PUBLICATIONS

MATHEMATIQUES

D'ORSAY

N° 84/04

D. E. A. D'ANALYSE NUMERIQUE

1983-1984

COURS DE FRANÇOIS THOMASSET

Université de Paris-Sud

Département de Mathématique

Bât. 425

91405 ORSAY France

PUBLICATIONS

MATHEMATIQUES

D'ORSAY

N° 84/04

D. E. A. D'ANALYSE NUMERIQUE

1983-1984

COURS DE FRANÇOIS THOMASSET

Université de Paris-Sud

Département de Mathématique

Bât. 425

91405 ORSAY France

DEA D'ANALYSE NUMERIQUE F. THOMASSET , 1983-1984

ORSAY

TABLE DES MATIERES

CHAPITRE I

LA MÉTHODE DU GRADIENT CONJUGUÉ.

LA METHODE DU GRADIENT CONJUGUE

Soit à résoudre dans $\mathbb{R}^n$

(1) $A \cdot x = b$,

avec $b \in \mathbb{R}^n$ et A matrice n x n symétrique définie positive.

Ceci est équivalent à résoudre le problème d'optimisation :

(2) Min f(y)

$y \in \mathbb{R}^n$

avec $F(y) = \frac{1}{2}(Ay, y) - (b, y)$

1/ Méthode de Richardson (ou "gradient", ou steepest descent)

$x^\circ \in \mathbb{R}^n$

$k \geq 0$:

$$M. x^{(k+1)} = M. x^{(k)} + \alpha_k (b - Ax^{(k)})$$

α_k est tel que :

$$F(x^{(k+1)}) = \text{Min } F(x^{(k)} + \alpha M^{-1} (b - Ax^{(k)}))$$

on vérifie que α_k est donné par :

(3) $$\alpha_k = \frac{(r^{(k)}, M^{-1} r^{(k)})}{(AM^{-1} r^{(k)}, M^{-1} r^{(k)})}$$

avec

$r^{(k)} = b - Ax^{(k)}$

M = matrice de préconditionnement (expliqué plus loin)

2/ Gradient conjugué : introduction - détermination des paramètres

L'itération du gradient conjugué est de la forme :

(4) $$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$$

(5) $$p^{(k)} = M^{-1} r^{(k)} + \gamma_k p^{(k-1)}$$

avec

(6) $$r^{(k)} = b - Ax^{(k)}$$

α_k ; γ_k = paramètres calculés comme suit :

On pose :

(7) $Z^{(k)} = M^{-1} r^{(k)}$

les paramètres sont déterminés de façon à vérifier

(8) $\boxed{(Z^{(i)}, M Z^{(j)}) = 0 \quad , i \neq j}$

M = matrice de préconditionnement déterminée plus loin. On la choisit symétrique définie positive et "facile" à inverser.

De (1) et (6) on déduit :

(9) $r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}$

$MZ^{(k+1)} = MZ^{(k)} - \alpha_k A p^{(k)}$

d'où avec l'hypothèse (8) ; appliquée avec i = k , j = k + 1

(10) $\boxed{\alpha_k = (M Z^{(k)} , Z^{(k)}) / (A p^{(k)} , Z^{(k)})}$

(9), (7) $\Rightarrow p^{(k)} = \frac{1}{\alpha_k} A^{-1} (M Z^{(k+1)} - M Z^{(k)})$

d'où avec (5) :

$$-\frac{1}{\alpha_k} A^{-1} (M Z^{(k+1)} - M Z^{(k)}) = Z^{(k)} - \frac{\gamma_k}{\alpha_{k-1}} A^{-1} (M Z^{(k)} - M Z^{(k-1)}$$

(11) $M Z^{(k-1)} = M Z^{(k)} - \alpha_k A Z^{(k)} + \frac{\gamma_k \alpha_k}{\alpha_{k-1}} (M Z^{(k)} - M Z^{(k-1)})$

En multipliant (11) par $Z^{(k+1)}$ et en appliquant la relation d'orthogonalité (8) on obtient :

(12) $\alpha_k = - \alpha_{k-1} (A Z^{(k)}, Z^{(k-1)}) / (M Z^{(k-1)}, Z^{(k-1)})$

En multipliant (11) par $Z^{(k+1)}$ on obtient :

$(M Z^{(k+1)}, Z^{(k+1)}) = - \alpha_k (A Z^{(k)}, Z^{(k+1)})$

ou encore (en remplaçant k par k-1)

$(M Z^{(k)}, Z^{(k)}) = - \alpha_{k-1} (A Z^{(k-1)}, Z^{(k)})$

et en reportant dans (12) :

(13) $\boxed{\gamma_k = (M Z^{(k)}, Z^{(k)}) / (M Z^{(k-1)}, Z^{(k-1)})}$

3/ L'algorithme du gradient conjugué préconditionné

Algorithme 1

$x^{(o)}$ choisi dans $\mathbb{R}^n$

$Z^{(o)} = M^{-1} \ (b - A\, x^{(o)}) \ , \ r^{(o)} = b - A\, x^{(o)}$

$p^{(o)} = Z^{(o)}$

$\alpha_o = (Z^{(o)}, M\, Z^{(o)}) \ / \ (p^{(o)}, A\, p^{(o)})$

$k \geq o$:

$x^{(k+1)} = x^{(k)} + \alpha_k\, p^{(k)}$

$r^{(k+1)} = b - Ax^{(k+1)}$ [variante : on peut aussi calculer $r^{(k+1)}$ par (9)]

$Z^{(k+1)} = M^{-1}\, r^{(k+1)}$

$\gamma_k = (Z^{(k)}, M\, Z^{(k)}) \ / \ (Z^{(k-1)}, M\, Z^{(k-1)})$

$p^{(k)} = Z^{(k)} + \gamma_k\, p^{(k-1)}$

$\alpha_k = (Z^{(k)}, M\, Z^{(k)}) \ / \ (Z^{(k)}, A\, p^{(k)})$

Test de convergence ..

Remarque :

On a montré que les relations d'orthogonalité (8) impliquent que les paramètres α_k, γ_k vérifient nécessairement (10) et (13). Il reste à prouver la réciproque.

Variante (Algorithme 1')

$$
\begin{cases}
r^{(o)} = b - A\, x^{(o)} \\
Z^{(o)} = M^{-1}\, r^{(o)} \\
p^{(o)} = Z^{(o)} \\
\beta_o = (Z^{(o)}, r^{(o)}) \\
s^{(o)} = A.\, p^{(o)} \\
x_o = \beta_o \,/\, (p^{(o)}, s^{(o)})
\end{cases}
$$

$$
\begin{cases}
k \geq o \\
x^{(k+1)} = x^{(k)} + \alpha_k\, p^{(k)} \\
r^{(k+1)} = r^{(k)} - \alpha_k\, s^{(k)} \\
Z^{(k+1)} = M^{-1}\, r^{(k+1)} \\
\beta_k = (Z^{(k)}, r^{(k)}) \\
\gamma_k = \beta_k \,/\, \beta_{k-1} \\
p^{(k)} = Z^{(k)} + \gamma_k\, p^{(k-1)} \\
S^{(k)} = A.\, p^{(k)} \\
\alpha_k = \beta_k \,/\, (p^{(k)}, s^{(k)})
\end{cases}
$$

4/ Preuve des relations d'orthogonalité

Théorème 1 : soient $Z^{(k)}$, $k \geq o$ les vecteurs définis par les algorithmes du paragraphe 3

alors, : $\forall\, k > 1$,

(14) $(Z^{(i)}, M\, Z^{(k)}) = 0$ pour $i \neq k$

Démontration :

On raisonne par récurrence :

k = 1 : i = o

$(Z^{(o)}, M\, Z^{(1)}) = o$ par construction de α_o

Hypothèse de récurrence : (14) est vrai judqu'à $k > 1$.

Il faut montrer que (14) est vrai pour $(k + 1)$.

Par l'hypothèse de récurrence :

$(Z^{(i)}, M Z^{(j)}) = o$ pour $i \neq j$, i et $j < k$

Par construction :

$(Z^{(i)}, M Z^{(k+1)}) = o$ pour $i = k$ et $k - 1$

Si $k = 2$ le résultat est démontré - sinon soit $i < k - 1$

Par application de (11) :

$$(15)\quad (M Z^{(k+1)}, Z^{(i)}) = \alpha_k (A Z^{(k)}, Z^{(i)})$$

Distinguons les cas $i > o$ et $i = o$.

Si $i > o$:

Appliquons à nouveau (11) :

$$M Z^{(i+1)} = M Z^{(i)} - \alpha_i A Z^{(i)} + \frac{\gamma_i \alpha_i}{\alpha_{i-1}} (M Z^{(i)} - M Z^{(i-1)})$$

En multipliant par $Z^{(k)}$ on obtient

$(A Z^{(i)}, Z^{(k)}) = o$

d'où le résultat grâce à (15) et à la symétrie de A.

Si $i = o$:

$M Z^{(1)} = M Z^{(o)} - \alpha_o A Z^{(o)}$

d'où le résultat en multipliant par $Z^{(k)}$ et comparaison avec (15) □

Notation : Si $a_1, a_2, .. a_m$ sont m vecteurs de $\mathbb{R}^n$, $[a_1, a_2, ..a_m]$ désigne le sous espace vectoriel engendré par ces vecteurs.

Théorème 2

$$(16)\quad (p^{(i)}, A p^{(j)}) = o \text{ si } i \neq j$$

Démonstration

Par (9) et (7) on obtient :

$A\, p^{(k)} \in [M\, Z^{(k)},\ Z^{(k+1)}]$

$p^{(i)} \in [Z^{(1)},\ p^{(i-1)}]$

d'où par récurrence :

$p^{(i)} \in [Z^{(i)},\ Z^{(i-1)},\ \ldots,\ Z^{(o)}]$

Donc

$(A\, p^{(k)},\ p^{(i)}) = o$ si $i < k$ □

Remarque :

Les directions de "descente" $p^{(k)}$ sont "conjuguées" par rapport à A, d'où l'appellation de la méthode.

Corollaire 1

(17) $$\boxed{\alpha_k = (M\, Z^{(k)},\ Z^{(k)}) \ /\ (A\, p^{(k)},\ p^{(k)})}$$

Démonstration

Par (5) :

$(A\, p^{(k)},\ Z^{(k)}) = (A\, p^{(k)},\ p^{(k)} - \gamma_k\, p^{(k-1)})$

d'où le résultat en utilisant le théorème 2, et en reportant dans l'expression de α_k (10). □

On utilisera l'expression (17) en pratique, de préférence à (10).

Remarque :

S'il existe k tel que $p^{(k)} = o$ et $p^{(k-1)} \neq o$,

alors :

$$Z^{(k)} + \gamma_k\, p^{(k-1)} = o$$

comme

$$p^{(k-1)} \in [Z^{(k-1)},\ Z^{(k-2)},\ \ldots,\ Z^{(o)}]\ ,$$

ceci n'est possible que si $Z^{(k)} = o$, autrement dit :

$r^{(k)} = b - Ax^{(k)} = o$

Donc tant que l'on n'a pas obtenu la solution les $p^{(k)}$ sont nuls et on peut poursuivre l'algorithme.

Corollaire (2)

L'algorithme du gradient conjugué converge en n itérations au plus.

Démonstration

Résulte des relations d'orthogonalité (8) □

Remarques

i/ Dans tout ce qui précède on a supposé que l'arithmétique était parfaite et qu'il n'y avait pas d'erreurs d'arrondi.
En pratique c'est faux : l'arithmétique en virgule flottante des ordinateurs introduit des erreurs d'arrondi ; si on laisse s'accumuler les erreurs d'arrondi, les relations d'orthogonalité (8) et (16) ne sont plus vérifiées ; la convergence n'est plus garantie. Pratiquement il suffit de tester périodiquement le produit scalaire $(M\, Z^{(k+1)}, Z^{(k)})$ et de réinitialiser l'algorithme si ce produit scalaire devient supérieur à une valeur préalablement choisie (par exemple 10^{-13} en double précision sur IBM).

ii/ La propriété du corollaire 2 n'est guère intéressante pour n grand. L'intérêt du gradient conjugué réside en fait dans ses propriétés de convergence qui seront étudiées dans les paragraphes suivants. Un choix judicieux du préconditionnement M permet d'accélérer la convergence dans des proportions spectaculaires.

5/ Pourquoi le gradient conjugué est-il une méthode itérative rapide ?

On pose $K = M^{-1} A$.

Lemme (1)

Il existe pour tout $k \geq o$ un polynôme de degré k tel que $Z^{(k+1)}$ s'exprime sous la forme :

(18) $$Z^{(k+1)} = (I - K P_k (K))\ Z^{(o)}$$

Démonstration :

par récurrence :

$$P_o(\lambda) = \alpha_o$$

$$P_1(\lambda) = \alpha_o + \alpha_1 + \gamma_1 \alpha_1 - \alpha_0 \alpha_1 \lambda$$

$$P_k(\lambda) = \alpha_k + \left(1 - \left(\alpha_k - \frac{\gamma_k \alpha_k}{\alpha_{k-1}}\right) \lambda\right) P_{k-1}(\lambda) - \frac{\gamma_k \alpha_k}{\alpha_{k-1}} \lambda \, P_{k-2}(\lambda) \qquad \square$$

On pose :

$$E(y) = (A.(x - y), x - y),$$

où x est la solution : $A x - b = 0$: $E(y)$ est donc une norme de l'erreur.

On va montrer que le polynôme P_k apparaissant dans (18) est optimal en un certain sens.

Pour démontrer cette propriété fondamentale on introduit un autre algorithme (Alg. 2) que l'on montrera équivalent à l'algorithme 1.

Algorithme (2)

$\tilde{x}^o = x^o \in \mathbb{R}^n$, $Z^{(o)} = M^{-1} (b - A x^{(o)})$

$k \geq o$:

On construit un polynôme Q_k de degré $\leq k$, tel que pour tous les polynômes R_k de degré $\leq k$, on ait :

$$E(x^{(o)} + Q_k(K) Z^{(o)}) \leq E(x^{(o)} + R_k(K) Z^{(o)}) \tag{19}$$

On pose :

$$\tilde{x}^{(k+1)} = x^{(o)} + Q_k(K) Z^{(o)} \tag{20}$$

Remarque :

$$\tilde{Z}^{(k+1)} = M^{-1} (b - A\tilde{x}^{(k+1)}) = (I - K Q_k(K)) Z^{(o)} \tag{21}$$

Théorème 3

i/ Il existe pour tout $k > o$ un polynôme Q_k unique vérifiant la relation d'orthogonalité (19)

ii/ $(M \tilde{Z}^{(i)}, \tilde{Z}^{(j)}) = o$ pour $i \neq j$

Notations :

On a : $E\,(\tilde{x}^{(k+1)}) = (M\,\tilde{Z}^{(k+1)},\ K^{-1}\,\tilde{Z}^{(k+1)})$

(utiliser les définitions de $K = M^{-1}$, A de $\tilde{Z}^{(k)}$, et la symétrie de M)

On note $\beta^{(k)}_j$ les coefficients de Q_k :

$$Q_k(\lambda) = \sum_{j=o}^{j=k} \beta^{(k)}_j\,\lambda^j$$

$$\tilde{Z}^{(k+1)} = Z^{(o)} - K \sum_{j=o}^{k} \beta^{(k)}_j\,K^j\,\tilde{Z}^{(o)}$$

On pose $u^{(j)} = K^j\,Z^{(o)}$

$$\tilde{Z}^{(k+1)} = Z^{(o)} - K \sum_{j=o}^{k} \beta^{(k)}_j\,u^{(j)}$$

Comme MK = A, on a encore :

$$(22)\quad E\,\tilde{x}^{(k+1)} = \left(M\,Z^{(o)} - \sum_{j=o}^{k} \beta^{(k)}_j\,A\,u^{(j)},\ K^{-1}\,Z^{(o)} - \sum_{\ell=o}^{k} \beta^{(k)}_\ell\,u^{(\ell)}\right)$$

Il s'agit de calculer les coefficients $\beta^{(k)}_j$: ces coefficients rendent minimale une fonctionnelle quadratique en les $\beta^{(k)}_j$; le résultat du théorème 3 sera acquis si l'on vérifie que les vecteurs $u^{(j)}$ sont linéairement indépendants.

Démonstration du théorème 3

Pour k = o, il suffit de prendre $Q_o = \alpha_o$ (cf paragraphe 1)

Pour k > o, on fait l'hypothèse de récurrence suivante :

$H\,R^{(k)}$:

i/ on peut construire les polynômes Q_j pour $j < k$ et donc également $\tilde{x}^{(j)}$, $\tilde{Z}^{(j)}$, $j < k$.

ii/ $\forall\ j < k$,

$\tilde{r}^{(j)} \neq o$ (remarque : quand $\tilde{r}^{(j)} = b - A\tilde{x}^{(j)} = o$, l'algorithme 2 s'arrête)

$\forall\ \ell \leqq j-1,\ (M\,\tilde{Z}^{(j)},\ u^{(\ell)}) = o$

$\forall\ \ell \leqq k,\ \ell \neq j,\ (\tilde{Z}^{(j)},\ M\,\tilde{Z}^{(\ell)}) = o$

Lemme (2)

$(k > o)$ H $R^{(k)} \Rightarrow$ les $u^{(j)}$ $(o < j < k)$ sont linéairement indépendants.

Démonstration du lemme (2) :

Supposons une relation de dépendance linéaire :

$$(22) \qquad \sum_{j=o}^{k} \delta_j \, u^{(j)} = o$$

Multiplions cette relation par M $Z^{(k)}$; l'hypothèse de récurrence implique que le seul terme non nul dans la sommation peut être obtenu pour j = k, d'où :

$$\delta_{(k)} \, (M \, \overset{\gamma}{Z}{}^{(k)}, \, u^{(k)}) = o$$

Or :

$$\overset{\gamma}{Z}{}^{(k)} = Z^{(o)} - K \, Q_{k-1} \, (K) \, Z^{(o)}$$

$$= Z^{(o)} - \sum_{j=o}^{k-1} \beta_j^{(k-1)} \, u^{(j+1)}$$

Appliquons encore l'hypothèse de récurrence :

$$(M \, \overset{\gamma}{Z}{}^{(k)}, \, \overset{\gamma}{Z}{}^{(k)}) = - \beta_{k-1}^{(k-1)} \, (M \, \overset{\gamma}{Z}{}^{(k)}, \, u^{(k)})$$

Comme $r^{(k)} \neq o$ (H R), $(M \, \overset{\gamma}{Z}{}^{(k)}, \, \overset{\gamma}{Z}{}^{(k)}) \neq o$,

donc $(M \, \overset{\gamma}{Z}{}^{(k)}, \, u^{(k)}) \neq o$,

d'où

$$\delta_k = o$$

Le même raisonnement permet de montrer que les autres coefficients δ_j de la relation linéaire (23) sont nuls : $\delta_j = o$, $j = o, 1 .., k$

□

Reprise de la démonstration du théorème 3

Soient $\beta^{(k)}$ et $h^{(k)}$ les vecteurs de $\mathbb{R}^{k+1}$ définis par :

$$\beta^{(k)} = \begin{bmatrix} \beta_o^{(k)} \\ \cdot \\ \cdot \\ \beta_k^{(k)} \end{bmatrix} \qquad h^{(k)} = \begin{bmatrix} h_o^{(k)} \\ \cdot \\ \cdot \\ h_k^{(k)} \end{bmatrix} \quad h_j^{(k)} = (u^{(j)}, MZ^{(o)})$$

$C^\circ = (M Z^{(o)}, K^{-1} Z^{(o)})$

Alors d'après (22) :

$$E(\tilde{x}^{(k+1)}) = (G^{(k)} \beta^{(k)}, \beta^{(k)}) - 2(\beta^{(k)}, h^{(k)}) + C^\circ$$

(produits scalaires dans $\mathbb{R}^{k+1}$),
où $G^{(k)}$ est la matrice carrée k x k de coefficients :

$G_{j\ell}^{(k)} = (Au^{(j)}, u^{(\ell)})$ (produit scalaire dans $\mathbb{R}^n$)

$G^{(k)}$ est une matrice k x k symétrique ; grâce au lemme 2, elle est définie positive.

Il existe donc un vecteur unique $\beta^{(k)}$, et donc un polynôme Q_k, minimisant $E(\tilde{x}^{(k+1)})$.

Il reste à vérifier l'hypothèse de récurrence HR (k + 1).

Le vecteur $\beta^{(k)}$ est solution du système linéaire :

$G^{(k)} \beta^{(k)} = h^{(k)}$

La jième ligne de ce système s'écrit :

$$\sum_{\ell=o}^{k} (Au^{(j)}, u^{(\ell)}) \beta_\ell^{(k)} = (u^{(j)}, M Z^{(o)})$$

soit encore grâce à la symétrie de A :

$$(u^{(j)}, M Z^{(o)} - \sum_{\ell=o}^{k} \beta_\ell^{(k)} Au^{(\ell)}) = 0$$

Finalement $\boxed{(u^{(j)}, M \tilde{Z}^{(k+1)}) = o \quad \forall\, j \leq k}$

D'autre part, par définition, pour $j \leq k$, $j > o$:

$$(M \tilde{Z}^{(k+1)}, \tilde{Z}^{(j)}) = (M \tilde{Z}^{(k+1)}, \tilde{Z}^{(o)} - \sum_{\ell=o}^{j-1} \beta_\ell^{(j-1)} \; u^{(\ell+1)})$$

$$= (M \tilde{Z}^{(k+1)}, \tilde{Z}^{(o)})$$

Mais $Z^{(o)} = u^{(o)}$, donc d'après ce qui précède,

$$\boxed{(M \tilde{Z}^{(k+1)}, \tilde{Z}^{(j)}) = o, \forall\, j \leq k}$$

Fin de démonstration du théorème 3.

L'algorithme 2 définit donc une suite unique de vecteurs $\tilde{x}^{(k)}$:

$$\tilde{x}^{(k+1)} = x^{(o)} + \sum_{j=o}^{k} \beta_j^{(k)} \; u^{(j)}$$

$$\tilde{r}^{(k+1)} = b - A \tilde{x}^{(k+1)}$$

$$= r^{(o)} - A \sum_{j=o}^{k} \beta_j^{(k)} \; u^{(j)}$$

$$\tilde{Z}^{(k+1)} = M^{-1} \tilde{r}^{(k+1)}$$

$$= Z^{(o)} - K \sum_{j=o}^{k} \beta_j^{(k)} \; u^{(j)}$$

D'autre part l'algorithme 1 (ou 1') (gradient conjugué) définit la suite de vecteurs $x^{(k)}$, que l'on peut développer de la même façon d'après le lemme 1 , en notant $\gamma_j^{(k)}$ les coefficients du polynôme P_k on obtient :

$$x^{(k+1)} = x^{(o)} + \sum_{j=o}^{k} \gamma_j^{(k)} \; u^{(j)}$$

$$r^{(k+1)} = r^{(o)} - A \sum_{j=o}^{k} \gamma_j^{(k)} \; u^{(j)}$$

$$Z^{(k+1)} = A^{(o)} - K \sum_{j=o}^{k} \gamma_j^{(k)} \; u^{(j)}$$

Il s'avère qu'il s'agit en fait des mêmes suites, comme le prouve le théorème suivant :

Théorème 4

Les algorithmes 1 et 2 définissent la même suite de vecteurs :

$\tilde{x}^{(k)} = x^{(k)} \quad \forall\, k$

Démonstration :

On raisonne par récurrence.

On suppose que les deux algorithmes partent du même vecteur :

$\tilde{x}^{(o)} = x^{(o)}$

Supposons $\tilde{x}^{(j)} = x^{(j)}$, $\forall\, j < k$.

$$K^{-1} (\tilde{Z}^{(k+1)} - Z^{(k+1)}) = - \sum_{o}^{k} (\beta_j^{(k)} - \gamma_j^{(k)})\, u^{(j)}$$

donc :

$$K^{-1} (\tilde{Z}^{(k+1)} - Z^{(k+1)}) \in [u^{(o)}, u^{(1)}, .., u^{(k)}]$$

or :

$$[Z^{(o)}, Z^{(1)}, .., Z^{(k)}] \subset [u^{(o)}, u^{(1)}, .., u^{(k)}]$$

Comme les $Z^{(j)}$ sont linéairement indépendants, ces deux sous espaces vectoriels sont égaux. . :

$$[Z^{(o)}, Z^{(1)}, .., Z^{(k)}] = [u^{(o)}, u^{(1)}, .., u^{(k)}]$$

Donc :

$$4)\quad K^{-1} (\tilde{Z}^{(k+1)} - Z^{(k+1)}) \in [Z^{(o)}, Z^{(1)}, .., Z^{(k)}]$$

D'autre part, $\forall\, j < k$:

$\tilde{Z}^{(j)} = Z^{(j)}$, $(M \tilde{Z}^{(k+1)}, \tilde{Z}^{(j)}) = o$, $(MZ^{(k+1)}, Z^{(j)}) =$

Donc : $(M (\tilde{Z}^{(k+1)} - Z^{(k+1)}), Z^{(j)}) = o$, $j \leq k$

d'où avec (24) :

$(K^{-1} (\tilde{Z}^{(k+1)} - Z^{(k+1)}, M(\tilde{Z}^{(k+1)}) - Z^{(k+1)})) = o$

Posons :

$$v = A^{-1} M(\tilde{Z}^{(k+1)} - Z^{(k+1)})$$

Sachant que $K^{-1} = A^{-1} M$, on obtient :

$$(Av, v) = o$$

d'où

$$v = o$$

d'où $\tilde{Z}^{(k+1)} = Z^{(k+1)}$

d'où le résultat

En conclusion on a le résultat suivant :

L'algorithme du gradient conjugué est, parmi les méthodes itératives de la forme :

$$x^{(k+1)} = x^{(o)} + P_k (K) Z^{(o)}$$

(avec P_k polynôme de degré k), celle qui à chaque étape minimise l'erreur $E (x^{(k+1)})$

6/ Propriétés de convergence liées au spectre de $K = M^{-1} A$

On pose :

$$e^{(k+1)} = x - x^{(k+1)} = - A^{-1} r^{(k+1)}$$

On vérifie :

$$e^{(k+1)} = (I - K P_k (K)) e^{(o)}$$

d'où :

$$\begin{aligned} (25)\ E (x^{k+1}) &= (A (x - x^{(k+1)}), x - x^{(k+1)}) \\ &= (A e^{(k+1)}, e^{(k+1)}) \\ &= (A (I - K P_k (K)) e^{(o)}, I - K P_k (K) e^{(o)}) \end{aligned}$$

M étant par hypothèse symétrique définie positive, on peut la diagonaliser : il existe une matrice V telle que :

$$M = V D V^T, \quad V^T V = I$$

où $D = \begin{pmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{pmatrix}$ = matrice diagonale

(d_i = valeurs propres de M)

On note :

$$M^{1/2} = V \begin{pmatrix} d_1^{1/2} & & 0 \\ & \ddots & \\ 0 & & d_n^{1/2} \end{pmatrix} V^T$$

On vérifie que $P_k(K)$ peut s'écrire ($K = M^{-1} A$) :

$$P_k(K) = M^{-1/2} P_k(M^{-1/2} A M^{-1/2}) M^{1/2} \tag{26}$$

D'autre part

$$K P_k(K) = P_k(K) . K .$$

On a donc :

$$M^{1/2} e^{(k+1)} = (I - P_k(M^{-1/2} A M^{-1/2}) M^{-1/2} A M^{-1/2}) M^{1/2} e^{(o)} \tag{27}$$

Soit $\lambda_1, \lambda_2, .., \lambda_n$ les valeurs propres de K.

Soit $\hat{K} = M^{-1/2} A M^{-1/2} = M^{1/2} K M^{-1/2}$:

La matrice $\hat{K}$ [1] a les mêmes valeurs propres que K ; on peut donc écrire :

$$\hat{K} = U \Lambda U^T, \quad U^T U = I, \quad \Lambda = \begin{pmatrix} \lambda_1 & & o \\ & \ddots & \\ o & & \lambda_n \end{pmatrix}$$

$$P_k(M^{-1/2} A M^{-1/2}) = P_k(\hat{K}) = U P_k(\Lambda) U^T$$

En reportant dans (27) on obtient :

$$\begin{aligned} \hat{K}^{1/2} M^{1/2} e^{(k+1)} &= \hat{K}^{1/2} M^{1/2} e^{(o)} - \hat{K}^{1/2} P_k(\hat{K}) \hat{K} M^{1/2} e^{(o)} \\ &= \hat{K}^{1/2} M^{1/2} e^{(o)} - P_k(\hat{K}) \hat{K} \hat{K}^{1/2} M^{1/2} e^{(o)} \\ &= U (I - P_k(\Lambda) \Lambda U^T K^{1/2} M^{1/2} e^{(o)} \end{aligned}$$

(28) Posons :

$$\bar{e}^{(k)} = (I - P_k(\Lambda) \Lambda) \bar{e}^{(o)}$$

D'autre part $A = M^{1/2} \hat{K} M^{1/2}$

Donc :

$$\begin{aligned} E(x^{(k+1)} &= (Ae^{(k+1)}, e^{(k+1)}) = (M^{1/2} \hat{K} M^{1/2} e^{(k+1)}, e^{(k+1)}) \\ &= (\hat{K} M^{1/2} e^{(k+1)}, M^{1/2} e^{(k+1)} \\ &= (\hat{K}^{1/2} M^{1/2} e^{(k+1)}, \hat{K}^{1/2} M^{1/2} e^{(k+1)}) \\ &= (\bar{e}^{(k+1)}, \bar{e}^{(k+1)}) \end{aligned}$$

---

(1) $\hat{K}$ est symétrique définie positive

Si $\bar{e}^{(k+1)} = (\bar{e}_1^{(k+1)}, \ldots, \bar{e}^{(k+1)})$,

$E(x^{(k+1)}) = \sum_{i=1}^{n} (\bar{e}_i^{(k+1)})^2$, $E(x^{(o)}) = \sum_i (\bar{e}_i^{(o)})^2$

D'autre part, grâce à (29), Λ étant diagonale :

$$\bar{e}_i^{(k+1)} = (1 - \lambda_i P_k(\lambda_i)) \bar{e}_i^{(o)}$$

Donc :

$$\frac{E(x^{(k+1)})}{E(x^{(o)})} \leq \max_i (1 - \lambda_i P_k(\lambda_i))^2$$

Le résultat fondamental sur la méthode du gradient conjugué est le suivant, résultant de ce qui précède et du théorème 4 :

Théorème 5

Pour tout polynôme Q_k de degré k ($k \geq o$), on a :

(30) $$\boxed{\frac{E(x^{(k+1)})}{E(x^{(o)})} \leq \max_i (1 - \lambda_i Q_k(\lambda_i))^2}$$

(λ_i = valeurs propres de $K = M^{-1}A$)

7/ Rappels sur les polynômes de Tchebycheff

On rappelle seulement les propriétés utiles pour les majorations du paragraphe suivant

Sur $]-1 + 1[$, $T_n(x)$ est défini par :

$T_n(x) = \cos(n \text{ Arc cos } x)$

Propriétés :

. T_n est un polynôme de degré n dont le coefficient principal est 2^{n-1}

. $T_o(x) = 1$

$T_1(x) = x$

$T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x)$, $n \geq 1$

. Si $|x| \leq 1$,

$$T_n(x) = \frac{1}{2}\left((x + \sqrt{x^2 - 1})^n + (x - \sqrt{x^2 - 1})^n \right)$$

. Soit $\P_n^1$ l'ensemble des polynômes de degré n valant 1 en o

Soit (a, b) un intervalle. Alors le polynôme de $\P_n^1$ qui a la plus petite norme du maximum s'exprime à l'aide du polynôme de Tchebycheff T_n :

$$\min_{Q_n \in \P_n^1} \max_{a \leq t \leq b} |Q_n(t)| = \max_{a \leq t \leq b} \left| \frac{T_n\left(\frac{a + b - 2t}{b-a}\right)}{T_n\left(\frac{a + b}{b - a}\right)} \right|$$

$$= 1 \Big/ \left| T_n\left(\frac{a + b}{b - a}\right) \right|$$

$$T_n\left(\frac{b + a}{b - a}\right) \geq \frac{1}{2}\left(\frac{\sqrt{b} + \sqrt{a}}{\sqrt{b} - \sqrt{a}}\right)^n$$

8/ Taux de convergence de la méthode du gradient conjugué

On ne connaît évidemment pas l'expression analytique du polynôme P_k réalisant le minimum dans le théorème 5.

Pour obtenir une majoration de l'erreur $E(x^{(k+1)})$ on applique le théorème 5 avec des polynômes P_k astucieusement choisis.

On suppose que λ_1 et λ_n sont les valeurs propres extrêmes de $K = M^{-1} A$:

$$\lambda_n = \min_i \lambda_i \qquad \lambda_1 = \max_i \lambda_i$$

On a alors :

$$\max_i \left(1 - \lambda_i Q_k(\lambda_i)\right)^2 \leq \max_{\lambda_n \leq \lambda \leq \lambda_1} \left(1 - \lambda Q_k(\lambda)\right)^2$$

D'après le paragraphe précédent on connaît le polynôme qui donnera la plus petite norme du maximum sur l'intervalle (λ_n, λ_1) : il suffit de prendre

$$1 - \lambda P_k(\lambda) = T_{k+1}\left(\frac{\lambda_1+\lambda_n-2\lambda}{\lambda_1-\lambda_n}\right) / T_{k+1}\left(\frac{\lambda_1+\lambda_n}{\lambda_1-\lambda_n}\right)$$

On a alors :

$$(31)\quad \boxed{\frac{E(x^{(k+1)})}{E(x^{(o)})} \leq 4\left(\frac{\sqrt{K}-1}{\sqrt{K}+1}\right)^{2(k+1)}}$$

où $K = \lambda_1/\lambda_n$ = conditionnement de K .

Ce résultat correspond à l'intuition suivante : la convergence est d'autant plus rapide que K est proche de 1.

Or si K est proche de 1, $K = M^{-1}A$ est " proche" de l'identité (à une constante multiplicative près).

Or si κ était égale à l'identité, la convergence serait obtenue en une itération.

Il faut donc choisir une matrice de préconditionnement M telle que les valeurs propres de $K = M^{-1} A$ soient le plus groupées possible.

En fait on connaît des résultats un peu plus précis ; par exemple la présence de quelques valeurs propres isolées n'altère que faiblement le taux de convergence :

Proposition :

Si

$$a \leqq \lambda_n \leqq \lambda_{n-1} \leqq \ldots \leqq \lambda_{n-m+1} \leqq b \leqq \lambda_{n-m} \leqq \ldots \leqq \lambda_1$$

alors, pour $k + 1 \geqq m =$

$$\frac{E\,(x^{(k+1)})}{E\,(x^{(o)})} \leqq 4 \left(\frac{\sqrt{b/a} - 1}{\sqrt{b/a} + 1} \right)^{2\,(k+1-m)}$$

Démonstration

Il suffit de poser :

$$1 - \lambda\, Q_{k+1}\,(\lambda) = R_{k+1}\,(\lambda) = \prod_{i=1}^{n-m} \left(1 - \frac{\lambda}{\lambda i}\right) \frac{T_{k+1-m}\left(\frac{a+b-2\lambda}{b-a}\right)}{T_{k+1-m}\,\frac{a+b}{b-a}}$$

et de remarquer que :

$$\max_{1 \leqq i \leqq n} R_{k+1}\,(\lambda_i)^2 = \max_{n-m+1 \leqq i \leqq n} R_{k+1}\,(\lambda_i)^2$$

9/ Comparaison des coûts du gradient conjugué et de la méthode de Choleski

On compare sur un cas simple une méthode itérative et une méthode directe (Choleski).

i/ Laplacien en 2 dimensions sur un carré -

discrétisation par différences finies avec I mailles par côté (schéma à 5 points)

Nombre d'inconnues : $n = O(\dot{I}^2)$

Largeur de bande : $b = O(\dot{I})$

Choleski : Factorisation : $O(\dot{I}^4)$ (nombre d'opérations)

Résolution : $O(4\ \dot{I}^3)$

Gradient conjugué :

On suppose que l'on a choisi M (matrice de préconditionnement) de la forme $M = LDL^T$, avec L triangulaire et D diagonale, où L a la même structure que A (c'est-à-dire des zéros dans les mêmes positions ; par exemple préconditionnement de Van der Vorst). Alors la résolution d'un système linéaire avec la matrice M coûte un nombre d'opérations du même ordre qu'une multiplication par A, soit $5\ I^2$.

Une itération de gradient conjugué (algorithme 1') coûte (en négligeant les opérations scalaires) :

3 combinaisons linéaires de vecteurs de $\mathbb{R}^n$

2 produits scalaires

1 résolution de système linéaire avec la matrice M

1 multiplication de vecteur par la matrice A

soit : $O(20\ \dot{I}^2)$

Le rapport gradient-conjugué/Choleski est donc :

$O\left(\frac{\text{Nombre d'itérations}}{I} \times 5\right)$ (en ne comptant que la résolution pour Choleski)

ii/ Laplacien en 3 dimensions sur un cube -

Discrétisation par différences finies avec I mailles par côté

(Schéma à 7 points)

Nombre d'inconnues : $n = O(\dot{I}^3)$

Largeur de bande : $b = O(\dot{I}^2)$

Choleski :

Factorisation : $O(\dot{I}^{7})$

Résolution : $O(4\dot{I}^{5})$

Gradient conjugué (mêmes hypothèses sur M)

Coût par itération : $o\ (24\dot{I}^{3})$

Rapport $\dfrac{\text{gradient conjugué}}{\text{Choleski (résolution}} = o\ \left(\dfrac{\text{Nombre d'itérations x 6}}{\dot{I}^{2}}\right)$

Nombre typiques d'itérations (2 D ou 3 D) : 10 à 100

La méthode du gradient conjugué réalise en outre des économies de mémoire (évite de passer sur disque).

R E F E R E N C E S

G. MEURANT & G. GOLUB

"Résolution numérique des grands systèmes linéaires",

Eyrolles, juin 1983

(La plupart des démonstrations de ce chapitre sont tirées de cet ouvrage)

M.R. HESTENES & E. STIEFEL

"Methods of conjugate gradient for solving linear systems",

NBS J. Res. 49 (1952) pp 409 - 436

J.K. REID

"On the method of conjugate gradients for the solution of large

sparse systems of equations",

pp 231-254 dans "Larges sparse sets of linear equations",

édité par J.K. Reid, Academic Press, 1971

A. JENNINGS & G.M. MALIK

"The solution of sparse linear equations by the conjugate gradient method",

Int J. Num. Meth. Engng., vol. 12, pp 141-158 (1978)

L. HAGEMAN & D.M. YOUNG

"Applied Iterative Methods", Academic Press (1971)

A. BJÖRCK & G. DAHLQUIST

"Numerical methods", Preitice Hall (1974)

CHAPITRE II

PRÉCONDITIONNEMENTS POUR LA MÉTHODE DU GRADIENT CONJUGUÉ

PRECONDITIONNEMENTS POUR LA METHODE DU GRADIENT CONJUGUE

On demande à la matrice de préconditionnement M de satisfaire deux exigences contradictoires :

- le conditionnement de $M^{-1}A$ doit être aussi proche de 1 que possible
- la résolution du système : MZ=r doit être beaucoup plus "facile" que la résolution du système d'origine Ax=b

Dans beaucoup de méthodes M est de la forme :

$$M = L.D.L^T$$

avec L triangulaire inférieure et D diagonale : il s'agit donc d'une factorisation "approchée" ou "incomplète" de la matrice A. On s'arrangera pour éviter en partie (ou en totalité) le remplissage qui survient lors de la factorisation "complète".

Sauf cas particuliers, on ne sait pas donner de bornes sur $K(M^{-1}a)$, autrement que par expérimentation numérique. Cependant, ces méthodes donnent de bons résultats sans nécessiter de connaissances préalables sur les spectres de A ou $M^{-1}A$.

1/ Préconditionnement par factorisation incomplète de Cholesky : motivation

On rappelle que la factorisation de Cholesky décompose A sous la forme :

$$A = \overline{L}\,\overline{D}\,\overline{L}^T$$

avec $\overline{D}$ = matrice diagonale,

$\overline{L}$ = matrice triangulaire inférieure, $\overline{l}_{ii} = 1$

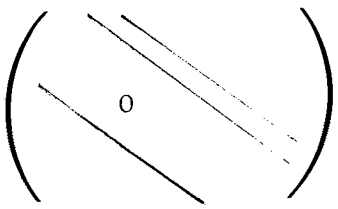
Si la matrice A est creuse, la méthode de Cholesky souffre du phénomène de remplissage : il existe beaucoup de couples i,j tels que :

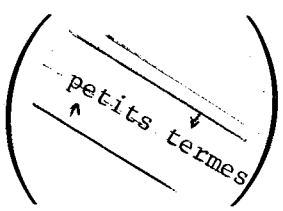
$a_{ij} = 0$ et $\overline{l}_{ij} \neq 0$

toutefois :

. le remplissage a lieu seulement à l'intérieur de la bande (ou du profil) de la matrice.

. On observe dans de nombreuses matrices de discrétisation de problèmes elliptiques que les termes $\overline{l}_{ij}$ sont de plus en plus petits à mesure que l'on s'éloigne des diagonales non nulles de A :

A = 

$\overline{L}$ = 

On peut donc s'attendre à obtenir une bonne approximation de A en négligeant les petits termes dans $\overline{L}$.

L'idée la plus simple pour atteindre cet objectif et de construire la factorisation complète à une certaine tolérance donnée ; on définit donc $L^{\varepsilon} = (\ell_{ij})$:

$$d^{\varepsilon}_{ii} = \overline{d}_{ii}$$

$$\ell^{\varepsilon}_{ij} = \begin{cases} 0 & \text{si } \overline{\ell}_{ij} < \varepsilon \min \overline{\ell}_{ij}, \overline{\ell}_{jj} \\ \overline{\ell}_{ij} & \text{sinon} \end{cases}$$

La matrice de préconditionnement du gradient conjugué est alors : $M = L^{\varepsilon}\ \overline{D}\ L^{\varepsilon T}$
La figure 1 donne le nombre d'itérations nécessaires pour la convergence (Laplacien en éléments finis) en fonction du pourcentage d'éléments de $\overline{L}$ gardés dans L^{ε} : on constate qu'un nombre assez faible d'éléments dans L suffit pour assurer une bonne convergence (5 à 10%).
Cette technique présente deux inconvénients :
i/ on doit construire au préalable la factorisation exacte $\overline{L}\ \overline{D}\ L^{T}$, qui ne tient peut-être pas en mémoire centrale pour un gros problème; il faut alors utiliser une méthode par blocs et passer par la mémoire secondaire (disques). Toutefois, ceci peut -être rentable si le système linéaire avec matrice A doit être résolu un grand nombre de fois avec des seconds membres différents.
iii/ on doit représenter en mémoire centrale les graphes de deux matrices creuses pour le gradient conjugué : A et L^{ε} ; ces deux graphes sont a priori distincts.

En revanche la technique ne repose sur aucune hypothèse supplémentaire, contrairement à la méthode de Van der Vorst exposée ci-dessous. (voir également KERSHAW, 1981, pour une autre stratégie d'application générale)

2/ Préconditionnement par Cholesky incomplet pour les H-matrices et M-matrices (Meijerink & Van der Vorst)

On se donne un ensemble G de couples i, j tels que

$$i > j :$$

$$G \subset \{1, \ldots, n\} \times \{1, \ldots, n\}$$

On définit L, D, R telles que :

$$(1)\begin{cases} A = L\ D\ L^{T} - R, \\ D = \text{matrice diagonale} > 0 \\ L = \text{matrice triangulaire inférieure, } \ell_{ii} = 1 \\ \ell_{ij} = 0 \text{ si } \{i, j\} \notin G \\ r_{ij} = 0 \text{ si } \{i, j\} \in G,\ r_{ii} = 0 \end{cases}$$

PERIAUX (1979)

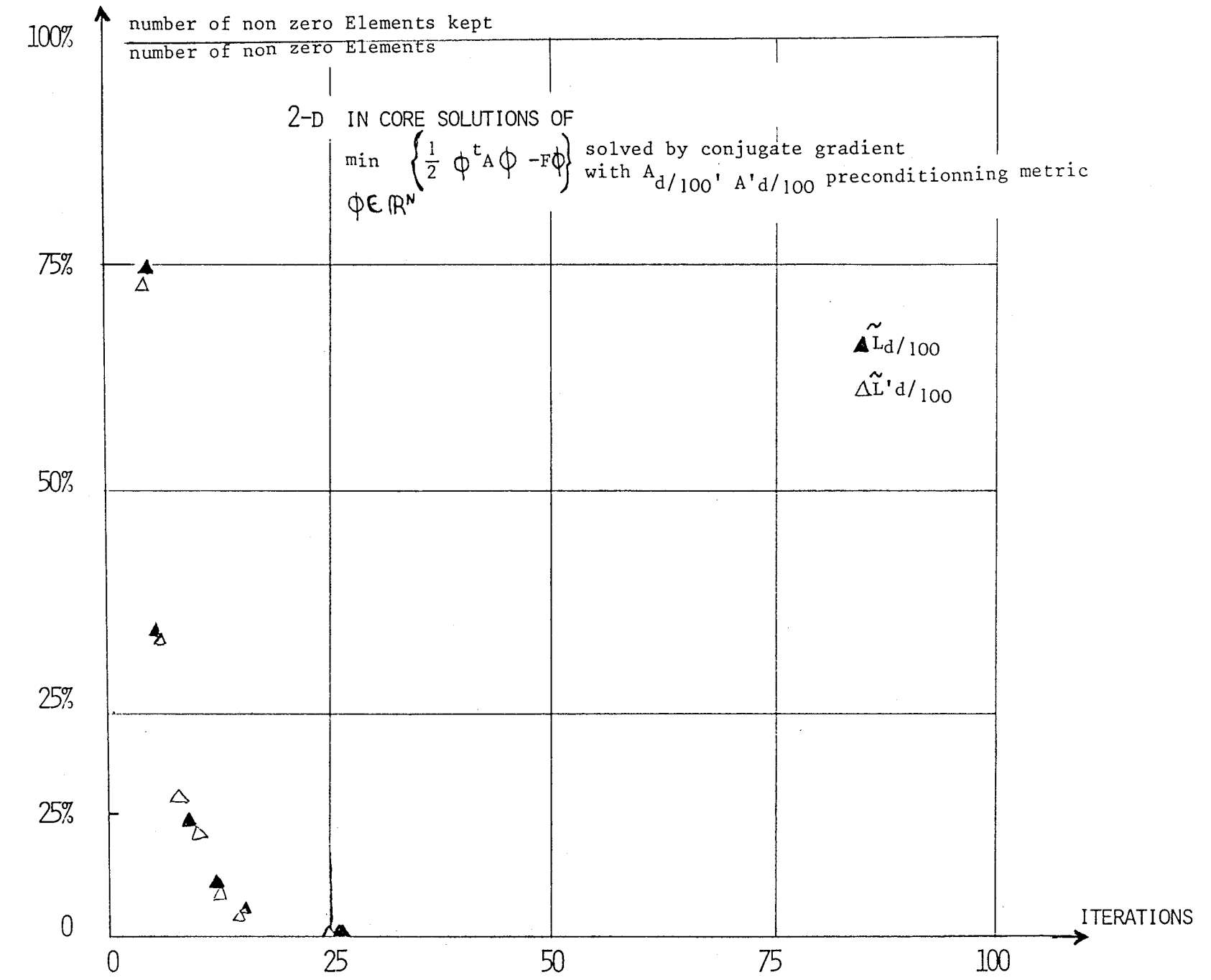


Fig. 1

PERIAUX (1979)

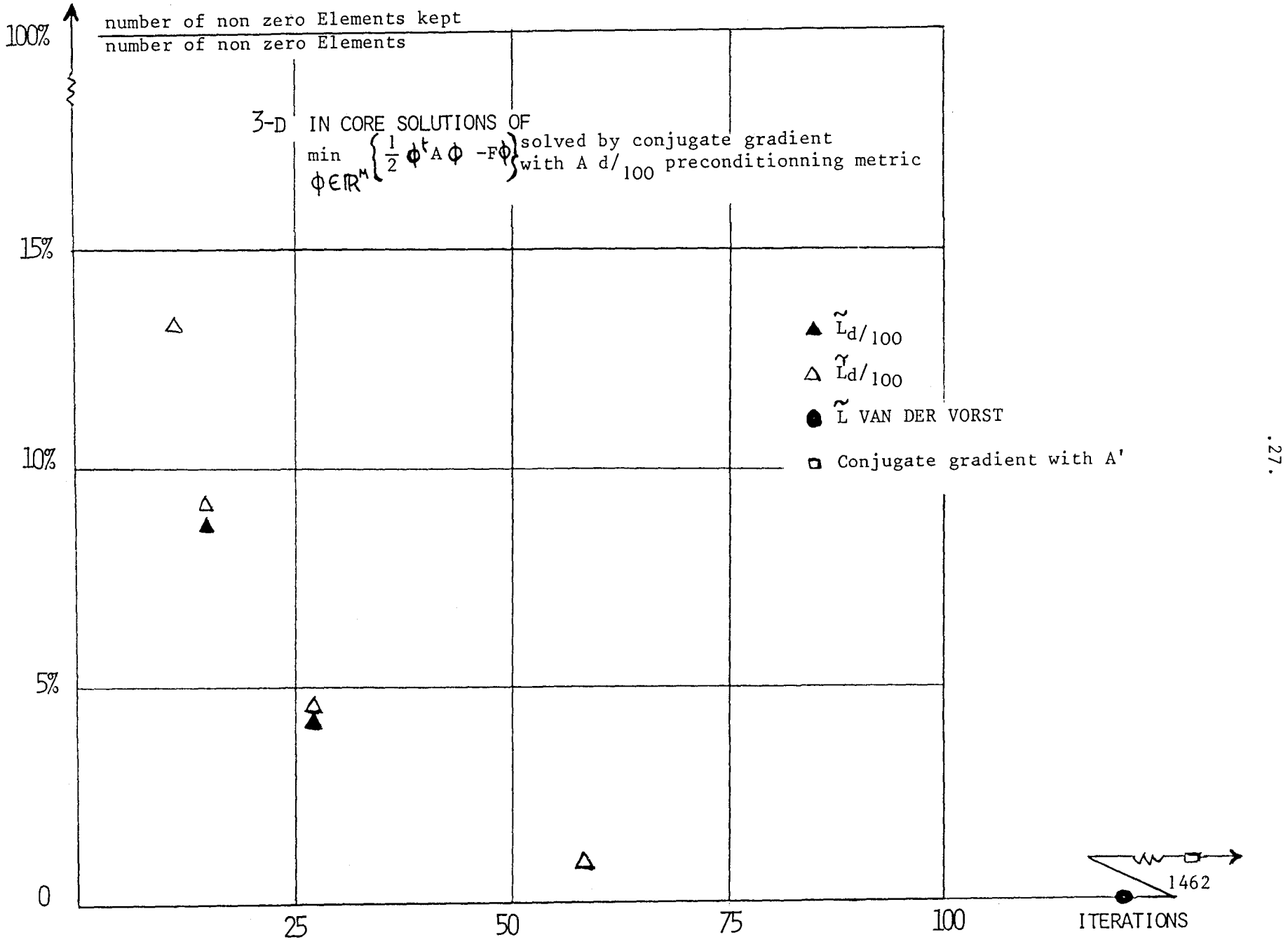


Fig. 2

Le préconditionnement M est alors :

$$\boxed{M = L\,D\,L^T}$$

On vérifiera que L et D satisfont les équations :

$(i \geqslant j)$

$$(2)\quad \begin{cases} a_{ij} = \sum\limits_{\substack{\{i,k\}\in G \\ \{j,k\}\in G \\ k \leqq j}} \ell_{ik}\, d_k\, \ell_{jk} \\ \ell_{ii} = 1, \quad \ell_{ij} = o \text{ si } i < j \end{cases}$$

Des formules (2) on déduit aisément l'algorithme pour calculer les ℓ_{ij} et d_i, analogue à la factorisation "complète".

Toutefois il n'est pas certain que la factorisation (1) existe ni que l'on puisse mener l'algorithme jusqu'au bout (il faut que les d_k soient > o).

Théorème 1

Soit A une H-matrice non singulière symétrique à diagonale positive.
Alors pour tout G la factorisation incomplète (1) existe et est déterminée par (2).

Remarques :

i/ cf annexe pour les définitions et propriétés des H-matrices

ii/ les hypothèses impliquent que A est symétrique définie positive.

Démonstration

Décomposons la factorisation en plusieurs étapes :

A la première étape, on pose :

$$A = \begin{vmatrix} a_{ii} & a_1^T \\ a_1 & B_1 \end{vmatrix} = \begin{vmatrix} a_{11} & b_1^T \\ b_1 & B_1 \end{vmatrix} - \begin{vmatrix} o & r_1^T \\ r_1 & o \end{vmatrix}$$

$$= M_1 - R_1$$

avec :

$\{j, 1\}$ $\quad G \Rightarrow b_{j1} = a_{j1}, \quad r_{j1} = o$

$\{j, 1\}$ $\quad G \Rightarrow b_{j1} = o, \quad r_{j1} = -a_{j1}$

Effectuons une étape de factorisation complète sur $M_{1\ i}$,

$$M_1 = \begin{pmatrix} 1 & 0 \\ \ell_1 & I \end{pmatrix} \begin{pmatrix} a_{11} & o \\ o & A_2 \end{pmatrix} \left(\begin{array}{c|c} 1 & \ell_1^T \\ \hline & I \end{array}\right)$$

$$= L_1 \cdot D_1 \cdot L_1^T ,$$

soit :

$$\ell_1 = b_1/a_{11} \ , \ A_2 = B_1 - \frac{1}{a_{11}} \ b_1 \ b_1^T$$

On vérifie aisément que A_2 est symétrique

Montrons que A_2 est une H-matrice non singulière à diagonale positive.

Posons $\tilde{A}_2 = B_1 - \frac{1}{a_{11}} \ a_1 \ a_1^T$

(Remarque $\tilde{A}_2$ correspond à la première étape de la factorisation complète de Cholesky) On sait déjà que $\tilde{A}_2$ est non singulière.

On montre d'abord que $\tilde{A}_2$ est une H-matrice non singulière à diagonale positive.

Par hypothèse, A est une H-matrice non singulière : il existe $e_1 \ e_2, \ \ldots, \ e_n$ tels que :

$$e_i > o$$

$$(3) \quad \sum_{j \neq i} e_j \ |a_{ij}| < e_i \ a_{ii} \qquad \forall \ i$$

Notons $\tilde{a}_{ij}$ $(i, j > 1)$ les coefficients de $\tilde{A}_2$:

$$(i > 1) \quad \tilde{a}_{ii} = a_{ii} - |a_{1i}|^2 \ / \ a_{ii}$$

$$(i, j > 1) \quad \tilde{a}_{ij} = a_{ij} - a_{1i} \ a_{1j} \ / a_{ii}$$

En multipliant par e_j et en sommant de 2 à n, on obtient :

$$(4) \quad \sum_{\substack{j>1 \\ j\neq i}} e_j \ |\tilde{a}_{ij}| = \sum_{j>1, j\neq i} e_j \ |a_{ij} - a_{1i} \ a_{1j} \ / \ a_{ii}|$$

$$\leqq \sum_{\substack{j>1 \\ j\neq i}} e_j \ |a_{ij}| + \left(\sum_{\substack{j>1 \\ j\neq i}} e_j \ |a_{1j}| \right) |a_{1i}| / \ a_{ii}$$

D'après (3) :

$$\sum_{\substack{j\neq i \\ j>1}} e_j|a_{ij}| = \sum_{\substack{j=1 \\ j\neq i}}^{n} e_j|a_{ij}| - e_1|a_{i1}| < e_i a_{ii} - e_1|a_{i2}|$$

$$\sum_{\substack{j>1 \\ j\neq i}} e_j|a_{1j}| = \sum_{j>1} e_j|a_{1j}| - e_i|a_{i1}| < e_1 a_{11} - e_i|a_{i1}|$$

D'où en reportant dans (4) :

$$\sum_{\substack{j>1 \\ j\neq i}} e_j|a_{ij}| < e_i a_{ii} - e_1|a_{i1}| + (e_1 a_{11} - e_i|a_{i1}|)\frac{|a_{i1}|}{a_{11}}$$

$$= e_i\,(a_{ii} - |a_{i1}|^2 / a_{11})$$

$$= e_i\,\tilde{a}_{ii}$$

Donc $\tilde{A}_2$ est une H-matrice non singulière à coefficients diagonaux positifs. Alors il en est de même pour A_2 par le théorème A6 (Annexe).

On peut alors recommencer la même opération sur A_2 :

$$A_2 = \begin{pmatrix} a_{22}^{(2)} & a_{2}^{(2)T} \\ a_{2}^{(2)} & B_2 \end{pmatrix} = \begin{pmatrix} a_{22}^{(2)} & b_2^T \\ b_2 & B_2 \end{pmatrix} - \begin{pmatrix} o & V_2^T \\ V_2 & o \end{pmatrix}$$

$$= M_2 - R_2$$

b_2 est obtenu à partir de $a_2^{(2)}$ en annulant les coefficients dont les indices n'appartiennent pas à G. On factorise M_2 :

$$M_2 = \begin{pmatrix} 1 & o \\ \ell_2 & I \end{pmatrix} \begin{pmatrix} a_{22}^{(2)} & o \\ o & A_3 \end{pmatrix} \begin{pmatrix} 1 & \ell_2^T \\ o & I \end{pmatrix}$$

On vérifie que l'on a :

$$A = L_1\,L_2\,D_2\,L_2^T\,L_1^T - L_1 \begin{pmatrix} o & o \\ o & R_2 \end{pmatrix} L_1^T - R_1$$

avec :

$$L_1 = \begin{pmatrix} 1 & & & o \\ & \ddots & & \\ \ell_1 & & \ddots & \\ & & & I \end{pmatrix} \qquad L_2 = \begin{pmatrix} 1 \ldots o \ldots & o \ldots \\ o \ldots 1 \ldots & o \ldots \\ o & \ddots & \ddots & \\ . & & \ell_2 & \ddots & I \end{pmatrix}$$

$$L_1 L_2 = \begin{pmatrix} 1 & o & & \\ \ell_2 & 1 & & 0 \\ & \ell_2 & \ddots & \\ & & \ddots & I \\ & & 0 & \end{pmatrix}$$

Le même raisonnement que ci-dessus montre que le processus se poursuit jusqu'à son terme : à chaque étape le bloc diagonal non encore factorisé est lui-même une H-matrice non singulière à coefficients diagonaux positifs.

Exemples :

i/ soit A la matrice de discrétisation du Laplacien avec conditions aux limites de Dirichlet par différences finies (schéma à 5 points) : A est une M-matrice (théorème A3) donc une H-matrice.

ii/ soit A la matrice de discrétisation du Laplacien avec conditions aux limites de Dirichlet par éléments finis P1 ; on suppose que tous les angles des triangles du maillage utilisé sont aigus.

Alors A est une M-matrice.

Ce résultat est vrai sous des conditions un peu plus générales : voir RUAS (1980).

Choix de G

Un choix simple consiste à prendre :

$G = \{ (i, j) = a_{ij} \neq o \}$ (ICCG(o))

Dans ce cas L possède la même structure de matrice creuse que A. On verra d'autres choix possibles plus loin.

3/ Factorisation incomplète d'une matrice définie positive

(Y. ROBERT, 1981)

Si A n'est pas une M-matrice, on ne peut pas toujours mener à bien la fac-

torisation incomplète du paragraphe précédent.

Un premier remède consiste à modifier la matrice que l'on factorise : en effet une matrice à diagonale strictement dominante est une H-matrice ; donc en chargeant suffisamment la diagonale de A on obtiendra une H-matrice. (MANTEUFFEL, 1978).

Une autre méthode (ROBERT, 1981) modifie la façon de factoriser la matrice.

On reprend les notations du paragraphe précédent : à la première étape de la factorisation, on écrit :

$$A = \begin{pmatrix} a_{11} & a_1^T \\ a_1 & B_1 \end{pmatrix}$$

Un ensemble G d'indices étant donné on définit de la même façon les vecteurs b_1 et r_1 :

$$\begin{cases} b_1 = (b_{j1}) \ , \ r_1 = (r_{j1}) : \\ b_{j1} = o \text{ et } r_{j1} = a_{j1} \quad \text{si } \{j, 1\} \notin G \\ b_{j1} = a_{j1} \text{ et } r_{j1} = o \quad \text{si } \{j, 1\} \in G \end{cases}$$

Posons :

$$r_{11} = \sum_{j=2}^{n} |r_{j1}|$$

$$r_{jj} = |r_{j1}|, \ j > 1$$

D_R = matrice diagonale d'ordre n - 1 : $\begin{pmatrix} r_{22} & & 0 \\ & \ddots & \\ 0 & & r_{nn} \end{pmatrix}$

$$R_1 = \left(\begin{array}{c|c} r_{11} & r_1^T \\ \hline r_1 & D_R \end{array}\right)$$

On a donc avec ces notations :

$$A = \left(\begin{array}{c|c} a_{11} + r_{11} & b_1^T \\ \hline b_1 & B_1 + D_R \end{array}\right) - \left(\begin{array}{c|c} r_{11} & r_1^T \\ \hline r_1 & D_R \end{array}\right)$$

$$= \qquad M_1 \qquad - \qquad R_1$$

Décomposons M_1 :

$$M_1 = \begin{pmatrix} 1 & 0 \\ \ell_1 & I \end{pmatrix} \begin{pmatrix} a_{11} + r_{11} & 0 \\ 0 & A_2 \end{pmatrix} \begin{pmatrix} 1 & \ell_1^T \\ 0 & I \end{pmatrix}$$

où :

$$\ell_1 = \frac{b_1}{a_{11} + r_{11}} \qquad A_2 = B_1 + D_R - \frac{1}{a_{11} + r_{11}} b_1 b_1^T$$

Cette étape est possible car par hypothèse A est définie positive, donc $a_{11} + r_{11} > o$.

Pour continuer il faut vérifier que A_2 est définie positive.

Lemme 1 :

R_1 est semi définie positive.

Démonstration :

R_1 est une somme de matrices de la forme :

$$\begin{pmatrix} |r| & .. & 0 & .. & r \\ & \ddots & & \ddots & \\ 0 & & \ddots & & 0 \\ & \ddots & & \ddots & \\ r & & 0 & & |r| \end{pmatrix}$$

, dont les valeurs propres sont o et $2\,|y|$

Lemme 2

A_2 est symétrique définie positive.

Démonstration

Grâce au lemme 1, $M_1 = A + R_1$ est définie positive.

soit $x \in \mathbb{R}^{n-1}$, $\alpha \in \mathbb{R}$, $y = \begin{pmatrix} \alpha \\ x \end{pmatrix}$; on a donc :

$$(M_1 y, y) > o$$

En développant cette inégalité on obtient :

$$\alpha^2 (a_{11} + r_{11}) + 2\alpha (b_1, x) + (x, B_1 + D_R)x) > o$$

Choisissons x pour que : $\alpha^2 (a_{11} + v_{11}) + \alpha (b_1, x) = o$:

$$\alpha = - \frac{(b_1, x)}{a_{11} + r_{11}}$$

Il vient :

$$-\frac{(b_1 , x)^2}{a_{11} + r_{11}} + (x, (B_1 + D_R)\, x) > 0$$

Autrement dit

$$(x, A_2\, x) > 0$$

d'où le lemme 2, car x est quelconque dans $\mathbb{R}^{n-1}$

Méthodes ICCG (n) (GUSTAFSSON , (1978) MEIJERINK & VANDERVORST (1981),

Dans le cadre des factorisations incomplètes du paragraphe 3 il est possible d'améliorer la convergence en enrichissant l'ensemble G : il est clair que plus on est proche du graphe de la factorisation complète, plus $M = LDL^T$ est proche de A.

On pose :

$$G^{(o)} = \{ \{ i, j\} : a_{ij} \neq o \}$$

soit $L^{(o)}\ D^{(o)}\ L^{(o)\,T}$

$M^{(o)}$ est différente de A et contient de nouveaux éléments non nuls ; on pose :

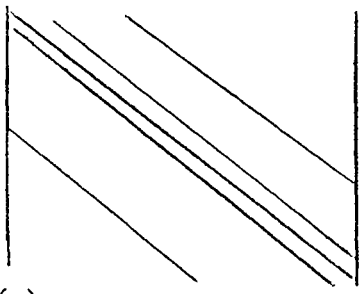
$G^{(1)} = G^{(o)} \cup \{i, j : m^{(o)}_{ij} \neq a \}$

et $M^{(1)} = L^{(1)}\ D^{(1)}\ L^{(1)T}$ est la factorisation incomplète construite à l'aide de $G^{(1)}$. Il est bien entendu possible de recommencer le processus et de construire une suite de graphes $G^{(k)}$ et de factorisations associées $M^{(k)} = L^{(k)}\ D^{(k)}\ L^{(k)T}$.

Cette construction est bien entendu possible avec une matrice de discrétisation par éléments finis (il faut représenter le graphe de $L^{(k)}$)

Dans le cas d'une matrice pentadiagonale à structure régulière (différences finies), la construction est assez simple : on peut vérifier par exemple que si A est de la forme :

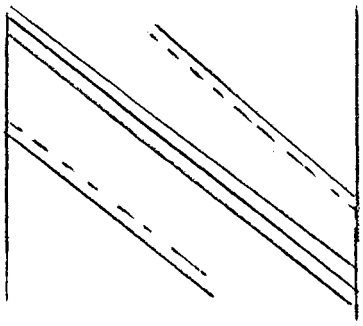
A =



alors $M^{(o)}$ contient 2 diagonales de plus :

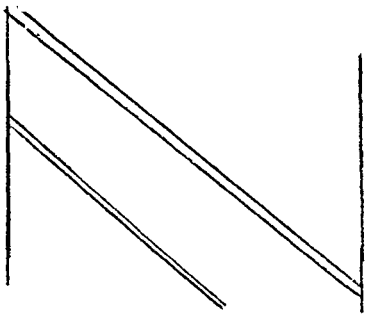
$M^{(o)} = L^{(o)} D^{(o)} L^{(o)T}$

=



et

$L^{(1)}$ =



(méthode ICC G (1))

Exercice : calculer les coefficients de $L^{(1)}$ en fonction des coefficients de A

Par contre $M^{(1)} = L^{(1)} D^{(1)} L^{(1)T}$ contient 6 diagonales de plus que A, soit 3 diagonales supplémentaires par rapport à $L^{(o)}$ pour $L^{(3)}$ (méthode ICC G(3)).

La figure 3 tirée de MEIJERINK & VANDER VORST donne une idée de la rapidité de convergence de ces méthodes.

5/ Factorisation incomplète par blocs

(CONCUS, MEURANT & GOLUB), 1983)

Soit A une M-matrice symétrique définie positive de la forme :

$A = D + L + L^T$

$$D = \begin{pmatrix} D_1 & & 0 \\ & \ddots & \\ 0 & & D_n \end{pmatrix} \qquad L = \begin{pmatrix} 0 & & \\ A_2 & \ddots & 0 \\ & \ddots & 0 \\ 0 & & A_n \end{pmatrix}$$

m_i = dimension du bloc diagonal D_i

On définit par récurrence les blocs Σ_i de même dimension m_i par :

$$\Sigma_1 = D_1$$

$$(5) \quad \Sigma_i = D_i - A_i \ \Sigma_{i-1}^{-1} \ A_i^T \quad , \ i > 1 :$$

On a alors la factorisation :

$$A = (\Sigma + L) \ \Sigma^{-1} \ (\Sigma + L^T)$$

où Σ est la matrice diagonale par blocs :

$$\Sigma = \begin{pmatrix} \Sigma_1 & & 0 \\ & \ddots & \\ 0 & & \Sigma_n \end{pmatrix}$$

$$\text{et } \Sigma + L = \begin{pmatrix} \Sigma_1 & & 0 \\ A_2 & \Sigma_2 & \\ & \ddots & \\ 0 & A_n & \Sigma_n \end{pmatrix}$$

Théorème : (CONCUS, GOLUB & MEURANT)

Les blocs Σ_i sont des M-matrices symétriques définies positives.

On notera que Σ_i^{-1} est a priori une sous matrice pleine.

Pour construire un préconditionnement on remplacera dans la formule précédente (s) Σ_{i-1}^{-1} par une sous matrice creuse Δ_{i-1} :

$$\Delta_1 = D_1$$

$$\Delta_i = D_i - A_i \Delta_{i-1} A_i^T \quad i > 1$$

(Δ_{i-1} "approche" Δ_{i-1}^{-1})

On prend alors pour préconditionnement :

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

$$\Delta = \begin{pmatrix} \Delta_1 & & 0 \\ & \ddots & \\ 0 & & \Delta_n \end{pmatrix}$$

En pratique on pourra introduire les factorisations de Cholesky des blocs Δ_i (1) :

$$\Delta_i = L_i L_i^T$$

M s'écrit alors :

$$M = \begin{pmatrix} L_1 & & & \\ W_2 & L_2 & & 0 \\ & \ddots & \ddots & \\ & 0 & W_n & L_n \end{pmatrix} \begin{pmatrix} L_1^T & W_2^T & & 0 \\ & L_2^T & \ddots & \\ & & & W_n^T \\ & 0 & & L_n \end{pmatrix}$$

où $W_i = A_i L_{i-1}^T$

Comme dans l'algorithme du gradient conjugué le préconditionnement M m'intervient que dans la résolution des systèmes linéaires MZ = v, il n'est pas nécessaire de stocker les blocs W_i.

---

(1) Il faut bien sûr s'assurer que les Δ_i sont symétriques définies positives.

CONCUS, GOLUB & MEURANT : Comparaison

Problème test : Laplacien 2500 inconnues

Table 3

Number of iterations and total work per point for $\| r^k \|_\infty / \| r^o \|_\infty < 10^{-6}$
Test problem 1, N = 2500.

M	≠ its.	work/N
I	109	1199
DIAG	109	1199
ICCG(o) : IC(1.1)	33	495
ICCG(1) : IC(1.2)	21	399
IC (1.3)	17	357
ICCG(3) : IC(2.4)	12	300
DKR	23	345
MIC(1.2)	17	323
MIC(1.3)	14	294
SSOR ω=1	40	640
SSOR ω=1.7	21	336
LJAC	80	1120
BSSOR ω=1	28	532
BSSOR ω=1.7	16	304
BDIA	22	418
POL(1.-1)	18	342
POL(0.9412,-0.4706)	21	399
POL(1.143,-1.143)	17	323
INV(1)	15	285
MINV(1)	11	209
CHOL(1)	16	304
CHOL(2)	12	252
CHOL(3)	9	225
CHOL(4)	8	232
CHOL(5)	7	231
UND(2.3)	15	255
UND(2.4)	15	255
UND(3.4)	11	231
UND(3.5)	11	231
UND(4.5)	9	225
UND(4.6)	9	225
UND(5.6)	7	203
MUND(2.3)	12	204
MUND(2.4)	10	170
MUND(2.5)	9	153
MUND(3.4)	10	210
MUND(3.5)	8	168
MUND(3.6)	8	168
MUND(4.5)	8	200
MUND(4.6)	7	175
MUND(5.6)	7	203

CONCUS, GOLUB & MEURANT : Spectre de $M^{-1}A$

Problème test : Laplacien 2500 inconnues

Table 5

Extremal eigenvalues and condition number of $M^{-1}A$.
Test problem 1. N = 2500 .

M	$\lambda min(M^{-1}A)$	$\lambda max(M^{-1}A$	$K(M^{-1}A)$
I	0.0076	7.992	1053
ICCG(o)IC(1.1)	0.0128	1.206	94.0
ICCG(1)IC(1.2)	0.033	1.179	35.6
IC(1.3)	0.049	1.131	23.2
ICCG(3)IC(2.4)	0.091	1.138	12.5
DKR	1.003	15.36	15.3
MIC(1.2)	1.003	8.83	8.3
MIC(1.3)	1.006	6.19	6.15
SSORω = 1	0.0075	1.	132.5
SSORω =1.7	0.040	1.	25.1
LJAC	0.0038	1.99	527.
BSSOR ω=1.	0.0150	1.	66.8
BSSOR ω=1.7	0.074	1.	13.5
BDIA	0.024	1.023	42.6
POL(1.-1)	0.035	1.	28.7
POL(0.9412,-0.4706)	0.027	1.002	37.2
POL(1.143,-1.143)	0.043	1.023	23.8
INV(1)	0.059	1.073	18.2
MINV(1)	1.006	4.261	4.24
CHOL(1)	0.050	1.050	20.8
CHOL(2)	0.090	1.065	11.8
CHOL(3)	0.142	1.076	7.56
CHOL(4)	0.204	1.078	5.29
CHOL(5)	0.272	1.078	3.97
UND(2,3)	0.058	1.07	18.5
UND(2,4)	0.059	1.073	18.2
UND(2.5)	0.059	1.073	18.2
UND(3.4)	0.104	1.086	10.5
UND(3.5)	0.106	1.089	10.2
UND(4.5)	0.162	1.091	6.75
UND(4.6)	0.166	1.096	6.59
UND(5.6)	0.228	1.088	4.78
MUND(2.3)	0.102	1.242	12.2
MUND(2.4)	0.202	1.564	7.74
MUND(2.5)	0.380	2.024	5.33
MUND(3.4)	0.164	1.242	7.58
MUND(3.5)	0.291	1.518	5.22
MUND(3.6)	0.483	1.887	3.91
MUND(4.5)	0.234	1.221	5.21
MUND(4.6)	0.375	1.449	3.87
MUND(5.6)	0.309	1.197	3.88

Considérons le cas particulier des discrétisations par différences finies : alors les D_i sont des sous matrices tridiagonales et les A_i des matrices diagonales ; en prenant pour Λ_{i-1} une sous matrice tridiagonale, alors Δ_i est également tridiagonale.

Le choix de Λ_{i-1} est alors ramené à l'approximation tridiagonale de l'inverse d'une matrice tridiagonale.

Le choix suivant s'avère parmi les plus performants parmi ceux étudiés par Concus, Golub & Meurant :

on prend $\Lambda_{i-1} = B(\Delta_i^{-1}, 3)$ = matrice tridiagonale formée des 3 diagonales principales de Δ_i^{-1} .

Comme Δ_i est tridiagonale on peut trouver aisément des formules explicites donnant les coefficients de la factorisée de Cholesky de Δ_i et en déduire ceux de Λ_{i-1} (Exercice !)

On démontre par récurrence que les Δ_i sont à diagonale strictement dominante.

6/ Le préconditionnement SSOR

(AXELSSON, EVANS)

Soit $A = D + L + L^T$, D diagonale ;

L dénote donc la partie triangulaire inférieure de A,

On pose :

$$(5)\quad M = \frac{1}{\omega\,(2-\omega)}\,(D + \omega L)\,D^{-1}\,(D + \omega L^T)$$

ω = paramètre, $o < \omega < 2$:

Lemmes :

. Les valeurs propres de $M^{-1}A$ sont réelles et comprises entre 0 et 1

. s'il existe μ et δ tels que :

$$\operatorname{Max}_{x\neq o} \frac{(D\,x,\,x)}{(Ax,\,x)} \leq \mu, \qquad \operatorname{Max}_{x\neq o} \frac{(L\,D^{-1}\,L^T\,x,\,x) - \frac{1}{4}(Dx,x)}{(Ax,\,x)} \leq \delta$$

alors :

i/ il existe ω_{opt} tel que, pour $\omega = \omega_{opt}$:

$$\kappa(M^{-1} A) \leq \frac{1}{2} \left(\sqrt{(\frac{1}{2} + \delta) \mu} + 1\right)$$

ii/ si $\omega = 1 = \kappa (M^{-1}A) \leq 1 + \frac{\mu}{4} + \delta$

. dans le cas de Laplacien sur un carré discrétisé par différences finies (schéma à 5 points) :

$$\mu = \frac{4}{\lambda_n} = \frac{1}{2 \sin^2(\frac{\pi}{2} h)}, \quad \delta = o$$

On en déduit que pour $\omega = 2/(1 + 2 \sin \frac{\pi}{2} h)$,

$\kappa (M^{-1}A) = O (\frac{1}{h})$, alors que $\kappa (A) = O (\frac{1}{h^2})$

(Mais on ne sait pas déterminer ω_{opt}).

7/ Approximation de A^{-1} par un polynôme en A

(JOHNSON, MICCHELLI & PAUL, DUBOIS, GREEBAUM & RODRIGUE, ADAMS)

Soit A une M-matrice non singulière, symétrique, définie positive.

Alors A peut s'écrire :

$A = D - B$,

avec D = matrice diagonale,

$B > o$, $\rho(B) < \rho(D)$,

et A^{-1} est la limite de la série :

$$A^{-1} = (I - D^{-1} B)^{-1} D^{-1}$$

$$= (I + D^{-1}B + \ldots + (D^{-1}B)^k + \ldots) D^{-1}$$

On peut donc songer à tronquer cette série pour définir le préconditionnement :

$$M^{-1} \equiv \sum_{k=o}^{\ell-1} (D^{-1} B)^k D^{-1}$$

avec ℓ faible ($\ell = 2$ ou 3)

Plus généralement, si A s'écrit :

$A = P - Q$, (P inversible)

On peut poser :

$$(6) \qquad M^{-1} = \sum_{k=o}^{\ell -1} (P^{-1} Q)^k \; p^{-1}$$

les théorèmes suivants sont démontrés dans Adams (1982)

Théorème :

Hypothèses :

A = P - Q est symétrique définie positive

P symétrique inversible

Alors :

. M définie par (6) est symétrique

. si ℓ est impair,

M définie positive $\Leftrightarrow$ P définie positive

. si ℓ est pair

M définie positive $\Leftrightarrow$ P + Q définie positive

Démonstration ii

i/ symétrie de M : exercice

ii/ ℓ impair :

soit $G = P^{-1}Q$, g une valeur propre de G, $g \neq 1$:

Il correspond à G une valeur propre de :

$R = I + G + G^2 + ... + G^{\ell - 1}$

donnée par :

$$1 + g + g^2 + ... + g^{\ell -1} = \frac{1-g^{\ell}}{1-g} > o$$

Sig = 1, la valeur propre correspondante de R est $\ell > o$

Comme P = MR et M est symétrique, on en déduit le résultat

On utilisera le lemme suivant :

Lemme Si X, Y, Z, sont des matrices, X = YZ, X symétrique définie positive, Y symétrique, Z a des valeurs propres réelles positives, alors Y est définie positive).

ii/ si ℓ est pair :

soit $G = P^{-1}Q$.

Vérifier que M^{-1} peut s'écrire :

$$M^{-1} = P^{-1} (P + Q) (I + G^2 + G^4 + \ldots G^{\ell-2}) P^{-1}$$

sachant que P est inversible et symétrique, en déduire le résultat ii/ en utilisant le lemme ci-dessus.

Théorème

Si A =(P - Q) symétrique définie positive

et P est symétrique inversible,

alors i/ $\rho(P^{-1} Q) < 1 \Leftrightarrow$ P + Q définie positive

ii/ les valeurs propres de G sont réelles.

Si en outre la plus petite valeur propre de $G = P^{-1}Q$ est $\geq o$, alors $K (M^{-1}A)$ est une fonction décroissante de m.

Exemple d'application :

P = préconditionnement SSOR (cf (5)), c'est-à-dire :

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

Plus généralement encore on peut chercher le préconditionnement sous la forme (cf Saad)

$$M = Q_m (A)^{-1}$$

avec Q_m polynôme de degré m.

On souhaiterait avoir Q_m (A) A proche de la matrice identité et donc minimiser :

$$(6) \quad \max_{\lambda_i \in S_p(A)} |1 - \lambda_i Q_m (\lambda_i)|$$

Soit (a, b) un intervalle encadrant les valeurs propres λ_i de A, on peut essayer de remplacer ce problème par :

trouver Q_m polynôme de degré m

tel que $\max_{a \leq \lambda \leq b} |1 - \lambda Q_m(\lambda)|$ soit minimum

On sait que la solution de ce problème s'exprime à l'aide de polynômes de Tchebycheff convenablement translatés et normalisés.

Malheureusement la qualité de la convergence obtenue est trés sensible à la connaissance d'un bon encadrement des valeurs propres de A, ce qui n'est pas toujours possible. En outre le polynôme obtenu ne réalise pas toujours le minimum de la quantité intéressante à savoir (6).
Il est en fait plus performant d'utiliser une autre norme pour définir Q_m.

Soit $\omega(\lambda)$ une fonction de pondération positive. On pose :

$$\|(f)\|_\omega = \int_a^b f^2(\lambda)\, \omega(\lambda)\, d\lambda)^{1/2}$$

On cherchera alors Q_m tel que $\| 1 - \lambda Q_m(\lambda) \|_\omega$ soit le plus petit possible.

Il s'agit d'un problème classique en théorie de l'approximation :
Par exemple, si $\omega(\lambda) = \sqrt{\lambda(1-\lambda)}$, on peut déterminer par récurrence les polynômes orthogonaux par rapport à $\lambda\omega(\lambda)$ et en déduire $Q_m(\lambda)$.
(cf Saad, Stiefel).

Par exemple les premiers polynômes sont, sur l'intervalle (0,1)

$Q_o(\lambda) = 3/4$

$Q_1(\lambda) = 4 - (16/5)\lambda$

$Q_2(\lambda) = \frac{2}{7}(28 - 56\lambda + 32\lambda^2)$

Pour un intervalle quelconque (a, b) les Q_m s'en déduisent par changement de variable.

En pratique on pourra prendre a = o et déterminer b par le théorème de Gerschgörin :

$$b = \max_i \left(\sum_j |a_{ij}|\right)$$

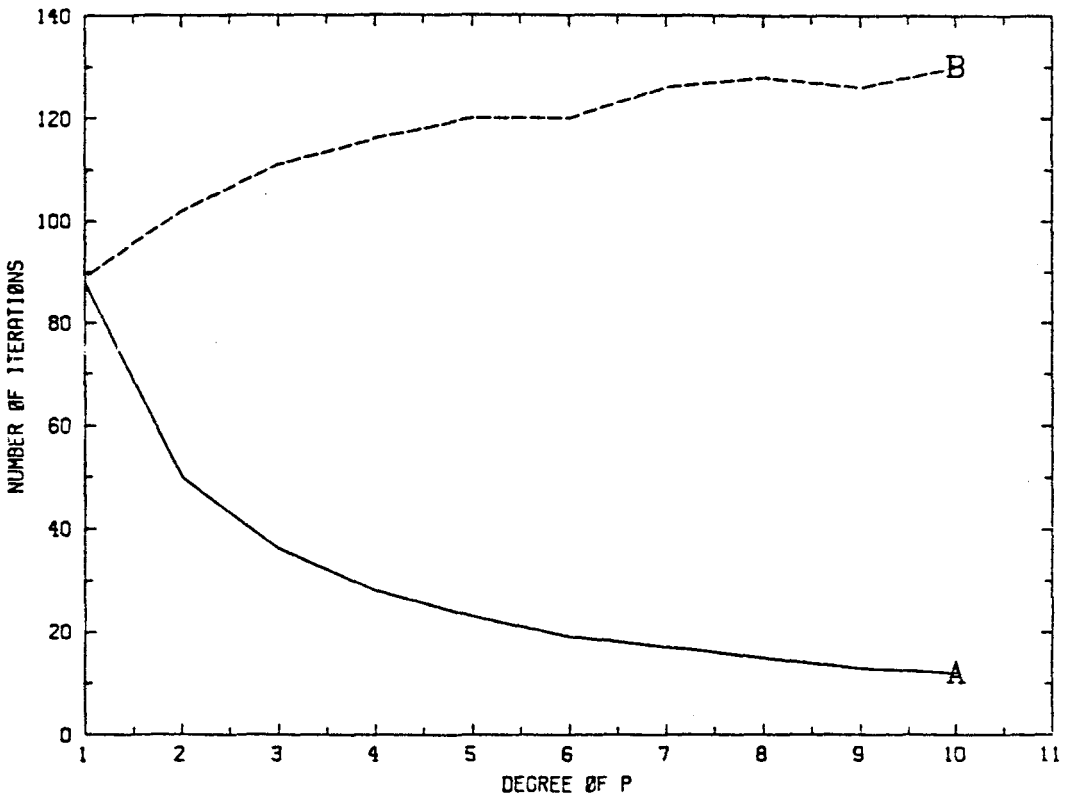
Y. SAAD

Approximation polynominale de A^{-1}

Influence du degré de Q_m

Problème test : Laplacien difference finies

40 x 30 (1200 inconnues)



A - CG ITERATIØNS ØN PRECØNDITIØNED PRØBLEM VERSUS DEGREE ØF P

B - TØTAL NUMBER ØF MATRIX-VECTØR MULTIPLIES VERSUS DEGREE ØF P

Figure 5-1: Iterations versus degree of preconditioning

ANNEXE : RAPPEL SUR LES M-MATRICES

(cf MEURANT & GOLUB, BERMANN & PLEMMONS)

Définitions

i/ Une matrice A est monotone si :

. A est inversible

. $A^{-1} \geqslant o$ (c'est-à-dire tous les coefficients de A^{-1} sont $\geqslant o$)

ii/ Une M-matrice A s'écrit :

$A = sI - B$

avec $S \in R$, $s > o$, B matrice $\geqslant o$, $\rho(B) \leqslant s$

Remarque :

Si $A = S\,I - B$ est une M-matrice,

A est non singulière $\Leftrightarrow$ $\rho(B) < s$

Théorème A1 :

Soit A une matrice non singulière

A est une M-matrice $\Leftrightarrow$ $\left| \begin{array}{l} a_{ij} \leqslant o \quad \forall\, i,j \quad i \neq j \\ \text{et } A^{-1} \geqslant o \end{array} \right.$

Déf. Une L-matrice et de la forme :

$A = sI - B$, $s > o$ $B \geqslant o$

Autrement dit :

$a_{ii} > o \quad \forall\, i$; $a_{ij} \leqslant o \quad i \neq j$

Théorème A2

i/ soit A une L-matrice à diagonale dominante irréductible alors A est une M-matrice non singulière

ii/ Même résultat si A est une L-matrice à diagonale strictement dominante.

Définition :

. A est à diagonale strictement dominante généralisée si il existe un vecteur $x \in R^n$, $x > o$, tel que :

$$\forall\, i \quad |a_{ii}|\, x_i \geqslant \sum_{j \neq i} |a_{ij}|\, x_j$$

. A est à diagonale strictement dominante si $\exists\ x > o$:

$$\forall\ i \quad |a_{ii}|\ x_i > \sum_{j \neq i} |a_{ij}|\ x_j$$

Remarques :

soit D = matrice diagonale de coefficients x_i

. A est à diagonale dominante généralisée

$\Leftrightarrow$ AD est à diagonale dominante

. A est à diagonale strictement dominante généralisée

$\Leftrightarrow$ AD est à diagonale strictement dominante

. A est diagonale dominante généralisée

$\Leftrightarrow$ D matrice diagonale $> o$;

D^{-1} AD à diagonale dominante

Théorème A3 :

A est une M-matrice non singulière

$\Leftrightarrow$ $a_{ij} \leqslant o\ \forall i \neq j$,

et A est à diagonale strictement dominante généralisée.

Démonstration :

i/ si A est à diagonale strictement dominante généralisée, D matrice diagonale $> o$, telle que AD soit à diagonale strictement dominante.

$\Rightarrow$ AD est une M-matrice par le théorème A2

$\Rightarrow$ A est une M-matrice par le théorème A1

ii/ soit A une M-matrice non singulière. par le théorème A1 :

$a_{ii} > o$, $a_{ij} \leqslant o$ pour $i \neq j$, $A^{-1} \geqslant o$

Soit $e = \begin{pmatrix} 1 \\ . \\ . \\ 1 \end{pmatrix}$, $x = A^{-1} e$: $x_i > o \quad \forall i$

Soit D matrice diagonale : $D = \begin{pmatrix} x_1 & & 0 \\ & \ddots & \\ 0 & & x_n \end{pmatrix}$: $De = x$

Alors : $A\,Se = Ax = e > o$

En déduire le résultat !

Définition

Soit A une matrice :

M (A) : matrice de coefficients m_{ij} :

$M_{ii} = |a_{ii}|$

$m_{ij} = - |a_{ij}|$, $i \neq j$

A est une H-matrice si M (A) est une M-matrice.

Caractérisations d'une H-matrice

(résulte aisément des définitions et des théorèmes ci-dessus)

. A est une H-matrice non singulière

⇔ A est à diagonale strictement dominante généralisée

(⋆) ⇔ ∃ D matrice diagonale $> o$,

D^{-1} AD à diagonale strictement dominante

. Si A est une matrice à diagonale dominante irréductible alors A est une H-matrice.

. Une M-matrice est une H-matrice.

Théorème Ao

Soit A une H-matrice et G un sous ensemble d'indices i, j (i.e. un sous-ensemble de $\{1, \ldots, n\} \times \{1, .., n\}$

Soit C la matrice définie par :

$C_{ii} : a_{ii}$ $\forall$ i

$C_{ij} = a_{ij}$ si $i \neq j$, $\{i, j\} \in G$

$C_{ij} = o$ si $i \neq j$, $\{i, j\} \notin G$

Alors C est une H-matrice.

Démonstration :

Appliquer la caractérisation (*) ci-dessus et utiliser le fait que $|a_{ij}| \geq |C_{ij}| \quad \forall i$ et j.

Références

G. MEURANT & G.H. GOLUB

"Résolution numérique des grands systèmes linéaires, Eyrolles, 1983.

Jacques PERIAUX

"Résolution de quelques problèmes non linéaires en Aérodynamique par des méthodes d'éléments finis et de moindres carrés fonctionnels",
Thèse 3e cycle, Paris 6, Juin 1979

Vitoriano RUAS

"On the strong maximum principle for some piece wise linear finite element approximate problems of non positive type",
INRIA, Rapport de Recherche n° 43, Nov. 1980

Y. ROBERT

"Regular incomplete factorizations of real Positive définite matrices",
Linear algebra and its applications, 1982

A.D. TUFF & A. JENNINGS

"An iterative method for large systems of linear structural equations",
Int. J. New. Meth. Engng., vol. 7, pp175-183 (1973)

T.A. MANTEUFFEL
"The shifted Incomplete Cholesky Factorisation",
Sandia Laboratories, Rep. SAND 78-8226, May 1978 ;
Sandia Labs, Albaquerque, New Mexico 87115, USA

D.S. KERSHAW
"The Incomplete Cholesky Factorization method for the Iterative solution of systems of linear equations", Journal of Computational Physis, 26 (1978), pp43-65.

J.A. MEIJERINK & H.A. VANDER VORST

"An Iterative solution Method for linear systems of which the coefficient Matrix is a symmetric M-matrix",
Mathematics of Computation, vol. 31, n° 1, January 1977, pp 148-162.

J.A. MEIJERINK & H.A. VAN DER VORST
"Guidelines for the usage of Imcomplete decompositions in solving sets of linear equations as they occur in Practical problems",
Journal of Computational Physics, n° 44, pp 134-155 (1981)

Ivar GUSTAFSSON

"A Class of First order Factorization methods", BIT 18 (1978), pp 142-156

O. AXELSSON & I. GUSTAFSSON

"An Iterative solver for a mixed variable variational formulation of the 1st biharmonic problems",
Comp. Meth. Appl. Mech. Engng 20 (1979) pp 9-16

O. AXELSSON

"A class of Iterative methods for finite element equations", Comp Meth. Appl. Meth. Engng 9 (1976) pp 123-137

O. AXELSSON

"A generalized SSOR method", BIT 13 (1972), pp 443-467

P. CONCUS, G.H. GOLUB & G. MEURANT

"Block preconditionning for the conjugate gradiant method", à paraître LLNL - 14866, July 1982

T. MANTEUFFEL

"An Incomplete factorization technique for positive definite matrices", Mathematics of Computation, vol 34, n° 150 (1979), pp 473-479

O.G. JOHNSON & George PAUL

"Vector Algorithms for elliptic Partial differential equations based on the Jacobi Method",
in Elliptic Problem solvers, edited by Martin Schultz,
Academic Press, 1981

A. BERMAN & Q.J. PLEMMONS

"Non negative matrices in the Mathematical Sciences,
Academic Press, 1970

P. DUBOIS, A. GREENBAUM & G. RODRIGUE
"Aporoximating the inverse of a matrix for use in Iterative Algorithms on Vector processors", Computing, vol 22, pp 257-268 (1979)

O.G. JOHNSON, C. MICCHELLI & G. PAUL

"Polynomial preconditionners for conjugate gradient calculations",
IBM Research Report, n° 40442,
IBM Thomas J. Watson Research Center, York town Heights.

Loyce ADAMS
"Iterative Algorithms for large sparse linear systems on parable computers",
Ph. D, University of Virginia, Nov 1982

D.J. EVANS

"The use of preconditionning in iterative methods for solving linear équations with symmetric positive definite matrices",
J. Inst. Maths. Applic., vol. 4, pp 295-314 (1967)

Youcef SAAD

"Practical use of polynômial preconditionnings for the conjugate gradiant method",
Research Report, Yale University,
YALEU/DCS/RR - 282, August 1983

E.L. STIEFEL

"Kernel Polynomials on Linear Algebra and their applications", US. NBS Applied Math. Series, n° 49, pp 1-24 (1958).

CHAPITRE III

CALCUL PARALLÈLE : UNE INTRODUCTION

1/ Evolution des calculateurs et des besoins scientifiques

2/ Classification des architecteures

2.1. La classification de Flynn
2.2. SISD
2.3. SIMD
2.4. Pipelines
2.5. MIMD
2.6. La classification de Kuck

CALCUL PARALLELE : UNE INTRODUCTION

1/ Evolution des calculateurs et des besoins scientifiques

Depuis l'apparition des calculateurs electroniques il y a environ 35 ans, la technologie des circuits a accompli (et continue d'accomplir) des progrès spectaculaires.
Pour prendre un exemple, le calculateur EDSAC réalisé vers 1950 à Cambridge (GB) avait les caractèristiques suivantes :

- . addition : 1,5 ms
- . multiplication : 6 ms
- . période d'horloge 2µs
- . environ 100 mots mémoire pour les données

En comparaison pour le CRAY-1 dont le premier exemplaire a été livré en 1976 :

- . la période d'horloge est de 12,5ns = $12,5 \times 10^{-9}$ S
- . pour les opérations numériques (virgule fixe ou flottante, mots de 64 bits) le CRAY-1 peut débiter jusqu'à 160 millions de résultats par seconde.
- . la mémoire est de 1 ou 2 millions de mots de 64 bits.

Ainsi : alors que les performances technologiques n'augmentaient que dans un rapport de l'ordre de 160, le nombre d'opérations arithmétiques exécutées augmentaient dans un rapport de l'ordre de 10^6 : cette différence considérable s'explique par une meilleure architecture, ou organisation, du calculateur.
L'importance croissante de l'architecture par rapport à la technologie dans les calculateurs rapides devrait s'accentuer dans les années à venir, car les progrès technologiques sont de plus en plus difficiles à obtenir.

Par ailleurs, les besoins exprimés par les utilisateurs, physiciens et ingénieurs, ne pourront sans doute pas être satisfaits par les seuls progrès technologiques dans les circuits.

---

(1) Un calculateur contient généralement une horloge envoyant des signaux à intervalles réguliers (période d'horloge, clock period); cet intervalle de temps donne une mesure de la rapidité du matériel; c'est par exemple le temps minimal qui sépare deux chargements de registres.

Ces besoins sont souvent en deçà des possibilités des calculateurs actuels les plus rapides : dans le domaine de l'aérodynamique on a évalué que la simulation d'un écoulement turbulent autour d'un avion (tridimensionnel) nécessiterait un calculateur débitant de l'ordre de 10^9 opérations flottantes par seconde (cf CHAPMAN, 1979).
Les besoins des météorologistes sont encore supérieurs.
Une caractèristique des nouveaux ordinateurs rapides est la suivante : pour obtenir le maximum de puissance le programmeur doit impérativement tenir compte de l'architecture; sinon il risque d'être fortement déçu : la puissance du CRAY-1 varie de 130 à 2 millions d'opérations par seconde selon le problème et la programmation !

Avant de décrire les différentes architectures existantes, il convient de s'en - tendre sur une mesure (grossière) des performances. On s'interesse ici aux applications numériques (équations aux dérivées partielles) où l'on doit réaliser un grand nombre d'opérations (additions, multiplications, divisions) en virgule flottante[1]. Une quantité intéressante pour l'utilisateur est donc le nombre de millions d'opérations flottantes par seconde ou megaflops (mega = 10^6, flops = F Loating point Operations Per Second)-
On espère utiliser le gigaflops vers 1990 ... A quand le Teraflops ?

Remarque :

Cette mesure n'est pas nécessairement adaptée à d'autres applications telles par exemple que le traitement des images (par exemple le MMP de Goodyear Aerospace débite 10^6 éléments d'images - "pixels" - par seconde, chaque élément d'image demandant 100 à 1000 opérations sur des informations élémentaires de longueurs variées : entiers sur 8 - ou 12 bits, flottants sur 32 bits).

2/ Classifications des architectures

2.1. La classification de Flynn

L'augmentation de puissance des calculateurs est obtenue, d'une manière générale en éxécutant en parallèle des opérations indépendantes.

---

(1) Dans ce type d'applications les calculs sur les entiers (virgule fixe) sont le plus souvent des calculs d'indices pour accéder à l'élément de tableau contenant l'information "utile" en virgule flottante.

Ce parallelisme peut être obtenu d'un grand nombre de manières différentes; la classification suivante, sommaire mais couramment utilisée, a été introduite par Flynn :

SISD : "Single Instruction, Single data" :
C'est le calculateur séquentiel classique (IBM 7090, CDC 6600)
à un instant donné l'unité de calcul éxécute une seule instruction traitant une seule donnée.

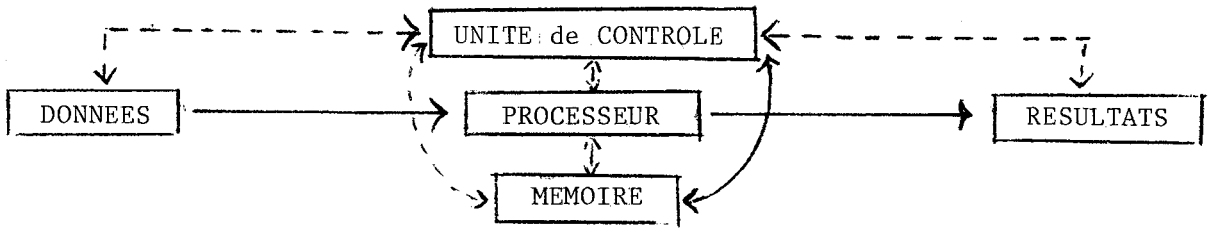
SIMD : "Single Instruction, Multiple Data" :
Le calculateur possède n unités de calcul identiques éxécutant à chaque instant la même instruction, sur des données différentes
(ILLIAC IV,ICL - DAP)

MIMD : "Multiple Instruction, multiple Data" :
Ce calculateur permet à n tâches différentes de travailler simultanément sur des données différentes, pour coopérer à un même travail (HEP-DENELCOR); ceci peut être réalisé par n processeurs travaillant chacun "à leur rythme" (Carnegie Mellon, C-mmp, CRAY-XMP)

Pipeline: Les opérations arithmétiques sont divisées en n étapes élémentaires à un instant donné l'unité de calcul est capable de contenir plusieurs instructions travaillant sur des données différentes, et rendues à des étapes différentes.
(exemple : IBM 360/91,CDC 7600;
CRAY-1, CDC - CYBER 205,
CRAY-XMP, FPS - AP120 ...)

2.2. SISD : Calculateur séquentiel classique :

La structure simplifiée est la suivante :
(---: flot de contrôle : ordre de l'unité de contrôle
(——: flot des données)



L'unité de contrôle est chargée de déclencher les mouvements de données et de lancer l'unité de calcul.
Il est pratique d'expliquer les architectures d'ordinateurs en utilisant l'image d'une chaîne de montage industrielle.

Prenons l'exemple d'une usine de mise d'eau minérale en bouteilles; sur chaque bouteille on doit effectuer les opérations suivantes :

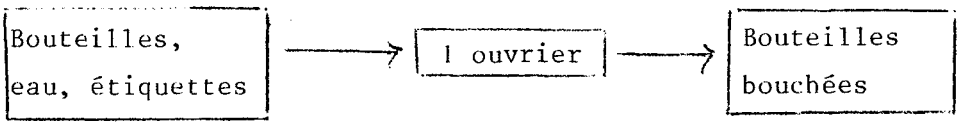
1/ Laver la bouteille : cette opération prend un temps t_1

2/ Remplir la bouteille : temps t_2

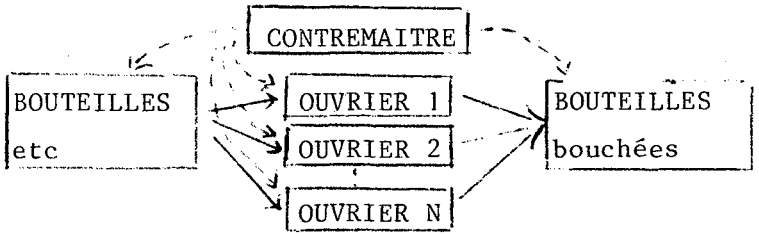
3/ Boucher la bouteille : temps t_3

4/ Poser l'étiquette : temps t_4

L'architecture SISD correspond à un ouvrier seul :



2.3. SIMD



Un certain nombre d'ouvrier N éxécutent les opérations précédentes sous les ordres d'un contremaître; les opérations sont synchronisées à chaque instant c'est la même opération qui est éxécutée (lavage,ou remplissage, ou bouchage, ou étiquettes).
Pour traiter N bouteilles il faut un temps :

$$t_N = t_1 + t_2 + t_3 + t_4$$

et pour k bouteilles :

$$t_k = (t_1 + t_2 + t_3 + t_4) \times k/N$$

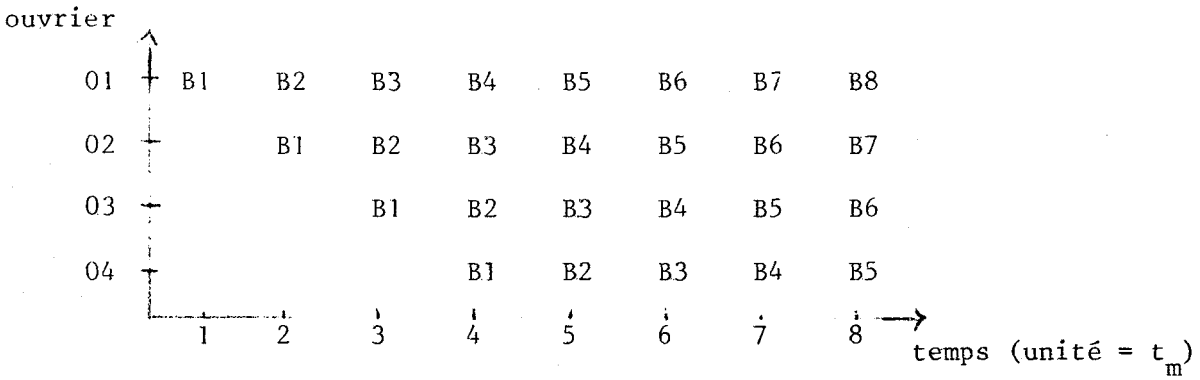
ou k/N désigne la partie entière supérieure de k/N. Noter qu'un certain nombre d'ouvriers restera partiellement inactif si k n'est pas un multiple de N. En laissant les ouvriers travailler à leur rythme on aboutit à la notion de SPMD (Single program, multiple Data) : chaque ouvrier exécute la même séquence d'instructions mais on abandonne la synchronisation.

2.4. Pipelines : la chaîne d'assemblage :

Gardant l'image de l'usine de bouteilles, cette organisation comptera 4 ouvriers : le premier ouvrier lave les bouteilles, le second les remplit, le troisième les bouche, le quatrième pose les étiquettes .
Soit t_m le maximum de temps d'opérations élémentaires :

$$t_m = \max (t_1, t_2, t_3, t_4)$$

A des instants régulièrement espacés par t_m, chaque ouvrier prend une bouteille pour le traitement dont il est chargé. Après une phase de démarrage les 4 ouvriers se trouvent occupés comme l'indique le diagramme suivant ou l'on a indiqué à chaque instant le numéro de la bouteille traitée par chaque ouvrier :



Ainsi la première bouteille sort de la chaîne après 4 périodes t_m; la kième bouteille est prête après un temps :

$$t_k = \sum_{i=1}^{4} t_i + (k-1) \times t_m$$

Donc passéela phase de démarrage il sort une bouteille tous les temps t_m.

L'organisation ci-dessus, shématise l'organisation des calculateurs pipelinés "vectoriels" (CRAY-1, CDC-CYBER205) prévus pour exécuter rapidement de longues séquences d'opérations identiques sur des ensembles de données appelés "vecteurs".

Les opérations arithmétiques peuvent être tronçonnées en opérations élémentaires d'un grand nombre de façons ; prenons l'exemple de l'addition en virgule flottante de deux nombres a_1 et a_2 : chaque opérande est représenté par le couple (exposant,mantisse) :

$$a_1 = m_1 \times b^{e_1} \qquad a_2 = m_2 \times b^{e_2}$$

(b= base de la représentation en virgule flottante, par exemple b=16)

On écrit :

$$a_1 + a_2 = (m_1 + m_2 \times b^{e_2 - e_1}) \times b^{e_1} \text{ si } e_1 > e_2$$

$$(m_1 \times b^{e_1 - e_2} + m_2) \times b^{e_2} \text{ si } e_1 > e_2$$

On voit donc que :

x décompose de manière évidente en 4 étape l'addition :

1ere étape : soustraction des exposants

2éme étape : multiplication de l'une des mantisses par $b^{\pm (e_1 - e_2)}$ (décalage)

3ème étape : additions des mantisses

4ème étape : normalisation du résultat

2.5. MIMD, ou l'affranchissement des esclaves :

Cette fois non seulement les ouvriers travaillent à leur rythme mais ils peuvent exécuter des séquences d'opérations, ou programmesdifférents. L'image la plus expressive pour décrire l'architecture MIMD est celle de la construction d'une maison où plusieurs corps de métiers travaillent simultanément. La première difficulté est la synchronisation : le peintre attend que le platrier est fini sa tâche pour poser le papier peint !

Un exemple de calculateur MIMD est le CRAY-XMP constitué de 2 processeurs[1] capable de coopérer pour servir un même utilisateur. Celui-ci doit alors découper son problème en deux tâches si possible équivalentes.

Prenons l'exemple de la factorisation $L\,L^T$ de la matrice d'un problème d'éléments finis. Il est généralement possible de renuméroter les inconnues de façon à ce que la matrice se présente sous la forme :

$$A = \begin{pmatrix} A_1 & 0 & B_1^T \\ 0 & A_2 & B_2^T \\ B_1 & B_2 & A_3 \end{pmatrix}$$

Alors :

$$A = L\,L^T,\quad L = \begin{pmatrix} L_1 & 0 & 0 \\ 0 & L_2 & 0 \\ C_1 & C_2 & L_3 \end{pmatrix}$$

où les blocs L_i et C_i sont définis par les équations :

$$L_i\,L_i^T = A_i\ ,\ i = 1,2$$

$$C_i = B_i\,(L_i^T)^{-1}$$

$$C_1C_1^T + C_2C_2^T + L_3L_3^T = A_3$$

En affectant le traitement des blocs A_1et A_3 au processeur 1, et le traitement de A_2 au processeur 2, on obtient le shéma suivant :

PROCESSEUR 1	PROCESSEUR 2
1/ Calcul de L_1: $L_1L_1^T = A_1$	1/ Calcul de L_2: $L_2L_2^T = A_2$
2/ Calcul de C_1: $C_1 = B_1(L_1^T)^{-1}$	2/ Calcul de C_2: $C_2 = B_2(L_2^T)^{-1}$
3/ attendre C_2 ⟵	3/ Prévenir le processeur 1 que C_2 est prêt
4: Calcul de L_3: $L_3L_3^T = A_3 - C_1C_1^T - C_2C_2^T$	

(1) Chaque processeur est lui même un calculateur pipeliné vectoriel du même genre que le CRAY-1.

Sur le CRAY-XMP toute la mémoire est partagée; le processeur 2 "avertit" le processeur 1 que le bloc C2 est disponible (synchronisation).

Dans d'autres architectures, C2 pourrait se trouver dans une mémoire privée du processeur 2 qui doit alors envoyer les données C2 vers le processeur 1.

High-Speed Machines and Their Compilers

D.J. Kuck

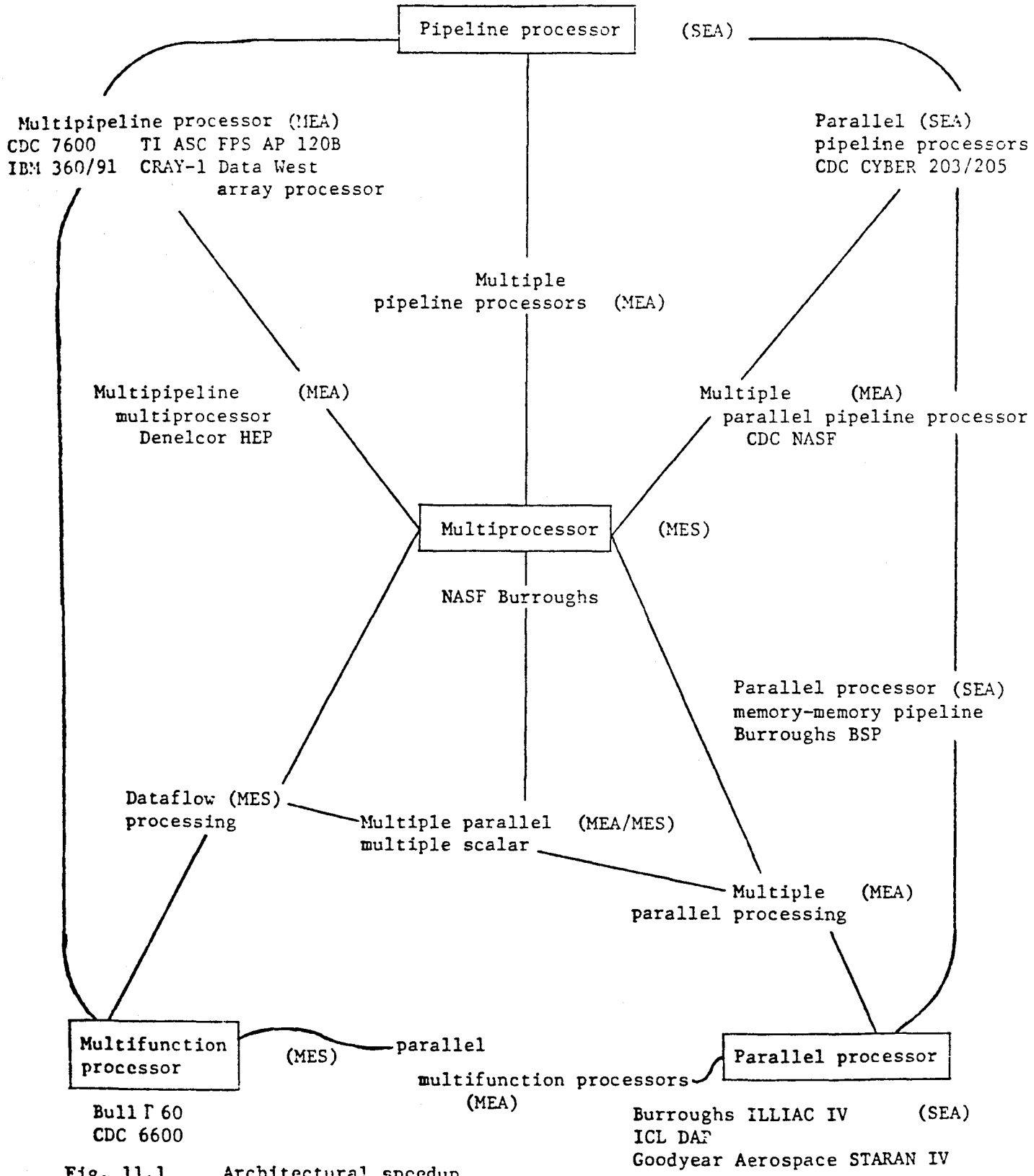


Fig. 11.1 Architectural speedup techniques

Références (Supercalculateur - calcul "parallèle")

R.W. HOCKNEY & C.R. JESSHOPE
"parallel Computers", Adam Hilger, Bristol, GB (1981)

Don HELLER
"A survey of parallel algorithms in numerical linear algebra", SIAM review, N°20 (1978) pp 740

C.V. RAMAMOORTHI & H.F. LI
"Pipeline architecture", Computing surveys, N° 9 (1977) pp 61 - 102

Richard M. RUSSELL
"The CRAY-1 computer system",
Communications of ACM N° 21 (1978) pp 63 -72

P.M. JOHNSON
"An Introduction to vector processing"
Computer Design N° 17 (2), pp 89 - 97, 1978

David J. KUCK, Duncan H. LAWRIE & Ahmed SAMEH
"High Speed computer and algorithm organisation" Academic Press, 1977

D.J. KUCK
"The Structure of computers and compulations" J. Wileg & sons, 1978

Peter M. KOGGE
"The architecture of Pipelined computers"
Mac Graw Hill, 1981

Ronald LEVINE
"Les superordinateurs", Pour la science,
Mars 1982

Dean R. CHAPMAN
"Computational Aerodynamics, Development and out look",
Vol 17,N 12, December 1979, pp 1293 - 1313 AIAA Journal

Jacques LENFANT
"Mémoires parallèles et réseaux d'interconnexion",
Technique et Science Informatique
Vol 1, N° 2, 1982, pp 135 - 142

Michael J. FLYNN
"Some computer organization and their effectiveness"
IEEE Transactions on computers, vol C - 21, N° 9, Sept 1972

CHAPITRE IV

UN CALCULATEUR VECTORIEL : LE CRAY-1

UN CALCULATEUR VECTORIEL : LE CRAY-1S

1 - INTRODUCTION

Considérons la boucle FORTRAN suivante qui additionne deux ensembles de nombres :

(1)
```
      DØ  1I = 1,N
         A(I)= B(I)+C(I)
   1 CØNTINUE
```

Sur un calculateur séquentiel "classique" cette séquence sera traduite en une suite d'opérations de la forme :

(2)
```
      . Initialiser le compteur I = 1 (1)
   1  . Tester si I ≦ N  = sinon aller en 2
      . Calculer les adresses des opérandes
      . Charger B(i) (mouvement de mémoire à registre)
      . Charger C(i)
      . Additionner
      . Ranger le résultat dans A(i)
      . I = I + 1
      . Aller en 1
   2  . .........
```

Dans un tel calculateur, une boucle très simple conduit en fait à répéter un grand nombre de fois une séquence d'une dizaine d'instructions, qu'il faut bien sûr décoder à chaque fois ... Une économie substantielle peut être réalisée si l'on dispose (au moins au niveau du langage machine) d'instructions "vectorielles" remplaçant l'ensemble des opérations élémentaires (2).

Un calculateur est dit "vectoriel" quand de telles instructions

(1) en général le compteur est maintenu dans un registre.

ont été prévues dès la conception du calculateur : le CRAY-1 (1) est un calculateur de ce type, qui peut exécuter rapidement des instructions vectorielles grâce à une architecture pipelinée.

On parlera de code scalaire pour désigner les portions de programme qui ne s'expriment pas à l'aide des instructions vectorielles de la machine (ou qu'on ne peut pas traduire en instructions vectorielles de la machine). La frontière entre le caractère scalaire et le vectoriel ne peut être précisée que pour un calculateur donné (la définition des "vecteurs" sur CRAY-1 sera donnée au § 3).
Vectoriser un programme c'est le transformer pour augmenter (si possible !) la partie vectorielle par rapport à la partie scalaire.

2 - L'ARCHITECTURE DU CRAY-1

2.1. Mémoire et registres

Un des problèmes à résoudre dans la réalisation d'un calculateur vectoriel est l'accès aux données qui doivent (2) impérativement alimenter le pipeline à un rythme régulier (1 donnée par période d'horloge, soit 12,5 ns).
Comme il est difficile de réaliser une mémoire de taille suffisante (quelques millions de mots) soutenant de tels débits les architectes de machines ont eu recours à diverses techniques :

- présence d'une mémoire locale (3) servant de tampon entre la mémoire principale et les unités de calcul : elle garantit au pipeline un approvisionnement régulier en données.
- Divisions de la mémoire en bancs : chaque banc de mémoire travaille de façon autonome.

---

(1) Le CRAY-1 a été conçu par Seymour Cray, qui était aussi le concepteur du 7600 chez CDC ; le premier CRAY-1 a été livré en 1976 à Los Alamos ; les CRAY-1 et CRAY-XMP sont fabriqués et commercialisés par Cray Research Inc.

(2) Dans une instruction vectorielle.

(3) Appelée mémoire "cache" sur d'autres calculateurs.

Sur le CRAY-1 la mémoire locale pour les unités de calcul pipelinées est constituée par 8 registres vectoriels (numérotés de V0 à V7). Chaque registre vectoriel contient 64 mots de 64 bits chacun et peut-être chargé directement depuis la mémoire.(1).
Les calculs "scalaires" utilisent 8 registres dits scalaires (S0 à S7) de 64 bits et les calculs d'adresse utilisent 8 registres de 24 bits (A0 à A7) ; des tampons s'interposent entre ces registres scalaires et la mémoire (T0 à T63), 64 tampons de 24 bits entre les Ai et la mémoire. Enfin 4 tampons d'instructions peuvent contenir chacun 16 mots de 64 bits (2) . Les unités de calcul prennent leurs données exclusivement dans les registres A, S, V.

La mémoire (jusqu'à 4 x 106 mots de 64 bits) est divisée en 16 bancs qui peuvent travailler en parallèle ; le temps de réponse est 50 ns : autrement dit il s'écoule 4 périodes d'horloge (4 x 12,5 ns) entre le lancement d'une requête-mémoire et le moment où l'on peut lancer une autre requête sur le même banc). Comme il y a 16 bancs on pourrait penser que les registres vectoriels peuvent être alimentés à raison de $16 \times \frac{1}{50\text{ns}} = 320 \times 10^6$ mots par seconde.(3).

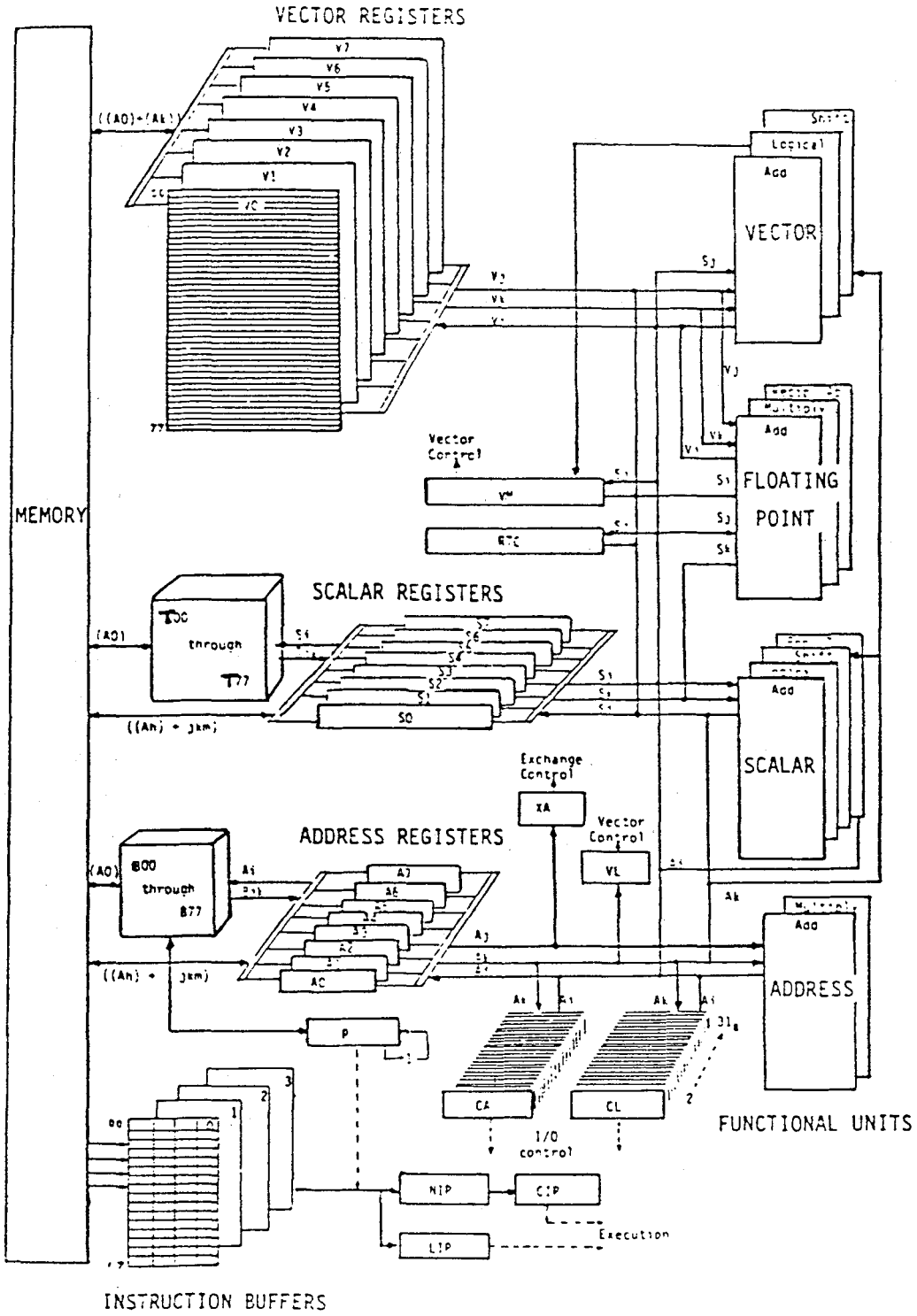
---

(1) L'arithmétique en virgule flottante est effectuée sur des mots de 64 bits.

(2) Une instruction pouvant être codée sur 16 ou 32 bits, un mot peut contenir 1, 2 ou 3 instructions.

(3) En anglais "débit de mémoire" se dit "bandwidth".

Fig. 1 = L'unité de calcul du CRAY-1 et la mémoire.



- Richard RUSSELL "The CRAY-1 computer system"
CACM Vol. 21 - n° 1 janv. 78 p. 26

- "CRAY 1 S series : Hardware reference Manual HR 0808".

En fait le débit est plus faible ; en effet, il n'existe qu'un seul chemin (1) entre la mémoire et les registres (vectoriels ou scalaires) ce qui ramène le débit maximum de chargement des registres à 80×10^6 mots par seconde. Ce débit est faible par rapport aux capacités de calcul du CRAY-1 comme on le verra plus loin.
Dans le plus mauvais cas, lorsque l'on adresse des mots se trouvant tous dans un même banc, le débit tombe à 20×10^6 mots par seconde.

En pratique il faut donc, pour obtenir la performance maximum :

- éviter d'adresser le même banc mémoire deux fois de suite (on verra plus loin comment s'y prendre) ;
- essayer d'effectuer un maximum de calculs sur une donnée pendant son séjour dans un registre,
- être en mode vectoriel, où toutes les adresses peuvent être calculées à l'avance.

On notera que les 4 tampons d'instructions disposent chacun d'un chemin d'accès à la mémoire et donc que les instructions peuvent être chargées à la vitesse maximum de 320×10^6 mots par seconde.

En fait, ces débits maximaux ne peuvent être obtenus qu'après un temps d'amorçage de 15 périodes d'horloge : en effet des étages de contrôle s'intercalent entre la mémoire et les registres, qui occupent 11 périodes ; d'où le délai de 4 + 11 = 15 périodes d'horloge entre le lancement de la première requête mémoire et la présence de la donnée dans un registre.

Un tableau FORTRAN x (N) est rangé en mémoire de la manière suivante : si m_o est le numéro du banc de mémoire contenant X(1), l'élément x(2) se trouve dans le banc m_o + 1 modulo 16 ; l'élément X(i) dans le banc = m_o + i - 1 modulo 16 $(o \leq m_o \leq 16)$

x(17)	x(18)	x(19)	x(20)	x(21)		x(30)	x(31)	x(33)
x(1)	x(2)	x(3)	x(4)	x(5)		x(14)	x(15)	x(16)
banco 1		2	3	4		13	14	15

(1) En (fr)anglais on dit un "bus".

Les instructions capables de charger les registres vectoriels depuis la mémoire sont prévues pour charger successivement des mots séparés par des intervalles constants, par exemple la séquence : X (1), X (3), X (5), X(7), X (9) ...

Grâce aux remarques précédentes on notera qu'une séquence avec un incrément de 8 peut être chargée à raison d'un mot toutes les deux périodes (40 X 10^6 mots/s) ; une séquence avec un incrément de 16 à raison d'un mot toutes les 4 périodes (20 x 10^6 mots/s). Les autres séquences "régulières" à raison d'un mot par période (80 x 10^6 mots/s).

2.2. Unités fonctionnelles.

Le CRAY-1 est doté de 12 unités fonctionnelles, listées au tableau 1, spécialisées chacune en vue d'un type d'opération.

Chaque unité fonctionnelle prend plus d'une période d'horloge pour être exécutée. Le traitement est accéléré par les techniques suivantes :

- ségmentation des unités fonctionnelles (pipeline)
- fonctionnement des unités en parallèle (découplage)
- chainage
- l'auto-enchainement

2.2.1. La segmentation des unités fonctionnelles. (pipeline) (1)

Chaque unité fonctionnelle est divisée en un certain nombre d'étages élémentaires, chaque étape durant exactement une période d'horloge. Une unité fonctionnelle peut alors éxécuter les étapes distinctes d'opérations successives.

---

(1) La traduction française de "pipeline" par "bitoduc" (BOSSAVIT, EDF, BUCCER 75, 1979) n'a pas reçu l'approbation de l'ensemble de la communauté scientifique.

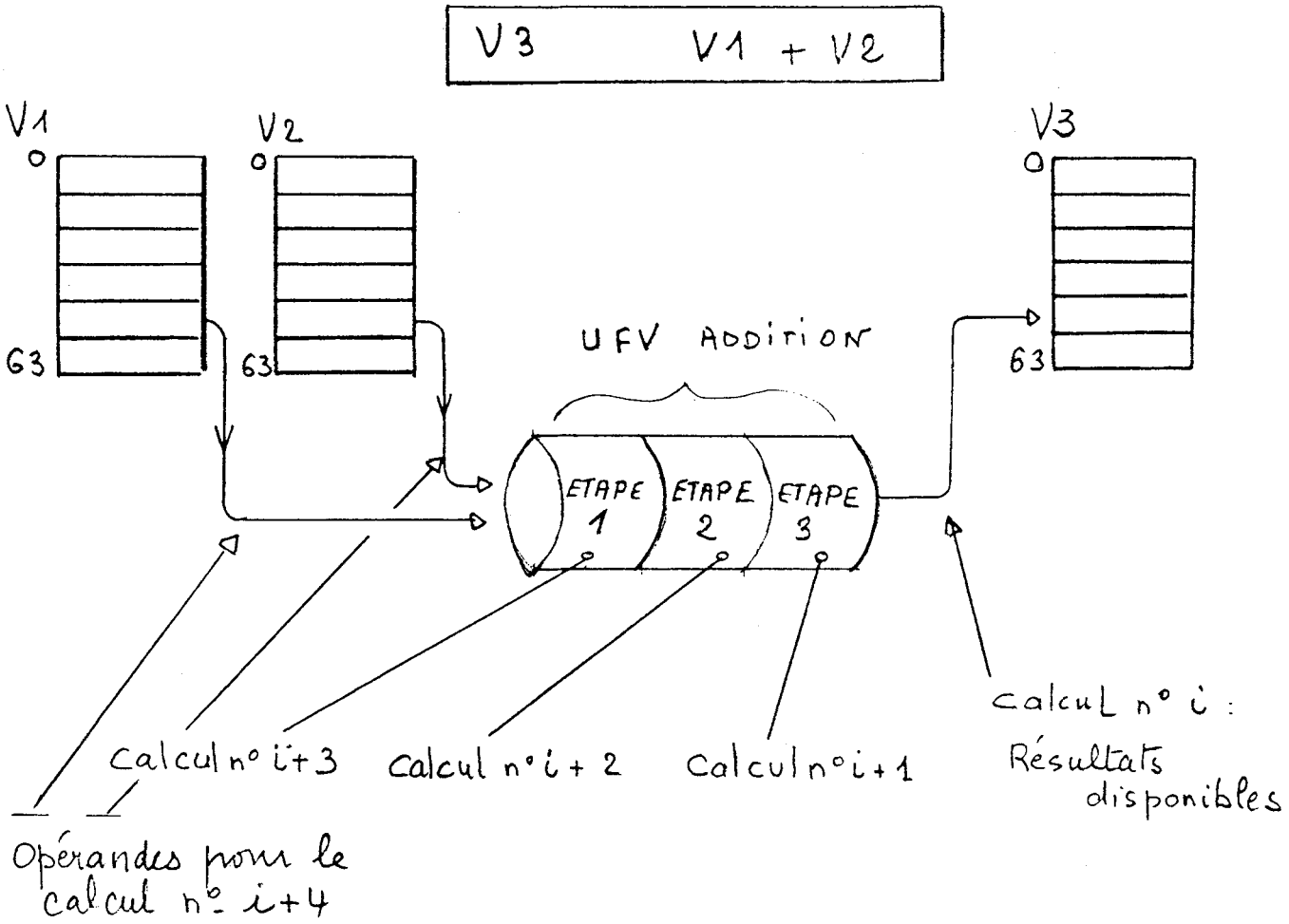
UNITES FONCTIONNELLES	TEMPS D'EXECUTION (en période d'horloge)
i/ Unités fonctionnelles pour le calcul d'adresses. (sur 24 bits)	
Addition - Soustraction	2
Multiplication	6
ii/ Unités fonctionnelles purement scalaires (sur 64 bits).	
Opérations booléennes	1
Décalages	2
Addition /soustraction entière	3
Comptage de bits à o (ou 1)	3
iii/ Unités fonctionnelles Flottantes (64 bits)	
Addition/Soustraction	6
Multiplication	7
Inversion approchée	14
iv/ Unités fonctionnelles purement vectorielles (64 bits)	
Opérations booléennes	2
Décalages	4
Addition/soustraction entières	3

TABLEAU 1 :

Les unités fonctionnelles du CRAY-1.

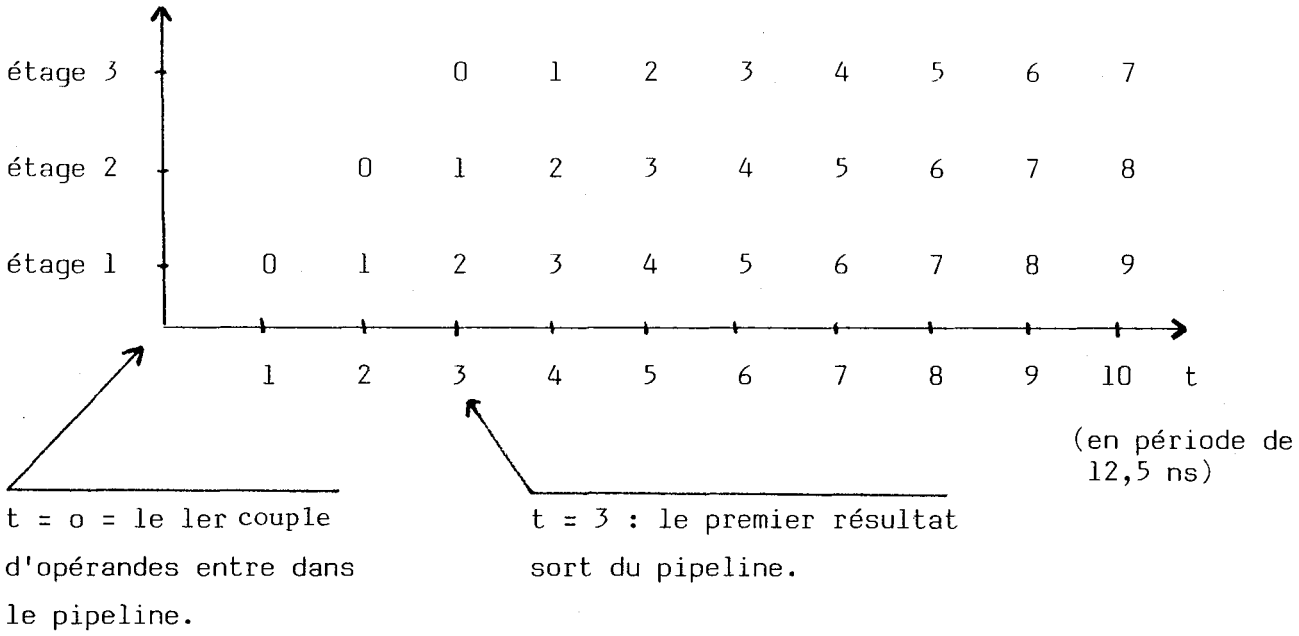
Six des unités fonctionnelles peuvent travailler en mode vectoriel : elles prennent alors leurs données dans les registres vectoriels Vo, V1, ..., V2 et peuvent produire un résultat par période d'horloge après un temps d'amorçage.

Exemple : addition des registres vectoriels V1 et V2, résultat dans V3 (virgule fixe).



L'état de la chaine des opérations à un instant donné est décrit par la figure ci-dessus : quand le résultat n°i (i étant compris entre o et 63) sort du pipeline les données du calcul n° i + 1 entrent dans l'étage 3 du pipeline, etc ... Si n est le nombre d'étage du pipeline, n + 2 éléments sont en opération à un instant donné.

Cette situation peut se représenter également par le diagramme temporel :



On notera que les opérations vectorielles sont toujours de longueur 64, comme la longueur des registres vectoriels.

En fait une unité fonctionnelle peut également prendre l'un de ses opérandes dans l'un des registres scalaires So,...,S7 (qui ne contiennent chacun qu'un mot de 64 bits) afin de traiter en mode vectoriel les opérations du type :

{ SCALAIRE, VECTEUR } → VECTEUR ;

par exemple la multiplication de tous les éléments d'un vecteur par un même scalaire.

2.2.2. Fonctionnement des unités fonctionnelles en parallèle, ou "découplage"

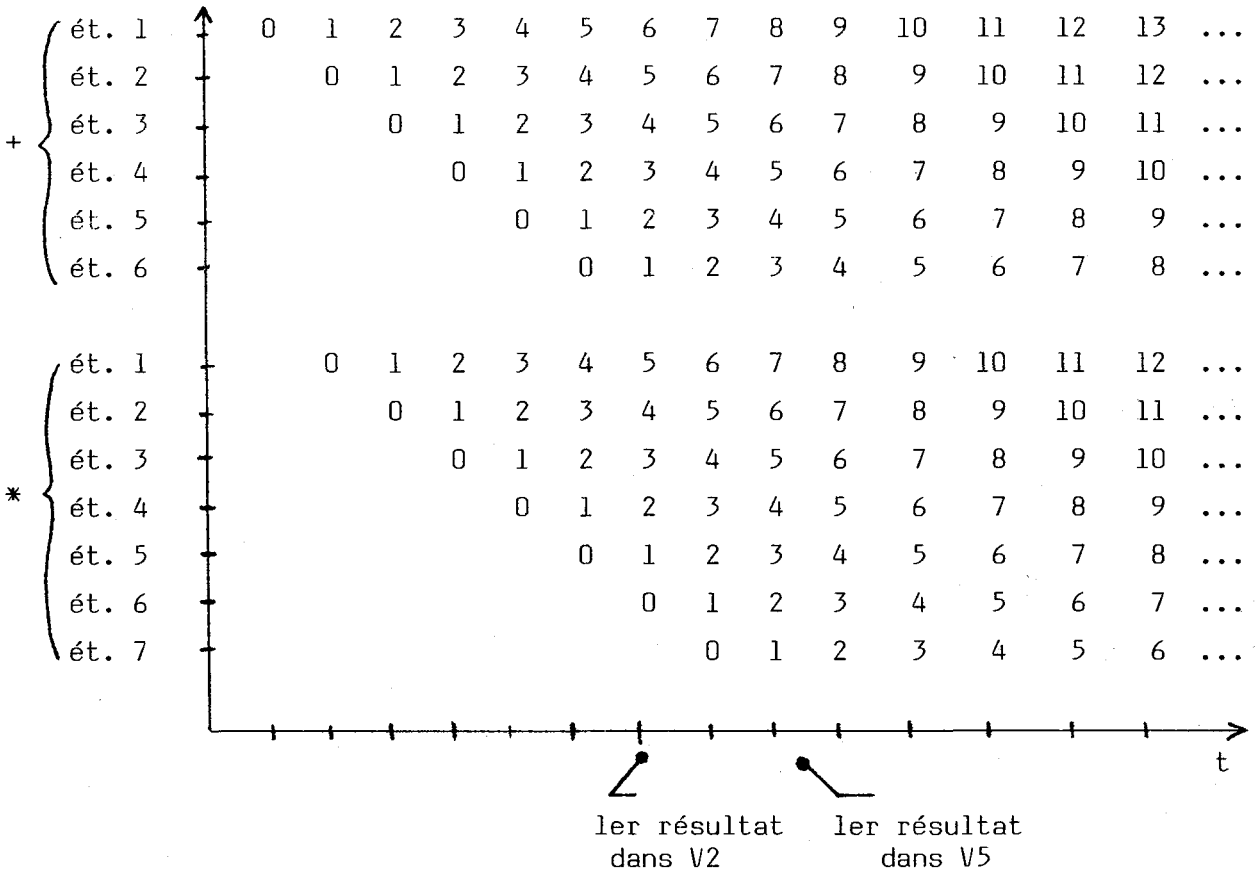
Deux unités fonctionnelles différentes peuvent fonctionner simultanément dès lors que leur opérandes sont disponibles dans des registres différents.

Exemple :

addition flottante de Vo et V1, résultat dans V2
multiplication flottante de V3 et V4, résultat dans V5.

V2	V0 + FV1
V5	V3 * FV4

Une restriction : une seule instruction peut être lancée pendant une période d'horloge ; il y aura donc au moins un décalage d'une période entre deux instructions indépendantes.



Une unité fonctionnelle ne peut servir plus d'une instruction vectorielle : on dit qu'elle est "réservée", pendant toute la durée de l'opération vectorielle.

Dernière remarque :

Les unités scalaires peuvent bien sûr fonctionner en parallèle avec les unités vectorielles.

2.2.3. Chainage

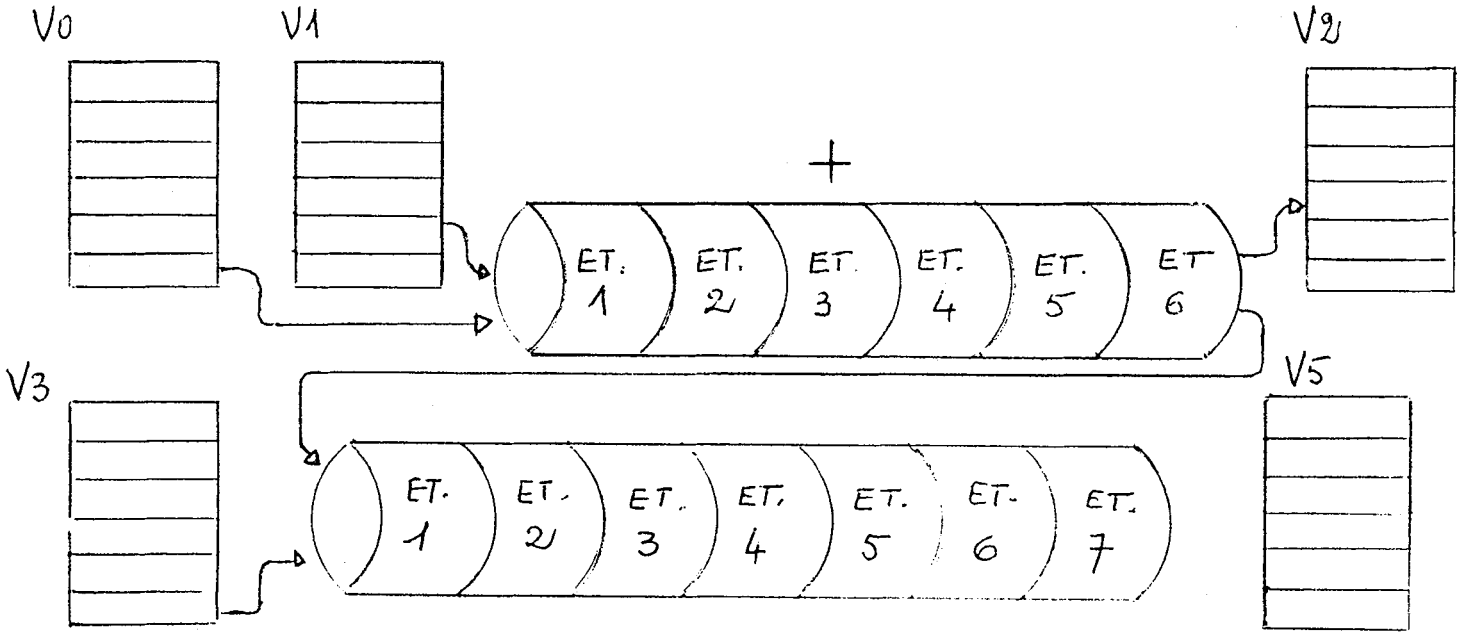
Considérons l'exemple précédent, un peu modifié : l'un des opérandes de la multiplication est pris dans V2 au lieu de V4 :

V2	V0 + FV1
V5	V3 * FV2

(autrement dit on veut mettre (V0 + V1) * V3 dans V5). Il serait vraiment dommage de devoir attendre la fin de l'addition si l'unité de multiplication et les opérandes de la multiplication sont disponibles ! De fait on n'attendra pas, mais seulement dans des circonstances bien précises : la réservation du registre résultat (V2 dans l'exemple) est levée pendant une période (1) de 12,5 ns commençant au temps n + 2 après le lancement de la première instruction.

(n étant le nombre d'étages : pour l'unité d'addition flottante, la période de chainage intervient à 8 périodes après le lancement).

Si au moment de cette période de chainage :
- l'unité de multiplication est libre ;
- le registre V3 est libre et contient l'opérande, alors le premier résultat sortant de la première instruction (c'est à dire le résultat de la première addition) est renvoyé en opérande vers l'unité fonctionnelle vectorielle de multiplication ; les deux unités (+ et *) continuent à travailler en parallèle.



Si au contraire la deuxième instruction n'est pas prête à être lancée au moment de la période de chainage, elle doit attendre la fin de la première instruction vectorielle pour démarrer.

(1) Cette période de chainage est appelée "chain slot time" dans les manuels Cray Res. Inc.

2.2.4. - Autochainage ("fonction récursive")

Dans tous les cas précédents le résultat de la ième opération dans une instruction vectorielle ne dépendait pas des résultats précédents dans la chaine (i-1, i-2, ...). L' "autochainage" est un dispositif des unités fonctionnelles qui permet de renvoyer le résultat sortant du pipeline vers l'entrée, ce qui permet de calculer des expressions comme la somme de 64 nombres (par exemple)

$$S = \sum_{i=i}^{64} x_i$$

Ce calcul s'écrirait en FORTRAN :

```
      S = 0
      DØ 10 i = 1, 64
10       S = S + X (I)
```

Cherchons à effectuer ce calcul par autochainage. Nous aurons besoin de l'unité fonctionnelle d'addition en virgule flottante et de deux registres vectoriels, notés A et B

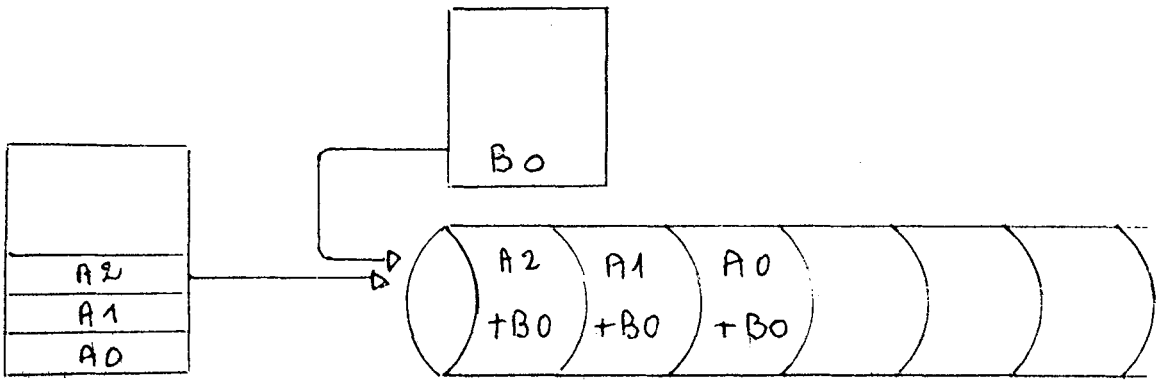
- l'un A contient les données X (i),
- l'autre B contiendra des résultats intermédiaires.

Le calcul se décompose en 2 phases :

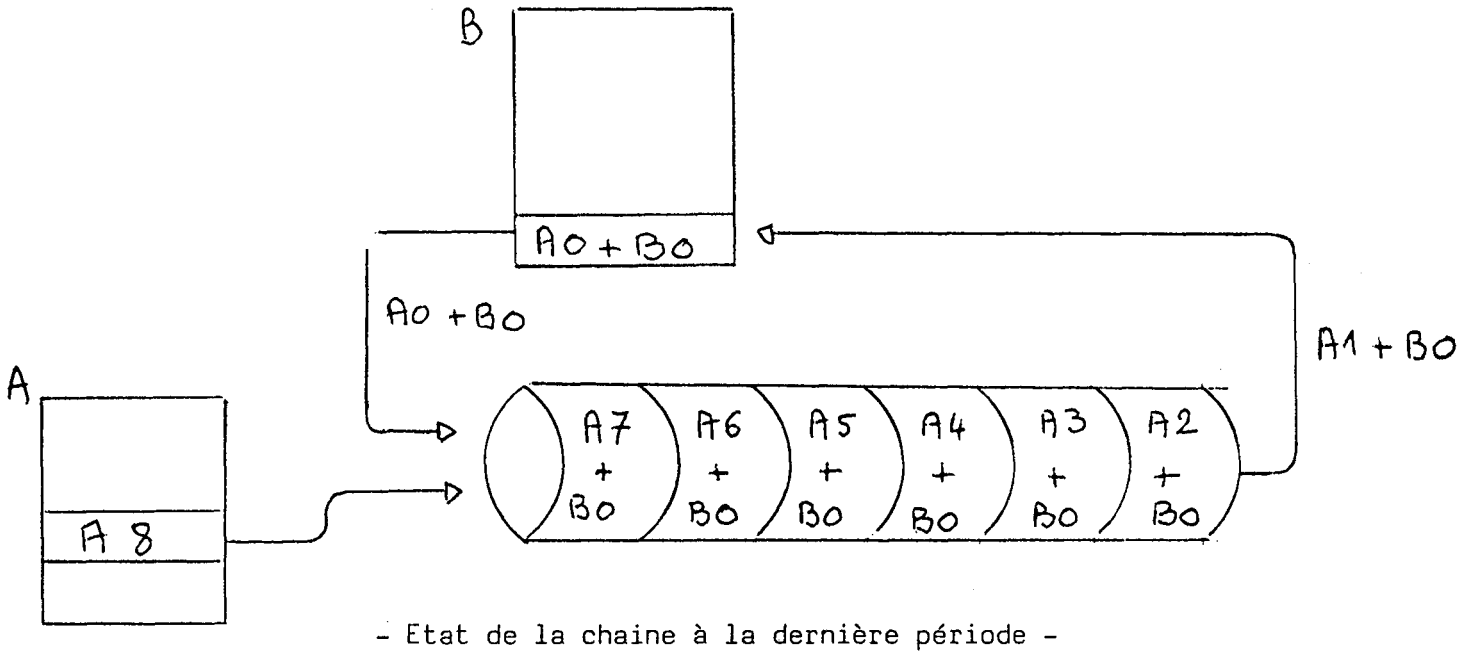
i/ Phase d'initialisation

Le premier élément de B (B0) a été initialisé à 0.

La phase d'initialisation dure le temps nécessaire à la production de la première somme A0 + B0

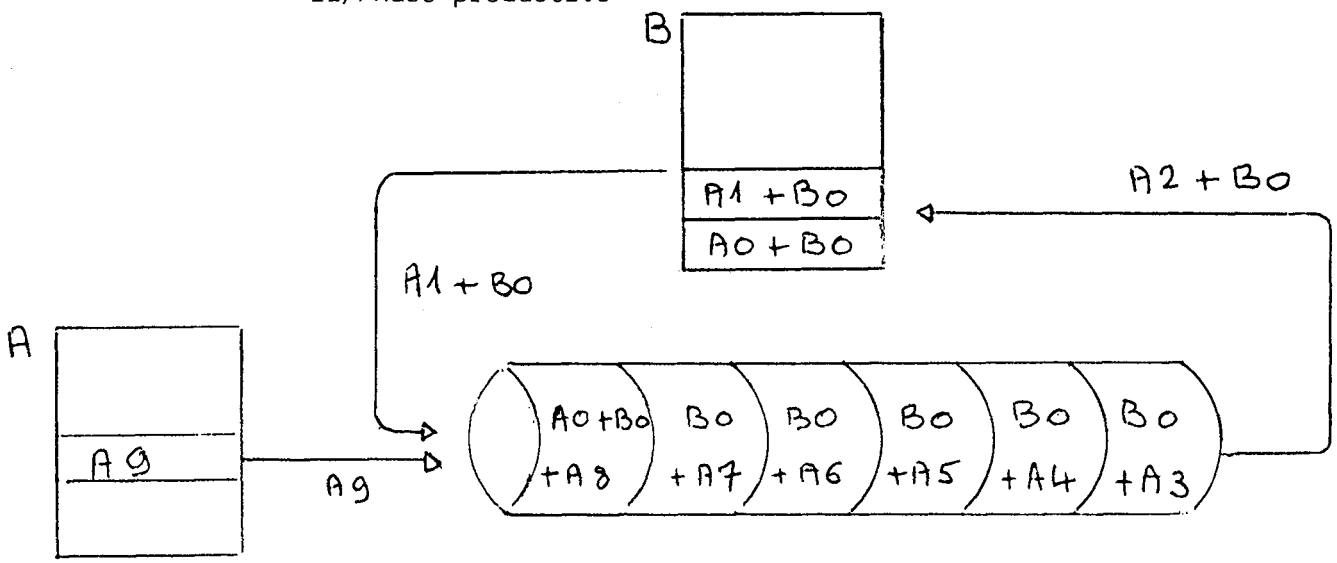


Ce temps est de 8 périodes (= 2 + nombre d'étages du pipeline) = pendant ces 8 périodes la composante issue de B est toujours la même, B0.
Dès que la phase d'initialisation se termine le rang de la composante envoyée depuis B est incrémenté à chaque période d'horloge.



- Etat de la chaine à la dernière période -
de la phase d'initialisation

ii/Phase productive



- Fin de la première période de la phase productive -

A la 2ème période de la phase productive, on commence donc le calcul de A1 + A9 (B0 = 0), et aux périodes suivantes le calcul de :

A2 + A10
A3 + A11
.....
A7 + A15

A la 9ème période de la phase productive on réinjecte A0 + A8 dans le pipeline pour calculer A0 + A8 + A16.

A la fin du traitement on obtient dans B des sommes partielles :

$$B_i = \sum A_j$$

$$\begin{cases} 0 \leq j < i \\ j \equiv i \text{ modulo } 8 \end{cases}$$

Ainsi par exemple :

$$B_{56} = A_0 + A_8 + A_{16} + A_{24} + A_{32} + A_{40} + A_{48} + A_{56}$$

$$B_{57} = A_1 + A_9 + \ldots\ldots\ldots\ldots\ldots + A_{49} + B_{57}$$

$$B_{58} = A_2 + A_{10} + \ldots\ldots\ldots\ldots\ldots + A_{50} + A_{58}$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$B_{63} = A_7 + A_{15} + A_{23} + \ldots\ldots\ldots\ldots + A_{55} + A_{63}$$

On obtient alors le résultat S en sommant ces huit sommes partielles : ce calcul n'est pas vectoriel, mais il n'y a plus que 8 additions à effectuer (cette présentation de l'autochainage est inspirée de De Drouas, 1981).

3 - LE CALCUL VECTORIEL EN FORTRAN.

3.1. Généralités

Le FORTRAN 77 ne contient pas d'instructions vectorielles. cependant le compilateur CFT (compilateur FORTRAN du CRAY, sur ensemble du FORTRAN 77) sait reconnaitre des structures vectorielles

(sous certaines réserves que l'on va détailler) et les traduire en instructions vectorielles (1). Les programmeurs doivent souvent adapter leur style de programmation pour aider cette reconnaissance.

Notons tout d'abord que les programmes ont généralement des boucles de longueur différentes de 64 : c'est le travail du compilateur de décvouper la boucle en tronçons de 64, avec éventuellement un tronçon de longueur inférieure. L'un des registres du calculateur (VL) est d'ailleurs spécialement utilisé pour contenir la longueur des opérations vectorielles.

3.2. Les boucles DO

. Seules les boucles DO sont candidates à la vectorisation ; ainsi CFT ne reconnait pas une opération vectorielle dans la répétition d'opérations suivantes :

```
I = 0
I = I + 1
A(i) = B(I) + C(I)
IF (I.LT.N) GØTØ1
```

3.3. Pas de rupture de chaine

. Pour qu'une boucle soit vectorisable, elle ne doit contenir aucune instruction causant une rupture de séquence dans la répétition des opérations, donc :

- pas d'entrées sorties
- pas de tests (2)
- pas de branchements
- pas d'appels à des sous programmes (3)
- pas de boucle DO (4)

Exemple de boucles non vectorisées :

```
      DO  1  i = 1,N
         A(i) = B(I) + 10
 1    WRITE (6;10)A(I)
10    FORMAT(...)
      DØ  2  I = 1,N
         IF (A(I).GT.0.)B(I) = SQRT(A(i))
 2    CØNTINUE
      DØ 3 I 1,N
         C(I) = 50 * I
 3    CALL SP (C(I))
```

---

(1) Une norme de fortran contenant des instructions vectorielles, dénommée FORTRAN 8X, est à l'étude à l'ANSI, association de contructeurs et d'utilisateurs.

(2) On verra plus loin comment lever cette restriction.

(3) Cependant l'appel aux fonctions spéciales(COS,SIN,...) est vectorisé.

(4) Donc seules les boucles les plus internes seront vectorisées.

Ceci n'autorise donc que les instructions d'affectation du type :

Z(I) = Expression

3.4. Eléments réguliers

. Pour qu'une boucle soit vectrorielle elle ne doit faire référence qu'à des éléments dits "réguliers", que le matériel est capable d'amener vers les registres vectoriels à un débit raisonnable ; ces éléments réguliers comprennent :

- les constantes par rapport à la boucle (1)
- l'indice de boucle
- un élément de tableau dont tous les indices sont constants par rapport à la la boucle, sauf éventuellement un, qui est de la forme :

A * I + B

ou encore

I + B,

avec A et B des constantes et I l'indice de boucle.
Exemples de boucles vectorisées :

```
      DØ 1 I = 1, N
        A (2 * I + 3) = D(5 * I - 1) /F(I)
1       B(I)  = F(I) + G(I)
```

(1) c'est-à-dire les variables qui n'apparaissent pas dans la boucle à gauche d'un signal =

```
      DØ  1  J  =  1, M
          DØ  2  I = 1, N
2             A(I + J * 2) = 16.5 * B(3 * I - J)
1     CONTINUE
```

(Vectorisation de la boucle interne = I est constant par rapport à cette boucle).
Exemples de boucles non vectorisées :

```
      DØ  1  I  =  1, N
          A (I * * 2) = 1.
1  CONTINUE
      DØ  2  I  =  1, N
          A (I,I) = 0.
2  CONTINUE
```

3.5. Dépendances arrières

. Pour qu'une boucle soit vectorisable, il ne faut pas qu'elle contienne de dépendance arrière. Ce cas se présente lorsque l'itération n°i utilise le résultat d'un calcul lancé lors d'une itération précédente, susceptible de ne pas être terminé.

Exemples de dépendances arrières :

```
      DØ  1  I  =  3, N
          A(I) = (A(I - 1)+1)/(A(I-1)+2)
1  CONTINUE
      DØ  2  I  = 5, N
          A (I) = A(I-3) + 4
2  CONTINUE
      DØ  3  I =  1, N
3         S = S + B(I)
```

(Cette boucle est "speudovectorisée" grâce à la technique d'autochainage, cf § 2.2.4.

Exemples de boucles vectorisées (pas de dépendance arrière) :

```
      DØ  1  I  = 2, N, 2
          A(I) = (A(I-1) +1)/2
1 CONTINUE
```

(à gauche du signe "=" on fait référence aux éléments pairs, à droite aux éléments d'indice impair).

```
      DØ  1  I  =  65, 1024
          A(I)  =  A(I-64)/1.52
1     CONTINUE
```

(cette boucle est vectorisée car la longueur d'une opération vectorielle est au maximum 64 sur le CRAY-1).

Plus généralement, soit une boucle de la forme :

```
      DØ  1  I  =  I1, I2, I3
          A(I) = exression (A(I - I4)
1 CONTINUE
```

le lecteur vérifiera qu'elle ne comporte pas de dépendance arrière dans les cas suivants (en supposant I3 >o pour simplifier) :

- si İ3 ne divise pas İ4
- si İ3 divise I4 et I4/I3 $\geqslant$ 64
- si I2 - I4 - I1 $<$ o

(pour obtenir ce résultat, poser J = (İ - İ1)/İ3 + 1

connaissant la valeur des paramètres il peut arriver que le programmeur sache qu'une boucle est sans dépendance arrière alors que le compilateur ne possède pas les mêmes informations.

Dans le doute le compilateur s'abstient par prudence de vectoriser (1). Le programmeur indique au compilateur qu'il peut vectoriser en confiance à l'aide d'une directive de compilation(2) : il s'agit d'une carte placée juste avant la boucle :

```
CDIR$  IVDEP   (c en lère colonne)
  DØ 1 i = I 1, I2,I3
     ..........
1  CONTINUE
```

Remarque : les dépendances "avant" ne gênent pas la vectorisation.

3.6. Dépendance croisée

On dit qu'une boucle contient une dépendance croisée si un élément de vecteur, à une intération i, peut être modifié à une autre itération j, avec j-i $\leqslant$ longueur des vecteurs = 64.

---

(1) une vectorisation dans des cas de dépendance arrière donnerait des résultats imprévisibles, éventuellement faux.
(2) Evidemment la carte IVDEP n'enlèvera pas les dépendances qui existent réellement.

Exemple de boucle avec dépendance croisée :

```
      DØ  1  I  =  1, N
          A(I) =  4 + D(I+16)
          B(I) = 8. *  A(I + 1)
    1 CONTINUE
```

L'interprétation classique de Fortran est la suite de calculs :

```
A(1) = 4 + D(17)
B(1) = 8 *  A(3)
.................
```

Par contre, l'interprétation vectorielle de cette boucle est plutôt obscure : on pourrait en fait trouver plusieurs transcriptions vectorielles, une seule représentant ce que le programmeur voulait dire. Dans le doute le compilateur s'abstient de vectoriser.
Comme pour les dépendances croisées, le programmeur peut lever les doutes du compilateur et le forcer à vectoriser en plaçant une commande :

```
CDIR $  IVDEP
```

avant la boucle (aux risques et périls du programmeur).

3.7. Fonctions masques CVMGZ,CVMGT ...

Considérons la boucle suivante qui comporte un test :

```
      DØ  1  i  =  1,N
          IF(X(I).GT.0.) Z(I) = 3 * SIN(X(I))
    1 CONTINUE
```

Cette boucle n'est pas vectorisée à cause de la présence du IF. Par contre, la boucle suivante est vectorisée et donne le même résultat sur le CRAY-1 :

```
      DØ  2  I  =  1, N
          Z(I) = CVMGT (X(I).GT.0.,3*SIN(X(i), Z(i))
    2 CONTINUE
```

CVMGZ et CVMGT sont deux fonctions spéciales reconnues par le compilateur FORTRAN du CRAY-1.

Pour chaque tronçon de 64 éléments, cette boucle vectorielle s'éxécute en 3 étapes :

1ère étape : on calcule toutes les expressions intervenant dans les affectations (3 * SIN(x(i)) sans tenir compte du test ; on les range dans un registre vectoriel, par exemple dans Vo.

2ème étape: on calcule pour chaque I la valeur de l'expression logique qui gouverne le test (x(I).GT.o) et l'on range le résultat (o ou 1) dans un registre spécialement réservé à cet usage (VM = Vector Mask = 64 bits).

3ème étape : soit par exemple V1 le registre contenant les données x(i) et V2 le registre résultat ; on éxécute alors une instruction "vector merge" : pour chaque position j de o à 63, cette instruction range Vo(j) dans V2(j) si le bit j du masque VM est à 1, et elle range V1(j) dans V2(j) si le même bit de VM est à o.

La boucle est bien vectorisée, mais au prix de quelques sacrifices :

- le code n'est plus portable, les fonctions CVMGZ et GVMGT n'étant pas connues hors du CRAY-1.

- On effectue tous les calculs avant de tenir compte du test. Ceci peut avoir des conséquences plus ou moins graves :

* Si le test s'avère faux pour un grand nombre de valeurs de l'indice I, on aura effectué des calculs pour rien ; dans les cas extrêmes la version "vectorisée" peut s'avérer plus lente que la version "non vectorisée".

* Si le test sert à garantir la validité du calcul,l'utilisation des fonctions ci-dessus peut conduire à des erreurs, par exemple dans le cas suivant :

```
      DØ  1  i = 1, N
         IF (A(I).GT.O.)A(I) = 1./A(I)
    1 CONTINUE
```

Si l'on écrit :

```
      DØ  2  I  = 1, N
        A(I) = CVMGT(A(i).GT.O.,1./A(i), A(i))
    2 CONTINUE
```

Alors la machine éxécutera toutes les divisions quoiqu'il arrive et dans ce cas on ne pourra se prémunir contre les divisions par zéro.

3.8. Fonctions de la bibliothèque.

Il est généralement avantageux du point de vue de la performance, d'utiliser des fonctions codées en assembleur, en particulier les fonctions de la bibliothèque LINPACK(1). Parmi les plus usuelles on trouve :

SDOT : produit scalaire de deux vecteurs
(utilise le procédé d'autochainage, cf. § 2.2.4).

SUM : Somme des éléments d'un vecteur.

SNRM2 : Norme d'un vecteur (à utiliser de préférence à SDOT dans ce cas)

SASUM : Somme des valeurs absolues des éléments d'un vecteur.

SAXPY : Combinaison linéaire de deux vecteurs ($a\,\vec{x} + \vec{y}$)

Cette bibliothèque étant assez largement répandue, la perte de portabilité est limitée.

3.9. Adressage indirect.

Comme on a pu s'en rendre compte, le compilateur CFT ne vectorise pas des boucles du genre :

```
      DØ  1  I  = 1, N
          A(i) = B(TAB(I)) + 1
    1 CONTINUE
```

(A,B et TAB étant des tableaux).

En effet, le compilateur n'a aucun moyen de vérifier que les données B (TAB(i)) sont rangées régulièrement et peuvent être accédées en mode vectoriel.

Ce type de boucle se rencontre néanmoins dans de nombreuses applications :

- éléments finis sur un maillage quelconque ;
- constitution d'histogrammes ;
- interpolation d'une fonction tabulée.

(1) qui est appelée par défaut.

3.10. Remarques générales sur la vectorisation.

Il convient de tempérer l'ardeur d'un programmeur à vectoriser par quelques règles ; il doit tout d'abord avant de transformer son programme se poser quelques questions :

. est-ce-que l'algorithme utilisé est bon ? l'efficacité de la vectorisation peut-être illusoire si la complexité de l'algorithme est de l'ordre O (n^2) alors qu'il existe un alrorithme de complexité O (logn) (par exemple). Eventuellement il faut envisager un autre algorithme mieux adapté au calcul vectoriel.

. Quelles sont les parties du programme qui prennent le plus de temps . (Elles ne représentent souvent qu'un faible pourcentage en nombre de lignes de code).

. Est-ce-que le programme doit continuer à tourner sur d'autres machines ? (1)

. Est-ce-que l'éxécution du programme est limitée par l'attente des opérations d'Entrées/Sorties (i/O) ?

Si le programmeur décide malgré tout de modifier son programme pour aider la vectorisation il doit garder à l'esprit la trivialité suivante : le programme doit continuer à donner des résultats justes !

Il est prudent de spécifier assez précisément les fonctionalités à remplir par chaque portion de programme et de s'assurer à chaque modification qu'elles sont respectées.

Des résultats faux n'ont qu'un intérêt médiocre, même obtenus très rapidement ...

3.11. Exemples de code assembleur généré par C.F.T.

Les instructions assembleur correspondant au corps de la boucle 10 sont comprises entre les lignes marquées par $ 10 A et $ 10 B : ces lignes correspondent à un tronçon de 64 opérations et se terminent par un transfert au début (JAN $ 10 A) pour le cas ou plusieurs tronçons seraient nécessaires.

On repère des instructions vectorielles :

`V3 ,A0,1`	Transfert depuis la mémoire (à l'adresse donnée par le registre A0), vers V3, avec incrément de 1 dans les adresses.
`,A0,1 V3`	Transfert depuis V3 vers la mémoire.
`V5 V6*RV7`	Multiplication en virgule flottante des registres vectoriels V6 et V7, résultat dans V5.
`V3 V4-FV5`	Soustraction en virgule flottante.

(1) l'écriture "vectorisée" risque de le rendre moins performant sur des machines "séquentielles".

```
1.          SUBROUTINE TEST1 (A,B,C,D,E,F,N)
2.          DIMENSION A(N),B(N),C(N),D(N),E(N),F(N)
3.          DO 10 I = 1, N
4.    10    A(I) = B(I) - C(I)*D(I) - E(I)*F(I)
5.          RETURN
6.          END
```

```
                BLOCK BEGINS AT SEQ. NO.        1, P=       15a
1107 00000000+  0200 00000000+                  ST3+0,0  A7                 A0  ST3+0
022700  035700  0200 00000002+                  A7  30          ,A0  B30,A7 A0  ST3+0
022700  037700  0207 00000000+                  A7  00          ,A0  T00,A7 A7  ST3+0
025701                                          301  A7
          VECTOR LOOP BEGINS AT SEQ. NO.        3, P=       20b
                BLOCK BEGINS AT SEQ. NO.        3, P=       20b
          1001 00000006+  1213 00000000                     A1  =TEST1+6,0                  S3  +0,A1
          023730  022601  1303 00000003+                     A7  S3              A6  01      =TEST1+3,0  S3
          031567  042677  1306 00000004+                     A5  A6-A7           S6  <01     I,J  S6
          031450  030004  012                               A4  A5-1            A0  A4      JAP     =106
          025403  1002 00000012+                           203  A4             A2  =TEST1+12,0
1006 00000011+  1007 00000013+                  A5  =TEST1+11,0                 A7  =TEST1+10,0
071204  1005 00000007+  042172                  S2  A4              A5  =TEST1+7,0          S1  <06
1001 00000013+  045712  1104 00000014+          A1  =TEST1+13,0                 S7  =S2&S1          A4  =TEST1+14,0
          023370  030330  025302                            A3  S7              A3  A3+1            202  A3
031300  075600  030032  024203                  =6  -1              T00  S6     A0  A3+A2           A2  202
030636  030737  025006  030535                  A6  A3+A6           A7  A3+A7   206  A0             A5  A3+A5
025605  030631  025704  030734                  205  A6             A6  A3+A1   204  A7             A7  A7+A6
025203                                          203  A2
=     110A
          024400  023440  024302                            S4  T00             A4  S4              A3  =02
024106  030014  002003  176300                  A1  =05             A0  A1+A4   VL  A3              V3  ,A0,1
024205  030024  176200  165723                  A2  =05             A0  A2+A4   V2  ,A0,1          V7  V2*RV3
024104  030014  176100  030054                  A1  =04             A0  A1+A4   V1  ,A0,1          A0  A3+A6
176000  165501  030364  176600                  V0  ,A0,1           V5  V0*RV1  A0  A4+A4           V6  ,A0,1
173467  173345  030074  071513                  V4  V6-FV7          V3  V4-FV5  A0  A7+A4           S5  +A3
172030  060345  024203  030023                  ,A0,1  V3           S3  S4+S5   A2  203             A0  A2+A3
027101  075300  025003  025102                  A1  =S0             T00  S3     206  A0             202  A1
011       346+  1303 00000004+                  JAN     =10A                    I,0  S3
                BLOCK BEGINS AT SEQ. NO.        5, P=       45a
=     110A
043100  0200 00000037+  022700                  S1  >00             A0  =TB+37                      A7  10
034700  0200 00000000+  022700                  200,A7  ,A0         A0  =TB+0                       A7  20
036700  005000  044444  044444                  T00,A7  ,A0         J  300              S4  S4&S4   S4  S4+S4
PAGE 2                  00=CDEGLMPRSTVX                 12/01/83-09:14:25       CFT 1.10(10/19/83)  PAGE 7
```

EXEMPLE 1 : Exemple de code assembleur généré par le compilateur fortran CFT : Une boucle vectorisée.

Le premier exemple montre le chainage entre le transfert de mémoire à registre, et les opérations arithmétiques. Par contre on voit que le compilateur commence par effectuer toutes les multiplications puis les additions (conformément aux règles de priorité de FORTRAN) ce qui n'est pas optimal comme le montre l'exemple 2. Dans l'exemple 2, on a forcé,(1) par un jeu de parenthèses, le compilateur à changer l'ordre des opérations (pour donner un programme équivalent au premier), et à alterner multiplications et soustractions : on a cette fois un chainage double : transfert mémoire à registre - multiplication - soustraction.

(1) le compilateur évalue toujours le niveau le plus interne de parenthèses en premier.

```
1.          SUBROUTINE TEST2 (A,B,C,D,E,F,N)
2.          DIMENSION A(N),B(N),C(N),D(N),E(N),F(N)
3.          DO 10 I = 1, N
4.  10      A(I) = (B(I) - C(I)*D(I)) - E(I)*F(I)
5.          RETURN
6.          END
```

```
          025403   1002 000000012+                    B73  A4          A2  @TEST2+12,0
1006 00000011+     1007 000000013+          A6  @TEST2+11,0            A7  @TEST2+13,0
071204   1005 00000010+   042172                      A5  @TEST2+10,0                  S1  <76
1001 00000007+   045712   1004 000000014+  A1  @TEST2+7,0             S7  #S2&S1       A4  @TEST2+14,0
         023370   030330   025302                                      A3  A3+1         J02  A3
031300   075600   030032   024203           A3  -1                     T00  S6         A0  A3+A2       A2  B03
030635   030737   025006   030535           A6  A3+A6                  A7  A3+A7       B06  A0          A5  A3+A5
025605   030631   025704   030734           B05  A3                    A6  A3+A1       B04  A7          A7  A3+A4
025203                                      B03  A2
=     $10A
         074400   023440   024302                                      S4  T00         A4  S4          A3  B02
024106   030014   002003   176300           11  B05                    A0  A1+A4       VL  A3          V3  ,A0,1
024205   030024   175200   165123           A2  B05                    A0  A2+A4       V2  ,A0,1       V1  V2+FV3
024104   030014   176000   173401           A1  B04                    A0  A1+A4       V0  ,A0,1       V4  V0-FV1
030054   176700   030064   176400           A2  A5+A4                  VT  ,A0,1       A0  A5+A4       V6  ,A0,1
165567   173345   030074   071513           V5  V6*RVT                 V3  V4-FV5      A0  A7+A4           +A5
177030   060345   024203   030023           ,A0,1  V3                  S3  S4+S5       A2  B03         A0  A2+A3
027100   075300   025003   025102           A1  Z51                    T00  S3        B03  A0         B02  A1
011        34b+   1303 000000004+           JAN     $10A                               I,0  S3
          BLOCK BEGINS AT SEQ. NO.      5, P=      45a
=     $10B
043100   0200 000000037+   022700           S1  >37                    A0  $T8+37                    A7  00
034700   0200 000000000+   022700           B00,A7  ,A0                A0  $T8+0                     A7  00
136700   005000   044444   044444           T00,A7  ,A0                J   000          S4  S4&S4       S4  S4&S4
AGE 2                  ON=CDEGLMNPRRSTVX              12/01/83-09:14:25          CFT 1.10(10/19/83)  PAGE 9
```

EXEMPLE 2 : Code assembleur généré par CFT : Une boucle vectorisée avec chaînage. Comparer avec l'exemple 1.

4 - QUELQUES REMARQUES EN CONCLUSION

. Le CRAY-1 est évidemment une machine rapide : avec 12,5 ns - il possède la plus petite période d'horloge parmi les ordinateurs sur le marché (1) (2).

. Les temps d'amorçage des pipelines sont assez faibles (quelques unités) ce qui donne des performances correctes même avec des vecteurs courts (20).

. On peut accéder en mode vecteur à des éléments espacés de pas quelconques, ce qui n'est pas vrai par exemple sur le CDC CYBER 205.

. La faiblesse du CRAY-1 semble être une bande passante insuffisante pour la mémoire : considérons à nouveau la boucle suivante :

```
      DØ  1  i = 1, N
        A(i)=B(i)+C(i)

    1 CONTINUE
```

Pour chaque tronçon de 64 éléments il faut :

Charger un registre opérande
Charger le deuxième registre opérande
Additionner
Ranger les résultats en mémoire.

} chainage

On voit qu'il y a recouvrement d'une lecture mémoire par l'addition ; néanmoins, la machine passe une grande partie de son temps à attendre la mémoire, avec les unités fonctionnelles inactives.
Encore s'est-on placé dans le cas favorable où l'on chainait la lecture mémoire et une opération arithmétique. Celà n'est pas vrai quand la lecture mémoire ne peut s'effectuer au rythme voulu par les unités fonctionnelles ; par exemple lorsque les adresses des éléments accédés sont en progression arithmétique de raison 8 ou 16, comme dans la boucle :

```
      DIMENSION A(16,N), B(16,N), C(16,N)
      DØ  1  J  =  1,N

            A(K,J) = B(K,J) + C(K,J)
```

. (1) le CRAY - XMP a une période d'horloge de 9,5 ns. Cette phrase était vraie en 1983.
(2) Par suite les calculs en mode scalaire sont également relativement rapides.

. Le secret pour obtenir de hautes performances est d'essayer de faire utiliser au mieux les données qui se trouvent dans les registres ; autrement dit quand une donnée a été amenée dans un registre, tenter d'effectuer un maximum de calculs avec cette donnée. Cet objectif amène à fusionner les corps de plusieurs boucles.

. Ce gonflement des boucles est tempéré par deux remarques :

i/ il n'y a que 8 registres vectoriels ;

ii/ il faut que le corps d'une boucle vectorisée tienne dans les buffers d'instructions, qui ne contiennent chacun pas plus de 64 instructions.

REFERENCES :

CRAY-1S series : Hardware Reference Manual

E. de DROUAS : "Vectoriser sur le CRAY-1",
EDF/DER/IMA, Atelier logiciel n°31, 1981.

B. MEYER "Un calculateur vectoriel : le CRAY-1 et sa programmation",
EDF/DER/IMA, Atelier logiciel n°24, 1982.

R. BUTEL Note EDF :
"Evaluation théorique des performances du Cray",
Note E.D.F. n° 10, HI-3947/00, Octobre 1981.

A. BOSSAVIT "Bientôt la vectorisation",
EDF, BUCCER n° 75, Septembre 1979.

CHAPITRE V

QUELQUES ALGORITHMES DE BASE EN CALCUL VECTORIEL

1 - Multiplication matrice x vecteur

2 - Récurrences linéaires d'ordre 1 -
Réduction cyclique

3 - Systèmes linéaires tridiagonaux

3.1. Elimination de Gauss "séquentielle"

3.2. "Recursive doubling"

3.3. Réduction cyclique

4 - Algorithme de Gauss Siedel : méthode des fronts

I - Multiplication matrice x vecteur

Soit A une matrice à M lignes et N colonnes; x et y sont deux vecteurs de $\mathbb{R}^N$ et $\mathbb{R}^M$:

On veut ranger dans y le produit A.x -

Une implantation classique part des équations du produit matrice x vecteur :

$$(1) \qquad y_i = \sum_{j=1}^{N} a_{ij}\, x_j$$

d'où l'on déduit le code FORTRAN :

```
(2)   DØ  1  I = 1,M
          Y (i)= 0
      DØ 2  J  = 1,N
            Y(i) = Y(i) + A (I,J) * X(J)
    2 CØNTINUE
    1 CØNTINUE
```

on remarque que la boucle 2 est un produit scalaire, et donc que ceci paut s'écrire sur CRAY-1 en utilisant la routine de la bibliothèque, SDOT :

```
(3)   DØ  1  I = 1,M
            Y(i) = SDØT (N,X,1, A(I,1), M)
    1 CØNTINUE
```

Cependant le produit scalaire, même optimisé par SDOT, n'est pas une opération vectorielle aussi performante que les autres : on ne passe en mode vectoriel que grâce à un dispositif d'autochaînage analogue à celui décrit au châpitre précedent et qui demande un temps de démarrage assez long.

Une autre façon de voir le même problème est de réecrire (1) sous la forme :

$$(4) \qquad y = \sum_{j=1}^{n} x_j\, a^j$$

où $a^j \in \mathbb{R}^M$ est le jème vecteur colonne de la matrice.

L'équation (4) conduit au code suivant (il faut d'abord initialiser y à 0 dans une boucle séparée) :

(5)
```
      DØ 1 I = 1,M
         Y(I) = 0
1     CØNTINUE
      DØ 3 J = 1,N
         DØ 2 I = 1,M
              Y(I) = Y(I) + A(I,J)  X(J)
2     CØNTINUE
3     CØNTINUE
```

Alors la boucle interne de ce programme est une combinaison linéaire de vecteurs. Il exis te d'ailleurs en bibliothèque une routine qui peut implémenter efficacement les combinaisons linéaires de vecteurs :

(6)
```
      DØ 1 I = 1,M
           Y(I) = 0
1     CØNTINUE
      DØ 3 J = 1,N
           CALL SAXPY (M,X(J), A(1,J),1,Y,1)
3     CØNTINUE
```

Ainsi en reprenant la spécification du problème et en l'écrivant en termes de vecteurs, on obtient aisément une écriture mieux adaptée à la machine.

On peut remarquer que la transformation qui fait passer du programme (2) au programme (5) est en fait très simple : on échange les boucles I et J, de sorte que la boucle en I devient la plus interne.

Les programmes (5) et (6) ne sont pas encore optimaux : les ennuis viennent de ce que la mémoire est trop sollicitée par rapport aux calculs effectués.

En effet, à chaque itération en J, le compilateur provoque le chargement des vecteurs Y( ) et A(•,j), met à jour Y(•), puis écrit en mémoire le vecteur Y(•), sans reconnaitre que Y(•) ne contient qu'un résultat temporaire.

En supposant pour simplifier que M≦64, on a donc la séquence d'instructions suivantes (dans laquelle X(J) est une constante) :

```
→ J ← J+1
  charger A(•,J) dans un registre vectoriel
  multiplier par X(J)
  charger Y(•)
  additionner
  Ranger Y(•) en mémoire
```

alors qu'on aurait pu avoir :

```
      charger Y(•) dans un registre vectoriel
┌──→  J ← J+1
│     charger A(•,J)
│     multiplier par X(J)
└──   additionner à Y(•)
      ranger Y(•)
```

(Il faut se rappeler qu'il n'y a qu'un seul chemin entre la mémoire et les registres).

Notons avant de continuer que la situation pourrait changer complétement :

- quand le compilateur évoluera sur le CRAY-1S;
- sur le CRAY-XMP, où 3 chemins d'accès sont disponibles entre la mémoire et les registres (2 en lecteure, 1 en écriture)

La machine actuelle (le CRAY-1S) étant ce qu'elle est, on peut accélerer le code (2) en tentant de suppléer aux insuffisances du compilateur : on essaie de profiter un peu plus de la présence des données dans les registres. Pour celà on "déroule" une partie de la boucle en J à l'intérieur de la boucle en I : on traite donc plusieurs valeurs de J en même temps (Y est initialisé à 0) :

(7)

```
      N4 = N/4
      DØ 1 J4 = 1,N4
         DØ 2 I = 1,M
            Y(i) =
            (((Y(i)
               + A(i,J4*4-3)* X (J4 *4-3))
               + A(i,J4*4-2)* X(J4 * 4-2))
               + A(i,J4*4-1)* X(J4 *4-1))
               + A(i,J4*4  )* X(J4 * 4  )
2     CØNTINUE
1     CØNTINUE
      DØ 3 J = N4 * 4+1,N
         DØ 4 i = 1,M
           Y(i)= Y(i) + A(i,J) * X(J)
4     CØNTINUE
3     CØNTINUE
```

(Le parenthésage permet le chainage des instructions de changement de registre, de multiplication et d'addition)-

Quelques mesures de temps (en nombre de périodes d'horloge)

	(2) produit scalaire	(3) SDOT	(5)	(6)	(7)	(7)' (*)
N=100 M=100	11.2×10^4 (14 M Flops)	7.14×10^4	5.4×10^4	6.7×10^4	2.39×10^4 (67 M Flops)	2.79×10^4
N=100 M=600	7.48×10^5 (13 M Flops)	5.28×10^5	2.78×10^5	2.4×10^5	1.32×10^5 (72 M Flops)	

(*) $7'$: avec u minimum de parenthèses !

2/ Récurrences linéaires d'ordre 1 - Réduction cyclique :

On appelle récurrence linéaire d'ordre 1 le calcul d'une suite x_i définie par :

$$\text{Rang (8)} \quad \begin{cases} x_1 = \text{donné} \\ x_i = a_{i-1} * x_{i-1} + b_{i-1} \qquad (1 < i \leq n) \end{cases}$$

où a_i et b_i sont donnés.

Le calcul de x_i dépend du résultat précédent x_{i-1} ; La boucle suivante qui implémente (8) de façon simple présente une dépendance arrière et n'est donc pas vectorisée par le compilateur CFT du CRAY-1 :

```
      DØ  1  I = 1,N
          X(I) = A(I) * X(I-1) + B(I)
    1 CØNTINUE
```

La réduction cyclique ramène le calcul de (8) à un problème de même structure mais portant sur une suite de longueur n/2, lui même réductible au calcul d'une suite de longueur n/4, etc...

Plus précisement écrivons deux équations successives de (8) :

$$x_i = a_{i-1} * x_{i-1} + b_{i-1}$$

$$x_{i+1} = a_i * x_i + b_i$$

On peut éliminer x_i entre ces deux équations :

$$x_{i+1} = a_i * (a_{i-1} * x_{i-1} + b_{i-1}) + b_i$$

On obtient une dépendance entre x_{i+1} et x_{i-1} de la même forme que (8) :

$$(9) \quad \begin{cases} x_{i+1} = a^{(1)}_{i-1} * x_{i-1} + b^{(1)}_{i-1} \\ \text{avec} \\ a^{(1)}_{i-1} = a_i * a_{i-1} \quad , \quad b^{(1)}_{i-1} = a_{i-1} * b_{i-1} + b_i \end{cases}$$

On peut réiterer ce processus d'élimination: la même procédure s'applique récursivement jusqu'à obtention d'un système à une seule équation.

Essayons de formaliser l'algorithme - Nous aurons besoin des notations suivantes : (Dans chacune de ces fonctions à valeur vectorielle, l'argument M représente le nombre d'éléments du vecteur résultat et $x \in \mathbb{R}^n$) :

(10)
$$\begin{cases} \text{. Pair } (x,2k,m) = \{x_{2k}, x_{2k+2}, \ldots, x_{2(k+m-1)}\} \in \mathbb{R}^m \\ \quad \text{(extraction à partir de l'indice 2k, de m éléments d'indice pair :} \\ \quad (x, 2k, m) \in \mathbb{R}^m \text{)} \\ \text{. Impair } (x,2k+1,m) = \{x_{2k+1}, x_{2k+3}, \ldots, x_{2(k+m)-1}\} \in \mathbb{R}^m \\ \text{. } \tau(x,m) = \{0, x_1, x_2, \ldots, x_{m-1}\} \in \mathbb{R}^m \\ \text{. } b = \{b_1, b_2, \ldots, b_{n-1}\} \in \mathbb{R}^{n-1}\} \\ b' = \{x_1, b_1, b_2, \ldots, b_{n-1}\} \end{cases}$$

On vérifie aisément les formules :

(11)
$$\begin{cases} \text{Pair } (\tau(x,m), 2, m') = \text{Impair } (x, 1, m') \\ \text{Impair } (\tau(x,m), 3, m') = \text{Pair } (x, 2, m') \end{cases}$$

Le problème qui est posé, à savoir (8), peut être réécrit au moyen des définitions ci-dessus [1] :

(12) $$\boxed{x = \tau(a,n) * \tau(x,n) + b'}$$

(noter que tous les vecteurs dans (12) sont bien de longueurs n)-

En applicant la fonction "pair" à l'équation (12) on obtient, en utilisant (11) :

(13) Pair (x, 2, n/2) = Impair (a, 1, $n/_2$) * Impair (x,1) , n/2) + Impair (b,1, n/2)

De même en applicant "Impair" :

(14) Impair (x,3, m)
= Pair (a, 2, m)* Pair (x, 2, m)
+ Pair (b,2,m)

---

(1) dans les opérations vectorielles qui vont suivre, "*" ou "+" désignent respectivement le produit, ou la somme, terme à terme .

avec :

$$m = \begin{cases} n/2-1 & \text{si } n \text{ est pair} \\ n/2 & \text{si } n \text{ est impair} \end{cases}$$

Donc en combinant (13) et (14) :

$$\text{(15)} \quad \left\lceil \begin{array}{l} \text{Impair } (x, 3, m) \\ \quad = \text{Pair } (a, 2, m) * \text{Impair } (a, 1, m) * \text{Impair } (x, 1, m) \\ \quad + \text{Pair } (a, 2, m) * \text{Impair } (b, 1, m) \\ \quad + \text{Pair } (b, 2, m) \end{array} \right.$$

Bien entendu ceci n'est pas autre chose que (9). En combinant tout cela on obtient la procédure récursive suivante :
(une procédure est dite récursive si elle s'appelle elle même; dans le "langage" ci-dessous cousin de l'ALGOL, l'appel à la procédure REC s'écrit simplement : REC (arguments, ...)).

(Résolution d'une récurrence linéaire du 1er ordre)

```
REC : PROCEDURE (X,A,B,N)
  X : VECTEUR (N); A,B : VECTEUR (N-1)
  SI N = 2 ALORS  X(2) ← X(1) * A(1) + B(1)
  SINON ALORS:
        SI N PAIR ALORS M ← N/2-1
        SINON     ALORS M ← N/2
        FIN SI

        REC (Impair (X,1,M+1),
             Pair (A,2,M) * Impair (A,1,M),
             Pair (A,2,M) * Impair (B,1,M)
             + Pair (B,2,M)
             )

        Pair (X,2,N/2) ← Impair (A,1,N/2)
                       * Impair (X,1,N/2)
                         + Impair (B,1,N/2)

    FIN SI
FIN REC
```

Exercice : Montrer par récurrence sur N que cette procédure se termine.

Ce programme ne peut être transcrit tel quel en FORTRAN sur le CRAY-1 :
Le FORTRAN n'accepte pas les procédures récursives ! si l'on veut transcrire cette procédure en FORTRAN on devra donc utiliser un mécanisme qui permettra de sauvegarder "l'environnement" du programme au niveau N avant de passer au niveau N/2 - un tel mécanisme s'appelle une "pile".
D'une manière générale une pile d'objets d'un certain type T [1] est une collection d'objets de ce type, sur laquelle on peut faire deux opérations :

- empiler : ajouter un objet au dessus de la pile;
- dépiler : prendre le dernier objet ajouté sur la pile si celle ci est non vide.

En supposant que l'on sache emplier des vecteurs, on obtient la version non récursive de la procédure REC :
(On notera qu'un compilateur acceptant des procédures récursives ferait à peu près la même chose).

---

(1) On peut vouloir empiler des vecteurs, des entiers, des adresses, des boites de conserves, etc... En anglais, pile se dit "stack".

```
RECITER (X,A,B,N);
    X : VECTEUR (N); A,B : VECTEUR (N-1); X',A',B', : VECTEURS
    TANT QUE N  2  REPETER
        EMPILER A,B,X,N
        SI N PAIR ALORS N'← N/2
        SINON     ALORS N'← N/2 + 1
        FIN SI
        A'← PAIR (A,2,N'-1)* Impair (A,1,N'-1);
        B'← PAIR (A,2,N'-1)* Impair (B,1,N'-1)
            + Pair (B,2,N'-1) ;
        X ← Impair (X,1,N') ;
        N ← N'; A ← A'; B ← B'
        FIN TANT QUE

   X(2) ← X(1) +  A(1) + B(1)

TANT QUE PILE PAS VIDE REPETER
        DEPILER A',B', X',N'
        Pair (X',2,N') ← Impair (A',1,N)* X
                       + Impair (B',1,N)
   X ← X'; A ← A'; B ← B'; N ← N'

        FIN TANT QUE

FIN RECITER
```

Comment implémenter la pile de vecteurs ?
On choisit dans le programme ci-dessous d'écrire les vecteurs empilés derrière les vecteurs A,B,X : on aura donc un vecteur de longueur N/2 (ou N/2 - 1) à la suite de A, puis un vecteur de longueur N/4, etc...
Si N est une puissance de 2, soit $N = 2^k$, le stockage supplémentaire est donc :

$$2^{k-1} + 2^{k-2} + \ldots\ldots + 1 = \frac{1 - 2^k}{-1} = N-1$$

En général on peut vérifier par récurrence que le stockage supplémentaire nécessaire pour stocker les vecteurs A est borné par N. Par contre le stockage supplémentaire pour les vecteurs X est majoré par 2xN.

L'empilage des vecteurs se traduit par une simple translation des adresses :
APRIM1 donne le début de A', etc.
Pour empiler N', on pourrait utiliser un vecteur d'entiers de longueur $\log_2 N+1$.
On peut aussi remarquer que dans la première étape, on empile soit N/2, soit N/2 +1:
il suffit on fait empiler la parité de N_1 c'est ce que l'on fait dans la variable entière
PILEN, qui constitue une pile d'entiers à elle toute seule !
Noter enfin l'usage obligatoire (sur CRAY) de la directive CDIR $ IVDEP

```
      SUBROUTINE  RECITER (X,A,B,N)
      DIMENSION X(1)
C     TABLEAU A DIMMENSIONNER A  3* N DANS LE PROGRAMME APPELANT
      DIMENSION A(1),B(2)
C     TABLEAUX DE DIMENSION 2* N
      INTEGER  N,  NPRIM, X1,X2,A1,A2
      INTEGER   APRIM1, APRIM2, XPRIM1, XPRIM2
      INTEGER   PILEN,ØRDRE
C     A1 = 0
      A2 = N-1
      X1 = 0
      X2 = N
C     VECTEUR A = A(A1 + 1), ..., A(A1+ N-1) = A (A2)
      ORDRE = N
      PILEN = 0
10    IF (ORDRE. EQ. 2) GØ TØ 30
      IF (MOD (ORDRE,2). EQ. 0)
         THEN
              NPRIM = ØRDRE/2
         ELSE
              NPRIM = ØRDRE /2+1
C     EMPILER X,A,B, ORDRE
      PILEN = PILEN  2 + MØD(ØRDRE,2)
      APRIM1 = A2+1
      APRIM2 = A2 + NPRIM-1
      XPRIM1 = X2+1
      XPRIM2 = X2 + NPRIM
      X(XPRIM1 + 1) = X(X1+1)
CDIR$ IVDEP
       DØ 20 i= 1,NPRIM-1
          A (i + APRIM1) = A (i * 2+A1) * A(i * 2-1 +A1)
          B (i + APRIM1) = B (i * 2+A1) * B(i * 2-1 +A1)+ B(i * 2+A1)
```

```
 20 CØNTINUE
    X1 = XPRIM1
    X2 = XPRIM2
    A1 = APRIM1
    A2 = APRIM2
    ORDRE = NPRIM
    GØ TØ10
C
C
30  X(X1+2)= A(A1+1) *X(X1+1) + B(A1+1)
C
C
40  IF (A1.EQ. 0) GØ TØ 60
C   DEPILER
    NPRIM = ORDRE * 2-MØD (PILEN,2)
    PILEN = PILEN/2
    XPRIM2= X1-1
    APRIM2= A2-1
    XPRIM1 = X1 - NPRIM
    APRIM1 = A1 - NPRIM + 1
CDIR$ IVDEP
    DØ  50  i = 1, ØRDRE
        X(i *2 + XPRIM1)= A(i *2-1 + APRIM1)* X(i + X1)
                        + B(i* 2-1 + APRIM1)
 50 CØNTINUE
    X1= XPRIM1
    X2 = XPRIM2
    A1 = APRIM1
    A2 = APRIM2
    ORDRE = NPRIME
    GØ TØ 40
 60 RETURN
    END
```

Exercice : Prouver que ce programme est correct.

Sur un ordinateur vectoriel, on voit l'une des limitations de la méthode : les vecteurs sont de plus en plus courts[1].

---

(1) Le nombre d'itérations est $\log_2 N$

En dessous d'un certain seuil [1] la vectorisation d'une boucle de longueur N n'apporte plus rien. On peut donc tester avant d'empiler de nouveaux vecteurs si ORDRE est bien supérieur à un seuil NO (à déterminer expérimentalement); sinon on sort des itérations empilantes, et l'on résoud ce qui reste en mode scalaire.

Enfin, il est moins fatigant, mais aussi moins instructif, d'utiliser la procédure de la bibliothèque du CRAY-1 qui fait la même chose :

```
CALL FOLR (N,A,1,X,1)
```

est équivalent à :

```
DØ  10 i = 2,N
10 x(i) = - A(i) * X(i-1) + X(i)
```

3/ Systèmes linéaires tridiagonaux

La résolution de systèmes dont la matrice est tridiagonale occupe une place fondamentale dans les outils de base pour la résolution de grands systèmes linéaires (par exemple méthodes de directions alternées). Malheureusement des algorithmes efficaces comme l'élimination de Gauss ou la factorisation LU sont séquentiels, et doivent donc être transformés (on abandonne....) pour utiliser au mieux un calculateur vectoriel ou parallèle.

On note $\{a_i\}$ $\{b_i\}$ $\{c_i\}$ les diagonales de la matrice

(8)
$$A = \begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{pmatrix}$$

On veut résoudre :

$$A \cdot x = k; \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad k = \begin{pmatrix} k_1 \\ \vdots \\ k_n \end{pmatrix}$$

---

(1) Inférieur à 10 sur CRAY-1

Les w_i, c_i, g_i ci-dessousdésignent les composantes de vecteurs auxilliaires.

3.1. Elimination de Gauss - Factorisation LU

. Factorisation : A = LU :

$$L = \begin{pmatrix} e_1 & & & \\ a_2 & c_2 & & 0 \\ & \ddots & \ddots & \\ 0 & & a_n & e_n \end{pmatrix} \qquad U = \begin{pmatrix} 1 & w_1 & & \\ & \ddots & \ddots & 0 \\ & & \ddots & w_{n-1} \\ 0 & & & 1 \end{pmatrix}$$

Les e_i et w_i sont définis par l'algorithme séquentiel suivant :

$$\begin{cases} c_1 = b_1 \\ w_1 = c_1 / e_1 \end{cases}$$

(16)

$$(i > 1) \quad \begin{cases} e_i = b_i - a_i \; w_{i-1} \\ w_i = c_i / e_i \end{cases}$$

. Résolution : descente (Lg = k)

$$(17) \quad \begin{cases} g_1 = k_1 / e_1 \\ (i > 1) \quad g_i = (k_i - a_i \; g_{i-1}) / e_i \end{cases}$$

. Résolution : remontée (U. x = g)

$$(18) \quad \begin{cases} x_n = g_n \\ (i < n) \quad x_i = g_i - w_i \; x_{i+1} \end{cases}$$

Les boucles correspondant à (16),(17,(18) présentent des dépendances arrières et ne sont en aucun cas vectorisables : le calcul à une étape i dépend du résultat de l'étape précédente.

Remarque :
Dans le cas où il s'agit de résoudre simultanément m systèmes tridiagonaux de la forme (8) (et de taille toujours égale à n), on obtient une parallèlisation (vectorisation) évidente en remplaçant toutes les opérations scalaires dans (16),(17),(18) par des opérations vectorielles de longueur m.

3.2. "Recursive doubling" (Stone, 1975).

Les boucles (17) et (18) sont des récurrences linéaires du premier ordre, dont une écriture "vectorielle" peut être obtenue par la technique de réduction cyclique décrite précédemment.
Par contre, le calcul des e_i et w_i par (16) est non linéaire: la technique ci-dessus ne peut s'appliquer du moins pas directement.
Réecrivons (16) en fonction de w_i seulement :

(19) $w_1 = c_1 / b_1$

$(i>1) \quad w_i = c_i / (b_i - a_i \; w_{i-1})$

Introduisons une variable auxilliaire y_i telle que :

(20) $w_i = - y_i / y_{i+1}$

En reportant dans (19) il vient la relation linéaire :

(21) $(i > 1) \quad a_i \; y_{i-1} + b_i \; y_i + c_i \; y_{i+1} = 0$

En prenant $y_1 = 1$, les autres termes de la suite y_i sont déterminés par :

$$y_2 = -1/w_1 = - b_1 / c_1$$

et par (21) (récurrence linéaire du second ordre).

On se ramène à une récurrence linéaire du premier ordre en posant :

$$v_i = \begin{pmatrix} y_i \\ y_{i+1} \end{pmatrix} \qquad Q_i = \begin{pmatrix} 0 & 1 \\ -\frac{a_i}{c_i} & -\frac{b_i}{c_i} \end{pmatrix}$$

On a alors

$$v_i = Q_i \cdot v_{i-1},$$

et la technique de réduction cyclique s'applique.

Remarques : L'algorithme échoue si l'un des coefficients c_i s'annule. Dans ce cas le problème (8) peut être découplé en deux problèmes d'ordre inférieur.

3.3. Réduction cyclique

Le principe de la réduction cyclique s'applique également à la résolution des systèmes linéaires tridiagonaux.

Ecrivons 3 équations du système à résoudre :

$$a_{i-1}\ x_{i-2} + b_{i-1}\ x_{i-1} + c_{i-1}\ x_i = k_{i-1}$$

$$a_i\ x_{i-1} + b_i\ x_i + c_i\ x_{i+1} = k_i$$

$$a_{i+1}\ x_i + b_{i+1}\ x_{i+1} + c_{i+1}\ x_{i+2} = k_{i+1}$$

Eliminons x_{i-1} et x_{i+1} entre ces 3 équations : on obtient un système tridiagonal d'ordre moitié :

$$a_i^{(1)}\ x_{i-2} + b_i^{(1)}\ x_i + c_i^{(1)}\ x_{i+2} = k_i^{(1)}$$

avec :

$$\begin{cases} a_i^{(1)} = -\ a_i\ a_{i-1} / b_{i-1} \\ b_i^{(1)} = b_i - a_i\ c_{i-1} / b_{i-1} - c_{i+1}\ a_{i+1} / b_{i+1} \\ c_i^{(1)} = -\ c_{i+1}\ c_i / b_{i+1} \\ k_i^{(1)} = k_i - a_i\ k_{i-1} / b_{i-1} - c_{i+1}\ k_{i+1} / b_{i+1} \end{cases}$$

On applique récursivement le même processus. Comme au chapitre 2, l'implémentation de cette technique utilise une pile. Le shéma des itérations est le suivant (en supposant que N + 1 est une puissance de 2 pour simplifier : $N = 2^q - 1$

```
P ← 1
(22) TANT QUE p<q REPETER
         Calculer a(p), b(p), c(p), k(p)
         Empiler  a(p), b(p), c(p), k(p)
         p ← p+1
     FIN TANT QUE

Résoudre le"système" d'ordre 1

     TANT QUE  p>1 REPETER
         p ← p+1
(23)     Dépiler a(p), b(p), c(p), k(p)
         calculer x(2^p * i - 2^(p-1)), i>1
     FIN TANT QUE
```

L'implémentation est laissée en exercice. (Utiliser les techniques du paragraphe 3).

Les opérations ci-dessus contiennent des opérations vectorielles: on peut vérifier par exemple qu'une même itération (22) comporte 10 opérations vectorielles de longueur 2^{q-p+1}, et qu'une itération (23) en comporte 4. En totalisant le nombre d'opérations effectuées, on trouve que cette méthode effectue un nombre d'opérations flottantes scalaires de l'ordre de : 14 x N. Ceci est supérieur au nombre d'opérations effectuées dans la méthode de Gauss classique (qui varie comme 6 x N). Ce phénomène(1) est courant dans les algorithmes parallèles : la réorganisation de l'algorithme entraine un nombre plus grand d'opérations.
Ceci à deux conséquences :
- sur un calculateur séquentiel la version parallèle sera moins performante que la version séquentielle;
- faisant plus d'opérations, on a plus de chances d'accumuler des erreurs d'arrondi.

4/ L'algorithme de Gauss Seidel : la méthode des fronts (Lamport 19..)

Considérons la résolution du Laplacieu sur un carré, discrétisé par différences finies; on utilise la méthode de Gauss Seidel, et on obtient donc un programme du type suivant (2):

```
      do 10  i = 1, nx
         do 20  j = 1, ny
S              x(i,j) = (x(i-1,j)+ x(i+1,j)+ x(i,j-1)+ x(i,j+1) .14
20       CONTINUE
10    CONTINUE
```

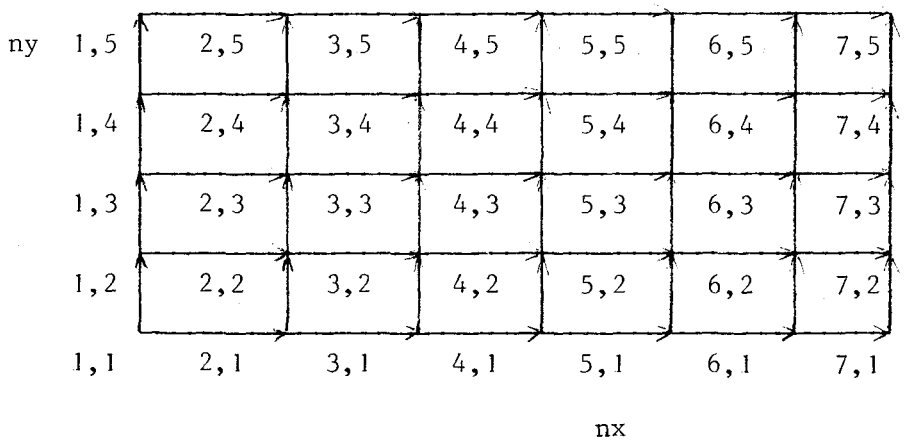
La boucle la plus interne présente une dépendance arrière qui inhibe la vectorisation : il en est de même si l'on permute les boucles en i et en j : la situation parait sans espoir !
Traçons le graphe de dépendances associé à ce programme : chaque noeud du graphe est associé à une instruction S(i,j) pour un couple de valeur i et j.

---

(1) Lié au fait qu'une équation est utilisée un plus grand nombre de fois.

(2) On suppose que les conditions aux limites sont données par x/ 0 .) etc...

On trace une flèche de $\{i,j\}$ vers $\{k,l\}$ si l'exécution de S_{kl} dépend de l'exécution de S_{ij} - on obtient le graphe suivant :



on s'aperçoit en examinant le graphe que certaines opérations pourraient s'effectuer en parallèle : il s'agit par exemple de $S_{3,1}$, $S_{2,2}$, $S_{1,3}$, ou plus généralement des S_{ij} avec i+j = constante.

Posons donc l= i+j : la plus petite valeur de l est obtenue pour i = j = 1, soit l = 2i

La plus grande valeur est obtenue pour i = nx, j =ny

On obtient le programme suivant :

```
      do  10 l = 2, nx + ny
          if (nx. le. ny)
          then
             if (l-1 . le . nx)
                then
                   j1 = 1
                   j2 = l-1
                else
                   j1 = l - nx
                   j2 = max 0 (l-1,ny)
              end if
           else
              if (l-1 . le . ny)
                then
                   j1 = 1
                   j2 = l-1
                else
                   j1 = l-max 0(l-1,nx)
                   j2 = ny
              end if
```

```
          do  20 j = j1,j2
              x (1-j,j)=
              (x(1-j-1,j) + x(1-j+1,j)
              + x(1-j,j-1) + x(1-j,j+1) / 4
20        CONTINUE
10      CONTINUE
```

La boucle 20 ci-dessus n'est pas reconnue comme vectorielle par la version actuelle du compilateur CFT, puisque l'indice de boucle j est présent dans les indices de deux dimensions différentes.

Pour obtenir une boucle vectorielle, il faut donc remplacer x(i,j) par un tableau équivalent à une seule dimension (de longueur ny * m);
On remplace alors la boucle 20 par :

```
        CDIR$  IVDEP
            do  20 j = j1,j2
                TX ((m-1) * j + 1- m)
                =
                (TX ((m-1)* j + 1- m- 1)
                + TX ((m-1) * j + 1- m + 1)
                + TX ((m-1) * j + 1- m * 2)
                + TX ((m-1) * j + 1       ) / 4
 20  CØNTINUE
```

La valeur de m doit être supérieure ou égale à nx+2 et prise de façon à éviter les conflits de bancs.
La boucle 20 est alors vectorisée; cependant la longueur de cette boucle est très courte(1,2,...) au début et à la fin de l'algorithme.

Références

Alain BOSSAVIT
"Comment j'ai vectorisé certains de mes programmes",
EDF/IMA, HI 4271-00, Septembre 1982

H.S. STONE
"Parallel tridiagonal solvers",
ACM transactions on mathematical softwave,
1, pp 289 - 307 (1975)

R.W. HOCKNEY & C.R. JESSHOPE
"Parallels computers", Adam Hilger, Bristol, 1983

J. ERHEL
"Parallélisation d'algorithmes numériques"
thèse de 3ème cycle, Paris 6, 11 Mars 1982.

C H A P I T R E VI

RÉSOLUTION DE SYSTÈMES TRIDIAGONAUX : RÉDUCTION "PAIR-IMPAIR".

Résolution de systèmes tridiagonaux : réduction "pair-impair"

Soit le système linéaire :

$$\begin{vmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & 0 \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ & 0 & & a_{n-1} & b_{n-1} & c_{n-1} \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{vmatrix} = \begin{vmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{vmatrix}$$

La première équation s'écrit :

$$(1) \quad a_i\, x_{i-1} + b_i\, x_i + c_i\, x_{i+1} = k_i$$

Utilisons les (i-1)ème et (i+1)ème équations pour éliminer x_{i-1} et x_{i+1} de la ième équation j on obtient une relation de la même forme, cette fois entre x_{i-2}, x_i, x_{i+2} :

$$(2) \quad a_i^{(1)}\, x_{i-2} + b_i^{(1)}\, x_i + c_i^{(1)}\, x_{i+2} = k_i$$

avec :

$$(3) \quad \begin{cases} a_i^{(1)} = - a_i \times (1/b_{i-1})\, a_{i-1} \\ c_i^{(1)} = - c_i\, (1/b_{i+1})\, c_{i+1} \\ b_i^{(1)} = b_i - a_i\, (1/b_{i-1})\, c_{i-1} - c_i\, (1/b_{i+1})\, a_{i+1} \end{cases}$$

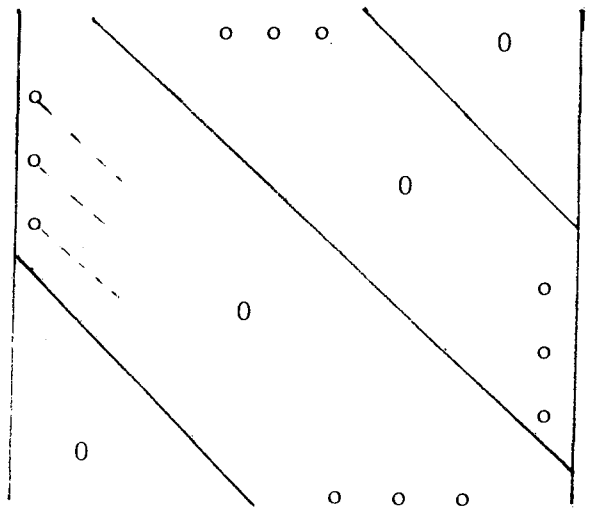
A la différence de la réduction cyclique on effectue cette transformation sur <u>toutes</u> les lignes, aussi bien les lignes paires que les lignes impaires.

Les équations que l'on obtient peuvent s'écrire, sous forme matricielle :

$$\begin{bmatrix} b_1^{(1)} & o & c_1^{(1)} & & & & & \\ o & b_2^{(1)} & o & c_2^{(1)} & & & & \\ a_3^{(1)} & o & b_3^{(1)} & o & c_3^{(1)} & & & \\ o & a_4^{(1)} & o & b_4^{(1)} & o & c_4^{(1)} & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & o & b_{n-1}^{(1)} & o & c_{n-2}^{(1)} & \\ & & & & a_n^{(1)} & o & b_n^{(1)} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} k_1^{(1)} \\ k_2^{(1)} \\ \vdots \\ k_n^{(1)} \end{bmatrix}$$

Ainsi on a encore une matrice contenant 3 diagonales non nulles ; mais les éléments non nuls hors de la diagonale principale ont été repoussés d'une position. On peut [1] réappliquer le même procédé, et éliminer x_{i-2} et x_{i+2} de la ième équation.

La matrice est de la forme :



A l'étape suivante on élimine x_{i-4}, x_{i+4}, de la ième équation et les éléments non nuls sur une ligne sont maintenant séparés par 8 portions, si l'on répète le processus, le nombre de colonnes séparant les coefficients non nuls d'une ligne double à chaque étape. Donc en au plus $\log_2 n$ étapes, la matrice est réduite à sa diagonale principale.

---

(1) on étudiera plus loin les conditions de stabilité.

Pour mettre en évidence les opérations vectorielles, on utilisera les notations suivantes ($x \in R^n$) :

$$s\,x = \{o, x_1, x_2, \ldots, x_{n-1}\} \in R^n$$
$$s^{-1}x = \{x_2, x_3, x_4, \ldots, x_n, o\} \in R^n$$

(on notera que s n'est pas bijective et que s^{-1} n'est pas réellement "l'inverse" de s).

$$s^2_x = s \circ s\,(x) = \{o, o, x_1, x_2, \ldots, x_{n-1}\}$$
$$s \circ s^{-1}\,(x) = \{o, x_2, x_3, \ldots, x_n\}$$
$$s^{-1} \circ s\,(x) = \{x_1, x_2 \ldots, x_{n-1}, o\}$$

alors les relations (1) peuvent s'écrire :

(4) $a \star sx + b \star x + x \star s^{-1} x = k$

(en prolongeant a et c à des vecteurs de $\mathbb{R}^n$ par des valeurs arbitraires,par exemple $a_1 = c_n = o$).

En appliquant s à l'équation précédente, on obtient (remarquer que $sc \star s\,(s^{-1} x) = sc \star x$) :

$sa \star s^2 x + sb \star sx + sc \star x = sk$

De même en appliquant s^{-1} :

$s^{-1}_a \star x + s^{-1} b \star s^{-1} x + s^{-1} c \star s^{-2} x = s^{-1} k$;

d'où après élimination, une version vectorielle de (2) :

(5) $a^{(1)} \star s^2 x + b^{(1)} \star x + c^{(1)} \star s^{-2} x = k^{(1)}$

avec

$$(6) \quad \begin{cases} a^{(1)} = - a \star s\,(1/b) \star sa \\ b^{(1)} = b - s(1/b) \star sc \; - s^{-1}\,(1/b) \star s^{-1} a \\ c^{(1)} = - c \star s^{-1}\,(1/b) \star s^{-1} c \end{cases}$$

A l'étape p, on part d'une relation entre $s^{2p}x$, x, $s^{-2p}x$:

(7) $a^{(p)} \star s^{2^p} x + b^{(p)} \star x + c^{(p)} \star s^{-2^p} x = k^{(p)}$

et l'on aboutit à une relation entre $s^{2^{p+1}} x$, x, $s^{-2^{p+1}} x$:

(8) $a^{(p+1)}\, s^{2^{p+1}} x + b^{(p+1)} x + c^{(p+1)}\, s^{-2^{p+1}} x = k^{(p+1)}$:

$$(9)\qquad \left|\begin{array}{l} a^{(p+1)} = - a^{(p)} \ast s^{2^p} (1/b^{(p)}) \ast s^{2^p} a^{(p)} \\ b^{(p+1)} = b^{(p)} - a^{(p)} \ast s^{2^p} (1/b^{(p)}) \ast s^{2^p} c^{(p)} \\ \qquad\qquad - c^{(p)} \ast s^{-2^p} (1/b^{(p)}) \ast s^{-2^p} a^{(p)} \\ c^{(p+1)} = \qquad - c^{(p)} \ast s^{-2^p} (1/b^{(p)}) + s^{-2p} c^{(p)} \\ k^{(p+1)} = k^{(p)} - a^{(p)} \ast s^{2^p} (1/b^{(p)}) \ast s^{2^p} k^{(p)} \\ \qquad\qquad - c^{(p)} \ast s^{-2^p} (1/b^{(p)}) \ast s^{-2^p} k^{(p)} \end{array}\right.$$

Comme $s^m x$ possède $n - m$ coefficients non nuls, on a
$s^{2^{p+1}} x = s^{-2^{p+1}} x = o$ dès que $2^{p+1} \geq n$.

Donc à l'étape $p = \log_2 n - 1$ on obtient un système à matrice diagonale :

$$(10)\qquad b^{(P+1)} \ast x = k^{(p+1)}$$

Chaque étape p comporte 14 opérations vectorielles de longueur $n - 2^p$;
finalement la résolution de (10) est une division vectorielle de longueur n.

Examinons des conditions suffisantes qui permettront à l'algorithme ci-dessus de fonctionner.

On suppose que les coefficients du système à résoudre satisfont :

$$(11)\qquad \left|\begin{array}{l} b_i > o, \quad a_i \leq o, \quad c_i \leq o \qquad \forall\, i \\ b_i \geq |a_i| + |c_i| \\ |a_i| / b_i \leq \alpha < 1 \quad |c_i| / b_i \leq \gamma < 1 \\ \alpha > o, \gamma > o \end{array}\right.$$

En revenant aux équations (3), on vérifie :

$$o \geq a_i^{(1)} \geq + \alpha\, a_i, , \qquad o \geq c_i^{(1)} \geq \gamma\, c_i$$

$$b_i^{(1)} \geq b_i - \alpha |a_i| - \gamma |c_i| \geq (1 - \mathrm{Max}\,(\alpha,\gamma))\, b_i$$

d'où :

$$\left|\begin{array}{l} b_i^{(1)} > o, \; a_i^{(1)} \leq o, \; c_i^{(1)} \leq o \\ |a_i^{(1)}| / b_i^{(1)} \leq \dfrac{\alpha}{1 - \mathrm{Max}\,(\alpha,\gamma)} |a_i| / b_i \\ |c_i^{(1)}| / b_i^{(1)} \leq \dfrac{\gamma}{1 - \mathrm{Max}\,(\alpha,\gamma)} |c_i| / b_i \end{array}\right.$$

Si $\mathrm{Max}\,(\alpha, \gamma) \leq 1/2$,

$$\text{Alors} \left(\begin{array}{l} |a_i^{(1)}| / b_i^{(1)} \leq \alpha \\ b_i^{(1)} \geq |a_i^{(1)}| + |c_i^{(1)}| \end{array}\right) \qquad |c_i^{(1)}| / b_i^{(1)} \leq \gamma$$

Donc si $\alpha \leq 1/2$ et $\gamma \leq 1/2$, le calcul peut être poursuivi jusqu'au bout ; en outre dans ces conditions,

$$\left|\frac{a_i^{(1)}}{b_i^{(1)}}\right| \leq \frac{|a_i|}{b_i}$$

$$|a_i^{(p)}| \leq 2^{-p}\,|a_i|$$

Si le système (1) est à dominance diagonale, les systèmes (2) et (8) sont encore à dominance diagonale et le calcul est stable.

Pour démontrer cette propriété on utilise les équations (3) avec les notations :

$$a'_i = |a_i| \,/\, b_i \,,\quad c'_i = |c_i| \,/\, b_i$$

Par l'hypothèse de dominance diagonale :

$$o \leq a'_i + c'_i \leq 1,\quad b_i > o$$

et d'autre part, d'après (3) :

$$|a_i^{(1)}| = |a_i|\; a'_{i-1},\quad |c_i^{(1)}| = c_i\; c'_{i+1}$$

$$b_i^{(1)} \geq b_i - |a_i|\; c'_{i-1} - |c_i|\; a'_{i+1}$$

$$b_i^{(1)} \geq b_i\,(1 - a'_i\, c'_{i-1} - c'_i\, a'_{i+1}) \tag{12}$$

Enfin :

$$a'_{i-1} + c'_{i-1} \leq 1 \tag{13}$$

$$a'_{i+1} + c'_{i+1} \leq 1 \tag{14}$$

En multipliant la première équation (13) par a'_i et la deuxième (14) par c'_i on obtient :

$$(a'_i\, c'_{i-1} + c'_i\, a'_{i+1}) + (a'_i\, a'_{i-1} + c'_i\, c'_{i+1}) \leq 1 \tag{15}$$

En reportant dans (12) on obtient

$$b_1^{(1)} \geq b_i$$

Finalement :

$$\frac{|a_i^{(1)}| + |c_i^{(1)}|}{b_i^{(1)}} \leq \frac{|a_i^{(1)}| + |c_i^{(1)}|}{b_i} = a'_i\, a'_{i-1} + c'_i\, c'_{i+1} \leq 1$$

CHAPITRE VII

UN CALCULATEUR PARALLÈLE MIMD : LE HEP

1/ Introduction

Le HEP[1] est un calculateur parallèle construit et commercialisé par la société DENELCOR (Denver,Colorado); sa conception comporte quelques caractères originaux.

Le parallèlisme est obtenu à deux niveaux :

- la machine se compose d'un ensemble de processeurs[2], de 1 à 16, partageant des données en mémoire; un réseau d'interconnexion (switch) s'interpose entre ces processeurs et la mémoire.
- Chaque processeur permet l'exécution simultanée de plusieurs processus (process) pour le compte d'un même utilisateur; un tel parallélisme est obtenu en pipelinant non seulement les unités fonctionnelles de calcul, mais aussi les unités de contrôle du processeur.

Les processus coopèrent en partageant des données dans la mémoire.
(on peut définir un processus comme une séquence d'instructions.)

Les synchronisations s'effectuent grâce à un dispositif original de la mémoire : à chaque mot dans la mémoire est associé un bit signifiant "mot plein" ou "mot vide", représentant l'état actuel du mot en question. Le matériel permet d'une manière indivisible :

- de tester l'état d'un mot, de lire son contenu s'il est "plein" et de mettre son état à "vide";
- de tester l'état d'un mot, d'y écrire une valeur, et de mettre son état à "plein".
- d'autres combinaisons sont possibles et seront détaillées dans ce chapître

Le programmeur scientifique a la possibilité d'écrire des programmes parallèles en FORTRAN 77, grâce à quelques extensions au langage, principalement les suivantes :

- l'ordre CREATE crée un nouveau processus;
- les variables dites "asynchrones" : elles sont reconnues par le compilateur par le premier caractère qui doit être un "$" : seules les variables asynchrones conduisent aux tests de l'état des mots correspondants.

---

(1) Heterogeneous Element Processor = HEP

(2) PEM = Process Execution module

Cette organisation apparente le HEP à la famille des calculateurs MIMD.
La performance obtenue, assez modeste, est certainement liée au choix de la technologie utilisée dans les modèles actuels, qui n'est pas la plus rapide: on peut obtenir 10 MIPS (Millions d'Instructions par Seconde) avec un processeur d'où un maximum de 160 MIPS avec 16 processeurs. Il faut remarquer que les utilisateurs en calcul scientifique ont l'habitude de compter en MFLOPS (Millions d'opérations flottantes par seconde) : pour beaucoup d'applications scientifiques un "MIPS" correspond à une fraction de "MFLOPS"; si le programme est en FORTRAN, ce rapport dépend évidemment de la qualité du compilateur.
Finalement le HEP permet au programmeur FORTRAN d'exprimer des processus parallèles, moyennant un coût[1] de gestion (overhead) minime.
Par rapport à un calculateur vectoriel :
- le HEP permet d'obtenir les opérations parallèles là où les calculateurs vectoriels sont impuissants ou inefficaces (accès aléatoire à la mémoire, exécutions asynchrones,...)
- dans les applications contenant un grand nombre d'opérations vectorielles, le HEP aura sans doute du mal à concurrencer les calculateurs vectoriels même à technologie égale; en effet sur le calculateur vectoriels, les accès mémoires sont alors accélerés; d'autre part à une instruction vectorielle correspondent sur le HEP une boucle réalisée par plusieurs instructions.

---

(1) Il ne s'agit pas ici du coût au sens financier.

2/ Principe des opérations sur un processeur du HEP

2.1. calculateur SISD

Pour comprendre le fonctionnement d'un processeur sur le HEP, reprenons le fonctionnement d'un calculateur séquentiel "classique" du type SISD : le processeur éxécute à un instant donné un seul processus, et lance les instructions les unes après les autres. Dans le cas le plus simple il faut attendre la fin d'une instruction avant d'entamer la suivante.

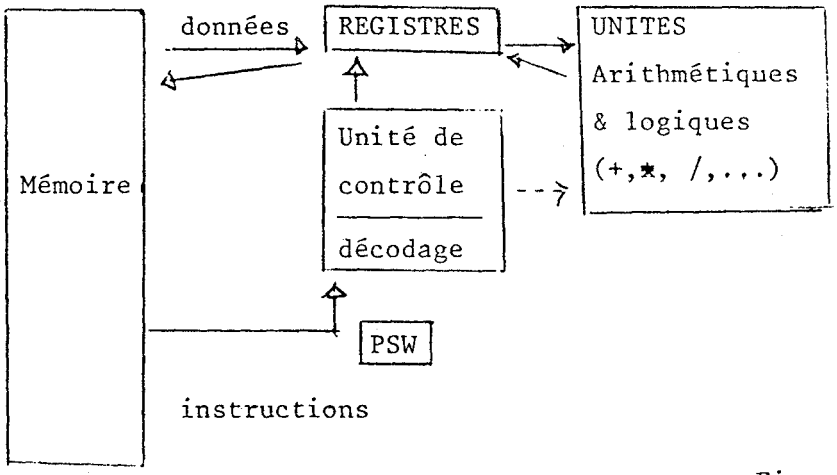


Figure 1 : shéma d'un calculateur SISD

les informations contenant l'état du processus et permettant de poursuivre le calcul, en par ticulier d'accéder à l'instruction suivante, sont concentrées dans un mot, ou un registre, spécial, désigné souvent sous le nom de PSW (Program Status Word).

2.2. La queue d'éxécution

Les concepteurs d'ordinateurs ont reconnu depuis longtemps la possibilité de pipeliner l'unité de contrôle (décodage), et de faire recouvrir dans le temps les opérations de décodage, d'accès aux données et de calcul pour plusieurs instructions : l'une des caractèristiques du HEP est d'associer ces instructions à différents processus.
Pour décrire de façon imagée le fonctionnement d'un processeur HEP, on peut imaginer que les différents PSW, qui décrivent les différents processus, sont placés sur une "roue", tournant à une vitesse constante (figure2)

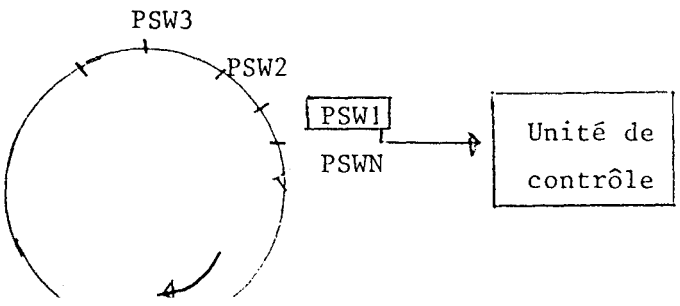


Figure 2

Quand le PSW décrivant l'état d'un processus "passe" devant l'unité de contrôle le processus à la possibilité de lancer une nouvelle instruction dans la mesure où les opérandes en sont disponibles. La "roue" tourne suffisamment lentement pour permettre que cette instruction soit terminée au moment de lancer la suivante pour le même processus.
Une autre façon de voir consiste à décrire les PSW associés aux différents processus comme inscrits dans une file d'attente, ou queue d'exécution[1] (figure 3) qui avance d'une position toutes les 100ns. Il existe une autre file d'attente contenant les PSW des processus attendant le résultat d'une requête[2]
Avant d'expliquer son fonctionnement il convient de décrire brièvement l'organisation de la mémoire..

2.3. La mémoire

Il en existe 4 sortes sur un système HEP :

- la mémoire de données, partagée entre les processus accessibles à travers le réseau d'interconnexion.
- Les autres mémoires sont réservées à chaque processeur:
- les registres : 2048 mots de 64 bits (par processeur)
- les mémoires de constantes : 4096 mots de 64 bits (par processeur)

accessibles seulement en lecture pour les programmes d'utilisateur; accessibles en écriture à travers des instructions privilégiées utilisées par le système.

- Les mémoires de programmes : de 128×10^3 mots à 10^6 mots (par processeur).

Ainsi qu'il a été exposé dans l'introduction, chaque mot dans la mémoire de données est affecté d'un bit supplémentaire dont les valeurs représentent l'état "plein" ou "vide". Chauqe registre peut de même être dans l'état "plein" ou "vide"; il peut également se trouver dans l'état "réservé".
La propriété essentielle de la mémoire est la suivante :

> Le test de l'état d'un mot, l'obtention du contenu de ce mot et le changement de l'état du mot s'effectue en une seule opération indivisible.

Sans ce caractère indivisible, la notion d'état d'un mot en mémoire serait vide de sens.

---

(1) "Control loop"

(2) "SFU queue", queue mémoire

2.4. Les unités fonctionnelles - la SFU

Chacune des unités fonctionnelles est segmentée[1] (pipelinée) ainsi que l'unité de contrôle; chaque segment durant 100 ns, ces unités peuvent accepter une nouvelle instruction toutes les 100 ns.

L'une des unités fonctionnelles, également pipelinée, est chargée de mouvements de données depuis, et vers, la mémoire de données : il s'agit de la SFU (Sheduler Function Unit).

Une opération de lecture ou d'écriture dans la mémoire de onnées prend un temps indéterminé[2] pour plusieurs raisons :
- la requête, qui passe à travers le réseau d'interconnexion, peut se trouver en conflit avec les requêtes émises par les autres processus,(le fonctionnement du réseau sera décrit brièvement au 3) qui essayent d'emprunter le même chemin;
- le mot mémoire peut ne pas être prêt : il peut être dans l'état "vide" dans une opération de lecture , ou dans l'état "plein" dans une opération d'écriture.

Aussi lorsqu'une opération de lecture ou d'écriture en mémoire se présente, la SFU :
- éxpédie la requête avec les renseignements identifiant le processeur et le processus émetteurs ;
- retire de la queue d'éxécution le PSW associé au processus émetteur de la requête.

Quand revient la réponse de la mémoire de données, et si l'état du mot concerné était satisfaisant (vide ou plein suivant le cas), la donnée est éventuellement copiée dans un registre et le PSW est réinséré dans la queue d'éxécution.

La queue d'exécution avance d'une position toutes les 100 ns : un nouveau PSW est alors considéré par l'unité de contrôle. Quand le résultat d'une instruction doit se trouver dans un registre, le registre résultat passe dans l'état "réservé" quand l'instruction démarre.

Mais une instruction ne peut s'exécuter tant que l'un des registres opérandes est dans l'état "réservé", puisque ceci signifie que les opérandes ne sont pas encore prêts; si tel est le cas, le PSW correspondant est inséré à la fin de la queue d'exécution, sans incrémenter le compteur d'instructions.

---

(1) sauf les unités de division

(2) mais que l'on peut en général majorer

Il en est de même si l'état (vide, plein, ou réservé) d'un registre opérande ou résultat n'est pas satisfaisant.

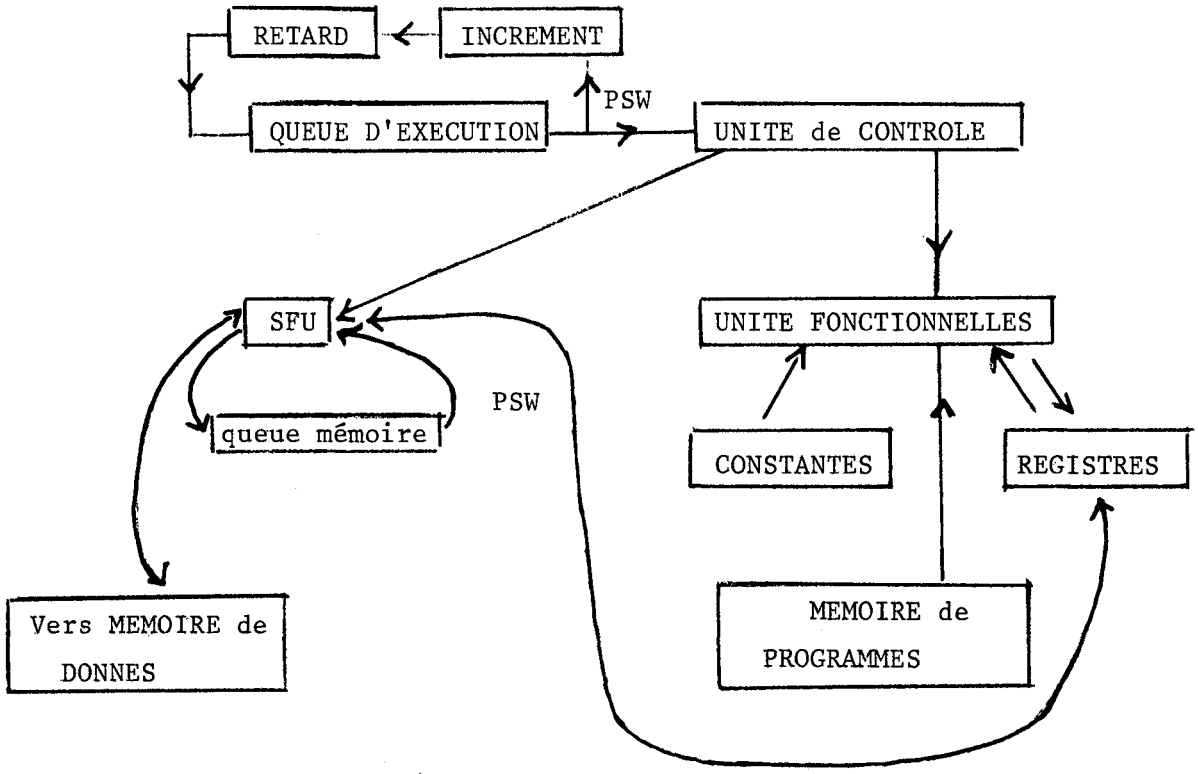


Figure 3:
Shéma simplifié d'un processeur

3/ Principes de fonctionnement du réseau d'interconnexion

Le réseau est dispositif matériel permettant d'acheminer les requêtes mémoire entre les processeurs et les bancs de mémoire, et vice et versa. Il est constitué d'un nombre(non figé) de noeuds : la figure 4 donne un exemple du réseau utilisé pour connecter 4 processeurs à 4 bancs mémoire.

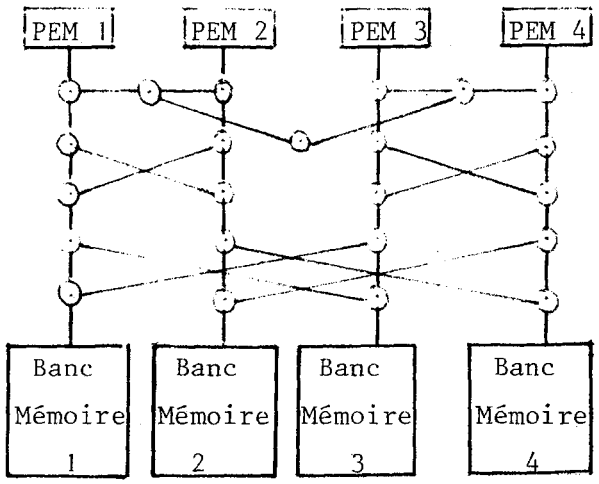


Figure 4 :
Shéma simplifié d'un système HEP
(on a volontairement oublié les disques et entrées sorties)

Chaque noeud possède 3 entrées et 3 sorties(figure 5) ,

Figure 5 : un noeud

et peut donc être connecté à 3 voisins dans les deux sens.
Le réseau fonctionne de façon synchrone : toutes les 100 ns, chaque noeud reçoit entre 0 et 3 messages et les retransmet aussitôt. Il existe plusieurs chemins vers une destination donnée dans le réseau : chaque noeud connait, en fonction de la destination d'un message, la porte de sortie correspondant au chemin le plus court; cette "connaissance" est inscrite dans des tables précalculées et présentes dans chaque noeud. Il peut arriver que plusieurs messages cherchent à sortir par la même porte à un instant donné, afin de résoudre ces conflits, chaque messages est affecté d'une priorité initialisée à 1 à l'entrée dans le réseau. Quand plusieurs messages sont en conflits sur une porte dans un noeud, un seul prend la route optimale compte tenu de sa destination : c'est le message porteur de la plus forte priorité. Les autres messages sont déroutés vers les autres portes du noeud et voient leur priorité augmentée de 1.

Il peut arriver avec ce mécanisme qu'un processeur ou un banc de mémoire reçoive un message qui ne lui est pas destiné : il est alors obligé de le renvoyer dans le réseau avant d'introduire ses propres messages.

4/ Programmation en FORTRAN - HEP

Deux sortes d'extensions ont été introduites dans le langage FORTRAN-sur le HEP : les ordres de lancement de processus (CREATE,RESUME) et les variables asynchrones (**$**)

4.1. Les ordres de lancement de processus

L'ordre CREATE a une syntaxe similaire à celle du CALL :

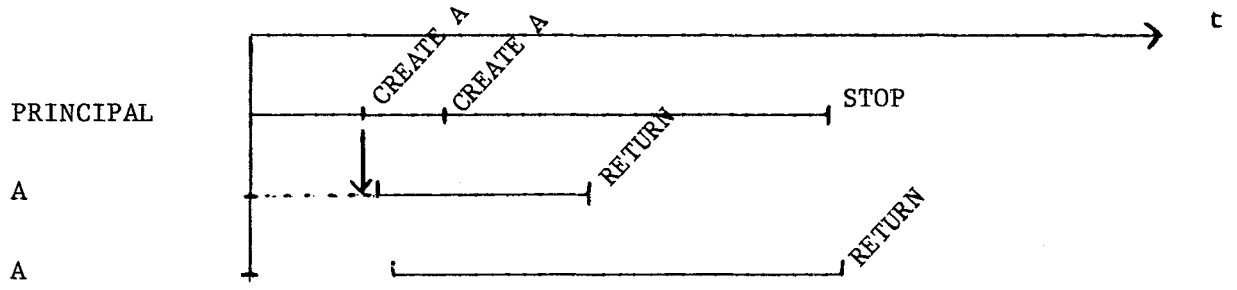
CREATE nom de sous programme (paramètres)

L'effet produit par cette instruction est de créer un nouveau processus pour le compte de l'utilisateur, constitué par les instructions du sous programme, et qui s'exécute en parallèle avec le programme appelant, et qui se termine en rencontrant l'ordre RETURN,

EXEMPLE :

```
PROGRAMME PRINCIPAL
-------
CREATE   A(TAB,N) ---> SUBROUTINE  A(X,N)
CREATE   A(TAB1,N)     DIMENSION   X(N)
--------               X(1) = ....
                       ----
STOP                   RETURN
END                    END
```

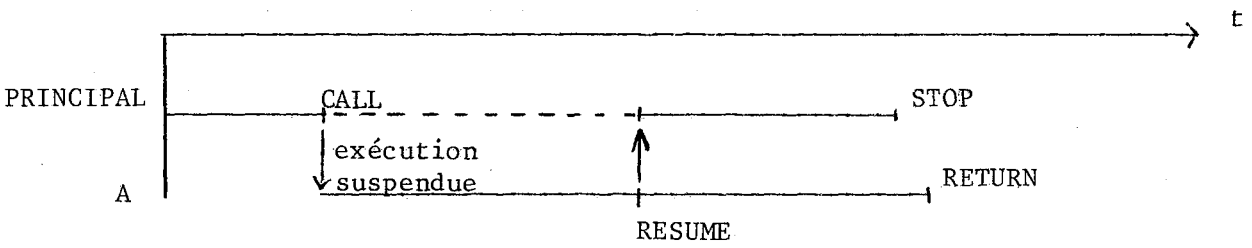
Dans cet exemple, 2 processus seront crées, selon le diagramme des temps qui suit :



Si un sous programme est appelé par un ordre CALL, l'éxécution du programme appelant est suspendue, ainsi qu'il est d'usage sur tout calculateur. Cependant, si le sous programme exécute un ordre RESUME, un autre processus est créé et l'éxécution du programme appelant continue en parallèle avec celle du sous-programme.

EXEMPLE :

```
PROGRAMME PRINCIPAL          SUBROUTINE  A(...)
-------                      -------
CALL  A(...)                 RESUME
-------                      -------
STOP                         RETURN
END                          END
```



Remarque: un ordre RESUME est ignoré dans un sous programme invoqué par CREATE.

4.2. Les variables"asynchrones":

Dans un programme FORTRAN une variable dont le nom commence par le caractère "$" est dite "asynchrone"; son utilisation implique le test de l'état du mot mémoire associé à la variable :

- on doit attendre que l'état du mot soit "plein" pour en lire le contenu; après lecture l'état de ce mot devient "vide"; aucune autre instruction ne peut intervenir sur ce mot entre la lecture de l'état "plein" et le passage à l'état "vide".

EXEMPLE :

```
I = $ J + 2
(on attend que $ J soit plein (1))
```

- on doit attendre que l'état du mot soit "vide" avant d'y écrire une valeur; après écriture l'état du mot passe à "plein"; aucune autre instruction ne peut intervenir sur ce mot entre la lecture de l'état "vide" et la passage à l'état "plein".

---

(1) Par abus de langage cette phrase signifie :"on attend que le mot de mémoire associé à $J soit plein"

EXEMPLES :

```
$ A = 3. * COS (B) + 5
(on attend que $ A soit "vide")
$ C = $ BD + 1
(on attend que $ BD soit  plein , puis que $ C soit vide)
```

4.3. Les fonctions de manipulation des variables asynchrones

PURGE : force l'état de variables asynchrone à "vide", sans tenir compte de l'état précédent. A utiliser au début d'un travail pour initialisation
exemple : PURGE $A,$B

VALUE : donne la valeur actuelle d'une variable asynchrone, sans tenir compte de l'état; ne change pas la valeur de l'état.

SETE : donne la valeur actuelle d'une variable asynchrone, sans tenir compte de l'état; force l'état à "vide"

WAITF : attend que l'état d'une variable asynchrone soit "plein", retourne le contenu sans changer l'état

FULL : teste l'état d'une variable asynchrone; renvoie la valeur . TRUE . ou . FALSE . ; ne change pas l'état.

4.4. Deux exemples simples

Synchronisation " FORK -JOIN" :

un flot séquentiel d'instructions se sépare (fork) en N flots parallèles; quand tous les processus parallèles ont atteint un point de synchronisation donné, où le parallèlisme cesse, tous les processus s'arrêtent sauf un (join)

```
      PROGRAM A
      COMMON /SYNCH/ $IC,$FINI
      ------
      PURGE $IC,$FINI
      $IC = N
      DØ  10 I = 1, N-1
      CREATE B(liste de paramètres)
10    CONTINUE
      CALL  B(liste de paramètres)
      I = $FINI

      STOP
      END
```

```
C       SUBROUTINE B (arguments)
        processus dupliqué N fois
        CØMMØN/SYNCH/$IC,$FINI

        calculs

C       tester la fin des autres processus
        I = $IC - 1
        IF (I . EQ . 0)
           THEN
              $FINI = 1
           ELSE
              $IC = 1
        RETURN
        END
```

"Auto - sheduling"

Considérons le code séquentiel :

```
 DØ  10  I = 1, N
     -----
     calculs dépendant de I
10   CONTINUE
```

supposons que les diverses itérations de la boucle soient indépendantes : on peut les répartir entre un certain nombre, NPRØC, de processus parallèles. On peut décider d'allouer à l'avance le travail à chaque processus, par exemple le Jème processus traitera les calculs correspondant, dans la boucle ci-dessus, aux indices I = multiple de J.

Mais, si les calculs des différentes itérations est très variable il est avantageux de laisser chaque processus choisir les indices des calculs qu'il effectuera; comme dans le programme ci-dessous :

(Remarquer que lorsqu'un processus P_0 a réussi à "vider" les variables $K à l'étiquette 5, aucun autre processus ne peut lire ou modifier $K avant que P_0 n'ait à nouveau rempli $K par $K = I + 1)

```
      PRØGRAM  AUTO
      -----
      LØGICAL $FAIT,L
      CØMMØN $K,N,$FAIT,$IACTIF
      PURGE $K,$FAIT,$IACTIF
      N = -----
      NPRØC = ----
      $K = 1
      $IACTIF = NPRØC
      DØ  10  J = 1,NPRØC -1
      CREATE SUB(---)
10    CØNTINUE
      CALL SUB(---)
      L = $FAIT
 c    Cette instruction garantit que tous les processus ont terminé
      ------
      STØP
      END
      SUBRØUTINE  SUB(---)
      CØMMØN $K,N,$FAIT,$IACTIF
      LØGICAL $FAIT
      ------
 5    i = $K
      IF (i . LE . N)
     .    THEN
             $K = I + 1
             -----
             calculs (I)
             ------
             GØ TØ 5
          ELSE
             K1 = $IACTIF -1
             IF (K1 . EQ . 0) $FAIT = . TRUE .
             $IACTIF = K1
             $K = N+1
             ENDIF
             RETURN
             END
```

Preuve du programme AUTO :

- aucun processus ne reste bloqué en attente sur une variable asynchrone;
- tous les processus se terminent : entre deux calculs sur un indice I, le compteur $K a été incrémenté de 1 ou plus; donc le nombre d'itération dans chaque processus est borné
- Toutes les valeurs de I sont utilisées (de 1 à N) = le compteur $K est utilisé à 1, le premier processus capable d'obtenir le compteur traite la valeur 1 de l'indice I, et passer le compteur $K à la valeur 2; un raisonnement par récurrence permet de conclure.
- Chaque valeur de I n'est utilisée qu'une fois.

Références :

SMITH (Burton. J.)
" A pipelined shared resource MIMD computer",
IEEE, International Conference on Parallel Processing, 1978

DENELCOR, Inc.
"Heterogeneous Element Processor : Principes of opérations", Avril 1981
"HEP : FORTRAN 77 User's guide", Février 1982

1 - Position du problème

Le problème posé est le suivant : des processus "producteurs" produisent en parallèle des " articles" utilisés par des processus "consommateurs".
Chaque article est rangé dans une zone de "tampons" en attendant sa consommation.
L'application envisagée est l'assemblage d'une matrice de discrétisation par éléments finis :

* Les producteurs calculent les matrices élémentaires
* Les consommateurs accumulent les matrices élémentaires dans la matrice globale.

On dispose d'un nombre limité de tampons : les producteurs sont en compétition pour obtenir les numéros des tampons dans lesquels ils pourront calculer; les consommateurs attendent que des tampons soient pleins. La gestion des tampons, c'est à dire leur allocation à divers processus, doit s'effectuer dans une section critique : on appelle ainsi une séquence d'instructions qui ne peut être exécutée que par un seul processus à la fois (on dit aussi que cette séquence est exécutée en exclusion mutuelle).
Sur le HEP une section critique pourra être réalisée à l'aide d'une variable asynchrone, vidée au début et remplie à la fin de la section critique :

```
R1 = $ VERROU
: section critique
:
$ VERROU = R2
```

Les valeurs présentent dans $ VERROU ne sont pas nécessairement significatives : c'est l'état du mot qui est important.

Enfin il faut résoudre les conflits entre les consommateurs de façon à garantir l'intégrité de la structure de données contenant la matrice globale; chaque processus consommateur contiendra dans la boucle la plus interne une instruction FORTRAN de la forme :

```
AG(KG) = AG(KG) + AELEM (KELEM)
```

où AELEM est un "article" ou matrice élémentaire.

Lorsqu'un processus P_o a réussi à lire la valeur AG(KG) il faut empêcher tout autre processus de lire ou de modifier le mot mémoire avant que P_o n'ait inscrit la nouvelle valeur résultant de l'accumulation de AELEM(KELEM) sur le HEP ceci peut être réalisé en écrivant à la place de l'instruction ci-dessus :

CHAPITRE VIII

UN EXERCICE DE PROGRAMMATION PARALLÈLE SHR LE HEP :

PRODUCTEURS - CONSOMMATEURS

APPLICATION À L'ASSEMBLAGE D'UNE

MATRICE D'ÉLÉMENTS FINIS.

$ AG(KG) = ($ AG(KG) + AELEM(KELEM)

(l'état est associé à chaque mot-mémoire, et non au tableau $ AG).

Cette solution pourrait être envisagée sur le HEP où les synchronisations sont réalisées grâce à des dispositifs purement matériels. On ne la retiendra pas pour deux raisons :

* Elle contraint à tester l'état de chaque variable dans la structure des données de la matrice globale, ce qui allonge les opérations et inhibe d'éventuelles optimisations par le compilateur;

* Ce type de solution serait totalement inefficace sur une machine où les synchronisations reposeraient sur des dispositifs logiciels plus lourds comme sur le CRAY-XMP.

Ainsi utilisera-t-on une démarche de type "sous-domaines" : le domaine de calcul est recouvert par une partition en sous-domaines disjoints (figure 1)

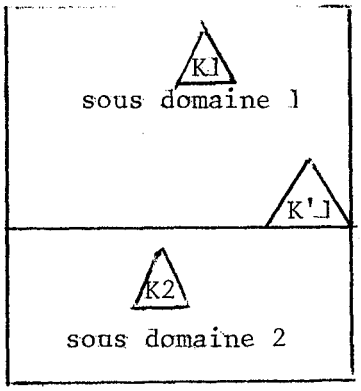


Figure 1 : partition en 2 sous domaines :
L'assemblage de K1 peut être effectué en parallèle avec celui de K2; l'assemblage de K'1 peut donner lieu à des conflits.

On affecte un processus consommateur à chaque sous domaine : alors l'assemblage des éléments intérieurs aux sous domaines peut s'effectuer sans conflit; l'assemblage d'un élément touchant une frontière inter-sous-domaine sera fait dans une section critique. Par contre les producteurs ne sont pas spécialisés et peuvent traiter des éléments quelconques.

2 - Notations et structures de données

2.1. Le maillage

NE = nombre d'éléments du maillage
NP = nombre de processus producteurs
NC = nombre de processus consommateurs
= nombre de sous-domaines
NUSD = tableau d'entiers de dimension NE
= donne pour chaque élément le numéro de sous-domaine auquel il appartient.
IFSD = tableau de valeurs logiques de dimension NE
= IFSD(j) = . TRUE . si élément j touche une frontière inter sous-domaine,
= . FALSE. sinon.

On supposera que ces données sont calculées dans une étape préliminaire; cette étape doit également préparer d'une part les structures permettant d'obtenir les caractéristiques géometriques de chaque noeud; d'autre part, les structures de données décrivant la matrice globale. On ne donnera pas ici le détail de ces structures : on suppose qu'elles sont utilisées dans deux sous programmes :

. CALMAT (JE,IBUF) : calcule la contribution de l'élément J dans le tampon du numéro IBUF
. ASSEM (JE,IBUF) : assemble la contribution de l'élément JE

Ces sous programmes ne sont pas listés ici : en fait l'algorithme utilisé est indépendant du choix du type d'éléments finis comme du choix de la représentation de la matrice globale.

Hypothèse sur la numérotation : les numérotations des sous-domaines sont entremélées: on numérote d'abord le premier élément du premier sous-domaine puis un élément du premier sous-domaine, etc; jusqu'au premier élément du NCème sous-domaine; puis le premier élément du 2ème sous-domaine...

2.2. les tampons

NBUF = nombre de tampons
LITEM = nombre de mots d'un "article"
BUFFER (LITEM,NBUF) = les tampons
JBUF (NBUF) = numéros d'éléments correspondant aux articles dans les tampons
= initialement à 0

Allocation des tampons :

On choisit de répartir statiquement les tampons entre les consommateurs.

Pour simplifier, on supposera que NBUF est un multiple entier du nombre de consommateurs NC.

Le ième consommateur aura les tampons :

$$IBUF = (i-1) * \frac{NBUF}{NC} + 1 \text{ à } IBUF = i * \frac{NBUF}{NC}$$

$ ETAT (NBUF) = donne l'état de chaque tampon :

= 0 : vide et libre

= 1 : réservé par un producteur

= 2 : plein et prêt à consommer

3 - Le programme

L'algorithme éxécuté par un producteur est le suivant :

```
PRODUCTEUR :
TANT QU'IL RESTE DES ELEMENTS REPETER :
- obtenir un numéro d'élément non calculé JE
- ic = NUSD (JE) = numéro de sous domaine
                 = numéro de consommateur
- obtenir un tampon vide, et le marquer réservé
- calculer la contribution dans le tampon obtenu
- marquer le tampon "plein"
DECREMENTER LE NOMBRE DE PRODUCTEURS ACTIFS
```

L'algorithme exécuté par un consommateur est :

```
CONSOMMATEUR (IC)
REPETER :
- obtenir un tampon plein et assembler une matrice élémentaire;
- mettre l'état du tampon à vide
- si tous les tampons sont vides, tester le nombre de producteurs encore
  actifs : si tous les producteurs sont arrêtés, décrémenter le nombre
  de consommateurs actifs et terminer.
SI TOUS LES CONSOMMATEURS ONT TERMINE, signaler la fin de l'algorithme.
```

L'implémentation en FORTRAN-HEP est donnée ci-dessous : On suppose que l'initialisation (ou la lecture sur fichier) des structures de données du maillage s'effectue dans la procédure INIMAIL, et que l'utilisation (ou l'écriture sur un autre fichier) de la matrice assemblée s'effectue dans la procédure UTMAT.

On notera que la procédure consommatrice a un paramètre, qui est le numéro de sous domaine : ce numéro est aussi l'indice IC de la boucle de "création".
Si l'on transmettait la variable IC sans précaution à CONSOM, toutes les incarnations de CONSOM utiliseraient en fait la même mémoire pour ranger leur paramètre , celle qui est réservée dans le programme principal!
L'utilisation de la variable asynchrone $IIC permet à chaque consommateur de recopier son numéro dans une variable locale.

```
      PRØGRAM  PRODCONS
      CØMMØN/SYNCH/$JEG,$FINI,$PACTIF,$CACTIF,$GO(NC),$SC
      INTEGER  $JEG,$FINI,$PACTIF,$CECTIF,$GO,$SC
      PARAMETER (NE= ...) NBUF = ..., NP=...,NC = ...,LITEM = ...)
      CØMMØN/BUF/BUFFER (LITEM, NBUF), JBUF (NBUF), $ ETAT (NBUF)
      INTEGER  $ETAT
      INTEGER  VIDE,RESER,PLEIN
      DATA  VIDE,RESER,Plein/0,1,2/
      PURGE  $JEG,$FINI,$PACTIF$CACTIF,$ETAT,$GO,$IIC
      DØ  1 IBUF = 1,NBUF
          JBUF (IBUF) = 0
1         $ETAT (IBUF) = VIDE
      $PACTIF = NP
      $CACTIF = NC
      $JEG = 1
      $SC = 1
      DØ  2  IC = 1,NC
2         $ GO(IC) = 1
      CALL INIMAIL
      DØ  3  IP = 1,NP
          CREATE PRODUC
3     CØNTINUE
      DØ 4  IC = 1,NC - 1
         $IIC= IC
         CREATE CONSOM ($IIC)
4     CØNTINUE
      $IIC = NC
      CALL CONSOM ($IIC)
      i = $ FINI
      CALL UTMAT
      STØP
      END
```

```
      SUBROUTINE  PRODUC
      COMMON/SD/ NUSD (NE), IFSD(NE)
      CØMMØN/SYNCH/ $JEG, $FINI, $PACTIF, $CACTIF, $GO(NC), $SC
      INTEGER   $JEG, $FINI, $PACTIF, $CACTIF, $GO, $SC
      PARAMETER (NE = ..., NBUF = ..., NP = ..., NC = ..., LITEM = ...)
      CØMMØN/ BUF/ BUFFER (LITEM,NBUF), J BUF(NBUF), $ETAT(NBUF)
      INTEGER  $ETAT
      INTEGER  VIDE, RESER, PLEIN
      DATA  VIDE, RESER, PLEIN /0,1,2/
C     D'ABORD OBTENIR UN NUMERO D'ELEMENT
C     PROPRIETE INVARIANTE (PO)
C     POUR TOUT K< $JEG, K >0, UN PRODUCTEUR EST DEJA PASSE
C     PAR L'ETIQUETTE 1 ET A LU LA VALEUR K POUR $JEG
1     JE = $JEG
      $JEG = JE + 1
      IF (JE . GT. NE)
     . THEN
           $ PACTIF = $ PACTIF - 1
      ELSE
           IC = NUSD (JE)
           IBUF1 = (IC-1) * (NBUF/NC) + 1
           IBUF2 = IC * (NBUF/NC)
2          DØ 3  IBUF = IBUF1, IBUF2
              IETA = $ETAT(IBUF)
              IF (IETA.EQ.VIDE) GØ TØ 4
3             $ETAT (IBUF) = IETA
           GØ TØ2
4          $ETAT (IBUF) = RESER
           JBUF (IBUF) = JE
           CALL CALMAT (JE, IBUF)
           IETA = $ETAT (IBUF)
           $ETAT (IBUF) = PLEIN
           PURGE $ GO(IC)
           GØTØ 1
      ENDIF
      RETURN
      END
```

```
      SUBROUTINE  CØNSØM($IIC)
      CØMMØN/SYNCH/ $JEG, $FINI,$PACTIF,$CACTIF,$GO(NC),$SC
      INTEGER $JEG, $FINI,$PACTIF,$CACTIF,$GØ, $SC
      PARAMETER (NE=...,NBUF=...,NP=...,NC=...,LITEM...)
      CØMMØN/BUF/BUFFER(LITEM,NBUF), JBUF(NBUF),$ETAT(NBUF)
      INTEGER  $ETAT, VIDE,RESER,PLEIN
      DATA VIDE,RESER,PLEIN /0,1,2/
      CØMMØN /SD/NUSD(NE),IFSD(NE)
      LOGICAL IFSD
C     RECOPIER TOUT DE SUITE LE NUMERO DE SOUS DOMAINE
      IC = $IIC
C     ATTENDRE QU'UN TAMPON SOIT PLEIN
      $GØ(IC) = 1
      IBUF1 = (IC-1) * (NBUF/NC) +1
      IBUF2 = IC * (NBUF/NC)
10    NBRES = 0
C     CHERCHER UN TAMPON PLEIN
      DØ  20  IBUF = IBUF1, IBUF2
          IETA = $ETAT (IBUF)
          IF (IETA.EQ.PLEIN) GØTØ40
          IF (IETA.EQ.RESER) NBRES = NBRES + 1
20        $ETAT (IBUF) = IETA
C     NBRES = nombre de tampons trouvés dans l'état réservé
C     SI NBRES = 0, tous les tampons ont été trouvés vides.
      IF ( NBRES.EQ.0.AND. VALUE($PACTIF).EQ.0)
      GØTØ 50
      GØTØ 10
40    $ETAT(IBUF) = IETA
      JE = JBUF(IBUF)
      IF (IFSD(JE)) ISC = $SC
      CALL ASSEM (JE, IBUF)
      IF (IFSD(JE)) $SC = ISC
      IETA = $ETAT (IBUF)
      $ETAT (IBUF) = VIDE
      GØTØ 10
50    NBC = $CACTIF - 1
      $CACTIF = NBC
      IF (NBC.EQ.0) $FINI = 1
      RETURN
      END
```

4 - Eléments de preuve du programme précédent

* On peut facilement vérifier que les variables asynchrones de ce programme sont généralement utilisées de la façon suivante : tout "vidage" de la variable par un processus est nécessairement suivi, quelques instruction plus loin[1], du "remplissage" de la variable par le même processus.
Tel est le cas des variables $ETAT, $PACTIF,$CACTIF,$SC,$JEG.
Il est donc impossble qu'un processus reste bloqué indéfiniment en attendant l'une de ces variables.
Les cas particuliers concernent $GØ,$FINI,$IIC :

* la variable $IIC est remplie par le programme principal avec la valeur IC, avant de "créer" le consommateur numéro IC, qui vide la variable $IIC.

* La variable $GØ(IC) est vidée par un ordre PURGE du processus producteur qui réussit à remplir un tampon à destination du consommateur numéro IC (le PURGE ne provoque pas d'attente).

* $FINI sert à tester la fin de l'algorithme : seul le processus consommateur qui réussit à décrémenter $CACTIF jusqu'à 0 peut remplir la variable $FINI (c'est le dernier consommateur actif).

* on a donc le même résultat concernant toutes les variables asynchrones : aucun processus ne reste bloqué indéfiniment pour l'état de ces variables.

* Quand un tampon est trouvé plein par un consommateur, celui-ci le libère en un temps fini -

* Supposons qu'un producteur trouve que tous les tampons associés au consommateur de numéro IC sont pleins : il est alors bloqué et réexécute la boucle 3 jusqu'à trouver un buffer vide.
Mais alors le consommateur IC, éxécutant la boucle 20, trouvera nécessairement un tampon plein, qu'il liberera après l'appel à ASSEM[2] donc il est impossible que tous les producteurs trouvent indéfiniment tous les tampons (associés à un consommateur) pleins. (de même il est impossible que ces tampons soient indéfiniment tous réservés ou pleins).

* Il en résulte que le compteur $JEG atteint la valeur limite NE en un temps fini : tous les producteurs sont alors conduits à terminer leur exécution; quand le dernier producteur a terminé, $PACTIF atteint la valeur 0.

---

(1) Sans intervention d'autre variable asynchrone

(2) On suppose bien sûr que tous les éléments font partie d'un sous domaine auquel correspond un consommateur unique et réciproquement que les consommateurs correspondent à des sous domaines non vides.

* D'après la propriété invariante PO(voir le texte de PRODUC), les contributions de tous les éléments de JE = 1 à JE = NE, sont calculées une fois et une seule.

* Donc pour tous consommateurs IC, un producteur aura réussi à exécuter PURGE $GO (IC) (après le calcul des contributions d'un élément du sous domaine IC), ce qui permet au consommateur IC de démarrer ($GO(IC) = 1).

* Si un consommateur trouve des tampons réservés (boucle 20) : ils sont passés dans l'état "réservé" à l'instruction 4 dans PRODUC; ils passeront dans l'état plein en un temps fini et le consommateur pourra continuer.

* Quand tous les producteurs ont terminé, il ne reste que des tampons "vides" ou "pleins" : le nombre de tampons pleins diminue à chaque passage dans la boucle du consommateur (à chaque passage par "$ETAT (IBUF) = VIDE"). Donc tous les tampons des consommateurs deviennent vides en un temps fini. Alors le test de la valeur de $PACTIF (en utilisant VALUE, donc tester ni changer l'état du mot-mémoire) permet de vérifier que tous les producteurs ont bien terminé.
Donc tous les consommateurs se terminent en un temps fini.

Remarques et Conclusion

1/ La solution qui a été présentée ici n'est peut être pas optimale :

* Il serait souhaitable d'éliminer ou de rendre aussi rares que possible les "attentes actives" (boucles 3 et 20) qui usent du temps de calcul de façon improductive.

* On n'a pas distingué entre les diverses parties de l'interface inter-sous domaines

2/ L'écriture d'un programme parallèle doit s'accompagner d'une preuve : il n'est pas raisonnable de s'en remettre aux tests à l'exécution pour trouver les erreurs.

Références :

David GRIES : "The Science of programming", Springerverlag, 198

David GRIES : "An exercice in Proving Parallel Programms corret", communications of the ACM, Déc 1977, Vol 20, N° 12, pp 921 - 930

Brent HAILPERN : "Temporal Logic", Lecture notes in computer science

Sur la programmation en général :

Nicklaus WIRTH : " Algorithme + Data structure = Programm" Prentice-Hall

Henri LEDGARD : "Proverbes de programmation", Dunnod

Claude BAUDOIN & Bertrand MEYER : "Méthodes de Programmation", EY Volles, 1978

N° D'ORDRE : 701
3È TRIMESTRE 1984