Étude des performances du calculateur vectoriel ST 100

Christine Eisenbeis, Jocelyne Erhel

IRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1)39.635511

Rapports Techniques

N° 51

ÉTUDE DES PERFORMANCES DU CALCULATEUR VECTORIEL ST 100

Christine EISENBEIS
Jocelyne ERHEL

Mai 1985

ETUDE DES PERFORMANCES DU CALCULATEUR VECTORIEL ST100

C. EISENBEIS
J. ERHEL

INRIA

Avril 1985

ETUDE DES PERFORMANCES DU CALCULATEUR VECTORIEL ST100

RESUME

Ce rapport présente des tests effectués sur l'ordinateur vectoriel ST100 de Star Technologies.

La première partie est consacrée à la description des caractéristiques matérielles et logicielles du calculateur. On développe ensuite différentes techniques de programmation pour exploiter au mieux les spécificités de la machine.

Un exemple d'application, la résolution d'une équation aux dérivées partielles par méthode spectrale, illustre les performances du ST100. A titre de comparaison, on a effectué les mêmes mesures sur un CRAY-1S.

Enfin, un exemple tiré de l'algèbre linéaire montre comment concevoir et optimiser un programme écrit en langage assembleur.

ABSTRACT

This report gives an outline of some applications we have run on the array-processor ST100 of Star Technologies.

In the first part, we describe the hardware and software components of the computer. Then we develop some programming tools to take into account the special features of the array-processor.

An example of application, the resolution of a partial differential equation by a spectral method, illustrates the ST100 capabilities and performances. We compare the results with those obtained on a CRAY 1S.

Finally, with the example of a linear algebra routine, we discuss how to design and optimize a program written in assembly language.

TABLE DES MATIERES

INTRODUCTION

L'introduction de parallélisme dans les ordinateurs a permis d'accroître sensiblement leur vitesse d'exécution. L'architecture de ces machines exploite deux types de parallélisme : d'une part, le traitement des données "à la chaîne" adapté au calcul vectoriel, d'autre part, la multiplication d'unités fonctionnelles traitant simultanément plusieurs jeux de données.

A côté des super-calculateurs tels que Cray-1S, Cray-XMP, Control Data Cyber 205, Fujitsu VP200, Hitachi S810, ... , ont été développés des ordinateurs vectoriels appelés array-processeurs comme FPS 164, ST100, ... Ces array-processeurs sont en moyenne moins puissants et plus spécialisés que les premiers, mais également moins coûteux. Par exemple, certains sont bien adaptés aux calculs numériques nécessitant des Transformées de Fourier.

Nous avons eu la possibilité de programmer plusieurs applications sur le ST100. Dans ce rapport, nous décrivons les aspects matériels et logiciels de ce calculateur, et les résultats obtenus.

Nous avons programmé une méthode spectrale pour résoudre un problème non linéaire évolutif. Cette application est écrite dans un langage APCL analogue au Fortran, et utilise des sous-programmes vectoriels de bibliothèque écrits en assembleur. Nous donnons des résultats d'exécution sur le ST100, et à titre comparatif sur un Cray-1S.

Nous avons également étudié sur un exemple la programmation d'une macro en langage assembleur, et les problèmes d'optimisation associés.

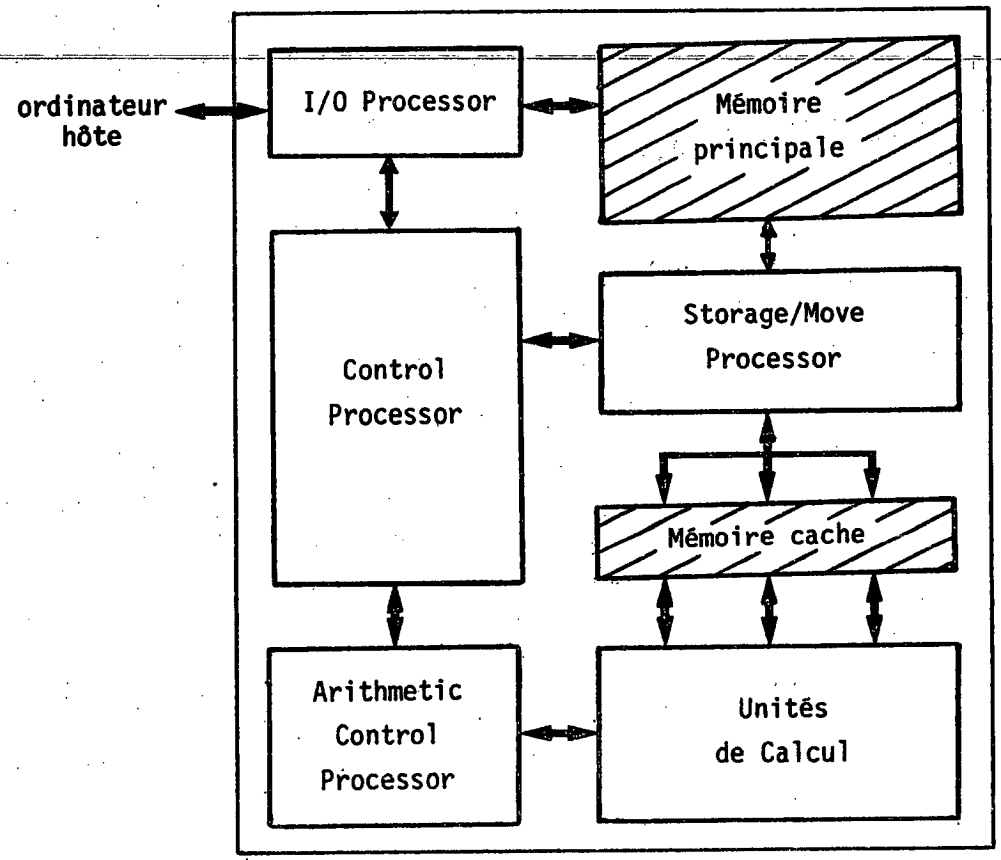
Nous tenons à remercier M. Calderaro de la société Star Technologies France d'avoir mis à notre disposition un ordinateur ST100. Nous sommes reconnaissantes à M. Laleous de son aide et de ses conseils efficaces pour la programmation et la mise en oeuvre d'applications sur ce calculateur vectoriel.

Le temps de calcul utilisé sur Cray-1S a été attribué par le Conseil Scientifique du Centre de Calcul Vectoriel pour la Recherche.

A - PRESENTATION DU ST100

Le ST100 est un calculateur vectoriel et un multi-processeur qui comporte plusieurs niveaux de parallèlisme, tous contrôlables à l'aide de langages de programmation adaptés. Il doit être rattaché à un ordinateur, dit ordinateur "hôte".

I - ARCHITECTURE



1 - PROCESSEURS

-- CP = Control Processor ou processeur de contrôle : il organise le travail des processeurs SMP et ACP.
-- ACP = Arithmetic Control Processor ou processeur de calcul.
-- SMP = Storage/Move Processor ou processeur de transfert des données.
-- I/OP = Processeur d'entrées/sorties des données : il effectue la liaison entre le ST100 et l'ordinateur hôte.

Le SMP transfère les données de mémoire principale en mémoire cache (mémoire de travail de l'ACP), l'ACP effectue des calculs sur ces données et le SMP retourne les résultats de la mémoire cache en mémoire principale. L'ACP et le SMP peuvent travailler simultanément, c'est là une première forme de parallélisme possible dans le ST100. Les programmes qu'ils exécutent (dits "MACROS"), sont écrits dans un langage assembleur (MAL = Macro Assembly Language).

Le CP est composé de 2 microprocesseurs Motorola 68000 qui peuvent opérer simultanément : l'un gère l'interaction des processeurs ACP et SMP (exécution de macros, synchronisation, ...) par l'intermédiaire d'un programme, dit "PROCESS", écrit en APCL (Array Processor Control Language, langage très proche du FORTRAN). Le second microprocessor contient l'APM (array processor monitor) qui contrôle le transfert des PROCESS et données entre le ST100 et l'ordinateur hôte.

L'I/OP sert de lien entre le ST100 et l'ordinateur hôte. Il est commandé par un programme FORTRAN ("PROGRAM") qui est chargé de transférer dans le ST100 les process et les tableaux de données. Notons que ce programme tourne sur l'hôte.

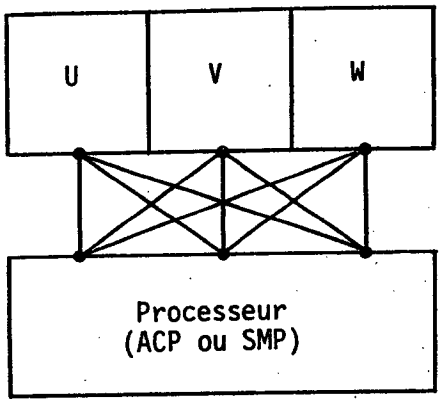
Le temps de cycle du ST100 est de 40 nanosecondes.

2 - MEMOIRES

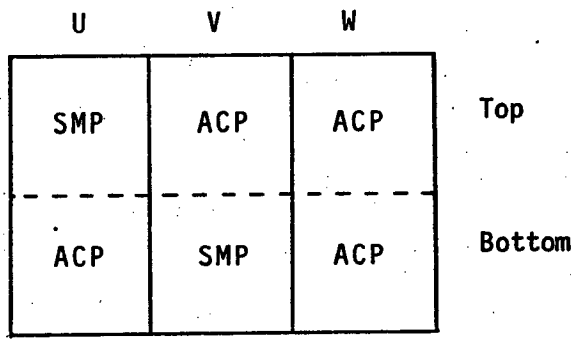
Le ST100 possède, dans le CP, une mémoire locale qui contient les données accessibles du process.

Les programmes (process et macros) et les données sont rangés dans la mémoire principale. Ils sont gérés par le CP et le SMP. La mémoire principale consiste en 8 bancs entrelacés, sa capacité peut aller jusqu'à 8 millions de mots de 32 bits.

La mémoire cache est partagée en trois bancs accessibles en parallèle par l'intermédiaire de réseaux "cross-bar" établissant une correspondance bi-univoque entre les 3 bancs du cache et les 3 accès des processeurs ACP et SMP. Chaque banc a une capacité de 16K mots de 32 bits, capacité qui sera portée à 64 K mots dans une prochaine version.



En outre, chaque banc peut être partagé en deux moitiés (Top et Bottom) qui peuvent être affectées indifféremment à l'un ou l'autre des processeurs ACP ou SMP.



- Exemple d'affectation -

Ceci permet aux 2 processeurs ACP et SMP de travailler en parallèle en partageant la mémoire cache (voir exemples ci-dessous). Le partage en 3 bancs permet, lui, 3 accès simultanés entre processeur et cache. Ceci est valable pour l'ACP, qui peut lire/écrire dans les 3 bancs en même temps (multiplication du débit mémoire), mais pas pour le SMP, qui n'a qu'un accès au cache par temps de cycle.

II - DIFFERENTS NIVEAUX DE PROGRAMMATION

1 - MACROS

Ce sont les programmes écrits en MAL (Macro Assembly Language) qui commandent le SMP et l'ACP. Deux formes de parallélisme apparaissent à ce niveau :

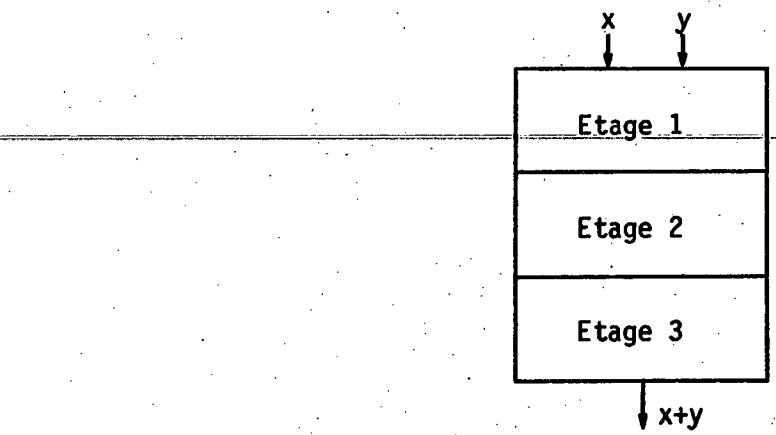
* La multiplicité des unités fonctionnelles

L'ACP contient 2 additionneurs et 2 multiplieurs qui peuvent

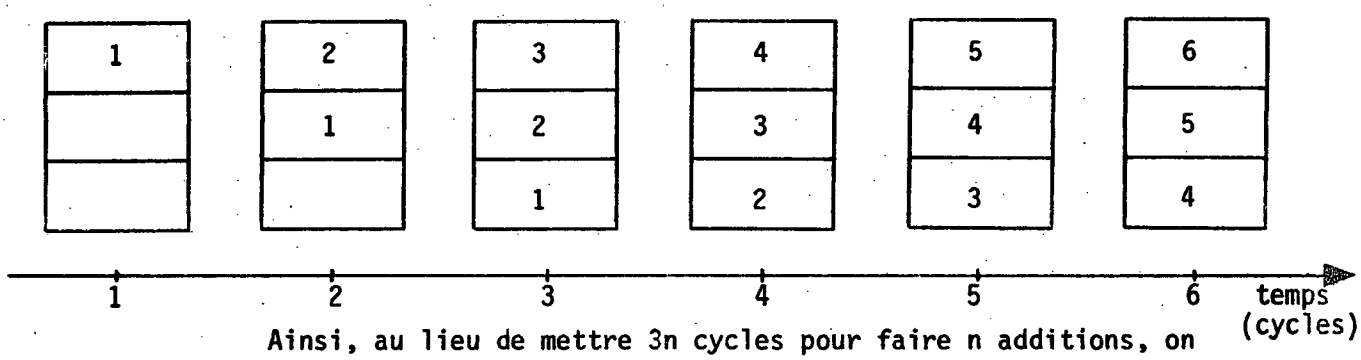
travailler tous les quatre en même temps. Il faut bien sûr que leurs registres soient alimentés en conséquence et d'autre part, que les tâches simultanées soient indépendantes.

* La structure pipeline des unités fonctionnelles

Par exemple, l'additionneur se compose de 3 étages



et il faut donc 3 cycles pour faire une addition. Le pipeline consiste, dans une addition de vecteurs, à alimenter chaque étage de façon continue : dès que le calcul sur une donnée y est terminé, on transmet le résultat à l'étage suivant et on y entre la donnée suivante.



Ainsi, au lieu de mettre 3n cycles pour faire n additions, on utilise seulement (n+2) cycles et un temps de calcul divisé par 3 pour les grands vecteurs.

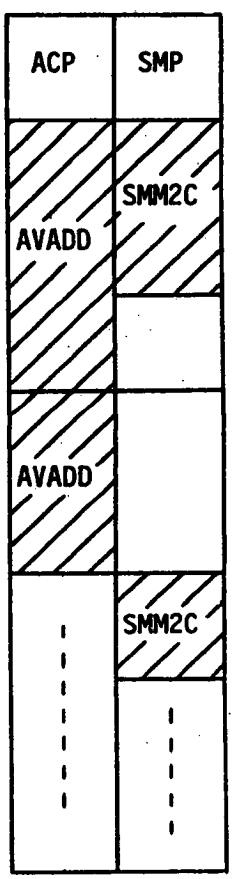
Remarquons que les principales "macros" (par exemple les plus couramment utilisées en calcul scientifique) sont fournies par le constructeur, depuis la simple addition de vecteurs jusqu'à des algorithmes plus compliqués tels que la FFT. On peut dès lors écrire la majorité des

programmes sans avoir recours à la programmation en assembleur. Mais, comme on le verra plus loin, les "overhead", dus aux temps d'appel des macros, sont importants et l'écriture d'un programme en assembleur peut éviter des pertes de temps non négligeables.

2 - PROCESS

Ce sont les programmes écrits en APCL qui contrôlent et synchronisent les processeurs ACP et SMP. Ils se présentent comme suit :

```
---------------------------------------------------------------------
        Process EXEMP(....)
C déclaration des tableaux et scalaires dans les 3 mémoires
        MAINMEMORY
        .......
        LOCALMEMORY
        .......
        CACHEMEMORY
        .......
              .
              .
              .
C commande de synchronisation ------------(1)------------
        CALL STSYNC(101010)
C exécution d'une MACRO ACP
        CALL AVADD(...)
C exécution d'une MACRO SMP
        CALL SMM2C(...)
C commande de synchronisation ------------(2)------------
        CALL STSYNC(111111)
C exécution d'une MACRO ACP
        CALL AVADD(...)
C commande de synchronisation ------------(3)------------
        CALL STSYNC(000000)
C exécution d'une MACRO SMP
        CALL SMC2M(...)
              .
              .
              .
              .
C commande d'attente
        CALL STWSMP
        END
---------------------------------------------------------------------
```



N.B. : (AVADD est une MACRO ACP qui additionne 2 vecteurs)
(SMM2C et SMC2M sont des MACROS SMP de transferts de vecteurs main → cache et cache → main respectivement).

Déroulement du process :

Comme l'ACP et le SMP travaillent indépendamment, deux macros ACP et SMP appelées à la suite l'une de l'autre vont s'exécuter simultanément dès le top de départ donné par la commande APCL : STSYNC(---(1)---). Celle-ci sert d'autre part à synchroniser les deux processeurs : elle signale que l'ACP et le SMP doivent s'attendre avant de commencer toute autre tâche (---(2)---). Son argument assigne aux deux processeurs leurs zones d'accès au cache : c'est un nombre binaire à 6 bits, chacun représentant une partie du cache, égal à 0 pour le SMP et 1 pour l'ACP ;

bit 1 : banc U Top
bit 2 : banc U Bottom
bit 3 : banc V Top
bit 4 : banc V Bottom
bit 5 : banc W Top
bit 6 : banc W Bottom

Ainsi STSYNC(010101) signale que les MACROS SMP qui vont suivre travailleront sur les parties "Top" des bancs de cache et les MACROS ACP sur les parties "Bottom". Les données doivent bien sûr être organisées en conséquence.

Il existe d'autres commandes d'attente de fin d'exécution des tâches telles que :

- STWACP : attente de l'ACP
- STWSMP : attente du SMP
- STWAP : attente des 2 processeurs.

On les utilise par exemple avant le "END" du process, pour s'assurer que les tâches SMP et ACP ont bien été menées à terme, avant de retourner au programme principal.

3 - PROGRAMS - LIAISON HOTE ⟷ ST100

Le ST100 est rattaché à un ordinateur hôte qui le dirige par un programme FORTRAN.

Les tâches de l'hôte sont :

- "ouvrir" l'AP : assigner un nombre logique à une "partition" dans le ST100, c'est-à-dire une région dans la mémoire principale où l'on rangera process et données.

- réserver dans le ST100 l'espace mémoire ("partition") nécessaire au chargement du process et des données.

- déclarer les tableaux dans le ST100 (identifiés par un tableau à 3 entiers dans l'hôte).

- transférer les données dans ces tableaux.

- charger le process, l'exécuter.

- lire les résultats.

- libérer l'AP.

- fermer l'AP.

Ces tâches se font par appel à des subroutines, voir exemple ci-après. On verra que ILUN et ISTAT sont 2 arguments communs à toutes ces subroutines :

- ISTAT est un code d'erreur, il permet de vérifier que toutes les tâches se passent bien (utile en particulier pour mettre au point un process !).

- ILUN est le numéro que l'on assigne à une "partition" dans le ST100.

EXEMPLE DE PROGRAMME : ADDITION DE 2 VECTEURS DE TAILLE 10000

```
      PROGRAM EXAMP1
C                          EXECUTE C=A+B
      REAL A(10000),B(10000),C(10000)
C                          A,B,C SONT 3 VECTEURS DE TAILLE 10000
      INTEGER ICA(3),ICB(3),ICC(3)
C                          CES TROIS TABLEAUX D'ENTIERS IDENTIFIENT
C                          LES 3 TABLEAUX CORRESPONDANT A A,B,C
C                          DANS LE ST100
      IF (ISTAT.NE.0) GOTO 100
C                          CAS D'ERREUR
      ILUN=50
      CALL STOPENW(ILUN,ISTAT)
C                          OUVRE L'AP, LUI ASSIGNE UN NUMERO LOGIQUE : 50
      IF (ISTAT.NE.0) GOTO 100
      CALL STSCH(ILUN,ISTAT,'DSIZE',40,'PSIZE',4)
C                          TAILLE DE LA PARTITION A RESERVER : 40 KW
C                          POUR LES DONNEES ET 4 KW POUR LE PROCESS
      IF (ISTAT.NE.0) GOTO 100
C                          DECLARE LES 3 TABLEAUX DANS LA PARTITION
      CALL STARAY(ILUN,ISTAT,ICA,10000,'REAL')
      IF (ISTAT.NE.0) GOTO 100
      CALL STARAY(ILUN,ISTAT,ICB,10000,'REAL')
      IF (ISTAT.NE.0) GOTO 100
      CALL STARAY(ILUN,ISTAT,ICC,10000,'REAL')
      IF (ISTAT.NE.0) GOTO 100
C                          TRANSFERE LES DONNEES DE L'HOTE
C                          VERS LA MEMOIRE PRINCIPALE
      CALL STWRW(ILUN,ISTAT,A,10000,ICA)
      IF (ISTAT.NE.0) GOTO 100
      CALL STWRW(ILUN,ISTAT,B,10000,ICB)
      IF (ISTAT.NE.0) GOTO 100
C                          CHARGE LE PROCESS ET L'EXECUTE
      CALL VADDW(ILUN,ISTAT,JSTAT,ICA,ICB,ICC,10000)
      IF (ISTAT.NE.0) GOTO 100
C                          TRANSFERE LES RESULTATS DE LA MEMOIRE
C                          PRINCIPALE VERS L'HOTE
      CALL STRDW(ILUN,ISTAT,C,10000,ICC)
      IF (ISTAT.NE.0) GOTO 100
C                          LIBERE L'AP
      CALL STREL(ILUN,ISTAT)
      IF (ISTAT.NE.0) GOTO 100
C                          FERME L'AP
      CALL STCLOS(ILUN,ISTAT)
      IF (ISTAT.NE.0) GOTO 100
      STOP
C                          TRAITEMENT DES CAS D'ERREURS
100   ..........
      END
```

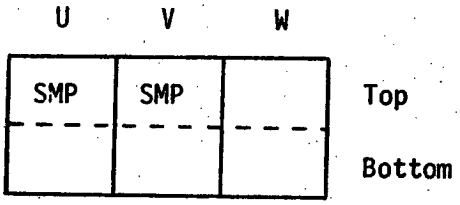
B - ORGANISATION D'UN PROCESS

I - DOUBLE BUFFER POUR LES GRANDS VECTEURS

La mémoire cache a une capacité limitée (3*16K mots). Les calculs sur les grands vecteurs requièrent donc un traitement par paquets. On utilise à cette fin l'indépendance des parties "Top" et "Bottom" du cache et le parallélisme entre les processeurs ACP et SMP, le SMP exécutant les transferts sur une partie du cache et l'ACP les calculs sur l'autre partie.

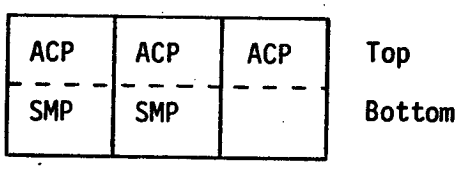
EXEMPLE : ADDITION DE 2 VECTEURS DE GRANDE TAILLE C=B+A

1) Transfert des 8K premiers mots de A et B dans les bancs UTop et VTop (tâche SMP) ;



2) Transfert des 8K mots suivants de A et B dans les bancs UBottom et VBottom (tâche SMP) ;

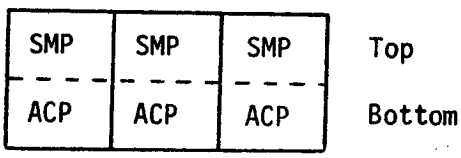
Calcul de UTop + VTop, résultat dans WTop (tâche ACP) ;



3) Transfert des 8K premiers résultats (WTop) du cache en mémoire principale (tâche SMP) ;

Transfert des 8K mots suivants de A et B de la mémoire principale dans les bancs UTop et VTop (tâche SMP) ;

Calcul de UBottom + VBottom, résultat dans WBottom (tâche ACP) ;



... etc

Ainsi, au lieu de s'ajouter, les temps de travail des 2 processeurs ACP et SMP se recouvrent ("overlapping").

II - DOUBLE BUFFER POUR LE TRAITEMENT DES PETITS VECTEURS

L'overlapping d'un processeur par l'autre pourrait aussi, en théorie s'utiliser pour diminuer le temps de calcul sur des petits vecteurs : supposons par exemple que l'on veuille additionner 2 vecteurs de taille N. Pour cela, on utilise les MACROS :

(SMP)	SMM2C		: Transfère un vecteur de mémoire principale vers le cache ;
(SMP)	SMC2M		: Transfère un vecteur de cache en mémoire principale ;
(ACP)	AVADD	(A,B,C)	: Calcule C=A+B ou A,B,C sont 3 vecteurs du cache, dans 3 bancs différents.

Pour des vecteurs de taille P, les durées d'exécution de ces MACROS sont :

SMM2C	P+25 cycles
SMC2M	P+20 cycles
AVADD	P+9 cycles

On divise alors les vecteurs à additionner en Q parties de taille P (N=P*Q) et on traite les vecteurs par paquets en utilisant le double buffer ; le programme s'écrit, pour Q pair :

PROGRAMME	TEMPS D'EXECUTION (en cycles)	ACP	SMP	
SMM2C en Top (2 fois)	2P+50		SMM2C	
			SMM2C	STSYNC
AVADD en Top	max(2P+50,P+9)=2P+50	AVADD	SMM2C	
SMM2C en Bottom (2 fois)			SMM2C	STSYNC
		AVADD	SMC2M	
FAIRE Q/2-1 FOIS :				
! SMC2M en Top	max(3P+70,P+9)=3P+70		SMM2C	
! SMM2C en Top (2 fois)			SMM2C	
! AVADD en Bottom				STSYNC
		AVADD	SMC2M	
! AVADD en Top	max(3P+70,P+9)=3P+70			
! SMC2M en Bottom			SMM2C	
! SMM2C en Bottom (2 fois)			SMM2C	STSYNC
AVADD en Bottom	max(P+9,P+20)=P+20	AVADD	SMC2M	
SMC2M en Top				STSYNC
			SMC2M	
SMC2M en Bottom	P+20			

Le temps d'exécution est donc, en cycles : (Q/2-2)(6P+140)+6P+140. Pour par exemple Q=4, on obtient 6P+40 = 3N/2+140 cycles alors que si on avait fait le calcul en bloc (en supposant N≤16K (= taille d'un banc du cache)) :

SMM2C (2 fois)	2N+50
AVADD	N+9
SMC2M	N+20

on aurait eu 4N+79 cycles !

Malheureusement, ce calcul n'est que théorique ; en pratique, il faut tenir compte du temps d'appel des MACROS ("overhead"). Ce temps d'appel est de 50 microsecondes environ mais peut être en partie masqué :

Exemple : soit la suite d'instructions

```
CALL  MACRO1
CALL  MACRO2
```

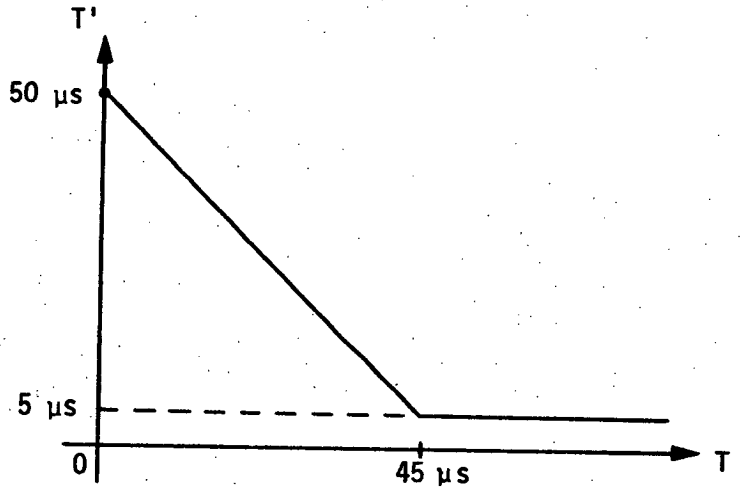
Pendant l'exécution de MACRO1, le CP commence à préparer MACRO2

(arguments à appeler, ...etc). Si le temps d'exécution de MACRO1 est assez long, il masque alors le temps d'appel de MACRO2, qui peut être réduit jusqu'à 5 microsecondes. Schématiquement :

Soit T le temps d'appel de MACRO1, T' le temps réel d'overhead (i.e. le temps écoulé entre la fin d'exécution de MACRO1 et le début de MACRO2) ;

si T≤45 microsecondes, T'=50 - T microsecondes ;

si T≥45 microsecondes, T'=5 microsecondes ;



Ainsi, on voit que le double buffer n'est efficace que pour des temps d'exécution assez longs, c'est-à-dire que la taille des vecteurs à traiter par macro doit être assez grande. Les mesures effectuées à ce sujet montreront qu'il est utopique d'espérer gagner du temps par double buffer sur des petits vecteurs.

III - MOINS D'OVERHEAD : LES "LOOPING-MACROS"

On remédie au problème d'overhead évoqué plus haut en utilisant les "looping-macros" : ce sont des MACROS fournies en bibliothèque qui bouclent autour des MACROS élémentaires ; on peut donc exécuter plusieurs fois une même MACRO sans sortir du processeur (ACP ou SMP), donc sans additionner les overheads dûs aux "call".

Au lieu d'avoir

```
faire de i=1 à n
     [call macro(i)
fin faire
```

qui demande n "call", on utilise

```
: call looping-macro(i,i=1..n)
```

qui ne demande plus qu'un "call".

Avantages :

(i) la diminution du nombre de "call".

(ii) le temps d'exécution par macro est plus long, donc susceptible de masquer l'overhead de la macro suivante.

Inconvénients :

(i) le temps de calcul est plus long entre 2 synchronisations (STSYNC) et ceci restreint les possibilités de faire travailler les 2 processeurs ACP et SMP en parallèle.

(ii) on est limité par la taille du cache ce qui peut conduire à combiner looping-macros et double buffer.

IV - RECAPITULATION

Nous allons ici comparer les différentes méthodes présentées ci-dessus sur un exemple modèle. Les chiffres indiqués ici (overhead, temps de calcul ..) ne sont donnés que pour indiquer un ordre de grandeur, ils ne correspondent que très grossièrement à la réalité.

Supposons qu'on ait le même calcul à faire sur M vecteurs de même taille N,V(1),V(2),...V(M). On note CV(1),CV(2),...CV(M) leurs adresses dans le cache (éventuellement identiques).

Appelons :

* SMPI(V,CV) la MACRO SMP qui transfère le vecteur V de mémoire principale dans le vecteur CV du cache ; durée d'exécution supposée : 0,1xN microsecondes.

* SMPO(V,CV) la MACRO SMP qui transfère le vecteur CV du cache dans le vecteur V de mémoire principale ; durée d'exécution supposée : 0.1xN microsecondes.

* ACP(CV) la MACRO ACP qui exécute le calcul sur CV et retourne le résultat en CV ; durée d'exécution supposée : 0.1xNLog2(N) microsecondes.

* LSMPI,LSMPO,LACP les 3 LOOPING-MACROS correspondantes ; leurs durées d'exécutions respectives sont donc :

pour M vecteurs, 0.1xMN, 0.1xMN, 0.1xMNLog2(N).

Un tableau donne les conclusions qualitatives (tableau A). Nous avons d'autre part représenté sous forme de graphique les temps CPU estimés avec les temps d'overhead calculés précédemment, et avec, N=M. L'abscisse représente le nombre d'opérations arithmétiques (N*N*Log2(N)) et l'ordonnée le temps CPU.

Examinons les différentes manières d'écrire le programme.

Note : les instructions à l'intérieur du [sont exécutées en parallèle.

1) Ecriture directe du programme

```
faire de i=1 à n
        [call SMPI(V(i),CV(i))
        [call ACP(CV(i))
        [call SMPO(V(i),CV(i))
fin faire
```

2) Double Buffer

```
[call SMPI(V(1),CV(1))

⌈call SMPI(V(2),CV(2))
⌊call ACP(CV(1))

faire de i=2 à (M-1)
                ⌈call SMPO(V(i-1),CV(i-1))
                |call SMPI(V(i+1),CV(i+1))
                ⌊call ACP(CV(i))
fin faire

⌈call SMPO(V(M-1),CV(M-1))
⌊call ACP(CV(M))

[call SMPO(CV(M),V(M))
```

3) Looping-macros

```
[call LSMIP(V(i),CV(i), i=1,..M)
[call LACP(CV(i))
[call LSMPO(V(i),CV(i), i=1,..M)
```

CONCLUSIONS

Le tableau A montre :

* le travail en parallèle des 2 processeurs (overlapping) ;
* le nombre de "call" (overhead) ;
* la taille des vecteurs traités (plus ils sont grands, plus le temps de calcul est long et donc susceptible de masquer l'overhead de la MACRO suivante) ;
* les contraintes sur le nombre (M) et la taille (N) des vecteurs.

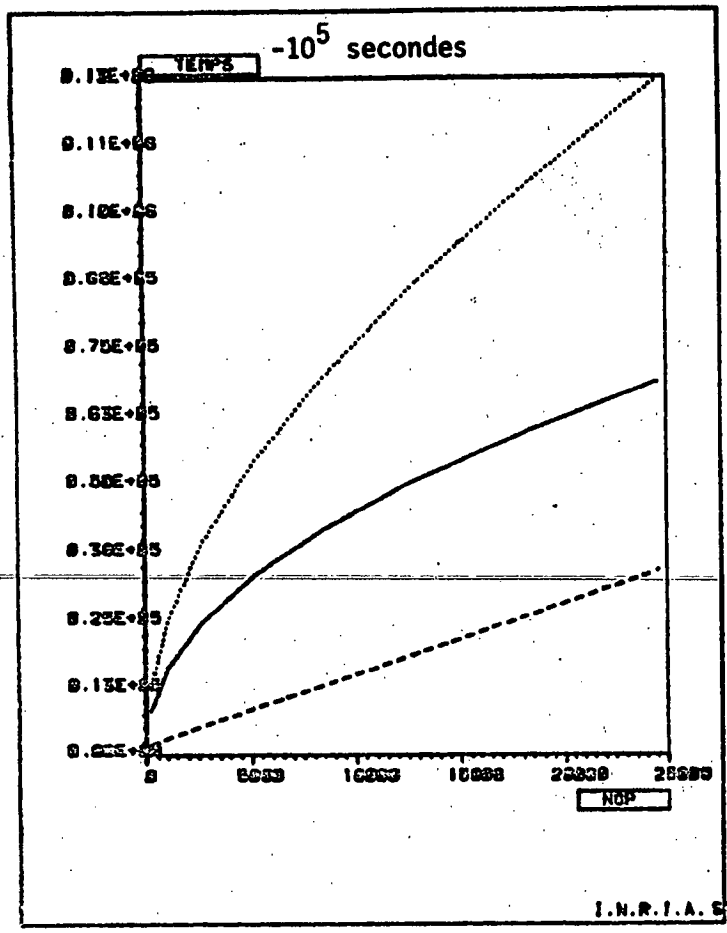
Il peut se résumer ainsi :

- Programmation normale : OVERHEAD et pas d'OVERLAPPING ;
- Double buffer : OVERHEAD mais OVERLAPPING ;
- Looping-macros : pas d'OVERHEAD mais pas d'OVERLAPPING.

Les graphiques d'estimation des temps de travail montrent qu'il est préférable d'utiliser les looping-macros pour les petits vecteurs et le double buffer pour les grands vecteurs, puisque asymptotiquement, la courbe des looping-macros s'identifie à celle de la programmation normale, moins bonne que le double buffer (la courbe des looping-macros pour le traitement des grands vecteurs est, de plus, irréaliste puisque la taille des vecteurs est, dans ce cas, limitée par la taille du cache).

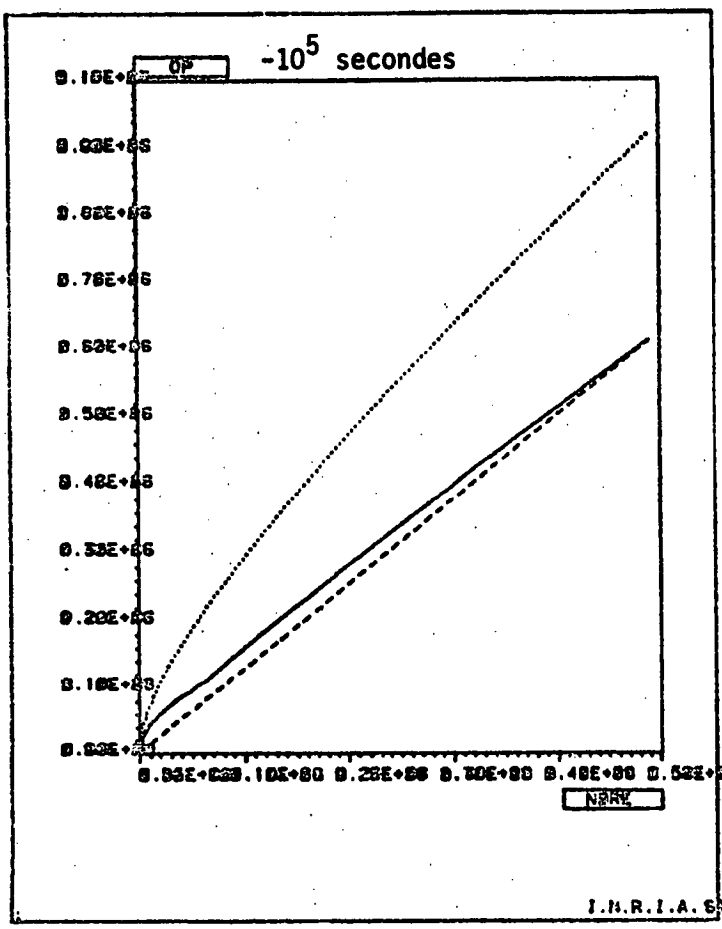
	Overlapping	Nombre de call	Taille des vecteurs traités	Contraintes sur M et N
Programmation normale	non	3M	N	N ≤ taille d'un banc de cache ; M illimité
Double buffer	oui	3M	N	N ≤ taille d'un 1/2 banc de cache ; M illimité
Looping macros	non	3	NM	MN ≤ taille d'un banc de cache
Double buffer + looping macros	oui	$3\frac{M}{P}$	PN	PN ≤ taille d'un 1/2 banc de cache ; M illimité

Tableau A



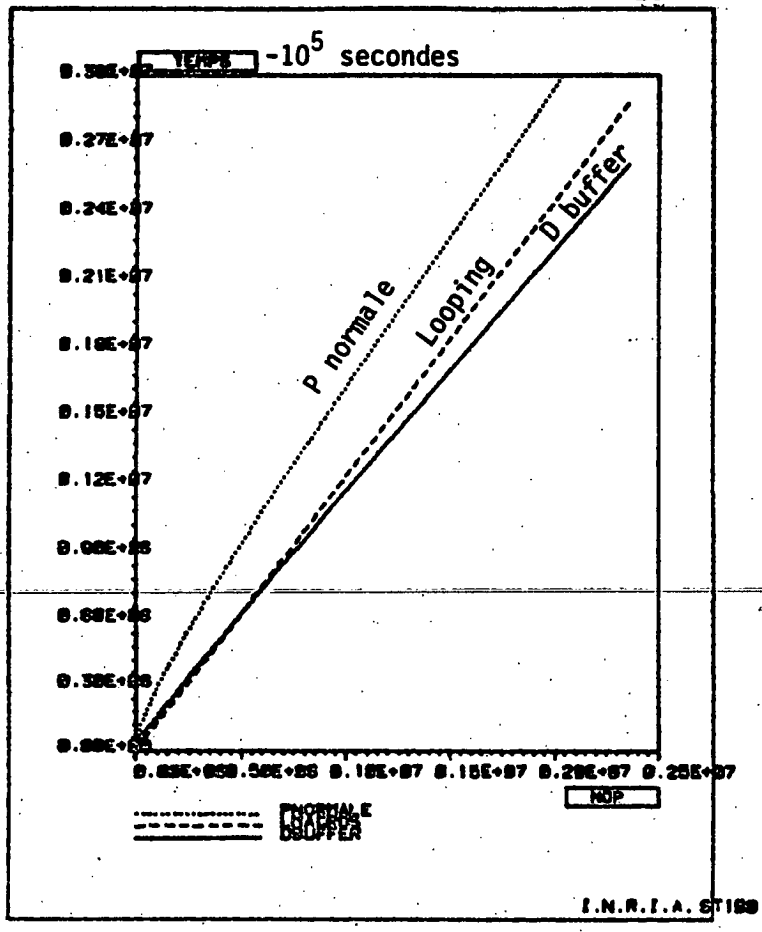
Graphique 1 : Estimation du temps de calcul sur les petits vecteurs. N = 0 ... 64

..... : programmation normale
——— : double buffer
----- : looping-macros.

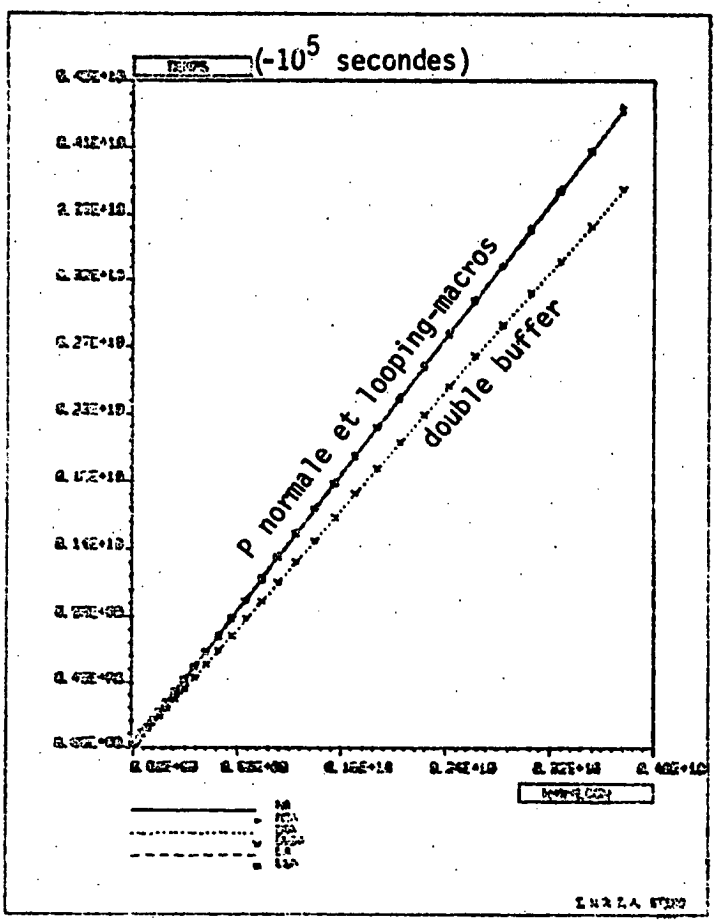


Graphique 2 : N = 0 ... 248

..... : programmation normale
——— : double buffer
----- : looping-macros.



Graphique 3 : N = 0 ... 512



Graphique 4 : Estimation du temps de calcul sur les grands vecteurs. N = 512 ... 16384

4) Une solution intermédiaire ?

Il semble a priori possible d'utiliser les looping-macros en double buffer pour, à la fois, faire travailler les 2 processeurs en parallèle et réduire l'overhead :

Soit M=PQ, le programme s'écrit :

```
[call LSMPI(V(i),CV(i), i=1,...P)

┌call LSMPI(V(i),CV(i), i=(P+1),...2P)
└call LACP(CV(i),i=1,...P)

faire de j=1 à (Q-2)
     ┌call LSMPO(V(i),CV(i), i=(j-1)P+1,...jP)
     ├call LSMPI(V(i),CV(i), i=(j+1)P+1,...(j+2)P)
     └call LACP(CV(i), i=jP+1,...(j+1)P)
fin faire

┌call LSMPO(V(i),CV(i), i=(Q-2)P+1,...(Q-1)P)
└call LACP(CV(i), i=(Q-1)P+1,...PQ)

[call LSMPO(V(i),CV(i), i=(Q-1)P+1,...PQ)
```

REMARQUES :

* contraintes sur M,N et P
  PN ≤ taille du cache
  M illimité
* travail en parallèle de l'ACP et du SMP
* nombre de "call" : 3Q
* taille des vecteurs traités : PN

Le choix de P est alors soumis à plusieurs contraintes :

- optimisation de l'overlapping d'un processeur par l'autre ;
- PN ≤ taille du cache ;
- P assez grand
  - (i) pour que le nombre de "call" soit petit (il est égal à 3M/P) ;
  - (ii) pour que la taille des vecteurs traités soit assez longue (pour masquer l'overhead).

On peut ainsi espérer diminuer le temps de calcul en optimisant la valeur de P. Le calcul du temps CPU pour des P différents, effectué avec les mêmes hypothèses que ci-dessus montre que l'on peut gagner environ 10 % par rapport au meilleur des 2 temps - double buffer, looping-macros - ceci pour les petites valeurs de N. Pour les grandes valeurs de N, l'overhead devient négligeable et le double buffer est bien sûr meilleur.

C - EXEMPLE : RESOLUTION D'UNE EQUATION D'EVOLUTION PAR METHODE PSEUDO-SPECTRALE

En guise d'exemple, on présente ici la résolution d'une équation aux dérivées partielles évolutive en dimension 2. On utilise pour cela une méthode pseudo-spectrale, ce qui nous conduit à implémenter une FFT bi-dimensionnelle. Celle-ci est programmée de 2 manières : double buffer et looping-macros.

I - LE PROBLEME A RESOUDRE

On étudie la résolution de l'équation sur $\Omega = [0,a] \times [0,b]$.

$$\frac{\partial u}{\partial x}(x,y,t) - \Delta u + \rho(u) = g(x,y) \quad \forall (x,y) \in \Omega \quad \forall t > 0$$

$$u(x,y,0) = u_0(x,y) \quad \forall (x,y) \in \Omega$$

avec conditions aux limites périodiques.

où $\rho(u) = |u|^{k-1} u \quad k \geq 1$.

1 - SCHEMA D'APPROXIMATION

* En temps :
  soit $\Delta t > 0$ et $u_p(x,y)$ l'approximation de u à $t = p\Delta t$,
  on calcule u_{p+1} par

$$\frac{u_{p+1} - u_p}{\Delta t} - \Delta u_{p+1} + \rho(u_p) = g.$$

* En espace :
  soient N et $M \in \mathbb{N}$, N et M puissances de 2,
  on se donne sur Ω les points de collocation $(xjk)_{\substack{j=0..N-1 \\ k=0..M-1}}$
  avec $xjk = (jh_N, kh_M)$ $\left[h_N = \frac{a}{N}, h_M = \frac{b}{M}\right]$

L'approximation pseudo-spectrale d'une fonction u périodique en x et y est alors donnée par :

$$\tilde{u}(x,y) = \frac{1}{NM} \sum_{p=-\frac{N}{2}}^{+N/2} \sum_{q=-\frac{M}{2}}^{M/2} \frac{\hat{u}_{pq}}{\alpha_p \beta_q} T_{pq}(x,y)$$

avec

$$T_{pq}(x,y) = \exp\left(-\frac{2i\Pi xp}{a} - \frac{2i\Pi yq}{b}\right)$$

$$\alpha_p = 2 \text{ pour } p = -\frac{N}{2}, +\frac{N}{2}$$
$$= 1 \text{ pour } |p| < \frac{N}{2}$$

$$\hat{U}_{pq} = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} u(xjk)\, T_{pq}(xjk)$$

$$\beta_q = 2 \text{ pour } q = -\frac{M}{2}, \frac{M}{2}$$
$$= 1 \text{ pour } |q| < \frac{M}{2}.$$

et la convention $\hat{U}_{pq} = \hat{U}_{p \bmod N,\, q \bmod M}$

i.e. $\tilde{u}(x,y) = \frac{1}{NM} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} \hat{U}_{pq}\, T_{\min(p,p-N),\, \min(q,q-N)}(x,y)$.

Alors $\tilde{u}(xjk) = u(xjk)$ et on passe de la matrice U des valeurs de u aux points $(xjk)_{\substack{j=0..N-1 \\ k=0..M-1}}$ à la matrice $\hat{U}$ des coefficients $(\hat{U}_{pq})_{\substack{p=0..N-1 \\ q=0..M-1}}$

par l'isomorphisme $F : M(N,M) \longrightarrow M(N,M)$
$$U \longrightarrow \hat{U}$$

où M(N,M) est l'ensemble des matrices NxM.

L'intérêt est que le calcul des dérivées spatiales est considérablement simplifié :

comme $\Delta\, T_{pq}(x,y) = \tilde{d}_{pq}\, T_{pq}(x,y)$

où $\tilde{d}_{pq} = -(\frac{2\Pi p}{a})^2 - (\frac{2\Pi q}{b})^2$,

on obtient

$$\Delta\, \tilde{u}(xjk) = \frac{1}{NM} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} \hat{U}_{pq}\, d_{pq}\, T_{pq}(xjk) \quad \text{avec } d_{pq} = \tilde{d}_{[\min(p,p-n)\min(q,q-M)]}$$

Posons $\Delta U = (\Delta\tilde{u}(xjk))_{\substack{j=0..N-1 \\ k=0..M-1}}$

alors

$$\Delta U = F^{-1}\, D\, F\, U$$

où $D : M(N,M) \to M(N,M)$ est l'opérateur diagonal qui à une matrice $A = (aij)$ associe la matrice $DA = (d_{ij} a_{ij})$.

On résoudra donc en fait

$$\frac{U_{p+1} - U_p}{\Delta t} - F^{-1}\, D\, F\, U_{p+1} + \rho(U_p) = G \quad \text{où} \quad G = (g(x_{jk})).$$

(Rem: si $U_p = (u_p^{ij})$

$$\tilde{\rho}(U_p) = (\rho(u_p^{ij})) \quad \text{et non pas} \quad \rho(U_p) = |U_p|^{k-1} U)$$

c'est-à-dire

$$(I-\Delta t\ D)\ F\ U_{p+1} = F\ (U_p - \Delta t\ \rho(U_p) + G).$$

$$U_{p+1} = F^{-1}\ (I - \Delta t\ D)^{-1}\ F\ (U_p - \Delta t\ \rho(U_p) + G)\ .$$

On note $\wedge$ l'opérateur diagonal $(I - \Delta t\ \Delta)^{-1}$

Ψ l'opérateur $\Psi(U) = U - \Delta t\ \rho(U) + G$

$$\boxed{U_{p+1} = F^{-1} \wedge F\ \Psi(U_p)}$$

2 - EVALUATION DU COUT DES CALCULS

* Ψ

En considérant la matrice U comme un vecteur (en rangeant par exemple colonne par colonne), alors le calcul de $\Psi(U)$ est vectoriel

$\Psi(U) = U - \Delta t\ |U|^{k-1}\ U + G$ de taille NM

* $\wedge$

De même A étant diagonal, son calcul revient à une multiplication vecteur (NM) par vecteur (NM).

* $F,\ F^{-1}$

Considérons toujours U comme un vecteur de taille NM, on peut alors représenter F comme une matrice NM x NM (Rem : $F^{-1} = \frac{1}{NM} F^*$) donc pratiquement le calcul de F(U) revient à NM multiplications vecteur (NM) x vecteur (NM) .

On voit que c'est le calcul de F et F^{-1} qui va prendre le plus de temps dans l'itération $U_p \longrightarrow U_{p+1}$.

On peut cependant réduire les calculs en utilisant la FFT (Fast Fourier Transform), qui calcule les sommes de la forme

$$X_k = \sum_{n=0}^{N-1} x_n \exp(\frac{2i\Pi kn}{N}) \quad k = 0 \ldots N-1.$$

(Rappelons que $FU = (\hat{u}_{pq})$ avec

$$\hat{u}_{pq} = \sum_{j=0}^{N-1} \sum_{k=0} u(xjk) \exp(\frac{2i\Pi jp}{N} + \frac{2i\Pi kq}{M}).)$$

3 - LA FFT EN DIMENSION 2

Les sommes à calculer sont de la forme

$$A_{j,k} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} a_{nm} \exp\left(\frac{2i\Pi nj}{N} + \frac{2i\Pi mk}{M}\right) \quad \begin{matrix} j=0\ldots N-1 \\ k=0\ldots M-1 \end{matrix}$$

Les sous-programmes FFT calculent

$$X_p = \sum_{\ell=0}^{p-1} x_\ell \exp \frac{2i\Pi \ell q}{p} \qquad q = 0 \ldots p-1$$

Un tableau contient en entrée les données $x_0 \ldots x_{p-1}$ et en sortie $X_0 \ldots X_{p-1}$.

Le calcul se fera donc comme suit :

- FFT par colonnes

de n = 0 à N-1 faire

FFT(a_{n0}, a_{n1}, $\ldots$ a_{nM-1})

{calcule $B_{n,k} = \sum_{n=0}^{M-1} a_{nm} \exp\left(\frac{2i\Pi mk}{M}\right)$; k = 0 ... M-1}.

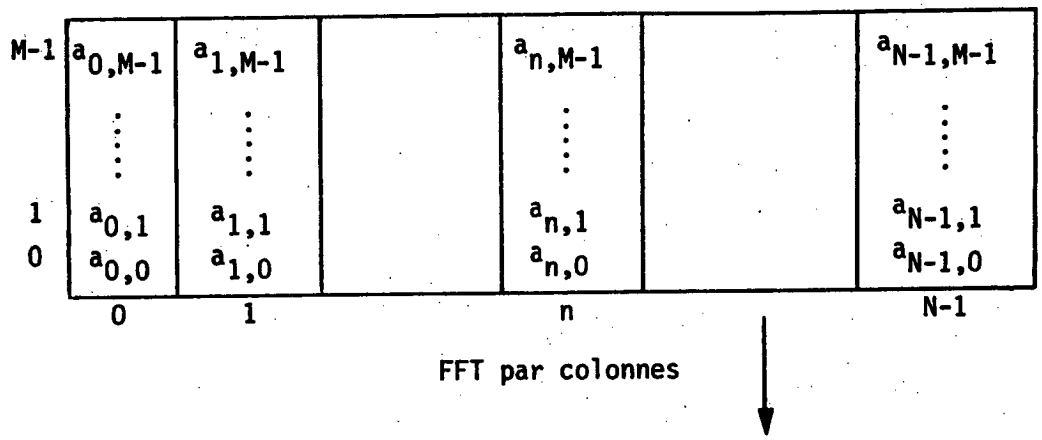
- FFT par lignes

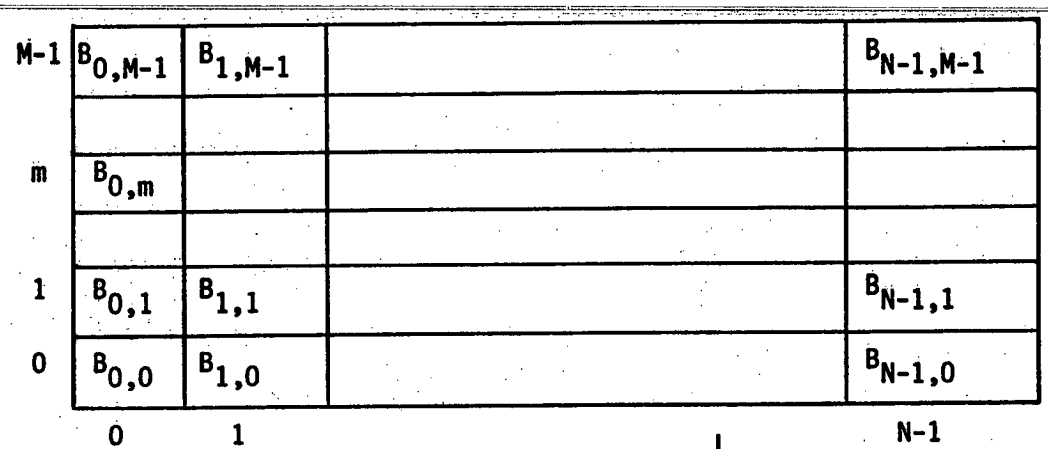
de k = 0 à M-1 faire

FFT($B_{0,k}$, $B_{1,k}$, $\ldots$ $B_{N-1,k}$)

{calcule $A_{j,k} = \sum_{n=0}^{N-1} B_{nk} \exp\left(\frac{2i\Pi nj}{N}\right)$; j = 0 ... N-1}.

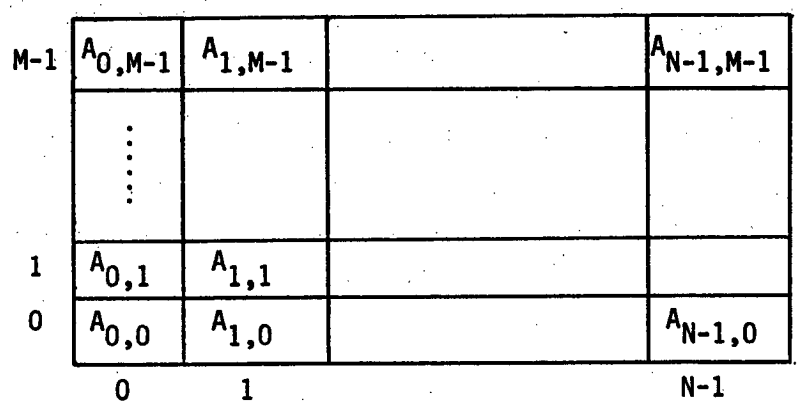
Un tableau N x M comporte données et résultats au long du calcul.





(B)

FFT par lignes



(A)

Comme les a_{nm} sont réels, on a les symétries suivantes :

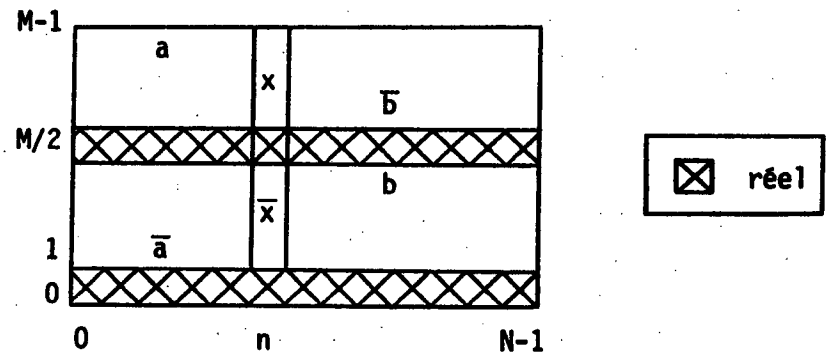
$$\overline{B_{n,k}} = \sum_{m=0}^{M-1} a_{nm} \exp(-\frac{2i\Pi mk}{M})$$

$$= \sum_{n=0}^{M-1} a_{nm} \exp(\frac{2i\Pi m(M-k)}{M})$$

$$= B_{N,M-k} \qquad k = 1, \ldots, M-1$$

$$B_{n,0} = \overline{B_{n,0}}$$

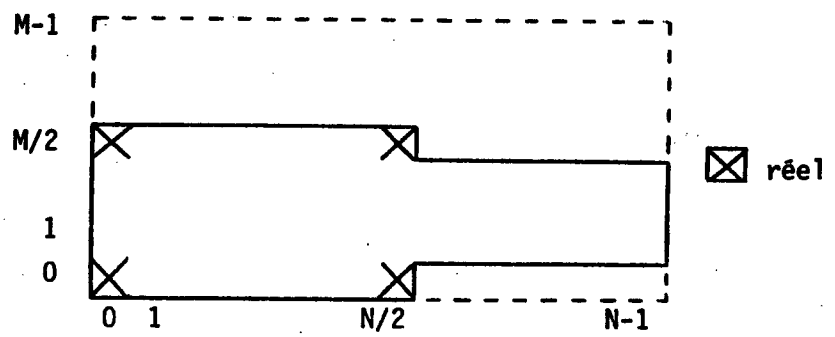
donc le tableau (B) présente les symétries



Il suffit donc de calculer $B_{n,k}$ $n = 0 \ldots N-1$ pour $k = 0, \ldots \frac{M}{2}$

Certaines subroutines FFT prennent en compte ces symétries (FFT réel → complexe).

Le nombre de FFT par lignes se réduit alors à M/2 + 1 au lieu de M. D'autre part, les lignes 0 et M/2 sont réelles et on peut aussi leur appliquer une FFT réel → complexe. Finalement, les valeurs à conserver sont donc



c'est-à-dire $(\frac{M}{2} - 1)\, N + 2(N/2 - 1) = \frac{NM}{2} - 2$ valeurs complexes et 4 valeurs

réelles, soit NM valeurs réelles , ce qui est le nombre minimum puisqu'on part de NM valeurs sur la grille, réelles et que F est un isomorphisme.

II - PROGRAMMATION DE LA FFT EN DIMENSION 2 SUR LE ST100

1 - LES MACROS UTILISEES POUR FAIRE UNE FFT

La FFT sur le ST100 se fait en 4 étapes :

- SMP - Transfert des tables de sinus et cosinus dans le cache ;
- SMP - Transfert des données main → cache ;
- ACP - Calcul dans le cache ;
- SMP - Transfert des résultats main → cache.

REMARQUES

* Un transfert de tables de sinus et cosinus suffit pour une série de FFT de même longueur (réelles ou complexes, c'est le nombre de points qui compte).

* Il existe deux MACROS ACP réalisant des FFT :

- FFTN - les données sont en ordre normal et les résultats en ordre bit-reversed ;
- FFTB - les données sont en ordre bit-reversed et les résultats en ordre normal.

Mais il existe des MACROS SMP qui peuvent réordonner les tableaux lors des transferts main → cache.

* seule FFTN peut exécuter une FFT réel → complexe.
  seule FFTB peut exécuter une FFT complexe → réel.

* Organisation des données ou résultats dans le cache

- FFT complexe ⟷ complexe.

Les N points complexes a(0),a(1),...a(N-1) sont répartis en 2 tableaux :

1) Re(a(0)) Re(a(1)) ... Re(a(N-1))
2) Im(a(0)) Im(a(1)) ... Im(a(N-1))

- FFT réel ⟷ complexe.

REELS :

Les N points réels a(0),a(1),...a(N-1) sont répartis en 2 tableaux :

1) a(0) a(2) ... a(N-2)
2) a(1) a(3) ... a(N-1)

COMPLEXES :

En tenant compte de la symétrie $a(N-k)=\overline{a(k)}$, on ne conserve que (N/2+1) points complexes, et comme a(0) et a(N/2+1) sont réels, on répartit les valeurs dans 2 tableaux de taille N/2 comme suit :

1) a(0)=Re(a(0)) Re(a(1)) ... Re(a(N/2))
2) a(N/2+1)=Re(a(N/2+1) Im(a(1)) ... Im(a(N/2))

Remarque : il existe des MACROS SMP qui décomposent ("unpack") les tableaux précédents en 2 tableaux de taille N/2+1 pour donner :

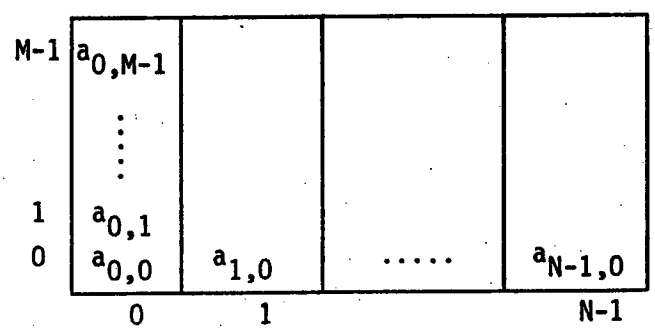
1) a(0) Re(a(1)) ... Re(a(N/2)) a(N/2+1)
2) 0 Im(a(1)) ... Im(a(N/2)) 0

ou qui, inversement, les compressent ("pack").

2 - STOCKAGE DES DONNEES ET RESULTATS

On conserve en mémoire principale un seul tableau N*M de réels. D'après la disposition donnée ci-dessus, on obtient les tableaux suivants pour la FFT-2D directe :

TABLEAU INITIAL



Après les FFT par colonnes (réel → complexe) :

TABLEAU INTERMEDIAIRE

M-1	$\mathrm{Im}\ B_{0,\frac{M}{2}-1}$	$\mathrm{Im}\ B_{1,\frac{M}{2}-1}$		$\mathrm{Im}\ B_{N-1,\frac{M}{2}-1}$
M-2	$\mathrm{Re}\ B_{0,\frac{M}{2}-1}$	$\mathrm{Re}\ B_{1,\frac{M}{2}-1}$		$\mathrm{Re}\ B_{N-1,\frac{M}{2}-1}$
⋮	⋮			⋮
3	$\mathrm{Im}\ B_{0,1}$	$\mathrm{Im}\ B_{1,1}$		$\mathrm{Im}\ B_{N-1,1}$
2	$\mathrm{Re}\ B_{0,1}$	$\mathrm{Re}\ B_{1,1}$		$\mathrm{Re}\ B_{N-1,1}$
1	$B_{0,\frac{M}{2}}$	$B_{1,\frac{M}{2}}$		$B_{N-1,\frac{M}{2}}$
0	$B_{0,0}$	$B_{1,0}$		$B_{N-1,0}$
	0	1		N-1

Les FFT par lignes sont toutes complexe → complexe, sauf pour les lignes 0 et 1 qui sont réel → complexe. On obtient finalement :

TABLEAU FINAL

M-1	$\mathrm{Im}\ A_{0,\frac{M}{2}-1}$	$\mathrm{Im}\ A_{1,\frac{M}{2}-1}$				$\mathrm{Im}\ A_{N-2,\frac{M}{2}-1}$	$\mathrm{Im}\ A_{N-1,\frac{M}{2}-1}$
M-2	$\mathrm{Re}\ A_{0,\frac{M}{2}-1}$	$\mathrm{Re}\ A_{1,\frac{M}{2}-1}$				$\mathrm{Re}\ A_{N-2,\frac{M}{2}-1}$	$\mathrm{Re}\ A_{N-1,\frac{M}{2}-1}$
	⋮	⋮				⋮	⋮
	$\mathrm{Im}\ A_{0,1}$	$\mathrm{Im}\ A_{1,1}$	$\mathrm{Im}\ A_{2,1}$			$\mathrm{Im}\ A_{N-2,1}$	$\mathrm{Im}\ A_{N-1,1}$
2	$\mathrm{Re}\ A_{0,1}$	$\mathrm{Re}\ A_{1,1}$	$\mathrm{Re}\ A_{2,1}$			$\mathrm{Re}\ A_{N-2,1}$	$\mathrm{Re}\ A_{N-1,1}$
1	$A_{0,\frac{M}{2}}$	$A_{\frac{N}{2},\frac{M}{2}}$	$\mathrm{Re}\ A_{1,\frac{M}{2}}$	$\mathrm{Im}\ A_{1,\frac{M}{2}}$		$\mathrm{Re}\ A_{\frac{N}{2}-1,\frac{M}{2}}$	$\mathrm{Im}\ A_{\frac{N}{2}-1,\frac{M}{2}}$
0	$A_{0,0}$	$A_{\frac{N}{2},0}$	$\mathrm{Re}\ A_{1,0}$	$\mathrm{Im}\ A_{1,0}$		$\mathrm{Re}\ A_{\frac{N}{2}-1,0}$	$\mathrm{Im}\ A_{\frac{N}{2}-1,0}$
	0	1	2			N-2	N-1

Ceci nous donne la structure du tableau $\wedge$ par lequel on multiplie le tableau précédent, avant d'exécuter la FFT-2D inverse.

3 - ORGANISATION DU SOUS-PROGRAMME DE FFT-2D : DOUBLE BUFFER

Dans un premier temps, on a écrit un process en double buffer pour exécuter les séries de FFT (par colonnes puis par lignes).

Les bancs de cache A et B contiennent les données et résultats des FFT, le banc C contient les tables de sinus et cosinus.

Organisation du process :

FFT par colonnes :

* Données de la 1ère colonne main → cache	BOTTOM
* FFT sur la 1ère colonne	BOTTOM
Données de la 2ème colonne main → cache	TOP
* Résultats de la (j-1)-ème colonne cache → main	BOTTOM
Données de la (j+1)-ème colonne main → cache	BOTTOM
FFT sur la j-ème colonne	TOP
* FFT sur la (j+1)-ème colonne	BOTTOM
Résultats de la j-ème colonne cache → main	TOP
Données de la (j+2)-ème colonne main → cache	TOP
* Résultats de la (N-1)-ème colonne cache → main	BOTTOM
FFT sur la N-ème colonne	TOP
* Résultats de la N-ème colonne cache → main	TOP

FFT par lignes :

Même chose en inversant N et M et en exécutant les 2 FFT réelles en dehors de la boucle (sur la 1ère et la (N/2+1)-ème ligne).

Comme le temps de calcul des FFT (tâche ACP de coût = O(MLogM) pour des vecteurs de taille M) est très long par rapport au temps de transfert (tâche SMP de coût = O(2M) pour des vecteurs de taille M), on recouvre presque tout le temps SMP par le temps ACP :

Nombre d'opérations pour les FFT par colonnes :

- sans double buffer : proportionnel à 2NM+NMLogM
- avec double buffer : proportionnel à 2M+NMLogM.

On a donc gagné 2(N-1) opérations sur un total de 2NM+NMLogM opérations soit $2(N-1)/(2N+NLogM) \simeq 2/(LogM+2)$. Pour par exemple M = 64, LogM = 6, on a gagné environ 1/4 du temps de calcul (sans compter l'overhead).

Malheureusement, comme on l'a déjà vu, l'overhead dû à l'appel des MACROS est trop important par rapport au temps d'exécution pour que le gain de temps soit réel. Ainsi, le graphique 5 montre que le temps total n'est que du temps d'overhead :

- on sait que le temps de calcul est environ proportionnel à NM Log NM ;
- le nombre de "call", donc l'overhead, est, lui, proportionnel à N+M/2 (N FFT verticales et M/2 FFT horizontales).

Les graphiques 5 et 6 montrent le temps réel T d'exécution du process en fonction respectivement de N+M/2 et NMLogNM pour

$$128 \leq NM \leq 8192$$
$$16 \leq N \leq 512$$
$$16 \leq M \leq 512$$

On voit que la loi T=f(N+M/2) est bien linéaire tandis que la loi T=g(NMLogNM) est plutôt logarithmique.

4 - FFT-2D EN LOOPING-MACROS

On a réécrit le process en essayant d'éviter au maximum les overhead. Pour cela, on se limite à une taille de tableau des valeurs sur la grille de 8K mots = 8192 mots et on utilise les looping-macros qui travaillent sur tout le tableau en un seul "call" : chaque étape qui suit consiste en un seul appel de MACRO.

Les bancs A et B contiennent les données et résultats. Le banc C contient les tables de sinus et cosinus, et le tableau d'inversion ^.

Le PROCESS s'exécute comme suit :

(i) Transfert des NM données réelles dans le cache, en séparant parties paires et impaires dans les bancs A et B et d'autre part, en réordonnant les paires de nombres (indice 2k, indice 2k+1) en bit-reversed (BRS).

BANC A	BANC B
a(0,BRS(0))	a(0,BRS(0)+1)
a(0,BRS(1))	a(0,BRS(1)+1)
.	.
.	.
a(0,BRS((M-2)/2))	a(0,BRS((M-2)/2)+1)
a(1,BRS(0))	a(1,BRS(0)+1)
a(1,BRS(1))	a(1,BRS(1)+1)
.	.
.	.
a(1,BRS((M-2)/2)	a(1,BRS((M-2)/2)+1)
.	.
.	.
.	.
.	.
a(N-1,BRS(0))	a(N-1,BRS(0)+1)
a(N-1,BRS(1))	a(N-1,BRS(1)+1)
.	.
.	.
a(N-1,BRS(M-2)/2))	a(N-1,BRS((M-2)/2)+1)

(ii) N FFT réel → complexe de taille M (données en ordre bit-reversed, résultats en ordre normal "packed").

BANC A	BANC B
B(0,0)	B(0,M/2)
Re B(0,1)	Im B(0,1)
.	.
.	.
Re B(0,M/2-1)	Im B(0,M/2-1)
B(1,0)	B(1,M/2)
Re B(1,1)	Im B(1,1)
.	.
.	.
Re B(1,M/2-1)	Im B(1,M/2-1)
.	.
.	.
.	.
.	.
B(N-1,0)	B(N-1,M/2)
Re B(N-1,1)	Im B(N-1,1)
.	.
.	.
Re B(N-1,M/2-1)	Im B(N-1,M/2-1)

(iii) "Unpackage" avant de revenir en mémoire principale.

BANC A	BANC B
B(0,0)	0
Re B(0,1)	Im B(0,1)
.	.
.	.
Re B(0,M/2-1)	Im B(0,M/2-1)
B(0,M/2)	0
B(1,0)	0
Re B(1,1)	Im B(1,1)
.	.
.	.
Re B(1,M/2-1)	Im B(1,M/2-1)
B(1,M/2)	0
.	.
.	.
.	.
.	.
B(N-1,0)	0
Re B(N-1,1)	Im B(N-1,1)
.	.
.	.
Re B(N-1,M/2-1)	Im B(N-1,M/2-1)
B(N-1,M/2)	0

(iv) Retour en mémoire principale pour réordonner le tableau avant d'effectuer les FFT par lignes (qui nécessitent un rangement ligne par ligne).

(v) Transfert du tableau dans le cache en ordre bit-reversed, ligne par ligne, parties réelles dans le banc A, parties imaginaires dans le banc B.

BANC A	BANC B
B(BRS(0),0)	0
B(BRS(1),0)	0
.	.
.	.
B(BRS(N-1),0)	0
Re B(BRS(0),1)	Im B(BRS(0),1)
Re B(BRS(1),1)	Im B(BRS(1),1)
.	.
.	.
Re B(BRS(N-1),1)	Im B(BRS(N-1),1)
.	.
.	.
.	.
.	.
Re B(BRS(0),M/2-1)	Im B(BRS(0),M/2-1)
Re B(BRS(1),M/2-1)	Im B(BRS(1),M/2-1)
.	.
.	.
Re B(BRS(N-1),M/2-1)	Im B(BRS(N-1),M/2-1)
B(BRS(0),M/2)	0
B(BRS(1),M/2)	0
.	.
.	.
B(BRS(N-1),M/2)	0

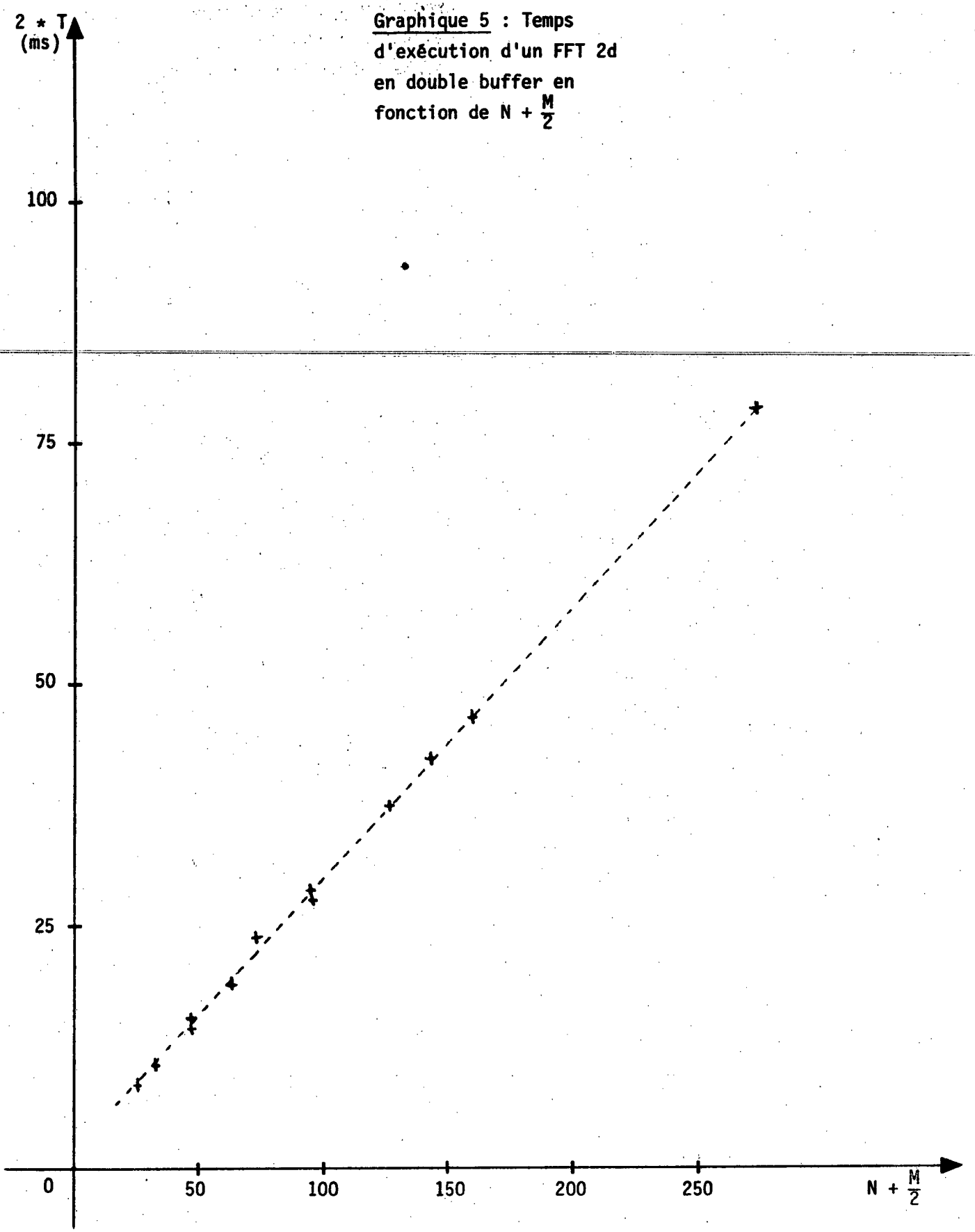
(vi) (M/2+1) FFT complexe ⟶ complexe de taille N (données en ordre bit-reversed, résultats en ordre normal).

BANC A	BANC B
Re A(0,0)	Im A(0,0)
Re A(1,0)	Im A(1,0)
.	.
.	.
Re A(N-1,0)	Im A(N-1,0)
Re A(0,1)	Im A(0,1)
Re A(1,1)	Im A(1,1)
.	.
.	.
Re A(N-1,1)	Im A(N-1,1)
.	.
.	.
.	.
.	.
Re A(0,M/2)	Im A(0,M/2)
Re A(1,M/2)	Im A(1,M/2)
.	.
.	.
Re A(N-1,M/2)	Im A(N-1,M/2)

Les données sont alors prêtes à être multipliées par le tableau Λ, qui reste en permanence dans le banc C ; ensuite, on fera la FFT-2D inverse en effectuant les tâches inverses de (v), (iv), (iii), (ii), (i), dans l'ordre.

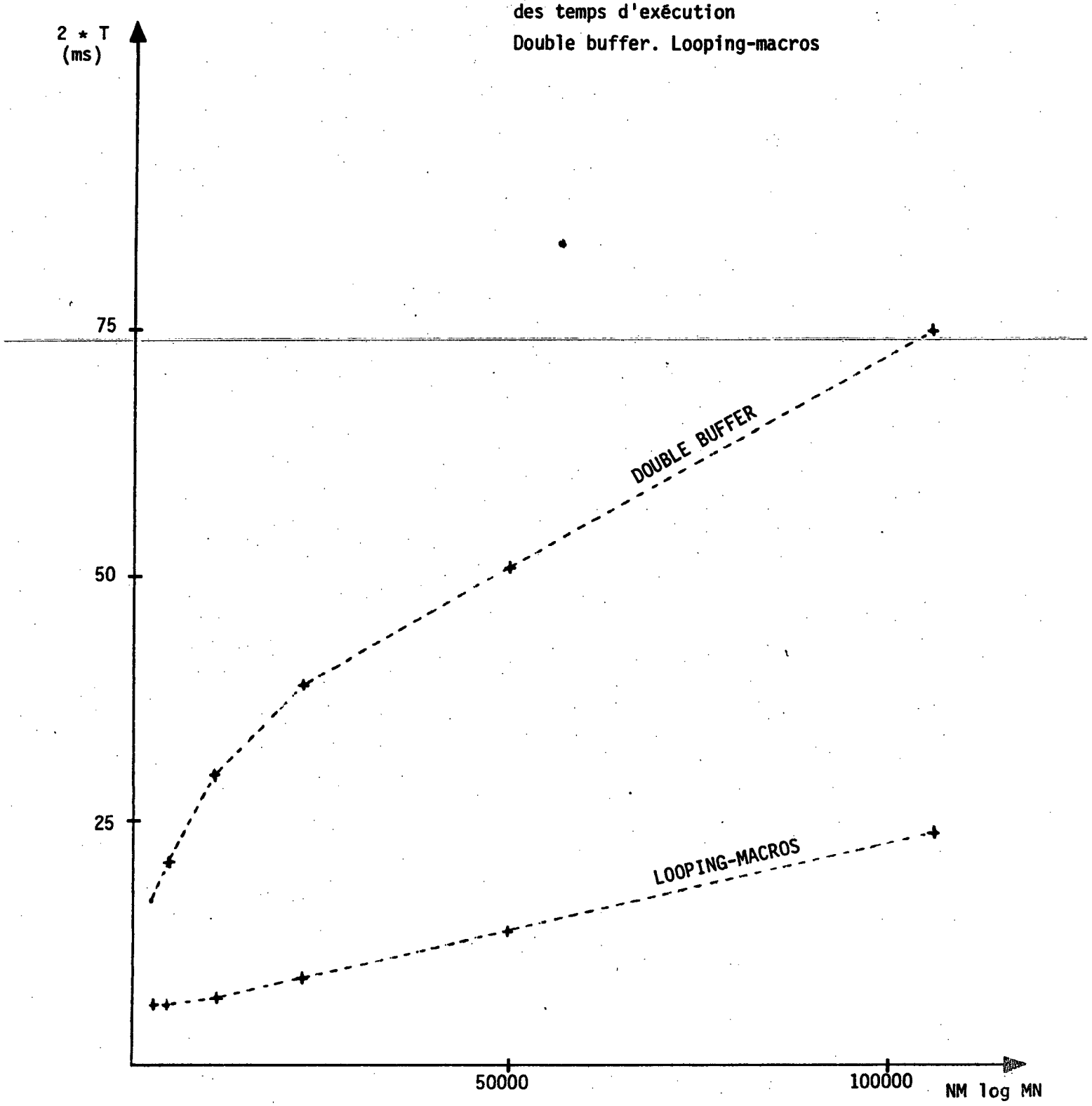
5 - COMPARAISON DES TEMPS DE CALCUL : DOUBLE BUFFER-LOOPING-MACROS

On a représenté sur un même graphique les temps de calcul de la FFT-2D en double buffer et en looping-macros, en fonction de NMLogNM (auquel le temps de calcul ACP est, en principe, proportionnel). On voit que la courbe correspondant aux looping-macros est bien linéaire, donc qu'on ne perd pas de temps en overhead, ce qui n'est pas le cas pour la courbe correspondant au double buffer puisque, comme on l'a vu, dans ce cas, le temps de calcul est plutôt fonction de l'overhead ... Le gain obtenu en utilisant les looping-macros est d'environ 75 %. Remarquons que le (graphique n° 6) obtenu est assez proche des estimations faites au chapitre précédent (graphique n° 1).

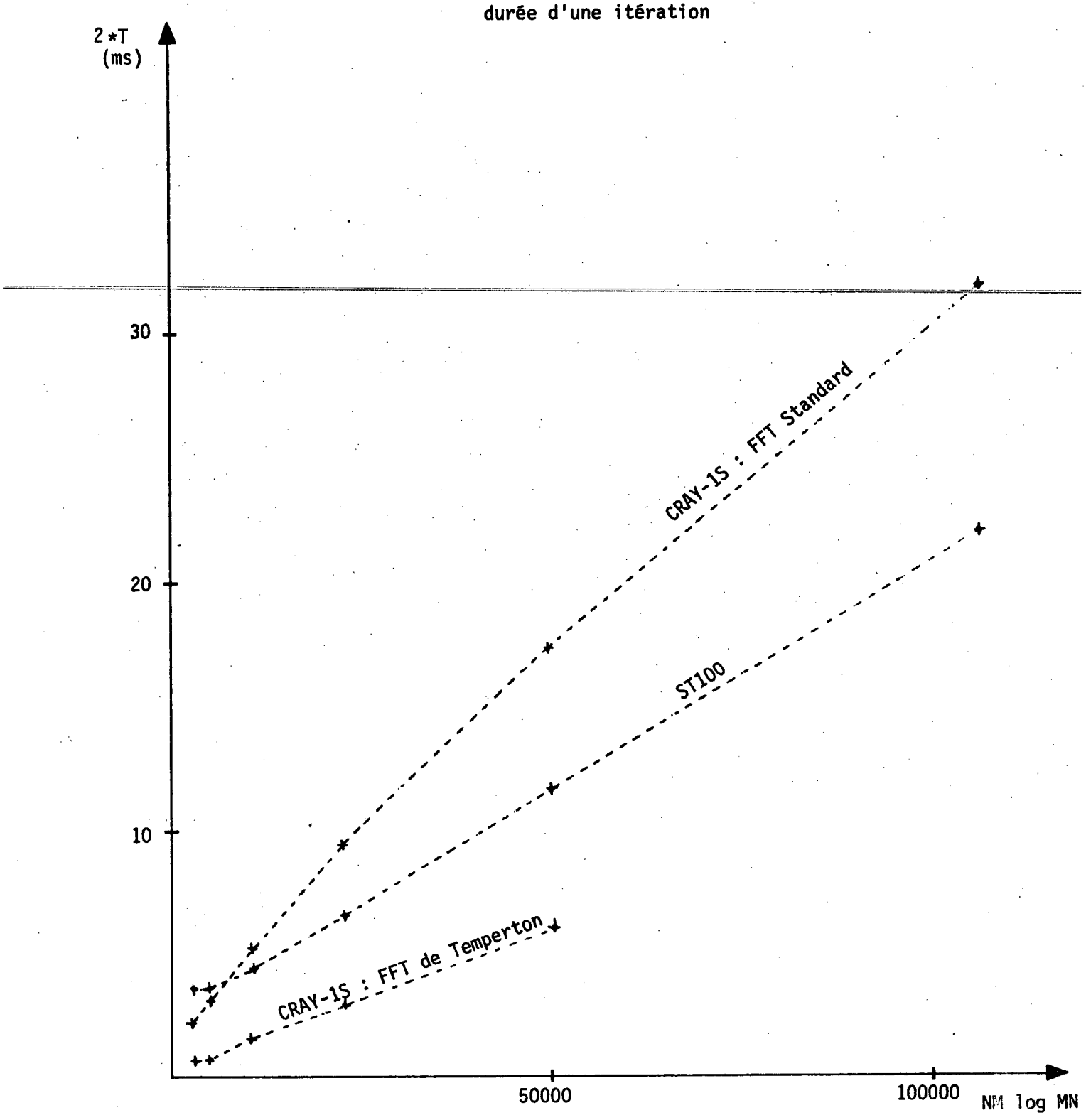


Graphique 5 : Temps d'exécution d'un FFT 2d en double buffer en fonction de $N + \frac{M}{2}$

Graphique 6 : Comparaison
des temps d'exécution
Double buffer. Looping-macros



Graphique 7 : Comparaison des temps de calcul ST100 - CRAY 1 = durée d'une itération



III - COMPARAISON DES TEMPS DE CALCUL CRAY-1S - ST100

On a mesuré les durées d'exécution dans le cadre du programme de résolution de notre équation. Rappelons que dans le passage du temps t au temps t+dt, la majeure partie du temps de calcul est consacrée aux FFT (directe et inverse). C'est pourquoi on compare ici les durées d'exécution d'une itération, qui correspondent pratiquement aux durées d'exécution de 2 FFT-2d. Il est à noter que les calculs sur le ST100 sont de précision 32 bits alors que la simple précision du CRAY-1S est de 64 bits ; la convergence de l'algorithme peut donc être meilleure sur le CRAY-1S et nécessiter moins d'itérations.

Deux séries de mesures ont été réalisées sur le CRAY-1S. La première utilise les subroutines de FFT standard de la librairie CRAY et la seconde des subroutines de FFT optimisée par Temperton /4/. Sur le ST100, on a fait des mesures avec les looping-macros, qui ont donné les meilleures performances. Les résultats sont présentés sur le graphique 7.

On voit que malgré l'overhead assez important encore pour les petites valeurs de N et M, le ST100 est plus rapide que le CRAY-1 avec FFT "standard" dès que N et M augmentent ; il est cependant moins rapide dès que l'on utilise les FFT plus performantes de Temperton. Ceci est assez naturel dans la mesure où ces FFT ont été optimisées en tenant compte des spécificités du CRAY ; de même, les FFT du ST100 sont très performantes, le ST100 offrant une architecture très bien adaptée à ce genre de calculs.

D - UN EXEMPLE DE MACRO ACP : PRODUIT D'UNE MATRICE CREUSE PAR UN VECTEUR

I - INTRODUCTION

Dans les méthodes d'éléments finis, une part non négligeable des calculs consiste à effectuer le produit d'une matrice d'ordre élevé mais creuse, c'est-à-dire avec beaucoup de coefficients nuls, par un vecteur. Il faut donc trouver une structuration de la matrice qui minimise à la fois la place mémoire et la durée d'exécution. Plusieurs facteurs interviennent dans la vitesse, les plus importants étant le nombre d'opérations, les pertes de temps dues au traitement des indices (indirection, ...), et la vectorisation.

1 - STRUCTURES DE DONNEES

On considère ici un mode de stockage de type creux où la matrice est rangée par lignes avec le même nombre de coefficients non nuls (largeur de bande constante). Un tableau de pointeurs permet de repérer les numéros de colonnes. La matrice peut être rectangulaire (c'est par exemple une sous-matrice d'une matrice-blocs). L'intérêt d'imposer une largeur de bande constante est de vectoriser sur le nombre de lignes au lieu du nombre de colonnes, trop petit en général. Un traitement préalable permet éventuellement d'extraire une sous-matrice avec des lignes de même largeur de bande. On utilise les variables suivantes :

N	nombre de lignes
LBD	largeur de bande
A(N,LBD)	matrice des coefficients non nuls rangés par lignes de largeur LBD
JA(N,LBD)	pointeur colonne correspondant

A_{ij} = A(I,K) avec j = JA(I,K) $1 \leq K \leq LBD$

2 - ALGORITHME 1

Il s'agit d'effectuer le calcul mathématique suivant :

$$y(i) = \sum_{k=1}^{lbd} a(i,k) * x(ja(i,k))$$

$$1 \leq i \leq n$$

Le programme comprend deux boucles, la boucle interne sur I étant vectorielle (y a été initialisé à 0.) :

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

3 - ALGORITHME 2

Afin d'éviter des appels mémoire pour Y (une lecture/écriture de Y(I) pour chaque valeur de K), une technique usuelle consiste à dérouler la boucle externe, c'est-à-dire à écrire explicitement les calculs pour plusieurs K. Dans l'exemple ci-dessous, on a déroulé par pas de 4 :

```
      DO  1  K = 1,LBD - 3,4
          DO  2  I = 1,N
              Y(I) = Y(I) + A(I,K) * X(JA(I,K))
     .                    + A(I,K+1) * X(JA(I,K+1))
     .                    + A(I,K+2) * X(JA(I,K+2))
     .                    + A(I,K+3) * X(JA(I,K+3))
2         CONTINUE
1     CONTINUE
C     TRAITER LES VALEURS RESTANTES DE K PAR DES TESTS
C     . . .
```

II - PROGRAMMATION D'UNE MACRO

1 - TABLE DE RESERVATION

Pour écrire et mettre au point des programmes en langage "Macro Assembleur", on utilise des "flowcharts" ou tables de réservation. A chaque colonne de ce tableau correspond un cycle de la machine, qui est l'intervalle de temps minimal entre deux changements d'état du processeur. (Le ST100 possède un cycle de durée 40 ns). Durant chaque cycle, on peut effectuer en parallèle plusieurs opérations élémentaires sur les différentes

unités fonctionnelles, telles que adressage d'un banc du cache, une étape de multiplication flottante, etc ... . A chaque ligne de la table de réservation correspond précisément une opération de durée un temps de cycle.

L'utilisateur remplit une table de réservation en indiquant, pour chaque cycle, quelles instructions élémentaires le programme doit exécuter, en précisant au besoin les opérandes (Figure 1). Il réserve donc les unités fonctionnelles pour une durée déterminée. Il lui reste ensuite à traduire la table de réservation en langage assembleur, c'est-à-dire à écrire une macroinstruction par cycle.

Pour accélérer l'exécution d'un programme assembleur, il faut "compacter" le code, c'est-à-dire effectuer le maximum d'opérations en parallèle à chaque cycle, et réduire ainsi le nombre total de cycles.

unité fonctionnelle \ cycle	1	2	3	...	k
unité 1	x	x	x		x
unité 2	x	x		x	
⋮					
unité n		x		x	x

Figure 1 : exemple de table de réservation

2 - CAS DU PROCESSEUR ARITHMETIQUE DU ST100

La table de réservation d'un processeur est évidemment liée à son architecture. Nous allons décrire ci-après celle de l'ACP (Arithmetic Control Processor). (Figure 2).

L'ACP contient quatre ALU (Arithmetic and Logic Unit) :

- une pour le contrôle des boucles, les tests, ...
- trois pour la génération d'adresses mémoire, une par accès aux bancs du cache.

Les registres CA, CB, CC contiennent les données lues du cache tandis que les registres RA, RB, RC sont destinés à l'écriture de résultats

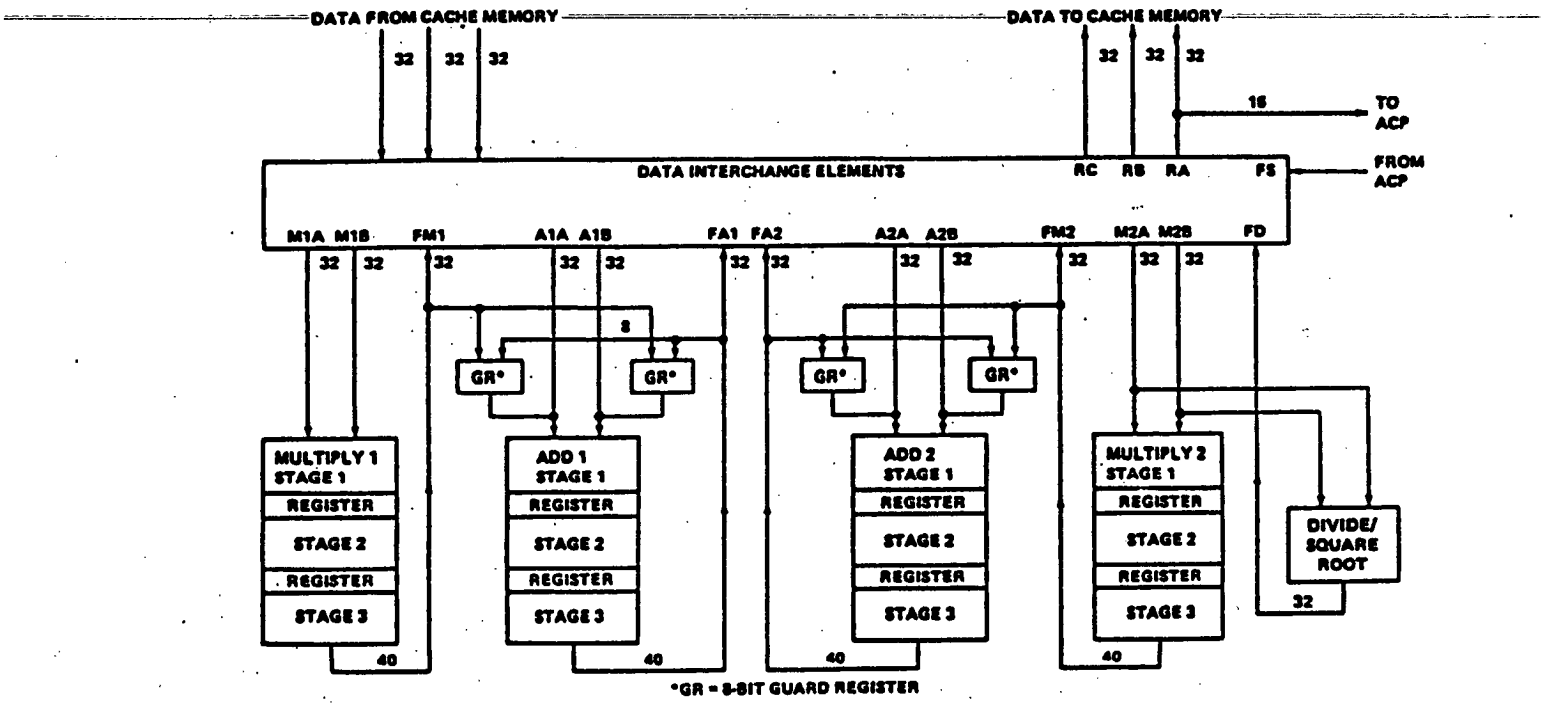


Figure 2 : Arithmetic Computation Unit

dans le cache.

Il existe cinq unités fonctionnelles flottantes :

- deux additionneurs pipelinés à trois étages ;
- deux multiplieurs pipelinés à trois étages ;
- un diviseur séquentiel.

L'architecture de l'ACP est adaptée au calcul vectoriel, avec chaînage des accès mémoire et des opérations flottantes. Considérons par exemple le produit de deux vecteurs :

$$C = A * B \Longleftrightarrow c_i = a_i * b_i \qquad 1 \leq i \leq N.$$

Un premier problème qui se pose est le rangement des données dans les bancs du cache. Dans ce cas très simple, il suffit de ranger chaque vecteur dans un banc. Le problème devient effectif lorsque l'algorithme contient plus de trois données.

La figure 3 est une table de réservation simplifiée où l'on n'a représenté que les unités fonctionnelles effectivement utilisées, à savoir les trois ALU des générateurs d'adresse et l'un des multiplieurs. On constate sur cet exemple que ces unités sont réservées à chaque cycle (cf. cycle 9) sauf bien sûr au début et à la fin du pipeline. On itère sur le cycle 9 en gérant la boucle dans la 1ère ALU. Le code est donc optimisé, et on atteint asymptotiquement la performance de 25 MFlop/s qui correspond à 1 opération par cycle.

unité fonctionnelle \ cycle		1	2	3	4	5	6	7	8	9
cache A	adresse transfert registre CA	A1	A2 A1	A3 A2 A1	A4 A3 A2	A5 A4 A3	A6 A5 A4	A7 A6 A5	A8 A7 A6	A9 A8 A7
cache B	adresse transfert registre CB	B1	B2 B1	B3 B2 B1	B4 B3 B2	B5 B4 B3	B6 B5 B4	B7 B6 B5	B8 B7 B6	B9 B8 B7
multiplieur	registres opération résultat				A1*B1	A2*B2 A1*B1	A3*B3 A2*B2 C1	A4*B4 A3*B3 C2	A5*B5 A4*B4 C3	A6*B6 A5*B5 C4
cache C	adresse transfert cache							C1	C2 C1	C3 C2 C1

Figure 3

La table de réservation de cette application pourtant triviale montre que le nombre de cycles croît très rapidement lorsqu'on écrit les micro-instructions de façon exhaustive. C'est pourquoi on introduit une hiérarchie des unités fonctionnelles.

On regroupe les trois opérations élémentaires relatives au cache A par exemple en une seule méta-opération, la vitesse apparente étant le nombre de mots par cycle lus ou écrits dans le cache A. De même chaque cache et les unités flottantes sont réduits à une entrée dans la méta-table de réservation. On peut ainsi étudier au niveau élémentaire chaque méta-opération et au niveau global leur enchaînement.

Pour l'exemple précédent, chaque méta-cycle correspond en fait à trois cycles de la machine et les indices à trois composantes du vecteur (Figure 4).

unité fonctionnelle \ cycle	1	2	3
CACHE A	A1	A2	A3
CACHE B	B1	B2	B3
MULTIPLIEUR		A1*B1	A2*B2
CACHE C			C1

Figure 4

Cette technique permet, en raisonnant à un niveau moins détaillé, de simplifier la table de réservation et de faciliter l'optimisation. Ceci apparaîtra plus clairement dans l'écriture de l'algorithme décrit au § I .

3 - RETARDS

Dans l'exemple précédent, les opérations s'enchaînent sans problème. Mais il est parfois nécessaire d'introduire des retards afin de synchroniser les vitesses des différentes unités fonctionnelles. On utilise alors les registres de l'ACP comme tampons pour stocker les opérandes en attendant de les transférer d'une unité à l'autre.

Supposons par exemple qu'un mot lu dans le cache A doive être

chargé dans un des registres d'un additionneur (FA1 par exemple) deux cycles plus tard, mais ne puisse rester dans le registre CA, un autre mot devant être lu. On peut, s'il est libre, charger le mot dans le registre RA et charger un cycle plus tard RA dans le registre FA1 (Figure 5).

registre \ cycle	1	2	3	4	5
CA	X1	Y1	X2	Y2	X3
RA		X1		X2	
FA1			X1		X2

Figure 5

III - MISE EN OEUVRE SUR ST100

1 - ANALYSE DU PROBLEME

Il s'agit d'effectuer le calcul de Y, donné par la formule suivante :

$$Y(I) = \sum_{K=1}^{LBD} A(I,K) * X(JA(I,K)) \qquad 1 \leq I \leq N$$

L'algorithme contient quatre opérandes, d'où un problème de rangement dans les trois bancs du cache.

Calculons la vitesse maximale de cet algorithme : pour effectuer 2 opérations (1 addition et 1 multiplication), il faut au moins :

- lire une valeur de JA
- lire une valeur de A
- lire une valeur de X.

En outre, selon la profondeur de la boucle déroulée en K (cf. algorithme 2), il faut lire et écrire une valeur de Y.

Les générations d'adresse d'un élément de A ou JA ne prennent qu'un cycle : il suffit d'ajouter l'indice courant (contenu dans un regis-

tre) à l'adresse de départ, et d'incrémenter cet indice. Par contre, il faut deux cycles pour générer l'adresse d'un élément de X : il faut d'abord transférer la valeur de l'indice (JA), puis ajouter ce déplacement à l'adresse de départ de X, ces deux opérations devant s'effectuer séquentiellement.

Pour obtenir une vitesse de n opérations par cycle, il faut donc effectuer par cycle n/2 accès mémoire pour lire A, de même pour lire JA, et n accès pour lire X, soit un total de 2n accès-mémoire, sans compter les accès relatifs à Y. Or, le cache contenant trois bancs, on en déduit que n vaut au plus 3/2, soit une vitesse théorique maximale de 37.5 MFlops/s.

Etudions maintenant le problème du stockage des données, d'abord dans le cas où le cache ne contient qu'une copie du vecteur X.

2 - CAS D'UNE SEULE COPIE DE X

Dans ce cas, X étant dans un seul banc, on ne peut effectuer qu'une lecture de X tous les deux cycles, donc la vitesse est limitée à 1 opération par cycle, soit 25 MFlops/s.

On peut alors stocker A et JA dans le même banc, et lire sur deux cycles une valeur de A et de JA. Le troisième banc contient le vecteur Y, ce qui permet de lire puis d'écrire Y durant ces deux cycles : il est inutile dans ce cas de dérouler la boucle. Le nombre d'unités flottantes est ici suffisant pour opérer à la même vitesse que les bancs du cache.

Il est nécessaire d'introduire des retards après la lecture de A et le calcul de Y, comme le montre la table de réservation de la Figure 6 . La boucle contient deux meta-cycles (6 et 7 par exemple). Le code est optimisé puisque les trois bancs du cache sont réservés à chaque cycle : on ne peut pas espérer mieux.

UNITE \ CYCLE	1	2	3	4	5	6	7	8	9
BANC A	J1	A1	J2	A2	J3	A3	J4	A4	
RA			A1		A2		A3		A4
BANC B		X1	X1	X2	X2	X3	X3	X4	X4
*				A1*X1	—	A2*X2	—	A3*X3	—
BANC C				Y1		Y2	(Y1)	Y3	(Y2)
+					A1*X1+Y1	—	A2*X2+Y2	—	A3*X3+Y3
RC						(Y1)	—	(Y2)	—

Figure 6

(Y1) : écriture de Y1

On atteint asymptotiquement la performance de 25 MFlops/s : en effet, on effectue 1 opération (* ou +) par cycle.

3 - CAS OU X EST STOCKE DANS PLUSIEURS BANCS

C'est la lecture de X qui limite la vitesse globale de l'algorithme. Une solution pour remédier à ce problème consiste à dupliquer X dans plusieurs bancs. Il faut alors restructurer le rangement des autres données.

Pour obtenir la vitesse maximale de 37.5 MFlops/s, il faut lire 3 valeurs de X tous les 4 cycles, et donc stocker X sur 3 bancs. Sur les 4 cycles, deux sont consacrés à la lecture de X, et les deux autres permettent de lire les 3 valeurs de A et JA : il faut pour cela stocker A et JA sur 3 bancs (Figure 7).

BANC A	BANC B	BANC C
X	X	X
A(3I)	A(3I+1)	A(3I+2)
JA(3I)	JA(3I+1)	JA(3I+2)

Figure 7 : stockage des données dans les 3 bancs

Mais il faut en outre insérer des cycles pour lire et écrire les éléments du vecteur Y, ce qui diminue la vitesse réelle d'exécution. Il faut également utiliser des registres pour introduire des retards et bien sûr dérouler la boucle sur K suffisamment.

Lorsque le vecteur X n'est stocké que sur deux bancs, le problème devient un peu plus simple. Il faut cette fois stocker le tableau A dans 2 bancs, et les tableaux JA et Y dans un seul banc (Figure 8).

BANC A	BANC B	BANC C
X	X	JA
A(I,1)	A(I,2)	Y
A(I,3)	A(I,4)	

Figure 8 : stockage des données

On obtient naturellement une boucle de longueur 6, comme le montre la table de réservation de la Figure 9. Le registre RB permet de retarder l'utilisation de Y après sa lecture ; c'est le seul tampon nécessaire. La boucle sur K est déroulée sur une longueur 4 selon l'algorithme 2. On exécute donc 8 opérations en 6 cycles, ce qui donne une vitesse de 33.33 MFlops/s.

cycle / unité fonctionnelle	1	2	3	4	...			
Cache A		a11	x12	x12	a13	x14	x14	a21
Cache B		x11	x11	a12	x13	x13	a14	x21
Cache C	j11	j12	Y1	j13	j14		j21	j22
*1				P11			P13	
*2					P12			P14
+1						P11+P12		
+2							Y1+P11+P12	
RB				Y1				

x22	x22	a23	x24	x24	a31	x32	x32
x21	a22	x23	x23	a24	x31	x31	a32
Y2	j23	j24	(Y1)	j31	j32	Y3	j33
	P21			P23			P31
		P22			P24		
P13+P14			P24+P22	•		P23+P24	
	Y1			Y2+P21+P22			Y2
	Y2						Y3

a33	x34	x34	a41	x42	x42	a43	x44
x33	x33	a34	x41	x41	a42	x43	x43
j34	(Y2)	j41	j42	Y4	j43	j44	(Y3)
		P33			P41		
P32			P34			P42	
	P31+P32			P33+P34			P41+P42
		Y3+P34+P32			Y3		
				a41	Y4		

Figure 9 : table de réservation

4 - RESULTATS

On a codé deux macros, l'une où X est stocké dans un banc, l'autre où le cache contient 2 copies de X. On a utilisé les macros SMP de la librairie pour réaliser les transferts des données avec le mode de stockage choisi.

La taille maximale des données dépend du choix du rangement dans les bancs.

Dans le 1er cas, A et JA étant dans un même banc, on doit avoir : $N \times LBD \leq 8192$. On introduit au besoin dans le process une partition de A et JA en paquets de colonne, de façon à réduire LBD dans la macro.

Dans le 2ème cas, on a fixé LBD à 4 dans la macro, la boucle externe sur K se faisant au niveau du process. A et Y sont stockés dans un même banc, donc on doit avoir $5\ N \leq 16384$, soit $N \leq 3276$.

On a introduit une boucle d'itération fictive de longueur 10 000 avant le CALL aux macros, afin de pouvoir négliger les pertes de temps dues au CALL et de ne mesurer que la durée effective d'exécution.

On a mesuré la 1ère macro avec LBD = 1, 4, 8 et N variant respectivement de 1 000 à 8 000, 500 à 2 000, 250 à 1 000. Dans la 2ème macro, LBD vaut 4 et N varie de 500 à 3 000.

Les vitesses obtenues sont indépendantes de N et LBD, ce qui prouve que les temps de start-up des boucles sont petits, et correspondent aux vitesses évaluées précédemment : on obtient respectivement 24.5 MFlops/s et 32.5 MFlops/s.

IV - CONCLUSION

Pour optimiser un code assembleur sur le ST100, il est nécessaire d'une part de répartir correctement les données dans les caches, d'autre part d'introduire des retards. On utilise à cette fin des tables de réservation simplifiées où les macro-cycles correspondent à trois cycles réels de la machine. Dans le cas étudié, il fallait optimiser le parallélisme des appels mémoire, celui des opérations s'en déduisait sans problème. L'exemple précédent montre que cette conversion de code est efficace, elle a en effet permis de passer de 24.5 à 32.5 MFlops/s. En outre, on peut évaluer facilement le gain obtenu à l'aide des tables de réservation.

CONCLUSION

Nous avons présenté ici un calculateur vectoriel qui offre de multiples possibilités de parallélisme et de vectorisation. Grâce à la programmation à plusieurs niveaux, la machine est adaptable efficacement au type d'algorithme étudié : programmation en assembleur pour la vectorisation, en APCL pour la synchronisation des tâches de calcul et de transfert des données. De plus, les outils fournis pour l'implémentation des programmes tels que les tables de réservation, les simulateurs de macros et de process facilitent l'utilisation du ST100.

Nous avons vu que les performances du ST100 sont très bonnes, en particulier qu'elles sont compétitives avec le CRAY-1S au niveau du traitement des FFT. Ce résultat a été acquis en réduisant l'overhead à l'aide de "looping-macros".

L'évolution future du logiciel tient compte de ce problème d'overhead et s'oriente vers la génération automatique de super-macros.

Un langage de haut niveau devrait à long terme faciliter la programmation de cette machine, tout en restant performant. En particulier, il serait souhaitable de gérer automatiquement le parallèlisme entre calculs et transferts des données. En outre, le problème de compaction du code assembleur, qui est en général très difficile, s'avère ici crucial pour exploiter au mieux les possibilités de vectorisation et de parallélisme. Un exemple très simple a montré comment une optimisation du code assembleur fournit une accélération non négligeable.

BIBLIOGRAPHIE

/1/ D. GOTTLIEB, S. ORSZAG,
"Numerical Analysis of Spectral Methods : Theory and Applications",
SIAM, Philadelphia, 1977.

/2/ STAR TECHNOLOGIES,
"ST100 : The 100 Megaflops array-processor",
Hardware and Software References Manuals.

/3/ P. KOGGE,
"The architecture of pipelined computers",
Mc Graw Hill Book Company, New York, 1981.

/4/ C. TEMPERTON,
"Fast Fourier Transforms on Cray-1",
European Center for Median Weather Forecasts Report N° 21, 1979.

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