Simulation par elements finis d'ecoulements compressibles sur calculateurs vectoriels

F. Angrand, Jocelyne Erhel

INRIA

CENTRE DE ROCQUENCOURT

Rapports de Recherche

N° 622

SIMULATION PAR ÉLÉMENTS FINIS D'ÉCOULEMENTS COMPRESSIBLES SUR CALCULATEURS VECTORIELS

Françoise ANGRAND
Jocelyne ERHEL

Février 1987

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

SIMULATION PAR ELEMENTS FINIS D'ECOULEMENTS COMPRESSIBLES SUR CALCULATEURS VECTORIELS

FINITE ELEMENT METHODS TO SIMULATE COMPRESSIBLE FLOWS ON VECTOR COMPUTERS

Françoise ANGRAND , Jocelyne ERHEL

Résumé

Les méthodes d'éléments finis sont bien adaptées à la résolution des équations d'Euler ou de Navier-Stokes pour des écoulements compressibles autour de géométries complexes. Elles requièrent un grand nombre d'opérations, aussi doit-on avoir une grande puissance de calcul. L'objet de cette étude est de montrer quelles solutions peuvent être apportées pour accroître le taux de vectorisation des codes utilisant des maillages non structurés. Par exemple, par une technique de coloriage on peut séparer les segments et les éléments en groupes n'ayant pas de sommets communs afin d'éliminer les dépendances des données lors de l'accumulation des résultats aux noeuds. Malgré tout, la nécessité de l'adressage indirect fait décroître les performances gobales de la vectorisation. Les tests sont effectués sur deux schémas différents servant à résoudre les équations de Navier-Stokes et d'Euler bidimensionnelles. Les résultats présentés ont été obtenus sur un CRAY 1-S et sur un CRAY XMP.

Summary

Finite Element methods are well suited to solve Euler or Navier-Stokes equations for compressible flows over complex geometries such as a complete aircraft. They involve many operations, so they require a high power of computing. We discuss vectorization issues related to unstructured meshes. Computations over edges or elements are highly vectorizable. The separation of edges or elements into groups without any common vertex eliminates data dependencies for the accumulation of results at the nodes. But the need of indirect addressing may decrease dramatically the global performance. We study two different schemes to solve 2-D Navier-Stokes and Euler equations, and give results for executions on a CRAY-1S and a CRAY-XMP.

CHAPITRE 1 : INTRODUCTION

L'avènement de super-ordinateurs, qui allient une très grande vitesse d'exécution à une mémoire de très grande taille, ont permis d'appliquer les simulations numériques à de nombreux phénomènes physiques. Les équations continues, qui modélisent un problème physique, sont approchées par des équations discrètes. Pour cela, le domaine d'étude est maillé par une grille de points où sont calculées les inconnues du problème. Diverses techniques d'interpolation fournissent une approximation des dérivées et des intégrales intervenant dans les équations. Généralement, les valeurs d'un champ de variables en un point sont calculées en fonctions des valeurs aux points voisins dans la grille.

Deux types de méthodes numériques, appelées Différences Finies et Eléments Finis, diffèrent notamment par la nature du maillage et la technique d'interpolation. Ces différences s'avèrent cruciales pour l'efficacité de la mise en oeuvre et la vitesse d'exécution, en particulier sur des ordinateurs vectoriels tels que CRAY-1, CRAY-XMP, FUJITSU VP, ETA^{10} [1,2,3].

Les grilles de Différences Finies sont très structurées, avec une numérotation régulière des points et une structure locale identique en chaque point (même nombre de voisins avec même orientation). Par contre, les maillages pour les Eléments Finis sont plus généraux et la structure n'est pas uniforme (nombre variable de voisins et topologie locale variable). Cette difficulté est compensée par une plus grande flexibilité pour discrétiser des géométries complexes, et pour raffiner le maillage localement. Cette souplesse autorise une même précision de l'approximation qu'en Différences Finies, avec moins de points dans le maillage.

Typiquement, les calculs de Différences Finies se font directement aux points, grâce à des équations liant les valeurs entre points voisins. Les indices des n voisins jk ($1 \leq k \leq n$, n constante) d'un point i sont des expressions linéaires simples en fonction de i ($jk = i+1, i-m$, par exemple). Par contre, les méthodes d'Eléments Finis utilisent une dualité entre les points et les éléments du maillage (triangles en 2D et tétraèdres en 3D par exemple). La correspondance entre les éléments et leurs sommets se fait par un tableau d'indices, du fait que la structure du maillage n'est pas uniforme ni régulière. On effectue d'une part des calculs aux éléments, qui font intervenir des valeurs aux sommets

de l'élément, et nécessitent l'usage du tableau de correspondance et d'autre part des calculs aux points. De manière générale, il faut sommer en chaque point les contributions élémentaires de tous les éléments contenant ce point.

Ces caractéristiques se répercutent sur l'adéquation des méthodes aux architectures vectorielles. Les opérations en Différences Finies sont en général vectorielles, les vecteurs étant les tableaux de variables aux points. L'accès aux tableaux se fait à incrément constant (fréquemment égal à un), donc au débit maximal de la mémoire. Les calculs en un point sont en général indépendants des calculs aux autres points et peuvent être exécutés en mode vectoriel. Deux aspects des méthodes d'Eléments Finis en pénalisent les performances. D'une part la dualité éléments/sommets requiert un accès aux tableaux par adressage indirect. Le débit mémoire est ralenti par des appels accrus (lecture de l'indice puis de la variable) et par un flot de données irrégulier (rangement en mémoire non structuré), qui peut occasionner des conflits mémoire, induisant un temps de réponse non déterministe de la mémoire. Sur le CRAY-1, de telles opérations doivent être exécutées en mode scalaire. Si elles sont vectorielles sur le CRAY-XMP, le CRAY-2, le FUJITSU VP, l'ETA^{10}, elles restent toutefois moins performantes que des opérations portant sur des vecteurs de mots contigüs en mémoire. D'autre part, on ne peut accumuler en mode vectoriel les valeurs de deux éléments ayant un sommet commun. Cette dépendance de données empêche la vectorisation de cette étape de calcul.

Diverses applications numériques exploitent de façon très efficace les méthodes de Différences Finies sur les super-ordinateurs [4]. Mais les performances des méthodes d'Eléments Finis sont en général moindres [5]. L'objet de cette étude est de montrer sur des exemples quelles solutions peuvent être apportées pour accroître l'efficacité des Eléments Finis.

Les calculs sont organisés de façon à isoler et à minimiser les opérations avec indexation indirecte (passage points/éléments). Les opérations purement vectorielles (sur des mots contigus en mémoire) ainsi introduites sont optimisées pour exploiter au maximum les caractéristiques de l'architecture. En outre, des techniques de coloriage permettent de partitionner les éléments en plusieurs groupes formés d'éléments disjoints, sans sommet commun. Au sein d'un même groupe d'éléments, on supprime ainsi les dépendances de données dans l'étape d'accumulation aux sommets des valeurs élémentaires qui peut donc être effectuée en mode vectoriel.

Les exemples traités sont caractéristiques des problèmes liés à la soufflerie numérique, dans un cadre industriel. Les problèmes numériques de résolution des équations d'Euler et de Navier-Stokes pour des fluides compressibles sont décrits au Chapitre 2. Le Chapitre 3 rappelle la structure des ordinateurs vectoriels et les techniques d'adéquation des programmes à l'architecture. Les méthodes utilisées pour optimiser les codes étudiés et accroître les performances font l'objet du Chapitre 4, où sont détaillés les problèmes spécifiques des Eléments Finis. Enfin, les résultats et vitesses d'exécution sur CRAY-1 et CRAY-XMP sont commentés au Chapitre 5.

CHAPITRE 2 : PROBLEME NUMERIQUE

I - EQUATIONS REGISSANT L'ECOULEMENT 2D D'UN FLUIDE DE NAVIER-STOKES COMPRESSIBLE

On introduit une notation vectorielle qui est souvent utilisée pour ce problème. On introduit le vecteur W représentant l'ensemble des variables :

$$(1.1)\quad W = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ e \end{bmatrix} = \begin{bmatrix} \rho \\ m \\ n \\ \ell \end{bmatrix}$$

Alors le système s'écrit sous la forme :

$$(1.2)\quad \frac{\partial W}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = \frac{1}{Re}\left(\frac{\partial R}{\partial x} + \frac{\partial S}{\partial y}\right)$$

avec

$$F(W) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(e+p) \end{bmatrix}, \qquad G(W) = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(e+p) \end{bmatrix},$$

$$(1.3)\quad R = \begin{bmatrix} 0 \\ \frac{4}{3}\frac{\partial u}{\partial x} - \frac{2}{3}\frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ u\left(\frac{4}{3}\frac{\partial u}{\partial x} - \frac{2}{3}\frac{\partial v}{\partial y}\right) + v\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) + \frac{\gamma}{P_r}\frac{\partial \epsilon}{\partial x} \end{bmatrix}$$

$$(1.4)\quad S = \begin{bmatrix} 0 \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{4}{3}\frac{\partial v}{\partial y} - \frac{2}{3}\frac{\partial u}{\partial x} \\ u\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) + v\left(\frac{4}{3}\frac{\partial v}{\partial y} - \frac{2}{3}\frac{\partial u}{\partial x}\right) + \frac{\gamma}{P_r}\frac{\partial \epsilon}{\partial y} \end{bmatrix}$$

γ : est le rapport des chaleurs spécifiques à pression et volume constants. On fait l'hypothèse qu'ils sont constants. $\gamma=1.4$ pour l'air.

P_r est le nombre de Prandtl $P_r = .72$ pour l'air

(u,v) : sont les composantes de la vitesse du fluide

ρ sa densité,

e l'énergie totale

p la pression

E l'énergie interne spécifique

Re nombre de Reynolds

De plus on a toujours

$$(1.5) \quad \epsilon = \frac{1}{\gamma-1}\frac{p}{\rho} ,$$

$$e = \rho\epsilon + \rho\frac{u^2+v^2}{2} .$$

On a mis au 1er membre de (1.2) les dérivées du 1er ordre et au 2ème membre les termes visqueux du 2ème ordre ; si on annule le 2ème membre (Re infini) on retrouve les équations d'Euler.

Notation :

Le nombre de Mach est défini comme suit :

$$M^2 = \frac{\rho(u^2+v^2)}{\gamma p}$$

La vitesse du son est notée

$$c^2 = \gamma\frac{p}{\rho}$$

On notera aussi que les opérateurs F et G sont homogènes d'ordre 1 pour W, c'est-à-dire que chaque composante, par exemple, de F vérifie $F_i(\lambda W) = \lambda F_i(W)$. On peut écrire :

$$F(W) = AW , \quad G(W) = BW$$

où on note A et B les matrices jacobiennes respectivement de F et G :

$$A = \frac{\partial F}{\partial W} \quad , \qquad B = \frac{\partial G}{\partial W}$$

Ce qui s'écrit :

$$A = -\begin{bmatrix} 0 & -1 & 0 & 0 \\ \frac{3-\gamma}{2}u^2 + \frac{1-\gamma}{2}v^2 & (\gamma-3)u & (\gamma-1) & 1-\gamma \\ uv & -v & -u & 0 \\ \gamma\frac{eu}{\rho} + (1-\gamma)u(u^2+v^2) & -\frac{\gamma e}{\rho} + \frac{\gamma-1}{2}(3u^2+v^2) & (\gamma-1)uv & -\gamma u \end{bmatrix}$$

$$B = -\begin{bmatrix} 0 & 0 & -1 & 0 \\ uv & -v & -u & 0 \\ \frac{3-\gamma}{2}v^2 + \frac{1-\gamma}{2}u^2 & (\gamma-1)u & (\gamma-3)v & 1-\gamma \\ \gamma\frac{ev}{\rho} + (1-\gamma)v(u^2+v^2) & (\gamma-1)uv & -\frac{\gamma e}{\rho} + \frac{\gamma-1}{2}(3v^2+u^2) & -\gamma v \end{bmatrix}$$

L'équation (1.2) avec Re = $+\infty$ est hyperbolique car la matrice $H = \eta_1 A + \eta_2 B$ a ses valeurs propres réelles pour chaque $(\eta_1,\eta_2) \in \mathbb{R}^2$ et de plus H est diagonalisable : c'est à dire qu'il existe une matrice $T_{\eta_1\eta_2}$ telle que

$$H = T_{\eta_1\eta_2} \, \Lambda_{\eta_1\eta_2} \, T^{-1}_{\eta_1\eta_2}$$

où $\Lambda_{\eta_1\eta_2}$ est la matrice diagonale contenant les valeurs propres de H.

Le calcul direct des valeurs propres de H et des matrices T, T^{-1} qui diagonalisent H est fastidieux. Il est plus simple de considérer les équations écrites dans les variables physiques - vecteur $\tilde{W}$

$$\frac{\partial \tilde{W}}{\partial t} + \tilde{A}\frac{\partial \tilde{W}}{\partial x} + \tilde{B}\frac{\partial \tilde{W}}{\partial y} = 0 \tag{1.6}$$

$$\tilde{W} = \begin{pmatrix} \rho \\ u \\ v \\ p \end{pmatrix} , \quad \tilde{A} = \begin{pmatrix} u & \rho & 0 & 0 \\ 0 & u & 0 & 1/\rho \\ 0 & 0 & u & 0 \\ 0 & \gamma p & 0 & u \end{pmatrix} , \quad \tilde{B} = \begin{pmatrix} v & 0 & \rho & 0 \\ 0 & v & 0 & 0 \\ 0 & 0 & v & 1/\rho \\ 0 & 0 & \gamma p & v \end{pmatrix}$$

Soit $S : \mathbb{R}^4 \to \mathbb{R}^4$ un opérateur défini par $S(\tilde{W}) = W$. Les équations (1.2) et (1.5) sont équivalentes donc on a

$$\tilde{A} = M^{-1} A M , \quad \tilde{B} = M^{-1} B M , \quad M = \frac{\partial S}{\partial \tilde{W}} .$$

Les matrices M et M^{-1} sont calculées ainsi

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ u & \rho & 0 & 0 \\ v & 0 & \rho & 0 \\ \frac{u^2+v^2}{2} & \rho u & \rho v & 1/\gamma-1 \end{pmatrix} \qquad M^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -u/\rho & 1/\rho & 0 & 0 \\ -v/\rho & 0 & 1/\rho & 0 \\ (\gamma-1)\frac{u^2+v^2}{2} & (1-\gamma)u & (1-\gamma)v & (\gamma-1) \end{pmatrix}$$

La matrice $\tilde{H} = \eta_1 \tilde{A} + \eta_2 \tilde{B}$ est reliée à H par $H = M \tilde{H} M^{-1}$. Par conséquent les valeurs propres de H et $\tilde{H}$ sont identiques. Les valeurs propres de $\tilde{H}$ sont

$$\begin{cases} \lambda_1(W) = \lambda_2(W) = \eta_1 u + \eta_2 v , \\ \lambda_3(W) = \lambda_1(W) + c(\eta_1^2 + \eta_2^2)^{1/2} , \\ \lambda_4(W) = \lambda_1(W) - c(\eta_1^2 + \eta_2^2)^{1/2} , \end{cases}$$

II - SCHEMA EXPLICITE DE TYPE LAX WENDROFF

Dans cette partie, on développe une méthode pour résoudre les équations de Navier-Stokes, à partir d'un solveur d'Euler, basé sur un schéma de Lax Wendroff à 2 pas. La méthode s'est montrée efficace pour résoudre les équations d'Euler [13] et donc est intéressante pour résoudre les équations de Navier-Stokes, tout particulièrement quand la partie hyperbolique du système est prédominante (grands nombres de Reynolds). La

stratégie consiste à ajouter explicitement les termes visqueux de Navier-Stokes en tenant compte des conditions sur le corps [14].

2.1. Discrétisation en temps

On considère l'équation linéaire scalaire

$$\frac{\partial w}{\partial t} + V.\nabla w = 0 \tag{2.1}$$

où V est un vecteur donné, constant.

Le schéma introduit par Lax et Wendroff est basé sur un développement en série de Fourier par rapport au temps :

$$w^{n+1} = w^n + \Delta t \frac{\partial w^n}{\partial t} + \frac{\Delta t^2}{2} \frac{\partial^2 w^2}{\partial t^2} + 0(\Delta t^3) , \tag{2.2}$$

où w^k correspond à une approximation de $w(k\Delta t)$. En utilisant (2.1) pour calculer les dérivées en temps, on obtient le schéma suivant qui est du 2ème ordre en temps.

$$w^{n+1} = w^n - \Delta t \, \underset{\sim}{V}.\underset{\sim}{\nabla} w^n + \frac{\Delta t^2}{2} \underset{\sim}{V}.\underset{\sim}{\nabla}(\underset{\sim}{V}.\underset{\sim}{\nabla} w^n) . \tag{2.3}$$

Lax et Wendroff ont appliqué ce schéma pour des différences finies centrées en espace, ce qui donne globalement un schéma du second ordre en précision.

2.2. Discrétisation en éléments finis pour l'équation d'advection

Soit V_h l'espace discret suivant

$$V_h = \{\phi_h \in C^0(\bar{\Omega}), \quad \phi_{h|T} \in P_1 , \ \forall T \in \delta_h\}$$

avec δ_h une triangulation standard de Ω.

P_1 ensemble des polynômes de degré ≤ 1

$C^0(\bar{\Omega})$ l'ensemble des fonctions continues à support compact.

On introduit la discrétisation en espace de 2.3, sous forme variationnelle

$$\left\{ \begin{array}{l} \text{Si } w_h^n \in V_h \text{ est connu, on calcule } w_h^{n+1} \text{ par :} \\ \\ \displaystyle\int_\Omega w_h^{n+1} \phi_h dx = \int_\Omega w_h^n \phi_h \, dx - \Delta t \int_\Omega \underset{\sim}{V}.\underset{\sim}{\nabla} w_h^n \phi_h \, dx \\ \\ \displaystyle - \frac{\Delta t^2}{2} \int_\Omega (\underset{\sim}{V}.\underset{\sim}{\nabla} w_h^n)(\underset{\sim}{V}.\underset{\sim}{\nabla}\phi_h) dx \\ \\ \displaystyle + \frac{\Delta t^2}{2} \int_\Gamma \underset{\sim}{V}.\underset{\sim}{\nabla} w_h^n \, \underset{\sim}{V}.\underset{\sim}{n} \, \phi_h \, d\Gamma , \quad \forall \phi_h \in V_h , \\ \\ w_h^{n+1} \in V_h . \end{array} \right. \tag{2.5}$$

Comme les fonctions approchées sont linéaires par morceaux, on peut calculer directement leurs dérivées.

Si le membre de gauche de (2.5) est calculé exactement, on doit résoudre un système linéaire pour obtenir la solution de 2.5. Aussi on utilise une variante "mass lumping" de (2.5).

$$
(2.6)\qquad
\begin{cases}
(w_h^{n+1},\phi_h)_h = (w_h^n,\phi_h)_h - \Delta t \int_\Omega V.\nabla w_h^n \phi_h \, dx \\
\\
- \dfrac{\Delta t^2}{2} \int_\Omega (V.\nabla w_h^n)(V.\nabla \phi_h) dx \\
\\
+ \dfrac{\Delta t^2}{2} \int_\Gamma V.\nabla w_h^n \, V.n \, \phi_h \, d\Gamma \quad , \quad \forall \phi_h \in V_h \\
\\
w_h^{n+1} \in V_h \quad ,
\end{cases}
$$

avec le produit scalaire $(.,.)_h$ défini par

$$
(2.7)\qquad (\phi_h,z_h)_h = \sum_{T \in \delta_h} \frac{\text{aire}(T)}{3} \sum_{i=1}^{3} \phi_h(P_i)\, z_h(P_i) \, , \quad \forall \phi_h, z_h \in V_h
$$

où

aire(T) est l'aire du triangle T

P_i sont les noeuds du triangle T i=1,2,3.

2.3. Schéma de Lax Wendroff à 2 pas pour les équations d'Euler

Le schéma présenté dans le paragraphe 2.2 a été proposé pour les systèmes hyperboliques. Pour simplifier les calculs, une version à 2 pas de ce schéma a été introduite par Richtmyer. Pour une équation non linéaire scalaire de la forme

$$
(2.8)\qquad \frac{\partial w}{\partial t} + \underset{\sim}{\nabla}.\underset{\sim}{f}(w) = 0
$$

La discrétisation en temps du schéma à 2 pas de Lax Wendroff est donné par : Si w^n est connu, on calcule le prédicteur $\hat{w}$ et w^{n+1} par

Etape prédicteur

$$\hat{w} = w^n - \alpha \Delta t \nabla.f(w^n) \tag{2.9}$$

Etape correcteur

$$w^{n+1} = w^n + \beta_1 \Delta t \nabla.f(w^n) + \beta_2 \Delta t \nabla.f(\hat{w}) \tag{2.10}$$

avec

$$\alpha = 1 + \frac{\sqrt{5}}{2} \quad , \quad \beta_1 = \frac{2\alpha - 1}{2\alpha} \quad , \quad \beta_2 = \frac{1}{2\alpha} . \tag{2.11}$$

Quand on applique à l'équation linéaire (1.1), le schéma (2.9) - (2.11) est équivalent à (2.3). Ce schéma a été appliqué à une discrétisation spatiale de type éléments finis.

Soit H_h l'espace discret

$$H_h = \{z_h \in L^2(\Omega), \quad z_{h|T} \in P_o \quad , \quad \forall T \in \delta_h\} \tag{2.12}$$

on applique la discrétisation en temps 2.9, 2.10 à la partie hyperbolique du système (1.2) (équations d'Euler) en utilisant des approximations constantes par morceaux pour l'étape prédicteur et linéaire par morceaux pour l'étape correcteur.

Etape prédicteur

Soit $W_h^n \in V_h^n$, on calcule $\hat{W}_h \in H_h^n$ par

$$\int_\Omega \hat{W}_h . z_h \, dx = \int_\Omega W_h^n . z_h \, dx - \alpha \Delta t \int_\Omega \left[\frac{\partial F_h(W_h^n)}{\partial x} . z_h \right.$$

$$\left. + \frac{\partial G_h(W_h^n)}{\partial y} . z_h \, dx \right] \quad , \quad \forall z_h \in H_h^4 \quad , \tag{2.13}$$

G_h et z_h étant constants sur chaque triangle et en intégrant par parties les termes dérivées, on obtient

$$\hat{W}_h(T) = \int_T W_h^n \, dx - \alpha \Delta t \int_{\partial T}^{*} (F_h(W_h^n)\nu_x + G_h(W_h^n)\nu_y) d\sigma \quad , \quad \forall T \in \delta_h . \tag{2.14}$$

(ν_x, ν_y) est le vecteur unitaire normal à ∂T, frontière du triangle T, dirigé vers l'extérieur de T

Etape correcteur

On calcule $W_h^{n+1} \in V_h^n$ par

$$
(2.15)\quad \left\{
\begin{array}{ll}
<1> & \frac{1}{\Delta t}(W_h^{n+1} - W_h^n), \phi_h)_h = \\
<2> & \int_\Omega^* \{\beta_1 \left[F_h(W_h^n)\frac{\partial \phi_h}{\partial x} + G_h(W_h^n)\frac{\partial \phi_h}{\partial y}\right] \\
<3> & + \beta_2 \left[F_h(\hat{W}_h)\frac{\partial \phi_h}{\partial x} + G_h(\hat{W}_h)\frac{\partial \phi_h}{\partial y}\right] \} \, dx \\
<4> & + \frac{\chi}{2} \int_\Omega^* \left[f_h(W_h^n)\frac{\partial W_h^n}{\partial x}\frac{\partial \phi_h}{\partial x} + g_h(W_h^n)\frac{\partial W_h^n}{\partial y}\frac{\partial \phi_h}{\partial y}\right] dx \\
<5> & - \int_\Gamma^* (F_h(\bar{W}_h)\nu_x + G_h(\bar{W}_h)\nu_y)\phi_h d\Gamma, \quad \forall \phi_h \in V_h^4
\end{array}
\right.
$$

on utilise une intégration numérique dans les intégrales avec une étoile.
(2.14) s'écrit

$$
(2.17)\quad \left\{
\begin{array}{l}
\int_{\partial T}^* (F_h(W_h^n)\nu_x + G_h(W_h^n)\nu_y)\, d\sigma = \\
\sum_{i=1}^{3} L_i \left[F_h(W_h^n(m_i))\nu_x + G_h(W_h^n(m_i))\nu_y\right]
\end{array}
\right.
$$

où L_i est la longueur du côté i du triangle T, i=1,2,3

m_i est le milieu du côté i de T

$$
(2.18)\quad F_h(W_h^n(m_i)) = \begin{bmatrix}
m(m_i) \\
\frac{(m(m_i))^2}{\rho(m_i)} + (\gamma-1)(e(m_i) - \frac{(m(m_i))^2+(n(m_i))^2}{2\rho(m_i)} \\
\frac{m(m_i)n(m_i)}{\rho(m_i)} \\
\frac{m(m_i)}{\rho(m_i)}(\gamma e(m_i) - (\gamma-1)\frac{(m(m_i))^2+(n(m_i))^2}{2\rho(m_i)}
\end{bmatrix}
$$

$G_h(U_h(m_i))$ est calculé de la même manière.

Dans < 1 > on utilise le produit scalaire défini en (2.7) et étendu à V_h^n.

Dans < 2 > on utilise une formule d'intégration exacte pour les polynômes de degré 2

$$(2.19)\qquad \begin{cases} \int_{\Omega}^{*} [F_h(W_h^n)\dfrac{\partial \phi_h}{\partial x} + G_h(W_h^n)\dfrac{\partial \phi_h}{\partial y}]\,dt = \\ \sum\limits_{T\in \delta_h} \dfrac{\text{aire}(T)}{3}\,[\dfrac{\partial \phi_n}{\partial x}\Big|_T \sum\limits_{i=1}^{3} F_h(W_h^n(m_i)) + \dfrac{\partial \phi_h}{\partial y}\Big|_T \sum\limits_{i=1}^{3} G_h(W_h^n(m_i))] \end{cases}$$

< 3 > est calculé exactement puisque $\hat{W}_k$ est constant par morceau.

< 4 > est un terme de viscosité artificielle avec un paramètre χ, les fonctions scalaires f_h et g_h sont constantes par morceaux et données par

$$(2.20)\qquad f_h(W_h^n)\Big|_T = \Delta x^2\ \text{Max}\,(\frac{\partial \rho}{\partial x}\Big|_T, \frac{\partial u}{\partial x}\Big|_T, \frac{\partial v}{\partial x}\Big|_T, \frac{\partial e}{\partial x}\Big|_T, \frac{\partial S}{\partial x}\Big|_T)$$

S est l'entropie,

Δx est une estimation de la longueur de T dans la direction x

$g_h(W_h^n)\Big|_T$ est défini de façon semblable pour la direction y.

< 5 > contient les conditions limites.

$\bar{W}_h$ est calculé à partir des valeurs intérieures de W_h^n et des conditions à l'infini.

2.4. Application aux équations de Navier Stokes

Pour résoudre les équations de Navier Stokes, sous forme conservative, on applique le schéma précédent à la partie hyperbolique des équations et on traite la partie visqueuse par un schéma du 1er ordre, aussi n'introduisons nous des termes visqueux que dans l'étape correcteur $V_{h,o} = V_h \cap \{\phi = 0 \text{ sur } \Gamma_\beta\}$.

On obtient l'algorithme suivant :

- étape prédicteur

Connaissant W_h^n, on calcule $\hat{W}_h$ par (2.13), (2.14)

- étape correcteur

$$(2.21)\qquad \begin{cases} \dfrac{1}{\Delta t}\,(W_h^{n+1} - W_h^n, \phi_h)_h = \langle 2\rangle + \langle 3\rangle + \langle 4\rangle + \langle 5\rangle \\ \langle 6\rangle - \dfrac{1}{Re}\int_{\Omega} (R_h(W_h^n)\dfrac{\partial \phi_h}{\partial x} + S_h(W_h^n)\dfrac{\partial \phi_h}{\partial y})\,dx \\ \langle 7\rangle + \dfrac{1}{Re}\int_{\Gamma} (R_h(W_h^n)\nu_x + S_h(W_h^n)\nu_y)\phi_h\,d\Gamma\ ,\ \forall\ \phi_h \in V_{h,o}^4 \end{cases}$$

<2> ... <5> sont donnés par (2.15).

Pour <6> et <7> on définit les fonctions R_h, S_h par

$$R_h = R(u_h, v_h, \epsilon_h)$$

$$S_h = S(u_h, v_h, \epsilon_h)$$

R, S sont donnés par (1.3), (1.4).

u_h, v_h, ϵ_h sont linéaires par morceau sur chaque triangle T et satisfont

$$u_h(M_i) = \frac{m(M_i)}{\rho(M_i)} \tag{2.22}$$

$$v_h(M_i) = \frac{n(M_i)}{\rho(M_i)} \tag{2.23}$$

$$\epsilon_h(M_i) = \frac{1}{\rho(M_i)} \left[e(M_i) - \frac{(m(M_i))^2 + (n(M_i))^2}{2\rho(M_i)} \right] \tag{2.24}$$

M_i sont les noeuds du triangle T, i=1,...3.

L'intégrale <6>, peut alors être calculée exactement. Pour <7> il faut faire une interpolation aux noeuds des valeurs obtenues sur les triangles.

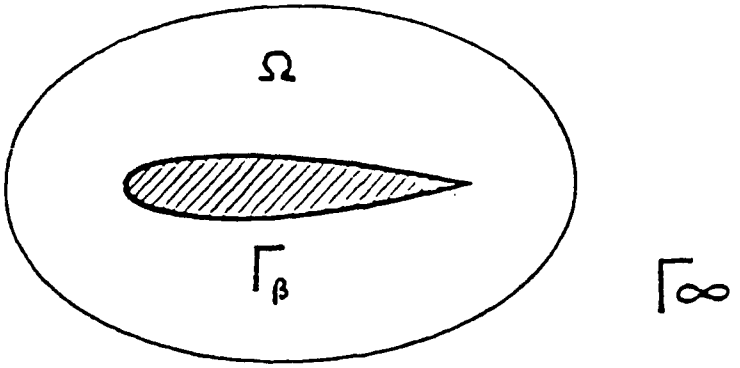
2.5. Conditions limites

Dans le paragraphe suivant on donne quelques détails sur les conditions limites. Elles ne sont prises en compte que dans l'étape correcteur. Elles sont donc du 1er ordre en temps.

On donne les conditions aux limites dans le cas d'un écoulement externe autour d'un corps. Soit Γ_∞ la frontière externe, Γ_β la frontière sur le corps (cf. figure 2.1) .

$\Gamma = \Gamma_\beta \cup \Gamma_\infty$

Figure 2.1



2.5.1. Conditions à l'infini

On donne à l'infini (sur la frontière externe du domaine) des conditions d'écoulement uniforme. Un traitement numérique permet de prendre en compte uniquement les informations linéaires (hyperbolicité du système à résoudre).

Pour donner une idée de la méthode appliquée, considérons l'intégrale des termes convectifs de (2.5)

$$I = \int_{\Omega} V.\nabla w_h^n \phi_h \, dx \,, \tag{2.25}$$

On intègre par parties

$$I = - \int_{\Omega} w_h^n \, V.\nabla \phi_h \, dx + \int_{\Gamma} \overline{w}_h \, V.n\phi_h \, d\Gamma \,, \tag{2.26}$$

on suppose que l'on veuille imposer

$$w_h = w_\infty \quad \text{sur} \quad \Gamma_\infty \tag{2.27}$$

Alors on définit $\bar{w}_h$ par

$$\begin{aligned} \bar{w}_h &= w_\infty \quad \text{if} \quad V.n < 0 \\ \bar{w}_h &= w_h^n \quad \text{if} \quad V.n \geq 0 \end{aligned} \tag{2.28}$$

on applique la même règle pour le calcul de l'intégrale de bord sur Γ_∞ du terme <5> de (2.21), les valeurs propres des matrices jacobiennes de F_h et G_h jouant le rôle de V et les conditions limites étant données par l'écoulement uniforme.

En utilisant cette technique, les conditions limites adéquates sont automatiquement prises aussi bien pour les écoulements subsoniques que supersoniques.

2.5.2. Conditions sur le corps pour Navier-Stokes

Sur la paroi du corps, on impose une condition d'adhérence, c'est-à-dire que la vitesse U est nulle sur le corps.

En ce qui concerne la température, on suppose la paroi isotherme ce qui se traduit par le fait que T, la température au corps, est égale à la température d'arrêt, ou encore $\epsilon = \epsilon_\beta$ donné sur Γ_β.

La solution au temps n+1 étant calculée pour les noeuds intérieurs par (2.13, 2.14, 2.21) on impose les conditions suivantes sur le corps :

(2.29) $U^{n+1} = 0$ condition d'adhérence

(2.30) $\epsilon^{n+1} = \epsilon_\beta$ ϵ_β donné

On déduit la pression par l'algorithme suivant

i) Pour tout noeud P_i appartenant au corps on définit le triangle T_i qui contient la normale issue de P_i et dirigée vers le fluide.

ii) Les termes visqueux sont calculés sur ce triangle et cette valeur est donnée sur le côté appartenant au corps.

iii) La pression est supposée linéaire sur le triangle T_i.

On écrit les équations de quantité de mouvement sur le corps.
On obtient alors

$$\nabla p = \frac{1}{Re} [\Delta U^n + \frac{1}{3} \nabla(\nabla . U^n)]$$

Deux cas se présentent alors :

- si P_i est le seul noeud du triangle T_i sur le corps (figure 2.2), l'interpolation linéaire pour calculer la pression en P_i est faite dans la direction normale $\nabla p : n = p_i^{n+1} \frac{\partial p_i}{\partial n} + \sum_{j=1,2} p_j^n \frac{\partial \phi_j}{\partial n}$

  j : autres noeuds de T_i
  ϕ_j fonctions de base

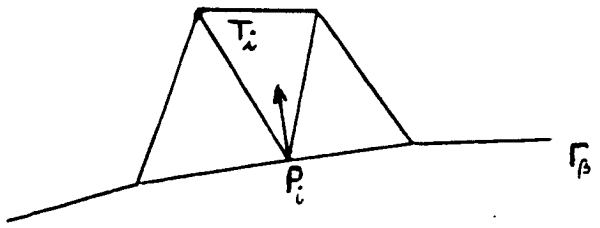


Figure 2.2

- si 2 noeuds de T_i appartiennent au corps (figure 2.3), l'équation linéaire est écrite sur le segment $P_i P_j$ où P_j est le sommet de T_i qui n'est pas sur le corps.

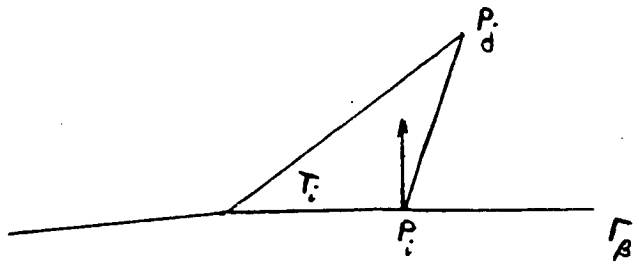


Figure 2.3

$$P^{n+1}(P_i) = p^n(P_j) + \overrightarrow{P_i P_j} \cdot \nabla p^n$$

La pression étant calculée sur le corps, (1.5) et (2.30) permettent de calculer ρ et e au pas de temps $(n+1)\Delta t$.

2.5.3. Conditions sur le corps pour Euler

Dans le cas des équations d'Euler ($Re = +\infty$) on remplace la condition d'adhérence par une condition de glissement $U.n = 0$.

L'intégrale <5> de (2.15) ne contient plus que des termes de pression pour les équations de quantité de mouvement. Aucune autre contribution n'est apportée aux autres équations. Les conditions de glissement sont prises de façon faible au travers de cette intégrale ; il n'y a pas de difficulté car la pression est connue en tous les points au temps $n.\Delta t$.

2.6. Pas de temps local

On ne considère que des écoulements stationnaires. Aussi on peut utiliser un pas de temps défini localement, ce qui permet de converger plus rapidement vers la solution stationnaire.

Dans ce cas, le pas de temps est une fonction discrète dépendant du pas de temps. On le définit aux noeuds de la triangulation δ_h par la formule

$$\Delta t(M_i) = K \frac{h^2(M_i)}{\frac{2}{PrRe} + (|U_i| + c_i)h(M_i)}$$

K : nombre de CFL < 1 pour la stabilité du schéma

M_i : noeud de δ_h

B_i : ensemble des triangles de δ_h ayant M_i comme noeud commun = support de M_i

$|U_i|$: approximation du module de la vitesse en M_i

c_i : approximation de la vitesse du son en M_i.

H(T) : grandeur du plus grand côté de T

$$h(M_i) = \min_{T \in B_i} \frac{2 \text{ aire}(T)}{H(T)}$$

III - SCHEMA EXPLICITE DECENTRE DE TYPE VAN-LEER

Une autre approche pour résoudre les équations d'Euler est d'utiliser un schéma décentré [15].

3.1. Approximation en temps

On définit un schéma explicite d'ordre 2 de type Van-Leer à deux pas.

On utilise la formulation continue en variables physique (1.6) pour l'étape prédiction qui s'écrit :

$$\tilde{W}^{n+\frac{1}{2}} = \tilde{W}^n - \frac{\Delta t}{2}(\tilde{A}(\tilde{W})\tilde{W}_x + \tilde{B}(\tilde{W}^n)\tilde{W}^n_y) \tag{3.1}$$

où Δt est le pas de temps.

La formulation de l'étape correction utilise la formulation conservative (1.2).

$$W^{n+1} = W^n - \Delta t\,(A(W^{n+\frac{1}{2}})\,W_x^{n+\frac{1}{2}} + B(W^{n+\frac{1}{2}})\,W_y^{n+\frac{1}{2}} \tag{3.2}$$

3.2. Formulation variationnelle (volume fini)

En multipliant par une fonction test ϕ, et en intégrant par parties, on obtient la formule :

$$\iint_\Omega \phi W_t \, d\Omega - \int_\Omega (F(W)\phi_x + G(W)\, d\Omega + \int_{\partial\Omega} \phi(F(W)\nu_x + G(W)\nu_y)\, d\sigma = 0$$

Pour déterminer l'ensemble des fonctions ϕ, on définit une cellule C_i autour de chaque noeud M_i, à partir de la triangulation (figure 2.4) (L_{ij} joint les centres des 2 triangles contenant M_i et M_j en passant par le milieu du segment M_iM_j).

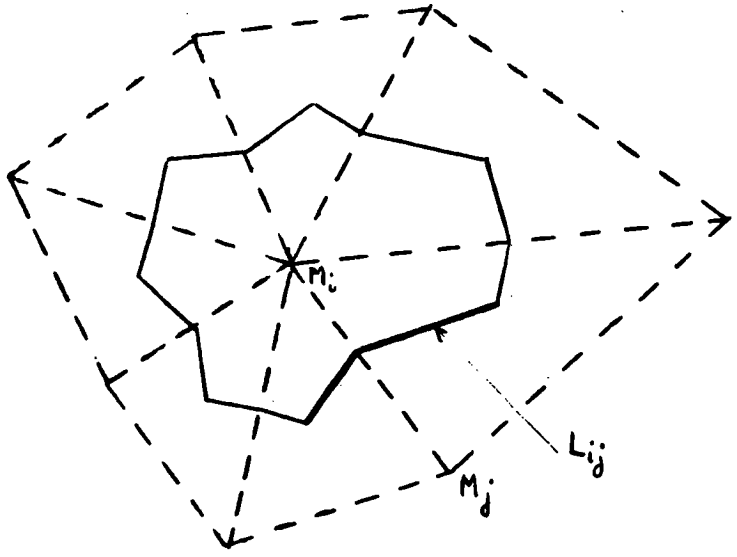


Figure 2.4 Cellule C_i

On construit un système discret d'équations en choisissant $\phi_i = \chi_i$, fonction caractéristique de la cellule C_i, pour chaque noeud M_i.

$$\iint_{C_i} W_t \, d\Omega + \int_{\partial C_i} (F(W)\nu_x + G(W)\nu_y) d\sigma = 0$$

$$\iint_{C_i} W_t \, d\Omega + \int_{\partial C_i} (A(W)\nu_x + B(W)\nu_y) \, W \, d\sigma = 0$$

soit

$$\iint_{C_i} W_t \, d\Omega + \sum_{j=1}^{N_i} \phi_{ij} = 0$$

où

$$\phi_{ij} = \int_{L_{ij}} (A(W)\nu_x + B(W)\nu_y) \, W \, d\sigma$$

N_i est le nombre d'arêtes (i,j) issues de M_i

L_{ij} est la portion de ∂C_i relative à l'arête (ij) comme indiqué sur la figure 2.4.

3.3. Intégration numérique

On utilise une formule d'intégration à un point pour calculer l'intégrale de surface :

$$\iint_{C_i} W_t \, d\Omega = \text{aire}\,(C_i)\, W_{t,i}$$

où aire(C_i) est l'aire de la cellule C_i

$W_{t,i}$ est la valeur au noeud i de W_t.

Le calcul des flux sur la frontière ∂C_i doit introduire des termes dissipatifs pour éviter les oscillations au niveau du choc. On utilise une méthode de décomposition de flux, de type Q-schéma :

$$P_{ij} = A\left(\frac{W_{ij} + W_{ji}}{2}\right) \nu_x + B\left(\frac{W_{ij} + W_{ji}}{2}\right) \nu_y$$

$$\phi_{ij} = \text{long}(L_{ij}) \left\{ \frac{1}{2} B\left(\frac{W_{ij} + W_{ji}}{2}\right) + \frac{1}{2} |P_{ij}| . (W_i - W_j) \right\}$$

$$B\left(\frac{W_{ij} + W_{ji}}{2}\right) = \frac{F(W_{ij})\nu_x + G(W_{ij})\nu_y + F(W_{ji})\nu_x + G(W_{ji})\nu_y}{2}$$

W_{ij} et W_{ji} sont définis de façon à conserver un schéma d'ordre deux, au moins sur des maillages suffisamment réguliers. On a choisi une interpolation linéaire à l'aide d'une approximation du gradient ∇W :

$$W_{ij} = W_i + \nabla W_i . \overrightarrow{M_i M_j}$$

$$W_{ji} = W_j - \nabla W_j . \overrightarrow{M_i M_j}$$

L'approximation de W est linéaire sur chaque triangle, ce qui conduit classiquement à une interpolation linéaire des dérivées

$$W_{x,i} = \frac{\sum_{T \in B_i} W_{x,i/T} * \text{aire}(T)}{\sum_{T \in B_i} \text{aire}(T)}$$

Le calcul de ces dérivées s'obtient donc en accumulant les dérivées élémentaires sur chaque triangle.

L'algorithme global s'écrit donc comme suit, pour chaque noeud M_i .

- Prédicteur

$$\tilde{W}_i^{n+\frac{1}{2}} = \tilde{W}_i^n - \frac{\Delta t}{2} (\tilde{A}(\tilde{W}_i^n)\, \tilde{W}_{x,i}^n + \tilde{B}(\tilde{W}_i^n)\, \tilde{W}_{x,i}^n)$$

La formulation est en variable physiques.

Cette étape correspond à une boucle sur les noeuds.

- Correcteur

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{\text{aire}(C_i)} \cdot \sum_{j=1}^{N_i} \phi_{ij}^{n+\frac{1}{2}}$$

Cette partie est en variables conservatives.

On effectue d'abord une boucle sur les arêtes $(M_i M_j)$ pour calculer les flux ϕ_{ij}, qui sont ensuite accumulés aux noeuds correspondant.

Comme pour le schéma de Lax-Wendroff, on utilise un pas de temps local (2.6). Les conditions limites sont traitées comme dans le paragraphe (2.5).

IV - SCHEMA IMPLICITE

Les delta schémas implicites se montrent fort intéressants pour construire des solveurs rapides des équations d'Euler stationnaires. Ils peuvent être appliqués aussi bien à des schémas centrés ou décentrés qu'à des approximations de type Galerkin. Les schémas implicites semblent à l'heure actuelle mieux adaptés aux maillages non structurés que les méthodes multigrilles.

4.1. Cas monodimensionnel

Considérons le système non linéaire :

$$\frac{\partial w}{\partial t} = f(w) \qquad (4.1)$$

Après un développement en temps "arrière", on obtient :

$$w^{n+1} = w^n + \Delta t \frac{\partial w^{n+1}}{\partial t}$$

$$w^{n+1} = w^n + \Delta t\, f(w^{n+1})$$

Si f est différentiable, de Jacobien A, il vient :

$$w^{n+1} = w^n + \Delta t \left[f(w^n) + A(w^n)(w^{n+1} - w^n)\right]$$

d'où, en posant

$$\Delta w = w^{n+1} - w^n$$

$$\left[I - \Delta t\, A(w^n)\right] \Delta w = \Delta t\, f(w^n)$$

$$\boxed{\left[\frac{1}{\Delta t} I - A(w^n)\right] \Delta w = f(w^n)} \qquad (4.2)$$

4.2. Cas bidimensionnel

Pour construire le Jacobien A, on utilise une linéarisation d'une décomposition de flux. Par exemple, on utilise une décomposition du premier ordre de type Q-schéma. [17],[18].

4.3. Algorithme

Le schéma de chaque itération en temps se décompose en deux phases :

phase explicite : $$\hat{W} - W^n = \Delta t\, f(W^n) \qquad (4.3)$$

phase implicite : $$\left[\frac{1}{\Delta t} I - A(W^{\tilde{n}})\right] \Delta W = f(W^{\tilde{n}}) \qquad (4.4)$$

Les schémas en temps utilisés dans la phase explicite (calcul de f) et dans la linéarisation de la phase implicite (calcul de A) peuvent être différents. On a combiné par exemple une phase explicite de type Lax-Wendroff avec une phase implicite de type Q-schéma.

La mise en oeuvre de l'équation (4.3) sur des ordinateurs vectoriels a fait l'objet d'études préalables [13], [14], [15].

L'équation (4.4) fait intervenir la construction de la matrice A, et la résolution du système linéaire induit. Ce système n'est pas symétrique, bien qu'à structure symétrique.

Il suffit en fait d'obtenir une approximation du résidu Δw, d'où l'idée d'utiliser une méthode de relaxation.

La matrice A n'est pas calculée à chaque pas de temps, mais est "gelée" durant un nombre fixé de pas de temps. On peut également déterminer la fréquence des mises à jours de A en fonction de la vitesse de convergence. On utilise un pas de temps local pour accélérer la convergence vers la solution stationnaire.

La mise en oeuvre et l'optimisation de l'équation (4.4) sont détaillées dans le chapitre 4.

CHAPITRE 3 : PROGRAMMATION VECTORIELLE

I - INTRODUCTION

Un ordinateur vectoriel peut fonctionner en mode scalaire, comme tout calculateur conventionnel, ou en mode vectoriel, grâce à son architecture spécialisée. Ce dernier est beaucoup plus rapide et doit donc prédominer pour obtenir de bonnes performances. Le code généré par le compilateur Fortran est exécuté en mode scalaire ou vectoriel selon certains critères liés à la machine et décrits ci-après. Vectoriser un programme, c'est essayer d'en augmenter la partie vectorielle.

Si l'on suppose que le calculateur est V fois plus rapide en mode vectoriel qu'en mode scalaire, la vitesse moyenne v_m (la vitesse scalaire étant l'unité), en fonction du taux de vectorisation α, est donnée par la formule ci-dessous dite loi d'Amdhal [6]

$$v_m = \frac{V}{\alpha + V(1-\alpha)}$$

La figure 3.1 montre la variation de v_m en fonction de α, pour différentes valeurs de V.

Cette remarque souligne la nécessité d'exploiter au mieux les caractéristiques vectorielles de la machine. Les paragraphes suivants décrivent succintement les composantes matérielles de la partie vectorielle et les règles de vectorisation d'un programme FORTRAN.

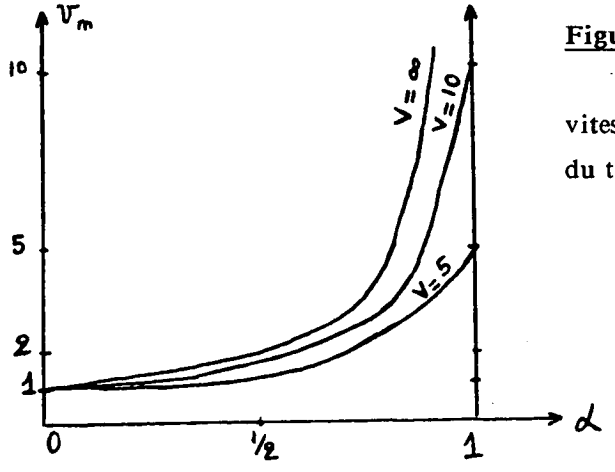


Figure 3.1

vitesse moyenne en fonction du taux de vectorisation.

II - ARCHITECTURE VECTORIELLE

2.1. Ordinateur séquentiel

Un ordinateur peut être grossièrement schématisé par la figure 3.2. La Mémoire contient les données et les programmes, l'Unité de Contrôle gère le flot d'instructions, tandis que l'Unité de Calcul effectue les opérations arithmétiques

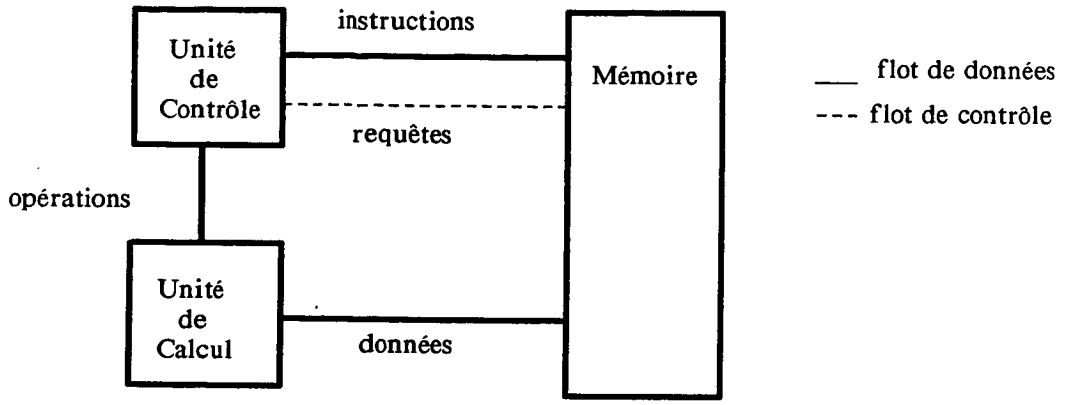


Figure 3.2 Schéma d'un ordinateur séquentiel

2.2. Principe du Pipeline

Afin d'accélérer l'exécution, les ordinateurs utilisent très souvent le principe du Pipeline ou du travail à la chaîne. Chaque instruction est décomposée en plusieurs étapes (par exemple, lecture de l'instruction en mémoire, décodage, lecture des opérandes, opération arithmétique, écriture du résultat, passage à l'instruction suivante). Les Unités de Contrôle et de Calcul travaillent sur plusieurs instructions à la fois, comme dans une chaîne d'assemblage, les postes de travail opèrent simultanément sur plusieurs objets à différents stades de fabrication (figure 3.3).

Après un temps de démarrage (le temps que la première instruction arrive au bout de la chaîne), le pipeline a un débit beaucoup plus rapide qu'une seule unité (en principe, le temps d'exécution est divisé par le nombre d'étapes dans la chaîne).

Ce mode de fonctionnement reste toutefois difficile à programmer au niveau des instructions. Il faut d'une part, alimenter la chaîne de production en instructions, d'autre

part s'assurer que chaque étape fonctionne à la même vitesse ou introduire des tampons (Buffers) pour synchroniser l'ensemble. En outre, il faut détecter et résoudre les conflits entre instructions ayant des opérandes ou résultats communs par exemple [7].

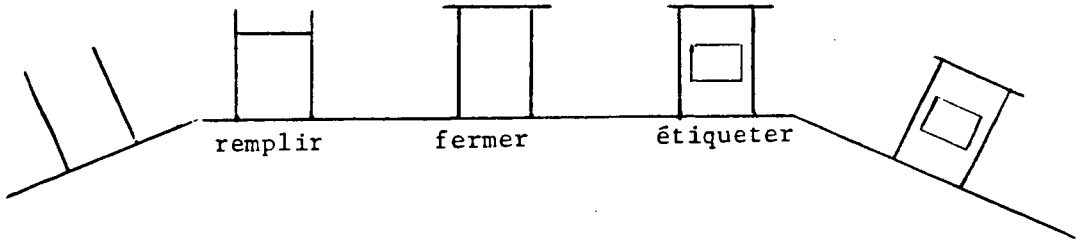


Figure 3.3. Schéma d'une chaîne d'assemblage

2.3. Opération vectorielle

Le concept vectoriel est basé sur deux principes généraux (Figure 3.4) :

- le mode pipeline est appliqué aux opérations arithmétiques, divisées en plusieurs étapes, afin d'augmenter le débit de production des résultats

- la même opération arithmétique est répétée un grand nombre de fois sur des jeux de données différents. Toutes ces opérations sont indépendantes (voir §. IV). Chaque jeu d'opérandes est un vecteur, et le nombre de données dans un jeu est la longueur du vecteur. La synchronisation de chaque pipeline se trouve ainsi extrêmement simplifiée.

La programmation de ce type d'architecture nécessite la détection de vecteurs et d'opérations vectorielles, ce qui fera l'objet des paragraphes suivants.

La vitesse d'une opération vectorielle dépend de la longueur des vecteurs. En effet, le remplissage du pipeline nécessite un temps d'amorçage (start-up) au bout duquel est fourni en général un résultat par cycle d'horloge, unité de temps d'un ordinateur (figure 3.5 : les numéros dans les colonnes indiquent les indices de vecteurs ou numéros de l'opération).

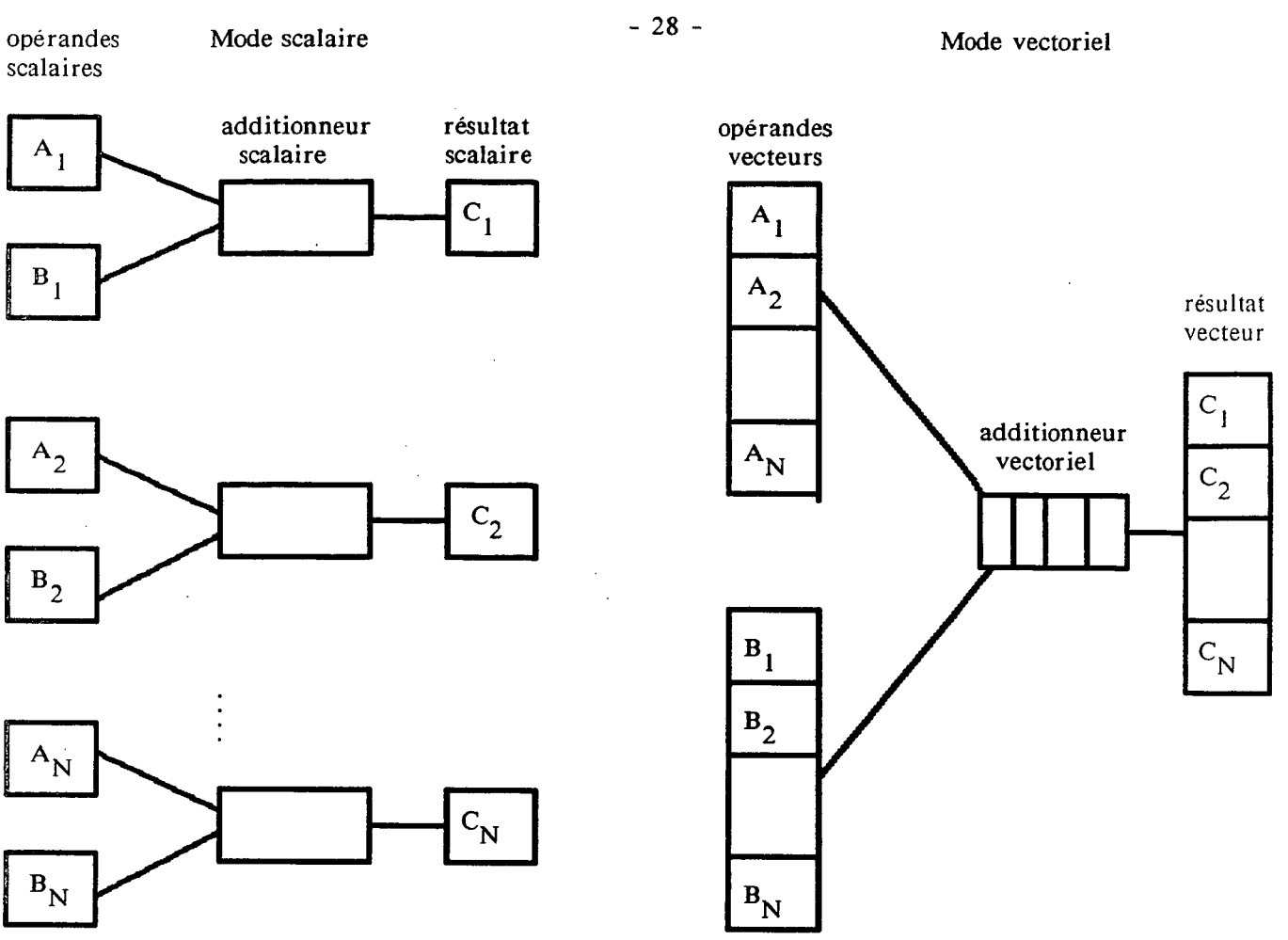


Figure 3.4. N additions en modes scalaire et vectoriel

	Mode scalaire				Mode vectoriel			
cycle	étape 1	étape 2	étape 3	étape 4	étape 1	étape 2	étape 3	étape 4
1	1				1			
2		1			2	1		
3			1		3	2	1	
4				1	4	3	2	1
5	2				5	4	3	2
6		2			6	5	4	3
7			2					
8				2				
9	3							
10		3						
11			3					
12				3	12	11	10	9

Figure 3.5. Vitesses scalaire et vectorielle

La formule ci-dessous donne la vitesse d'exécution d'une opération vectorielle en fonction de la longueur des vecteurs [8]

$$v = r_\infty \frac{n}{n+n_{\frac{1}{2}}}$$

r_∞ est la vitesse asymptotique (pour n=∞)
$n_{\frac{1}{2}}$ est la longueur de demi-performance (pour $v = r_\infty/2$)
$n_{\frac{1}{2}}$ caractérise le temps de start-up et r_∞ le cycle d'horloge

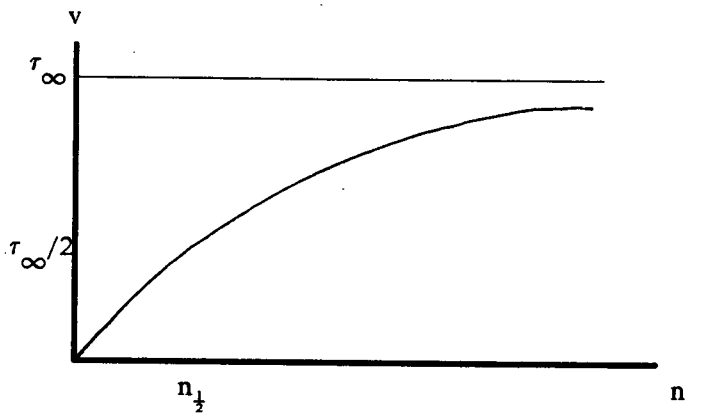


Figure 3.6. vitesse vectorielle et longueur de vecteur

2.4. Parallélisme et chaînage

Une unité de calcul contient en général plusieurs unités fonctionnelles, chaque unité étant vectorielle, c'est-à-dire exécutant en mode pipeline des instructions vectorielles sur des jeux d'opérandes appelés vecteurs et notés V_j. Ces unités peuvent opérer en parallèle sur des vecteurs distincts, c'est-à-dire effectuer des opérations vectorielles indépendantes simultanément (figure 3.7).

Au lieu d'exécuter en même temps des instructions indépendantes, deux unités vectorielles peuvent enchaîner les opérations vectorielles de façon à former un super-pipeline (figures 3.8 et 3.9). Le résultat de l'addition est par exemple un opérande de la multiplication, qui peut débuter dès que les premiers indices sont prêts et fonctionner en mode pipeline avec l'addition.

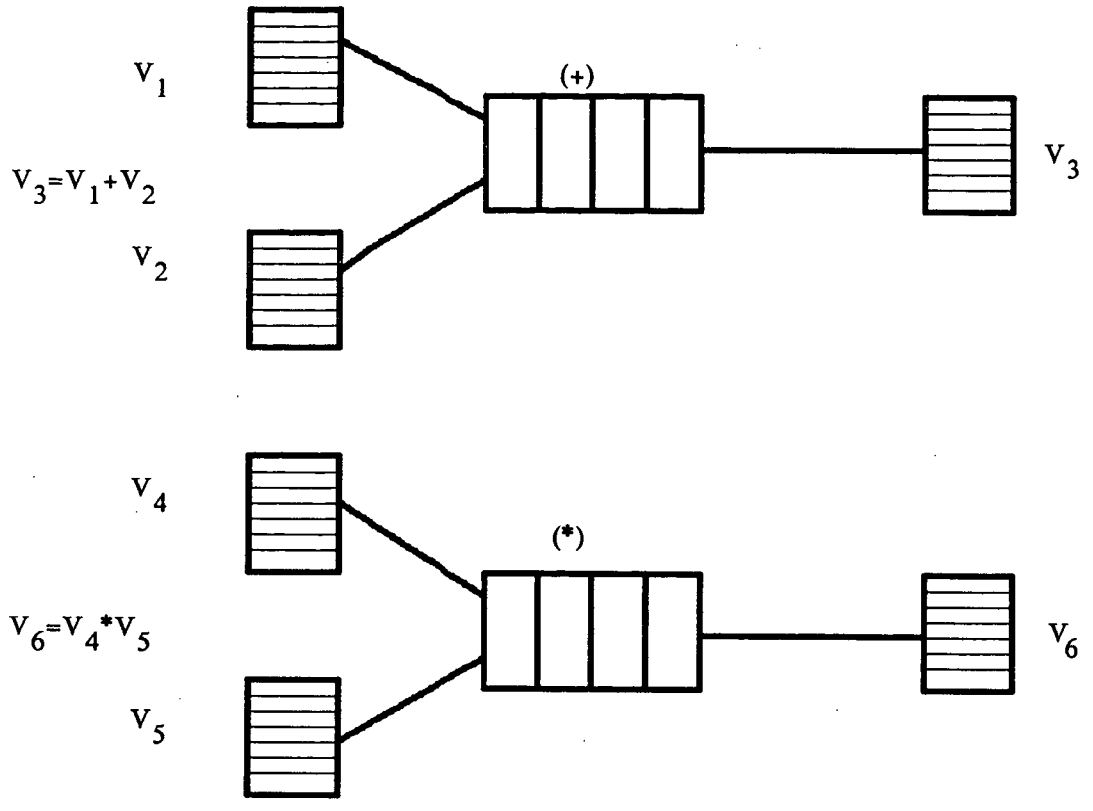


Figure 3.7 Opérations vectorielles en parallèle

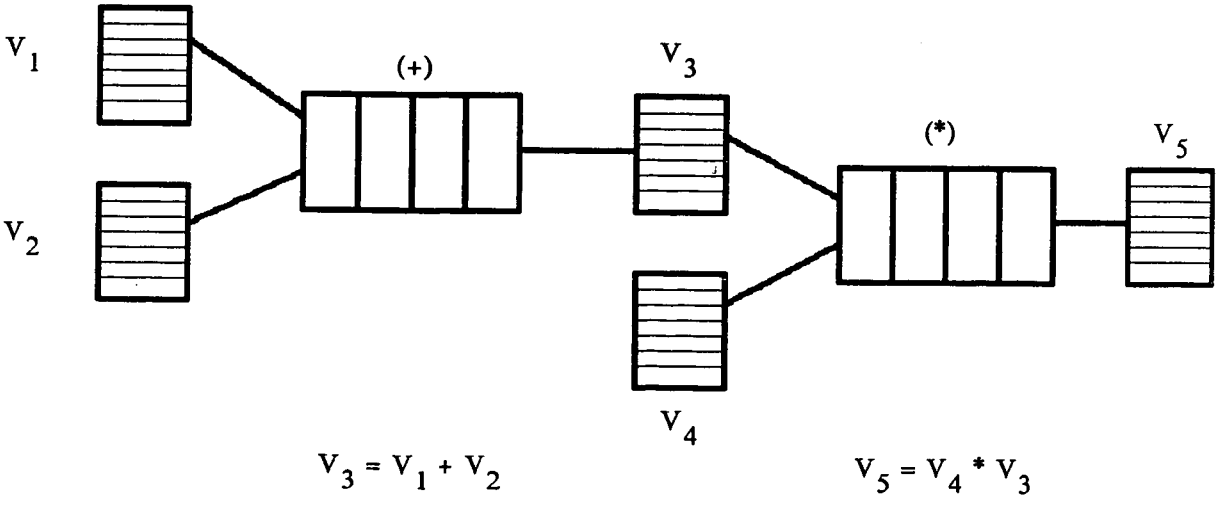


Figure 3.8. Opérations vectorielles enchaînées

	Addition (+)					Multiplication (*)			
cycle	étape 1	étape 2	étape 3	étape 4		étape 1	étape 2	étape 3	étape 4
1	1								
2	2	1							
3	3	2	1						
4	4	3	2	1					
5	5	4	3	2		1			
6	6	5	4	3		2	1		
7	7	6	5	4		3	2	1	
8	8	7	6	5		4	3	2	1
9	9	8	7	6		5	4	3	2

Figure 3.9. Exécution en mode chaînage

2.5. Alimentation des pipelines

Il faut fournir les opérandes et stocker les résultats des unités vectorielles à une cadence suffisante pour soutenir leur débit de production, qui ne peut être ralenti. En général, les données doivent donc arriver à raison de deux opérandes par cycle et repartir à raison d'un résultat par cycle. Or la mémoire qui contient les données est, dans les ordinateurs actuels, environ 10 fois plus lente que l'unité de calcul. Pour soutenir le débit, deux types de solutions ont été adoptés.

- Les données et résultats peuvent transiter directement de mémoire à mémoire, sans intermédiaire apparent entre la mémoire et l'unité de calcul. La mémoire est alors divisée en bancs entrelacés pour soutenir un débit, suffisant, maximal lorsque les éléments de vecteurs sont contigus en mémoire. Cette approche a été retenue par Control-Data et ETA Systems [3].

- Des tampons, appelés registres vectoriels, reçoivent les opérandes de la mémoire et alimentent les unités de calcul à la bonne cadence (vice-versa pour les résultats). Toutefois, la mémoire est encore divisée en bancs entrelacés pour accélérer le transfert des données vers les registres. Mais ceux-ci peuvent être utilisés pour différents calculs afin de limiter les échanges avec la mémoire. Le programme peut jouer avec les registres vectoriels de façon à optimiser les transferts-mémoire qui pénalisent souvent la vitesse d'exécution. Cette approche, est adoptée par beaucoup de vendeurs, tels que CRAY, FUJITSU, HITACHI. Sur CRAY, 8 registres vectoriels contiennent 64 mots (de 64 bits)

chacun, tandis que les registres du FUJITSU VP sont reconfigurables dynamiquement (à l'exécution) en 8 registres de longueur 1024 mots, (8 × 1024) ou 16 × 512, ou 32 × 256, ou 64 × 128, ou 128 × 64. [1,2].

III - ORGANISATION DE LA MEMOIRE

3.1. Interleaving

Le paragraphe précédent soulignait l'importance d'un débit-mémoire suffisant pour alimenter les unités de calcul (directement ou via les registres vectoriels) en données. La mémoire est à cette fin divisée en plusieurs parties appelées bancs, auxquels on peut accéder en parallèle. Si la mémoire contient m bancs, le débit est ainsi multiplié, en théorie, par m.

Les bancs sont dits entrelacés (interleaved) par référence au mode de rangement des tableaux.

Considérons un tableau X de longueur N (déclaré en FORTRAN 77 par DIMENSION X(N)). X est rangé dans les m bancs-mémoire "horizontalement", en entrelaçant les bancs. La figure 3.10 montre le cas m=4.

B1	B2	B3	B4
X(1)	X(2)	X(3)	X(4)
X(5)	X(6)	X(7)	X(8)
X(9)	X(10)	X(11)	X(12)
X(13)	...		

Figure 3.10. Rangement en mémoire d'un tableau X

En mode vectoriel, il faut accéder le plus rapidement possible aux éléments {X(I)/I=1,2,3,...N}. Le débit est accéléré par des accès parallèles aux bancs. A chaque cycle, peut débuter une requête-mémoire (lecture ou écriture). L'accès complet à une donnée nécessite k cycles, de telle sorte qu'un banc est réservé pendant k cycles, dès qu'une requête lui est soumise ; par contre, un autre banc non réservé peut satisfaire une nouvelle requête.

Le schéma de fonctionnement pour accéder aux éléments {X(I)/I=1,2,...N} en mode vectoriel est illustré sur la figure 3.11 (avec m=4 et k=4). Après un temps de démarrage (start-up), on multiplie le débit par m (une donnée tous les [k/m]*) cycles au lieu de k cycles).

cycle	B1	B2	B3	B4
1	X(1)			
2	X(1)	X(2)		
3	X(1)	X(2)	X(3)	
4	X(1)	X(2)	X(3)	X(4)
5	X(5)	X(2)	X(3)	X(4)
6	X(5)	X(6)	X(3)	X(4)
7	X(5)	X(6)	X(7)	X(4)
8	X(5)	X(6)	X(7)	X(8)
9	X(9)	X(6)	X(7)	X(8)
10	X(9)	X(10)	X(7)	X(8)

Figure 3.11. Accès au vecteur {X(I)/I=1,2,...N}

cycle	B1	B2	B3	B4
1		X(2)		
2		X(2)		X(4)
3		X(2)		X(4)
4		X(2)		X(4)
5		X(6)		X(4)
6		X(6)		X(8)
7		X(6)		X(8)
8		X(6)		X(8)
9		X(10)		X(8)
10		X(10)		X(12)

Figure 3.12. Accès au vecteur {X(I)/I=2,4,6,...}

Toutefois, si on veut accéder maintenant aux seuls éléments pairs {X(I)/I=2,4,6,...} du tableau X, alors on n'utilise plus que la moitié des bancs. (figure 3.12). Une requête arrive sur un banc tandis qu'il est encore réservé, ce qui produit un conflit-mémoire et diminue le débit. Dans ce cas, le débit est divisé par deux par rapport au cas précédent.

*) [k/m] est le plus petit entier supérieur à k/m.

De manière générale, il y a conflit-mémoire dès qu'on accède aux éléments d'un tableau avec un pas ℓ qui n'est pas premier avec m. [9]

Dans la plupart des calculateurs vectoriels, m est une puissance de 2 pour des raisons matérielles (hardware). Considérons par exemple le tableau COOR(NDIM,N) représentant les coordonnées de N points dans un espace de dimension NDIM. La figure 3.13 montre le rangement du tableau COOR, dans 4 bancs, dans le cas du plan (NDIM=2). Les éléments du vecteur des coordonnées en x sont donc rangés avec un pas de NDIM. Dans le cas du plan et d'un nombre pair de bancs-mémoire, les conflits divisent le débit par 2. Il est donc préférable de déclarer un tableau ACOOR(N,NDIM), afin d'éviter un conflit-mémoire potentiel (figure 3.13).

En outre, certains calculateurs vectoriels, notamment ETA^{10} qui fonctionne de mémoire à mémoire, sont nettement plus efficaces si les éléments des vecteurs sont contigus en mémoire.

En règle générale, il vaut mieux utiliser des vecteurs avec un pas égal à 1.

B1	B2	B3	B4
COOR(1,1)	COOR(2,1)	COOR(1,2)	COOR(2,2)
COOR(1,3)	COOR(2,3)	COOR(1,4)	COOR(2,4)
...	...	...	...
ACOOR(1,1)	ACOOR(1,2)	ACOOR(1,3)	ACOOR(1,4)
...	...	...	...
...	...	ACOOR(1,N)	ACOOR(2,1)
ACOOR(2,2)	ACOOR(2,3)	ACOOR(2,4)	ACOOR(2,5)
...	...	...	...

Figure 3.13. Rangement des tableaux COOR(2,N) et ACOOR(N,2)

3.3. Chemins d'accès

La plupart des opérations nécessitent deux opérandes et un résultat, donc trois registres vectoriels et souvent trois transferts-mémoire. Le CRAY-1 par exemple ne

contient qu'un seul chemin d'accès à la mémoire, qui constitue un goulot d'étranglement dans une majorité de programmes. Il est alors primordial d'optimiser l'utilisation des registres pour limiter le nombre des transferts-mémoire, en gardant les opérandes communs à plusieurs opérations et ne stockant pas en mémoire les résultats intermédiaires. Si cette remarque reste vraie pour tous les ordinateurs à registres vectoriels, elle est tempérée sur le CRAY-XMP et le FUJITSU VP par exemple par l'introduction de chemins multiples. Le CRAY-XMP a deux chemins d'accès dans le sens mémoire-registres et un dans l'autre sens, tandis que le FUJITSU VP dispose de deux chemins d'accès à double sens. On peut ainsi transférer en parallèle plusieurs données et accélérer la cadence d'alimentation en vecteurs. Toutefois, un mécanisme doit synchroniser les transferts pour résoudre les conflits (lecture après écriture par exemple).

3.4. Chaînage

De la même façon que deux opérations arithmétiques peuvent être enchaînées pour former un super pipeline, la lecture d'un opérande ou l'écriture d'un résultat peuvent s'enchaîner avec une opération. On peut ainsi obtenir un double chaînage, combinant une lecture, une addition et une multiplication (voire un triple chaînage avec l'écriture, mais ce n'est pas possible sur le CRAY-1 par exemple) (figure 3.14). En effet, la lecture (LOAD) ou l'écriture (STORE) de mémoire à registre est une unité fonctionnant en mode pipeline.

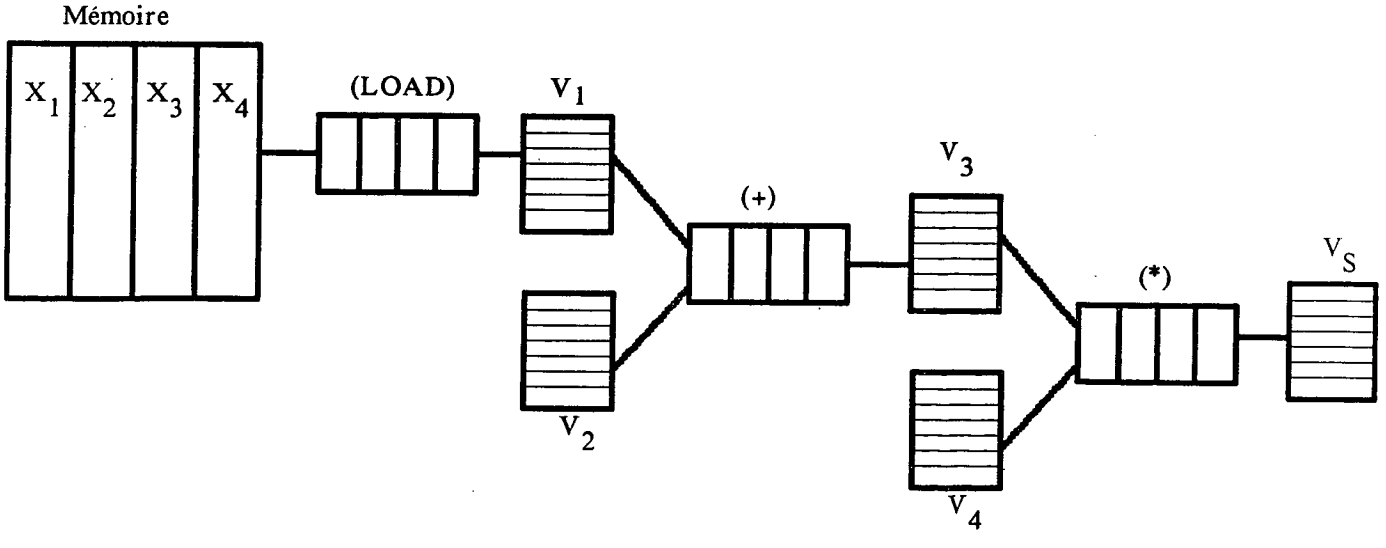


Figure 3.14. Double chaînage

IV - VECTORISATION

4.1. Code vectoriel

Nous avons vu qu'un fort pourcentage d'un programme doit être exécuté en mode vectoriel pour obtenir de bonnes performances. Nous allons maintenant définir la notion de code vectoriel.

Les parties de code vectorisables sont les boucles. Tout code autre qu'une boucle est scalaire. Le compilateur vectoriel essaie de traduire les boucles en instructions vectorielles. Il suit pour cela un certain nombre de règles, communes à tous les compilateurs, sauf pour certains cas particuliers. De nombreux vectoriseurs, de plus en plus efficaces ont été développés récemment. [10,11]. Contrairement aux compilateurs qui génèrent du code assembleur, les vectoriseurs transforment le programme source (en FORTRAN) en FORTRAN, mieux adapté à la vectorisation par le compilateur.

4.2. Vecteurs

L'indice de la boucle joue un rôle essentiel dans la vectorisation et permet de définir la notion de vecteur. La figure 3.15 indique les différentes indexations possibles, pour un tableau unidimensionnel.

I : indice de boucle (DO 1 I=1,...)
X : tableau unidimensionnel (DIMENSION X(1000))
A,B : constantes entières dans la boucle (indépendantes de I)
INDEX : tableau d'entiers unidimensionnel (INTEGER INDEX(100))

indexation	type d'adressage
X(I+B)	Vecteur contigu
X(A*I+B)	Vecteur à pas constant
X(INDEX(I))	Vecteur à adressage indirect

Figure 3.15. Vecteurs (tableaux unidimensionnels)

Le cas des tableaux multidimensionnels peut être traité différemment selon l'architecture vectorielle. De manière générale, un seul indice doit dépendre de l'indice de boucle I, les autres étant des constantes indépendantes de I. La figure 3.16 donne quelques exemples.

I : indice de boucle (DO 1 I=...)

Y : tableau à deux dimensions (DIMENSION Y(1000,1000))

A,B,C,D,J : constantes entières dans la boucle (indépendantes de I)

indexation	adressage
X(I+B,J)	Vecteur contigu
Y(J,I+B) Y(A*I+B,J) Y(J,A*I+B)	Vecteur à pas constant
Y(INDEX(I),J) Y(J,INDEX(I))	Vecteur à adressage indirect
Y(A*I+B,C*I+D)	Vecteur à pas constant

Figure 3.16. Vecteurs - cas des tableaux multidimensionnels

Certaines variables scalaires peuvent être "étendues" à des vecteurs par le vectoriseur, si cette transformation s'avère nécessaire et possible. Dans l'exemple de la figure 3.17, T est transformé en vecteur TV, S reste scalaire.

```
      DO 1 I=1,N
         T = X(I) + S * Y(I)
         Z(I) = T * Z(I)
1     CONTINUE
```

code original

```
      DO 1 I=1,N
         TV(I) = X(I) + S * Y(I)
         Z(I) = TV(I) * Z(I)
1     CONTINUE
```

code après extension de T en TV

Figure 3.17. Extension de scalaires en vecteurs

4.3. Boucle vectorielle

Une boucle contient en général des instructions de contrôle (IF,...) et des instructions d'affectation (X(I)=...). Un cas important est celui de boucles ne contenant que

des instructions d'affectation. De telles boucles doivent remplir certaines conditions pour être vectorielles. Dans la suite, les indices de boucle sont normalisées de façon à se ramener au cas : DO 1 I=1,N.

4.4. Une instruction d'affectation

Soit B la boucle :

```
      DO  1  I=1,N
           (S)
  1   CONTINUE
```

l'instruction d'affectation S est vectorielle si toutes ses occurrences, S(I) pour toutes les valeurs de l'indice I de la boucle, peuvent être exécutées simultanément. En effet, plusieurs indices sont traités en parallèle dans la chaîne vectorielle. Donc l'instruction ne doit pas être récurrente, les opérandes d'un indice J ne peuvent être le résultat d'un indice antérieur I < J.

Définition :

Soit IN(S(I)) l'ensemble des opérandes de l'occurrence S(I), et OUT(S(I)) le résultat.

Proposition :

Une boucle normalisée de longueur N avec une instruction d'affectation S est vectorielle si et seulement si

$$(\forall I,J)\ (1 \leq I < J \leq N)\quad IN(S(J)) \cap OUT\ (S(I)) = \emptyset$$

Exemples :

```
      DO  1  I=1,N
           X(I) = Y(I) + A(I) * X(I-1)
  1   CONTINUE
```

La boucle n'est pas vectorielle car

$$IN(S(I)) \cap OUT(S(I-1)) = \{X(I-1)\}$$

```
      DO  2  I=1,2 * N,2
           X(I)  = Y(I) + A(I) * X(I-1)
2     CONTINUE
```

Après normalisation, la boucle s'écrit

```
      DO  2  I=1,N,1
           X(2*I) = Y(2*I) + A(2*I) * X(2*I-1)
2     CONTINUE
```

La boucle est vectorielle car $\forall\ I < J\ \ 2*I \neq 2*J-1$ d'où $OUT(S(I)) \cap IN(S(J)) = \emptyset$.

Proposition :

La récurrence linéaire ci-dessous n'est pas vectorielle pour $1 \leq M \leq N-1$

```
      DO  1  I=1,N
           X(I) = ... X(I-M) ...
1     CONTINUE
```

Proposition :

```
      DO  3  I=1,N
           X(INDEX(I)) = X(INDEX(I)) + Y(I)
3     CONTINUE
```

Cette boucle est vectorielle si et seulement si $\forall\ I < J\ \ INDEX(I) \neq INDEX(J)$ c'est à dire si INDEX est injective.

4.5. Plusieurs instructions d'affectation

Soit la boucle suivante

```
      DO  1  I=1,N
           S1
           S2
           .
           .
           Sk
           .
           .
           Sℓ
           .
           .
           Sm
1     CONTINUE
```

où toutes les instructions S_k ($1 \leq k \leq m$) sont des affectations. Pour effectuer cette boucle en mode vectoriel, sans modifier l'ordre des instructions, il faut et suffit d'une part que toutes les instructions soient vectorielles et d'autres part qu'on puisse distribuer la boucle c'est-à-dire exécuter toutes les occurrences de S_1, puis toutes celles de S_2, etc. Si l'on compare au mode scalaire, il faut donc pouvoir exécuter $S_k(J)$ avant $S_\ell(I)$, avec $1 \leq k < \ell \leq m$ et $1 \leq I < J \leq N$ (figure 3.18), afin d'obtenir les mêmes résultats.

mode scalaire	mode vectoriel
$S_k(I)$	$S_k(I)$
$\vdots$	$\vdots$
$S_\ell(I)$	$S_k(J)$
$\vdots$	$\vdots$
$S_k(J)$	$S_\ell(I)$
$\vdots$	$\vdots$
$S_\ell(J)$	$S_\ell(J)$

Figure 3.18. Condition de vectorisation

L'interversion de $S_k(J)$ et $S_\ell(I)$ est possible à deux conditions :

1) les opérandes de $S_k(J)$ ne sont pas un résultat de $S_\ell(I)$, autrement dit, on ne peut consommer avant d'avoir produit, soit

$$\text{IN}\ (S_k(J)) \cap \text{OUT}\ (S_\ell(I)) = \emptyset$$

2) le résultat de $S_k(J)$ n'est pas un opérande de $S_\ell(I)$, autrement dit, on ne peut reproduire avant d'avoir déjà consommé, soit

$$\text{OUT}\ (S_k(J)) \cap \text{IN}\ (S_\ell(I)) = \emptyset$$

En ajoutant la condition de non récurrence pour chaque instruction, on obtient le théorème suivant, appelé Conditions de Bernstein.

Théorème :

Une boucle normalisée de longueur N, avec m instructions d'affectation S_k ($1 \leq k \leq m$) est vectorielle sans réordonner si et seulement si

$$(\forall I,J) \quad (1 \leq I < J \leq N)$$

(1) $$(\forall k)\,(1 \leq k \leq m) \quad IN\ (S_k(J)) \cap OUT\ (S_\ell(I)) = \emptyset$$

(2) $$(\forall k,\ell)\,(1 \leq k < \ell \leq m) \quad \begin{cases} IN\ (S_k(J)) \cap OUT\ (S_\ell(I)) = \emptyset \\ OUT\ (S_k(J)) \cap IN\ (S_\ell(I)) = \emptyset \end{cases}$$

(1) est la condition de non-récurrence des instructions

(2) est la condition de distribution des instructions en plusieurs boucles.

4.6. Réordonner les instructions

Les compilateurs de première génération limitent leur analyse à ces conditions. Mais les vectoriseurs de seconde génération vont plus loin et cherchent à réordonner les instructions ou à vectoriser partiellement en distribuant la boucle, ce qui devait être fait "à la main" avant leur apparition. Toutefois, une bonne connaissance du problème permet au programmeur de réordonner et de distribuer les instructions, de façon à optimiser le chaînage des opérations vectorielles et l'utilisation des registres vectoriels.

Exemples :

	DO 1 I=1,N		DO 2 I=1,N	
S_1	X(I) = Y(I) + 1		Y(I+1) = A(I) + 2	S'_1
S_2	Y(I+1) = A(I) + 2		X(I) = Y(I) + 1	S'_2
1	CONTINUE	2	CONTINUE	
	IN $(S_1(J))$ = {Y(J)}		OUT $(S'_1(J))$ = {Y(J+1)}	
	OUT $(S_2(J-1))$ = {Y(J)}		IN $(S'_2(I))$ = {Y(I)}	
	Boucle non vectorielle		Boucle vectorielle après interversion des instructions	

Pour réordonner les instructions d'une boucle sans changer le résultat, il faut et il suffit, d'une part que les interversions soient possibles en mode scalaire, d'autre part que les conditions de Bernstein soient satisfaites pour la boucle réordonnée.

Mode scalaire	Mode scalaire après interversion	Mode vectoriel
$S_k(I)$	$S_\ell(I)$	$S_\ell(I)$
$S_\ell(I)$	$S_k(I)$	$S_\ell(J)$
$S_k(J)$	$S_\ell(J)$	$S_k(I)$
$S_\ell(J)$	$S_k(J)$	$S_k(J)$

Figure 3.19. Conditions d'interversion et vectorisation

Les conditions sont analogues aux conditions de Bernstein inversées, avec cette fois $(\forall I,J)\ (1 \leq I \leq J \leq N)$, pour inclure la condition d'interversion des deux instructions.

4.7. Dépendances de données

Pour détecter automatiquement les possibilités de vectorisation, on est amené à formaliser l'analyse "manuelle" en introduisant la notion de graphe de dépendances.

Définition :

Soit S_k et S_ℓ $(1 \leq k < \ell \leq m)$ deux instructions d'affectation d'une boucle normalisée de longueur N.

S_k dépend d'elle-même si $(\exists I,J)(1 \leq I < J \leq N)\ \ IN\left(S_k(J)\right) \cap OUT\left(S_k(I)\right) \neq \emptyset$

S_ℓ dépend de S_k si :

$(\exists I,J) \quad (1 \leq I \leq J \leq N)$

$IN\left(S_\ell(J)\right) \cap OUT\left(S_k(I)\right) \neq \emptyset$ (dépendances de données)

$OUT\left(S_\ell(J)\right) \cap IN\left(S_k(I)\right) \neq \emptyset$ (anti dépendances)

$OUT\left(S_\ell(J)\right) \cap OUT\left(S_k(I)\right) \neq \emptyset$ (dépendances externes)

S_k dépend de S_ℓ si :

$(\exists I,J) \quad (1 \leq I < J \leq N)$

$IN\left(S_k(J)\right) \cap OUT\left(S_\ell(I)\right) \neq \emptyset$

$OUT\left(S_k(J)\right) \cap IN\left(S_\ell(I)\right) \neq \emptyset$

$OUT\left(S_k(J)\right) \cap OUT\left(S_k(I)\right) \neq \emptyset$

Note : Les définitions diffèrent par l'inégalité entre I et J.

Si S_k dépend de S_ℓ, la boucle n'est pas vectorielle sans réordonner. Réciproquement, si S_ℓ dépend de S_k, on ne peut intervertir S_k et S_ℓ. Par suite, si S_k et S_ℓ dépendent l'une et l'autre, la boucle doit être scalaire.

Définition :

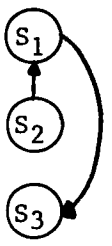
Le graphe orienté G de dépendances de données d'une boucle normalisée de longueur N avec m instructions d'affectations est défini de la façon suivante :

- les m noeuds du graphe sont les instructions S_k ($1 \leq k \leq m$)
- un arc va de S_k vers S_ℓ si S_ℓ dépend de S_k

Exemple :

```
        DO 1 I = 1,N
(S1)        X(I) = Y(I) + B(I)
(S2)        Y(I+1) = A(I) + 2
(S3)        B(I) = B(I) + 1
```

Graphe :



S_1 dépend de S_2 (en données)

S_3 dépend de S_1 (anti -)

Les arcs du graphe indiquent l'ordre d'exécution qu'il faut respecter. Un cycle dans le graphe doit être exécuté en mode scalaire. Un algorithme permet de construire le graphe et de déterminer les cycles.

Définition :

Le graphe quotient $G/\mathcal{C}$ du graphe de dépendances G par les cycles $\mathcal{C}$ est obtenu ainsi :

- les noeuds de $G/\mathcal{C}$ sont les cycles (ou les instructions seules)
- un arc relie un cycle C_1 à un cycle C_2 s'il existe un arc d'une instruction de C_1 vers une instruction de C_2.

Remarque :

Ce graphe est le quotient de G par la relation d'équivalence "$S_k \mathcal{R} S_\ell$ <=> il existe un chemin de S_k vers S_ℓ ou S_ℓ vers S_k".

Ce graphe-quotient permet de déterminer ce qui est vectoriel ou non dans la boucle. La première étape consiste à distribuer la boucle entre les cycles, en les réordonnant en fonction des arcs du graphe, pour respecter l'ordre des dépendances de données. Ensuite, on analyse chaque boucle formée d'un seul cycle :

- un cycle à plus de deux éléments est scalaire : en effet, dans quelque ordre des instructions, les Conditions de Bernstein ne sont pas satisfaites.

- un cycle à une instruction récurrente est scalaire.

- un cycle à une instruction non récurrente est vectoriel.

La troisième étape cherche à optimiser le chaînage en regroupant dans une même boucle plusieurs instructions vectorielles. Il existe plusieurs solutions depuis une instruction par boucle jusqu'à toutes les instructions dans la même boucle, en passant par une interversion d'instructions indépendantes. L'optimisation dépend de l'architecture cible (nombre de registres vectoriels notamment), et de nombreux critères.

Un programmeur peut actuellement trouver une meilleure solution qu'un vectoriseur automatique.

4.8. Cas particuliers d'instructions récurrentes

En fait, les ordinateurs vectoriels disposent de mécanismes spéciaux dans les pipelines pour vectoriser certaines instructions récurrentes. Le compilateur sait les reconnaître et générer du code vectoriel.

Le cas le plus fréquent est la somme des éléments d'un vecteur utilisée pour effectuer un produit scalaire. Les boucles ci-dessous sont vectorielles :

```
      S = 0.0                    S = 0.0
      DO  1  I=1,N               DO  1  I=1,N
           S = S + X(I)               S = S + X(I) * Y(I)
1     CONTINUE                   1  CONTINUE
```

Toutefois, le calcul vectoriel d'un produit scalaire est souvent plus lent qu'une opération purement vectorielle.

De même, la recherche du plus grand (plus petit) élément d'un vecteur est en général vectorisable (pas sur CRAY-1 toutefois). La boucle ci-dessous est vectorielle (sauf exception)

```
      BIG = 0.0
      DO  1  I=1,N
           BIG = MAX(BIG,ABS(X(I))
1     CONTINUE
```

Certains ordinateurs (Hitachi S810 par exemple) peuvent également vectoriser les récurrences linéaires d'ordre un, comme ci-dessous :

```
      DO  1  I=1,N
           X(I) = X(I) - A(I) * X(I-1)
1     CONTINUE
```

Ces récurrences interviennent dans la résolution de systèmes tridiagonaux. Toutefois, même en mode vectoriel, on n'obtient pas les performances optimales de la machine.

4.9. Instructions de contrôle

Jusqu'ici, seules les instructions d'affectation ont été analysées. Les instructions de contrôles les plus fréquentes sont les DO, CALL, READ/WRITE, GOTO, IF.

En général seule la boucle la plus interne est vectorisée. Une boucle vectorielle ne contient donc pas d'instruction DO. Toutefois, certaines boucles imbriquées peuvent être compilées comme une seule boucle vectorielle, ou interverties pour vectoriser en fait la boucle externe.

Actuellement, seuls les appels à des fonctions mathématiques usuelles (ABS, SQRT, MAX, SIN, COS,...) sont vectoriels. Une boucle vectorielle ne contient donc pas d'instruction CALL.

De même les lectures/écritures ne sont vectorisées que dans certains cas particuliers.

Une programmation sans GOTO est toujours préférable, c'est encore plus vrai en mode vectoriel.

Si le test d'une instruction conditionnelle IF est vectoriel (non récurrent) et si les instructions des groupes THEN, ELSE sont vectorielles, alors la séquence IF ... THEN ... ELSE ... END est vectorielle. La technique de vectorisation varie avec la machine. Un vecteur masque détermine les indices pour lesquels le test est vrai et les vecteurs de calcul ne contiennent que les "bons" indices grâce à une opération de type GATHER, effectuée soit au niveau de la mémoire, soit au niveau des registres. Dans tous les cas, les performances sont pénalisées par le traitement de la condition.

4.10. Directives de compilation

Dans certains cas, le compilateur ne peut détecter si une instruction est récurrente ou non. L'utilisateur peut ajouter au code FORTRAN des cartes commentaires d'un format spécifique, (variable d'une machine à l'autre), reconnues par le compilateur comme des directives. Par exemple, les boucles ci-dessous ne seront vectorisées que si le programmeur ajoute en tête de boucle une directive de non récurrence.

```
      DO  1  I=1,N                    DO  1  I=1,N
          X(I+J1) = ... X(I+J2) ...           X(INDEX(I)) = ... X(INDEX(I)...
1     CONTINUE                        1     CONTINUE
```

vectorielle si $J_1 \leq J_2$ ou $J_1 \geq J_2 + N$

vectorielle si INDEX injective

D'autres directives, variables suivant la machine, permettent de spécifier la longueur d'une boucle, le taux de vérité d'une instruction conditionnelle, etc.

4.11. Résumé

Une boucle est vectorielle au moins dans le cas suivant :

- elle est la plus interne
- elle n'a pas d'instruction de contrôle (DO, CALL, READ/WRITE,GOTO,IF) donc pas de rupture de séquence
- les instructions d'affectation ne sont pas récurrentes
- une instruction est indépendante des instructions suivantes
- un seul indice dans les tableaux varie avec l'indice de boucle.

D'autre part :

- L'adressage des vecteurs à pas constants voire contigus est le plus rapide
- L'adressage indirect est lent, voire scalaire
- Les vecteurs longs donnent de meilleurs performances
- Les appels mémoire sont coûteux
- Beaucoup d'opérations dans une boucle facilitent le chaînage et le parallélisme des opérations arithmétiques.

CHAPITRE 4 : CODES EULER ET NAVIER-STOKES VECTORIELS

I. ALGORITHMES

1.1. Schéma de Lax-Wendroff

Chaque itération en temps comporte plusieurs phases de calcul :

- Viscosité, artificielle pour les équations d'Euler, physique dans le cas des équations de Navier-Stokes. Les opérations sont analogues à des calculs de dérivées aux noeuds.
- Pas de temps local, à chaque noeud
- Prédicteur, correcteur, avec calcul de flux
- Conditions aux limites.

1.2. Schéma de Van Leer

Les étapes de calcul d'une itération en temps sont similaires :

- Dérivées aux noeuds des variables "physiques"
- Pas de temps local
- Prédicteur
- Correcteur, avec calcul de flux
- Conditions aux limites

1.3. Structures de données

Les données de l'algorithme sont stockées dans trois types de tableaux en fonction de l'élément géométrique :

- noeuds : pour chaque noeud, on définit les variables aux itérations précédente et courante, les coordonnées du point, ...
- triangles : caractéristiques topologiques telles que aire, numéros de sommets,...
- arêtes : le même genre de tableau.

En outre, on distingue les données dites constantes, qui définissent par exemple la topologie du domaine, des données dites variables comme la vitesse, le pas de temps...

1.4. Vectorisation

Les paragraphes suivants décrivent en détail les techniques utilisées pour optimiser la vectorisation de chaque étape de calcul. De manière générale, chacune peut être représentée par l'un des deux schémas suivants :

* boucle vectorielle sur les noeuds (conditions aux limites notamment)
* boucle sur les triangles ou arêtes, décomposée en trois parties :
  - transfert des tableaux de noeuds à des tableaux de triangles (GATHER)
  - boucle vectorielle sur les triangles
  - accumulation des résultats élémentaires aux noeuds (SCATTER).

La partie GATHER est en fait un adressage indirect, vectorisé ou non suivant le compilateur. Les boucles vectorielles ont été distribuées en plusieurs boucles de façon à optimiser le code. Cette technique a permis d'exploiter le parallélisme et le chaînage des unités vectorielles, tout en minimisant les appels-mémoire et en améliorant la gestion des registres vectoriels.

Toutefois, cette solution introduit de nombreux vecteurs temporaires. Pour limiter la place mémoire occupée, on a découpé l'ensemble des indices (nombre de triangles par exemple) en paquets de longueur égale, qui représente ainsi la longueur des vecteurs dans chaque boucle. Ce paramètre est choisi en fonction de l'espace mémoire disponible et du calculateur (temps de start-up des unités vectorielles ou longueur de demi-performance $n_{\frac{1}{2}}$ [8]).

L'accumulation des résultats élémentaires aux noeuds n'est vectorielle que si les noeuds considérés sont différents. Un coloriage des éléments permet de définir des paquets d'éléments sans noeud commun. Les contributions aux noeuds des éléments d'une même couleur sont ainsi indépendantes et vectorielles. Cette opération nécessite un adressage indirect de type SCATTER.

II. CALCUL DES DERIVEES

Pour calculer les dérivées aux noeuds, il est nécessaire de calculer les dérivées élémentaires puis de les assembler aux noeuds. En effet, l'interpolation étant P1, les dérivées approchées sont de type P_0. Cette remarque est générale aux codes d'éléments finis et l'analyse qui suit reste valable dans d'autres contextes de simulations.

On peut formuler ainsi l'expression des dérivées :

$$dtri(jt) = \sum_{k=1}^{3} u(nu(k,jt)) * cq(k,jt)$$

$$du(is) = \Big(\sum_{jt \in supp(is)} dtri(jt) \Big) / airs(is)$$

où

dtri(jt) est la dérivée dans le triangle jt

u est la variable à dériver

nu(k,jt) est le numéro du k^e noeud du triangle jt

cq(k,jt) est la dérivée de la fonction de base associée

du(is) est la dérivée de u au noeud is

supp(is) est le support du noeud is, c'est-à-dire l'ensemble des triangles contenant le noeud is (figure 4.1)

airs(is) est l'aire du support de is

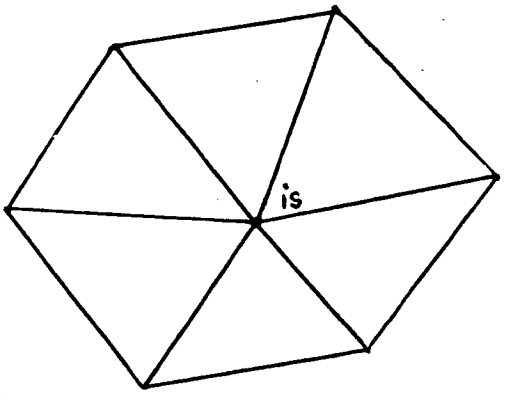


Figure 4.1. Support du noeud is

Le code Fortran associé est analogue à celui-ci :

```
      DO  1  IS = 1,NS
      DU(IS) = O.DO
1     CONTINUE
      DO  2  JT = 1,NT
      DTRI =   U(NU(1,JT)) * CQ(1,JT) +
               U(NU(2,JT)) * CQ(2,JT) +                    (2.1)
               U(NU(3,JT)) * CQ(3,JT)
      DU(NU(1,JT)) = DU(NU(1,JT)) + DTRI
      DU(NU(2,JT)) = DU(NU(2,JT)) + DTRI                   (2.2)
      DU(NU(3,JT)) = DU(NU(3,JT)) + DTRI
2     CONTINUE
      DO  3  IS = 1,NS
      DU(IS) = DU(IS) * USAIRS(IS)
3     CONTINUE
```

Les boucles 1 et 3 sont trivialement vectorielles. Nous allons considérer simplement les instructions (2.1) et (2.2) de la boucle 2. Pour cela, nous pouvons distribuer la boucle 2, en introduisant le vecteur temporaire DTRI(1:NT)

```
      DO  20  JT = 1,NT
      DTRI(JT) = U(NU(1,JT)) * CQ(1,JT) + ...
20    CONTINUE
      DO  21  JT = 1,NT
      DU(NU(1,JT)) = DU(NU(1,JT)) + DTRI(JT)
21    CONTINUE
```

La boucle 20 est vectorielle. Toutefois, elle n'est efficacement vectorisée que si le calculateur possède des instructions matérielles traitant l'adressage indirect des vecteurs. Sinon, notamment sur le Cray-1, il faut utiliser une primitive dite GATHER (figure 4.2)

```
      DO   1   I = 1,N
      Y(I) = X(INDEX(I))          <=>  CALL GATHER (N,Y,X,INDEX
1     CONTINUE

      DO   2   I = 1,N
      X(INDEX(I)) = Y(I)          <=>  CALL SCATTER (N,X,INDEX,Y)
2     CONTINUE
```

Figure 4.2. GATHER et SCATTER opérations

La boucle 20 peut être réécrite en utilisant le vecteur temporaire UT(1:NT) :

```
      CALL GATHER (3 * NT , UT, U, NU)
      DO   20  JT = 1,NT
      DTRI(JT) =   UT(3 * JT - 2) * CQ(1,JT) +
                   UT(3 * JT - 1) * CQ(2,JT) +
                   UT(3 * JT) * CQ(3,JT)
20    CONTINUE
```

La boucle 21 contient des dépendances de données et n'est donc pas vectorielle. En effet, la fonction NU : JT → {NU(1,JT),NU(2,JT),NU(3,JT)} n'est pas injective, comme l'illustre la figure 4.3

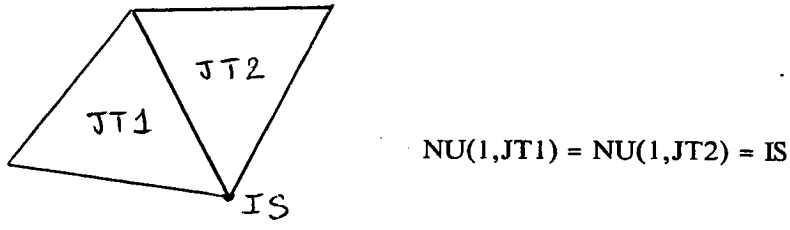


Figure 4.3. La fonction NU n'est pas injective

Pour vectoriser la boucle 21, on peut séparer l'ensemble des triangles en paquets de telle sorte que deux triangles d'un même groupe n'aient aucun sommet commun. Dans ce cas, la restriction de la fonction NU à chaque partie est injective, et on peut donc vectoriser le code correspondant.

Pour définir une telle partition, on est amené à considérer le graphe suivant $(\mathcal{C}, \mathcal{V}_{\mathcal{C}})$, schématisé sur la figure 4.4.

$\mathcal{C}$ est l'ensemble des triangles

Une arête de $\mathcal{V}_{\mathcal{C}}$ relie deux triangles qui ont au moins un sommet commun

$$(JT1, JT2) \in \mathcal{V}_{\mathcal{C}} \iff \exists\ IS \ /\ IS \in JT1 \cap JT2$$

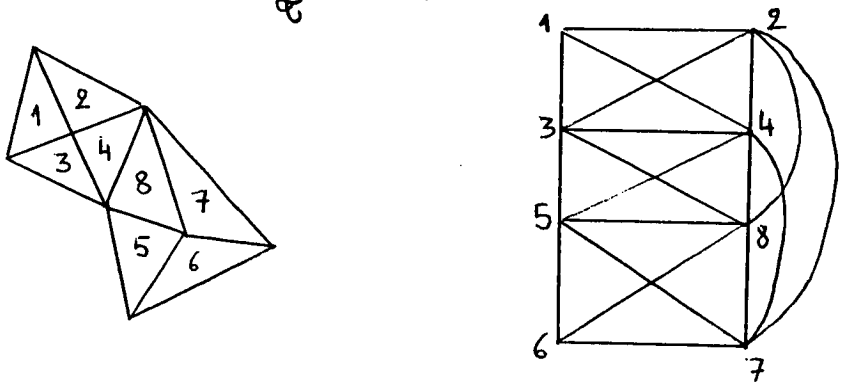


Figure 4.4. Maillage et graphe $(\mathcal{C}, \mathcal{V}_{\mathcal{C}})$ associé

Déterminer un groupe d'éléments sans noeud commun consiste à colorier le graphe $(\mathcal{C}, \mathcal{V}_{\mathcal{C}})$, c'est à dire à attribuer une couleur à chaque élément de façon que deux éléments de même couleur ne soient pas adjacents dans le graphe $(\mathcal{C}, \mathcal{V}_{\mathcal{C}})$. On utilise un algorithme de coloriage général, [12], car le graphe considéré n'est pas planaire, comme l'illustre la figure 4.4 ; le nombre chromatique NC, c'est à dire le nombre minimal de couleurs nécessaires, est borné par la formule suivante :

$$\max_{1 \le i \le NS} d_i \le NC \le 3 * \max_{1 \le i \le NS} d_i - 5$$

où d_i est le nombre de noeuds voisins du noeud i dans la triangulation.

On renumérote les éléments par couleur, ce qui permet de remplacer la boucle 21 par deux boucles imbriquées, dont la plus interne est cette fois vectorielle, puisqu'on a supprimé les dépendances de données.

```
      DO  22  IC = 1,NC
      DO  23  JT = ICOL(IC-1) + 1, ICOL(IC)
      DU(NU(1,JT)) = DU(NU(1,JT)) + DTRI(JT)
      ...
23    CONTINUE
22    CONTINUE
```

Il faut ajouter une directive de compilation pour forcer la vectorisation qui ne peut être détectée automatiquement. D'autre part, on utilise comme dans la boucle 20, un adressage indirect des tableaux. Le gain est très faible si l'on doit avoir recours aux primitives GATHER puis SCATTER (figure 4.24).

La technique de coloriage est donc plus efficace sur le Cray-XMP que sur le Cray-1.

III - CALCUL DU PAS DE TEMPS

La plupart des méthodes de résolution d'équations évolutives requièrent le calcul d'un nouveau pas de temps à chaque itération. L'utilisation d'un pas de temps local accélère la convergence vers une solution stationnaire, tout en respectant le critère de stabilité (condition de CFL). Il est important d'optimiser cette étape de calcul qui est commune à de nombreux codes d'éléments finis.

il faut calculer des pas de temps dans chaque triangle pour déterminer les pas de temps aux noeuds. Le calcul peut se formuler ainsi :

$$dt = \min_{jt\in\{1...nt\}} (dtt(jt)) * cfl$$

$$dtl(is) = \min_{jt\in supp(is)} (dtt(jt)) * cfl/airs(is)$$

$$dtt(jt) = \min (dxm/(umax + cmax), dym/(vmax+cmax))$$

où

dxm et dym sont les hauteurs minimales du triangle

$umax = \max(|u_1|, |u_2|, |u_3|)$

$vmax = \max(|v_1|, |v_2|, |v_3|)$

$cmax = \sqrt{(\gamma * pmax/\rho min)}$

$pmax = \max(|p_1|, |p_2|, |p_3|)$

$\rho min = \min(|\rho_1|, |\rho_2|, |\rho_3|)$

où les variables physiques (ρ,u,v,p) sont considérées aux trois noeuds du triangle jt.

Le code Fortran contient donc des boucles du type suivant :

```
      DT = DTMAX
      DO  1  IS = 1,NS
      DTL(IS) = DTMAX
1     CONTINUE
      DO  2  JT = 1,NT
      UMAX =    MAX (ABS(UA2(NU(1,JT))),
                     ABS(UA2(NU(2,JT))))
      UMAX = MAX(UMAX,ABS(UA2(NU(3,JT))))
      VMAX = ...
      PMAX = ...
      ROMIN = ...
      CMAX = SQRT(GAMMA * PMAX / ROMIN)
      DTT = MIN(DXM(JT)/(UMAX + CMAX),
                DYM(JT) / (VMAX + CMAX))
      DT = MIN(DT,DTT)
      DTL(NU(1,JT)) = MIN(DTL(NU(1,JT)),DTT)
      DTL(NU(2,JT)) =  MIN(DTL(NU(2,JT)),DTT)
      DTL(NU(3,JT)) = MIN(DTL(NU(3,JT)),DTT)
2     CONTINUE
      DO  3  IS = 1,NS
      DTL(IS) = DTL(IS) * CFL * USAIRS(IS)
3     CONTINUE
```

Les boucles 1 et 3 sont trivialement vectorielles. Les calculs de UMAX,VMAX,...DTT de la boucle 2 sont également vectoriels, bien que peu rapides car ils effectuent beaucoup plus d'appels mémoire que d'opérations arithmétiques, mais ils nécessitent un adressage indirect des tableaux UA2,...

Le calcul de DT présente une dépendance de données, mais cette récurrence est vectorisée sur la plupart des calculateurs. Enfin le report aux noeuds (calcul de DTL) n'est pas vectoriel car la fonction NU n'est pas injective et introduit des dépendances de données.

La structure du programme est analogue à celle du paragraphe précédent concernant le calcul des dérivées, et consiste, outre des boucles vectorielles sur les noeuds, en une boucle sur les éléments qui se décompose comme suit :

- des indirections pour utiliser des valeurs stockées aux noeuds
- des instructions purement vectorielles relatives aux éléments
- des reports aux noeuds non vectoriels sauf sur des groupes d'éléments non adjacents (obtenus par coloriage).

La technique de programmation est donc identique à la précédente, mais les performances sont moins bonnes du fait de la formulation des calculs (peu d'opérations arithmétiques pour beaucoup de requêtes mémoire).

IV. CALCUL DU FLUX DANS LE SCHEMA DE VAN LEER

1. Prédiction

L'étape de prédiction se formule directement de façon vectorielle sur les variables physiques stockées aux noeuds. Elle utilise en outre les dérivées et le pas de temps local. Elle effectue l'opération :

$$\tilde{W}^{n+\frac{1}{2}} = \tilde{W}^{n} - \frac{\Delta t}{2} \left(\tilde{A}(\tilde{W}^{n})\, \tilde{W}^{n}_{x} + \tilde{B}(\tilde{W}^{n})\tilde{W}^{n}_{y}\right)$$

ce qui conduit à des instructions vectorielles du type suivant :

```
      DO   1   IS = 1,NS
      ...
      CE2(IS) = UA2(IS) - 0.5 * DTL(IS) *
                (UA2(IS) * DX2(IS) + UA3(IS) * DX3(IS) + DX4(IS))
      ...
1     CONTINUE
```

La boucle contient en fait quatre instructions vectorielles, correspondant aux quatre composantes de W. Le calculateur Cray-1 ne contient pas assez de registres vectoriels pour optimiser les appels mémoire (8 registres vectoriels de 64 mots), et le compilateur génère des sauvegardes inutiles et coûteuses de variables temporaires.

Pour éviter ce problème, il est préférable de distribuer la boucle (ici en deux), de façon à optimiser l'équilibre entre requêtes mémoire et opérations arithmétiques. Chacune des boucles est mieux vectorisée, notamment grâce au chaînage des unités fonctionnelles.

Le code Fortran comporte donc 2 boucles

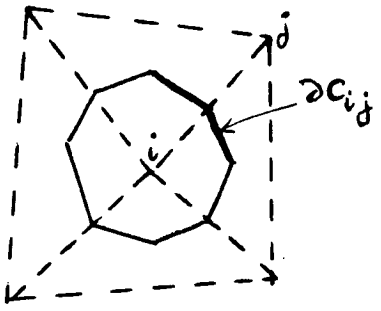
```
      DO  11  IS = 1,NS
      ...
      CE2(IS) = UA2(IS) * DX2(IS) + UA3(IS) * DX3(IS) + DX4(IS)
      ...
11    CONTINUE
      DO  12  IS = 1,NS
      ...
      CE2(IS) = UA2(IS) - 0.5 * DTL(IS) * CE2(IS)
      ...
12    CONTINUE
```

La lecture du code généré permet ce genre d'optimisation, tout en programmant en langage Fortran. Elle a été réalisée sur un Cray-1, mais reste valable pour une large gamme de calculateurs vectoriels.

2. Correction

Dans l'étape de correction, on calcule des flux sur les frontières des cellules C_i, par un Q-schéma d'ordre deux, et on accumule ces flux aux noeuds des cellules :

$$\begin{cases} W_{ij} = W_i^{n+\frac{1}{2}} + \frac{1}{2} \vec{\nabla} W_i^n \cdot \vec{ij} \\ W_{ji} = W_j^{n+\frac{1}{2}} - \frac{1}{2} \vec{\nabla} W_j^n \cdot \vec{ij} \\ \phi_{ij} = H\left(\frac{W_{ij} + W_{ji}}{2}\right) + \frac{1}{2} \left|P\left(\frac{W_{ij} + W_{ji}}{2}\right)\right| (W_{ij} - W_{ji}) \\ W_i^{n+1} = W_i^n - \frac{\Delta t}{\text{aire}(C_i)} \sum_{j=1}^{N_i} \phi_{ij} \end{cases}$$



N_i = nombre de voisins j du noeud i

∂C_{ij} : portion de frontière de la cellule C_i relative à l'arête (ij)

Figure 4. 5. Cellule C_i

L'algorithme correspondant à cette formulation est le suivant :

boucle sur les arêtes (ij)

(i) calcul des valeurs moyennes W_{ij} et W_{ji}

(ii) calcul du flux moyen $H(\frac{W_{ij} + W_{ji}}{2})$

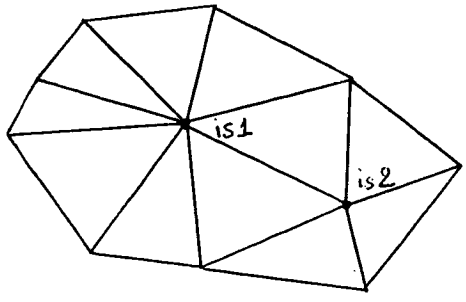
(iii) calcul de la fonction flux ϕ_{ij}

(iv) report aux deux noeuds i et j dans W_i^{n+1} et W_j^{n+1}

Remarque :

Il est nécessaire d'effectuer les calculs sur les arêtes puis de les accumuler aux noeuds. En effet, on considère des maillages non structurés. Par conséquent, le nombre d'arêtes issues d'un noeud est variable, et il n'y a pas de fonction simple faisant correspondre les numéros d'arêtes et les numéros de noeuds. (figure 4.6).

On pourrait faire une boucle sur les noeuds i, puis sur les voisins j, mais on obtiendrait une longueur de vecteurs égale au nombre de voisins (ce qui est très petit en général, même en 3.D). Contrairement aux maillages structurés, on ne peut pas intervertir les boucles en i et j, le nombre de voisins dépendant de i.



8 arêtes issues de is1

5 arêtes issues de is2

Figure 4.6. Correspondance noeuds - arêtes

(i) utilise les résultats de l'étape de prédiction et les dérivées. c'est une opération vectorielle, mais nécessitant un adressage indirect des vecteurs (passage des noeuds aux arêtes).

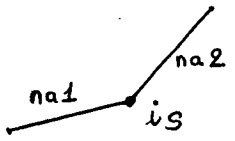
(ii) et (iii) sont des expressions vectorielles contenant un grand nombre d'opérations. Pour optimiser le code, on a distribué les calculs en plusieurs boucles, en regroupant les instructions opérant sur les mêmes variables (par exemple, séparation en quatre boucles des quatre équations définissant le flux moyen). Le but recherché est d'éviter les sauvegardes mémoire et d'augmenter le chaînage en optimisant l'utilisation des huit registres vectoriels. Cette étude a été conduite en analysant le code généré.

Toutefois, il faut stocker dans des vecteurs (au lieu de scalaires) les résultats intermédiaires. Pour ne pas saturer la place mémoire disponible, on a découpé l'ensemble des arêtes en groupes de longueur fixe, qui est donc la dimension de tous les vecteurs temporaires introduits. On a choisi une taille de 64 suffisante sur le Cray-1 pour obtenir de bonnes performances. Le coût suplémentaire dû à la gestion des indices est négligeable devant la durée des opérations.

Le code Fortran contient donc des boucles imbriquées, dont les plus internes sont vectorielles :

```
      DO  1  JT1 = 1, NT, LVECT
      JT2 = MIN(LVECT, NT - JT1 + 1)
      DO  2  JT = 1, JT2
      ...
2     CONTINUE
      DO  3  JT = 1, JT2
      ...
3     CONTINUE
      ...
1     CONTINUE
```

Enfin, l'étape (iv) n'est pas vectorielle, car il y a conflit d'écriture pour deux arêtes adjacentes (Figure 4.7).



is = nubo (1,na1) = nubo(2,na2)

Figure 4.7. La fonction NUBO n'est pas injective

Le problème - donc la solution - sont les mêmes que pour les calculs des dérivées et du pas de temps. On colorie donc le graphe $(\mathcal{A}, \mathcal{V}_{\mathcal{A}})$, où $\mathcal{A}$ est l'ensemble des arêtes, et $\mathcal{V}_{\mathcal{A}}$ l'ensemble des couples d'arêtes ayant un sommet commun. (iv) est alors vectoriel, si l'on sépare les indices d'arêtes en groupes de même couleur, et moyennant un adressage indirect des tableaux. Le nombre de couleurs d'arêtes NC, est borné par la formule suivante :

$$\max_{1\le i\le NS} d_i \le NC \le 2 * \max_{1\le i\le NS} d_i - 1$$

où d_i est le nombre de noeuds voisins du noeud i.

V. CONDITIONS LIMITES

On a vu dans le paragraphe (2.5) qu'il existe 3 types de conditions limites :

- condition à l'infini
- condition de glissement
- condition d'adhérence

seules les 2 premières ont fait l'objet d'une vectorisation conséquente.

1. Conditions à l'infini

Dans la section (2.5.1) on a montré quel genre de méthode on utilisait. Dans la pratique on prend un décentrage de type Van Leer, restreint au premier ordre.

La frontière Γ_∞ à l'infini du domaine plan considéré est en général une courbe fermée. Donc, sur la frontière Γ_∞, deux arêtes exactement sont issues de chaque noeud, tandis que ce nombre est variable pour les noeuds internes. On peut donc effectuer les calculs directement aux noeuds.

Pour chaque noeud, on doit donc calculer les contributions des deux arêtes. Les équations sont les mêmes que pour le calcul du flux. Mais le code comprend ici une boucle sur les noeuds, avec une boucle externe de longueur deux correspondant aux deux arêtes.

On a stocké pour chaque noeud, les valeurs des normales aux deux arêtes ce qui supprime toute indirection, tout en utilisant peu de mémoire : le nombre de noeuds ou d'arêtes frontières est en effet petit.

La boucle sur les noeuds est distribuée en plusieurs boucles vectorielles pour optimiser le code. On divise éventuellement le nombre de noeuds frontière en paquets de longueur fixe pour limiter la place mémoire occupée par les variables temporaires.

2. Condition de glissement

Comme pour Γ_∞ , Γ_β est un contour fermé.

Le principe du programme est le même que pour les conditions à l'infini. Pour chaque noeud, on ajoute la contribution des deux arêtes issues de ce noeud. On peut même stocker directement la somme des normales aux deux arêtes pour minimiser le nombre d'opérations.

VI. RESOLUTION DU SYSTEME LINEAIRE IMPLICITE

A chaque pas de temps du schéma implicite, il faut résoudre un système linéaire, où la matrice A est creuse non symétrique, mais à structure symétrique :

$$\left(\frac{1}{\Delta t} I - A\right) \Delta w = f(w^n) \qquad (6.1)$$

On utilise une méthode de relaxation de type Gauss-Seidel ou Jacobi.

Bien que la méthode de Gauss-Seidel converge plus rapidement que celle de Jacobi, cette dernière est mieux adaptée aux architectures vectorielles. Dans la mesure où quelques itérations de relaxation suffisent, la méthode de Jacobi peut s'avérer compétitive sur un ordinateur vectoriel. L'implémentation de ces deux méthodes repose sur l'analyse du produit d'une matrice par un vecteur.

Remarque : On se restreint dans cette étude aux relaxations par points, les relaxations par blocs semblant peu adaptées aux maillages non structurés.

6.1. Produit d'une matrice creuse par vecteur

Soit A une matrice creuse à structure symétrique, à coefficients non symétriques, d'ordre N. En pratique A est issue d'une discrétisation par éléments finis, donc la

structure est irrégulière et très creuse. En 2D,N est de l'ordre de 3000 et chaque ligne contient environ 6 éléments non nuls.

Etant donné un vecteur X d'ordre N, on calcule le vecteur Y par :

$$\begin{cases} Y_i = A_{ii} * X_i + \sum_{\substack{j \neq i \\ A_{ij} \neq 0}} A_{ij} * X_j \\ 1 \leq i \leq N \end{cases} \qquad (6.2)$$

L'indépendance des N équations, jointe aux propriétés de l'addition (associativité et commutativité), autorisent un ordre quelconque des opérations. L'algorithme et les structures de données mis en oeuvre exploitent ce parallélisme, tout en prenant en compte la structure creuse de la matrice A.

La structure de A est représentée par le graphe G_A, comprenant N noeuds, dont les arêtes sont les couples (i,j) $1 \leq i, j \leq N$, tels que A_{ij} et A_{ji} soient non nuls (la structure est symétrique). Le tableau NUBO (NA,2) contient les descripteurs des NA arêtes (numéros des deux sommets de l'arête) et suffit à définir le graphe G_A.

Les coefficients de la matrice A sont alors stockés dans deux tableaux

- le tableau D(N) contient les termes diagonaux A_{ii} ($1 \leq i \leq N$)
- le tableau A(NA,2) contient les termes extradiagonaux A_{ij} et A_{ji} pour chaque arête ia = (i,j) $1 \leq ia \leq NA$.

Le choix de la structure de donnée conditionne celui de l'algorithme qui effectue les calculs par arête de la manière suivante :

$$\begin{cases} Y_i = D_i * X_i \\ 1 \leq i \leq N \end{cases}$$

$$\begin{cases} Y_i = Y_i + A_{ia,1} * X_j \\ Y_j = Y_j + A_{ia,2} * X_i \\ ia = (i,j) \quad 1 \leq ia \leq NA \end{cases} \qquad (6.3)$$

Cette formulation induit des dépendances de données. En effet, deux arêtes adjacentes (i,j_1) et (i,j_2) contribuent au calcul du même Y_i. Pour lever ces dépendances, on définit une partition des arêtes, de façon à ce que deux arêtes d'un même sous-ensemble soient disjointes. Pour chaque sous-ensemble, les calculs sont alors indépendants et peuvent être effectués en parallèle.

La partition est réalisée par un algorithme de coloriage, qui attribue une couleur à chaque sous-ensemble d'arêtes disjointes, comme dans le §. IV.

La figure 4.8 contient le programme qui implémente l'algorithme (6.3). La première boucle est une simple multiplication de vecteurs. Il faut noter que la boucle sur les arêtes d'une même couleur n'est vectorisée que par une directive explicite de compilation. En outre, l'adressage des vecteurs est indirect et requiert un matériel adapté. La longueur des vecteurs est le nombre d'arêtes dans chaque couleur, qui n'est petit en pratique que pour les deux dernières couleurs.

```
      SUBROUTINE MATVEC (N,NA,NC,NUBO,IC,DIAG,A,X,Y)
c     produit d'une matrice creuse A par un vecteur X
c     résultat dans Y
c     paramètres
c     N : nombre de noeuds = ordre de A
c     NA : nombre d'arêtes du graphe de A
c     NC : nombre de couleurs
c     NUBO(NA,2) : tableau d'arêtes
c     ICOL(0:NC) : coloriage des arêtes
c     DIAG(N) : diagonale principale
c     A(NA,2) : coefficients extradiagonaux
c     X(N) : vecteur opérande
c     Y(N) : vecteur résultat

c     produit par DIAG
      DO 1 I=1,N
            Y(I) = DIAG(I) * X(I)
      1  CONTINUE

c     boucle sur les arêtes par couleur
      DO 2 IC = 1,NC
C DIRECTIVE : NO-RECURRENCE
      DO 3 IA = ICOL(IC-1) + 1,ICOL(IC)
            Y(NUBO(IA,1)) = Y(NUBO(IA,1)) + A(IA,1) * X(NUBO(IA,2))
            Y(NUBO(IA,2)) = Y(NUBO(IA,2)) + A(IA,2) * X(NUBO(IA,1))
3     CONTINUE
2     CONTINUE
      RETURN
      END
```

Figure 4.8. Programme MATVEC

6.2. Résolution par Jacobi

Les itérations de Jacobi s'écrivent :

$$\begin{cases} X^0 = 0 \\ X_i^{n+1} = A_{ii}^{-1} \left(F_i - \sum_{\substack{j \neq i \\ A_{ij} \neq 0}} A_{ij} * X_j^n \right) \\ 1 \leq i \leq N \end{cases} \qquad (6.4)$$

Ces équations ont les mêmes propriétés que les équations (6.2) et on peut donc y appliquer le même algorithme. Ici, le tableau D contient les inverses des termes diagonaux A_{ii} .

$$\begin{cases} X_i = D_i * F_i \\ 1 \leq i \leq N \end{cases}$$

Boucle n = 1, NITER

$$\left| \begin{array}{l} \begin{cases} Y_i = F_i \\ 1 \leq i \leq N \end{cases} \\ \\ \begin{cases} Y_i = Y_i - A_{ia,1} * X_j \\ Y_j = Y_j - A_{ia,2} * X_i \\ 1 \leq ia \leq NA \quad ia = (i,j) \end{cases} \\ \\ \begin{cases} X_i = D_i * Y_i \\ 1 \leq i \leq N \end{cases} \end{array} \right. \qquad (6.5)$$

Fin boucle

Pour chaque itération, les calculs sont vectoriels, pourvu que l'on sépare les arêtes en groupes de différentes couleurs.

6.3. Résolution par Gauss-Seidel

Les itérations de Gauss-Seidel s'écrivent :

$$\begin{cases} X^0 = 0 \\ X_i^{n+1} = A_{ii}^{-1} \left(F_i - \sum_{\substack{j > i \\ A_{ij} \neq 0}} A_{ij} * X_j^n - \sum_{\substack{j < i \\ A_{ij} \neq 0}} A_{ij} * X_j^{n+1} \right) \\ 1 \leq i \leq N \end{cases} \qquad (6.6)$$

Les équations (6.6) font intervenir la numérotation des noeuds et présentent des dépendances de données. L'équation i dépend des équations j, avec j voisin de i dans G_A et j plus petit que i.

Cette contrainte implique une autre représentation du graphe G_A. Outre le tableau d'arêtes, où les numéros de sommets sont ordonnés (ia = (i,j) i < j), on stocke pour chaque noeud i, l'ensemble des arêtes (j,i) issues de i, avec j < i. Les coefficients restent stockés dans les tableaux D(N) et A(NA,2).

L'algorithme modifie les équations (6.5) pour prendre en compte les dépendances de données.

$$\begin{cases} X_i = D_i * F_i & (7.0) \\ 1 \le i \le N \end{cases}$$

boucle n=1, NITER

$$\left|\begin{array}{ll} \begin{cases} Y_i = F_i & (7.1) \\ 1 \le i \le N \end{cases} & \\ \begin{cases} Y_i = Y_i - A_{ia,1} * X_j & (7.2) \\ 1 \le ia \le NA \qquad ia = (i,j) \end{cases} & \\ Y_i = Y_i - \sum_{ia=(j,i)} A_{ia,2} * X_j & (7.3) \\ X_i = D_i * Y_i & (7.4) \\ 1 \le i \le N & \end{array}\right. \qquad (6.7)$$

fin boucle

Les boucles (7.0), (7.1) et (7.4) sont trivialement vectorielles. La boucle (7.2) est vectorielle pour chaque couleur d'arête. La boucle (7.3) est externe, la boucle interne étant vectorielle de petite longueur (3 à 6 en 2D) et nécessitant des adressages indirects.

6.4. Comparaison Jacobi/Gauss-Seidel

La durée d'exécution de chaque méthode dépend d'une part de la vitesse de convergence, d'autre part de la vitesse d'exécution liée à l'architecture de l'ordinateur.

Le nombre d'opérations par itération est identique pour les deux méthodes, seule la vitesse d'exécution de la boucle (7.3), contenant 2*NA opérations, diffère.

Soit α l'efficacité de Gauss-Seidel par rapport à Jacobi pour la convergence ($\alpha > 1$) et τ l'efficacité de Jacobi par rapport à Gauss-Seidel pour la vitesse d'exécution de la boucle (7.3) ($\tau > 1$). L'efficacité globale de Gauss-Seidel vaut :

$$\eta = \frac{2\alpha}{1+\tau} \qquad (6.8)$$

En mode scalaire, τ vaut 1, et on retrouve $\eta = \alpha$. En mode vectoriel, selon les valeurs de α, qui dépend de la précision requise, et de τ, qui dépend de l'architecture, η peut être supérieur ou inférieur à 1.

6.5. Application au schéma implicite

Les coefficients de la matrice sont en fait des blocs 4x4, ce qui ne modifie pas l'algorithme. Dans la mesure où la phase implicite n'agit que comme un préconditionneur, seules quelques itérations de relaxation suffisent. La précision de la résolution influe sur la vitesse de convergence du schéma global vers une solution stationnaire. La formule (6.8) donne alors l'efficacité de la méthode de Gauss-Seidel pour la phase implicite, qui est la partie prédominante en temps calcul, en prenant pour α l'efficacité de convergence globale du schéma en temps.

L'efficacité globale dépend de la durée relative de la phase implicite par rapport à la phase explicite, et du nombre de relaxations dans la résolution du système linéaire.

VII. CONSTRUCTION DE LA MATRICE DU SCHEMA IMPLICITE

Le système linéaire de la phase implicite s'obtient par linéarisation d'une approximation du premier ordre, induite par la décomposition de flux. Par exemple, si on choisit une décomposition de type Vijayasundaram, on obtient :

$$\begin{cases} A_{ii} = \sum_j P^+ \left(\frac{W_i+W_j}{2}\right) \\ A_{ij} = P^- \left(\frac{W_i+W_j}{2}\right) \end{cases}$$

Les calculs sont donc analogues à ceux effectués pour la décomposition de flux. Le traitement des noeuds frontière est semblable à celui des conditions aux limites, et intervient dans le calcul des termes diagonaux A_{ii}.

Les calculs de flux sont effectués aux arêtes ia=(i,j). L'algorithme est structuré comme les autres étapes de calcul, conformément aux méthodes d'éléments finis :

- opérations de type GATHER pour déterminer les valeurs des variables aux sommets de l'arête
- opérations vectorielles pour calculer la contribution de chaque arête
- report aux sommets de l'arête. Cette opération, de type SCATTER, est vectorisée par coloriage des arêtes, en regroupant dans une même couleur des arêtes disjointes.

Il est à noter que les termes extradiagonaux A_{ij} sont stockés directement aux arêtes (i,j), ce qui ne nécessite aucun adressage indirect. Seuls les termes diagonaux A_{ii} sont accumulés aux noeuds après calcul aux arêtes.

Les opérations vectorielles sur les arêtes sont optimisées pour exploiter l'architecture de la machine. Sur CRAY, le faible nombre de registres vectoriels conduit à distribuer les opérations en plusieurs boucles, comme pour les autres étapes de calcul.

Les calculs aux frontières se font directement aux noeuds, dans la mesure où chaque noeud frontière a deux voisins exactement sur la frontière. Les techniques d'optimisation sont calquées sur le traitement des conditions aux limites.

CHAPITRE 5 : RESULTATS NUMERIQUES ET PERFORMANCES

I - PROBLEMES TESTS

Le domaine d'étude est un profil d'aile NACA0012, entouré d'un maillage contenant 1006 points (figures 5.1 et 5.2). On a aussi raffiné localement le maillage pour obtenir des résultats plus précis (figures 5.3 et 5.4) [16].

On a testé les cas suivants :

(i) Mach 0.8 à l'infini et angle d'attaque nul pour les équations d'Euler, en utilisant les schémas centré (a) et décentré (b) - (figure 5.5)

(ii) Mach 0.85 à l'infini, angle d'attaque nul et Reynolds égal à 500. pour les équations de Navier-Stokes, en utilisant le schéma centré (c) - (figure 5.6).

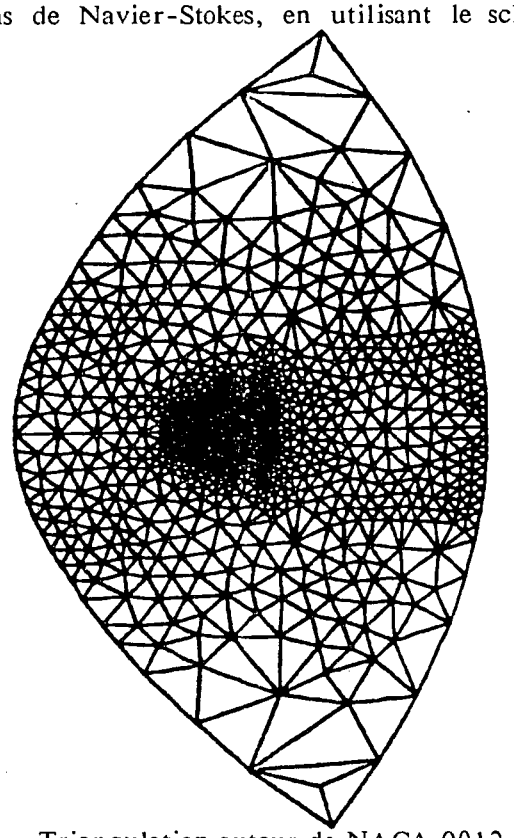


Figure 5.1. Triangulation autour de NACA 0012

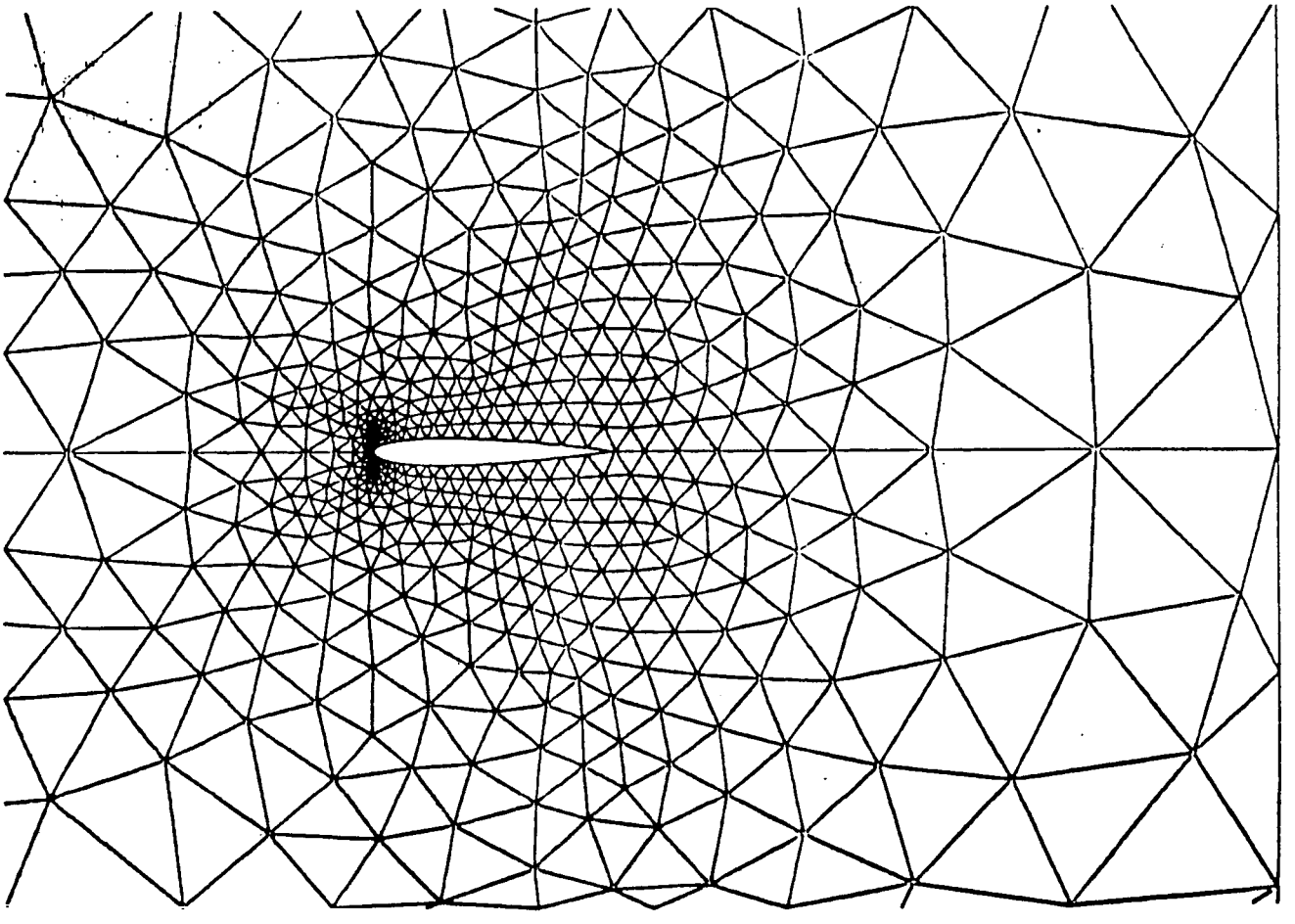


Figure 5.2. : Vue partielle du maillage

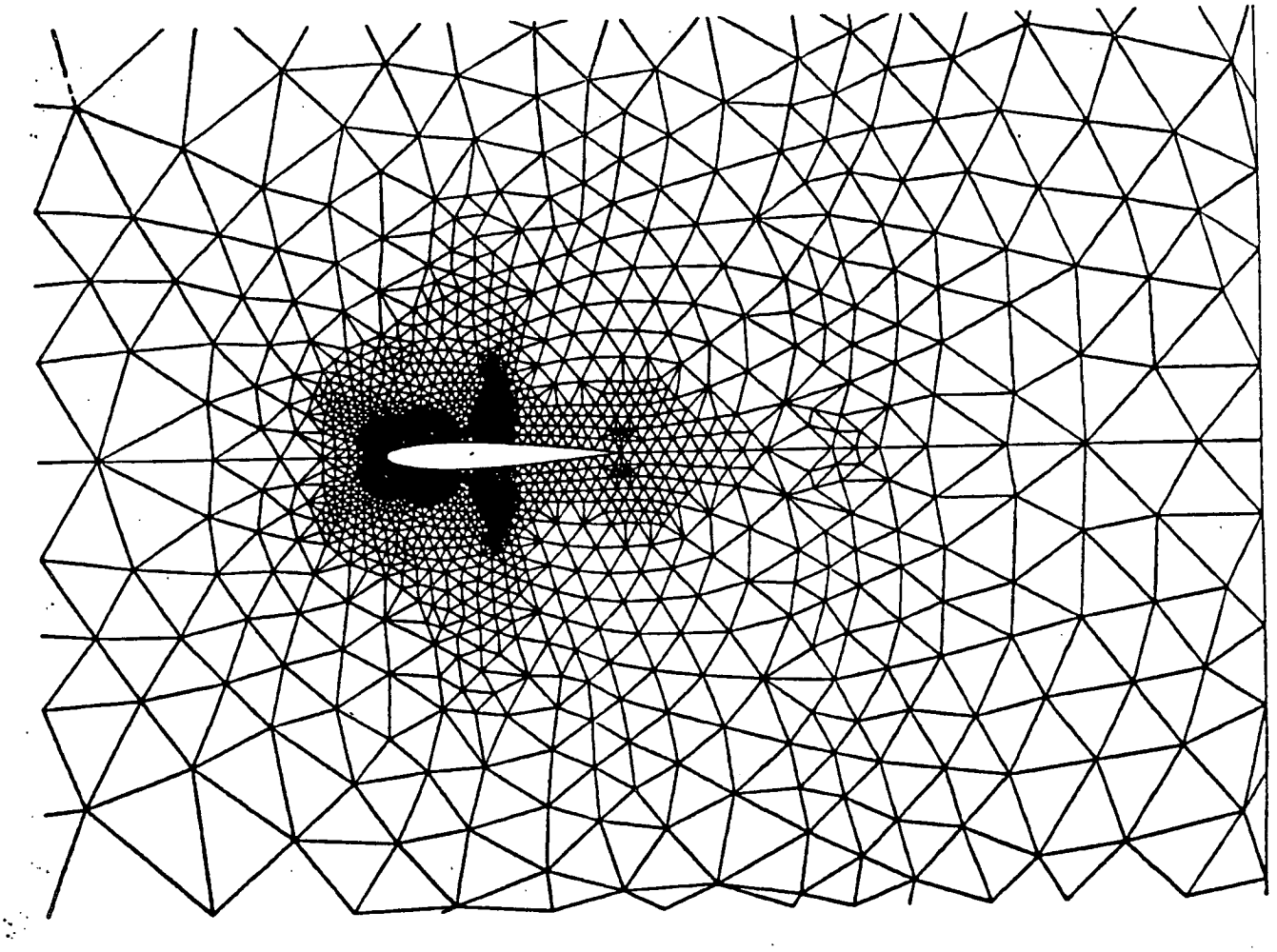


Figure 5.3. : Raffinement local pour les équations d'Euler

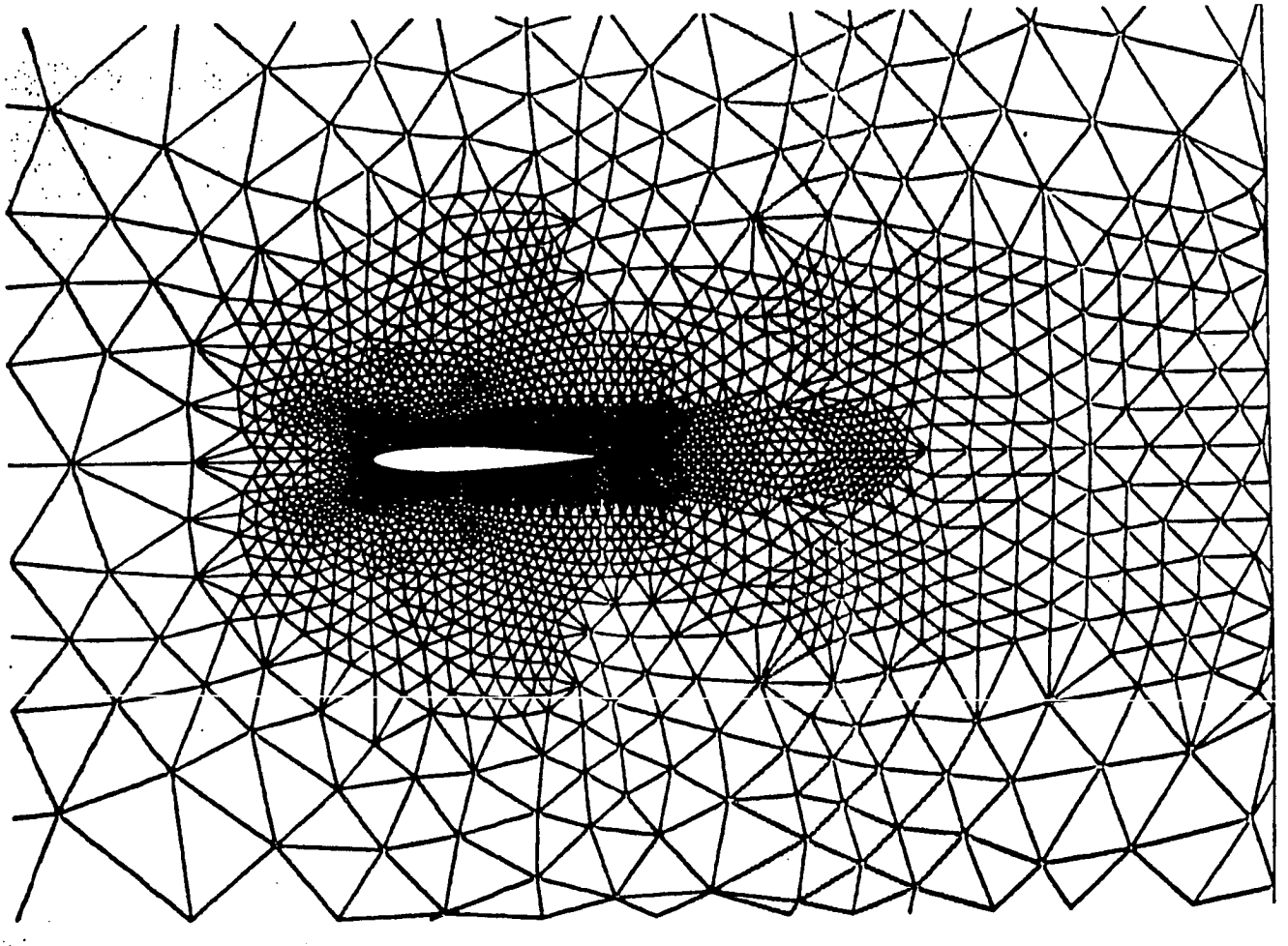


Figure 5.4. : Raffinement local pour les équations de Navier-Stokes

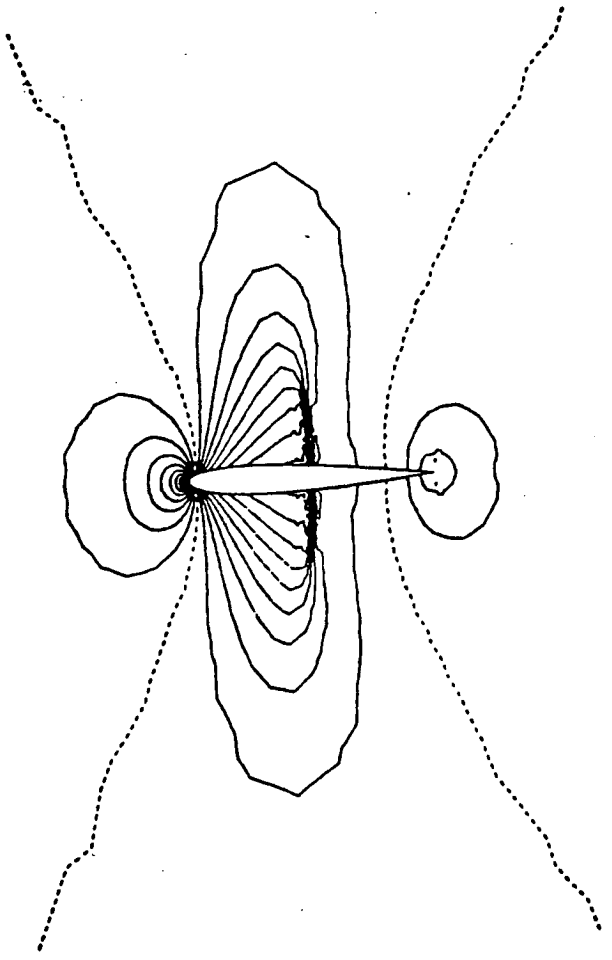


Figure 5.5. : iso-Kp pour les équations d'Euler (i)

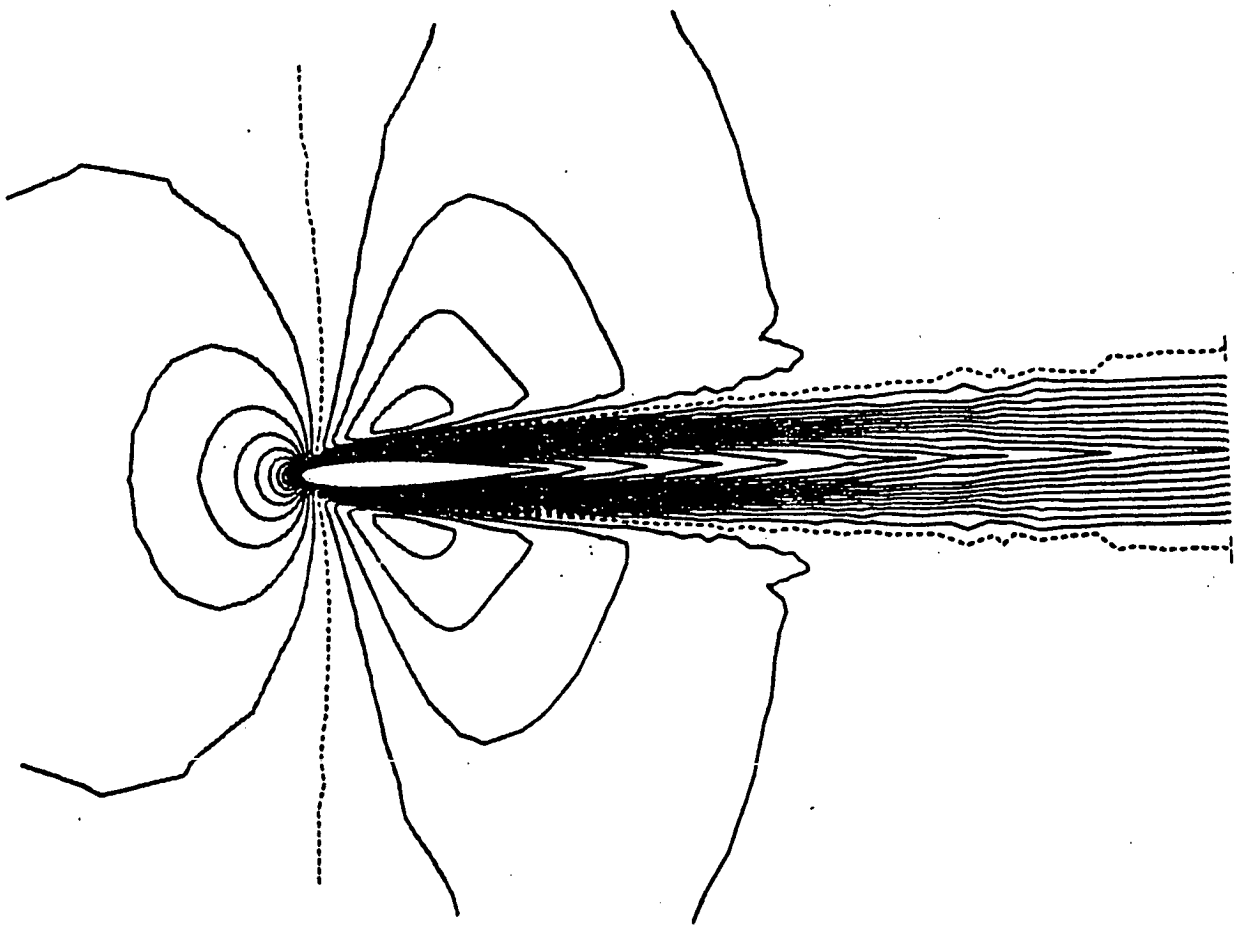


Figure 5.6. : isomach pour les équations de Navier-Stokes (ii)

En pratique, la durée d'exécution d'une seule itération et la place mémoire sont environ proportionnelles au nombre de points du maillage. C'est pourquoi les mesures de performances sont données pour la triangulation à 1006 points, et pour un seul problème-test. La vitesse d'exécution ne dépend pas du maillage considéré, mais seulement du nombre de points.

Toutefois, la vitesse de convergence dépend du schéma choisi et du problème traité. On compare ici les schémas explicite et implicite sur le problème-test et la triangulation à 1006 points.

II - PERFORMANCES DES SCHEMAS EXPLICITES

Les codes (a), (b), (c) ont été exécutés sur les ordinateurs vectoriels CRAY-1 et CRAY-XMP 48, avec respectivement les compilateurs CFT1.14 et CFT 1.15. Le CRAY-XMP 48 utilisé est une machine monoprocesseur sur laquelle les instructions avec adressage indirect (GATHER/SCATTER) sont vectorielles, de même que les instructions conditionnelles.

La figure 5.7 donne les temps et vitesses d'exécution des diverses étapes de calcul, pour les schémas (a), (b), (c). L'efficacité est le rapport entre les durées d'exécution en mode séquentiel (option OFF=V sur le compilateur CFT) et vectoriel (option ON=V).

La figure 5.8 indique le taux de vectorisation des codes en fonction de leur efficacité. L'efficacité maximale a été fixée à 14, à la fois sur CRAY-1 et CRAY-XMP.

		CRAY 1 CFT 1.14			CRAY XMP CFT 1.15		
		temps CPU (secondes)	efficacité	vitesse (M flops)	temps CPU (secondes)	efficacité	vitesse (M flops)
schéma (a)	pas de temps	0.009062	2.0	2.2	0.004669	3.2	4.3
	viscosité	0.012763	2.1	13	0.005607	3.4	29.6
	Lax Wendroff (flux + C.L.)	0.021431	2.4	24.4	0.009748	4.3	53.7
schéma (b)	pas de temps	0.019017	1.6	7.5	0.003864	5.9	36.7
	Van Leer (Flux)	0.020280	3.1	32.2	0.007866	6.3	83.1
	conditions aux limites (CL)	0.000290	4.6	51.0	0.000147	7.5	100.5
schéma (c)	Viscosité	0.022480	1.5	13.6	0.009276	2.6	33
	Lax Wendroff (flux + C.L.)	0.029513	2.3	25.4	0.015583	3.6	48.1

Figure 5.7.

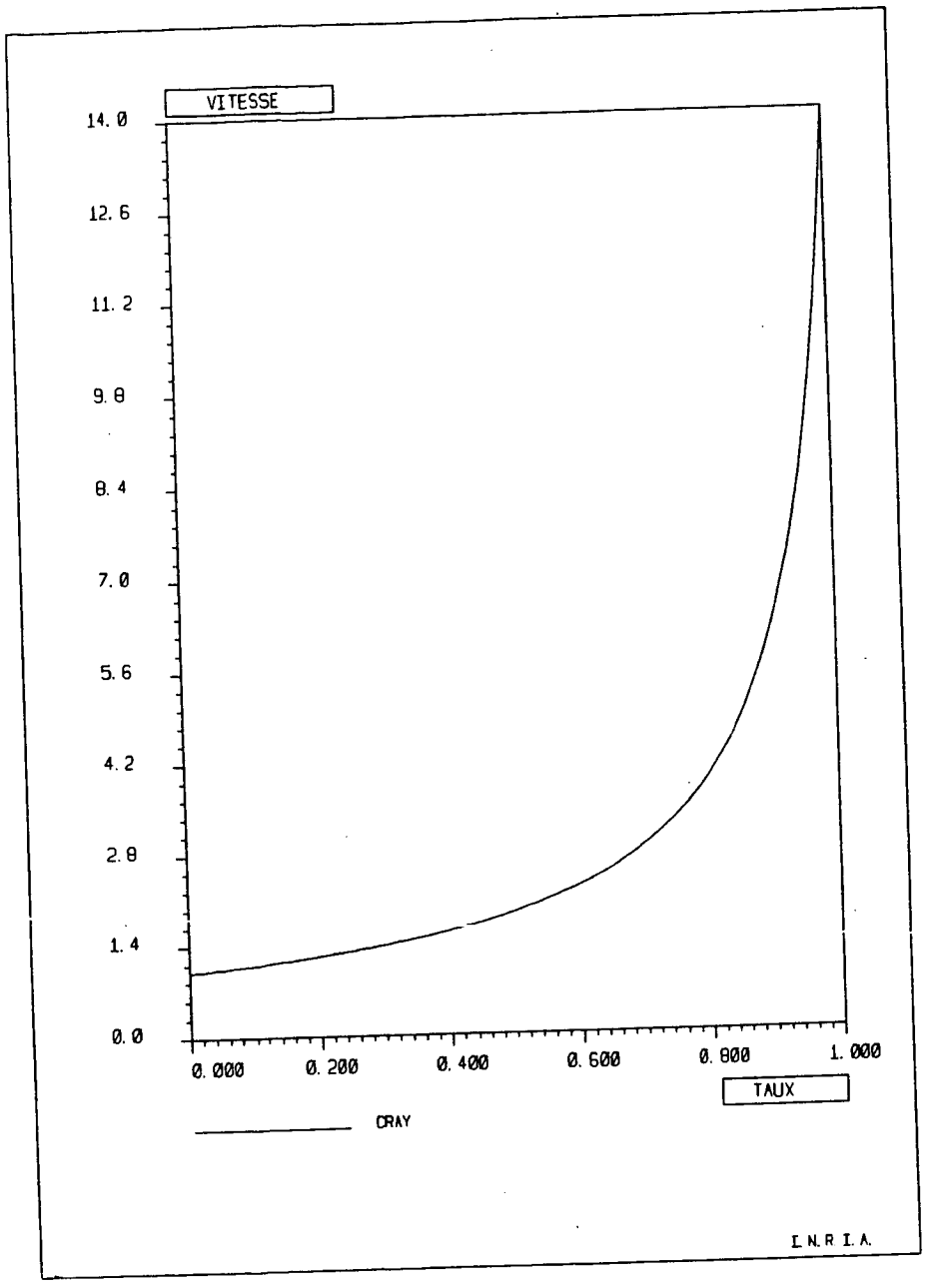

VITESSE
14. 0
12. 6
11. 2
9. 8
8. 4
7. 0
5. 6
4. 2
2. 8
1. 4
0. 0
0. 000
0. 200
0. 400
0. 600
0. 800
1. 000
TAUX
CRAY
I. N. R. I. A.

Figure 5.8

On a distribué les boucles en plusieurs boucles vectorielles de longueur 64, ce qui s'avère optimal pour le CRAY.

Sur le CRAY-XMP, on a utilisé la vectorisation de l'adressage indirect, tandis que l'on a utilisé les fonctions de bibliothèque GATHER et SCATTER du CRAY-1.

Le calcul du pas de temps nécessite beaucoup d'opérations de type MIN ou MAX et peu d'opérations arithmétiques, d'où une faible vitesse d'exécution, mais cependant une bonne efficacité par rapport au mode scalaire.

La technique de coloriage n'est mise en oeuvre que dans le schéma (b), dont les parties purement vectorielles sont également mieux optimisées localement. Les performances des schémas (a) et (c) peuvent donc être améliorées. La triangulation à 1006 points nécessite 10 couleurs pour les triangles et les arêtes, tandis que la triangulation raffinée à 2765 points requiert 11 couleurs de triangles et 10 couleurs d'arêtes.

Pour réduire le résidu de trois ordres de grandeur, avec la triangulation à 1006 points (figure 5.1), le schéma (a) consomme 16.6 secondes de temps CPU sur le CRAY-XMP, le schéma (b) 10 secondes et le schéma (c) 23.6 secondes. Le schéma (b) est légèrement plus rapide car il est mieux vectorisé. L'introduction des termes visqueux pour résoudre les équations de Navier-Stokes est d'un coût raisonnable.

III. PERFORMANCES DE SCHEMAS IMPLICITES

3.1. Choix des paramètres

Plusieurs paramètres influent sur la vitesse de convergence du schéma implicite, et sur la vitesse d'exécution :

- le choix de la méthode de résolution (Jacobi/Gauss-Seidel),
- le nombre d'itérations de relaxation,
- la fréquence des mises à jour de la matrice du système,
- le calcul du pas de temps (montée en CFL).

Les expériences actuellement réalisées calculent les coefficients du système linéaire toutes les 10 itérations. Un critère envisagé est la valeur du résidu. De même, le CFL est fixé à min (KT,200), où KT est le numéro de l'itération, mais il peut être choisi en fonction du résidu.

Le nombre d'itérations optimal, qui minimise la durée d'exécution pour une même précision (même résidu) dépend du problème traité et de l'ordinateur utilisé. Sur CRAY-1, on a mesuré un nombre optimal de quatre itérations de Jacobi et deux itérations de Gauss-Seidel symétrique (en balayant les points alternativement dans l'ordre croissant et dans l'ordre décroissant), pour le problème-test.

Cependant, la résolution de Jacobi s'est avérée plus efficace en temps CPU, bien que convergeant plus lentement.

3.2. Comparaison avec un shcéma explicite

La figure 5.9 montre les courbes de convergence des schémas explicite et implicite. Le résidu est ici normalisé et vaut RES^n/RES^o, où

$$RES^n = \frac{||\Delta W^n||_2}{\Delta t^n}$$

Le schéma explicite est le schéma décentré de type Van Leer, qui est également utilisé dans la phase explicite du schéma implicite. La phase implicite consiste en quatre itérations de Jacobi

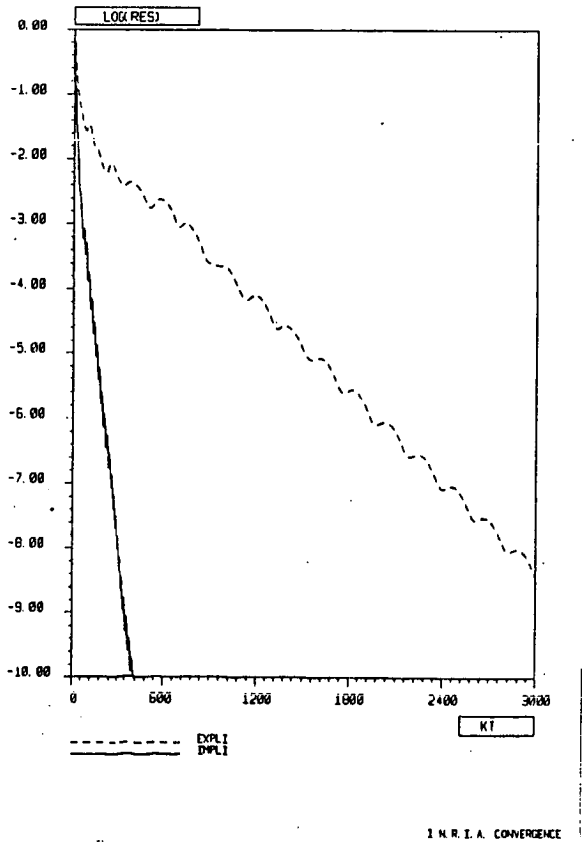


Figure 5.9

La figure 5.10 donne les performances sur CRAY-1 des deux schémas.

SCHEMA	# ITER RESIDU=10^{-3}	TEMPS CPU (secondes)	TEMPS CPU PAR ITERATION	EFFICACITE	MFLOPS
EXPLICITE	674	28	0.0396	2.4	20
IMPLICITE	57	8	0.1203	1.7	16
RESOLUTION IMPLICITE	4 par itération	3	0.0513	1.8	18

Figure 5.10

Sur CRAY-1, le schéma implicite est donc environ trois fois plus rapide que le schéma explicite. Sur CRAY-XMP, le rapport devrait être plus grand, car la phase de résolution sera plus efficace en mode vectoriel.

Il est à noter également que le schéma implicite requiert environ deux fois plus de place mémoire que le schéma explicite.

Le nombre de mots-mémoire nécessaire est évalué ci-dessous, pour un problème bidimensionnel, en fonction du nombre de noeuds NS :

explicite :	110 × NS	mots-mémoire
implicite :	240 × NS	mots-mémoire

Remerciements

L'accès au CRAY-1 a été fourni par le comité scientifique du CCVR. Les tests sur CRAY-XMP ont été effectués à EDF par Alex Azar et Daniel Charpin de la société CRAY-Recherche France.

REFERENCES

[1] Cray Research Inc.
Cray-1 Computer Systems Reference Manuals, 1980
Cray-XMP Computer Systems Reference Manuals, 1982

[2] FUJITSU Ltd.
FACOM VP : General Description, 1984

[3] ETA Systems Inc.
Introduction to the ETA System, 1986

[4] Borrel M., Montagne J.L., Néron N., Veuillot J.P., Vuillot A.M.
Implementation of 3D explicit Euler codes on a CRAY-1S vector computer.
GAMM-Workshop, Karsruhe, 14 mars 1985

[5] Jameson A., Baker T.J., Weatherill N.P.
Calculation of inviscid transonic flow over a complete aircraft.
AIAA 24th Aerospace Sciences Meeting, Reno/Nevada, January 1986

[6] Buzbee B.L.
A strategy for vectorization
Parallel Computing 3 (1986) 187-192, North holland

[7] Kogge P.M.
The architecture of Pipelined Computers
Mc Graw Hill, New York 1981

[8] Hockney R.W., Jesshope C.R.
Parallel computers
Adam Hilger, Bristol, 1981

[9] Shapiro H.D.
Theoretical limitations on the efficient use of parallel memories
IEEE Transactions on Computers, Vol. C-27, 421-428, May 1978

[10] Kuck D.J.
Automatic program restructuring for high-speed computation
CONPAR-81

[11] Lichnewsky A., Thomasset F.
Techniques de base sur l'exploitation automatique du parallélisme dans les programmes
Rapport de recherche INRIA n° 460, Décembre 1985

[12] Welsh D.J.A., Powell M.B.
An upper bound for the chromatic number of a graph and its application to time tabling problems.
Computer Journal, Vol. 10, n° 1, May 67

[13] Angrand F., Dervieux A.
Some explicit triangular finite element schemes for the Euler equations.
Int. J. for Numerical Methods in Fluids, Vol. 4, 749-764 (1984)

[14] Angrand F.
Viscous perturbation for the compressible Euler equations. Application to the numerical simulation of compressible viscous flows.
Proceedings du GAMM Workshop on Numerical Simulation of Compressible Navier-Stokes Flows, Nice (France), Décembre 1985, à paraître chez Vieweg.

[15] Fezoui F.
Résolution des équations d'Euler par un schéma de Van Leer en éléments finis.
Rapport INRIA n° 358.

[16] Palmerio B., Billey V., Dervieux A., Periaux J.
Self adaptive mesh refinements and finite element methods for solving the Euler equations
ICFD Conf. Reading (1985).

[17] Vijayasundaram G.
Résolution numérique des équations d'Euler pour des écoulements transsoniques avec un schéma de Godunov en Eléments Finis.
Thèse de 3e cycle, Paris 6, 1983.

[18] Stoufflet B.
Résolution numérique des équations d'Euler des fluides parfaits compressibles par des schémas implicites en éléments finis.
Thèse de 3e cycle, Paris 6, 1984.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique