cas général.

6.2.1. Calcul du temps total

Au chapitre 2, § 2.4.2, la matrice H a été choisie de telle sorte que deux noeuds de l'espace d'itération, joints par un vecteur $\vec{w}$, soient aussi joints par un chemin critique comprenant $< \vec{1}_H \ \vec{w} >$ supernoeuds. En prenant en compte le facteur d'échelle λ et en appelant L la longueur de ce chemin, on obtient en discret:

$$L = \left\lceil \frac{< \vec{1}_H \ \vec{w} >}{\lambda} \right\rceil + K - N + 1$$

où N est la dimension de l'espace et K une constante valant 0, 1 ou 2 suivant la position de l'origine du partitionnement par rapport aux deux points considérés. En effet, les fronts de supernoeuds sont *bosselés* et deux noeuds dont les vecteurs d'itération $\vec{j}$ et $\vec{j}'$ vérifient $\vec{1}_H \ \vec{j} = \vec{1}_H \ \vec{j}'$ peuvent appartenir à deux fronts différents. Le phénomène pouvant se produire à l'arrivée aussi bien qu'au départ, l'erreur possible atteint la valeur de 2.

Comme les domaines d'itération intéressants sont très grands, les vecteurs $\vec{w}$ représentatifs des chemins critiques de ces espaces auront une norme importante et nous ferons l'approximation:

$$L = \frac{< \vec{1}_H \ \vec{w} >}{\lambda}$$

Le temps total T va être égal au temps d'exécution d'un supernoeud, dont nous avons vu (cf. § 6.1) que c'était un polynôme en λ de degré N à condition de ne pas prendre en compte le parallélisme entre échanges et calculs, multiplié par le nombre de supernoeuds du chemin critique:

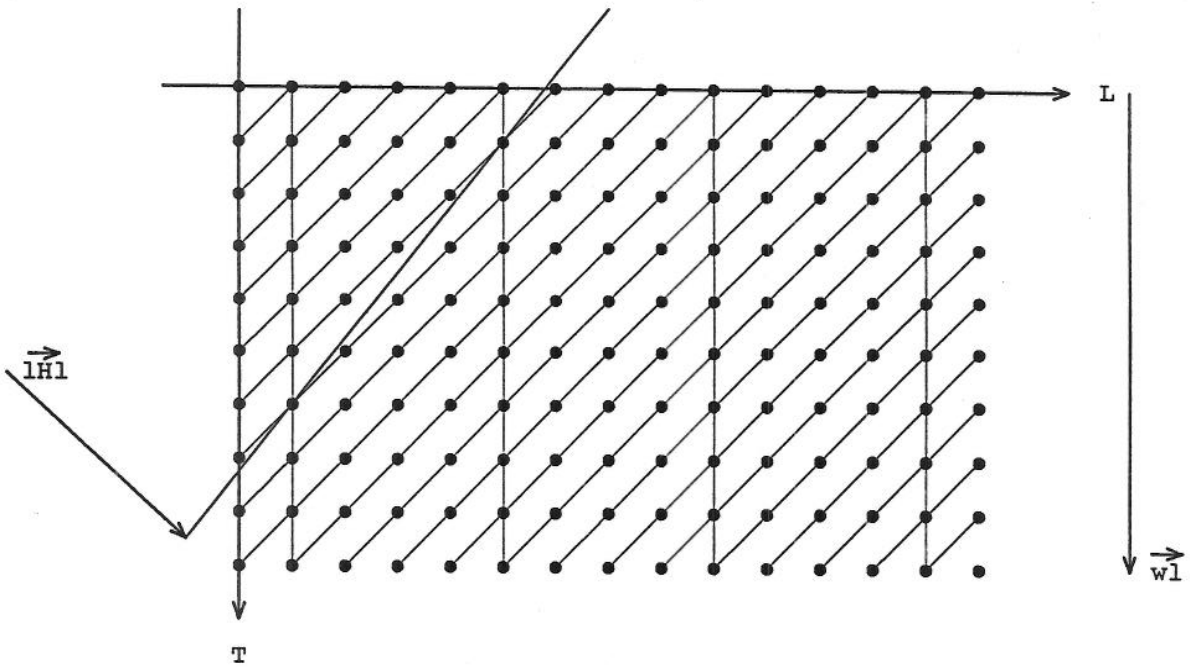
$$T = < \vec{1}_H \ \vec{w} > \left(\frac{\alpha_0}{\lambda} + \alpha_1 + \alpha_2 \lambda + \cdots \alpha_N \lambda^{N-1} \right)$$

Cette formule simple cache plusieurs problèmes. Premièrement, il n'existe pas un unique vecteur critique $\vec{w}$, représentatif du domaine d'itération. Le vecteur $\vec{w}$ est une fonction de la direction de $\vec{1}_H$. D'autre part, la norme de $\vec{1}_H$ varie avec sa direction et avec la base de partitionnement choisie.

Pour illustrer ces points, trois partitionnements différents du même espace d'itération, soumis aux mêmes dépendances, sont représentés en figures 6.2.1.a, b, c. Le premier est défini par:

$$P_1 = \begin{pmatrix} 1/2 & -2 \\ 0 & 2 \end{pmatrix} \quad H_1 = \begin{pmatrix} 2 & 0 \\ 2 & 1/2 \end{pmatrix} \quad \vec{1}_H = \begin{pmatrix} 2 \\ 3/2 \end{pmatrix} \quad \lambda = 2 \quad \vec{w}_1 = \begin{pmatrix} 9 \\ 14 \end{pmatrix}$$

ce qui donne avec l'approximation continue 26.5 supernoeuds de 4 noeuds, contre 28 en prenant en compte les effets de discrétisation.



Premier partitionnement

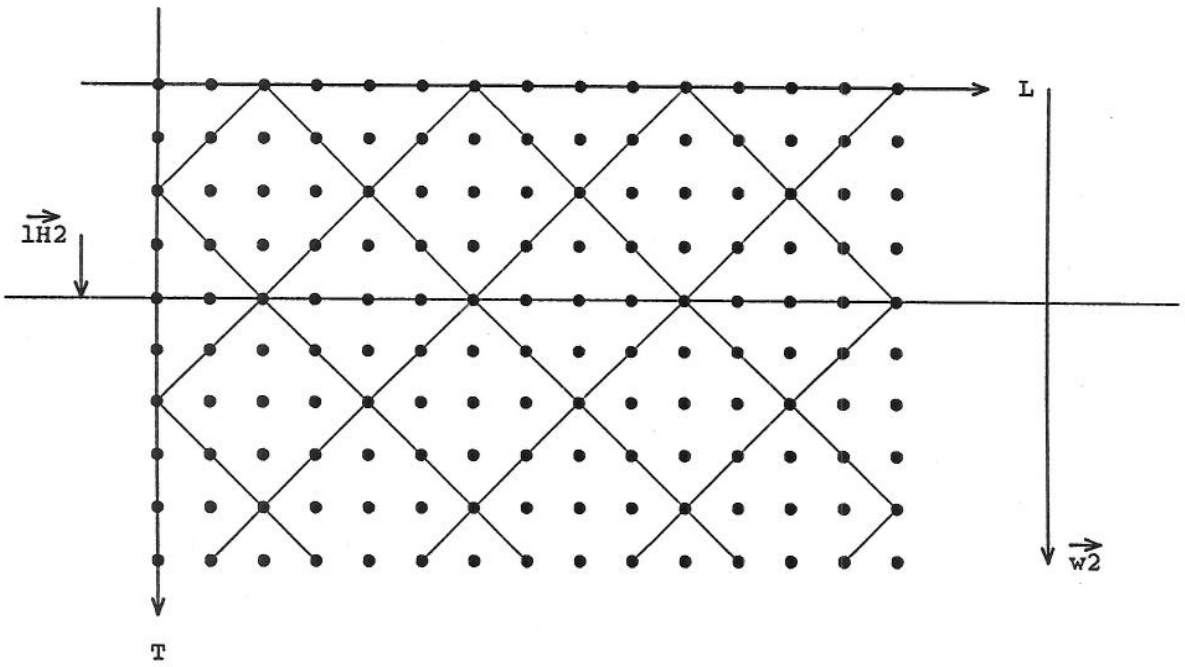
- Figure 6.2.1.a -

Le deuxième partitionnement est défini par:

$$P_2 = \begin{pmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ -\sqrt{2}/2 & \sqrt{2}/2 \end{pmatrix} \quad H_2 = \begin{pmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ -\sqrt{2}/2 & \sqrt{2}/2 \end{pmatrix} \quad \vec{1}_H = \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix} \quad \lambda = 2\sqrt{2} \quad \vec{w}_2 = \begin{pmatrix} 9 \\ 0 \end{pmatrix}$$

La longueur du chemin critique est maintenant de 4.5 supernoeuds de 8 noeuds chacun soit 36 noeuds. En prenant $\lambda = 2$ pour nous remettre dans les conditions du premier exemple, le nombre de noeuds à calculer dans le chemin critique serait:

$$T = \frac{< \vec{1}_H \ \vec{w} >}{\lambda} = \frac{9\sqrt{2}}{2} = 6.363$$



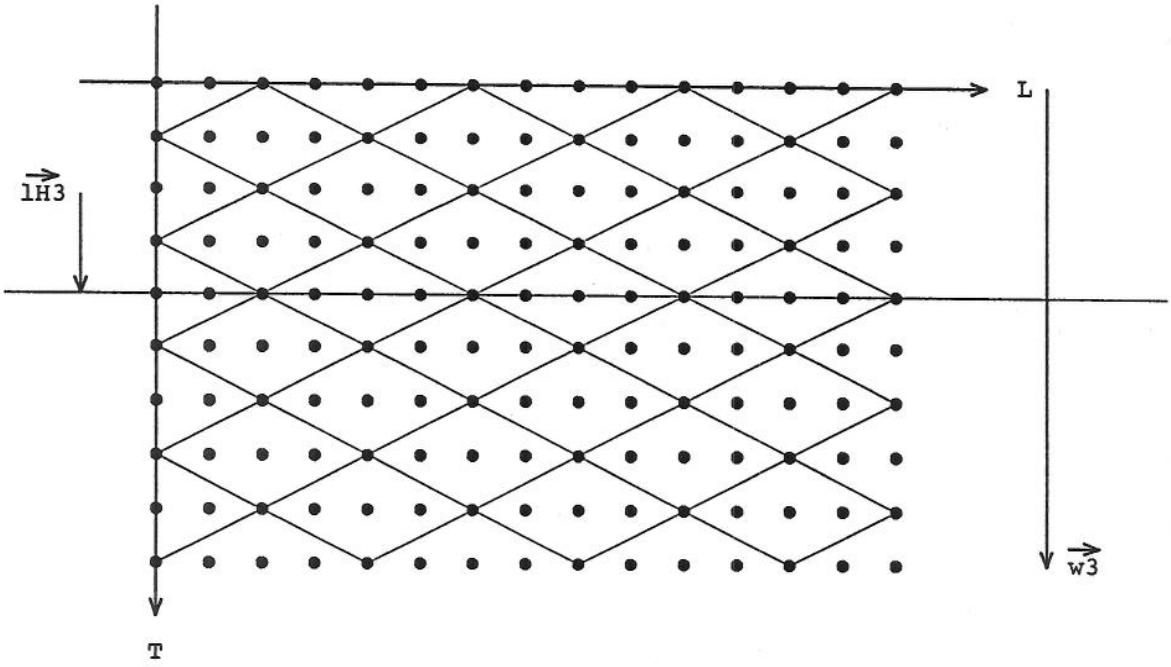
Deuxième partitionnement

- Figure 6.2.1.b -

Enfin, le troisième exemple utilise des directions de partitionnement différentes. Elles ne sont pas critiques et introduisent un peu de sursynchronisation. Voici les matrices et coefficients de définition:

$$P_3 = \begin{pmatrix} 1/2 & 1/2 \\ -1 & 1 \end{pmatrix} \quad H_3 = \begin{pmatrix} 1 & 1 \\ -1/2 & 1/2 \end{pmatrix} \quad \vec{1}_H = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \quad \lambda = 2 \quad \vec{w}_3 = \begin{pmatrix} 9 \\ 0 \end{pmatrix}$$

L'approximation continue donne 9 supernoeuds de 4 noeuds, soit 36 noeuds à calculer au total.



Troisième partitionnement

- Figure 6.2.1.c -

Dans l'expression donnant T, les coefficients α_0 et α_N ne dépendent pas de $\vec{1}_H$. En effet, la forme du partitionnement ne joue ni sur le volume de calcul ni sur les transferts de tableaux dont la dimension réelle est égale à la dimension de l'espace d'itération, ni sur les synchronisations. Le coefficient du terme de plus haut degré, α_N, est donc indépendant de H bien qu'il ait été réduit par le partitionnement: un bon générateur de code va pouvoir réutiliser la donnée obtenue pour une référence pour satisfaire les autres références de la même famille et les références à des tableaux dont la dimension réelle est inférieure à N ne génèrent plus un coût en λ^N. Par contre, le coefficient α_{N-1} dépend de H comme on l'a vu au chapitre 4, § 4.2.2. Pour simplifier les calculs, nous allons l'ignorer ainsi que les coefficient α_1, α_2, etc... au profit du terme de plus haut degré et du seul terme qui soit décroissant en λ et susceptible d'avoir une très forte valeur (Cray, hypercubes,...). Si nous obtenons des valeurs de λ suffisamment grandes au terme de l'optimisation, cette approximation sera validée a posteriori. Le temps d'exécution total devient donc:

$$T = < \vec{1}_H \ \vec{w} > \left(\frac{\alpha_0}{\lambda} + \alpha_N \lambda^{N-1} \right)$$

6.2.2. Les étapes de l'optimisation

Nous nous restreignons au cas des processeurs élémentaires scalaires pour pouvoir choisir complètement la base de partitionnement P. Le choix de la direction d'itération globale et des directions de partitionnement se fait en plusieurs étapes:

- construction de l'ensemble des vecteurs critiques;
- choix de la base de partitionnement optimale à directions de partitionnement fixées et pour un vecteur critique;
- sélection des meilleures directions de partitionnement;
- ajustement du facteur d'échelle λ.

Les trois premières étapes ne se préoccupent que du terme $< \vec{1}_H \ \vec{w} >$, profitant de l'indépendance de α_0 et de α_N vis-à-vis de H.

Cela ne garantit pas que le supernoeud résultant soit exécutable efficacement sur processeur vectoriel.

6.2.3. Construction de l'ensemble des vecteurs critiques

L'ensemble des vecteurs critiques est obtenu en cherchant le maximum des formes linéaires du type $< \vec{1}_H \ \vec{w} >$ pour tous les $\vec{1}_H$ possibles. De tels vecteurs joignent donc des sommets du domaine d'itération, ce qui, vu la sémantique des boucles DO, en limite le nombre à $2^N \times (2^N - 1)$. En effet, chaque boucle définit deux contraintes et pour chacune des boucles il faut choisir l'une ou l'autre de ces contraintes pour déterminer un sommet. Il y a donc 2^N sommets au plus, car de temps en temps les deux contraintes d'une boucle peuvent être confondues. Le nombre de vecteurs se déduit immédiatement du nombre de sommets, puisqu'on peut choisir librement l'origine (2^N choix) mais que l'extrémité doit être différente ($2^N - 1$ choix). Cela donne 12 vecteurs en dimension 2. On ne garde que les vecteurs lexicopositifs, ce qui en divise le nombre par 2. Il est donc possible de chercher un partitionnement pour chacun d'entre eux, c'est-à-dire de considérer dans la suite que $\vec{w}$ est fixé.

En reprenant le domaine d'itération utilisé figure 6.2.1, on obtient:

$$W = \left\{ \vec{w}_1 = \begin{pmatrix} 9 \\ 0 \end{pmatrix}, \vec{w}_2 = \begin{pmatrix} 9 \\ 14 \end{pmatrix}, \vec{w}_3 = \begin{pmatrix} 9 \\ -14 \end{pmatrix}, \vec{w}_4 = \begin{pmatrix} 0 \\ 14 \end{pmatrix} \right\}$$

Ces vecteurs sont représentés figure 6.2.3.

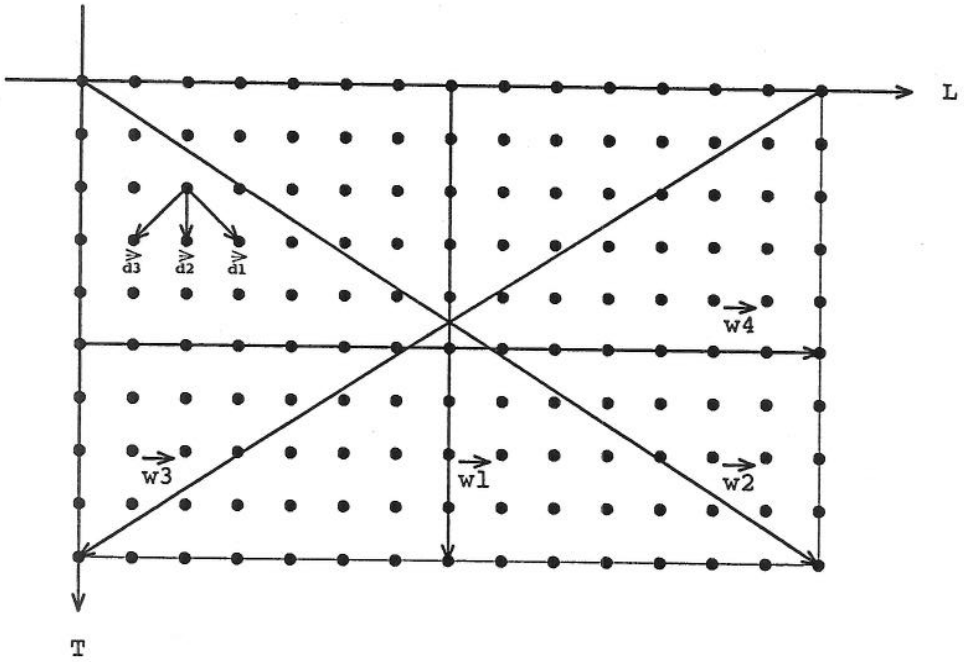
6.2.4. Choix d'une base de partitionnement à directions fixées

Si les directions de partitionnement sont fixées par une base H, l'ensemble des directions d'itérations possibles $\vec{1}_H$ est donné par les conditions suivantes:

$$\vec{1}_H = \sum_{i=1}^{N} \lambda_i \ \vec{h}_i$$

$$\prod_{i=1}^{N} \lambda_i = 1$$

$$\forall\, i \in [1..N] \quad \lambda_i > 0$$



Vecteurs critiques

- Figure 6.2.3 -

qui correspondent à un changement de base par une matrice diagonale Λ de déterminant égal à 1. Les λ_i peuvent donc prendre n'importe quelle valeur **positive** pour respecter les dépendances mais leur produit doit rester égal à 1 pour respecter la condition $det\,(H)=1$. En posant:

$$w_i = \vec{h}_i \; \vec{w}$$

on trouve par dérivation de $L = \vec{1}_H \; \vec{w}$ et de la contrainte sur le produit des λ_i :

$$d\,\lambda_i \vec{h}_i \, \vec{w} + d\,\lambda_j \vec{h}_j \, \vec{w} = 0 \qquad \frac{d\,\lambda_i}{\lambda_i} + \frac{d\,\lambda_j}{\lambda_j} = 0$$

On en déduit que:

$$\frac{\lambda_i}{\lambda_j} = \frac{\vec{h}_j \, \vec{w}}{\vec{h}_i \, \vec{w}} = \frac{w_j}{w_i}$$

En calculant tous les λ_i à partir de λ_j et en utilisant la contrainte sur le produit des λ_i , on trouve:

$$\prod_{i=1}^{N} \frac{w_j}{w_i} \lambda_j = 1$$

ce qui implique:

$$\lambda_j = \frac{\left(\prod_i w_i \right)^{1/N}}{w_j}$$

quand tous les w_j sont strictement positifs. Si un des w_j est nul ou négatif, cela veut dire que les deux sommets extrémités de $\vec{w}$ peuvent être exécutés simultanément parce qu'il n'y a pas dépendance entre eux et que $\vec{w}$ n'aurait pas dû être pris en considération (i.e. il faut que $\vec{w}$ appartienne au cône de dépendance).

Dans le cas de la figure 6.2.3, avec le partitionnement P_2 de la figure 6.2.1, un seul des vecteurs de W, $\vec{w}_1$, est intéressant car les autres peuvent conduire à des temps d'exécution nuls et ne peuvent donc pas être vecteurs critiques. En faisant les calculs, on obtient:

$$w_1 = \frac{9\sqrt{2}}{2} \qquad w_2 = \frac{9\sqrt{2}}{2} \qquad \lambda_1 = \lambda_2 = 1 \qquad \vec{1}_H = \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix}$$

et on vérifie qu'aucun des vecteurs potentiellement critiques ne conduit à un chemin plus long que celui qu'on a pour $\vec{w}_1$:

$$L_{\vec{w}_1} = 9\sqrt{2} \qquad L_{\vec{w}_2} = 9\sqrt{2} \qquad L_{\vec{w}_3} = 9\sqrt{2} \qquad L_{\vec{w}_4} = 0$$

En reportant les valeurs optimales des λ_i dans le calcul de la longueur du chemin, on trouve pour des directions de partitionnement définies par une base H vérifiant $det\,(H) = 1$:

$$L_H = \sum_i \lambda_i \, \vec{h}_i \cdot \vec{w} = \sum_i \frac{\left(\prod_j \vec{h}_j \cdot \vec{w} \right)^{1/N}}{\vec{h}_i \cdot \vec{w}} \, \vec{h}_i \cdot \vec{w}$$

$$L_H = N \left(\prod_j \vec{h}_j \cdot \vec{w} \right)^{1/N}$$

Les coefficients λ_i nous donnent aussi un partitionnement optimal pour les directions de H:

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{pmatrix} \qquad \hat{H} = \Lambda H \qquad \hat{P} = \left(\hat{H}^{-1} \right)^t = \left((\Lambda H)^{-1} \right)^t = \hat{P} = \Lambda^{-1} P$$

On remarque enfin que le vecteur $\vec{1}_{\hat{P}}$ est colinéaire au vecteur $\vec{w}$. En effet, en posant:

$$K = \left(\prod_j \vec{h}_j \cdot \vec{w} \right)^{-1/N}$$

on a par définition de $\hat{P}$ et de l'inverse d'une matrice diagonale:

$$\vec{1}_{\hat{P}} = K \ (<\vec{h}_1 \vec{w} > \vec{p}_1 + <\vec{h}_2 \vec{w} > \vec{p}_2 + \ \cdots \ + <\vec{h}_N \vec{w} > \vec{p}_N \ )$$

En utilisant la propriété qui sert à la définition de P à partir de H, $\vec{h}_i \ \vec{p}_j = \delta_{ij}$ (cf. § 2.4.2), on calcule les N produits scalaires de $\vec{1}_{\hat{P}}$ avec les vecteurs $\vec{h}_j$:

$$\vec{1}_{\hat{P}} \ \vec{h}_1 = K \ \vec{w} \ \vec{h}_1 \quad \vec{1}_{\hat{P}} \ \vec{h}_2 = K \ \vec{w} \ \vec{h}_2 \quad \cdots \quad \vec{1}_{\hat{P}} \ \vec{h}_N = K \ \vec{w} \ \vec{h}_N$$

ce qui montre la propriété. Géométriquement, cela signifie que la diagonale principale des supernoeuds est parallèle au vecteur critique. Intuitivement, on essaie d'avancer le plus possible dans la direction critique.

6.2.5. Sélection des meilleures directions de partitionnement

Il reste maintenant à justifier le choix des directions de partitionnement que nous avons supposé données par une base H. Le dernier cas présenté sur la figure 6.2.1 est un exemple fondé sur une base H différente de celle qui a été utilisée pour les deux premiers cas et pour les calculs que nous venons d'effectuer.

Soit H' une base dont tous les vecteurs sont des combinaisons linéaires positives d'une autre base H, c'est-à-dire que le cône des directions d'itération générables par H' est inclus dans le cône généré par H et, dualement, que le cône de P, qui exprime l'ordre partiel d'exécution, est inclus dans le cône de P'. La matrice de passage de H' à H, appelée $A = (a_{ij})$, a donc tous ses coefficients positifs et, pour respecter la condition habituelle $det(H) = det(H') = 1$, $det(A) = 1$.

Les coefficients w_j' utilisés pour le calcul du parcours optimal dans la nouvelle base H' peuvent s'exprimer en fonction des w_j valables dans l'ancienne base et des coefficients de la matrice A :

$$w_j' = \vec{h}_j' \ \vec{w} = \sum_i a_{ij} \vec{h}_i \vec{w} = \sum_i a_{ij} w_i$$

On en déduit:

$$\prod_j \ w_j' = \prod_j \ \sum_i \ a_{ij} w_i$$

En développant les deux expressions en dimension 2 ($N = 2$):

$$w_1' \ w_2' = \prod_{j=1}^{2} \ \sum_{i=1}^{2} \ a_{ij} w_j$$

$$= \prod_{j=1}^{2} \ (a_{1j} w_1 + a_{2j} \ w_2)$$

$$= (a_{11} w_1 + a_{21} w_2) \ (a_{12} w_1 + a_{22} w_2)$$

$$= a_{11}a_{12}w_1^2 + (a_{11}a_{22} + a_{21}a_{12})w_1w_2 + a_{21}a_{22}w_2^2$$

Tous les coefficients de A sont positifs par hypothèse. On a donc:

$$w_1' \; w_2' \geq (a_{11}a_{22} + a_{21}a_{12})w_1w_2$$

Par hypothèse aussi, on a:

$$det(A) = 1 = a_{11}a_{22} - a_{21}\,a_{12} \implies a_{11}a_{22} + a_{21}a_{12} = 1 + 2a_{21}a_{12}$$

$$\implies a_{11}a_{22} + a_{21}a_{12} \geq 1$$

On en déduit que

$$w_1' \; w_2' \geq w_1w_2$$

et donc que:

$$L_{H'} \geq L_H$$

L'égalité est vérifiée quand A est **diagonale** puisque L_H est le chemin le plus court qu'on peut obtenir sur l'ensemble des bases dérivables de H par un changement de base diagonale. En effet, dans ce cas:

$$w_1' \; w_2' \; \cdots \; w_N' = \prod_i a_{ii} \; w_i = \prod_i w_i$$

puisque le déterminant de A vaut 1.

Cette démonstration est généralisable pour N quelconque. Le coefficient de $w_1w_2 \cdots w_N$ est identique au déterminant de A mais sans les signes négatifs. Il est donc toujours supérieur ou égal à 1.

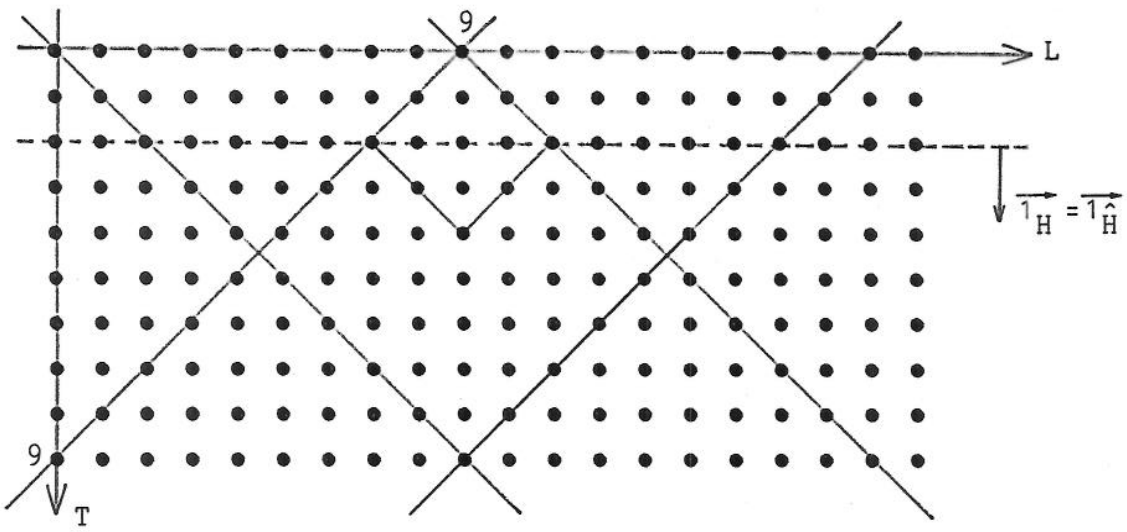
Ce résultat est illustré par les exemples des figures 6.2.5.a et 6.2.5.b où figurent deux partitionnements basés sur des directions différentes. Celui de la figure 6.2.5.a est caractérisé par:

$$P = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix} \quad H = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix} \quad \vec{1}_H = \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix} \quad \vec{w} = \begin{pmatrix} 9 \\ 0 \end{pmatrix} \quad L = 9\sqrt{2} = 12.726$$

Comme on l'a vu précédemment, ce partitionnement est optimal pour le vecteur $\vec{w}$ choisi.

Le partitionnement représenté sur la figure 6.2.5.b n'est, lui, pas optimal. C'est pourquoi deux vecteurs d'itération globale sont représentés.

$$P' = \begin{pmatrix} \frac{1}{\sqrt{5}} & \frac{1}{\sqrt{5}} \\ \frac{-3}{\sqrt{5}} & \frac{2}{\sqrt{5}} \end{pmatrix} \quad H' = \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{3}{\sqrt{5}} \\ \frac{-1}{\sqrt{5}} & \frac{1}{\sqrt{5}} \end{pmatrix} \quad \vec{1}_{H'} = \begin{pmatrix} \sqrt{5} \\ 0 \end{pmatrix} \quad \vec{w} = \begin{pmatrix} 9 \\ 0 \end{pmatrix} \quad L = 9\sqrt{5} = 20.124$$



Partitionnement optimal

- Figure 6.2.5.a -

L'optimisation à directions fixées conduit au deuxième partitionnement:

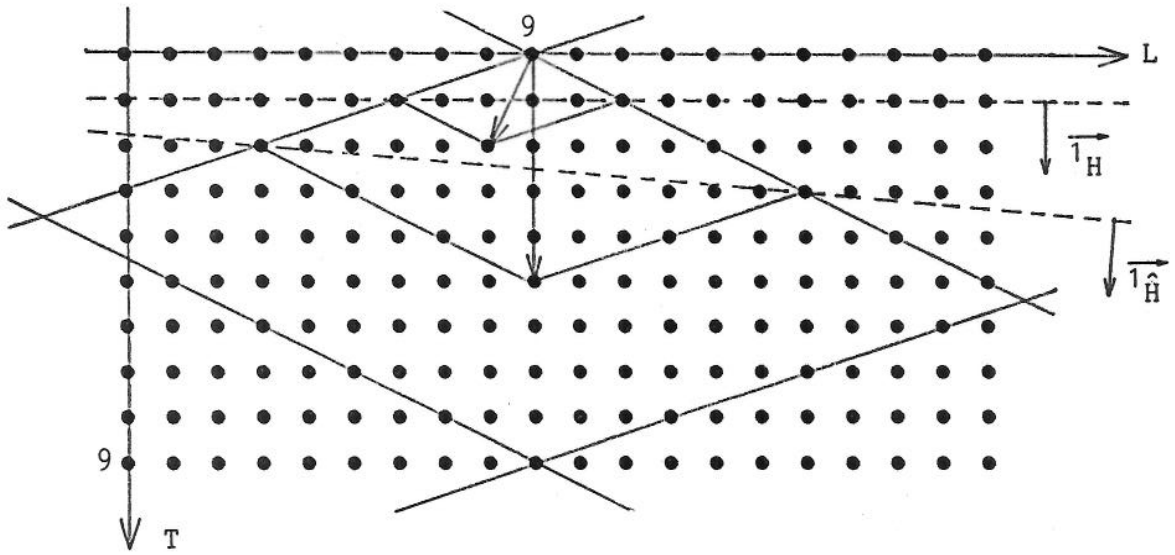
$$\hat{P}' = \frac{\sqrt{6}}{\sqrt{5}} \begin{pmatrix} \frac{1}{3} & \frac{1}{2} \\ -1 & 1 \end{pmatrix} \quad \hat{H}' = \frac{\sqrt{6}}{\sqrt{5}} \begin{pmatrix} 1 & 1 \\ -\frac{1}{2} & \frac{1}{3} \end{pmatrix} \quad \vec{1}_{\hat{H}'} = \begin{pmatrix} 2\frac{\sqrt{6}}{\sqrt{5}} \\ \frac{1}{\sqrt{5}\,\sqrt{6}} \end{pmatrix}$$

$$\vec{w} = \begin{pmatrix} 9 \\ 0 \end{pmatrix} \quad L = 18\frac{\sqrt{6}}{\sqrt{5}} = 19.714$$

On peut vérifier au passage que $\vec{1}_{\hat{P}'}$ est bien colinéaire à $\vec{w}$ (§ 6.2.4). La matrice de changement de base A vaut:

$$A = H^{-1} H' = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix} \times \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{3}{\sqrt{5}} \\ \frac{-1}{\sqrt{5}} & \frac{1}{\sqrt{5}} \end{pmatrix} = \frac{\sqrt{2}}{\sqrt{5}} \begin{pmatrix} \frac{3}{2} & 1 \\ \frac{1}{2} & 2 \end{pmatrix}$$

Ce dernier résultat montre qu'il faut utiliser les vecteurs définissant les hyperplans critiques (cf. § 2.3.4) pour construire la base de partitionnement. En dimension 2, il n'y a pas de problème car il n'y a que 2 hyperplans critiques et une seule base H optimale possible. Pour les dimensions supérieures, il n'y a pas non plus de problème quand le nombre d'hyperplans critiques est égal à N mais quand leur nombre



Optimisation de la direction d'itération

- Figure 6.2.5.b -

est supérieur nous savons seulement que la base optimale se trouve parmi les bases obtenues en choisissant N vecteurs libres sur la surface du cône des directions de partitionnement valides. Nous ne savons pas si les vecteurs optimaux sont nécessairement des rayons (i.e. des hyperplans critiques) et l'on peut même montrer que dans les espaces de dimension supérieure à 4, N rayons d'un cône ne sont pas nécessairement libres.

Intuitivement, il semble qu'il faille choisir la base H, contenue dans le cône, dont l'angle solide est maximum. Nous n'avons pas eu le temps de poursuivre nos recherches dans cette direction car de nombreux problèmes se posaient déjà en dimension 2 pour la génération de code.

6.2.6. Optimisation de la taille des supernoeuds

L'optimisation de la taille des supernoeuds se fait en dérivant la formule qui donne T en § 6.2.1 par rapport au facteur d'échelle λ. Le rapport des coefficients α_0 et α_N, que l'on ne sait malheureusement pas calculer (cf. § 6.1), va contrôler la taille de λ:

$$\frac{dT}{d\lambda} = -\frac{\alpha_0}{\lambda^2} + \alpha_N (N-1)\lambda^{N-2} = 0$$

$$\hat{\lambda} = \left(\frac{\alpha_0}{(N-1)\alpha_N} \right)^{1/N}$$

Le calcul de $\hat{T}$ montre que le terme en α_0 donne la même contribution que le terme en α_N et donc que l'approximation de 6.2.1 est valable si $\hat{\lambda} >> 1$:

$$\hat{T} = 2\alpha_N^{\frac{1}{N}} \alpha_0^{\frac{N-1}{N}} (N-1)^{\frac{1}{N}}$$

Pour ce partitionnement optimal, les processeurs passent autant de temps à se synchroniser qu'à faire des calculs.

La valeur du facteur d'échelle optimal $\hat{\lambda}$ doit respecter certaines contraintes. Il doit être suffisamment grand pour que $\hat{\lambda}P$ soit une matrice à coefficients entiers (cela implique que le déterminant est supérieur ou égal à 1 et donc qu'il y a au moins un noeud par supernoeud) et il doit être suffisamment faible pour que $\hat{\lambda}P$ ne couvre pas le domaine tout entier. Dans le premier cas, il faut retourner à la méthode hyperplane. Dans le deuxième, il faut exécuter le code séquentiellement. Il reste un troisième cas où les supernoeuds ne permettent pas d'adapter le programme à la machine cible. C'est celui où le nombre maximum de supernoeuds par front est inférieur au nombre de processeurs élémentaires.

6.2.7. Calcul du nombre maximum de processeurs utilisables

Le calcul du nombre maximum de processeurs utilisables simultanément s'effectue d'une manière similaire à celui du volume de données à conserver en cache (cf. § 4.3.1.) à partir du vecteur $\vec{w}$. Comme dans cette section, nous allons nous limiter au cas bidimensionnel.

Soit $\vec{f}_1$, $\vec{f}_2$, $\vec{f}_3$ et $\vec{f}_4$ les quatre vecteurs qui délimitent le domaine d'itération. Le plus grand front va être perpendiculaire à la direction d'itération et être de la forme:

$$\vec{w} = \mu \vec{f}_i + \vec{f}_{i+1} \qquad \mu \in [0..1] \qquad \vec{w}\ \vec{1}_H = 0$$

ou bien:

$$\vec{w} = \vec{f}_i + \nu \vec{f}_{i+1} \qquad \nu \in [0..1] \qquad \vec{w}\ \vec{1}_H = 0$$

L'indice $i+1$ est pris modulo 4.

Les deux formes sont équivalentes puisqu'on trouve $\mu = \nu^{-1}$. Il suffit donc de faire les calculs avec une des deux formes et sous la condition que le coefficient soit positif, puis de choisir l'autre forme s'il est plus grand que 1.

L'occupation d'un supernoeud sur un front est donné par $\vec{p}_2 - \vec{p}_1$. Le nombre de supernoeuds maximum sur un front, i.e. le degré de parallélisme au niveau des supernoeuds, est égal à la valeur absolue de la constante K qui donne le rapport entre $\vec{w}$ et $\vec{p}_2 - \vec{p}_1$.

$$\vec{w} = K\ (\vec{p}_2 - \vec{p}_1)$$

Exemple

Nous allons reprendre l'ensemble des étapes d'optimisation qui se terminent par le calcul de K sur le cas particulier suivant:

```
DO I = 1, 13
  DO J = I/3+1, 28-I/2
    A(I,J) = (A(I,J)+A(I-1,J)+A(I,J-1))/3.
```

Calcul du degré de parallélisme

La première étape consiste à rechercher le cône de dépendance et à en déduire les deux hyperplans critiques. En dimension 2 et avec ce type de références on obtient immédiatement:

$$D = I \qquad H = I \qquad P = I$$

où I est la matrice identité.

Nous supposons que le processeur cible est scalaire et qu'on peut donc optimiser les deux directions de partitionnement. Il faut commencer par recenser les vecteurs critiques du domaine d'itération, représenté en figure 6.2.7, et choisir celui d'entre eux qui va être réellement critique. Pour accélérer la présentation, nous limitons W aux vecteurs qui semblent être des candidats possibles au vu des dépendances:

$$W = \left\{ \vec{w}_1, \vec{w}_2, \vec{w}_3 \right\} = \left\{ \begin{pmatrix} 12 \\ 4 \end{pmatrix}, \begin{pmatrix} 12 \\ 21 \end{pmatrix}, \begin{pmatrix} 0 \\ 27 \end{pmatrix} \right\}$$

La durée de l'exécution, avec λ fixé à 1, est égale à la longueur du chemin critique L qui est donnée en dimension 2 par la formule:

$$L = 2\sqrt{w_1 w_2} \qquad w_1 = \vec{w}\ \vec{h}_1 \qquad w_2 = \vec{w}\ \vec{h}_2$$

qui a été démontrée en § 6.2.4. Dans ce cas, on obtient:

$$\begin{aligned} \vec{w}_1 \rightarrow L &= 8\sqrt{3} = 13.856 \\ \vec{w}_2 \rightarrow L &= 12\sqrt{7} = 31.748 \\ \vec{w}_3 \rightarrow L &= \perp \end{aligned}$$

La valeur de la longueur est indéfinie pour $\vec{w}_3$ parce que la direction d'itération optimale correspondante, qui est colinéaire à $\vec{w}_3$ dans ce cas particulier, est impossible. Elle viole la contrainte $\vec{h}\ \vec{d} > 0$ puisque

$\vec{w}_3 \, \vec{d}_1 = 0.$

On en conclut donc que le vecteur critique est $\vec{w}_2$ et que la diagonale des supernoeuds doit être colinéaire à $\vec{w}_2$. On aurait pu aussi calculer $\hat{H}$ et en déduire $\hat{P}$ (cf. § 6.2.4). Quoi qu'il en soit, les deux coefficients de $\vec{w}_2$, 12 et 21, peuvent être réduits et donnent comme solution entière pour P^I la matrice:

$$P^I = \begin{pmatrix} 4 & 0 \\ 0 & 7 \end{pmatrix}$$

Si jamais les calculs conduisent à une matrice P^I dont la surface est trop grande, il est possible d'en faire une approximation de la pente optimale au moyen des séries de Farey[3] . Supposons que l'on veuille approximer 4/7 pour pouvoir avoir des supernoeuds plus petits. La série de Farey s'obtient en réduisant des intervalles d'approximation. Le premier intervalle est donné par les fractions 0/1 et 1/1 et les suivants s'obtiennent en additionnant les numérateurs et dénominateurs des bornes de l'intervalle précédent pour déterminer une nouvelle borne et en choisissant celui des deux nouveaux intervalles qui contient la fraction à approximer. Dans notre cas, la série est:

$$(0/1,1/1) \rightarrow (1/2,1/1) \rightarrow (1/2,2/3) \rightarrow (1/2,3/5) \rightarrow 4/7$$

En estimant que $3/5 = 0.600$ est une approximation correcte de $4/7 = 0.571$, qui diminue la taille du plus petit supernoeud presque par deux, nous avons:

$$P^I = \begin{pmatrix} 3 & 0 \\ 0 & 5 \end{pmatrix} \quad P = \frac{1}{\sqrt{15}} \begin{pmatrix} 3 & 0 \\ 0 & 5 \end{pmatrix} \quad H = \frac{1}{\sqrt{15}} \begin{pmatrix} 5 & 0 \\ 0 & 3 \end{pmatrix}$$

et le temps d'exécution est allongé d'un facteur $\sqrt{5}/\sqrt{3} = 1.290$:

$$L = 12\sqrt{7}\left(\frac{\sqrt{5}}{\sqrt{3}}\right) = 40.975$$

Les quatre vecteurs qui définissent les frontières du domaine d'itération sont:

$$\vec{f}_1 = \begin{pmatrix} 12 \\ 4 \end{pmatrix} \quad \vec{f}_2 = \begin{pmatrix} 0 \\ 17 \end{pmatrix} \quad \vec{f}_3 = \begin{pmatrix} -12 \\ 6 \end{pmatrix} \quad \vec{f}_4 = \begin{pmatrix} 0 \\ -27 \end{pmatrix}$$

à une légère approximation entière près, qui se voit sur la figure 6.2.7. aux deux sommets gauches du domaine d'itération. En donnant la direction d'itération par le vecteur $\vec{h}$:

$$\vec{h} = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

qui permet de faire les calculs plus simplement que $\vec{1}_H$, on obtient:

$$< \vec{f}_1 + \nu \vec{f}_2 \, . \, \vec{h} > = 0 \quad \rightarrow \quad \nu = -\frac{72}{51}$$

$$< \vec{f}_2 + \nu \vec{f}_3 \,.\, \vec{h} > = 0 \quad \rightarrow \quad \nu = \frac{51}{42}$$

$$< \vec{f}_3 + \nu \vec{f}_4 \,.\, \vec{h} > = 0 \quad \rightarrow \quad \nu = -\frac{42}{51}$$

$$< \vec{f}_4 + \nu \vec{f}_1 \,.\, \vec{h} > = 0 \quad \rightarrow \quad \nu = \frac{9}{8}$$

Les solutions 1 et 3 ne vérifient pas la condition $\nu \geq 0$ et sont éliminées. Les solutions 2 et 4 nécessitent le choix de la forme en μ parce que ν est supérieur à 1. On a donc comme fronts maximaux:

$$\vec{w} = \frac{42}{51}\vec{f}_2 + \vec{f}_3 = \begin{pmatrix} -12 \\ 20 \end{pmatrix}$$

ou bien

$$\vec{w} = \frac{8}{9}\vec{f}_4 + \vec{f}_1 = \begin{pmatrix} 12 \\ -20 \end{pmatrix}$$

qui sont équivalents.

En l'absence de moyen de connaître les coefficients α_0 et α_N, on suppose que la formule de la section 6.2.6 donne:

$$\lambda = \frac{\sqrt{\alpha_0}}{\sqrt{\alpha_N}} = \sqrt{15} = 3.872$$

Cette valeur est un peu faible pour justifier l'approximation du polynome donnant le temps par ces deux termes extrêmes, mais elle a été choisie pour obtenir un dessin lisible.

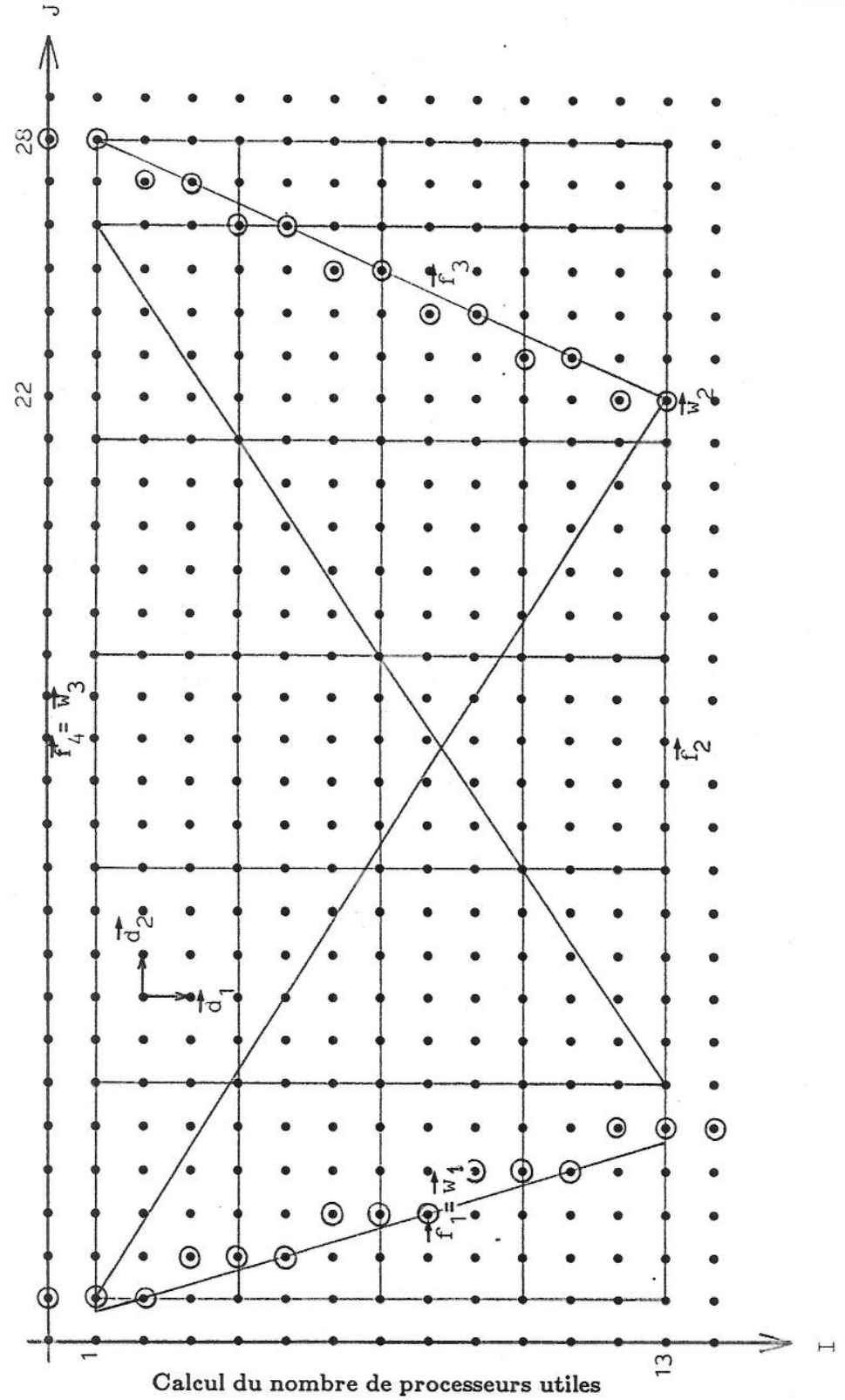
On obtient finalement:

$$\lambda(\vec{p}_2 - \vec{p}_1) = \begin{pmatrix} 3 \\ -5 \end{pmatrix}$$

ce qui donne $K = 4$ comme degré de parallélisme. Cette valeur correspond exactement à celle qu'on lit sur la figure 6.2.7. Un multiprocesseur à nombre infini de processeurs élémentaires ne donnera pas de résultats meilleurs qu'un quadri-processeur si le temps de synchronisation est 15 fois supérieur au temps d'exécution d'un corps de boucle.

6.2.8. Conclusion

L'équation gouvernant la taille des supernoeuds leur semble assez défavorable à première vue, à moins que α_0 ait une valeur très importante comme c'est le cas avec la bibliothèque de multitasking du Cray X-MP. Les gros supernoeuds sont systématiquement défavorisés parce qu'ils augmentent le volume du chemin critique en λ^{N-1} tout en ne réduisant l'overhead qu'en λ^{-1}.



Calcul du nombre de processeurs utiles

- Figure 6.2.7 -

Ce résultat négatif doit être atténué en remarquant que le coefficient α_N, qui a été optimisé en favorisant le calcul vectoriel et la réutilisation des données, n'est valable que pour des λ assez grands. Autrement dit, la formule donnant le temps n'est pas valable pour les petits λ et le temps réel est plus élevé.

Signalons qu'en dehors du contexte des supernoeuds un algorithme de recherche de l'ordonnancement linéaire optimal est proposé dans [2].

6.3. Nombre de processeurs limité

Le calcul du temps d'exécution avec un nombre infini de processeurs effectué en section 6.2 ne prend pas en compte l'ordonnancement car il est à la limite possible d'affecter un processeur par supernoeud. Garder un processeur inactif ne pénalise alors pas le temps d'exécution globale. Ce n'est plus le cas maintenant. Le but n'est plus de garantir que chaque supernoeud va être exécuté au plus tôt mais que chaque processeur va rester actif aussi continûment que possible. Lors de la génération de code (chapitre 3), nous avons opté pour un ordonnancement particulier: les fronts. C'est sur lui que nous allons étudier l'effet du choix de λ et de P.

La discussion sur le temps d'exécution d'un supernoeud (§ 6.1), basée sur les résultats des chapitres 4 et 5, reste valable.

Partant du code généré au chapitre 3, nous allons successivement étudier l'overhead global, l'overhead dû aux noeuds partiels, l'équilibrage de charge dans un front et l'évolution du degré de parallélisme des fronts. Nous en déduirons une formule approximative du temps d'exécution.

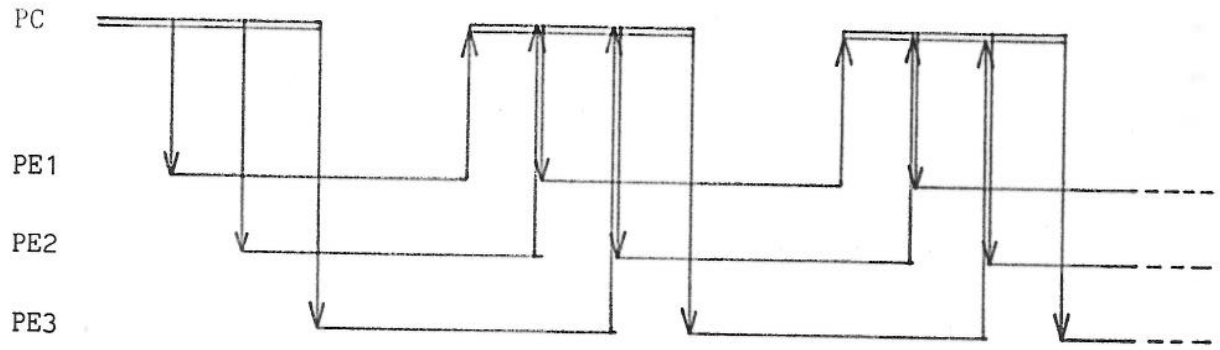
6.3.1. Overhead global

L'incrémentation du compteur de front, l'attribution d'un numéro de supernoeud à un processeur et l'incrémentation du numéro courant sont les seules opérations qu'il faut effectuer séquentiellement. Suivant les primitives de synchronisation et leur durée d'exécution, deux approches sont possibles. La première consiste à définir une tâche maîtresse lançant l'exécution de chaque supernoeud comme une sous-tâche. La tâche maîtresse peut être exécutée sur un processeur spécialisé (PC) ou être swappée de processeur élémentaire (PE) en processeur élémentaire. La seconde consiste à avoir une tâche par processeur, chaque tâche essayant d'exécuter un maximum de supernoeuds prêts. L'allocation des supernoeuds se fait via une section critique et une tâche auxiliaire assure les synchronisations globales entre fronts. Ces deux solutions sont équivalentes pour l'évaluation du temps d'exécution comme on peut le voir sur la figure 6.3.1. L'essentiel de l'overhead peut être reporté dans le temps d'exécution du supernoeud, dans le coefficient α_0.

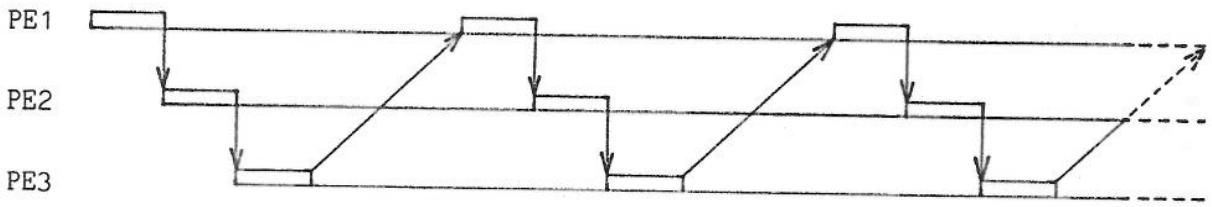
En supposant que les sections critiques n'entraînent pas d'attentes considérables, l'overhead global de parcours de la boucle sur les fronts et des boucles de balayage des supernoeuds est de la forme:

$$\sum_{k=0}^{N-1} \alpha_k \frac{V^{k/N}}{\lambda^k}$$

dans la mesure où la racine N-ième du volume du domaine d'itération peut être assimilée aux hauteurs du domaine d'itération. Quand le domaine d'itération a une forme spéciale, allongée selon une direction,



Déroulement de l'exécution avec une unité de contrôle



Auto-allocation des tâches de calcul par les processeurs

Overhead global

- Figure 6.3.1 -

cette approximation, qui nous conduit plus loin à considérer la surface du domaine d'itération comme proportionnelle à:

$$V^{\frac{N-1}{N}}$$

est fausse.

Les supernoeuds ont été définis de manière à pouvoir faire croître leur taille comme λ^N et à représenter une durée de calcul excédant largement cet overhead, quelle que soit la solution choisie, pourvu que le domaine d'itération soit assez grand.

Nous allons donc négliger l'overhead global.

6.3.2. Temps de calcul d'un supernoeud partiel

Nous n'avons étudié jusqu'à maintenant que le temps d'exécution des supernoeuds complets. Le cas des supernoeuds partiels a longtemps semblé secondaire car leur nombre, proportionnel à la surface, était d'un ordre de grandeur inférieur à celui des supernoeuds complets, proportionnel au volume.

En fait les expériences menées sur l'Alliant FX/8^4 , ont montré que le rapport des temps d'exécution d'un supernoeud partiel et d'un supernoeud complet pouvait dépasser un ordre de grandeur si la génération de code n'était pas faite soigneusement pour les supernoeuds partiels.

Nous avons montré en § 3.4 que les techniques nécessaires à la génération de code vectoriel pour les supernoeuds partiels avaient été développées, sinon toujours mises en oeuvre dans les cas les plus compliqués.

Nous considérons donc que le temps d'exécution d'un supernoeud partiel va toujours être inférieur au temps d'exécution d'un supernoeud complet et nous allons ignorer la distinction existant entre les deux.

6.3.3. Equilibrage de charge dans un front

Comme nous effectuons une resynchronisation globale à la fin de chaque front, il faudrait que le nombre de supernoeuds soit un multiple du nombre de processeurs, souvent inconnu à la compilation, et que le temps d'exécution des supernoeuds partiels qui se trouvent aux frontières ne soit pas trop court. Il faut aussi que l'hypothèse assurant que tous les supernoeuds s'exécutent dans le même temps soit vérifiée (§ 1.3.1).

Il serait envisageable de garder plusieurs versions du programme restructuré en fonction du nombre de processeurs, pour tenir compte de la charge de la machine ou de pannes partielles. Il serait aussi envisageable de *recoller* deux supernoeuds partiels de taille inférieure à la moitié de la taille standard pour n'avoir que des tâches de durées égales. Pour le moment, cela semble utopique et nous préférons faire une approximation.

Cette approximation de "charge équilibrée" et l'oubli des supernoeuds partiels sont d'autant plus valables que le nombre de supernoeuds par front est grand par rapport au nombre de processeurs puisque le temps maximal perdu par un processeur sera inférieur à la durée d'exécution d'un supernoeud. Soit P le nombre de processeurs, K le nombre de supernoeuds et V le volume de calculs à effectuer. Le temps d'exécution est donné par:

$$T = \left\lfloor \frac{K}{P} \right\rfloor \frac{V}{K} + mod\,(K,P)\,\frac{V}{K}$$

Le deuxième terme, qui représente indirectement le temps perdu par les processeurs qui n'ont plus de travail avant la fin, tend vers 0 quand K tend vers l'infini ($mod\,(K,P) < P$) tandis que le premier est

stable.

L'augmentation du nombre de supernoeuds par front nécessite une minimisation de λ et/ou le choix de directions de partitionnement aussi proches que possible des hyperplans critiques.

La diminution de la taille des supernoeuds est aussi favorable quand les temps d'exécution des supernoeuds varient comme ce serait le cas si nous étions capables de traiter les tests et les appels de procédures.

La dernière solution envisageable serait de changer l'ordonnancement global (cf. § 3.1) et d'autoriser le recouvrement de deux fronts. Nous n'avons pas eu le temps d'étudier la complexité des synchronisations qu'il faudrait alors mettre en place.

6.3.4. Evolution du degré de parallélisme des fronts

Il n'y a pas de raison pour que le degré de parallélisme soit constant d'un front à l'autre. Cela nécessite en effet que le domaine d'itération soit un cylindre coupé par deux hyperplans parallèles et que la normale à ces hyperplans soit une direction d'itération admissible vis-à-vis des dépendances. Les fronts sont alors des hyperplans parallèles aux deux extrémités du cylindre (cf. figure 6.3.5.a). En dimension 2, la condition à vérifier est équivalente à la minimisation du nombre de types de supernoeuds partiels puisque le parallélisme de la frontière et des fronts est donné par:

$$\vec{p}_2 - \vec{p}_1 = \mu \vec{f}$$

Le problème est encore compliqué quand il faut générer des vecteurs de longueur constante dans les supernoeuds ou quand on veut minimiser le nombre de types de supernoeuds le long des autres frontières!

Néanmoins, ce critère peut guider le choix de la matrice de partitionnement. Mais quand aucune solution à degré de parallélisme constant n'est possible ou intéressante à cause de la forme trop aplatie qu'elle imposerait aux supernoeuds, il faut minimiser le problème que posent les zones d'amorçage et de terminaison où le degré de parallélisme est inférieur au nombre de processeurs.

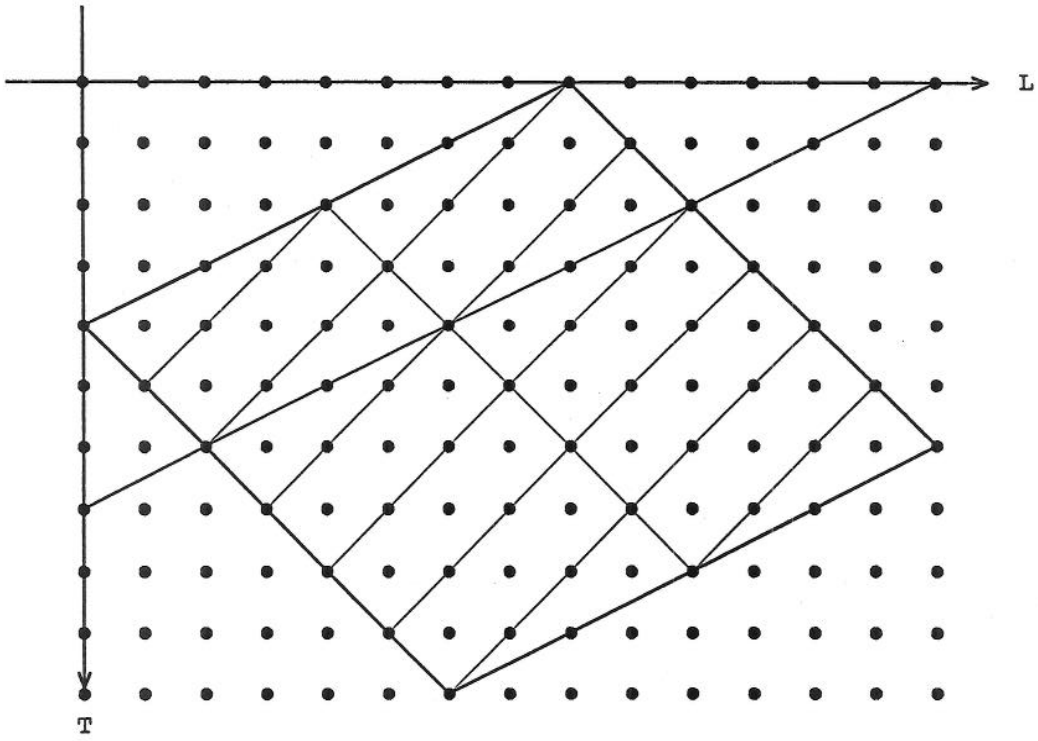
Ces deux zones sont des domaines pointus dont le volume dépend de λ, le facteur d'échelle, et de p, le nombre de processeurs. Quel que soit l'angle solide des domaines, le degré de parallélisme d va augmenter avec les numéros de front h suivant la formule:

$$d \approx O\left(h^{N-1}\right)$$

On en déduit que le nombre de fronts nécessaires pour obtenir un degré de parallélisme égal à p, le nombre de processeurs, est:

$$h \approx O\left(p^{\frac{1}{N-1}}\right)$$

Chaque front va être exécuté, totalement en parallèle, en un temps égal à celui de l'exécution d'un



Choix d'un partitionnement à degré de parallélisme constant

- Figure 6.3.5.a -

supernoeud.

Tous ces calculs sont effectués à des constantes multiplicatives près et conduisent au temps d'exécution de ces deux zones pointues:

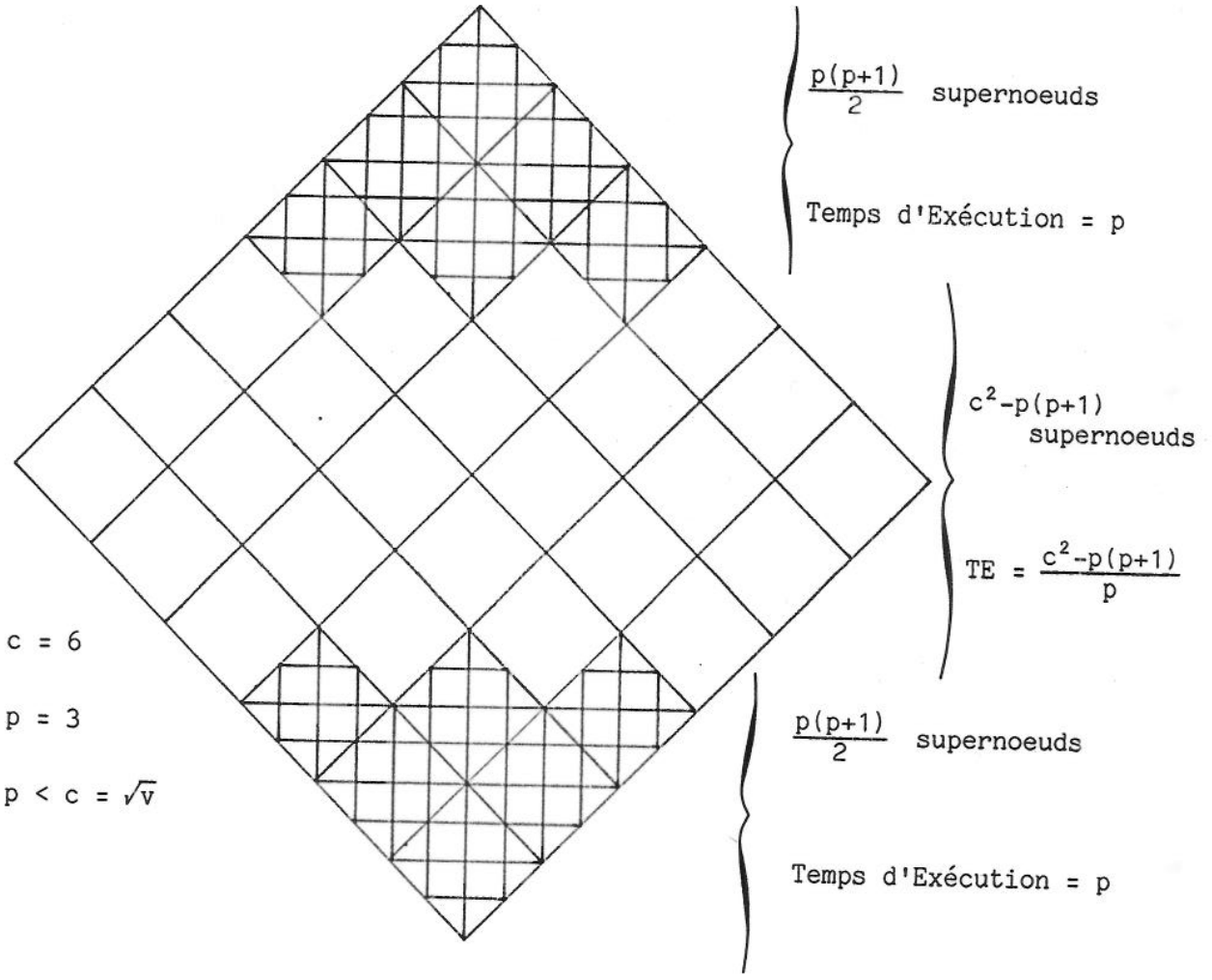
$$T = \beta\, p^{\frac{1}{N-1}} \left(\alpha_0 + \alpha_1 \lambda + \cdots + \alpha_N \lambda^N \right)$$

qui est le produit de la durée d'exécution d'un front, i.e. un supernoeud, par le nombre de fronts. La constante β s'annule quand le domaine est cylindrique et que la direction d'itération globale $\vec{I}_H$ est orthogonale, ou presque orthogonale, à deux extrémités identiques du domaine d'itération. Elle prend en compte à la fois la zone d'amorçage et la zone de terminaison qui ont le même type de forme.

Ce temps, durant lequel tous les processeurs ne peuvent pas travailler simultanément, va être d'autant plus court que la dimension du problème N est élevée et que le facteur d'échelle λ est plus faible.

Il n'est pas évident qu'il faille prendre en compte ce temps, si on néglige le déséquilibre de la charge dans les fronts qui suivent ou précèdent ces zones. Après tout, en dimension 2, le temps d'exécution réel du front suivant, qui va compter $p+1$ supernoeuds, va être deux fois supérieur au temps calculé.

Ces résultats sont illustrés sur la figure 6.3.5.b où les zones d'amorçage et de terminaison sont mises en évidence par un quadrillage. Le dessin est fait pour trois processeurs, mais les formules données sont valables en dimension 2 quand il n'y a pas de supernoeuds partiels (les frontières sont parallèles aux deux droites (hyperplans) de partitionnement. Les volumes des différentes zones sont aussi indiqués car ils vont être nécessaires à l'évaluation du temps de calcul de la zone intermédiaire.



Evaluation des temps d'exécution

- Figure 6.3.5.b -

6.3.5. Fonctionnement en régime stationnaire

Il ne reste plus qu'à trouver le temps nécessaire à l'évaluation du sous-domaine où le degré de parallélisme est supérieur au nombre de processeurs. Nous faisons l'hypothèse que la charge est équilibrée ou que nous aurons développé une synchronisation globale moins contraignante que les fronts.

Ce temps va dépendre du nombre de supernoeuds que doit exécuter chaque processeur et du temps d'exécution de chaque supernoeud. Ce temps d'exécution est toujours de la forme:

$$\alpha_0 + \alpha_1\lambda + \alpha_2\lambda^2 + \cdots + \alpha_N\lambda^N$$

tandis que le nombre de supernoeuds affectés à chaque processeur est égal au nombre total de supernoeuds présents dans le volume où ces conditions sont vérifiées. Ce volume s'obtient en retirant du volume du domaine d'itération, le volume des zones d'amorçage et de terminaison et éventuellement le volume des supernoeuds partiels. En ne prenant en compte que le volume des zones pointues parce que la génération de code vectoriel pour les supernoeuds partiels est possible, on obtient le volume de la zone intermédiaire:

$$V' = V - \delta\lambda^N p^{\frac{N}{N-1}}$$

en intégrant la hauteur des zones extrêmes, qui a été calculée dans la section précédente: le volume des zones pointues augmente comme la puissance N-ième de la hauteur, h^N, et $h = p^{\frac{1}{N-1}}$ quand les p processeurs peuvent être utilisés simultanément.

Le temps s'en déduit en divisant V' par le volume d'un supernoeud pour obtenir le nombre de supernoeuds puis en multipliant le résultat par le temps d'exécution d'un supernoeud et en divisant le tout par le nombre de processeurs:

$$T = \frac{1}{p}\,\frac{1}{\lambda^N}\left(V - \delta\lambda^N p^{\frac{N}{N-1}}\right)(\alpha_o + \alpha_1\lambda + \ldots \alpha_N\lambda^N)$$

Cette formule peut être vérifiée sur la figure 6.3.5.b où les calculs de volumes ont été effectués en nombres entiers. L'hypothèse de pleine utilisation des processeurs est vérifiée si l'ordonnancement permet les recouvrements de fronts. Le dernier (resp. premier) front de la zone d'amorçage (resp. terminaison) pourrait être inclus dans la zone intermédiaire. Cela ne changerait pas le résultat, mais les formules feraient apparaître $p-1$ au lieu de p dans le calcul de volume.

6.3.6. Approximation du temps global d'exécution

Il est maintenant possible de donner une approximation du temps global d'exécution en rassemblant tous les éléments précédents. V est toujours le volume total du domaine d'itération, N le nombre de boucles imbriquées, λ le facteur d'échelle et p le nombre de processeurs.

$$T = \frac{V - \delta\lambda^N p^{\frac{N}{N-1}}}{p\,\lambda^N}(\alpha_o + \alpha_1\lambda + \ldots \alpha_N\lambda^N)$$

$$+ \beta p^{\frac{1}{N-1}}(\alpha_o + \alpha_1\lambda + \ldots \alpha_N\lambda^N)$$

Le premier terme correspond à la zone où tous les processeurs sont utilisés en permanence. Le deuxième terme correspond aux zones pointues où tous les processeurs ne peuvent pas être utilisés. Il n'y a pas de terme pour tenir compte des noeuds partiels compte tenu de la possibilité que nous avons de générer du code vectoriel pour eux.

Les coefficients α_i reflètent à la fois le temps d'exécution et le temps d'échange. S'il y a du parallélisme entre échanges et calculs, ces coefficients ne permettront pas de rendre compte du choix *dominant calcul* ou *dominant échange* en fonction de λ. Il y aurait ajout d'un MAX.

Les coefficients β et δ dépendent beaucoup de la génération de code et du partitionnement. Les figures 6.3.7.a et 6.3.7.b montrent deux partitionnements du même domaine d'itération pour $\lambda = 2$. Dans le premier cas, on compte 24 supernoeuds complets et 61 supernoeuds partiels. De plus, les deux volumes d'amorçage et de terminaison, délimités par les fronts indiqués en pointillés, représentent une part considérable du domaine d'itération total. Dans le deuxième cas, la direction d'itération $\vec{I}_H$ est orthogonale à deux des frontières et aussi bien le balayage que le test de complétude d'un supernoeud sont effectués en nombres entiers[†]. On compte maintenant 45 noeuds complets et 28 noeuds partiels. Et les zones d'amorçage et de terminaison sont pratiquement vides. Indépendamment de λ, le choix d'un bon partitionnement va sans doute influer considérablement sur les performances.

On peut simplifier l'expression de T en regroupant les termes en β et δ:

$$T = \frac{V}{p\,\lambda^N}(\alpha_o + \alpha_1\lambda + \ldots \alpha_N\lambda^N)$$

$$+ (\beta - \delta)p^{\frac{1}{N-1}}(\alpha_o + \alpha_1\lambda + \ldots \alpha_N\lambda^N)$$

[†]Cela pose des problèmes si les boucles ne sont pas parfaitement imbriquées. Il faut alors ajouter des instructions aux noeuds se trouvant sur certaines frontières.

Sachant que δ s'obtient par intégration, il varie comme l'inverse de factoriel(N). Dans le cas de la figure 6.3.5.b, $\beta = 2$ et $\delta = 1$. Le deuxième terme va donc rester positif et augmenter avec le nombre de processeurs, ce qui est intuitivement normal: le volume des zones pointues sur lesquelles tous les processeurs ne sont pas utilisés croît. Il augmente aussi avec λ. On peut étudier la sensibilité de T aux variations de λ:

$$\frac{dT}{d\lambda} = -\frac{V}{p}\left(\frac{\alpha_0}{N\lambda^{N+1}} + \frac{\alpha_1}{(N-1)\lambda^N} + \cdots \frac{\alpha_{N-1}}{\lambda^2}\right)$$

$$+ (\beta - \delta)p^{\frac{1}{N-1}}\left(\alpha_1 + 2\alpha_2\lambda + \cdots + N\alpha_N\lambda^{N-1}\right)$$

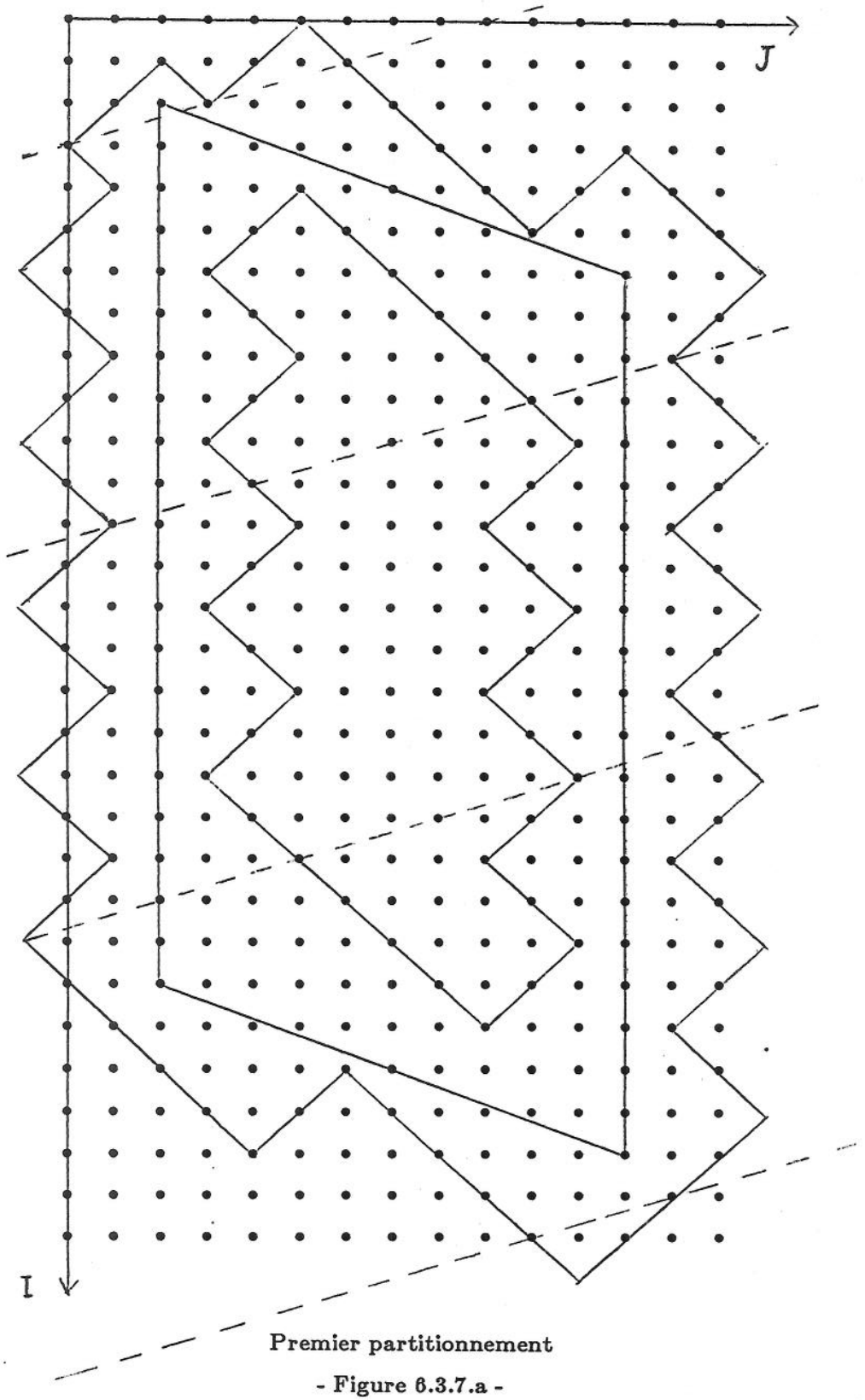
Le seul terme qui provoque un accroissement de T quand λ augmente est celui qui correspond aux zones d'amorçage et de terminaison, psuique c'est le seul terme positif (la dérivée du temps d'exécution d'un supernoeud par rapport à λ est physiquement positive). Le volume correspondant aux noeuds partiels augmente aussi, mais ces derniers n'ont pas été pris en compte explicitement.

Comme **aucun** des coefficients n'est calculable sans connaître les détails du générateur de code utilisé, une optimisation en λ n'est pas réaliste. Mais si λ peut être très grand, si $V >> \lambda^N$ malgré le grand λ et si γ est très faible, le temps d'exécution peut être approché par le terme:

$$T \approx \frac{\alpha_N V}{p}$$

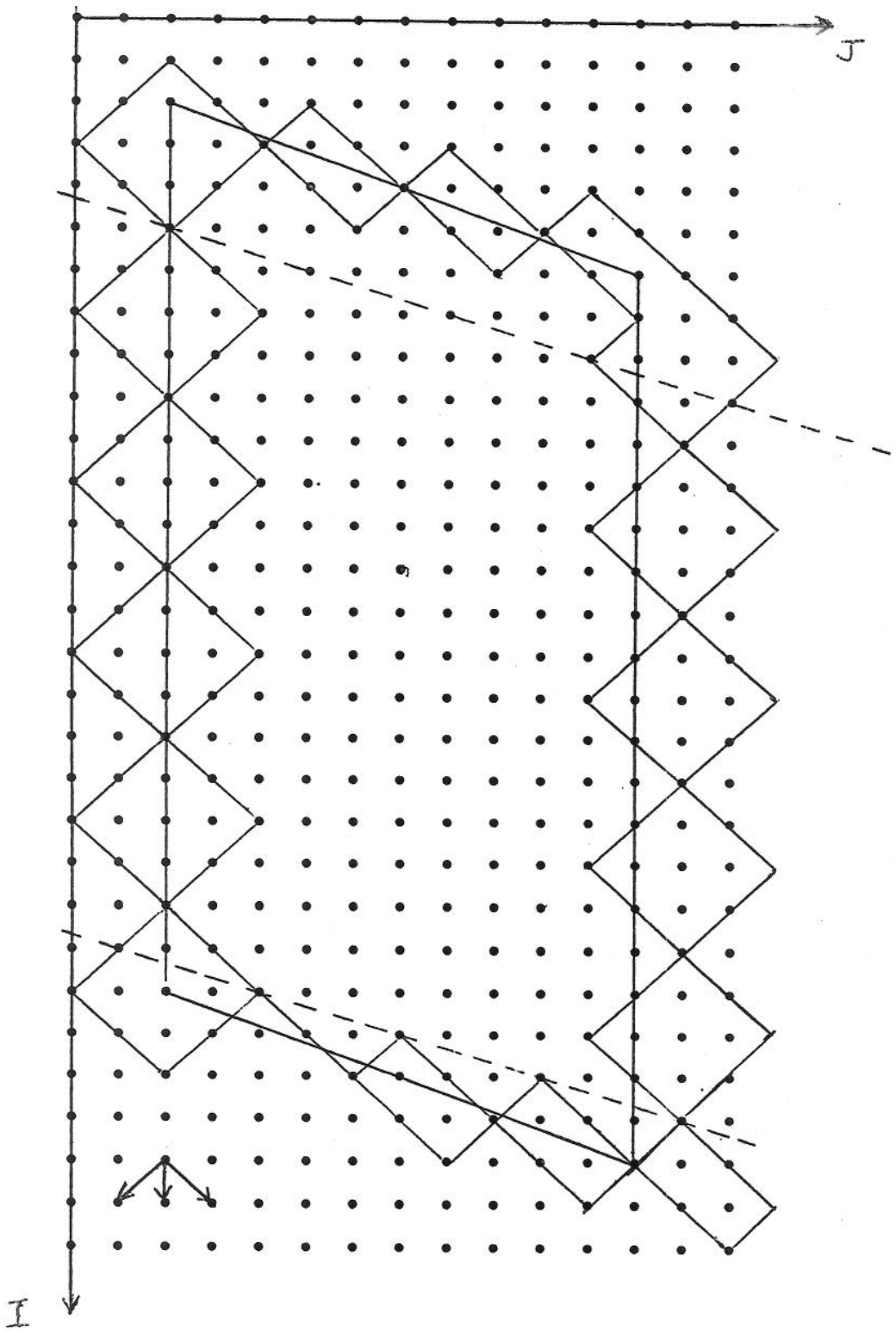
L'optimisation apportée par les supernoeuds est donc cachée dans le calcul de α_N. La réduction des échanges et leur passage dans les coefficients α_{N-1}, α_{N-2}, etc... permet de réduire le coefficient α_N qu'on obtiendrait en appliquant un tronçonnage après la méthode hyperplane.

L'optimisation peut aussi se faire par essais successifs, surtout dans le cas des processeurs élémentaires vectoriels dont on ne modélise pas la sensibilité aux vecteurs de longueurs variables.



Premier partitionnement

- Figure 6.3.7.a -



Deuxième partitionnement

- Figure 6.3.7.b -

6.4. Critères de choix

L'ensemble des critères de choix que nous avons rencontrés dans les chapitres 3, 4 ,5 et 6 est listé dans la colonne de gauche de la table 6.4. suivant les trois rubriques: temps d'exécution d'un supernoeud, temps d'exécution de l'ensemble des supernoeuds, problèmes de génération de code.

L'influence sur chaque critère des paramètres qu'il faut déterminer est indiquée par une lettre X ou par une flèche. Les paramètres sont:

- la direction d'itération $\vec{h}$ locale à un supernoeud (chapitre 5);
- la direction d'itération globale $\vec{1}_H$ (§ 2.4.2);
- les hyperplans de partitionnement, représentés par H;
- le facteur d'échelle λ (§ 2.5);
- l'origine du partitionnement $\vec{p}_o$ (cf. § 3.2.3).

De plus, trois données du problèmes, l'ensemble des dépendances D, le domaine d'itération A et l'ensemble des références, symbolisé par $\vec{\alpha}$, contraignent les choix.

Nous allons commenter cette table récapitulative, un peu sèche en elle-même, en suivant les cinq colonnes des paramètres, plutôt que les lignes, car ce sont eux qu'il faut déterminer.

Le choix de $\vec{h}$ est essentiellement un compromis à effectuer entre la difficulté de la génération de code et la possibilité d'effectuer une exécution vectorielle. La minimisation du volume de cache nécessaire (à λ fixé) est limitée dès qu'il y a plusieurs vecteurs d'autodépendance non colinéaires, d'où la présence du facteur $\vec{\alpha}$ dans la table, mais non négligeable car il arrive souvent qu'il n'y ait qu'un tel vecteur. Certaines machines sont aussi très sensibles au pas avec lequel les données sont accédées: à cause des blocs dans les caches, des pages quand il y a une mémoire virtuelle, ou du partitionnement de la mémoire en bancs. Le choix de $\vec{h}$ est contraint par les dépendances D.

Le choix de la direction d'itération globale $\vec{1}_H$ a une influence sur le volume des données transférées au niveau du supernoeud, avec les mêmes restrictions que pour $\vec{h}$. $\vec{1}_H$ peut aussi être choisi de manière à diminuer le nombre de fronts en prenant la direction optimale avec un nombre infini de processeurs ou, pourvu que ce ne soit pas incompatible avec D, de manière à avoir un degré de parallélisme constant. Enfin, le choix de $\vec{1}_H$ conditionne le nombre de types de supernoeuds partiels qu'on rencontre le long des frontières (cf. § 3.2.2).

Les directions de partitionnement (H) peuvent être choisies de manière à avoir des vecteurs de longueur constante, ce qui accélère probablement l'exécution des supernoeuds sur les machines à fort $n_{1/2}$ (cf. § 5.1.3) et simplifie la génération de code (cf. § 3.3.3). La recherche d'un partitionnement aussi proche que possible des contraintes introduites par les dépendances permet d'augmenter le nombre de supernoeuds par front tout en diminuant le nombre de fronts (§ 6.4.4). La matrice de partitionnement H

doit respecter deux contraintes. Elle doit définir un partitionnement valide (cf § 2.4.1) et, dans le cas des processeurs vectoriels à fort $n_{1/2}$ qui n'a pas été pris en compte dans ce chapitre, la matrice P qui est dérivée de H doit contenir au moins un vecteur dont la direction joint des noeuds qui peuvent être exécutés en parallèle.

Le choix du facteur d'échelle λ est à la fois gouverné par des objectifs contradictoires et soumis à des contraintes nombreuses. Commençons par les objectifs. Pour diminuer l'overhead de balayage des supernoeuds, y compris les supernoeuds vides, pour réutiliser au maximum les données (i.e. augmenter le rapport $\alpha_N \lambda / \alpha_{N-1}$) et, dans certains cas, pour améliorer les performances vectorielles des processeurs élémentaires, il faut un grand λ. Mais pour limiter le problème de l'équilibrage de charge sur les fronts et la taille des zones où tous les processeurs ne peuvent pas travailler simultanément, il faut plutôt choisir un petit λ. Enfin, l'utilisation efficace des registres vectoriels peut conduire à choisir les λ parmi les multiples d'une constante (32 ou 64 par exemple) plutôt que parmi les entiers naturels.

Le facteur d'échelle λ est aussi soumis à plusieurs contraintes. Tout d'abord, les supernoeuds ne doivent pas utiliser plus d'espace de copie que ne peut en contenir la mémoire rapide. Deuxièmement, ils doivent toujours contenir un nombre entier de noeuds. Comme nous avons choisi pour simplifier la génération de code de les prendre tous identiques, cela veut dire que le plus petit supernoeud, à directions de partitionnement fixées, est donné par la matrice P^I (cf. § 2.4.3). Si l'optimisation conduit à des supernoeuds ne contenant qu'un seul noeud, la notion de supernoeud devient inutile et l'on retrouve une méthode hyperplane (cf. § 8.1). Troisièmement, il n'est pas possible d'avoir un supernoeud dont la taille est supérieure à celle du domaine d'itération. Cela signifierait que les overheads de contrôle, et éventuellement de transfert, seraient supérieurs au gain qu'on peut attendre de la mise en parallèle des calculs des supernoeuds et donc que les supernoeuds sont inutiles: l'exécution séquentielle est alors l'exécution la plus efficace qu'on puisse trouver.

Enfin, le choix de l'origine du partitionnement, $\vec{p}_o$, ne semble pas critique, bien que la diminution du nombre de supernoeuds partiels soit agréable quand la programmation est effectuée manuellement et que la réduction du volume occupé par des supernoeuds partiels conduise probablement à une amélioration des performances.

L'architecture de la machine joue un rôle lors de la génération de code et lors de l'attribution de poids aux différents critères. Il ne faut pas oublier de vérifier que le nombre de processeurs réels est inférieur au nombre de processeurs qu'on trouve avec l'hypothèse du nombre infini de processeurs, dans la mesure où l'on a pu évaluer les coefficients α_0, ... α_N qui donnent le temps d'exécution d'un supernoeud.

		$\vec{h}$	$\vec{1}_H$	H	λ	$\vec{p}_o$	D	A	$\vec{\alpha}$
1	Volume du cache ou de la mémoire locale (λ_{max})	X			↓				X
2	Exécution vectorielle (α_N)	X			X		X		
3	Vecteur de longueur constante (α_N)			X					
4	Transferts de données (α_N, α_{N-1}, $\cdots$)		X	X	↑		X		X
5	Accès aux tableaux avec un pas de un (bancs, blocs cache, page virtuelle)	X							X
6	Constance du degré de parallélisme (β, δ)		X					X	
7	Nombre de fronts		X	X	↑			X	
8	Nombre de supernoeuds par front (supérieur au nombre de processeurs)				↓				
9	Durée d'exécution variable (IFs, CALLs)				↓				
10	Overhead de balayage				↑				
12	Nombre de supernoeuds partiels		X		↑	X			
13	Nombre de types de supernoeuds partiels		X			X		X	
14	Code d'un supernoeud	X							

Table 6.4: Récapitulatif des tous les critères, choix et contraintes

References

1. D.A. Calahan, "Task Granularity Studies on a Many-Processor CRAY X-MP," *Parallel Computing* **2**(2), pp.109-118, North-Holland (1985).

2. J. A. B. Fortes and F. Parisi-Presicce, "Optimal Linear Schedules for the Parallel Execution of Algorithms," *Int'l Conference on Parallel Processing*, pp.322-329 (Aug. 1984).

3. G. H. Hardy and E. M. Wright, *The Theory of Numbers*, Clarendon Press, Oxford (1959).

4. F. Irigoin and R. Triolet, "Supernodes and Alliant FX/8 Minisupercomputer," Tech. Rep. ENSMP-CAI-86-081, Ecole des Mines de Paris (Aug. 1986).

5. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

7. EXEMPLE

Nous allons maintenant appliquer l'ensemble des techniques et des résultats présentés dans les chapitres 2 à 6 à l'exemple traité informellement en § 1.2. La machine cible est l'Alliant FX/8 (cf. § 1.1.1.3) parce que c'est le seul multiprocesseur vectoriel auquel nous avons eu accès et pour lequel nous avons quelques résultats expérimentaux. Elle a de plus l'avantage pour nous d'être une machine très sensible à la localité des accès mémoire[1] .

7.1. Programme à restructurer

Plusieurs versions du programme ont été données dans la section 1.2 pour montrer dans quelle mesure il était possible d'éliminer automatiquement les trucs et astuces qu'utilisent les programmeurs pour économiser l'espace mémoire. Nous supposerons donc qu'il nous est possible de repartir du programme le plus proche de l'équation:

```
DO T = 1, TFIN
  DO L = 1, LNGBARRE
    BARRE(T,L) = (BARRE(T-1,L-1)+BARRE(T-1,L)+BARRE(T-1,L+1))/3.
```

7.2. Partitionnements possibles

Le calcul des dépendances est particulièrement simple dans ce cas. Nous allons néanmoins poser le système d'équations qui permet de trouver les vecteurs de dépendances relatifs aux trois datadépendances qui sont présentes. La référence BARRE(T,L) correspond à la définition d'une valeur à l'itération (T,L) et la référence BARRE(T'–1,L'–1) à une utilisation à l'itération (T',L'). Le vecteur de datadépendance est défini par:

$$\vec{d}_1 = \begin{pmatrix} T' \\ L' \end{pmatrix} - \begin{pmatrix} T \\ L \end{pmatrix}$$

selon la définition donnée en § 1.4.4. Pour qu'il y ait dépendance, il faut aussi que les deux références concernent le même élément de tableau:

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

On obtient donc pour ce premier couple de références:

$$\vec{d}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

et de la même manière avec les deux autres références utilisant BARRE:

$$\vec{d}_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \vec{d}_3 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Ces trois vecteurs sont bien lexicopositifs et ce sont eux qui génèrent le cône de dépendance par toutes leurs combinaisons linéaires à coefficients positifs. On peut aussi vérifier que le cône de dépendance est de dimension 2 comme l'espace d'itération (§ 2.2.1) et que le graphe de dépendance est connexe (§ 2.2.3). En effet:

$$det\,(\vec{d}_1, \vec{d}_2) = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix} = -1$$

ce qui montre d'une part que $\vec{d}_1$ et $\vec{d}_2$ engendrent un cône de dimension 2 puisque le déterminant est non nul et d'autre part que la maille formée par les vecteurs $\vec{d}_1$ et $\vec{d}_2$ couvre tous les points entiers de l'espace d'itération puisque la valeur absolue du déterminant vaut 1. Si $\vec{d}_3$ avait été choisi à la place de $\vec{d}_2$, ce deuxième résultat n'aurait pas été acquis.

Comme le problème est de dimension 2, les hyperplans critiques peuvent être trouvés sans recourir au calcul des rayons et du sommet du cône de dépendance grâce à la méthode décrite en § 2.3.4.3. Il suffit de calculer les angles directeurs a_i des vecteurs de dépendances et de choisir les deux angles extrêmes:

$$a_1 = \frac{\pi}{4} \qquad a_2 = 0 \qquad a_3 = -\frac{\pi}{4}$$

La notion d'angle *extrême* a un sens parce que tous les vecteurs se trouvent dans un demi-plan. Le vecteur $\vec{d}_2$ est donc éliminé et l'on obtient la matrice $\hat{H}$ définissant les hyperplans critiques, dont on a montré en § 6.2.5 qu'ils constituaient les meilleurs partitionnements possibles, en prenant un vecteur orthogonal à $\vec{d}_1$ et un vecteur orthogonal à $\vec{d}_3$ puis en les normalisant et en les ordonnant de sorte que $det\,(\hat{H}) = 1$. Cela donne:

$$\hat{H} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}$$

La connaissance de $\hat{H}$ nous permet de trouver la meilleure direction d'itération (§ 6.1) dans le cas où le nombre de processeurs est infini et où les processeurs élémentaires sont scalaires, ce qui n'est pas le cas sur l'Alliant FX/8. $\hat{H}$ peut néanmoins être utile car les unités vectorielles de cette machine ont un faible $n_{1/2}$ (§ 5.1.2). Les synchronisations extrêmement efficaces de l'Alliant donnent qualitativement une valeur faible à la constante α_0 et conduisent donc à une petite valeur de λ (§ 6.2.6), qui permettrait d'utiliser de nombreux processeurs sur de petits domaines d'itération si les processeurs élémentaires de

l'Alliant FX/8 n'étaient vectoriels.

7.3. Calcul des transferts de données

Les calculs des mouvements de données sont effectués à partir des vecteurs d'autodépendance attachés à chaque famille de références (§ 1.4.3, § 3.5.2 et 2). Dans le cas présent, nous avons une seule famille de références, caractérisée par les formes linéaires:

$$\vec{\alpha}_1 = \begin{pmatrix}1\\0\end{pmatrix} \qquad \vec{\alpha}_2 = \begin{pmatrix}0\\1\end{pmatrix}$$

Cette famille de référence est de dimension 2 car $det\,(\vec{\alpha}_1, \vec{\alpha}_2) = 1$ et les vecteurs $\vec{\alpha}_1$ et $\vec{\alpha}_2$ ne nécessitent pas de réduction par leurs PGCDs. Comme la dimension de l'espace d'itération est aussi de 2, il n'est pas possible de diminuer le nombre de transferts effectués entre les processeurs et la mémoire lente (4.2.1) à moins d'essayer de profiter de la présence de quatre références dans la famille pour réutiliser des données conservées dans une mémoire locale ou dans des registres vectoriels (§ 4.6.1).

L'exploitation de cette idée nécessite la présence dans P d'une direction parallèle et la présence d'un vecteur de datadépendance ou d'inputdépendance compatible avec la direction parallèle et les dépendances. Il n'est pas nécessaire d'appliquer la méthode hyperplane (§ 5.1.3) puisque la boucle sur L est parallèle. Les six vecteurs d'inputdépendance sont colinéaires à cette direction parce que leurs composantes en T sont nulles:

$$\vec{d}_{12} = \vec{d}_{23} = -\vec{d}_{21} = -\vec{d}_{32} = \begin{pmatrix}0\\1\end{pmatrix} \qquad \vec{d}_{13} = -\vec{d}_{31} = \begin{pmatrix}0\\2\end{pmatrix}$$

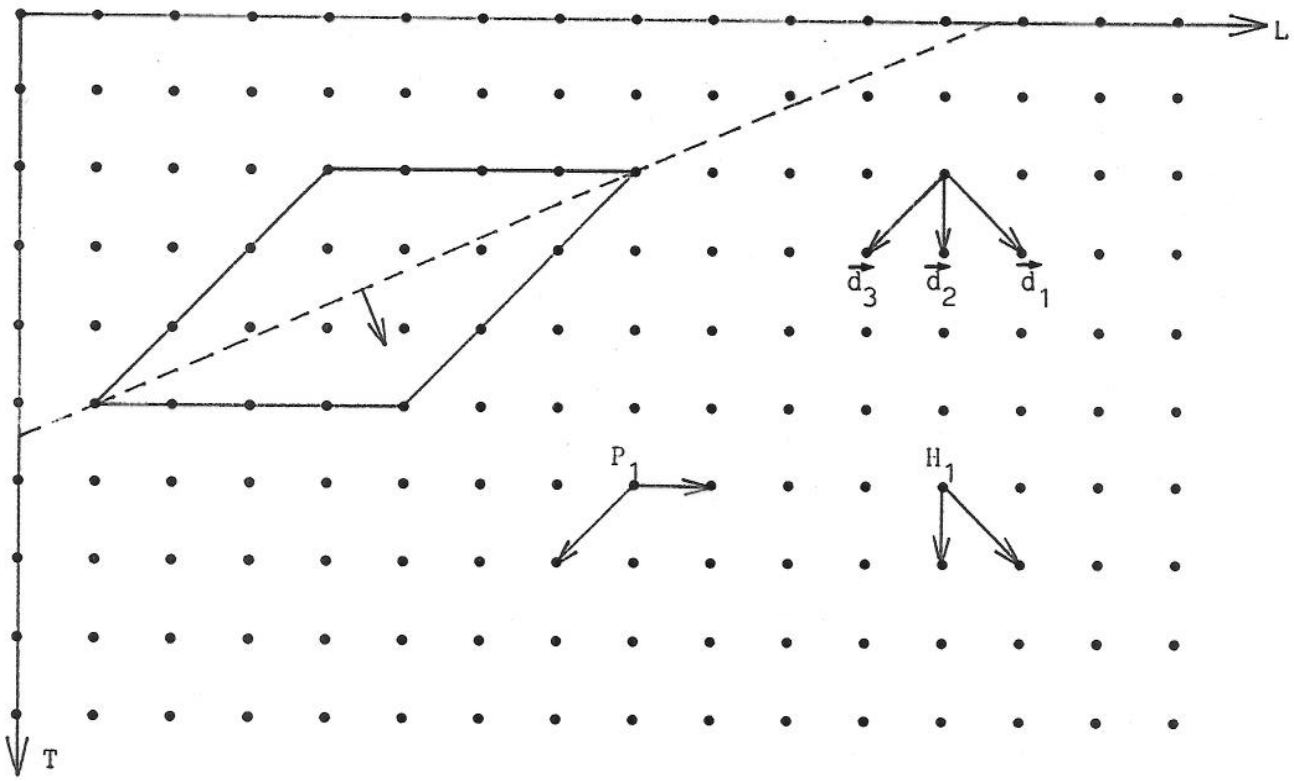
Il reste donc les trois vecteurs de datadépendance. $\vec{d}_2$ est inutilisable comme vecteur de P car sa normale, qui devrait définir un hyperplan valide, n'appartient pas au cône positif de H (§ 2.3.5 et § 2.4.2).

Les deux partitionnements possibles sont donc définis par:

$$H_1 = \begin{pmatrix}1 & 1\\0 & 1\end{pmatrix} \quad P_1 = \begin{pmatrix}1 & 0\\-1 & 1\end{pmatrix} \qquad H_2 = \begin{pmatrix}1 & 1\\-1 & 0\end{pmatrix} \quad P_2 = \begin{pmatrix}1 & 0\\1 & 1\end{pmatrix}$$

Ces deux partitionnements, représentés figure 7.3.a et 7.3.b, avec deux types de supernoeuds qui peuvent en découler, permettent de n'effectuer que deux *loads* et un *store* par itération tout en réutilisant l'essentiel des données chargées dans le cache pendant le premier *load* par la suite. Le tronçonnage de la boucle sur L qu'effectue automatiquement le restructureur VAST de l'Alliant nécessite trois *loads*.

Pour profiter davantage du cache de l'Alliant FX/8, il faut introduire une autodépendance en diminuant l'espace de stockage. Cela est possible si seules les valeurs finales, BARRE(TFIN:LNGBARRE), sont utiles dans la suite du programme. Nous reviendrons sur cette possibilité en § 7.10.



Premier partitionnement à vecteurs de longueur constante

- Figure 7.3.a -

7.4. Choix d'une direction d'itération globale

Il faut commencer par recenser les vecteurs critiques du domaine d'itération:

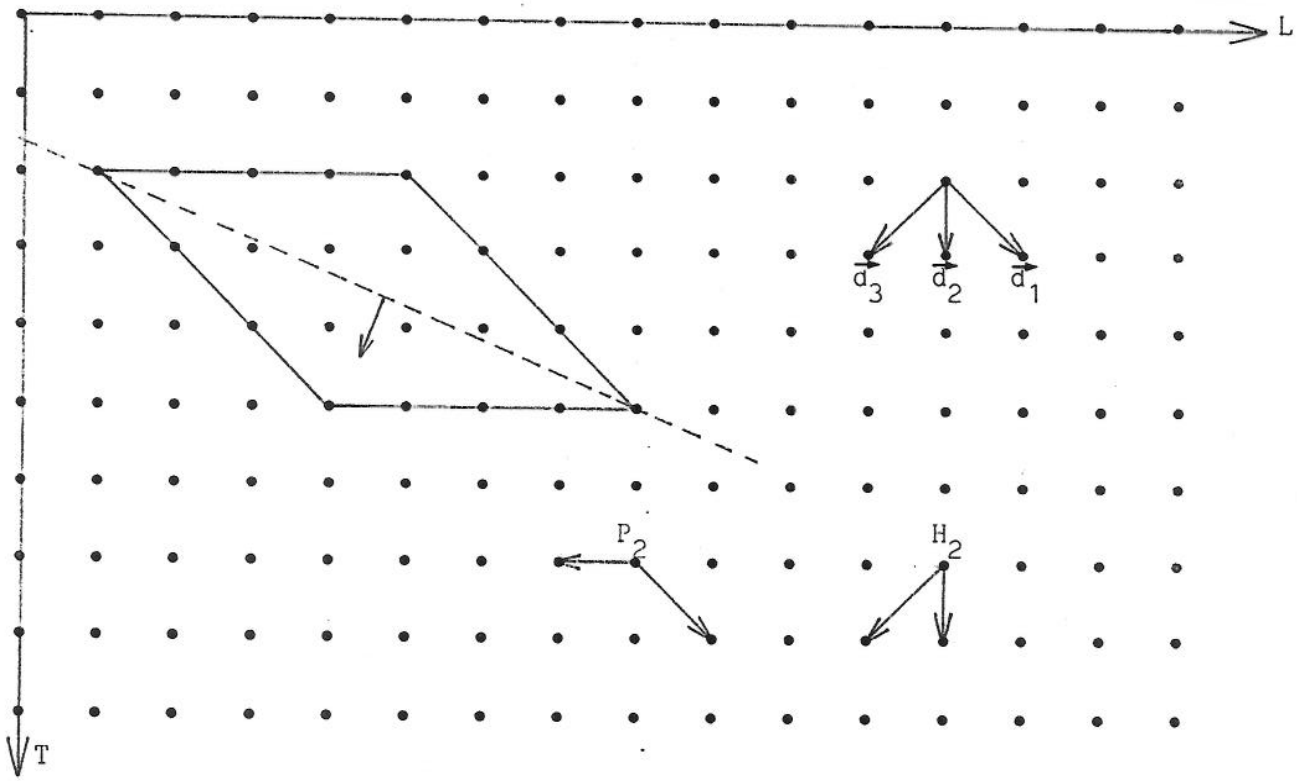
$$W = \{\vec{w}_1, \vec{w}_2, \vec{w}_3\} = \left\{ \begin{pmatrix} \text{TFIN-1} \\ 0 \end{pmatrix}, \begin{pmatrix} \text{TFIN-1} \\ \text{LNGBARRE-1} \end{pmatrix}, \begin{pmatrix} \text{TFIN-1} \\ \text{1-LNGBARRE} \end{pmatrix}, \right\}$$

Pour simplifier la suite des calculs, on incorpore la constante -1 dans TFIN et LNGBARRE. En utilisant d'abord les directions de partitionnement optimales qui sont définies par $\hat{H}$, on obtient:

$$L_{\vec{w}_1} = \sqrt{2}\ \text{TFIN} \qquad L_{\vec{w}_2} = L_{\vec{w}_3} = \sqrt{2}\ \sqrt{\text{TFIN}^2 - \text{LNGBARRE}^2}$$

Les vecteurs $\vec{w}_2$ et $\vec{w}_3$ ne sont critiques que si LNGBARRE < TFIN, c'est-à-dire si chaque point du résultat final BARRE(TFIN,1:LNGBARRE) dépend de chaque point des conditions initiales BARRE(0,0:LNGBARRE+1), et, dans ce cas, ils donnent des durées d'exécution plus courtes que $L_{\vec{w}_1}$.

Le calcul de λ_1 et de λ_2 (§ 6.2.4) donne:



Deuxième partitionnement à vecteurs de longueur constante

- Figure 7.3.b -

$$\lambda_1 = \lambda_2 = 1$$

La base $\hat{H}$ définit donc le meilleur partionnement, à un facteur d'échelle près.

La direction d'itération qu'on obtient est parallèle à un des deux axes du domaine d'itération. Cela nous garantit un degré de parallélisme constant et un bon équilibrage de la charge (§ 6.3.4). De plus le nombre de types de supernoeuds partiels va être faible car les deux vecteurs définissant les frontières s'obtiennent par des combinaisons linéaires à faibles coefficients des vecteurs de la base de partitionnement $\hat{P} = \hat{H}$ (§ 3.2.2):

$$\vec{f}_1 = \begin{pmatrix} \text{TFIN-1} \\ 0 \end{pmatrix} = \mu\,(\vec{p}_1 + \vec{p}_2) \qquad \vec{f}_2 = \begin{pmatrix} 0 \\ \text{LNGBARRE-1} \end{pmatrix} = \nu\,(\vec{p}_1 - \vec{p}_2)$$

Ces coefficients réduits par leur PGCD donnent deux types de supernoeuds partiels pour chacune des quatre frontières, soit 8 au total. Il est possible de réduire ce nombre en mettant l'origine $\vec{p}_o$ du partitionnement (§ 3.2.3) à l'un des sommets du domaine d'itération. Nous prenons dans notre cas:

$$\vec{p}_o = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

ce qui réduit le nombre de types de supernoeuds partiels à 6. Si les constantes TFIN et LNGBARRE ne sont pas critiques[†] , il faut les ajuster de manière à avoir des origines de supernoeuds aux autres sommets du domaine d'itération:

$$(TFIN,1) \quad (TFIN,LNGBARRE) \quad (1,LNGBARRE)$$

Cela dépend bien entendu du facteur d'échelle λ et de la base P^I qui définit le plus petit supernoeud possible:

$$P^I = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$$

Il faut que les équations:

$$\vec{p}_o + \lambda P^I \begin{pmatrix} \mu \\ \nu \end{pmatrix} = x$$

où x est l'un quelconque des trois sommets restants, soient solubles en nombres entiers pour μ et ν. Le troisième sommet étant obtenu par la somme des deux vecteurs frontières, il suffit de vérifier:

$$\begin{pmatrix} \mu \\ \nu \end{pmatrix} = \frac{1}{2\lambda} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} \text{TFIN}-1 \\ 0 \end{pmatrix}$$

ce qui implique que TFIN doit être de la forme $2K\lambda + 1$ et:

$$\begin{pmatrix} \mu \\ \nu \end{pmatrix} = \frac{1}{2\lambda} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ \text{LNGBARRE}-1 \end{pmatrix}$$

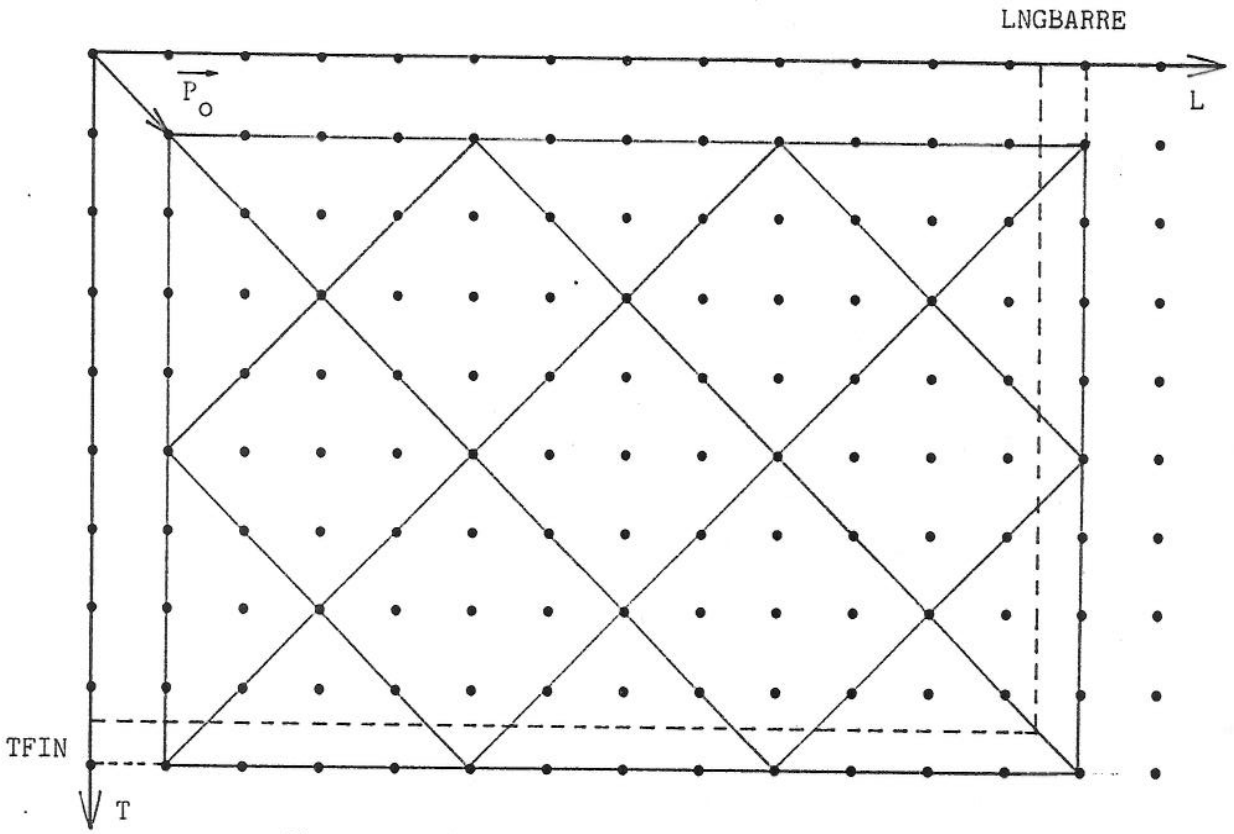
ce qui implique aussi LNGBARRE $= 2K'\lambda + 1$.

Un tel partitionnement est représenté figure 7.4 pour $\lambda = 2$. On remarque qu'il n'y a même pas de supernoeuds spéciaux dans les coins (§ 3.2.5).

Pour obtenir un résultat correct en Fortran, il faut tenir compte des remarques sur le caractère strict ou non strict des inégalités définissant les supernoeuds (§ 3.2.2) et décaler la frontière terminale de manière à transformer les inégalités strictes utilisées dans les calculs en inégalités larges nécessaires à la génération de bornes de boucles DO. Les deux types de frontières, large et stricte, sont dessinés sur la figure 7.4. Cela revient, dans notre cas, à décrémenter TFIN et à poser TFIN $= 2K\lambda$.

Les vecteurs ne sont pas de longueur constante dans les supernoeuds.

[†] Pour éviter des conflits de bancs, il est déjà souvent demandé aux programmeurs d'éviter les puissances de 2 comme dimensions de tableaux.



Ajustement du domaine d'itération au partitionnement

- Figure 7.4 -

7.5. Choix du facteur d'échelle λ

Les critères de choix du facteur d'échelle λ sont énumérés en § 6.4.

Ce facteur est tout d'abord usuellement contraint par la taille du cache disponible pour chaque processeur, mais le cache s'est avéré inutile pour notre problème (§ 7.3). Aucune optimisation des transferts de données n'est possible (critères 1 et 4).

Il n'y a pas non plus de problème majeur avec l'équilibre de la charge des processeurs dans un front puisque le degré de parallélisme est constant. Il faut seulement vérifier qu'il reste supérieur au nombre de processeurs, 8 dans le cas de l'Alliant, et même largement supérieur puisque le nombre de supernoeuds varie d'un front à l'autre à cause des supernoeuds partiels. Manuellement, il est tentant de regrouper ces deux supernoeuds... Le calcul du degré de parallélisme a été présenté en 6.2.7. Le critère d'orthogonalité conduit à choisir $\vec{f}_2$ comme représentatif des fronts et à avoir:

$$d = \frac{\text{LNGBARRE}-1}{2\lambda}$$

La constante K' doit donc être supérieure à 8, ou plus exactement:

$$\lambda < \frac{\text{LNGBARRE}-1}{16}$$

(critère 8)

Le facteur d'échelle doit aussi permettre d'utiliser correctement les unités vectorielles et leurs registres. Nous ne connaissons pas les valeurs de $n_{1/2}$ pour l'additionneur et le multiplieur, mais les registres ont une longueur de 32 et des expériences ont montré que des longueurs de vecteurs comprises entre 64 et 256 donnnaient des résultats optimaux. Comme la longueur moyenne d'un vecteur dans un supernoeud est justement λ, on peut envisager d'appliquer le partitionnement quand LNGBARRE est supérieur à 1025 (critère 2).

Le nombre de processeurs est fini, il est donc inutile de minimiser le nombre de fronts (critère 7). Le temps d'exécution du corps de boucle est constant, la génération du code de contrôle global va permettre d'éviter totalement le balayage des noeuds vides (cf. § 7.6.1) et les noeuds partiels seront exécutés vectoriellement, ce qui permet vraisemblablement d'en négliger l'influence. Les critères 7,9,10 et 11 ne s'appliquent donc pas.

Dans la suite, nous prenons une très faible valeur, $\lambda = 2\sqrt{2}$, pour pouvoir illustrer le partitionnement sur des figures d'une taille raisonnable. On a donc:

$$P = \begin{pmatrix} 2 & 2 \\ -2 & 2 \end{pmatrix} \quad P^{-1} = \frac{1}{8}\begin{pmatrix} 2 & -2 \\ 2 & 2 \end{pmatrix} \quad G = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \quad PG = \begin{pmatrix} 0 & 2 \\ -4 & 2 \end{pmatrix}$$

7.6. Génération du code de contrôle global

7.6.1. Principes généraux

La génération des expressions de bornes ou des tests nécessaires au code de contrôle global s'effectue par projection et/ou élimination de redondance de systèmes d'inéquations et d'équations linéaires en nombres entiers sur les variables de contrôle.

Soit T et L les coordonnées d'un noeud quelconque et T0 et L0 les coordonnées de l'origine d'un supernoeud. Les systèmes qui nous intéressent sont constitués de la disjonction de plus petits systèmes exprimant les conditions de base suivantes:

- le noeud (T,L) appartient au domaine d'itération:

$$A \begin{pmatrix} T \\ L \end{pmatrix} \leq \vec{\alpha}$$

- le noeud (T,L) appartient au supernoeud (T0,L0):

$$\vec{0} \leq P^{-1} \begin{pmatrix} T - \mathrm{T0} \\ L - \mathrm{L0} \end{pmatrix} < \vec{1}$$

  La matrice P intègre λ pour alléger les écritures et ce système d'inéquations est systématiquement multiplié par $det(P)$ pour pouvoir faire les cacluls en nombres entiers;

- (T0,L0) est une origine de supernoeud:

$$\begin{pmatrix} \mathrm{T0} \\ \mathrm{L0} \end{pmatrix} = PG \begin{pmatrix} \mathrm{L1} \\ \mathrm{L2} \end{pmatrix} + \vec{p}_o \qquad ou\ bien \qquad \begin{pmatrix} \mathrm{T0} \\ \mathrm{L0} \end{pmatrix} = P \begin{pmatrix} \mathrm{L1} \\ \mathrm{L2} \end{pmatrix} + \vec{p}_o$$

  ceci exprime que les origines de supernoeuds ont des coordonnées entières dans la base de partitionnement et dans la base de balayage des supernoeuds. L1 et L2 sont des entiers quelconques. Dans la suite, nous utiliserons systématiquement l'expression faisant intervenir G car L1 et L2 sont alors les indices des deux boucles de balayage.

- les demi-plans extérieurs qui définissent les frontières du domaine d'itération sont obtenus par négation d'**une** des inégalités du domaine d'itération.

7.6.2. Calcul des bornes des boucles DO

Comme le domaine d'itération est un rectangle, la matrice APG va être triangulaire et permettre de balayer le domaine sans utiliser d'opérateurs MIN ou MAX dans les expressions de bornes de boucles (§ 3.2.4).

On construit un système d'inégalités exprimant le fait qu'au moins un noeud (T,L) de chaque supernoeud (T0,L0) appartient au domaine d'itération et que le supernoeud (T0,L0) est obtenu en fonction des indices de boucle L1 et L2, sur lesquels on cherche des contraintes:

$$\begin{cases} \vec{0} \leq P^{-1} \begin{pmatrix} T-T0 \\ L-L0 \end{pmatrix} < \vec{1} \\ A \begin{pmatrix} T \\ L \end{pmatrix} \leq \vec{\alpha} \\ \begin{pmatrix} T0 \\ L0 \end{pmatrix} = PG \begin{pmatrix} L1 \\ L2 \end{pmatrix} + \vec{p}_o \end{cases}$$

où PG est la matrice permettant de passer de la base G à la base B. Comme le domaine d'itération est rectangulaire, la matrice A est constituée par la superposition d'une matrice identité I et d'une matrice $-I$. Dans notre cas, cela donne le système d'inéquations suivant:

$$\begin{cases} 0 \leq 2T - 2T0 - 2L + 2L0 < 8 \\ 0 \leq 2T - 2T0 + 2L - 2L0 < 8 \\ 1 \leq T \leq TFIN = 8 \\ 1 \leq L \leq LNGBARRE = 13 \\ T0 = 1 + 2L2 \\ L0 = 1 + 2L2 - 4L1 \end{cases}$$

La projection de ce système sur L2 permet d'obtenir les bornes de la boucle externe sur L2. Elle donne:

$$-1 \leq L2 \leq 3$$

La projection avec élimination de redondance sur L1 et sur L2 permet d'obtenir les bornes de la boucle interne:

$$\begin{cases} -1 \leq L2 \leq 3 \\ 2L1 - L2 \leq 0 \\ -2L1 + L2 \leq 6 \end{cases}$$

On retrouve le résultat de la projection sur L2, mais ce n'est pas le cas en général. Il reste à exprimer L1 en fonction de L2 en utilisant les règles et la division définies en § 1.4.5:

$$\begin{cases} L1 \leq \dfrac{L2}{2} \\ L1 \geq \dfrac{L2 - 5}{2} \end{cases}$$

La validité de ces bornes peut être vérifiée sur la figure 7.6.2.

Ceci se traduit en Fortran, en modifiant le code donné en § 3.2.6 pour tenir compte de la couverture du domaine d'itération par un système triangulaire et non par un système rectangulaire, par:

```
TFRONT = p0  e1 + BI 1  g2 e1 = -1
LFRONT = p0  e2 + BI 1  g2 e2 = -1
DOSEQ L2 = -1, 3
  DOALL L1 = mydiv(L2-5,2),mydiv(L2,2)
    T0 = TFRONT + L1 *  g1 e1 = TFRONT
    L0 = LFRONT + L1 *  g1 e2 = LFRONT - 4*L1
    ...
  ENDDOALL
  TFRONT = TFRONT +  g2 e1 = TFRONT + 2
  LFRONT = LFRONT +  g2 e2 = LFRONT + 2
ENDDOSEQ
```

7.6.3. Test des supernoeuds vides

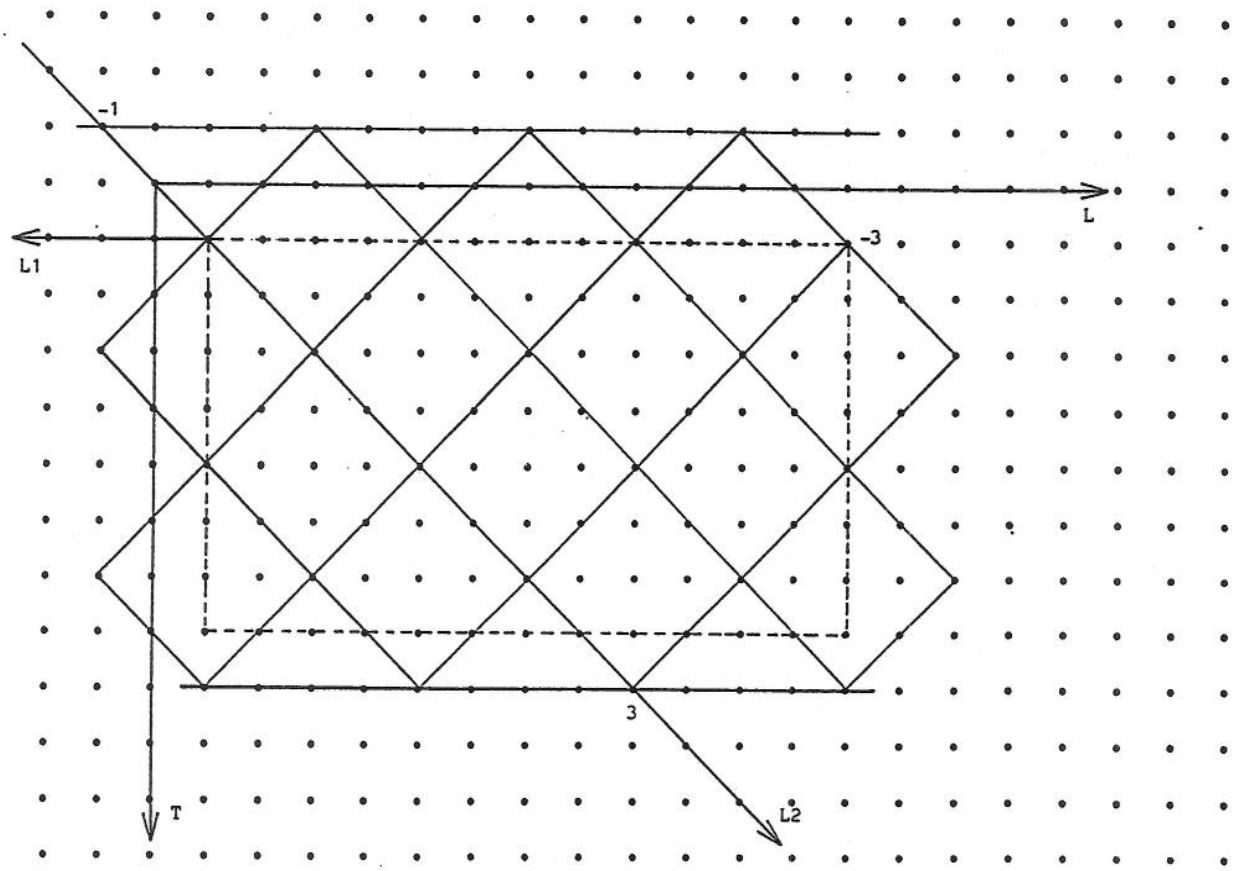
Le test des supernoeuds vides est inutile si l'algorithme de projection en nombres entiers que l'on a utilisé au paragraphe précédent est exact. Tout couple (L1,L2) correspond par définition de la projection à un supernoeud dont au moins un noeud appartient au domaine d'itération.

Nous avons utilisé un algorithme de projection développé par R.Triolet[4] pour réaliser un test de faisabilité par la méthode de Fourier-Motzkin. Nous l'avons complété par un algorithme d'élimination de redondance par test de faisabilité, différent de celui qu'utilise N. Halbwachs[2] . Enfin, les équations ont été traitées manuellement pour éviter leur transformation en inéquations et la perte d'information qui en résultait parfois. Tous les résultats donnés dans ce chapitre ont été vérifiés par programme.

Pour notre exemple, la projection est correcte et le test des supernoeuds vides est donc **inutile.**

7.6.4. Test des supernoeuds complets

Les supernoeuds complets sont des polyèdres convexes inclus dans le domaine d'itération, qui est lui aussi convexe. Il suffit donc de vérifier que chacun des sommets du supernoeud appartient au domaine d'itération: les coordonnées des noeuds du supernoeuds sont des combinaisons linéaires positives des coordonnées des sommets et donc des combinaisons linéaires de points du domaine d'itération. Ces conditions s'expriment dans notre cas particulier par le système d'inégalités suivant (§ 3.2.5):



Balayage des supernoeud dans la base G par les indices L1 et L2

- Figure 7.6.2 -

$$
\begin{cases}
1 \leq \mathrm{T0} \leq \mathrm{TFIN} \\
1 \leq \mathrm{L0} \leq \mathrm{LNGBARRE} \\
0 \leq \mathrm{T0} + \vec{p}_1 \vec{e}_1 = \mathrm{T0} + 2 \leq \mathrm{TFIN} + 1 \\
0 \leq \mathrm{L0} + \vec{p}_1 \vec{e}_2 = \mathrm{L0} - 2 \leq \mathrm{LNGBARRE} + 1 \\
0 \leq \mathrm{T0} + \vec{p}_2 \vec{e}_1 = \mathrm{T0} + 2 \leq \mathrm{TFIN} + 1 \\
0 \leq \mathrm{L0} + \vec{p}_2 \vec{e}_2 = \mathrm{L0} + 2 \leq \mathrm{LNGBARRE} + 1 \\
0 \leq \mathrm{T0} + < \vec{p}_1 + \vec{p}_2 \,.\, \vec{e}_1 > = \mathrm{T0} + 4 \leq \mathrm{TFIN} + 1 \\
0 \leq \mathrm{L0} + < \vec{p}_1 + \vec{p}_2 \,.\, \vec{e}_2 > = \mathrm{L0} \leq \mathrm{LNGBARRE} + 1 \\
\mathrm{T0} = 1 + 2\mathrm{L2} \\
\mathrm{L0} = 1 + 2\mathrm{L2} - 4\mathrm{L1}
\end{cases}
$$

Les "0" de gauche et les "+ 1" qui apparaissent dans les termes droits traduisent l'inégalité **stricte** des

frontières de sortie du supernoeud. Si un noeud du supernoeud atteignait ces limites, il serait combinaison linéaire de deux sommets atteignant aussi cette limite. L'origine du supernoeud ne peut pas être un de ces deux sommets puisqu'elle vérifie les limites exactes du domaine d'itération. Un des deux sommets doit être le *plus grand* point du supernoeud:

$$(T0,L0) + \vec{p}_1 + \vec{p}_2$$

sinon le noeud considéré serait un des deux autres sommets, ce qui est exclu par définition des supernoeuds. Le noeud considéré appartient donc à un des deux bords de sortie du supernoeud et non au supernoeud. La généralisation de cette solution au cas où A n'est pas la matrice identité comme dans notre exemple reste à faire.

Pour pouvoir traiter le cas de boucles mal imbriquées, il faudra imposer que le supernoeud appartienne à l'intérieur du domaine d'itération en utilisant simplement les contraintes usuelles du domaine d'itération.

La simplification automatique de ce système est possible en éliminant sa redondance et donne:

$$\begin{cases} 1 \leq T0 \leq TFIN - 3 \\ 3 \leq L0 \leq LNGBARRE - 1 \end{cases}$$

L'algorithme de projection en nombres entiers profite de la connaissance des valeurs numériques de TFIN et de LNGBARRE car il peut appliquer des critères de divisibilité. Avec TFIN $=$ 8 et LNGBARRE $=$ 13, on obtient:

$$\begin{cases} 1 \leq T0 \leq 5 \\ 3 \leq L0 \leq 11 \end{cases}$$

Le gain est ici limité à la borne supérieure sur L.

On voit sur la figure 7.6.2 que les deux contraintes sur T sont atteintes avec le partitionnement choisi et que le système réduit est valable puisque les noeuds appartenant aux faces de *sortie* d'un supernoeud ne font pas partie de ce dernier.

7.6.5. Test d'un supernoeud frontière

Il y a toujours quatre frontières. Nous ne développons les calculs que pour la première frontière définie par $\vec{f}_1$:

$$L = 1$$

Pour qu'un supernoeud (T0,L0) soit un supernoeud partiel appartenant à cette frontière, il faut qu'il ait un noeud (T,L) appartenant à l'extérieur de cette frontière et un noeud (T',L') appartenant au domaine d'itération. L'origine du supernoeud (T0,L0) doit aussi avoir des coordonnées entières dans la base G.

L'appartenance au demi-plan est donnée par l'inégalité:

$$L < 1$$

L'appartenance au domaine d'itération par:

$$A \begin{pmatrix} T' \\ L' \end{pmatrix} \leq \vec{\alpha}$$

tandis que l'appartenance au supernoeud de ces deux noeuds s'écrit:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \leq P^{-1} \begin{pmatrix} T - T0 \\ L - L0 \end{pmatrix} < \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \leq P^{-1} \begin{pmatrix} T' - T0 \\ L' - L0 \end{pmatrix} < \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Les contraintes sur l'origine du supernoeud sont données par:

$$\begin{pmatrix} T0 \\ L0 \end{pmatrix} = PG \begin{pmatrix} L1 \\ L2 \end{pmatrix} + \vec{p}_o$$

Cela donne dans notre cas, après multiplication par $det(P)$ pour revenir à des calculs en nombres entiers:

$$\begin{cases} L < 1 \\ 1 \leq T' \leq 8 \\ 1 \leq L' \leq 13 \\ 0 \leq 2T - 2T0 - 2L + 2L0 < 8 \\ 0 \leq 2T - 2T0 + 2L - 2L0 < 8 \\ 0 \leq 2T' - 2T0 - 2L' + 2L0 < 8 \\ 0 \leq 2T' - 2T0 + 2L' - 2L0 < 8 \\ T0 = 1 + 2\,L2 \\ L0 = 1 + 2\,L2 - 4\,L1 \end{cases}$$

Il faut éliminer les variables auxiliaires T, L, T', L', L1 et L2 de ce système en le projetant pour obtenir un test sur T0 et L0. La projection donne:

$$\begin{cases} L0 = 1 \\ 1 \leq T0 \leq 5 \\ T0 - L0 \leq 4 \\ -T0 + L0 \leq 12 \\ T0 + L0 \leq 18 \\ -T0 - L0 \leq -2 \end{cases}$$

Les quatre dernières inégalités sont redondantes avec les deux premières. Pour optimiser le test, il faut aussi se rendre compte que l'intervalle de définition de T est inutile. C'est évident pour $T0 \geq 1$ à cause du domaine d'itération. Ça l'est moins pour $T0 \leq 5$ qui traduit le fait qu'il n'y a pas de noeud coin. Sachant que ce test sera placé à l'intérieur des deux boucles sur L1 et L2, on souhaite donc montrer que

les deux inégalités $1 \leq T0$ et $T0 \leq 5$ sont redondantes dans le système suivant:

$$\begin{cases} 1 \leq T0 \\ T0 \leq 5 \\ L0 = 1 \\ -1 \leq L2 \leq 3 \\ 2L1 - L2 \leq 0 \\ -2L1 + L2 \leq 6 \\ T0 = 1 + 2\,L2 \\ L0 = 1 + 2\,L2 - 4\,L1 \end{cases}$$

La redondance est très facile à montrer manuellement car on peut rétrécir les bornes sur L2 à 0 et 2 en les déduisant des bornes sur L1 et de la relation L2=2L1. Mais nous n'avons pas actuellement de programme qui sache exploiter correctement les équations.

Les trois autres frontières se traduisent en tests de la manière suivante:

$$\begin{array}{lll} T > \text{TFIN} & \rightarrow & T0 \geq \text{TFIN} - 2 \\ L > \text{LNGBARRE} & \rightarrow & L0 \geq \text{LNGBARRE} \\ T < 1 & \rightarrow & T0 \leq 0 \end{array}$$

En remplaçant TFIN et LNGBARRE par leur valeurs numériques, on profite de la résolution en nombres entiers:

$$\begin{array}{lll} L < 1 & \rightarrow & L0 = 1 \\ T > 8 & \rightarrow & T0 = 7 \\ L > 13 & \rightarrow & L0 = 13 \\ T < 1 & \rightarrow & T0 = -1 \end{array}$$

Ces valeurs peuvent être vérifiées sur la figure 7.6.2.

7.6.6. Tests des supernoeuds coins

Le choix de l'origine du partitionnement et des dimensions du domaine d'itération permet d'éviter l'existence de supernoeuds *coins*. La génération des tests reposerait aussi sur la constitution d'un système d'inégalités exprimant qu'un des sommets du domaine d'itération appartient au supernoeud et que le supernoeud appartient à la grille de balayage des supernoeuds. Si l'origine du supernoeud obtenu est le sommet du domaine d'itération, le supernoeud est simplement noeud frontière, à moins qu'on ne veuille prendre en compte des boucles mal imbriquées ou des domaines d'itération très pointus dans lesquels un supernoeud peut appartenir à deux frontières.

On peut vérifier l'absence de deux coins en testant la non-faisabilité des quatre systèmes définissant un supernoeud possédant trois noeuds dont un appartient au domaine d'itération et dont les deux autres violent chacun une frontière différente.

7.6.7. Code de contrôle global complet

Le code donné en 7.6.2 est enrichi des tests sur les supernoeuds complets et les supernoeuds frontières. Le test sur les supernoeuds vides est éliminé parce qu'il est inutile.

```
TFRONT = -1
LFRONT = -1
DOSEQ L2 = -1, 3
  DOALL L1 = mydiv(L2-5,2),mydiv(L2,2)
    T0 = TFRONT
    L0 = LFRONT - 4*L1
    IF(1.LE.T0.AND.T0.LE.5.AND.2.LE.L0.AND.L0.LE.11) THEN
      CALL COMPLET(T0,L0,...)
    ELSEIF(L0.EQ.1) THEN
      CALL FRONTIERE1(T0,L0,...)
    ELSEIF(T0.EQ.7) THEN
      CALL FRONTIERE2(T0,L0,...)
    ELSEIF(L0.EQ.13) THEN
      CALL FRONTIERE3(T0,L0,...)
    ELSEIF(T0.EQ.-1) THEN
      CALL FRONTIERE4(T0,L0,...)
    ELSE
      CALL ERREUR(T0,L0)
  ENDDOALL
  TFRONT = TFRONT + 2
  LFRONT = LFRONT + 2
ENDDOSEQ
```

Bien que la taille des supernoeuds doive permettre d'amortir le quadruple test précédant l'appel d'un supernoeud complet, il est envisageable d'exprimer ce test en terme de L1 et de L2, puis de le découper en plusieurs morceaux, au niveau de la boucle sur L2 d'abord, puis sur les bornes de la boucle L1. Le schéma de contrôle serait alors:

```
traiter la frontière 4
pour chaque front contenant des supernoeuds complets
   frontière 1
   traiter le front de supernoeuds complets
   frontière 3
traiter la frontière 2
```

Les tests et l'overhead associé seraient alors complètement supprimés.

7.7. Code de contrôle des supernoeuds

7.7.1. Code d'un supernoeud complet

La machine cible est un multiprocesseur vectoriel. On va donc garder une boucle interne vectorielle ce qui impose d'itérer le long de T. On applique d'abord la méthode décrite en § 3.3.1.3 car elle est la plus simple. Il faut projeter le supernoeud sur la direction d'itération pour obtenir les deux bornes de la boucle externe. Soit t et l les coordonnées d'un noeud de supernoeud, relatives à l'origine du supernoeud. Elles vérifient le système:

$$\begin{pmatrix}0\\0\end{pmatrix} \leq P^{-1} \begin{pmatrix}t\\l\end{pmatrix} < \begin{pmatrix}1\\1\end{pmatrix}$$

Dans le cas général, il faudrait exprimer la coordonnée selon la direction d'itération en fonction de t et l, mais ici on utilise justement la direction t. La projection du système précédent sur t donne:

$$0 \leq t \leq 3$$

Il ne reste plus qu'à classer les contraintes du système en contraintes à coefficients négatifs, qui définissent la borne inférieure des itérations sur l, et en contraintes à coefficients positifs, qui définissent la borne supérieure (cf. figure 7.7.1):

$$\begin{cases}2t - 2l \leq 7\\-2t - 2l \leq 0\end{cases} \rightarrow \begin{cases}t - l \leq 3\\-t - l \leq 0\end{cases} \rightarrow \quad l \geq \mathrm{MAX}(-t, t-3)$$

$$\begin{cases}-2t + 2l \leq 0\\2t + 2l \leq 7\end{cases} \rightarrow \begin{cases}-t + l \leq 0\\t + l \leq 3\end{cases} \rightarrow \quad l \leq \mathrm{MIN}(t, 3-t)$$

En appelant maintenant T = T0 + t et L = L0 + l les coordonnées relatives au repère initial et (T0,L0) les coordonnées du supernoeud, le code de contrôle résultant est donc:

```
      SUBROUTINE COMPLET(T0,L0,...)
      ...déclarations
      DOSEQ 100 T = T0, T0+3
        DOVEC 100 L = L0+MAX(T0-T,T-T0-3),L0+MIN(T-T0,3-T+T0)
          ... corps de la boucle
100   CONTINUE
```

Code de contrôle d'un supernoeud complet

Pour diminuer l'overhead dû aux MIN et aux MAX, il faut décomposer les supernoeuds en 1, 2 ou 3 polygones *triangulaires*. La méthode de Kuhn a été proposée en § 3.3.1.4. Dans le cas de la dimension 2, il est possible de procéder plus simplement en classant les quatre sommets des supernoeuds suivant la valeur de leurs premières coordonnées dans la base d'itération souhaitée. On obtient 2, 3 ou 4 valeurs (une seule valeur impliquerait que $\vec{p}_1$ et $\vec{p}_2$ sont liés) suivant le nombre de sommets qui ont la même valeur. S'il n'y a que deux valeurs, la méthode de génération précédente doit conduire à une seule paire de contraintes et MIN et MAX sont inutiles. S'il y a trois valeurs, il faut découper le supernoeud en deux parties en utilisant la valeur intermédiaire pour définir deux demi-plans d'intersection **vide** comme contraintes supplémentaires. Les deux parties sont des triangles. Enfin, s'il y a quatre valeurs, les deux triangles sont séparés par un parallélogramme et il faut utiliser l'intersection de deux demi-plans pour définir la zone centrale.

Dans notre cas, on trouve trois valeurs: 0, 2 et 4. Deux systèmes en découlent:

$$\begin{pmatrix}0\\0\end{pmatrix} \leq P^{-1}\begin{pmatrix}T\\L\end{pmatrix} < \begin{pmatrix}1\\1\end{pmatrix} \quad \text{et} \quad T < 2$$

et:

$$\begin{pmatrix}0\\0\end{pmatrix} \leq P^{-1}\begin{pmatrix}T\\L\end{pmatrix} < \begin{pmatrix}1\\1\end{pmatrix} \quad \text{et} \quad T \geq 2$$

qu'il faut projeter pour trouver les bornes sup et inf sur T. On ajoute ensuite les deux inégalités trouvées au système initial et on en élimine la redondance sans toucher à ces deux nouvelles inégalités.

```
      SUBROUTINE COMPLET(T0,L0,...)
      ...déclarations
      DOSEQ 100 T = T0, T0+1
        DOVEC 100 L = L0+T0-T,L0+T-T0
          ... corps de la boucle
100   CONTINUE
      DOSEQ 110 T = T0+2, T0+3
        DOVEC 110 L = L0+T-T0-3,L0+3-T+T0
          ... corps de la boucle
110   CONTINUE
```

Code de contrôle optimal d'un supernoeud complet

7.7.2. Code de contrôle d'un supernoeud partiel

Pour des raisons d'efficacité, il faut aussi générer du code vectoriel. Les deux méthodes utilisées dans le paragraphe précédent restent valides mais il faut ajouter au système de départ les contraintes propres à chaque frontière et au domaine d'itération.

Dans le cas de la frontière 1, on a trouvé précédemment une contrainte sur les coordonnées (T0,L0) de l'origine des supernoeuds:

$$L0 = 1$$

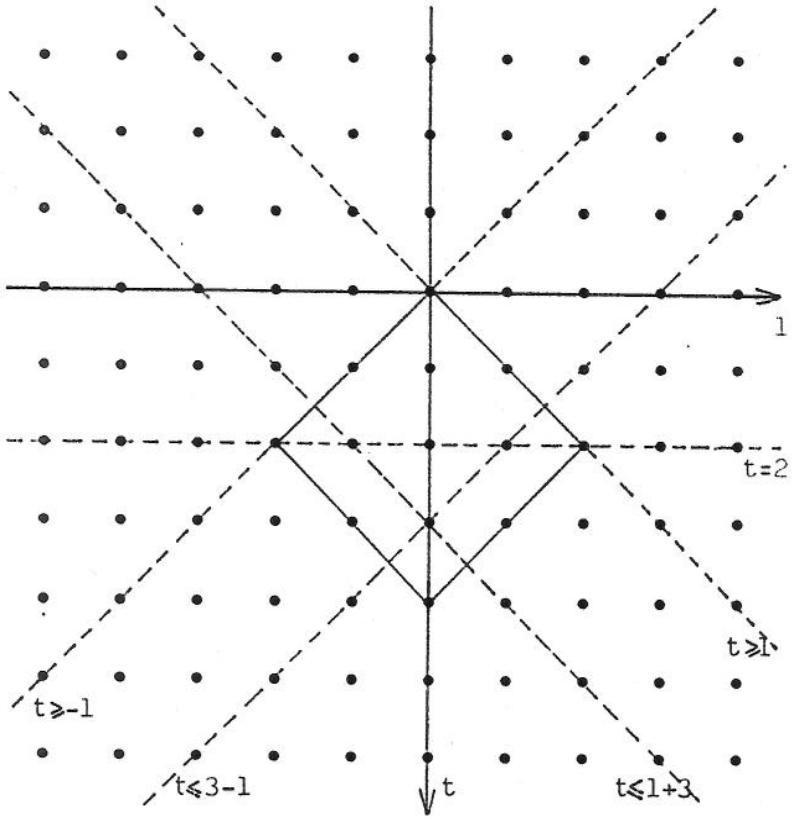
En appelant t et l les coordonnées relatives à l'origine (T0,L0), on a aussi:

$$\begin{cases} 1 \leq T0 + t \leq 8 \\ 1 \leq L0 + l \leq 13 \end{cases}$$

qui caractérise le domaine d'itération, et:

$$\begin{cases} 0 \leq 2t - 2l \leq 7 \\ 0 \leq 2t + 2l \leq 7 \end{cases}$$

qui définit le supernoeud. Il ne faut pas oublier non plus les deux équations qui définissent T0 et L0 en fonction de L1 et de L2.



Expressions des bornes des boucles balayant un supernoeud complet dans le repère local

- Figure 7.7.1 -

On peut projeter sur t et l et éliminer la redondance pour obtenir des tests simples mais il faut ensuite vérifier que les deux contraintes d'appartenance au domaine d'itération sont redondantes. Le respect des deux premières contraintes, qui font aussi intervenir T0 et L0, est garanti par le test de supernoeud frontière.

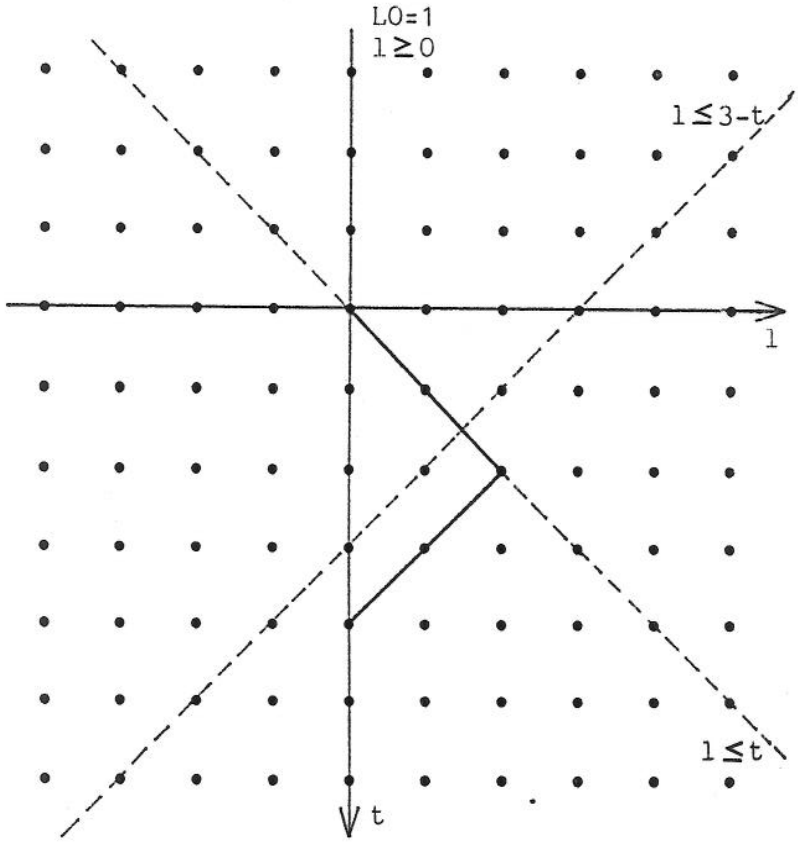
Dans notre cas, l'appartenance à la frontière entraîne $l \geq 0$:

$$\begin{cases} L0 = 1 \\ \\ 1 \leq L0 + l \end{cases} \quad => \quad -l \leq 0$$

Ceci rend deux des inéquations de définition du supernoeud, $2t-2l \leq 7$ et $-2t-2l \leq 0$ redondantes, ainsi que les quatre inéquations définissant le domaine d'itération. Le code de contrôle du supernoeud est donc défini par:

$$\begin{cases} -2t + 2l \leq 0 \\ 2t + 2l \leq 7 \\ -l \leq 0 \end{cases} \quad => \quad \begin{cases} -t + l \leq 0 \\ t + l \leq 3 \\ -l \leq 0 \end{cases}$$

Cette élimination de redondance peut être effectuée automatiquement. Il faut aussi projeter sur t pour obtenir les bornes de la boucle extérieure (cf. figure 7.7.2).



Expressions des bornes des boucles balayant un supernoeud partiel situé sur la frontière 1

- Figure 7.7.2 -

En utilisant maintenant les coordonnées dans le repère initial, T = T0 + t et L = L0 + l, le code devient:

```
      SUBROUTINE FRONTIERE1(T0,L0,...)
      ... déclarations
      DOSEQ 100 T = T0, T0+3
        DOVEC 100 L = L0,L0+MIN(T-T0,3-T+T0)
          ... corps de la boucle
100   CONTINUE
```

Code de contrôle d'un supernoeud partiel le long de la frontière F1

La suppression de l'appel à MIN nécessite le même partitionnement que pour les supernoeuds complets.

7.8. Modification des références aux variables

La machine cible, l'Alliant FX/8, dispose d'un cache qui nous permet de garder les références initiales inchangées. La génération de code s'achève donc en insérant le corps de boucle dans les codes de contrôle obtenus comme résultats des paragraphes 7.7.1 et 7.7.2:

```
      SUBROUTINE COMPLET(T0,L0,BARRE,TFIN,LNGBARRE)
      REAL BARRE(0:TFIN,0:LNGBARRE+1)
      DOSEQ 100 T = T0, T0+3
        DOALL 100 L = L0+MAX(T0-T,T-T0-3),L0+MIN(T-T0,3-T+T0)
          BARRE(T,L) = (BARRE(T-1,L-1)+BARRE(T-1,L)+BARRE(T-1,L+1))/3.
  100 CONTINUE
      RETURN
      END
```

Code d'un supernoeud complet pour machine à cache ou à mémoire globale accessible par les processeurs

---

Ce code est également valable pour les machines à mémoires locales dont les processeurs peuvent accéder la mémoire globale puisque nous avons laissé de côté, faute de temps, l'optimisation des accès à un même élément par des références différentes d'une même famille.

Les machines sans mémoire globale ou dont les processeurs ne peuvent pas accéder la mémoire globale, nécessitent des copies. Nous allons illustrer la seule méthode qui nous permet de garder la direction d'itération locale que nous avons choisie en § 7.7: la copie par excès avec accès linéaire (§ 3.5.3).

Nous n'avons qu'une famille de références du type BARRE(T+α,L+β) où α et β sont des constantes. Il n'y a donc pas de problème de cohérence (§ 3.5.5). Cette famille compte quatre membres, BARRE(T,L), BARRE(T–1,L–1), BARRE(T–1,L) et BARRE(T–1,L+1). En reprenant les notations de R. Triolet, il faut calculer le MIN et le MAX de ϕ_1 et de ϕ_2 pour ces quatre références en prenant t et l comme des coordonnées locales au supernoeud. Elles sont donc contraintes par:

$$\begin{cases} 0 \leq 2t - 2l \leq 7 \\ 0 \leq 2t + 2l \leq 7 \end{cases}$$

Il faut projeter **successivement** sur ϕ_1 et sur ϕ_2 les quatre systèmes obtenus en ajoutant à la définition du supernoeud les expressions d'indices. Comme ϕ_1 et ϕ_2 ne dépendent que d'une variable le calcul est particulièrement simple dans notre cas:

$$\begin{cases} -1 \leq \phi_1 \leq 3 \\ -2 \leq \phi_2 \leq 2 \end{cases}$$

La déclaration du tableau de copie s'en déduit immédiatement:

REAL COPIE(-1:3,-2:2)

et les références ne font plus intervenir que les coordonnées locales au supernoeud. La partie calcul du code d'un supernoeud devient donc:

```
      SUBROUTINE COMPLET(T0,L0,...)
      REAL COPIE(-1:3,-2:2)
      ...
      DOSEQ 100 T = 0, 3
        DOALL 100 L = MAX(-T,T-3), MIN(T,3-T)
          COPIE(T,L) = (COPIE(T-1,L-1)+COPIE(T-1,L)+COPIE(T-1,L+1))/3.
100   CONTINUE
      ...
      RETURN
      END
```

Code d'un supernoeud complet pour machine à mémoire locale

7.9. Génération du code de copie

L'unique famille de références est utilisée en entrée et en sortie. Il faut donc la copier avant et après l'exécution du code du supernoeud.

Nous traitons d'abord le cas des résultats, car il est plus simple dans la mesure où l'on fait l'hypothèse simplificatrice que toutes les valeurs produites par l'ensemble de boucles imbriquées initial sont utilisées dans la suite du programme. Si ce n'était pas le cas et si l'on était capable de caractériser les résultats utiles par une **région**, il faudrait faire en plus un calcul d'intersection.

Nous faisons l'hypothèse que, dans notre cas, toutes les donnnées définies doivent être recopiées. Comme il n'y a qu'une référence de définition, il suffit de reprendre le code de contrôle défini pour les calculs du supernoeud.

```
      DOALL 100 T = T0, T0+3
        DOALL 100 L = L0+MAX(T0-T,T-T0-3),L0+MIN(T-T0,3-T+T0)
          BARRE(T,L) = COPIE(T-T0,L-L0)
100   CONTINUE
```

Code de recopie des résultats d'un supernoeud complet

Les familles qui ne sont constituées que de références en utilisation doivent être copiées intégralement. Il n'y en a pas dans l'exemple que nous traitons ici.

Les données référencées en utilisation et en définition ne nécessitent une copie que si elles sont produites à l'extérieur du supernoeud, ou si elles sont utilisées avant d'être redéfinies (i.e. il existe une antidépendance).

Le premier cas ne s'exprime pas bien en terme de système d'inéquations puisqu'il faut caractériser **l'extérieur** d'un supernoeud. Ceci nécessite la projection dans l'espace des $\vec{\phi} = \Phi$ des n systèmes qui sont implicitement définis par:

$$\begin{cases} \vec{\phi}_u = \vec{\phi}_d \\ \vec{1}_H \ \vec{j}_u > \vec{1}_H \ \vec{j}_d \\ \vec{0} \leq P^{-1}(\vec{j}_u - \vec{j}_0) < \vec{1}_P \\ not\left(\vec{0} \leq P^{-1}(\vec{j}_d - \vec{j}_0) < \vec{1}_P\right) \end{cases}$$

La multiplicité des systèmes est due à la présence d'une négation. Il faut ensuite copier

indépendamment chaque polyèdre qu'on obtient dans l'espace Φ en considérant l'enveloppe convexe des polyèdres élémentaires obtenus pour chaque référence de la famille traitée. Les symboles $\vec{\phi}_u$ et $\vec{\phi}_d$ représentent les éléments du tableau référencés par les vecteurs d'itérations $\vec{j}_u$ et $\vec{j}_d$ et leur égalité traduit la collision des références. Ces symboles sont calculés en appliquant les formes linéaires qui définissent les références aux vecteurs d'itérations. La deuxième condition écarte les définitions postérieures à l'utilisation.

Quant au deuxième cas, celui où les données apparaissant dans une référence d'utilisation peuvent avoir été produites par le supernoeud lui-même, il s'exprime avec un seul système linéaire. Il n'est nécessaire de copier les données avant de commencer les calculs que si leur utilisation précède leur définition. Le volume référencé à copier vérifie donc le système suivant:

$$\begin{cases} \vec{\phi}_u = \vec{\phi}_d \\ \vec{1}_H \ \vec{j}_u < \vec{1}_H \ \vec{j}_d \\ \vec{0} \leq P^{-1}(\vec{j}_u - \vec{j}_0) < \vec{1}_P \\ \vec{0} \leq P^{-1}(\vec{j}_d - \vec{j}_0) < \vec{1}_P \end{cases}$$

Il faut calculer l'enveloppe convexe des polyèdres obtenus dans Φ pour chaque référence de la famille puis dériver des contraintes sur des indices de boucle à partir des contraintes dans Φ.

Dans l'exemple que nous traitons, le deuxième type de système est vide car nous n'avons pas d'antidépendance. En effet, le sous-système:

$$\vec{\phi}_u = \vec{\phi}_d \quad \text{et} \quad \vec{1}_H \ \vec{j}_u < \vec{1}_H \ \vec{j}_d$$

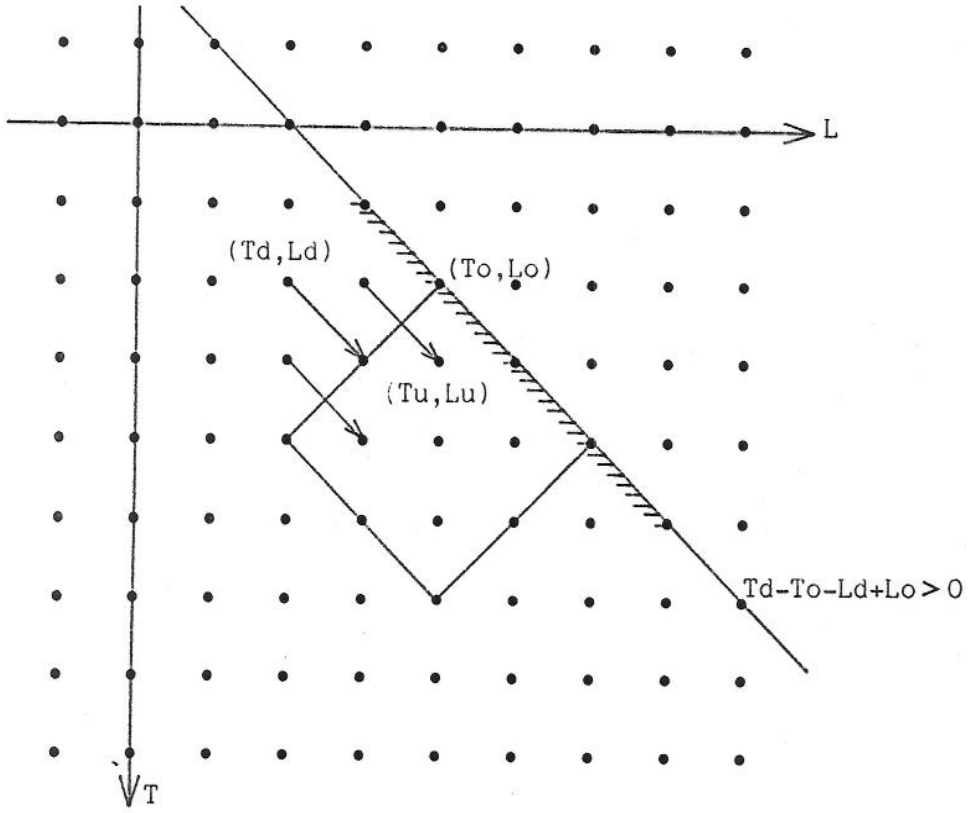
s'écrit:

$$T_d = T_u - 1 \quad \text{et} \quad T_u < T_d$$

en posant $\vec{j}_u = (T_u, L_u)$ et $\vec{j}_d = (T_d, L_d)$. Il est infaisable.

Par contre, nous avons quatre systèmes à projeter, et ce, pour trois références, avant d'obtenir les zones à copier. Pour la négation de la première condition d'appartenance au supernoeud et pour la première référence, on a (cf. figure 7.9.a):

$$\begin{cases} T_d = T_u - 1 \\ L_d = L_u - 1 \\ T_u > T_d \\ 0 \leq 2T_u - 2T0 - 2L_u + 2L0 < 8 \\ 0 \leq 2T_u - 2T0 + 2L_u - 2L0 < 8 \\ 0 > 2T_d - 2T0 - 2L_d + 2L0 \end{cases}$$



Données à copier en entrée

- Figure 7.9.a -

Ce premier système est infaisable et l'ensemble des éléments du tableau BARRE qu'il faut copier pour cette référence et pour cette frontière est vide. Cela est dû au fait que la frontière de supernoeud et la dépendance choisies sont parallèles (cf. figure 7.9.a). Les frontières *sortantes* du supernoeud comme:

$$2T_d - 2T0 - 2L_d + 2L0 \geq 8$$

donneraient aussi des systèmes infaisables à cause de la condition venant de l'ordonnancement, $T_u > T_d$. Mais la deuxième frontière *entrante* définit un sous-ensemble non vide d'éléments de BARRE:

$$\begin{cases} T_d = T_u - 1 \\ L_d = L_u - 1 \\ T_u > T_d \\ 0 \leq 2T_u - 2T0 - 2L_u + 2L0 < 8 \\ 0 \leq 2T_u - 2T0 + 2L_u - 2L0 < 8 \\ 0 > 2T_d - 2T0 + 2L_d - 2L0 \end{cases}$$

Comme $\vec{\phi}_d = (T_d, L_d)$, il faut projeter ce polyèdre sur (T0, L0, T_d, L_d) et éliminer la redondance pour trouver le système qui définit ce sous-ensemble:

$$\begin{cases} T_d - T0 + L_d - L0 \leq -1 \\ -T_d + T0 + L_d - L0 \leq 0 \\ -T_d + T0 + L_d + L0 \leq 2 \\ -L_d + L0 \leq 2 \end{cases}$$

Ce polyèdre est représenté sur la figure 7.9.b.

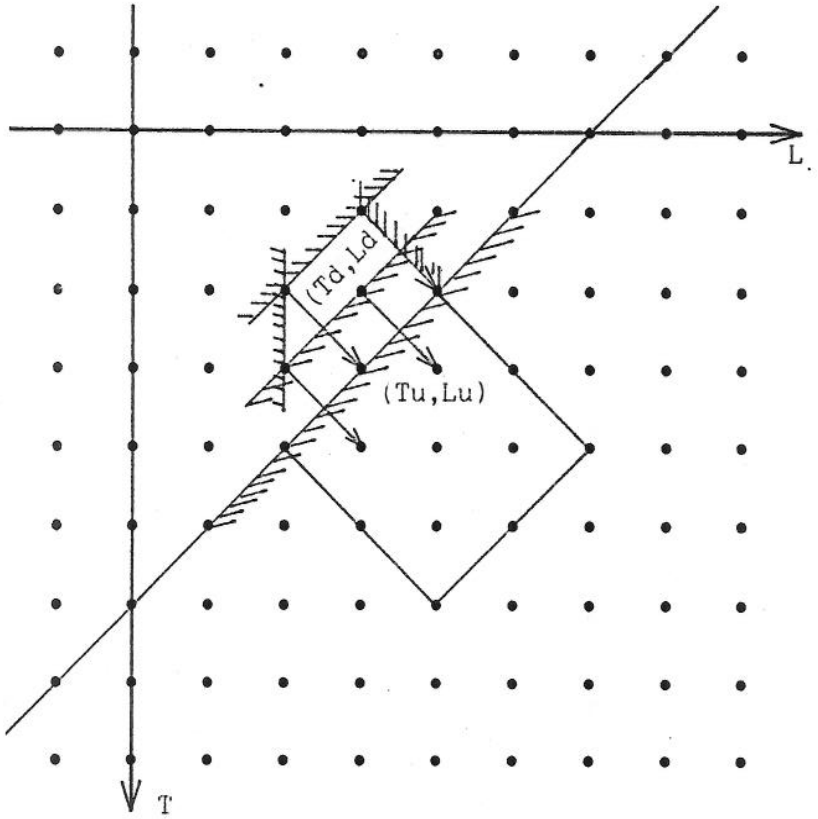
Avant d'exploiter ces inégalités pour générer le code de copie, il faudrait en calculer l'enveloppe convexe avec les inégalités produites par les deux autres références, BARRE(T–1,L) et BARRE(T–1,L+1). On voit sur la figure 7.9.b que la première d'entre elles ajouterait le point (T0–1,L0) et que la seconde ne donnerait qu'un ensemble vide.

La génération de code consiste à projeter le système final sur la variable de contrôle qu'on veut utiliser comme indice de la boucle externe (T_d dans notre cas) pour obtenir les deux bornes de cette boucle, puis à le projeter une deuxième fois sur les deux variables de contrôle (T_d et L_d) pour obtenir les bornes de la boucle interne. Si les inégalités sont trop nombreuses, il faut utiliser les fonctions MIN et MAX dans les expressions de bornes.

En négligeant le calcul de l'enveloppe convexe et en reprenant le système précédent, on obtient:

$$-1 \leq T_d - T0 \leq 1$$

On en déduit le code de copie:



Polyèdre des données lues

- Figure 7.9.b -/

```
      DO 100 T = T0-1, T0+1
        DO 100 L = L0+MAX(2,T-T0-2),L0+MIN(T-T0-1,T-T0)
          COPIE(T-T0,L-L0) = BARRE(T,L)
100   CONTINUE
```

Copie d'une partie des données du supernoeud (T0,L0)

La deuxième zone à copier est symétrique de la première:

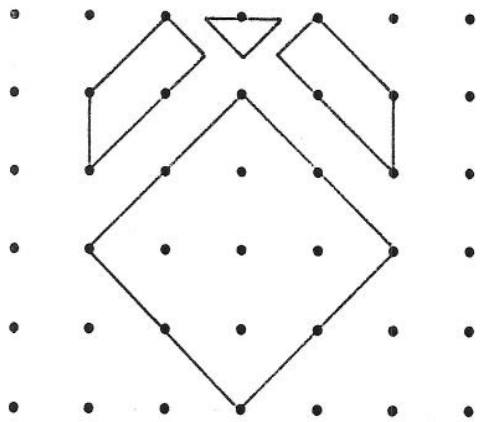
```
      DO 110 T = T0-1, T0+1
        DO 110 L = L0+MAX(-T+T0,T-T0+1),L0+MIN(2,T-T0+2)
          COPIE(T-T0,L-L0) = BARRE(T,L)
110   CONTINUE
```

Copie de la 2ème partie des données du supernoeud (T0,L0)

Il ne reste plus qu'un élément à copier:

$$\text{COPIE}(-1,0) = \text{BARRE}(T0-1,L0)$$

L'ensemble des données à copier en entrée est représenté en figure 7.9.c et le code complet est donné en figure 7.9.d.



Régions lues par le supernoeud

- Figure 7.9.c -

Le même principe s'applique aux supernoeuds partiels, en ajoutant la ou les contraintes sur (T0,L0) qui font que le supernoeud est partiel, et les inégalités du domaine d'itération.

```
      SUBROUTINE COMPLET(T0,L0,BARRE,TFIN,LNGBARRE)
      REAL BARRE(0:TFIN,0:LNGBARRE)
      REAL COPIE(-1:3,-2:2)

      DO 200 T = T0-1, T0+1
        DO 200 L = L0+MAX(2,T-T0-2),L0+MIN(T-T0-1,T-T0)
          COPIE(T-T0,L-L0) = BARRE(T,L)
200   CONTINUE
      DO 210 T = T0-1, T0+1
        DO 210 L = L0+MAX(-T+T0,T-T0+1),L0+MIN(2,T-T0+2)
          COPIE(T-T0,L-L0) = BARRE(T,L)
210   CONTINUE
      COPIE(-1,0) = BARRE(T0-1,L0)

      DOSEQ 100 T = 0, 3
        DOALL 100 L = MAX(-T,T-3), MIN(T,MYDIV(7-2T,2))
          COPIE(T,L) = (COPIE(T-1,L-1)+COPIE(T-1,L)+COPIE(T-1,L+1))/3.
100   CONTINUE

      DOALL 300 T = T0, T0+3
        DOALL 300 L = L0+MAX(T0-T,T-T0-3),L0+MIN(T-T0,MYDIV(7-2T+2T0,2))
          BARRE(T,L) = COPIE(T-T0,L-L0)
300   CONTINUE

      RETURN
      END
```

Code d'un supernoeud complet pour machines à mémoires locales

- Figure 7.9.d -

7.10. Résultats expérimentaux

La floraison de machines parallèles que l'on observe à l'heure actuelle sur le marché est survenue trop tard pour que nous puissions valider le partitionnement sur plusieurs machines et sur plusieurs programmes. Aucun multiprocesseur vectoriel n'était disponible en France à part quelques Cray X-MP auxquels nous n'avions pas accès.

Il a heureusement été possible de faire **un** essai[3] sur un minisuperordinateur Alliant FX/8 aux Etats-Unis grâce au professeur Sameh du Center for Supercomputer Research and Development (University of Illinois at Urbana-Champaign). L'Alliant FX/8, qui a été présentée au paragraphe § 1.1.1.3, n'atteint ses performances nominales que si les données se trouvent dans le cache ou les registres vectoriels. Elle était donc une cible naturelle pour notre méthode car la localité propre aux supernoeuds allait permettre de conserver les données dans le cache beaucoup plus longtemps que ne le faisait le tronçonnage de la boucle parallèle sur L qu'effectuait le restructureur VAST.

La génération du code partitionné a été effectuée entièrement à la main car nous ne disposions pas alors des programmes de calculs en nombres entiers nécessaires aux calculs des tests et des bornes de boucles. Bien que les structures de contrôle soient voisines de celles que nous venons de dériver systématiquement, elles ne sont pas aussi propres. En particulier, le balayage des supernoeuds est obtenu par déroulement de boucle plutôt que par changement de base et les dimensions ne sont pas adaptées de la même manière à la taille des supernoeuds.

Nous avons aussi optimisé l'utilisation de la mémoire en ne conservant pas en mémoire le tableau de valeurs BARRE(0:TFIN,0:LNGBARRE+1) mais seulement deux lignes de ce tableau, BARRE1(0:LNGBARRE+1) et BARRE2(0:LNGBARRE+1). Cette optimisation a plusieurs conséquences fâcheuses dont nous n'avons pas pris conscience immédiatement:

- l'initialisation des extrémités de BARRE1 et de BARRE2 ne permet plus d'avoir des boucles parfaitement imbriquées;
- les calculs sur BARRE1 et BARRE2 correspondent à deux boucles situées au même niveau;
- en conséquence, les conditions d'application de notre méthode ne sont pas vérifiées.

L'augmentation de 50 % de la vitesse que nous avons mesurée quand les données ne tiennent plus en cache (cf. figure 7.10) n'est donc pas directement significative pour notre méthode. Elle montre seulement que la localité des accès a, sur l'Alliant FX/8, une importance considérable. Le nombre des défauts de cache et le temps d'exécution total ont été réduits selon les formules suivantes:

$$V \rightarrow \alpha\frac{V}{\lambda}$$

$$V(t_c + t_e) \rightarrow Vt_c + \alpha\frac{Vt_e}{\lambda}$$

puisque le nombre des défauts de cache est proportionnel à la frontière du supernoeud. V est le nombre de noeuds (i.e. le volume) du domaine d'itération, le symbole t_c représente le temps de calcul d'un noeud quand toutes les données sont disponibles en cache ou en registre et t_e le temps nécessaire en cas de défaut de cache. Pour simplifier, nous supposons ici que les conflits de bancs dans la mémoire ont un coût moyen. La validation de cette fonction avec nos donnnées expérimentales est impossible à cause des effets de λ sur t_c . Quand λ est faible et que le terme d'échange de données varie fortement, les vecteurs sont trop courts pour que les unités vectorielles donnent des t_c asymptotiques. A ces deux premiers effets bénéfiques s'ajoute la diminution relative de l'overhead de contrôle, ce qui ne permet pas de faire une analyse précise en faisant seulement varier λ.

Vu le gain de vitesse obtenu par rapport au programme de référence, on peut déduire qu'approximativement un tiers du temps d'exécution est passé à satisfaire des défauts de cache.

Si nous avions utilisé l'exemple développé dans ce paragraphe 7, le nombre d'accès aurait été divisé par une constante légèrement inférieure à 2 et le temps d'exécution aurait dû passer de 3 à 2.5. On peut donc conjecturer que la vitesse aurait encore augmenté de 20 %.

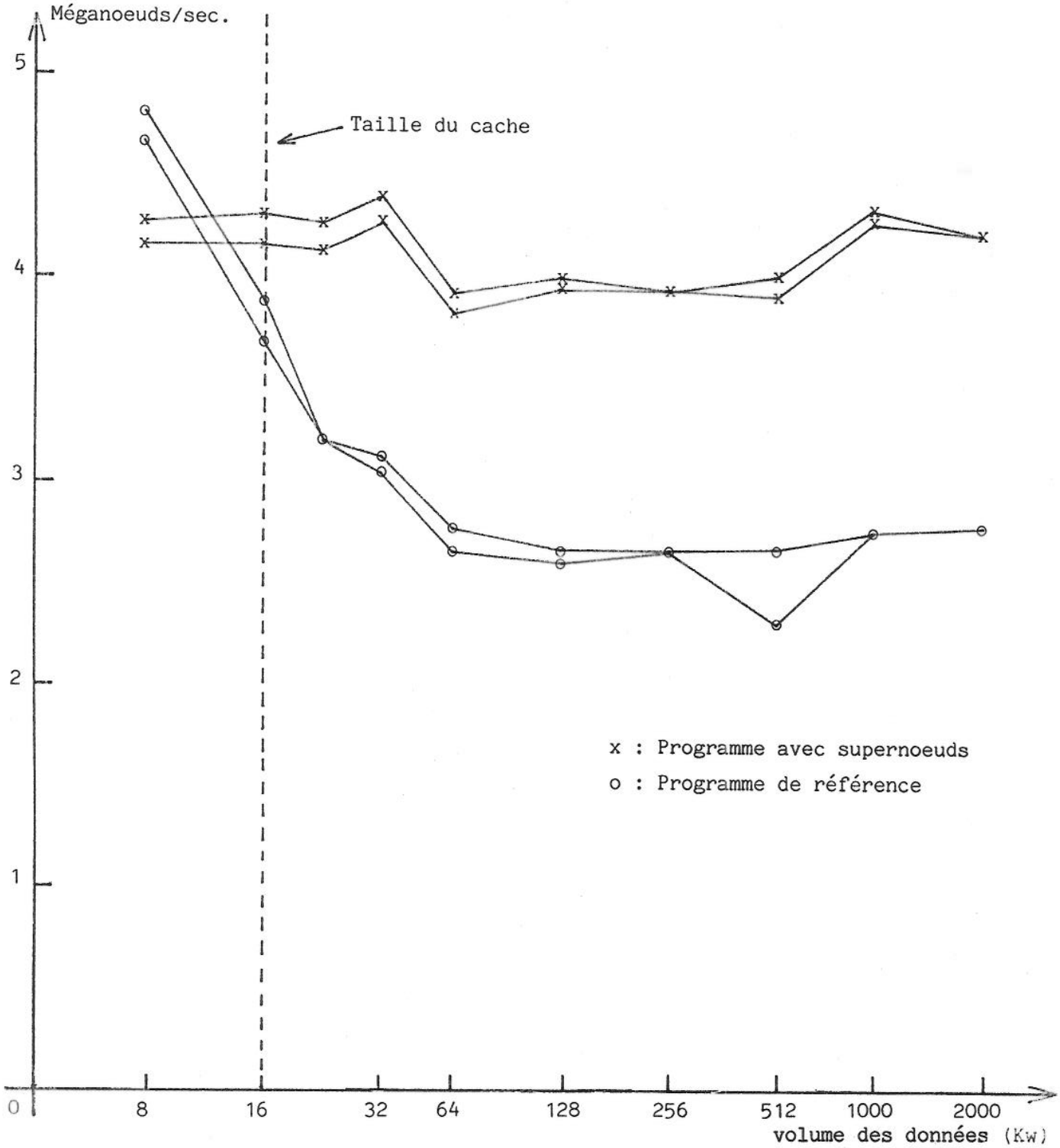
7.11. Conclusion

Ce chapitre nous a permis d'appliquer à un exemple l'ensemble des résultats exposés dans les chapitres 2 à 6: partitionnement, calcul des transferts de données, choix d'une direction d'itération globale et du facteur d'échelle, génération du code de contrôle global, génération du code de contrôle des supernoeuds complets et des supernoeuds frontières, modification des références aux variables et génération du code de copie.

L'écriture d'un algorithme de choix d'un partitionnement optimal n'est pas encore faisable car il faudrait d'abord faire des expériences pour voir sur plusieurs machines quelle est l'importance relative des divers facteurs énumérés à la fin du chapitre 6.

Par contre, une fois la matrice P et la direction d'itération locale $\vec{h}$ connues, tous les algorithmes nécessaires pour effectuer la transformation de programme partitionnant des boucles parfaitement imbriquées sont disponibles. Il semble même maintenant envisageable de générer le code de copie pour des machines comme les hypercubes, que nous avons laissé de côté dans ce travail, si le problème de la répartition des données sur les mémoires locales est résolu.

Enfin, l'unique expérience à laquelle nous avons pu procéder vu le manque de machines parallèles disponibles (pour les chercheurs en informatique) montre l'intérêt du partitionnement, même sur une machine sans mémoire locale disposant de primitives de synchronisation très rapides.



Vitesse de calcul en fonction du logarithme de la taille mémoire utilisée

- Figure 7.10 -

References

1. W. Abu-Sufah and A. D. Malony, "Vector Processing on the Alliant FX/8 Multiprocessor," *Int'l Conference on Parallel Processing*, pp.559-566 (Aug. 1986).

2. N. Halbwachs, "Détermination automatique de relations linéaires par les variables d'un programme," Thèse 3ème cycle, I.N.P., Grenoble (1979).

3. F. Irigoin and R. Triolet, "Supernodes and Alliant FX/8 Minisupercomputer," Tech. Rep. ENSMP-CAI-86-081, Ecole des Mines de Paris (Aug. 1986).

4. R. Triolet, "Contribution à la parallélisation automatique de programmes," Thèse de Docteur-Ingénieur, Université Paris VI (Déc. 1984).

8. COMPARAISONS AVEC D'AUTRES METHODES

Il n'est pas possible d'étudier toutes les méthodes qui ont été proposées pour découper un programme en tâches et/ou en améliorer la localité des références en en réordonnançant l'exécution. Nous laisserons de côté une première famille qui consiste essentiellement à dérouler *suffisamment* les boucles à la compilation et à travailler sur un graphe d'exécution défini en extension. Les travaux de D'Holander[9] et, dans une moindre mesure, ceux qui sont liés au compilateur Bulldog[10] et aux machines VLIW en sont deux illustrations.

La deuxième famille est constituée de méthodes qui peuvent être appliquées automatiquement par un compilateur, et qui utilisent le graphe de dépendance. La plus connue de ces méthodes est la méthode des vagues, aussi appelée méthode hyperplane. Mais le simple échange de boucles est déjà susceptible de diminuer considérablement le temps d'exécution. Les travaux d'Abu-Sufah sur l'adaptation des programmes à la mémoire virtuelle trouvent aussi une nouvelle jeunesse puisqu'on peut les reprendre pour les machines à cache ou à mémoire locale. Récemment, J.K. Peir a proposé plusieurs partitionnements de programmes et un outil de simulation permettant de rechercher le meilleur partitionnement. Sa méthode est donc semi-automatique. Enfin, il est impossible de ne pas mentionner les travaux effectués sur la systolisation d'algorithmes tellement ils sont proches, par les hypothèses et les méthodes, de la restructuration automatique de programmes.

Finalement, la troisième famille de travaux que nous allons étudier est constituée d'études manuelles d'un ou de plusieurs algorithmes particuliers remaniés ou partitionnés pour les adapter à une machine particulière comme Cm* ou l'Alliant FX-8, ou de modélisation du comportement des algorithmes visant à évaluer le nombre maximum de processeurs utilisables. Cette troisième famille est très large et de nombreux travaux dignes d'intérêt sont simplement cités faute de place et de temps.

8.1. Méthode des vagues et méthode hyperplane

Ces méthodes ont été développées depuis très longtemps sous des noms divers. Dans l'ordre chronologique, Karp[18] , Muraoka[28] , Lamport[20] (Hyperplane and Coordinate Methods), Kuhn[19] (Loop Reindexing), Wolfe[40] (Index Set Shifting) et Cytron[8] (DO ACCROSS) ont étudié les possibilités de parallélisation que donnaient des vecteurs de dépendance constants, aussi appelés récurrences uniformes. Le calcul des vecteurs de dépendance varie d'un auteur à l'autre, ainsi que les conditions d'application. Le principe de base reste le même mais le problème de la génération de code tient une place de plus en plus importante au fil des années. Les vagues aux formes irrégulières ont été explorées à fond dans le projet Vesta[34] alors qu'elles avaient déjà été remplacées ailleurs par des hyperplans, ensuite réduits eux-mêmes à un unique hyperplan, au prix d'une diminution du parallélisme, pour ne pas faire dépendre la génération de code de la forme du domaine d'itération. La difficulté de génération automatique statique de code en vagues se conçoit bien à la vue de la figure 8.1.a. Et elle augmente encore s'il faut définir des instructions vectorielles.

Malgré la simplification considérable qu'apporte l'introduction d'hyperplans, la génération de code est restée problématique et Kuhn, Wolfe et Cytron ont tous proposé des améliorations sur ce plan, soit en lisibilité, soit en facilité. Il faut cependant noter que les DO ACCROSS de Cytron sont plus généraux mais ne peuvent être utilisés que par une machine parallèle et non par une machine vectorielle.

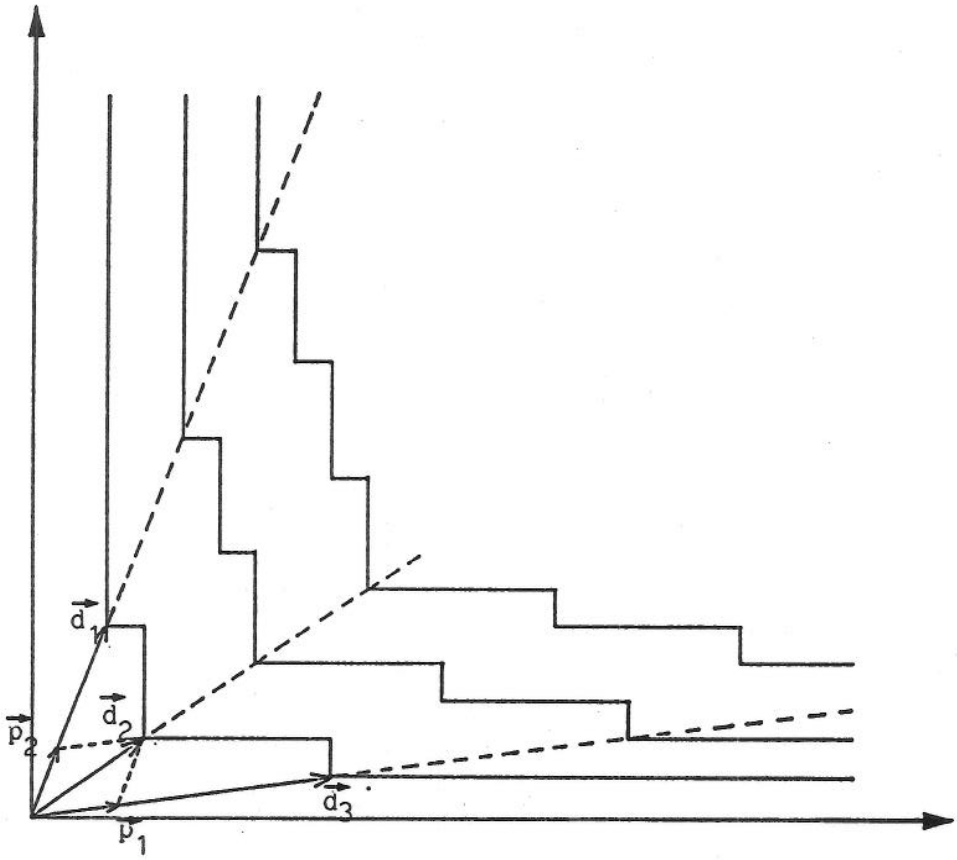
Ces méthodes, reposant toutes sur la longueur des vecteurs de dépendance et sur l'angle d'ouverture du cône de dépendance, ne peuvent être utilisées que si notre partitionnement en supernoeud peut aussi l'être. Mais elles n'apportent pas d'amélioration de localité. Dans le cas des essais effectués sur l'Alliant, l'hyperplan était réduit à une droite et correspondait justement à la boucle vectorielle interne: la méthode hyperplane ne modifiait donc pas le programme de référence.

Lors de l'optimisation de la taille des supernoeuds, de nombreuses contraintes doivent être observées. Si une contrainte de taille minimale est saturée, on retrouve des supernoeuds à un noeud et les fronts deviennent des hyperplans. La technique que nous avons développée permet d'obtenir de nouveaux résultats car les travaux précédents se sont limités à une optimisation en dimension 2, appliquée itérativement sur les paires de boucles successives, en commençant par la paire la plus interne. Kuhn montre par un contre-exemple (page 86 et suivantes) que cette itération ne donne pas nécessairement le résultat optimal, bien que cela ne se soit jamais produit sur les exemples réels qu'il a traités.

Son contre-exemple, illustré figure 8.1.b, est basé sur les vecteurs de dépendance suivants:

$$\vec{d}_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad \vec{d}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{d}_3 = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$$

En appliquant sa méthode, Kuhn trouve successivement deux droites et les combine en un hyperplan. Le temps d'exécution résultant est:



Exemple de vagues

- Figure 8.1.a -

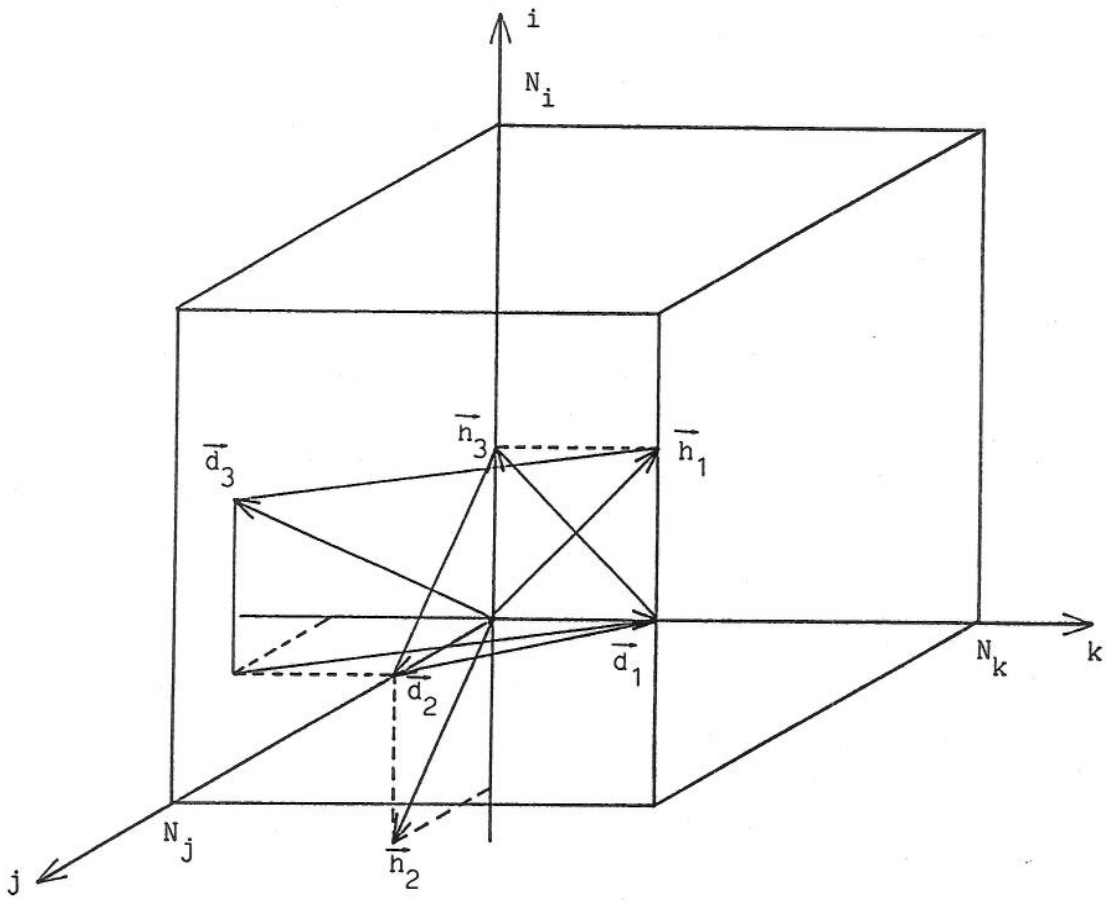
$$T = N_i + N_j + N_k$$

en supposant que l'on dispose d'un nombre illimité de processeurs, ce qui est une hypothèse implicite de Kuhn. N_i est la borne supérieure de la boucle externe sur i, et la même convention vaut pour les boucles sur j et k, la dernière étant la plus interne. Il montre ensuite qu'un autre plan permet d'obtenir le temps d'exécution T' :

$$T' = 2\,N_j + N_k$$

qui peut être plus court si $N_j \leq N_i$.

En utilisant les bases P et H, ces résultats s'obtiennent mathématiquement. Comme les vecteurs $\vec{d}_1$, $\vec{d}_2$ et $\vec{d}_3$ forment une base, nous pouvons directement poser (cf. 2.4.1):



Contre-exemple de Kuhn

- Figure 8.1.b -

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & -1 \end{pmatrix} \qquad det(P) = -1$$

D'après la valeur de $det(P)$, chaque supernoeud ne comprend qu'un noeud et notre partitionnement est équivalent à la méthode hyperplane. Toutes les coordonnées de P sont entières et le partitionnement est régulier (cf. 2.4.3). En inversant et en transposant P, on obtient la base duale:

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

L'ensemble des vecteurs de direction d'itération possibles constitue le cône entier strictement positif généré par H:

$$\vec{h} = \lambda\vec{h}_1 + \mu\vec{h}_2 + \nu\vec{h}_3 = \begin{pmatrix} \lambda-\mu+\nu \\ \mu \\ \lambda \end{pmatrix} \qquad \lambda, \mu, \nu \in \mathbf{N}^*$$

Enfin, en prenant le vecteur critique le plus évident avec un domaine d'itération parallélépipédique $(N_i, N_j, N_k)^t$, le temps d'exécution est:

$$T = (\lambda-\mu+\nu)\, N_i + \mu\, N_j + \lambda N_k$$

Une simple dérivation montre que λ et ν doivent être minimisés, ce qui donne $\lambda=\nu=1$, mais que la sensibilité pour μ dépend du signe de $N_i - N_j$. Le contre-exemple de Kuhn est obtenu pour $\mu=2$, ce qui donne d'ailleurs la solution optimale du problème, si $N_i - N_j$ est positif. Les différents vecteurs utilisés et les intersections avec le cube unité des deux plans proposés sont représentés figure 8.1.b.

Avec une grande valeur de μ, le temps ne devient pas négatif mais (N_i, N_j, N_k) n'est plus un vecteur critique. Il faut le remplacer par $(-N_i, N_j, N_k)$ et changer la formule donnant T.

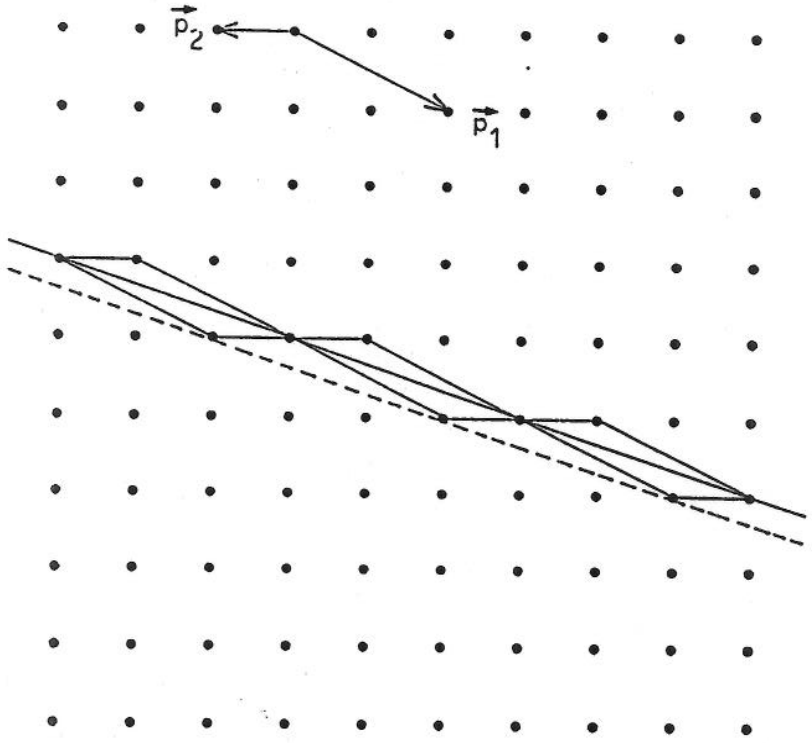
Quelle que soit la pente rationnelle de l'hyperplan, il existe une infinité de matrices de partitionnement unimodulaires produisant des fronts confondus avec cet hyperplan (cf. figure 8.1.c). Cela est dû au fait que la normale à l'hyperplan est une direction d'itération globale valide. La méthode hyperplane est donc incluse dans notre technique de partitionnement. Le passage des vecteurs de dépendances au cône de dépendance permet de lever la condition d'uniformité des récurrences tout en obtenant des résultats plus fins qu'avec l'Execution Ordering Vector de Wolfe[40].

Notons cependant que cela n'est pas vrai pour un ordonnancement hyperplan *tronçonné*. On peut l'approcher soit en choisissant une matrice de partitionnement très ouverte, ce qui laisse quelques *trous* dans l'hyperplan, soit en utilisant l'hyperplan lui-même pour partitionner le domaine d'itération, ce qui donne un front différent de l'hyperplan. Les exécutioons de certains noeuds sont retardés dans les deux cas (cf figure 8.1.d).

La méthode de partitionnement que nous proposons étant fondée sur des conditions moins restrictives que la méthode hyperplane, nous pouvons l'appliquer aux exemples que cite Kuhn: recherche de valeurs propres par l'algorithme de Jacobi (dimension 3), un algorithme de moindres carrés, un algorithme de réduction de largeur de bande et les méthodes SOR à 5 et 9 points.

8.2. Echange de boucles

L'échange de boucles est une transformation de programme ancienne et relativement simple quand elle est limitée à deux boucles adjacentes. Elle ne peut être appliquée que si les dépendances vérifient des conditions d'échange. Elle est utilisée en vectorisation quand une boucle parallèle englobe une boucle séquentielle pour rendre le parallélisme exploitable par une machine vectorielle. Elle est aussi utilisée pour améliorer la localité des données, en particulier dans les environnements à mémoire virtuelle. Un exemple présenté en 2.2.2 montrait par exemple qu'une inversion de boucle permettait de garder une



Simulation de la méthode hyperplane par partitionnement

- Figure 8.1.c -

valeur en registre.

Cette méthode peut aussi être vue comme un cas particulier de méthode hyperplane, en limitant les hyperplans parallèles potentiels aux seuls hyperplans définis par les vecteurs de base.

Allen et Kennedy en ont sans doute fait l'étude la plus générale dans[4] , où, partant du cas de la multiplication de matrice A=B $\times$ C, ils montrent que les objectifs d'un paralléliseur ou même d'un simple vectoriseur dépendent fortement de la machine cible. En effet, les dépendances doivent être éloignées du coeur des boucles pour obtenir du parallélisme mais doivent aussi être conservées autant que possible à un bas niveau pour qu'on puisse réutiliser les mêmes données, réutilisation qu'impliquent les dépendances. Les dépendances impliquent aussi des synchronisations si le programme est exécuté sur une machine parallèle MIMD: le nombre de synchronisations sera minimisé si les dépendances sont maintenues au coeur des boucles, une itération du coeur étant exécutée par un seul processeur, donc sans besoin de synchronisation.

Ces objectifs contradictoires conduisent Kennedy et Allen à faire deux échanges de boucles séparés par une phase de strip mining. Le premier échange permet d'obtenir du parallélisme de nature vectorielle,

le strip mining permet de doser la quantité de parallélisme qu'on va garder, et le deuxième échange permet de faire redescendre les dépendances qui vont améliorer la localité sans perturber le petit peu de parallélisme dont on a besoin (64 sur un Cray). Voici le résultat pour un Cray:

```
      DO 100 J=1,128
        DO 100 I=1,128,64
          DO 100 K=1,128
            C(I:I+63) = C(I:I+63) + A(I:I+63,J)*B(K,J)
100   CONTINUE
```

L'effet du strip mining est équivalent à la constitution d'un supernoeud mais l'optimisation qui suit dépend de l'héritage qui peut être propagé de supernoeud en supernoeud. L'absence d'un deuxième niveau de parallélisme dans les machines envisagées par les auteurs simplifie le traitement de l'héritage.

Leur étude ne prend pas en compte les autodépendances sur les matrices B et C et depuis lors le problème de la multiplication de matrice a été étudié en tant que tel, dans un environnement multiprocesseur vectoriel, par une équipe du Center for Supercomputing Research and Development. Nous étudierons le résultat de leurs travaux en 8.6.

Enfin, il faut noter que l'étude des échanges de boucles est limitée aux deux boucles les plus internes dans le projet PFC[3] , où la méthode de tests d'échangeabilité développée par Kennedy et Allen a été implémentée, sans doute pour respecter l'objectif de performance de ce projet.

8.3. Adaptation des programmes à la mémoire virtuelle

Ce problème s'est posé dès l'apparition de machines à mémoire virtuelle dans les années 1960. La recherche d'un algorithme de remplacement de page minimisant les défauts de page a conduit à étudier le comportement des programmes vis-à-vis de la mémoire.

Les premiers travaux ont été basés sur des traces réelles (une trace est une liste d'accès à la mémoire) mais leur production et leur exploitation étaient si coûteuses que des modèles stochastiques ou déterministes ont été ensuite employés[36] . Quoique des résultats intéressants aient été obtenus, surtout pour les références aux instructions[6, 2] , Abu-Sufah montre dans son PhD[1] , qu'il vaut mieux partir du code source que des traces qu'il engendre.

Il propose un modèle de boucle élémentaire appelé ELM dont on peut analyser facilement le comportement puis montre que de nombreux cas de boucles plus complexes peuvent être ramenés à ce modèle par des transformations de programme syntaxiques. Quelques cas résistent à toute transformation et leur exécution sur une machine à mémoire virtuelle ne peut être que catastrophique. Quand elles peuvent être appliquées, ces transformations conduisent non seulement à des programmes dont le comportement est facile à modéliser mais encore à des programmes qui s'exécutent efficacement, plus efficacement que le programme originel.

Plusieurs transformations sont appliquées d'office pour supprimer des dépendances susceptibles d'interdire la réorganisation du code:

- normalisation des boucles;
- If Pattern matching;
- renommage de scalaire;
- induction variable substitution and subscript cleaning;
- type - A if removal from Do-loop.

Ensuite, les boucles sont distribuées suivant les composantes connexes du graphe de dépendance† et/ou fusionnées pour ne traiter que le minimum de données simultanément (distribution) et pour essayer d'effectuer un maximum de calcul sur les données présentes en mémoire principale (fusion).

Pour diminuer la taille des composantes connexes, une expansion de scalaire modifiée et la scalar forward substitution sont envisagées. La deuxième de ces transformations est préférée quand elle est valide parce qu'elle n'augmente pas l'espace mémoire nécessaire.

Enfin, pour les tableaux multidimensionnels, le stockage par sous-matrice (une sous-matrice par page) et l'indexage par page sont proposés comme dans[23] où le coût supplémentaire d'adressage est présenté comme un inconvénient à peser soigneusement en fonction de la taille de la matrice traitée. L'indexage par page et ce mode de stockage donnent des résultats **équivalents** à la définition de supernoeuds hypercubiques. Le cache ou le code de copie se chargent de modifier temporairement le mode de stockage des tableaux dans notre cas.

Pour augmenter le nombre de cas qu'il sait traiter, Abu-Sufah propose aussi de transformer les boucles imbriquées contenant des instructions à différents niveaux en des boucles parfaitement imbriquées. Appliquée au cas de l'élimination de Gauss, cette transformation produit des affectations conditionnelles qui provoqueraient, pour des supercalculateurs, un overhead de test et du code vectoriel inefficace. Le coût d'un cache miss n'est pas du même ordre de grandeur que celui d'un défaut de page. Il faudrait donc utiliser une technique sophistiquée de IF removal basée sur un calcul interprocédural de prédicats pour obtenir des assignations simples, vectorisables sans problème, dans le code correspondant à un supernoeud plein.

En conclusion, notre méthode étant basée sur les vecteurs de dépendance et non sur des vecteurs de proximité ne peut pas donner d'aussi bons résultats qu'une méthode spécifique au problème de la mémoire virtuelle. Par exemple, une initialisation de tableau par lignes ou une transposition de matrice ne seront pas modifiées, ce qui est normal puisqu'on ne peut rien faire pour améliorer le fonctionnement d'un cache ou d'une mémoire locale dans ces deux cas. Mais dans le cas où les vecteurs de dépendances lient des noeuds qui réfèrent des éléments de tableaux proches l'un de l'autre, nous obtenons une

†Il s'agit dans ce cas du graphe de dépendance sur les instructions et non sur les itérations d'instructions.

augmentation de localité et un réordonnancement des calculs similaires.

Malgré les excellents résultats exposés dans le PhD d'Abu-Sufah, sa méthode n'a pas été implémentée dans un compilateur commercial à notre connaissance, sans doute à cause des temps de restructuration considérables entraînés par l'utilisation de Parafrase. Ces temps ne sont pas indiqués par l'auteur.

Les travaux sur l'optimisation de programmes exécutés dans un environnement à mémoire virtuelle ont été peu nombreux ces dernières années. Mais les langages orientés objets et les volumes de code considérables auxquels ils conduisent ont renouvelé le problème[37] . De même, les performances désastreuses des garbage collectors utilisés avec LISP redonnent un peu d'actualité à ce vieux problème. Mais, dans les deux cas, les relations entre objets sont données par des graphes quelconques et non par les graphes réguliers des programmes scientifiques.

8.4. Partitionnement avec synchronisation de J.-K. Peir

Deux méthodes de partitionnement sont proposées dans le PhD[29] et diverses publications[31, 30] de J.-K. Peir: le partitionnement par blocs et le partitionnement entrelacé. Dans le premier cas, chaque processeur traite N/p itérations successives d'une boucle. N est le nombre d'itérations de la boucle. Ce premier partitionnement favorise d'après l'auteur la réutilisation des données et limite le nombre de synchronisations. Nous le voyons comme un cas dégénéré de supernoeud.

Dans le second cas, les itérations i telles que $mod\,(i\,,p\,)=k$ où p est le nombre de processeurs sont exécutées sur le k-ième processeur. Le degré de parallélisme est plus important mais le terme de *partitionnement* semble impropre pour caractériser un simple DOALL exécuté par plusieurs processeurs.

Le partitionnement selon un hyperplan généré par la méthode hyperplane est exclu à cause de l'overhead de contrôle *inacceptable* qui en résulte. Il nous semble que cet overhead est surestimé.

Dans le cas de boucles imbriquées, chaque boucle peut être partitionnée d'une manière différente et la décomposition de l'ensemble des boucles est définie par l'*aspect ratio*, AR, qui définit le nombre de partitions produites par chaque boucle. Le choix du AR optimal est un compromis entre la réutilisation des données et l'overhead de synchronisation. Plusieurs AR sont sélectionnés pour les deux types de partitionnements d'une manière heuristique et le choix final s'effectue par simulation.

Enfin, Peir montre que cette technique peut être appliquée aux systèmes à communication par messages (i.e. sans mémoire globale).

Aucun résultat expérimental, obtenu sur une machine réelle, n'est présenté dans les travaux que nous avons pu consulter et il est difficile de comparer cette méthode et la nôtre. Il est sûr qu'elle ne permet pas de réutiliser les données aussi bien qu'une méthode bidimensionnelle mais son champ d'application est plus large. Elle n'impose de restrictions ni sur la dimension du problème ni sur le caractère parfait des boucles imbriquées.

8.5. Systolisation

De nombreux chercheurs ont étudié la dérivation automatique d'un algorithme ou d'une machine systolique à partir de programmes Fortran[24, 25, 26] ou à partir d'équations récurrentes uniformes[18, 32, 27] . Nous avons laissé de côté la réalisation d'un compilateur[15] pour le processeur systolique Warp[5] , bien que son architecture soit presque MIMD (chaque processeur élémentaire contient son code), car les dépendances entre modules doivent être exprimées dans un langage spécial, W2, par des primitives send/receive et non par des références à la même valeur. Nous sommes parti de l'étude la plus récente, la thèse de C. Mongenet[27] , pour établir une comparaison entre les travaux effectués en parallélisation et en conception de réseaux systoliques.

Ces deux domaines possèdent les mêmes notions de base, seul change le vocabulaire:

- Un domaine d'itération convexe, fini ou non dans le cas systolique.
- Des vecteurs de datadépendance constants en systolique. D'autres types de dépendances sont pris en compte en parallélisation et la constance des vecteurs de dépendance n'est pas indispensable. Dans tous les cas, on aboutit à un cône de dépendance défini par des rayons. Ceci nous permet d'envisager le partitionnement des algorithmes systoliques.
- Un espace d'itération de dimension 2 ou 3.
- Identité des calculs effectués dans chaque cellule systolique (sauf avec le WARP) et des calculs effectués par chaque noeud.

Deux problèmes, communs aux deux domaines, doivent être résolus:

- Trouver une fonction temporelle pour ordonner les calculs dans le temps.
  Nous l'avons appelée direction d'itération, globale ou locale. Les critères (cf. § 2.3.5) que nous utilisons sont plus faibles que ceux de P. Quinton[32] parce que nos domaines d'itérations sont finis et que nous n'avons pas besoin d'obtenir un nombre de supernoeuds égal au nombre de processeurs. Nous n'avons pas non plus besoin de garantir que la fonction temporelle est positive sur tout le domaine d'itération. Il n'existe pas de critère d'optimalité unique pour choisir une des fonctions valides. Le plus utilisé semble être celui de la méthode hyperplane, qui découle de l'hypothèse d'un nombre infini de processeurs. Il consiste à minimiser les coefficients de la fonction temporelle car cela augmente la densité des calculs sur chaque front, diminue le nombre total de fronts et donc le temps d'exécution. L'adjonction des fonctions quasi-affines que propose P. Quinton ne nous semble pas intéressante pour notre partitionnement car les cas où elles pourraient être utiles sont exclus (cf. § 2.2.3).

- Garantir la localité des accès.

  En systolique, parce qu'une cellule ne peut accéder que quelques cellules voisines (C. Mongenet propose d'étendre le systolique pour incorporer la diffusion de donnnées). En parallèle, pour utiliser efficacement la hiérarchie mémoire.

Un certain nombre de problèmes sont propres au systolique:

- Trouver une exécution causale.

  Il faut vérifier que les définitions équationnelles sont explicites, c'est-à-dire qu'il n'y a pas de cycle dans les dépendances. Ce problème n'existe pas pour ceux qui dérivent leur réseau systolique à partir de programmes Fortran.

- Allouer les calculs aux cellules.

  Le problème ne se pose pas en parallélisation du fait de la présence d'une mémoire locale mais P. Leca[21] a utilisé la technique systolique de P. Quinton pour écrire du code destiné à LCAP 1 (cf. § 1.1.1.4) n'utilisant pas de mémoire globale. Le problème de l'allocation pourrait être transposé aux tableaux de données. Par exemple, le passage de BARRE(T,L) à BARRE1(L) et BARRE2(L) qui a été effectué en § 7.10 pour économiser de la mémoire ressemble au repliement effectué pour affecter les calculs sur des cellules. En reprenant les mêmes fonctions modulo (ausi appelées quasi-linéaires), il suffit de vérifier que l'on ne crée pas de nouvelles dépendances.

- Supprimer la diffusion des données.

  Toutes les variables doivent être indicées par des formes linéaires formant un espace dont la dimension est égale à celle de l'espace d'itération. Cela pose des problèmes quand on veut partir d'un programme Fortran car il faut ajouter des pseudo-indices un peu arbitrairement. Nous avons rencontré le même problème en § 2.2.2, mais uniquement du fait d'un manque de dépendance. Les tenants de la spécification par des équations non procédurales en profitent pour critiquer toute approche liée à Fortran mais ils imposent cette même condition à leurs équations ou bien ils élargissent la définition du systolique.

Problèmes propres au partitionnement:

- Trouver une base de partitionnement.

  Une fois connu l'ensemble des directions d'itérations globales possibles, il faut encore en trouver une réalisation optimale par supernoeuds. Le problème n'est résolu pour le moment qu'en dimension 2 (cf. § 6.2.5).

- Regrouper les calculs qui doivent être exécutés dans le même processus.

  Même si l'on identifie processus et processeur, le partitionnement par supernoeud ne s'obtient pas par des fonctions d'allocations linéaires mais par des fonctions quasi-affines (les coefficients sont fractionnaires).

En conclusion, on peut dire que systolisation et parallélisation sont deux domaines très proches. Partir d'équations plutôt que d'un vieux langage impératif procédural comme Fortran peut sembler séduisant à première vue mais ne semble modifier en rien le problème. Il n'y a d'ailleurs pas un système unique d'équations pour spécifier un problème et les systèmes d'aide à la conception d'algorithmes systoliques proposent des outils de transformation d'équations... La recherche de la causalité entre équations et la recherche dépendances de toute nature entre instructions aboutissent à définir un ordre partiel sur les calculs dans tous les cas. C'est cet ordre qui est fondamental.

8.6. Etude manuelle de stratégies de décomposition

D. Vrsalovic et al. ont présenté dans [39] une étude systématique des effets de partitionnements 2D sur les temps d'accès aux données et les temps de calcul pour trois algorithmes d'analyse numérique (multiplication de matrice, résolution de l'équation de Poisson sur une grille carrée, équations différentielles linéaires).

Dans ces algorithmes, on étudie le partitionnement d'un **plan** de calculs parallèles. L'analyse des dépendances ne joue donc aucun rôle. Toute l'optimisation est basée sur la nature des accès mémoire, locaux ou globaux.

Le temps d'exécution est décomposé en temps de calcul T_p, qui inclut le temps CPU nécessaire pour effectuer les copies et qui est évalué en supposant que tous les accès ont lieu en mémoire locale, et en temps d'accès T_a. Le paramètre de base est le nombre de processeurs N et non la taille des partitions. Comme tous les calculs peuvent être effectués en parallèle, il suffit de diviser le domaine d'itération par le nombre de processeurs pour trouver la taille des partitions.

Les algorithmes sont caractérisés par des couples de fonctions:

$$\left(\frac{T_p}{t_p(N)}, \frac{T_a}{t_a(N)} \right)$$

T_p et T_a sont relatifs à l'exécution avec un seul processeur tandis que t_p et t_a correspondent à l'exécution par N processeurs. La nature de ces fonctions permet de prévoir l'existence ou non d'un nombre de processeurs optimal.

Le calcul de ces fonctions est effectué **manuellement** et plusieurs stratégies d'utilisation de la mémoire locale sont étudiées pour chaque algorithme. Rien n'est prévu pour automatiser ces calculs à partir de la forme des partitions et des références.

La machine cible est le multiprocesseur Cm* (cf. § 1.1.1.5), malgré ses unités arithmétiques en nombres entiers codés sur 16 bits. Le temps d'accès intercluster de cette machine explique l'intérêt des copies en mémoire locale: dès qu'une donnée doit être utilisée plusieurs fois, il vaut mieux la copier. Seul parmi les algorithmes présentés, celui de la multiplication de matrice permet de réutiliser des données un

grand nombre de fois. Le modèle utilisé pour établir les différentes fonctions et calculer le nombre de processeurs optimal ou la meilleure utilisation de la mémoire locale est validé par simulation et non par expérience. Le résultat des validations est satisfaisant.

Enfin, plusieurs formes de partitions, le carré, le triangle et l'hexagone, sont essayées pour la résolution de l'équation de Poisson. L'hexagone donne de meilleurs résultats grâce à son faible rapport périmètre / surface. Remarquons que les triangles peuvent s'obtenir avec un partitionnement par hyperplans contraint (cf. § 2.3.1) et que le polygone convexe qu'est le triangle se prête à notre méthode de génération de code de supernoeuds. Mais il faudrait aussi savoir balayer les deux ensembles d'origines des triangles. L'hexagone ne peut pas être obtenu avec un partitionnement par hyperplan, mais le code d'un supernoeud est aussi dérivable du système de contraintes définissant un hexagone.

La résolution de l'équation de Poisson a un domaine d'itération tridimensionnel qui est coupé en tranches par un test de convergence. L'application de notre méthode nécessiterait une adaptation de ce test aux fronts crénelés que produisent les supernoeuds.

8.7. Optimisation du produit de matrice sur l'Alliant FX/8

Le partitionnement du produit de matrice en plusieurs produits de matrices-blocs a été proposé depuis longtemps pour améliorer les performances sur les processeurs à mémoire virtuelle[23, 38] .Cette technique est aussi utile pour exploiter au mieux un cache *données*. W. Jalby et U. Meier[17] ont proposé un modèle permettant de choisir les tailles de blocs optimales pour l'Alliant FX/8 et ont fait de très nombreuses expériences pour valider les résultats. La bonne utilisation du cache et la minimisation des conflits d'accès à la mémoire globale qui en découle fait passer la vitesse d'exécution du produit de matrice de 9 Mflops avec l'algorithme standard vecteur $\times$ vecteur à environ 40 Mflops.

Leur méthode consiste d'abord à séparer l'optimisation du temps de calcul proprement dit et l'optimisation des échanges pour ne pas avoir un problème trop compliqué. La première étape leur permet de déterminer quelle version du produit de matrice produit les meilleurs résultats quand toutes les données sont en cache. C'est une étape que nous supposons faite, dans notre cas, par une passe préalable de paralléliseur (cf. § 1.3.2). La deuxième étape est consacrée à minimiser le nombre de défauts de cache sous l'hypothèse d'une politique de remplacement optimale. Enfin, la solution est prise dans l'intersection de l'ensemble des programmes satisfaisants pour les unités vectorielles et pour le nombre de processeurs élémentaires et de l'ensemble des solutions ne provoquant pas trop de défauts de cache.

Contrairement à ce que nous proposons en affectant chaque supernoeud à un processeur, Jalby et Meier utilisent les huit processeurs et leurs unités vectorielles pour chaque produit de matrice bloc. Il en résulte des contraintes plus fortes sur la taille minimale de leurs blocs, mais l'enchaînement purement séquentiel des calculs sur les blocs permet de prendre en compte sans difficulté ce que nous avons appelé le problème de l'*héritage*, la réutilisation des données préchargées pour un bloc par le bloc suivant. La différence entre les deux méthodes se voit dans l'évaluation du nombre de lectures et du volume du

cache.

Voici le code du produit de deux matrices-blocs A et B que nous obtiendrions à partir d'un programme naturel. Il diffère bien sûr du code optimal qui utiliserait soit une instruction de réduction vectorielle s'il en existe une efficace sur l'Alliant FX/8, soit un strip-mining de la boucle sur s et le transfert vers l'intérieur de la boucle sur t comme le suggère Kennedy (cf. § 8.2). La boucle séquentielle sur t se trouve à l'extérieur car c'est ce que donnerait une génération de code basée sur un vecteur d'itération local $\vec{h}$. Rien ne dit que ce choix justifié en dimension 2 le soit encore en dimension 3 et au delà. Nous reprenons les notations de [17] pour faciliter la comparaison:

DO t = 1, m_2
  DO r = 1, m_3
    DO s = 1, m_1
      $C_{s,r} = C_{s,r} + A_{s,t} * B_{t,r}$

Produit de deux matrices-blocs

Ce code correspond au code supernoeud qu'on obtiendrait en complétant l'unique dépendance du produit de matrice par les deux autres vecteurs de la base initiale (cf. § 2.2.2).

Il reste à ajuster m_1, m_2 et m_3 à la taille du cache pour garantir un nombre minimal de défauts de cache. On trouve dans la table suivante les volumes des régions accédées par chaque référence, le nombre de lectures sans cache (NLx), le nombre d'écritures, les données contenues dans le cache et leur volume.

Les volumes des régions accédées, qui correspondent exactement au nombre d'accès non satisfaits par le cache (cf. § 4.6.1) puisque la taille des supernoeuds est choisie pour éviter les accès multiples à la même case mémoire, peuvent être calculés par la méthode duale (cf. § 4.2.2) car toutes les références sont de dimension N-1:

$$\lambda P = \begin{pmatrix} m_2 & 0 & 0 \\ 0 & m_3 & 0 \\ 0 & 0 & m_1 \end{pmatrix} \quad \frac{1}{\lambda} H = \begin{pmatrix} \frac{1}{m_2} & 0 & 0 \\ 0 & \frac{1}{m_3} & 0 \\ 0 & 0 & \frac{1}{m_1} \end{pmatrix} \quad \lambda^N = m_1 m_2 m_3 \quad \vec{d}_A = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$$V_A = \lambda^N < \frac{\vec{1}_H}{\lambda} . \vec{d}_A > = m_1 m_2$$

On retrouve bien le résultat trivial qu'on attendait pour un partitionnement aussi simple.

Le calcul du volume de la zone écrite est sans intérêt parce que cette zone est déjà prise en compte comme zone lue: les deux références à C sont identiques.

Nous n'avons pas développé en § 4.3 de méthode valide pour calculer les volumes de cache avec des dimensions supérieures à 2 mais la multiplication de matrice entre dans le cas des expressions d'index mono-indicées. Le volume se calcule en oubliant les expressions d'index dont l'indice n'est pas dominé par un indice n'appartenant pas aux expressions d'index de ce tableau. Un indice est *dominé* par un autre si la boucle du premier est incluse dans la boucle du second. B devient donc un scalaire, A est monodimensionnel (une colonne) et C reste bidimensionnel. Le calcul s'effectue comme si le processeur élémentaire était scalaire car les accès vectoriels sont aussi séquentiels.

La table est organisée en partant du corps de la boucle puis en ajoutant les boucles englobantes en partant de l'intérieur.

	corps	DO s=1,m_1	DO r=1,m_3	DO t=1,m_2
$C_{s,r}$	1	m_1	m_1m_3	m_1m_3
NLC	1	m_1	m_1m_3	$m_1m_2m_3$
$A_{s,t}$	1	m_1	m_1	m_1m_2
NLA	1	m_1	m_1m_3	$m_1m_2m_3$
$B_{t,r}$	1	1	m_3	m_2m_3
NLB	1	1	m_3	m_2m_3
$C_{s,r}$	1	m_1	m_1m_3	m_1m_3
NEC	1	m_1	m_1m_3	$m_1m_2m_3$
Cache	$\emptyset$	$\emptyset$	$A_{*,t}$	$A_{*,t}$, $B_{t,r}$ et $C_{*,*}$
Volume	0	0	m_1+1	$m_1m_3 + m_1 + 1$
Lectures	3	$2m_1 + 1$	$m_1m_3 + m_1 + m_3$	$m_1m_2 + m_2m_3 + m_3m_1$

Etude du cache et des accès

Soit n_1, n_2 et n_3 les trois dimensions de la matrice complète. Le nombre de supernoeuds est:

$$\frac{n_1 n_2 n_3}{m_1 m_2 m_3}$$

ce qui donne un nombre d'échanges l :

$$l = n_1 n_2 n_3 \left(\frac{1}{m_1} + \frac{1}{m_2} + \frac{1}{m_3} \right)$$

sous la contrainte de taille du cache:

$$m_1 m_3 + m_1 + 1 < CS$$

En supposant le cache saturé, en négligeant la constante 1 et en posant $K = n_1 n_2 n_3$, on obtient:

$$m_1 = \frac{CS}{1 + m_3}$$

$$l = K \left(\frac{1}{m_2} + \frac{1}{m_3} + \frac{1 + m_3}{CS} \right)$$

$$\frac{\partial l}{\partial m_3} = K \left(-\frac{1}{{m_3}^2} + \frac{1}{CS} \right) = 0$$

Les dimensions optimales sont donc:

$$m_1 = \frac{CS}{1 + \sqrt{CS}}$$
$$m_2 = n_2$$
$$m_3 = \sqrt{CS}$$

Dans le cas de l'Alliant FX/8 qui a un cache global de 16 KW à partager entre ses huit processeurs élémentaires (CS =2048), on obtient:

$$m_1 = 44.277 \qquad m_2 = n_2 \qquad m_3 = 45.254$$

Le nombre de lectures qui en résulte est:

$$l = n_1 n_2 n_3 \left(\frac{1 + \sqrt{CS}}{CS} + \frac{1}{n_2} + \frac{1}{\sqrt{CS}} \right) \approx \frac{2}{\sqrt{CS}}$$

quand n_2 est grand par rapport à $\sqrt{CS}\ /2 = 22.627$

$$l = 0.044\ n_1 n_2 n_3$$

au lieu de $3 n_1 n_2 n_3$ en l'absence de cache ou avec un cache très mal utilisé comme ce serait le cas en

l'absence de blocs ou de supernoeuds.

Etant donné la contrainte que représentent les registres vectoriels, il faudrait prendre:

$$m_1 = 32 \qquad m_2 = n_2 \qquad m_3 = 64$$

ce qui donne $l = 0.046\ n_1 n_2 n_3$, assez proche de l'optimum continu. Ce coefficient est à diviser par 2 avec le quadruplement de la taille du cache qu'annonce Alliant.

Ce résultat peut ne pas sembler intéressant vis-à-vis de ceux qu'obtiennent Jalby et Meier puisqu'ils arrivent à des coefficients compris entre 0.016 et 0.020 et qu'ils montrent qu'une augmentation de ce coefficient conduit à une diminution de vitesse appréciable bien qu'il n'y ait pas de corrélation directe entre eux deux. Cependant, notre méthode assure l'essentiel du gain puisqu'elle fait passer le coefficient du nombre de lectures de 3 à 0.05 et elle est susceptible d'être automatisée, ne laissant à l'optimisation manuelle qu'un gain de 0.05 à 0.02.

La relation entre le modèle mathématique de Jalby et Meier et les résultats expérimentaux n'a pu être établie qu'en codant directement en assembleur pour éviter d'introduire des effets supplémentaires dus au générateur de code Fortran. Ils dérivent analytiquement leurs coefficients à partir d'un code plus parallèle que le nôtre qui oblige à garder davantage d'éléments en cache mais qui évite la division du cache global en 8 parties:

$$m_1 m_2 + m_1 m_3 < CS$$

Le nombre de lectures tient compte de l'ordre des boucles sur les produits de matrices-blocs pour amortir certains chargements de données sur un ensemble de produits partiels. Le calcul final d'une sous-matrice de C nécessite $m_1 n_2 + m_3 n_2 + m_1 m_3$ lectures et le nombre de telles sous-matrices est:

$$\frac{n_1 n_3}{m_1 m_3}$$

Ils en déduisent:

$$l = \frac{m_1 + m_3}{m_1 m_3} n_1 n_2 n_3 + n_1 n_3$$

Partant de deux équations différentes des nôtres, il n'est pas étonnant qu'ils arrivent à des résultats autres.

Nous n'avons malheureusement pas eu le temps de comparer expérimentalement notre partitionnement avec le leur sur l'Alliant FX/8 pendant la courte période que nous avons passée au CSRD. Il aurait fallu coder en assembleur et étudier l'optimisation de la partie calcul, ce qui n'est pas le propos de ce travail.

Qualitativement, la méthode de Jalby et Meier permet d'exploiter deux niveaux de parallélisme dans les supernoeuds. La granularité de leurs supernoeuds (i.e. blocs) est plus importante que celle des nôtres mais ils n'ont pas besoin d'en avoir des paquets de huit pour occuper les processeurs. Ils génèrent donc moins de contraintes pour l'usager qui dimensionne ses tableaux ou fixe les coefficients de discrétisation. Comme eux, nous n'obtenons pas un découpage carré des sous-matrices puisque nous prenons $m_2 = n_2$ pour réexploiter au maximum les $C_{s,r}$ qui se trouvent dans le cache. Ce n'est pas forcément la meilleure chose à faire puisqu'un registre vectoriel pourrait jouer le même rôle, mais nous avons laissé les registres hors du champ de ce travail sauf en § 7.3.

Etant donné la sensibilité de l'Alliant FX/8 aux mouvements de données, l'équipe "applications" du projet CEDAR a publié de nombreux rapports sur les problèmes de localité dans divers algorithmes d'analyse numérique.

8.8. Autre travaux d'optimisation manuelle de code ou d'algorithmes

Le problème de l'adaptation des programmes et des algorithmes aux machines parallèles a toujours été un sujet fascinant pour les chercheurs. Mais le domaine a récemment explosé avec la multiplication commerciale des machines parallèles. Même en laissant de côté l'adaptation des algorithmes qui concerne principalement les numériciens et un peu les chercheurs en combinatoire, il n'est pas possible de recenser ce qui ne concerne que l'adaptation des programmes.

Le partitionnement de programmes et l'importance des mouvements de données dans les performances ont été étudiés très tôt à cause de la structure SIMD de l'Illiac IV. Gentleman[14] a montré par des calculs de complexité que les mouvements de données étaient prédominants dans la multiplication de matrice effectuée sur un multiprocesseur à nombre fixe d'interconnexions par processeur élémentaire. L'adjonction d'une facilité de diffusion de données ne supprime pas le goulot d'étranglement que constituent les échanges. Il faut mettre en oeuvre un réseau d'interconnexion global pour faire descendre la part des échanges par rapport aux calculs.

Lint et Agerwala[22] ont proposé deux méthodes de partitionnements: la dichotomie, qui consiste à partir du programme complet et à la diviser par deux jusqu'à ce que cela n'apporte plus rien, et l'agglomération qui revient à mettre ensemble des calculs élémentaires tant que les communications entre blocs ne sont pas trop importantes. Ils donnent aussi des modèles qui doivent permettre d'estimer le temps d'exécution pour des machines synchrones ou asynchrones et pour différents réseaux d'interconnexion.

Gannon et Rosendale[13] ont proposé deux modèles pour prévoir le comportement d'algorithmes sur les multiprocesseurs vectoriels à mémoire globale et sur les multiprocesseurs massifs à mémoire locale uniquement. Ils calculent manuellement les volumes. Ils n'étudient pas l'influence d'une hiérarchie mémoire dans le premier cas et se consacrent à des calculs de complexité pour montrer l'influence du réseau d'interconnexion dans le second cas.

L'architecture hypercube a aussi entraîné des études sur le partitionnement des algorithmes au Caltech[12, 11, 33] , dans les universités avec les hypercubes d'Intel[35] , puis chez Floating Point Systems[16] .

M. Cosnard[7] propose de faire du partitionnement par blocs variables sur les domaines triangulaires pour conserver un degré de parallélisme constant. L'automatisation de cette idée est envisageable, mais elle serait difficile dans le cas des supernoeuds comme on l'a vu en § 2.3.3.

8.9. Conclusion

La méthode de partitionnement que nous proposons ne donne pas systématiquement des résultats optimaux. Elle ne permet pas d'obtenir des solutions aussi efficaces que celles qui ont été trouvées manuellement par Jalby et Meier et ne repose pas sur une analyse aussi fine que celles qu'a effectuées Vrsalovic. Mais elle semble pouvoir à la fois être automatisée à court terme et fournir des résultats intéressants par rapport aux optimisations manuelles. Accessoirement, elle englobe la méthode hyperplane qui n'en est qu'un cas particulier.

References

1. W. A. Abu-Sufah, "Improving the Performance of Virtual Memory Computers," Ph.D. Thesis, DCS Report No. UIUCDCS-R-78-945, University of Illinois at Urbana-Champaign (Jan. 1979).

2. M. S. Achard, J. Y. Babonneau, M. Carpentier, G. Morisset, and M. B. Mounajjed, "The clustering algorithms in the OPALE restructuring system," pp. 137-153 in *Proc. Int. Conf. Perf. Computer Installations* (1978).

3. J. R. Allen and K. Kennedy, "PFC: a Program to Convert Fortran to a Parallel Form," in *Supercomputers, Design and Application*, ed. K. Hwang (1982). COMPSAC, Tutorial

4. J. R. Allen and K. Kennedy, "Automatic Loop Interchange," *SIGPLAN'84 Symposium on Compiler Construction*, pp.233-246 (June 1984).

5. M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, K. Sarocky, and J. A. Webb, "WARP Architecture and Implementation," *13th Annual International Symposium on Computer Architecture*, pp.346-356 (June 1986).

6. J. Y. Babonneau, M. S. Achard, G. Morisset, and M. B. Mounajjed, "Automatic and General Solution to the Adaptation of Programs in a Paging Environment," *6th ACM Symposium on Operating Systems Principles*, pp.109-116 (Nov. 1977).

7. M. Cosnard, J. M. Muller, Y. Robert, and D. Trystram, "Communication Costs Versus Computation Costs in Parallel Gaussian Elimination," *Int. Workshop on Parallel Algorithms and Architectures*, pp.19-29 (April 1986).

8. R. G. Cytron, "Compile-Time Scheduling and Optimization for Asynchronous Machines," PhD Thesis, Report No. UIUCDCS-R-84-1739, University of Illinois at Urbana-Champaign (1984).

9. E. D'Hollander, "Automatic Decomposition and Scheduling of Sequential Programs for a Homogeneous Multiprocessor," Report, State University of Ghent (1985).

10. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge (1985).

11. G. C. Fox, "Matrix Operations on the Homogeneous Machine," Technical Report CALT-68-939, Caltech (Aug. 1982).

12. G. C. Fox, "Decomposition of Scientific Problems for Concurrent Processors," Technical Report CALT-68-986, Caltech (March 1983).

13. D. B. Gannon and J. V. Rosendale, "On the Impact of Communication Complexity on the Desing of Parallel Numerical Algorithms," *IEEE Transactions on Computers* **33**(12), pp.1180-1194 (Dec. 1984).

14. W. M. Gentleman, "Some Complexity Results for Matrix Computations on Parallel Processors," *Journal of the ACM* **25**(1), pp.112-115 (Jan. 1978).

15. T. Gross and M. S. Lam, "Compilation for a High-performance Systolic Array," *SIGPLAN'86 Symposium on Compiler Construction* , pp.27-38 (June 1986).

16. J. L. Gustafson, "Programmation du FPS T 64," pub, Floating Point Systems (1986).

17. W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy," Technical Report CSRD, University of Illinois at Urbana-Champaign, Urbana-Champaign (Feb. 1986).

18. R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM* **14**(3), pp.563-590 (July 1967).

19. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

20. L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM* **17**(2), pp.83-93 (Feb. 1974).

21. P. Leca, "Conception d'algorithmes parallèles pour des systèmes multiprocesseurs à mémoires distribuées: application à la programmation du système multi-FPS164 LCAP1," *International Seminar on Scientific Supercomputers*, pp.251-279 (Feb. 1987).

22. B. Lint and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Transactions on Software Engineering* **7**(2), pp.174-188 (March 1981).

23. A. McKellar and E. Coffman, "Organizing Matrices and Matrix Operations for Paged Memory Systems," *Communications of the ACM* **12**(3), pp.153-165 (March 1969).

24. W. L. Miranker and A. Winkler, "Spacetime Representations of Systolic Computational Structures," Research Report RC 9775, IBM (Dec. 1982).

25. D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays.," *Proceedings of the IEEE* **71**(1), pp.113-120 (Jan. 1983).

26. D. I. Moldovan and J. A. E. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Transactions on Computers* **35**(1), pp.1-12 (Jan. 1986).

27. C. Mongenet, "Une méthode de conception d'algorithmes systoliques. Résultats théoriques et réalisation.," Thèse de Doctorat, INPL, Nancy (1985).

28. Y. Muraoka, "Parallelism Exposure and Exploitation in Programs," Ph.D. Thesis, DCS 424, University of Illinois at Urbana-Champaign (Feb. 1971).

29. J.-K. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," PhD Thesis, Report UIUCDCS-R-86-1259, University of Illinois at Urbana-Champaign (March 1986).

30. J.-K. Peir and D. D. Gajski, "Toward Computer-Aided Programming for Multiprocessors," *Int. Workshop on Parallel Algorithms and Architectures*, pp.191-202 (April 1986).

31. J.-K. Peir, D. D. Gajski, and M. Y. Wu, "Programming Environments for Multiprocessors," *International Seminar on Scientific Supercomputing*, pp.47-68 (Feb. 1987).

32. P. Quinton, "The Systematic Design of Systolic Arrays," Rapport 193, IRISA, Rennes (Avril 1983).

33. D. Rapp, "Matrix Multiplication and Inversion on a Concurrent Processing Array," Technical Report, Caltech (March 1984).

34. C. Renvoise, G. Rouquié, J. C. Cottet, and P. Feautrier, "Détection automatique d'opérations parallèles dans les programmes," Note de programmation, C.I.I. Honeywell Bull, Louveciennes (Nov. 1980).

35. Y. Saad, "Gaussian Elimination on Hypercubes," *Int. Workshop on Parallel Algorithms and Architectures*, p.15 (April 1986).

36. J. Spirn, *Program Behavior: Models and Measurements*, Elsevier North Holland, N.Y. (1977).

37. J. W. Stamos, "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," *ACM Transactions on Computer Systems* **2**(2), pp.155-180 (May 1984).

38. K. Trivedi, "Prepaging and Applications to Structured Array Problems," Report UIUCDCS-R-74-662, University of Illinois at Urbana-Champaign (1974).

39. D. Vrsalovic, E. F. Gehringer, Z. Z. Segall, and D. P. Siewiorek, "The Influence of Parallel Decomposition Strategies on the Performance of Multiprocessor Systems," *SIGARCH Newsletter* **13**(3), pp.396-405 (June 1985).

40. M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD. Thesis, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign (1982).

CONCLUSION

Partant des architectures des superordinateurs, des contraintes que ces dernières imposent aux programmeurs, et des résultats obtenus depuis une quinzaine d'années en matière de restructuration de programmes, nous avons proposé une nouvelle transformation, appelée partitionnement, qui réordonne les calculs effectués au sein de boucles imbriquées de manière à utiliser au mieux les deux niveaux de parallélisme que l'on trouve dans les multi-multiprocesseurs ou dans les multiprocesseurs vectoriels, à profiter du niveau le plus rapide d'une hiérarchie mémoire, qu'il s'agisse d'antémémoires ou de mémoires locales, et à contrôler le pourcentage du temps perdu à effectuer des synchronisations entre tâches.

L'ensemble des partitionnements valides a été caractérisé par des équations et des inéquations linéaires ce qui permet à la fois de rechercher d'une manière automatique un *meilleur* partitionnement et de générer automatiquement la version partitionnée du code initial, une fois le partitionnement fixé par des coefficients numériques. La génération de code est fondée sur la projection de systèmes d'égalités et d'inégalités linéaires, sur l'élimination de la redondance de tels systèmes, sur les tests de faisabilité et dans une moindre mesure sur l'enveloppe convexe de deux systèmes. Ces algorithmes doivent être conçus pour des variables entières relatives et ne peuvent être dérivés immédiatement des algorithmes similaires qui travaillent sur un domaine réel car, contrairement à ce qui est usuellement admis en détection du parallélisme, aucune imprécision n'est acceptable. Les algorithmes de base ont été programmés et utilisés pour assurer la validité des systèmes donnés dans le chapitre 7.

La génération de code proposée comprend aussi les déclarations de données locales, la détection des parties de tableaux qu'il faut importer ou exporter avant de lancer l'exécution d'une tâche et les instructions de transferts entre les deux niveaux de la hiérarchie mémoire qui sont pris en compte. Cette génération devrait pouvoir être étendue au cas des machines sans mémoire locale, comme les hypercubes, sous forme de génération de messages mais aucun élément n'est fourni sur la manière d'allouer les tableaux sur de nombreuses mémoires locales car ce problème a été laissé hors du champ de ce travail.

Le choix du meilleur partitionnement est guidé à la fois par les transferts de données qu'il impose entre mémoire lente et mémoire rapide, par le volume de données qui peut être stocké dans la mémoire rapide et par la vitesse de calcul pur que pourra atteindre un processeur élémentaire sur une tâche. L'évaluation symbolique des quantités correspondantes n'est pas toujours possible mais les résultats partiels que nous obtenons sont suffisants pour traiter les termes de plus haut degré du volume transféré et du volume de cache ou de mémoire locale nécessaire pour éviter les transferts inutiles. Ceci permet d'envisager à terme une automatisation du choix du partitionnement pour un ensemble de boucles imbriquées donné et pour une machine précise. Mais, pour le moment, ce choix reste encore sous la responsabilité du programmeur.

La classe de partitionnement proposée est la plus large que nous connaissons. Elle intègre les partitionnements primitifs qui sont obtenus par une combinaison de blocage de boucle (*Strip-Mining*) et

d'échange de boucles (*Loop Interchange*). Elle contient aussi l'échange de boucles et la méthode hyperplane pour lesquel notre méthode de génération de code est applicable. Elle étend aussi le domaine d'application de la méthode hyperplane à des programmes dont les dépendances ne sont pas uniformes en utilisant les rayons du cône de dépendance et non les vecteurs de dépendances eux-mêmes.

Un problème fondamental nous a gêné tout au long de ce travail, la qualité des générateurs de code de beaucoup de compilateurs modernes. Bien que de nombreuses techniques d'optimisation aient été publiées depuis le milieu des années 60, le code compilé produit en 1986 contient encore des séquences STORE/LOAD, des STORE ou des LOAD multiples de la même valeur, des calculs d'adresses nécessitant des multiplications alors qu'il pourraient faire l'objet d'une réduction de force, des instructions à effet constant dans des corps de boucle, etc... Cette situation est doublement gênante pour celui qui propose une nouvelle technique d'optimisation. D'abord parce qu'il vaudrait mieux implémenter en priorité les techniques plus simples qui sont déjà connues. Et ensuite, parce que le code généré à partir de programmes partitionnés va être si mauvais avec les générateurs que nous avons vus que le gain espéré risque fort d'être annulé par l'exécution d'instructions totalement inutiles.

Les générateurs de code de mauvaise qualité nous empêchent aussi d'évaluer le temps nécessaire à l'exécution des corps de boucles et le temps utilisé à effectuer les tests et initialisations de boucle, ce qui est un obstacle majeur à l'automatisation du choix d'un bon partitionnement. Enfin, les compilateurs Fortran ne disposent pas de directives permettant d'indiquer que telle ou telle variable peut être conservée en registre et d'éviter des rangements en mémoire totalement inutiles. En conséquence, l'équipe *Applications* du CSRD a dû effectuer une grande partie de ses travaux en assembleur.

Avant de passer aux suites que ce travail pourrait recevoir, je pense qu'il est important de préciser que les quelques jours passés à faire des expériences sur une Alliant FX-8 aux Etats-Unis ont eu une importance beaucoup plus considérable sur ce travail que celle que pourraient suggérer les quelques pages qui leur sont consacrées. C'est là-bas que se sont concrétisés les problèmes de génération de code partitionné et l'importance particulière qu'avait la génération d'un code soigneux pour les supernoeuds partiels alors que ce problème avait auparavant été écarté à coups de O(n). C'est aussi aux Etats-Unis que s'est révélée l'influence un peu perverse des générateurs de code et que nous avons obtenu les premiers et les seuls résultats expérimentaux positifs montrant que le partitionnement pouvait être utile en dépit de l'overhead de contrôle supplémentaire qu'il nécessite.

Il est aussi intéressant de noter que le partitionnement, tel que nous l'avons défini dans ce travail, est difficile à vendre aux chercheurs américains qui travaillent sur la parallélisation automatique de programmes, qu'ils se trouvent à l'Université Rice dans l'équipe de K. Kennedy, à l'Université d'Illinois dans l'équipe du projet Cedar ou dans les laboratoires d'IBM à Yorktown Heigths sous la direction de F. Allen. Et ce, pour plusieurs raisons. Premièrement, il n'est pas possible de dire quel est le gain potentiel ou

quelle proportion de programmes réels pourrait bénéficier de la méthode proposée car aucune expérience à grande échelle n'a été faite a priori pour valider l'idée avant de se lancer dans les développements un peu théoriques. Deuxièmement, cette méthode nécessite des informations aussi précises que la méthode hyperplane qui n'est déjà généralement pas implémentée car considérée comme trop compliquée et trop lente pour être intégrée dans un programme qui, à terme, doit n'être qu'une toute petite partie d'un compilateur.

Il faudrait donc poursuivre ce travail par une série d'expériences portant sur plusieurs machines et sur plusieurs programmes, et ce, même si un générateur de partitionnements semi-automatique n'est pas encore disponible. L'étude de quelques programmes va d'ailleurs montrer immédiatement qu'il faut étendre la méthode aux boucles non parfaitement imbriquées si l'on veut que son domaine d'application ne soit pas trop limité.

Des extensions sont aussi nécessaires en ce qui concerne les machines cibles. Tout d'abord, il faudrait savoir traiter le niveau mémoire ultra-rapide que constituent les registres. Une première ébauche a été proposée dans ce travail, mais il reste beaucoup à faire. Il faudrait aussi savoir traiter plus de deux niveaux de hiérarchie mémoire et se préoccuper davantage de la génération de code d'une tâche pour éviter que le compilateur ne procède à des partitionnements supplémentaires pour utiliser des registres vectoriels.

Le nombre de niveaux de parallélisme va encore augmenter. Par exemple, le projet CEDAR en prévoit trois. C'est trop pour un espace de calcul de dimension 2, mais le partitionnement dégage au moins quatre niveaux de parallélisme avec un espace de dimension 3: deux à l'intérieur des tâches et deux entre les tâches. Faudra-t-il exploiter le parallélisme de grain fin qui existe dans une tâche élémentaire ou le parallélisme plus grossier qui se présente entre tâches? L'architecture de la machine et les performances de son système d'exploitation ou de ses primitives de synchronisation devront guider la réponse.

Il nous semble donc que l'adaptation automatique de programmes, qu'ils soient séquentiels ou non, qu'ils soient écrits dans un vieux langage comme Fortran ou au moyen d'équations, restera, quelques années encore, un domaine de recherche.

Résumé

L'architecture compliquée des superordinateurs modernes doit être prise en compte lors de la conception des programmes pour assurer à la fois une saturation du parallélisme vectoriel et des processeurs élémentaires, une bonne localité des références de manière à ne pas se trouver limité par la bande passante de la mémoire principale, et un coût de synchronisation acceptable.

Nous proposons une nouvelle transformation de programme, appelée partitionnement en supernoeuds, qui s'applique aux boucles imbriquées, et qui permet d'atteindre les quatre objectifs cités précédemment. Cette transformation est basée sur la connaissance précise des dépendances entre instructions du corps de boucle. Elle consiste à découper l'espace des calculs par des familles d'hyperplans respectant les dépendances pour ne pas créer de cycles entre les partitions. La validité d'une famille peut être testée automatiquement.

Nous montrons comment cette transformation peut être appliquée de manière automatique avec des algorithmes sur les systèmes linéaires d'inégalités dès que les caractéristiques du partitionnement sont connues numériquement.

Le choix d'un partitionnement optimal, qui dépend à la fois des transferts de données, du volume de données que peut contenir la mémoire la plus rapide, des caractéristiques des unités fonctionnelles et des performances des primitives de synchronisation disponibles ne peut pas être fait par dérivation analytique car de nombreuses données de base, relatives à la génération de code ou à l'ensemble des calculs considérés, sont inconnues en général. Nous proposons donc une liste de paramètres et de facteurs qu'il faut prendre en compte pour déterminer manuellement un partitionnement intéressant. Nous avons ainsi réussi à obtenir une augmentation de vitesse de 50 % sur l'Alliant FX-8 en améliorant la localité des références.

Cette méthode est une généralisation de l'échange de boucles, de la méthode hyperplane et de la classe des partitionnements qu'on obtient par blocage et échange de boucles. La méthode de génération de code développée s'applique donc aussi à ces cas.

Mots-clés

Parallélisme, parallélisation, vectorisation, multiprocesseur, transformation de programme, partitionnement.