Université Strasbourg I – Louis Pasteur
U.F.R. de mathématique et d'informatique
École Doctorale des Sciences Pour l'Ingénieur

Étude d'outils logiciels pour la parallélisation et la transformation de programmes dans les applications de calculs numériques

THÈSE

présentée et soutenue publiquement le 5 juillet 2001

pour l'obtention du

Doctorat de l'Université Strasbourg I – Louis Pasteur
(discipline informatique)

par

Frédérique VOISIN-DEMERY

Composition du jury

Président et rapporteur interne :	Jan DUSEK, Professeur Université Louis Pasteur, Strasbourg
Rapporteurs externes :	Denis TRYSTRAM, Professeur INPG, Grenoble Serge PETITON, Professeur Université des Sciences et Technologies de Lille
Examinateur :	Philippe CLAUSS, Maître de Conférence, HDR Université Louis Pasteur, Strasbourg
Directeur de thèse :	Guy-René PERRIN, Professeur Université Louis Pasteur, Strasbourg

Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection / ICPS – UA CNRS 7005

Mis en page avec la classe thloria.

Remerciements

Je suis tout d'abord reconnaissante à Guy-René Perrin d'avoir proposé ce sujet de thèse et de m'avoir encadrée tout au long de ce travail.

Je remercie aussi Jan Dusek qui a bien voulu accepter la double casquette de président de jury de cette thèse et de rapporteur interne ; il a aussi soumis un problème issu de la mécanique des fluides à l'ICPS, problème qui est à l'origine d'une partie conséquente du travail effectué pour cette thèse. Merci à Denis Trystram et à Serge Petiton, rapporteurs de cette thèse, pour leurs remarques constructives sur le manuscrit et lors de la soutenance ; merci enfin à Philippe Clauss, qui a bien voulu être examinateur, pour son regard pertinent sur ce travail.

Je sais aussi gré à Paul Feautrier de son aide via le réseau, sans laquelle la quatrième et dernière partie de cette thèse n'aurait pas pu voir le jour.

Je tiens bien sûr à remercier tous les membres de l'ICPS pour leur accueil chaleureux à Strasbourg, avec une pensée particulière pour
– les occupants du bureau B221, Philippe Clauss, Vincent Loechner et plus tard Benoit Meister, pour leurs explications sur leurs travaux dont il est question dans cette thèse ;
– l'équipe Pei, Eric Violard et Stéphane Genaud ;
– Romaric David, toujours disponible pour répondre à mes questions sur l'utilisation de l'Origin 2000 ;
– et enfin tous ceux qui furent mes *co-bureau*, plus ou moins longtemps, entre autres Philippe Gerner et Michel Salomon.

Citons aussi tous les pourvoyeurs de bonne humeur, gage d'un bon moral durant la préparation de ce travail : la funky_team et mes camarades de l'ADDAℓ.

Et enfin, *last but not least*, je n'oublie pas mes parents, toujours encourageants, et Marc, présent chaque jour à mes côtés.

Résumé

Ce travail vise à étudier l'adéquation de divers outils de parallélisation s'inscrivant dans le cadre du parallélisme de données à l'écriture, la transformation et la parallélisation de programmes de calculs scientifiques.

Après une première partie consacrée à la présentation de deux outils, le deuxième chapitre s'intéresse à l'utilisation de PEI, langage équationnel, pour la transformation de programmes et leur parallélisation : on montre comment ce langage permet de prendre un programme pour matrices pleines, et de le transformer en programme pour matrices creuses par la définition de quelques fonctions. La parallélisation du programme résultant s'en suit, grâce à la représentation basée sur l'alignement des données que permet ce langage.

Dans une troisième partie, on montre l'intérêt de la modélisation polyédrique des nids de boucles telle qu'elle est faite par le logiciel OPERA, en s'appuyant sur la parallélisation d'un code de mécanique des fluides. Ce type de modélisation permet en effet de représenter visuellement le problème, ce qui aide à la détection du parallélisme. D'autre part, nous montrons comment la modélisation mathématique sous-jacente permet de minimiser les communications dans ce problème, par application de techniques d'optimisation d'accès à la mémoire aux cas de communications. Une série d'expérimentations permet finalement de valider l'approche.

Enfin, une dernière partie présente une comparaison entre deux outils pour la parallélisation de la factorisation de Cholesky. L'un est axé sur une approche par parallélisme de données alors que l'autre propose une approche centrée sur les instructions. Il s'avère que cette dernière approche permet une parallélisation automatique un peu meilleure, alors que la première propose une meilleure représentation du programme en vue de sa compréhension.

Mots-clés: parallélisme, transformation de programmes, applications numériques, parallélisation automatique, parallélisme de données, modèle polyédrique, minimisation des communications, calculs creux

Abstract

This work studies several programs' parallelization and transformation tools, coming from the data parallelism framework, in the context of scientific computation.

After a first part dedicated to the description of two tools, a second chapter deals with PEI, an equational language, and its interest for program transformation and parallelization. A program for dense matrix can be transformed in a program for sparse matrix thanks to this language, with the simple definition of some functions. Matrix vector multiplication is used as an example : the dense program is first transformed into a sparse one and then easily parallelized, due to the program representation based on data alignment.

A third part presents a study about the interest of the OPERA tool for the parallelization of a fluid mechanics program. OPERA's nest loops modelization, based on the polytope model, allows a useful visual representation of the problem, helping to detect parallelism. Furthermore, the mathematical modelization underlying the tool allows to minimize the number of communications in this problem. More precisely, techniques of memory access optimization depending on the polytope model can easily be adapted to the problem of communications minimization. At last, a series of experimentations helps to validate this approach.

A fourth and last part shows a comparison between two parallelization tools, on the Cholesky factorization's probleme. One of them has a data parallel approach whereas the other deals with instructions rather than datas. This last one appears to be a bit more efficient when it comes to automatic parallelization, whereas the first one turns out be more helpfull to understand programs behaviour.

Keywords: parallel computing, programs transformation, numerical applications, automatic parallelization, data parallelism, polytope model, communications minimization, sparse computation

Table des matières

Introduction

Plus vite, plus haut, plus fort
JO Sydney, Septembre 2000

La devise précédente illustre bien l'évolution de notre société : nous voulons plus, plus d'argent (plus de temps libre, plus de sécurité, plus de liberté, ...), plus vite (démarrage en flèche des *start-up*, communication instantanée par *e-mail*, ...) et mieux ! Dans ce contexte, la nécessité d'utiliser des super-calculateurs, ayant parfois des centaines de processeurs n'est plus à démontrer. Malgré l'augmentation régulière des performances des processeurs, nous voulons travailler avec des données de taille toujours plus importante, et nous voulons travailler de plus en plus rapidement. Ainsi, nous pouvons faire des simulations toujours plus précises et toujours plus rapides afin de connaître le temps qu'il fera dans cinq jours, ou de simuler des explosions nucléaires plutôt que de les réaliser sous les eaux du Pacifique.

Néanmoins, l'écriture de programmes efficaces à destination des machines multiprocesseurs est loin d'être triviale. Dans la pratique, on peut différencier deux types d'approches pour ce qui est d'aborder la parallélisation des programmes. D'un côté, on trouve le parallélisme explicite. Il s'agit d'un paradigme à l'intérieur duquel le programmeur doit analyser son programme pour y découvrir le parallélisme. Il doit aussi réfléchir aux échanges de données que son analyse induit et coder explicitement ces échanges. Cette approche est très répandue malgré les difficultés qu'elle présente, toutes les tâches de la parallélisation incombant au programmeur. La mise au point des bibliothèques de passage de messages, telles MPI [83] ou PVM [39] qui ont réussi à s'imposer en standard sont sans doute la cause du développement et de la popularité de cette approche.

Un peu plus tard s'est développé un parallélisme implicite, destiné à alléger la tâche du programmeur. L'insertion de directives à l'attention du compilateur dans le code séquentiel, mettant en évidence le parallélisme simplifie grandement l'écriture des programmes pour machines multi-processeurs. Néanmoins, notre programmeur reste tributaire de la façon dont le compilateur va traiter les informations fournies, et il a un moindre contrôle sur le déroulement et les performances de son programme. Cette approche, centrée sur les données plus que sur sur les instructions du programme, ne connaît pas (encore ?) le succès des approches précédentes. La difficulté à faire émerger un standard – après High Performance Fortran (HPF) [51], OpenMP [27] – l'absence de compilateurs intégrant toutes les spécifications des standards, ainsi que l'impossibilité d'exprimer des structures de données complexes ou irrégulières sont sans doute responsables, ainsi que la difficulté à obtenir des réalisations vraiment efficaces.

Les deux approches précédentes restent très pragmatiques et s'attachent à la mise en œuvre du parallélisme. Aussi aisée cette mise en œuvre soit-elle, l'homme derrière la machine est toujours en charge d'une des tâches pricipales de la parallélisation : la détection du parallélisme dans le calcul qu'il veut effectuer et/ou la transformation de ces calculs pour faire émerger le parallélisme. Des techniques d'analyse et de transformations des programmes ont donc été, et sont toujours, étudiées afin de faciliter cette étape. En particulier, une littérature riche s'est développée autour de l'analyse des nids de boucles qui sont au cœur des programmes de simulation numérique, grands consommateurs de ressources et demandeurs de parallélisme.

La détection du parallélisme potentiel dans un code informatique nécessite deux phases : une analyse du programme permettant de mettre en évidence quelles sont les transformations autorisées ou non – en effet, la parallélisation ne doit pas se faire au détriment de la correction du programme – et la détection des transformations utiles.

Si des techniques simples de parallélisation automatique ont été mises en place dans des outils commerciaux (par exemple Developper Magic de SGI pour produire du code OpenMP), il n'en reste pas moins que bon nombre d'outils d'analyse complexe restent éloignés du code réel : nécessité de donner le bout de code à traiter sous une forme particulière, pas d'analyse inter-procédurale, etc. On se trouve alors devant une difficulté : d'un côté, des codes de calculs demandent à être parallélisés efficacement ; de l'autre, des outils ou techniques de parallélisation pointus existent, mais sont pour l'instant inadaptés à la parallélisation d'applications réelles.

Le travail présenté dans la suite s'attelle à combler ce fossé et vise à étudier les techniques et outils de parallélisation et d'analyse de programmes développés à l'ICPS. Il s'agit d'analyser quel est leur intérêt pour la parallélisation, en dépassant les cas d'écoles.

Le premier chapitre présente brièvement deux outils développés à l'ICPS : OPERA (Outils de Parallélisation des Équations Récurrentes Affines) et Pei (Parallel Equation Interpretor). Le second chapitre présente un travail original sur l'utilisation de Pei dans le cadre de la transformation de programmes manipulant des structures de données denses en programmes à structures creuses et de leur parallélisation. On passe ensuite à la parallélisation d'un code de mécanique des fluides : OPERA nous permet d'analyser les communications induites par une parallélisation possible, alors qu'on utilisera Pei pour valider une transformation de cette parallélisation permettant d'optimiser les communications. Enfin, on présente une étude sur les techniques de parallélisation issues notamment des méthodes polyédriques : en prenant l'exemple de la factorisation de Cholesky, on compare deux outils – OPERA et PAF – notamment en terme de communications induites par les parallélisations proposées.

Chapitre 1

Présentation des outils développés à l'ICPS

1.1 Introduction à Pei

Pei a été défini dans le but de fournir une notation mathématique permettant de décrire et transformer des programmes parallèles [104, 41]. La description d'un programme repose sur les données de ce programme, auxquelles on applique un certain nombre d'opérations. Les données sont vues comme des valeurs sur un ensemble de points discrets arrangés selon une certaine géométrie. On peut appliquer des fonctions sur ces valeurs, ce qui correspond à des calculs ; on peut aussi déplacer des données dans le domaine géométrique, ce qui peut être vu comme des dépendances ou communications ; enfin, on peut changer la forme du domaine géométrique, ce qui correspond à transformer le programme en changeant la distribution des données.

1.1.1 Opérations sur les champs de données

Un programme est une relation entre des *multi-ensembles* de valeurs, qu'on peut considérer comme les entrées et sorties du programme. Il est nécessaire d'avoir une organisation pour ces données afin de travailler avec elles. Par exemple, les matrices sont représentées par des tableaux qui sont en fait des fonctions sur des indices. L'ensemble des indices, que nous appelons *domaine de référence*, est une partie de $\mathbb{Z}^n$. Un tel ensemble de valeurs placées sur un domaine de référence discret est appelé *champ de données*, illustré par la figure 1.1. Il s'agit de la brique de base sur laquelle nous opérons en Pei.

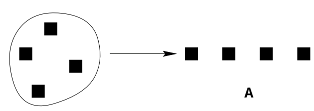
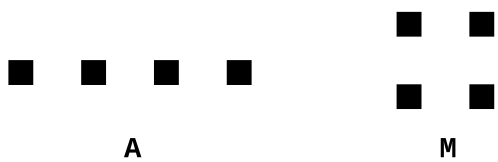


Fig. 1.1 – Multi-ensemble de valeurs et champ de données associé

Un programme est composé d'un ensemble d'équations mettant en jeu des champs de données, sur lesquels ont peut appliquer diverses opérations [41]. La syntaxe des fonctions permettant de décrire ces opérations est inspirée du lambda-calcul. Voyons maintenant quelles sont ces opérations.

- **Changement de base** : cette opération modifie la géométrie du référentiel de base. Elle est représentée par l'opérateur `::` et illustrée par la figure 1.2. Sur cet exemple, on modifie la géométrie du champ de données `A` à une dimension pour le transformer en un champ `M` à 2 dimensions, par le biais de la fonction `align`.



`M = align : : A`
`align` $= \lambda i|(0 \leq i \leq 3)(i\%2, i \div 2)$

Fig. 1.2 – Changement de base

- **Opération de superposition** : cette opération notée `/&/` permet de construire un champ de données à partir de deux autres champs. Les valeurs de ce troisième champ sont la séquence des valeurs des deux champs que l'on a superposés. La figure 1.3 illustre la superposition de deux champs de données `A` et `B`.

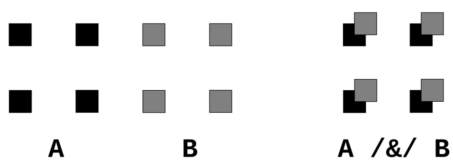
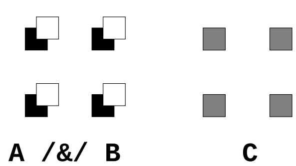


Fig. 1.3 – Opérateur de superposition

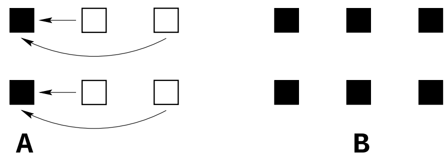
- **Opération fonctionnelle** : elle permet de faire des calculs sur les valeurs des champs de données, comme le montre la figure 1.4 qui illustre l'addition de deux champs de données superposés. Cette opération est notée par $\rhd$.



`C` $=$ add $\rhd$ `A /&/ B`
add $= \lambda(a, b).a + b$

Fig. 1.4 – Opération fonctionnelle

- **Opération géométrique** : notée par $\lhd$, elle permet de modifier l'emplacement des données sur le domaine géométrique. Si on considère chaque point du domaine comme un processeur virtuel, cette opération induit alors des comunications pour aligner des données en vue d'un calcul. Cette opération est illustrée par la figure 1.5



$\texttt{B} = \texttt{A} \lhd \texttt{spread}$

$\texttt{spread} = \lambda\, i \mid (0 \leq i \leq 2, 1 \leq j \leq 2).(1, j)$

Fig. 1.5 – Opération géométrique

L'exemple du produit matrice-vecteur donné en figure 1.6 permet de mieux appréhender ces différentes opérations.

```
MatVec : (A,V) ↦ X
{
matrix::A = A
align::B = V
P = prod ▷ (A/&/(B ◁ spread))
S = add ▷ (P/&/(S ◁ shift))
X = project::(S ◁ last)
}
matrix   =  λ(i,j) |(1≤i≤n, 1≤j≤n) . (i,j)
align    =  λ(i,j) |(i=1, 1≤j≤n) . j
project  =  λ(i,j) |(1≤i≤n, j=n) . i
shift    =  λ(i,j) |(1≤i≤n, 1<j≤n) . (i,j−1)
last     =  λ(i,j) |(1≤i≤n, j=n) . (i,j)
spread   =  λ(i,j) |(1≤i≤n, 1≤j≤n) . (1,j)
add      =  id # λ(a·b) . a+b
prod     =  λ(a·b) . a×b
```

Fig. 1.6 – Multiplication matrice-vecteur en Pei

On peut expliquer chaque équation de la façon suivante :

- La première équation définit le placement du champ de données `A` donné en entrée. La fonction `matrix`, définie sur $\mathbb{Z}^2$ est un changement de base représentant l'identité sur $[1..n] \times [1..n]$. Cela signifie que que les valeurs de `A` sont placées sur le carré $[1..n] \times [1..n]$.
- La deuxième équation définit implicitement le champ de données `B`. La fonction `align`, définie sur $\mathbb{Z}^2$, est un changement de base, qui, appliqué à la donnée unidimensionnelle `B` permet d'obtenir `V` placée sur un domaine bi-dimensionnel. On notera que la fonction `align` définit le contexte du programme.
- Pour la troisième équation, on applique tout d'abord une opération géométrique sur `B`, ceci afin de placer la “première ligne” de `B` suivant les colonnes (direction $(1, 0)$ dans $\mathbb{Z}^2$). Ensuite, grâce à l'opérateur de superposition `/&/`, on obtient la superposition des champs de données `A` et (`B` $\lhd$ `spread`) ce qui permet d'effectuer la multiplication qui apparaît dans le produit matrice-vecteur. Il reste ensuite à effectuer la somme de ces

valeurs.
- La quatrième équation définit le champ de données `S` comme résultat d'une opération fonctionnelle sur 2 champs de données, ce qui permet de calculer la somme sur les lignes. On remarque que les sommes partielles permettant d'arriver au résultat sont accumulées dans le champ de données `S`.
- La dernière équation permet d'obtenir le résultat final, le champ de données `X` unidimensionnel, par projection de la dernière colonne de `S` sur $\mathbb{Z}$ par le biais du changement de base `project`.

L'ensemble de ces opérations est illustré par la figure 1.7.

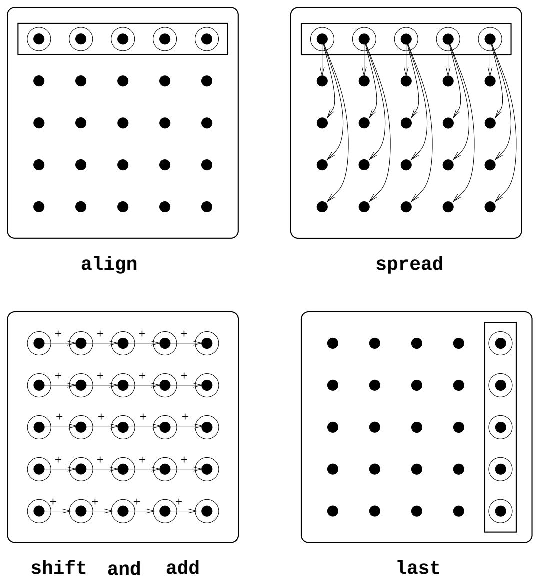


FIG. 1.7 – Illustration de la multiplication matrice-vecteur avec PEI.

1.1.2 Sémantique

Les concepteurs de PEI ont défini une sémantique pour toutes ces opérations. Tout d'abord un champ de données, placement de valeurs d'un multi-ensemble V sur un domaine discret, est défini comme une application $v : \mathbb{Z}^n \to V$. Néanmoins, le choix du placement est arbitraire : on peut définir un champ de données à partir de la bijection σ qui le sépare d'un placement de référence, et de la fonction v. Un champ de données est donc un couple (v, σ) pour lequel on suppose vérifiée la contrainte suivante : $dom(v) \subseteq dom(\sigma)$. Voyons maintenant la sémantique des opérations appliquées à un champ de données `X`$= (v, \sigma)$.
- **Opération fonctionnelle** : soit `f` une fonction $\mathtt{f} : V \to W$ où W est un ensemble de valeurs, telle que $img(v) \subseteq dom(\mathtt{f})$, alors $\mathtt{Y} = \mathtt{f} \rhd \mathtt{X}$ définit `Y` comme $(\mathtt{f} \circ v, \sigma)$
- **Opération géométrique** : soit `g` une fonction $\mathtt{g} : \mathbb{Z}^n \to dom(v)$ telle que $dom(\mathtt{g}) \subseteq dom(\sigma)$, alors $\mathtt{Y} = \mathtt{X} \lhd \mathtt{g}$ définit `Y` comme $(v \circ \mathtt{g}, \sigma)$.
- **Changement de base** : soit `h` une fonction bijective d'une partie de $\mathbb{Z}^n$ vers une partie de $\mathbb{Z}^p$ telle que $dom(v) \subseteq dom(\mathtt{h})$, alors $\mathtt{Y} = \mathtt{X} \lhd \mathtt{h}$ définit `Y` comme $(v \circ \mathtt{h}^{-1}, \sigma \circ \mathtt{h}^{-1})$.

À partir de cela, on peut définir une série de règles qui permettent de combiner les diverses opérations entre elles. En particulier :

$$
\begin{aligned}
\mathtt{f1} \rhd (\mathtt{f2} \rhd \mathtt{X}) &= (\mathtt{f1} \circ \mathtt{f2}) \rhd \mathtt{X} && (1.1)\\
\mathtt{X} \lhd (\mathtt{g1} \circ \mathtt{g2}) &= (\mathtt{X} \lhd \mathtt{g1}) \lhd \mathtt{g2} && (1.2)\\
& \quad \textit{si } \textit{codom}(\mathtt{g1}) \subseteq \textit{dom}(v) &&\\
(\mathtt{X1/\&/X2}) \lhd \mathtt{g} &= (\mathtt{X1} \lhd \mathtt{g})\mathtt{/\&/}(\mathtt{X2} \lhd \mathtt{g}) && (1.3)\\
\mathtt{h::}(\mathtt{X1/\&/X2}) &= (\mathtt{h::X1})\mathtt{/\&/}(\mathtt{h::X2}) && (1.4)\\
\mathtt{h::}(\mathtt{f} \rhd \mathtt{X}) &= \mathtt{f} \rhd (\mathtt{h::X}) && (1.5)\\
& \quad \textit{si } \textit{dom}(v) \subseteq \textit{dom}(\mathtt{h}) &&\\
\mathtt{h::}(\mathtt{X} \lhd \mathtt{g}) &= (\mathtt{h::X}) \lhd \mathtt{h} \circ \mathtt{g} \circ \mathtt{h}^{-1} && (1.6)\\
& \quad \textit{si } \textit{codom}(\mathtt{g}) \subseteq \textit{dom}(v) \subseteq \textit{dom}(\mathtt{h}) &&\\
& \quad \textit{et } \ \textit{codom}(\mathtt{g}) \subseteq \textit{dom}(v) &&\\
& && (1.7)
\end{aligned}
$$

Par l'utilisation de ces règles, on peut appliquer des transformations à un programme.

1.1.3 Pei et HPF

Les exemples précédents montrent bien que Pei se base sur le parallélisme de données [85], c'est-à-dire un modèle de programmation centré sur les données et dans lequel le parallélisme est obtenu par distribution des données. Cette approche *data-parallel* de Pei est mise en avant notamment avec la notion d'alignement : avant de faire un calcul, on aligne les données qui interagissent au cours de ce calcul. C'est une approche proche de celle de High Performance Fortran [57, 51], qui propose de paralléliser des programmes via des directives de compilation. Ces directives permettent entre autres d'aligner des tableaux de données par rapport à d'autres tableaux via la directive `ALIGN`. Il est donc assez naturel de vouloir traduire les programmes Pei en HPF [40].

Pour ce faire, il est tout d'abord nécessaire de mettre le programme sous forme traduisible. Des champs de données supplémentaires peuvent en effet être introduits pour permettre à une expression Pei d'être traduite. Dans l'exemple du produit matrice-vecteur, l'équation

$$\mathtt{P} = \mathtt{add} \rhd (\mathtt{A/\&/}(\mathtt{B} \lhd \mathtt{spread})) \tag{1.8}$$

doit être découpée en :

$$
\begin{aligned}
\mathtt{D} &= \mathtt{B} \lhd \mathtt{spread} && (1.9)\\
\mathtt{P} &= \mathtt{add} \rhd (\mathtt{A/\&/D}). && (1.10)
\end{aligned}
$$

Un vérificateur de type (*type-checker*) vérifie qu'il n'y a pas de définitions contradictoires dans le programme et retourne le domaine de chaque champ de données, ce qui permet la déclaration des variables correspondantes. Chaque équation est ensuite traduite. Un `template` (au sens de HPF) défini de façon à ce qu'on puisse y aligner toutes les variables est tout d'abord mis en place, suivi par l'initialisation des variables. Les changements de base sont traduits sous la forme de directives HPF `align` sur le template défini au départ. Les opérations fonctionnelles sont traduites par des instructions HPF `forall` qui expriment le parallélisme de données. De même pour les opérations géométriques. La traduction des équations récurrentes se fait en 2 temps : initialisation des variables qui collectent le résultat final, puis calcul.

Le code HPF résultant du programme Pei de la figure 1.6 se trouve en figure 1.8.

```
      program MatVec
      real A(1:n,1:n), V(1:n), B(1:n,1:n)
      real D(1:n,1:n), P(1:n,1:n), S(1:n,1:n)
      real X(1:n)+
!hpf$ template T(1:n,0:n)
!hpf$ align A,B,P,S with T
!hpf$ align V(j) with T(1,j)
!hpf$ align X(i) with T(i,n)
      forall (j=1:n) B(1,j)=V(j)
      forall (i=1,n, j=1:n) D(i,j)=B(1,j)
      forall (i=1:n, j=1:n) P(i,j)=A(i,j)*D(i,j)
      forall (i=1:n) S(i,0)=0
      forall (i=1:n)
        do j=1,n S(i,j)=S(i,j-1)+P(i,j)
      endforall
      forall (j=1:n) X(j)=S(j,n)
```

Fig. 1.8 – Traduction du programme de multiplication matrice-vecteur Pei en HPF

1.2 Introduction à OPERA

Une importante part des codes de calculs scientifiques se passe dans l'exécution de boucles. Lors de la parallélisation de tels codes, il est intéressant de se pencher plus particulièrement sur la parallélisation de ces boucles. Il existe deux approches : on peut tout d'abord chercher à modifier le nid de boucles pour faire apparaître du parallélisme tout en vérifiant qu'on respecte les contraintes du programmes. De telles approches sont décrites dans [108, 8, 56]. Les transformations appliquées aux boucles peuvent être décrites sous une forme matricielle qui permet de transformer le programme correspondant.

Une autre approche consiste à représenter le nid de boucles de départ sous la forme d'un systèmes d'équations affines paramétrés [91], ce qui permet de modéliser le problème sous une forme géométrique (modèle polyèdrique [65]) et de faire l'étude des dépendances pour aboutir à la parallélisation du nid de boucles. L'outil OPERA a été développé dans le but de proposer une parallélisation des systèmes d'équations affines paramétrées.

1.2.1 Systèmes d'équations affines paramétrées

Prenons le nid de boucles suivant :

```
for i=1, n1
  for j=1, n2
    a[i,j] = b[i+3j+2] + 4
```

On peut écrire $a = f(\ b(\ g(i,j,n_1,n_2)\ )\ )$, où $f(x) = x+4$ et $g(i,j,m,n) = i+3j+2$. Ceci nous définit en fait une équation récurrente affine paramétrée (PARE), qui est de façon plus générale une équation de la forme

$$X[z] = f(\dots, Y[g(z,m)], \dots) \quad z \in D_E(p)$$

avec X et Y des variables, p un vecteur de $\mathbb{Z}^m$ représentant les m paramètres : dans notre exemple, $m = 2$, et on a $p = (n_1, n_2)$. z est le vecteur des indices : ici $z = (i,j)$. $D_E(p)$ est un polyèdre convexe, qui décrit le domaine d'itération. Il est défini par des contraintes, dépendant des bornes des boucles, pouvant s'écrire sous une forme matricielle :

$$D_E(p) = \{z \in \mathbb{Z}^n | Az \leq Bp + b\}.$$

Dans notre exemple les indices i et j vérifient $1 \leq i \leq n_1$ et $1 \leq j \leq n_2$ ce qui peut se traduire sous la forme

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} (i,j) \leq \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} (n1, n2) + \begin{pmatrix} 0 \\ -1 \\ 0 \\ -1 \end{pmatrix}.$$

Enfin, $g(z,p)$ est une fonction affine du type $g(z,p) = Qz + Rp + c$ où, comme pour A et B, R et Q sont des matrices entières et c est un vecteur entier. Dans l'exemple précédent, on a

$$g(z,p) = (1,3)(i,j) + (0,0)(n_1,n_2) + 2.$$

A partir de là, on peut définir ce qu'est un système d'équations récurrentes affines paramétrées. Il est constitué d'un vecteur de paramètres p, avec son domaine de validité D_p inclus dans $\mathbb{Z}^m$, un ensemble de variables d'entrées et sorties (avec leur domaines de définition), un ensemble de variables intermédiaires, utilisées dans les équations récurrentes et finalement, un ensemble d'équations récurrentes affines paramétrées. On peut décrire de nombreux programmes d'algèbre linéaire sous cette forme (produit matrice vecteur, élimination de Gauss, etc.). Les PARE sont équivalents à des nids de boucles mis en assignation unique, c'est-à-dire à l'intérieur desquels les variables ne sont affectées qu'une seule et unique fois, et ayant des propriétés particulières : indices de boucles entiers et s'incrémentant de 1, bornes des boucles étant des expressions affines des paramètres ou des indices de boucles extérieures, idem pour les indices des tableaux, etc.

1.2.2 Modélisation des dépendances

La transformation de nids de boucles visant à leur parallélisation implique que l'on tienne compte des dépendances qui apparaisent. En effet, lors de la parallélisation, il faut respecter la sémantique du programme. Prenons le morceau de programme suivant :

```
S1  x = ...
S2  y = x ...
```

Il est clair qu'on ne peut pas effectuer les instructions S1 et S2 en parallèle, car le calcul de `y` dépend de `x`. La prise en compte des dépendances est donc essentielle lors de la parallélisation. Il existe plusieurs types de dépendances : anti-dépendances, dépendances d'entrée, de sortie et de flot [108], mais certaines astuces [25] comme la mise en assignation unique pour les nids de boucles par exemple permettent de ne considérer au final que les dépendances de flots, du type de celle donnée en exemple. On fait généralement appel aux graphes pour modéliser les dépendances, les nœuds étant les variables, les arêtes les dépendances entre les variables. On peut ne représenter que les variables (graphe réduit) ou au contraire tous les points de calcul (graphe complet). Nous allons nous intéresser à la modélisation des dépendances par les *ensembles d'utilisation* et les *ensembles d'émission* [81], qui, lorsqu'on considère des dépendances affines comme c'est le cas ici, ont des propriétés géométriques intéressantes.

Ensemble d'utilisation : l'ensemble d'utilisation associé à une variable $Y[z_0]$ dans une équation E du type $X[z] = f(\ldots, Y[g(z,m)], \ldots)$ pour la dépendance particulière donnée par g est défini par :

$$Ut_{E,Y}(z_0) = \{z \in D_E(p) | g(z,p) = z_0\}.$$

C'est l'ensemble des points qui ont besoin de $Y[z_0]$ pour être calculés (pour la dépendance induite par la fonction g).

Ensemble d'émission : l'ensemble d'émission associé à une variable Y dans une équation E pour une dépendance particulière est défini par :

$$\begin{aligned} Em_{E,Y}(p) &= \{z_0 \in \mathbb{Z}^n | \exists z \in D_E(p) \text{ tel que } z_0 = g(z,p)\} \\ &= g(D_E(p), p) \end{aligned}$$

Cet ensemble correspond à tous les points de Y qui sont nécessaires à un calcul dans une équation donnée.

Lorsque tous les espaces ont la même dimension, on peut parler de vecteurs de dépendances, qui relient un point de l'ensemble d'émission à un sommet de l'ensemble d'utilisation correspondant.

1.2.3 Cadencement, allocation

La parallélisation d'un nid de boucles nécessite qu'on définisse une allocation et un cadencement. L'allocation permet d'associer un processeur à chaque instruction ou donnée, alors que le cadencement permet d'associer une date d'exécution à une instruction, ou à la mise à jour d'une donnée. En se plaçant dans le cadre du parallélisme de données et pour respecter les contraintes induites par les dépendances, on doit avoir :

$$\text{si le calcul de } X[x] \text{ dépend de } Y[y] \text{ alors } t(X[x]) > t(Y[y]) \tag{1.11}$$

où t est le cadencement. On peut définir diverses sortes de cadencement [54, 34, 28], comme le cadencement affine paramétré $t_X(z,p) = \lambda_X x + \lambda'_X p + a_X$, ou encore les cadencements multidimensionnels qui renvoie un temps à plusieurs dimensions. Pour trouver un cadencement, il suffit d'écrire l'ensemble des inégalités qui découlent de la condition (1.11) et de trouver une solution qui respectent les inégalités :
- soit par la méthode Farkas [98] qui consiste à supprimer les variables z et p via l'application du lemme de Farkas permettant de linéariser les conditions, pour obtenir un nouveau système d'inéquations qu'on peut alors résoudre par la méthode du simplexe ;
- soit par la méthode des sommets [91] qui consiste à s'affranchir des paramètres intervenant dans les inéquations en considérant leurs valeurs extrémales.

On peut définir des surfaces de temps, rassemblant les points qui s'éxécutent à une même date. Ces surfaces sont orthogonales aux vecteurs lignes de la matrice λ_X. Dans le cas d'un cadencement mono-dimensionnel, les différents points ayant les mêmes dates d'exécution peuvent former des hyper-plans (voir [62] à la base de cette approche).

Dans le cas d'un cadencement mono-dimensionnel, on peut définir l'allocation à partir d'un vecteur dans la direction duquel on va projeter les points d'itération. Les processeurs virtuels sont le résultat de cette projection, et les communications entre processeurs résultent des projections des vecteurs de dépendances. Il est donc intéressant de projeter dans la direction d'un vecteur de dépendance : aucune communication n'est générée pour la dépendance en question. Remarquons que bien souvent, cadencement optimal et allocation optimale (en terme de communications par exemple) sont inconciliables. Les concepts d'allocation et de cadencement sont illustrés par la figure 1.9

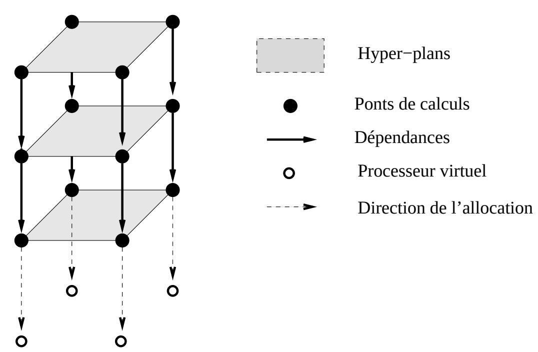


FIG. 1.9 – Allocation et cadencement

1.2.4 Et OPERA ?

OPERA est un lociciel qui a été développé à l'ICPS par V. Loechner durant sa thèse [74, 73]. Le logiciel permet de visualiser les points de calculs intervenant dans un nid de boucles, les vecteurs de dépendances, les vecteurs extrémaux, les ensembles d'utilisation et d'émission. Il est possible de définir un cadencement et/ou une allocation : on visualise alors les hyperplans des points s'exécutant à la même date. D'autre part, le code correspondant à l'allocation et au cadencement donné peut être généré, suivant la méthode détaillée dans [93], où la donnée d'un vecteur de cadencement (perpendiculaire aux hyperplans dans le cas monodimensionnel) et d'un vecteur d'allocation se traduit par la construction d'une matrice de transformation qui permet de calculer les bornes des nouvelles boucles, et ainsi le nouveau parcours des points de calculs correspondant à une exécution parallèle.

Les programmes d'entrée d'OPERA sont des programmes ALPHA [77], donc mis en assignation unique. Il est d'autre part nécessaire que les variables aient toutes la même dimension, quitte à ajouter une dimension a priori inutile. La visualisation du programme et des divers ensembles nécessite de donner une valeur aux paramètres, mais ce n'est utile que pour la visualisation, tout cela via une interface graphique, voir la figure 1.11 qui montre le programme de l'algorithme de Gauss pour la résolution de système linéaire. Le programme en entrée est donnée par la figure 1.10

```
system GaussianElimination : { N | N>=1 }
                ( m : { i,j | 1<=i<=N; 1<=j<=N+1 } of integer )
       returns ( g : { i,j | 1<=i<=N; i<=j<=N+1 } of integer );

var
   A  : { i,j,k | k+1<=i<=N; k+1<=j<=N+1; 0<=k<=N-1 } of integer;

let
   A[i,j,k]  = case
               { | k=0 } : m[i,j];
               { | 1<=k<=N-1 } : A[i,j,k-1] -
                        A[i,k,k-1] * A[k,j,k-1] / A[k,k,k-1] ;
               esac;
   g[i,j] = A[i,j,i-1];
tel;
```

FIG. 1.10 – Programme ALPHA de la résolution de système linéaire par l'algorithme de Gauss

Pour la visualisation, on a choisi de prendre $N = 4$. On peut voir qu'on a donné un cadencement (Schedule Vector). Les surfaces de temps - points qui s'exécutent aux mêmes instants - sont dessinées et on a quelques indications comme la latence du programme - le temps d'exécution du programme, puisqu'il s'agit de la durée qui sépare le premier point calculé du dernier - qui vaut 11 et le parallélisme potentiel - nombre de processeurs actifs au maximum - qui vaut 6. Ces valeurs sont calculées gâce à la bibliothèque de manipulation des polyèdres Polylib [90].

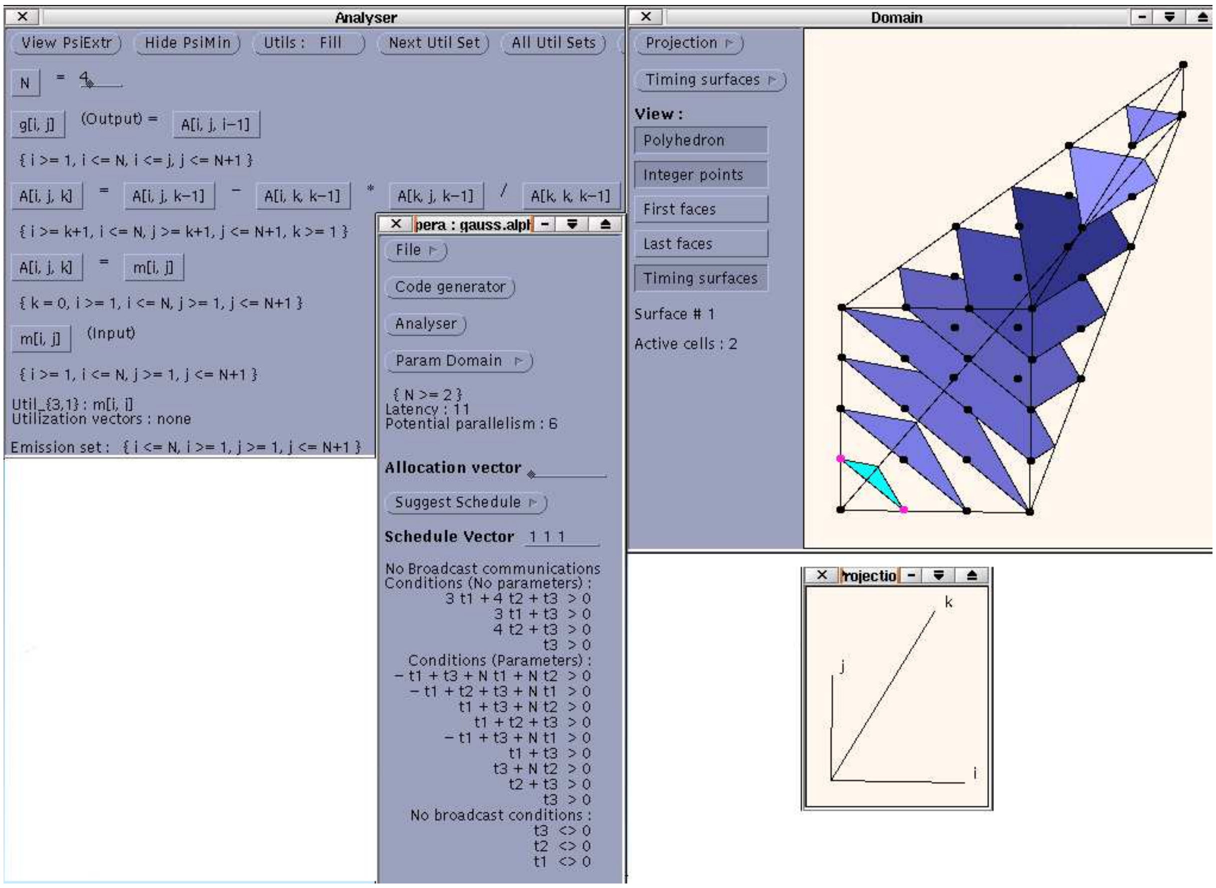


Fig. 1.11 – Copie d'écran d'OPERA

Chapitre 2

PEI et le calcul creux

L'objectif de cette partie est de montrer comment PEI, que nous avons présenté précédemment (voir partie 1), permet de faciliter l'écriture de programme faisant intervenir des calculs creux. Après une présentation générale de ce que sont les matrices creuses, des problèmes particuliers que leur utilisation pose et des algorithmes spécifiques qu'il faut mettre en œuvre, on montrera comment on a utilisé PEI pour faire face à deux problèmes : passage d'un programme plein à un programme creux pour faire face au problème du stockage, et parallélisation de ces programmes.

2.1 Introduction au calcul creux

On appelle matrice creuse une matrice qui a une large proportion de ses éléments égaux à 0. Les matrices tridiagonales en sont un exemple :

$$\begin{pmatrix} a & b & & 0 \\ c & \ddots & \ddots & \\ & \ddots & \ddots & b \\ 0 & & c & a \end{pmatrix}$$

Les matrices creuses interviennent généralement lors de la résolution des équations aux dérivées partielles. Considérons l'équation

$$\begin{gathered} -u''(x) = f(x) \text{ sur } [0,1] \\ u(0) = u(1) = a \end{gathered}$$

Par définition, on a

$$u'(x) = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h}$$

Si on discrétise le domaine $[0,1]$ en N morceaux de taille $\frac{1}{N}$ et on va chercher les solutions u sur les $N+1$ points $x_n = n\frac{1}{N},\ n \in \mathbb{N},\ 0 \leq n \leq N$. On approxime :

$$u'(x) = \frac{u(x-h) - u(x+h)}{2h}$$

Prenons $h = \frac{1}{2N}$ et posons $u_i = u(x_i)$ le problème devient alors :

$$\begin{gathered} u_0 = a \\ -u_{i+1} + 2u_i - u_{i-1} = f_i \\ u_{N-1} = a \end{gathered}$$

ce qui peut se réécrire sous forme matricielle :

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ u_i \\ \vdots \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} f_1 + u_0 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} + u_n \end{pmatrix}$$

Des domaines beaucoup plus complexes, en 2 ou trois dimensions avec des maillages discrets qui peuvent être irréguliers, conduisent à des matrices creuses plus irrégulières, n'ayant pas forcément une forme particulière.

Les calculs faisant intervenir des matrices creuses posent des problèmes spécifiques : tout d'abord, il est nécessaire d'adopter un stockage particulier pour ces matrices afin de tirer profit de leur structure. Il est d'autre part nécessaire de revoir certains algorithmes afin de limiter le remplissage qu'ils induisent, afin, encore, de tenir compte du caractère creux de ces matrices ; toujours pour ces raisons, il peut être utile d'utiliser d'autres algorithmes, mieux adaptés. Enfin, la parallélisation des calculs creux pose des problèmes spécifiques car les techniques applicables dans les cas pleins ne sont plus utilisables telles quelles.

Ces divers problèmes sont abordés dans la suite : tout d'abord les problèmes du remplissage et des algorithmes utilisés pour les matrices creuses sont présentés afin de fixer certains points. Ensuite, les sections 2.2 et 2.3 montrent comment PEI peut être utile pour aborder et résoudre les problèmes particuliers liés au stockage et à la parallélisation.

2.1.1 Le remplissage

Supposons que l'on veuille résoudre un système linéaire impliquant une matrice creuse A par la méthode bien connue du pivot de Gauss [63].

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

Après une étape d'élimination, on obtient la matrice

$$A' = \begin{pmatrix} 1 & 2 & -3 & 0 \\ 0 & 1 & 0 & -3 \\ 0 & \boxed{6} & -8 & 0 \\ 0 & -3 & 0 & 1 \end{pmatrix}$$

On voit que l'élément a_{32} encadré est devenu non-nul. Ceci pose deux problèmes. Tout d'abord au niveau du stockage : comme on le verra dans la suite, on ne stocke que les valeurs non-nulles des matrices (donc on n'utilise pas de tableaux de taille $N \times N$ entre autres). Il faut prévoir le remplissage ou avoir des structures de données dynamiques (mais d'accès plus complexes que de simples tableaux...) D'autre part, sur de très grosses matrices, le remplissage induit plus de calculs. On perd alors une bonne partie du gain qu'on a avec une matrice creuse en terme de calculs moins nombreux : une valeur de plus induit une donnée de plus entrant dans un calcul.

Il peut alors être utile de prévoir et/ou réduire le remplissage pour pallier ces deux inconvénients.

Une façon de prévoir le remplissage est d'utiliser le profil (voir définition ci-après) de la matrice, pour les matrices symétriques. En effet, le remplissage de la factorisation de Cholesky (ou LU d'ailleurs) reste cantonné à certaines zones, en particulier ne sort pas du profil.

Définition 1 (Profil)
Soit $A = (a_{ij})_{1 \le i \le n, 1 \le j \le n}$ une matrice symétrique. On note j_i l'indice de la colonne du premier élément non-nul de la ligne i, soit

$$j_i = \min_{1 \le j \le i} (a_{ij} \neq 0)$$

Le profil P_A de A est défini par :

$$P_A = \{(i, j); 1 \le i \le n, j_i \le j \le i\}$$

Ci-dessous, un exemple, où les "X" représente les éléments non-nuls :

$$\begin{array}{cccccc} X & & & & & \\ & X & & & & \\ X & & X & & & \\ & X & & X & & \\ X & X & & & X & \\ & & & X & X & X \end{array} \qquad \qquad \begin{array}{cccccc} X & & & & & \\ & X & & & & \\ X & X & X & & & \\ & X & X & X & & \\ X & X & X & X & X & \\ & & & X & X & X \end{array}$$

Matrice creuse — Profil de la matrice creuse

Une autre façon de résoudre le problème est de calculer à l'avance le remplissage qui va intervenir lors d'une factorisation. Pour ce faire, on représente une matrice $A(n, n)$ par un graphe : $G(V, E)$ où V est l'ensemble des sommets et E l'ensemble des arêtes :

$$V = \{v_1, \ldots, v_n\}$$
$$(v_i, v_j) \in E \text{ si et seulement si } a_{ij} \neq 0$$

En particulier, si A est symétrique on a :

$$(v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E$$

Dans ce cas, G est un graphe non-orienté.

L'élimination de Gauss peut alors être interprétée comme la construction d'une série de graphes dits *d'élimination* [69]. La construction est la suivante :
- on part de $G_0 = G$
- on parcourt une boucle k de 0 à $n-1$ au cours de laquelle on va construire un graphe G_{k+1} à partir de G_k :
  - on obtient V_{k+1} en supprimamt v_k de V_k
  - on obtient E_{k+1} en prenant E_k duquel on supprime toutes les arêtes incidentes à v_k et en lui ajoutant les arrêtes $(v.w)$ telles que $(v, v_k) \in E_k$ et $(v_k, w) \in E_k$.

Durant cette opération, on construit un graphe avec les sommets V et les arêtes E auxquelles on ajoute toutes les arêtes créées lors de la construction des ensembles successifs E_k, et on obtient le graphe de la matrice avec le remplissage induit par la factorisation. En effet, les arêtes (v, w) telles que $(v, v_k) \in E_k$ et $(v_k, w) \in E_k$ correspondent aux valeurs formellement non-nulles qui induisent un remplissage.

Enfin, diverses stratégies visant à réduire le remplissage existent. L'intérêt d'une telle opération est de diminuer le coût en mémoire du stockage de la matrice mais aussi de diminuer le nombre de calculs, ce qui du même coup réduit l'impact des erreurs numériques. Néanmoins, le calcul du stockage minimal est un problème NP-complet [109], donc la réduction du stockage se fait par diverses heuristiques.

Stratégies locales de réduction du remplissage

Il s'agit, à chaque étape, de choisir le pivot qui va minimiser le stockage. On peut parler tout d'abord de la stratégie de Markowitz [76]. On note $l_i^{(k)}$ le nombre d'éléments non-nuls qui apparaissent dans la ligne i à l'étape k et $c_i^{(k)}$ le nombre d'éléments non-nuls qui apparaissent dans la colonne i à l'étape k. On définit alors une fonction de coût de Markowitz qui vaut $r_i^{(k)} = (l_i^{(k)} - 1).(c_i^{(k)} - 1)$, ce qui pour chaque élément a_{ij} nous donne le nombre d'éléments non-nuls dans la matrice mise à jour. On choisit alors le pivot qui a un coût moindre, à condition que cela n'induise pas un pivot nul, bien entendu.

On a aussi la méthode du *degré minimal* (ou Minimum Degree Ordering [42, 3]). On l'utilise pour les matrices symétriques, mais on peut l'étendre aux non-symétriques en l'appliquant à $A + A^T$. On cherche le pivot $a_{kk}^{(k)}$ tel que $r_k^{(k)} \leq max_{k \leq i \leq n} r_i^{(k)}$, en modélisant la matrice sous forme de graphe ce qui induit un algorithme spécifique. Cette stratégie connaît des variantes.

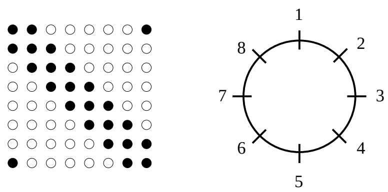
Stratégies globales de réduction du remplissage

Il s'agit là de réordonner la matrice a priori pour que le remplissage reste cantonné à certaines zones.
Il existe l'*algorithme de Cuthill-Mc Kee* [71]. L'idée est d'essayer de mettre la matrice sous forme tri-diagonale. On va partitionner les sommets en niveaux, ce qui permettra ensuite

de les renumeroter et donc d'obtenir une matrice ré-ordonnée. On choisit pour cela un sommet de départ, qui va être notre premier niveau.

$$S_0 = \{v_k\}$$

On construit ensuite S_i à partir de S_{i-1} en prenant les voisins de tous les nœuds de S_{i-1} qui n'appartiennent encore à aucun niveau. Ensuite, on renumérote les nœuds niveau par niveau. La figure 2.1 montre un exemple de matrice et de son réordonnancement, avec la matrice initiale et son graphe, puis le partitionnement et enfin la matrice réordonnée et son graphe.



$$S_0 = \{1\}$$
$$S_1 = \{2, 8\}$$
$$S_2 = \{3, 7\}$$
$$S_3 = \{4, 6\}$$
$$S_5 = \{5\}$$

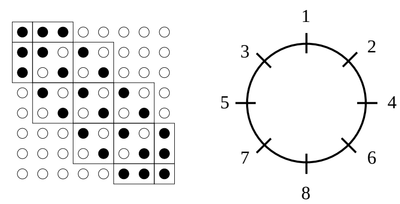


FIG. 2.1 – Exemple de matrice avec son graphe et son réordonnancement après application de l'algorithme de Cuthill-Mc Kee

On peut aussi utiliser l'algorithme des *Dissections emboîtées* (Nested dissections [68, 42, 50]). Ici, on choisit des points qui vont séparer le graphe en 2, puis on va séparer les 2 sous-graphes de la même façon, récursivement jusqu'à avoir des points séparés. Ceci permet de contruire des niveaux, il n'y a ensuite plus qu'à renuméroter les points, en renumérotant tout d'abord les points séparés, puis en numérotant les points dans les séparateurs, dans l'ordre inverse de calcul de ces séparateurs. La figure 2.2 montre un exemple de graphe de matrice ainsi que la matrice obtenue après application de l'algorithme.

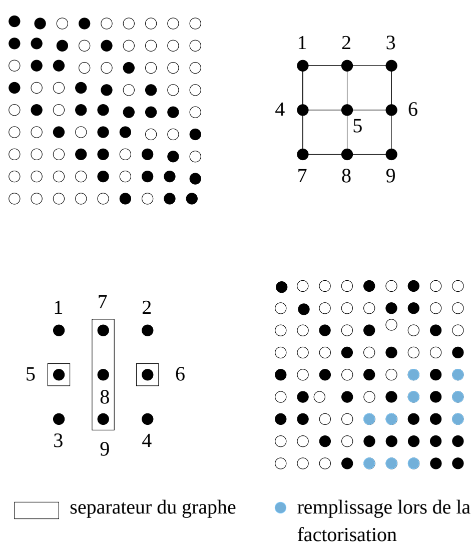


FIG. 2.2 – Exemple d'application de l'algorithme des dissections emboîtées

2.1.2 Résolution des systèmes creux

Le problème du remplissage évoqué précédemment est intimement lié à l'utilisation des méthodes directes de résolution de systèmes linéaires (Gauss, factorisation LU, factorisation de Cholesky,...). La mise en œuvre des méthodes directes se fait via le biais des méthodes *multifrontales*. On trouve dans [70] une bonne approche agrémentée d'un exemple. On peut aussi se réferrer à [31]. Ces méthodes consistent à découvrir dans la matrice creuse des ensembles de colonnes sur lesquelles il est possible de travailler indépendamment sans se soucier des 0, comme sur une matrice pleine, et de revoir les diverses étapes de la factorisation LU ou de Cholesky en fonction de cela.

Cependant, ces méthodes ne sont peut-être pas les plus adaptées aux matrices creuses, malgré leur robustesse : leur mise en œuvre demande un travail conséquent : réduction du remplissage, adaptation des méthodes de résolution. Il existe d'autres techniques, robustes elles-aussi, permettant de mieux prendre en compte le caractère creux des matrices.

Ainsi, on peut envisager d'utiliser d'autres méthodes que les méthodes directes pour résoudre le systèmes linéaires creux. Les matrices creuses sont plus adaptées à la mise en œuvre efficace de calculs comme les produits matrices-vecteurs. En effet, ces calculs n'impliquent aucun remplissage et ne nécessitent donc pas de réordonnancement de la matrice. Les méthodes indirectes, qui cherchent la solution d'un système linéaire par itérés successifs, se prêtent justement bien à ce type de calculs.

Les méthodes indirectes consistent à chercher la solution du système linéaire $Ax = b$ par des itérés successifs : $x_{k+1} = \ldots x_k \ldots$. Les méthodes de Jacobi ou Gauss-Seidel, avec ou sans relaxation en font partie. On pourra se réferrer à [64] pour une présentation de ces méthodes. On va plutôt ici se pencher sur les méthodes de Krylov, qui sont plus robustes et comparables dans leur performance aux méthodes directes.

Introduisons quelques notations. On note $r_k = Ax_k - b$ et $K_k = Vect\{r_0, Ar_0, \ldots, A^{k-1}r_0\}$. r_k est le résidu et K_k est appelé espace de Krylov. L'idée ici pour résoudre les systèmes linéaires est de minimiser le résidu r_k à chaque étape tout en cherchant x_{k+1} sous la forme $x_{k+1} = x_k + d_k$ avec $d_k \in K_{k+1}$.

On peut tout d'abord parler du *gradient conjugué*. Cette technique s'applique pour les matrices symétriques définies positives (abrégé par la suite en SPD). On trouve dans [64] une présentation détaillée. On cherche $d_k = \alpha_k p_k$, c'est-à-dire qu'on va d'abord choisir la direction, puis la taille du vecteur, de façon à minimiser le résidu. On obtient l'algorithme de la figure 2.3.

Cet algorithme minimise le résidu sur l'espace K_k c'est-à-dire qu'on a $r_k \perp K_k$ par rapport au produit scalaire induit par la matrice A : $(A.,.)$. On note qu'après n itérations, l'algorithme rend théoriquement la valeur exacte de la solution, puisqu'on a $r_k = 0$. Néanmoins, cet algorithme est utilisé comme un algorithme itératif, avec un test de convergence pour savoir quand s'arrêter.

L'algorithme précédent est un peu limité dans la mesure où il n'est adapté qu'aux matrices SPD. Divers autres algorithmes ont été développés pour l'améliorer. Tout d'abord, l'algorithme du *bigradient* [38] construit le résidu r_k tel que $r_k \perp L_k$ avec

Initialisation
choix de x_0
$p_0 = r_0 = b - Ax_0$
Pour $k = 0, 1 \ldots$
$\quad \alpha_k = \frac{\|r_k\|^2}{(Ap_k, p_k)}$
$\quad x_{k+1} = x_k + \alpha_k p_k$
$\quad r_{k+1} = r_k - \alpha_k A p_k$
$\quad \beta_{k+1} = \frac{\|r_{k+1}\|^2}{\|r_k\|^2}$
$\quad p_{k+1} = r_{k+1} + \beta_{k+1} p_k$

FIG. 2.3 – Algorithme du gradient conjugué

$L_k = Vect\{r_0, A^T r_0, \ldots, (A^T)^{k-1}\}$. Cet algorithme a ensuite été modifié pour accélérer sa convergence : on obtient alors l'algorithme CGS (ou *Conjugate Gradient Squared* [101]). Une écriture sous forme polynomiale des vecteurs sous la forme $r_k = P_k(A) r_0$ permet de réécrire les produits scalaires et de supprimer des étapes, accélérant le processus. Enfin, Bi-CGStab (ou *bigradient stabilisé* [106]) est une amélioration de CGS pour lequel on ajoute une minimisation dans le calcul des polynômes : on va chercher $r_k = P_k(A).Q_k(A) r_0$ où Q_k est de la forme $Q_k = (1 - \omega_1 x) \ldots (1 - \omega_k x)$ avec les ω qui minimisent le résidu. On obtient finalement l'algorithme de la figure 2.4.

Les algorithmes précédents induisent une récurrence courte sur le calcul de x_k lorsqu'on les développe : $x_k = f(x_{k-1}, x_{k-2})$. On peut essayer d'avoir une récurrence beaucoup plus longue, en cherchant $r_k \perp L_k$ où cette fois-ci $L_k = Vect\{Ar_0, A^2 r_0, \ldots, A^k r_0\}$. Pour cela, on cherche à construire une base orthonormée de K_k qui nous permettra de facilement calculer d_k. On construit cette base orthonormée par l'algorithme d'Arnoldi [6] :

Algorithme d'Arnoldi
Pour $j = 1, \ldots, k$
$\quad \hat{v}_{j+1} = A v_j$
$\quad$ Pour $i = 1, \ldots, j$
$\qquad \beta_{i,j} = (\hat{v}_{j+1}, v_i)$
$\quad \hat{v}_{j+1} = \hat{v}_{j+1} - \sum_{i=1}^{j} \beta_{i,j} v_i$
$\quad \beta_{j+1,j} = \|\hat{v}_{j+1}\|$
$\quad v_{j+1} = \frac{\hat{v}_{j+1}}{\beta_{j+1,j}}$

Ainsi, au bout de k itérations, on a construit un système orthonormal $(v_j)_{j=1,\ldots,k}$, une matrice $k \times k$, la matrice des $(\beta_{i,j})_{(i=1,\ldots,k\ ,j=1,\ldots,k)}$ complétée de 0, notée H_k et une matrice $((k+1) \times k)$ notée $\bar{H}_k$ qui est la matrice H_k à laquelle on a ajouté une ligne de 0 avec au bout $\beta_{k+1,k}$. Notons V_k la matrice des (v_j). On a alors :

$$H_k = V_k^T A V_k$$
$$A V_k = V_{k+1} \bar{H}_k$$

Comme V_k est un changement de base dans K_k, on peut écrire que pour tout z de K_K, il existe un y tel que $z = V_k y$; y est alors le vecteur z exprimé dans la base des v_i.

Initialisation :
on choisit x_0
$r_0 = b - Ax_0$
$\bar{r}_0$ est choisi au hasard
de telle sorte que $(r_0, \bar{r_0}) \neq 0$
on peut prendre $\bar{r}_0 = r_0$
$\rho_0 = \alpha = \omega_0 = 1$
$v_0 = p_0 = 0$
Pour $k = 1, 2, 3, \ldots$
$\rho_k = (\bar{r}_0, r_{k-1})$
$\beta = \frac{\rho_k}{\rho_{k-1}} \frac{\alpha}{\omega_{k-1}}$
$p_k = r_{k-1} + \beta(p_{k-1} - \omega_{k-1} v_{k-1})$
$v_k = Ap_k$
$\alpha = \frac{\rho_k}{(\bar{r}_0, v_k)}$
$s = r_{k-1} - \alpha v_k$
$t = As$
$\omega_k = \frac{(t,s)}{(t,t)}$
$x_k = x_{k-1} + \alpha p_k + \omega_k s$
Test d'arrêt
$r_k = s - \omega_k t$

FIG. 2.4 – Algorithme BiCGStab

Le problème de minimisation du résidu $\min_{z \in K_k} \|b - A(x_0 + d)\|$ devient alors un problème de minimisation de J définit par :

$$J(y) = \| \, \|r_0\| e_1 - \bar{H}_k y \|$$

où e_1 est le premier vecteur de la base euclidienne. On cherche y_k de telle sorte que r_k soit orthogonal à K_k (on cherche à minimiser le résidu sur K_k). On obtient alors :

$$y_k = H^{-1} \|r_0\| e_1$$

Cette résolution se fait par une décomposition QR de H, qui de par sa forme presque triangulaire supérieure s'y prête bien. L'algorithme converge, mais la construction de la base d'Arnoldi lorsque k est grand est très coûteuse. C'est pourquoi on utilise un algorithme un peu différent, GMRes(m), qui consiste à calculer la base d'Arnoldi pour m vecteurs, à calculer la solution x_m qui s'y rapporte et en cas de non-convergence, de prendre ce nouveau x_m comme point de départ x_0. Cet algorithme est décrit à la figure 2.5.

Initialisation :
$r_0 = b - Ax_0$
$v_1 = \frac{r_0}{\|r_0\|}$
Algorithme d'Arnoldi
Pour $j = 1, \ldots, m$
  $\hat{v}_{j+1} = Av_j$
  Pour $i = 1, \ldots, j$
    $\beta_{i,j} = (\hat{v}_{j+1}, v_i)$
  $\hat{v}_{j+1} = \hat{v}_{j+1} - \sum_{i=1}^{j} \beta_{i,j} v_i$
  $\beta_{j+1,j} = \|\hat{v}_{j+1}\|$
  $v_{j+1} = \frac{\hat{v}_{j+1}}{\beta_{j+1,j}}$
Construction de la solution approchée :
Minimisation de $\| \, \|r_0\| e_1 - \bar{H}_m y \|$
  cette minimisation se fait par une décomposition QR
$x_m = x_0 + V_m y_m$
Critère de convergence
$r_m = Ax_m - b$
s'il n'y a pas convergence
  $r_0 = r_m$
  $v_1 = \frac{r_0}{\|r_0\|}$
  recommencer à partir de l'algorithme d'Arnoldi

FIG. 2.5 – Algorithme GMRes(m)

Les algorithmes itératifs précédents convergent théoriquement. Ce n'est pas toujours la cas lors de l'implémentation sur ordinateur à cause des problèmes numériques en résultant. Il est donc nécessaire de renforcer la convergence à l'aide de préconditionnements. Cela consiste à changer le système à résoudre $Ax = b$ en

- $K^{-1}Ax = K^{-1}b$ (préconditionnement à gauche)
- ou $AK^{-1}y = b$ et $K^{-1}y = x$ (préconditionnement à droite)
- ou $K_1^{-1}AK_2^{-1}y = K_1^{-1}b$ et $K_2^{-1}y = x$ si K^{-1} s'écrit $K_1^{-1}K_2^{-1}$

K^{-1} est une approximation de l'inverse de A. On peut choisir plusieurs types de préconditionnements :

- *Préconditionnement diagonal*
  On prend $K = D$ diagonale de A. Ce préconditionnement n'est pas très intéressant, car il arrive que le conditionnement de A soit le même que celui de $K^{-1}A$.
- *Préconditionnement SSOR*
  Si A est SPD on peut écrire $A = D + L + L^T$ où D est diagonale, L triangulaire strictement inférieure. On prend

  $$K = (\Delta + L)\Delta^{-1}(\Delta + L^T)$$

  avec Δ matrice diagonale à déterminer. Un cas particulier est

  $$K = \frac{1}{\omega(2-\omega)}(D + \omega L)D^{-1}(D + \omega L^T)$$

- *Préconditionnement LU incomplet*
  Dans le cas d'une matrice A quelconque inversible, on peut la factoriser sous la forme $A = LU$ Malheureusement, si A est creuse, on la remplit, d'où l'idée d'une factorisation incomplète, soit en déterminant à l'avance les emplacements que l'on souhaite remplir, soit en négligeant certains éléments de la décomposition. Pour plus de précisions, on pourra consulter [95] et [96]. On a :

  $$A = LU + R$$

  et on prend

  $$K = LU$$

  Ce préconditionnement nécessite alors la résolution de 2 systèmes triangulaires.
- *Préconditionnement polynomial* [96]
  On approxime A^{-1} par un polynôme en A. Par exemple

  $$A^{-1} \approx I + (I - A) + (I - A)^2 + \cdots + (I - A)^k$$

  qui est un préconditionnemnt de type Neumann. Lorsqu'on dispose d'informations sur les valeurs propres de la matrice, on peut utiliser des préconditionnements de type Tchebysheff.

Si on reprend les algorithmes de résolution présentés (gardient conjugué, Bi-CGSTab, GMRes(m), ...) on voit que les calculs les plus lourds concernent les produits matrices-vecteurs ainsi que l'application des préconditionnements. Les produits matrice-vecteur sont relativement bien adaptés au calcul creux (voir la section 2.2.1 qui donnent des exemples de codes). Restent les préconditionnements, qui sont généralement choisis de façon à ne pas trop bousculer la structure de la matrice.

2.2 Transformation de programme avec Pei

Un des problèmes lors de l'écriture des programmes pour matrices creuses réside dans le fait qu'on ne peut pas réutiliser le code séquentiel existant. En effet, les stockages adaptés à ces matrices nécessitent une réécriture des calculs. On se propose d'utiliser Pei afin de transformer automatiquement les programmes "pleins" en programmes "creux". Ces transformations sont conditionnées par le stockage utilisé pour la matrice.

2.2.1 Stockage des matrices creuses

La différence de traitement entre les matrices creuses et les matrices pleines tient surtout au fait qu'on ne stocke pas les 0 des matrices creuses. Les calculs tels qu'on les effectue pour les matrices pleines doivent alors être revus pour s'adapter au stockage creux. Il existe plusieurs types de stockages creux. Ceci est abordé dans [61] par exemple. Nous en présentons quelques uns.

Le stockage le plus simple est le *Coordinate storage*. Il est constitué de 3 tableaux, qu'on appellera A, IA et JA, stockant respectivement les nnz valeurs non-nulles de la matrice, les lignes, et les colonnes correspondantes. Le code du produit matrice-vecteur utilisant ce stockage est donné par la Fig. 2.6. Ce stockage a le désavantage de n'avoir aucune régularité et engendre de nombreuses indirections.

```
do i=1, nnz
  v[ia[i]] = v[ia[i]]+b[ja[i]]*a[i]
enddo
```

Fig. 2.6 – Produit matrice-vecteur avec le Coordinate storage

On peut aussi citer le stockage *Compress Sparse Row* (CSR) et son analogue *Compress Sparse Column* (CSC). On stocke les éléments non-nuls d'une matrice dans 3 tableaux à une dimension : le premier, A stocke les valeurs non-nulles de la matrice ligne par ligne (resp. colonne par colonne) les unes à la suite des autres. Un deuxième tableau, JA, stocke les numéros de colonnes (resp. lignes) correspondant à chaque valeur non-nulle. Enfin, un troisième tableau, IA, stocke un pointeur sur les débuts de lignes dans les tableaux A et JA. La figure 2.7 donne un exemple. Si on note n la taille de la matrice et nnz le nombre total d'éléments non-nuls, IA est donc de taille n alors que A et JA sont de taille nnz.
Le produit matrice-vecteur utilisant ce stockage est illustré dans la Fig. 2.8.

On remarque que ces stockages sont bien adaptés à la mise en œuvre de calculs comme le produit matrice-vecteur : les données dans les tableaux sont accédées consécutivement, ce qui favorise la localité spatiale. Les algorithmes mettant en jeu des produits matrice-vecteur sont donc particulièrement intéressants quand on manipule des matrices creuses.

Il existe d'autres stockages, pouvant tirer parti de la forme de la matrice, tel le stockage diagonal adapté aux matrices diagonales. Enfin, certains stockages peuvent être adaptés au parallélisme, permettant de passer aisément du stockage ligne au stockage colonne (et

$$\begin{pmatrix} 3 & 0 & 0 & 1 & 0 & 2 \\ 0 & 6 & 0 & 0 & 3 & 0 \\ 4 & 0 & 9 & 0 & 5 & 0 \\ 0 & 6 & 0 & 12 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 \\ 7 & 0 & 0 & 0 & 0 & 18 \end{pmatrix}$$

IA	1			4		6			9		10	11		12
	↓			↓		↓			↓		↓	↓		↓
A	3	1	2	6	3	4	9	5	6	12	15	7	18	*
JA	1	4	6	2	5	1	3	5	2	4	5	1	6	*

Fig. 2.7 – Exemple de matrice et son stockage CSR

```
do i=1,n
  v[i] = 0
  do j=ia[i],ia[i+1]-1
    v[i] = v[i]+b[ja[j]]*a[j]
  enddo
enddo
```

Fig. 2.8 – Produit matrice-vecteur avec le stockage CSR

réciproquement), transformation utile lorsqu'un passage à la transposée est nécessaire par exemple.

2.2.2 Exemple de transformation de programme avec Pei

Pei est outil de transformation de programmes fonctionnant dans un cadre formel. Notre idée est de l'utiliser pour transformer les programmes pleins en programmes creux. Il faut adapter le code pour que le stockage plein devienne un stockage creux, ce qui, nous allons le voir, s'apparente à un changement de base.

On considère l'exemple du produit matrice-vecteur donné lors de l'introduction à Pei au chapitre 1.1.1 (programme MatVec de la figure 1.6) en prenant une matrice composée de trois diagonales. Les éléments non-nuls sont situés aux emplacements (i, j) qui vérifient $i = j + 2$, $i = j$ et $i = j - 2$, comme illustré sur la figure 2.9. Il faut tout d'abord choisir un stockage creux, à partir duquel on pourra définir le changement de base. Pour l'exemple, on peut choisir un stockage bien adapté à ce type de matrice : le stockage diagonal, illustré par la Fig.2.9.

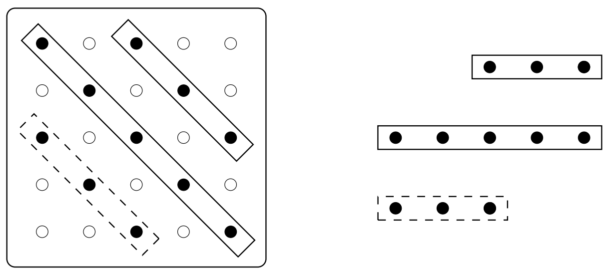


Fig. 2.9 – Matrice tridiagonale et son stockage diagonal

Le changement de base qui correspond à cette transformation est donné par :

$$\begin{aligned}
\texttt{gather} \;=\; & \quad \lambda(i,j)\ |(3{\le}i{\le}n, 1{\le}j{\le}n-2, j{=}i{-}2)\,.\,(1,j)\\
& \#\ \lambda(i,j)\ |(1{\le}i{\le}n, 1{\le}j{\le}n, i{=}j)\,.\,(2,j)\\
& \#\ \lambda(i,j)\ |(1{\le}i{\le}n-2, 3{\le}j{\le}n, i{=}j{+}2)\,.\,(3,j)
\end{aligned}$$

dont l'inverse est

$$\begin{aligned}
\texttt{gather}^{-1} \;=\; & \quad \lambda(i,j)\ |(i{=}1, 3{\le}j{\le}n)\,.\,(j{-}2,j)\\
& \#\ \lambda(i,j)\ |(i{=}2, 1{\le}j{\le}n),\,.\,(j,j)\\
& \#\ \lambda(i,j)\ |(i{=}3, 1{\le}j{\le}n{-}2)\,.\,(j{+}2,j)
\end{aligned}$$

Alignement matrice-vecteur

Pour pouvoir appliquer le changement de base qui passe d'une géométrie à une autre, il est nécessaire que les valeurs non-nulles de la matrice et du vecteur appartiennent au domaine du changement de base. Or, si on aligne le vecteur avec la première ligne de la matrice

comme précédemment, cette condition n'est pas respectée. En effet, des composantes non-nulles du vecteur sont alignées avec des composantes nulles de la matrice qui seront donc éliminées lors du changement de base. On perd des valeurs du vecteur ! Un premier pas de la transformation consiste donc à changer l'alignement du vecteur en choisissant la diagonale principale de la matrice plutôt que la première ligne. Cette façon de procéder est assez générale car les matrices provenant de discrétisations des équations aux dérivées partielles ou des équations différentielles ont rarement des 0 sur la diagonale. Sinon, on est amené à considérer les éléments nuls de la diagonale comme non-nuls logiquement, ce qui n'est pas dommageable.
Ce nouvel alignement se fait en réécrivant la fonction `spread` sous la forme :

$$\texttt{spread} = \texttt{pivot} \circ \texttt{diagonal} \tag{2.1}$$

où

$$\begin{aligned} \texttt{pivot} &= \lambda(i,j) \;|(1 \le i \le n, j = i) \,.\, (1,j) && (2.2)\\ \texttt{diagonal} &= \lambda(i,j) \;|(1 \le i \le n, 1 \le j \le n) \,.\, (j,j). && (2.3) \end{aligned}$$

Ce changement est illustré par la Fig.2.10.

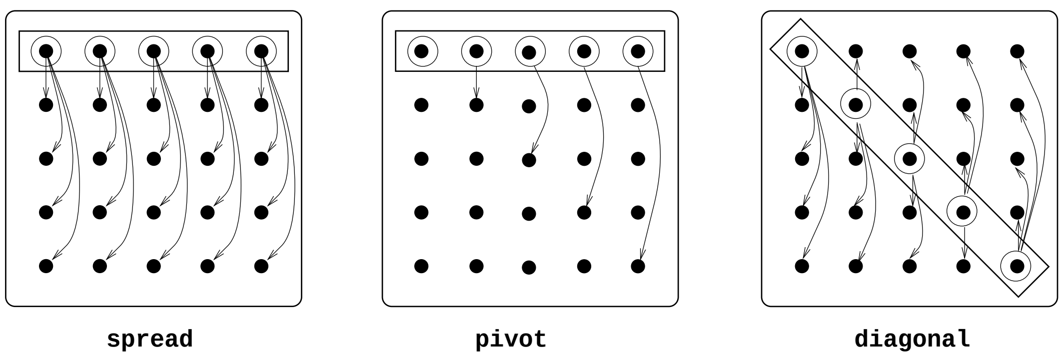


FIG. 2.10 – Fonctions géométriques `spread`, `pivot` et `diagonal`

Application du changement de base

On peut commencer à appliquer le changement de base aux équations. Tout d'abord, on doit l'appliquer au champ `P`, défini par

$$\texttt{P} \;=\; \texttt{add} \rhd (\texttt{A/\&/}(\texttt{B} \lhd \texttt{spread})).$$

On peut lui appliquer `gather`, et en utilisant la règle (1.5) précédemment énoncée on obtient

$$\begin{aligned} \texttt{gather::P} \;=\; \texttt{prod} \rhd (&\texttt{gather::A /\&/} \\ &\texttt{gather::}((\texttt{B} \lhd \texttt{pivot}) \lhd \texttt{diagonal})). \end{aligned} \tag{2.4}$$

Il faut ensuite appliquer la règle (1.6) à $\texttt{gather::((B} \lhd \texttt{pivot)} \lhd \texttt{diagonal)}$ ce qui nous donne

$$\begin{aligned}\texttt{P} \;=\; & \texttt{prod} \rhd (\texttt{gather::A /\&/} \\ & (\texttt{gather::(B} \lhd \texttt{pivot})) \lhd (\texttt{gather} \circ \texttt{diagonal} \circ \texttt{gather}^{-1}).\end{aligned} \tag{2.5}$$

Posons

$$\texttt{P'} = \texttt{gather::P} \tag{2.6}$$
$$\texttt{A'} = \texttt{gather::A} \tag{2.7}$$
$$\texttt{T'} = \texttt{gather::(B} \lhd \texttt{pivot}). \tag{2.8}$$

On peut finalement simplifier en

$$\texttt{P'} = \texttt{prod} \rhd (\texttt{A'/\&/(T'} \lhd \texttt{spread'})) \tag{2.9}$$

où la nouvelle fonction `spread'`, illustrée par la Fig.2.11 est définie par

$$\begin{aligned}\texttt{spread'} \;=\; & \texttt{gather} \circ \texttt{diagonal} \circ \texttt{gather}^{-1} \\ =\; & \quad \lambda(i,j) \;\; |(i{=}1, 3{\leq}j{\leq}n) \,.\, (2,j) \\ & \#\; \lambda(i,j), |(i{=}2, 1{\leq}j{\leq}n), \,.\, (2,j) \\ & \#\; \lambda(i,j) \;\; |(i{=}3, 1{\leq}i{\leq}n{-}2) \,.\, (2,j).\end{aligned} \tag{2.10}$$

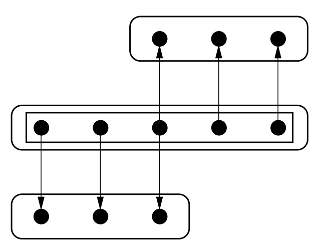


FIG. 2.11 – Nouvelle fonction `spread'`

Il faut ensuite appliquer le changement de base à `S`. On ne peut pas appliquer directement le changement de base, car les domaines de $\texttt{gather}^{-1}$ et `shift` ne se correspondent pas. Il est nécessaire de restreindre le domaine de `S` pour ne garder que les valeurs non-nulles. L'équation précédente ne posait pas ce problème puisqu'on appliquait le changement de base seulement sur les valeurs diagonales où il n'y pas de zéros. La fonction `sparse` donnée ci-dessous décrit la structure non-nulle de la matrice et permet de se restreindre à ces seules valeurs.

$$\begin{aligned}\texttt{sparse} \;=\; & \quad \lambda(i,j) \;\; |(3{\leq}i{\leq}n, j{=}i{-}2) \\ & \#\; \lambda(i,j) \;\; |(1{\leq}i{\leq}n, j{=}i) \\ & \#\; \lambda(i,j) \;\; |(1{\leq}i{\leq}n{-}2, j{=}i{+}2)\end{aligned} \tag{2.11}$$

On peut alors appliquer `sparse` à `S` dans le cas de la matrice tridiagonale :

$$\begin{aligned}
\mathtt{S} \lhd \mathtt{sparse} &= \mathtt{add} \rhd (\mathtt{P} \lhd \mathtt{sparse}\ /\&/ \\
&\qquad (\mathtt{S} \lhd \mathtt{shift} \circ \mathtt{shift}) \lhd \mathtt{sparse}) \\
&= \mathtt{add} \rhd (\mathtt{P} \lhd \mathtt{sparse}\ /\&/\mathtt{S} \lhd (\mathtt{shift} \circ \mathtt{shift} \circ \mathtt{sparse})) \\
&= \mathtt{add} \rhd (\mathtt{P} \lhd \mathtt{sparse}\ \ /\&/ \\
&\qquad \mathtt{S} \lhd (\mathtt{sparse} \circ \mathtt{shift} \circ \mathtt{shift} \circ \mathtt{sparse})) \\
&= \mathtt{add} \rhd (\mathtt{P} \lhd \mathtt{sparse}\ /\&/(\mathtt{S} \lhd \mathtt{sparse}) \lhd \mathtt{sparse_shift})
\end{aligned} \tag{2.12}$$

où, toujours dans le cas particulier de la matrice tridiagonale

$$\begin{aligned}
\mathtt{sparse_shift} &= \mathtt{shift} \circ \mathtt{shift} \circ \mathtt{sparse} \\
&= \quad \lambda(i,j)\ \ |(3{\le}i{\le}n, j{=}i) \,.\, (i, j{-}2) \\
&\quad \#\ \lambda(i,j)\ \ |(1{\le}i{\le}n{-}2, j{=}i{+}2) \,.\, (i, j{-}2).
\end{aligned}$$

Il faut ensuite appliquer `gather` à la somme le long des lignes. Soit

$$\mathtt{S1} = \mathtt{S} \lhd \mathtt{sparse}. \tag{2.13}$$

En utilisant la règle (1.5) suivie de la règle (1.6) on peut écrire

$$\begin{aligned}
\mathtt{gather::S1} &= \mathtt{add} \rhd (\mathtt{gather::}(\mathtt{P} \lhd \mathtt{sparse})\ /\&/ \\
&\qquad \mathtt{gather::}(\mathtt{S1} \lhd \mathtt{sparse_shift})) \\
&= \mathtt{add} \rhd (\mathtt{gather::}(\mathtt{P} \lhd \mathtt{sparse})\ /\&/ \\
&\quad (\mathtt{gather::S1}) \lhd \mathtt{gather} \circ \mathtt{sparse_shift} \circ \mathtt{gather}^{-1}).
\end{aligned} \tag{2.14}$$

Comme `gather` ne s'applique qu'aux éléments non-nuls, on a

$$\mathtt{P'} = \mathtt{gather::}(\mathtt{P} \lhd \mathtt{sparse}). \tag{2.15}$$

Soit

$$\mathtt{S'} = \mathtt{gather::S1}. \tag{2.16}$$

On obtient au final

$$\mathtt{S'} = \mathtt{add} \rhd (\mathtt{P'/\&/}(\mathtt{S'} \lhd \mathtt{shift'})) \tag{2.17}$$

où la nouvelle opération géometrique `shift'` illustrée Fig.2.12 est définie par

$$\begin{aligned}
\mathtt{shift'} &= \mathtt{gather} \circ \mathtt{sparse_shift} \circ \mathtt{gather}^{-1} \\
&= \quad \lambda(i,j)\ \ |(i{=}1, 3{\le}j{\le}n) \,.\, (2, j{-}2) \\
&\quad \#\ \lambda(i,j)\ \ |(i{=}2, 3{\le}j{\le}n) \,.\, (3, j{-}2).
\end{aligned} \tag{2.18}$$

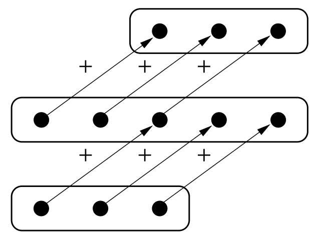


FIG. 2.12 – Nouvelle fonction géométrique `shift'`

Finalement, il faut recueillir le résultat, ce qui demande de modifier la fonction `last` pour que son domaine soit compatible avec celui de `gather`. L'équation qui définit `X` doit être découpée. De $\mathtt{X} = \mathtt{project::}(\mathtt{S} \lhd \mathtt{last})$, on extrait tout d'abord

$$\mathtt{Z} = \mathtt{S} \lhd \mathtt{last}$$

Le champ de données dense `S` doit tout d'abord être restreint au domaine creux, de façon à pouvoir appliquer le changement de base à `Z`. On réécrit $\mathtt{S} \lhd \mathtt{last}$ sous la forme $\mathtt{S} \lhd \mathtt{last} \lhd \mathtt{sparse\_last}$ qui se simplifie en $\mathtt{S1} \lhd \mathtt{sparse\_last}$ où `S1` a été défini dans l'équation (2.13) et avec

$$\begin{aligned} \mathtt{sparse\_last} \;=\; & \quad \lambda(i,j)\ |(1{\le}i{\le}n{-}2, j{=}i{+}2) \\ & \# \ \lambda(i,j)\ |(n{-}1{\le}i{\le}n, j{=}i) \end{aligned}$$

On peut alors appliquer le changement de base, ce qui mène à :

$$\begin{aligned} \mathtt{gather::Z} &= \mathtt{gather::}(\mathtt{S1} \lhd \mathtt{sparse\_last}) \\ &= \mathtt{S'} \lhd (\mathtt{gather} \circ \mathtt{sparse\_last} \circ \mathtt{gather}^{-1}) \end{aligned}$$

où `S'` est défini par (2.16). Avec

$$\mathtt{Z'} \;=\; \mathtt{gather::Z}$$

et

$$\begin{aligned} \mathtt{last'} &= \mathtt{gather} \circ \mathtt{sparse\_last} \circ \mathtt{gather}^{-1} \\ &= \quad \lambda(i,j)\ |(i{=}1, 3{\le}j{\le}n) \\ & \quad \# \ \lambda(i,j)\ |(i{=}2, n{-}1{\le}j{\le}n) \end{aligned} \tag{2.19}$$

on obtient l'équation (illustrée Fig.2.13)

$$\mathtt{Z'} \;=\; \mathtt{S'} \lhd \mathtt{last'}. \tag{2.20}$$

Le vecteur solution peut ensuite être aligné avec la diagonale de la matrice en appliquant `project'` (définie au-dessous) à `Z'`.

$$\begin{aligned} \mathtt{project'} \;=\; & \quad \lambda(i)\ |(i{=}2, 1{\le}j{\le}n{-}2).(1, j+2) \\ & \# \ \lambda(i)\ |(i = 2, n{-}1{\le}i{\le}n).(i,j) \end{aligned}$$

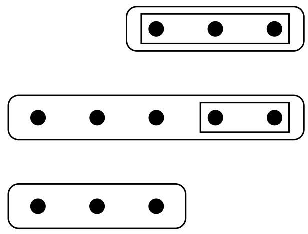


FIG. 2.13 – Nouvelle fonction géométrique `last'`

Finalement, on obtient `X'` par

$$\texttt{X'} = \texttt{project'::Z'}. \tag{2.21}$$

Enfin, on fusionne les équations (2.20) et (2.21) pour obtenir

$$\texttt{X'} \;=\; \texttt{project'::}(\texttt{S'} \lhd \texttt{last'}). \tag{2.22}$$

Ceci termine la transformation et définit un programme creux du produit matrice-vecteur. Le programme final se trouve dans la la Fig.2.14 où $\texttt{sparsematrix} = \texttt{gather} \circ \texttt{matrix}$ et $\texttt{align'} = \texttt{gather} \circ \texttt{pivot} \circ \texttt{align}$. Les fonctions `spread'`, `shift'`, et `last'` sont décrites dans les équations (2.10), (2.18) et (2.19)

```
SparseMatVec : (A,V) ↦ Z
{
sparsematrix::A = A
align'::B = V
P = prod ▷ (A/&/(B ◁ spread'))
S = add ▷ (P/&/(S ◁ shift'))
X = project'::(S ◁ last')
}
```

FIG. 2.14 – Multiplication matrice-vecteur creuse en PEI

L'utilisation de PEI pour la parallélisation de programme faisant intervenir des matrices bandes a aussi été abordée dans [94]. Un programme PEI décrivant la factorisation incomplète *LU* d'ordre 0 (voir l'introduction au calcul creux pour la factorisation incomplète) a été écrit pour des matrices bandes à 7 diagonales, de même que les changements de base permettant l'écriture de ce programe.

2.2.3 Autres transformations, autres stockages

L'exemple précédent nous a montré que PEI permet de transformer un programme plein en programme creux. Il faut pour cela définir un stockage creux, puis en déduire le changement de base qui en découle et l'appliquer au programe plein. Dans l'exemple donné, le stockage diagonal était utilisé, mais d'autres stockages peuvent être décrits, et d'autres matrices peuvent être utilisées.

Stockage SGP

Le stockage SGP (Sparse General Pattern [88, 87]) est un stockage adapté du stockage *Ellpack*. Il demande 3 tableaux à deux dimensions : A_{cr} pour les valeurs, A_i pour la ligne d'origine d'un élément et J_c pour le rang d'un élément dans sa colonne. Ces trois tableaux ont les mêmes dimensions, et le tableau J_c est ajouté au stockage Ellpack habituel pour faciliter le parallélisme. Le stockage de la matrice utilisée précédemment est donné par la Fig.2.15.

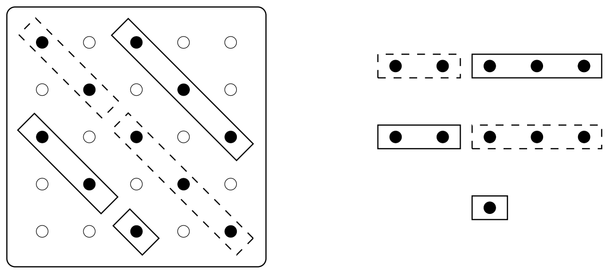


FIG. 2.15 – Une matrice et son stockage SGP

On peut alors définir un nouveau changement de base `gather1` qui exprime le passage du stockage plein au stockage creux

$$
\begin{array}{lcl}
\texttt{gather1} & = & \lambda(i,j)\ \ |(1{\leq}i{\leq}n-2, 3{\leq}j{\leq}n, j{=}i{+}2)\,.\,(1,j) \\
 & \# & \lambda(i,j)\ \ |(1{\leq}i{\leq}2, 1{\leq}j{\leq}2, i{=}j)\,.\,(1,j) \\
 & \# & \lambda(i,j)\ \ |(3{\leq}i{\leq}n, 3{\leq}j{\leq}n, i{=}j)\,.\,(2,j) \\
 & \# & \lambda(i,j)\ \ |(3{\leq}i{\leq}4, 1{\leq}j{\leq}2, i{=}j+2)\,.\,(2,j) \\
 & \# & \lambda(i,j)\ \ |(5{\leq}i{\leq}n, 3{\leq}j{\leq}n-2, i{=}j+2)\,.\,(3,j)
\end{array}
$$

et son inverse

$$
\begin{array}{lcl}
\texttt{gather1}^{-1} & = & \lambda(i,j)\ \ |(i=1, 1{\leq}i{\leq}2)\,.\,(j,j) \\
 & \# & \lambda(i,j)\ \ |(i=1, 3{\leq}j{\leq}n)\,.\,(j-2,j) \\
 & \# & \lambda(i,j)\ \ |(i=2, 1{\leq}i{\leq}2)\,.\,(j+2,j) \\
 & \# & \lambda(i,j)\ \ |(i=2, 3{\leq}i{\leq}n)\,.\,(j,j) \\
 & \# & \lambda(i,j)\ \ |(i=3, 3{\leq}i{\leq}n-2)\,.\,(j+2,j)
\end{array}
$$

On peut alors, de la même façon que précédemment, définir de nouvelles fonctions `spread1`, `shift1` et `last1`.

$$
\begin{aligned}
\texttt{spread1} &= \texttt{gather1} \circ \texttt{diagonal} \circ \texttt{gather1}^{-1} \\
&= \quad \lambda(i,j)\ |(i = 1, 1 \leq j \leq 2)\,.\,(1,j) \\
&\quad \#\ \lambda(i,j)\ |(i = 1, 3 \leq j \leq n)\,.\,(2,j) \\
&\quad \#\ \lambda(i,j)\ |(i = 2, 1 \leq j \leq 2)\,.\,(1,j) \\
&\quad \#\ \lambda(i,j)\ |(i = 2, 3 \leq j \leq n)\,.\,(2,j) \\
&\quad \#\ \lambda(i,j)\ |(i = 3, 3 \leq j \leq n-2)\,.\,(2,j)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{shift1} &= \texttt{gather1} \circ \texttt{sparse\_shift} \circ \texttt{gather1}^{-1} \\
&= \quad \lambda(i,j)\ |(i = 1, 3 \leq i \leq 4)\,.\,(1,j-2) \\
&\quad \#\ \lambda(i,j)\ |(i = 1, 5 \leq i \leq n)\,.\,(2,j-2) \\
&\quad \#\ \lambda(i,j)\ |(i = 2, 3 \leq i \leq 4)\,.\,(2,j-2) \\
&\quad \#\ \lambda(i,j)\ |(i = 2, 3 \leq i \leq n-2)\,.\,(3,j-2)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{last1} &= \texttt{gather1} \circ \texttt{sparse\_last} \circ \texttt{gather1}^{-1} \\
&= \quad \lambda(i,j)\ |(i = 1, 3 \leq j \leq n) \\
&\quad \#\ \lambda(i,j)\ |(i = 2, n-1 \leq j \leq n)
\end{aligned}
$$

Les divers mouvements de données mis en jeu par la nouvelle formulation sont illustrés par la Fig.2.16

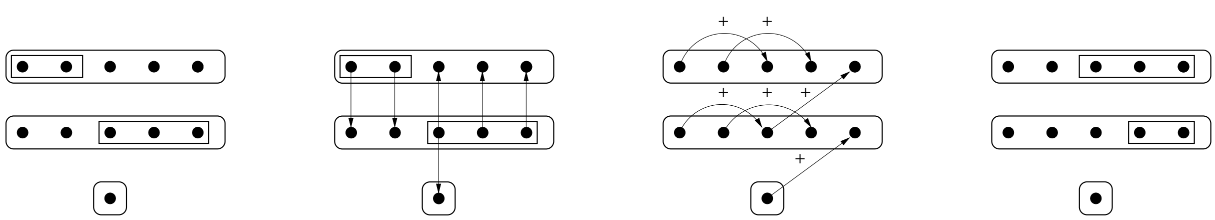


FIG. 2.16 – Fonctions géométriques `spread1`, `shift1` et `last1`.

Stockage MSR

Le stockage MSR (*Modified Sparse Row*) est un stockage adapté de CSR. On stocke simplement la diagonale à part, avant les autres éléments. Comme on connaît alors les numéros de lignes et colonnes de ces éléments, on ne les stocke pas et les $n+1$ premiers éléments du tableaux JA contiennent le tableau IA qui disparaît. La figure 2.17 illustre ce stockage.

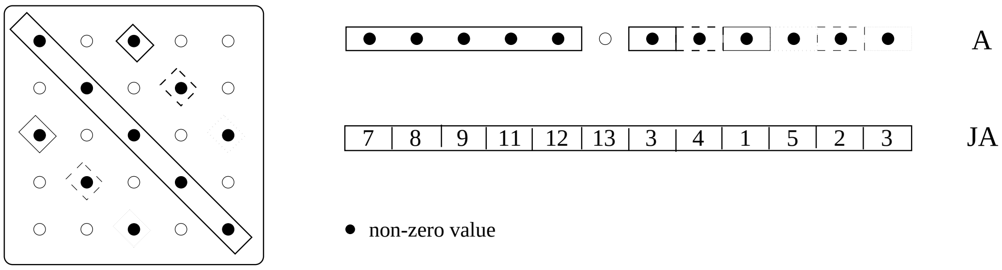


Fig. 2.17 – Matrice creuse et son stockage MSR

Le changement de base qui illustre ce stockage est donné ci-après.

$$
\begin{aligned}
\texttt{gather2} \;=\; & \lambda(i,j) \;|(1{\leq}i{\leq}2, j{=}i{+}2)\,.\,(n{+}i{+}1)\\
\# \;& \lambda(i,j) \;|(3{\leq}i{\leq}n{-}2, j{=}i{+}2)\,.\,(n{+}2i{-}1)\\
\# \;& \lambda(i,j) \;|(1{\leq}i{\leq}n, j{=}i)\,.\,(i)\\
\# \;& \lambda(i,j) \;|(3{\leq}i{\leq}n{-}2, j{=}i{-}2)\,.\,(n{+}2i{-}2)\\
\# \;& \lambda(i,j) \;|(n{-}1{\leq}i{\leq}n, j{=}i{-}2)\,.\,.(2n{+}i{-}3).
\end{aligned}
\tag{2.23}
$$

Et son inverse :

$$
\begin{aligned}
\texttt{gather2}^{-1} \;=\; & \lambda(i) \;|(1{\leq}i{\leq}n)\,.\,(i,i))\\
\# \;& \lambda(i) \;|(n{+}2{\leq}i{\leq}n{+}3)\,.\,(i{-}n{-}1, i{-}n{+}1)\\
\# \;& \lambda(i) \;|(n{+}4{\leq}i{\leq}3n{-}5), (i{-}n) \bmod 2{=}0)\,.\\
& \qquad ((i{-}n{-}1)/2+2), (i{-}n{-}1)/2)\\
\# \;& \lambda(i) \;|(n{+}4{\leq}i{\leq}3n{-}5), (i{-}n) \bmod 2{=}1)\,.\\
& \qquad ((i{-}n{-}1)/2{+}1), (i{-}n{-}1)/2{+}3)\\
\# \;& \lambda(i) \;|(3n{-}4{\leq}i{\leq}3n{-}3)\,.\,(i{-}2n{+}3, i{-}2n{+}1).
\end{aligned}
\tag{2.24}
$$

En suivant les mêmes étapes que précédemment (voir [105] pour les détails), on obtient les nouvelles fonctions “creuses” `spread2`, `shift2` et `last2` décrites ci-dessous et illustrées

Fig.2.18.

$$
\begin{aligned}
\texttt{spread2} &= \texttt{gather2} \circ \texttt{diagonal} \circ \texttt{gather2}^{-1} \\
&= \quad \lambda(i) \ |(1 \leq i \leq n) \,.\, (i) \\
&\quad \# \ \lambda(i) \ |(n{+}2 \leq i \leq n{+}3) \,.\, (i{-}n{+}1) \\
&\quad \# \ \lambda(i) \ |(n{+}4 \leq i \leq 3n{-}5, (i{-}n) \bmod 2 = 0) \,.\, ((i{-}n{-}1)/2) \\
&\quad \# \ \lambda(i) \ |(n{+}4 \leq i \leq 3n{-}5, (i{-}n) \bmod 2 = 1) \,.\, ((i{-}n{-}1)/2 + 3) \\
&\quad \# \ \lambda(i) \ |(3n{-}4 \leq i \leq 3n{-}3) \,.\, (i{-}2n{+}1) \\
\texttt{shift2} &= \texttt{gather2} \circ \texttt{sparse\_shift} \circ \texttt{gather2}^{-1} \\
&= \quad \lambda(i) \ |(n{+}2 \leq i \leq n{+}3) \,.\, (i{-}n{-}1) \\
&\quad \# \ \lambda(i) \ |(n{+}4 \leq i \leq 3n{-}5, (i{-}n) \bmod 2{=}1) \,.\, ((i{-}n{-}1)/2{+}1) \\
&\quad \# \ \lambda(i) \ |(3 \leq i \leq n{-}2) \,.\, (n{+}2i{-}2) \\
&\quad \# \ \lambda(i) \ |(n{-}1 \leq i \leq n) \,.\, (2n{+}i{-}3) \\
\texttt{last2} &= \texttt{gather} \circ \texttt{sparse\_last} \circ \texttt{gather}^{-1} \\
&= \quad \lambda(i) \ |(n{+}1 \leq i \leq n{+}2).(i) \\
&\quad \# \ \lambda(i) \ |(n{+}3 \leq i \leq 3n{-}5, (i{-}n) \bmod 2 = 1).(i) \\
&\quad \# \ \lambda(i) \ |(n{-}1 \leq i \leq n).(i)
\end{aligned}
$$

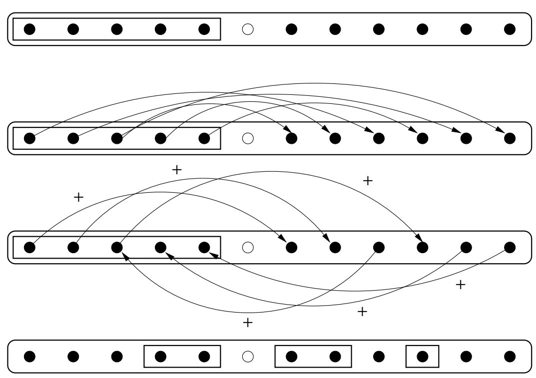


FIG. 2.18 – Fonctions géométriques `spread2`, `shift2` et `last2`.

Coordinate Storage

Pour finir, on peut prendre le cas d'une matrice de forme plus générale, comme par exemple la matrice

$$
A = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 3 & 0 & 4 & 0 \\ 5 & 0 & 6 & 0 & 7 \\ 0 & 8 & 0 & 9 & 0 \\ 0 & 0 & 10 & 0 & 11 \end{pmatrix}.
$$

en utilisant le “coordinate storage” :

$$
\begin{array}{rcl}
a & = & \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 4 & 6 & 7 & 3 & 9 & 2 & 5 & 8 & 10 & 11 \\ \hline\end{array} \\
ia & = & \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 3 & 3 & 2 & 4 & 1 & 3 & 4 & 5 & 5 \\ \hline\end{array} \\
ja & = & \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 4 & 3 & 5 & 2 & 4 & 3 & 1 & 2 & 3 & 5 \\ \hline\end{array} .
\end{array}
$$

On obtient alors le changement de base

$$
\begin{aligned}
\texttt{gather3} \;=\; & \quad\lambda(i,j)\ |(i=1,j=1)\,.\,(1) \\
& \#\ \lambda(i,j)\ |(i=1,j=3)\,.\,(7) \\
& \#\ \lambda(i,j)\ |(i=2,j=2)\,.\,(5) \\
& \#\ \lambda(i,j)\ |(i=2,j=4)\,.\,(2) \\
& \#\ \lambda(i,j)\ |(i=3,j=1)\,.\,(8) \\
& \#\ \lambda(i,j)\ |(i=3,j=3)\,.\,(3) \\
& \#\ \lambda(i,j)\ |(i=3,j=5)\,.\,(4) \\
& \#\ \lambda(i,j)\ |(i=4,j=2)\,.\,(9) \\
& \#\ \lambda(i,j)\ |(i=4,j=4)\,.\,(6) \\
& \#\ \lambda(i,j)\ |(i=5,j=3)\,.\,(10) \\
& \#\ \lambda(i,j)\ |(i=5,j=5)\,.\,(11). \qquad (2.25)
\end{aligned}
$$

Il est décrit par une fonction point-à-point. On doit suivre le même raisonnement que pour la partie 2.2.2 pour appliquer ce changement de base. En particulier, il est nécessaire de recalculer de nouvelles fonctions `sparse_shift` et `sparse_last` et de donner la nouvelle fonction `sparse` correspondant aux éléments non-nuls de cette fonction.

2.3 Application au parallélisme

Si l’écriture d’un programme creux à partir d’un programme plein pose problème, la parallélisation de programmes creux est elle aussi problématique. On ne passe pas facilement d’une version séquentielle à une version parallèle, d’autant plus que les nombreuses indirections générées par les stockages creux ne permettent pas les analyses habituelles sur les nids de boucles. Il est donc intéressant de voir quelles solutions PEI peut apporter à ce problème de la parallélisation.

2.3.1 Parallélisation des programmes creux

Les programmes faisant intervenir des matrices creuses ont bien souvent pour noyau de calcul des résolutions de systèmes linéaires. Ce sont donc sur ces parties de calculs que se concentrent les efforts de parallélisation.
En ce qui concerne les résolutions directes (par décomposition *LU* ou de Cholesky), la formulation creuse à partir des méthodes multifrontales permet d’obtenir une formulation parallèle : on peut arriver à traiter plusieurs colonnes en parallèle quand leurs structures

sont indépendantes. D'autre part, dès qu'on arrive à un traitement avec de nombreuses valeurs, on peut faire appel à des procédures "pleines" déjà parallélisées (via des bibliothèques par exemple). Le solveur MUMPS (Multifrontal Massively Parallel Solver) [4] fait partie des solveurs qui intègrent de telles méthodes en parallèle, mais diverses approches existent, [32] par exemple.

Si on reprend les algorithmes de résolution par méthodes indirectes présentées dans 2.1.2 on voit que les calculs les plus lourds concernent les produits matrice-vecteur et l'application des préconditionnements.
La parallélisation de ces algorithmes consiste en fait à partitionner le graphe de la matrice (défini dans 2.1.1) afin de distribuer les données, ce qui guidera ensuite l'exécution parallèle du programme. On distribue généralement les lignes entières de la matrice, et les composantes correspondantes des vecteurs sont alignées avec les lignes : $v[i]$ se trouve sur le même processeur que $A(i, *)$. Les calculs sont donc effectués selon la *owner computes rule*. Des partitionneurs [55] calculent la distribution de la matrice en essayant à la fois d'équilibrer la charge entre les processeurs et de minimiser les communications induites par les vecteurs distribués. Les préconditionnements sont quant à eux souvent appliqués par blocs et donc localement, en suivant la distribution de la matrice.

Des structures de données adaptées sont mises en place, plus ou moins selon les schémas abordés dans [9, 103]. On sépare tout d'abord, sur chaque processeur, les composantes de vecteurs en quatre groupes.
- Les points internes : il s'agit des composantes du vecteur qui ne nécessitent aucune communication pour être calculées.
- Les points frontières : ce sont les composantes du vecteur qui nécessitent des communications pour être calculés.
- Les points externes : ce sont les composantes distantes de vecteurs qu'il sera nécessaire de communiquer pour calculer une des composantes locales.
- Les autres points.

En rapport avec cette distinction, on met en place une structure locale de données. L'ensemble des lignes de la matrice A qui sont locales à un processeur est noté A_{loc}. Cette matrice locale est séparée en 2 : B_{loc} qui correspond aux valeurs de la matrice qui interviennent dans un calcul avec les points internes et B_{ext} qui prend en compte les autres valeurs. Le produit matrice-vecteur peut alors s'effectuer :

1. multiplication locale de B_{loc} avec les points concernés ;
2. communication des variables externes ;
3. multiplication de B_{ext} avec ces points externes.

Cette façon de procéder a l'avantage de permettre un recouvrement calcul/communication : durant la multiplication locale, on reçoit les composantes extérieures.

Différentes bibliothèques tirent d'ailleurs parti de ce système. On peut citer :
- PetsC : `http://www-fp.mcs.anl.gov/petsc/`.
  Il s'agit d'une bibliothèque écrite en C++, basée sur MPI [45, 7]. La bibliothèque comporte des routines de résolution de système par les méthodes de Krylov ainsi que

de résolution d'équations différentielles, tout en proposant divers stockages creux et les routines d'opérations matrice/vecteurs qui vont avec.

- Aztec [52, 99] : `http:www.cs.sandia.gov/CRF/aztec1.html`
  Aztec permet de résoudre des systèmes linéaires. La distribution des matrices est basée sur le partionneur Chaco [48, 49] et de nombreuses méthodes de Krylov sont disponibles (CG, CGS, BI-CGStab, etc.) de même que de nombreux préconditionnements (ILU, Jacobi, ...).
- PSparslib [97] : `http://www.cs.umn.edu/Research/arpa/p_sparslib/psp-abs.html`
  Il s'agit de la version parallèle d'une bibliothèque séquentielle, Sparskit. Elle permet de résoudre le systèmes linéaires par des méthodes de Krylov en visant une exploitation sur les machines à mémoire distribuées [72]. Plusieurs formats de matrices sont possibles, diverses méthodes de résolution sont disponibles (GMRES, FGMRES, ...) ainsi que de nombreux préconditionnements.

2.3.2 Parallélisation avec Pei

De la même façon qu'on a donné une traduction HPF du programme Pei plein du produit matrice vecteur dans la partie 1.1.3, on peut proposer une traduction HPF du programme creux. Prenons le cas du stockage MSR par exemple.

Le processus de traduction rencontre alors quelques difficultés, en particulier pour la traduction de la réduction. On se souvient qu'il faut donner une transformation espace-temps qui permet d'avoir un ordre temporel pour l'exécution de la somme. Or, ici, après la transformation plein/creux, on obtient un domaine à une dimension qu'on ne peut plus associer à une dimension temporelle. Dans le cas plein, le vecteur $(0, 1)$ représentait en fait une direction du temps.

D'autre part, le traducteur n'intègre pas les instructions de distribution de données irrégulières qu'on trouve dans HPF2. Ceci empêche la traduction de se faire. Néanmoins, on peut proposer une traduction basée sur l'esprit dans lequel le traducteur a été développé. Cette traduction est proposée en figure 2.19.

2.4 Autres travaux

D'autre travaux ayant une approche similaire (partir de la structure pleine pour arriver à la structure creuse et/ou parallèle) ont été faits.

Signalons tout d'abord les travaux de Bik et Wijschoff [11, 10, 12]. Leur idée a été de construire un compilateur creux prenant en entrée un programme plein et le transformant en un programme creux. L'utilisateur signale par des directives de compilation quelles sont les structures de données creuses. Le compilateur va choisir le stockage utilisé. Tout d'abord, une phase d'analyse détecte les parties du programme dans lesquelles les données creuses interviennent. Des instructions conditionnelles sont insérées pour séparer les calculs faisant intervenir les éléments non-nuls des calculs avec des éléments nuls. Ces derniers peuvent souvent être éliminés (c'est la cas avec le produit matrice-vecteur). Si A est la représentation pleine d'une structure de donnée creuse A', on note $E(A)$ l'ensemble des

```
      program SparseMatVec
      real A(1:3*n-4), V(1:n), B(1:3*n-4)
      real D(1:3*n-4), P(1:3*n-4), S(1:3*n-4), Z(1:3*n-4)
!hpf$ template W(1:3*n-4)
!hpf$ align A,B,P,S,Z with T
!hpf align V(1:n) with T(1:n)
      forall (i=1:n) T(i)=V(i)
      forall (i=1:3*n-4)
        if (1.lt.i .and. i.lt.n) D(i)=T(i)
        if (n+2.lt.i .and. i.lt.n+3) D(i)=T(i-n+1)
        if (n+4.lt.i .and. i.lt.3*n-5) then
           if (mod((i-n),2).eq.0) then
              D(i)=T((i-n-1)/2)
           else
              D(i)=T((i-n-1)/2 +3)
        endif
        if (3*n-4.lt.i .and. i.lt.3n-3) D(i)=T(i-2*n+1)
      endforall
!
      forall (i=1:3*n-4) P(i)=A(i)*D(i)
!
      forall (i=1:3*n-4)
        if (1.lt.i .and. 2.lt.i) S(i)=P(i)
        if (i.eq n+1)  S(i)=P(i)
        if (n+4.lt.i .and i.lt.3*n-5) then
          if (mod(i,2) .eq. 0) S(i)=P(i)
        endif
        if (i.eq.3*n-4) then S(i)=P(i)
      endforall
      forall (i=3:n+3)
        if (3.lt.i .and. i.lt.n-2) S(i)=S(i)+S(n+2*i-2)
        if (n-1.lt.i .and. i.lt.n) S(i)=P(i)+S(2*n+i-3)
        if (n+2.lt.i .and. i.lt.n+3) S(i)=P(i)+S(i-n-1)
      endforall
      forall (i=n+4:3*n-5)
        if (mod(i,2) .eq. 1) S(i)=P(i)+S((i-n-1)/2+1)
      endforall
      forall (i=1,n)
        if (1.lt.i .and. i.lt.2) X(i)=S(i-n)
        if (3.lt.i .and. i.lt.n-2) X(i)=S(2*i+n-1)
        if (n-1.lt.i .and. i.lt.n) X(i)=S(i)
       endforall
```

FIG. 2.19 – Traduction HPF du programme matrice-vecteur en stockage MSR

indices des valeurs non-nulles, et σ la fonction qui associe un indice dans la représentation pleine avec son correspondant dans la représentation creuse. Le type de transformation évoquée ci-dessus peut-être illustré par l'exemple suivant :

```
                          if (i,j) ∈ E(A)
                            x = A'(σ(i,j))
x = A(i,j)   devient      else
                            x = 0.
```

Du point de vue de PEI , σ est un changement de base, mais l'utilisateur ne le choisit pas. Le compilateur s'en charge. Il choisit en fonction des indications fournies par l'utilisateur ou de la structure de la matrice analysée par le compilateur, la matrice étant lue dans un fichier. La façon dont les données sont accédées durant le calcul est aussi pris en compte. Seuls les stockages pour lesquels au moins une direction d'accès existe (comme CSR, par exemple, accessible ligne par ligne) peuvent être utilisés. D'autre part, les nids de boucles sont restructurés pour que la direction d'accès choisie soit utilisée dans le plus de boucles possibles. Des fonctions d'expansion permettent d'adapter le stockage pour d'autres boucles, permettant aussi d'insérer ou d'enlever des éléments dans la matrice. Enfin, la transformation ne se fait pas comme dans PEI par composition de fonctions, mais en calculant des tableaux d'indirections pour réordonner les matrices.

Une autre approche est celle de Pingali [60]. Les accès aux données creuses sont vues comme des requêtes dans une base de données relationnelle. Les accès aux données dans un nid de boucles sont représentées par une énumération de n-uplets satisfaisant une requête. Dans le cas du produit matrice-vecteur

```
Do i = 1, n
  Do j = 1, n
    B(j) = B(j) + A(i,j) * X(j)
```

la matrice A peut être vue comme une relation dont les n-uplets sont $\langle i, j, a\rangle$, i et j étant les indices de lignes, a étant les valeurs. On peut faire la même chose avec X, représenté par $\langle j, x\rangle$. Finalement, le domaine d'itération $1 \leq i \leq n, 1 \leq j \leq n$ peut être considéré comme une relation R sur $\langle i, j\rangle$. La sélection des données utiles au calcul du produit matrice-vecteur AX est en fait la sélection des valeurs non-nulles a de A pour i et j respectant R, ce qui se résume par la formule :

$$\sigma_{a\neq 0}(R(i,j) \bowtie A(i,j,a) \bowtie X(j,x)$$

où σ est un opérateur de sélection qui sélectionne les n-uplets satisfaisant le prédicat en indice.

A partir de ces bases, il faut décrire les structures de données au compilateur. En effet, il est plus intéressant de proposer une interface de haut niveau pour l'accès aux données que de coder "en dur" un nombre fixe de formats. Ici, l'utilisateur doit donner la structure hiérarchique des indices : par exemple dans le cas d'un stockage CSR, on énumère les lignes, à l'intérieur desquelles on énumère les colonnes, contrairement à CSC où lignes et colonnes sont inversées. Puis l'utilisateur doit fournir un certain nombre de fonctions d'accès aux donnnées. Il faut ensuite formuler les requêtes et découvrir les jointures, ce qui peut se révéler moins simple que dans le cas donné en exemple, si les indices de tableau ne sont plus les indices de boucles, mais des fonctions affines de ces indices. Néanmoins, grâce à la représentation matricielle qu'on peut faire des fonctions affines, ce problème se résout. Il faut finalement sélectionner l'ordre des jointures et les implémenter, ce qui se fait grâce aux indications données par l'utilisateur.

Cette suite d'opérations permet de transformer un programme plein en un programme creux. Les auteurs se proposent aussi de générer du code parallèle [58, 59]. Ils suppposent que la matrice est distribuée ligne par ligne, chaque ligne i étant placée sur le processeur $p(i)$, comme on l'a vu à la fin de la section 2.1.2. Les matrices ont donc une représentation locale et une représentation globale, le passage d'une représentation à l'autre se fait par une fonction, qui peut être vue comme une relation entre les indices locaux, globaux et le numéro du processeur. On peut alors évaluer les diverses requêtes parallèles, tout en choisissant le site sur lesquel elles doivent être évaluées.

Une troisième approche est celle de R. Adle et F. Delaplace [1, 30]. Ici, il n'est pas question de transformer un programme plein en creux pour ensuite le paralléliser, mais de se placer directement dans l'optique d'une parallélisation. Une étape préliminaire à la parallélisation est l'analyse de dépendances qui permet par la suite d'ordonnancer les calculs. L'analyse de dépendances d'un programme plein sert de base à l'analyse de dépendances du programme creux correspondant ; il s'agit ensuite de supprimer les dépendances qui n'ont plus lieu d'être à cause de la spécificité des éléments nuls. A partir de cette analyse de dépendance creuse, on peut générer du code parallèle creux. Comme il est difficile de faire l'analyse de dépendances creuse avant l'exécution du programme car la structure creuse n'est pas toujours connue à l'avance, le principe de l'inspecteur-exécuteur est suggéré : l'inspecteur, à partir de l'analyse pleine, construit un nouveau graphe de dépendances creux et génère le parallélisme. L'exécuteur exécute ensuite le programme selon le schéma établi par l'inspecteur. Le parallélisme est donc généré lors de l'exécution, dynamiquement.

Enfin, une dernière approche de l'écriture des programmes creux est celle de HPF. Il n'existe pas de bibliothèques en HPF pour le calcul creux, d'ailleurs il y en assez peu pour le calcul plein (il n'y a pas de LAPACK pour HPF par exemple). Il faut dire qu'HPF ne permet pas vraiment les distributions contenant des indirections. Il s'agit d'une directive qui fait partie des extensions approuvées de la norme HPF2.0 [51], via un nouveau format de distribution `INDIRECT` qui permet d'associer un numéro de processeur à un élément particulier d'un tableau. Elle n'est pas encore bien supportées par les compilateurs : pghpf [15] qui ne supporte que `GE_BLOCK` une distribution par blocs contigus de taille différente, ADAPTOR [17], etc., ce qui explique que HPF ne soit pas très utilisé. Néanmoins, des travaux ont été menés pour voir comment concilier parallélisme de données et structures irrégulières [110, 46, 16], en se basant sur HPF.

Chapitre 3

Résolution des équations de Navier-Stokes

L'étude de l'écoulement des fluides nécessite de nombreux calculs parfois longs et couteux ; la parallélisation des programmes de modélisation de ces écoulements est donc d'un intérêt certain. Nous présentons par la suite l'étude de la parallélisation d'un code de mécanique des fluides développé à l'Institut de Mécanique des Fluides de Strasbourg simulant l'écoulement d'un fluide autour d'un obstacle. Cela permet de modéliser les échanges de chaleur dans les centrales thermiques ou la sedimentation des particules.

3.1 Équation - Schéma de résolution

L'écoulement d'un fluide est décrit par des équations aux dérivées partielles appelées *équations de Navier-Stokes*. Ces équations font intervenir les variables décrivant le fluide, par exemple la vitesse v et la pression p. On a alors les équations :

$$\begin{aligned}
&\frac{\partial v}{\partial t} + v\nabla v - \nu\nabla^2 v + \nabla p = 0\\
&\nabla . v = 0
\end{aligned}$$

où $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$.

Par résolution de ces équations on peut, suivant les cas, mettre en évidence l'apparition d'instabilités dans l'écoulement du fluide, instabilités qui se caractérisent le plus souvent par une période de transition. On peut observer par exemple des oscillations de l'écoulement avec une fréquence qui varie peu, mais dont l'amplification varie.

Résoudre une équation aux dérivées partielles, c'est calculer une fonction - ou *la* fonction s'il y a unicité de la solution - qui vérifie l'équation donnée. On pourra se réferrer à [92] pour en savoir plus. On ne peut généralement pas obtenir une formulation analytique de cette solution, bien que certaines équations aient des solutions analytiques connues. On se contente donc de calculer les valeurs de la fonction en un certain nombre de points donnés, que ce soit au niveau temporel (valeurs de la fonction à certaines dates) ou spatial (valeurs de la fonction en certains points du domaine où le phénomène à étudier se produit).

Ceci nécessite une discrétisation au niveau temporel comme au niveau spatial. La discrétisation temporelle suppose le choix d'un pas de temps dont la valeur dépend du phénomène étudié : dans le cas de l'étude des instabilités si l'écoulement connaît des oscillations, il faut un pas de temps assez petit par rapport aà la période des oscillations, ce qui est toujours le cas suite au critère de convergence du schema de résolution choisi. La discrétisation spatiale quant à elle consiste souvent à choisir un certain nombre de points dans le domaine étudié. On recherche alors la valeur de la fonction solution en ces points. Le placement des points dans le domaine géométrique dépend généralement du phénomène physique étudié : il peut par exemple être judicieux de placer plus de points de discrétisation là où on devine qu'un phénomène va apparaître. Enfin, le nombre de points dépend de la précision recherchée.

Revenons au problème : l'étude d'un phénomène de transition à la turbulence entraînant des oscillations. Le caractère presque périodique du phénomène suggère une modélisation particulière des variables : vitesse et pression du fluide peuvent être représentées sous forme de décomposition de Fourier. On peut en effet réécrire le terme $\frac{\partial v}{\partial t}$ pour faire intervenir une seconde variable temporelle sous la forme : $\frac{\partial v}{\partial t} + \frac{\partial v}{\partial s}$ ce qui permet de séparer les effets périodiques des effets transitoires [33, 19]. On peut alors réécrire les variables v et p sous la forme

$$
\begin{aligned}
v(s,t,X) &= \sum_{n=-\infty}^{n=+\infty} c_n(s,X)e^{in\omega t} \\
p(s,t,X) &= \sum_{n-\infty}^{n=+\infty} d_n(s,X)e^{in\omega t}.
\end{aligned}
$$

s et t sont les variables de temps, X représente les variables spatiales, n est l'harmonique, T la période et on pose $\omega = \frac{2\pi}{T}$. La dépendance de c_n et d_n vis-à-vis de s permet de prendre en compte des variations faibles de la période. Notons que c_n et d_n vérifient :

$$
\begin{aligned}
c_n &= -\bar{c}_n \\
d_n &= -\bar{d}_n
\end{aligned}
$$

où $\bar{c}$ désigne le conjugué de c. On injecte cette nouvelle formulation dans les équations de Navier-Stokes pour obtenir finalement :

$$
\begin{aligned}
&\frac{\partial}{\partial s}c_n + (\mathrm{i}\; n\; \omega - \nu\nabla^2)c_n + \sum_{n=-\infty}^{n=+\infty}(c_k.\nabla).c_{n-k} + \nabla d_n = 0 \\
&\nabla.c_n = 0
\end{aligned}
$$

où on a $\mathrm{i}^2 = -1$.

On remarque que pour $n = 0$ et $k = 0$ on retrouve les équations de Navier Stokes d'origine. Afin de faire apparaître les harmoniques lors du calcul il faut empêcher le mode $n = 0$ de capter les echelles temporelles "rapides" (de l'ordre de la période). Il faut alors choisir un pas de temps Δs de l'ordre de la période dans les équations nouvellement obtenues.

La discrétisation temporelle se passe normalement en trois temps. Considérons ce qui se passe entre les étapes de temps i et $i+1$. On approxime :

$$\frac{\partial v}{\partial t} \approx \frac{v^{i+1} - v^i}{\Delta t}$$

On fait intervenir deux variables temporaires v^* et $\hat{v}$ à calculer, pour résoudre successivement

$$\frac{v^* - v^i}{\Delta t} = -v^i \nabla v^i \tag{3.1}$$

$$\frac{\hat{v} - v^*}{\Delta t} = -\nabla p^i \text{ et div } \hat{v} = 0 \tag{3.2}$$

$$\frac{v^{i+1} - \hat{v}}{\Delta t} = \nu \nabla^2 v^{i+1} \tag{3.3}$$

Le calcul de v^* dans (3.1) ne pose pas problème particulier car la résolution est explicite, ne nécessitant pas l'inversion d'un opérateur. La résolution de (3.2) permet de calculer tout d'abord la pression par résolution de l'équation de Poisson :

$$\nabla^2 p^{n+1} = \frac{1}{\Delta t} \mathrm{div} v^*$$

La calcul de $\hat{v}$ s'ensuit.

Enfin, la résolution de (3.3) est traitée implictement. On obtient v^{i+1} en résolvant les équations de Helmholtz

$$(1 - \nu \Delta t \nabla^2) v^{i+1} = \hat{v}.$$

Néanmoins, avec la modélisation en série de Fourier, il est nécessaire de n'avoir que des résolutions implicites qui sont des méthodes de résolution plus stables, afin de pouvoir choisir un pas de temps assez long. Il faut donc légèrement reprendre le schéma précédent. En particulier, l'équation à résoudre devient :

$$\begin{aligned} &\frac{c_n^{j+1} - c_n^j}{\Delta s} = \mathrm{i}\, n\, \omega c_n^{j+1} - \sum_{n=-\infty}^{n=+\infty} (c_k^{j+1}.\nabla).c_{n-k}^{j+1} - \nu \nabla^2) c_n^{j+1} - \nabla d_n^{j+1} = 0 \\ &\nabla c_n^{j+1} = 0. \end{aligned}$$

On peut résoudre cette équation en faisant intervenir des champs intermédiaires :

$$\begin{aligned} \tilde{c}_n^0 &= c_n^j \\ c_n^{j+1} &= \tilde{c}_n^q. \end{aligned}$$

c'est-à-dire en introduisant un deuxième pas de discrétisation. En combinant la discrétisation temporelle et la modélisation en série d Fourier, on obtient les trois équations

(3.4), (3.5), (3.6) à résoudre successivement, et ce pour chaque n qui intervient dans la décomposition en série de Fourier.

$$
\begin{aligned}
&\frac{\tilde{c}_n^*}{\Delta h} + \frac{\tilde{c}_n^*}{\Delta t} = \frac{\tilde{c}_n^q}{\Delta h} + \frac{c_n^j}{\Delta t} + \mathrm{i}\, n\, \omega \tilde{c}_l^q - \sum_{n=-\infty}^{n=+\infty} (\tilde{c}_k^q.\nabla).\tilde{c}_{n-k}^q = 0 && (3.4)\\
&\hat{\tilde{c}}_n\left(\frac{1}{\Delta h} + \frac{1}{\Delta t}\right) = \tilde{c}_n^*\left(\frac{1}{\Delta h} + \frac{1}{\Delta t}\right) - \nabla \tilde{d}_n^{q+1} \text{ et } \mathrm{div}\hat{\tilde{c}} = 0 &&\\
&\text{avec } \nabla^2 \phi = \nabla \tilde{c}_n\left(\left(\frac{1}{\Delta h} + \frac{1}{\Delta t}\right) - \nabla^2\right)\tilde{d}_n^q &&\\
&\tilde{d}_n^{q+1} = \tilde{d}_n^q + \phi && (3.5)\\
&\left(\frac{1}{\Delta h} + \frac{1}{\Delta t} - \nu \nabla^2\right) w = \hat{\tilde{c}}\left(\frac{1}{\Delta h} + \frac{1}{\Delta t}\right) + \left(\frac{1}{\Delta h} + \frac{1}{\Delta t} - \nu \nabla^2\right)\tilde{c}_n^q && (3.6)\\
&\text{où } \tilde{c}_n^{q+1} = \tilde{c}_n^q + w &&
\end{aligned}
$$

On constate dans (3.4) que le terme en $\mathrm{i}\omega c$ est traité de façon explicite. En fait, le traitement implicite de ce terme reviendrait à inverser un opérateur non-hermitien, ce qui reviendrait à abandonner la méthode du gradient conjugué réservée aux opérateurs symétriques.

Il faut ensuite se préocupper de la discrétisation spatiale. Le domaine physique est maillé, voir la figure 3.1 : on reconnaît l'obstacle rond alors que la partie à sa droite correspond à l'endroit où se situe le sillage et donc le phénomène que l'on souhaite étudier. Sur chaque élément, la fonction recherchée est approximée.

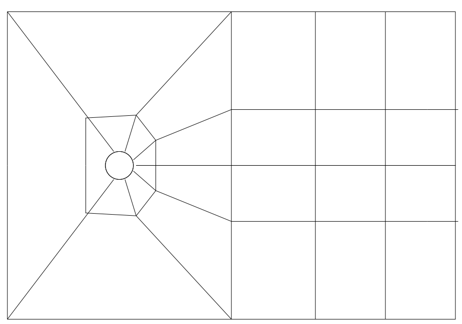


FIG. 3.1 – Un des maillages

Dans le cas des éléments finis [92, 89], la restriction à chaque élément de la fonction cherchée est un polynôme. Plus on souhaite que ce polynôme soit de degré élevé, plus il faut prendre de points dans l'élément. Dans notre cas, une discrétisation par éléments spectraux [47] a été choisie. Les éléments spectraux sont des sortes d'éléments finis qui contiennent beaucoup de points, ce qui revient à dire que les approximations sur chaque élément se font par des polynômes d'ordre élevé. D'ailleurs, le nom même d'éléments spectraux provient de leur lien avec les méthodes spectrales [43] qui permettent de résoudre certaines équations par une approximation précise en utilisant une "bonne" interpolation sur le domaine concerné. La difference entre la méthode des éléments spectraux et celle des éléments finis tient au fait que la convergence ne s'obtient pas en multipliant le nombre d'éléments mais en augmentant leur ordre, c'est-à-dire la dimension de la base spectrale au sein d'un élément.

Sur un élément donné, on va définir un ensemble de fonctions formant une base dans laquelle on cherche à exprimer la solution. Ces fonctions sont des polynômes d'interpolation, et toute la difficulté consiste à prendre de bons points à l'intérieur de l'élément pour avoir une approximation correcte. C'est ici qu'on fait appel aux méthodes dites pseudospectrales qui permettent d'effectuer de bons choix (interpolation de Gauss-Lobato Legendre [43]). On prend donc des éléments finis plus gros que pour les méthodes d'éléments finis classiques, en faisant une approximation plus précise sur chacun d'eux. On garde les propriétés de continuité de la fonction finale recherchée sur le domaine entier, continuité assurée par la méthode et la conformité du maillage.

3.2 Parallélisation

La parallélisation de tels problèmes issus de la physique se fait généralement sous la forme de *décomposition de domaine*. Il s'agit de partitionner le domaine physique en morceaux qui sont distribués sur les processeurs, voir figure 3.2. Une partie des calculs est local à chaque processeur, mais certaines données doivent être comuniquées, notamment des valeurs sur des points de la frontière entre deux domaines, ou communes à deux domaines en cas de *recouvrement*.

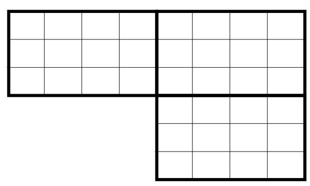


FIG. 3.2 – Domaine maillé et partitionné

On a un code parallèle qui suit le canevas suivant :

```
Boucle séquentielle sur le temps
  Boucle parallele sur chaque domaine
    calcul sur le domaine
    communication aux domaines proches des valeurs aux interfaces
```

Dans notre cas, la formulation du problème en série de Fourier avec deux pas de temps induit une nouvelle parallélisation possible :

```
Boucle 1 séquentielle sur le temps lent s
  Boucle 2 séquentielle sur le temps h
    Boucle parallèle sur les harmoniques n
      Résolution de (3.4)
      Résolution de (3.5)
      Résolution de (3.6)
```

En effet, si au lieu de distribuer les sous-domaines spatiaux on distribue les données c_n et d_n de façon à les aligner, c'est-à-dire en plaçant c_n et d_n sur le processeur n, la plupart des calculs sont locaux : les équations (3.5) et (3.6) ne font intervenir que des variables d'indice n et donc leur résolution, avec cette parallélisation, est locale. La méthode choisie pour résoudre les équations suggère donc une nouvelle parallélisation possible, qu'il n'est pas possible d'appliquer dans le cas d'une modélisation plus conventionnelle. On doit néanmoins se pencher sur la résolution de (3.4), qui nécessite des communications lors du calcul de la somme.

Un code FORTRAN de 30000 lignes existe, qui prend en compte la modélisation en série de Fourier des variables vitesse et pression. Dans ce code, la résolution de (3.4) représente en moyenne un quart du temps de calcul, la résolution de (3.5) et (3.6) utilisant les trois quart du temps restant, comme le montre le tableau 3.1. Il est d'autant plus intéressant de paralléliser efficacement la résolution de (3.5) et (3.6) que ce sont les calculs les plus gourmands. La parallélisation proposée précédemment répond justement à ces impératifs.

Nombre d'harmoniques	Temps de calcul total (en s)	Temps de résolution de (3.4)	Pourcentage du temps total
1	406	54	13,3%
2	933	269	28,3%
3	3460	650	18,7%
4	4572	1192	26,0%
5	6461	1889	29,2%
6	8035	2751	34,2%
7	10506	3773	35,9%

TAB. 3.1 – Temps d'exécution du programe séquentiel de résolution du problème, pour 7 itérations externes et 200 internes.

Face à cette parallélisation, c'est finalement la résolution de (3.4) qui pose problème puisque la distribution des calculs de résolution de (3.5) et (3.6) et par conséquent des données sur les processeurs induit des communications coûteuses. Il ne faudrait pas que le coût de cette étape devienne prohibitif.

3.2.1 Reformulation de problème

Il s'agit donc de minimiser le temps d'exécution de la résolution de (3.4), de telle sorte que cette étape secondaire lors d'une exécution séquentielle ne devienne pas d'un coût trop important en parallèle. Pour cela, on ne doit pas optimiser la résolution, proprement dite, mais les mouvements de données qu'elle met en jeu. Le calcul à effectuer pour la résolution de cette étape se ramène au calcul de :

$$A_n = \sum_{k=-\infty}^{k=+\infty} A(c_{n-k}, c_k) \tag{3.7}$$

où A est une forme bilinéaire, et où c_k représente c_k^q qui intervient dans la décomposition de v. On a simplement reformulé l'équation (3.4) de façon exploitable.

Bien évidemment, il n'est pas question de garder des bornes infinies pour calculer la somme (3.7), d'autant plus que physiquement cela n'aurait pas de sens car les harmoniques de degré élevé contribuent très peu au phénomène. On modélise donc les variables avec $N+1$ harmoniques, d'où $N+1$ variables c_k et d_k (c_0, ... c_N). Ceci entraîne les contraintes suivantes :

$$\begin{aligned} |n-k| &\leq N \\ |k| &\leq N \end{aligned}$$

La réécriture de A_n en tenant compte du fait que $c_{-k} = \bar{c}_k$ se traduit donc par :

$$A_n = \sum_{k=1}^{k=N-n} A(\bar{c}_k, c_{n+k}) + \sum_{k=0}^{k=n} A(c_k, c_{n-k}) + \sum_{k=n+1}^{k=N} A(c_k, \bar{c}_{k-n}) \tag{3.8}$$

Cette formulation n'est pas unique : en effet, on a $c_0 = \bar{c}_0$ donc c_0 est réel. On peut donc faire commencer la première boucle à 0 et la deuxième à 1. Cependant, la formulation choisie est mieux adaptée au problème de la parallélisation, comme on le verra ultérieurement.

Afin d'analyser les mouvements de données mis en jeu par ce calcul, on va finalement l'exprimer sous forme de nids de boucles. Le calcul de A_n dépend de c_k de 3 façons différentes (utilisation ou non du conjugué, place dans la forme bilinéaire), qui correspondent aux trois membres de la somme de l'équation (3.8). On transcrit cette somme en remplaçant n par I et k par K, A_n par $A[I]$ et c_n par $C[I]$. Les fonctions `F1`, `F2`, `F3` correspondent au 3 cas énumérés ci-dessus. Le nid de boucles obtenu est illustré par la figure 3.3.

```
Do I = 0 , N
   Do K = 1 , N-I
       A[I] = A[I] + F1(C[K],C[I+K])
   Do K=0 , I
       A[I] = A[I] + F2(C[K],C[I-K])
   Do K = I+1 , N
       A[I] = A[I] + F3(C[K],C[K-I])
```

FIG. 3.3 – Nid de boucles à paralléliser

L'objectif est de paralléliser la boucle en I au mieux, c'est-à-dire en minimisant les communications, tout en respectant certaines contraintes :

– L'allocation est donnée. En effet, ce problème n'étant qu'une part d'un problème plus gros, il n'est pas souhaitable d'avoir à changer l'allocation fixée par ce problème englobant. On a donc $N+1$ processeurs, $A[I]$ étant sur le processeur I, de même que $C[I]$.

– On suppose que la mémoire de chaque processeur ne peut pas contenir l'ensemble des $C[K]$. On élimine donc la solution qui serait de faire une communication globale All-To-All au départ : chaque processeur communiquant à tous les autres le $C[I]$ qu'il a en mémoire et effectuant ses calculs, ayant toutes les données dont il a besoin sur place.

On peut s'étonner de voir privilégier une solution qui ne permet pas d'utiliser autant de processeurs que l'on en dispose. Une telle solution serait néanmoins possible en couplant la parallélisation par mode proposée avec une décomposition de domaines. Pour M sous-domaines, on aurait ainsi besoin de $N.M$ processeurs, ce qui nous permet d'utiliser plus de processeurs - autant qu'on en dispose suivant le choix de M - pour notre parallélisation.

3.2.2 Première parallélisation

La première idée est de prendre le nid de boucles tel qu'il est et à chaque fois qu'un processeur a besoin d'une donnée, il y a une communication.

Calcul du nombre de communications

On se place sur le processeur I. On compte le nombre de calculs : $N-I+I+1+N-I = 2N-I+1$ Ceci nous donne, pour l'ensemble des $N+1$ processeurs un nombre total de $\frac{3N^2+5N}{2}$ calculs.
Chaque calcul demande 2 valeurs, donc $4N-2I+2$ communications, desquelles on doit retirer les utilisations de $C[I]$ qui est local. On compte combien de fois on utilise $C[I]$: cela dépend si $I \leq N/2$ (cas 1) ou si $I > N/2$ (cas 2)
– cas 1, $I \leq N/2$: on utilise $C[I]$ 2 fois dans la deuxième boucle et 1 fois dans la première, une fois dans la troisième. D'où le nombre de communications : $4N-2I-2$.
– cas 2, $I > N/2$: on n'utilise pas I dans la première boucle, ni dans la troisième. Par contre, dans la deuxième boucle sur K, on l'utilise 2 fois. D'où le nombre de communications : $4N-2I$.
Soit pour l'ensemble des processeurs, un nombre de communications totale de $3N^2+N$ (à 1 près, suivant si N est pair ou impair).
On remarque enfin que chaque processeur a dans sa mémoire 3 $C[I]$ simultanément, en comptant celui qui lui est local.

Génération des communications

Nous nous situons dans l'hypothèse où les communications de données sont exprimés par des primitives SEND/RECEIVE. Chaque processeur k détenteur d'une donnée locale `C[K]` doit donc déterminer les processeurs amenés à utiliser ce `C[K]` et quand cette utilisation doit avoir lieu (on rappelle que l'on s'interdit une communication ALL-to-ALL). Ce problème revient à déterminer les ensembles d'utilisation de chaque $C[K]$. A cette fin, on va utiliser OPERA, ce qui nous amène à réecrire le nid de boucles sous forme d'assignation unique, voir Fig. 3.4

La visualisation d'une partie des dépendances pour ce nid de boucles est visible figure 3.5, copie d'écran de ce que retourne OPERA. On a choisi de prendre $N = 7$ (soit un

```
Do I = 0 , N
    Do K = 1 , N-I
        A[I,K] = A[I,K-1] + F1(C[K,0],C[I+K,0])
    Do K=N-I+1 , N+1
        A[I,K] = A[I,K-1] + F2(C[K-N+I-1,0],C[N-K+1,0])
    Do K = N+2 , 2N-I+1
        A[I,K] = A[I,K-1] + F3(C[K-N-1+I,0],C[K-N-1,0])
```

FIG. 3.4 – Mise en assignation unique du nid de boucle

calcul destiné à être parallélisé sur 8 processeurs) pour la visualisation, et de montrer les points d'utilisation de la variable $C[3]$. Notons aussi que toutes les variables doivent avoir la même dimension ; on a donc été amené à ajouter une dimension au tableau C dans la formulation sous assignation unique précédente. On a utilisé un "truc" consistant à remplacer $C[K]$ par $C[K,0]$. Ainsi, sur la figure 3.5, les points de la première colonne peuvent être considérés comme les points avec lesquels le tableau C est aligné.

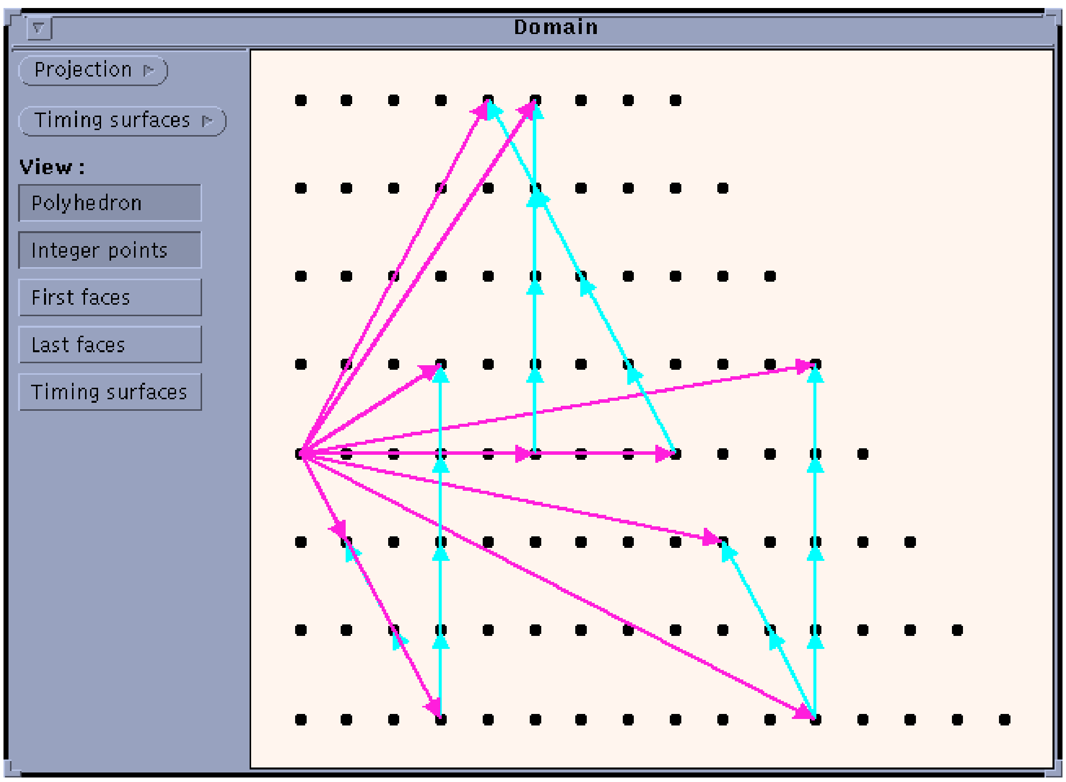


FIG. 3.5 – Ensembles d'utilisations de $C[3]$ avec 8 processeurs. Verticalement : les processeurs (donc I). Horizontalement : K

L'utilisation brute d'OPERA pour générer les communications pose certains problèmes. En particulier, il n'y a pas de génération globale du code, mais seulement une approche boucle par boucle et par instance de tableau. Une fois qu'on a choisi de paralléliser la boucle sur I, on peut générer un code pour la boucle

```
Do K = 1, N-I
  A[I,K] = A[I,K-1] + F1(C[K],C[I+K])
```

pour, par exemple, $C[K]$: on sait alors quel processeur doit communiquer, à qui et quand ce qui correspond à un $C[K]$; pareil pour $C[I+K]$. De la même façon, on peut s'occuper de la boucle

```
Do K=N-I+1 , N+1
  A[I,K] = A[I,K-1] + F2(C[K-N+I-1],C[N-K+1])
```

mais cela indépendamment de la première boucle. On peut facilement générer des directives de réception (on sait de quelles valeurs on a besoin). Par contre, on ne peut générer les directives d'envoi aussi facilement car il est possible qu'il faille faire un SEND dans la première boucle, pour un calcul qui a lieu dans la deuxième. Comme d'autre part les boucles générées ont des bornes différentes, on ne peut pas fusionner facilement les boucles obtenues.

On doit donc procéder différemment, par exemple, en ayant une structure de données qui à chaque K (donc à chaque pas de temps) associe les processeurs auquel un processeur doit envoyer une donnée. OPERA permet de construire facilement cela à la main, grâce à la visualisation des ensembles d'utilisation. On visualise chaque ensemble, pour un $C[I]$ donné, et on regarde quel(s) processeur(s) l'utilise(nt) à l'étape K.
Un exemple pour $N = 7$, et pour le processeur $I = 3$:

Etape k	Processeur(s)	Etape k	Processeur(s)
1	2	9	2
2	1	10	1
3	0,0,1,2,3,4	11	0,0,1,2,3,4
4	7	12	-
5	3,4,5,6,6,7	13	-
6	5	14	-
7	4		
8	3		

obtenu à partir de l'image 3.5.

Ceci aboutit au code final de la Fig 3.6 dans lequel `Nb_envoi[I,K]` est le nombre d'envois que doit effectuer le processeur I à l'étape K et `Proc[I,K,.]` est la liste des processeurs auquel le processeur I doit envoyer $C[I]$ à l'étape K. Il faut prévoir les cas où la donnée est sur place car alors aucune communication n'est nécessaire, ou le cas où $I - K = K$ car on communique 2 fois la même donnée, ce qui est inutile. L'idée que le programmeur doive expliciter lui-même les communications dans les programmes est peu élégante. Néanmoins, on pourrait imaginer implémenter les algorithmes sous-jacents à OPERA pour calculer ces structures de données. Dans la mesure où on peut obtenir une visualisation des communications, il est clair qu'on peut obtenir cette même information sous une autre forme.
Un autre façon de générer les structures `Proc[.,.,.]` et `Nb_envoi[.,.]` est d'exécuter un nid de boucles identique qui ne fait aucun calcul mais stocke les communications qui seraient effectuées. Il s'agit donc d'un pré-calcul des communications. Cette façon de procéder est d'autant plus rentable qu'au cours du programme original, la boucle est

utilisée à chaque pas de temps et que la faire tourner sans calcul n'occasionne pratiquement pas de surcoût.

```
Do I = 0 , N
    Do K = 1 , N-I
        Do j=1, Nb_envoi[I,K]
            Send C[I] to  Proc[I,K,j]
        Receive C[K]
        Receive C[I+K]
        A[I,K] = A[I,K-1] + F1(C[K],C[I+K])
    Do K=N-I+1 , N+1
        Do j=1, Nb_envoi[I,K]
            Send C[I] to  Proc[I,K,j]
        Receive C[K-N+I-1]
        Receive C[N-K+1]
        A[I,K] = A[I,K-1] + F2(C[K-N+I-1],C[N-K+1])
    Do K = N+2 , 2N-I+1
        Do j=1, Nb_envoi[I,K]
            Send C[I] to  Proc[I,K,j]
        Receive C[K-N-1+I]
        Receive C[K-N-1]
        A[I] = A[I,K-1] + F3(C[K-N-1+I],C[K-N-1])
```

FIG. 3.6 – Pseudo-code parallélisant la boucle Fig.3.3

3.2.3 Une première amélioration

Présentation

La solution qui vient d'être présentée s'avère coûteuse en communications et nous allons chercher à l'optimiser en restructurant les boucles pour profiter de la commutativité de l'addition. Les calculs que nous avons effectués pour évaluer le nombre de communications nous ont montré que $C[I]$ par exemple est utilisé 2 fois dans la deuxième boucle. On va donc voir si il est possible de réunir ces calculs pour éviter d'avoir à communiquer 2 fois la même donnée. En faisant un changement de variables $K = K - I$ dans la troisième boucle on obtient :

```
Do I = 0 , N
    Do K = 1 , N-I
        A[I] = A[I] + F1(C[K],C[I+K])
    Do K=0 , I
        A[I] = A[I] + F2(C[K],C[I-K])
    Do K = 1 , N-I
        A[I] = A[I] + F3(C[I+K],C[K]).
```

En s'intéressant aux données accédées, on voit que la première et la troisième boucle utilisent les mêmes données. On remarque que c'est dû au choix fait en 3.2.1, de ne pas faire intervenir $C[0]$ dans la première somme, mais dans la deuxième. On peut finalement fondre les boucles 1 et 3 en une seule. De même pour la deuxième boucle que l'on peut couper en deux (*loop-splitting*) :

```
Do K=0, [I/2]-1
    A[I] = A[I] + F2(C[K],C[I-K])
Do K=[[I/2]]+1 , I
    A[I] = A[I] + F2(C[K],C[I-K])
Si I est pair
    A[I] = A[I] + F2(C[I/2],C[I/2])
```

où la notation `[I/2]` dans les bornes des boucles désignent la partie entière supérieure et `[[I/2]]` la partie entière inférieure. En appliquant le changement de variable $K = I - K$ la deuxième boucle devient :

```
Do K=0 , [I/2]-1
    A[I] = A[I] + F2(C[I-K],C[K])
```

Il apparaît clairement que les deux demi-boucles utilisent les mêmes données. On peut réécrire le nid de boucles complet, ce qui est présenté Fig. 3.7 (où `Mod` est la fonction modulo).

```
Do I = 0 , N
    Do K = 1 , N-I
        A[I] = A[I] + F1(C[K],C[I+K])
        A[I] = A[I] + F3(C[I+K],C[K])
    Do K=0 , [I/2]-1
        A[I] = A[I] + F2(C[K],C[I-K])
        A[I] = A[I] + F2(C[I-K],C[K])
    Si (I Mod 2) = 0
        A[I] = A[I] + F2(C[I/2],C[I/2])
```

FIG. 3.7 – Nid de boucles après une première transformation

Par rapport à la formulation précédente, on divise le nombre de communications par 2, donc $2N-I+1$, par processeur, autant que de calculs, ou encore de l'ordre de $\frac{3}{2}N^2$ communications au total. Et comme précédemment, chaque processeur a 3 $C[I]$ simultanément en mémoire.

La génération des communications

Comme précédemment, on peut déterminer les communications en écrivant le programme sous forme d'assignation unique.
`F4` représente l'application des fonctions précédentes `F1` et `F3` au sein d'une seule fonction. De la même façon que dans le cas précédent, on peut générer les communications. Il est plus intéressant de s'interroger sur la façon dont on pourrait obtenir «automatiquement»

```
Do I = 0, N
    Do K = 1 , N-I
        A[I,K] = A[I,K-1] + F4(C[K],C[I+K])
    Do 2K = 2N-2I+2, 2N-I+2
        A[I,K] = A[I,K-1] + F2(C[K-N+I-1],C[N+1-K])
```

FIG. 3.8 – Mise en assignation unique du nid de boucles 3.7

cette solution. Cela nécessite tout d'abord de se rendre compte que pour un I donné, deux variables C sont utilisées simultanément, et cela à deux pas de temps différents (pour 2 K différents).

L'outil graphique OPERA permet de se rendre compte de ce phénomène, comme l'illustre la Fig. 3.9. La sélection des bonnes données à visualiser met en évidence un schéma de communications identique : sur la figure, on voit que $C[5]$ est utilisé 2 fois sur les mêmes processeurs.

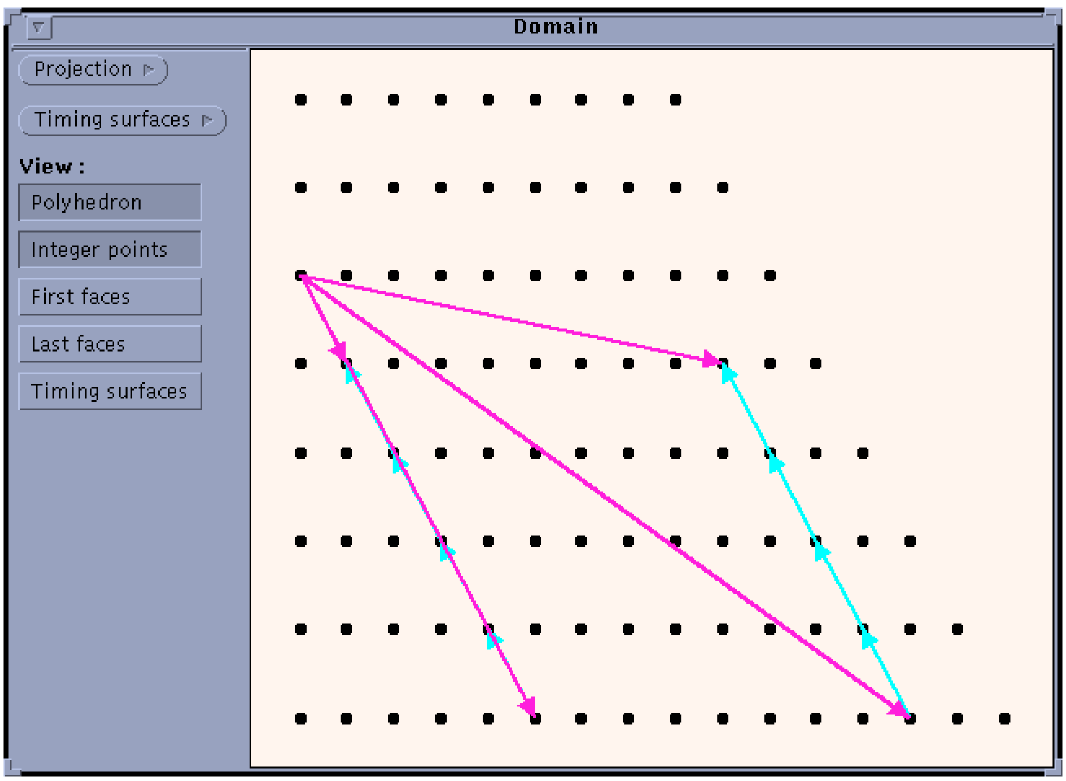


FIG. 3.9 – Ensemble d'utilisation de 2 occurences de $C[5]$

La génération automatique de cette solution ne semble pas possible à partir des outils dont on dispose. Il a été nécessaire de modifier l'ordre d'exécution des calculs pour l'obtenir, ce qui est rendu possible par la commutativité de l'addition. Cependant, la formulation sous forme d'assignation unique de la figure 3.4 nous oblige à donner des dépendances qui ne sont plus respectées par la solution proposée, et le mode de fonctionnement des outils est bien évidemment de respecter les dépendances pour garder la sémantique du programme original.

3.2.4 Solution minimisant le nombre de communications

La boucle précédente fait encore des communications «en trop». On remarque que le minimum de communications de chaque processeur est N. En effet, si on reprend la formulation de départ (Fig. 3.3), la deuxième boucle utilise les données $C[O], \cdots, C[I]$, et la troisième boucle $C[I+1], \cdots, C[N]$. Toutes les données sont donc utilisées au moins un fois par chaque processeur.

Pour obtenir ce nombre minimum de communications, il «suffit», une fois qu'on a une donnée localement, de l'utiliser dans tous les calculs où elle apparaît. Le problème qui se pose est que l'exécution de ces calculs nécessite d'autres données qui entraînent d'autres calculs, etc... Il ne faut pas que cela se ramène à une solution où on aurait besoin de toutes les données simultanément sur chaque processeur.

Pour voir ce qu'il en est, on va transformer les nids de boucles en 3 temps :
- étude de la première boucle précédente
- étude de la deuxième boucle précédente
- fusion de ces deux boucles si nécessaire.

La première boucle

Rappel :

```
Do K = 1 , N-I
    A[I] = A[I] + F1(C[K],C[I+K])
    A[I] = A[I] + F3(C[I+K],C[I])
```

Y'a-t-il des données qui sont réutilisées dans cette boucle ? Sachant que `C[I+K]` est utilisé au pas K, le pas $I+K$ est-il également un pas de cette boucle ? C'est le cas si $I+K = L$, où $L \leq N-I$, c'est-à-dire si $I+K \leq N-I$. On calcule alors : `A[I] + F1(C[I+K],C[2I+K])` et `A[I] + F3(C[2I+K],C[I+K])`. Le raisonnement peut continuer si $K + 2I \leq N - I$, auquel cas on aura besoin de $C[K + 3I]$, etc. Pour un K donné, on est donc amené à écrire :

```
J = K
Tant que J <= N-I
    A[I] = A[I] + F1(C[J],C[I+J])
    A[I] = A[I] + F3(C[I+J],C[J])
    J = J+I
```

Reste un dernier travail à accomplir : changer la boucle sur K pour ne pas refaire des calculs effectués par la boucle `Tant que` rajoutée. Pour cela, il suffit de limiter la boucle à I. Si $N - I > I$, alors les valeurs de K supérieures à I sont de la forme $JI + L$ (avec $L \leq I$), et donc sont parcourues par la boucle rajoutée : il n'y a pas de trous. Si $N-I < I$, on ne fait pas de calcul en trop, car l'indice K tel que $N - I < K < I$ est supérieur à $N - I$ et donc on n'effectue pas de calcul, car on ne rentre pas dans la boucle `Tant que`.

Par contre, on est obligé de faire un test (pour voir si $J < N - I$), ce qui ajoute un surcoût, qui doit être négligeable par rapport au temps de communications épargné. La boucle est illustrée par la Fig. 3.10

```
Do K=1, I
  J = K
  Tant que J <= N-I
      A[I] = A[I] + F1(C[J],C[I+J])
      A[I] = A[I] + F3(C[I+J],C[J])
      J = J+I
```

FIG. 3.10 – Réécriture de la boucle faisant intervenir `F1` et `F3` pour réutiliser au mieux les variables communiquées

La deuxième boucle

```
Do K=0 , [I/2]-1
    A[I] = A[I] + F2(C[K],C[I-K])
    A[I] + A[I] + F2(C[I-K],C[K])
Si (I Mod 2) = 0
    A[I] = A[I] + F2(C[I/2],C[I/2]
```

Ici, il n'y a pas de réutilisation des variables. En effet, il faudrait avoir $L = I - K$ et $L < \lceil I/2 \rceil - 1$ ce qui nous amène à : $I - K < \lceil I/2 \rceil - 1$, soit $K > \lfloor I/2 \rfloor + 1$, ce qui est exclu, car on ne va que jusqu'à $\lceil I/2 \rceil - 1$.
Cette boucle reste donc telle qu'elle est.

La fusion des 2 boucles

Il est à noter qu'une fusion des 2 boucles est nécessaire, car il y a des valeurs identiques de $C[K]$ utilisées dans chacune d'elles, si $I > 0$ et $N > 1$.
Deux problèmes se posent :
– les bornes de ces 2 boucles ne sont pas les mêmes ;
– peut-on réutiliser les $C[I - K]$ à la façon des $C[I + K]$?
Comme la boucle sur K du premier cas étudié commence à 1, on traitera le cas $K = 0$ à part. Comme la deuxième boucle ne va que jusqu'à $\lceil I/2 \rceil - 1$, on prend cette boucle comme boucle originale, dans laquelle on insère les calculs de la première boucle. On s'occupera ensuite de rajouter les calculs laissés de côté.

```
Do K=0 , [I/2]-1
    A[I] = A[I] + F2(C[K],C[I-K])
    A[I] = A[I] + F2(C[I-K],C[K])
    J = K
    Tant que J <= N-I
        A[I] = A[I] + F1(C[J],C[I+J])
        A[I] = A[I] + F3(C[J+I],C[J])
```

```
        J = J+I
```

Il manque bien évidemment des calculs, vu que la boucle sur K ne va que jusqu'à $\lceil I/2 \rceil - 1$ alors que le boucle originale de la Fig. 3.10 va jusqu'à I. Pour combler ce vide, on peut songer à réutiliser $C[I-K]$. Si $I-K \leq N-I$, alors on peut calculer : `A[I] = A[I] + F1(C[I-K],C[2I-K])` et `A[I] = A[I] + F3(C[2I-K],C[I-K])` et réutiliser $C[2I-K]$ avec $C[3I-K]$, etc... Il faut noter que ces calculs n'interfèrent pas avec ceux dejà effectués (c'est-à-dire qu'on ne fait pas de calculs en double). En effet, supposons que les calculs se superposent :

$\exists \quad L \leq \lceil I/2 \rceil - 1, \quad n \in N \quad | \quad nI + L = mI - K$; alors $K = (m-n)I - L$ donc $K \geq (m-n)I - \lceil I/2 \rceil - 1$.

Comme $K \leq \lceil I/2 \rceil - 1$, on doit avoir $m - n < 2$.

- Si $m-n < 0$ alors $K < 0$, impossible.
- Si $m-n = 0$ alors $K = -L < 0$ impossible car $K \geq 0$.
- Si $m-n = 1$, $K = \lfloor I/2 \rfloor + 1$ exclus.

On s'aperçoit également qu'il n'y aucun problème en constatant que quand I varie de 1 à $\lceil I/2 \rceil - 1$, $I-K$ varie de $I-1$ à $\lfloor I/2 \rfloor + 1$; on récupère alors les calculs de la première boucle qu'on avait mis de côté pour écrire une boucle ayant $\lceil I/2 \rceil - 1$ comme borne. Ceci confirme qu'il n'y a pas de calculs qui se répètent et prouve qu'on fait bien tous les calculs, mis à part le cas $K = I$ traité en même temps que le cas $K = 0$. Le cas $K = I/2$ est aussi traité à part. On aboutit finalement au programme de la Fig. 3.11

Dans ce pseudo-code, on peut remplacer les `Tant que` par des `Do`, avec des bornes faisant intervenir la division entière de 2 entiers.

D'autre part, le nombre maximum de $C[K]$ sur un processeur est 4 : $C[I]$ sur le processeur I, $C[nI+K]$, $C[(n+1)I+K]$, et $C[I-K]$, ce qui, on l'espère est raisonnable au vu de la taille des $C[K]$.

Génération des communications

Ici, le problème de la génération des communications vient du fait qu'on fait une boucle avec un pas de I, I qui est lui-même un indice de boucle, ce qui signifie qu'on utilise des éléments de la forme $C[IJ+K]$, qui n'est pas une fonction affine des indices de boucles. On sort donc du cadre du modèle polyèdrique : OPERA n'est alors d'aucune aide pour générer les communications.

On peut néanmoins toujours appliquer ce qui a été dit pour les boucles précédentes c'est-à-dire faire tourner une boucle à vide qui génère `Nb_Envoi` et `Nb_Proc`.

```
I=0
A[I] = A[I] + F2(C[0],C[0])
Do K=1 , N
    A[I] = A[I] + F2(C[K],C[K])
Do I = 1 , N
    K = 0
    A[I] = A[I] + F2(C[0],C[I])
    A[I] = A[I] + F2[C[I],C[0])
    J = I
    Tant que J <= N-I
        A[I] = A[I] + F1(C[J],C[J+I])
        A[I] = A[I] + F3(C[J+I],C[J])
        J = J+I
    Do K = 1 , [I/2]-1
        A[I] = A[I] + F2(C[K],C[I-K])
        A[I] = A[I] + F2(C[I-K],C[K])
        J = K
        Tant que J <= N-I
            A[I] = A[I] + F1(C[J],C[J+I])
            A[I] = A[I] + F3(C[J+I],C[J])
            J = J+I
        J = I-K
        Tant que J <= N-I
            A[I] = A[I] + F1( C[J],C[J+I])
            A[I] = A[I] + F3(C[J+I],C[J])
            J = J+I
    Si (I Mod 2) = 0
        A[I] = A[I] + F2(C[I/2],C[I/2])
        J = I/2
        Tant que J <= N-I
            A[I] = A[I] + F1(C[J],C[J+I])
            A[I] = A[I] + F3(C[J+I],C[J])
            J = J+I
```

Fig. 3.11 – Pseudo-code obtenu après réécriture et fusion des boucles

3.3 Obtention automatique de la solution en liaison avec les techniques d'optimisation des accès à la mémoire

3.3.1 Formalisation

On peut formaliser les diverses transformations que nous avons effectuées en nous appuyant sur la notion d'ensemble, à la manière de [23]. Il s'agissait dans cet article d'optimiser les accès à la mémoire dans un nid de boucles pour améliorer la localité spatiale.

Reprenons le nid de boucles de départ (Fig. 3.3), en ne gardant que les 2 boucles internes sur K qui peuvent être fusionnées. On la réécrit en faisant varier l'indice de boucle K sur des domaines disjoints, voir figure 3.12.

```
DO I=1,N
  DO K=1, N-I
     A[I] = A[I] + F1(C[K], C[I+K])
  DO K=N-I+1, 2N-2I
     A[I] = A[I] + F3(C[K-N+2I], C[K-N+I])
```

FIG. 3.12 – Nid de boucles simplifié, K variant sur des domaines disjoints

Il s'agit maintenant d'écrire $P(I_0, I)$, ensemble des itérations qui utilisent un $C[I_0]$ donné pour une itération I donnée. Notons $P_1(I_0, I)$ l'ensemble des itérations qui utilisent $C[I_0]$ lors de l'exécution de la première boucle et $P_2(I_0, I)$ l'ensemble des itérations qui utilisent $C[I_0]$ lors de l'exécution de la deuxième boucle. Clairement, $P(I_0, I) = P_1(I_0, I) \bigcup P_2(I_0, I)$.

On a :

$$\begin{aligned} P_1(I_0, I) &= \{K \in D_1 | K = I_0\} \bigcup \{K \in D_1 | I + K = I_0\} \\ P_2(I_0, I) &= \{K \in D_2 | K - N + 2I = I_0\} \bigcup \{K \in D_2 | K - N + I = I_0\} \end{aligned}$$

où D_1 est le domaine de la première boucle et D_2 celui de la deuxième :

$$\begin{aligned} D_1 &= \{K | 1 \leq K \leq N - I\} \\ D_2 &= \{K | N - I + 1 \leq K \leq 2N - 2I\}. \end{aligned}$$

Finalement on obtient :

$$
\begin{aligned}
P(I_0, I) \;=\; & \{1 \le K \le N - I | K = I_0\} \\
& \bigcup \{1 \le K \le N - I | K = I_0 - I\} \\
& \bigcup \{N - I + 1 \le K \le 2N - 2I | K = I_0 + N - 2I\} \\
& \bigcup \{N - I + 1 \le K \le 2N - 2I | K = I_0 + N - I\}
\end{aligned}
$$

On voit ici apparaître le pas de I qui nous a servi ultérieurement à réécrire le nid de boucles en nous servant de la réutilisation d'un $C[I_0]$ donné entre une boucle et une autre. Ici, on voit qu'un $C[I_0]$ est utilisé à la fois à l'étape $I_0 - I$ et I_0, (pourvu que l'on reste dans le domaine valide).
On peut réécrire $P(I_0, I)$ en se concentrant sur I_0

$$
\begin{aligned}
P(I_0, I) \;=\; & \{1 \le I_0 \le N - I | K = I_0\} \\
& \bigcup \{1 + I \le I_0 \le N | K = I_0 - I\} \\
& \bigcup \{I + 1 \le I_0 \le N | K = I_0 + N - 2I\} \\
& \bigcup \{1 \le I_0 \le N - I | K = I_0 + N - I\}
\end{aligned}
$$

Si à partir de cela on calcule les données `C` accédées par les deux boucles internes, on voit que ce sont les mêmes, d'où l'idée de les réunir en une seule, comme nous l'avons fait dans la section 3.2.3.

Cette approche du problème nous permet de modéliser et calculer les accès aux données afin de nous aider à paralléliser au mieux. Cependant, on ne peut appliquer l'idée complète énoncée sans [23] dans notre cas, puisque qu'il faut, pour que la technique soit valide, que la référence à une donnée soit unique. Ce n'est pas le cas, puique la première boucle référence deux fois le tableau C. Pour voir où se situe le problème, remarquons que cela consisterait dans notre cas à parcourir une boucle sur I_0 et à faire tous les calculs qui utilisent le $C[I_0]$ correspondant : on connaît ces calculs grâce à l'ensemble $P(I_0, I)$. Mais ceci nous amènerait à faire 2 fois certains calculs, puisqu'avec $C[I_0]$, on utilise $C[I_0 - I]$, pour un calcul qui a déjà été effectué lors des calculs mettant en jeu $C[I_0 - I]$.

3.3.2 Transformation des accès aux données

Par contre, on peut ré-écrire la boucle en ayant fait la transformation pour rassembler les deux boucles en une seule et en la modélisant de façon simplifiée sous la forme de la figure 3.13. On peut alors se servir de la technique décrite dans [80] pour opérer la transformation qui nous donnera un ordre d'accès aux données similaire à celui que nous avons obtenu à la main. La technique énoncée, qui vise à optimiser la localité dans l'accès aux données est parfaitement transposable à notre problème de communications.

```
Do I = 0, N
   Do J = 1, N-I
      A[I] = A[I] + F4(C[J], C[J+I])
```

FIG. 3.13 – Nid de boucles à optimiser

Comme nous avons vu que les dépendances sur `A` n'en étaient pas vraiment au vu de la commutativité de l'addition, l'hypothèse de non-dépendances portée par les boucles est respectée. Il faut ensuite nous assurer que la matrice d'accès à la données `C` est inversible. Ce n'est pas le cas avec le nid tel qu'il est où la matrice d'accès à la première référence de `C` est donnée par :

$$\begin{pmatrix} 0 1 \end{pmatrix} \tag{3.9}$$

Pour satisfaire cette hypothèse, on peut soit ajouter une dimension à la variable `C`, en ajoutant par exemple `I` : ce n'est pas dommageable car on parallélise suivant `I`. Mais on peut plus simplement se séparer de la boucle sur `I` qui devient inutile pour notre réflexion. On se base désormais sur la boucle très simplifiée 3.14 en étudiant les accès à la variables `C`.

```
Do J = 1, N-I
   A[I] = A[I] + F4(C[J], C[J+I])
```

FIG. 3.14 – Boucle sur laquelle on va procéder à l'optimisation des accès

On a deux “matrices” et vecteurs d'accès pour chaque utilisation de `C` :

$$\begin{gathered} R_1 = (1), t_1 = (0) \\ R_2 = (1), t_2 = (I) \end{gathered}$$

I et N étant tous deux des paramètres.
On peut ensuite écrire qu'il y a réutilisation des données pour deux itérations j_a et j_b si

$$\begin{gathered} R_1 j_a + t_1 = R_2 j_b + t_2 \\ R_2^{-1} R_1 j_a + R_2^{-1} t_1 - t2 = j_b \\ T j_a + t = j_b \end{gathered}$$

Dans notre cas, cette condition s'écrit

$$j_a - I = j_b$$

c'est-à-dire

$$\begin{gathered} T j_a + t = j_b \\ T = (1) \;\text{ et }\; t = (-I) \end{gathered}$$

et on note R la relation de réutilisation entre 2 références.

Il nous faut ensuite découper l'ensemble des itérations suivant le type de réutilisation qu'elles induisent. On définit tout d'abord l'ensemble P_1 des itérations du domaine D d'itérations qui font appel à des données qui seront réutilisées par la suite, puis l'ensemble P_2 des itérations qui font appel à des données utilisées lors des itérations de P_1. Reste l'ensemble L des itérations qui font appel à des données qui ne sont jamais re-référencées. Ensuite, on peut définir C, ensemble des itérations qui appartiennent à la fois à P_1 et P_2 (donc $C = P_1 \bigcap P_2$). Pour terminer le partitionnement, reste à garder les points de P_1 qui ne sont pas dans C, et ceux de P_2 qui ne sont pas dans C, $D_1 = P_1 - C$ et $D_2 = P_2 - C$.

Une fois ces ensembles calculés, reste à les partitionner à nouveau de façon à parcourir les itérations dans l'ordre optimal, c'est-à-dire en réutilisant au mieux les données. On va donc considérer :

- $DC1$: ensemble des itérations de D_1 qui sont directement liées à des itération de D_2 ;
- C_1 : ensemble des itérations de C directement liées à des itérations de C ;
- $DC2$: ensemble des itérations de D_1 liées à une itération de C, liée elle-même à une itération de D_2 (donc itérations de D_1 liées à D_2 par R^2) ;
- C_2 : itérations de C liées à elle-même par R^2 ;
- ...
- DC_k : ensemble des itérations de D_1 liées à D_2 par R^k ;
- C_k : itérations de C liées à elle-même par R^k ;

Il est bien évidemment nécessaire de savoir arrêter le partitionnement lorsque qu'on a complètement partitionner D. Il est possible de s'arrêter si l'application de R $kmax$ fois se résume à l'identité, c'est-à-dire s'il existe k tel que :

$$\begin{pmatrix} T & t \\ 0 & 1 \end{pmatrix}^k = Id$$

Ce n'est pas notre cas, où on a

$$\begin{pmatrix} T & t \\ 0 & 1 \end{pmatrix}^k = \begin{pmatrix} 1 & -I \\ 0 & 1 \end{pmatrix}^k = \begin{pmatrix} 1 & -kI \\ 0 & 1 \end{pmatrix}$$

La seule solution reste donc de fixer un k maximum en espérant que cela sera suffisant pour obtenir la formulation réellement optimale. C'est un peu dommage, car dans notre cas, nous sommes capables de savoir quelle est la longueur de la "chaîne" de réutilisation. La boucle qui utilise les toutes les données avec un pas de I est :

```
Do K = 1,I
   J = 0
   While K+JI <= N-I
      A[I] = A[I] + F4(C[K+JI],C[K+(J+1)I]
```

et donc on a $J = \lfloor \frac{N-I-K}{I} \rfloor$ itérations pour chaque K. Ceci nous permet néanmoins de calculer une borne à donner pour $kmax$ afin d'atteindre l'optimalité.

Afin de générer le code correspondant, il faut parcourir les ensemble C_k et DC_k, ce qui se fait très bien grâce à la Polylib [90] qui permet de manipuler des polyèdres paramétrés.

3.3.3 Discussion et conclusion

La méthode précédente permet de réordonner les calculs de façon à ce que les accès aux données soient optimisés. On utilise ici la technique présentée dans [80] pour minimiser les communications induites. On remarque néanmoins que la formulation par nids de boucles de pas I est plus lisible que la formulation que nous obtiendrions par la technique précédente qui reviendrait à dérouler des boucles que nous arrivons à décrire par 3 nids imbriqués.

Néanmoins, un travail récent [79] s'est préocuppé de l'optimisation des données dans un cas particulier du cas précédent : les opérations stencil, c'est-à-dire où les accès aux données sont des translations, ce qui est notre cas : accès en $K+I$, donc translation de I. Le code obtenu est similaire au nôtre : ajout d'une boucle interne qui permet de parcourir les données afin d'obtenir de la localité temporelle.

Ces avancées montrent bien les progrès obtenus et leur possible utilisation sur des problèmes proches des problèmes réels. Néanmoins, l'obtention automatique de la solution "optimale" demande de mettre en œuvre diverses techniques. Ainsi, il a tout d'abord fallu détecter l'aspect de réduction du calcul, qui malgré des dépendances permet d'en modifier l'ordre. Puis, on a dû réécrire une boucle pour qu'elle soit parcourue en sens inverse, puis mis en œuvre la fusion de deux boucles, puis couper une boucle en deux pour ensuite fusionner les 2 morceaux en une seule. Ces transformations sur les boucles pour améliorer la localité des données ont été décrites dans [78] par exemple qui donne un modèle permettant de choisir des transformations utiles, mais elles ne permettent pas à elles-seules d'aboutir à la solution finale. Inversement, le modèle utilisé à partir du nid simplifié pour aboutir au meilleur parcours ne permet pas de faire les étapes préliminaires qui portent sur plusieurs boucles. Ainsi, ces diverses techniques ne sont nullement exclusives les unes des autres mais leur utilisation concertée permet d'aboutir au meilleur résultat.

On remarquera que de toutes ces techniques sur l'optimisation de la mémoire, seules les techniques visant à améliorer la localité temporelle ont été utilisées ici. Quel peut être l'intérêt d'utiliser aussi la localité spatiale, c'est-à-dire le réaménagement des données en mémoire pour que des données utilisées à des dates proches soient proches aussi ? Si on se remet dans le problème initial, il faut bien voir que les données $C(K)$ sont en fait des matrices de complexes en 4 dimensions et qu'un $C(K)$ est en fait formé de trois matrices. Mettre dans un ordre ou un autre ces matrices en mémoire n'a donc que peu d'influence tel quel.

Les techniques d'optimisation spatiales seraient utiles dans la version séquentielle. Pour les calculs des fonctions `F`, on peut en effet imaginer découper les matrices C en morceaux qu'on réordonnerait en mémoire et qui participeraient d'un coup à tous les calculs où ils interviennent. Il s'agit donc là de faire une forme de tuilage, non-trivial vu que la fonction `F` est en fait une routine de 200 lignes qui fait elle-même appel à d'autres routines... Néanmoins, les opérations matricielles opérant sur des éléments toujours à la même place (opération de type addition de matrice et non pas multiplication), cela semble faisable. Si on découpe les matrices C en blocs avant le nid de boucle en I, on peut appliquer notre réécriture des boucles, les C étant des morceaux de notre C initial et faire cette fois un réordonnancement en mémoire qui serait utile.

3.4 Transformation des nids de boucles et PEI

PEI étant un outil de transformation de programme, il nous paraît bien indiqué de l'utiliser pour exprimer la transformation du nid de boucles précédent : passage du parcours d'un ensemble d'itérations à un parcours dans un ordre différent, qu'on peut aussi voir comme la transformation qui viserait à réordonner en mémoire les données pour optimiser la localité spatiale. Une première solution est apportée par [79], mais il s'agit aussi d'apporter une preuve à un niveau plus formel de la validité de la transformation effectuée en 3.2.4. Plus précisément, cette transformation consiste à réarranger les points de calculs dans un ordre différent pour que les accès à certaines variables soient modifiées. C'est une transformation de programme, exprimable avec PEI.

3.4.1 Réécriture en PEI du nid de boucles

On va partir d'un nid de boucle simplifié, qui ne garde que la partie concernant la transformation avec mise en place d'un pas de I :

```
Do I = 0, N
   Do K = 1, N-I
       A[I,K] = A[I,K-1] + F4(C[K,N+1], C[K+I,N+1])
```

Comme lorsqu'on a écrit le nid de boucles pour OPERA (section 3.2.3, figure 3.5), on ajoute ici une dimension au tableau C, mais contrairement à OPERA, où on alignait avec la première colonne, on aligne ici avec ce qui serait la dernière colonne, afin de ne pas engendrer de confusion entre les points de calculs et les données. La traduction de ce programme en PEI s'exprime par le programme de la figure 3.15.

$$
\begin{array}{l}
\{\\
\texttt{align} :: \texttt{A} = \texttt{A}\\
\texttt{align2} :: \texttt{C} = \texttt{V}\\
\texttt{A} = \texttt{calc} \rhd \ ((\texttt{A} \lhd \texttt{shift})\ /\&/(\texttt{C} \lhd \texttt{spreada})\ /\&/(\texttt{C} \lhd \texttt{spreadb}))\\
\}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{align} & = & \lambda(i,k)\ |(0{\le}i{\le}n, k=0).(i,k)\\
\texttt{align2} & = & \lambda(i,k)\ |(i=n+1, 0 \le k \le n).(i,k)\\
\texttt{calc} & = & \texttt{identity}\ \#\ \lambda(a{\cdot}b{\cdot}c).a{+}F4(b,c)\\
\texttt{shift} & = & \quad\lambda(i,k)\ |(0{\le}i{\le}n, i{\le}k{\le}n-i).(i,k-1)\\
\texttt{spreada} & = & \quad\lambda(i,k)\ |(0{\le}i{\le}n, 1{\le}k{\le}n-i).(k,n+1)\\
\texttt{spreadb} & = & \quad\lambda(i,k)\ |(0{\le}i{\le}n, 1{\le}k{\le}n-i).(i+k,n+1)
\end{array}
$$

FIG. 3.15 – Programme PEI correspondant au nid de boucles original

À partir de ce premier nid de boucles, on en a construit un deuxième (c'était l'objet de la partie 3.2.4) :

```
Do I = 0, N
    Do K = 1 , I
       Do  J = 0, [(N-I-K+1)/I]
         A[I] = A[I]+F4(C[JI+K],C[(J+1)I+K])
```

ce qui se traduit en Pei par le programme de la figure 3.16 : on a supprimé la boucle "artificielle" en J sans bien évidemment modifier l'ordre dans lequel on accède aux données (voir la figure 3.17).

```
{
align :: A = A
align2 :: C = V
A = calc ▷ ((A ◁ shift) /&/(C ◁ spread1) /&/(C ◁ spread2))
}
```

$$
\begin{array}{lcl}
\texttt{align} & = & \lambda(i,k)\ |\,(0\le i\le n, k=0)\,.\,(i,k) \\
\texttt{align2} & = & \lambda(i,k)\ |\,(i=n+1, 0\le k\le n)\,.\,(i,k) \\
\texttt{calc} & = & \texttt{identity}\ \#\ \lambda(a\cdot b\cdot c)\,.\,a{+}b{*}c{+}b{*}c \\
\texttt{shift} & = & \quad\lambda(i,k)\ |\,(0\le i\le n, i\le k\le n-i)\,.\,(i,k-i) \\
 & & \#\ \lambda(i,k)\ |\,(0\le i\le n, 1\le k\le i\,.\,(i,\lfloor n-i-k+1)/2\rfloor) \\
\texttt{spread1} & = & \quad\lambda(i,k)\ |\,(0\le i\le n, 1\le k\le n-i)\,.\,(n+1,k) \\
\texttt{spread2} & = & \quad\lambda(i,k)\ |\,(0\le i\le n, 1\le k\le n-i)\,.\,(n+1,i+k)
\end{array}
$$

Fig. 3.16 – Programme Pei correspondant à la formulation optimale du nid de boucles original

Ce programme et le précédent diffèrent sur l'ordre dans lequel on accède à la variable `A`, mais le domaine reste le même, voir la figure 3.17. Cette transformation, qui modifie les dépendances, n'est pas exprimable par Pei .

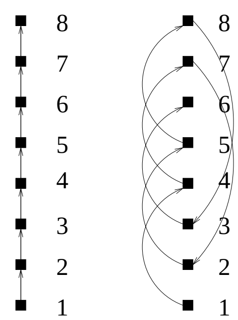


Fig. 3.17 – Dépendances entre `A[I]` (N=11, I=3), pour les programmes 3.15 et 3.16

Le programme auquel on veut aboutir est celui dans lequel on a "remis les points dans l'ordre", c'est-à-dire où les dépendances entre les points de calculs sont ceux illustrés par la figure 3.18. Notre transformation consiste donc à prendre le premier domaine unidimensionnel de la figure 3.18 et à en organiser les points dans un ordre différent pour obtenir le second domaine unidimensionnel de cette même figure : il s'agit-là d'un changement de base noté `h` pour `A`, qu'il nous faut définir.

3.4.2 Changement de base

Pour trouver ce changement de base, on se rend compte que les dépendances en A auxquelles on veut aboutir forment une chaîne qui passe par tous les points, et qu'on peut

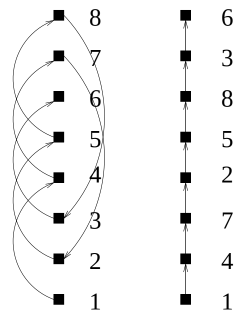


FIG. 3.18 – Dépendances entre `A[I]` pour N=11, I=3 pour le programme 3.16 et celui à construire

décomposer cette chaîne en sous-chaînes, qui correspondent aux dépendances dans la boucle en J de la formulation précédente. Sur l'exemple de la Fig. 3.18, la chaîne est donnée par :

$$1 \to 4 \to 7 \to 2 \to 5 \to 8 \to 3 \to 6$$

et les sous-chaînes :

$$\begin{aligned}&1 \to 4 \to 7\\&2 \to 5 \to 8\\&3 \to 6.\end{aligned}$$

Pour trouver quelle est la place d'un élément dans la nouvelle représentation, il suffit d'avoir la place du début de la sous-chaîne à laquelle il appartient ainsi que son rang dans la sous-chaîne.

1. Place de la sous-chaîne

   Déterminons tout d'abord le nombre d'éléments des sous-chaînes. Il y a i sous-chaînes, car le nombre de sous-chaînes est déterminé par le nombre d'itérations de la boucle en `K` pour le nid de boucles avec les pas de I, pour $n-i$ constituants. Le nombre d'éléments par sous-chaîne est donc grosso-modo $\frac{n-i}{i}$. Plus précisément, écrivons la division euclidienne de $n-i$ par i :

   $$n - i = i.q + r \quad \text{avec } r < i.$$

   q correspond alors au nombre minimal d'éléments par sous-chaîne : dans $n-i$ éléments, on loge i sous-chaînes de q éléments, et r éléments. On interprète donc r comme le nombre de points qui n'appartiennent pas à des sous-chaînes de q éléments. Les points appartenant à une même sous-chaîne sont espacés de i, par construction des sous-chaînes. Comme $r < i$, ces r points appartiennent à des sous-chaînes différentes, et par conséquent, les sous-chaînes n'ayant pas q éléments en ont $q+1$.

   Le nombre maximum d'éléments Max d'une sous-chaîne est donc :
   - $(n-i)$ div i si $(n-i) \equiv 0\ [i]$ et dans ce cas, toutes les sous-chaînes ont $(n-i)$ div i éléments.
   - $[(n-i)$ div $i] + 1$ sinon. Dans ce cas, le nombre nbc de chaînes à avoir le maximum d'éléments est le r précédent, c'est-à-dire $(n-i)$ mod i.

Et on peut écrire de façon plus générale que $nbc = (n - i)$ modB i, avec :

$$a \text{ modB } b = a \bmod b \text{ si } a \bmod b \neq 0$$
$$a \text{ modB } b = b \text{ sinon}$$

Deux cas se présentent alors :
- si k modB $i \leq nbc$ alors la formule donnant $\mathtt{h}(k)$ est de la forme : $l.Max + rang$. En effet, k appartient alors à une sous-chaîne qui contient Max éléments, et il y a éventuellement d'autres sous-chaînes de Max éléments avant, dans la nouvelle représentation.
- si k modB $i > nbc$ alors la formule est $nbc.Max + m.(Max - 1) + rang$, car alors, il y a toutes les sous-chaînes ayant Max éléments avant et peut-être d'autres avec $Max - 1$ éléments.

Déterminons l. Il s'agit en fait du nombre de sous-chaînes situées avant la sous-chaîne qui nous intéresse, soit encore le numéro de cette sous-chaîne auquel on ôte 1, si on numérote les sous-chaînes par leur ordre « d'apparition » : celle qui commence avec le premier élément porte le numéro 1, celle commençant avec le suivant porte le numéro 2, etc... jusqu'à i. On voit alors que ce numéro est donné par k modB i. En effet, on a $k = i.q + r$ avec r qui est bien le numéro de début de la sous-chaîne puisque le

$$l = (k \text{ modB } i) - 1$$

Déterminons m. On a en fait

$$k \text{ modB } i = nbc + m + 1$$

d'où

$$m = k \text{ modB } i - nbc - 1$$

2. Le rang

   On peut trouver celui-ci en connaissant le premier élément de la sous-chaîne, et qui est donné par k modB i. Ensuite, son rang $rang$ vérifie

$$k = k \text{ modB } i + i.(rang - 1)$$

   Donc

$$rang = \frac{k - (k \text{ modB} i)}{i} + 1$$

Pour définir l'inverse $\mathtt{h}^{-1}$ du changement de base : il faut remettre chaque élément à sa place initiale. Les éléments sont espacés de i en i : il suffit de savoir à quelle sous-chaîne $c(k)$ un élément appartient et son rang $r(k)$. Alors :

$$\mathtt{h}^{-1}(k) = [r(k) - 1].i + c(k)$$

Pour cela, on a besoin des mêmes données que précédemment, notamment combien de sous-chaînes ont Max éléments.

– si $k \leq Max.nbc$ alors

$$c(k) = (k-1) \text{ div } Max + 1$$
$$r(k) = k \text{ modB } Max$$

– si $k > Max.nbc$ alors

$$c(k) = nbc + (k - nbc.Max - 1) \text{ div } (Max - 1) + 1$$
$$r(k) = (k - nbc.Max) \text{ modB } (Max - 1)$$

On peut finalement écrire le changement de base h qui permet de passer d'une formulation à l'autre :

$$\begin{array}{lll}
\mathtt{h} & = & \lambda(i,k)\ |(0 \leq i \leq n, (n-i) \bmod i = 0, 1 \leq k \leq n-i, 0 \leq k \text{ modB } i \leq (n-i) \text{ modB } i, \\
 & & \quad (i, [(k \text{ modB } i) - 1].[(n-i) \text{ div } i] + \frac{k-(k \text{ modB } i)}{i} + 1\ ) \\
 & \# & \lambda(i,k)\ |(0 \leq i \leq n, (n-i) \bmod i \neq 0, 1 \leq k \leq n-i, 0 \leq k \text{ modB } i \leq (n-i) \text{ modB } i, \\
 & & \quad (i, [(k \text{ modB } i) - 1].[(n-i) \text{ div } i + 1] + \frac{k-(k \text{ modB } i)}{i} + 1\ ) \\
 & \# & \lambda(i,k)\ |(0 \leq i \leq n, (n-i) \bmod i \neq 0, 1 \leq k \leq n-i, (n-i) \text{ modB } i < k \text{ modB } i \leq i, \\
 & & \quad (i, [(k \text{ modB } i) - 1].[(n-i) \text{ div } i] + (n-i) \text{ modB } i + \frac{k-(k \text{ modB } i)}{i} + 1\ )
\end{array}$$

et son inverse h^{-1}

$$\begin{array}{lll}
\mathtt{h}^{-1} & = & \lambda(i,k)\ |(\ 0 \leq i \leq n, (n-i) \bmod i = 0, 1 \leq k \leq n-i\ )\ . \\
 & & \qquad (i,\ (k \text{ modB } [(n-i) \text{ div } i] - 1).i + (k-1) \text{ div } [(n-i) \text{ div } i] + 1\ ) \\
 & \# & \lambda(i,k)\ |(\ 0 \leq i \leq n, (n-i) \bmod i \neq 0, \\
 & & \qquad 1 \leq k \leq ((n-i) \text{ div } i + 1).((n-i) \text{ modB } i)\ )\ . \\
 & & \quad (i,\ (k \text{ modB } [(n-i) \text{ div } i + 1] - 1).i + (k-1) \text{ div } [(n-i) \text{ div } i] + 1\ ) \\
 & \# & \lambda(i,k)\ |(\ 0 \leq i \leq n, (n-i) \bmod i\ \neq 0, \\
 & & \qquad (n-i) \text{ div } i + 1 \leq k \leq [(n-i) \text{ div } i + 1].[(n-i) \text{ modB } i]\ )\ . \\
 & & \quad (i,\ (k - [(n-i) \text{ div} i + 1].[(n-i) \text{ modB } i]) \text{ modB } [(n-i) \text{ div} i] - 1].i \\
 & & \quad +(n-i) \text{ modB } i \\
 & & \quad +(k - [(n-i) \text{ div } i + 1].[(n-i) \text{ modB } i] - 1) \text{ div } [(n-i) \text{ div } i] + 1\ ).
\end{array}$$

3.4.3 Application du changement de base

En appliquant la transformation induite par la fonction `h` vue comme un changement de base au programme PEI 3.16 qui illustre le nid de boucles, on obtient le nid de boucles correspondant au programme transformé.

On veut appliquer `h` au calcul de `A`. Il suffit pour cela d'appliquer les règles de transformation :

$$\begin{array}{lll}
\mathtt{h::A} & = & \mathtt{h::(calc} \rhd ((\mathtt{A} \lhd \mathtt{shift})/\&/(\mathtt{C} \lhd \mathtt{spread1})/\&/(\mathtt{C} \lhd \mathtt{spread2}))) \\
 & = & \mathtt{calc} \rhd\ ((\mathtt{h::A} \lhd \mathtt{h} \circ \mathtt{shift} \circ \mathtt{h}^{-1})/\&/(\mathtt{h::C} \lhd \mathtt{h} \circ \mathtt{spread1} \circ \mathtt{h}^{-1}) \\
 & & /\&/(\mathtt{h::C} \lhd \mathtt{h} \circ \mathtt{spread2} \circ \mathtt{h}^{-1})).
\end{array}$$

En posant

$$\begin{aligned}
\texttt{Ab} &= \texttt{h::A} \\
\texttt{Cb} &= \texttt{h::C} \\
\texttt{shiftb} &= \texttt{h} \circ \texttt{shift} \circ \texttt{h}^{-1} \\
\texttt{spread1b} &= \texttt{h} \circ \texttt{spread1} \circ \texttt{h}^{-1} \\
\texttt{spread2b} &= \texttt{h} \circ \texttt{spread2} \circ \texttt{h}^{-1}
\end{aligned}$$

on obtient une équation de calcul

$$\texttt{Ab} = (\texttt{calc} \rhd ((\texttt{Ab} \lhd \texttt{shiftb})\texttt{/\&/}(\texttt{Cb} \lhd \texttt{spread1})\texttt{/\&/}(\texttt{Cb} \lhd \texttt{spread2})))$$

On note que le changement de base n'affecte pas le domaine de `C`, puisqu'il laisse la partie correspondant à $i = 2n + 1$ en place.
On obtient ainsi un nouveau programme permettant de faire le calcul.

```
{
align :: A = A
align2 :: C = V
A = calc ▷  ((A ◁ shiftb) /&/ (C ◁ spread1b) /&/ (C ◁ spread2b))
}
align     =  λ(i,k) |(0≤i≤n, k = 0) . (i,k)
align2    =  λ(i,k) |(i = n+1, 0 ≤ k ≤ n) . (i,k)
calc      =  identity # λ(a·b·c) . a+b*c+b*c
shiftb    =  λ(i,k) |(0≤i≤n, i≤k≤n−i) . (i,k−1)
spread1b  =  h ∘ spread1 ∘ h^-1
spread2b  =  h ∘ spread2 ∘ h^-1
```

FIG. 3.19 – Programme transformé

En l'état actuel des choses, on ne peut pas calculer automatiquement les compositions de fonctions mettant en jeu des fonctions ausi complexes que `h`, mais PEI offre malgré tout un cadre permettant de mettre une œuvre cette transformation.

3.5 Expérimentations

A la section 3.2, on a vu trois parallélisations possibles du programme, de la moins efficace théoriquement à la plus efficace. On va voir si ces gains théoriques se remarquent à un niveau pratique.

Au niveau du problème posé, on utilise un maillage semblable à celui de 3.1, avec 26 éléments spectraux contant chacun 36 points, d'où des données (matrices) de 936 éléments complexes, ce qui est finalement assez peu.

Le code a été parallélisé en utilisant MPI [83], à partir d'une version séquentielle existante. Il s'agit d'un code en FORTRAN 77 de plus de 33000 lignes. Les tests ont été effectués sur une Origin2000 avec 32 processeurs (R10000 à 195 MHZ). Les programmes étaient soumis via le système de files mis en place, avec les travaux d'autres utilisateurs, le but étant de voir les performances dans des conditions normales d'utilisation. En particulier, il nous fallait un certain nombre de pas de temps afin que de voir le comportement du programme lorsqu'il évolue, car cela est susceptible de jouer sur l'équilibrage de charge : les calculs en-dehors de la procédure parallélisée peuvent être différents pour chaque mode suivant l'evolution des calculs. Le grand nombre de pas de temps et donc la durée du programme qui en découlait rendait difficile des tests en utilisateur unique.

La parallélisation du code entier nous a amené à revoir la façon dont étaient gérées les entrées-sorties. Le fichier d'entrée, contenant les valeurs d'initialisation de chaque point du domaine pour chaque mode a été découpé pour obtenir un fichier par mode afin que chaque processeur ne lise que ce qui le concerne. De même, l'unique fichier de sortie généré a été remplacé par N fichiers, chaque mode générant un fichier de sortie afin d'éviter des synchronisations qui auraient été nécessaires pour ordonnancer les écritures dans ce fichier unique. Une phase de post-processing est alors nécessaire pour reformer un fichier unique après l'exécution à partir des divers fichiers générés, ceci afin que les résultats puissent être analysés de la même façon que l'étaient les résulats obtenus après une exécution séquentielle.

3.5.1 Comparaison exécution séquentielle - exécution parallèle

Dans un premier temps, on va étudier l'impact de la parallélisation par rapport à une exécution séquentielle. Rappelons que dans la version parallèle, on place chaque donnée en rapport avec l'harmonique n sur le processeur n. Lorsqu'on augmente le nombre d'harmoniques (donc le nombre de processeurs utilisés), on augmente la taille du problème.

La figure 3.20 nous montre les temps d'exécutions obtenus. Il s'agit d'une moyenne sur plusieurs exécutions avec 5 pas d'itérations externes et 200 internes, soit 1000 itérations. La courbe concernant le temps parallèle est elle aussi une moyenne. Nous comparerons les diverses parallélisations dans la section suivante.

On voit avec le tableau 3.2 que la parallélisation apporte un gain notable sur le temps d'exécution, avec des chiffres satisfaisants. La parallélisation s'avère efficace : on ne divise pas le temps séquentiel par le nombre de processeurs (ce serait l'idéal à atteindre) lors d'une exécution parallèle, mais la parallélisation offre des performances constantes et intéressantes : efficacité supérieure à 0.65. Le cas N = 2 est à part, puisque la parallélisation n'apporte aucun gain ! Si on analyse plus finement les performances, on se rend compte que ce n'est pas la partie parallélisée qui induit cette baisse de performance, mais le reste du calcul. L'explication plus précise se trouve dans la suite.

L'efficacité est différente de 1, et l'explication de ce phénomène n'est pas très compliquée. On a vu que les équations (3.5) et (3.6) se parallélisent sans communication. Par contre,

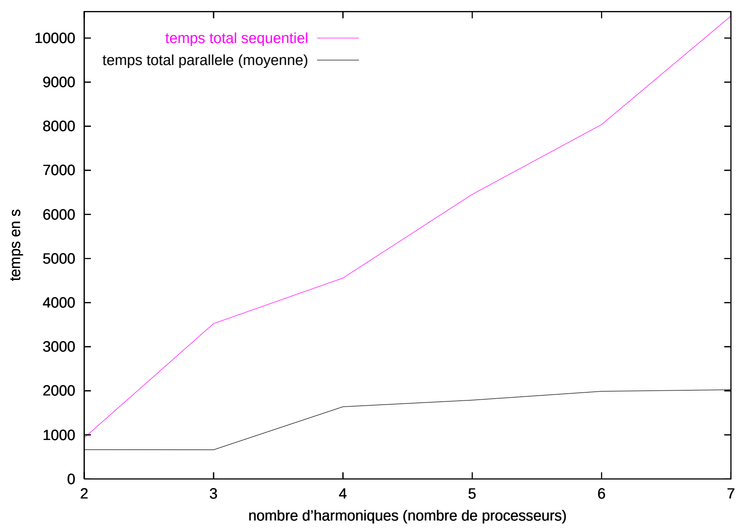


Fig. 3.20 – Temps d'exécution du programme (1000 itérations) en fonction du nombre d'harmoniques

Nombre d'harmoniques	Temps séquentiel (en s)	Temps parallèle (en s)	Efficacité $(\frac{Tseq}{Tpar*proc})$
2	932.15	921.03	0.51
3	3524.96	1530.02	0.77
4	4557.58	1700.78	0.67
5	6454.59	1866.65	0.70
6	8037.12	1994.05	0.67
7	10500.61	2107.37	0.71

Tab. 3.2 – Comparaison du temps d'exécution séquentielle et parallèle

la résolution de (3.4) nécessite des communications qui induisent un surcoût, faisant bien évidemment diminuer l'efficacité

D'autre part, il faut aussi prendre en compte les synchronisations inhérentes à ce type de programme : les boucles temporelles créent une séquentialisation. À chaque pas de temps, il faut attendre que le processeur le plus lent ait terminé pour continuer. Lors de l'exécution parallèle, le calcul le plus long impose la durée de chaque itération sur le temps, alors que lors de l'exécution séquentielle, la durée de chaque itération est la somme de temps de calculs plus ou moins longs.

Si on considère ce qui se passe dans le reste du programme (c'est-à-dire hors résolution de (3.4)), on a à faire à des résolutions de systèmes linéaires (pour résoudre l'équation (3.5) notamment), qui se font par des méthodes itératives de type gradient conjugué préconditionné. Le temps passé dans ces résolutions est important. Par conséquent, le nombre d'itérations nécessaires à la résolution des systèmes conditionne fortement le temps d'exécution de chaque itération. On va donc se pencher sur ce nombre d'itérations afin de voir quel est le comportement du programme hors la partie parallélisée.

Les tableaux de la figure 3.21 représente une partie de ce qui se passe dans le reste du programme : il s'agit d'une moyenne du nombre d'itérations par harmoniques nécessaires à la résolution des systèmes linéaires qui sont des nœuds de calculs importants dans le reste du programme, ainsi que le nombre de fois où il y a eu des résolutions de système à faire.

2 modes		
Mode	Nb iter	Nb résol
0	14.23	5711
1	17.49	6000

3 modes		
Modes	Nb iter	Nb résol
0	14.24	5793
1	17.41	6000
2	24.35	6000

FIG. 3.21 – Nombre moyen d'itérations pour une résolution de système, pour une exécution sur 2 puis 3 processeurs, et nombre de résolutions sur lequel la moyenne a été calculée

On remarque que pour l'exécution sur 2 processeurs, le mode 0 fait moins d'itérations pour résoudre un système que le mode 1, et qu'en plus, il a moins de systèmes à résoudre. Un des modes prend beaucoup plus de temps par rapport à l'autre, d'où le gain médiocre apporté par la parallélisation dans ce cas : la charge est mal équilibrée. Quand on passe à trois processeurs, on a une moins grande différence car le mode 3 est gourmand en calculs comme le mode 2, et l'efficacité s'en trouve donc naturellement augmentée.

3.5.2 Comparaisons des diverses exécutions parallèles

On peut ensuite étudier les différences au niveau de la parallélisation elle-même. Les courbes des figures 3.22 (resp. 3.23 et 3.24) montrent les temps d'exécution du programme total (resp. de la partie parallélisée - résolution de (3.4) - et le temps des communications pour cette partie). Les courbes étiquettées (1) correspondent au programme pour lequel on a parallélisé "bêtement", les courbes (2) au programme dans lequel on a divisé le volume des communications par 2 et les courbes (3) au programme pour lequel on a optimisé les communications.

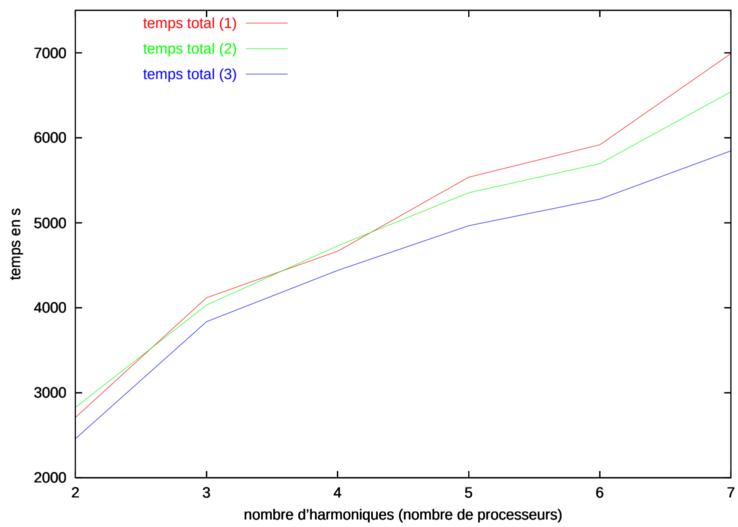


FIG. 3.22 – Temps total d'exécution du programme (3000 itérations) en fonction du nombre d'harmoniques

En ce qui concerne les temps d'exécution du programme entier, l'ordre des courbes est pratiquement celui prévu : le programme le plus long à s'exécuter est celui qui a le plus de communications ; ensuite, on trouve celui dont a divisé les communications par 2, puis enfin, le plus rapide, le programme "optimisé". Le temps de communication influe bien sur le temps d'exécution total et il est utile de prendre en compte ce critère lors de la parallélisation.

Comparons maintenant les courbes de type (1) - communications les plus nombreuses - avec celles de type (2) - communications divisées par 2. Le résultat est conforme à notre attente dans la mesure où la courbe (1) se trouve au-dessus de la courbe (2) ; la différence entre les deux courbes s'accentue d'ailleurs quand on augmente le nombre de données à communiquer (c'est-à-dire qu'on augmente le nombre de modes, et donc le nombre de processeurs). Si on se concentre sur la partie concernant les communications, on ne passe pas d'une courbe à l'autre en divisant par 2 le temps, alors qu'on a pourtant divisé le nombre des communications par 2. Ceci s'explique par ce qu'on entend par temps de communications. Il inclut les temps d'envoi (implémenter de façon à être recouvert par les temps de réception et de calculs), ainsi que les temps d'attente liés à la synchronisation entre étapes (une étape étant composée d'une passe "envoi-réception-calcul"), en particulier du temps d'attente de la réception. Ce temps est bien sûr fortement réduit lorsqu'on fait moins de communications, puisqu'on fait moins de réceptions. Si le temps passé à envoyer les données est faible comme c'est le cas ici, on ressent moins les effets de l'optimisation. C'est ce qui se passe ici.

Comparons ensuite les courbes avec les courbes (3) correspondant à la version optimale. On voit ici que les temps d'exécution sont inférieurs aux temps des autres courbes, ce

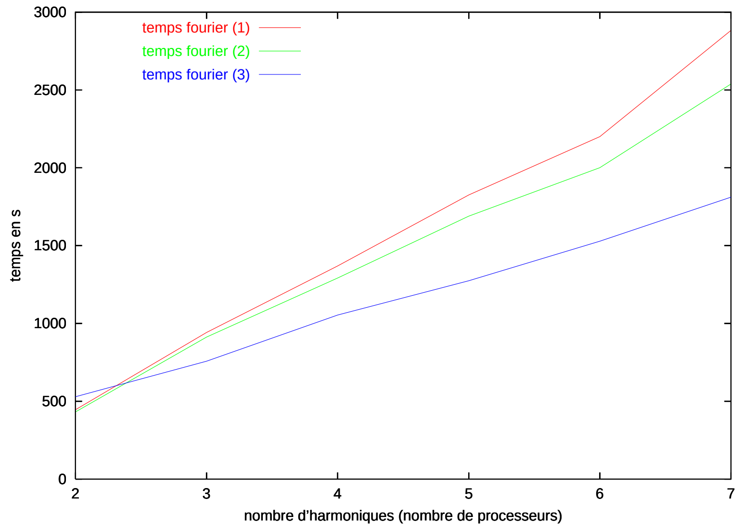


FIG. 3.23 – Temps total d'exécution de la résolution de l'équation 3.4 - (3000 itérations) en fonction du nombre d'harmoniques

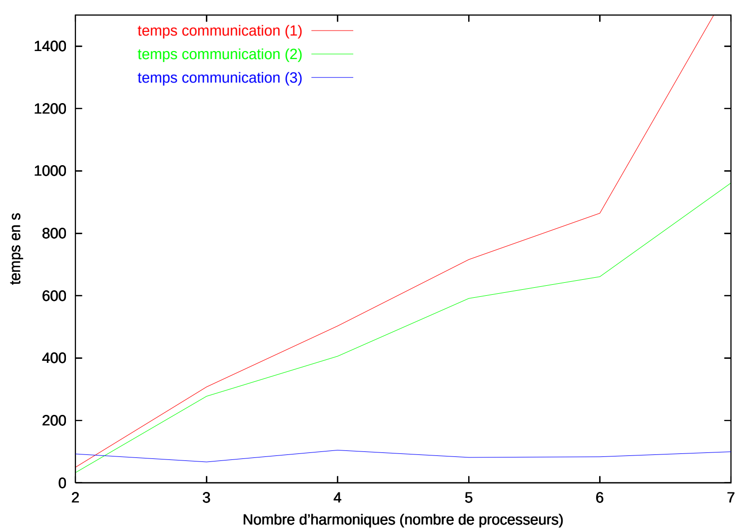


FIG. 3.24 – Temps de communication pour la partie parallélisée (3000 itérations) en fonction du nombre d'harmoniques

qui est conforme à nos prévisions. Par contre, la courbe concernant le temps des communications a une allure plate, alors que pourtant le nombre des communications n'est pas constant (il est de l'ordre de N^2 au total comme on l'a vu précédemment, ou N par processeur). On peut expliquer ce phénomène en se rappelant que ce temps concerne le temps d'attente. Dans cette version, on n'attend moins de données : la réutilisation des données qu'on a introduite fait qu'à chaque étape, on n'attend qu'une seule données au lieu de deux. D'où un temps d'attente qui se réduit forcément. Ceci explique que la courbe des temps de communications soient nettement en-dessous des autres. Le caractère quasi plat de la courbe ne peut s'expliquer que par le fait que cette version-là du programme permet une meilleure synchronisation des processus entre eux, ce qui diminue l'attente.

Si on regarde enfin l'influence du gain obtenu par les diverses parallélisations sur le programme total, il faut se reporter au tableau 3.3 qui nous montre la différence entre les temps de la procédure parallélisée (résolution de l'équation 3.4), notés "par", pour les cas (1) et (2) et la différence entre les temps d'exécution totale, notés "tot". Même chose pour les les cas (1) et (3).

nproc	par (1) - par (2)	tot (1) - tot (2)	par (1) - par (3)	tot (1) - tot (3)
2	14.49	-118.67	250.21	-83.06
3	29.90	84.63	281.07	185.04
4	77.55	-62.30	226.92	315.67
5	136.03	182.33	571.46	550.67
6	199.35	222.90	640.01	671.89
7	343.88	449.05	1144.45	1071.32

TAB. 3.3 – Influence du gain dûe à la parallélisation sur le temps total d'exécution

Pour la comparaison entre (1) et (2), on se rend compte que le temps des communications joue, mais qu'il est soit compensé, soit au contraire appuyé, par un gain du reste du programme. On observe de la même façon une perte de performance de la version (3) par rapport à ce que devrait apporter la parallélisation. Néanmoins, une étude sur le comportement du reste du programme ne permet pas de conclure : les diverses versions résolvent le même nombre de systèmes linéaires et le nombre moyen d'itérations est sensiblement le même. Néanmoins, il peut s'agir d'un problème non-significatif dû aux circonstances des expérimentations, bien qu'il s'agisse là déjà de moyennes sur une dizaine de tests. Néanmoins, les résultats qui seront présentés dans la section suivante permettent d'apporter un autre éclairage.

3.6 Autre problème traité

3.6.1 Présentation

A partir du problème original, nous avons eu un autre problème à traiter : il s'agit de la résolution des équations de Navier-Stokes dans le cadre d'un écoulement axi-symétrique. Ceci signifie qu'il y a périodicité de la vitesse et de la pression par rapport à la direction azimutale (la direction dénotée par θ en coordonnées cylindriques). Il s'agit-là en fait du passage en trois dimensions du problème précédent, lorsque que l'obstacle rencontré par le fluide est de type sphérique ou ellipsoïde.

Cette périodicité de l'écoulement dans une des directions nous pousse alors à écrire les variables v et p en série de Fourier pour en tenir compte. On abandonne alors la formulation précédente en série de Fourier qui découlait d'une discrétisation particulière sur le temps pour se concentrer sur ce nouveau problème. On se place tout d'abord en coordonnées cylindriques. Les équations de Navier-Stokes reformulées dans ce contexte :

$$\begin{aligned}
\frac{\partial v_z}{\partial t} + v_z\frac{\partial v_z}{\partial z} &+ v_\theta\frac{\partial v_z}{r\partial\theta} = -\frac{\partial p}{\partial z} \\
&+ \nu(\frac{1}{r}\frac{\partial}{\partial r}(r\frac{\partial v_z}{\partial r}) + \frac{1}{r^2}\frac{\partial^2 v_z}{\partial\theta^2} + \frac{\partial^2 v_z}{\partial z^2}) \qquad (3.10)\\
\frac{\partial v_r}{\partial t} + v_z\frac{\partial v_r}{\partial z} &+ v_r\frac{\partial v_r}{\partial r} + \frac{v_\theta}{r}\frac{\partial v_r}{\partial\theta} - \frac{\partial v_\theta^2}{r} = -\frac{\partial p}{\partial r} \\
&+ \nu(\frac{\partial}{\partial r}[\frac{1}{r}\frac{\partial}{\partial r}(rv_r)] + \frac{1}{r^2}\frac{\partial^2 v_r}{\partial\theta^2} + \frac{\partial^2 v_z}{\partial z^2} - \frac{2}{r^2}\frac{\partial v_\theta}{\partial\theta}) \qquad (3.11)\\
\frac{\partial v_\theta}{\partial t} + v_z\frac{\partial v_\theta}{\partial z} &+ v_r\frac{\partial v_\theta}{\partial r} + v_\theta\frac{\partial v_\theta}{r\partial\theta} + \frac{v_\theta v_r}{r} = -\frac{\partial p}{r\partial\theta} \\
&+ \nu(\frac{\partial}{\partial r}[\frac{1}{r}\frac{\partial}{\partial r}(rv_\theta)] + \frac{1}{r^2}\frac{\partial^2 v_\theta}{\partial\theta^2} + \frac{\partial^2 v_\theta}{\partial z^2} + \frac{2}{r^2}\frac{\partial v_r}{\partial\theta}) \qquad (3.12)\\
\frac{\partial v_z}{\partial z} &+ \frac{1}{r}\frac{\partial}{\partial r}(rv_r) + \frac{1}{r}\frac{\partial v_\theta}{\partial\theta} = 0 \qquad (3.13)
\end{aligned}$$

L'écriture des variable sous forme de séries de Fourier nous donnent :

$$\begin{aligned}
v(z,r,\theta,t) &= \sum_{n=-\infty}^{n=+\infty} v_m(z,r,t)e^{im\theta} \\
p(z,r,\theta,t) &= \sum_{n=-\infty}^{n=+\infty} p_m(z,r,t)e^{im\theta}
\end{aligned}$$

On peut ensuite remettre cela dans les équations de Navier-Stokes précédentes (voir [102]) et découper les équations comme précédemment en trois parties pour résoudre le problème. Il n'y a ensuite pas de différences fondamentales avec ce qui précéde en ce qui concerne la parallélisation, les étapes restent les mêmes. La décomposition en série de Fourier ne porte pas sur la même variable que précédemment, mais le couplage entre les équations reste le même.

3.6.2 Comparaison exécution séquentielle - exécution parallèle

Les expérimentations sur ce nouveau problème portent sur un domaine maillé de façon beaucoup plus précise : 169 éléments avec chacun 36 noœuds d'interpolation dans un plan

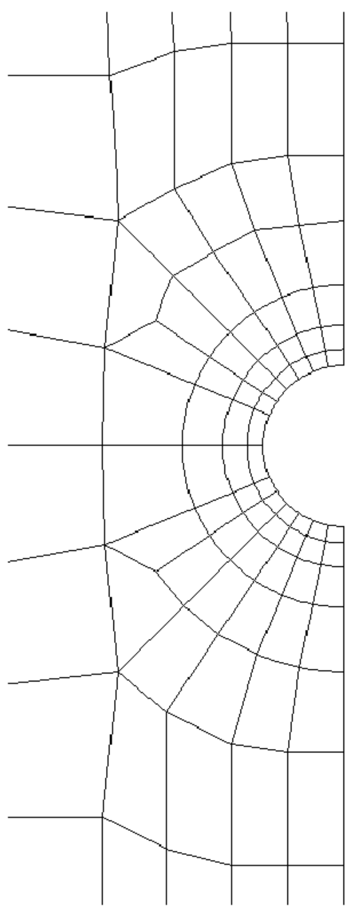


Fig. 3.25 – Morceau du maillage à 169 éléments

radial - axial. Une figure d'une partie du maillage, autour de l'obstacle est jointe : figure 3.25.

La figure 3.26 montre les temps d'exécution de la version séquentielle et de la version parallèle (qui est une moyenne entre les temps des trois versions parallélisées différemment). On peut étudier l'accélération pour cette version, voir le tableau 3.4.

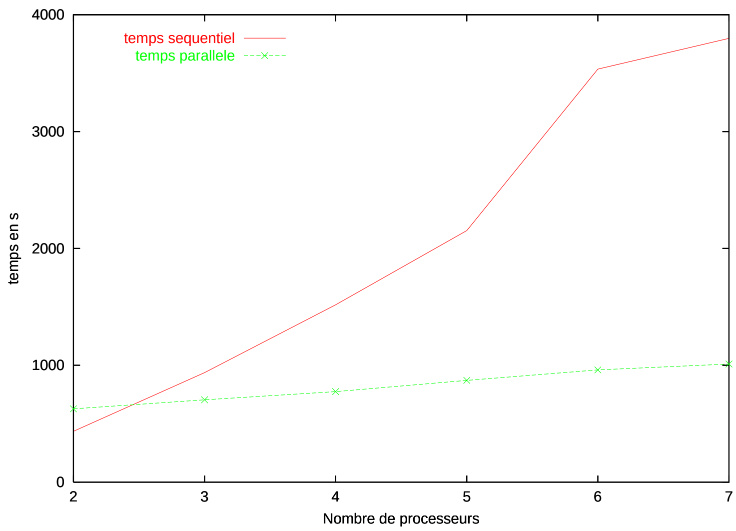


FIG. 3.26 – Temps d'exécution du programme (400 itérations) en fonction du nombre d'harmoniques

Nombre d'harmoniques	Temps séquentiel (en s)	Temps parallèle (en s)	Efficacité $(\frac{Tseq}{Tpar*proc})$
2	435.25	627.09	0.35
3	936.93	704.03	0.44
4	1517.78	0.49	
5	2152.49	870.52	0.49
6	3534.34	960.48	0.61
7	3797.70	1010.34	0.54

TAB. 3.4 – Comparaison du temps d'exécution séquentielle et parallèle

Les performances sont moins intéressantes qu'avec la version précédente qui mettait en jeu des données moins grosses. Néanmoins, la parallélisation permet d'avoir un gain sur le temps d'exécution total, avec 7 harmoniques, on divise le temps d'exécution par 3. Les explications sont les mêmes que précédemment : temps des communications à prendre en compte, et ce d'autant plus que les données sont plus grosses, ainsi que les synchronisations dûes au pas de temps qui oblige à prendre comme temps de référence d'une itération le temps le plus long.

Examinons de plus près la parallélisation suivant les trois techniques utilisées pour paralléliser les communications. Les figures 3.27, 3.28 et 3.29 montrent respectivement les temps d'exécution du programme total, les temps d'xécution de la procédure parallélisée et les temps de communication. Les courbes (1) correspondent aux courbes de la version avec le maximum de communication ; les courbes (2) correspondent aux courbes avec le nombre de communications divisées par 2 et les courbes (3) correspondent aux courbes avec le nombre de communications optimisées. Le nombre total d'itérations est de 2000 pour chaque exécution.

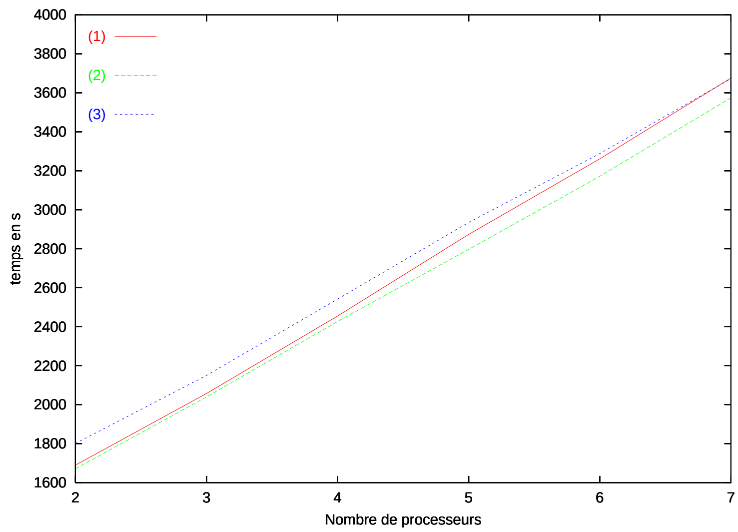


FIG. 3.27 – Temps d'exécution total du programme (2000 itérations)

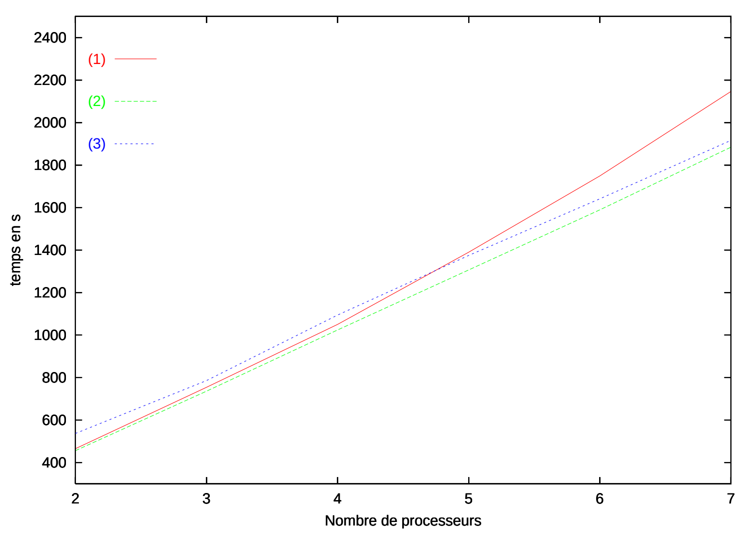


FIG. 3.28 – Temps d'exécution de la procédure parallélisée (2000 itérations)

On se rend compte que la parallélisation censée être optimale ne l'est plus, que ce soit en terme de temps de communications ou sur le temps total. Pour ce qui est du temps de communications, on ne peut l'expliquer qu'en considérant qu'il y a un temps d'attente

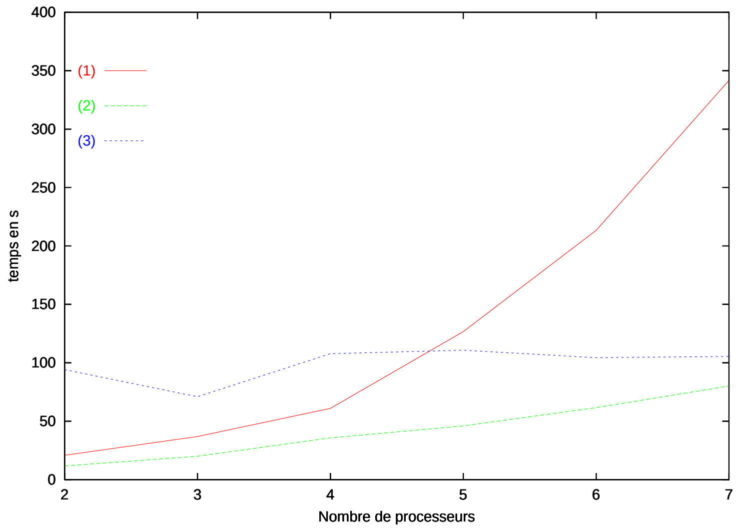


FIG. 3.29 – Temps de communication (2000 itérations)

plus grand dans le cas de communications optimisées que dans les autres. Par rapport au premier programme, on travaille sur des données plus grandes, donc sur des temps calculs plus élévés, ce qui modifie l'ordre relatif dans lequel les processeurs exécutent leurs calculs les uns par rapport aux autres. Par voie de conséquence, cela modifie aussi l'instant auquel ils sont prêts à envoyer leurs données aux processeurs qui en ont besoin et donc cela modifie les temps d'attente des uns et des autres. La parallélisation optimale en terme de nombre de communications ne l'est plus en terme de synchronisation avec les autres processeurs ce qui occasionne des retards. Néanmoins, dès que le volume de communication s'élève, c'est-à-dire ici dès 4 processeurs, donc 3 tableaux C à communiquer, la version 3 est tout de même meilleure que la version (1) qui induit le maximum de communications. Pour ce qui est de comparer (1) et (2), on ne retrouve pas le rapport de $1/2$ entre les temps de communications ce qui tend à prouver que la version (2) induit en plus un meilleur équilibre entre les communications et les calculs.

Ce temps sur les communications se ressent sur le temps d'exécution de la procédure parallélisée. On constate qu'il y a bien un gain entre les versions (1) et (2) pour lesquelles le temps de communication est plus élévé dans un cas que dans l'autre, ce qui confirme cette fois la théorie. Les temps d'exécution respectifs entre (1), (2) et (3) sont ceux attendus à la vue des courbes sur les temps de communication, confirmation de ce que l'on attendait.

On peut enfin regarder comment tout cela se répercute sur les temps d'exécution du programme en entier, via le tableau 3.5, qui montre la différence entre les temps de la procédure parallélisée (notés par) pour les cas (1) et (2) et la différence entre les temps d'exécution totale (notés tot). Même chose pour les les cas (1) et (3).
On s'aperçoit sur les courbes que la version (2) est la meilleure, ce qui n'est pas surprenant au vu des remarques précédentes sur le temps de calcul de la procédure parallélisée. Néanmoins, le gain par rapport à la version (1) n'est pas entièrement répercuté sur le programme total, surtout à partir de 6 modes.

nproc	par (1) - par (2)	tot (1) - tot (2)	par (1) - par (3)	tot (1) - tot (3)
2	9.37	18.21	-71.5	-111.01
3	17.98	17.69	-31.61	-92.60
4	26.22	28.06	-43.76	-86.88
5	83.59	76.63	15.52	-61.37
6	159.71	89.33	108.14	-26.98
7	262.37	101.16	230.27	3.3

Tab. 3.5 – Influence du gain dû à la parallélisation sur le temps total d'exécution (version azimutale, 2000 itérations)

La version (3) quant à elle est encore plus mauvaise qu'espérée ! Alors que pour un nombre d'harmoniques supérieure à 3 la procédure parallélisée (3) est plus rapide que (1), la courbe sur le temps total (3) reste au-dessus de (1), ce qui se lit très bien dans le tableau 3.5 où la différence positive pour (3) devient négative.

On va s'intéresser à ce qui se passe dans le reste du programme pour expliquer ces différences. Ainsi, les courbes présentées en figure 3.30 montrent le nombre moyen d'itérations nécessaires à la résolution des systèmes linéaires dans le reste du programme.
Pour ce qui concerne le nombre d'itérations qui ont permis de calculer les moyennes, il est de 8000 pour toutes les exécutions et pour tous les modes, à l'exception du mode 0 qui a un nombre de résolutions compris entre 5000 et 6000, mais on remarque que ce nombre pour les exécutions de type (3) est supérieur aux autres. On voit clairement sur les courbes que les résolutions dans le cas (3) prennent environ une itération de plus, ce qui qui concourent à augmenter encore plus les pertes dûes aux mauvaises synchronisations, ou à compenser les gains qu'on aurait pu avoir. On a un phénomène de dégradation numérique dûe aux problèmes de calcul en flottant. Pour les diverses parallélisations, on change l'ordre dans lequel on effectue les calculs, donc on n'obtient pas les mêmes résultats au bit près. La signification physique des données ainsi codées reste la même, mais la parallélisation (3) induit une mauvais couplage numérique.

3.6.3 Autres facteurs d'influence sur les temps d'exécution

Une série d'autres courbes a été réalisée sur la même machine Origin2000, avec les mêmes programes et les mêmes données en entrée et le même nombre d'itérations, mais juste après l'ajout de 20 processeurs R12000. Afin d'avoir des exécutions homogènes, on s'est restreint à n'utiliser que des processeurs R10000. Les temps d'exécution obtenus sont présentés sur les figures 3.31, 3.32, 3.33.
On voit que ces courbes sont complètement différentes des courbes précedentes, alors qu'il s'agit de la même chose... On note plusieurs différences : tout d'abord les temps de l'exécution totale peuvent varier du simple au double entre cette série d'exécutions et celles de la section précédente. Ensuite, la version du programme générant le moins de communications se comporte mieux que précédemment, en particulier pour un nombre de modes élevé. Comment expliquer ces différences ? Il est difficile de faire une analyse précise, mais il est probable que nous n'avions pas le monople de l'utilisation des processeurs sur

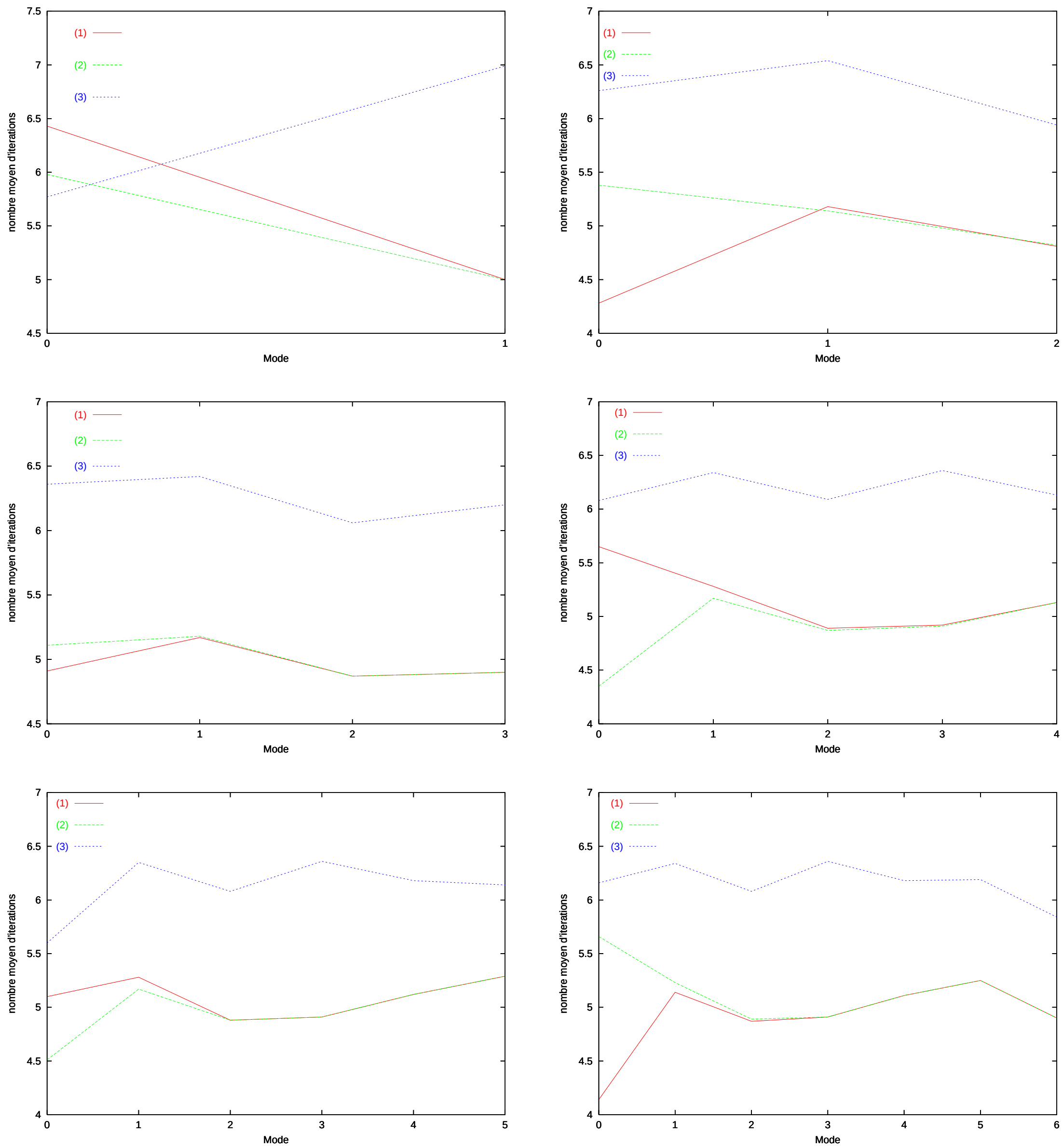


FIG. 3.30 – Nombre moyen d'itérations par mode, pour des exécutions sur 2, 3, 4, 5, 6 et 7 processeurs

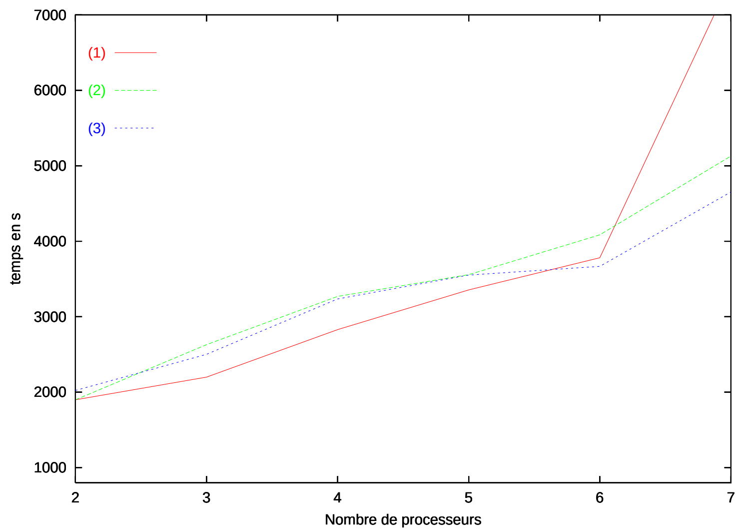


FIG. 3.31 – Temps d'exécution total du programme (2000 itérations) - 2

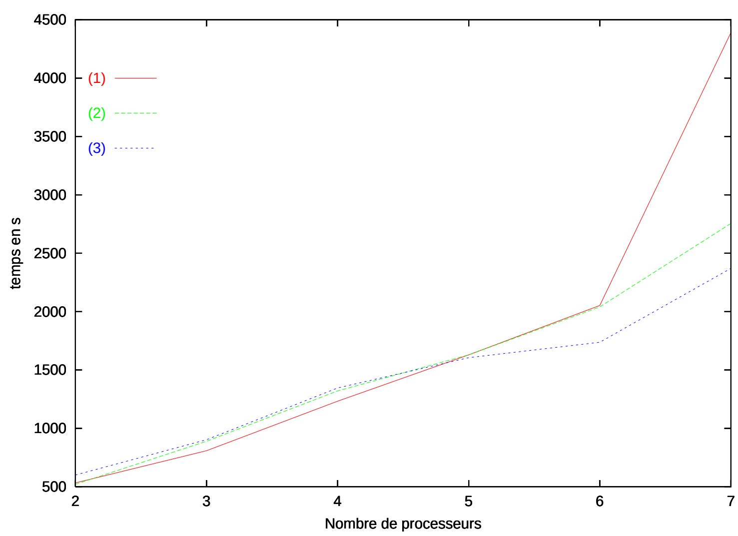


FIG. 3.32 – Temps d'exécution de la procédure parallélisée (2000 itérations) - 2

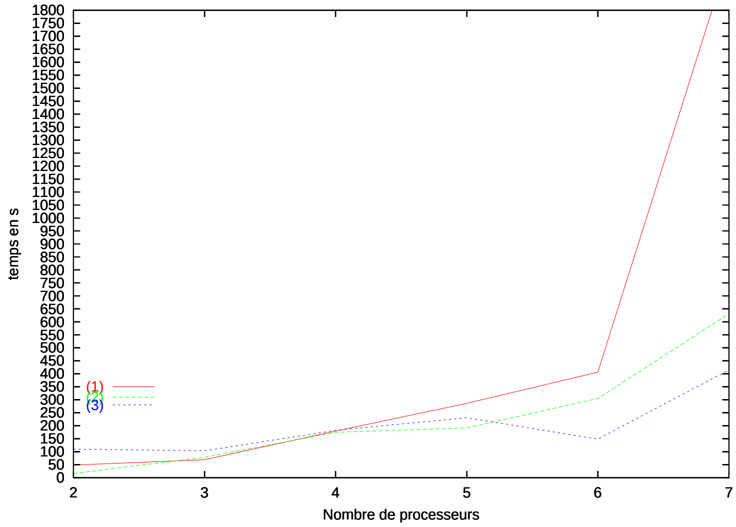


FIG. 3.33 – Temps de communication (2000 itérations) - 2

lesquels le programme s'éxécutait. En effet, forçant le placement sur certains processeurs, il est possible que la puissance disponible était partagée avec d'autres programmes qui n'étaient pas migrés par le système vers d'autres processeurs libres. L'équilibrage de charge se faisait alors différemment, privilégiant la solution optimale.

Forcer le placement n'a rien d'exotique, on aurait pu vouloir le faire aussi afin de profiter de l'architecture de la machine : on dispose d'un hypercube dont les nœuds sont des blocs de 2 processeurs qui partagent la même zone mémoire. Les communications entre ces 2 processeurs sont donc extrêmement rapides, d'où l'intérêt de placer les processus sur de tels blocs.

Maintenant, regardons quelle était la charge entre les deux type d'exécutions, figure 3.34.

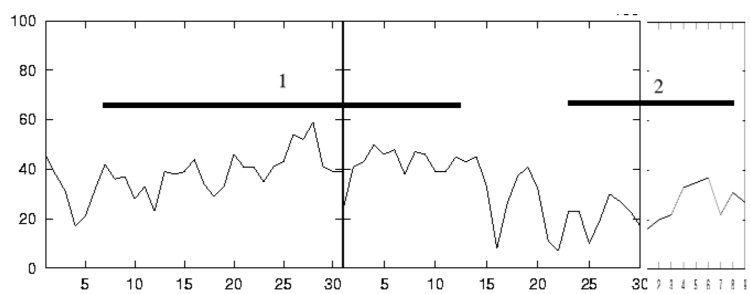


FIG. 3.34 – Pourcentage d'utilisation CPU de l'Origin 2000 en fonction du temps

La période 1 est celle durant laquelle les exécutions pour obtenir les courbes de cette section ont été effectuées et la période 2 est celle des courbes de la section précédente. On constate une baisse sensible : on passe de plus de 40% à environ 30%, et on peut estimer que ces 40% correspondent à une utilisation de 55% environ des R10000, en comparant charge totale et temps CPU des R10000 avec des cas précédents où seuls les R10000 étaient disponibles. Vu l'utilisation des processeurs durant cette période, l'hypothèse de partage des ressources CPU des processeurs est fort possible.

Par contre, nous ne partagions pas les processeurs sur la période 2 grâce au nouveau système de files mis en place qui assure de lui-même un placement fixe et sépare clairement les types de processeurs disponibles.

On peut aussi penser que les pertes d'efficacité abordées dans la partie 3.5.2 sont dûes au système. Les expérimentations faites dans cette section ne plaçaient pas les processus sur des processeurs précis : il est possible que le système ait fait migrer des processus à cause des interactions possibles entre programmes, afin d'optimiser placement et place mémoire nécessaire. Vu que la machine était très utilisée à ce moment-là, à environ 75%, c'est une hypothèse raisonnable, d'autant plus que la version (3) est un peu plus gourmande en mémoire et est susceptible d'interagir différemment avec les autres programmes.

3.7 Conclusion et perspectives

Nous avons étudié dans cette section la parallélisation d'un problème issu de la mécanique des fluides. Cela nous a amené à nous restreindre à l'étude d'un nid de boucles à paralléliser en minimisant les communications induite par la parallélisation dans un contexte de place mémoire limitée.

Un étude à la main nous a permis de réécrire ce nid de boucle pour avoir le minimum de communication possible au prix d'un petit surplus de mémoire par rapport au problème initial. Ensuite, diverses modélisations du problème nous ont permis de voir quelles étaient les limitations du modèle polyédrique pour l'étude directe de ce nid, ainsi que d'étudier l'intérêt des techniques d'optimisation d'accès à la mémoire dans le cadre de l'optimisation du nombre de communications. Les toutes dernières innovations en ce domaine sont particulièrement intéressantes puisqu'elles permettent d'approcher notre solution écrite à la main. Le problème reste l'élargissement de la technique à plusieurs boucles successives, ainsi que la détection des réductions auxquelles la technique peut facilement être adaptée.

Après cette série d'études théoriques, une étude pratique sur l'intérêt réel de la parallélisation a été conduite, avec divers tests de performance sur une machine Origin 2000 de 32 processeurs R10000. Il s'avère que les performances attendues dépendent de la taille du problème ainsi que de l'utilisation qui est faite de la machine par ailleurs. Cette étude a aussi permis de mettre en lumière le problème de la parallélisation des réductions, qui bousculant l'ordre des calculs, modifie les résultats des calculs lorsqu'ils sont faits par des ordinateurs. Notre parallélisation "optimale" génère ainsi des instabilités numériques qui allongent les temps de calculs et ne sont pas forcément compensées par un nombre de communications moindre. Néanmoins, cette optimisation du nombre de communications s'avère efficace lorsque la machine est très utilisée car elle permet de compenser les temps de communication longs entre processeurs distants.

La parallélisation a été testée sur deux types de problèmes : dans l'un, on utilisait une certaine modélisation des données par série de Fourier permettant la parallélisation, alors que dans l'autre, c'est le passage à la troisième dimension qui permettait d'obtenir une modélisation similaire. Par la suite, on pourrait envisager de coupler les 2 méthodes : passer en trois dimensions avec la décomposition de Fourier tout en gardant la double discrétisation temporelle qui permet une seconde décomposition de Fourier. Danc ce cas, on aurait un volume de données encore plus conséquent à traiter : si la décomposition spatiale comporte N modes de Fourier et celle temporelle en comporte M, on a alors $N.M$ données d'entrée à traiter. Néanmoins, avant d'en arriver à implémenter ce problème il est nécessaire de réécrire les équations de départ en y intégrant les 2 modélisations...

Ce problème serait sans doute à paralléliser différemment. En effet, $N.M$ risque d'être supérieur au nombre de processeurs dont on dispose, contrairement à nos parallélisations précédentes. Il faudrait alors réfléchir à la manière de regrouper les divers modes sur des processeurs identiques, ou alors à ne pas prendre en compte tous les modes pour toutes les variables afin d'avoir un nombre total de données à traiter raisonnable au vu du nombre de processeurs dont on dispose.

Chapitre 4

Comparaison d'outils pour la parallélisation de la décomposition de Cholesky

4.1 Factorisation de Cholesky

4.1.1 Présentation

La résolution des systèmes linéaires matriciels se fait généralement en décomposant la matrice en 2 matrices triangulaires inférieure (L) et supérieure (U), en permutant les lignes de la matrice originale si nécessaire. Pour une matrice A on calcule donc :

$$PA = LU$$

où P est une matrice de permutation. Cette décomposition est basée sur l'application de l'algorithme du pivot de Gauss. La résolution d'un système $Ax = b$ une fois cette décomposition effectuée est simple : il se ramène à résoudre $LUx = Pb$. Il suffit de résoudre deux systèmes triangulaires : $Ly = Pb$, puis $Ux = y$.
Néanmoins, cet algorithme est inadapté dans le cas des matrices symétriques : on perd la symétrie lors de la décomposition LU. La factorisation de Cholesky est un algorithme qui s'applique aux matrice symétriques, définies positives (SPD), c'est-à-dire vérifiant :

$$\begin{aligned}
&\text{symétrique} : A = A^T \\
&\text{définie} : (Ax, x) = 0 \Leftrightarrow x = 0 \\
&\text{positive} : \forall x \ \ (Ax, x) \geq 0
\end{aligned}$$

et qui garde la symétrie. En effet, on montre qu'on peut décomposer les matrices SPD sous la forme $A = LL^T$, L étant une matrice triangulaire inférieure.
Il existe divers algorithmes de calcul de L, suivant qu'on la calcule colonne par colonne ou ligne par ligne.

Construction colonne par colonne de L

première colonne On veut $A = LL^T$. Ceci se traduit par :

$$a_{11} = l_{11}.l_{11} \text{ donc } l_{11} = \sqrt{a_{11}}$$

Ensuite, on a, pour $2 \leq i \leq n$:

$$a_{i1} = l_{11}.l_{i1} \Rightarrow l_{i1} = \frac{a_{i1}}{l_{11}}$$

j-ème colonne On suppose que les $j-1$ premières colonnes sont calculées. On a :

$$a_{jj} = l_{jj}^2 + \sum_{k=1}^{j-1} l_{jk}^2 \text{ et donc } l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$$

Une fois calculé l'élément diagonal, on peut calculer le reste de la colonne :

$$a_{ij} = l_{ij}.l_{jj} + \sum_{k=1}^{j-1} l_{ik}l_{jk} \text{ et donc } l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk}}{l_{jj}}$$

Le critère "définie positive" de la matrice assure que les calculs $\sqrt{a_{jj} - \sum_{i=1}^{j-1} l_{ji}^2}$ sont possibles et ne sont jamais nuls (pour éviter une division par 0).
La figure 4.1 illustre ces calculs

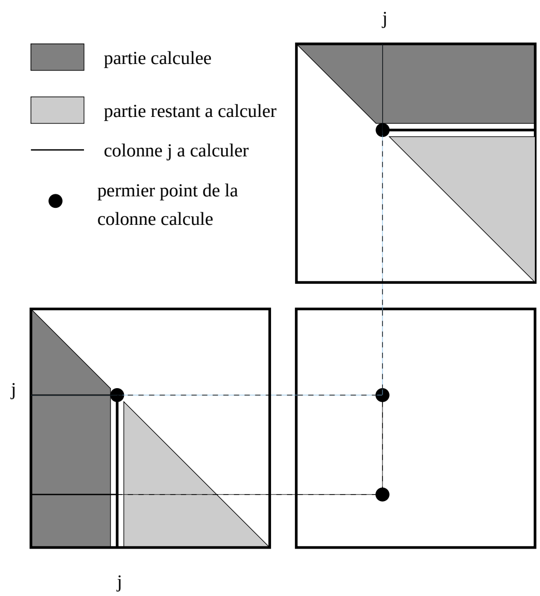


FIG. 4.1 – Construction de L pour $A = LL^T$ par colonnes

Finalement, l'algorithme qui correspond à cette formulation est donné en Fig. 4.2.

```
Do j = 1, n
  L[j,j] = A[j,j]
  Do k = 1, j-1
     L[j,j] = L[j,j] + L[j,k] * L[j,k]
  L[j,j] = sqrt(L[j,j])
  Do i = j+1, N
     L[i,j] = A[i,j]
     Do k = 1, j-1
          L[i,j] = L[i,j] - L[i,k] * L[j,k]
     L[i,j] = L[i,j] / L[j,j]
```

FIG. 4.2 – Algorithme de la factorisation de Cholesky colonne par colonne

Construction ligne par ligne

première ligne : elle est en fait constituée d'un unique élément

$$a_{11} = l_{11}.l_{11} \text{ donc } l_{11} = \sqrt{a_{11}}$$

i-ème ligne : sachant que les $i-1$ premières lignes sont déjà calculées on peut écrire, pour $j \leq 1 \leq i-1$:

$$a_{ij} = \sum_{k=1}^{j} l_{ik} l_{kj} \text{ d'où } l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j} l_{ik} l_{kj}}{l_{jj}}$$

Et on peut alors calculer la valeur sur la diagonale :

$$l_{ii} = \sqrt{a_{ii} - \sum_{j=1}^{i-1} l_{i,j}^2}$$

La figure 4.3 illustre cette deuxième manière de procéder.
Finalement, l'algorithme qui correspond à cette formulation est donné en Fig. 4.4.

Construction par transformations successives

Enfin, on peut construire L peu à peu, sous une forme qui s'apparente à une construction par blocs, et qui permet d'ailleurs d'obtenir un telle construction. On veut avoir :

$$A = \left(\begin{array}{c|c} d_1 & b_1^T \\ \hline b_1 & B_1 \end{array}\right) = \left(\begin{array}{c|c} l_1 & 0 \\ \hline m_1 & I \end{array}\right) \times \left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & B_2 \end{array}\right) \times \left(\begin{array}{c|c} l_1 & m_1^T \\ \hline 0 & I \end{array}\right)$$

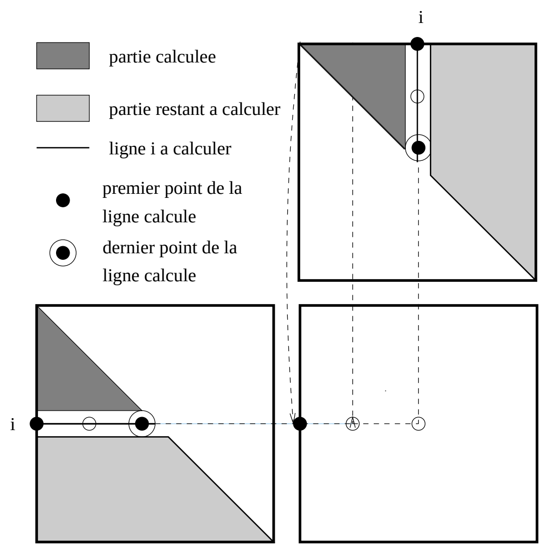


FIG. 4.3 – Construction de L pour $A = LL^T$ par lignes

```
Do i = 1, n
  Do j = 1, i-1
     L[i,j] = A[i,j]
     Do k = 1, j-1
          L[i,j] = L[i,j] - L[i,k] * L[j,k]
     L[i,j] = L[i,j] / L[j,j]
  L[i,i] = A[i,i]
  Do k = 1, i-1
     L[i,i] = L[i,k] * L[i,k]
  L[i,i] = sqrt (L[i,i])
```

FIG. 4.4 – Algorithme de la factorisation de Cholesky ligne par ligne

On obtient alors, par identification :

$$l_1^2 = d_1 \Rightarrow l_1 = \sqrt{d_1}$$
$$l_1 m_1 = b_1 \Rightarrow m_1 = \frac{1}{l_1} b_1$$
$$B_2 = B_1 - m_1.m_1^T$$

Il suffit de recommencer le procédé avec B_2 à la place de A et on construit ainsi peu à peu la matrice L pour obtenir finalement :

$$A = \begin{pmatrix} l_1 & & & \\ & l_2 & & \\ m_1 & m_2 & \ddots & \\ & & & l_n \end{pmatrix} \times \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \times \begin{pmatrix} l_1 & & m_1^T & \\ & l_2 & m_2^T & \\ & & \ddots & \\ & & & l_n \end{pmatrix}$$

on obtient cette fois l'algorithme de la figure 4.5.

```
L = A
Do j = 1, n
  L[j,j] = sqrt(L[j,j])
  Do i = j+1, n
     L[i,j] = L[i,j] / L[j,j]
  Do k = j+1, N
     Do i = k, N
     L[i,k] = L[i,k] - L[i,j] * L[k][j]
```

FIG. 4.5 – Algorithme de la factorisation de Cholesky par transformations successives

4.1.2 Parallélisation des bibliothèques

La parallélisation de cette méthode telle qu'elle est implémentée dans ScaLAPACK [20, 21] (version parallélisée par passage de messages de LAPACK) est basée sur la dernière formulation, étendue à la version par blocs. La matrice A est distribuée par blocs sur les processeurs, vus sous la forme d'une grille à 2 dimensions. Chaque processeur contient donc un bloc de la matrice.

On obtient facilement, à partir de la formulation précdente, une formulation par bloc en cherchant A sous la forme :

$$A = \left(\begin{array}{c|c} D_1 & A_1^T \\ \hline A_1 & B_1 \end{array}\right) = \left(\begin{array}{c|c} L_1 & 0 \\ \hline M_1 & I \end{array}\right) \times \left(\begin{array}{c|c} I & 0 \\ \hline 0 & B_2 \end{array}\right) \times \left(\begin{array}{c|c} L_1^T & M_1^T \\ \hline 0 & I \end{array}\right)$$

On calcule alors :

$L_1 L_1^T = D_1$: on fait la composition de Cholesky de D_1
$L_1 M_1 = A_1$: résolution d'un système triangulaire pour obtenir $M_1 = L_1^{-1} A_1$
$B_2 = B_1 - M_1 D_1^{-1} M_1^T$

On obtient le programme parallèle en parcourant la matrice par colonnes de processeurs. On fait localement la factorisation de Cholesky du bloc diagonal, en faisant appel à la procédure séquentielle de LAPACK. Les résolutions triangulaires, ainsi que le produit de matrices se font par appel aux procédure PBLAS3 (BLAS de niveau 3 parallélisées).

4.2 Parallélisation "automatique" de la factorisation de Cholesky

4.2.1 Avec OPERA

Comme nous l'avons dit au tout début, pour étudier un programme sous OPERA, il faut l'écrire en assignation unique dans le langage Alpha, ce qui est présenté en figure 4.6. Les calculs sont effectués dans le tableau M : l'étape 0 consiste en l'initialisation. Le deuxième calcul constitue le cœur du calcul, alors que les 2 calculs suivants sont d'une part le calcul de la racine carrée pour les éléments diagonaux, et d'autre part la division par l'élément diagonal pour les autres valeurs de la matrice à calculer.

```
system DecompChol : { N | N>=1 }
               ( A : { i,j | 0<=i<=N; i<=j<=N } of real )
       returns ( L : { i,j | 1<=i<=N; j<=i<=N ; 1<=j} of real );

var
   M  : { i,j,k | 1<=i<=N; 1<=j ; j<=i<=N+1; 0<=k<=j} of real;

let
   M[i,j,k]  = case
               { | k=0 } : A[i,j] ;
               { | j<=i<=N ; 1<=k<=j-1} : M[i,j,k-1] - M[i,k,k]*M[j,k,k] ;
               { | 1<=j<=N ; i = j ; i = k } :  M[j,j,k-1] ;
               { | j+1<=i<=N ; k = j } : M[i,j,k-1] / M[j,j,j] ;
               esac;

   L[i,j] = M[i,j,j];
tel;
```

FIG. 4.6 – La factorisation $A = LL^T$ mise en assignation unique

On peut alors donner ce programme au logiciel OPERA pour l'étudier. La figure 4.7 nous montre le domaine d'itération (domaine de M) pour une valeur de N égal à 5.

Les points gris, situés sur la portion de plan triangulaire à la base du polyèdre constituent les points initialisés avec A, le points verts, situés sur la portion de plan triangulaire inclinée vers la droite, sont les points terminaux. Les axes du domaines dont données par le petit schéma à côté de la copie d'écran du logiciel.

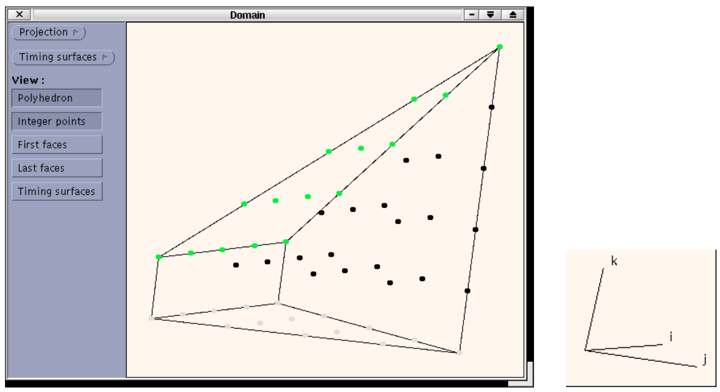


FIG. 4.7 – Le domaine d'itération et l'orientation des axes (N=5)

On peut ensuite faire apparaître l'ensemble des dépendances générées par le programme, via la visualisation des ensembles d'utilisation pour chaque utilisation du tableau M. Ceci est illustré par la figure 4.8, que l'on peut découper pour explications.

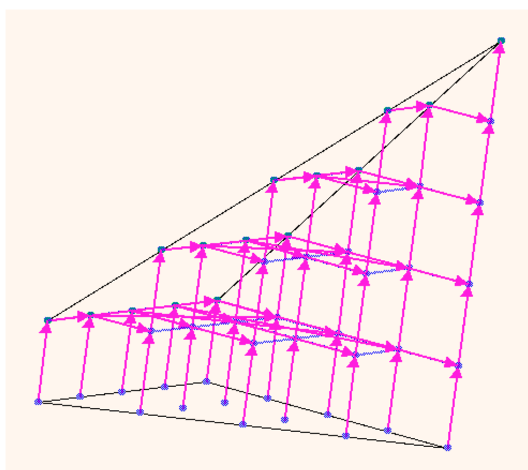


FIG. 4.8 – Visualisations de tous les ensembles d'utilisation (N=5)

Tout d'abord, la série de captures d'écrans de la figure 4.9 nous montre les ensembles d'utilisation pour le calcul de : $M[i,j,k] = M[i,j,k-1] - M[i,k,k] * M[j,k,k]$. Plus précisément, on voit les ensembles d'utilisation des valeurs du type $M[i,j,k-1]$ puis $M[i,k,k]$ et enfin $M[j,k,k]$. Les vecteurs dans la direction k illustrent les dépendances entre $M[i,j,k]$ et $M[i,j,k-1]$.

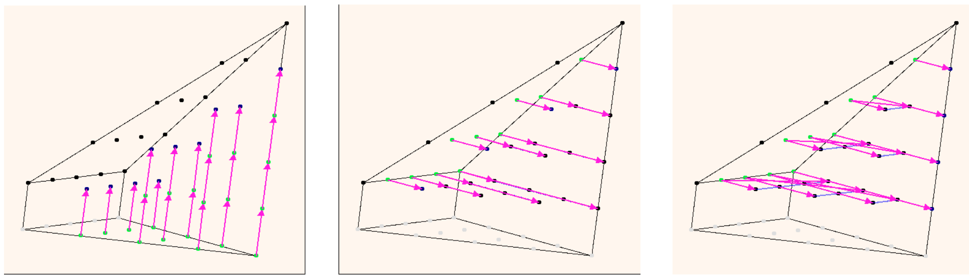


FIG. 4.9 – Ensembles d'utilisation nécessaire au calcul $M[i, j, k] = M[i, j, k-1] - M[i, k, k] * M[j, k, k]$

Ensuite, nous avons la figure 4.10 qui représente les ensembles d'utilisation pour le calcul des valeurs diagonales $M[i, j, k] = sqrt(M[i, j, k-1])$. On remarque que ces valeurs se trouvent sur l'arrête qui vérifie $i = j = k$ comme elles devraient l'être.

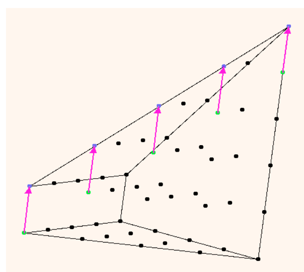


FIG. 4.10 – Ensembles d'utilisation nécessaire au calcul $M[i, j, k] = sqrt(M[i, j, k-1])$

Enfin, on trouve les ensembles d'utilisation pour la division par l'élément diagonal, qui nous permet d'obtenir les valeurs finales du calcul par $M[i, j, k] = M[i, j, k-1]/M[j, j, j]$. Ils sont illustrés par la figure 4.11 avec 2 copies d'écran, pour les ensembles correspondants à l'utilisation des $M[i, j, k-1]$ et des $M[j, j, j]$. On note que les valeurs ainsi calculées se retrouvent effectivement sur la face contenant les points terminaux du calcul.

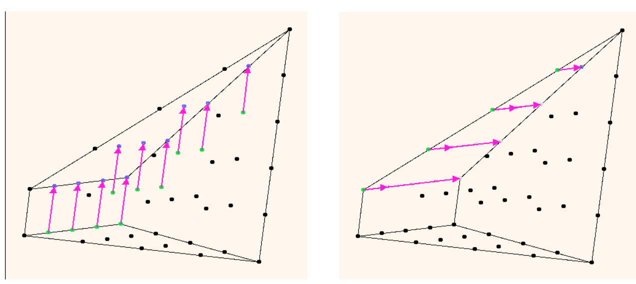


FIG. 4.11 – Dépendances pour le calcul de $M[i, j, k] = M[i, j, k-1]/M[j, j, j]$

On peut ensuite voir à donner une allocation et un cadencement. On peut choisir de donner un cadencement, ou laisser le sytème choisir celui qui lui semble le meilleur. Si on laisse le système choisir, le cadencement se fait selon le vecteur $(1, 1, 1)$. Les surfaces de temps, correspondant aux points qui ont la même date d'exécution sont illustrées sur le premier schéma de la figure 4.12. On obtient alors un temps de latence de 14 et un parallélisme potentiel de 6 : 6 processeurs virtuels actifs au même instant au maximum.

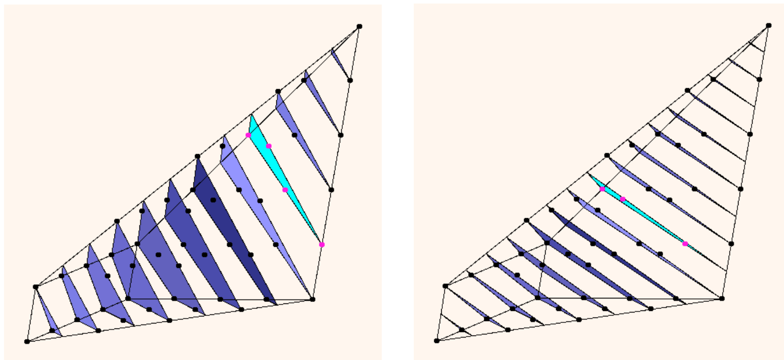


FIG. 4.12 – Surfaces de temps avec le vecteur de cadencement $(1, 1, 1)$, puis $(1, 1, 2)$

On peut aussi choisir de donner son propre cadencement. Essayons avec $(1, 1, 2)$. Le résultat est illustré par le second schéma de la figure 4.12. Notre choix est plus mauvais, puisque le temps de latence est alors de 19.

Enfin, on peut choisir l'allocation. Afin de minimiser les communications, il est sensé de prendre un vecteur de projection parallèle à l'une des directions de communications, par exemple suivant i ou j. On peut alors voir le résultat de la projection, et les ensembles d'utilisation projetés permettent de voir les communications résiduelles. Deux exemples sont illustrés : avec une allocation suivant $(1, 1, 1)$ et suivant $(1, 0, 0)$:

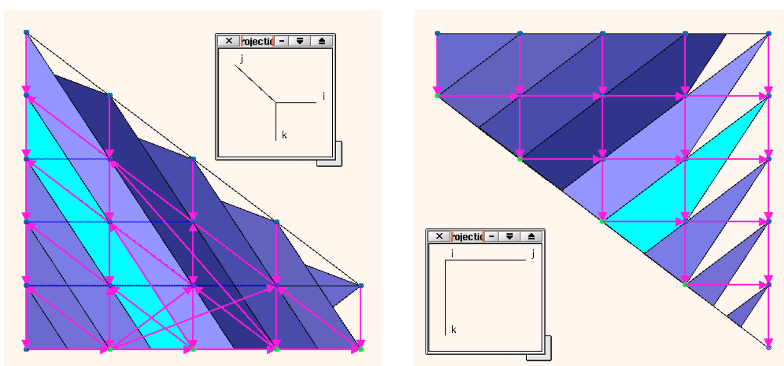


FIG. 4.13 – Surface de temps avec le vecteur d'allocation $(1, 1, 1)$, puis $(0, 1, 0)$

4.2.2 Avec PAF

PAF [26, 34, 35, 36] est un acronyme pour Paralléliseur Automatique de FORTRAN. Développé au PRISM à l'université Versailles St Quentin, il a pour but d'implémenter certains algorithmes de placement et d'ordonnancement de façon à paralléliser automatiquement du code FORTRAN.

L'un des avantages est que l'on peut donner le programme à traiter directement en FORTRAN, même s'il s'agit d'un sous-ensemble de FORTRAN. Le programme d'entrée est donné en figure 4.14.

```
      program cholesky

      real a(100,100)
      real l(100,100)
      integer i,j,k
      integer N

      do j=1,N
C
C     calcul de l'element diagonal l(j,j)
C
         l(j,j) = a(j,j)
         do k=1,j-1
           l(j,j) = l(j,j) - l(j,k)*l(j,k)
        enddo
        l(j,j) = sqrt(l(j,j))
C
C     calcul de la colonne j
C
        do i=j+1,N
           l(i,j) = a(i,j)
           do k=1,j-1
              l(i,j) = l(i,j) - l(i,k) * l(j,k)
           enddo
           l(i,j) = l(i,j) / l(j,j)
        enddo
      enddo
      end
```

FIG. 4.14 – Programme FORTRAN de décomposition de Cholesky à donner en entrée de PAF

On remarque que contrairement à OPERA, pour lequel le programme d'entrée contenait un paramètre, N la taille de la matrice, il se retrouve ici fixé. Il ne s'agit pas d'une limitation de l'outil, qui fait ses calculs sans prendre en compte les dimensions des tableaux, mais d'une contrainte d'écriture sur les programmes à donner en entrée. Les résultats sont donc tous aussi généraux dans un cas que dans l'autre.
Les résultats intéressants nous sont donnés via un ordonnancement et un placement ; ceux-ci sont décrits instruction par instruction. Il convient de numéroter les instructions qui font les calculs pour s'y retrouver. Elles sont numérotées à partir de 1, dans l'ordre d'apparition du programme, voir figure 4.15

```
     do j=1,N
i1     l(j,j) = a(j,j)
            do k=1,j-1
i2             l(j,j) = l(j,j) - l(j,k)*l(j,k)
            enddo
i3          l(j,j) = sqrt(l(j,j))
            do i=j+1,N
i4             l(i,j) = a(i,j)
               do k=1,j-1
i5                l(i,j) = l(i,j) - l(i,k) * l(j,k)
               enddo
i6             l(i,j) = l(i,j) / l(j,j)
            enddo
     enddo
```

FIG. 4.15 – Numérotation des instructions

On peut donc donner le cadencement rendu par PAF, toujours instruction par instruction :

$$\begin{array}{ll} i_1 & t = 0 \\ i_2 & t = 3k \\ i_3 & t = 3j - 2 \\ i_4 & t = 0 \\ i_5 & t = 3k \\ i_6 & t = 3j - 1 \end{array}$$

Les indices font référence aux indices utilisés dans le calcul pour une instruction donnée. Les instructions i_1 et i_4, qui correspondent à l'initialisation, sont effectuées à t = 0. Ensuite, sont effectuées en séquence les instructions i_3, i_1 et finalement i_2 et i_5. Si on étudie de plus près ce résultat, on constate que ce cadencement correspond, pour cet exemple, à une exécution au plus tôt. On peut faire une petite “simulation” sur un court exemple pour s'en convaincre.
Prenons $N = 3$. À $t = 0$, on exécute les instructions i_1 et i_4 soit :

```
i1  l(1,1) = a(1,1)
i1  l(2,2) = a(2,2)
i1  l(3,3) = a(3,3)
i4  l(2,1) = a(2,1)
i4  l(3,1) = a(3,1)
i4  l(3,2) = a(3,2)
```

À $t = 1$, on calcule une instruction i_3, pour $j = 1$, soit :

```
l(1,1) = sqrt(l(1,1))
```

À $t = 2$, on calcule des instructions i_6, pour $j = 1$:

```
l(2,1) = l(2,1) / l(1,1)
l(3,1) = l(3,1) / l(1,1)
```

À $t = 3$, on calcule des instruction i_2 et i_5, pour $k = 1$:

i_2 `l(2,2) = l(2,2) - l(2,1) * l(2,1)`
i_2 `l(3,3) = l(3,3) - l(3,1) * l(3,1)`
i_5 `l(3,2) = l(3,2) - l(3,1) * l(2,1)`

À $t = 4$, on repasse à une instruction i_3, pour $j = 2$ cette fois :

```
l(2,2) = sqrt(l(2,2))
```

À $t = 5$, on a une instruction i_6, pour $j = 2$:

```
l(3,2) = l(3,2) / l(2,2)
```

À $t = 6$, on a une instruction i_2 (il n'y a plus d'instructions i_4) pour $k = 2$:

i_2 `l(3,3) = l(3,3) - l(3,2) * l(3,2)`

Enfin, à $t = 7$, il reste une instruction i_3 :

```
l(3,3) = sqrt(l(3,3))
```

On voit bien que toute donnée calculée est aussitôt réutilisée si nécessaire.

Le placement obtenu par PAF est le suivant :

i_1 p = (0,j)
i_2 p = (0,j)
i_3 p = (0,0)
i_4 p = (i,j)
i_5 p = (i,j)
i_6 p = (0,i)

Le placement tel qu'on le voit là est plutôt effectué dans le cadre d'une machine à mémoire partagée car on a un placement par instruction et non pas par emplacement mémoire ou par donnée. Le placement proposé ne permet pas de déduire directement un placement possible pour machine à mémoire distribuée, placement respectant la règle de la *owner computes rule* (le processeur dont la mémoire contient une donnée calcule les modifications de cette donnée). Prenons le cas du calcul de l'élément $l(3,2)$. Il est modifié aux instants $t = 3$ et $t = 5$. Le premier calcul correspond à une instruction i_5, qui s'exécute sur le processeur (3,2) et le second à une instruction i_6 qui s'exécute sur (0,2).

Ceci n'empêche pas d'obtenir du code respectant la *owner computes rule*, car un certain nombre de techniques [14] permettent de s'y ramener, par exemple en ajoutant des variables temporaires qui servent lorsqu'un processeur effectue un calcul sur une donnée qu'il n'est pas censé mettre à jour : il effectue la mise à jour sur la variable temporaire et le processeur qui s'occupe de la variable non-temporaire correspondante en question effectue la mise à jour à partir de la variable temporaire.

4.3 OPERA / PAF : les différences, les ressemblances

Si on compare les résultats obtenus avec les 2 outils, on se rend compte qu'ils sont différents. Les schémas de la figure 4.16 illustrent les résultats obtenus pour le cas $N = 3$.

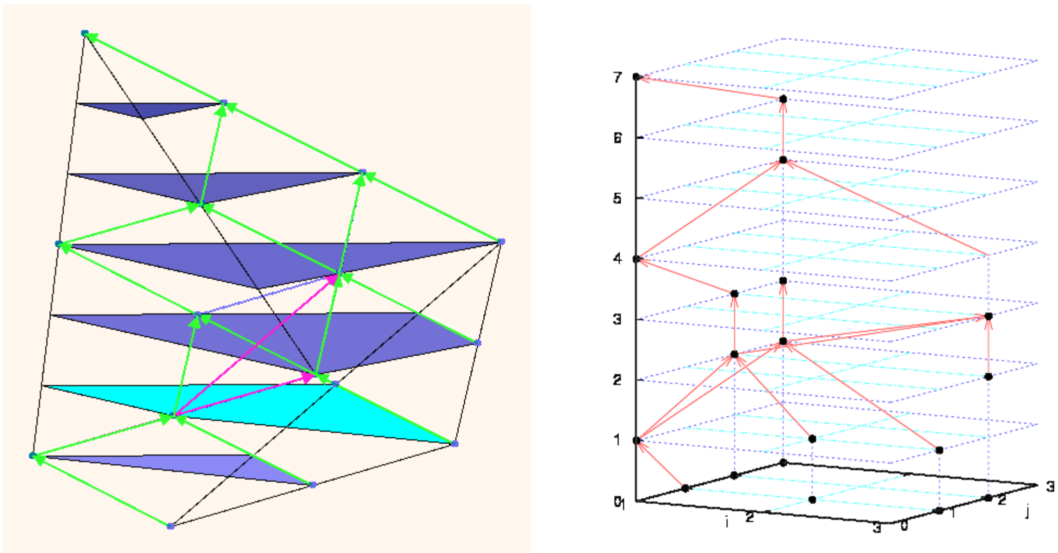


FIG. 4.16 – Placement et ordonnancement de la factorisation de Cholesky avec OPERA et PAF ($N = 3$)

4.3.1 Cadencement

Les cadencements par exemple ne sont pas les mêmes, et bien qu'on puisse choisir son cadencement avec OPERA, on ne peut obtenir le même résultat qu'avec PAF. Si on se réfère à la figure 4.8, il nous faudrait choisir un vecteur de cadencement orthogonal au plan (i, j), c'est à dire au plan contenant les points gris afin que ces derniers soient calculés à $t = 0$, comme dans le cadencement de PAF. Cela signifie alors qu'on calcule en même temps des points entre lesquels il y a une dépendance dans les pas de temps qui suivent, puiqu'il y a des dépendances à l'intérieur des plans orthogonaux à $(0, 0, 1)$: les points qui se trouvent dans les plans facilement visualisables sur la dernière image de la figure 4.9 via les vecteurs de dépendance s'exécuteraient aux mêmes dates...

Néanmoins, le temps d'exécution des deux programmes reste le même sur cet exemple puisqu'il correspond au chemin critique dans le graphe de dépendances.

Ces différences sont dûes aux différentes stratégies d'ordonnancement implémentées. Dans OPERA, on cherche une transformation espace-temps pour chaque variable du système d'équations récurrentes, avec un ordonnancement affine du type : $t_X(z, p) = \lambda_X z + \lambda'_X p + a_X$ (pour la variable X, avec le vecteur de paramètres p). Pour trouver les λ qui conviennent, il faut tenir compte de l'ensemble des dépendances pour le calcul de X et écrire les inégalités qui s'en suivent sur les dates d'exécutions respectives de X et des variables desquelles elle dépend.

Par exemple, on a $M[i, j, k] = M[i, j, k-1]/M[j, j, j]$, ce qui signifie que $t_M((i, j, k), n) > t_M(i, j, k-1, n)$ et que $t_M((i, j, k), n) > t_M((j, j, j), n)$ (en prenant les bons domaines de validité des paramètres). En développant, on obtient une série d'inéquations dont la résolution permet de calculer les valeurs des λ. OPERA calcule ainsi toutes les contraintes en se restreignant aux valeurs z extrêmales du polyèdre paramétré représentant le domaine de validité des indices : c'est la méthode des sommets. La résolution, par appel à la librairie Polylib [90] permet d'avoir le cadencement optimal pour cette classe de cadencement.

Dans PAF, on ne décrit pas le programme sous forme d'équations récurrentes. Sur notre exemple, on a un unique cadencement pour le programme pour OPERA car notre système d'équations récurrentes affines n'a qu'une seule variable. Or, si on décrit le programme en terme d'instructions, comme c'est le cas pour PAF, on est amené à calculer un cadencement par instruction, soit dans notre cas six fonctions de cadencement à calculer.

A partir de là, le raisonnement est similaire à celui qui précède, sauf que les données sont remplacées par des instructions. On cherche des cadencements affines paramétrés pour chaque instruction S, de la forme : $t_S(z,p) = \lambda_S z + \lambda'_S p + a_S$. On écrit les contraintes qui découlent des dépendances, par exemple pour l'instruction i_6 `l(i,j) = l(i,j) / l(j,j)`. Une instruction $i_6(i,j)$ est donc en dépendance avec une instruction $i_3(j')$ (calcul de $l(j,j)$) telle que $j = j'$ et une instruction $i_5(i',j',k')$ (calcul de $l(i,j)$), telle que $i = i'$, $j = j'$ et $k' = j - 1$. On est donc amené à écrire : $t_{i_6}(i,j) > t_{i_3}(j)$ et $t_{i_6}(i,j) > t_{i_5}(i,j,j-1)$. En faisant cela avec toutes les instructions et en développant les formules en t_S et toutes les dépendances, on obtient un système d'inéquations linéaires, du genre : $t_{i_6}(i,j) - t_{i_3}(j) > 0$. En appliquant le lemme de Farkas sur chacune des inéquations, on peut les transformer en équations, et se débarrasser des indices pour finalement retrouver un système d'inéquations que l'algorithme du simplexe permet de résoudre et qui donne les λ que nous cherchions. Tous ces calculs sont effectués à l'aide PIP [37]

Il n'y a donc aucune raison pour que le cadencement proposé par PAF soit le même que celui proposé par OPERA (et réciproquement) vu la différence des techniques sous-jacentes, la plus importante étant qu'OPERA opère sur les variables du PARE, d'où moins de calculs à effectuer, alors que PAF opère sur les instructions du programme, d'où une fonction de cadencement pour chaque instruction.

4.3.2 Placement

Comme on l'a déjà remarqué, les fonctions de placement diffèrent elles aussi. Avec OPERA, on place clairement des données, ce qui fait que le programme se place de facto dans le contexe de la *owner computes rule*. Mais il faut bien se rappeler qu'il s'agit de programmes écrits en assignation unique et que la règle qui s'applique n'est pas celle à laquelle on peut penser intuitivement : ce n'est pas forcément un même processeur qui va effectuer l'ensemble des calculs permettant d'obtenir un certain $l(i,j)$. Prenons la figure 4.16 : un certain $l(i,j)$ est calculé via $M[i,j,k]$ pour des valeurs de k successives. Pour respecter la règle des écritures locales au sens du programme initial, il faudrait donc projeter selon (0,0,1), ce qui nous donne le premier schéma de la figure 4.17. Mais on peut très bien choisir une allocation dans le sens de (0,1,0) illustrée par le second schéma, qui a une structure de communication plus régulière. Chaque $M[i,j,k]$ est bien calculé par un seul processeur, mais plusieurs participent néanmoins au calcul de l'ensemble de $M[i,j,k]$ pour un i et un j fixé.

On notera que l'on est obligé de donner soi-même le placement : l'outil n'en choisit pas un pour nous. Néanmoins, diverses stratégies permettant de choisir le meilleur placement ont

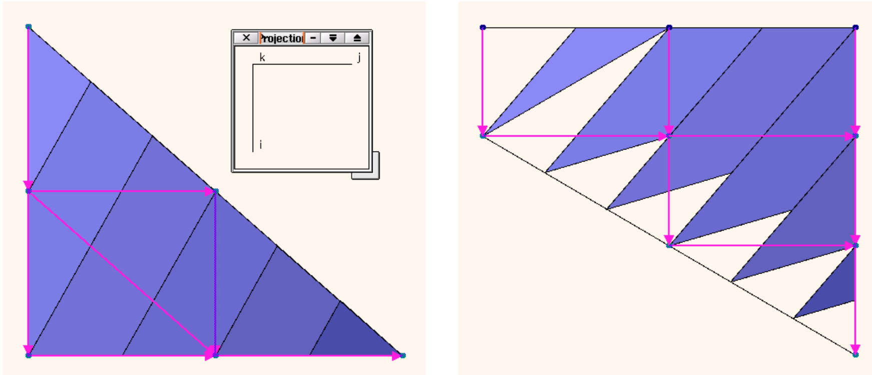


FIG. 4.17 – Allocation suivant le vecteur (0,0,1) puis (0,1,0)

été décrites [75] : on peut construire un arbre des solutions pour le placement. La racine comporte les contraintes de validité, typiquement le fait qu'on ne peut allouer sur un même processeur des calculs qui ont lieu à la même date. On crée ensuite des successeurs à un nœud en ajoutant des contraintes supplémentaires à celle existantes, pour chaque dépendance (qui créent ainsi des niveaux dans l'arbre) afin de tenir compte du volume des communications générées et de leur type (de voisin à voisin ou distantes). On garde le nœud ainsi créé s'il existe une solution respectant toutes les contraintes, et on peut au final choisir le placement qui nous convient.

Pour PAF, on explore des possibilités de placement pour les instructions. Le placement obtenu dans notre cas ainsi que les communications qui peuvent en résulter sont illustrés par la figure 4.18

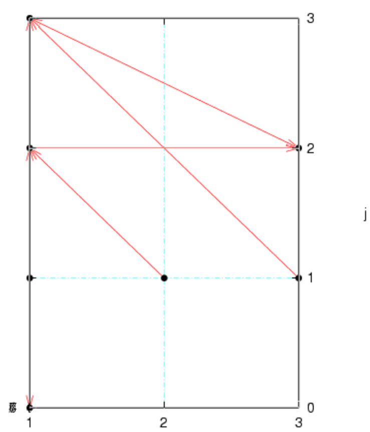


FIG. 4.18 – Projections suivant l'allocation

On obtient là-encore un placement différent parce qu'on place des instructions et des données et non pas des données seules. La technique utilisée [36] consiste à écrire le maximum d'équations décrivant que l'on souhaite le minimum de communications, c'est-à-dire que l'on souhaite que le processeur qui exécute une instruction possède les données utiles localement. Afin de pouvoir automatiser les calculs, on cherche des fonctions de placement affines. Enfin, pour éviter de trouver une solution triviale comme celle qui consisterait à mettre tout sur le même processeur, ce qui minimise bien les communications mais est très peu satisfaisant si on souhaite générer du parallélisme, on oublie certaines contraintes, en fonction des communications qu'elles génèrent.

On notera que l'allocation donnée par PAF se situe “au milieu” de celles qu'on obtient avec OPERA en terme de nombre de processeurs : l'allocation suivant (0,0,1) demande six processeurs, celle suivant (0,1,0) en demande neuf et celle de PAF sept. En terme de communications résiduelles, l'allocation de PAF génère toutes les communications qui ne sont pas verticales (selon l'axe z). On peut en compter le volume.

La dépendance entre i_1 et i_3, ou entre i_2 et i_3 génère N communications au total (on communique une donnée à chaque fois, N fois) ; les dépendances entre i_3 et i_6 génèrent 2 communications pour chaque instruction i_3, leur nombre total étant $\sum_{i=N-1}^{i=1} i$, d'où $N^2 - N$ communications au total. Les dépendances entre i_6 et i_2 ne génèrent pas de communications vu l'allocation. Enfin, les dépendances entre i_6 et i_5 génèrent à chaque fois 2 communications pour un nombre total $T1$ d'itérations :

$$
\begin{aligned}
T1 &= \sum_{j=1}^{j=N} \sum_{i=j+1}^{i=N} \sum_{k=1}^{k=j-1} 1 \\
&= \sum_{j=1}^{j=N} \sum_{i=j+1}^{i=N} j-1 \\
&= \sum_{j=1}^{j=N} (j-1) * (N-j) \\
&= -\sum_{j=1}^{j=N} N + (N+1) \sum_{j=1}^{j=N} j - \sum_{j=1}^{j=N} j^2 \\
&= -N^2 + \frac{1}{2} N(N+1)^2 - \frac{1}{6} N(2N+1)(N+1) \\
&= \frac{1}{6}(N^3 - 3N^2 - 3N)
\end{aligned}
$$

Au total, on a donc $\frac{1}{3}N^3 - N$ communications.

Pour OPERA, si on choisit l'allocation suivant (0,0,1), on garde les communications qui ne sont pas suivant k, donc on supprime les dépendances de $M(i,j,k-1)$ vers $M(i,j,k)$. Il reste donc les 2 communications pour le calcul $M(i,j,k) = M(i,j,k-1) - M(i,k,k) *$

$M(j,k,k)$, pour un nombre total d'itérations $T2$:

$$
\begin{aligned}
T2 &= \sum_{j=1}^{j=N} \sum_{i=j}^{i=N} \sum_{k=1}^{k=j-1} 1 \\
&= \sum_{j=1}^{j=N} \sum_{i=j}^{i=N} j-1 \\
&= \sum_{j=1}^{j=N} (j-1) * (N+1-j) \\
&= -\sum_{j=1}^{j=N} (N+1) + (N+2) \sum_{j=1}^{j=N} j - \sum_{j=1}^{j=N} j^2 \\
&= -N^2 - N + \frac{1}{2} N(N+1)(N+2) - \frac{1}{6} N(2N+1)(N+1) \\
&= \frac{1}{6}(N^3 - N)
\end{aligned}
$$

ainsi que la communication pour le calcul de $M(i,j,k) = M(i,j,k-1)/M(j,j,j)$ pour un nombre $T3$ d'itérations :

$$
\begin{aligned}
T3 &= \sum_{j=1}^{j=N} \sum_{i=j}^{i=N} 1 \\
&= \sum_{j=1}^{j=N} (N-j+1) \\
&= N(N+1) - \frac{1}{2} N(N+1) \\
&= \frac{1}{2} N^2 + \frac{1}{2} N.
\end{aligned}
$$

Au total, on a donc : $\frac{1}{3}N^3 + \frac{1}{2}N^2 + \frac{1}{6}N$ communications. L'allocation de PAF est donc meilleure dans ce cas, bien qu'elle génère plus de processeurs virtuels.

Si on choisit (0,1,0) comme vecteur d'allocation, on a N communications pour le calcul $M(i,j,k) = sqrt(M(i,j,k-1))$, $T3$ communication pour le calcul de $M(i,j,k) = M(i,j,k-1) - M(i,k,k) * (j,k,k)$ (à cause des dépendances de k vers $k-1$), on a $T3$ communications pour le calcul de $M(i,j,k) = M(i,j,k-1) - M(i,k,k) * (j,k,k)$ (toujours à cause des dépendances en de $k-1$ vers k). La dépendance entre $M(i,j,k)$ et $M(i,k,k)$ ne génère pas de communication puisque c'est une dépendance dans le sens de j. la dépendance entre $M(i,j,k)$ et $M(j,k,k)$ ne génère pas toujours une communication, car quand on a $i = j$, il n'y a pas de communications, d'où un nombre de communications égal à $T4$:

$$\begin{aligned} T4 &= \sum_{j=1}^{j=N} \sum_{i=j+1}^{i=N} \sum_{k=1}^{k=j-1} 1 \\ &= T1 \\ &= \frac{1}{6}(N^3 - 3N^2 - 3N). \end{aligned}$$

On peut calculer le nombre total de communications $T1 + T2 + T3 + N$ qui nous donne $\frac{1}{3}N^3 + \frac{5}{6}N$. Ce cadencement, le meilleur qu'on puisse obtenir, est encore un peu moins bon que celui obtenu par PAF. On remarque néanmoins qu'il génère des communications plus régulières.

On notera aussi que l'allocation donnée par PAF n'utilise pas la colonne de processeurs en $i = 1$. Ces processeurs ont d'ailleurs été éliminés sur les schémas 4.17 et 4.18 et ne sont pas comptabilisés dans le nombre de processeurs utiles. Il serait aussi possible d'améliorer sensiblement l'allocation en supprimant l'utilisation du processeur $(0, 0)$ et en exécutant i_3 sur le processeur $(0, j)$: on éviterait les communications dûes aux dépendances entre i_2 et i_3.

Notons pour finir que cette allocation est purement virtuelle dans le sens qu'il s'agit d'un placement sur des processeurs virtuels. Il faut ensuite mettre ces processeurs virtuels sur des processeurs réels. Diverses techniques sont envisageable, suivant le niveau où on se place. Ainsi, un procédé de tuilage est envisageable, comme celui décrit dans [18] qui permettrait de découper le domaine défini après l'allocation de façon à regrouper des zones rectangulaires de processeurs virtuels en un super-processeur virtuel : on supprime ainsi les communications intra-tuile et si on avait un bon placement au départ au niveau du volume de communication, notre résultat se répercute sur les communications inter-tuiles dans la mesure où il s'agit de restes de communications inter-processeurs. Il faut ensuite placer ces super-processeurs virtuels sur des processeurs réels et c'est généralement une distribution cyclique qui s'avère un bon choix au niveau de l'équilibrage de charge. Les tuiles formant déjà des blocs, on obtient finalement une distribution bloc-cycliques, comme celle utilisée dans les bibliothèques parallélisées.

Une autre façon de restreindre le nombre de processeurs virtuels consiste à utiliser des techniques d'allocation non-linéaire, par sérialisation : on constate en effet que tous les processeurs virtuels définis par notre cadencement ne sont pas tous actifs au même instant, et même que certains ne sont plus actifs après une certaine date alors que d'autres ne sont pas actifs avant une certaine date. On peut donc limiter le nombre de processeurs en réutilisant des processeurs devenus inactifs au lieu de prendre de nouveaux processeurs. [24, 22] décrit une technique particulière de sérialisation pour la décomposition de Cholesky qui permet de garder une bonne régularité dans le réseau obtenu, ce qui pourrait permettre ensuite de placer plus facilement les processeurs virtuels résultant sur une grille de processeurs réels. On peut ainsi imaginer "tuiler" le domaine résultant : on obtient directement une sorte de distribution bloc-cyclique puisqu'à l'intérieur de nos tuiles, les processeurs virtuels sont actifs plusieurs fois.

On peut aussi remettre cette étape à la génération de code : ainsi, si on génère du code HPF, il suffit de définir un *template* sur lequel aligner nos données et distribuer ce template par bloc cyclique, ce qui permet généralement d'atteindre un bon équilibrage de charge sur ce type d'algorithme. C'est une autre façon de décrire la première solution envisagée.

4.4 Conclusion

En l'état actuel des choses, on peut dire que les deux outils ont des utilisations différentes. Grâce à son interface graphique et à la possibilité de modifier et visualiser de nombreux paramètres, OPERA s'avère très utile pour étudier un programme et ces différentes parallélisations. Notons aussi qu'il propose tout de même de bonne parallélisations, limitées bien évidemment par le modèle sous-jacent utilisé, et que les réflexions connexes ont permis un travail sur les communications en profondeur.

PAF s'avère être plus proche du paralléliseur automatique : programme FORTRAN en entrée, possibilité de code parallèle en sortie. L'approche par instruction plutôt que par les données donne une complexité supérieure à l'approche et génère de meilleurs résultats. Par contre, il faut se satisfaire de la solution proposée et le non-suivi de la règle des écritures locales complexifie la génération de code. On remarque que si on se place dans un contexte de machine à mémoire distribuée, il est plus simple de suivre cette règle, mais que cela risque de générer des communications supplémentaires.

Pour finir, d'autres travaux autour d'outils permettant d'aboutir à une parallélisation automatique ont été menés. On peut tenter de les regrouper en diverses catégories qui ne sont pas totalement disjointes les unes des autres.

La parallélisation automatique d'un programme comporte diverses phases : il faut tout d'abord transformer le programme de départ, donné dans un langage de plus ou moins haut niveau (C, FORTRAN, Java, ALPHA) en une structure qui sera utilisable pour l'analyser. Puis il faut faire l'analyse elle-même : analyser les dépendances, déterminer allocation et cadencement. Enfin, il faut générer le code parallèle.

Certains outils se proposent de donner un cadre pour réaliser toutes ces étapes. C'est le cas de Nestor [100] qui est un outil permettant de manipuler des programmes FORTRAN en fournissant une représentation interne d'un programme basée sur une conception orientée objet. Le pendant pour langage C de Nestor est SUIF [107, 66] développé au Computer System Laboratory de l'université de Stanford, dont la deuxième version est disponible depuis 1999. L'analyse des programmes C en vue de leur parallélisation est plus complexe que celle des programmes FORTRAN, en particulier à cause de l'utilisation des pointeurs qui rend l'analyse des programmes difficile. SUIF propose un environnement qui permet de représenter un programme et d'en faire l'analyse par un système de passes et il permet l'analyse interprocédurale. On peut aussi parler de PIPS [53] de l'Ecole des Mines de Paris qui offre un environnement permettant d'implémenter des techniques d'analyse, de parallélisation ou d'optimisation de programmes. Dès le départ, son grand avantage a été de faire de l'analyse interprocédurale, ce qui le rend proche des vraies applications, surtout celles de calcul numérique puisqu'il supporte seulement FORTRAN (ou HPF) en entrée.

Ces environnements qui proposent un cadre pour développer des compilateurs, proposent aussi des techniques de parallélisation, ce qui est assez logique : une fois qu'on a un tel environnement, pourquoi ne pas l'utiliser ? Ainsi, Nestor fournit deux types de parallélisations. La première est basée sur la reconnaissance des boucles directement parallélisables c'est-à-dire les boucles séquentielles qu'on peut transformer en FORALL. La seconde, plus complexe repose sur l'algorithme d'Allen et Kennedy [2]. De même, SUIF propose des techniques de parallélisation [82, 67] ou d'optimisation de la localité des données [5] basées sur les transformations affines. Dans PIPS, c'est l'algorithme d'Allen et Kennedy qui est utilisé comme base de la parallélisation, mais il est aussi possible d'obtenir du parallélisme à gros grain ainsi que d'exhiber du parallélisme par des opérations sur les boucles.

Notons que ces outils s'appuient parfois eux-mêmes sur d'autres outils pour certains de leur calculs : ainsi, le code FORTRAN nécessaire à Nestor est analysé par Petit [86] afin de fournir le graphe de dépendances du programme. D'autre part, PIPS se repose sur C3Linear, une bibliothèque construite en partie via une traduction en FORTAN de PIP[37], pour tout ce qui concerne les calculs d'inéquations ou les manipulations de matrices et vecteurs. Dans le même genre, on trouve la Polylib [90] pour ce qui est manipulation de polyèdres, ou Omega [84], sur lequel reponse Petit, d'ailleurs et qui permet de manipuler des ensembles d'entiers de façon symbolique.

On peut ensuite s'intéresser aux développements plus tournés vers l'implémentation de stratégies de parallélisation, comme Loopo [44] ou Bouclettes [13]. Loopo a pour but d'implémenter des méthodes de transformations espace-temps, basées sur le travail de Christan Lengauer [65] sur les boucles. Ce sont ajoutées par la suite des techniques permettant de prendre en compte les structures `WHILE` ou de tenir compte des opérateurs conditionnels. Le travail porte dorénavant sur la génération efficace de code HPF pour ces programmes. Bouclettes, développé au LIP, s'intéresse aussi à la parallélisation des nids de boucles et la genération de code HPF. Les algorithmes utilisés pour déterminer un cadencement et une allocation sont décrits dans [28, 29]. Les techniques se situent entre celles d'OPERA et PAF : on a moins de liberté au niveau du cadencement que dans PAF, mais il s'agit néanmoins d'un cadencement par instruction. Sur l'allocation aussi on a moins de liberté : allocation identique pour toutes les instructions à une constante prêt. A partir de là, du code HPF peut être généré [14]. OPERA et PAF se situent aussi dans cette catégorie d'outils.

De nombreux outils ont donc vu le jour, visant à la parallélisation automatique de programmes. Néanmoins, il s'agit le plus souvent d'outils visant à valider une certaine approche de la parallélisation, certains algorithmes. À des fins de comparaisons des diverses techniques, il serait intéressant de voir dans quelle mesure cette multitude d'approches pourrait être unifiée à l'intérieur d'outils spécifiquement développés pour l'application de techniques, comme ceux que nous avons présenté plus avant.

Conclusion

- *Qu'est-ce qui est plus intelligent qu'un ordinateur ?*
- `Deux ordinateurs`

Philipe Gelück, Ma langue au Chat

Nous avons étudié divers outils et techniques de parallélisation et de transformation sur divers problèmes, en restant dans le cadre des nids de boucles, le but étant de voir l'intérêt des approches théoriques pour des problèmes plus ou moins concrets qui apparaissent dans les programmes de calculs numériques.

Les méthodes polyédriques se sont révélées efficaces sur plusieurs terrains : optimisation de programmes, que ce soit dans le cadre de la résolution des équations de Navier-Stokes pour la modélisation et la transformation de nids de boucles dans un but d'optimisation, ou alors dans la parallélisation de la décomposition de Cholesky où la solution retournée supporte bien la comparaison avec d'autres outils. L'intérêt du modèle réside dans la modélisation mathématique de la transformation de programme et qui permet l'automatisation de cette transformation, via l'utilisation d'objets mathématiques comme les polyèdres. Ceci permet d'obtenir une parallélisation, c'est-à-dire une allocation et un cadencement, et aussi d'appliquer cette transformation et de générer du code [93]. D'autre part, c'est aussi cette modélisation qui permet l'obtention de code dans lequel on optimise les accès à la mémoire, temporellement ou spatialement, puisque c'est grâce à cette modélisation qu'on peut calculer des ensembles de points à parcourir, calculer des bornes de boucles, etc. De ce point de vue-là, il s'agit réellement d'un outil puissant.

Néanmoins, ce modèle supporte difficilement les programmes comportant des irrégularités : les programmes creux et les accès irréguliers (par indirections) aux tableaux qu'ils entraînent ne permettent pas d'être représentés et traités dans ce contexte. C'est ici que des méthodes plus formelles de représentation et transformation de programmes peuvent faire valoir leurs avantages. Ainsi, un programme numérique est bien souvent constitué d'opérations sur des matrices (tableaux à deux dimensions) qui concourrent à un calcul commun en combinant leurs éléments de façon correcte. On peut dire que c'est cette approche générale que Pei a choisi, via le concept *d'alignement* qui le sous-tend. Cela permet de décrire bon nombre de nœuds de calcul des programmes de calculs numériques. Nous avons décrit dans ce cadre le produit matrice-vecteur, dont on peut obtenir relativement facilement une version creuse et une parallélisation. Le cœur de l'approche est *l'alignement* des données, dans notre cas l'alignement d'un vecteur sur une matrice. Cette approche centrée sur les données mène tout naturellement à la parallélisation et permet une identification forte avec le langage HPF, ce qui fait de notre point de vue une grande part de son intérêt.

D'autre part, l'utilisation de l'objet mathématique "fonction" pour modéliser les diverses opérations permet la représentation d'une large palette de programmes. On se heurte alors à la difficulté de calculer réellement les compositions de fonctions nécessaires à la description et à la transformation d'un programme, notamment lorsqu'elles ne sont pas affines ou s'il s'agit de fonctions définies par morceaux. On peut raisonnablement penser qu'un interfaçage avec un logiciel de calcul formel permettrait de résoudre une bonne partie de ces problèmes.

D'une certaine façon, les deux outils au cœur de notre étude sont complémentaires en ce qui concerne la parallélisation de programmes. L'un permet d'étudier un programme et d'en obtenir une parallélisation, alors que le second fournit un cadre pour appliquer la transformation, c'est-à-dire se rapprocher d'un code parallèle utilisable. Notons aussi que la modélisation par ensembles d'utilisation / d'émission rapproche OPERA d'un modèle d'exécution sur machine à mémoire distribuée, c'est-à-dire où on raisonne en terme de communications, comme on a pu le constater sur le cas de la parallélisation de la décomposition de Cholesky. PEI permet d'avoir une autre vision, car la notion d'alignement qui le sous-tend lui permet de plus se rapprocher des machines à mémoire partagée, puisque la notion de communications est évacuée au profit simplement de "données utilisées simultanément", ce qui permet au final de couvrir un large panel de modèles d'exécution.

Un autre point commun à ces deux outils est leur approche centrée sur les données, plutôt que sur les instructions. Le gros avantage de cette approche est de permettre l'étude et la compréhension des programmes. L'approche data-parallèle d'OPERA permet une visualisation utile dans le cadre de l'étude de programmes en vue de leur parallélisation en faisant ressortir les régularités, ce qui permet une bonne compréhension du problème, comme nous l'avons montré en ce qui concerne l'étude de la parallélisation de Cholesky ou encore l'étude du nid de boucles à paralléliser dans le problème des équations de Navier-Stokes. Le comportement des programmes est parfaitement compréhensible : on peut voir les dépendances entre données, donc les possibles communications et choisir en fonction de cela une parallélisation. De la même façon, l'approche centrée sur les données de PEI permet de bien comprendre quels sont les mouvements de données en jeu et la parallélisation d'un programme devient alors évidente.

Cette approche est néanmoins limitée puisqu'elle ne permet pas d'obtenir les meilleures solutions pour la parallélisation d'un programme. Cependant, la notion d'optimalité reste floue dans le cadre du nid de boucles remis dans le contexte de l'application complète, comme on l'a vu : d'autres facteurs, comme l'environnement peuvent influer sur le comportement du programme.

Ce qui manque néanmoins à ces approches pour en faire des outils réellement proches de l'utilisateur dans le cadre de la parallélisation est la prise en compte de la réalité physique des machines sur lesquelles un programme est amené à s'exécuter. Sans parler des caractéristiques spécifique des architectures de chaque machine parallèle existante, on peut regretter que nulle méthode ne soit proposée afin de placer les processeurs virtuels sur des proceseurs réels. C'est pourtant une question importante pour l'utilisateur final.

Si on a une solution parallèle dont les communications sont minimisées d'une façon ou d'une autre, mais que le placement des processeurs virtuels sur les processeurs réels est fait n'importe comment, l'intérêt de cette optimisation s'en trouve réduit.

Plus globalement, le modèle data-parallèle sous-jacent aux approches étudiées a les inconvénients de ses avantage : il reste cantonné à une certaine régularité, ce qui ne permet pas forcément d'obtenir une solution optimale, ni de modéliser tous les problèmes. On trouve la régularité à plusieurs niveaux : restriction aux dépendances affines, structures de contrôle limitées aux boucles, hypothèse que chaque calcul a un coût unitaire. Ceci restreint son champ d'application, mais aussi le type d'exécution visée. Ainsi, l'intérêt de cette approche dans le cadre du metacomputing, c'est-à-dire l'exécution d'applications sur différents sites resterait à étudier. Si les applications sont susceptible de présenter des régularités et d'être analysables, c'est le support d'exécution qui présente désormais des irrégularités. On se trouve à l'opposé des réseaux systoliques qui ont vu naître l'approche data-parallèle ou les méthodes polyédriques.

Il n'empêche que ce mode de raisonnement reste très efficace pour ce qui est de la représentation des programmes en vue de leur compréhension. On notera aussi qu'une approche centrée sur les données s'avère efficace non seulement dans le cadre de la parallélisation, mais aussi pour ce qui est de l'optimisation des accès aux données pour des programmes séquentiels ou parallèles.

Bibliographie

[1] ADLE R. *Outils de parallélisation automatique pour des programmes denses pour les structures creuses*. Thèse de doctorat, Université d'Evry Val d'Essonne, 1999

[2] ALLEN R. et KENNEDY K. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4) : 491–542, octobre 1987

[3] AMESTOY P.R., DAVIS T.A. et DUFF I.S. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4) : 886–905, 1996

[4] AMESTOY P.R., DUFF I.S. et L'EXCELLENT J.Y. Mumps multifrontal massively parallel solver version 2.0. Rapport technique TR/PA/98/02, CERFACS, 1998

[5] ANDERSON J., AMARASINGHE S. et LAM M. Data and computation transformation for multiprocessors. Dans *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Santa Barbara, CA, juillet 1995

[6] ARNOLDI W. The principle of minimized iteration in the solution of of the matrix eigenvalue. *Quart. Appl. Math.*, 9 : 17–29, 1951

[7] BALAY S., GROPP W., MCINNES L.C. *et al. Modern Software Tools in Scientific Computing,*, chapitre Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries, pp. 163–202. Birkhauser Press, 1997

[8] BANERJEE U. *Loop Transformations for Restructuring Compilers : The Foundations*. Norwell, Mass. : Kluwer Academic Publishers, 1993

[9] BASERMANN A. Conjugate gradient and Lanczos methods for sparse matrices on distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1) : 46–52, août 1997

[10] BIK A.J.C., KNIJNENBURG P.M.W. et WIJSHOFF H.A.G. Reshaping access patterns for generating sparse codes. Dans *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing* (edité par K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau et D. Padua), Lecture Notes in Computer Science, pp. 406–420. Springer-Verlag, Ithaca, New York, août 8–10, 1994

[11] BIK A.J.C. et WIJSHOFF H.A.G. Advanced compiler optimizations for sparse computations. *J. of Parallel and Distributed Computing*, 31 : 14–24, 1995

[12] BIK A.J.C. et WIJSHOFF H.A.G. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2) : 109–126, février 1996. Also on LNCS

[13] Boulet P. Bouclettes : A Fortran loop parallelizer. *Lecture Notes in Computer Science*, 1067 : 784–791, 1996

[14] Boulet P. et Dion M. Code generation in bouclettes. Rapport technique 95-43, Laboratoire de l'Informatique et du Parallélisme, ENS Lyon, 1995

[15] Bozkus Z., Meadows L., Nakamoto S. *et al.* PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1) : 29–40, 1997

[16] Brandes T., Brégier F., Counilh M.C. *et al.* Contribution to better handling of irregular problems in HPF2. Dans *Europar'98* (edité par Springer-Verlag), tome 1470 de *Lecture Notes in Computer Science*. septembre 1998

[17] Brandes T. et Zimmermann F. Adaptor — A transformation tool for HPF programs. Dans *Programming environments for massively parallel distributed systems : working conference of the IFIP WG10.3, April 25–29, 1994, Ascona, Italy* (edité par K.M. Decker et R.M. Rehmann), pp. 91–96. Birkhäuser Boston Inc., Cambridge, MA, USA, 1994

[18] Calland P., Dongarra J. et Robert Y. Tiling with limited resources. Dans *IEEE International Conference on Application-Specific Systems, Architectures and Processors : proceedings, Juillet 14–16, 1997, Zurich, Switzerland* (edité par L. Thiele), pp. 229–238. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997

[19] Carte G., Dusek J. et Fraunié P. A spectral time discretization for flows with dominant periodicity. *Journal of Computational Physics.*, 120 : 171–183, 1995

[20] Choi J., Demmel J., Dhillon I. *et al.* LAPACK working note 95 : ScaLAPACK : A portable linear algebra library for distributed memory computers – design issues and performance. Rapport technique UT-CS-95-283, Department of Computer Science, University of Tennessee, mars 1995

[21] Choi J., Dongarra J.J. et Walker D.W. The design of scalable software libraries for distributed memory concurrent computers. Dans *Proceedings : Eighth International Parallel Processing Symposium, April 26–29, 1994, Cancun, Mexico* (edité par H.J. Siegel), pp. 792–799. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994

[22] Clauss P. Efficient and regular mappings od affine recurrences by serializing. Rapport technique 95-14, ICPS, 1995

[23] Clauss P. et Meister B. Automatic memory layout transformation to optimize spatial locality in parameterized loop nests. Dans *4th Annual Workshop on Interaction between Compilers and Computer Architectures, INTERACT-4, Toulouse (France)*. janvier 2000

[24] Clauss P. et Perrin G.R. Optimal mapping of systolic algorithms by regular instruction shifts. Dans *International conference on Application Specific Array Processors, ASAP'94* (edité par E.E.S. P. Cappello, R.M. Owens et B.W. Wah), pp. 224–235. IEEE Computer Society Press, août 1994

[25] COLLARD J.F. Analyse des définitions visibles : État de l'art et applications en parallélisation automatique. *Techniques et Sciences Informatiques*, mai 1999

[26] Construction systématique de programmes parallèles et distribués. World Wide Web Document : `http://www.prism.uvsq.fr/french/parallel/paf/autom_fr.html`

[27] DAGUM L. et MENON R. OpenMP : An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1) : 46–55, janvier/mars 1998

[28] DARTE A. et ROBERT Y. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems*, 5(8) : 814–822, août 1994

[29] DARTE A. et ROBERT Y. On the alignment problem. *Parallel Processing Letters*, 4(3) : 259–270, septembre 1994

[30] DELAPLACE F. et ADLE R. Extension of the dependence analysis for sparse computations. Dans *Parallel & Distributed Computing Systems*. octobre 1997

[31] DUFF I.S., ERISMAN A.M. et REID J.K. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986

[32] DUMITRESCU B., DOREILLE M., ROCH J.L. *et al.* Two-dimensional block partitionings for the parallel sparse Cholesky factorization. *Numerical Algorithms*, 16(1) : 17–38, 1997. Sparse matrices in industry (Lille, 1997).

[33] DUSEK J., LE GAL P. et FRAUNIÉ P. A numerical and theoretical study of the first Hopf bifurcation in a cylinder wake. *J. Fluid Mech.*, 264 : 59–80, 1994

[34] FEAUTRIER P. Some efficient solutions to the affine scheduling problem. I. one-dimensional time. *International Journal of Parallel Programming*, 21(5) : 313–347, octobre 1992

[35] FEAUTRIER P. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *International Journal of Parallel Programming*, 21(6) : 389–420, décembre 1992

[36] FEAUTRIER P. Placement automatique des données et des calculs. *Technique et Sciences Informatiques*, 15 : 529–558, 1996

[37] FEAUTRIER P. et TAWBI N. Résolutions de système d'inéquations linéaires : mode d'emploi du logiciel PIP. Rapport technique 90-2, Institut Blaise Pascal, Laboratoire MASI, 1990

[38] FLETCHER R. Conjugate gradient methods for indefinite systems. *Lecture Notes in Mathematics*, 506 : 73–89, 1976

[39] GEIST A., BEGUELIN A., DONGARRA J. *et al.* *PVM : Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994

[40] GENAUD S. On deriving HPF code from PEI programs. Rapport technique RR97-04, ICPS, septembre 1997

[41] GENAUD S. *Transformations de programmes Pei : applications au parallélisme de données*. Thèse de doctorat, Université Louis Pasteur, 1997

[42] GEORGE A. et LIU J.W.H. The evolution of the minimum degree ordering algorithm. *SIAM Rev.*, 31(1) : 1–19, 1989

[43] GOTTLIEB D. et ORSZAG S.A. *Numerical analysis of spectral methods : theory and applications*. Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1977. CBMS-NSF Regional Conference Series in Applied Mathematics, No. 26

[44] GRIEBL M. et LENGAUER C. The loop parallelizer LooPo — announcement. *Lecture Notes in Computer Science*, 1239 : 603–, 1997

[45] GROPP W., MCINNES L. et SMITH B. The PETSc 2.0 users manual. Rapport technique ANL-95/11, Argonne National Laboratory, juin 1995

[46] GUTIÉRREZ E., ASENJO R., PLATA O. *et al.* Automatic parallelization of irregular applications. *Parallel Computing*, 26(13–14) : 1709–1738, décembre 2000

[47] HENDERSON R.D. *In High-Order Methods for Computational Physics*, chapitre Adaptive spectral elements methods for turbulence and transition, pp. 225–324. Springer, 1999

[48] HENDRICKSON B. et LELAND R. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2) : 452–469, mars 1995

[49] HENDRICKSON B. et LELAND R. A multi-level algorithm for partitioning graphs. Dans *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA* (edité par S. Karin). ACM Press and IEEE Computer Society Press, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995

[50] HENDRICKSON B. et ROTHBERG E. Improving the run time and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20(2) : 468–489 (electronic), 1999

[51] HPF FORUM. *High Performance Fortran Language Specification*, janvier 1997. Version 2.0

[52] HUTCHINSON S.A., PREVOST L.V., SHADID J.N. *et al.* *Aztec User's Guide : Version 2.0*, septembre 1998

[53] IRIGOIN F., JOUVELOT P. et TRIOLET R. Semantical interprocedural parallelization : An overview of the PIPS project. Dans *International Conference on Supercomputing, Cologne*. juin 1991

[54] KARP R.M., MILLER R.E. et WINOGRAD S. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3) : 563–590, juillet 1967

[55] KARYPIS G. et KUMAR V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1) : 359–392, janvier 1999

[56] KELLY W. et PUGH W. A unifying framework for iteration reordering transformations. Technical Report CS-TR-3430, University of Maryland, College Park, février 1995

[57] KOELBEL C.H. *The High Performance Fortran handbook*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, janvier 1994

[58] KOTLYAR V., PINGALI K. et STODGHILL P. Compiling parallel code for sparse matrix applications. Dans *SC'97 : High Performance Networking and Computing :*

Proceedings of the 1997 ACM/IEEE SC97 Conference : November 15–21, 1997, San Jose, California, USA. (edité par ACM). ACM Press and IEEE Computer Society Press, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997

[59] Kotlyar V., Pingali K. et Stodghill P. Compiling parallel sparse code for user-defined data structures. Dans *Proceedings of Eights SIAM Conference on Parallel Processing for Scientific Computing*. Minneapolis, mars 1997

[60] Kotlyar V., Pingali K. et Stodghill P. A relational approach to the compilation of sparse matrix programs. *Lecture Notes in Computer Science*, 1300 : 318–, 1997

[61] Kumar V., Grama A., Gupta A. *et al.* *Introduction to Parallel Computing : Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994

[62] Lamport L. The parallel execution of DO loops. *Communications of the ACM*, 17(2) : 83–93, février 1974

[63] Lascaux P. et Théodor R. *Analyse numérique matricielle appliquée à l'art de l'ingénieur. Tome 1*. Masson, Paris, deuxième édition, 1993

[64] Lascaux P. et Théodor R. *Analyse numérique matricielle appliquée à l'art de l'ingénieur. Tome 2*. Masson, Paris, deuxième édition, 1994. Méthodes itératives. [Iterative methods]

[65] Lengauer C. Loop parallelization in the polytope model. Dans *CONCUR '93 : 4th International Conference on Concurrency Theory* (edité par E. Best), tome 715 de *Lecture Notes in Computer Science*, pp. 398–416. Springer-Verlag, Hildesheim, Germany, 23–26août 1993

[66] Liao S.W., Diwan A., Bosch R.P. *et al.* SUIF Explorer : an interactive and interprocedural parallelizer. *ACM SIGPLAN Notices*, 34(8) : 37–48, août 1999

[67] Lim A.W. et Lam M.S. Maximizing parallelism and minimizing synchronization with affine transforms. Dans *The 24th ACM Sigplan-Sigact Symposium on Principles of Programming Languages (POPL '97)*, pp. 201–214. Association for Computing Machinery, New York, janvier 1997

[68] Lipton R.J., Rose D.J. et Tarjan R.E. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16(2) : 346–358, 1979

[69] Liu J.W.H. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1) : 134–172, 1990

[70] Liu J.W.H. The multifrontal method for sparse matrix solution : theory and practice. *SIAM Rev.*, 34(1) : 82–109, 1992

[71] Liu W.H. et Sherman A.H. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.*, 13(2) : 198–213, 1976

[72] Lo G.C. et Saad Y. Iterative solution of general sparse linear systems on clusters of workstations. Rapport technique UMSI 96/117 & UM-IBM 96/24, Minnesota Supercomputing institute, 1996

[73] Loechner V. *Contribution à l'étude des polyèdres paramétrés et applications en parallélisation automatique*. Thèse de doctorat, Université Louis Pasteur, décembre 1997

[74] Loechner V. et Mongenet C. A toolbox for affine recurrence equations parallelization. *Lecture Notes in Computer Science*, 919 : 263–268, 1995

[75] Loechner V. et Mongenet C. Solutions to the communication minimization problem for affine recurrence equations. *Lecture Notes in Computer Science*, 1300 : 328, 1997

[76] Markowitz H.M. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3) : 255–269, 1957

[77] Mauras C. Definition de ALPHA : un langage pour la programmation systolique. Rapport technique RR-1090, Inria, Institut National de Recherche en Informatique et en Automatique, 1989

[78] McKinley K.S., Carr S. et Tseng C.W. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4) : 424–453, juillet 1996

[79] Meister B. Localité de données dans les opérations stencil. Dans *RenPar 2001, Rencontres francophones du parallélisme, Cité des Sciences, La Villette, Paris*. juin 2001

[80] Meister B., Loechner V. et Clauss P. The polytope model for optimizing cache locality. Rapport technique ICPS-RR-00-02, ICPS, 2000

[81] Mongenet C., Clauss P. et Perrin G.R. Geometrical tools to map systems of affine recurrence equations on regular arrays. *Acta Informatica*, 31(2) : 137–160, 1994

[82] Moon S., So B. et Hall M.W. Evaluating automatic parallelization in SUIF. *IEEE Transactions on Parallel and Distributed Systems*, 11(1), janvier 2000

[83] MPIForum. MPI : A message-passing interface MPI forum. Rapport technique CS/E 94-013, Department of Computer Science, Oregon Graduate Institute, mars 94

[84] Omega library and calculator. World Wide Web Document : `http://www.cs.umd.edu/projects/omega/omega-lib.html`

[85] Perrin G.R. et Darte A. *The data parallel programming model : foundations, HPF realization, and scientific applications*, tome 1132 de *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1996

[86] Petit. World Wide Web Document : `http://www.cs.umd.edu/projects/omega/petit.html`

[87] Petiton S. et Emad N. *The data parallel programming model : foundations, HPF realization, and scientific applications*, chapitre A data parallel scientific computing introduction, pp. 45–64. Springer-Verlag Inc., 1996

[88] Petiton S., Saad Y., Wu K. *et al.* Basic sparse matrix computations on the CM-5. *Internat. J. of Modern Physics*, 4 : 65–83, 1993

[89] Pironneau O. *Méthodes des éléments finis pour les fluides*. Masson, 1988

[90] Polylib, a library of polyhedral functions. World Wide Web Document : `http://icps.u-strasbg.fr/PolyLib/`

[91] Quinton P. et Van Dongen V. The mapping of linear recurrence equations ons on regular arrays. *The Journal of VLSI Signal Processing*, pp. 95–113, 1989

[92] Raviart P., Thomas J., Ciarlet P. *et al.* *Introduction à l'analyse numérique des équations aux dérivées partielles*. Sciences sup. Mathématiques. Dunod, 1998

[93] Reffay C. *Génération de code parallèle à partir d'équations récurrentes*. Thèse de doctorat, Université de Franche-Comté, 1996

[94] Rosenkranz G. et Brageul S. Programmation parallèle sous PEI et OPERA de problèmes industriels du CEA. Rapport de stage DESS Mathématiques discrètes, Université Louis Pasteur, 1995

[95] Saad Y. ILUT : a dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1 : 387–402, 1994

[96] Saad Y. *Iterative Methods for Sparse Linear System*. PWS Publishing Company, 1996

[97] Saad Y. et Sosonkina M. Solution of distributed sparse linear systems using PSPARSLIB. *Lecture Notes in Computer Science*, 1541 : 503–509, 1998

[98] Schrijver A. *Theory of Linear and Integer Programming.*. A Wiley-Interscience publication. Wiley, Chichester, New York, 1986

[99] Shadid J.N. et Tuminaro R.S. Sparse iterative algorithm software for largescale MIMD machines : an initial discussion and implementation. *Concurrency, Pract. Exp.*, 4 : 481–497, 1992

[100] Silber G.A. et Darte A. The Nestor library : A tool for implementing Fortran source to source transformations. *Lecture Notes in Computer Science*, (1593) : 653, 1999

[101] Sonnenveld P. CGS, a fast lanczos-type solver for nonsymetric linear solver. *SIAM J. Sci. Stat. Comput*, 10 : 36–52, 1989

[102] Tomboulides A.G., Orszag S.A. et Karniadakis G.E. Direct and large-eddy simulations of axisymmetric wakes. Aerospace Sciences Meeting, 31st, Reno, NV, Jan. 11-14, 1993 A93-23286, AIAA, 1993

[103] Tuminaro R.S., Shadid J.N. et Hutchinson S.A. Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency : Practice and Experience*, 10(3) : 229–247, mars 1998

[104] Violard E. A mathematical theory and its environment for parallel programming. *Parallel Processing Letters*, 4(3) : 313–328, septembre 1994

[105] Voisin F. et Perrin G.R. Sparse computation with PEI. *International Journal of Foundations of Computer Science*, 10(14) : 425–442, 1999

[106] Vorst H.V.D. Bi-CGStab : A fast and smoothly converging variant of Bi-CG for the solution of nonsymetric linear systems. *SIAM J. Sci. Stat. Comput*, 13 : 631–644, 1992

[107] WILSON R., FRENCH R., WILSON C. *et al*. SUIF : A parallelizing and optimizing research compiler. Rapport technique CSL-TR-94-620, Computer Systems Laboratory, Stanford University, mai 1994

[108] WOLFE M. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1986

[109] YANNAKAKIS M. Computing the minimal fill-in is NP-complete. *SIAM Journal of Algebraic and Discrete Methods*, 2 : 77–79, 1981

[110] ZAPATA E.L., PLATA O., ASENJO R. *et al*. Data-parallel support for numerical irregular problems. *Parallel Computing*, 25(13–14) : 1971–1994, décembre 1999