LE PARALLÉLISME DE DONNÉES EN FORTRAN

Philippe Marquet
marquet@lifl.fr
Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France

DESS Informatique Parallèle

15 octobre 1993
révisé le 18 octobre 1994
révisé le 5 novembre 1995

HISTORIQUE

Table des matières

- Fortran = langage scientifique
  ⇒ choix important pour le parallélisme
- **1954-57** John BACKUS, IBM
  ⋮
  1973 Fortran V → norme **Fortran 77**
  ⋮ Fortran 8x : manipulation de tableaux
  1991 Fortran 90 : nouvelle norme
  1992-93 High Performance Fortran : **HPF**
  Prise en compte des architectures à mémoire distribuée
- Traditionnellement, les compilateurs Fortran sont très optimisés et produisent du code de très bon niveau

BIBLIOGRAPHIE

Bibliographie

- *International Standard 1539:*
  Information Technologie/Programming languages/Fortran
  ISO/IEC, 1991
- *Fortran 90 Explained*
  Michael Metcalf et John Reid
  Oxford Science Publication, 1994
- *Programmer's Guide to Fortran 90*
  Walter Brainerd, Charles Goldbelg, et Jeanne C. Adams
  Mc Graw-Hill, 1994
- *High Performance Fortran Language Specification*
  High Performance Forum, version 1.0, May 1993
- *Digital Parallel Software Environment:*
  HPF and PSE Manual
  Digital Equipment Corporation, January 1995
- *Cours Fortran 90*
  Patrick Corde et Hervé Delouis
  IDRIS, CNRS, 1994

FORTRAN 77

Table des matières

- C'est encore aujourd'hui *LE* langage Fortran
- Rappels :
  - Aspects lexicaux
  - Déclarations
  - Équivalence — Common
  - Fonctions

Aspects lexicaux

Ligne

- Colonnes 1 à 5 étiquette (n° de ligne)
  6 suite
  7 à 77 instruction
- Une instruction par ligne
- Commentaire : C ou * en 1re colonne
- Directive : commentaire, sauf pour certains compilateurs

```
CMPF  <directive du compilateur MP Fortran>
*HPF$ <directive du compilateur HP Fortran>
```

Mots-clés

- Les mots-clés ne sont pas des mots réservés !
- Les espaces blancs ne sont pas significatifs

```
DO 10 I = 1, 10
DO10I = 1
```


Déclaration

- Attention, la déclaration des variables n'est pas obligatoire :

  I, J, K, L, M, N entiers
  Autres réels

```
IMPLICIT NONE
```

- Déclaration de variables :

  <*type*> <*name-list*>

```
INTEGER           LAST, LONG(100), MEM(0:10, 0:10)
REAL              TEMP (0:100)
DOUBLE PRECISION  FINE (50, 10)
COMPLEX           Q1(20), Q2(20)
LOGICAL           FLAG
```

- Déclarations DIMENSION

```
REAL      A, X
DIMENSION A(2:10,5), X(10)
```

- Déclarations PARAMETER – Constantes

```
PARAMETER (LENGTH=10)
PARAMETER (LSQ=LENGTH**2)
```


EQUIVALENCE

- Allocation de plusieurs variables au même endroit
- Était nécessaire dans la préhistoire, avec les mémoires limitées !
- Dangereux
- Exemple

```
EQUIVALENCE (AA, ALPHA)

AA    = 3
ALPHA = 5
```

- Équivalence entre un élément de tableau et une variable

```
REAL A (3), ALPHA, BETA
EQUIVALENCE (ALPHA, A(1)), (BETA, A(2))
```

- Équivalence entre deux tableaux

```
REAL A (3,3), B (9), COL (3)
EQUIVALENCE (COL, A, B)
```

DATA

- Déclaration/initialisation de variables

    DATA <*liste-de-variables*> /<*liste-de-valeurs*>/

- Exemple

```
DATA A, B, C /1, 3, 7/
```

- Répétition d'une valeur

```
DATA I, J, K /3*0/
```

- Initialisation d'un tableau

```
REAL A (5,5)
DATA A /25*1.0/

REAL B (5,5)
DATA B /5*(1., 2., 3., 4., 5.)/
```

- Initialisation DATA est faite une seule fois et non à chaque fois que le code contenant la déclaration DATA est exécuté

Instructions

Instructions de contrôle

- GOTO <*line #*>
  Il existe aussi des GOTO calculés !
- IF (<*logical-expr*>) THEN
      <*instructions*>
  ENDIF
- IF (<*logical-expr*>) THEN
      <*instructions*>
  ELSE
      <*instructions*>
  ENDIF
- IF (<*logical-expr*>) <*instruction*>
- ELSEIF
- Il existe aussi un IF arithmétique !

- Boucle

```
   DO 10 I = 1, 100
     SUM = SUM + A(I)
10 CONTINUE

   DO 20 I = 1, 10, 2
      IMP = IMP + A(I)
20 CONTINUE
```

Entrées/sorties

- READ * , <*liste-de-variables*>
- PRINT * , <*liste-de-valeurs*>

```
PRINT *, 'ANGLE=', THETA, 'LONGUEUR=', A
```

- Il existe une notion de format d'entrée ou de sortie : mot clé FORMAT
- Manipulation de fichiers : OPEN, CLOSE, READ, WRITE, ENDFILE...

Programme

- Un programme Fortran 77 est de la forme

    PROGRAM <*nom*>
        [<*déclarations*>]
        [<*instructions*>]
    END

- Un fichier source Fortran 77 est constitué d'un ensemble de fonctions (FONCTION et SUBROUTINE) et d'un éventuel programme principal
- Les variables déclarées au niveau du programme ne sont, normalement, pas visibles par les fonctions
- La compilation d'un fichier source Fortran se fait fonction par fonction

  Le découpage en fichiers n'importe donc pas au niveau de la déclaration

Procédures

- Deux types de procédures : les fonctions FONCTION et les SUBROUTINE
- Exemple

```
SUBROUTINE SBRT (A, B)
REAL A, B, X
...
END
```

- Les paramètres sont toujours passés par *adresse*

```
FUNCTION FNCT (T, TAB, TABLO, N)
DIMENSION T(10), TAB(N), TABLO(*)
```

- Un appel

```
DIMENSION TT (10,10)
... F (TT)

FUNCTION F (T)
DIMENSION T (100)
```

est légal, mais déconseillé !

- Pas de récursivité (même indirecte)

Fonctions

- La fonction

```
FUNCTION MAXI (A,B)
INTEGER MAXI, A, B
IF A .GT. B THEN
  MAXI = A
ELSE
  MAXI = B
ENDIF
END
```

- Est appelée :

```
INTEGER I, J, K
...
K = MAXI (I, J)
```

- Retour d'une valeur scalaire

- Fonction « en ligne »

```
REAL A, ALPHA
REAL X, Y
RAC (B,C) = (-B+SQRT(B**2-4.*A*C))/(2.*A)
...
A =
X = RAC (4., -6.)
...
X = RAC (ALPHA, SIN(Y))
```


Subroutines

- Définition

```
SUBROUTINE PERMUT (LISTE, TAILLE)
INTEGER TAILLE
INTEGER LISTE (TAILLE)
...
...
END
```

- Appel

```
CALL PERMUT (L, LSIZE)
```

- Il existe aussi des entrées multiples d'appel : mot-clé ENTRY
À éviter

Procédures paramètres

- Déclaration/identification d'un nom comme étant celui d'une fonction ou sous-routine

```
EXTERNAL  NOME
INTRINSIC NOMI
```

- Appel

```
CALL MYSBRT (A, B, NOME)
```


Procédures intrinsèques

- Nombreuses fonctions et sous-routines prédéfinies
- « Généricité » au niveau du typage

```
INTEGER          A, B
REAL             C, D
DOUBLE PRECISION E, F
COMPLEX G
A = ABS (B)
C = ABS (D)
E = ABS (F)
C = ABS (G)
```

(Pour les complexes : $\sqrt{G_r^2 + G_i^2}$)

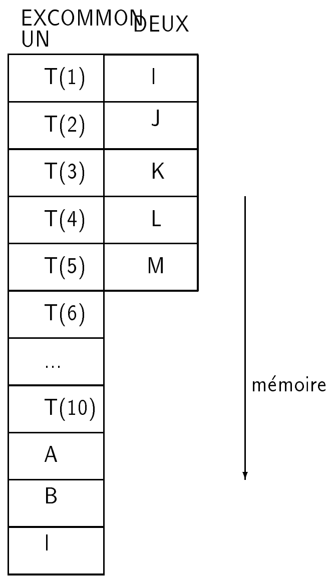
- Nombre de paramètres variable

```
INTEGER A, B, C, D
A = MAX (B, C)
A = MAX (B, C, D)
```


COMMON

- Échange/partage d'information entre deux « modules » (fonction/sous-routine/programme)
- Chacun des modules déclare une *zone mémoire* par une spécification COMMON
- Exemple « common blanc »

```
            PROGRAM EXCOMMON
            INTEGER T (10), I
            REAL    A, B
            COMMON  T, A, B, I
            ...
            END
SUBROUTINE UN          SUBROUTINE DEUX
INTEGER T(10), I       INTEGER I, J, K, L, M
REAL    A, B           COMMON  I, J, K, L, M
COMMON  T, A, B, I     ...
...                    ...
END                    END
```

- Plusieurs COMMON ⇒ il faut les nommer

```
COMMON /NOM1/ A, B, C
COMMON /NOM2/ G, F, R
```

Une variable ne peut apparaître que dans une seule zone

- Attention à la cohérence lors de l'utilisation des COMMON
- Les contraintes de spécification d'EQUIVALENCE ne doivent pas être contradictoires avec une déclaration COMMON
- Initialisation de COMMON par BLOCK DATA, équivalent à DATA pour les variables simples

SAVE

- Les variables locales d'une procédure perdent leur valeur entre deux appels
- Les variables peuvent être spécifiées comme rémanentes et ainsi garder leur valeur par une spécification SAVE

```
SUBROUTINE ANY (A,B)
REAL A(*), B(*)
SAVE KOUNT
DATA KOUNT /0/
KOUNT = KOUNT+1
...
END
```


- SAVE utilisé pour des COMMON non partagés par le programme principal (ou l'ancêtre commun)

```
            SUBROUTINE A
            ...
            CALL B
            ...
            CALL C
            ...
            END

SUBROUTINE B             SUBROUTINE C
SAVE INDEX               SAVE INDEX
COMMON /INDEX/ I, J, K   COMMON /INDEX/ I, J, K
...                      ...
I = 3                    ...
...                      ...
END                      END
```


Conclusion : attention !

- Attention aux erreurs syntaxiques
- Attention aux déclarations implicites des variables
- Attention aux EQUIVALENCE
- Attention aux COMMON
- Attention au passage de paramètres par adresse

FORTRAN 90

Table des matières

Fortran 90

- C'est la nouvelle norme Fortran, Fortran 8x
- Manipulation explicite de tableaux
- Sur-ensemble de Fortran 77 :
  - Tout programme Fortran 77 correct est un programme Fortran 90 correct
  - Constructions identifiées comme « obsolètes »
    - ◇ Boucle `DO` réelles
    - ◇ `IF` arithmétique
    - ◇ `GOTO` assignée

Aspects syntaxiques

- Deux formats de ligne
  1. forme fixe Fortran 77
  2. forme libre, pas de zone réservée
     séparateur d'instructions ;
- Commentaires : `! fin de ligne commentaire`
- Caractère de continuation `&`
- Opérateurs

`<`	`<=`	`==`	`/=`	`>=`	`>`
`.LT.`	`.LE.`	`.EQ.`	`.NE.`	`.GE.`	`.GT.`

Types

- Types prédéfinis
- Précision des variables de type INTEGER et REAL peut être précisée par le programmeur

```
INTEGER (KIND=2) :: I
```

→ 2 bytes

- Types « dérivés » — Ce sont des structures

```
TYPE TRIPLET
  INTEGER A, B, C
END TYPE TRIPLET
TYPE TRIPLET :: TT
TT % A = 3
```

- Type pointeur : POINTER

Une variable pouvant être référencée par un pointeur doit être déclarée TARGET

Déclaration

- Fortran 77

```
REAL      T, U
DIMENSION T(10), U (10)
```

- Fortran 90

```
REAL, DIMENSION (10) :: T, U
```


Affectation de tableaux

- Fortran 77

```
    REAL      T, U
    DIMENSION T(10), U (10)
    DO 10 I = 1, 10
      T (I) = U (I) + 5
10  CONTINUE
```

- Fortran 90

```
    REAL, DIMENSION (10) :: T, U
    T = U + 5
```


Allocation des tableaux

- Fortran 77 : allocation de tous les objets est statique
- Fortran 90 : allocations statique, automatique, et dynamique des tableaux

Tableaux statiques

- Idem Fortran 77
- Allocation déterminée à la compilation
- Taille connue dès la compilation

Tableaux automatiques

- Alloués automatiquement à l'entrée d'une fonction
- Désalloués automatiquement à la sortie de la fonction
- Par ex. la taille est un paramètre de la fonction

```
SUBROUTINE SBRT (T)
  REAL, DIMENSION (:) :: T
  REAL, DIMENSION (SIZE (T)) :: TEMPO
  ...
END
```

Tableaux dynamiques

- Explicitement alloués et désalloués par le programmeur
- Instructions ALLOCATE et DESALLOCATE
- Doit être spécifié ALLOCATABLE lors de sa déclaration
- Taille non précisée lors de la déclaration, seul le nombre de dimension est précisé

```
INTEGER N
REAL, DIMENSION (:,:), ALLOCATABLE :: A
...
READ *, N
ALLOCATE (A(N,N*N))
...
DESALLOCATE(A)
```

- Instruction ALLOCATED permet de savoir si un tableau dynamique est alloué ou non

Littéraux

- Tableau mono-dimensionnel : liste de valeurs immédiates entre (/ et /)

```
(/ 1, 2, 3, 5, 10 /)
```

- Liste de valeurs consécutives ou régulièrement espacées

```
(/ (I, I= 1, 5) /)                (/ 1, 2, 3, 4, 5 /)
(/ (I, I= 1, 10, 2) /)            (/ 1, 3, 5, 7, 9 /)
```

- Combinaison des deux constructions

```
(/ 1, 4, (I, I=1,3) /)            (/ 1, 4, 1, 2, 3 /)
(/ ((I, I=1, 3), J=1, 2) /)       (/ 1, 2, 3, 1, 2, 3 /)
```

- Multi-dimension par fonction intrinsèque RESHAPE

```
RESHAPE (SHAPE  = (/ 2, 3 /),
         SOURCE = (/ 1, 2, 3, 4, 5, 6 /))
```

désigne la matrice

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

- Non limitée aux entiers, s'appliquent à tous les types Fortran

Description

- Une opération de description permet d'obtenir
  - un sous-vecteur/sous-tableau

  à partir
  - d'un vecteur/tableau
- Le résultat d'une opération de description peut être utilisé en partie droite (valeur) ou en partie gauche (accès) d'une affectation
- Deux descriptions en Fortran 90
  1. Description par triplet
  2. Description par un vecteur
- Description indépendante pour chaque dimension d'un tableau

Description par triplet

- Sélectionne des éléments
  - contigus, ou
  - régulièrement espacés

  dans une dimension
- Syntaxe générale :

  [<*lower*>] : [<*upper*>] [: <*step*>]
- Exemple, les éléments de 1 à N de la ligne I

```
A (I, 1:N)
```

- Exemple, les éléments impairs de 1 à M de la colonne J

```
A (1:M:2, J)
```

- Valeurs par défaut de <*lower*> et <*upper*> : indices limites de variation (définis lors de la déclaration ou de l'allocation)
  Valeur par défaut du pas : 1

Description par un vecteur d'index

- Réalise les opérations de « gather/scatter »
- Par exemple :

```
V ( (/1, 7, 3, 2/) )
```

est composé, dans l'ordre, des éléments

```
V(1)   V(7)   V(3)   V(2)
```

- Pour deux vecteurs U et V

```
U = (/1, 3, 2/)
V = (/2, 1, 1, 3/)
```

```
Z(3,V) -->  Z(3,2)   Z(3,1)   Z(3,1)   Z(3,3)
Z(U,V) -->  Z(1,2)   Z(1,1)   Z(1,1)   Z(1,3)
            Z(3,2)   Z(3,1)   Z(3,1)   Z(3,3)
            Z(2,2)   Z(2,1)   Z(2,1)   Z(2,3)
```

- Opération illégale en écriture si doublon(s)

```
! V( (/1,7,3,7/) ) = (/1, 2, 3, 4/)
```

V(7) vaut 2 ou 4 ?

Pointeurs comme alias

- Utilisation particulière des pointeurs proposés par Fortran 90
- La déclaration d'un pointeur sur un tableau définit le nombre de dimensions de ce tableau

```
REAL, DIMENSION (:,:), POINTER :: PTR
```

- Suite à la déclaration

```
REAL, DIMENSION (3,5), TARGET :: TAB
```

on peut associer le pointeur au tableau :

```
PTR => TAB
```

et utiliser PTR comme un alias pour TAB

- Associer une description par triplet au pointeur

```
PTR => TAB(L1:U1:S1, L2:U2)
```

évaluation de L1, U1, S1, L2, U2 lors de l'association

PTR référence les éléments de TAB sélectionnés par la description

Constructeur WHERE

- Affectation masquée
- Bloc WHERE

```
WHERE (<logical-expr> )
    <assignment-stmt-list>
[ ELSEWHERE
    <assignment-stmt-list> ]
END WHERE
```

- Uniquement affectations de tableaux de même forme que l'expression logique
- La partie droite n'est évaluée que pour les éléments correspondants à une valeur VRAI de l'expression

  Les éléments sont ensuite rangés dans les éléments correspondants de la partie gauche

```
WHERE (PRESURE <= 1.0)
  PRESURE = PRESURE + 1.0
  TEMP = TEMP - 5
ELSEWHERE
  RAINING = .TRUE.
END WHERE
```


- Fonction « élémentaire » (elemental functions)
  - Ce sont des fonctions prédéfinies de Fortran
  - Application d'une opération scalaire élément par élément
  - Appel d'une telle fonction dans un WHERE
    $\Rightarrow$ les paramètres sont masqués

```
WHERE (A>0)
  B = LOG (A)
END WHERE
```

Procédures

- Extensions des fonctions et des programmes Fortran 77
- Retour de valeur tableau
- Paramètre tableau « assumed-shape »
- Attribut spécifiant les caractéristiques des paramètres formels
- Extension des fonctions intrinsèques aux paramètres tableaux
- Nouvelles fonctions intrinsèques de manipulation de tableaux

Paramètres « assumed-shape »

- Un paramètre formel d'une procédure est dit « assumed-shape » :
  - les tailles de ses dimensions ne sont pas connues lors de la définition de la procédure
  - ces tailles seront celles du paramètre effectif lors de l'appel de la fonction

```
SUBROUTINE FOO (TAB)
   REAL, DIMENSION (:,:) :: TAB
   ...
END SUBROUTINE

REAL, DIMENSION (10, 20)  :: T1
REAL, DIMENSION (100, 30) :: T2
...
CALL FOO (T1)
CALL FOO (T2)
```


- Seule la taille est passée avec le paramètre effectif, les bornes inférieures et supérieures ne sont pas connues
- Sauf spécification, la borne inférieure est 1, la borne supérieure est la taille dans la dimension

```
REAL, DIMENSION (0:10) :: A

SUBROUTINE FOO (ASA)
   REAL, DIMENSION (:) :: ASA
```

  $\Rightarrow$ correspondance $\mathtt{A}(i-1) \equiv \mathtt{ASA}(i)$
- Cette caractéristique « assumed-shape » des tableaux oblige les compilateurs à passer
  - l'adresse du tableau
  - la taille de celui-ci dans chaque dimension

  pour tout paramètre tableau

Retour d'une valeur tableau

- Une fonction peut retourner un tableau
- Nombre de dimensions de ce tableau fixé par la définition de la fonction
- Taille du tableau
  - fixe
  - dépend des paramètres effectifs

```
FUNCTION FOO (A)
   REAL, DIMENSION (100) :: A, FOO
   ...
END FUNCTION

FUNCTION MULTMAT (A,B)
   REAL, DIMENSION (:,:) :: A
   REAL, DIMENSION (SIZE (A,2)) :: B
   REAL, DIMENSION (SIZE (A,1)) :: MULTMAT
   ...
   MULTMAT = ...
END FUNCTION
```

Attributs des paramètres

- Pour éviter d'éventuelles copies entre procédure appelante et procédure appelée, des attributs peuvent être associés aux paramètres tableaux :
  - INTENT (IN)
  - INTENT (OUT)
  - INTENT (INOUT)

```
SUBROUTINE FOO (T)
   REAL, DIMENSION (:), INTENT (IN) :: T
   ...
END SUBROUTINE
```


Paramètres tableaux

- Les tableaux sont passés comme paramètre aux procédures
- Comme de règle en Fortran, le passage se fait par adresse
- Les paramètres effectifs sont généralement évalués et une copie de la valeur est passée à la fonction (pas de copie au retour)
- Mais, les paramètres qui sont des variables tableaux éventuellement décrites par
  - triplet, ou scalaire

  sont directement passés à la procédure
  ⇒ partage appelé/appelante

- Exemple : suite à la définition

```
SUBROUTINE ADD1 (TAB)
   REAL, DIMENSION (:) :: TAB
   TAB = TAB +1
END SUBROUTINE

REAL, DIMENSION (10) :: V
```

  - CALL ADD1 (V)
    Incrémente tous les éléments de V
  - CALL ADD1 (V (1:5))
    Incrémente les éléments V(1) à V(5)
  - CALL ADD1 (V ((/6,7,8,9,10 /)))
    Ne modifie aucune des valeurs de V

Procédures intrinsèques

- Extension des procédures intrinsèques de Fortran 77
  → « procédures élémentaires »

  Fonctionnellement, un appel de fonction élémentaire avec des arguments tableaux est appliqué à tous les éléments

  Par exemple :

  MIN le minimum de ses arguments

  SIN le sinus de son argument
- Fonctions « inquiring »

  Retourne des informations sur ses arguments, par exemple

  ASSOCIATE retourne vrai ssi un pointeur et une cible passés en paramètre sont associés (le pointeur référence la cible)
- Fonctions de manipulation de vecteurs et de matrices

  DOTPRODUCT somme du produit élément à élément des paramètres

  MATMUL multiplication de matrices

- Fonctions de réduction

  `SUM` somme des éléments d'un tableau

  `MINLOC` indice du plus petit élément d'un tableau

  - Argument `MASK` : vecteur de `LOGICAL` sélectionnant les éléments sur lesquels appliquer la fonction

    `SUM (A, MASK= A>0)`

  - Argument `DIM` : numéro de la dimension sur laquelle réaliser la réduction

- Fonctions de transformation

```
MERGE (TRUESOURCE, FALSESOURCE, MASK)
PACK (ARRAY, MASK)
UNPACK (ARRAY, MASK, DEFAULTVALUE)
RESHAPE (...)
```

⇒ + de 100 fonctions intrinsèques

Caractéristiques d'un tableau

Ces fonctions retournent des informations à propos des tableaux passés en paramètre

- `MAXLOC` Indice(s) de l'élément maximal d'un tableau
- `MINLOC` Indice(s) de l'élément minimal d'un tableau
- `LBOUND` Borne(s) inférieure d'un tableau
- `UBOUND` Borne(s) supérieure d'un tableau
- `SHAPE` Nombre de dimension et taille dans chacune d'elles d'un tableau
- `SIZE` Taille d'une dimension d'un tableau

Réduction

Ces fonctions retournent un résulat scalaire qui est combinaison des valeurs des éléments d'un tableau
Elles peuvent aussi retourner un tableau qui contient les résulats de cette combinaison par exemple pour chacunes des lignes d'un tableau

- `ALL` Vrai ssi tous les éléments sont `.true.`
- `ANY` Faux ssi tous les éléments sont `.false.`
- `COUNT` Nombre d'élémenst `.true.`
- `MAXVAL` Plus grande valeur d'un tableau
- `MINVAL` Plus petite valeur d'un tableau
- `PRODUCT` Produit des valeurs d'un tableau
- `SUM` Somme des valeurs d'un tableau

Multiplication

Deux fonctions de multiplication

- `DOT_PRODUCT (v1,v2)` Produit scalaire de deux vecteurs

`v1` de type entier/réel	⇒	`sum(v1*v2)`
`v1` de type complexe	⇒	`sum(conjg(v1)*v2)`
`v1` et `v2` de type logique	⇒	any (v1 .and. v2)

- `c = MATMUL (a,b)` Produit matricel de deux matrices

si `shape(a)` $= (n,p)$ et `shape(b)` $= (p,q)$	⇒	`shape(c)` $= (n,q)$ $c_{i,j}$ = `sum(a(i,:)*b(:,j))`
si `shape(a)` $= (p)$ et `shape(b)` $= (p,q)$	⇒	`shape(c)` $= (q)$ c_j = `sum(a*b(:,j))`
si `shape(a)` $= (n,p)$ et `shape(b)` $= (p)$	⇒	`shape(c)` $= (n)$ c_i = `sum(a(i,:)*b)`
Si `a` et `b` de type logique	⇒	remplacer `sum` par `any` et `*` par `.and.` dans les formules

Construction

- RESHAPE construction d'un tableau de "shape" donnée
  - Nombre de dimension
  - Taille de chacune des dimensions

  à partir des valeurs d'un autre tableau

Décalage

- `CSHIFT (array, dim [, mask])`
  Décalage **torique** sur les éléments d'une dimension donnée
  Par défaut, `dim=1`
- `EOSHIFT (array, shift [,boundary] [,dim])`
  Décalage des éléments d'une dimension donnée
  Remplissage des éléments perdus par la valeur `boundary`
  Valeurs par défaut selon le type du tableau : 0, 0.0, (0.0, 0.0), `.false.`

Compression

- `PACK (array, mask [,vector])`
  Compresse un tableau sous le contrôle d'un masque
  L'exemple

```
integer, dimension (3,3) :: a = 0
a = eoshift (a, shift = (/ 0,-1,1 /),
                boundary = (/ 1, 9, 7 /))
print *, pack (a, mask= a/=0)
print *, pack (a, mask= a/=0,
                  vector = (/ (i**3, i=1,5) /))
```

produit le résultat suivant pour `a` :

$$a = \begin{pmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{pmatrix}$$

et affiche

```
(/ 9,7 /)
(/ 9,7,27,64,125 /)
```


- `UNPACK (vector, mask, field)`
  Décompresse un vecteur sous le contrôle d'un masque
  L'exemple

```
integer, dimension (3) :: v2 = 1, v = (/1,2,3 /)
logical, dimension (3,3) :: m
integer, dimension (3,3) :: a, fld
m = reshape ((/ (((i==j), i=1,3), j=1,3) /), (/ 3,3 /))
fld = unpack (v2, mask=m, field=0)
m = cshift (m, shift=(/-1,1,0 /), dim=2)
a = unpack (v, mask=m, field=fld)
```

produit

$$m = \begin{pmatrix} T & F & F \\ F & T & F \\ F & F & T \end{pmatrix} \qquad fld = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$m = \begin{pmatrix} F & T & F \\ T & F & F \\ F & F & T \end{pmatrix} \qquad a = \begin{pmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Sélection

- `MERGE (tsource, fsource, mask)`
  - Séléctionne une valeur entre deux selon une valeur logical (scalaire)
    `MERGE (1.0, 0.0, R<0)` si R = -3 $\Rightarrow$ 1.0
    si R = 7 $\Rightarrow$ 0.0
  - Séléctionne entre deux éléments correspondants de deux tableaux en fonction d'un masque
    L'exemple

    ```
    integer, dimension (3,3) : a, b, c
    c = merge (a, b, a>b)
    ```

    est équivalent à

    ```
    c = max (a, b)
    ```

Réplication

- `SPREAD (source, dim, ncopies)`

  Duplique un tableau par réplication suivant une nouvelle dimension

  L'exemple

```
integer, parameter :: nc=3
integer, dimension (3) :: a = (/ 4,8,2 /)
print *, spread ('c', 1, 4)
print *, spread (a, 2, nc)
```

produit

$$\begin{pmatrix} c \\ c \\ c \\ c \end{pmatrix} \qquad \begin{pmatrix} 4 & 4 & 4 \\ 8 & 8 & 8 \\ 2 & 2 & 2 \end{pmatrix}$$

Transposition

- `TRANSPOSE (matrix)`

  Transpose une matrice passée en argument

  L'exemple

```
integer, dimension (3,3) :: a, b, c
a = reshape ( (/ (i**2, i=1,9) /), (/ 3,3 /))
b = transpose (a)
c = reshape (a, (/ 3,3 /), order= (/ 2,1 /))
```

produit

$$a = \begin{pmatrix} 1 & 16 & 49 \\ 4 & 25 & 64 \\ 9 & 36 & 81 \end{pmatrix} \qquad b = \begin{pmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{pmatrix} \qquad c = \begin{pmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{pmatrix}$$

Structures de contrôle

- Structures de contrôle plus puissantes que Fortran 77
- `IF`

```
[ <name> : ] IF ( < l-expr > ) THEN
                 <block>
             [ ELSE IF ( <l-expr> ) THEN [ <name> ]
                 <block> ]...
             [ ELSE [ <name> ]
                 <block> ]
             END IF [ <name> ]
```

- `CASE`, par exemple :

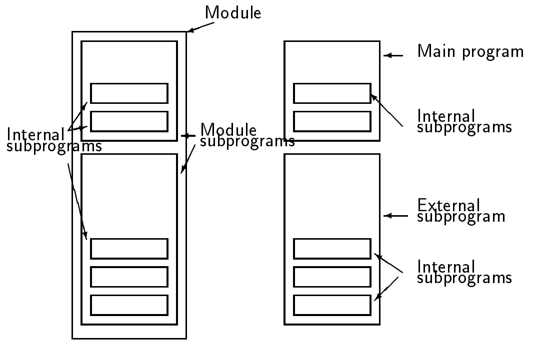
```
SELECT CASE (NUMBER)
   CASE (:-1)
      SIGN = -1
   CASE (0)
      SIGN = 0
   CASE (1:)
      SIGN = 1
END CASE
```


- `DO`, boucle sans numéro

```
[ <loop-name> : ]
   DO < var-name> = <lower> , <upper> [ , < step> ]
      <...>
      [ EXIT <loop-name> ]
      [ CYCLE <loop-name> ]
      <...>
   END DO [ <loop-name> ]
```

Unité de programme

- Programmes, procédures internes, procédures externes, modules, visibilité...




```
PROGRAM
MODULES        CONTAINS
FUNCTION       INTERFACE
SUBROUTINE
```


Conclusion

- Fortran 90 est largement critiqué pour sa taille et sa complexité trop importantes
  - Langage généraliste
  - Historique de Fortran
- Insuffisance :

  Pas de description par vecteur de bit

  Incomplétude :

  Pointeur sur un vecteur décrit par triplet et non par liste d'index

  Incohérence :

  Paramètres tableaux
  Masquage des paramètres dans un `WHERE`
- Compatibilité avec Fortran 77
- Puissance des constructions littérales
- Nouvelle norme, longue élaboration

  Prise en compte de certaines constructions MasPar Fortran, CM Fortran...
- Extension pour le parallélisme :

  Parallélisme de Données

HIGH PERFORMANCE FORTRAN

Table des matières

- High Performance Fortran Forum, 1992/1993
- Proposition d'évolution du standard Fortran pour les machines parallèles à mémoires distribuées (SIMD et MIMD)
- Notion de distribution des données
- Reprise de différents langages :
  - Fortran 90
  - MP Fortran (DEC/MasPar)
  - Connection Machine Fortran (TMC)
  - Fortran D (Rice university)
  - Vienna Fortran (Vienna university)

DEC/MASPAR FORTRAN

Table des matières

Compilateur MP Fortran

Avantages

- Gestion d'une interface DPU/frontal (asynchrone) prise en compte uniquement par le compilateur, i.e. sans intervention du programmeur
- Virtualisation des tableaux plus grands que la grille de PEs
- Gestion fine des registres des PEs assurée par le compilateur

Résumé Cacher l'architecture au programmeur tout en tirant parti de la puissance de la machine

Architecture de la DECmpp

Frontal DECstation 5000/240

- Station ULTRIX
- Machine de développement
- Exécution du code scalaire

PEs Processeurs Élémentaires

- 1024 à 16384 PEs
- ALU, registres, mémoire
- Bit de contexte
- Matrice de PEs (32×32 à 128×128)

DPU Data Parallel Unit

- Ensemble des PEs
- Mécanismes de communications entre les PEs
  - 8 voisins : Xnet
  - Général : Global Router
- ACU — Array Control Unit
  - Contrôle des PEs
  - (Exécution du code scalaire)

Les tableaux MP Fortran

Notions

- Scalaire
- Tableau
  - Une ou plusieurs dimensions, « rang »
  - Nombre d'éléments dans chaque dimension « shape »

Types de tableau

- Tableau complets (whole array)
- Sections de tableaux (array sections)
- Constructeurs (array constructors)

Conformité

- Opération sur des tableaux conformes
- Même rang, même shape
- Scalaire conforme avec tout tableau

Manipulation de tableaux

Fortran 77

```
      REAL A (10, 10),B (10, 10), C (10, 10)
      ...
      DO 30, I = 1, 10
         DO 30, J = 1, 10
            A(I,J) = B(I,J) + C(I,J)
   30 CONTINUE
```

$\Rightarrow$ Exécution sur le **frontal**

Fortran 90

```
      REAL A (10, 10),B (10, 10), C (10, 10)
      ...
         A = B + C
```

$\Rightarrow$ Exécution sur le **DPU**

Sections de tableaux

- Une section d'un tableau est
  - un tableau (sous-tableau)
  - formée d'éléments d'un tableau
  - obtenue par une opération de description
  - utilisée en partie droite comme en partie gauche
- Deux spécifications de description :
  - description par triplet
  - description par un vecteur

Notation triplet

- Équivalent « data-parallèle » de la boucle DO Fortran
- Sous-ensemble d'éléments d'un tableau spécifié par
  - borne inférieure
  - borne supérieure
  - pas
- Exemple

```
INTEGER    SIZE
PARAMETER (SIZE = 20)
REAL      X (SIZE/2), Y (SIZE), Z (SIZE)

   X = Y (2:SIZE:2) * Z (::2)
```

- Exemple

```
C (I,J) = SUM (A (:,J) * B (I,:))
```


Description par un vecteur

- Opération de « gather/scatter »
- Un descripteur vecteur est un tableau d'entiers de dimension 1
  Chaque élément est un index dans la dimension décrite
- Exemple

```
INTEGER A (10)
INTEGER V (5), R(5)
DATA A /21,22,23,24,25,26,27,28,29,30/
DATA V /2, 4, 10, 8, 10/

   R = A (V)
```

range les valeurs 22, 24, 30, 28, et 30 dans R

- Une description par un vecteur utilise le « global router »

- En partie gauche d'une affectation, les index ne peuvent pas se répéter (résultat non garanti)

```
INTEGER A (10), V (5), R(5)
DATA A /21,22,23,24,25,26,27,28,29,30/
DATA V /2, 4, 10, 8, 10/
DATE R /1, 2, 3, 4, 5/

   A (V) = R          ! ERREUR
```

A (10) vaut 3 ou 5?

- La norme Fortran 90 interdit de tel doublons
  MP Fortran ne garantit pas le résultat

- Description indépendante pour chaque dimension

```
INTEGER, ARRAY (5) :: V
INTEGER, ARRAY (5,5) :: A, RSLT
DATA V /5, 4, 3, 2, 1/

   RSLT = A (V, 5:1:-1)
```

équivalent data-parallèle au code Fortran 77

```
   DO 10 J=1, 5
      DO 10 I=1, 5
         RSLT (I,J) = A (V(I), 6-J)
10 CONTINUE
```

Constructeurs de tableaux

- Utilisés pour
  - initilialisation de tableaux
  - valeur littérale de tableaux
- Liste de valeurs scalaires

```
REAL A4(4)
   A4 = (/31.2, 83.5, 90.6, 22.1/)
```

- Ensembles en compréhension (limité aux valeurs entières[1])

```
INTEGER I6 (6)
   I6 = (/ (I, I=32,42,2) /)
   I6 = (/ (3, I=1,6) /)
   I6 = (/ ((/3,4/), I=1,3) /)
```

1. Ce n'est pas une limitation de Fortran 90.

- Combinaisons

```
INTEGER I6 (6)
   I6 = (/12, 24, (I, I=1,4) /)
```

- Limitations (MP Fortran n'existant pas en Fortran 90)

```
(/ (I+1, I=32, 42, 2) /)     ! ERREUR
(/ (I, I=1,3), J=1,2 /)      ! ERREUR
```

- Littéraux de rang supérieur à 1

```
INTEGER I23 (2, 3)
   I23 = RESHAPE (
&           SOURCE = (/1, 2, 3, 4, 5, 6/),
&           SHAPE  = (/2, 3/))
```

range les valeurs suivantes dans I23:

1	2	3
4	5	6

Affectation de tableaux

- 3 types d'affectation :
  1. Affectation directe d'un tableau
  2. Affectation conditionnelle par valeur WHERE
  3. Affectation conditionnelle par position FORALL
- Affectation directe
  - Affectation entre deux tableaux conformes (en rang et en shape)
  - Ne pas référencer deux fois le même élément en partie gauche

Constructeur WHERE

- Une affectation incluant ou excluant des éléments d'un tableau selon leur valeur
- Exemple d'instruction

```
WHERE (A .NE. 0) C = B/A
```

évite les divisions par zéro

- Exemple de constructeur (affectation de tableaux uniquement)

```
WHERE (PRESURE <= 1.0)
   PRESURE = PRESURE + 1.0
   TEMP = TEMP - 5
ELSE WHERE
   RAINING = .TRUE.
END WHERE
```

- Appel de fonction dans un WHERE : fonctions « elemental »

```
   WHERE (Z > 0)
10    Z = LOG (Z)
20    Z = SUM (LOG (Z))
   END WHERE
```

en 10, le $\log$ est réalisé sur les éléments positifs
en 20, le $\log$ est réalisé sur tous les éléments

Constructeur FORALL

- Équivalent data-parallèle de la boucle DO
- N'a pas été retenu dans le standard Fortran 90
- Exprime une affectation parallèle de tableau, élément par élément

```
  FORALL (I=1:10, J=1:10)
&    A (I,J) = B (J,I) / C(I,I)
```

- L'affectation peut être associée à un masque

```
  FORALL (I=1:10, J=1:10, C(I,I) .NE. 0)
&    A (I,J) = B (J,I) / C(I,I)
```

- Opération très générale ne pouvant pas toujours produire du code parallèle

Allocation de tableaux

Tableaux « assumed-shape »

- Paramètre formel d'une fonction
- La shape est transmise par le paramètre effectif
- Les rangs du paramètre formel et du paramètre effectif doivent correspondre
- Exemple

```
SUBROUTINE ASA  (A, B, C, M)
   REAL A (:), B (0:, :), C (M:, :)
```

- La valeur de la borne supérieure d'un tableau assumed-shape est calculée à partir de

  b_i la borne inférieure déclarée

  t la taille du paramètre effectif

  $b_s = t - b_i + 1$

Tableaux automatiques

- Tableaux locaux d'une fonction
  dont la taille diffère d'un appel à l'autre
- Alloués dynamiquement à l'entrée de la fonction
  Désalloués automatiquement à la fin de la fonction
- Syntaxe :
  - le tableau doit être déclaré local (i.e. ne pas être un paramètre, ne pas être dans un common)
  - sa taille doit être spécifiée par une variable paramètre ou common
- Exemple : échange de deux tableaux de taille quelconque

```
SUBROUTINE SWAP (A,B)
   REAL, DIMENSION (:) :: A, B
   REAL, DIMENSION (SIZE(A)) :: TEMPOAUTO
   TEMPOAUTO = A
   A = B
   B = TEMPOAUTO
END SUBROUTINE
```


DPU et FE arrays

- Deux espaces mémoire :
  - Frontal (FE)
  - PE (DPU)

  $\Rightarrow$ deux types de tableaux
- Des règles déterminent comment (où) sont alloués les tableaux
- Des directives peuvent forcer l'allocation d'un tableau
- Compréhension de l'allocation est nécessaire pour éviter les coûteux transferts de données entre FE et DPU
- Le programmeur doit assurer la cohérence entre les paramètres formels et effectifs des fonctions pour éviter les copies

DPU/FE arrays : règles

- Variables locales et paramètres formels
  - Tableaux numériques et logiques utilisés à la Fortran 90 ⇒ DPU
  - Autres tableaux et les scalaires ⇒ FE
- Avec l'option `-fecommon` (défaut), toutes les variables `COMMON` sont allouées sur le FE
- Avec l'option `-nofecommon`, les variables `COMMON` sont allouées
  - Tableaux numériques et logiques → DPU
  - Tableaux caractères et scalaires → FE
  - S'il y a conflit → le `COMMON` est alloué sur le FE

- Exemple

```
SUBROUTINE FOO (A, M, N)
   INTEGER M, N
   INTEGER A (M,N)
   A = 0                          ! DPU
   DO J = 1, N
      DO I = 1, M
         A (I,J) = A (I,J) + 1  ! Inefficace
      END DO
   END DO
END
```

- Exemple

```
SUBROUTINE  FOO (A)
   INTEGER A (1000,1000)
   ...
END

PROGRAM EX
   INTEGER A (1000000)          ! FE
   CALL FOO (A)                 ! Légal car FE
END
```

Uniquement en Fortran 77 (i.e. sur le FE) les tableaux déclarés avec un nombre de dimensions peuvent être référencés comme possédant un autre nombre de dimensions

- Exemple

```
PROGRAM EX
   INTEGER A (1000)             ! FE
   CALL FOO (A)
END

SUBROUTINE FOO (A)
   INTEGER A (1000)
   A = 0                        ! DPU
END
```

Lors de l'appel, le transfert entre le FE et le DPU est réalisé
Au retour le transfert inverse est réalisé

- Directives `ONDPU` et `ONFE` permettent au programmeur de spécifier où allouer un tableau

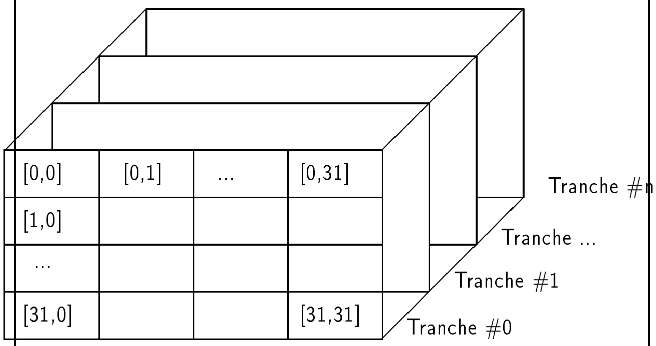
```
      PROGRAM EX
         INTEGER A (1000)
CMPF     ONDPU A
         CALL FOO (A)           ! DPU
      END

      SUBROUTINE FOO (A)
         INTEGER A (1000)
         A = 0                 ! DPU
      END
```


- Directives valides pour
  - les tableaux logiques et numériques
  - les `COMMON` ne comportant que des tableaux
- Directives invalides pour
  - les tableaux appartenant à un `COMMON`
  - les scalaires
  - les tableaux de caractères

⇒ l'allocation des tableaux est source de nombreuses erreurs en MP Fortran

Mapping des tableaux

- Les tableaux MP Fortran sont de taille quelconque, différentes de la taille de la machine
- Le mapping définit comment ces tableaux sont alloués dans la mémoire des processeurs
- Mapping par défaut (Canonical Array Allocation) : *Cut & Stack*
- Tranches mémoire sur les PEs



[PE x-coordinate, PE y-coordinate, # tranche]

Mapping canonique, tableaux 1-D

- Considère les processeurs ligne par ligne et alloue les éléments du tableau successivement sur les processeurs
- Exemple : `A (2048)` sur une machine 32×32

Tranche # 0

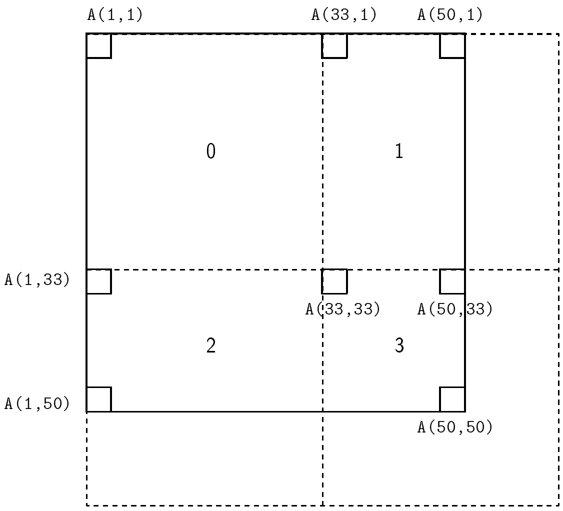
A (1)	A (2)	...	A (32)
A (33)	...		
...			
...			
A (993)			A (1024)

Tranche # 1

A (1025)	A (1026)	...	A (1056)
A (1057)	...		
...			
...			
A (2017)			A (2048)

Mapping canonique — tableaux 2-D

- Tableau `A (50,50)` sur une machine 32×32



4 tranches
Par exemple, sur le PE [0,0], sont alloués les 4 éléments
`A (1,1)`, `A (33,1)`, `A (1,33)`, et `A (33,33)`

- Attention `A (2,256)` sur une machine 32×32

  ⇒ 8 tranches
  32×32×8 = 8192 emplacements mémoire
  pour 2×256 = 512 éléments

  et n'utilise que 64 PEs/1024

Mapping canonique — tableaux de plus de 2 dimensions

- Les deux premières dimensions sont « mappées » comme pour un tableau 2-D
- Les dimensions d'ordres supérieurs sont « mappées » en mémoire
- `INTEGER A (50, 50, 8)`

  $\Rightarrow 4\times 8 = 32$ tranches (32×32)
- `INTEGER A (50, 8, 50)`

  $\Rightarrow 2\times 50 = 100$ tranches (32×32)

Directives de mapping

- Contrôler finement le mapping des tableaux sur les PEs
- Exemple, le mapping canonique

```
      INTEGER A (100,100,100,100)
CMPF  MAP A (XBITS, YBITS, MEMORY, MEMORY)
```

- Exemple, allouer une seule dimension sur la grille

```
      INTEGER A (100,100,100,100)
CMPF  MAP A (MEMORY, ALLBITS, MEMORY, MEMORY)
```

- Exemple, 1re dim. en mémoire, 3e dim. sur les PEs puis 2e dim. sur le reste des PEs

```
      INTEGER B (10,50,50)
CMPF  MAP A (MEMORY, PACKBITS:2, PACKBITS:1)
```

  Les dimensions ne pouvant être allouées sur les processeurs le sont en mémoire

Problème C'est au programmeur d'assurer la cohérence du mapping entre les paramètres formel et effectif

Communications en MP Fortran

- En MP Fortran, ce n'est pas le programmeur qui gère les communications
- La maîtrise des communications est essentielle
- Le programmeur doit comprendre comment le compilateur gère les communications

Communications Frontal/DPU

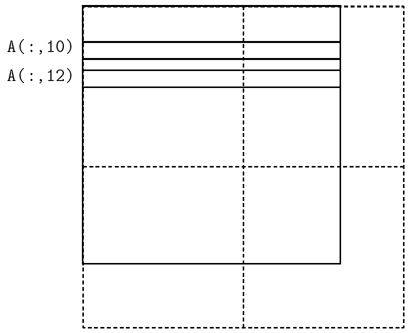
- Opération « Fortran 77 » : manipulation de scalaires, de tableaux par indiçage scalaire

  Opération « Fortran 90 » : manipulation de tableaux en tant qu'entité
- La manipulation d'un tableau DPU dans une opération « Fortran 77 »
  ⇒ Communication DPU → FE
  Manipulation
  Communication FE → DPU

Communication DPU/DPU

- Les manipulations lignes/lignes ou colonnes/ colonnes de tableaux ⇒ communications « Xnet »

```
INTEGER A (50,50)
        A (:,10) = A (:, 12)
```



4 tranches mémoire d'une grille 32×32

- Les autres manipulations introduisent des communications via le « Global router »

```
INTEGER A (50,50)
INTEGER V (50)
   A (:,10) = A (12,:)
   A (V, 10) =  A (12,:)
```

- Attention aussi aux directives de mapping

Interface MPL/MP Fortran

- MPL ⇒ accès direct à la machine
- MP Fortran appelle MPL pour contrôler plus finement la machine
- Identifier les fonctions MPL ; directive `MPL`

```
CMPF  MPL FOO, BAR
      CALL FOO
      CALL BAR
```

- MP Fortran : FE
  MPL : ACU

  ⇒ 3 espaces d'adressage : FE : MPF
  ACU : MPL
  PEs : MPF & MPL

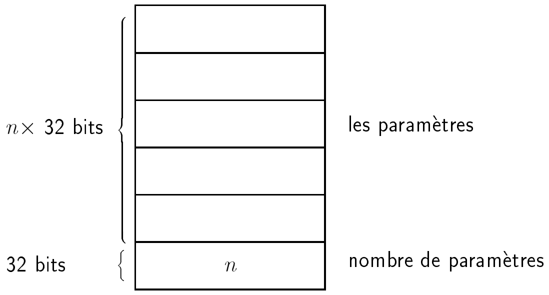
Blocs d'arguments

- À toute fonction MPL appelée depuis MPF sont passés deux paramètres :
  1. FE argument block
  2. ACU argument block
- Dans ces deux blocs, la fonction MPL retrouve les paramètres passés par MP Fortran

```
mplFuncCallFromMPF (fe_arg_blk, dpu_arg_blk)
   void *fe_arg_blk [] ;
   DopeVector *dpu_arg_blk [] ;
{
...
}
```


FE block

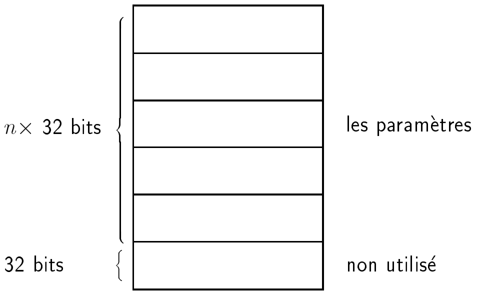
- FE block est un pointeur dans la mémoire du FE, d'où le `void *`



- Un mot de 32 bits correspond à chaque paramètre
- Un paramètre scalaire → son adresse (FE)
- Un paramètre tableau FE (non utilisé dans une instruction Fortran 90 ou directive `ONFE`)
  → son adresse de base (FE)
- Un paramètre tableau DPU (utilisé Fortran 90 ou `ONDPU`)
  → indéfini

ACU block

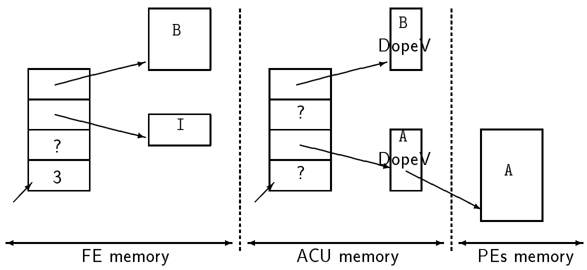
- ACU block est un tableau de pointeur dans la mémoire de l'ACU, d'où le `DopeVector *dpu_arg_blk []`



- À chaque paramètre correspond un pointeur (32 bits)
- Un paramètre scalaire → non défini
- Un paramètre tableau → pointeur sur un `DopeVector`, structure (cf. ci-après)

- Exemple

```
CMPF  MPL FOO
      INTEGER I
      REAL, DIMENSION (100) :: A, B
CMPF  ONDPU A
CMPF  ONFE B
      CALL FOO (A, I, B)
```



Dope vector

- Structure
- Présente sur l'ACU
- Information sur un paramètre tableau MP Fortran

```
typedef struct {
      /* adresse du tableau dans la mémoire des PEs */
   plural int * adr ;
      /* rank = nombre de dimension du tableau */
   unsigned int rank ;
      /* VRAI si le tableau réside sur les PEs
         (sinon adr est invalide) */
   unsigned int OnDpu ;
      /* tailles des dimensions du tableau */
   unsigned int extent [7] ;
} DopeVector ;
```


- Exemple : afficher les caractéristiques du tableau passé en paramètre
- Code MP Fortran `test.f`

```
      PROGRAM TESTDOPEVECTOR
         INTEGER ARRAY (10, 20) :: A
CMPF     MPL PRINTDOPE
         A = 42
         CALL PRINTDOPE (A)
      END
```


- Code MPL `dope.m`

```
PRINTDOPE (fe_arg_blk, dpu_arg_blk)
   /* nom en majuscule ! */
   void *fe_arg_block ;
   DopeVector *dpu_arg_block [] ;
{
   DopeVector *Dptr ;
   int i ;
   Dptr = dpu_arg_blk [1] ;
   printf ("rank = %d\n", Dptr-> rank) ;
   if (Dptr-> OnDpu) printf ("OnDpu is true\n") ;
             else printf ("OnDpu is false\n") ;
   printf ("extent [") ;
   for (i=0; i<Dptr-> rank; i++)
      printf ("%d,", Dptr-> extent [i]) ;
   printf ("]\n") ;
}
```

- Compilation

```
mpl -c dope.m
mpfortran test.f dope.o
```


Mapping des tableaux

- Les tableaux MPF sont virtuels ; de taille quelconque, indépendante de la machine
- Le mapping définit comment ces tableaux sont découpés pour être alloués sur la grille (32×32 ou 128×128) des processeurs
- Les fonctions MPL qui reçoivent des entêtes de vecteurs MPF doivent tenir compte de ce mapping pour accéder aux tableaux

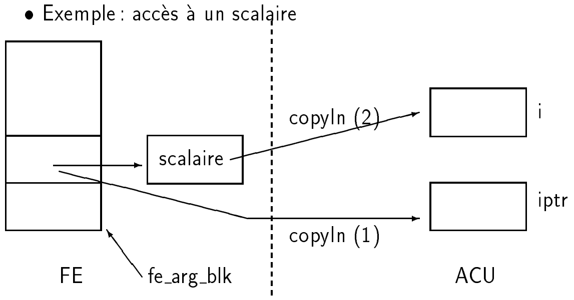
Accès aux scalaires et aux tableaux FE

- À partir du paramètre `fe_arg_blk` passé à la fonction MPL, on peut accéder aux
  - scalaires
  - tableaux FE
- Utilisation de

  `copyIn()` pour ramener les valeurs FE → ACU

  `blockIn()` pour ramener les valeurs FE → PEs

- Exemple : accès à un scalaire



- Code MPL :

```
int i, *iptr ;

/* (1) */
copyIn (&fe_arg_blk [j], &iptr, sizeof(iptr)) ;
/* on a (sur l'ACU) l'adresse du scalaire */

/* (2) */
copyIn (iptr, &i, sizeof(i)) ;
/* on a la valeur de i sur l'ACU */
```


Conclusion

- MP Fortran fournit une virtualisation de la DECmpp
- MP Fortran est une extension de Fortran 77 incluant des constructions de Fortran 90
- MP Fortran fournit les moyens d'allouer finement les tableaux sur les PEs par des directives

  Ces directives vont évoluer vers le prochain standard : *High Performance Fortran*
- Fortran sur DECmpp, c'est aussi *VAST-2* : vectoriseur/paralléliseur automatique

Fortran 77 ⟶ VAST-2 ⟶ MP Fortran

MP Fortran ⟶ VAST-2 ⟶ Fortran 77

CM FORTRAN

Table des matières

CM Fortran

- Connection Machine Fortran
- Proposé par Thinking Machine Corp. pour la CM-2, repris sur la CM-5
- Devient le langage stratégique pour TMC
- Extension Fortran 77 dans la direction de Fortran 90
- Ajoute des constructions relatives à l'alignement et à la distribution des tableaux

Tableau

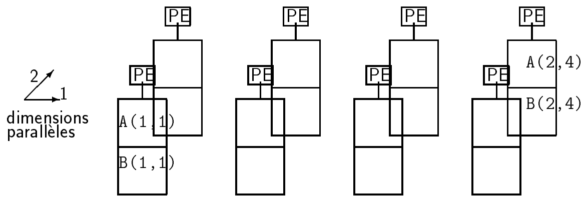
- Entité du langage (Fortran 90)
- Description par triplet, vecteur d'index
- Allocation : la machine cible
  - accédé uniquement en scalaire → FE
  - accédé en parallèle → PEs

Allocation des tableaux

- Au niveau du programmeur Fortran, la Connection Machine est constituée d'un nombre virtuellement infini de PEs
- Les éléments de vecteurs sont alloués sur les PEs
  - à raison de un par PE virtuel
  - suivant une allocation « canonique »
  - dépend des informations de déclaration

⇒ deux tableaux de même forme et même taille sont alloués sur le même ensemble de PEs

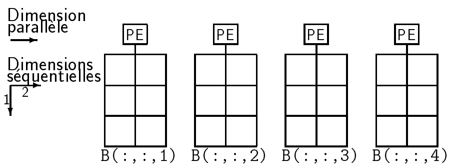
```
DIMENSION A (2,4)
DIMENSION B (2,4)
```



Directive LAYOUT

- Indique, pour chaque dimension d'un tableau si les éléments doivent être alloués
  - sur différents PEs : `:NEWS`
  - dans la mémoire d'un PE : `:SERIAL`
- Exemple

```
     DIMENSION B (3,2,4)
CMF$ LAYOUT    B (:SERIAL, :SERIAL, :NEWS)
```



- Une allocation `:SERIAL` ne produit aucune communication lors de traitements mettant en œuvre uniquement des éléments de tableaux alloués sur un même PE

```
B(1,1,:) = ( B(2,1,:) + B(3,1,:)
           + B(2,2,:) + B(3,2,:) ) / 4
```

- Cette directive LAYOUT peut aussi forcer l'alignement de d'éléments de deux tableaux sur un même jeu de PEs

```
      DIMENSION A (10, 256, 256)
      DIMENSION B (256, 256)
CMF$ LAYOUT    A (:SERIAL, :NEWS, :NEWS)
CMF$ LAYOUT    B (:NEWS, :NEWS)

      ! pas de communications inter-PEs
      B = A (3, :, :)
```

- Des LAYOUT différents entraînent d'éventuelles communications

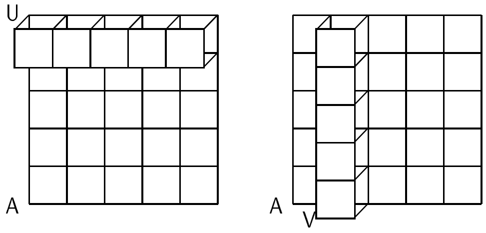
```
      DIMENSION A (10, 256, 256)
      DIMENSION C (10, 256, 256)
CMF$ LAYOUT    A (:SERIAL, :NEWS, :NEWS)
CMF$ LAYOUT    C (:NEWS, :NEWS, :NEWS)

      A = C            ! communications inter-PEs
```

- Poids sur les directives

```
      DIMENSION A (100, 100, 100)
CMF$ LAYOUT    A (:SERIAL, 2:NEWS, :NEWS)
```

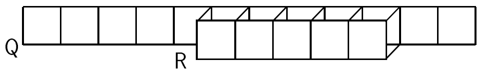

Directive ALIGN

- Allocation des éléments de deux tableaux dans les mêmes PEs, de manière contrôlée
- Exemple : « Alignement de projection »
  - aligner les éléments de U avec ceux de la première ligne de A
  - aligner les éléments de V avec ceux de la 2e colonne de A




```
      DIMENSION U(5), V(5), A(5,5)
CMF$ ALIGN U(I) WITH A(I,1)
CMF$ ALIGN V(I) WITH A(2,I)
```


- Exemple : « Alignement de déplacement »
  - aligner les éléments de R sur ceux de Q




```
      DIMENSION R(5), Q(12)
CMF$ ALIGN R(I) WITH Q(I+5)
```

- Alignements trop complexes non autorisés
  - « transposée de tableaux »
  - ...

  ⇒ Allocation d'un tableau toujours
  - contiguë
  - croissante

  sur les PEs

Visibilité des directives

- Une directive d'alignement LAYOUT ou ALIGN n'est « visible » que dans la suite du code où elle apparaît

⇒ Si un tableau « aligné » est utilisé dans différents modules, les directives doivent être répétées dans tous les modules

Passage d'arguments

- Le passage en paramètre d'un tableau « aligné »
  - copie du tableau dans un temporaire « canonique »
  - appel de la procédure avec cette copie
  - réalisation de la copie inverse au retour
- On peut éviter les copies :
  1. Définition d'un « Interface Block »
  2. Répétition de l'alignement
     - dans la procédure appelante
     - dans la procédure appelée
     - dans l'interface block

FORTRAN D

Table des matières

Fortran D

- Extension de Fortran
  - → machines scalaires
  - → machines parallèles SIMD
  - → machines parallèles MIMD
- Rice university, Ken Kennedy
- D : Data
  Decomposition
  Distributed

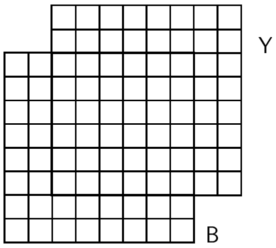
Distribution des données

- 2 étapes :
  - problem mapping
  - machine mapping
- Problème mapping
  - définition de `DECOMPOSITION`
  - $\simeq$ tableaux non-alloués
  - placement des tableaux sur les décompositions : `ALIGN`
- Machine mapping
  - placement des décompositions sur les processeurs physiques de la machine : `DISTRIBUTE`

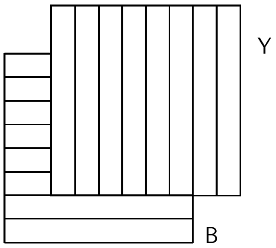
Alignement

- Déplacement par rapport à une décomposition
- Éclatement (rangement régulier suivant un pas)
- Permutation (ex. row-wise)
- Regroupement (plusieurs éléments sur un même élément d'une décomposition)
- Placement (d'un vecteur sur une matrice 2-D par ex.)

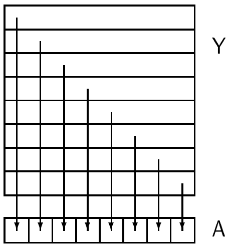
```
      REAL Y(N,N)
!FD$ DECOMPOSITION A (N), B(N,N)

!FD$ ALIGN Y(I,J) WITH B(I-2,J+2)
```




```
!FD$ ALIGN Y(I,J) WITH B(J+2,I-2)
```




```
!FD$ ALIGN Y(I,J) WITH A(I)
```



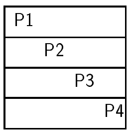
Distribution

- Pour chaque dimension d'une décomposition :

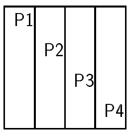
  `BLOCK` les éléments consécutifs d'une décomposition sont « alloués » sur un même processeur

  `CYCLIC` l'élément i est sur le processeur $i \bmod \#PE$

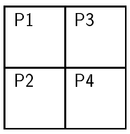
  `BLOCKCYCLIC` regroupement en blocs qui seront alloués cycliquement sur les PEs



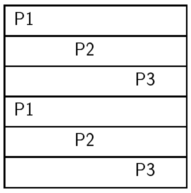
DISTRIBUTE B(BLOCK,*)



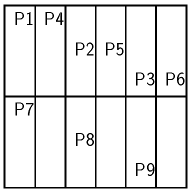
DISTRIBUTE B(*,BLOCK)



DISTRIBUTE B(BLOCK,BLOCK)



DISTRIBUTE B(CYCLIC,*)



DISTRIBUTE B(CYCLIC,CYCLIC)

Dynamicité

- Les spécifications d'alignement ALIGN et de distribution DISTRIBUTE ne sont pas seulement des directives, ce sont des instructions

⇒ Alignement et distribution des tableaux varient au cours de l'exécution du programme

Passage de paramètres

- Avant l'appel, ré-alignement, re-distribution suivant un alignement/une distribution standards
- Copie au retour

⇒ Inefficacité !!

Autres constructions

- Boucle FORALL
- Opérations de réduction
- Localisation des traitements (ON)
- Exemple :

```
REAL X(1024), Y(1024), Z(1024)
DECOMPOSITION A(1024)
ALIGN X, Y, Z WITH A
DISTRIBUTE A(BLOCK)
FORALL I = 1, 512 on HOME (A (I))
    X(I+512) = F(X(I), Y(I), Z(I))
END FORALL
```


HIGH PERFORMANCE FORTRAN

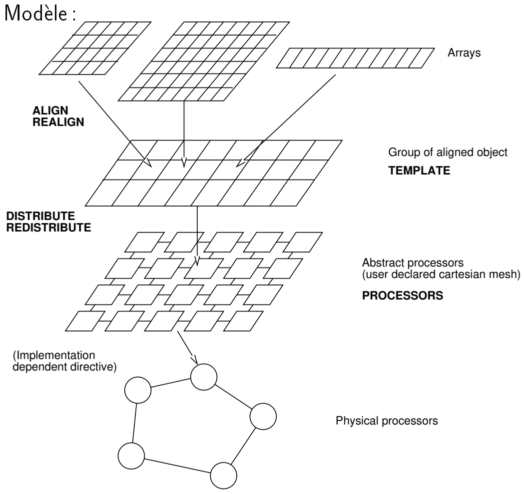
Table des matières

High Performance Fortran

- Extensions (+ modifications) de la norme Fortran 90
- Data Parallel Programming
- Top performance on MIMD + SIMD
  (with non-uniform memory access cost)
  (pas de perte de perfs. sur les autres machines)
- `FORALL` + fonctions intrinsèques
- Directives de distribution des données
- Minimum de conflits avec les autres langages
- Simplicité (!)
- Interface avec les autres langages
- Penser à une intégration dans les futurs standards C et Fortran
- Diffusion des travaux & Prise en compte des réactions
- Définition finale jan. 93 (mars)
- Disponibilité rapide de compilateurs
  ⇒ montrer les perfs. de HPF sur des tests
- Abandon : explicit MIMD programming (message passing, synchronization)

Distribution des données

- Modèle :



- Intuitions :
  - Une opération sur différents objets est plus efficace si les objets sont sur le même processeur
  - De telles opérations peuvent être exécutées en parallèle si elles peuvent être réalisées sur différents processeurs

- Deux types de directives
  - specification statements
  - executable statements
- Specification statements
  - « exécutés » une fois : à l'entrée du programme ou de la fonction
  - « déclarations »
- Executable statements
  - instruction, exécuté comme toute instruction

- Immédiatement et ultimement aligné :
  - si un objet A est aligné avec un objet B
    et que B est ensuite aligné avec C
    alors
    - ◇ A est aligné *ultimement* avec C
    - ◇ A est aligné *immédiatement* avec B
  - si B est ensuite réaligné, l'alignement ultime entre A et C subsiste
- Une fois un objet créé, il peut être déplacé par réalignement et/ou redistribution d'un objet avec lequel il est ultimement aligné

  I.e. la redistribution d'un objet garantit toujours la validité de la relation d'ultime alignement

Terminologie

`TEMPLATE` Un template peut être vu comme un tableau d'éléments ne possédant ni valeur ni allocation

Il est utilisé pour spécifier des alignements comme un « cadre » ou un « repère »

`PROCESSORS` Grille virtuelle de processeurs

Utilisés pour distribuer les `TEMPLATE` ou les tableaux

`DYNAMIC` Pour pouvoir être dynamiquement (i.e. à l'exécution) re-aligné ou re-distribué, un tableau doit être déclaré `DYNAMIC`

Mapping relation entre un objet et un processeur virtuel/un ensemble d'objets et un ensemble de processeurs virtuels

Directive PROCESSORS

- Autorise la déclaration de grilles (1-, 2-, 3-...D) de processeurs virtuels
- L'intention (sic !) est que deux objets « mappés » sur le même processeur virtuel soient alloués sur le même processeur physique
- Si $\#\text{virtuels} \geq \#\text{physiques} \Rightarrow$ ok
- Si $\#\text{virtuels} < \#\text{physiques}$?
  1. processeurs physiques inutilisés ?
  2. ne garantit pas l'allocation sur le même processeur physique des objets mappés sur le processeur virtuel ?
- Valeurs intrinsèques

```
NUMBER_OF_PROCESSORS
PROCESSORS_SHAPE
```

$\Rightarrow$ caractéristiques de la machine physique

- Exemples

```
!HPF$ PROCESSORS P (N)
!HPF$ PROCESSORS Q (NUMBER_OF_PROCESSORS()),
!HPF$            R (NUMBER_OF_PROCESSORS()/8, 8)
!HPF$ PROCESSORS SCALARPROCESSOR
```

- Déclaration PROCESSORS sans shape déclare un processeurs scalaire

  Selon l'implémentation
  - sur le hôte ou le processeur de contrôle
  - sur un processeur arbitraire
  - répliqué sur tous les processeurs

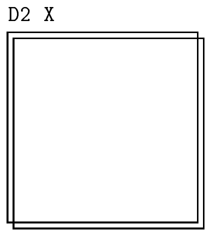
- VIEW permet de voir un même ensemble de processeurs sous différentes géométries
  « EQUIVALENCE pour les PROCESSORS»

```
!HPF$ PROCESSORS P (10, 10), Q (100)
!HPF$ VIEW Q OF P

!HPF$ PROCESSORS R (10, 10), S (100)
!HPF$ VIEW OF R (/1,2/) :: S

!HPF$ PROCESSORS T (10, 10), U (100)
!HPF$ PROCESSORS VIEW OF T (/2,1/) :: U
```


ALIGN et REALIGN

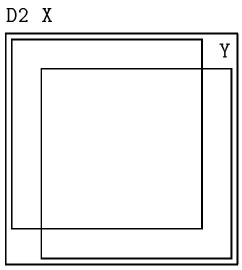
- Directives spécifiant que certains objets sont distribués de la même manière que d'autres
- Les opérations entre ces objets alignés devraient être réalisées plus efficacement qu'entre des objets non-alignés
- Exemple simple, syntaxe

```
!HPF$ TEMPLATE D1 (N), D2 (N,N)
      REAL, DIMENSION (N,N) :: X
!HPF$ ALIGN X WITH D2
!HPF$ ALIGN WITH D2 :: X
!HPF$ ALIGN X(I,J) WITH D2(I,J)
!HPF$ ALIGN (I,J) WITH D2(I,J) :: X
!HPF$ ALIGN X(:,:) WITH D2(:,:)
!HPF$ ALIGN (:,:) WITH D2(:,:) :: X
```



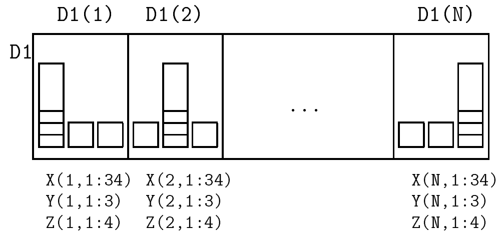
- Exemple

```
!HPF$ TEMPLATE D2 (N+1,N+1)
      REAL, DIMENSION (N,N) :: X, Y
!HPF$ ALIGN X(I,J) WITH D2(I,J)
!HPF$ ALIGN Y(I,J) WITH D2(I+1,J+1)
```



- Exemple, réduction du parallélisme à un axe

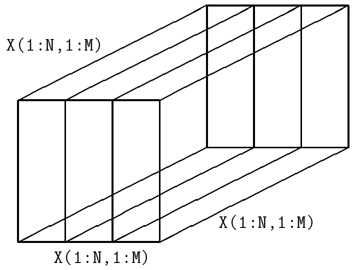
```
!HPF$ TEMPLATE D1 (N)
      REAL X (N, 34)
      REAL Y (N, 3)
      REAL Z (N, 4)
!HPF$ ALIGN X(J,K) WITH D1(J)
!HPF$ ALIGN Y(J,K) WITH D1(J)
!HPF$ ALIGN Z(J,K) WITH D1(J)
```



- Syntaxe
  ∀ K indice valide
  `ALIGN X(J,K) WITH D1(J)`
  est équivalent à
  `ALIGN X(J,*) WITH D1(J)`
  ou encore
  `ALIGN X(:,*) WITH D1(:)`

- Exemple, duplication d'un axe suivant une 3e dimension

```
!HPF$ TEMPLATE D3 (N, L, M)
      REAL X (N, M)
!HPF$ ALIGN X(I,J) WITH D3(I,K,J)
!HPF$ ALIGN X(I,J) WITH D3(I,*,J)
!HPF$ ALIGN X(:,:) WITH D3(:,*,:)
```



L copies de X(1:N,1:M) sont alors réparties sur le template D3 (sur sa 2e dimension)

HPF assure la cohérence entre ces copies
attention au coût !

- Exemple, transposition de deux axes

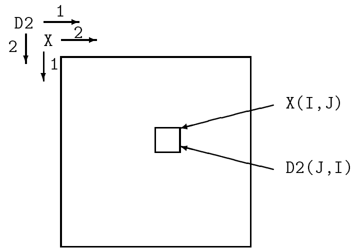
```
!HPF$ TEMPLATE D2(N,N)
      REAL     X (N,N)
!HPF$ ALIGN    X (I,J) WITH D2 (J,I)
```

ou

```
!HPF$ ALIGN    X (:,J) WITH D2 (J,:)
```

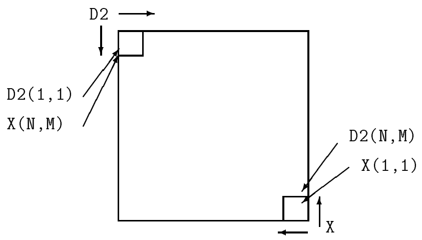
ou

```
!HPF$ ALIGN    X (I,:) WITH D2 (:,I)
```



- Exemple, inversion d'axes

```
!HPF$ TEMPLATE D2(N,M)
      REAL     X (N,M)
!HPF$ ALIGN    X (J,K) WITH D2(N-J+1, M-K+1)
```



- Alignement des paramètres de subroutines ou fonctions

(A) La déclaration d'un paramètre peut inclure une spécification d'alignement

```
      SUBROUTINE HUBBLE (TELESCOPE)
      REAL TELESCOPE (:)
!HPF$ ALIGN TELESCOPE WITH MOON
```

1. L'appelant ne connaît pas la déclaration du paramètre TELESCOPE (il n'y a pas de bloc INTERFACE)
   ⇒ le paramètre effectif DOIT être aligné avec MOON
   ⇒ il n'y aura pas de copie
2. L'appelant connaît la déclaration du paramètre TELESCOPE (il y a un bloc INTERFACE)
   ⇒ le paramètre effectif peut être aligné de n'importe quelle manière
   ⇒ il y aura une copie si nécessaire (détecter les cas de non-copie ?)

- Alignement des paramètres de subroutines ou fonctions

(B) La déclaration d'un paramètre peut inclure un alignement « à la »

```
      SUBROUTINE HUBBLE (TELESCOPE)
      REAL TELESCOPE (:)
!HPF$ ALIGN TELESCOPE WITH *
```

◇ Le paramètre TELESCOPE sera aligné comme l'est le paramètre effectif lors de l'appel

◇ Aucune copie ne peut avoir lieu lors de l'appel

◇ HPF doit gérer DYNAMIQUEMENT cet alignement

◇ C'est le moyen d'écrire des bibliothèques pour tous les paramètres, quelque soit leur alignement

- Alignement des paramètres de subroutines ou fonctions

(C) L'alignement d'un paramètre peut être réalisé sur un autre paramètre

```
      SUBROUTINE NICOLAS (TSAR, CZAR)
      REAL, DIMENSION (1918) :: TSAR, CZAR
!HPF$ ALIGN WITH *    :: TSAR
!HPF$ ALIGN WITH TSAR :: CZAR
```

1. Si les deux paramètres sont déjà alignés, il n'y aura pas de copie lors de l'appel
2. Sinon (et si bloc INTERFACE), il y aura une copie pour aligner CZAR avec TSAR

- Réalignement d'un paramètre au sein d'une fonction

  Pour être réaligné au sein d'une fonction, un paramètre doit

  1. Être déclaré DYNAMIC au niveau du paramètre effectif (appelant)
  2. Être déclaré DYNAMIC au niveau du paramètre formel

- Limitations
  - Pas d'alignement de COMMON « non-simples »
  - Pas d'alignement des tableaux de caractères
  - etc.

DISTRIBUTE et REDISTRIBUTE

- Directives spécifiant le « mapping » d'objets data sur un ensemble de processeurs abstraits
- L'idée est de rapprocher sur la grille de processeurs les $\neq$ éléments d'un tableau qui interagissent
- Directive de déclarations : DISTRIBUTE
- Directive d'exécution : REDISTRIBUTE
  $\Rightarrow$ DYNAMIC
- DISTRIBUTE : « paramètres » doivent être fixés à la compilation
- REDISTRIBUTE : toute valeur/expression entière est valide
- Deux distributions : BLOCK et CYCLIC

- Distribution par BLOCK

```
!HPF$ TEMPLATE   TT (N)
!HPF$ PROCESSORS PP (P)
!HPF$ DISTRIBUTE TT(BLOCK) ONTO PP
```

- On considère des bloc de B éléments de TT tels que

$$B = \lceil \frac{N}{P} \rceil$$

- On a donc P blocs de B éléments consécutifs de TT
- Ces blocs sont « mappés », un sur chaque processeur de PP

PP(1)	PP(2)	PP(5)
TT(1)	TT(4)	TT(13)
TT(2)	TT(5)	TT(14)
TT(3)	TT(6)	

Exemple : N =14, P =5

- Utilisation :

```
      REAL T(14)
!HPF$ ALIGN T WITH TT
      T(2:14:3) = T(1:13:3)
```

$\Rightarrow$ pas de communications

- On peut spécifier la taille des blocs

```
!HPF$ DISTRIBUTE TT(BLOCK(2))
```

- Ce sont alors ces blocs (de taille 2) qui seront distribués sur les processeurs

PP(1)	PP(2)	PP(3)	PP(4)	PP(5)
TT(1) TT(2)	TT(5) TT(6)	TT(9) TT(10)	TT(13) TT(14)	
TT(3) TT(4)	TT(7) TT(8)	TT(11) TT(12)		

- Attention :
  - perte de place mémoire
  - inactivité de certains processeurs

- Ces mêmes blocs peuvent être distribués cycliquement sur les processeurs

```
!HPF$ DISTRIBUTE TT(CYCLIC(2))
```

PP(1)	PP(2)	PP(3)	PP(4)	PP(5)
TT(1) TT(2)	TT(3) TT(4)	TT(5) TT(6)	TT(7) TT(8)	TT(9) TT(10)
TT(11) TT(12)	TT(13) TT(14)			

- L'attribut CYCLIC peut aussi être utilisé sans valeur ; les éléments sont alors distribués cycliquement
- Cette distribution peut être réalisée indépendamment sur chaque dimension d'un tableau/d'un template

Instruction FORALL

- Nouvelle instruction (non une directive !)
- Affectation « parallèle » d'éléments de tableaux

  (il existe déjà affectation tableaux, WHERE, et description par triplet, par vecteur d'index)
- Instruction FORALL et constructeur FORALL
- Il existait un FORALL dans les premiers drafts de la nouvelle norme Fortran 90

Instruction FORALL

- Affectation d'un tableau en terme
  - d'éléments du tableau
  - de groupes d'éléments de tableaux (sections)
- Inclut un « masque » par une valeur LOGICAL
- Plus général qu'une affectation + description + WHERE
- Exemples :

```
REAL H(N,N), X(N,N), Y(N,N)
...
FORALL (I=1:N, J=1:N)                    &
   H(I,J) = 1.0/REAL(I+J-1)
...
FORALL (I=1:N, J=1:N, Y(I,J) .NE. 0.0) &
   X(I,J) = 1.0/Y(I,J)
```

- Syntaxe

FORALL (<*triplet-list*> , [<*scalar-mask*>]) <*assignment*>

Exemples d'instructions FORALL

- `FORALL ( i=1:n, j=1:m ) a(i,j)=i+j`

as opposed to standard Fortran 90

```
a = SPREAD((/(i,i=1,n)/), DIM=2, NCOPIES=m) + &
    SPREAD((/(i,i=1,m)/), DIM=1, NCOPIES=n)
```

•

```
FORALL (j=1:m, k=1:n) x(k,j) = y(j,k)
FORALL (k=1:n) x(k,1:m) = y(1:m,k)
```

These statements both copy columns 1 through n of array y into rows 1 through n of array x. This is equivalent to the standard Fortran 90 statement

```
x(1:n,1:m) = TRANSPOSE(y(1:m,1:n))
```


- `FORALL (i=1:n, j=1:n) x(i,j) = 1.0 / REAL(i+j-1)`

This FORALL sets array element $x(i,j)$ to the value $\frac{1}{i+j-1}$ for values of i and j between 1 and n. In Fortran 90, the same operation can be performed by the statement

```
x(1:n,1:n) =                               &
     1.0/REAL(                             &
     SPREAD((/(i,i=1,n)/),DIM=2,NCOPIES=n) &
   + SPREAD((/(j,j=1,n)/),DIM=1,NCOPIES=n) - 1 )
```

Note that the FORALL statement does not imply the creation of temporary arrays and is much more readable.

•

```
FORALL (i=1:n, j=1:n, y(i,j).NE.0.0) &
              x(i,j) = 1.0 / y(i,j)
```

This statement takes the reciprocal of each nonzero element of array $y(1:n,1:n)$ and assigns it to the corresponding element of array x. Elements of y that are zero do not have their reciprocal taken, and no assignments are made to the corresponding elements of x. This is equivalent to the standard Fortran 90 statement

```
WHERE (y(1:n,1:n) .NE. 0.0) &
              x(1:n,1:n) = 1 / y(1:n,1:n)
```

- `FORALL (i=2:4) x(i) = x(i-1) + x(i) + x(i+1)`

  If this statement is executed with

  $$x = [1.0, 20.0, 300.0, 4000.0, 50000.0]$$

  then after execution the new values of array x will be

  $$x = [1.0, 321.0, 4320.0, 54300.0, 50000.0]$$

  This has the same effect as the Fortran 90 statement

```
x(2:4) = x(1:3) + x(2:4) + x(3:5)
```

  Note that it does *not* have the same effect as the Fortran 90 loop

```
DO i = 2, 4
   x(i) = x(i-1) + x(i) + x(i+1)
END DO
```


- `FORALL (i=1:n) a(i,i) = x(i)`

  This `FORALL` statement sets the elements of the main diagonal of matrix a to the elements of vector x. This cannot be done by an array assignment in Fortran 90 unless `EQUIVALENCE` or `WHERE` is also used.

- `FORALL (i=1:4) a(i,ix(i)) = x(i)`

  This `FORALL` statement sets one element in each row of matrix a to an element of vector x. The particular elements in a are chosen by the integer vector ix. If

  $$x = [10.0, 20.0, 30.0, 40.0]$$
  $$ix = [1, 2, 2, 4]$$

  and array a represents the matrix

  $$\begin{matrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \end{matrix}$$

  before execution of the `FORALL`, then a will represent

  $$\begin{matrix} 10.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 20.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 30.0 & 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 40.0 \end{matrix}$$

  after its execution. This operation cannot be accomplished with a single array assignment in Fortran 90.

- `FORALL (k=1:9) x(k) = SUM(x(1:10:k))`

  This `FORALL` statement computes nine sums of subarrays of `x`. If before the `FORALL`

  $$x = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]$$

  then after the `FORALL`

  $$x = [55.0, 25.0, 22.0, 15.0, 7.0, 8.0, 9.0, 10.0, 11.0, 10.0]$$

  This computation cannot be done by Fortran 90 array expressions alone.

- Interprétation du `FORALL`
  1. Évaluation dans un ordre quelconque de la liste de triplets
  2. Évaluation du *scalar-mask*
  3. Évaluation de la partie droite de l'affectation (pas d'ordre)
  4. Rangement dans la partie gauche (pas d'ordre)
- « Scalarisation » du `FORALL`
  - le code scalaire est « équivalent » à une boucle `FORALL`
  - c'est à partir de ce code que l'on devra tirer le parallélisme pour une machine parallèle

- Une instruction FORALL de la forme

```
FORALL(v1=l1:u1:s1, v2=l2:u2:s2, ...vn=ln:un:sn, mask)
   a(e1,..em) = rhs
```

est équivalente à

```
! Evaluate subscript and stride expressions.
! These assignments may be executed in any order.
templ1 = l1
tempu1 = u1
temps1 = s1
templ2 = l2
tempu2 = u2
temps2 = s2
  ...
templn = ln
tempun = un
tempsn = sn
```



```
! Evaluate the scalar mask expression, and evaluate the
! forall-assignment subexpressions where the mask is true.
! The iterations of this loop nest may be executed in any order.
! The assignments in the loop body may be executed in any order,
! provided that the mask element is evaluated before any other
! expression in the same iteration.
! The loop body need not be executed atomically.
! The DO statements may be nested in any order
DO v1=templ1,tempu1,temps1
  DO v2=templ2,tempu2,temps2
    ...
      DO vn=templn,tempun,tempsn
        tempmask(v1,v2,...,vn) = mask
        IF (tempmask(v1,v2,...,vn)) THEN
          temprhs(v1,v2,...,vn) = rhs
          tempe1(v1,v2,...,vn) = e1
          tempe2(v1,v2,...,vn) = e2
          ....
          tempem(v1,v2,...,vn) = em
        END IF
      END DO
     ...
  END DO
END DO

! Perform the assignment of these values to the corresponding
! elements of the array on the left-hand side.
! The iterations of this loop nest may be executed in any order.
! The DO statements may be nested in any order.
DO v1=templ1,tempu1,temps1
  DO v2=templ2,tempu2,temps2
    ...
      DO vn=templn,tempun,tempsn
        IF (tempmask(v1,v2,...,vn)) THEN
          a(tempe1(v1,v2,...,vn),...,tempem(v1,v2,...,vn)) =    &
            temprhs(v1,v2,...,vn)
        END IF
      END DO
    ...
  END DO
END DO
```


Constructeur FORALL

- Généralisation de l'instruction FORALL
- Plusieurs instructions d'affectation et une seule liste de triplets, un seul masque
- Syntaxe

FORALL (*<triplet-list>*, [*<scalar-mask>*])
    <forall-stmt-list>
END FORALL

<forall-stmt-list> ::= *<assignment>*
    <where>
    <forall>

- Exemple

```
FORALL (I=2:N-1, J=2:N-1)
   A(I,J) = A(I,J-1) + A(I,J+1)     &
          + A(I+1,J) + A(I-1,J)
   B(I,J) = A(I,J)
END FORALL
```


- Exemple

```
FORALL (I=1:N-1)
   FORALL (J=I+1:N)
     A(I,J) = A(J,I)
   END FORALL
END FORALL
```

- Une instruction de la forme

```
FORALL (... e1 ... e2 ... en)
   s1
   s2
   ...
   sn
END FORALL
```

(si : <*assignment*> ou <*forall*>, mais pas <*where*>)
est équivalente à

```
temp1 = e1
temp2 = e2
...
tempn = en
FORALL (... temp1 ... temp2 ... tempn) s1
FORALL (... temp1 ... temp2 ... tempn) s2
...
FORALL (... temp1 ... temp2 ... tempn) sn
```


Exemples de constructions FORALL

```
• FORALL ( i=2:n-1, j=2:n-1 )
    a(i,j) = a(i,j-1) + a(i,j+1) &
           + a(i-1,j) + a(i+1,j)
    b(i,j) = a(i,j)
  END FORALL
```

This FORALL is equivalent to the two Fortran 90 statements

```
a(2:n-1,2:n-1) = a(2:n-1,1:n-2) + a(2:n-1,3:n) &
               + a(1:n-2,2:n-1) + a(3:n,2:n-1)
b(2:n-1,2:n-1) = a(2:n-1,2:n-1)
```

In particular, note that the assignment to array b uses the values of array a computed in the first statement, not the values before the FORALL began execution.


```
• FORALL ( i=1:n-1 )
    FORALL ( j=i+1:n )
      a(i,j) = a(j,i)
    END FORALL
  END FORALL
```

This FORALL construct assigns the transpose of the lower triangle of array a (i.e., the section below the main diagonal) to the upper triangle of a. For example, if $n = 5$ and a originally contained the matrix

$$\begin{array}{rrrrr} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 4.0 & 8.0 & 16.0 & 32.0 \\ 3.0 & 9.0 & 27.0 & 81.0 & 243.0 \\ 4.0 & 16.0 & 64.0 & 256.0 & 1024.0 \end{array}$$

then after the FORALL it would contain

$$\begin{array}{rrrrr} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 1.0 & 4.0 & 9.0 & 16.0 \\ 2.0 & 4.0 & 8.0 & 27.0 & 64.0 \\ 3.0 & 9.0 & 27.0 & 81.0 & 256.0 \\ 4.0 & 16.0 & 64.0 & 256.0 & 1024.0 \end{array}$$

This cannot be done using array expressions without introducing mask expressions. (sic !)

```
DO 10 I = 1, N
   A(I+1:N,I) = A(I,I+1:N)
```


- Dépendance au sein du FORALL

```
REAL V (3,10,10)
FORALL (I=1:10,J=1:10)
   V(:,I,J) = f(V(:,I,J))
END FORALL
```

- La fonction `f` ne doit pas modifier les éléments de `V`
  Plus généralement, cette fonction ne doit pas avoir d'effets de bords
- HPF introduit donc une directive « assurant » une telle propriété

```
      INTERFACE
         FUNCTION F(X)
!HPF$    PURE F
         REAL, DIMENSION (3) :: F, X
         END FUNCTION F
      END INTERFACE

      REAL V (3,10,10)
      FORALL (I=1:10,J=1:10)
         V(:,I,J) = F(V(:,I,J))
      END FORALL
```

Directive INDEPENDENT

- Directive précédant une boucle DO ou FORALL ou un nid de boucles DO ou FORALL
- Intuitivement : indique que les opérations contenues dans la construction DO/FORALL peuvent être exécutées indépendamment
- Exemple

```
!HPF$ INDEPENDENT
      DO I=1, 100
         A(P(I)) = B(I)
      END DO
```

$\Rightarrow$ il n'existe pas de doublets dans P, et A et B ne partagent pas un même emplacement mémoire

- Exemple

```
!HPF$ INDEPENDENT (I1, I2, I3)
      DO I1=1, N1
         DO I2= 1, N2
            DO I3=1, N3
               DO I4 = 1, N4 ! NON INDEPENDENT !
                  A(I1,I2,I3) = A(I1,I2,I3)    &
                              + B(I1,I2,I4)    &
                              + C(I1,I2,I4)
               END DO
            END DO
         END DO
      END DO
```


- Exemple

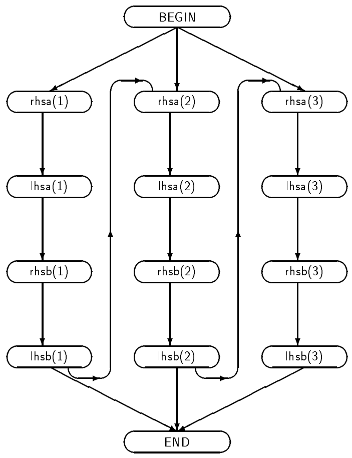
```
!HPF$ INDEPENDENT, NEW (J)
      DO I = 2, 100, 2
!HPF$ INDEPENDENT, NEW(VL, VR, UL, UR)
         DO J = 2 , 100, 2
            VL = P(I,J) - P(I-1,J)
            VR = P(I+1,J) - P(I,J)
            UL = P(I,J) - P(I,J-1)
            UR = P(I,J+1) - P(I,J)
            P(I,J) = F(I,J) + P(I,J) +    &
                     0.25 * (VR-VL+UR-UL)
         END DO
      END DO
```

- Sans l'option NEW, les itérations ne seraient pas indépendantes, car les valeurs de VR, VL, UR, UL d'une itération pourraient être utilisées lors d'une autre itération
- Avec le NEW, on spécifie que les boucles sont indépendantes si l'on utilise une NOUVELLE variable VR, VL, UR, UL pour chaque itération

- Soit le code

```
DO i = 1, 3
   lhsa(i) = rhsa(i)
   lhsb(i) = rhsb(i)
END DO
```

On a le graphe de précédence

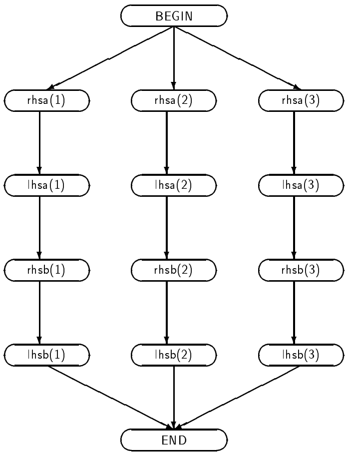


- Une $L \rightarrow R$ indique, par exemple, que R peut utiliser des valeurs produites par L et donc que l'exécution de R ne peut se faire avant la terminaison de celle de L

- En présence d'une directive `INDEPENDENT`

```
!HPF$ INDEPENDENT
      DO i = 1, 3
         lhsa(i) = rhsa(i)
         lhsb(i) = rhsb(i)
      END DO
```

Les dépendances sont

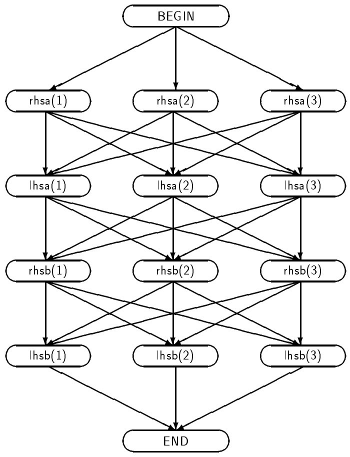


- Les différentes itérations peuvent être exécutées en parallèle

- Pour la boucle `FORALL`

```
FORALL ( i = 1:3 )
   lhsa(i) = rhsa(i)
   lhsb(i) = rhsb(i)
END FORALL
```

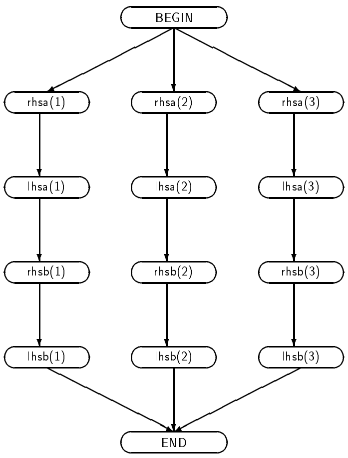
On a



- La boucle `FORALL` avec `INDEPENDENT`

```
!HPF$ INDEPENDENT
      FORALL ( i = 1:3 )
         lhsa(i) = rhsa(i)
         lhsb(i) = rhsb(i)
      END FORALL
```

donne



Nouvelles procédures intrinsèques

- HPF reprend les nombreuses fonctions intrinsèques de Fortran 90
- HPF ajoute
  - des nouvelles classes de fonctions
    - ◇ « System inquiry intrinsic functions »
    - ◇ « Mapping inquiry intrinsic function »
  - de nouvelles fonctions de calcul
  - une nouvelle bibliothèque
  - et étend les fonction `MINLOC` et `MAXLOC`

System inquiry intrinsic functions

- État du système (de l'architecture physique)
- Constant pour toute la durée du programme
- `NUMBER_OF_PROCESSORS()`
  - retourne le nombre de processeurs de la machine
  - accepte un paramètre optionnel `DIM` dont la valeur $1 \leq$ `DIM` $\leq n$ avec n, rank de la machine physique
  - exemple, DECmpp 12000, 8K PEs
    `NUMBER_OF_PROCESSORS()` $\rightarrow$ 8192
    `NUMBER_OF_PROCESSORS(DIM=1)` $\rightarrow$ 128
    `NUMBER_OF_PROCESSORS(2)` $\rightarrow$ 64
  - exemple, DECstation alpha
    `NUMBER_OF_PROCESSORS()` $\rightarrow$ 1
    `NUMBER_OF_PROCESSORS(DIM=1)` $\rightarrow$ 1

Le parallélisme de données en Fortran High Performance Fortran

- `PROCESSORS_SHAPE()`
  - retourne la shape de la machine
  - exemple, DECmpp 12000, 8K PEs
    `PROCESSOR_SHAPE()` $\rightarrow$ (/128,64/)
  - exemple, CM-2, 8K PEs
    `PROCESSOR_SHAPE()` $\rightarrow$ (/2,2,2,2,2,2,2,2,2,2,2,2,2/)
  - exemple, CM-5, 512 PEs
    `PROCESSOR_SHAPE()` $\rightarrow$ (/512/)
  - exemple, DECstation alpha
    `PROCESSOR_SHAPE()` $\rightarrow$ (/1/)
- Bien que ces fonctions retournent une valeur qui peut ne pas être connue à la compilation, elles peuvent apparaître dans des directives HPF

```
!HPF$ template t(100,3*NUMBER_OF_PROCESSORS())
```


Le parallélisme de données en Fortran High Performance Fortran

Extension de `MINLOC` et `MAXLOC`

- Retourne la position du plus petit (ou du plus grand) élément d'un tableau
- Ajout d'un paramètre optionnel `DIM`
- Exemple

$$\mathtt{A} = \begin{pmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 0 & 4 & 6 & -4 \end{pmatrix}$$

`MINLOC(A)`	$\rightarrow$	(/1,2/)	(le -5)
`MAXLOC(A)`	$\rightarrow$	(/1,3/)	(le 8)
`MINLOC(A,DIM=1)`	$\rightarrow$	(/1,1,2,3/)	(les 0,-5,-1,-4)
`MINLOC(A,DIM=2)`	$\rightarrow$	(/2,3,4/)	(les -5,-1,-4)

- Il existe aussi un paramètre optionnel `MASK`

Le parallélisme de données en Fortran High Performance Fortran

Bibliothèque

- Ce ne sont pas des fonctions intrinsèques

  (⇒
  utilisation `USE`
  pas de valeur à la compilation
)

- Fonctions de réduction

Fortran 90	+	SUM, COUNT
	*	PRODUCT
	.AND.	ALL
	.OR.	ANY
	MAX	MAXVAL
	MIN	MINVAL
HPF	IAND	AND[a]
	IOR	OR[a]
	IEOR	EOR[a]
	.NEQV.	PARITY

[a] Opération bit à bit

```
LOGICAL   T, F
PARAMETER (T=.TRUE., F=.FALSE.)  ! MACROS
PARITY (/T,F,F,T,T,T,F,F/)       ! -> .TRUE.
PARITY (/T,F,F,T,T,T,F,T/)       ! -> .FALSE.
```

Fonctions « scatter »

- Pour chaque opération XXX du langage, HPF fournit une fonction

```
XXX_SCATTER (SOURCE, BASE, IDX1, ...IDXn, M
```

- Les XXX sont

```
SUM    COUNT  PRODUCT
MAXVAL MINVAL AND     OR
EOR    ALL    ANY     PARITY
```

- Les IDX sont des tableaux d'index
  n, nombre de IDX, est le rank de BASE
  rank des IDX = rank de SOURCE
- Le résultat est de la forme de BASE
  du type de SOURCE et BASE
- Pour chaque s dans SOURCE, l'ensemble des éléments correspondants dans les IDX est un vecteur $(i_1, i_2, ...i_n)$

  On considère BASE$(i_1, i_2, ...i_n)$

  On lui associe tous les s de SOURCE qui lui correspondent : $s_1, s_2, ...s_m$

  Le résultat est $s_1 \otimes s_2 \otimes ...s_n \otimes b$

  ($\otimes$: opérateur XXX de la fonction)

- Le MASK détermine éventuellement les seuls éléments de SOURCE qui participent à l'opération
- SUM_SCATTER(SOURCE,IDX1,IDX2... IDXn MASK) peut être calculé par

```
result = BASE
DO i1 = LBOUND(SOURCE,1), UBOUND(SOURCE,1)
DO i2 = LBOUND(SOURCE,2), UBOUND(SOURCE,2)
...
DO ik = LBOUND(SOURCE,k), UBOUND(SOURCE,k)
   IF (MASK(i1,i2,... ik))
      result (IDX1(i1,i2,... ik),
              IDX2(i1,i2,... ik),
              ...
              IDXn(i1,i2,... ik))
      = result (IDX1(i1,i2,... ik),
              IDX2(i1,i2,... ik),
              ...
              IDXn(i1,i2,... ik))
      + SOURCE (i1,i2,... ik)
   END IF
END DO
END DO
...
END DO
```


- Exemple (enfin !)

```
A = (/10, 20, 30, 40, -10/)
V = (/3,  2,  2,  1,  1  /)
BASE = (/1,2,3,4/)

BASE = SUM_SCATTER (A, BASE, V, MASK=(A>0))
```

donne

```
BASE =    1   2   3  4
     +           10
     +       20
     +       30
     +   40
     = (/41, 52, 13, 4/)
```

équivalent à

```
BASE(V) = BASE(V) + MERGE (A,0.,A>0)
```

si V ne contient pas de doublons

Parallel prefix

- Pour chaque opération de réduction du langage
  - XXX_PREFIX
  - XXX_SUFFIX

```
XXX_YYYFIX (ARRAY [,DIM] [,MASK]
               [,SEGMENT] [,EXCLUSIVE])
```

- Calcul, par exemple, les sommes partielles d'un tableau
- Les XXX sont

```
SUM    COUNT  PRODUCT
MAXVAL MINVAL AND     OR
EOR    ALL    ANY     PARITY
```

- Exemple

A $\begin{pmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 0 & 4 & 6 & -4 \end{pmatrix}$

SUM_PREFIX(A) $\begin{pmatrix} 0 & -2 & 14 & 16 \\ 3 & 2 & 13 & 18 \\ 3 & 6 & 19 & 14 \end{pmatrix}$

SUM_PREFIX(A,DIM=1) $\begin{pmatrix} 0 & -5 & 8 & -3 \\ 3 & -1 & 7 & -1 \\ 3 & 3 & 13 & -5 \end{pmatrix}$

- Exemple MASK

```
MAXVAL_PREFIX(  (/3, 4, 5, 6, 7/),
          MASK=(/T, F, F, T, F/))
```

donne

```
                (/3, 3, 3, 6, 6/)
```

en fait,le résultat n'est souvent pas nécessaire pour les valeurs fausses du MASK :

```
WHERE (FOO) A = SUM_PREFIX (B,MASK=FOO)
```


- SEGMENT : scan par segment

  un segment est une suite d'éléments successifs dont la valeur de SEGMENT est constante (.TRUE. ou .FALSE.)
- EXCLUSIVE est une valeur scalaire LOGICAL indiquant si le scan inclut ou non l'élément
  - inclusive-sum(/1,2,3,4/) $\rightarrow$ (/1,3,6,10/)
  - exclusive-sum(/1,2,3,4/) $\rightarrow$ (/0,1,3,6/)
- SUFFIX $\Rightarrow$ opération depuis les indices les plus grands

Fonctions de tri

- Deux fonctions :

```
GRADE_UP (ARRAY, DIM)
GRADE_DOWN(ARRAY, DIM)
```

- Elles retournent un vecteur d'indices
- À partir de ce vecteur d'indices, on peut faire une permutation de éléments du vecteur pour obtenir le vecteur trié

Fonctions de manipulations de bits

- ...

Mapping inquiry intrinsic functions

- Les directives de « mapping » d'HPF ne sont pas des « ordres », mais des « indications »

  Le compilateur (un compilateur) HPF peut ne pas prendre en compte certaines distributions ou alignements
- Le « mapping » des tableaux est dynamique
- Le programmeur peut savoir exactement comment un tableau est « mappé » par l'appel de ces fonctions d'« inquiry »
- C'est particulièrement important pour l'interface avec d'autres langages

- Alignment inquiry

  CALL HPF_ALIGNMENT (

  ARRAY le seul input

  LB tableau (7) contenant l'indice dans le template du 1[er] élément de ARRAY sur un axe

  UB tableau (7) ... dernier élément ...

  STRIDE tableau (7) ... stride élément dans le template ...

  AXIS_MAP tableau (7), le n° d'axe du template associé à l'axe. 0 ⇒ « collapsed »

  IDENTITY_MAP logical, true ⇒ le tableau est de même shape que le template, les axes sont mappés de la même manière et les strides sont égaux à 1

  DYNAMIC logical, le tableau est dynamique

  NUMBER_OF_COPIES entier, le produit des tailles des axes du template sur lequel le tableau est répliqué

)

- Exemple

```
      DIMENSION A(10,10), B(20,20), C(20,40,10)
!HPF$ TEMPLATE T(40,20)
!HPF$ DYNAMIC A
!HPF$ ALIGN A(I,:) WITH T(1+3*I, 2:20:2)
!HPF$ ALIGN C(I,*,J) WITH T(J, 21-I)
!HPF$ PROCESSORS PROCS (4,2)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE B(CYCLIC, BLOCK) ONTO PROCS
```

- HPF_ALIGNMENT donne

	A	B	C
LB	4,2,..	1,1,..	1,–,1,..
UB	31,20,..	20,20,..	20,–,10,..
STRIDE	3,2,..	1,1,..	-1,–,1,..
AXIS_MAP	1,2,..	1,2,..	2,0,1,..
IDENTITY_MAP	.FALSE.	.TRUE.	.FALSE
DYNAMIC	.TRUE.	.FALSE.	.FALSE
NUMBER_OF_COPIES	1	1	1

- Distribution inquiry

  CALL HPF_DISTRIBUTION (

  ARRAY tableau dont on veut l'info sur le template

  AXIS_TYPE tableau de caractères, distribution de chaque dimension du template. "BLOCK", "CYCLIC", "COLLAPSED"

  AXIS_INFO tableau d'entiers.
  AXIS_TYPE : BLOCK ou CYCLIC → taille des blocs

  PROCESSORS_RANK tableau d'entiers, le rank de l'arrangement de processeurs associé au template

  PROCESSORS_SHAPE la shape...

)

- Exemple

	A	B
AXIS_TYPE	"BLOCK", "BLOCK",..	"CYCLIC", "BLOCK",..
AXIS_INFO	10, 10, ..	5, 10, ..
PROCESSORS_RANK	4, 2, ..	4, 2, ..
PROCESSORS_SHAPE	2	2

- Template inquiry

  CALL HPF_TEMPLATE (

  ARRAY le tableau pour lequel les informations sur le template seront retournées (c'est là la différence avec HPF_ALIGNMENT)

  LB, UB tableaux contenant les lower et upper bound déclarées pour le template

  AXIS_TYPE tableau de tableau de caractères, info sur chacun des axes du template
  - "NORMAL" l'axe est aligné à l'axe d'un tableau. AXIS_INFO est le n° de cet axe
  - "SINGLE" le tableau est aligné avec une seule coordonnées de l'axe du template. AXIS_INFO contient cette coordonnée
  - "REPLICATED" le tableau est dupliqué le long de cet axe. AXIS_INFO contient le nombre de copies

  AXIS_INFO tableau d'entiers

  NUMBER_ALIGNED nombre total de tableaux alignés avec le template, c'est le nombre de tableau qui devront être « déplacés » si le template est redistribué

  DYNAMIC logical

  TEMPLATE_RANK nombre d'axes du template

- Exemple

```
      REAL A(4,20)
!HPF$ TEMPLATE T(30, 150, 8, 200)
!HPF$ ALIGN A(I,J) WITH T(J+5,100,110-2*I,*)
```

AXIS_TYPE → "NORMAL", "SINGLE", "NORMAL", "REPLICATED"
AXIS_INFO → 2, 100, 1, 200

- Ces fonctions d'inquiry mapping dévoilent une partie de l'implementation !!

Fonctions locales

- L'idée est la suivante :
  - on considère un ensemble de tableaux, paramètre d'une fonction
  - la distribution/alignement de ces tableaux est tel que chaque processeur contient un certains nombre d'éléments (peut-être différent) de ces tableaux
  - une fonction est dite « locale » si le calcul de la fonction sur un processeur donné ne nécessite que la donnée des éléments se trouvant sur ce processeur
  - on peut de cette caractéristique définir une exécution efficace : les exécutions des différents processeurs sont indépendantes
  - ces fonctions locales sont écrites en « pure » Fortran 90
- Exemple : réalisation de la somme des éléments d'un tableau

- Code de l'appelant

```
      INTERFACE
         FUNCTION SREDUCE (A)
!HPF$       EXTRINSIC
            REAL, DIMENSION(NUMBER_OF_PROCESSORS()) :: SREDUCE
!HPF$       DISTRIBUTE (BLOCK) :: SREDUCE
            REAL, DIMENSION(:), INTENT(IN) :: A
         END FUNCTION SREDUCE
      END INTERFACE
      ...
      TOTAL = SUM(SREDUCE(A))
```

- Code de la fonction SREDUCE

```
      SUBROUTINE SREDUCE (AA)
!HPF$    LOCAL
         REAL AA(:)
         INTEGER COPIES

         CALL GLOBAL_ALIGNMENT (AA,NUMBER_OF_COPIES=COPIES)
         IF (COPIES .GT. 1)
            THEN
               ! LE TABLEAU EST DUPLIQUE : ERREUR
               CALL ERROR ()
            ELSE
               ! CALCUL DELA SOMME LOCALE
               RETURN (SUM(AA))
         END IF
      END
```

GLOBAL_ALIGNMENT → HPF_ALIGNMENTpour les fonctions locales

Entrées/sorties parallèles

- Lors de la conception d'HPF, un groupe de travail était chargé de mettre au point une extension des entrées/sorties de Fortran 90
- Ce groupe a abouti à la conclusion suivante : il n'y aura pas d'extension des I/O de Fortran 90 dans HPF !
- Cependant
  - une machine massivement parallèle a besoin d'I/O massivement parallèle
  - un programme efficace doit éviter le goulot d'étranglement des processeurs vers le système de fichiers
  - Fortran spécifie que les éléments sont rangés dans l'ordre dans un fichier. Cela est conflictuel avec les alignements/distributions

- 3 approches avaient été envisagées
  1. Associer une distribution aux données lors de leur écriture, cette distribution devra être semblable lors de la lecture
  2. Associer un « FILE TEMPLATE» lors de l'ouverture d'un fichier (un fichier est vu comme un tableau 1-D d'enregistrements)
  3. Définir des I/O parallèles indépendantes des I/O séquentielles

- Pourquoi l'abandon d'I/O spécifiques?
  - les architectures d'I/O parallèles sont trop différentes d'une machine à l'autre. Par exemple :
    - ◇ disque sur chaque nœud, ou
    - ◇ disque rapide accessible globalement via le réseau
  - Fortran 90 a déjà des instructions d'I/O (trop?) complexes et performantes en terme d'expressivité du langage
  - le compilateur HPF sait déjà quand il réalise des I/O sur des tableaux distribués et peut optimiser ces opérations sans extensions du langage
  - la gestion de fichier distribués (ou parallèles), et leur implémentation) incombe au système d'exploitation et non au langage

Modèles d'allocation mémoire

- Le problème est le suivant

  Fortran 77 et Fortran 90 supposent un modèle de mémoire linéaire dans laquelle sont alloués les variables (scalaires et tableaux)

  Ce modèle est mal adapté au mapping des variables tel qu'il est introduit en HPF

  Cela entraîne quelques limitations (attention aux portages de codes Fortran 77/90 vers HPF !!)
- Deux points sont à considérer
  - sequence association
  - storage association

Storage association

- Le modèle de mémoire est une mémoire linéaire et adressée linéairement
- `EQUIVALENCE` $\rightarrow$ partage de cette mémoire
- `COMMON` $\rightarrow$ partage de cette mémoire
- HPF impose des contraintes pour la distribution et l'alignement de variable appartenant à des `EQUIVALENCE` ou `COMMON`
- Attention aussi au passage de tableaux en paramètre

  ```
  FOO (A(10))
  ```

  en Fortran 77/90 $\Rightarrow$ tableau commençant à l'élément 10 de `A` !

Sequence association

- C'est le rangement des tableaux multi-dimensionnels dans la mémoire linéaire
- Attention

  On peut en Fortran 77/90 déclarer un tableau avec un certain nombre de dimensions et passer ce tableau à une fonction qui attend une tableau avec un autre nombre de dimensions

  (C'est écrit dans la norme !)

Fortran 77/90 vers HPF

- HPF propose un certain nombre de spécifications pour permettre au programmeur d'adapter les programmes Fortran 77/90 en indiquant si la « storage association » et la « sequence association » sont respectées ou non pour un tableau

Restrictions pour une première implémentation

- Afin de ne pas décourager les gens développant les compilateurs, HPF intègre dans sa définition un sous-ensemble de ses constructions
- L'idée et que les directives sont là pour aider le compilateur à réaliser certaines optimisations et que ces directives peuvent être ignorées !
- De base, HPF inclut aussi un `FORALL` (à partir duquel il n'est pas facile de générer du code parallèle !)

Conclusion

- Évolution du standard Fortran 90 pour les machines parallèles SIMD et MIMD
- Parallélisme de données
- Notion de distribution des données
- Reprise de différents langages :
  - Fortran 90
  - MP Fortran (DEC/MasPar)
  - Connection Machine Fortran (TMC)
  - Fortran D (Rice university)
  - Vienna Fortran (Vienna university)
- HPF-2
- Compilation ?
- Performances ?