Calcul Parallèle
2017-2018

Abdeljalil Nachaoui

Types dérivés:

variables MPI définies par l'utilisateur

Déclarer un type dérivé

Les fonctions de communication MPI utilise comme argument des types MPI standard comme MPI_INTEGER ou MPI_REAL. Il est possible de définir des types plus complexes. Cela peut servir, par exemple:

- A envoyer une section de tableau.
- A envoyer une variable de type structure définie dans le programme.

Exemple de déclaration d'un type dérivé simple:

```
INTEGER :: err,MPI_vector
call MPI_TYPE_CONTIGUOUS(3,MPI_REAL,MPI_vector,err)
call MPI_TYPE_COMMIT(MPI_vector,err)
... communications ...
call MPI_TYPE_FREE(MPI_vector,err)
```

On définit ici un type **MPI_vector**, constitué de **3 MPI_REAL** stockés consécutivement en mémoire. MPI_vector peut servir à définir d'autres types dérivés.

Il faut "compiler" le type avec **MPI_TYPE_COMMIT** avant de pouvoir l'utiliser dans une communication.

Types de données dérivés : introduction

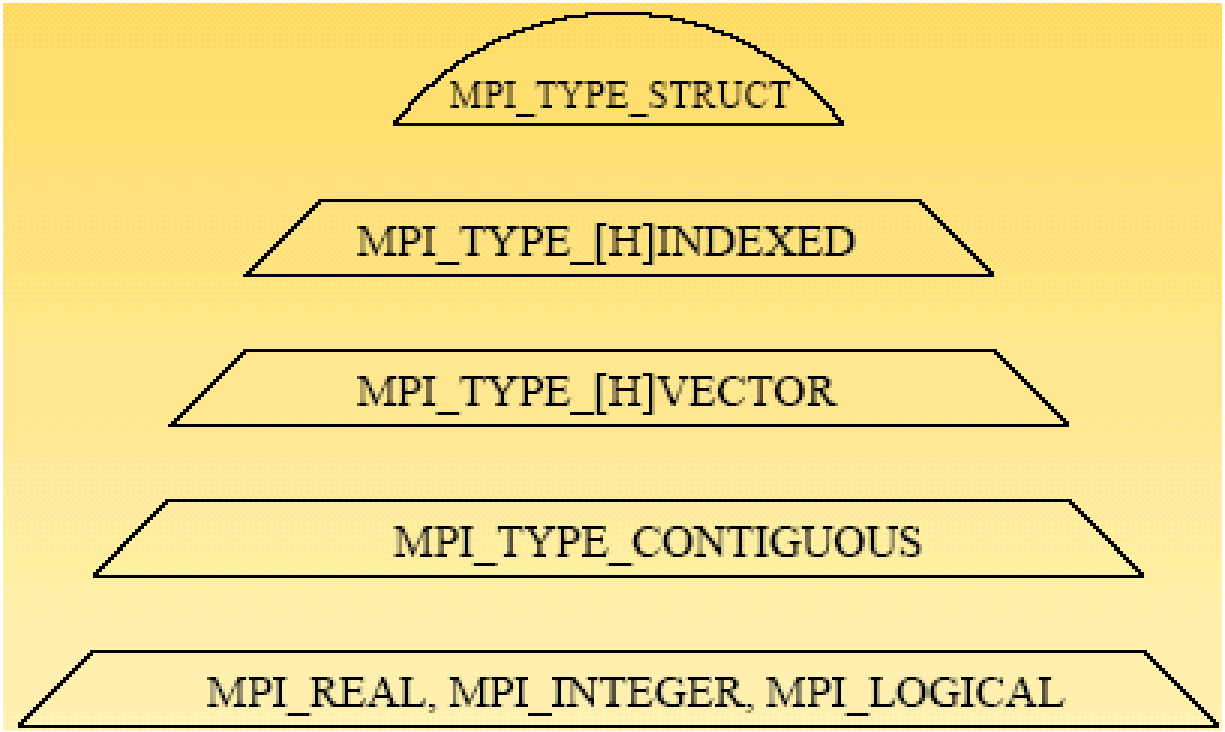
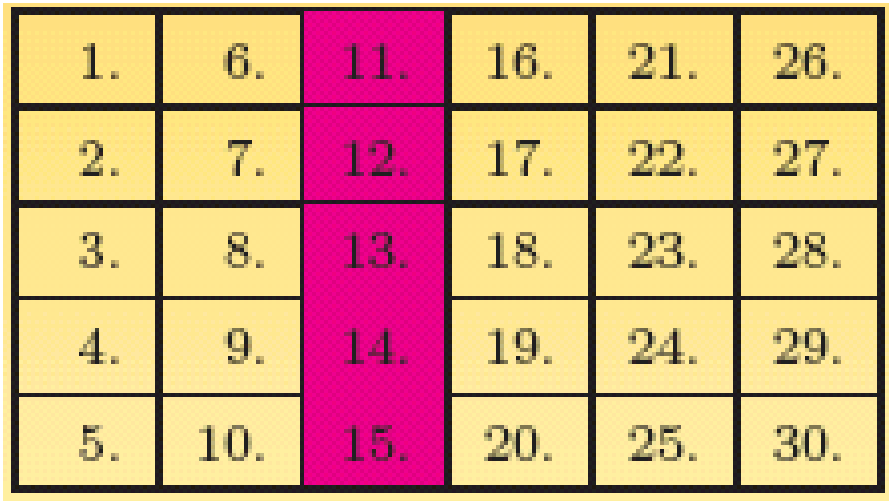


Fig. – Hiérarchie des constructeurs de type MPI

Types de données dérivés : contigus

TYPE CONTIGUOUS() crée une structure de données à partir d'un ensemble homogène de type prédéfini de données contiguës en mémoire.

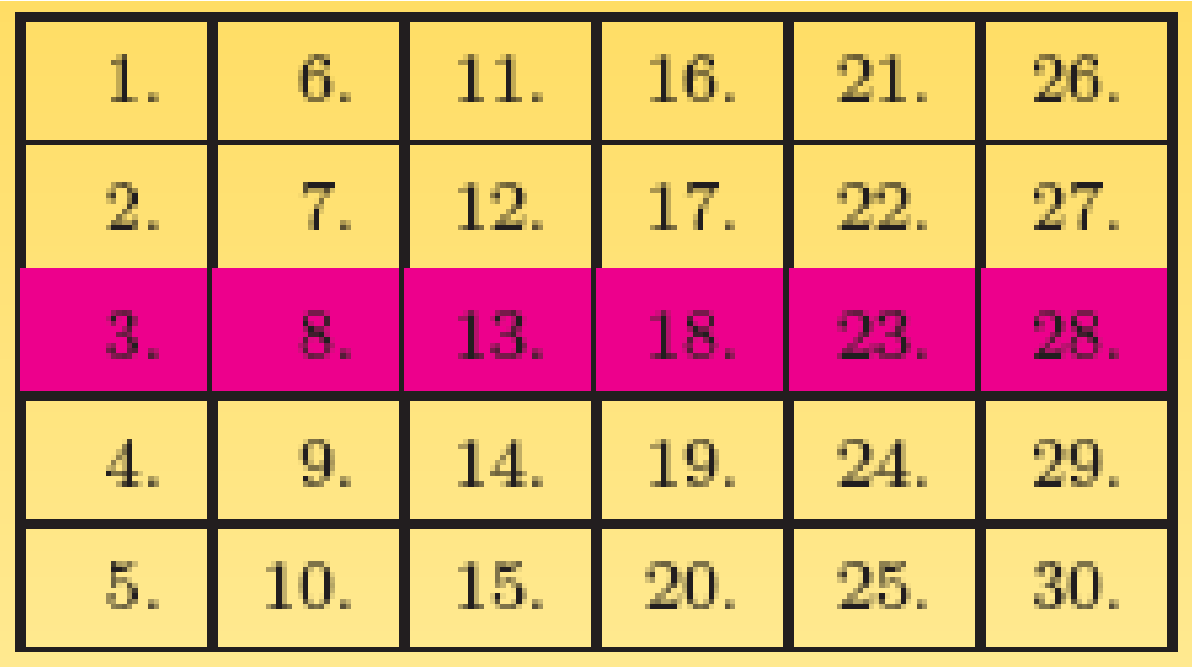


1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

call MPI_TYPE_CONTIGUOUS (5, MPI_REAL ,nouveau_type,code)

```
integer, intent(in) :: nombre, ancien_type
integer, intent(out) :: nouveau_type,code
call MPI_TYPE_CONTIGUOUS (nombre,ancien_type,nouveau_type,code)
```

Types de données dérivés :vecteur



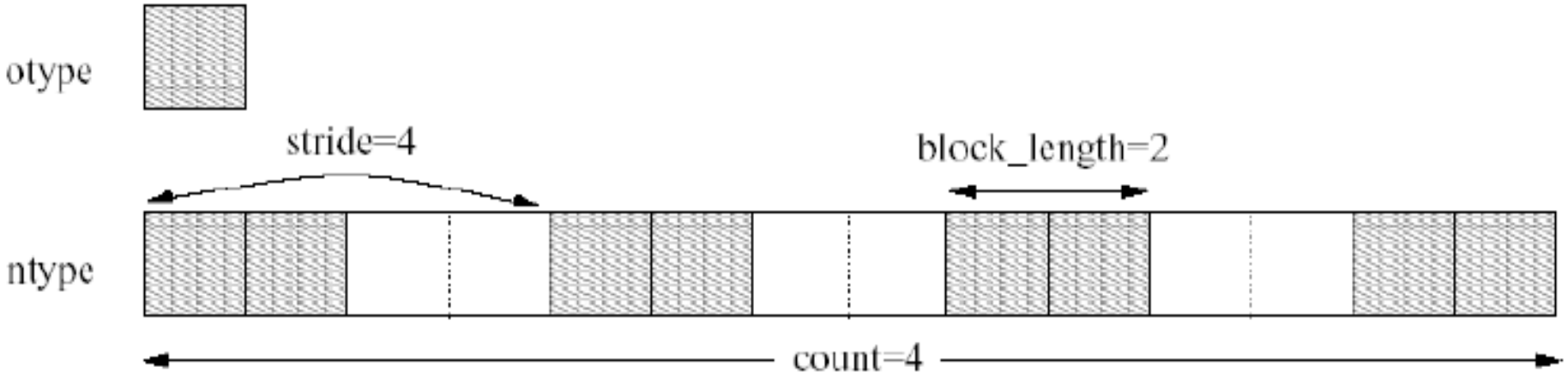
1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

call MPI_TYPE_VECTOR (6,1,5, MPI_REAL ,nouveau_type,code)

Fig. – Sous-programme MPI TYPE VECTOR()

Types de données dérivés : vecteur

```
MPI_Type_vector(count,block_length,stride, type,ntype,ierr)
   INTEGER, INTENT(IN)  :: count        ! Nombre de blocs
   INTEGER, INTENT(IN)  :: block_length ! Longueur d'un blocs
   INTEGER, INTENT(IN)  :: stride       ! Pas
   INTEGER, INTENT(IN)  :: otype        ! Ancien type
   INTEGER, INTENT(OUT):: ntype         ! Nouveau type
```



call MPI_TYPE_VECTOR (4,2,4, MPI_REAL ,ntype,code)

Types de données dérivés : hvecteur

MPI_TYPE _HVECTOR() crée une structure de données à partir d'un ensemble homogène de type prédéfini de données distantes d'un pas constant en mémoire.

Le pas est donné en nombre d'octets.

Cette instruction est utile lorsque le type générique n'est plus un type de base (MPI INTEGER, MPI REAL,...) mais un type plus complexe construit à l'aide des sous-programmes MPI vus précédemment

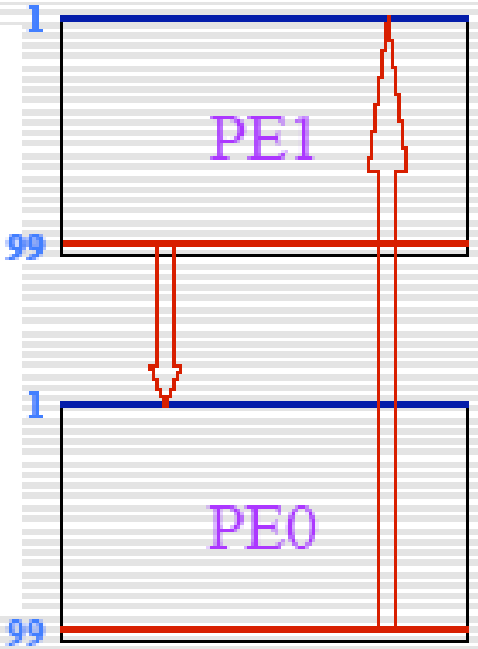
Le pas ne peut plus alors être exprimé en nombre d'éléments du type générique.

Exemple 1

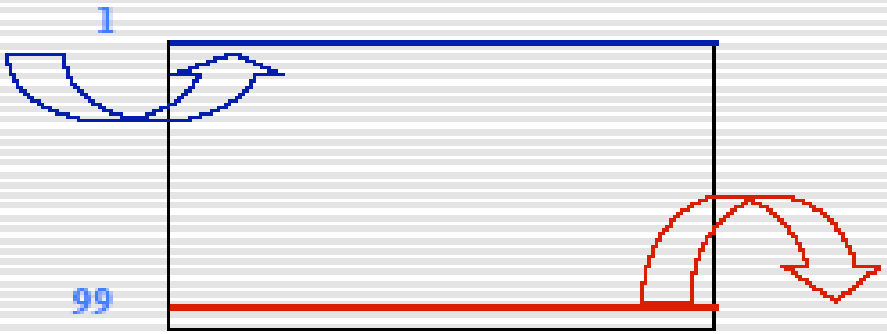
```
CHARACTER*256 :: class
INTEGER :: n
REAL :: a(100)
...
CALL MPI_BCAST(class, 256, MPI_CHAR, 0, comm, ierr)
CALL MPI_BCAST(n, 1, MPI_INTEGER, 0, comm, ierr)
CALL MPI_BCAST(a, 100, MPI_REAL, 0, comm, ierr)
```

Exemple 2

```
REAL :: a(100,100), buf1(100), buf2(100)
...
voisin = 1
IF( me.eq.1 ) voisin = 0
DO j=1,100
  buf1(j) = a(99,j)
END DO
CALL MPI_SENDRECV(buf1, 100, MPI_REAL, voisin, 0, &
   &              buf2, 100, MPI_REAL, voisin, 0, &
   &              comm, status, ierr)
DO j=1,100
  a(1,j) = buf2(j)
END DO
```



Exemple 2 avec **MPI_Type_vector**



REAL, **dimension(100,100) :: a**

INTEGER :: type_ligne

```
...
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, type_ligne, ierr)
CALL MPI_TYPE_COMMIT(type_ligne, ierr)
CALL MPI_SENDRECV(a(99,1), 1, type_ligne, voisin, 0, &
    &                a(1,1),  1, type_ligne, voisin, 0, &
    &                comm, status, ierr)
CALL MPI_TYPE_FREE(type_ligne, ierr)
```

Exercice

- Écrire un programme assurant les taches suivantes entre les processus 0 et 1:
- Initialisation de la matrice sur chaque processus
- Définition du type type_colonne
- Envoi de la première colonne du processus 0
- Réception dans la dernière colonne du processus occurrence

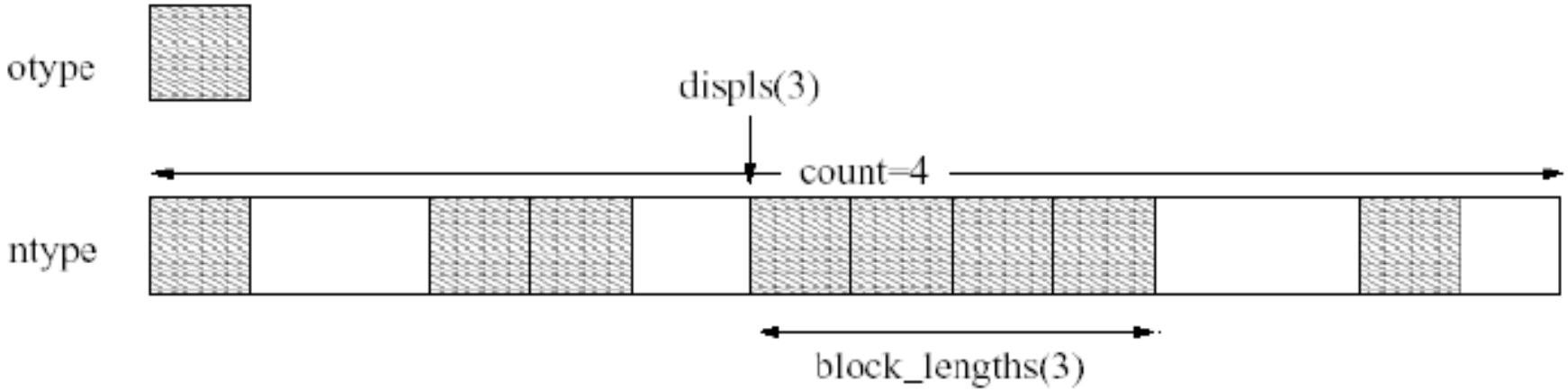
Exercice

construire le type MPI_vector_section, constitué de 4 blocs de 1 MPI_REAL, avec un pas entre les débuts de blocs de 3 MPI_REAL.

Les éléments foncés constituent la section x(2:12:3) du tableau x(1:12). Afficher les éléments 2, 5, 8 et 11 des colonnes 3, avant dernière et dernière.

Type indexé

```
MPI_Type_indexed(count,block_lengths,displs,otypes,ntype,ierr)
  INTEGER,INTENT(IN) :: count              ! Nombre de blocs
  INTEGER,INTENT(IN) :: block_lengths(:)! Longueur d'un blocs
  INTEGER,INTENT(IN) :: displs(:)          ! Deplacements (otype))
  INTEGER,INTENT(IN) :: otype              ! Ancien type
  INTEGER,INTENT(OUT):: ntype              ! Nouveau type
```



MPI_Type_hindexed: les déplacements displs sont exprimés en *bytes*

Type indexé

nb=3, longueurs blocs=(2,1,3), deplacements=(0,3,7)

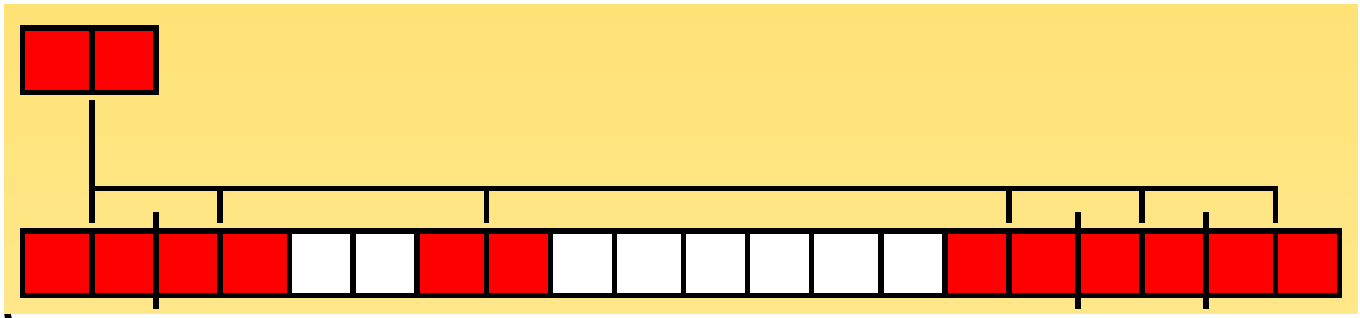


Fig. – Le constructeur MPI_TYPE_INDEXED

```
integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: longueurs_blocs
! Attention les déplacements sont donnés en éléments
integer,intent(in),dimension(nb) :: deplacements
integer,intent(in) :: ancien_type
integer,intent(out) :: nouveau_type,code
call MPI_TYPE_INDEXED (nb,longueurs_blocs,deplacements,ancien_type,nouveau_type,code)
```

Type structure

```
MPI_Type_struct(count,block_lengths,displs,types,ntype,ierr)
  INTEGER,INTENT(IN) :: count              ! Nombre de blocs
  INTEGER,INTENT(IN) :: block_lengths(:)   ! Longueur d'un blocs
  INTEGER,INTENT(IN) :: displs(:)          ! Deplacements (bytes)
  INTEGER,INTENT(IN) :: otypes(:)          ! Anciens types
  INTEGER,INTENT(OUT):: ntype              ! Nouveau type
```

Détermination des déplacements:

```
MPI_Address(loc, iadd, ierr)
  <type>, INTENT(IN) :: loc        ! Debut de la memoire
  INTEGER, INTENT(OUT) :: iadd     ! Adresse absolue de loc
```

Remarque: Similaire à l'opérateur & de C; mais il est préférable d'utiliser MPI_Address, même en C!

Type structure

nb=5, longueurs blocs=(3,1,5,1,1), deplacements=(0,7,11,21,26),
anciens types=(type1,type2,type3,type1,type3)

type1 type2 type3

anciens types

nouveau type

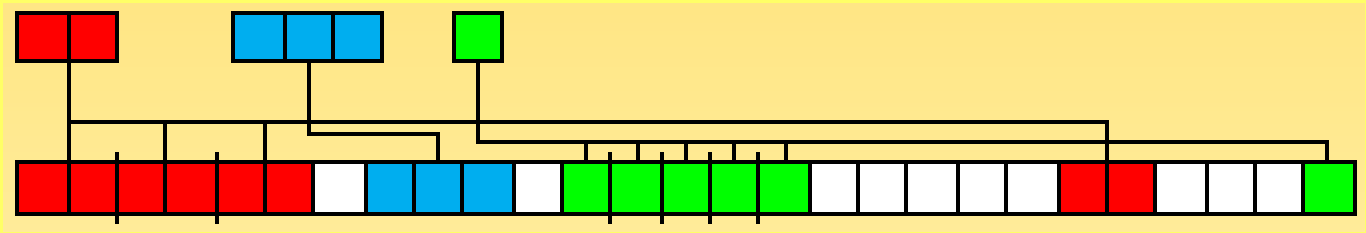


Fig. 32 – Le constructeur MPI TYPE STRUCT

```
integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: longueurs_blocs
integer,intent(in),dimension(nb) :: deplacements
integer,intent(in),dimension(nb) :: anciens_types
integer, intent(out) :: nouveau_type,code
call MPI_TYPE_STRUCT (nb,longueurs_blocs,deplacements,anciens_types,nouveau_type,code)
```

Informations sur un type dérivé MPI

Taille totale: **MPI_TYPE_SIZE()**

Bornes inférieures et supérieures:

MPI_TYPE_LB()

MPI_TYPE_UB()

MPI_TYPE_SIZE≤MPI_TYPE_Ub()-MPI_TYPE_LB()

MPI_TYPE_EXTENT()

Résumé

- Types de données homogènes
  - Pas de sauts: **MPI_Type_contiguous**
  - Sauts constants: **MPI_Type_[h]vector**
  - Sauts variables: **MPI_Type_[h]indexed**
- Types de données hétérogènes
  - **MPI_Type_struct**

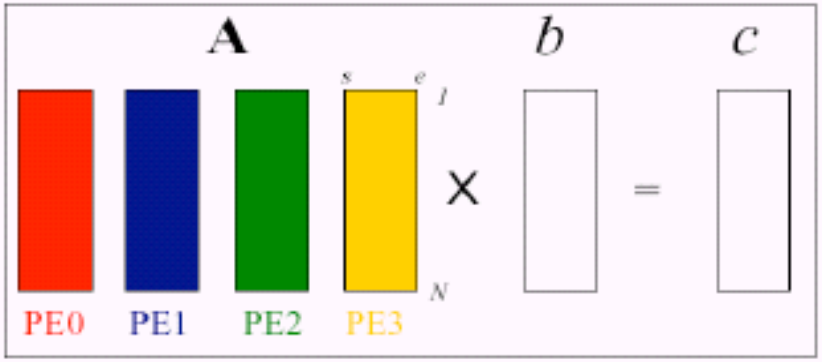
Résumé (suite)

- Utilitaires
  - **MPI_Type_extent, MPI_Type_size**
  - **MPI_Address**
- Utilisation:
  - Plusieurs communications avec le même type
  - On veut éviter de faire des copies temporaires
  - Permet de diminuer l'effet de latence

Multiplication matrice vecteur

- Contribution **locale**:

$$c_{ip} = \sum_{j=s_p}^{e_p} A_{ij} b_j, \quad i = 1 \ldots N$$



- Somme **globale**.

$$c_i = \sum_p c_{ip}, \quad i = 1 \ldots N$$