RECHERCHE

Distribution automatique des données et des calculs

Paul Feautrier

PRiSM, Université de Versailles
45, Avenue des Etats-Unis
78035 Versailles Cedex
Paul.Feautrier@prism.uvsq.fr

RÉSUMÉ. *Pour un multiprocesseur à mémoire distribuée, le facteur de performance le plus critique est le taux d'accès à des informations éloignées. Ce taux d'accès peut être considérablement réduit si l'on distribue habilement les données et les calculs parmi les processeurs et leurs mémoires. Si l'on utilise un langage de programmation à parallélisme de données comme HPF, ce travail est de la responsabilité de l'utilisateur. On explore ici une autre possibilité, celle d'une distribution automatique à la compilation à partir des seules informations disponibles dans le texte source. On montre qu'il est possible, en utilisant des techniques peu complexes d'algèbre linéaire, de trouver des placements satisfaisants à condition que le programme source se limite à des boucles* DO *et à des tableaux indexés au moyen de fonctions affines.*

ABSTRACT. *The most critical factor in the performance of a distributed memory computer is the access frequency to remote data. This frequency may be reduced by a clever distribution of data and computations among processors and their memories. In the context of data parallel languages – as for instance, HPF – finding the proper distribution is the responsibility of the programmer. This paper explores another possibility, namely having the compiler determine the distribution using only information available in the source program. The paper shows that, with the help of elementary linear algebra techniques, one may find satisfactory placements provided the source program is limited to* DO *loops and arrays with affine subscripts.*

MOTS-CLÉS : *parallélisation automatique, mémoire distribuée, placement, alignement.*
KEY WORDS : *automatic parallelization, distributed memory, placement, alignment.*

1. Introduction

Les applications informatiques modernes ont des besoins en puissance de traitement qu'un monoprocesseur courant n'est pas toujours capable de satisfaire. Ceci est vrai aussi bien dans le domaine du calcul à hautes performances que pour la gestion des bases de données ou la simulation par événements. On peut naturellement tenter de réaliser des monoprocesseurs de haute puissance. Dans cette direction, on se heurte rapidement aux limites des technologies disponibles. Le progrès "naturel", en particulier en matière de gravure sur silicium, fait croître la puissance des microprocesseurs au rythme d'un facteur deux tous les 18 mois. Par contre, la réalisation de superordinateurs ou de *mainframes* sans faire appel aux microprocesseurs standards est une entreprise difficile et de plus en plus coûteuse.

La solution, naturellement, est la réalisation de superordinateurs à partir de composants standards. Mais même ainsi, les choses ne sont pas si simples. Supposons qu'à un certain instant on ait besoin du double de la puissance du processeur courant. On décide naturellement de fabriquer un biprocesseur. Mais si cette fabrication doit prendre 18 mois, il vaut mieux ne rien faire et attendre la sortie de la génération suivante. Pour que la conception d'un ordinateur parallèle soit rentable, il faut que son parallélisme soit élevé, ou sa structure suffisamment simple pour que sa réalisation soit rapide à l'échelle de l'évolution technologique.

De ce point de vue, les ordinateurs à mémoire distribuée ont un avantage certain. Rien de plus simple que leur réalisation : il suffit d'interconnecter des équipements standards – par exemple des stations de travail – à l'aide d'un réseau qui peut être également standard, tel un Ethernet ou un réseau ATM. Le système d'exploitation d'un tel équipement peut être également standard, bien que des bibliothèques spécialisées comme PVM [BEG 91] aient été spécialement développées. La programmation de ce type d'architecture se fait par *échange de messages*. Un échange de message doit être prévu dès qu'un processeur a besoin d'une information qui réside dans la mémoire d'un autre processeur. Un tel échange est toujours très coûteux – on peut le chiffrer à l'équivalent de quelques milliers d'instructions de traitement. Il est donc essentiel que ces échanges soient le plus rare possible si l'on veut que le programme conserve un rendement acceptable.

Pour caractériser les diverses méthodes de construction de programmes parallèles, il est d'usage d'utiliser le concept de *grain*. Nous pensons pour notre part qu'il y a plusieurs sortes de grains, et que c'est leur comparaison qui permet une évaluation grossière des performances du programme parallèle.

Le nombre d'instructions qui peuvent être exécutées pendant la durée d'un échange typique est le *grain de parallélisme* de l'architecture et sera noté G dans ce qui suit. On peut se rendre compte très simplement de l'importance de ce paramètre en imaginant un programme composé de phases de calcul de N instructions suivies d'un échange de message impliquant tous les processeurs. Si on répartit les N instructions

également entre les P processeurs, l'ordre de grandeur du rendement sera :

$$\epsilon = \frac{1}{1 + PG/N}.$$

On voit qu'il ne faut pas utiliser plus de N/G processeurs si l'on veut conserver un rendement correct. La quantité N/P est également appelée le *grain* du programme parallèle. On voit que pour avoir un bon rendement, il faut que le grain du programme parallèle soit nettement plus grand que celui de l'architecture.

Pour utiliser efficacement un ordinateur à mémoire distribuée, il faut que les programmes qu'il exécute aient le plus gros grain possible et qu'il ait, au contraire, le grain le plus petit possible. La première condition est du ressort du programmeur, la seconde du ressort de l'architecte.

On a donc vu apparaître des architectures à mémoire distribuée spécialement conçues pour que l'échange de messages soit aussi rapide que possible. Un des points importants est la réduction de la latence – la durée qui s'écoule entre le lancement de l'émission et la réception du premier octet du message – et des réseaux ont été spécialement conçus pour améliorer ce point, avec des résultats spectaculaires. Mais comme pendant ce temps les processeurs n'ont pas cessé de progresser il n'y a pas eu de réduction notable du grain. Ce qui reste en tout cas invariant, c'est la *structure* de ces machines : la brique de base est constituée d'un processeur et de sa mémoire. Les processeurs sont interconnectés par un réseau qui peut avoir des topologies diverses : grille à une, deux ou plus de dimensions, hypercube, réseaux récursifs, etc. Le point important est qu'il y a plusieurs ordres de grandeur entre le temps nécessaire pour qu'un processeur accède à sa mémoire locale et la durée d'un échange de messages. Les performances d'un programme vont donc dépendre de façon critique de la fréquence des communications entre processeurs. On peut envisager a priori deux méthodes pour en diminuer le nombre :

— On peut partir d'une répartition arbitraire et l'améliorer au cours du fonctionnement du programme, en faisant migrer soit les programmes, soit les données. Certains systèmes d'exploitation distribués fournissent des outils pour faciliter la migration de tâches. Les systèmes à mémoire partagée virtuelle, au contraire, déplacent les données vers les tâches qui les utilisent.

— L'autre solution est de rechercher la distribution optimale des données et des calculs. Cette responsabilité peut être confiée au programmeur, ou au compilateur.

La première solution a l'avantage de la simplicité : aucune analyse compliquée n'est nécessaire. D'autre part, il n'est pas sûr qu'une distribution unique convienne pour tout un gros programme. Par contre, si l'on applique la deuxième solution à des noyaux pas trop gros, on peut espérer obtenir des résultats meilleurs, puisque l'on dispose, au moins en théorie, d'informations complètes sur le comportement du programme source. Cependant, ces informations sont complexes et le problème à résoudre est un problème délicat d'optimisation. L'objectif essentiel de cet article est d'explorer la possibilité de confier la détermination d'une distribution optimale au compilateur.

1.1. *Un exemple élémentaire*

Pour fixer les idées, nous allons étudier un exemple très simple. Soit l'instruction

```
program A
x = y + z
```

Pour «distribuer» cette instruction, il y a quatre décisions à prendre : quels processeurs hébergeront `x`, `y` et `z`, et quel processeur exécutera l'addition. On peut par exemple affecter une mémoire – et par conséquent un processeur – pour chaque variable, et utiliser un quatrième processeur pour effectuer le calcul. Cette solution induit évidemment trois communications. A l'autre extrême, on peut regrouper toutes les données et le calcul sur un seul processeur, et plus aucune communication n'est nécessaire. Il est probable que la deuxième solution sera la plus efficace ; elle n'a pas de parallélisme, mais après tout le programme original n'en avait pas non plus.

Considérons maintenant un exemple voisin :

```
program B
do i = 1,n
    x(i) = y(i) + z(i)   {S}
end do
```

On peut tout d'abord considérer qu'il faut placer comme ci-dessus les tableaux `x`, `y` et `z`, et l'instruction S. On obtient un résultat aussi médiocre que le précédent, avec beaucoup de communications ou peu de parallélisme. Si on cherche plutôt à placer les *cellules* des trois tableaux en cause, ainsi que les *itérations* de l'instruction S, on voit facilement que si `x(i)`, `y(i)`, `z(i)` et l'itération `i` de S sont placés sur le même processeur, aucun échange de message ne sera nécessaire ; par contre, il n'y a aucune contrainte sur le placement relatif des objets associés à des valeurs distinctes de `i`. Si P est le nombre de processeurs disponibles, on peut par exemple découper l'intervalle $[1, n]$ en P segments (presque) égaux, et attribuer à chaque processeur les fragments homologues des tableaux `x`, `y` et `z`, ainsi que les itérations correspondantes de la boucle.

Nous voyons apparaître ici un troisième grain, le *grain de l'analyse*, qui caractérise ce que le compilateur considère comme la plus petite partie analysable du programme source. Ici l'analyse a été menée à grain fin : la cellule de tableau et l'itération de boucle. Comme le compilateur procède par agrégation, le grain du programme résultant est au moins aussi gros que celui de l'analyse. Ici, on obtient un grain de n/P, ce que l'on peut considérer comme un grain «moyen» pourvu que n soit suffisamment grand. Une tendance naturelle est de mener l'analyse à grain aussi gros que celui de l'architecture cible. L'expérience montre que ce choix fait en général perdre du parallélisme, et qu'il vaut mieux, paradoxalement, mener l'analyse sur des objets aussi petits que possible pour les regrouper ensuite.

1.2. *Etat de l'art*

Le problème de la distribution est au centre de la problématique des langages à parallélisme de données [BOU 93], dont le développement, à son tour, a été impulsé par celui des ordinateurs à mémoire distribuée. Dans ces langages – le plus connu est HPF [HPF 94] – c'est l'utilisateur qui spécifie la distribution des données, le compilateur en déduisant celle des calculs par application de la règle des écritures locales (*owner computes*). La justification de cette procédure est que l'utilisateur est censé connaître la circulation des informations dans son programme, celle-ci devant reproduire le comportement du système modélisé. En conséquence, le compilateur doit être prêt à traduire n'importe quelle spécification, aussi inefficace soit-elle. En effet, si l'utilisateur arrive en général assez bien à trouver une bonne distribution quand il en existe une, il est souvent moins habile lorsqu'il s'agit de trouver le bon compromis entre des contraintes contradictoires. D'où l'idée de confier la détermination de la distribution au compilateur.

Un premier groupe de travaux, ([JIN 90] est un exemple typique) se place dans le cadre du paradigme de la satisfaction de contraintes. Une analyse du programme source permet de spécifier les relations qui doivent exister entre le placement des divers tableaux du programme pour obtenir une exécution efficace. On obtient ainsi un graphe de contraintes (*affinity graph*), qui est usuellement inconsistant. On utilise alors des heuristiques pour satisfaire autant de contraintes que possibles. En général, les types de placement que recherchent ces algorithmes sont moins généraux que ceux qui seront explorés dans cet article.

Le paradigme du placement affine qui est présenté dans cet article a été formulé par Ramanujam et Sadayappan [RAM 91] qui n'ont cependant pas présenté de méthode générale de résolution. L'article [FEA 94] a introduit la technique de l'élimination de Gauss incrémentale, qui sera présentée section 3.2.1. Les auteurs de [BAU 94] ont mis en évidence la relation entre les solutions du problème du placement et le noyau de la matrice de communication.

Il faut signaler enfin une série d'articles [DAR 94a, DAR 93, DIO 94b] où Y. Robert et. al. résolvent le problème du placement par des techniques issues du calcul matriciel. On y trouvera en particulier plusieurs résultats de NP-complétude, qui justifient le recours à des méthodes heuristiques. Ces articles se placent en gros dans le même cadre conceptuel que le nôtre. On recherche des fonctions de placement affines multidimensionnelles distinctes pour les calculs et les données. Les articles [DAR 94a, DAR 93] ne traitent que le cas des accès uniformes (la fonction d'indexation est une translation) dans un nid de boucles parfait. L'article plus récent [DIO 94b] traite des nids arbitraires et des fonctions d'indexation affines. Les auteurs construisent le graphe d'accès du programme. Celui-ci donne, en gros, le sens dans lequel les équations de communication doivent être résolues pour être sûr de l'existence d'une solution. L'analyse de ce graphe (la recherche d'un branchement de poids maximal) indique dans quel ordre les fonctions de placement doivent être calculées, ce

qui se fait par une succession d'inversions de matrices. Comme dans le présent travail, on parvient à rendre locales une grande partie des communications. Les arcs du graphe d'accès qui ne se retrouvent pas dans le branchement maximal donnent des communications résiduelles, qu'il est parfois possible de supprimer par une analyse plus approfondie.

2. Formalisation du problème

Notre objectif est donc de trouver un placement efficace d'un programme à partir de la simple analyse du texte de ce programme. Dans ce travail, nous nous limiterons à l'analyse d'une procédure unique. Le problème du placement d'un ensemble de procédures pose des problèmes conceptuels difficiles, que la recherche du placement se fasse automatiquement ou «manuellement». Dans cette section, l'objectif est de poser le problème et de montrer quelle est sa nature. Le développement d'une technique de résolution fera l'objet de la section suivante.

2.1. *Notations et conventions*

En l'état actuel des techniques de distribution, il n'est pas envisageable de trouver un placement pour un programme arbitraire. En effet, même dans les langages les plus simples, comme Fortran, il est possible de spécifier un schéma d'accès aux données de façon entièrement dynamique, par exemple à l'aide d'un tableau d'indices :

```
program C
do i = 1,n
  a(b(i)) = ....
end do
```

Dans cet exemple, il est impossible d'identifier à la compilation la correspondance entre une itération de la boucle et la cellule du tableau `a` qu'elle modifie, puisque ceci nécessite la connaissance du contenu du tableau `b`. On est donc naturellement conduit à imposer des restrictions aux programmes que l'on peut espérer compiler.

Il est tout d'abord indispensable de pouvoir identifier et dénommer les *opérations* du programme. Une opération est l'exécution d'une instruction. Comme dans la plupart des programmes, chaque instruction est répétée un grand nombre de fois, on doit pouvoir identifier chacune de ces répétitions. Ceci n'est facile que pour les programmes ne comportant que des boucles `DO` et des tests bien structurés comme seules instructions de contrôle. Dans ce cas, une itération d'une instruction est identifiée par la liste des valeurs des compteurs des boucles englobantes. On convient d'écrire cette liste, ordonnée de l'extérieur vers l'intérieur, sous la forme d'un vecteur, le *vecteur d'itération* de l'instruction. On peut alors prendre comme *repère* d'une opération un couple $\langle S, \vec{x} \rangle$ où S est le nom de l'instruction et $\vec{x}$ le vecteur d'itération,

vecteur à coordonnées entières. Pour chaque instruction, sa dimension est fixée : c'est le nombre de boucles englobantes. Les valeurs qu'il peut prendre ne sont pas arbitraires : chaque composante est contrainte par les bornes de la boucle correspondante. L'ensemble ainsi délimité est le *domaine d'itération* de l'instruction étudiée. On notera $\mathcal{D}_S$ le domaine d'itération de S, et $\mathcal{D}$ l'ensemble des opérations, c'est-à-dire la somme disjointe de tous les domaines d'itération.

Si l'instruction S est englobée par un ou plusieurs tests, son domaine d'itération peut n'être qu'un sous-ensemble de celui qui résulte de l'analyse des boucles englobantes. Dans ce cas, la détermination précise du domaine peut être impossible si les prédicats testés sont complexes ou dépendent des données du calcul. On doit se restreindre à une analyse approximative [COL 95]. Il n'y a par contre pas de difficulté si le test porte sur le signe d'une forme affine des compteurs des boucles englobantes : c'est le cas auquel nous nous limiterons ici.

Soit l'exemple :

```
program D
do i = 1,n
  do j = 1,i-1
    a(j,i) = a(i,j)      {S}
  end do
end do
```

Le domaine d'itération de l'instruction S est l'ensemble des points à coordonnées entières contenues dans le triangle :

$$1 \leq i \leq n, \;\; 1 \leq j \leq i-1.$$

L'objectif du compilateur est, en simplifiant, de placer chaque opération sur le processeur qui héberge la majorité des cellules de mémoire utilisées. Il lui est donc indispensable de pouvoir déduire du texte du programme la liste des cellules de mémoire accédées. Ceci n'est simple que si les structures de données se réduisent à des tableaux et à des scalaires. Il faut supposer de plus que dans les programmes à compiler, tous les indices de tableaux sont des fonctions explicitement connues des compteurs des boucles englobantes. Pour simplifier les notations, on considérera que les indices d'un tableau forment un vecteur. On postule donc l'existence, pour chaque variable indexée, d'une fonction à valeur vectorielle, dont l'argument est le vecteur d'itération, et qui donne les indices de la référence au tableau. Les scalaires seront traités comme des tableaux à zéro indice.

Dans le cas de la référence à `a` à droite de l'instruction S dans l'exemple D, cette fonction peut s'écrire :

$$f(\vec{x}) = \begin{pmatrix} x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \vec{x},$$

où $\vec{x}$ est le vecteur d'itération courant. La dernière formulation montre que f peut être associée à la matrice $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ mais il ne s'agit pas là d'une propriété générale.

Il est maintenant possible de formaliser le problème du placement des données et des calculs. On y parvient en associant à chaque opération $\langle S, \vec{x} \rangle$ une fonction Π telle que $\Pi(S, \vec{x})$ soit le nom du processeur qui exécute $\langle S, \vec{x} \rangle$. Sans perte de généralité, on peut supposer que les processeurs sont numérotés, et donc que la valeur de Π est entière[1].

De même, on notera $\Pi(\mathtt{A}, \vec{i})$ le nom du processeur qui héberge la cellule d'indice $\vec{i}$ du tableau $\mathtt{A}$. Π est la *fonction de placement* du programme.

Si q est le numéro d'un processeur particulier, l'ensemble

$$\mathcal{D}_q = \{u \mid u \in \mathcal{D}, \Pi(u) = q\} \qquad \text{[i]}$$

est l'ensemble des opérations que q doit exécuter. On pourrait de même définir le sous-tableau de chaque tableau qui est hébergé par q. On a évidemment intérêt à choisir les fonctions de placement de façon à ce que ces ensembles soient de structure pas trop complexe. Un autre point est que les ensembles $\mathcal{D}_q$ doivent être de tailles comparables si l'on veut que la charge des processeurs soit équilibrée.

On peut supposer, plus généralement, que les Π sont non plus des fonctions mais des relations. Cet artifice permet de représenter aussi bien la duplication des données que la redondance des calculs. Cet aspect du problème du placement est encore mal compris et devra faire l'objet d'études ultérieures. On pourra cependant consulter [BAU 94] qui traite un cas particulier, la duplication des données constantes.

2.2. *Condition de coupure*

Dans ce cadre, les conditions que doivent remplir les fonctions de placement pour qu'aucune communication ne soit nécessaire s'écrivent simplement. Soit S une instruction, $\vec{x}$ son vecteur d'itération, $\mathtt{A}[f(\vec{x})]$ une référence au tableau $\mathtt{A}$ figurant dans S. L'opération $\langle S, \vec{x} \rangle$ est exécutée par le processeur $\Pi(S, \vec{x})$. La cellule $\mathtt{A}[f(\vec{x})]$ est hébergée par le processeur $\Pi(\mathtt{A}, f(\vec{x}))$. Pour éviter une communication, il suffit que ces deux processeurs soient identiques :

$$\Pi(S, \vec{x}) = \Pi(\mathtt{A}, f(\vec{x})). \qquad \text{[ii]}$$

Les équations ainsi obtenues sont appelées *conditions de coupure*, car elles expriment à quelle condition la communication entre la cellule $\mathtt{A}[f(\vec{x})]$ et l'opération $\langle S, \vec{x} \rangle$ peut être «coupée». Le problème du placement peut être vu comme celui de la détermination d'un jeu de fonctions Π satisfaisant à l'ensemble des conditions de coupure, pour toutes les instructions du programme et pour toutes les références qui y figurent. S'il se révèle impossible de trouver des fonctions Π ayant les bonnes propriétés, on peut caractériser les communications résiduelles dues à la référence $\mathtt{A}[f(\vec{x})]$ dans l'instruction S comme l'ensemble :

$$\mathcal{R}_{S\mathtt{A}} = \{\vec{x} \mid \vec{x} \in \mathcal{D}_S, \Pi(S, \vec{x}) \neq \Pi(\mathtt{A}, f(\vec{x}))\} \qquad \text{[iii]}$$

1. On verra plus loin qu'il est parfois utile de nommer les processeurs à l'aide de *vecteurs d'entiers*.

et le problème est de minimiser la somme des tailles de ces ensembles[2].

Il faut observer que cette recherche de minimum doit se faire sous contrainte, sans quoi il a une solution triviale, qui consiste à placer toutes les données et les calculs sur le même processeur. Si l'objectif est de compiler un programme distribué, on doit éviter à tout prix ce phénomène de *collapsus*. La contrainte correspondante est en fait l'équilibre des charges, c'est-à-dire l'égalité des tailles des ensembles $\mathcal{D}_q$ définis ci-dessus [i].

2.3. *Résolution formelle*

Sous la forme ci-dessus, le problème peut être résolu simplement. Considérons le graphe non orienté biparti suivant. Les sommets en sont les opérations et les cellules de tableau du programme. Il y a un arc entre une opération et une cellule si et seulement si l'opération accède à la cellule, que ce soit en lecture ou en écriture. Le graphe ainsi défini est le *graphe de communication* du programme. Trouver un placement, c'est associer un numéro de processeur à chaque sommet du graphe, et l'équation [ii] exprime que deux sommets voisins doivent porter le même numéro. Par transitivité, on en déduit que deux sommets reliés par un chemin du graphe doivent porter le même numéro, et enfin que tous les sommets d'une même composante connexe doivent avoir le même numéro.

On trouve donc la *solution principale* du problème du placement en construisant les composantes connexes du graphe de communication et en affectant un processeur à chaque composante. Soit ϖ la solution principale. Tout autre solution s'en déduit par composition avec une fonction arbitraire :

$$\Pi = \xi \circ \varpi. \qquad [iv]$$

Quand un placement est spécifié, comme ci-dessus, par la composition de deux fonctions, on a pris l'habitude de dire que la fonction ϖ définit un placement sur un système de *processeurs virtuels*, et que la deuxième fonction spécifie la distribution des processeurs virtuels sur les processeurs réels. Ce vocabulaire a été introduit par les architectes de la CM-2 et repris, sous une forme légèrement différente, dans le langage HPF.

On voit que le nombre maximal de processeurs utilisables sans communication par un programme est une caractéristique intrinsèque du programme. En effet, l'application de la fonction ξ – la fonction de repliement – permet de réduire ce nombre (en affectant plusieurs composantes connexes au même processeur) mais pas de l'augmenter. Or en général ce nombre est trop faible : la plupart des programmes ont un graphe de communication *connexe*. Nous allons en voir les raisons et indiquer quelques remèdes à cette situation.

2. Il s'agit là d'une formalisation très schématique. Il faut tenir compte de phénomènes comme la réutilisation d'une même valeur par plusieurs opérations.

2.4. *Quelques exemples élémentaires*

Considérons tout d'abord le code suivant, dont l'effet est de transposer la matrice `t` :

```
program E
do i = 1,n
  do j = 1,i-1
    r = t(i,j)          {S1}
    t(i,j) = t(j,i)     {S2}
    t(j,i) = r          {S3}
  end do
end do
```

La composante connexe de `r` contient évidemment toutes les opérations de S_1 et toutes celles de S_3. On déduit ensuite de S_1 que cette composante connexe contient également les `t(i,j)`, pour $j < i$, et, d'après S_3, les `t(i,j)` pour $j > i$. Il s'ensuit enfin que toutes les opérations de S_2 sont dans la composante connexe de `r`. Au total, on voit que le graphe de communication est connexe, et que la seule façon d'exécuter ce programme sans communication est de n'utiliser qu'un seul processeur. Il est clair que cet état de choses est dû à l'utilisation du scalaire `r`. Le remède est bien connu : il faut *expanser* `r`, par exemple en un tableau à deux dimensions :

```
program EE
do i = 1,n
  do j = 1,i-1
    r(i,j) = t(i,j)     {S1}
    t(i,j) = t(j,i)     {S2}
    t(j,i) = r(i,j)     {S3}
  end do
end do
```

Cette transformation peut être entièrement automatisée à partir de l'analyse du flot des données du programme original [FEA 88, FEA 91, MAY 93, PUG 93]. Dans le graphe de communication du programme EE, à partir du sommet correspondant à la cellule `r(i,j)` avec $j < i$, on peut atteindre les cellules `t(i,j)` et `t(j,i)`, ainsi que les opérations $\langle S_1, i, j\rangle, \langle S_2, i, j\rangle$ et $\langle S_3, i, j\rangle$. Par contre, une itération de compteur $\langle i, j\rangle$ n'accède qu'à des éléments de tableau d'indice $\langle i, j\rangle$ ou $\langle j, i\rangle$. Il ne pourrait y avoir communication entre deux itérations différentes que dans le cas $i = j$, ce qui est exclu par les bornes des boucles[3].

3. On remarquera que ce raisonnement suppose que deux accès à un même tableau avec des indices différents conduisent à des cellules différentes, c'est-à-dire que les indices restent dans les bornes des tableaux. Il s'agit là d'une hypothèse courante en parallélisation automatique, qui appartient à la classe des raisonnements *garbage in, garbage out*.

Au total, la solution principale du problème du placement pour le programme EE peut s'écrire :

$$\begin{aligned}\varpi(S_1,i,j) &= \varpi(S_2,i,j) = \varpi(S_3,i,j) = \varpi(\mathtt{t},i,j) = \varpi(\mathtt{r},i,j) = \\ &= \textbf{if } i \geq j \textbf{ then } \begin{pmatrix} i \\ j \end{pmatrix} \textbf{ else } \begin{pmatrix} j \\ i \end{pmatrix}\end{aligned}$$

On remarquera que les noms des composantes connexes sont ici des vecteurs. Cette solution n'engendre aucune communication et s'exécute en temps $O(1)$ sur $O(n^2)$ processeurs, nombre que l'on peut réduire en employant une fonction de repliement convenable.

Soit maintenant le code du produit de matrices, dans une version totalement expansée, ou encore, en assignation unique :

```
program F
do i = 1,n
  do j = 1,n
    c(i,j,0) = 0                                    {S1}
    do k = 1,n
      c(i,j,k) = c(i,j,k-1) + a(i,k)*b(k,j)         {S2}
    end do
  end do
end do
```

Les conditions de coupure sont :

$$\begin{aligned}\Pi(S_1,i,j) &= \Pi(\mathtt{c},i,j,0) && \text{[v]}\\ \Pi(S_2,i,j,k) &= \Pi(\mathtt{c},i,j,k) && \text{[vi]}\\ &= \Pi(\mathtt{c},i,j,k-1) && \text{[vii]}\\ &= \Pi(\mathtt{a},i,k) && \text{[viii]}\\ &= \Pi(\mathtt{b},k,j) && \text{[ix]}\end{aligned}$$

Il est facile de voir que [viii] et [ix], entraînent que $\Pi(S_2,i,j,k)$ ne dépend ni de i ni de j en s'appuyant sur des raisonnements du genre «si $\phi(x) = \psi(y)$ où x et y sont des variables indépendantes distinctes, alors ϕ et ψ sont des fonctions constantes». Pour la même raison, la conjonction de [vi] et [vii] entraîne que $\Pi(\mathtt{c},i,j,k)$ ne dépend pas de k. En combinant tous ces résultats, on voit que les fonctions Π sont constantes, que le graphe de communication est connexe et donc qu'il n'existe pas d'implémentation du programme F sans communication. De plus, comme on est parti d'une version en assignation unique, il est impossible de remédier à cette situation par des expansions. On peut par contre penser que les communications sont dues au partage de constantes, en l'espèce les matrices `a` et `b`. Si on ignore les deux dernières conditions de coupure ([viii] et [ix]), on trouve la solution principale :

$$\varpi(S_1,i,j) = \varpi(S_2,i,j,k) = \begin{pmatrix} i \\ j \end{pmatrix},$$

mais cette solution implique la diffusion préalable de tout ou partie de `a` et `b`. Ce couplage par partage de constantes est très fréquent et peut souvent être résolu par duplication.

Pour terminer cette revue d'exemples, soit le code de l'élimination de Gauss :

```
program G
do i = 1,n
  do j = i+1,n
    do k = i+1,n
      a(j,k) = a(j,k) - a(j,i)*a(i,k)/a(i,i)    {S}
    end do
  end do
end do
```

Les conditions de coupure sont :

$$\Pi(S,i,j,k) = \Pi(\mathtt{a},j,k) = \Pi(\mathtt{a},j,i) = \Pi(\mathtt{a},i,k) = \Pi(\mathtt{a},i,i), \qquad \text{[x]}$$

ce qui implique évidemment que la fonction Π est constante : le programme G a un graphe de communication connexe. Ici, il ne s'agit ni d'un problème de scalaire insuffisamment expansé, ni d'un partage de constante, mais d'un trait intrinsèque à l'algorithme. Pour trouver un placement non trivial pour G – et pour les très nombreux programmes qui ont le même comportement – il faut ignorer une partie des conditions de coupure, auxquelles correspondront des communications résiduelles. La section suivante est consacrée au développement d'un algorithme permettant de choisir les communications à conserver et de calculer les fonctions de placement associées.

3. Méthode de résolution

Dans les exemples simples que nous venons de traiter, la résolution des équations de coupure pouvait se faire sans hypothèse préalable sur les fonctions de placement. Dans le cas général, les équations à traiter sont beaucoup plus complexes ; de plus, l'objectif de ce travail est de trouver un *algorithme* de résolution et non pas une collection de techniques disparates. Il y a peu d'espoir d'y parvenir sans faire des hypothèses sur les données du problème, ici les fonctions d'indexation qui permettent, telle f dans la formule [ii] de passer d'un repère d'opération aux indices d'un tableau.

On conviendra ici que toutes les fonctions d'indexation sont affines :

$$f(\vec{x}) = F\vec{x} + \vec{h}. \qquad \text{[xi]}$$

Si on note $|S|$ le nombre de boucles englobant l'instruction S, et $|\mathtt{A}|$ le nombre d'indices du tableau `A`, alors dans la formule ci-dessus, F est une matrice de dimension $|\mathtt{A}| \times |S|$ et $\vec{h}$ est un vecteur de dimension $|\mathtt{A}|$.

Cette hypothèse est courante en parallélisation automatique. Les programmes qui ne comportent que des boucles `DO` et dont les fonctions d'indexation sont affines

forment ce qu'il est convenu d'appeler les programmes *à contrôle statique* [FEA 88]. Il y a des raisons de penser, d'une part, que c'est la seule classe de programmes pour laquelle il est possible de résoudre exactement le problème de la compilation pour machines parallèles. Lorsque l'on veut traiter des programmes plus généraux, on est forcé, soit d'utiliser des méthodes d'approximation [COL 95], soit de renvoyer le problème de la parallélisation à l'exécution.

D'autre part, l'étude [ZHI 89] a montré qu'une proportion importante des programmes de calcul scientifique – de l'ordre de 80% – est à contrôle statique. Un domaine de recherche important est l'étude des méthodes permettant d'augmenter cette proportion par restructuration du programme source (élimination des `GOTO` [AMM 92], élimination des variables inductives [AHO 86], etc.).

Mais l'hypothèse que l'on vient de faire n'est pas suffisante pour permettre une résolution facile des conditions de coupure. On doit y rajouter l'hypothèse que les fonctions inconnues Π sont également affines :

$$\Pi(S, \vec{x}) = P_S \vec{x} + \vec{q}_S , \qquad \text{[xii]}$$

$$\Pi(\mathtt{A}, \vec{x}) = P_{\mathtt{A}} \vec{x} + \vec{q}_{\mathtt{A}} . \qquad \text{[xiii]}$$

Les nouvelles inconnues sont maintenant les matrices P_S, $P_{\mathtt{A}}$, et les vecteurs $\vec{q}_S$ et $\vec{q}_{\mathtt{A}}$.

Il faut bien dire que cette hypothèse est plus difficile à justifier. D'une part, un placement doit servir à la construction d'un programme parallèle. Or les techniques connues de restructuration de boucles [ANC 91, COL 94, XUE 94, KEL 94] ne s'appliquent qu'à des placements affines et peut-être affines par morceaux.

Mais d'autre part, il s'agit d'une heuristique : ce n'est que pour le cas des fonctions linéaires qu'il a été possible de trouver des algorithmes de placement automatiques. Il est d'ailleurs facile de construire des exemples simples ayant des placements compliqués, comme :

```
program H
do i = 1,n
  a(i) = a(2*i)
end do
```

Pour ce programme, la condition de coupure se ramène à :

$$\Pi(\mathtt{a}, i) = \Pi(\mathtt{a}, 2i).$$

Tous les indices de la forme $(2p + 1).2^k$ sont clairement dans la même composante connexe du graphe de communication. On peut donc prendre comme solution principale de cette équation la fonction qui associe à i son plus grand facteur impair! Cette fonction n'est évidemment pas linéaire, ni même linéaire par morceaux.

Dans la formule [xii], la matrice P_S est de dimension $g \times |S|$. Le nombre g est la dimension du placement. Pour comprendre son rôle, on peut tout d'abord faire référence à la structure de la machine cible. On se rappelle que celle-ci

est un multiprocesseur où les différentes unités sont interconnectées à l'aide d'un réseau de communication. Il est fréquent que ce réseau ait une forme géométrique régulière : on peut utiliser des processeurs ayant tous le même nombre de liens vers le réseau, ce qui en simplifie la réalisation. Parmi les réseaux réguliers les plus utilisés, on trouve les grilles : le réseau des processeurs est isomorphe à un sous-ensemble de $\mathbb{N}^d$, chaque processeur étant relié à ses $2d$ plus proches voisins. d est la dimension de la grille ; on trouve pour d les valeurs 1 (réseau linéaire), 2 et 3. Sur une grille à d dimensions, il est naturel de repérer un processeur par d nombres qui représentent ses coordonnées dans la grille. Dans ce cas, il est naturel de prendre $g = d$, et de chercher un placement ayant la dimension du réseau.

Mais une autre considération peut s'appliquer, même si le réseau n'est pas une grille ou quand la dimension de la grille est ajustable – ce qui est le cas des réseaux hypercubes. Supposons que le programme source comporte des nids de boucles de taille de l'ordre de n. Alors une forme affine sur les compteurs des boucles va également prendre de l'ordre de n valeurs. Si l'on utilise un placement de dimension g, on générera de l'ordre de n^g processeurs virtuels. Ce nombre doit être comparé d'une part au nombre de processeurs réels de la machine cible, et d'autre part au taux maximum de parallélisme du programme source.

Soit à inverser une matrice d'ordre $n = 1000$ sur une machine à mémoire distribuée ayant, comme il est fréquent dans les architectures actuelles, 100 processeurs relativement puissants. Un placement à une dimension engendre 1000 processeurs virtuels qu'il est facile de répartir sur les 100 processeurs réels. On obtiendra, suivant le jargon en usage, un *vp-ratio* de 10.

Si la machine, maintenant, est une CM-2 de 64000 processeurs, on devra passer à un placement à deux dimensions pour maintenir un vp-ratio convenable. Si enfin la taille de la matrice n'est plus que $n = 100$, on pourrait être tenté de passer à un placement à trois dimensions. Mais comme le taux de parallélisme moyen de l'inversion de matrice est de l'ordre de $n^2 = 10000$, on voit qu'il sera de toutes les façons impossible d'utiliser 64000 processeurs ; un placement à trois dimensions est inutile.

La conclusion de cette discussion est que le choix de la dimension du placement est un problème complexe, qui fait intervenir à la fois les caractéristiques du programme source et celles de la machine cible. Il est probablement impossible d'y procéder sans recourir à l'expérience. Dans ce qui suit, on supposera que la valeur de g a été imposée, et que l'algorithme de placement doit s'y adapter.

3.1. *Construction de la matrice de communication pour un placement à une dimension*

Pour ne pas trop compliquer les notations, on va tout d'abord traiter le cas d'un placement à une dimension ($g = 1$), ce qui conviendrait à un réseau linéaire. Les matrices P_S et $P_{\mathtt{A}}$ de [xii,xiii] deviennent des vecteurs $\vec{p}_S$ et $\vec{p}_{\mathtt{A}}$. De même, les vecteurs $\vec{q}_S$ et $\vec{q}_{\mathtt{A}}$ deviennent des scalaires. La condition de coupure

[ii] devient :

$$\vec{p}_S.\vec{x} + q_S = \vec{p}_{\mathtt{A}}.(F\vec{x} + \vec{h}) + q_{\mathtt{A}}.$$

Cette équation représente autant de contraintes qu'il y a de points dans le domaine d'itération de S. Mais en fait, toutes ces contraintes ne sont pas indépendantes. Il est facile de se rendre compte que si elles sont satisfaites en d points $\vec{x}_1, \ldots, \vec{x}_d$, elles seront satisfaites en tout point du sous-espace affine engendré par ces points. En général, le sous-espace engendré par le domaine d'itération de S est l'espace à $|S|$ dimensions tout entier. Il contient entre autres l'origine et les vecteurs de base. En écrivant la condition ci-dessus en ces points distingués, on trouve :

$$\vec{p}_S = \vec{p}_{\mathtt{A}} F, \qquad \text{[xiv]}$$

$$q_S = \vec{p}_{\mathtt{A}}.\vec{h} + q_{\mathtt{A}} \qquad \text{[xv]}$$

Il arrive parfois que le domaine d'itération d'une instruction ne soit pas de la dimension apparemment indiquée par le nombre de boucles englobantes. L'instruction S du programme ci-dessous en est un exemple :

```
do i = 1,n
  do j = 1,n
    if(i.eq.j) then
      a(i,j) = 1.        {S}
    else
      a(i,j) = 0.
    end if
  end do
end do
```

On peut toujours régler la difficulté en identifiant le plus petit sous-espace contenant le domaine d'itération, puis en reprenant le raisonnement ci-dessus dans ce sous-espace au moyen d'un simple changement de base.

Les conditions à satisfaire pour qu'un placement affine n'engendre aucune communication s'obtiennent en rassemblant toutes les équations [xiv,xv] pour toutes les références à la mémoire du programme. Dans ces équations, les inconnues sont les (composantes des) vecteurs $\vec{p}_S$, $\vec{p}_{\mathtt{A}}$ et les scalaires q_S et $q_{\mathtt{A}}$. Les autres termes, la matrice F et le vecteur $\vec{h}$, au contraire, se déduisent du texte du programme, par simple lecture dans les cas favorables, ou par une analyse plus complexe, par exemple s'il faut identifier des variables inductives. Le point important est que du point de vue des inconnues ci-dessus, chaque référence engendre un système d'équations linéaire et *homogène*.

Les deux équations obtenues jouent d'ailleurs un rôle différent. La quantité :

$$d(S, \mathtt{A}, \vec{x}) = \Pi(S, \vec{x}) - \Pi(\mathtt{A}, f(\vec{x}))$$

est la *distance* qui sépare le processeur qui exécute $\langle S, \vec{x}\rangle$ de celui qui possède $\mathtt{A}[f(\vec{x})]$. Cette distance peut avoir un sens physique, dans le cas d'un réseau

en grille, ou non ; mais elle donne toujours une indication du délai nécessaire à l'échange d'information. Dans le cas d'un placement affine, elle s'écrit :

$$d(S, \mathtt{A}, \vec{x}) = (\vec{p}_S - \vec{p}_{\mathtt{A}} F).\vec{x} + q_S - \vec{p}_{\mathtt{A}}.\vec{h} - q_{\mathtt{A}}.$$

Cette formule montre que si l'équation [xiv] est satisfaite, la distance de communication ne dépend pas de $\vec{x}$. Or, sur presque tous les réseaux, ce type de communication à distance constante, encore appelé décalage, s'effectue de façon beaucoup plus efficace qu'une communication où l'émetteur et le récepteur sont arbitraires – un facteur supérieur à 30 sur la CM-5 [PLA 95]. Si les équations [xiv] sont satisfaites, alors on peut tenter de satisfaire [xv]. Si on y parvient, toutes les communications auront été éliminées. Cependant, il ne s'agit plus là d'un objectif prioritaire : on peut parfaitement tolérer quelques communications résiduelles à distance constante. De toutes les façons, les méthodes qui vont être présentées pour résoudre [xiv] peuvent s'étendre sans difficulté à la résolution de [xv].

Pour caractériser le problème à résoudre, la première chose à faire est de rassembler toutes les inconnues en un seul vecteur obtenu en concaténant les $\vec{p}_S$ et les $\vec{p}_{\mathtt{A}}$. Le *vecteur de placement* $\vec{p}$, ainsi obtenu, a $N = \sum_S |S| + \sum_{\mathtt{A}} |\mathtt{A}|$ composantes. L'ordre dans lequel les instructions et les tableaux sont énumérés est arbitraire. Nous écrirons, par exemple, $S < T$ pour indiquer que l'instruction S vient avant l'instruction T dans cet ordre.

Avec ces conventions, l'équation [xiv] peut s'écrire :

$$\vec{p} C_{S\mathtt{A}} = 0,$$

où $C_{S\mathtt{A}}$ est une matrice bloc :

$$C_{S\mathtt{A}} = \begin{pmatrix} Z_1 \\ I \\ Z_2 \\ -F \\ Z_3 \end{pmatrix}.$$

I est la matrice unité de taille $|S| \times |S|$ et F est la matrice d'indexation de [xi]. Les matrices Z_i sont des matrices nulles de dimension convenable. Par exemple, la dimension de Z_1 est $N_1 \times |S|$, avec

$$N_1 = \sum_{R<S} |R|.$$

La matrice $C_{S\mathtt{A}}$ est la matrice de communication élémentaire qui correspond à la référence au tableau $\mathtt{A}$ dans l'instruction S. Il ne reste plus qu'à rassembler toutes les équations ainsi obtenues sous la forme :

$$\vec{p} C = 0. \qquad \text{[xvi]}$$

La matrice C s'obtient en assemblant les matrices $C_{S\mathtt{A}}$ [4] et s'appelle la *matrice de communication* du programme. Sous cette forme, la solution du problème est évidente. Le vecteur de placement $\vec{p}$ peut être choisi arbitrairement dans le noyau $\ker(C)$. En général, on observe le phénomène de collapsus parce que dans beaucoup de cas, la matrice de communication est de rang plein ; son noyau se réduit au vecteur 0, qui correspond à un placement sur un seul processeur.

Le collapsus ne se produit pas dans le cas de l'exemple D ci-dessus. Il faut placer l'instruction S et le tableau a, tous deux à deux dimensions. Le vecteur $\vec{p}$ est donc de dimension 4. Il y a deux matrices d'indexation :

$$F_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; F_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

En substituant ces matrices dans l'équation [xiv] et en assemblant on trouve la matrice de communication suivante :

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

Le lecteur vérifiera que le noyau de C est engendré par le vecteur $(1, 1, 1, 1)$.

3.2. *Algorithme glouton*

Que faire lorsque la matrice de communication a un noyau nul ? La solution proposée est d'accepter des communications résiduelles, ce qui revient à ignorer certaines matrices de communication élémentaires. On obtient ainsi une matrice de communication partielle C', et il est assez clair que si l'on ignore suffisamment de références, C' aura un noyau non trivial. A la limite, si on ignore toutes les références, C' est vide et son noyau occupe tout l'espace, ce qui veut dire que le placement peut être choisi arbitrairement.

On pourrait penser que la démarche correcte est de calculer le volume des communications selon la formule [iii] et de le minimiser. Mais il s'agit là d'un raffinement illusoire. En effet, l'ensemble des itérations donnant lieu à communication peut se mettre sous la forme :

$$\mathcal{R}_{S\mathtt{A}} = \mathcal{D}_S - \{\vec{x} \mid \vec{x} \in \mathcal{D}_S, d(S, \mathtt{A}, \vec{x}) = 0\}.$$

Si la distance est identiquement nulle, alors $\mathcal{R}_{S\mathtt{A}}$ est vide. Sinon, soit, comme plus haut, n l'ordre de grandeur de la taille des boucles. $\mathcal{D}_S$ est un ensemble de taille $O(n^{|S|})$. Par contre, comme l'équation $d = 0$ définit un hyperplan, la taille du deuxième terme est $O(n^{|S|-1})$, ce qui est négligeable si n est grand. Il résulte de ces évaluations que l'on peut raisonner par tout ou rien : ou bien la matrice de communication élémentaire $C_{S\mathtt{A}}$ figure dans la matrice C', et les

4. Cette formulation suppose que dans chaque instruction il n'y a qu'une référence par tableau. La généralisation n'est qu'une question de notations.

communications correspondantes sont éliminées[5]. Ou bien $C_{S\mathtt{A}}$ n'est pas prise en compte, ce qui engendre un volume de communication qu'il faudra estimer, mais qui dépend peu du placement choisi. Nous donnerons plus loin, section 3.2.2, quelques indications sur la façon d'effectuer cette estimation.

En résumé, le problème est de sélectionner celles des matrices de communication élémentaires qui figureront dans C' de façon à ce que le noyau de C' ne soit pas trivial et que le volume des communications résiduelles soit minimal. Ceci suggère immédiatement l'utilisation d'un algorithme glouton. On suppose dans ce qui suit que les références du programme ont été numérotées de 1 à L, que la matrice élémentaire correspondant à la référence k est C_k, et que l'énumération a été faite par volume de communication *décroissant*.

Algorithme E

1. Initialement C' est la matrice vide.
2. Pour $k = 1, L$:
   (a.) Former $C'' = \begin{pmatrix} C' & C_k \end{pmatrix}$
   (b.) Si $\ker(C'') \neq \{0\}$, $C' = C''$.
3. Choisir comme vecteur de placement un vecteur quelconque de $\ker(C')$.

Le test de l'étape 2b, qui permet de savoir si $\ker(C')$ contient un vecteur non nul, sera appelé le *test de trivialité* dans ce qui suit.

Comme tout algorithme glouton, il n'y a pas de garantie que l'on trouve l'optimum. L'algorithme peut être très facilement transformé en algorithme *branch and bound* moyennant les modifications suivantes. On suppose qu'à chaque référence est associé un poids w_k qui mesure le volume des communications engendrées. Le problème est de minimiser la somme des poids des communications qui ne figurent pas dans la matrice de communication partielle. L'algorithme consiste en la construction d'un arbre de choix successifs. Un noeud de l'arbre est caractérisé par les deux ensembles C des références acceptées et D des références rejetées. On confond ici les ensembles de références et les matrices de communication associées. Pour un noeud de hauteur k, toutes les références de 1 à k ont été réparties :

$$C \cup D = [1, k],\ C \cap D = \emptyset.$$

La valeur du noeud est $\sum_{i \in D} w_i$. On suppose que l'on dispose d'une meilleure solution courante, de valeur W.

Algorithme B

1. Si pour le noeud courant, $\ker(C)$ est trivial, ce noeud est un échec.
2. Si la hauteur du noeud courant est L, ce noeud est un succès, mettre à jour W.
3. Si la valeur du noeud courant est supérieure à W, il est inutile de le développer.

5. ou, à tout le moins, transformées en décalages.

4. Sinon, soit k la hauteur du noeud courant. On obtient le fils gauche en adjoignant C_{k+1} à C, et le fils droit en l'adjoignant à D.

Des expérimentations limitées avec cet algorithme ont montré que les résultats ne sont pas sensiblement meilleurs qu'avec l'algorithme glouton, qui correspond d'ailleurs à la phase initiale de l'algorithme B si l'on convient de développer l'arbre en profondeur d'abord et en privilégiant le fils gauche.

Pour compléter la spécification de ces algorithmes, il faut expliquer comment déterminer efficacement le noyau de C, comment classer les références par volume de communication décroissant, et enfin comment calculer les constantes de décalage une fois que le vecteur de placement a été déterminé.

3.2.1. *Calcul des noyaux*

Le point important dans la détermination des noyaux des matrices de communication partielles est d'exploiter le fait que celles-ci sont construites incrémentalement et d'éviter de refaire plusieurs fois le même travail. La meilleure façon d'y parvenir est de reformuler le problème comme la résolution d'un système d'équations linéaires et homogènes [FEA 94]. On suppose pour cela que l'on a introduit des symboles pour représenter les composantes de $\vec{p}$, on utilisera ici les inconnues $p_1, \ldots, p_N$. La solution courante peut se représenter comme une substitution σ qui remplace certaines des p_i par des expressions linéaires des inconnues résiduelles. Dans ce qui suit, le symbole $\mathcal{C}$ représente le système d'équations :

$$(p_1, \ldots, p_N)C = 0$$

associé à la matrice C. A une certaine étape de l'algorithme on suppose que σ est la solution du système d'équations associé à la matrice de communication courante. Une base du noyau de celle-ci s'obtient tout simplement en appliquant σ au vecteur $(p_1, \ldots, p_N)$ et en séparant les coefficients de chacune des inconnues résiduelles.

Soit à adjoindre à $\mathcal{C}$ le système $\mathcal{C}_k$ associé à la matrice de communication élémentaire C_k. On commence par appliquer σ à $\mathcal{C}_k$ et on élimine les lignes triviales $(0 = 0)$ et les lignes redondantes. Les autres lignes forment un système d'équations linéaires et homogènes en les inconnues résiduelles, que l'on résout par la méthode de Gauss. On obtient une nouvelle substitution τ et on forme $\sigma' = \sigma \circ \tau$. Le noyau de la nouvelle matrice C' est trivial si et seulement si σ' donne la valeur 0 à toutes les variables.

On reprend l'exemple D. Ici les deux références correspondent à un même volume de communication et peuvent être traitées dans un ordre arbitraire. Le système correspondant à la référence 1 est :

$$\begin{aligned} p_1 - p_4 &= 0 \\ p_2 - p_3 &= 0 \end{aligned}$$

et la solution est la substitution $\sigma_1 = [p_1 \leftarrow p_4, p_2 \leftarrow p_3]$. Comme il y a deux inconnues résiduelles p_3 et p_4, le noyau correspondant est de dimension 2.

Le système suivant est :

$$\begin{aligned} p_1 - p_3 &= 0 \\ p_2 - p_4 &= 0 \end{aligned}$$

Si on applique σ_1 à ce système, on obtient deux fois $p_4 - p_3 = 0$ à quoi correspond $\tau = [p_3 \leftarrow p_4]$, puis $\sigma_2 = [p_1 \leftarrow p_4, p_2 \leftarrow p_4, p_3 \leftarrow p_4]$. Le vecteur de placement correspondant est $p_4(1,1,1,1)$, ce qui correspond bien au noyau trouvé plus haut.

3.2.2. *Classement des références*

Pour appliquer l'algorithme glouton, nous avons besoin de classer les références par volume de communication décroissant. Ici, il faut quelques informations sur la structure de la machine cible. Le cas le plus simple est celui d'une machine NUMA (Non Uniform Memory Access) : chaque processeur peut lire et écrire dans la mémoire des autres processeurs, mais le temps d'accès d'une mémoire éloignée est très supérieur à celui d'une mémoire locale. Dans le code objet, il n'y a pas de distinction entre accès local et accès éloigné. Le volume de communication associé à une référence est donc égal au nombre d'itérations de cette référence, c'est-à-dire au volume[6] de $\mathcal{D}_S$.

Cependant, presque toutes les machines NUMA sont équipées de caches cohérents. Si une cellule de mémoire est accédée, une copie en est conservée dans le cache, et les accès suivants sont locaux. Si cependant un autre processeur modifie la cellule considérée, le mécanisme de cohérence entre en jeu et provoque l'invalidation de la copie, et donc une nouvelle lecture distante au prochain accès. Le fonctionnement d'une machine à passage de message n'est pas très différent. Après qu'une cellule ait été lue par un processeur, celui-ci peut la conserver dans sa mémoire locale aussi longtemps qu'elle n'est pas modifiée par un autre processeur. A ce moment là, un nouvel échange de message est nécessaire. On peut résumer cette discussion en disant que ce qu'il faut compter, ce sont non pas les cellules, mais les *valeurs* accédées.

Soit à évaluer les volumes de communications associés aux références du programme G. Une étude attentive du programme montre, par exemple, que la valeur contenue dans `a(i,i)` ne varie pas pendant l'exécution des boucles sur `j` et `k` – ceci résulte de la valeur des bornes inférieures de ces boucles. Par contre, la valeur de `a(i,i)` change à chaque tour de la boucle sur `i`; il lui correspond donc $O(n)$ communications.

Par contre, il est facile de voir que chaque lecture de `a(j,k)` correspond à une valeur nouvelle, ce qui engendre un trafic en $O(n^3)$. On remarquera au passage que, comme on n'a aucune information sur les valeurs contenues dans la matrice `a`, on est obligé de supposer que les valeurs écrites par S sont toutes distinctes. Le trafic engendré par la référence à gauche de l'affectation est toujours pris égal au volume de $\mathcal{D}_S$. Le lecteur se convaincra facilement que le trafic des deux autres références est en $O(n^2)$.

---

6. Par volume d'un sous-ensemble de $\mathbb{N}^d$, nous entendons ici le nombre de points à coordonnées entières contenus dans ce sous-ensemble

Le point important est que l'analyse ci-dessus ne se fait pas à simple lecture du programme source. On se tromperait grossièrement, par exemple, en disant que puisque la référence `a(j,k)` dépend de deux indices de valeur maximale n, elle engendre un trafic en $O(n^2)$. Ce qu'il faut ici, c'est une analyse du flot des données dans l'esprit de [FEA 91], dans lequel on trouvera d'ailleurs une étude complète du programme G. Cette analyse fournit, pour chaque valeur lue, le repère de l'opération qui l'a produite, sous la forme d'une fonction du repère de l'opération de lecture. Moyennant les hypothèses que nous avons faites plus haut sur les indices des tableaux et les bornes des boucles, cette fonction est en général affine ou affine par morceaux.

Dans tous les cas, on est ramené au problème de compter les points d'un sous-ensemble de $\mathbb{N}^d$. Ceci est possible [TAW 91] puisque les bornes des boucles sont des formes affines, et qu'en conséquence les ensembles en question sont délimités par des polyèdres. Mais l'algorithme est complexe. Le résultat se présente sous forme de polynômes sur les bornes des boucles, et peut être difficile à interpréter. Enfin, il n'est pas sûr qu'une telle précision soit nécessaire. L'expérience a montré qu'il était suffisant de déterminer l'ordre de grandeur du volume de communication, qui est en relation directe avec la dimension du polyèdre qui borne les informations à transmettre. Dans le cas où ce polyèdre est $\mathcal{D}_S$, cette dimension est $|S|$ sauf, comme on l'a vu plus haut, dans certains cas pathologiques. Si le volume à transmettre est délimité par un ensemble de la forme $f(\mathcal{D}_S)$, où f est la fonction source fournie par l'analyse du flot des données, la détermination de sa dimension est plus complexe, parce qu'il faut déterminer si la fonction f est injective ou non. Mais comme, par hypothèse, f est affine, cette détermination n'est pas difficile.

Au total, on obtient de bons résultats en se contentant de déterminer la dimension de chaque référence et en les classant par ordre décroissant de cette dimension.

3.2.3. *Calcul des constantes de décalage*

Les mêmes considérations que ci-dessus s'appliquent à la résolution de [xv]. Il s'agit d'un système linéaire qui n'est pas en général homogène. Ce système peut parfaitement être surdéterminé et ne pas avoir de solution, comme le montre l'exemple :

```
program K
do i = 2,n
  a(i) = a(i-1)
end do
```

Ici encore, la solution est de classer les références par volume de communication décroissant, et de procéder à une résolution incrémentale avec *backtracking* si un système partiel se révèle sans solution. Ce calcul peut d'ailleurs être intégré à l'algorithme E. Il ne s'agit que de bien ordonner les équations.

3.3. *Règle des écritures locales et forme en assignation unique*

La règle des écritures locales (*owner computes rule*) veut qu'un calcul soit toujours effectué sur le processeur qui en héberge le résultat. On peut traduire cette règle dans notre formalisme en s'imposant de toujours satisfaire la condition de coupure associée à la référence de gauche de chaque affectation.

On a vu plus haut l'une des raisons qui justifient cette règle. Le volume de communication associé à la référence gauche est au moins aussi important que celui des autres références. L'autre raison est que cette règle simplifie la programmation en évitant la réalisation d'un protocole d'écriture distante.

La règle des écritures locales peut toujours être suivie, à ceci près qu'elle peut entrer en conflit avec les conditions que l'on s'est imposé sur la dimension du placement. Par exemple, il est impossible de trouver un placement non trivial pour :

```
do i = 1,n
  s = ...
end do
```

si l'on exige de suivre la règle des écritures locales. Ce problème ne se pose jamais si le programme a été expansé jusqu'à atteindre la forme à assignation unique, ce qui est un sous-produit immédiat de l'analyse du flot des données. Dans ce cas, chaque affectation est de la forme :

```
do x⃗ ∈ D_S
   A_S[x⃗] = ···
end do
```

et chaque instruction modifie un tableau différent. La condition de coupure associée à la référence à gauche s'écrit :

$$\Pi(S, \vec{x}) = \Pi(\mathtt{A}_S, \vec{x}).$$

Suivre la règle des écritures locales, c'est utiliser ces équations pour éliminer, au choix, soit les $\Pi(S, .)$ soit les $\Pi(\mathtt{A}_S, .)$.

Il n'empêche que la règle des écritures locales est une contrainte supplémentaire imposée au placement, et que comme toute contrainte supplémentaire elle peut dégrader la qualité du résultat. On trouvera dans [DAR 93] un exemple de ce phénomène.

3.4. *Placement à plusieurs dimensions*

On a vu plus haut que dans certains cas l'emploi d'une fonction de placement scalaire peut être pénalisant en ce que le parallélisme dégagé peut être insuffisant pour saturer les processeurs disponibles. On est naturellement conduit, par analogie avec la structure des réseaux en grille, à l'utilisation de fonctions de placement à g dimensions. Toutes les considérations de la

section précédente se transposent sans modification à ce nouveau contexte. Dans les prototypes [xii,xiii], on remplace les vecteurs de placement $\vec{p}_S, \vec{p}_{\mathtt{A}}$ par des matrices $P_S, P_{\mathtt{A}}$. On peut former une matrice de placement globale P de dimension $N \times g$, qui doit satisfaire à l'équation analogue à [xvi] :

$$PC = 0.$$

La matrice de communication C est la même que dans le cas unidimensionnel. Il est clair que les vecteurs lignes de P satisfont à l'équation [xvi], donc appartiennent au noyau de C. La solution du problème est donc toute trouvée : pour obtenir un placement à g dimensions, on sélectionne, si possible, g vecteurs dans $\ker(C)$.

Mais ceci ne suffit pas. Si en conséquence d'un tel choix, les vecteurs lignes d'une des matrices P_S ne sont pas linéairement indépendants, seul un sous-ensemble de la grille des processeurs va être utilisé par S, ce qui correspond à une perte de performance. On est donc conduit à renforcer le test de trivialité de l'algorithme glouton. Pour qu'une matrice C' convienne à l'étape 2b de l'algorithme E, il faut que son noyau contienne au moins g vecteurs dont les projections dans les sous-espaces correspondant à chaque instruction soient linéairement indépendantes. Ceci est évidemment impossible si la dimension de ce sous-espace est inférieure à g. On voit déjà qu'il est inutile de prendre g supérieur au niveau maximum d'emboîtement des boucles du programme. Mais même si cette condition est respectée, il peut arriver qu'une instruction particulière ait un niveau d'emboîtement inférieur à g. Il faudra alors se résigner à perdre de la puissance de traitement pour cette instruction.

Dans le cas où les calculs de noyaux se font par la méthode de Gauss incrémentale que nous avons présenté section 3.2.1, le test ci-dessus se fait très simplement. Il suffit, pour chaque instruction, de vérifier que les composantes de son vecteur de placement dépendent au moins de g variables indépendantes.

Il est important d'observer que le test de trivialité est d'autant plus contraignant que g est plus élevé. Il s'ensuit que le placement obtenu aura d'autant plus de communications résiduelles que sa dimension sera plus élevée. En compensation, le débit du réseau croît en général avec sa dimension. La question de savoir si cet effet compense l'augmentation des communications résiduelles doit être résolue expérimentalement dans chaque cas.

3.5. *Exemple*

Soit à trouver un placement affine pour l'exemple G. Il y a une instruction incluse dans 3 boucles et un tableau à 2 indices, soit 5 inconnues p_1 à p_5. Il y a d'autre part 5 références, mais deux d'entre elles sont identiques, ce qui donne donc 4 conditions de coupure, qui, après substitution de la forme affine

et identification des coefficients, donnent – c'est un cas particulier de [xvi] :

$$\begin{array}{lll} p_1 = 0, & p_2 = p_4, & p_3 = p_5 \\ p_1 = p_5, & p_2 = p_4, & p_3 = 0 \\ p_1 = p_4, & p_2 = 0, & p_3 = p_5 \\ p_1 = p_4 + p_5, & p_2 = 0, & p_3 = 0 \end{array}$$

Les volumes de communication associés aux références de ce programme ont été déterminées à la section 3.2.2. On trouve que la première référence (l'écriture) et la seconde correspondent à un trafic en $O(n^3)$, les deux suivantes à un trafic en $O(n^2)$ et la dernière en $O(n)$. On voit donc que la règle des écritures locales va être respectée tout naturellement. La résolution par la méthode de Gauss des premières équations conduit à la solution :

$$\sigma_1 = [p_1 \leftarrow 0, p_2 \leftarrow p_4, p_3 \leftarrow p_5].$$

Il y a encore deux inconnues résiduelles ; nous pouvons donc trouver un placement à deux dimensions. Si on reporte la solution ci-dessus dans les équations de la ligne suivante, on trouve :

$$0 = p_5,\ 0 = 0,\ p_5 = 0.$$

d'où la nouvelle solution :

$$\sigma_2 = [p_1 \leftarrow 0, p_2 \leftarrow p_4, p_3 \leftarrow 0, p_5 \leftarrow 0].$$

Il n'y a plus maintenant qu'une seule variable libre, la dimension maximale du placement est donc 1. Le lecteur se convaincra facilement que si l'on tente d'exploiter les autres lignes, on ne trouve plus que le placement trivial. Il est facile de voir également qu'avec les placements proposés, on trouve une solution aux équations [xv] avec des constantes de décalage toutes nulles.

La conclusion de cette analyse est qu'il existe tout d'abord un placement à deux dimensions :

$$\Pi(S, i, j, k) = \begin{pmatrix} j \\ k \end{pmatrix},$$

qui demande deux communications d'ordre n^2 et une communication d'ordre n, et un placement à une dimension : $\Pi(S, i, j, k) = j$, qui ne demande qu'une communication d'ordre n^2 et une communication d'ordre n. On trouverait d'ailleurs le placement «symétrique» : $\Pi(S, i, j, k) = k$ en traitant les équations de la troisième ligne avant celles de la seconde.

3.6. *Distribution*

Le nombre de valeurs distinctes prises par le placement donne le nombre de processeurs virtuels nécessaires à l'exécution du programme objet. Il n'a aucune raison de correspondre au nombre de processeurs physiques de la machine cible. L'adaptation se fait par utilisation d'une fonction de repliement

[iv] qui n'est pour le moment soumise à aucune contrainte. Le choix de cette fonction correspond à la détermination d'une distribution dans les langages à parallélisme de données. On utilise en général des distributions simples, comme la distribution par blocs :

$$\xi(x) = x \div B,$$

la distribution cyclique :

$$\xi(x) = x \bmod P,$$

ou la distribution cyclique par blocs :

$$\xi(x) = (x \div B) \bmod P.$$

Un critère de choix est l'observation des communications résiduelles. Si celles-ci sont des décalages, l'emploi d'une distribution par bloc permet une réduction supplémentaire du trafic. La distribution par bloc n'apporte par contre aucun avantage pour les communications générales ou les diffusions. Dans ce cas, on doit préférer la distribution cyclique, qui permet un meilleur équilibrage de la charge des processeurs. On peut en gros suggérer les règles suivantes :

— Si les communications résiduelles sont en majorité des communications générales, utiliser une distribution cyclique.

— Si les communications résiduelles sont des décalages, choisir une distribution cyclique par bloc. La taille du bloc doit être choisie grande devant la longueur des vecteurs de communication.

3.7. *Génération de code*

Le problème de la génération du code parallèle connaissant un placement est en dehors du sujet de cet article. Il fait d'ailleurs l'objet de recherches extrêmement actives à l'échelle mondiale.

Une des méthodes les plus simples [ZIM 88] consiste à reproduire dans chaque processeur le programme original, en protégeant chaque instruction par une garde :

$$\textbf{if } \Pi(S, \vec{x}) = q \textbf{ then } S$$

où q est le numéro du processeur courant. Des gardes analogues permettent de savoir, pour chaque référence, si elle est locale ou s'il faut appeler un module d'échange de messages dont l'écriture peut être simplifiée en faisant appel à des bibliothèques du genre de PVM [BEG 91]. Le programme ainsi obtenu est en général inefficace, mais il peut être optimisé à l'aide de techniques de parcours de polyèdres maintenant bien maîtrisées [ANC 91, COL 94, XUE 94, KEL 94].

4. Conclusion

La technique présentée dans cet article peut être résumée de la façon suivante :

— Faire subir au programme une phase de pré-traitement, non présentée ici, en particulier pour régulariser les accès aux tableaux, déterminer le grain de parallélisme et la taille des boucles, analyser le flot des données, éventuellement expanser des scalaires ou des tableaux.

— Choisir la dimension du placement souhaité en prenant en compte l'architecture de la machine cible (nombre de processeurs, topologie du réseau) et les caractéristiques du programme source (grain du parallélisme, taille des boucles).

— Appliquer l'algorithme E pour trouver un placement minimisant les communications résiduelles.

— Choisir une fonction de distribution suivant les indications de la section **3.6** et générer le code objet suivant les indications de la section **3.7**.

Plusieurs implémentations ont été réalisées pour des architectures diverses ; on citera en particulier celles de [PLA 95], qui prennent la forme de traducteurs de Fortran 77 vers le CM-Fortran ou le CRAFT du Cray T3D. Les temps de compilation sont acceptables et les performances du programme objet sont encourageantes. Naturellement, il reste de gros efforts à faire avant que ces maquettes deviennent des produits exploitables.

On peut réinterpréter cette technique dans le cadre des recherches modernes sur la parallélisation automatique [FEA 92c, FEA 95], dans lesquelles un programme parallèle est représenté comme un ordre partiel sur ses opérations. Les techniques d'ordonnancement [FEA 92a, FEA 92b] cherchent les anti-chaînes de cet ordre et s'adaptent bien aux architectures synchrones. La technique du placement, au contraire, cherche les chaînes (ensembles d'opérations totalement ordonnées) et s'adapte aux architectures distribuées. Les deux techniques sont des méthodes de compilation à part entière. Leurs fonctionnements sont en gros opposés. La méthode de l'ordonnancement *ajoute* des arcs à l'ordre partiel donné, jusqu'à ce qu'il soit suffisamment régulier pour être représenté par une séquence de synchronisations globales ou barrières ; c'est la forme `SEQ of PAR` de [BOU 93]. Inversement, la méthode du placement *enlève* des arcs jusqu'à ce que l'ordre apparaisse comme la composition parallèle d'un ensemble de chaînes, la forme `PAR of SEQ`. Les arcs omis doivent être rétablis sous forme de communications résiduelles.

Il n'y a pas de raison objective de préférer l'une ou l'autre méthode. Certains ordres partiels prennent la forme `SEQ of PAR` en ajoutant très peu d'arcs, d'autres deviennent `PAR of SEQ` au moyen de très peu d'omissions. Par contre, chaque architecture a sa forme naturelle, et dans certains cas – les architectures synchrones à mémoire répartie et les réseaux systoliques – il faut à la fois un ordonnancement et un placement pour engendrer le programme parallèle. Ce n'est pas ici le lieu de débattre de cet aspect, pour lequel nous renvoyons à [FEA 95].

Dans le cas où l'ordre partiel initial est déjà sous la forme `PAR of SEQ`, l'algorithme devrait trouver immédiatement toutes les chaînes indépendantes. Comme le montre l'exemple D, cet objectif n'est pas tout à fait atteint. Une étude directe permettrait de trouver un placement bidimensionnel analogue à celui de l'exemple EE, donc avec $O(n^2)$ chaînes. Le fait de se restreindre aux placements affines ne permet de trouver qu'un seul vecteur de placement, qui génère n chaînes. Il serait donc intéressant d'étendre les méthodes présentées ici à des placements plus généraux. Cependant, l'exemple H montre que les progrès possibles dans cette direction seront forcément limités.

Une autre contrainte résulte du fait que le placement et la distribution ne sont que des intermédiaires, l'objectif final étant l'écriture du code parallèle. En l'état actuel de la technique, ceci n'est envisageable que pour des placements affines ou affines par morceaux. Or, l'exemple H montre que des fonctions d'indexation très simples suffisent pour engendrer des placements complexes. Le traitement exact de fonctions d'indexation complexes est donc probablement impossible. Une solution de compromis est évidemment d'ignorer les indexations complexes. Il en résultera des communications résiduelles, dont on ne peut qu'espérer qu'elles ne dégraderont pas trop les performances du programme cible. Il est peut-être envisageable de rechercher un placement à partir d'une analyse approximative du programme source [COL 95], mais tout reste à faire dans cette direction.

Certaines architectures distribuées ont été optimisées pour traiter plus efficacement certains types de communications. On a vu le cas des décalages, mais il en est d'autres, comme les diffusions (communications de un vers plusieurs processeurs) et leur inverse les réductions (de plusieurs vers un processeur). Une étude [PLA 95] récente a montré que la prise en compte même partielle de ces optimisations augmentait considérablement les performances des programmes objets.

Enfin, il semble clair que quand la taille du programme augmente, le nombre de contraintes augmente alors que le nombre de paramètres à ajuster reste fixe si le nombre de tableaux utilisés reste fixe. En conséquence, la qualité du placement obtenu diminue. Il paraît donc intéressant de découper un programme en phases que l'on place de façon indépendantes, les phases étant reliées par des opérations de redistribution d'ailleurs prévues dans un langage comme HPF. Si la mécanique de ce découpage paraît simple – il s'agit de renommer les tableaux – le choix des points de coupure est sans aucun doute un problème difficile qui mérite de nouvelles études.

5. Bibliographie

[AHO 86] Aho A., Sethi R. and Ullman J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AMM 92] Ammarguellat Z.. « A control-flow normalization algorithm and its complexity ». *IEEE Transactions on Software Engineering*, vol. 18, n° 3, p. 237–251, 1992.

[ANC 91] Ancourt, C. and Irigoin, F. « Scanning polyhedra with DO loops ». *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, p. 39–50, 1991.

[BAU 94] Bau D., Kokula I., Kotlyar V., Pingali K., and Stodghill P. « Solving alignment using elementary linear algebra » *Seventh International Workshop on Languages and Compilers for Parallel Computing*, LNCS 892, 1994

[BEG 91] Beguelin A., Dongarra J., Geist A., Manchek R., and Sunderam V. *A user guide to PVM: Parallel Virtual Machine.* Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, 1991.

[BOU 93] Bougé L. « Le modèle de programmation à parallélisme de donnés : une perspective sémantique. » *T.S.I.*, vol. 12, n° 5, p. 541–562, 1993.

[COL 94] Collard J-F, Feautrier P., and Risset T. « Construction of DO loops from systems of affine constraints. » *Parallel Processing Letters*, to appear, 1994.

[COL 95] Collard J-F., Barthou D., and Feautrier P. « Fuzzy array dataflow analysis. » *ACM Symp. on Principles and Practice of Parallel Programming*, p. 92–102, 1995.

[DAR 93] Darte A. and Robert Y. *A graph-theoretic approach to the alignment problem.* Technical Report 93-20, LIP-IMAG, 1993.

[DAR 94a] Darte A. and Robert Y. « Mapping uniform loop nests onto distributed memory architectures. » *Parallel Computing*, vol. 20, p. 679–710, 1994.

[DIO 94b] Dion M. and Robert Y. *Mapping Affine Loop Nests: New Results. HPCN Int. Conf.* p. 184–189, Springer, 1995

[FEA 88] Feautrier P. « Array expansion. » *ACM Int. Conf. on Supercomputing, St-Malo*, p. 429–441, 1988.

[FEA 91] Feautrier P. « Dataflow analysis of scalar and array references. » *Int. J. of Parallel Programming*, vol. 20, n° 1, p. 23–53, 1991.

[FEA 92a] Feautrier P. « Some efficient solutions to the affine scheduling problem, I, one dimensional time. » *Int. J. of Parallel Programming*, vol. 21, n° 5, p. 313–348, 1992.

[FEA 92b] Feautrier P. « Some efficient solutions to the affine scheduling problem, II, multidimensional time. » *Int. J. of Parallel Programming*, vol. 21, n° 6, p. 389–420, 1992.

[FEA 92c] Feautrier. « Techniques de parallélisation. » *Algorithmique Parallèle*, p. 243–257, Masson, 1992.

[FEA 94] Feautrier. « Toward automatic distribution. » *Parallel Processing Letters*, vol. 4, n° 3, p. 233–244, 1994.

[FEA 95] Feautrier. « Compiling for massively parallel architectures: a perspective. » *Microprogramming and microprocessing*, vol. 41, p. 425–439, 1995

[HPF 94] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.1.* Technical Report, Rice University, November 1994.

[JIN 90] Li Jinke and Chen M. « Index domain alignment: minimizing cost of cross-referencing between distributed arrays. » *Proc. Third Symp. on the Frontiers of Massively Parallel Computation*, p.424–433, 1990.

[KEL 94] Kelly W., Pugh W., and Rosser E. *Code Generation for Multiple Mappings.* Technical Report CS-TR-3317, U. of Maryland, 1994.

[MAY 93] Maydan D., Amarasinghe S., and Lam M. « Array dataflow analysis and its use in array privatization. » *Proc. of ACM Conf. on Principles of Programming Languages*, p. 2–15, 1993.

[PLA 95] Platonoff A. Contribution à la distribution automatique des données pour machines massivement parallèles. thèse de doctorat, Université P. et M. Curie, 1995.

[PUG 93] Pugh W. and Wonnacott D. « An evaluation of exact methods for analysis of value-based array data dependences. » *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, 1993.

[RAM 91] Ramanujan J. and Sadayappan P. « Compile-time techniques for data distribution in distributed memory machines. » *IEEE Trans. on Paralell and Distributed Systems*, vol. 2, p. 472–482, 1991.

[TAW 91] Tawbi N. Parallélisation Automatique : Estimation des Durées d'Exécution et Allocation Statique de Processeurs. thèse de doctorat, Université P. et M. Curie, Paris, 1991.

[XUE 94] Xue J. « Automatic non-unimodular transformations of loop nests. » *Parallel Computing*, vol. 20,n° 5, p. 711–728, 1994.

[ZHI 89] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. « An empirical study on array subscripts and data dependencies. » *1989 Int. Conf. on Parallel Processing*, p. 145–152, 1989.

[ZIM 88] Zima H., Bast H., and Gerndt M. « SUPERB : A tool for semi-automatic MIMD/SIMD parallelization. » *Parallel Computing*, vol. 6, p. 1–18, 1988.