



# Programmation Parallèle et Distribuée

---

Cours 5 : MPI avancé

Patrick Carribault

David Dureau

Marc Pérache ([marc.perache@cea.fr](mailto:marc.perache@cea.fr))



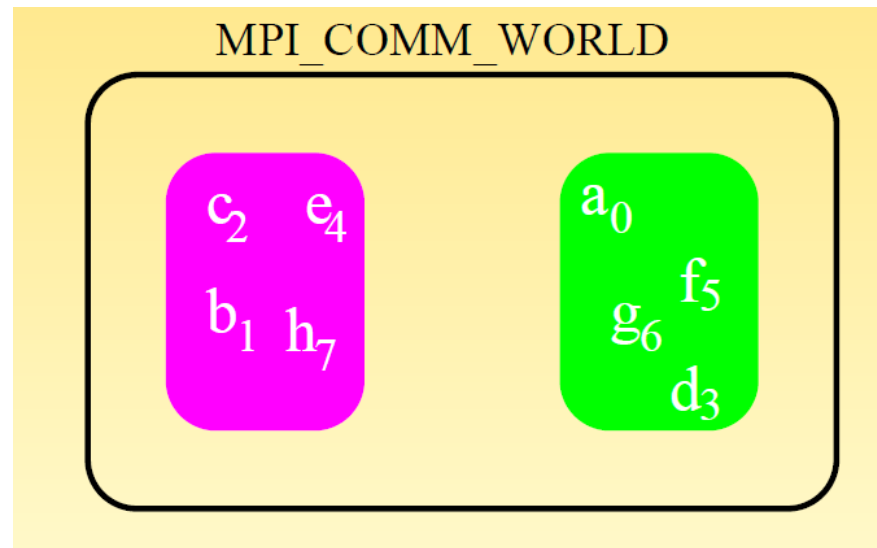
# Plan du cours 5

---

- Les communicateurs
- Communication collectives avec fonction utilisateur
- Introduction aux types dérivés
- Slides issus du cours IDRIS sur MPI

# Les communicateurs

- Il s'agit de partitionner un ensemble de processus afin de créer des sous-ensembles sur lesquels on puisse effectuer des opérations telles que des communications point à point, collectives, etc. Chaque sous-ensemble ainsi créé aura son propre espace de communication.





# Les communicateurs

---

- C'est l'histoire de la poule et de l'œuf...
  - On ne peut créer un communicateur qu'à partir d'un autre communicateur
  - Fort heureusement, cela a été résolu en postulant que la poule existait déjà. En effet, un communicateur est fourni par défaut, dont l'identificateur MPI COMM WORLD est un entier défini dans les fichiers d'en-tête.
  - Ce communicateur initial MPI COMM WORLD est créé pour toute la durée d'exécution du programme à l'appel du sous-programme MPI INIT()
  - Ce communicateur ne peut être détruit que via l'appel à MPI FINALIZE()
  - Par défaut, il fixe donc la portée des communications point à point et collectives à tous les processus de l'application



# Les communicateurs

---

- Dans l'exemple qui suit, nous allons :
  - regrouper d'une part les processus de rang pair et d'autre part les processus de rang impair ;
  - ne diffuser un message collectif qu'aux processus de rang pair et un autre qu'aux processus de rang impair.

# Les communicateurs

MPI\_COMM\_WORLD

$a_0$	$e_4$	$h_7$	$f_5$
$g_6$	$c_2$	$b_1$	$d_3$

$a_0^0$	$e_4^2$
$g_6^3$	$c_2^1$

$h_7^3$	$f_5^2$
$b_1^0$	$d_3^1$

$a_0^0$	$e_4^2$
$g_6^3$	$c_2^1$

$h_7^3$	$f_5^2$
$b_1^0$	$d_3^1$

$a_0$	$e_4$	$h_7$	$f_5$
$g_6$	$c_2$	$b_1$	$d_3$

```
$ mpirun -np 8 CommPairImpair
```

call MPI\_INIT(...)

call MPI\_COMM\_SPLIT(...)

call MPI\_BCAST(...)

call MPI\_COMM\_FREE(...)



# Les communicateurs

---

- Que faire pour que le processus 2 diffuse ce message au sous-ensemble de processus de rang pair, par exemple ?
  - Boucler sur des send/recv peut être très pénalisant surtout si le nombre de processus est élevé. De plus un test serait obligatoire dans la boucle pour savoir si le rang du processus auquel le processus 2 doit envoyer le message est pair ou impair.
  - La solution est de créer un communicateur regroupant ces processus de sorte que le processus 2 diffuse le message à eux seuls.

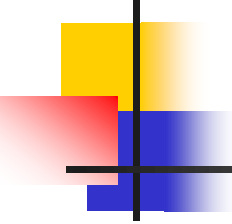


# Les communicateurs: groupes et communicateurs

---

- Un communicateur est constitué :
  - d'un groupe, qui est un ensemble ordonné de processus ;
  - d'un contexte de communication mis en place à l'appel du sous-programme de construction du communicateur, qui permet de délimiter l'espace de communication.
- Les contextes de communication sont gérés par MPI (le programmeur n'a aucune action sur eux : c'est un attribut « caché »)
- En pratique, pour construire un communicateur, il existe deux façons de procéder :
  - par l'intermédiaire d'un groupe de processus ;
  - directement à partir d'un autre communicateur.





# Les communicateurs : groupes et communicateurs

---

- Dans la bibliothèque MPI, divers sous-programmes existent pour construire des communicateurs : MPI CART CREATE(), MPI CART SUB(), MPI COMM CREATE(), MPI COMM DUP(), MPI COMM SPLIT()
- Les constructeurs de communicateurs sont des opérateurs collectifs (qui engendrent des communications entre les processus)
- Les communicateurs que le programmeur crée peuvent être gérés dynamiquement et, de même qu'il est possible d'en créer, il est possible d'en détruire en utilisant le sous-programme MPI COMM FREE()

# Les communicateurs: issu d'un autre

- L'utilisation directe des groupes présente dans ce cas divers inconvénients, car elle impose de :
  - nommer différemment les deux communicateurs (par exemple comm pair et comm impair) ;
  - passer par les groupes pour construire ces deux communicateurs ;
  - laisser le soin à MPI d'ordonner le rang des processus dans ces deux communicateurs ;
  - faire des tests conditionnels lors de l'appel au sous-programme MPI\_BCAST() :

```
if (comm_pair /= MPI_COMM_NULL) then
  ...
  ! Diffusion du message seulement aux processus de rangs pairs
  call MPI_BCAST(a,m,MPI_REAL,rang_ds_pair,comm_pair,code)
elseif (comm_impair /= MPI_COMM_NULL) then
  ...
  ! Diffusion du message seulement aux processus de rangs impairs
  call MPI_BCAST(a,m,MPI_REAL,rang_ds_impair,comm_impair,code)
end if
```

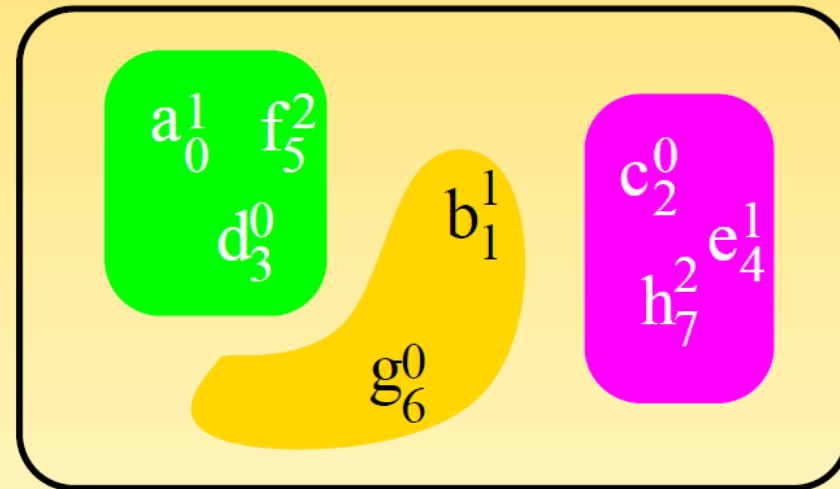
# Les communicateurs : issu d'un autre

- Le sous-programme MPI COMM SPLIT() permet de partitionner un communicateur donné en autant de communicateurs que l'on veut...

```
integer, intent(in)  :: comm, couleur, clef
integer, intent(out) :: nouveau_comm, code
call MPI_COMM_SPLIT(comm, couleur, clef, nouveau_comm, code)
```

processus	a	b	c	d	e	f	g	h
rang_monde	0	1	2	3	4	5	6	7
couleur	0	2	3	0	3	0	2	3
clef	2	15	0	0	1	3	11	1
rang_nv_com	1	1	0	0	1	2	0	2

MPI\_COMM\_WORLD





# Les communicateurs

---

```
program PairsImpairs
  use mpi
  implicit none

  integer, parameter :: m=16
  integer             :: clef,CommPairsImpairs
  integer             :: rang_dans_monde,code
  real, dimension(m) :: a

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang_dans_monde,code)

  ! Initialisation du vecteur A
  a(:)=0.
  if(rang_dans_monde == 2) a(:)=2.
  if(rang_dans_monde == 5) a(:)=5.
```



# Les communicateurs

---

```
clef = rang_dans_monde
if (rang_dans_monde == 2 .OR. rang_dans_monde == 5 ) then
  clef=-1
end if

! Création des communicateurs pair et impair en leur donnant une même dénomination
call MPI_COMM_SPLIT(MPI_COMM_WORLD,mod(rang_dans_monde,2),clef,CommPairsImpairs,code)

! Diffusion du message par le processus 0 de chaque communicateur aux processus
! de son groupe
call MPI_BCAST(a,m,MPI_REAL,0,CommPairsImpairs,code)

! Destruction des communicateurs
call MPI_COMM_FREE(CommPairsImpairs,code)
call MPI_FINALIZE(code)
end program PairsImpairs
```



# Opérations collectives utilisateur

---

- `int MPI_Op_create( MPI_User_function *function, int commute, MPI_Op *op );`
- On fournit un pointeur sur fonction. La fonction doit être associative et peut être commutative ou pas. Elle a un prototype défini.

# Opérations collectives utilisateur

```
void addem( int *, int *, int *, MPI Datatype * );  
void addem( int *invec, int *inoutvec, int *len, MPI Datatype  
            *dtype )  
{  
    int i;  
    for ( i = 0 ; i < *len ; i++ )  
        inoutvec[i] += invec[i];  
}
```

- Déclaration

```
MPI_Op_create( (MPI_User_function *)addem, 1, &op );
```

- Libération

```
MPI_Op_free(op);
```



# Types dérivés

---

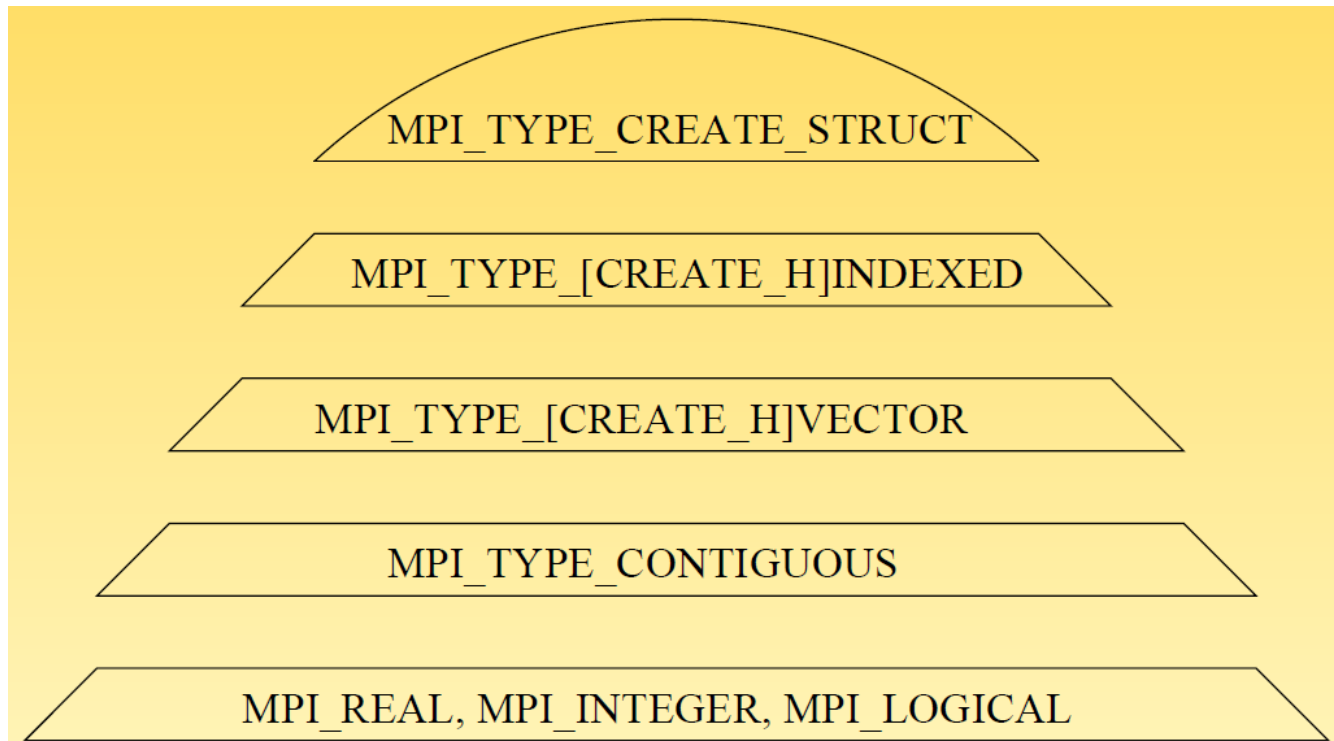
- Dans les communications, les données échangées sont typées : MPI INTEGER, MPI REAL, MPI COMPLEX, etc.
- On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que MPI TYPE CONTIGUOUS(), MPI TYPE VECTOR(), MPI TYPE CREATE HVECTOR().
- A chaque fois que l'on crée un type de données, il faut le valider à l'aide du sous-programme MPI TYPE COMMIT().
- Si on souhaite réutiliser le même type, on doit le libérer avec le sous-programme MPI TYPE FREE().





# Types dérivés

---



# Types dérivés: types contigus

- MPI 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 dérivés: avec un pas constant

- MPI TYPE VECTOR() 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'éléments.

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)
```

```
integer, intent(in)  :: nombre_bloc, longueur_bloc  
integer, intent(in)  :: pas ! donné en éléments  
integer, intent(in)  :: ancien_type  
integer, intent(out) :: nouveau_type, code
```

```
call MPI_TYPE_VECTOR(nombre_bloc, longueur_bloc, pas, ancien_type, nouveau_type, code)
```

# Types dérivés : avec un pas constant

- MPI TYPE CREATE 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, parce qu'alors le pas ne peut plus être exprimé en nombre d'éléments du type générique.

```
integer, intent(in) :: nombre_bloc, longueur_bloc
integer(kind=MPI_ADDRESS_KIND), intent(in) :: pas ! donné en octets
integer, intent(in) :: ancien_type
integer, intent(out) :: nouveau_type, code

call MPI_TYPE_CREATE_HVECTOR(nombre_bloc, longueur_bloc, pas,
                             ancien_type, nouveau_type, code)
```



# Types dérivés

---

- Il est nécessaire de valider tout nouveau type de données dérivé à l'aide du sous-programme `MPI_TYPE_COMMIT()`.
- La libération d'un type de données dérivé se fait par le sous-programme `MPI_TYPE_FREE()`.



# Types dérivés

```
program colonne
  use mpi
  implicit none

  integer, parameter                :: nb_lignes=5,nb_colonnes=6
  integer, parameter                :: etiquette=100
  real, dimension(nb_lignes,nb_colonnes) :: a
  integer, dimension(MPI_STATUS_SIZE) :: statut
  integer                           :: rang,code,type_colonne

  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  ! Initialisation de la matrice sur chaque processus
  a(:, :) = real(rang)

  ! Définition du type type_colonne
  call MPI_TYPE_CONTIGUOUS(nb_lignes,MPI_REAL,type_colonne,code)

  ! Validation du type type_colonne
  call MPI_TYPE_COMMIT(type_colonne,code)
```



# Types dérivés

---

```
! Envoi de la première colonne
if ( rang == 0 ) then
    call MPI_SEND(a(1,1),1,type_colonne,1,etiquette,MPI_COMM_WORLD,code)

! Réception dans la dernière colonne
elseif ( rang == 1 ) then
    call MPI_RECV(a(1,nb_colonnes),1,type_colonne,0,etiquette,&
        MPI_COMM_WORLD,statut,code)
end if

! Libère le type
call MPI_TYPE_FREE(type_colonne,code)

call MPI_FINALIZE(code)

end program colonne
```



# Types dérivés

---

- `MPI TYPE INDEXED()` permet de créer une structure de données composée d'une séquence de blocs contenant un nombre variable d'éléments et séparés par un pas variable en mémoire. Ce dernier est exprimé en éléments.
- `MPI TYPE CREATE HINDEXED()` a la même fonctionnalité que `MPI TYPE INDEXED()` sauf que le pas séparant deux blocs de données est exprimé en octets. Cette instruction est utile lorsque le type générique n'est pas un type de base MPI (`MPI INTEGER`, `MPI REAL`, ...) mais un type plus complexe construit avec les sous-programmes MPI vus précédemment. On ne peut exprimer alors le pas en nombre d'éléments du type générique d'où le recours à `MPI TYPE CREATE HINDEXED()`.
- Pour `MPI TYPE CREATE HINDEXED()`, comme pour `MPI TYPE CREATE HVECTOR()`, utilisez `MPI TYPE SIZE()` ou `MPI TYPE GET EXTENT()` pour obtenir de façon portable la taille du pas en nombre d'octets.