ÉLECTRICITÉ DE FRANCE
DIRECTION DES ÉTUDES ET RECHERCHES

Service Informatique
et Mathématiques Appliquées

Le 5 Mai 1980

DEPARTEMENT METHODES ET MOYENS
DE L'INFORMATIQUE

1, Avenue du Général de Gaulle
92141 CLAMART

Tél 645.21.61

Bertrand MEYER

UN CALCULATEUR VECTORIEL : LE CRAY-1 ET SA PROGRAMMATION

(Atelier logiciel n° 24)

HI/3452-01

70 pages

Version 2 : le 4 juin 1980
Version 3 : le 1er avril 1981
Version 4 : le 1er janvier 1982

B. MEYER

UN CALCULATEUR VECTORIEL : LE CRAY-1 ET SA PROGRAMMATION

Atelier logiciel n°24

Résumé : On présente les traits les plus originaux de l'ordinateur Cray-1 et les particularités qu'ils entraînent pour sa programmation efficace. Une série de règles, nécessaire et suffisante, est donnée pour l'utilisation des possibilités du calcul vectoriel.

		Pages
I	INTRODUCTION	6
II	PLACE D'UN CRAY-1 DANS UN CENTRE DE CALCUL	7
III	LE MATERIEL : CARACTERISTIQUES ORIGINALES DU CRAY-1	9
	III.1. - Rien ne sert de courir : il faut partir ensemble	9
	III.2. - La mémoire	13
	III.3. - Les unités fonctionnelles	16
	III.4. - Les tampons d'instructions	17
	III.5. - Les registres	19
	III.6. - Entrées et sorties	21
	III.7. - Le calcul vectoriel	21
	III.8. - En guise de synthèse	25
IV	AVANT DE PROGRAMMER VECTORIELLEMENT	27
	IV.1. - Qu'est-ce qu'un programme vectoriel ?	27
	IV.2. - Politique de vectorisation	27
	IV.3. - Problèmes de langage	29
	IV.4. - Le compilateur Fortran et la vectorisation	30
V	LES GRANDES TECHNIQUES VECTORIELLES	33
	V.1. - Les conditions de la vectorisation	33
	V.2. - Séries continues	34
	V.3. - Opérations primitives	35
	V.4. - Ensembles réguliers de données	39
	V.5. - La dépendance arrière	41
	V.5.1. - Définition	41
	V.5.2. - Exemples de dépendances arrière; solutions	42
	V.5.3. - Les opérations de réduction et la pseudo-vectorisation	45
	V.6. - La dépendance croisée	45
	V.7. - Directives du compilateur - Principe de parallélisme	47
	V.8. - L'adressage indirect	49

ANNEXE A : PRINCIPALES CARACTERISTIQUES DU CRAY-1 52

ANNEXE B : PRINCIPALES CARACTERISTIQUES DU LANGAGE FORTRAN DU CRAY-1 53

ANNEXE C : INSTRUCTIONS COMMANDE POUR L'EXECUTION D'UN TRAVAIL SUR CRAY-1 55

ANNEXE D : DEFIN

ANNEXE D : DEFINITION FORMELLE DE LA DEPENDANCE 62

ANNEXE E : RECAPITULATION DES REGLES PROPOSEES 63

BIBLIOGRAPHIE

NOTE SUR LA VERSION 4

La préparation des versions 3 et 4 de cette note a permis :

- de corriger quelques erreurs (en particulier, la définition des "éléments réguliers" au § V.4 était inexacte);

- de prendre en compte l'évolution du logiciel et des publications Cray, en particulier l'arrivée de la version 1.09 du compilateur Fortran avec les possibilités de pseudo-vectorisation des opérations de réduction (cf. V.5.3).

Elle a bénéficié de nombreuses conversations avec des responsables de la société Cray à Mendota Heights, Chippewa Falls et Clamart. Je remercie en particulier L. Higbie, R. Nelson, I. Qualters et D. Robbe.

Signalons qu'un document audiovisuel (bande vidéo) d'une demi-heure a été réalisé à partir de cette note et sous le même titre. Il présente la machine et sa programmation sous une forme imagée. Pour tout renseignement, s'adresser à E. de Drouas, ou à l'auteur.

Depuis sa première parution, cette note a été complétée par de nombreux autres documents EDF sur le Cray (langage de commande, vectorisation, etc.). On se reportera aux références signalées par un astérisque dans la bibliographie.

I INTRODUCTION

Le Cray-1, dont un exemplaire, exploité en commun par EDF et CISI, est disponible au Centre de Calcul des Etudes et Recherches d'EDF à Clamart, est un ordinateur destiné au calcul scientifique. Conçu grâce à des techniques de pointe en matière d'architecture et de construction de machines, il permet en particulier le traitement efficace de séries de données ou *vecteurs*. Selon les représentants de la firme qui l'a conçu, il s'agit, même en mode scalaire, du "plus rapide calculateur aujourd'hui disponible" [Dungworth 79].

Dans cette note de présentation, nous décrivons la place d'un calculateur de ce type dans un centre de calcul (section II); nous introduisons ensuite brièvement les concepts les plus originaux de l'architecture du Cray-1, pour autant qu'ils importent aux programmeurs (section III); enfin, après avoir indiqué les précautions méthodologiques nécessaires (section IV), nous discutons les méthodes permettant de programmer en Fortran de façon à tirer au mieux parti des possibilités de *vectorisation* offertes par cette machine (section V). En annexe, sont fournies une bibliographie, la récapitulation des caractéristiques de base de la machine, de celles du Fortran proposé, les cartes de contrôle nécessaires, des définitions mathématiques de quelques concepts délicats introduits à la section V, et la liste des règles proposées dans le texte.

Un mot de précaution : la présente note tente une synthèse entre les aspects matériels et logiciels du Cray-1; on a tenté de relier les règles de la programmation efficace aux contraintes du calcul vectoriel, bien qu'aucun document synthétique n'existât chez le constructeur. Il a donc fallu interpréter, au risque de se tromper.

La section IV (vectorisation) a bénéficié de nombreuses suggestions d'E. de Drouas, qui est par ailleurs l'auteur de l'annexe D (cartes de contrôle).

II PLACE D'UN CRAY-1 DANS UN CENTRE DE CALCUL

Un ordinateur très puissant (on parle souvent de "super-ordinateur") tel que le Cray-1 vise un type de traitement bien précis : le calcul scientifique lourd, vectoriel en particulier; il est très efficace pour cette application, mais non pour l'ensemble des tâches d'"intendance" auxquelles les calculateurs consacrent ordinairement une bonne partie de leur puissance (dialogue, communication, gestion de ressources, manipulations complexes de fichiers, etc.).

Aussi le Cray-1 n'est-il pas destiné à constituer le "noyau" d'un grand centre de calcul, en remplacement d'un ordinateur plus classique comme un IBM 3081. Sa place normale (figure 1) est celle d'un calculateur arrière *(back-end processor)* venant "épauler" un tel ordinateur, déjà présent auparavant, et qui sera désormais considéré comme frontal *(front-end)*.

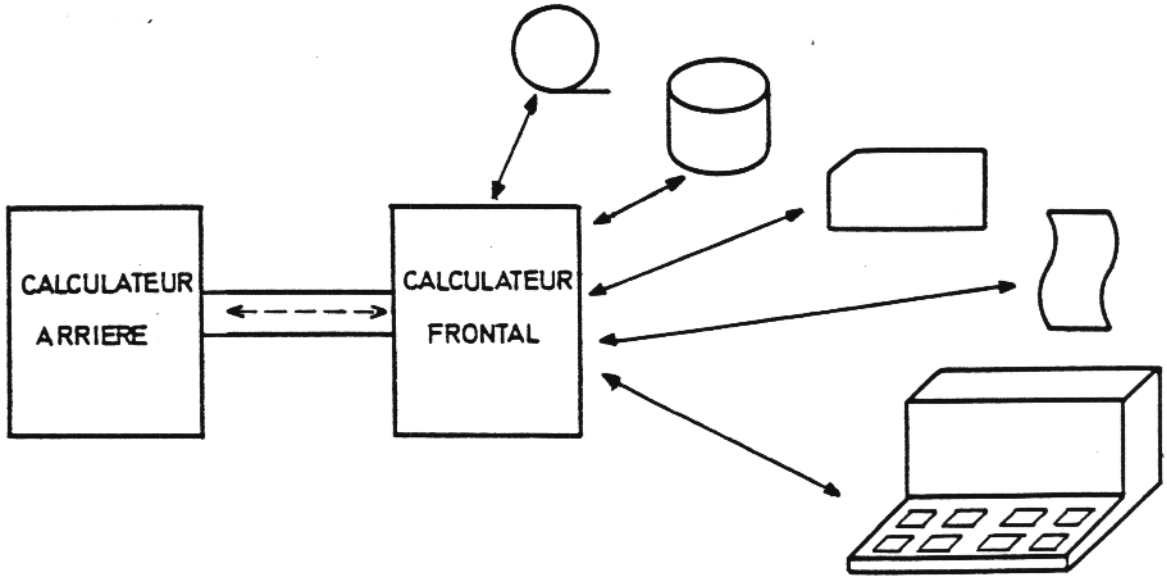


Figure 1
Place d'un super-ordinateur dans un centre de calcul

Le calculateur frontal continuera comme par le passé d'assurer l'interface avec les utilisateurs, l'entrée et la sortie des travaux, les contrôles de routine , la gestion des périphériques, etc. A ces tâches s'ajoutera désormais celle de filtrer les travaux : le calculateur frontal

en gardera une partie par-devers lui, pour les traiter comme par le passé; les autres seront transmis pour exécution au calculateur arrière.

Trois séries de conditions sont nécessaires pour qu'un travail puisse être transmis au calculateur arrière.

1. Le travail doit en comporter la *demande* explicite, grâce à un code compris par le frontal.

2. Les caractéristiques du travail doivent être telles que les *règles d'exploitation* du centre de calcul permettent de l'affecter au calculateur arrière.

3. Les *ressources* logicielles et matérielles demandées pour l'exécution du travail doivent être disponibles sur le calculateur arrière; elles incluent en particulier :

- les périphériques;
- les sous-programmes de bibliothèques;
- les langages de programmation et leurs compilateurs.

Au Centre de Calcul des Etudes et Recherches, le Cray-1 est accessible à partir de deux machines frontales : l'IBM 3081 et le concentrateur CII-HB 66, qui sert d'interface avec le réseau RETINA (cf. [Glaziou 81]).

III LE MATERIEL : CARACTERISTIQUES ORIGINALES DU CRAY-1

III.1. - RIEN NE SERT DE COURIR : IL FAUT PARTIR ENSEMBLE

Pour construire des ordinateurs permettant d'exécuter les programmes de plus en plus vite, deux voies sont possibles :

1. Augmenter la vitesse de base avec laquelle les circuits électroniques exécutent les instructions.

2. Augmenter le degré de parallélisme, c'est-à-dire chercher à exécuter de plus en plus d'opérations simultanément.

La voie 1 a conduit à des résultats spectaculaires (c'est peu dire) dans le passé. Le Cray-1 peut effectuer une série d'opérations arithmétiques en 12,5 nanosecondes chacune (1 ns = 10^{-9} s), c'est-à-dire *deux millions de fois plus vite* que l'ENIAC de 1946 (cf. [Moreau 81]). Les techniques de miniaturisation et d'intégration (LSI, VLSI) permettent d'aller toujours plus loin dans cette voie. Il faut reconnaître cependant que l'on approche aujourd'hui des limites absolues, liées à la vitesse de la lumière; c'est d'ailleurs cette constatation qui a amené les concepteurs du Cray-1 à donner à leur machine des dimensions si étonnamment réduites (6,6 m^2 au sol, pour un poids d'environ 5 tonnes, cf. figure 2), le but poursuivi étant de minimiser la longueur des connexions, donc la vitesse de transmission. La simplicité de la construction (trois types de "puces" électroniques seulement étaient utilisés à l'origine, complétés ultérieurement par deux autres) joue également un rôle important.

Au point où en est aujourd'hui la technique, chaque nanoseconde épargnée par la voie 1 est chèrement gagnée. Il reste donc la voie 2, qu'on peut décrire par le principe suivant :

> PRINCIPE DE PARALLELISME
>
> *Pour faire plus de choses en moins de temps, faisons plusieurs choses en même temps.*

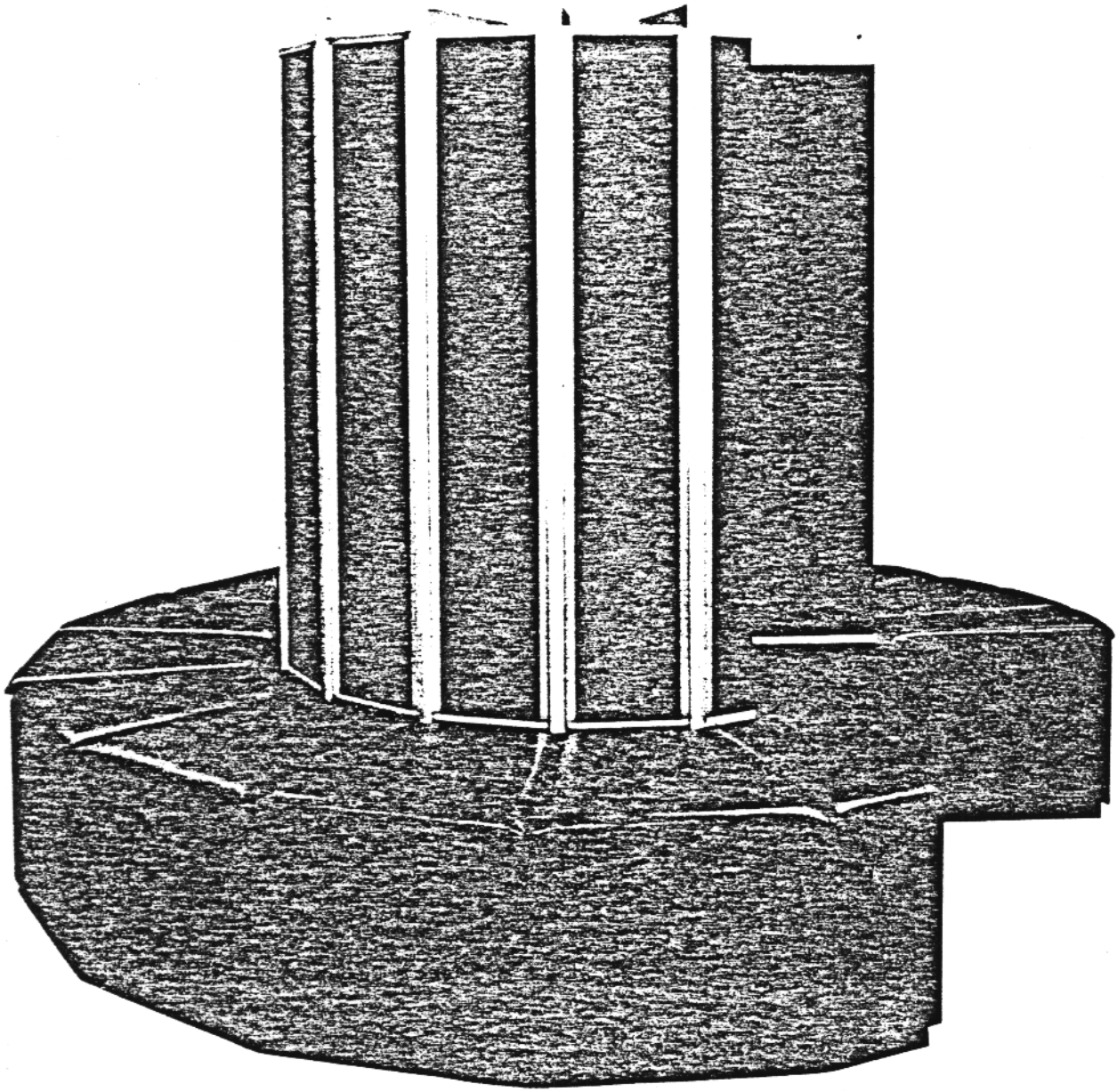


Figure 2 : Un Cray-1

Cette méthode est appliquée depuis longtemps, par exemple, aux systèmes d'exploitation, qui permettent d'affecter l'unité centrale à un programme pendant qu'un autre effectue des entrées ou des sorties, afin de ne pas perdre de temps en attente. Une autre application du principe de parallélisme est constituée par les réseaux et les systèmes distribués, par exemple les réseaux de micro-processeurs, permettant de répartir une tâche entre plusieurs calculateurs. Ce type de structures, où des processus autonomes se synchronisent à des instants initialement inconnus, est étudié, entre autres, dans [Brinch Hansen 73] et [Hoare 78], avec une bonne introduction dans [Brinch Hansen 79].

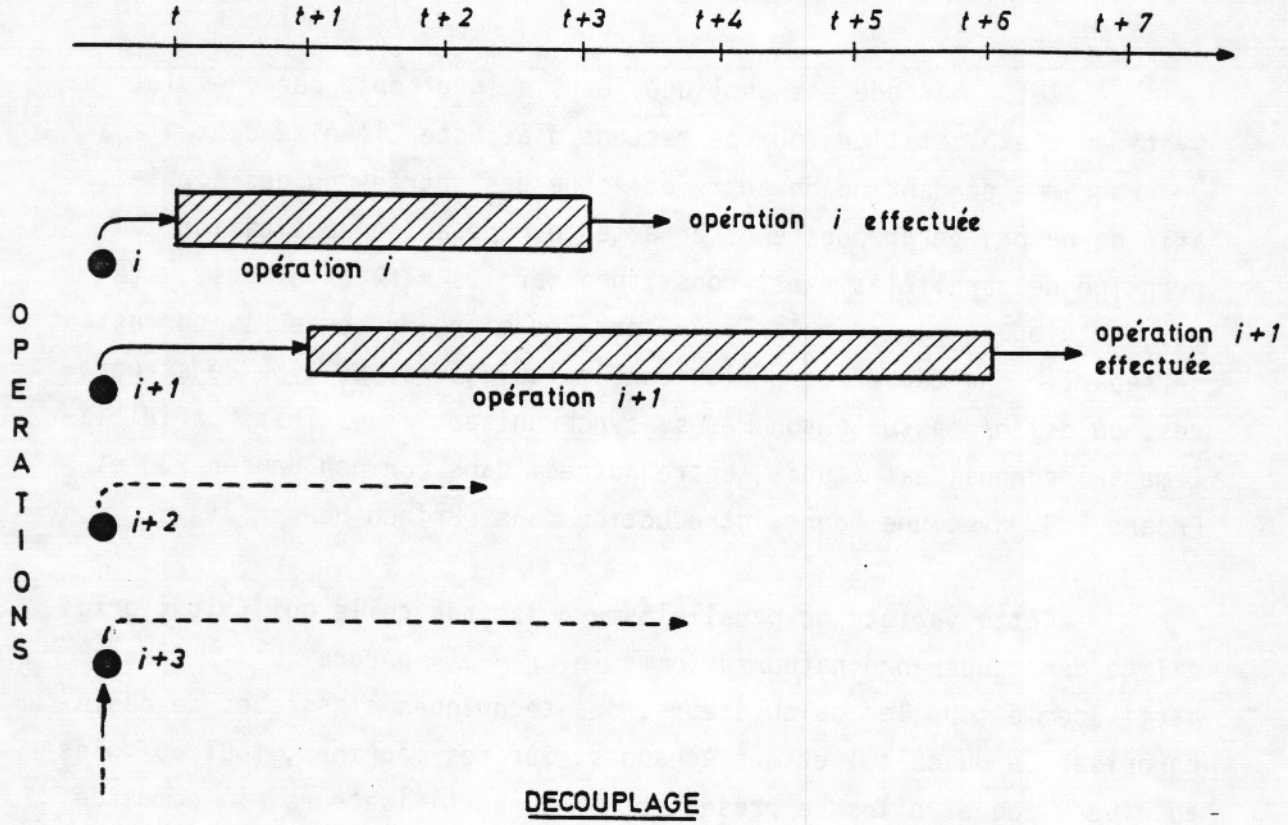
Cette variété de parallélisme n'est pas celle qui fait l'originalité des "super-ordinateurs" comme le Cray-1 - encore qu'ils tirent parti, comme tous les calculateurs, des techniques classiques de désynchronisation du calcul et des échanges. Sur ces machines, tout au moins au niveau auquel elles se présentent à leurs utilisateurs, la commande du processus de calcul reste centralisée, mais le matériel offre la possibilité de *commencer une opération avant que la précédente soit terminée*. Nous constaterons deux variantes de cette technique de "parallélisme contrôlé" (figure 3) :

- le *découplage* : cas où les opérations considérées sont différentes, et peuvent être confiées à des dispositifs disjoints;

- la *segmentation* : cas où les opérations sont les mêmes, et confiées au même dispositif, mais s'effectuent en n étapes indépendantes, le matériel pouvant exécuter simultanément des étapes distinctes d'opérations successives. On obtient alors, à condition que l'alimentation en opérations soit continue, un effet de parallélisme apparent - mais seulement en régime stationnaire, après un *temps d'amorçage* nécessaire à la production du premier résultat (n étapes)[1].

---

(1) Le terme de "segmentation" se trouve dans les publications Cray (à propos de la segmentation des unités fonctionnelles). "Découplage" est introduit par l'auteur.



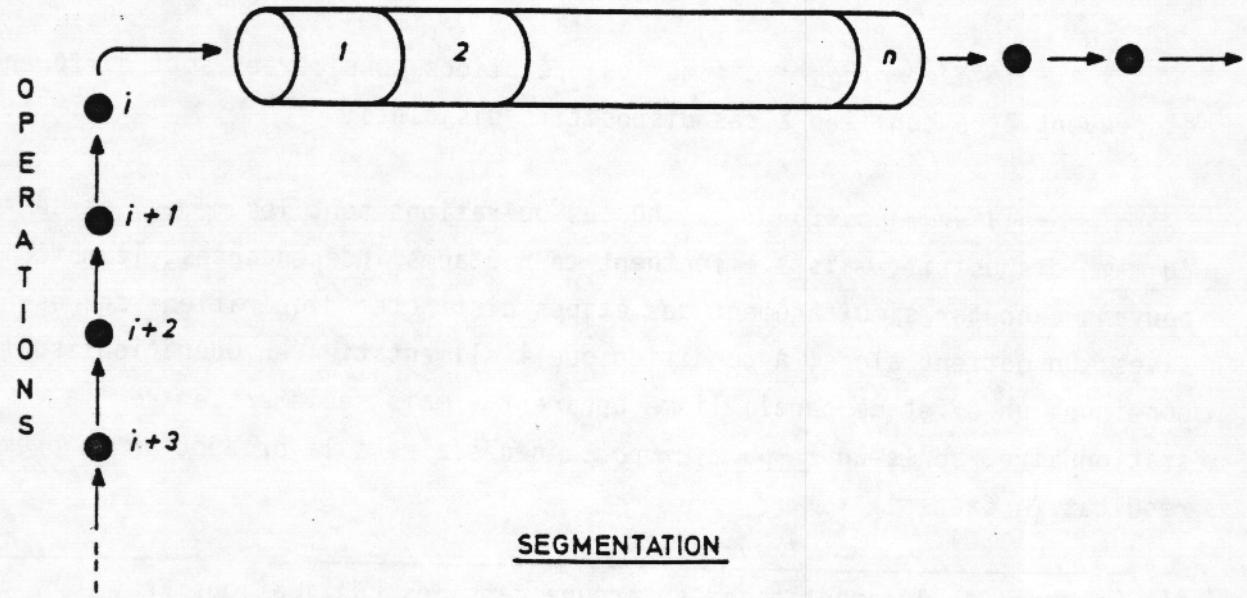


Figure 3

Découplage et segmentation

Un cas particulier de segmentation privilégié par les "super-ordinateurs" est l'application d'une *même instruction* à des données successives différentes. C'est ce qu'on nomme le calcul vectoriel, étudié ci-après en III.7. Cette forme de parallélisme est souvent appelée SIMD (*Single Instruction, Multiple Data* : instruction unique, données multiples), par opposition à des structures plus complexes (MIMD).

Nous étudions dans la suite de cette section comment le principe de parallélisme s'applique aux différents éléments de la conception du Cray-1 : la mémoire (III.2); les unités fonctionnelles, qui effectuent les calculs (III.3); les tampons d'instruction,qui abritent les instructions prêtes à être exécutées (III.4); les registres, qui reçoivent opérandes et résultats (III.5); les entrées et les sorties (III.6); les techniques de calcul vectoriel (III.7). On récapitulera en III.8 la signification du principe de parallélisme pour le programmeur.

PERIODE DE L'HORLOGE

Dans tous les cas, le *temps de référence* auquel le Cray-1 cherche, par application du principe de parallélisme, à ramener des opérations plus longues si le matériel les effectue séquentiellement, est la période de l'horloge, temps de base de l'ordinateur.

Une période d'horloge = 12,5 nanosecondes.

III.2. - LA MEMOIRE

Sur les ordinateurs classiques, le "goulot d'étranglement" qui limite la vitesse des calculs est moins l'exécution des instructions que le temps d'accès à la *mémoire* qui fournit les opérandes et reçoit les résultats. L'organisation de la mémoire d'un Cray-1 cherche à éviter, grâce au principe de parallélisme, un tel blocage par les données.

La mémoire centrale du Cray-1 comprend, en configuration maximale, quatre mégamots (4 194 304 mots de 64 bits)*. Le temps d'accès à un mot est normalement de quatre périodes d'horloge (50 ns); le temps de transfert de mémoire à registre est d'onze périodes (137,5 ns).

L'influence de ce temps de transfert est, en régime stationnaire, tempérée par la segmentation : on peut lancer un accès à la mémoire avant la fin du transfert précédent.

Pour diminuer le temps d'accès observé, on fait naturellement appel au découplage : la mémoire est divisée en seize sections indépendantes appelées *bancs*; les adresses, prises consécutivement, recouvrent cycliquement les seize bancs** (figure 4). Le temps d'accès de 4 périodes s'applique à des accès successifs à un même banc, mais il n'y a pas de contrainte pour des accès à des bancs différents (par exemple pour des adresses consécutives).

En situation optimale, donc, la mémoire pourra transmettre aux registres, ou en recevoir, *une donnée par période d'horloge* en régime stationnaire, après un temps d'amorçage de quinze périodes (4 + 11).

On notera que cette situation optimale ne pourra être obtenue que si les deux conditions suivantes sont réunies :

a) accès à des adresses prévisibles : on devra donc être en mode vectoriel, où les adresses des éléments successifs peuvent être calculées à l'avance (cf. III.7 ci-dessous);

b) pas de conflits d'accès à un même banc : le temps d'accès étant de quatre périodes et les bancs au nombre de seize, ceci exige que la différence entre deux adresses successives ne soit pas un multiple de 16 (accès se faisant alors toutes les quatre périodes)** ni un multiple impair de 8 (un accès toutes les deux périodes).

Ces points devront être pris en compte, dans la mesure du possible, par le programmeur soucieux d'utiliser au mieux les possibilités de calcul vectoriel (section IV).

---

* La mémoire disponible sur la machine de Clamart est actuellement (janvier 82) d'un mégamot (1 048 576 mots).

** La machine de Clamart ne possède actuellement (janvier 82) que huit bancs.

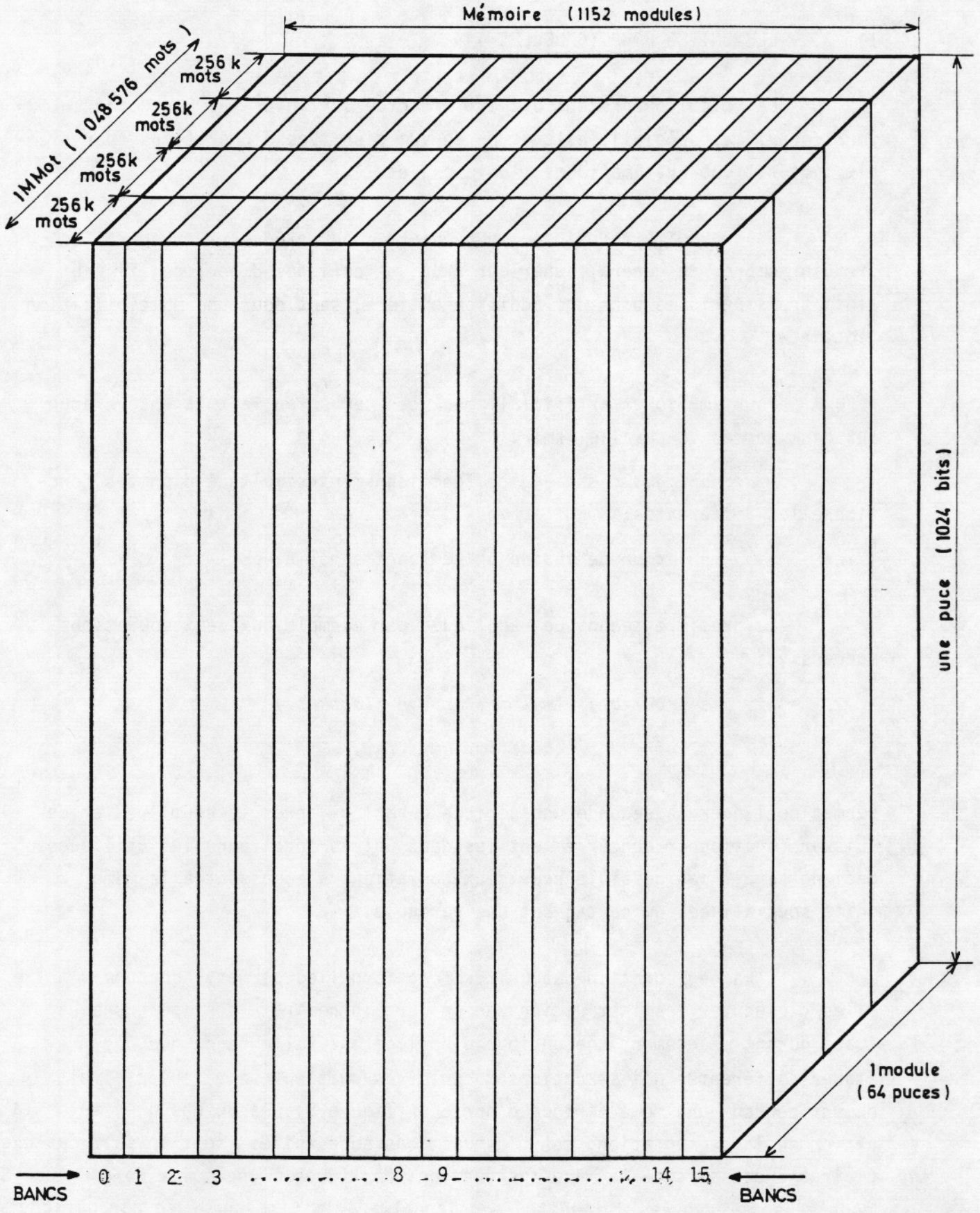


Figure 4

Organisation physique de la mémoire du Cray-1

III.3. - LES UNITES FONCTIONNELLES

Le calcul proprement dit est effectué sur le Cray-1 par douze *unités fonctionnelles*, spécialisées chacune en vue d'un type d'opérations (par exemple : opérations arithmétiques, logiques, etc.).

Chaque unité fonctionnelle requiert, pour exécuter une opération, un certain nombre, en général supérieur à un, de périodes d'horloge; il faut ainsi trois périodes pour une addition entière, sept pour une multiplication flottante.

Pour améliorer artificiellement ces temps, on fait là encore appel aux deux formes de parallélisme :

- *découplage*, c'est-à-dire fonctionnement simultané d'unités fonctionnelles différentes;

- *segmentation* de chaque unité fonctionnelle.

La première technique, appliquée par exemple aux deux opérations successives

$a \leftarrow b * c$; {*multiplication flottante*}

$m \leftarrow i + j$ {*addition entière*},

permet de lancer la seconde une période d'horloge après la première, et de laisser fonctionner concurremment les deux unités fonctionnelles utilisées. Ceci ne serait pas possible pour deux opérations s'adressant à la même unité spécialisée, ou partageant une variable.

La segmentation des unités fonctionnelles permet à chacune d'entre elles d'exécuter les instructions en un certain nombre n d'étapes, chaque étape durant exactement une période d'horloge, de telle façon que des étapes différentes d'instructions successives puissent se dérouler simultanément pendant une même période d'horloge (figure 5). Rappelons que, selon la définition de la segmentation, ces "instructions successives" sont en fait des exemplaires successifs de la même instruction, appliquée à une suite de données. Si ces données sont envoyées à l'unité fonctionnelle à la vitesse d'une par période

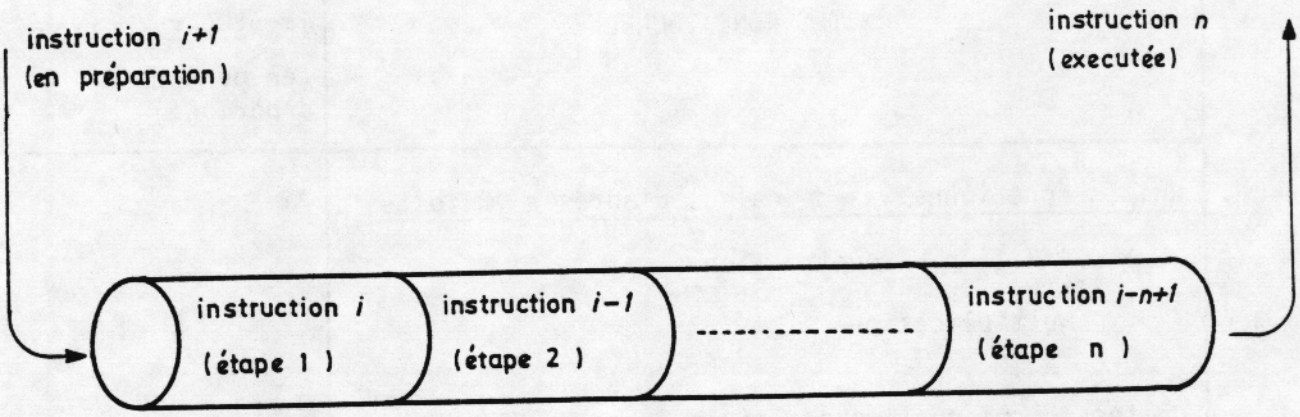


Figure 5

Segmentation d'une unité fonctionnelle

(cf. III.4 ci-dessous), l'unité produira donc également en régime stationnaire un résultat d'instruction par période, comme si son temps d'exécution avait été divisé par n.

Toutes les unités fonctionnelles du Cray-1 sont entièrement segmentées.

On trouvera à la figure 6 la liste des unités fonctionnelles, des types d'instructions qu'elles exécutent, et des temps correspondants. On notera que le Cray-1 ne possède pas d'instruction câblée de division flottante, mais seulement l'inversion approchée. Une division sera calculée par une inversion suivie d'une multiplication.

On notera la présence d'une unité vectorielle booléenne permettant de constituer des masques (vecteurs binaires), et d'extraire grâce à eux les éléments d'un vecteur vérifiant une certaine propriété. Un registre de masque *VM* est présent à cet effet (III.5 ci-après).

III.4. - LES TAMPONS D'INSTRUCTIONS

L'utilisation efficace du découplage et de la segmentation exige que l'arrivée des instructions ne soit retardée en aucune façon.

UNITES FONCTIONNELLES	TEMPS D'EXECUTION (en périodes d'horloge)
Unités fonctionnelles de calcul d'adresse (24 bits)	
Addition - soustraction	2
Multiplication	6
Unités fonctionnelles scalaires (mots de 64 bits)	
Opérations booléennes	1
Décalage	2 ou 3
Addition ou soustraction entière	3
Comptage des bits à 0 (ou 1)	4 ou 3
Unités fonctionnelles flottantes (mots de 64 bits)	
Addition ou soustraction	6
Multiplication	7
Inversion approchée	14
Unités fonctionnelles vectorielles (mots de 64 bits)	
Opérations booléennes (masquage et extraction)	2
Décalage	4
Addition ou soustraction entière	3

Figure 6

Unités fonctionnelles du Cray-1

Le chargement d'une instruction à partir de la mémoire requiert quatorze périodes d'horloge (il est vrai que les instructions placées aux adresses suivantes sont ensuite chargées au rythme de quatre mots par période). Pour éviter ce retard, quatre *tampons d'instruction* se trouvent dans l'unité de calcul; chacun d'eux a une capacité de 64 mots; le format des instructions étant de 16 ou 32 bits, c'est-à-dire un quart ou une moitié de mot, les quatre tampons peuvent donc abriter de 128 à 256 instructions.

Lorsque des instructions se trouvent dans les tampons d'instructions, leur exécution séquentielle est lancée au rythme d'une "parcelle" de 16 bits (instruction ou demi-instruction) par période. Un branchement à une autre instruction présente dans le tampon requiert deux périodes. Une référence à une série d'instructions non présente dans le tampon entraîne un retard initial de quatorze périodes puisqu'elle implique un accès à la mémoire.

Ce mode d'exécution favorise de toute évidence les *boucles courtes*, dont le corps, comportant peu d'instructions, tiendra dans 256 parcelles (les termes "boucle courte" et "boucle longue" font ici référence au nombre d'instructions contenues dans la boucle et non, bien sûr, au nombre d'itérations de la boucle à l'exécution). C'est en vertu de ce principe qu'on est amené à remplacer les boucles longues par plusieurs boucles courtes lorsque c'est possible (voir la *règle d'ouverture des boucles* en III.7 ci-après).

On notera l'analogie avec la notion d'espace de travail dans les systèmes d'exploitation à mémoire virtuelle (hiérarchie de ressources dont les plus rapides sont de capacité limitée).

III.5. - LES REGISTRES

Parmi les opérandes et le résultat d'une instruction, deux éléments au moins (sur trois en général) doivent se trouver dans l'un des *registres* de la machine. Les registres jouent en outre pour les données le même rôle que les tampons pour les instructions : accueillir des éléments à l'avance, de façon à éviter des accès abusifs à la mémoire.

Le Cray possède sept types de registres désignés par les lettres *A, B, S, T, V, VL* et *VM* (figure 7).

Type de registre	Longueur d'un registre de ce type	Nombre de registres de ce type	Rôle
A	24 bits	8	Adresses Index (pour l'adressage), compteurs de boucles, compteurs de décalages, contrôle des canaux (entrées et sorties).
B	24 bits	64	"Réservoir" pour les registres *A* transfert *B* ↔ *A* : une période; transfert *B* ↔ mémoire, par blocs : une période par mot.
S	64 bits	8	Source et destination des opérations scalaires (c'est-à-dire les opérations entières ou flottantes portant sur des mots).
T	64 bits	64	"Réservoir" pour les registres transfert *S* ↔ *T* : une période; transfert *B* ↔ mémoire, par blocs : une période par mot.
V	64 mots de 64 bits	8	Traitement de vecteurs (cf. III.6)
VL	7 bits	1	Longueur de vecteur
VM	64 bits	1	"Masque" binaire pour la sélection d'éléments d'un vecteur.

Figure 7

Registres du Cray-1

III.6. - ENTREES ET SORTIES

Les échanges du Cray sont effectués grâce à 24 calculateurs périphériques ou *canaux*, spécialisés par moitié en entrée ou en sortie, et répartis en quatre "groupes" de six,également spécialisés.

A chaque période d'horloge, on examine l'un des groupes, cycliquement, pour détecter une demande d'accès à la mémoire; s'il s'en trouve, elle n'est servie que quatre périodes plus tard (sauf si elle entre en conflit avec un accès à la mémoire demandé par l'unité de calcul; elle sera alors soumise à nouveau huit périodes plus tard). Les six canaux d'un groupe ont des priorités différentes.

Chaque canal peut donc émettre des demandes d'accès à la fréquence maximale d'une toutes les huit périodes d'horloge. Quatre périodes après une demande non satisfaite, cependant, on peut satisfaire une demande d'un canal moins prioritaire du même groupe, et servir les autres groupes dans les trois périodes suivantes.

On peut donc atteindre, pour les échanges, la vitesse de référence du Cray : un traitement par période d'horloge.

III.7. - LE CALCUL VECTORIEL

L'aptitude du Cray à faire du "calcul vectoriel" résulte pour l'essentiel de la combinaison de trois caractéristiques :

- la présence des huit registres vectoriels *V* (figure 7), de 64 mots chacun, pouvant donc contenir des séries de 64 entiers ou réels;

- l'accès rapide à la mémoire pour des adresses en progression régulière (III.2);

- la segmentation des unités fonctionnelles (III.3).

Le Cray-1 permet, grâce à ces propriétés, d'obtenir en régime stationnaire, dans les cas favorables, un traitement continu de données fournies à la vitesse d'une par période d'horloge. Ces données seront consommées par tranches de 64 éléments au plus.

Le fonctionnement continu permis par ces propriétés (figure 8) est connu sous le nom de mode "pipeline"; on peut proposer *bitoduc* en français [Bossavit 79]. La figure 8 représente le cas où les deux entrées *E1* et *E2* et la sortie *SO* sont vectorielles, c'est-à-dire effectuées sur des registres *V*. *E1* pourrait être aussi un registre scalaire du type *S* (exemple : multiplication d'un vecteur par un scalaire).

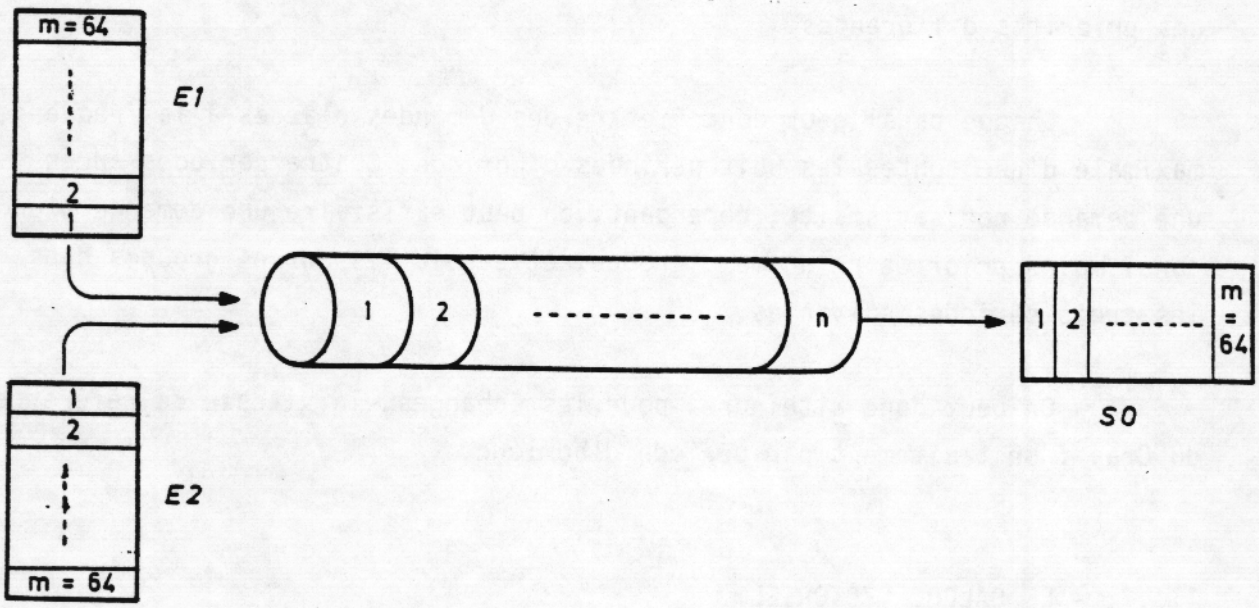


Figure 8

Fonctionnement d'un bitoduc

Le traitement en mode "bitoduc" permet, en régime stationnaire et dans les bons cas, la production d'un résultat par période d'horloge.

Le résultat ainsi obtenu peut à son tour alimenter, comme opérande, une autre unité fonctionnelle; c'est ce qu'on nomme le *chaînage* des opérations vectorielles (figure 9). On peut même créer des boucles en réinjectant la sortie d'un bitoduc à son entrée. Cette technique de chaînage récursif est utilisée pour améliorer l'efficacité de certaines opérations qui ne sont pas vectorisables au sens strict; c'est ce qu'on appelle la *pseudo-vectorisation*, appliquée en particulier par le compilateur Fortran à partir de sa version 1.09 (cf. ci-après V.5.3).

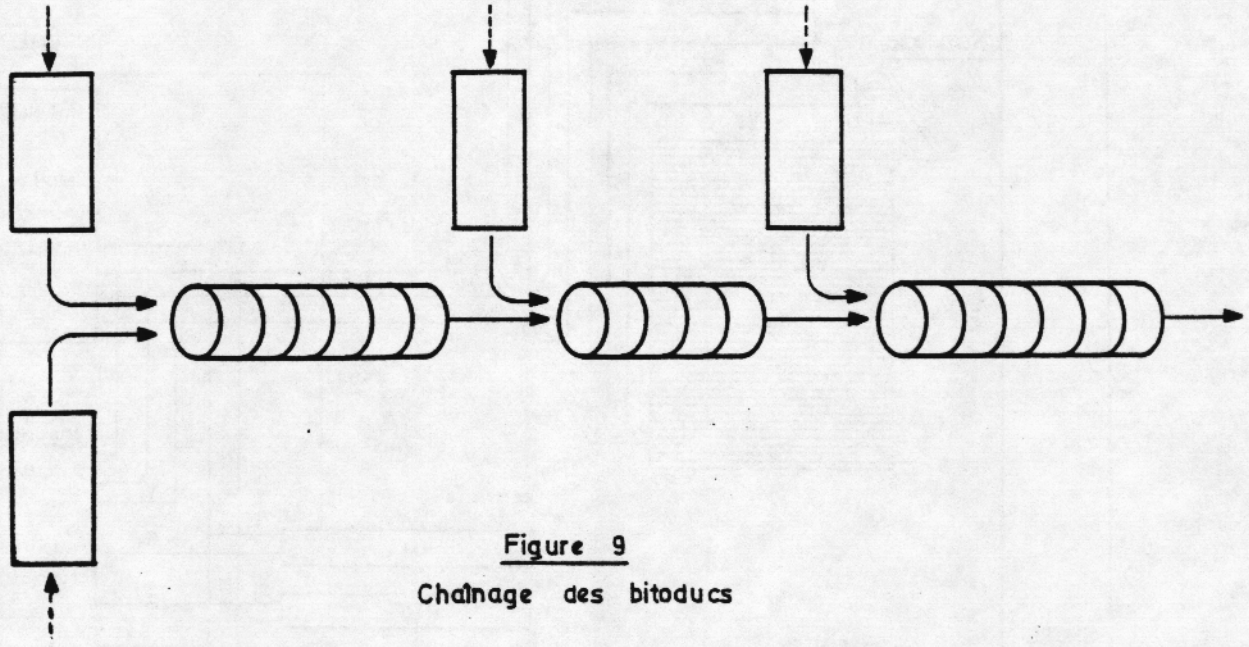


Figure 9
Chaînage des bitoducs

Le temps d'amorçage, plaie de tous les calculateurs vectoriels, donne la limite de taille des vecteurs au-dessous de laquelle le traitement vectoriel n'est pas avantageux. Il est clair que sur le Cray-1 cette limite est directement liée au nombre de périodes requis par chaque unité fonctionnelle (figure 6).

Six unités fonctionnelles sur douze peuvent être utilisées en liaison avec un registre V; il s'agit des six dernières mentionnées sur la figure 6 : les trois unités dites "flottantes" (accessibles également au traitement en mode scalaire), et les trois dites "vectorielles" (réservées au calcul vectoriel).

Lorsqu'une opération vectorielle est lancée sur une unité fonctionnelle, celle-ci est réservée jusqu'à la fin de l'opération, ce qui interdit jusque là toute autre opération sur la même unité.

Cette contrainte, jointe à celle qui exige qu'en mode vectoriel les instructions à exécuter soient en petit nombre, pour pouvoir rester dans les tampons d'instructions (III.4), entraîne le principe suivant :

REGLE D'OUVERTURE DES BOUCLES

Le calcul vectoriel préfère aux boucles longues les suites de boucles courtes.

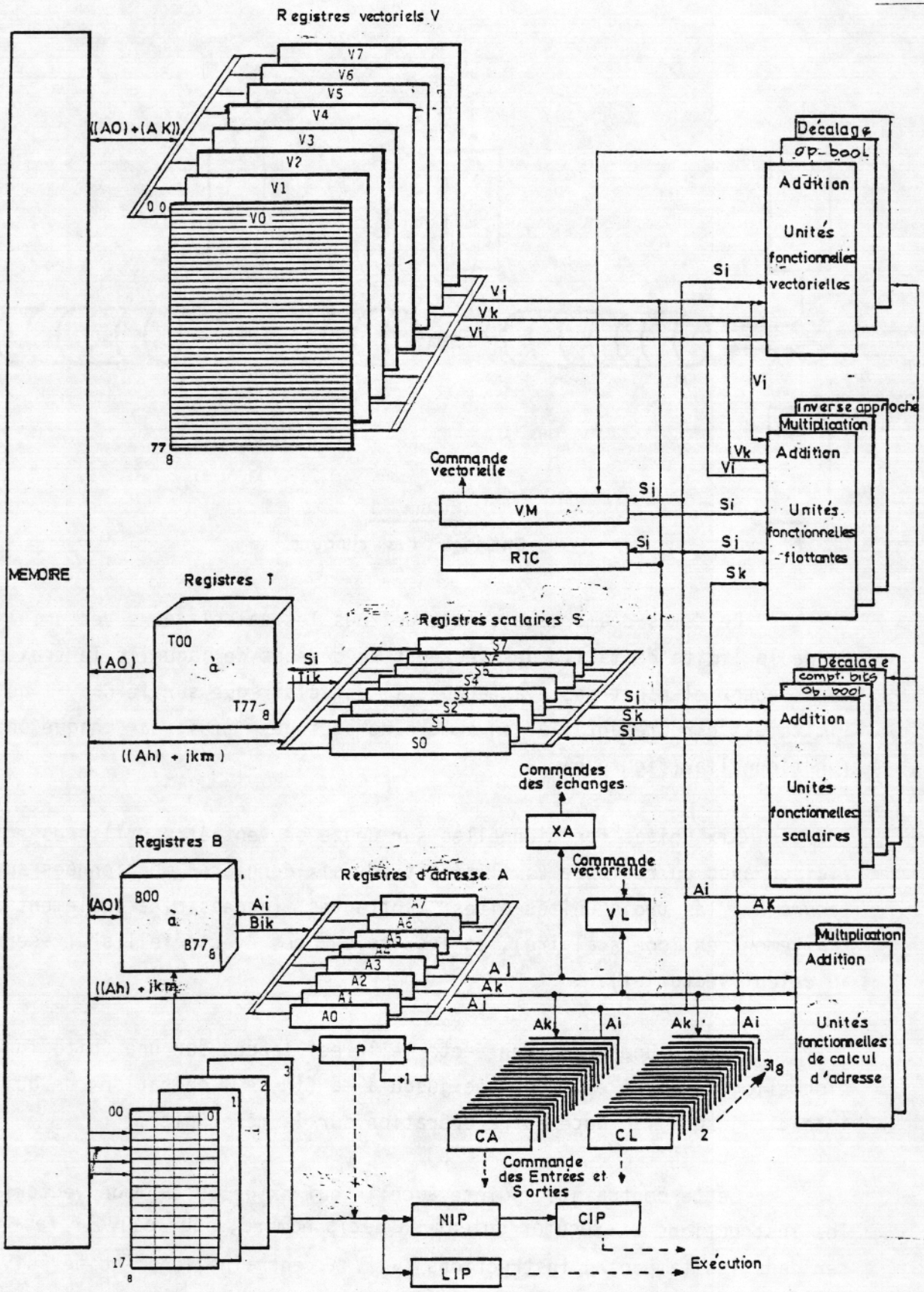


Figure 10

Schéma général de l'unité de calcul d'un Cray 1
(unité de calcul)

Le remplacement pourra le plus souvent être effectué par les compilateurs si l'on programme dans un langage de haut niveau. Sa possibilité entraîne cependant une contrainte assez rigoureuse pour la programmation vectorielle : la règle de non-dépendance croisée (V.6).

III.8. - EN GUISE DE SYNTHESE

Délaissant (enfin) les détails techniques, rassemblés sur le schéma d'organisation générale de l'unité de calcul du Cray-1 (figure 10), nous pouvons essayer de résumer ce que signifient pour le programmeur les structures qui viennent d'être décrites.

L'image mentale qui est apparue peu à peu à travers la description des différents éléments du Cray-1 est celle d'un réseau mythique de tuyauteries complexes, par lesquelles s'écoule un fluide étrange, transportant des instructions, des données et des résultats. L'équilibre idéal, toujours recherché, jamais atteint, est l'état où ce fluide parcourt tous les conduits avec une grâce égale et un débit constant - la période de l'horloge.

Les rouages de ce système parfait répètent à l'unisson, depuis l'aube des temps, la mécanique inhumaine rythmée par la période de l'horloge : avaler une donnée - cracher un résultat. Avaler-cracher. Avaler-cracher. Avaler - ...

La règle de diamant, pour qui veut s'approcher de ce Nirvaña, est de *prévoir* : pour alimenter continûment valves, soupapes et embouchures, il faut savoir à l'avance où chercher les éléments à venir. Ceci se traduit de deux façons :

- Pour les instructions, on cherchera des combinaisons aussi simples que possible : séries contiguës enchaînées, boucles courtes (on préférera deux boucles simples à une boucle complexe); éviter les branchements lointains, les appels de sous-programmes effectués dans le feu de l'action.

- Pour les données, on traitera des séries munies de structures très rigides - essentiellement des suites de données dont les adresses forment des progressions arithmétiques -, pour pouvoir appliquer les techniques de calcul vectoriel.

C'est à ce dernier aspect qu'est consacrée la suite de cette note, après une présentation d'ordre méthodologique.

IV AVANT DE PROGRAMMER VECTORIELLEMENT

IV.1. - QU'EST-CE QU'UN PROGRAMME VECTORIEL ?

Un élément de programme écrit dans un langage de haut niveau sera dit vectorisable si le compilateur peut le traduire sous une forme utilisant les dispositifs de calcul vectoriel offerts par la machine (III.7.). Les éléments non vectorisés seront traités en mode non vectoriel, dit "scalaire".

Nous étudierons à la section suivante (V) les conditions nécessaires et suffisantes pour qu'un élément de programme soit vectorisable. La poursuite de cet objectif peut entraîner, par ordre de difficulté croissante :

- l'adjonction de directives à l'intention du compilateur;

- des modifications locales des programmes;

- des changements d'algorithme.

Avant d'examiner en détail les techniques vectorielles, telles qu'elles découlent de la description précédente du matériel, il est important de bien comprendre la place de la vectorisation par rapport aux autres contraintes de la programmation.

IV.2. - POLITIQUE DE VECTORISATION

La présence d'un calculateur vectoriel ne fait pas disparaître les difficultés classiques de la programmation, et en soulève de nouvelles. A la première catégorie se rattachent la question de la validité des programmes et celle du choix des algorithmes; à la seconde, celle de la portabilité des programmes. Nous nous limiterons à les traiter par deux "règles" (règle de relativité, règle de portabilité). Ceci ne doit pas conduire à sous-estimer leur importance, qui ne peut que croître avec l'arrivée de calculateurs de plus en plus puissants.

REGLE DE RELATIVITE

a) L'aptitude au traitement vectoriel n'est qu'*un des éléments de l'efficacité* d'un programme. En particulier :

   i) de nombreux programmes sont limités non par le calcul, mais par les échanges;

   ii) la vectorisation n'a de sens que si l'algorithme utilisé est bon. Toute vectorisation produit par exemple un effet négligeable par rapport au remplacement d'un algorithme de tri en temps n^2 par un algorithme en temps $n \log n$.

b) L'amélioration de l'efficacité n'est intéressante que *relativement aux boucles les plus internes* des programmes, représentant souvent une faible proportion du code. On considère couramment que les programmes de type scientifique pur passent de 80 à 90% de leur temps dans quelques pourcent de leur texte.

c) L'efficacité n'est qu'*un des éléments de la qualité* d'un programme. La qualité la plus importante est la validité.

Voir aussi le chapitre VIII de [Meyer 78]. La compréhension de la règle de relativité devrait permettre d'éviter toute frénésie de vectorisation.

Lorsqu'un centre de calcul reçoit un calculateur vectoriel, qui coexistera avec un "frontal" (section II), il cesse d'être totalement homogène. Les programmeurs ne peuvent donc plus se permettre d'ignorer les problèmes de la portabilité. Aussi proposons-nous la règle suivante, au demeurant modérée.

REGLE DE PORTABILITE

Tous les programmes écrits en vue d'être vectorisés doivent pouvoir s'exécuter sur le calculteur frontal (non vectoriel), éventuellement au prix de modifications minimes.

Sur la programmation portable en Fortran, on consultera [APCOL 82].

Nous verrons que le Cray-1 permet de respecter cette règle au prix d'une certaine discipline.

Les deux règles méthodologiques qui précèdent resteront à l'arrière-plan de toute la discussion sur la vectorisation.

IV.3. - PROBLEMES DE LANGAGE

Les langages actuellement diffusés sur Cray-1 de façon officielle sont un langage d'assemblage (CAL) et Fortran.

Un compilateur Pascal existe, développé au laboratoire atomique de Los Alamos mais non soutenu par Cray. Un compilateur Pascal "officiel", développé pour Cray par l'Université de Manchester (G.B.) est annoncé pour la fin de 1982.

Les règles de vectorisation qui vont suivre, et les documents existants, sont relatifs à Fortran. On notera que Fortran est probablement le langage le plus mal adapté au type de manipulation entraîné par la recherche de vectorisation (comme à bien d'autres critères), du fait de la pauvreté de ses structures de contrôle et de données. On peut en fait caractériser une bonne partie des techniques appliquées par le compilateur comme consistant à reconstituer à grand-peine une structure globale des manipulations de données détruite par l'emploi de Fortran, et qui se serait exprimée tout naturellement à travers APL, Algol 68 ou PL/I.

Le langage Fortran disponible sur le Cray-1 (cf. Annexe) est un dialecte original, plus étoffé que celui de la norme 1966 (Fortran IV), et comprenant un certain nombre d'éléments de Fortran 77 [Meyer 79], mais encore en deçà de cette nouvelle norme, en particulier à l'article des manipulations de caractères. Il s'en rapproche peu à peu, plus lentement hélas que nous ne l'espérions dans la première édition de cette note; le point le plus gênant est la manipulation de caractères à la mode Fortran 77 qui, disponible aujourd'hui (janvier 82) sur IBM (compilateur Fortran VS), n'est toujours pas prête sur Cray.

Dans la situation actuelle, dont on souhaite qu'elle soit transitoire, les différents cas sont les suivants :

- un programme écrit en Fortran IV (Fortran H Extended sur IBM) sera assez facilement adapté au Cray-1 (voir cependant à l'annexe B quelques sources d'ennuis possibles);

- un programme écrit en Fortran 77 (Fortran VS sur IBM) soulèvera des difficultés tant que le Fortran Cray n'inclura pas l'ensemble de la norme Fortran 77;

- un programme écrit en Fortran Cray sera facile à adapter à Fortran VS ou à une autre version de langage conforme à Fortran 77 (en revanche, le retour à Fortran IV n'est pas un exercice recommandé).

IV.4. - LE COMPILATEUR FORTRAN ET LA VECTORISATION

Le compilateur Fortran du Cray-1, appelé CFT (Cray Fortran Translator) dans la documentation technique, produit du code vectorisé lorsqu'il reconnaît des structures de programme se prêtant à cette vectorisation. Les programmeurs écrivent donc dans un Fortran conforme aux règles normales du langage, non sans adapter toutefois leur style de programmation afin de faciliter cette reconnaissance. C'est l'objet des "techniques vectorielles" vues ci-après.

La politique de vectorisation adoptée par le Cray est, de ce point de vue, plus favorable à la portabilité des programmes que les solutions consistant à concevoir des langages spécialisés pour le traitement vectoriel [Perrot 79a, 79b, 79c], à proposer des extensions à un langage [Flanders 79], ou à vectoriser seulement les appels à des sous-programmes spécialisés.

Cet avantage, considérable au vu de notre "règle de portabilité", doit cependant être tempéré par deux remarques :

a) La vectorisation extrême peut exiger une réécriture considérable du code, rendant inefficace l'exécution sur une machine non vectorielle, et peu compréhensible le programme résultant (voir en V.5 le programme calculant la somme des éléments d'un vecteur).

b) Pour des traitements vectoriels d'emploi fréquent, il est tentant de faire appel aux sous-programmes très efficaces de la bibliothèque du super-ordinateur.

L'inconvénient *a* doit être limité à de petites portions de programme (cf. ci-après, règle des boucles internes). Pour éviter des surprises désagréables avec *b*, nous préconiserons la règle d'exploitation suivante, qui découle tout naturellement de la règle de portabilité :

> REGLE DE COMPATIBILITE
>
> Ne diffuser un programme de la bibliothèque du super-ordinateur qu'après avoir inclus dans la bibliothèque du calculateur frontal un sous-programme de même nom et de même spécification externe, écrit de préférence dans un langage normalisé.

La règle de relativité mettait l'accent sur la petitesse des portions de programmes accélérables de façon significative (b), et sur la validité des programmes (c). Les deux règles qui suivent témoignent de l'attitude conservatrice du compilateur, bien conforme à ces principes.

> REGLE DES BOUCLES INTERNES
>
> Seules les boucles les plus internes sont susceptibles d'être vectorisées par le compilateur.

Il est donc inutile de changer quoi que ce soit aux autres. Le volume de code à réécrire restera ainsi proportionnellement faible dans les cas normaux.

> REGLE DE SECURITE
>
> Le compilateur ne vectorise de lui-même que les boucles répondant de façon démontrée aux conditions de vectorisation.

La nécessité de cette règle vient de ce que les résultats d'un même programme peuvent être différents selon qu'il est exécuté en mode scalaire ou vectoriel. L'interprétation "par défaut" définie par les normes des langages étant celle des ordinateurs classiques, c'est-à-dire l'interprétation scalaire, le compilateur CFT ne vectorisera jamais sans être sûr de pouvoir le faire, de peur de produire des résultats erronés. Les programmeurs disposent donc d'une bonne sécurité à cet égard.

On notera que des cas peuvent se produire où un élément de programme est vectorisable, mais le compilateur ne dispose pas des éléments pour le démontrer; ceci se produit en particulier en liaison avec le problème de la dépendance arrière (V.5) et celui de la dépendance croisée (V.6). Les programmeurs peuvent le signaler au compilateur grâce à des directives de vectorisation qui vont lever ses hésitations (s'il n'y a pas d'impossibilité détectée par ailleurs). Ceci, bien sûr, est alors aux risques et périls du programmeur.

Les directives de vectorisation adressées au compilateur Fortran du Cray-1 ne soulèvent pas de difficultés de portabilité car elles revêtent la forme de commentaires (on parlerait de "pragmas" en Algol 68 ou en Ada).

V LES GRANDES TECHNIQUES VECTORIELLES

V.1. - LES CONDITIONS DE LA VECTORISATION

Les contraintes que doivent respecter les programmes pour que le compilateur puisse les vectoriser peuvent se ramener aux cinq conditions nécessaires et suffisantes de la règle ci-dessous(1).

REGLE DE VECTORISATION

Un calcul est vectorisable si et seulement s'il respecte les cinq conditions suivantes :

[C] . c'est une *série continue* d'opérations (V.2);

[P] . chacune de ces opérations est *primitive* (V.3);

[R] . les données traitées sont *régulières* (V.4);

[DC] . elles ne présentent pas de *dépendance croisée* (V.5);

[DA] . elles ne présentent pas de *dépendance arrière* (V.6).

Ces différentes conditions sont détaillées dans les cinq paragraphes qui suivent. Pour donner une vue d'ensemble, on peut les expliquer grossièrement ainsi :

- une "série continue" d'opérations est, en pratique, une boucle;

- une opération "primitive" est une opération n'entraînant aucune rupture de contrôle (en FORTRAN, ceci ne laisse guère apparemment que

---

(1) Cette règle est une interprétation, par l'auteur, des nombreux exemples épars dans les publications Cray. La terminologie de cette section est celle de l'auteur, non celle des documents Cray.

l'affectation est le *CONTINUE*; nous verrons qu'on peut élargir un peu cette classe);

- un ensemble "régulier" de données est, grosso modo, une constante, un entier variant en progression arithmétique ou une sous-suite d'un tableau dont l'adresse varie en progression arithmétique (exemple : la sous-suite *A(M)*, *A(M+L)*, *A(M+2*L)*, *A(M+3*L)* ... du tableau *A*);

- une relation de "dépendance"(1) lie un élément apparaissant à gauche d'une affectation (ou la suite à laquelle il appartient) aux éléments de la partie droite (exemple : dans *A(I) = C*B(I+1)* *A* dépend de *C* et de *B*);

- une dépendance est "croisée" si elle lie des éléments de deux suites différentes modifiées l'une et l'autre dans les opérations d'une série (exemple : si une boucle d'indice *I* contient les affectations *A(I) = ...* et *B(I) = A(I+K)*, alors *B* dépend de *A* qui est modifié).

- une dépendance est une dépendance "arrière" lorsqu'elle fait dépendre un élément d'une suite d'un élément antérieur de la même suite (exemple : *A(I+7) = C*A(I-1)*).

Examinons maintenant ces cinq conditions.

V.2. - SERIES CONTINUES.

Le fonctionnement continu en mode vectoriel exige que les instructions nécessaires restent en permanence disponibles dans les tampons d'instructions (III.4). Ces tampons ne peuvent abriter que 128 à 256 instructions, qui doivent donc contenir une répétition pour que le calcul vectoriel présente un intérêt.

Le compilateur vectorisera donc des boucles. La règle appliquée par le compilateur Cray (CFT) est plus précise encore :

---

(1) Le terme de "récursion" est employé dans les publications Cray.

REGLE DU *DO*

Seules les boucles *DO* sont vectorisables.

Combinée à la "règle des boucles internes" (IV.4), cette règle indique donc que les seuls éléments susceptibles d'être vectorisés sont les boucles *DO* au niveau le plus interne.

Les boucles par *IF* et *GOTO* ne le sont donc pas. Ceci serait particulièrement gênant en FORTRAN IBM (où les boucles *DO* sont de type répéter ... jusqu'à et non tant que); l'instruction *DO* du Cray-1, conforme à Fortran 77, équivaut à une instruction vide si *(borne_sup - borne_inf)*pas < 0*.

Il vaut mieux donc sortir d'une boucle *DO* que d'hésiter à y rentrer. Méthodologiquement, ceci n'est pas très sain.

On peut espérer que les versions ultérieures du compilateur lèveront cette restriction.

V.3. - OPERATIONS PRIMITIVES

Le principe d'alimentation continue exige, nous l'avons dit, que les séries continues d'instructions qui forment les boucles vectorielles soient prévisibles, c'est-à-dire n'entraînent aucune rupture de séquence. Elles doivent donc ne contenir aucune opération non "primitive" au sens de la définition suivante.

REGLE DES OPERATIONS PRIMITIVES

Pour qu'une boucle soit vectorisable, il faut qu'elle ne contienne aucune des opérations non primitives suivantes :

a) les entrées et les sorties;

b) les tests et les branchements;

c) les appels de sous-programmes.

Le cas a) semble irrémédiable. b) et c) justifient une discussion plus fine.

Tests dans les boucles

Le cas *b* est ennuyeux dans des exemples même très simples, comme celui de l'affectation conditionnelle; ainsi la boucle suivante, non vectorisable, qui affecte une valeur ou une autre aux éléments de A selon le signe de leur valeur initiale :

```
      DO  100  I = 1, N
            IF (A(I).GT.0) A(I) = COS (A(I))
            IF (A(I).LE.0) A(I) = SIN (A(I))
100         CONTINUE
```

(N.B. On peut préférer un style de programmation utilisant des branchements plutôt qu'un test redondant).

La structure matérielle du Cray-1 permet ici en fait de vectoriser une boucle équivalente. Le principe est de calculer toutes les valeurs éventuellement requises, et d'extraire ensuite celles qui sont véritablement nécessaires grâce aux instructions de masquage (cf. III.7). Cette opération équivaut en termes de Fortran à remplacer l'extrait ci-dessus par :

```
      REAL A(N), P(N), Q(N)
      LOGICAL MASQUE(N)
      ----------------
      DO  101  I = 1,N
            P(I) = COS (A(I))
            Q(I) = SIN (A(I))
            MASQUE(I) = A(I).GT.0
101         CONTINUE
C     — EXTRACTION —
      DO  102  I = 1,N
            IF (MASQUE(I))     B(I) = P(I)
            IF (NOT.MASQUE(I)) B(I) = Q(I)
102         CONTINUE
```

La première boucle correspond au (double) calcul de toutes les valeurs possibles, la seconde à une extraction, opérée par des instructions vectorielles de masquage. Bien entendu, ce texte Fortran représente symboliquement le programme équivalent produit par le compilateur, et ne correspond à rien qui existe concrètement sous la forme donnée.

Pour atteindre l'effet de ces boucles vectorielles, la structure logicielle proposée en Fortran est un jeu de deux sous-programmes spéciaux *CVMGT* (comparer pour supérieur strictement) et *CVMGZ* (comparer à zéro). On peut ici remplacer la boucle par :

```
      DO  103  I = 1,N
            B(I) = CVMGT (A(I).GT.0, COS (A(I)), SIN (A(I)))
103         CONTINUE
```

On aura noté que cette méthode va tout à fait à l'encontre du critère de portabilité énoncé en IV.2., et constitue une exception à l'emploi exclusif par le Cray-1 de constructions Fortran normales.

La reconnaissance par le compilateur des constructions de boucle telles que *100* (ou *102*), sans exiger l'emploi de fonctions spécialisées, fait partie des extensions annoncées par Cray. On peut donc espérer que la difficulté liée à la portabilité sera bientôt résolue.

Il reste cependant, fonctions spécialisées ou non, deux points délicats. Le premier est celui des branches d'alternative exigeant plus d'une instruc-

tion Fortran. L'autre est celui des quantités non définies. L'appel à *CVMGT* ou *CVMGZ*, produit par un programmeur ou dans l'avenir par le compilateur, évalue des vecteurs complets pour n'en garder que certains éléments. Les autres ont pu être écartés volontairement, comme dans l'exemple ci-dessous :

```
      DO  150  I = 1, N
            IF (A(I).GE.0) B(I) = SQRT (A(I))
            IF (A(I).LT.0) B(I) = SQRT (-2*A(I))
150   CONTINUE
```

Il est clair qu'on ne veut pas ici évaluer complètement les deux vecteurs $\sqrt{A}$ et $\sqrt{-2\ A}$. Cette boucle ne peut donc pas, en tout état de cause, être vectorisée.

Appels de sous-programmes

Pour résoudre le cas c) (appels dans une boucle), on peut appliquer la règle suivante :

> REGLE DES APPELS
>
> Mettre le sous-programme dans la boucle, ou la boucle dans le sous-programme.

La première solution consiste à remplacer l'appel

```
CALL SP( .... )
```

dans le corps d'une boucle, par le texte même du corps du sous-programme après remplacement des arguments.

La seconde consiste à remplacer toute entière une boucle de la forme :

```
      DO  200  I = 1, N
          ..........
          CALL SP (A(I), B(I), ...)
          ..........
200       CONTINUE
```

par un seul appel de sous-programme

CALL SPVEC (A, B, ...)

où *SPVEC* est un sous-programme "vectoriel", opérant répétitivement, par une boucle, sur les éléments de *A* et *B*.

Lorsqu'elle est applicable, c'est cette seconde solution (mettre la boucle dans le sous-programme) qui doit être recommandée. La première est en effet contraire à tout principe de modularité et de structuration des programmes (voir la règle de validité); la seconde permet au contraire de conserver, voire d'améliorer la structure de contrôle, en l'associant à celle des données : on aboutit à une architecture dans laquelle les sous-programmes manipulent des tableaux entiers; en d'autres termes, on écrit des algorithmes véritablement vectoriels, manipulant des objets de type "tableau".

On notera que cette seconde solution entre dans le cadre des techniques visant à ramener un programme Fortran à son équivalent dans un langage permettant la manipulation de tableaux, comme APL, PL/I ou Algol 68 (cf. IV.3).

V.4. - ENSEMBLES REGULIERS DE DONNEES

Pour qu'une boucle soit vectorisable, toutes les références qu'elle effectue à des données doivent être prévisibles. En pratique, ceci signifie que la variation des valeurs de ces données (pour des entiers) ou, plus généralement, de leurs adresses, doit s'effectuer en progression arithmétique (ce qui comprend en particulier les constantes). De tels éléments seront dits "réguliers".

REGLE DE REGULARITE

Un élément *E*(constante, variable, élément de tableau, expression) intervenant dans une boucle est dit régulier si et seulement s'il vérifie l'une des conditions suivantes :

a) *E* n'est pas modifié dans le corps de la boucle; nous dirons alors qu'*E* est constant relativement à la boucle;

b) *E* est l'indice de boucle;

c) E est une expression de la forme $\pm C*E'$, où C, E, E' sont entiers, C est constant relativement à la boucle et E' régulier;

d) E est une expression de la forme $E'\pm C$, où C, E, E' sont entiers, C est constant relativement à la boucle et E' régulier;

e) E est un élément de tableau dont tous les indices sont constants relativement à la boucle, sauf éventuellement un qui est alors une variable régulière.

Pour qu'une boucle soit vectorisable, il faut que tous les objets qui y apparaissent (variables, éléments de tableaux) soient réguliers.

La règle de régularité est récursive, ce qui ne devrait pas poser de problème particulier de compréhension.

Exemple

Soit une boucle d'indice I. Soient $A(20)$, $B(10,50)$, $C(20,30,10)$ des tableaux réels, M et N des entiers n'apparaissant pas dans la boucle à gauche d'un signe =.

Les éléments suivants sont réguliers :

I

M

N

$I + N$

$I - N$

$M * I$

$M * I - 3 * M * N$

$A(I)$

$B(I, M)$

$C(2*N - M, M * I - 3*M*N, 72 + M)$

Les éléments suivants ne sont pas réguliers :

B(I, I + 1)	(deux indices non constants relativement à la boucle)
*M/(N*I)*	(division)
*N*C(I)*	(*C(I)* n'est pas entier)

Nota : Le compilateur CFT impose des restrictions supplémentaires (relatives par exemple à l'imbrication des parenthèses), qui semblent évoluer rapidement. En l'absence d'une documentation précise, et en particulier d'une grammaire en BNF, il nous a paru préférable de ne pas les détailler. On trouvera des indications plus concrètes dans [de Drouas 81].

V.5. - LA DEPENDANCE ARRIERE

V.5.1. - Définition

Le fonctionnement en mode "bitoduc" entraîne qu'à tout instant de l'exécution d'une boucle *DO* une itération de rang i peut être lancée alors qu'une itération précédente, de rang $j \leq i$, n'est pas encore terminée. Plus précisément, ceci ne peut se produire que si :

$$i - j < cb$$

où cb est la capacité du bitoduc (64 pour le Cray-1). Les calculs seront donc erronés si la seconde utilise, dans le programme, des résultats produits par la première. C'est un cas de dépendance arrière.

Exemple :

La boucle

```
      DO  300  I = 2, N
          .............
          A(I) = 3*A(I-1)
300       CONTINUE
```

n'est pas vectorisable du fait de la dépendance arrière directe. Il existe aussi des dépendances arrière indirectes :

```
      DO  350  I = 1, N
          B(I) = 3/I
          A(I) = I**B(I-8)
350       CONTINUE
```

A utilise ici un élément de *B* calculé 8 itérations plus tôt. Cette dépendance arrière est en fait ici un cas particulier de la "dépendance croisée" vue plus loin.

On peut résumer cette situation par la règle suivante.

> REGLE DE NON-DEPENDANCE ARRIERE
>
> Pour qu'une boucle soit vectorisable, il faut qu'aucune des affectations qu'elle contient ne fasse dépendre un élément d'un vecteur, à une itération i quelconque, d'un élément du même vecteur qui a pu être modifié à une itération j, avec $i - cb < j < i$ ($cb = 64$).

On trouvera à l'annexe D une définition formelle de la dépendance arrière[1].

V.5.2. - Exemples de dépendances arrière; solutions

Un cas trivial de dépendance arrière ne faisant pas nécessairement intervenir d'indice est celui d'une affectation de la forme $X = f(X)$ (où f n'est pas l'identité). Il y a dépendance arrière entre la variable X (considérée comme vecteur à un élément) et elle-même. Deux cas importants en pratique sont la somme d'un vecteur

```
      S = 0
      DO  400 I = 1, N
          S = S + A(I)
400       CONTINUE
```

(1) Dans son état actuel, le compilateur CFT refuse toujours la vectorisation en cas de dépendance arrière, même s'il peut vérifier que $j \leq i - cb$. Il faut donc, si l'on sait que $j \leq i - cb$, utiliser une directive *IVDEP* (cf. V.7). Cette situation est regrettable.

où l'on remarque d'ailleurs que S n'est pas régulière, et le produit matriciel :

```
      DO  600 I = 1, M
          DO  550 J = 1, N
              DO  500 K = 1, P
                  C (I,K) = C(I,K) + A(I,K) * B(K,J)
500               CONTINUE
550           CONTINUE
600       CONTINUE
```

On a supposé ici qu'une première boucle (vectorisable), non écrite, avait initialisé C à zéro.

La dépendance arrière n'est gênante que si elle ne dépasse pas la capacité cb du bitoduc (cb= 64).

La dépendance arrière est de toute évidence une propriété de l'algorithme utilisé; on ne peut guère s'attendre à la voir supprimée par une modification superficielle du style de programmation. Il serait intéressant d'examiner les grands algorithmes numériques à la lumière de ce critère; il est clair par exemple que des algorithmes où l'approximation se déplace parallèlement sur un "front" (Jacobi) sont préférables de ce point de vue à ceux (Gauss-Seidel) pour lesquels l'approximation de la solution au [*point p, instant t*] dépend d'un certain nombre de valeurs approchées [p', t] pour d'autres points p', et non pas seulement de valeurs [p', $t - 1$]. Gauss-Seidel possède cependant une version vectorielle ; sur la vectorisation de ces deux algorithmes, voir [Bossavit 81a]

Dans certains cas, une modification relativement simple d'un algorithme à dépendance arrière peut le rendre vectorisable. C'est le cas du calcul matriciel : si au lieu de penser "élément"

$$c_i^j = \Sigma\ a_i^k\ b_k^j$$

on pense "vecteur (ligne)"

$$c_i = \Sigma\ a_i^k\ b_k$$

on obtient la version vectorisable ci-après :

```
      DO  800 I = 1, M
C         -- LIGNE I --
          DO  750  K = 1, P
              DO  700  J = 1, N
                  C(I,J) = C(I,J) + A(I,K) * B(K,J)
700               CONTINUE
750           CONTINUE
800       CONTINUE
```

(Note importante : Il devient ici fondamental d'avoir effectué l'initialisation à zéro de la matrice *C* dans une première boucle disjointe).

La version vectorisable du calcul de la somme d'un vecteur est beaucoup moins convaincante. Nous la donnons ici à titre d'illustration; avant de s'inquiéter, on lira le paragraphe V.5.3 ci-après. Elle utilise un sous-tableau auxiliaire *B* permettant de se ramener à 64 sommes de sous-vecteurs :

```
C     CALCUL DE LA SOMME DE N ELEMENTS DU VECTEUR A
C     -- BOUCLE VECTORISABLE --
        DO  900  I = 1, N
            B(I) = A(I)
900         CONTINUE
C     -- BOUCLE VECTORISABLE (EVENTUELLEMENT NULLE) --
        DO  930  I = 65, N
            B(I) = B(I) + B(I - 64)
930         CONTINUE
C     -- BOUCLE NON VECTORISABLE (64 ITERATIONS AU PLUS) --
        S = 0.
        M = MAXO (N - 63, 1)
        DO  960  I = M, N
            S = S + B(I)
960         CONTINUE
```

L'utilisation d'une constante "magique" telle que 64 est évidemment exécrable d'un point de vue méthodologique. C'est ce genre de pratique qui donne, après quelques années, des programmes mutilés, portant les stigmates des systèmes successifs. On préfèrera utiliser la constante symbolique *CB*, après avoir déclaré (cf. ci-après V.7) :

```
C       -- TAILLE DU BITODUC DU CRAY-1
        PARAMETER (CB = 64)
        INTEGER CB
```

V.5.3. - Les opérations de réduction et la pseudo-vectorisation

A partir de la version 1.09 du compilateur (janvier 1981), une amélioration importante est venue remédier aux difficultés soulevées par les problèmes tels que le calcul d'un produit scalaire ou celui de la somme des éléments d'un vecteur. C'est la pseudo-vectorisation des opérations de réduction. On appelle opération de réduction sur des vecteurs A, B ... le calcul d'une valeur x par une boucle de la forme

```
        x = valeur initiale
        DO 970  I = 1, N
970         x = x ⊕ g (A(I), B(I), ...)
```

où $\oplus$ et g sont des opérations arithmétiques. Le produit scalaire et la somme d'un vecteur sont typiquement des opérations de réduction. D'après ce qui précède, une réduction n'est pas vectorisable du fait de la dépendance arrière entre x et lui-même. Mais elle est pseudo-vectorisable; la technique de pseudo-vectorisation consiste à utiliser un bitoduc par étapes successives, en réinjectant la sortie dans l'entrée (voir [de Drouas 81] pour les détails de cette technique). Le résultat sera un temps de calcul plus grand que celui d'une opération vectorielle au sens propre, mais bien inférieur à celui d'une opération non vectorielle.

V.6. - LA DEPENDANCE CROISEE

La règle de non-dépendance arrière s'applique à des éléments d'une même suite. Pour des suites différentes, la règle est plus stricte :

> REGLE DE NON-DEPENDANCE CROISEE
>
> Pour qu'une boucle soit vectorisable, il faut qu'aucune des affectations qu'elle contient ne fasse dépendre un élément d'un vecteur, à une itération i quelconque, d'un élément de vecteur qui peut être modifié par une autre affectation à une itération j, avec $|j-i| < cb$ $(cb = 64)$.

On trouvera une définition formelle de la dépendance croisée à l'annexe D[1]. Ainsi, la boucle :

```
      DO  1000  I = 1, N
          A(I) = 2*A (I+2)
          B(I) = 3*A (I+1)
1000  CONTINUE
```

n'est pas vectorisable du fait de la dépendance croisée entre *B* et *A* (et non pas de la dépendance directe entre *A* et lui-même, qui est une dépendance avant).

A quoi est due cette règle, draconienne puisqu'elle peut faire référence à des éléments postérieurs des suites considérées (comparer $|i-j| < cb$ dans cette règle à $i - cb < j < i$ dans la précédente) ? Elle découle en fait de la règle d'ouverture des boucles (III.7). Nous avons vu que, pour vectoriser une boucle "longue", on peut être obligé de la scinder en plusieurs boucles. Ici le résultat d'une telle ouverture de boucle serait donc le même que si le programmeur, au lieu de la boucle *1000*, avait écrit :

```
      DO  1010  I = 1, N
1010      A(I) = A (I+1) + 3
      DO  1011  I = 1, N
1020      B(I) = A (I+1) - 1
```

Le résultat est ici différent : en supposant le vecteur *A* initialisé à zéro, la boucle *1000* donne un vecteur *B* tout à zéro (puisque la seconde affectation de la boucle utilise les anciennes valeurs de *A (I+1)*, alors que les boucles *1010-1020* affectent la valeur 1 à tous les éléments de *B* (dans la boucle *1020*, on utilise les nouvelles valeurs de *A*).

---

(1) Dans son état actuel, le compilateur CFT refuse toujours la vectorisation en cas de dépendance croisée, même s'il peut vérifier que $|j-i| \geq cb$. Il faut donc, si l'on sait que $|j-i| \geq cb$, utiliser une directive *IVDEP* (cf. V.7). Cette situation est regrettable.

Bien entendu, l'interprétation classique de Fortran est l'interprétation séquentielle (*1000*). De peur d'induire une erreur par l'interprétation vectorielle en cas d'ouverture (*1010-1020*), le compilateur ne vectorisera donc pas une boucle présentant une telle dépendance croisée.

Aucune règle simple ne permet d'éviter la dépendance croisée, qui est en général, comme la dépendance arrière, une propriété de l'algorithme.

Ce type de manipulation de boucles rejoint les problèmes classiques de la théorie des transformations de programme. On trouvera une étude en ce sens dans [Bossavit 82].

V.7. - DIRECTIVES AU COMPILATEUR - PRINCIPE DE PARALLELISME

Dans certains cas, une connaissance approfondie du programme permet d'affirmer qu'il n'y a pas de dépendance (arrière ou croisée), alors que la lettre du texte ne le garantit pas.

Dans

```
      DO  1200  I = 1, N
            A(I) = A(I+P)
            B(I) = A(I+Q)
1200        CONTINUE
```

si l'on sait que $(P \geq 0 \text{ ou } P \leq -64) \geq (|Q| \geq 64)$, alors la boucle est vectorisable. On peut alors inciter le compilateur à larguer ses scrupules (cf. IV.4, règle de sécurité) et à la vectoriser, en la faisant précéder de la directive spéciale suivante, qui apparaît comme un commentaire :

```
CDIR$  IVDEP
```

Il est clair qu'une telle opération est aux risques et périls du programmeur.

Les documents Cray indiquent que, même avec une telle directive, la vectorisation ne sera pas effectuée si le compilateur la détecte comme impossible. En d'autres termes la clause *IVDEP* sert uniquement à lever les soupçons de dépendance (arrière ou croisée) que le compilateur nourrit à l'égard des variables qui sont constantes relativement à la boucle (P et Q ci-dessus).

On peut regretter que le mode d'expression offert au programmeur ne soit pas plus précis : ALGOL W ou Ada auraient proposé des assertions permettant de dire exactement ce que l'on sait, ou croit savoir, en écrivant quelque chose comme :

```
CDIRØ ASSERT (P.GE.0.OR.P.LE. - CB) .AND. (ABS(Q) . GE. CB)
```

[ATTENTION, CECI EST UNE SUGGESTION DE L'AUTEUR ET NON UNE DIRECTIVE LEGALE DU FORTRAN CRAY !]

Une situation assez fréquente est celle où les éléments comme *P*, *Q* ne sont pas en fait des variables, mais des paramètres du problème, qui restent constants pendant une exécution du programme; on les désigne par leurs noms symboliques plutôt que par leurs valeurs numériques en prévision d'un changement possible. Si leur valeur est fournie par un ordre *DATA*

```
DATA P/7/,Q/2250/
```

le compilateur n'en tire aucune assurance particulière, car des variables initialisées ainsi peuvent être modifiées. Par contre, l'ordre *PARAMETER* présent dans le Fortran du Cray-1 et en Fortran 77 permet de définir des constantes symboliques, non susceptibles d'affectation :

```
PARAMETER (P = 7, Q = 2250)
```

Si les valeurs des éléments soupçonnés d'interdire la vectorisation pour fait de dépendance sont indiquées de cette façon, le compilateur CFT vectorisera les boucles du type *1200* sans qu'il soit besoin d'inclure une directive *IVDEP*.

On notera enfin, pour terminer sur les problèmes de dépendance, qu'un cas particulièrement agréable est celui où il n'y a aucune dépendance entre les itérations d'une boucle, c'est-à-dire où elles pourraient toutes s'exécuter en même temps. Il s'agit d'une condition suffisante, mais non nécessaire (sinon nous n'aurions pas autant développé la dépendance arrière et la dépendance croisée). Un exemple est la version vectorielle du produit matriciel (V.5, boucle *700*).

V.8. - L'ADRESSAGE INDIRECT

Introduisons pour conclure une technique utile de vectorisation. Une boucle de la forme :

```
      DO  1300 I = 1, N
            J = f(I)
            A(I) = g(B(J))
1300  CONTINUE
```

où f et g sont des expressions régulières (cf. V.4; par exemple, f pourrait être un nom de tableau, et g une expression plus compliquée) n'est pas vectorisable car J n'est pas une variable régulière.

Le dédoublement de cette boucle, avec utilisation d'un vecteur auxiliaire *IND*, permet de rendre "à-demi" vectorisable :

```
C     -- BOUCLE NON VECTORISABLE --
         DO  1330  I = 1, N
               J = f(I)
               IND (I) = A(J)
1330           CONTINUE
C     -- BOUCLE VECTORISABLE --
         DO  1360  I = 1, N
               A(I) = g(IND(I))
1360           CONTINUE
```

Le gain pourra être appréciable si g est une expression compliquée.

ANNEXE A

PRINCIPALES CARACTERISTIQUES DU CRAY-1

UNITE DE CALCUL

Structure

modes de traitement : scalaire, vectoriel
instructions : 128 codes différents
arithmétique : entière et flottante (N.B. pas de division flottante câblée)
unités fonctionnelles : 12, segmentées
interruptions : système à priorités

Constantes de base

période de l'horloge : 12,5 nanosecondes (1 ns = 10^{-9} s)
taille des mots : 64 bits
adresses : sur 24 bits

Registres

4 tampons d'instructions (64 éléments de 16 bits chacun)
8 registres *A* (adresses : 24 bits)
64 registres *B* (adresses intermédiaires : 24 bits)
8 registres *S* (scalaires : 64 bits)
64 registres *T* (scalaires intermédiaires : 64 bits)
8 registres *V* (vecteurs : 64 éléments de 64 bits chacun)
1 registre *VL* de longueur de vecteur
1 registre *VM* de masque de vecteur
1 registre d'horloge temps réel (64 bits)

ANNEXE B

PRINCIPALES CARACTERISTIQUES DU LANGAGE FORTRAN DU CRAY-1

Nous nous contenterons de quelques points particuliers susceptibles de soulever des difficultés pour le passage d'IBM à Cray. Une note plus complète sera publiée séparément; par ailleurs, la version actuelle du langage [Cray Fortran] évolue progressivement pour être mise en concordance avec Fortran 77.

Parmi les points délicats, signalons :

a) Le passage des arguments

Sur IBM, les arguments simples (non tableaux) à des sous-programmes sont passés par "valeur-résultat", c'est-à-dire recopiés localement. Sur Cray, c'est l'adresse qui est transmise. Ceci ne peut en principe entraîner d'incohérences que dans des programmes un peu bizarres (données accessibles de deux façons : comme argument et comme membre d'un commun). Voir le chapitre IV de [Meyer 78] pour plus de détails.

b) Les boucles *DO*

Une boucle *DO* est nulle si $(borne\_sup - borne\_inf) * pas < 0$.
En Fortran IV IBM, le corps de boucle est toujours exécuté au moins une fois. Là encore, seuls des programmes vraiment particuliers peuvent se trouver en défaut. La convention du FORTRAN du Cray-1 est celle de la nouvelle norme Fortran 77 et du Fortran VS d'IBM. Une option du compilateur permet cependant d'exécuter les boucles selon la convention du Fortran IV IBM (*ON = J* ; cf. table 9 à l'annexe C).

c) La taille des mots

Les mots du Cray-1 sont de 64 bits, taille double de celle sur IBM. La précision des calculs numériques devrait donc être meilleure, sauf bien sûr pour les programmes utilisant des critères d'arrêt non paramétrés.

On notera les correspondances suivantes (qui ne sont pas des identités, les systèmes de numérotation interne étant différents) :

IBM	CRAY
*INTEGER*2 (16 bits)*	
INTEGER (32 bits)	*INTEGER*2 (64 bits)* ou *INTEGER*
REAL ou *REAL*4 (32 bits)* *REAL*8* ou *DOUBLE PRECISION (64 bits)*	*REAL* ou *REAL*4* ou *REAL*8 (64 bits)*
*REAL*16 (128 bits)*	*DOUBLE PRECISION (128 bits)*

d) Caractères

Le code interne du Cray-1 est le code ASCII (IBM : EBCDIC). Un mot, de 64 bits, peut contenir 8 caractères (IBM : 4). Les programmes utilisant les propriétés du codes EBCDIC et/ou la propriété selon laquelle un mot contient exactement quatre caractères devront donc être modifiés.

L'ordre alphabétique des lettres,et l'ordre relatif des chiffres, sont préservés dans les deux cas.

Le Fortran du Cray-1 contient de nombreuses autres possibilités absentes du Fortran IV IBM; beaucoup d'entre elles, mais non toutes, existent en Fortran 77 et donc en particulier dans le Fortran VS d'IBM; les plus intéressantes actuellement sont les entrées et sorties sur des fichiers à accès direct, et les conversions internes (remplaçant *ENCODE-DECODE*). Par contre, le type *CHARACTER* (chaîne de caractères) n'est pas encore disponible.

Les problèmes de taille de mots sont particulièrement délicats : le passage à une machine à mots plus longs est numériquement sûr, mais non l'inverse. Sur tous ces problèmes de portabilité, on consultera le *Guide méthodologique de programmation Fortran* publié par la Division APCOL [APCOL 82].

ANNEXE C

INSTRUCTIONS COMMANDE POUR L'EXECUTION D'UN TRAVAIL SUR CRAY-1

Un travail soumis au Cray-1 est matérialisé par trois fichiers :

- le fichier de commande
- le fichier source
- le fichier de données si nécessaire.

Les paragraphes qui suivent décrivent la constitution du fichier de commande lorsque le fichier source contient un programme Fortran. Pour tous les détails, on se reportera à [Magnier 82].

1. - LA CARTE DE TRAVAIL

La carte de travail figure en tête du paquet; elle indique au système d'exploitation le nom du travail et ses caractéristiques.

2. - LES ALLOCATIONS DE FICHIERS

Fortran ne permettant pas d'adresser un fichier autrement que par un numéro d'unité logique, la correspondance entre ce numéro et un fichier physique doit être indiquée au système. On utilise pour cela l'ordre *ASSIGN* .

Les cartes comportant des ordres *ASSIGN* sont placées dans le paquet après la carte de travail et avant l'ordre de chargement du module exécutable.

Les ordres d'allocation de fichier se présentent sous une syntaxe différente selon que le fichier est préexistant ou non.

2.1. - Allocation avec création de fichier

La syntaxe dans ce cas est la suivante :

ASSIGN(DN = FTn)

Cet ordre a pour effet de créer un fichier nommé *FTn* dans lequel le programme Fortran pourra lire ou écrire en adressant l'unité logique *n* .

Exemple

```
ASSIGN(DN = FT10)
```

demande la création du fichier *FT10*.

Les instructions Fortran

```
      WRITE(10,1) X
1     FORMAT(F7.3)
```

provoquent l'écriture de la valeur de *X* dans *FT10* selon le format 1.

2.2. - Allocation d'un fichier préexistant

La correspondance entre le fichier *nomdefichier* et l'unité logique *n* est obtenue par l'ordre

```
ASSIGN(DN = nomdefichier, A = FTn)
```

à la condition que *nomdefichier* ait été créé auparavant et soit connu du système.

2.3. - Valeurs du numéro d'unité logique *n*

Ainsi que le prévoit la norme Fortran, *n* peut prendre toutes les valeurs de 0 à 99 inclus. Toutefois le système d'exploitation du Cray-1 comprend les valeurs suivantes :

n = 100 pour désigner le fichier prédéfini en lecture *$IN* (également adressable avec *n* = 5).

n = 101 pour désigner le fichier prédéfini en écriture *$OUT* (également adressable avec *n* = 6).

n = 102 pour désigner le fichier prédéfini *$PUNCH* qui n'est autre que le perforateur de cartes.

3. - COMPILATION

L'ordre *CFT* permet l'appel du compilateur Fortran. La syntaxe est la suivante :

CFT I = idn, L = ldn, B = bdn, C = cdn, E = n, ON = listeon, OFF = listeoff, TRUNC = nn.

Les paramètres sont optionnels. Si l'on veut n'en mentionner aucun, il suffit de faire suivre l'appel de *CFT* par un point : *CFT.*

Signification des paramètres :

I = idn : nom du fichier source à compiler.

L = ldn : nom du fichier qui recevra la liste de compilation. Le nom par défaut est *$OUT*. *L = 0* inhibe l'édition de la liste si la compilation se déroule correctement.

B = bdn : désigne le fichier binaire dans lequel est stocké le module objet généré. Le nom par défaut est *$BLD*.

C = cdn : nom du fichier contenant la traduction en langage d'assemblage du programme source générée par le compilateur. La traduction n'a pas lieu si l'option *C* n'est pas spécifiée.

E = n : supprime les messages du compilateur de niveau inférieur à *n*. La valeur par défaut est 3. On distingue 5 niveaux décrits dans le tableau suivant :

Niveau	Type	Description
1	*COMMENT*	Commentaires sur les parties inefficaces du programme.
2	*NOTE*	Signale les instructions non standard.
3	*CAUTION*	Erreur possible (exemple : cas d'un programme ne passant jamais par une instruction).
4	*WARNING*	Erreur probable (exemple : indice manquant dans la référence à un tableau).
5	*ERROR*	Erreur rédhibitoire entraînant l'arrêt du travail.

ON = listeon : Liste des options de compilation activées; ces options sont précisées sur la figure 9.

OFF = listeoff : Liste des options de compilation désactivées.

TRUNC = nn : Indique le nombre de bits tronqués lors de l'évaluation des résultats en virgule flottante. *nn* est compris entre 0 et 47 inclus. La valeur par défaut est 0.

4. - *SWITCH*

L'instruction *SWITCH* permet de positionner six indicateurs (numérotés de 1 à 6) à la valeur *ON* ou *OFF*.

Le programme Fortran peut consulter la valeur de l'indicateur à l'aide de la fonction prédéfinie *SSWITCH*.

Syntaxe :

(SWITCH, n = OFF) ou *(SWITCH, n = ON)*

pour *n* entier prenant ses valeurs de 1 à 6 inclus.

5. - RELIURE ET CHARGEMENT

Le relieur et le chargeur sont lancés par une seule instruction :

LDR.

Sans autres précisions, l'éditeur-chargeur suppose que le module objet ou chargeable se trouve dans le fichier *SBLD.*

6. - EXEMPLE

Voici un exemple de fichiers de commande particulièrement simple :

```
JOB, JN = THEODORE
CFT.
LDR.
/EOF
```

Notons cependant que la carte de fin de fichier */EOF* est destinée à l'ordinateur frontal; sa syntaxe n'est donc pas définitive.

OPTIONS DE COMPILATION

Option	Description	Valeur par défaut
A	Abandon du travail après compilation si une unité de programme contient une erreur rédhibitoire.	*OFF*
B	Liste le début du code généré de chaque bloc.	*OFF*
C	Fournit la liste des noms de communs et des longueurs de chaque unité de programme.	*ON*
D	Dresse la table des boucles *DO*.	*OFF*
E	Permet la prise en compte des directives adressées au compilateur.	*ON*
F	Permet la trace de l'exécution (directives *FLOW, NOFLOW)*.	*OFF*
G	Liste le code généré.	*OFF*
H	Seule la première instruction de chaque unité de programme est listée. Cette option l'emporte sur les autres.	*OFF*
I	Joint les étiquettes générées par le compilateur à la table des symboles.	*OFF*
J	Entraîne l'exécution de toutes les boucles *DO* au moins une fois (compatibilité avec le Fortran IV d'IBM)	*OFF*
L	Permet la prise en compte des directives de mise en page	*ON*
M	Met en service le dernier passage de l'optimiseur	*ON*
N	Insère dans la table des symboles les objets définis mais non utilisés	*OFF*

O	Entraîne l'impression d'un message en cas de débordement dans les tableaux à l'exécution.	*OFF*
P	Si *OFF* remplace tous les objets en double précision par des objets réels simples.	*ON*
Q	Arrête la compilation après 100 erreurs.	*ON*
R	Arrondit le résultat des multiplications.	*ON*
S	Imprime le programme Fortran source.	*ON*
T	Fournit la table des symboles après chaque unité de programme.	*ON*
V	Vectorise les boucles *DO* internes.	*ON*
W	Compile les opérations en virgule flottante comme des appels de sous-programmes écrits par l'utilisateur.	*OFF*
X	Dresse la table des symboles avec références croisées après chaque unité de programme (*X* l'emporte sur *T*).	*OFF*
Y	Pour la mise au point du compilateur	*OFF*
Z	Ecrit la table de symboles en mode *DEBUG* sur *ØBLD*	*OFF*

ANNEXE D

DEFINITION FORMELLE DE LA DEPENDANCE

Considérons une boucle exécutée ni fois. Elle calcule ni éléments d'un certain nombre de suites. Soit ns le nombre de ces suites. Chaque exécution de la boucle opère ns affectations à des variables ou éléments de tableaux ; nous ne considérons que des éléments de tableaux en traitant les variables simples comme des tableaux à un element.

Pour i compris entre 1 et ni, l'exécution de la i-ieme itération de la boucle modifie la valeur de nt éléments de tableaux qui sont donnés par

$$v_t\,(g_t(i)) \qquad 1 <= t <= nt$$

g pour "partie gauche" (d'une affectation)

La valeur affectée à chacun de ces éléments peut être notée :

$$f_t\,(v_{h_{t,1}}(d_{t,1}(i)),\ v_{h_{t,2}}(d_{t,2}(i)),\ldots\ldots,v_{h_{t,p(t)}}(d_{t,p(t)}(i)))$$

(d pour "partie droite" ; p(t) est une borne dépendant de t)

Exprimons l'existence de ces différentes fonctions par une "classe" en langage Z, qui nous permettra d'énoncer les conditions de non-dépendance :

```
non_dépendance ==
     class with
          cb : NAT
          ni, ns : NAT ;
          INDICE, TABLEAU : subset (NAT) ;
          g : TABLEAU -> (INDICE -> INDICE) ;
          p : TABLEAU -> NAT ;
          d : TABLEAU * NAT -/> (INDICE -> INDICE) ;
          h : TABLEAU * NAT -/> TABLEAU ;
     def
          cb == 64 ; INDICE == 1..ni ; TABLEAU == 1..ns
     where
          dom (d) = set t, n where n (- 1..p(t)) end ;
          dom (h) = dom (d) ;
          forall i, t1, t2 , k1 for
               i : 1..ni ; t1, t2 : 1..ns ; k1 : 1..p(t1)
          then
               arrière :
                    h(t1, k1) = t1 ==>
                         d (t1, k1) (i) (/- g (t1)(i-cb+1..i) ;
               croisée :
                    h(t1, k1) = t2 ==>
                         d (t1, k1)  (i) (/- g (t2) (i-cb+1..i+cb-1)
          end
     end
```

ANNEXE E

RECAPITULATION DES REGLES PROPOSEES

PRINCIPE DE PARALLELISME

Pour faire plus de choses en moins de temps, faisons plusieurs choses en même temps.

PERIODE DE L'HORLOGE

Dans tous les cas, le *temps de référence* auquel le Cray-1 cherche, par application du principe de parallélisme, à ramener des opérations plus longues si le matériel les effectue séquentiellement, est la période de l'horloge, temps de base de l'ordinateur :

une période d'horloge = 12,5 nanosecondes.

REGLE D'OUVERTURE DES BOUCLES

Le calcul vectoriel préfère aux boucles longues les suites de boucles courtes.

REGLE DE RELATIVITE

a) L'aptitude au traitement vectoriel n'est qu'*un des éléments de l'efficacité* d'un programme. En particulier :

i) de nombreux programmes sont limités non par le calcul, mais par les échanges;

ii) la vectorisation n'a de sens que si l'algorithme utilisé est bon. Toute vectorisation produit par exemple un effet négligeable par rapport au remplacement d'un algorithme de tri en temps n^2 par un algorithme en temps $n \log n$.

b) L'amélioration de l'efficacité n'est intéressante que *relativement aux boucles les plus internes* des programmes, représentant souvent une faible proportion du code. On considère couramment que les programmes de type scientifique pur passent de 80 à 90% de leur temps dans quelques pour-cent de leur texte.

c) L'efficacité n'est qu'*un des éléments de la qualité* d'un programme. La qualité la plus importante est la validité.

REGLE DE PORTABILITE

Tous les programmes écrits en vue d'être vectorisés doivent pouvoir s'exécuter sur le calculateur frontal (non vectoriel) éventuellement au prix de modifications minimes.

REGLE DE COMPATIBILITE

Ne diffuser un programme de la bibliothèque du super-ordinateur qu'après avoir inclus dans la bibliothèque du calculateur frontal un sous-programme de même nom et de même spécification externe, écrit de préférence dans un langage normalisé.

REGLE DES BOUCLES INTERNES

Seules les boucles les plus internes sont susceptibles d'être vectorisées par le compilateur.

REGLE DE SECURITE

Le compilateur ne vectorise de lui-même que les boucles répondant de façon démontrée aux conditions de vectorisation.

REGLE DE VECTORISATION

Un calcul est vectorisable si et seulement s'il respecte les cinq conditions suivantes :

[C] . c'est une *série continue* d'opérations (V.2);

[P] . chacune de ces opérations est *primitive* (V.3);

[R] . les données traitées sont *régulières* (V.4);

[DC] . elles ne présentent pas de *dépendance croisée* (V.5);

[DA] . elles ne présentent pas de *dépendance arrière* (V.6).

REGLE DU *DO*

Seules les boucles *DO* sont vectorisables

REGLE DES OPERATIONS PRIMITIVES

Pour qu'une boucle soit vectorisable, il faut qu'elle ne contienne aucune des opérations non primitives suivantes :

a) les entrées et les sorties;

b) les tests et les branchements;

c) les appels de sous-programmes.

REGLE DES APPELS

Mettre le sous-programme dans la boucle, ou la boucle dans le sous-programme.

REGLE DE REGULARITE

Un élément E(constante, variable, élément de tableau, expression) intervenant dans une boucle est dit régulier si et seulement s'il vérifie l'une des conditions suivantes :

a) E n'est pas modifié dans le corps de la boucle; nous dirons alors qu'E est constant relativement à la boucle;

b) E est l'indice de boucle;

c) E est une expression de la forme $\pm C*E'$, où C,E,E' sont entiers, C est constant relativement à la boucle, et E' régulier;

d) E est une expression de la forme $E'\pm C$, où C,E,E' sont entiers, C est constant relativement à la boucle, et E' régulier;

e) E est un élément de tableau dont tous les indices sont constants relativement à la boucle, sauf éventuellement un qui est alors une variable régulière.

Pour qu'une boucle soit vectorisable, il faut que tous les objets qui y apparaissent (variables, éléments de tableaux) soient réguliers.

REGLE DE NON-DEPENDANCE ARRIERE

Pour qu'une boucle soit vectorisable, il faut qu'aucune des affectations qu'elle contient ne fasse dépendre un élément d'un vecteur, à une itération i quelconque, d'un élément du même vecteur qui a pu être modifié à une itération j, avec $i - cb < j < i$ ($cb = 64$).

REGLE DE NON-DEPENDANCE CROISEE

Pour qu'une boucle soit vectorisable, il faut qu'aucune des affectations qu'elle contient ne fasse dépendre un élément d'un vecteur, à une itération i quelconque, d'un élément de vecteur qui peut être modifié par une autre affectation à une itération j, avec $|j-i| < cb$ ($cb = 64$).

BIBLIOGRAPHIE

Pour les documents Cray, la date et le numero donnes sont ceux de la version disponible au moment de la publication de cette note. Ces renseignements sont evidemment susceptibles de modification.

[APCOL 82] Division APCOL : Vers une norme : 101 Conseils pour la Programmation en Fortran ; Note EDF, Atelier Logiciel no. 35, mars 1982.

* [Beauchamp 81] Anne Beauchamp : LECIBM et ECRIBM ; Note EDF Cray no. 5, HI-3953/01, octobre 1981.

[Beauchamp 81a] Anne Beauchamp : Sous-Programme d'Allocation dynamique de Memoire ; Note Cray no. 7, HI-3984/01, novembre 1981.

* [Bossavit 79] Alain Bossavit : Bientot, la Vectorisation (Chronique d'Analyse numerique) ; BUCCER Numero 75, septembre 1979.

[Bossavit 81] Alain Bossavit, Bertrand Meyer : The Design of Vector Programs ; Note EDF Cray no. 9, HI-3694/01, octobre 1981 (Communication au Congres "International Symposium on Algorithmic Languages", Amsterdam, 26-29 octobre 1981 ; J. W. de Bakker (Ed.), North-Holland, 1982.

* [Bossavit 81a] Alain Bossavit : Vectorisation de quelques Algorithmes numeriques ; Note EDF Cray no. 6, HI-4001/01, decembre 1981.

[Bossavit 82] Alain Bossavit, Bertrand Meyer : Methods for Vector Programming ; a paraitre.

[Brinch Hansen 73] Per Brinch Hansen : Operating Systems Principles ; Prentice-Hall, 1973.

[Brinch Hansen 79] Per Brinch Hansen : A Keynote Address on Concurrent Programming ; Computer (IEEE), 12, 5, pages 50-56, mai 1979.

* [Butel 81] Remy Butel : Evaluation Theorique des Performances du Cray ; Note EDF Cray no. 10, HI-3947/00, octobre 1981.

[Cray Assemblage] Cray-1 Computer System - CAL Assembler Version 1 Reference Manual, Document Cray numero SR-0000, version I-01, juin 1981.

[Cray Bibliotheque] Cray-1 Computer System - Library Reference Manual, Document Cray numero SR-0014, version E-01, juin 1981.

[Cray Fortran] Cray-1 Computer System - FORTRAN (CFT) Reference Manual, Document Cray numero SR-0009, version H, aout 1981.

[Cray Machine] Cray-1 S Series Hardware Reference Manual, Document Cray numero HR-0808, juin 1980.

[Cray Presentation] The Cray-1 Computer System - Document Cray de presentation, sans numero ni date.

[Cray Progiciels] Scientific Applications Package Handbook, Document Cray sans numero, version C, aout 1981.

[Cray Systeme] Cray-1 Computer System - Cray OS Reference Manual ; Manual, Document Cray numero SR-0011, version I-02, juillet 1981.

* [De Drouas 81] Eric de Drouas : Vectoriser sur le Cray-1 ; Note EDF Cray no. 8, Atelier Logiciel no. 31, HI-3919/01, juin 1981.

* [De Drouas 81a] Eric de Drouas, Marie Farges : Performances du Cray-1 en Fortran ; Note EDF Cray no. 4, HI-3919/01, septembre 1981.

[Dungworth 79] : M. Dungworth : The Cray 1 Computer System ; dans [Infotech 79], tome 2, pages 51-76, numero de reference 80H324843 a la Documentation.

[Flanders 79] P.M. Flanders : Fortran Extensions for a Highly Parallel Processor ; dans [Infotech 79], tome 2, pages 117-134, numero de reference 80H324846 a la Documentation.

* [Glaziou 81] Y. Glaziou : Acces au Cray des Utilisateurs du Reseau Retina ; Buccer (Bulletin du Centre de Calcul des Etudes et Recherches), no. 98, decembre 1981.

[Higbie 79] Lee Higbie : Vectorization and Conversion of FORTRAN Programs for the Cray-1 (CFT) Compiler - Document Cray numero 2240207, juin 1979.

[Hoare 78] C.A.R. Hoare : Communicating Sequential Processes ; Communications of the ACM, volume 21, numero 8, pages 666-677, aout 1978.

[Infotech 79] Infotech State of the Art Report on Supercomputers ; tome 1 (Total Systems Issues) ; tome 2 (Invited Papers) ; Maidenhead, 1979. Numeros de reference 80H324840 et 80H324841 a la Documentation (certains articles ont en outre des numeros individuels).

* [Magnier 81] M. Magnier : Langage de Commande du Cray-1, Manuel de Reference ; Note EDF Cray no. 2, HI-3569/01, avril 1981 (version 2, 1982)

[Meyer 78] Bertrand Meyer et Claude Baudoin : Methodes de Programmation - Eyrolles, Paris, 1978.

[Meyer 79] Bertrand Meyer : La nouvelle Norme Fortran 77 ; Note EDF Atelier Logiciel numero 10, 1979.

* [Meyer 80] Bertrand Meyer, Eric de Drouas : Un Calculateur vectoriel - le Cray-1 et sa programmation ; Film video, septembre 1980.

[Moreau 1981] Rene Moreau : Ainsi naquit l'informatique ; Dunod, Paris, 1981.

[Perrot 79a] R.H. Perrot : Parallel Languages, dans [Infotech 79], tome 1, pages 117-149, numero de reference 80H324836 a la Documentation.

[Perrot 79b] R.H. Perrot : A Standard for Supercomputer Languages, dans [Infotech 79] , tome 2, pages 291-308, numero de reference 80H324848 a la Documentation.

[Perrot 79c] R.H. Perrot : A Language for Array and Vector Processors ; TOPLAS (Transactions on Programming Languages and Systems, ACM), volume 1, numero 2, pages 177-195, 1979.

[Williams 79] S.A. Williams : The Portability of Programs and Languages for Vector and Array Processors ; dans [Infotech 79], tome 2, pages 371-394, numero de reference 80H324849 a la Documentation.

* Chronique reguliere "Cray" au Buccer (Bulletin du Centre de Calcul des Etudes et Recherches).

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 : Atelier logiciel n° 24

TITRE : UN CALCULATEUR VECTORIEL ; LE CRAY-1 ET SA PROGRAMMATION.

VERSION : 2, le 4 Juin 1980

Votre nom :

Votre adresse précise :

Désirez-vous

- recevoir à l'avenir les nouvelles notes de la série Atelier logiciel ?

☐ oui ☐ non

- recevoir dès maintenant les notes AL n° :

Vos commentaires sur cette note et/ou le produit qu'elle décrit :

SERIE "ATELIER LOGICIEL"

Les notes de cette série présentent, soit des éléments de méthodologie de la programmation, soit ces outils logiciels d'intérêt général, destinés à faciliter le travail des programmeurs.
Dans le second cas, les programmes correspondants sont écrits et documentés selon les normes strictes destinées à leur donnner une bonne fiabilité et une utilisation commode. L'Atelier logiciel entend assurer une *garantie de qualité* aux programmes et à leur documentation. Il est donc instamment demandé aux utilisateurs de faire part aux auteurs de toute déficience constatée et de toute suggestion quant aux améliorations possibles.

LISTE DES NOTES "ATELIER LOGICIEL"

(état au 29 Mars 1982)

			n° et date de la dernière version
AL 1	B. Meyer	- *L'Atelier logiciel.*	2, octobre 1981
AL 2	E. Audin	- *MASQUE : Récupération des erreurs à l'exécution.*	2, avril 1981
AL 3	B. Logez	- *FORTRACE : Aide à la mise au point (trace et profil)*	2, avril 1981
AL 4	B. Meyer	- *ENSORCELE 1 : Entrées et sorties sans format. (1ère partie : sorties)*	5, avril 1981
AL 5	B. Meyer	- *TEXTES : Manipulation de caractères en Fortran.*	à paraître
AL 6	B. Meyer	- *CHRONOS : Mesures précises de temps d'exécution.*	3, avril 1981
AL 7	G. Brisson B. Meyer F. Vapné	- *ENSORCELE 2 : Entrées et sorties sans format. (2ème partie : entrées)*	1, mai 1981
AL 8	B. Meyer	- *Un ramasse-miettes par tri.*	2, septembre 1978
AL 9	E. Audin	- *APOTHECE : Gestion et documentation automatiques des bibliothèques de programmes.*	4, avril 1981
AL 10	B. Meyer	- *Le langage FORTRAN 77.*	1, juin 1980
AL 11	N.Andriantsizafy	- *MORTRAN : Un processeur de macros et un sur-langage de FORTRAN.*	1, mai 1978
AL 12	B. Logez	- *Utilisation pratique de l'analyseur syntaxique ANAGIC.*	1, février 1979
AL 13	G. Brisson	- *TRI : Tri interne rapide.*	2, avril 1981
AL 14	M. Demuynck B. Meyer	- *Les languages de spécification.*	4, avril 1981
AL 15	B. Meyer	- *Les concepts de SIMULA 67.*	1, juin 1979
AL 16	N. Penot	- *SIMULA 67.*	4, juin 1979
AL 17	N. Penot	- *Résolution d'un problème de simulation en SIMULA 67.*	1, janvier 1979
AL 18	G. Brisson	- *Procédures TSO conversationnelles d'un intérêt général : la bibliothèque AL.*	2, avril 1981
AL 19	E. Audin	- *AXEDIR : Accès direct.*	3, avril 1981
AL 20	P. Gaudron B. Lalande B. Meyer	- *REDUCE : Calcul symbolique.*	1, août 1979
AL 21	B. Meyer	- *ADA(GREEN), le langage du DOD.*	à paraître
AL 22	E. Audin G. Brisson B. Meyer	- *GESCRAN : Gestion d'écrans en mode page.*	3, avril 1981
AL 23	B. Meyer	- *Sur le formalisme dans les spécifications.*	1, novembre 1979
AL 24	B. Meyer	- *Un calculateur vectoriel : le Cray-1 et sa programmation.*	4, janvier 1982
AL 25	E.de Drouas B.Meyer	- *INDEX : constitution de l'index d'un document.*	2, août 1980
AL 26	B. Logez	- *CONTREX : Contrôle conversationnel de l'exécution d'un programme.*	1, avril 1981
AL 27	B. Meyer	- *Panorama des langages de programmation.*	1, avril 1981
AL 28	B. Meyer	- *La spécification.*	6, avril 1981
AL 29	B. Meyer	- *Principles of Package Design.*	1, mai 1981
AL 30	B. Meyer	- *Errare : Outils de traitement des erreurs.*	à paraître
AL 31	E.de Drouas	- *Vectoriser sur le Cray-1.*	1, août 1981
AL 32	E.de Drouas	- *Manuel d'utilisation de SVP.*	1, octobre 1981
AL 33	B. Meyer	- *Un environnement conversationnel à deux dimensions*	1, janvier 1982
AL 34	B. Meyer	- *J'ai même rencontré des programmeurs heureux.*	1, janvier 1982
AL 35	F. Vapné	- *Vers une norme : cent-un conseils pour la programmation en Fortran.*	1, avril 1982

Toutes les notes ci-dessus peuvent être demandées à Mme Fumadelles (Clamart C121, Tél.765.37.46). On peut aussi demander à recevoir régulièrement les nouvelles notes de la série.