ÉLECTRICITÉ DE FRANCE
DIRECTION DES ÉTUDES ET RECHERCHES

Service Informatique
et Mathématiques Appliquées

Décembre 1981

6

DEPARTEMENT METHODES ET MOYENS
DE L'INFORMATIQUE
1, Avenue du Général de Gaulle
92141-CLAMART

BOSSAVIT A.

VECTORISATION DE QUELQUES ALGORITHMES NUMERIQUES

Note CRAY n° 6

HI 4001/01

36 Pages

Résumé Deux exemples de réécriture d'algorithmes classiques pour obtenir une meilleure efficacité sur le Cray-1 tout en restant en Fortran : factorisation de Choleski et méthode itérative de Gauss-Seidel. Deux approches du problème : repenser les programmes "en termes de vecteurs" ou passer par transformation du programme séquentiel au programme vectoriel équivalent. Discussion de ces deux approches.

VECTORISATION DE QUELQUES ALGORITHMES NUMERIQUES

La présente "note Cray" rassemble deux articles déjà parus sur la factorisation de Choleski et la méthode itérative de Gauss-Seidel, écrits avant le premières exploitations du Cray-1 de Clamart. Des essais numériques ont été faits depuis, dont on pourra trouver les résultats dans la chronique du BUCCER(*) "Comment j'ai vectorisé certains de mes programmes", publiée courant 1981.

Il y a deux façons d'aborder le problème de la vectorisation des programmes, c'est-à-dire de leur réécriture pour leur donner plus d'efficacité sur une machine telle que le Cray-1. La première, propre à la réutilisation rapide de programmes existants, consiste à opérer certaines transformations, dont la plus connue est la permutation d'indices de boucles, en suivant certaines règles d'équivalence, développées dans la chronique "Cray" d'E. de Drouas dans le BUCCER. Toutefois, il est rare - comme nous le montrons plus loin - que de telles transformations donnent les "bonnes" versions des algorithmes, que l'on ne peut trouver en fait que par une approche toute différente, exposée ci-dessous, et qui peut se résumer par l'aphorisme : *pour bien vectoriser, penser vecteurs*. On verra plus loin comment il faut l'entendre.

D'autres exemples suivront, au fil des essais actuellement en cours. Le lecteur intéressé immédiatement par des développements plus approfondis peut utilement se reporter à l'ouvrage de Kuck, Lawrie et Sameh, *High Speed Computer and Algorithm Organization*, Academic Press (1977).

(*) Bulletin du Centre de Calcul des Etudes et Recherches, EDF Clamart.

- I -

LA VECTORISATION DE CHOLESKI

L'intérêt pour le "calcul parallèle" ne date pas d'hier. Vers la fin des années 60, il existait un ordinateur parallèle, l'ILLIAC 4, et c'était la mode de rechercher le parallélisme dans les algorithmes classiques. Beaucoup d'idées actuelles se trouvent déjà exposées au congrès de l'IFIP de 1971 [1], entre autres sources. Cet engouement était prématuré et passa vite. Puis vinrent le CDC 7600, avec déjà un certain degré de parallélisme, puis à partir de 1976 le Cray 1 dont un exemplaire arrivera à EDF en 1981, et, chez beaucoup de constructeurs, des "processeurs de tableaux", conçus comme des périphériques. On voit aussi apparaître sur le marché des machines qu'on peut assimiler à des "tableaux de processeurs", exécutant simultanément la même instruction sur des données différentes. Tout cela suscite un renouveau d'intérêt pour le parallélisme, manifeste dans plusieurs congrès récents ou annoncés [2] [3], et on va sans doute redécouvrir beaucoup de vieilles idées. Le présent article ne fait rien d'autre, à partir des acquis récents en méthodologie de la programmation [4] [5] [11]. Son but est de montrer qu'on peut mettre en évidence les possibilités de parallélisme, ou de vectorisation, au niveau de la spécification même des algorithmes et développer à partir de là des programmes corrects. Cette démarche semble supérieure à celle qui consisterait à "vectoriser" des programmes existants (bien qu'on obtienne déjà de cette façon des résultats appréciables).

Le plan est le suivant :

1) Parallélisme dans les programmes et dans les machines.

2) Calcul sur les vecteurs.

3) Systèmes linéaires triangulaires.

4) Choleski "vectoriel".

5) Conclusion.

1. - PARALLELISME DANS LES PROGRAMMES ET DANS LES MACHINES

Il y a parallélisme dans un programme, relativement à un certain ensemble de variables, si l'exécution simultanée et indépendante d'une certaine suite d'actions a le même effet (sur ces variables) que son exécution séquentielle.

Cette définition est d'une simplicité trompeuse, puisqu'elle fait appel à des concepts (programmes, variables, actions, etc.) qui eux-mêmes auraient grand besoin d'être définis. Le problème n'est pas nouveau en informatique, où on a rencontré les processus parallèles dès qu'on s'est mis à concevoir des systèmes d'exploitation, mais n'est pas encore résolu de façon définitive. Citons seulement [6] et [7] parmi les formalisations proposées.

Nous nous contenterons ici d'esquisser une description plus rigoureuse du parallélisme, tout en introduisant progressivement les notations et conventions qui seront employées par la suite.

L'exemple le plus connu d'algorithme relevant du calcul parallèle est celui de Jacobi pour la résolution des systèmes linéaires Ax = b. Il s'écrit :

```
(J)   déclaration a : MATRICE, x, y, b : tableaux [1:n]
                  de REELS {ordre(a) = n};
      données a; b;
      x ← 0 ;
      répéter
      |     pour i de 1 à n répéter
      |     |     y_i ← (b_i - Σ_{j≠i} a_ij x_j)/a_ii;
      |     x ← y
      jusqu'à satisfaction
```

(Les notations sont empruntées à [9] , avec des modifications mineures; "x ← y" signifie "affecter à la variable x la valeur y"; la partie déclarative précise quelles seront les variables du programme

et donne leur "type" (REELS, ENTIERS, MATRICES, etc.); les textes entre { } sont soit des commentaires soit des assertions sur les variables. Ce langage de programmation n'étant pas destiné à être compilé, on peut se permettre beaucoup de libertés typographiques, par exemple x_i au lieu de x(i), chaque fois qu'on y gagne en lisibilité.)

Cet algorithme est parallélisable car il est équivalent à la version transformée ci-dessous :

```
                répéter
                |       exécuter parallèlement, pour i de 1 à n,
(J')            |       |       y_i ← ...
                |       x ← y
                jusqu'à satisfaction
```

A l'opposé, l'algorithme de Gauss-Seidel, qui s'écrit

```
                répéter
(G.S.)          |       pour i de 1 à n répéter
                |       |       x_i ← (b_i - ...)
                jusqu'à satisfaction
```

n'est pas parallélisable, car la même transformation donne cette fois quelque chose de différent.

Essayons de généraliser. Soit un programme opérant sur des variables de types X, Y et Z, où x est le résultat cherché et les y_i des résultats intermédiaires de certaines actions A :

```
                déclarations x : X, y_1, ..., y_n : Y, z : Z,
                données ... ;
                répéter
(Seq)           |       pour i de 1 à n répéter
                |       |       A_i {action qui, entre autres choses
                |       |                   modifie y_i};
                |       x ← f(x, y_1, ..., y_n)
```

Le programme sera parallélisable si on peut substituer à *pour i de 1 à n répéter* l'ordre *exécuter parallèlement* ... sans modifier le résultat x (par contre les z peuvent être affectés : il y a parallélisme "relativement aux X et aux Y"). Il faut pour cela que les A_i laissent invariants x, d'une part, et tous les y_j sauf y_i d'autre part. On écrira cela, pour tout i,

(1) $$\forall \xi \in X, \quad \{x = \xi\} \quad A_i \quad \{x = \xi\}$$

(2) $$\forall \eta \in Y, \ \forall j \neq i, \quad \{y_j = \eta\} \ A_i \ \{y_j = \eta\}$$

Dans cette notation, imitée de Hoare [5], une écriture telle que

(3) $$\{P\} \ A \ \{Q\}$$

signifie que si l'on exécute A au moment où les variables vérifient la propriété P, alors Q devient vrai. Sous la forme (3), ou sous la forme quantifiée (1) ou (2), il s'agit d'axiomes sur les actions A. On conçoit comment ce formalisme permet de "prouver" des programmes : si l'action A intervient dans un état des variables vérifiant {R}, avec R => P, alors on peut affirmer que Q est vrai après l'exécution de A, et l'on pourra voir apparaître dans le programme la séquence {R} A {Q} où R et Q sont maintenant des assertions sur l'état des variables.

Ce formalisme permet d'aborder la question de l'équivalence des programmes, par exemple de prouver que la section de programme "Seq" ci-dessus est équivalente à

(Par)

répéter
|
exécuter parallèlement, pour i de 1 à n
| A_i ;
$x \leftarrow f(x, y_1, \ldots, y_n)$

Ainsi, avec (1), (2) et d'autres hypothèses sur les actions A_i, on peut énoncer :

Théorème - Pour tout couple de prédicats P et Q ne portant que sur x et y on a

(4) $\{P\}\ Seq\ \{Q\} \Leftrightarrow \{P\}\ Par\ \{Q\}$

Bien entendu tout cela reste trivial, mais la question n'est pas de démontrer des théorèmes profonds : elle est de se donner un moyen de reconnaître le parallélisme, là où il existe, et de façon aussi automatique que possible. Les recherches dans cette voie pourraient conduire par exemple à l'écriture de compilateurs reconnaissant le parallélisme potentiel dans les programmes pour exploiter les possibilités de parallélisme d'une machine donnée.

On est loin d'en être là. Les compilateurs actuels ne repèrent que certaines constructions parallélisables (des cas particuliers par rapport à ce qu'on vient de voir), correspondant étroitement à certaines particularités des machines.

Le parallélisme est réalisé, selon les constructeurs, de deux façons :

1) On peut confier chaque A_i à des processeurs physiquement différents, c'est-à-dire exécuter simultanément la même instruction sur une multiplicité de jeux de données. C'est ce que faisait l'ILLIAC 4, et ce que fait par exemple le DAP d'ICL.

2) On peut exécuter successivement les A_i, mais profiter du fait qu'il s'agit des mêmes instructions pour accélérer (dans des proportions considérables) leur déroulement, en organisant à l'avance le flux des données. C'est ce qui se passe, selon des modalités différentes, chez CDC avec la série Cyber et chez Cray.

Dire en quoi consiste cette "organisation du flux des données" demanderait trop de développements. On se contentera d'évoquer l'image

de la chaîne de montage pour illustrer le cas numéro 2(*). On peut aussi parler de "décentralisation" pour (1) et d'"organisation scientifique du travail" pour (2), ou évoquer Mao dans le premier cas, et Stakhanov (ou Taylor, selon les goûts) dans le second. Mais passons, et renvoyons à [10] pour plus de détails.

Les actions A qui peuvent s'exécuter en parallèle sont en pratique très limitées : elles se réduisent à des suites (pas trop longues) d'instructions élémentaires (affectations, opérations arithmétiques, à l'exclusion des tests, des appels de sous-programmes et des boucles ; on trouvera des règles très précises à ce sujet, pour le cas particulier du Cray-1, dans [10]). En pratique, le parallélisme des machines ne s'applique qu'aux sections de programme de la forme :

répéter pour i de 1 à n
| $x_i \leftarrow f(y_i, z_i, \ldots)$

où f est une fonction ne dépendant pas de i. Autrement dit, il n'y a guère de parallélisme possible que lorsqu'on se trouve en présence de la même opération sur des éléments de tableaux homologues (c'est-à-dire de même indice) donc d'une opération de type VECTEUR × VECTEUR × ... → VECTEUR, vecteurs de types différents mais de même longueur. C'est pourquoi on parle de vectorisation :

> *Il y a vectorisation lorsque la machine est en mesure de traiter des tableaux parallèles de longueur n en (beaucoup) moins de n fois le temps qu'elle prendrait pour traiter un jeu d'éléments homologues de ces tableaux.*

Peu importe que cet effet soit obtenu par l'un ou l'autre des procédés ci-dessus, ou une combinaison des deux. On parle de "machines vectorielles" et aussi de "processeurs de tableaux"; ces derniers sont

---

(*) Les lecteurs assidus de Marx penseront tout de suite aux "deux formes fondamentales [hétérogène et sérielle] de la manufacture" [8].

des périphériques spéciaux, branchés sur une machine ordinaire, auxquels on "sous-traite" les opérations vectorielles.

Le mot "vectorisation" s'emploie aussi, en un autre sens, pour désigner l'exercice consistant à modifier un algorithme ou un programme pour tirer parti des possibilités de vectorisation de la machine. C'est ce qui va nous occuper plus loin.

En résumé : le parallélisme est une caractéristique des programmes que l'on peut espérer définir avec rigueur et peut-être détecter par des moyens automatiques. Mais à l'heure actuelle seuls certains cas de parallélisme sont exploitables (au moyen de l'exécution simultanée ou séquentielle-accélérée des actions parallèles). Ils correspondent en gros aux opérations "sur les vecteurs" et on parle à leur propos de "machines vectorielles" et de "vectorisation".

2. - CALCUL SUR LES VECTEURS

Puisque le parallélisme se réduit en pratique à la vectorisation, nous allons centrer notre attention sur les programmes qui travaillent sur des tableaux de scalaires, par exemple des REELS.

Si $\oplus$ est une opération arithmétique, le programme

(5)
```
déclarations x, y, z : tableaux [1:n] de REELS,
données x, y ;
pour i de 1 à n répéter
|        z(i) ← x(i) ⊕ y(i)
```

est parallélisable. Par contre,

(6)
```
d ← d_o ;
pour i de 1 à n répéter
|       d ← d ⊕ x(i)
```

ne l'est pas. Il en résulte qu'une opération aussi fondamentale que le produit scalaire n'est pas vectorisable si on le programme comme on a l'habitude de le faire, par exemple en Fortran comme suit :

```
      PS = 0.
      DO 1  I = 1, N
            PS = PS + X(I) * Y(I)
1           CONTINUE
```

Le produit scalaire étant une opération primitive pour la plupart des algorithmes numériques, on a lieu a priori d'être sceptique quant aux avantages de la vectorisation. La première idée qui vient à l'esprit est de faire d'abord $z(i) \leftarrow x(i) * y(i)$ puis de couper en deux le vecteur z, d'additionner les deux moitiés (opération "vectorielle") et ainsi de suite, mais la complexité du flot d'instructions fera perdre le bénéfice de la vectorisation si n n'est pas très grand.

La solution réside dans un changement de point de vue complet, qu'il suffira d'illustrer sur l'exemple suivant, fréquemment cité. Il s'agit de la multiplication d'un vecteur par une matrice, version classique à gauche, version parallélisable à droite.

déclarations a : n-matrice, x,y : tableaux [1:n] de REELS,
i, j : ENTIERS, d : REEL ;

```
pour i de 1 à n répéter
|       y_i ← 0;
|       pour j de 1 à n répéter
|       |       y_i ← y_i + a_i^j x_j
```

```
y ← 0 ;
pour j de 1 à n répéter
|       pour i de 1 à n répéter
|       |
|       |       y_i ← y_i + a_i^j x_j
```

A gauche la boucle la plus interne est du type (6), donc n'est pas vectorisable. A droite au contraire les y_i peuvent être incrémentés en parallèle et on peut vectoriser. On remarquera que le "taux de parallélisme" de ces deux programmes est le même : dans les deux cas les actions de la

boucle indexée par i sont parallèles (on peut s'employer à le vérifier en utilisant (1) et (2)). Si la version de droite est seule vectorisable, c'est parce que la vectorisation n'exploite le parallélisme que dans les boucles les plus internes (cf. [10]).

Une autre remarque va nous fournir une clé pour aller plus loin. A droite, on a fait en sorte de retarder jusqu'au dernier moment (la boucle la plus interne) l'individualisation des éléments du tableau y, on a "travaillé sur des vecteurs" (y et les colonnes de a) le plus longtemps possible. Il est de simple bon sens d'ériger cela en principe : s'il est avantageux de faire des opérations vectorielles, il faut "penser vecteurs" et essayer d'exprimer les algorithmes en termes de vecteurs.

Introduisons donc le type VECTEUR (de REELS par exemple; en fait, c'est un type générique) par l'énumération des fonctions opérant sur les objets de type VECTEUR (voir [9] pour d'autres applications de cette démarche) :

type VECTEUR :

	longueur : VECTEUR → ENTIER
	composante : VECTEUR × ENTIER → REEL
	addition : VECTEUR × VECTEUR → VECTEUR
	multiplication : - *id* -
	... (*multiplication par un scalaire, projections, etc.*)

Il y aurait lieu de préciser un certain nombre de propriétés de ces fonctions, en exprimant des relations telles que

$$\forall v \in VECTEUR,\ longueur(v) > 0$$

etc., mais n'insistons pas. Le point important est qu'on ne fait figurer dans cette définition "fonctionnelle" du type VECTEUR que des opérations qu'on sait être vectorisables. En particulier, on a omis l'opération

$$produit\text{-}scalaire : VECTEUR \times VECTEUR \rightarrow REEL$$

Par contre, on peut introduire des choses comme

$$projection : VECTEUR \times ENTIER \rightarrow VECTEUR$$

avec la relation

$$k \geq \ell \Rightarrow composante\ (projection\ (v,\ell),\ k) = 0$$

pour tout vecteur v (c'est la projection selon les ℓ premières composantes).

Comme on l'a déjà remarqué, on pourra utiliser des notations équivalentes mais plus légères telles que + pour l'addition, P_ℓ pour la projection précédente, etc.

La règle du jeu va être d'écrire les algorithmes à l'aide des seules opérations primitives de la définition fonctionnelle. Si c'est possible, ces algorithmes seront par construction vectorisables. (On peut objecter que tout algorithme s'écrit en termes de ces primitives, puisqu'on dispose de l'opération *composante* ; mais on ne dispose pas, par contre, d'un moyen de reconstruire un vecteur à partir de ses composantes. Le résultat serait donc une collection de scalaires, et non un vecteur.)

Appliquons ces idées à l'exemple en cours. On introduit un type MATRICE :

type MATRICE :

largeur	: MATRICE → ENTIER
hauteur	: - *id* -
colonne	: MATRICE × ENTIER → VECTEUR
ligne	: - *id* -
transposition	: MATRICE → MATRICE
projection	: - *id* -

...

relations

$\forall\ m \in MATRICE, \forall\ j,\ 1 \leq j \leq largeur\ (m)$,

longueur (colonne(m,i)) = largeur(m);

...

$\forall\ \ell$, *colonne* $(P_\ell\ m,\ i) = P_\ell$(*col.*(m,i))

colonne(m,i) = ligne(transposition(m,i))

...

...

et ainsi de suite. (On remarque, d'après les seules propriétés fonctionnelles ci-dessus, que VECTEUR est un sous-type de MATRICE (les matrices à une colonne) donc que du point de vue logique (par opposition à fonctionnel) une matrice est une famille de vecteurs. Mais on peut ne pas s'en souvenir et on a même intérêt en fait à l'oublier.)

Munis de ce bagage formel, réécrivons l'algorithme de multiplication d'un vecteur par une matrice.

```
déclaration
|       a : MATRICE, x, y : VECTEURS {longueur(a) =
|       longueur (x); hauteur(a) = longueur(y)},
|       n, ℓ : ENTIERS;
données a, x;
|       n ← longueur(x) ;
|       y ← 0;
|       pour k de 1 à n répéter
(7)     |       y ← y + colonne(a, k) ∧ composante(x, k)
```

où "$\wedge$" est provisoirement, le symbole de la multiplication d'un vecteur par un scalaire. Pour simplifier, on écrira (7) désormais

$$y \leftarrow y + a^{k} \wedge x_{k} \tag{8}$$

La première remarque, évidente, est qu'il n'y a aucune raison de s'en tenir à un seul vecteur x, ou encore que x peut très bien être une matrice. On vient donc de programmer un produit matriciel

```
déclarations
|       a, x, y = MATRICES {...} , ... ;
données a, x ;
|       x ← longueur(x) ;
|       y ← 0 ;
|       pour k de 1 à n répéter
|       |       y ← y + a^k ∧ x_k
```

à condition de préciser le sens de l'opérateur $\wedge$. C' est le produit tensoriel ou extérieur bien connu d'un vecteur colonne par un vecteur ligne. On introduit donc dans la définition fonctionnelle

$$\wedge : VECTEUR \times VECTEUR \rightarrow MATRICE$$

avec les propriétés voulues

$$\forall\, x, y \in VECTEUR,$$
$$\quad hauteur\ (x \wedge y) = longueur\ (x)$$
$$\quad \ldots$$

etc., et on sait désormais vectoriser le produit matriciel, à condition que l'opération $\wedge$ soit vectorisable. Or c'est bien le cas. Sa traduction en Fortran, soit

```
      DO 1  I = 1,  longueur (x)
      DO 1  J = 1,  longueur (y)
1           Z (I, J) = X(I) * Y(J)
```

vérifie bien les conditions de vectorisation de Cray-1 exposées dans [10]. Et quant au DAP, pour prendre un exemple de parallélisme "vrai" (simultanéité), les gens d'ICL proposent une extension de Fortran où figure cet opérateur.

A titre d'exercice voici la traduction Fortran de la multiplication matricielle (Fig. 1).

Passons à l'opération inverse, c'est-à-dire à la résolution des systèmes linéaires.

3. - SYSTEMES LINEAIRES TRIANGULAIRES

Beaucoup d'algorithmes pour la résolution des systèmes linéaires consistent à se ramener à des systèmes triangulaires. On va d'abord examiner ce problème du point de vue de la vectorisation, et en profiter pour rappeler les principes de la programmation descendante [12] [13] et de la synthèse des programmes à partir des assertions finales.

Appelons TRIMAT un sous-type de MATRICE, caractérisé par

si $m \in$ *TRIMAT alors*
| *hauteur (m) = largeur (m) ;*
| $1 \leq i \leq$ *hauteur (m)* $\Rightarrow P_{i-1}\, m^i = 0$;

ce qui caractérise comme on le voit les matrices carrées triangulaires inférieures.

On se propose d'écrire un programme *résoltri* dont la spécification est la suivante

(9)
| *déclaration s : TRIMAT, x, b : VECTEURS*
| *{hauteur (s) = longueur (x) = longueur (b) = n} ;*
| *données s; x;*
| *resoltri (s, b, x)*
| $\{ \sum_{k=1,n} s^k \wedge x_k = b,$ *i.e.* $Sx = b\}$

(on omettra désormais le symbole $\wedge$).
Partant de l'assertion finale (9) nous procédons par découplage en introduisant une variable auxiliaire c et en remarquant que

$$\Sigma\, s^k x_k \Leftrightarrow (c + \sum_{k\leq n} s^k x_k = b \text{ et } c = 0)$$

$$\Leftrightarrow (c + \sum_{k\leq \ell} s^k x_k = b \text{ et } P_\ell\, c = 0) \text{ et } \ell = n)$$

$$\Leftrightarrow \underbrace{\qquad\qquad}_{(I(\ell) \quad \text{et}} \quad \ell = n)$$

Ici, I(ℓ) est un affaiblissement de la condition de sortie I(n) et I(0) peut être trivialement réalisé. Donc un "affinage" de *résoltri* consiste à écrire :

(résoltri)

```
déclaration ℓ : ENTIER;
ℓ ← 0; c ← b  {I(0)} ;
tant que ℓ < n répéter
|       ℓ ← ℓ + 1;
|       rétablir I(ℓ);
{ℓ = n  et I(ℓ)}
```

Ce programme est par construction correct car I(ℓ) étant toujours vrai à la sortie du corps de boucle "tant que", il est vrai aussi à la sortie de la boucle elle-même, ainsi que la condition d'arrêt ℓ = n; or l'intersection logique des deux est la condition de sortie voulue.

Passons à l'affinage suivant, qui consiste à détailler l'action "rétablir I(ℓ)". Il s'agit de passer de l'assertion d'entrée I(ℓ - 1) soit

$$c + \sum_{k<\ell} s^k x_k = b \quad \underline{et} \quad P_{\ell-1}\, c = 0 \tag{10}$$

à l'assertion de sortie I(ℓ), c'est-à-dire

$$c + \sum_{k<\ell} s^k x_k = b - s^\ell x_\ell \quad \underline{et} \quad P_\ell\, c = 0 \tag{11}$$

L'action nécessaire pour cela est $c \leftarrow c - s^\ell x_\ell$ après que l'on ait trouvé x_ℓ tel que $P_\ell(c - s^\ell x_\ell) = 0$. Or par hypothèse $P_{\ell-1}\, s^\ell = 0$ et $P_{\ell-1}\, c = 0$, donc cette équation se réduit à $c_\ell - s^\ell_\ell x_\ell = 0$, d'où x_ℓ .
La version finale du programme est donc, en supposant s régulière :

```
ℓ ← 0 ;  c ← b ;  {I(0)}
tant que  ℓ < n répéter
|       ℓ ← ℓ + 1 ;
|       {rétablir l'invariant de boucle I(ℓ) :}
|       |       x_ℓ ← c_ℓ / s^ℓ_ℓ
|       |       c ← c - s^ℓ x_ℓ
```

Partant d'une spécification "en termes de vecteurs" et programmant sur une "machine virtuelle" dont les instructions sont vectorisables il est normal que le produit de cette synthèse soit un programme vectorisable. La traduction Fortran de celui-ci apparaît Fig. 2. Bien entendu on parviendrait au même résultat en inversant l'ordre des boucles DO dans le programme séquentiel usuel, mais la démarche exposée ici doit s'imposer par son caractère général et systématique. Pour s'en convaincre, on va examiner le problème de la factorisation de Choleski.

4. - CHOLESKI "VECTORIEL"

Tout ce qui va suivre s'applique à la factorisation LU, avec pivotage partiel, mais on s'en tiendra aux matrices symétriques définies positives pour simplifier.

Sans répéter ce qui a déjà été dit, voici les étapes successives de la synthèse. D'abord la spécification :

(10)
$$\left|\begin{array}{l} \underline{\textit{déclarations}}\ a : \textit{MATRICE},\ s : \textit{TRIMAT}, \ldots ; \\ \underline{\textit{donnée}}\ a\ \{\textit{symétrique définie positive d'ordre } n\}; \\ \textit{Choleski} ; \\ \{a = \sum_{k \le n} s^k \wedge s^k,\ \textit{c'est-à-dire } A = S\,S^t\} \end{array}\right.$$

puis le découplage de (10), après avoir introduit la variable auxiliaire c, de type MATRICE :

$$(10) \Leftrightarrow \underbrace{((c + \sum_{k \le \ell} s^k \wedge s^k = a \ \underline{et}\ P_\ell\ c = 0)}_{(I(\ell)} \ \underline{et}\ \underbrace{\ell = n)}_{\ell = n)}$$
$$\Leftrightarrow \qquad (I(\ell) \quad \underline{et} \quad \ell = n)$$

puis le premier affinage

$\ell \leftarrow 0$; $c \leftarrow a$ $\{I(0)\}$

tant que $\ell < n$ *répéter*

| $\ell \leftarrow \ell + 1$;
| $\{c + \sum_{k<\ell} = a$ *et* $P_{\ell-1}\, c = 0$
| *rétablir* $I(\ell)$
| $\{c + \sum_{k<\ell} = a - s^{\ell} \wedge s^{\ell}$ *et* $P_{\ell}\, c = 0\}$

Pour rétablir I(ℓ) il faut faire $c \leftarrow c - s^{\ell} \wedge s^{\ell}$ après avoir trouvé s^{ℓ} tel que $P_{\ell-1}\, s^{\ell} = 0$ et que

$$P_{\ell}(c - s^{\ell} \wedge s^{\ell}) = 0$$

Or $P_{\ell-1}\, c = 0$, donc seule la ligne ℓ est concernée. On doit donc avoir

$$(c - s^{\ell} \wedge s^{\ell})_{\ell} = 0$$

c'est-à-dire

(11) $$c_{\ell} - s^{\ell}_{\ell} \wedge s^{\ell} = 0$$

donc en particulier (composante ℓ du premier membre de (11))

$$c^{\ell}_{\ell} = (s^{\ell}_{\ell})^2$$

d'où les deux instructions composant "rétablir I(ℓ)"

$$s^{\ell}_{\ell} \leftarrow rac(c_{\ell}) \; ; \; s^{\ell} \leftarrow c_{\ell}/s^{\ell}_{\ell}$$

Puisque c est symétrique (c'est aussi un invariant de boucle), $P_{\ell-1}\, c = 0$ entraîne $P_{\ell-1}\, c_{\ell} = 0$, d'où $P_{\ell-1}\, s^{\ell} = 0$.

La version finale est donc :

```
ℓ ← 0 ;  c ← a ;
tant que ℓ < n répéter
|        ℓ ← ℓ + 1 ;
|        pivot ← rac (c_ℓ^ℓ) ;
|        s^ℓ ← c_ℓ / pivot ;
|        c ← c - s^ℓ ∧ s^ℓ
```

Il manque quelques détails (traitement du cas d'erreur $c_\ell^\ell < 0$) mais de toute façon, on ne peut encore proposer de traduction Fortran à cause des problèmes de représentation physique (comment ranger en mémoire les éléments de a et s, comment tenir compte du "creux" des matrices). Celle de la Fig. 3, incomplète, est destinée à fixer les idées et à montrer comment on se débarrasse, en pratique, de la matrice auxiliaire c : puisque la partie non nulle de c correspond aux colonnes de s non encore calculées, on peut juxtaposer les éléments de s et ceux de c dans une même matrice, désignée par s (Fig. 4). Remarquer aussi comment on implante le produit extérieur en tenant compte de la symétrie.

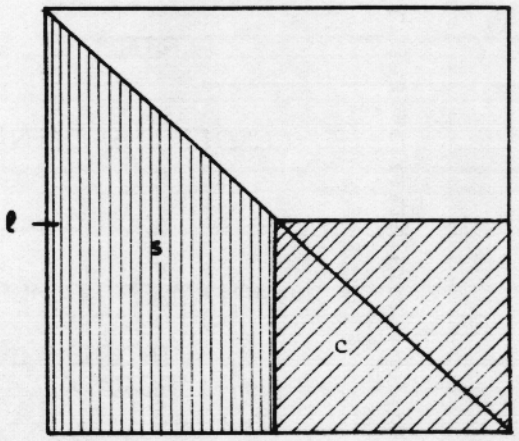

Fig. 4
Complémentarité de s et c.

5. CONCLUSION

Une nouvelle génération de machines permet d'accélérer très sensiblement les calculs"vectoriels" (opération de type VECTEUR × VECTEUR × ... → VECTEUR qui pourraient se faire parallèlement sur les composantes de même indice). On a donné une définition fonctionnelle du type VECTEUR, limitée aux seules opérations "vectorielles". On a montré, sur l'exemple de l'algorithme de Choleski, comment à partir d'une spécification de programme basée sur cette définition fonctionnelle, le procédé de synthèse descendante conduit automatiquement à un programme correct vectorisable (bénéficiant de toutes les possibilités nouvelles de ces machines).

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE
                          R E S T R I
     .                    (N, S,  B, X)
     .
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C                         FONCTION
C
C     RESOLUTION D'UN SYSTEME LINEAIRE TRIANGULAIRE INFERIEUR PAR
C     L'ALGORITHME VECTORISABLE
C
C     DONNEES
C
          INTEGER    N
C                                    Ordre de la matrice S
          REAL       S    ( 1 )
C                                    Tableau contenant les éléments de la
C                                    matrice S. Sij se trouve à la posi-
C                                    tion ((J - 1)(2N - J) + 2I)/2
          REAL       B    ( 1 )
C                                    Vecteur de longueur N, second membre.
C     RESULTATS
C
          REAL       X    ( 1 )
C                                    Vecteur de longueur N, solution de
C                                              S x  =  b
C     CONDITIONS D'ENTREE
C                                    S régulière, i.e. S(I*(I + 1)/2) /= 0
C                                    pour 1 <= I <= N.
C-------------------------------------------------------------------------
C
C     VARIABLES LOCALES
C
C          INTEGER L, I, ADRLL, ADRLI
           REAL XL
C
C     /l <--- 0 ; C <--- B/
        L = 0
        DO 1 I = 1, N
             X(I) = B(I)
1
C            -- Le vecteur C est logé dans la partie i > l, non en-
C            -- re calculée, du tableau X, puisque Ci = 0 pour i<= l
C     (* I(l) : C + Sigma(Sk Xk) = B  et  PlC = 0, invariant de boucle)
C     tant que l < n, répéter
2     IF ( L .GE. N) GO TO 5
C           l <--- l + 1 ;
             L = L + 1
C         /rétablir l'invariant de boucle I(l)/
C            Xl <--- Cl/Sll
                ADRLL = ((L - 1)*(2*N - L) + 2*L)/2
                XL = X(L)/S(ADRLL)
C            C  <--- C - Sl Xl
                ADRLI = ADRLL
                LP1 = L + 1
                X(L) = XL
                IF (LP1 .GT. N) GO TO 4
C               -- Les X(I) ci-dessous contiennent les valeurs de C
                DO 3 I = LP1, N
                     ADRLI = ADRLI + 1
3                    X(I) = X(I) - XL*S(ADRLI)
4               CONTINUE
          GO TO 2
5     CONTINUE
C     (* I(N) :  Sigma(Sk Xk, k = 1, ..., N) = B  *)
      RETURN
      END
```

Figure 1

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE
     .                        P R O M A T
     .
     .                (N, M, P, LCX, LCY, LCZ, X, Y, Z)
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C                                FONCTION
C
C       PRODUIT MATRICIEL Z = X*Y, VERSION VECTORISABLE. LES MATRICES
C     SONT REPRESENTEES DE LA FACON USUELLE DANS DES TABLEAUX BIDIMEN-
C     SIONNES CORRESPONDANT A X, Y, Z DANS LE PROGRAMME APPELANT.
C
C                           LISTE DES ARGUMENTS
C
C ***************************************************************************
C        TYPE        NOM      TABLEAU ?                  SIGNIFICATION
C ***************** ******** ************ ***************************************
C  DONNEES
C
      INTEGER        N                      | NOMBRE DE LIGNES DE X
C
C
      INTEGER        M                      | NOMBRE DE COLONNES DE Y
C
C
      INTEGER        P                      | NOMBRE DE COLONNES DE X (ET
C                                           |  DE LIGNES DE Y)
      REAL           X         (LCX, P)
      REAL           Y         (LCY, M)     | TABLEAUX CONTENANT X ET Y
C
      INTEGER        LCX, LCY, LCZ          | LONGUEURS EFFECTIVES, DANS LE
C                                           | PROGRAMME APPELANT, DES CO-
C                                           | LONNES DE X, Y ET Z.
C
C  RESULTATS
C
      REAL           Z         (LCZ, M)
C
C///////////////////////////////////////////////////////////////////////////
C     VARIABLES LOCALES :
            INTEGER I, J, K
C
C     /INITIALISER Z/
           DO 1 I = 1, N
           DO 1 J = 1, M
1                Z(I, J) = 0.
C     /POUR CHAQUE COLONNE DE X, ET POUR LA LIGNE HOMOLOGUE DE Y, FAIRE
C     /LE PRODUIT EXTERIEUR ET L'AJOUTER A Z :
           DO 3 K = 1, P
C               /CES OPERATIONS SONT VECTORISABLES :
                DO 2 I = 1, N
                DO 2 J = 1, M
2                    Z(I, J) = Z(I, J) + X(I, K)*Y(K, J)
3               CONTINUE
      RETURN
      END
```

Figure 2

Résolution d'un système triangulaire par la méthode "vectorielle"

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE
     .                        C H O V E C
     .                      (N, A,  S, NDP)
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C                                FONCTION
C
C     FACTORISATION DE CHOLESKI D'UNE MATRICE SYMETRIQUE PAR
C     L'ALGORITHME VECTORISABLE
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C     DONNEES
C
            INTEGER     N
C                                    Ordre de la matrice A
            REAL        A    ( 1 )
C                                    Tableau contenant les éléments de la
C                                    matrice A. Aij se trouve à la posi-
C                                    tion ((J - 1)(2N - J) + 2I)/2 (range-
C                                    ment "symetrique par colonnes")
C     RESULTATS
C
            REAL        S    ( 1 )
C                                    Tableau contenant les éléments de la
C                                    matrice S. Sij se trouve à la posi-
C                                    tion ((J - 1)(2N - J) + 2I)/2.
C                                    Au retour, si NDP = N, A = S (=S)
C
            INTEGER     NDP
C                                    Nombre de colonnes effectivement
C                                    prises en compte dans la factorisa-
C                                    tion.SiNDP < N, c'est qu'un radicande
C                                    <= 0 est apparu à la colonne NDP + 1.
C-------------------------------------------------------------------------
C
C     VARIABLES LOCALES
C
           INTEGER L, NNP1S2, ADRLL, ADRJL, ADRJJ, I, J, LP1
           REAL PIVOT, MUL
C     FONCTION ARITHMETIQUEMENT DEFINIE :
           INTEGER ADRESS
           ADRESS(I, J) = ((J - 1)*(2*N - J) + 2*I)/2
C
```

Figure 3

Choleski "vectoriel" (en-tête du programme). Cf. p. 16.

```
C-----------------------------------------------------------------------
C
            NDP = 0
Cc      ℓ <--- 0 ;
            L = 0
Cc      C <--- A ;
            NNP1S2 = (N*(N + 1))/2
            DO 1 I = 1, NNP1S2
                S(I) = A(I)
1           -- Le tableau S contient à la fois C et S.
C
Cc      tant que ℓ < n répéter
        IF (L .GE. N) GO TO 6
Cc          ℓ <--- ℓ + 1 ;
                L = L + 1
                ADRLL = ADRESS(L, L)
Cc          pivot <--- rac(Cℓℓ);
                RADIC = S(ADRLL)
                IF (RADIC .LE. 0.) GO TO 6
                    -- Exception si A n'est pas définie positive
C
                PIVOT = SQRT(RADIC)
                NDP = L
Cc          Sℓ <--- Cℓ/pivot ;
                DO 2 I = L, N
                    S(ADRLL + I - L) = S(ADRLL + I - L)/PIVOT
2
Cc          C <--- C - Sℓ >< Sℓ ;
                LP1 = L + 1
                IF (LP1 . EQ. N) GOTO 5
                DO 4 J = LP1, N
                    ADRJJ = ADRESS(J, J)
                    ADRJL = ADRESS(J, L)
                    MUL = S(ADRJL)
                    DO 3 I = J, N
                        S(ADRJJ+I-J) = S(ADRJJ+I-J) - MUL*S(ADRJL+I-J)
                        -- Cette boucle est la seule vectorisable
                        CONTINUE
3                   CONTINUE
4               CONTINUE
5       RETURN
6       END
```

Figure 3 (suite)

Choleski "vectoriel" (corps du programme).

REFERENCES

[1] *IFIP Congress*, Ljubljana, Août 1971, North-Holland, Amsterdam.

[2] *Conference on Elliptic Solvers*, Santa-Fe, 30 juin - 2 juillet 1980.

[3] CONPAR 81, *Conference on Analysing Problem-classes and Programming for Parallel Computing*, Nuremberg, 10-12 Juin 1981.

[4] E.W. DIJKSTRA : *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[5] C.A.R. HOARE : An Axiomatic Basis for Computer Programming, *Comm. ACM, 12*, 10 (1969), pp. 576-582.

[6] C.A.R. HOARE; "Communicating Sequential Processes", *Comm. ACM, 21*, 8 (1978), pp. 169 - 180.

[7] G. KAHN : "The Semantics of a Simple Language for Parallel Programming", in. *Proc. IFIP Congress* 1974, North-Holland, Amsterdam, 1974.

[8] K. MARX : *Le Capital* (critique de l'Economie Politique), I, 14, Ed. Sociales, Paris, 1962.

[9] B. MEYER et C. BAUDOIN : *Méthodes de Programmation*, Eyrolles, Paris, 1978.

[10] B. MEYER : Un calculateur vectoriel, le Cray-1 et sa programmation, note EDF HI 3452/01, Mai 1980.

[11] N. WIRTH : "Program Development by Stepwise Refinements", *Comm. ACM, 14*, 4 (1971), pp. 221-227.

- II -

Vectorisation de l'algorithme de Gauss-Seidel

1 Introduction

On illustre souvent les notions de parallélisme dans les algorithmes numériques par l'exemple simple des deux algorithmes itératifs de Jacobi et de Gauss-Seidel pour la résolution des systèmes linéaires. Le premier présente un "parallélisme" dans l'exécution des opérations, que le second n'a pas. Mais vectorisation et parallélisme, bien qu'étroitement liés, ne sont pas des concepts équivalents. Il est très simple de constater, comme nous allons le faire ci-dessous, que les deux algorithmes sont également vectorisables. Mais cela n'est apparent que si on s'astreint à "penser en termes de vecteurs", au sens que l'on va voir. Au contraire, si l'on part d'un programme séquentiel classique écrit en Fortran, et que l'on cherche à le transformer en une version vectorisable, on y parvient aisément dans le cas de Jacobi, mais non dans le cas de Gauss-Seidel. Tous ces points sont illustrés ci-dessous, échantillons de programmes à l'appui.

2 Jacobi et Gauss-Seidel

Soit b un vecteur donné, de dimension n, et A une matrice n X n, écrite sous la forme

(1) $$A = D - E - F$$

où D est la diagonale de A, - E et - F ses parties sous-diagonale et sur-diagonale. On veut résoudre l'équation

(2) $$Ax = b \ .$$

L' algorithme de Jacobi consiste en ceci :

(3)
```
x := 0 ;
répéter
    x := D^-1((E + F)x + b)
```

alors que celui de Gauss-Seidel s'écrit(*)

(4)
$$x := 0 \ ;$$
$$\underline{répéter}$$
$$\quad x := (D - E)^{-1}(F x + b)$$

Dans les deux cas, le corps de boucle est exécuté jusqu'à ce qu'un certain critère de convergence soit satisfait. La convergence est assurée à certaines conditions sur A, très fréquemment rencontrées en pratique, pour lesquelles nous renvoyons à /1/. Ces conditions impliquent en particulier que les termes diagonaux de A sont tous différents de zéro.

Dans (3) comme dans (4), on rencontre une multiplication d'un vecteur par une matrice, et une résolution de système linéaire triangulaire (et même diagonal dans le cas de Jacobi). Nous avons déjà montré dans /2/ et /3/ comment ces deux opérations peuvent s'exprimer comme enchaînement d'opérations élémentaires de type *VECTEUR* x *VECTEUR* → *VECTEUR*, avec parallélisme par rapport aux différentes composantes. Or de telles opérations peuvent être "vectorisées", c'est-à-dire écrites en Fortran sous forme de boucles que le compilateur peut traduire de façon particulièrement efficace, à l'intention d'une machine comme le Cray-1, en utilisant les registres vectoriels et les tuyaux à bits.

Avant de développer les algorithmes (3) et (4), un point doit être signalé. La stratégie de vectorisation d'un système linéaire triangulaire dépend largement de la structure de la matrice. Dans /2/ et /3/, où les matrices étaient supposées pleines, on les considérait comme collections de vecteurs-colonnes. On verra dans /4/ que dans le cas de matrices-bandes, il est plus naturel de considérer les parallèles à la diagonale comme les vecteurs dont A est composée, et que cela conduit à des algorithmes vectorisables tout différents, basés sur l'idée de la "réduction cyclique". Mais cela ne change rien à la remarque fondamentale : Gauss-Seidel s'exprime en termes de deux opérations primitives, (à savoir multiplication matrice-vecteur et système triangulaire), donc peut être vectorisé au même titre que ces deux opérations.

(*) Dans les deux cas, on remplace Ax = b par une équation de point fixe, soit Dx = (F + E)x ou (D - E) x = Fx + b, que l'on résout par la méthode itérative naturelle.

3 Développement des deux algorithmes

Un langage de programmation peut être considéré comme une "machine virtuelle", caractériséee par ses types et des opérations possibles sur les types. Sur une même machine, telle que le Cray-1, on peut disposer soit de Fortran, donc d'une machine virtuelle ne comportant que les types numériques et les opérations arithmétiques usuelles, soit de langages plus évolués comme Pascal, où le type *VECTEUR* serait explicitement fourni ; autre possibilité, APL. Soit d'abord une machine virtuelle de cette dernière catégorie.

On dispose donc du type *VECTEUR*, avec les opérations d'addition, de changement de signe, de multiplication et de division, toutes du type $VECTEUR \times VECTEUR \to VECTEUR$, et des opérations $REEL \times VECTEUR \to VECTEUR$ comme la multiplication ou la division d'un vecteur par un réel, etc. Si n est la longueur d'un vecteur, on peut définir pour tout entier i, compris entre 1 et n, l'opération *composante*. Pour la brièveté, on notera x_i ce qui devrait s'écrire formellement *composante*(x, i). Quand aux *MATRICES*, nous renvoyons à /2/ pour une description formelle. Il suffit ici de dire que si a est un objet de type *MATRICE*, a^j dénote la j-ème colonne de a, qui est un objet de type *VECTEUR*. On aura aussi besoin de considérer les colonnes de F ou de D - E comme des *VECTEURS*. Pour cela, on introduit l'opération P, de type $ENTIER \times VECTEUR \to VECTEUR$, et on note $P_i v$ la projection du vecteur v sur le sous-espace des i premières composantes. Avec cette notation, la colonne j de F, par exemple, est $P_{j-1} a^j$, où a est l'objet de programme, de type *MATRICE*, qui correspond à A. Enfin, l'opération $diag : MATRICE \to VECTEUR$, permet de passer d'une matrice au vecteur de ses composantes sur la diagonale.

Avec ces notations, l'algorithme de Jacobi s'écrit comme suit :

(5)

```
données a : MATRICE, b : VECTEUR {longueur(b) = ordre(a) = n} ;

variables x, y : VECTEURS ;

x := 0 ;
répéter
|   y := b ;
|   pour j de 1 à n répéter
|   |   y := y - x_j a^j
|   x := x + y/diag(a) ;
```

et celui de Gauss - Seidel, d'abord par un premier affinage de (4) :

(6)
```
x := 0 ;
répéter
    y := F x + b ;
    x := (D - E)^-1 y
jusqu'à convergence
```

puis, par un deuxième affinage (nécessaire, puisque (6) n'est pas encore exécutable sur la machine virtuelle que nous avons définie) :

(7)
```
x := 0 ;
répéter
    y := b ;
    pour j de 1 à n répéter
        y := y - x_j P_{j-1} a^j
    {y = F x + b}
    x := y ;
    pour j de 1 à n répéter
        x_j := x_j / a_j^j ;
        x := x - x_j (1 - P_j) d^j
    {x = (D - E)^-1 y}
jusqu'à convergence
```

Quelques raffinements permettraient de se passer complètement du vecteur auxiliaire y, mais cela relève de la représentation interne, et sera mieux vu sur la version suivante. Remarquons que (7) demande deux fois plus d'opérations vectorielles que (5) (Jacobi), mais qu'elles portent sur des vecteurs de longueur moyenne $n/2$ au lieu de n. On sait par ailleurs que Jacobi est asymptotiquement deux fois moins rapide que Gauss-Seidel, donc le rapport d'efficacité n'est pas évident a priori. Il dépend de caractéristiques de la machine physique telles que le rapport du temps d'amorçage des boucles vectorielles au temps d'exécution d'une instruction d'une telle boucle. Quoi qu'il en soit, ce n'est pas l'algorithme de Gauss-Seidel ni celui de Jacobi que l'on utilise en pratique, mais la surrelaxation, que le lecteur

pourra écrire sans peine en modifiant (7) (garder y au lieu de x dans le corps de boucle, et ajouter l'instruction $x := x + oméga*(y - x)$ en fin de boucle).

Passons maintenant à une machine virtuelle semblable à la machine Fortran, où il n'y a que des instructions scalaires. Pour abréger, seul le corps de la boucle *répéter* la plus externe est indiqué. D'abord Jacobi :

(8)

```
{Jacobi, un pas sur machine scalaire}
pour i de 1 à n répéter
|   y(i) := b(i)
pour j de 1 à n répéter
|   pour i de 1 à n répéter
|   |   y(i) := y(i) - x(j)*a(i, j)
pour i de 1 à n répéter
|   x(i) := x(i) + y(i)/a(i, i)
```

puis Gauss-Seidel :

(9)

```
{Gauss-Seidel, un pas sur machine scalaire}
pour j de 1 à n répéter
|   pour i de 1 à j - 1 répéter
|   |   x(i) := x(i) - x(j)*a(i, j)
|   x(j) := b(j)
pour j de 1 à n répéter
|   x(j) := x(j)/a(j, j) ;
|   pour i de j + 1 à n répéter
|   |   x(i) := x(i) - x(j)*a(i, j)
```

(Comparer (7) et (9) pour voir comment on peut faire l'économie du tableau auxiliaire représentant y.)

Si le programme (9) est confié à une machine capable de vectoriser

les boucles *pour* les plus internes, cette machine ne fera rien d'autre que reconstituer (7) à partir de (9). Il y a donc quelque absurdité à compiler en quelque sorte "à la main" le programme (7) pour produire (9), alors que la machine fera tout aussitôt l'opération inverse. Mais c'est inévitable si l'on programme en Fortran (du moins en Fortran actuel (66 ou 77) ; des extensions de Fortran comportant les opérations vectorielles sont d'ores et déjà à l'étude, et on peut espérer une situation plus normale dans quelques années). Au moins a-t-on ainsi limité les dégâts en ne proposant au compilateur qu'une tâche à sa portée. Comme on va le voir, le problème serait tout différent si on ne s'astreignait pas dès le début à "penser vecteurs".

4 Les versions séquentielles classiques des deux algorithmes

Partons de l'idée heuristique suivante : le système Ax = b comporte n équations, que l'on cherche à satisfaire successivement, pour chaque indice i, en ne jouant que sur la valeur de l'inconnue x_i. Dans le cas de Jacobi, ce processus s'opère simultanément sur les n équations (après quoi on actualise en même temps les n inconnues), alors que Gauss-Seidel prend en compte la nouvelle valeur de x_i dès qu'elle est trouvée.

La traduction algorithmique de cette idée est, pour Jacobi,

(10)

```
répéter
|   pour i de 1 à n répéter
|   |   y(i) := b(i) ;
|   |   pour j de 1 à n répéter
|   |   |   y_i := y_i - a(i, j)*x(j)
|   pour i de 1 à n répéter
|   |   x(i) = x(i) + y(i)/a(i, i)
```

et pour Gauss-Seidel :

(11)

```
répéter
|   pour i de 1 à n répéter
|   |   s := b(i) ;
|   |   pour j de 1 à n répéter
|   |   |   s := s - a(i, j)*x(j)
|   |   x(i) := x(i) + s/a(i, i)
```

Essayons maintenant de "passer du séquentiel au vectoriel", c'est-à-dire de retrouver (8) en partant de (10), et (9) en partant de (11). Dans le premier cas (Jacobi), c'est très simple : il suffit de faire éclater la première boucle i en deux, et de permuter les indices i et j dans le second morceau. On vérifie sans peine que la sémantique n'est pas changée par cette opération. Par contre, il faut beaucoup d'ingéniosité, pour ne pas dire de la ruse et une certaine perversité, pour passer de (11) à (9). Ces qualités se rencontrent couramment chez les humains, mais on n'a pas encore pleinement réussi à les enseigner aux compilateurs.

On voit sur cet exemple que la vectorisation est tout autre chose qu'un exercice de permutation des i et j, même si le bon résultat s'obtient souvent de cette façon, ou une stratégie de transformations de programmes à implanter dans les compilateurs. Il est sûrement plus productif de penser les algorithmes en termes de vecteurs dès le début.

A ce propos, une dernière remarque s'impose. Nous n'avons pas voulu jusqu'ici considérer l'opération *produit-scalaire*, de type *VECTEUR* x ... ... *VECTEUR* → *REEL*, comme une opération vectorielle (au sens : opération qui s'exécute rapidement sur un ordinateur vectoriel). Or ceci est relatif : les performances du compilateur CFT de Cray, par exemple, évoluent. Actuellement, le temps d'exécution d'un produit scalaire sur deux vecteurs assez longs est du même ordre que le temps d'exécution d'une multiplication et d'une addition vectorielles. Si l'on accepte le produit scalaire parmi les opérations vectorielles, il devient possible d'obtenir (10) et (11) en "pensant vecteurs" : il suffit de considérer les vecteurs lignes de A (au lieu des vecteurs colonnes) et de développer (3) et (4) en conséquence.

5 Versions Fortran

Les deux programmes présentés ci-dessous effectuent un pas de l'algorithme de Gauss-Seidel. Pour les utiliser, il faut les intégrer à une boucle de répétition, comportant un test d'arrêt, et éventuellement une surrelaxation, facile à mettre en œuvre.

```
      SUBROUTINE PASGS(LCOL, N, A, B,. X)
C
C     /UN PAS DE GAUSS-SEIDEL, VERSION SEQUENTIELLE/
C
C     /DONNEES :
            INTEGER N                  -- ordre de A
            INTEGER LCOL               -- longueur des colonnes du
C                                      -- tableau A dans l'appelant
            REAL A(LCOL, N), B(N), X(N)
C -------------------------------------------------------------------
C
      REAL XAUX                        -- variable locale
C
      DO 1 I = 1, N
           XAUX = B(I)
           DO 1 J = 1, N
                XAUX = XAUX - A(I, J)*X(J)
1          X(I) = X(I) + XAUX/A(I, I)
      RETURN
      END
```

Ceci concerne évidemment une matrice pleine, point sur lequel nous revenons en conclusion. Ce programme est très simple, et il n'y a pas de raison de chercher autre chose si les produits scalaires sont efficacement vectorisés.

Si ce n'est pas le cas, on préfèrera la version vectorielle (qui est par contre un peu moins efficace dans le cas des machines séquentielles, à cause du temps perdu aux démarrages de boucles, deux fois plus nombreux) :

```
      SUBROUTINE PASGSV(LCOL, N, A, B, X)
C
C     /UN PAS DE GAUSS-SEIDEL, VERSION VECTORIELLE/
C
                .................          -- même entête
C -------------------------------------------------------------------
C
      DO 2 J = 1, N
           JM1 = J - 1
           DO 1 I = 1, JM1                 -- Norme FORTRAN 77
1               X(I) = X(I) - X(J)*A(I, J)
2          X(J) = B(J)
      DO 4 J = 1, N
           X(J) = X(J)/A(J, J)
           JP1 = J + 1
           DO 3 I = JP1, N                 -- Id.
3               X(I) = X(I) - X(J)*A(I, J)
4          CONTINUE
      RETURN
      END
```

Il est possible que l'on ait intérêt, du point de vue de l'efficacité, à sortir l'instruction d'affectation de B à X de la boucle 2, mais on peut penser que c'est au compilateur que cette tâche doit incomber. Remarquons à nouveau la difficulté qu'il y aurait à produire *PASGSV* par des transformations de programme sur *PASGS*.

6 Vers la vectorisation "automatique" de Gauss-Seidel ?

Dans ce qui va suivre, et afin d'écarter de la discussion des difficultés secondaires, on se limitera à la résolution par Gauss-Seidel de Ax = b en supposant tous les termes diagonaux de A égaux à 1. De la sorte, les divisions disparaissent des programmes. Pour la même raison, on suppose b = 0, et on part de x non nul, examinant donc l'algorithme de relaxation qui ramène x à 0.

Un pas de Gauss-Seidel "par lignes" s'écrit :

```
pour i de 1 à n répéter
|   pour j de 1 à i-1  répéter
|   |   x(i) := x(i) - a(i, j)*x(j)
|   pour j de i + 1 à n répéter
|   |   x(i) := x(i) -a(i, j)*x(j)
```

et un pas "par colonnes" :

```
pour j de 1 à n répéter
|   pour i de 1 à j - 1 répéter
|   |   x(i) := x(i) - a(i, j)*x(j)
pour j de 1 à n répéter
|   pour i de j + 1 à n répéter
|   |   x(i) := x(i) - a(i, j)*x(j)
```

Sous ces formes "épurées", on va pouvoir comprendre comment ces deux algorithmes - dont nous savons par ailleurs qu'ils sont équivalents - peuvent se déduire l'un de l'autre. En effet, si l'on note $A_{i,j}$ l'action $x(i) := x(i) - a(i, j)*x(j)$, ces deux algorithmes apparaissent comme la réalisation successive des opérations $A_{i,j}$ correspondant à tous les couples i, j tels que $i \neq j$, mais dans des ordres différents. La version par lignes correspond, pour n = 4, à l'ordre

1,2 1,3 1,4 2,1 2,3 2,4 3,1 3,2 3,4 4,1 4,2 4,3

et celle par colonnes à l'ordre

1,2 1,3 2,3 1,4 2,4 3,4 2,1 3,1 4,1 3,2 4,2 4,3 .

Or, nous constatons que deux actions $A_{i,j}$ et $A_{k,l}$ sont permutables, autrement dit que les séquences d'actions

$$x(i) := x(i) - a(i, j)*x(j) ;$$
$$x(k) := x(k) - a(k,l)*x(l)$$

d'une part, et

$$x(k) := x(k) - a(k, l)*x(l) ;$$
$$x(i) := x(i) - a(i, j)*x(j)$$

d'autre part, sont équivalentes, à condition (suffisante) que $i \neq l$ et $j \neq k$. On démontrera donc que les deux ordres ci-dessus sont équivalents en prouvant qu'on peut passer de l'un à l'autre par des permutations de couples adjacents sans violer la condition ci-dessus. On s'en convainc aisément.

Le problème, si l'on veut automatiser cette démarche, comporte deux aspects : trouver des critères permettant de permuter deux instructions, et savoir passer d'un ordre à un autre, meilleur selon un certain critère (ici, celui de la vectorisation possible ou non), par des permutations permises. C'est le genre de problèmes qu'on résout lorsqu'on écrit un compilateur optimiseur. Malheureusement, des critères de permutabilité assez permissifs sont difficiles à trouver. Par exemple, ici, c'est à cause de la commutativité de l'addition que $A_{i,j}$ et $A_{i,k}$ sont permutables, une analyse sémantique assez poussée est donc nécessaire.

Cette discussion montre les limites de l'automatisation du processus de vectorisation.

7 Conclusion

L'exercice auquel nous nous sommes livrés est un peu académique dans la mesure où les algorithmes itératifs envisagés s'appliquent le plus souvent à des matrices creuses. Toutefois :

- Il y a toute une classe de problèmes où l'on rencontre de grandes matrices pleines, pour lesquelles la relaxation est la seule méthode commode (pour des raisons de place). Voir /5/. Dans ces cas, l'analyse ci-dessus s'applique.

- Pour ces mêmes problèmes, les algorithmes en faveur sont du type "gradient conjugué pré-conditionné" (qui malheureusement ne s'applique pas aux problèmes envisagés dans /5/). On peut les vectoriser par une démarche analogue.

- Pour des matrices-bandes, si la bande est assez large, on peut appliquer ce qui précède au prix de modifications faciles. L'essentiel est que les termes d'une même colonne forment un "ensemble régulier de données", au sens de /6/.

Le point important, toutefois, est qu'on doit partir de (4). Donc, si l'on sait vectoriser les résolutions de systèmes linéaires triangulaires, on sait vectoriser Gauss-Seidel et donc les méthodes de surrelaxation. Or, si A est une matrice obtenue par différences finies, ou par éléments finis sur un maillage régulier, on sait vectoriser la résolution du système associé à D - E (voir /4/). Le "retour aux différences finies", que toutes ces considérations semblent suggérer, sera peut-être un nouvel exemple de l'influence déterminante des facteurs matériels sur l'évolution de l'Analyse Numérique.

Références

/1/ R.S. Varga, Matrix Iterative Analysis, Prentice-Hall, Englewood Cliffs, N.J., 1962.

/2/ A. Bossavit, "La vectorisation de Choleski", Note EDF HI 3542/02 et Bulletin de la DER, à paraître, 1981.

/3/ A. Bossavit et B. Meyer, "Methods for Vector Programming", Note EDF HI 3694/01, février 1981.

/4/ A paraître (cf. forum de programmation du 11 Mars 1981, Clamart).

/5/ J.C. Vérité, "Projet de code TRIFOU (calcul des courants de Foucault en dimension 3)", Note EDF HI 3729/02, mars 1981.

/6/ B. Meyer, "Un ordinateur vectoriel : le Cray-1, et sa programmation", Note EDF HI 3452/01, juin 1980.

VOS COMMENTAIRES S'IL VOUS PLAIT !

Votre opinion nous est précieuse. Aidez-nous à améliorer nos produits en renvoyant cette feuille à Mme. Fumadelles, Division APCOL, Clamart, après y avoir porté vos remarques.

NOTE : CRAY n° 6

TITRE : VECTORISATION DE QUELQUES ALGORITHMES NUMERIQUES

VERSION : Décembre 1981

Votre nom :

Votre adresse précise :

Désirez-vous

- **recevoir à l'avenir les nouvelles notes de la série** CRAY :

  ☐ oui ☐ non

- **recevoir dès maintenant les notes** CRAY n° :

Vos commentaires sur cette note CRAY :