

# Langages du parallélisme

# Message Passing Interface

1. Manipulation de structures MPI
  1. Communicateurs
  2. Opérateurs définis par l'utilisateur
2. Topologie MPI
3. Types dérivés

# Message Passing Interface

## 1. Manipulation de structures MPI

- 1. Communicateurs

- 2. Opérateurs définis par l'utilisateur

## 2. Topologie MPI

## 3. Types dérivés

# Message Passing Interface

## 1. Manipulation de structures MPI

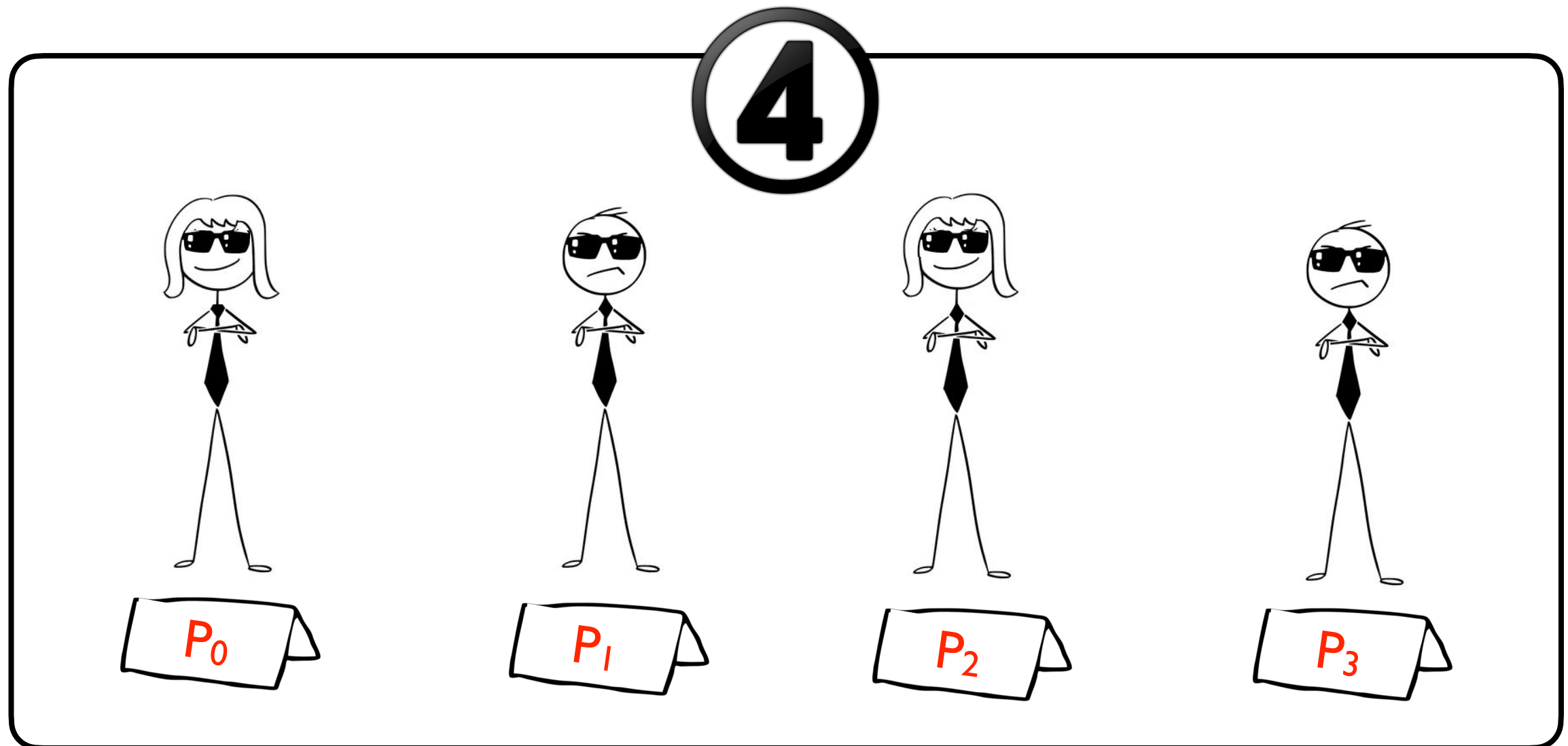
### 1. Communicateurs

### 2. Opérateurs définis par l'utilisateur

### 2. Topologie MPI

### 3. Types dérivés

# Un concept de base: Communicateur



**Communicateur = groupe de processus + contexte de communication**

- ➔ Prédéfini: `MPI_COMM_WORLD` avec tous les processus
- ➔ Type: `MPI_Comm`

# Communicateurs et groupes

## Petit nombre de processus

➔ `MPI_COMM_WORLD` suffisant

## Grand nombre de processus

➔ `MPI_COMM_WORLD` pas toujours pratique, on veut pouvoir:

- faire des communications entre un sous ensemble de processus
- travailler sur des jeux de données différents

## Deux notions

1. **Groupe**: ensemble ordonné de processus
2. **Communicateur**: groupe mais qui spécifie un domaine de communication

# Communicateurs

Communicateur = groupe de processus + contexte de communication  
( canal entre des processus pour échanger des messages )

Les opérations pour

- créer les communicateurs sont des **collectives**
- accéder à une information d'un communicateur sont **locales**

Deux types de communicateurs existent:

- **Intracommunicateur** : communication dans un groupe
- **Intercommunicateur** : communication entre deux groupes

# Créer des communicateurs

## Duplication, split et comparaison

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```

```
▸ result = MPI_IDENT (même communicateur)  
          MPI_CONGRUENT (même processus, mêmes rangs)  
          MPI_SIMILAR (même processus, rangs différents)  
          MPI_UNEQUAL
```

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,  
                   MPI_Comm *newcomm)
```



# Créer des communicateurs

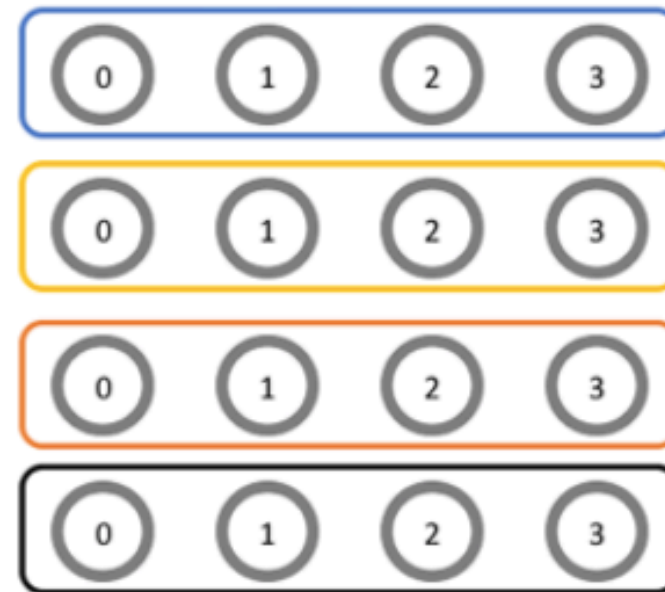
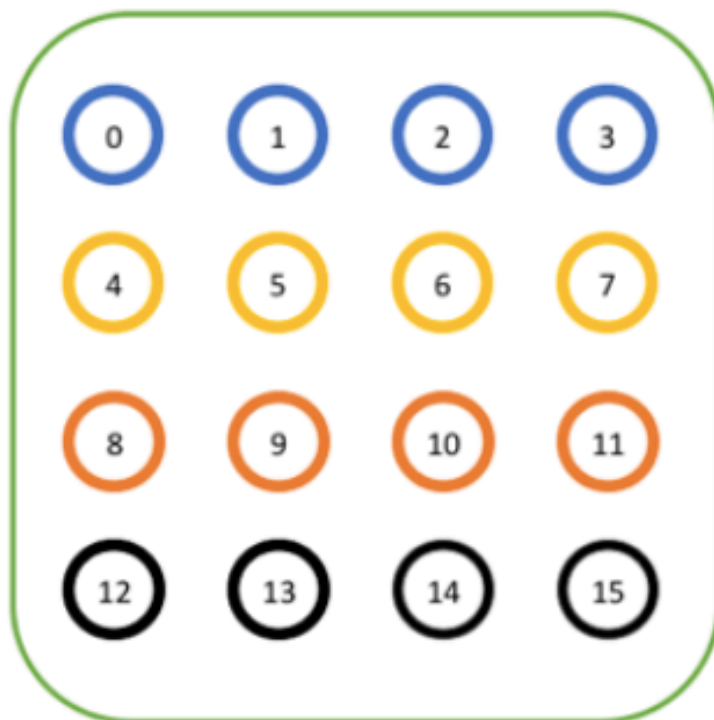
```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```



determine à quel  
communicateur le  
processus va  
appartenir



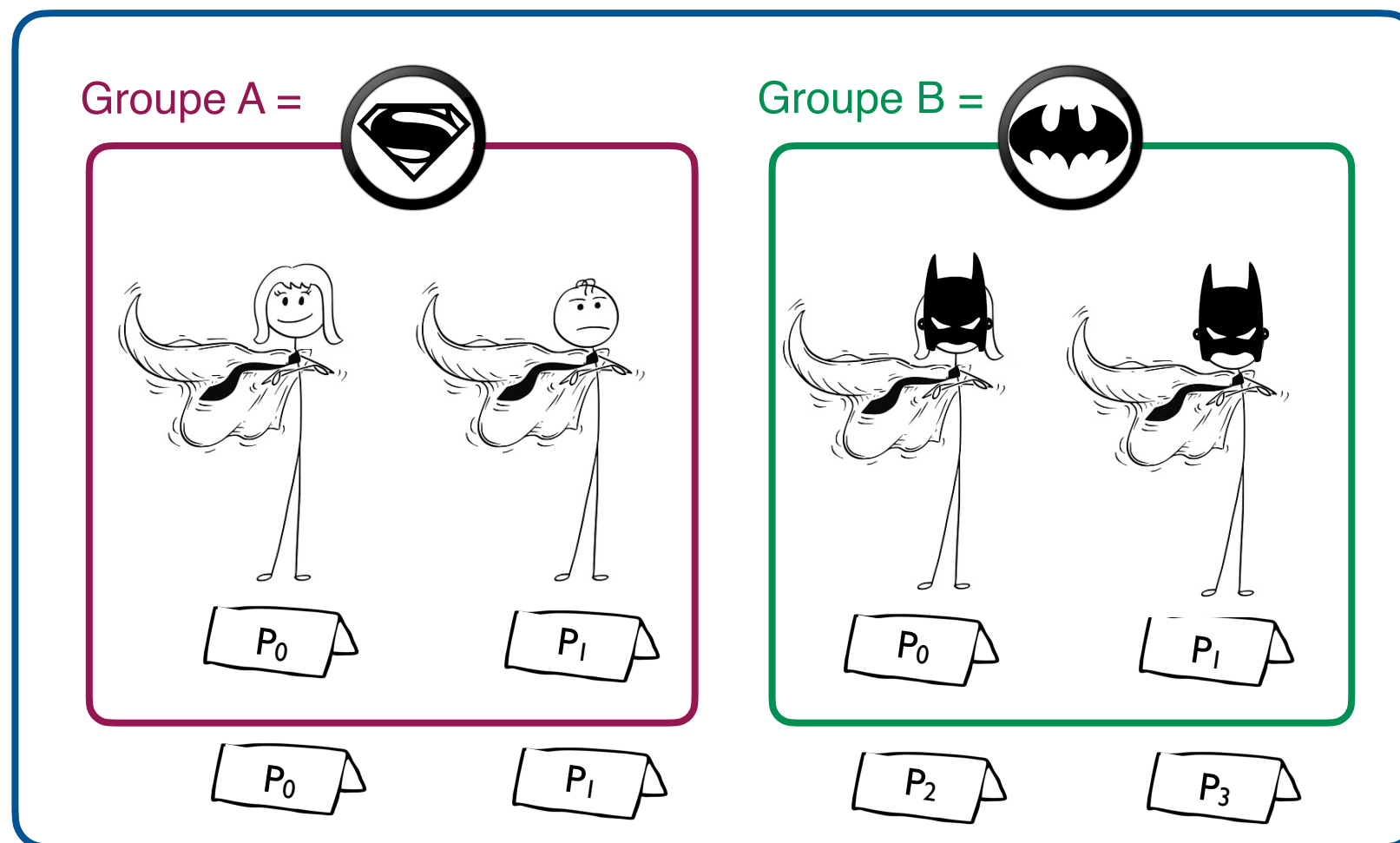
determine l'ordre(rank)  
au sein de chaque  
nouveau communicateur



# Groupes

Autres moyens de créer des communicateurs de manière **plus flexible** via des **groupes**

Groupe = **ensemble de processus ordonnés**



Groupe de base, associé à MPI\_COMM\_WORLD

# Créer des groupes

Un constructeur à partir du communicateur:

- MPI\_Comm\_group**(MPI\_Comm comm, MPI\_Group \*group)
- MPI\_Group\_size**(MPI\_Group group, int \*size)
- MPI\_Group\_rank**(MPI\_Group group, int \*rank)
  - ➡ retourne le rang du processus dans le groupe ou MPI\_UNDEFINED

Correspondance entre les rangs des processus du group1 et leur rang dans group2

**MPI\_Group\_translate\_ranks**(group1, n, rank1, group2, rank2)

Comparer deux groupes **MPI\_Group\_compare**(group1, group2, result). Le résultat est

- **MPI\_IDENT** si les éléments des groupes sont les mêmes ainsi les rangs ;
- **MPI\_SIMILAR** si les éléments des groupes sont les mêmes mais avec des rangs différents ;
- **MPI\_UNEQUAL** sinon

Supprimer un groupe : **MPI\_Group\_free**(group)

# Créer des groupes

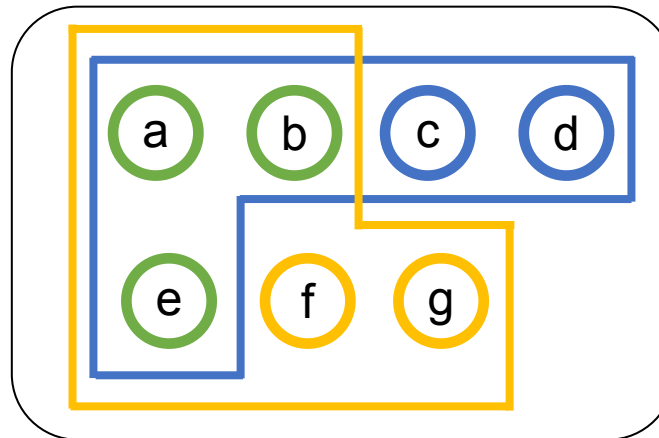
```
int MPI_Group_*(MPI_Group group1, MPI_Group group2,  
                MPI_Group *newgroup)
```

- Où \* ∈ {union, intersection, difference}
- **newgroup** contient les processus satisfaisants l'opération \* d'abord ordonnés selon l'ordre dans le groupe 1 et ensuite en fonction de l'ordre dans le groupe 2.
- Dans le nouveau groupe chaque processus sera présent seulement 1 fois.

# Exemple

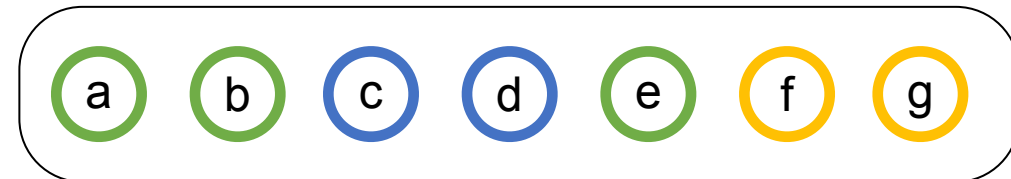
group1 = {a, b, c, d, e}

group2 = {e, f, g, b, a}



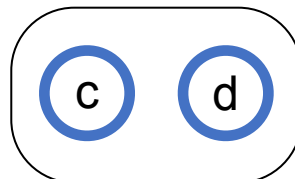
## Union

-newgroup = {a, b, c, d, e, f, g}



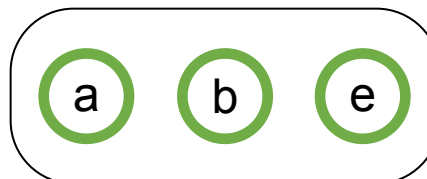
## Différence

-newgroup = {c, d}



## Intersection

-newgroup = {a, b, e}



# Intracommunicateurs

## Qu'est ce qu'un intracommunicateur?

- Des processus
- Un groupe
- Un contexte de communication
- Une topologie (optionnel)

## Creation à partir d'un ancien communicateur et d'un groupe

### Prototypes:

- `int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)`
- `int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm)`

# Intracommunicateurs

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,  
                    MPI_Comm *newcomm)
```

↑  
Doit être appelée par tous  
les processus dans le  
groupe associé à comm

↑  
Sous ensemble du groupe  
associé à comm

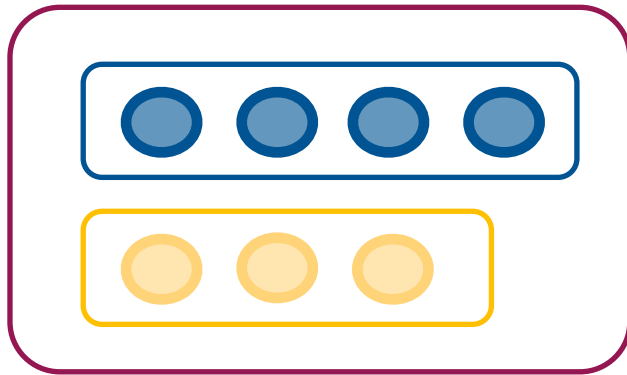
```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,  
                          MPI_Comm *newcomm)
```

↑  
Doit être appelée par tous  
les processus dans le  
groupe group

↑  
Sous ensemble du groupe  
associé à comm

# Intercommuniqueurs

Qu'est-ce qu'un intercommuniqueur?



- Plus de processus
- **Deux** groupes
- Un communiqueur

Objet qui construit un domaine de communication entre deux groupes disjoints de processus.

Autorise des communications entre les groupes: très utile pour les couplages de codes, les simulations client/serveur

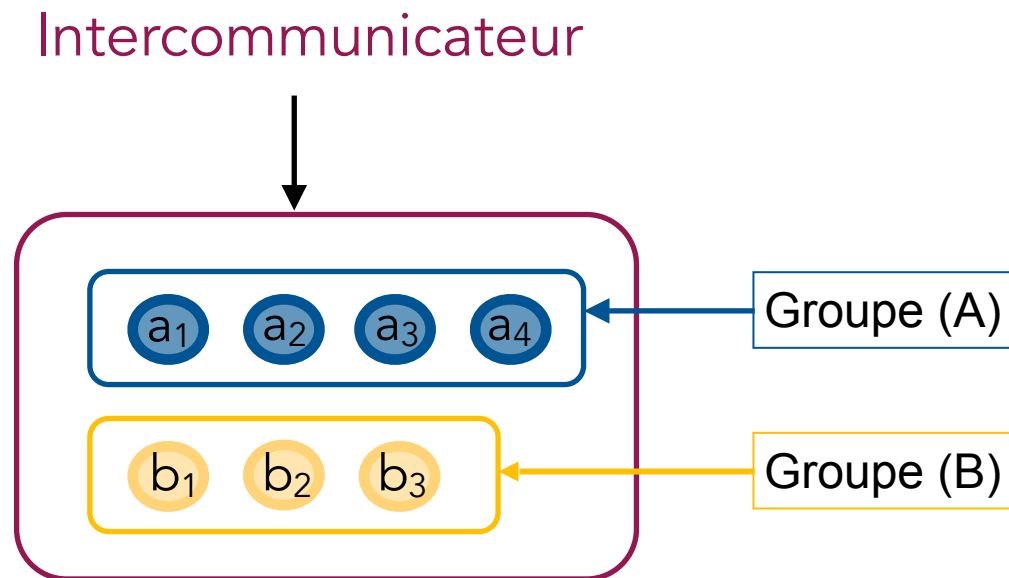
## Prototype

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,  
MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)
```



# Intercommuniqueurs

Objet qui construit un domaine de communication entre deux groupes disjoints de processus



**NB:** Il est IMPOSSIBLE d'envoyer un message à un processus dans le même groupe en utilisant ce communicateur.

Pour tous les processus de (A)

- (A) est le groupe **local** ;
- (B) est le groupe **remote**

Pour tous les processus de (B)

- (A) est le groupe **remote** ;
- (B) est le groupe **local**.

# Groupes et communicateurs spéciaux

## Groupes spéciaux :

Groupe sans aucun processus: `MPI_GROUP_EMPTY`

`MPI_GROUP_NULL` signifie que l'objet groupe n'existe pas.

## Communicateurs spéciaux :

Communicateur initial: `MPI_COMM_WORLD`

Communicateur qui contient le processus d'appel: `MPI_COMM_SELF` (utile pour les I/O)

`MPI_COMM_NULL` signifie que le communicateur n'existe pas

# Message Passing Interface

## 1. Manipulation de structures MPI

### 1. Communicateurs

### 2. Opérateurs définis par l'utilisateur

## 2. Topologie MPI

## 3. Types dérivés

# Opérations définies par l'utilisateur

```
int MPI_Op_create (
```

```
    MPI_User_function *user_fn(in),
```

```
    int commute(in),
```

```
    MPI_Op *op(out),
```

```
);
```

Pointeur de fonction **user\_fn**

Prototype:

```
void (*f) (void* invec, void* inoutvec,  
int *len, MPI_Datatype *datatype);
```

# Opérations définies par l'utilisateur

```
int MPI_Op_create (  
    MPI_User_function *user_fn(in),  
    int commute(in),  
    MPI_Op *op(out),  
);
```



Vaut 1 si l'opération  
est commutative

# Opérations définies par l'utilisateur

- **Exemple**

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

- **Creation de l'opération**

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

- **Libération de l'opération**

```
MPI_Op_free(&op);
```

# Message Passing Interface

## 1. Manipulation de structures MPI

### 1. Communicateurs

### 2. Opérateurs définis par l'utilisateur

## 2. Topologie MPI

## 3. Types dérivés

# Topologie MPI

Dans les méthodes de partitionnement de domaine sur des structures régulières où l'on fait correspondre le domaine de calcul à la grille de processus, il est intéressant de pouvoir disposer les processus suivant une topologie régulière

MPI permet de définir des topologies virtuelles de type :

**-Cartésien :**

- ➡ chaque processus est défini dans une grille de processus ;
- ➡ la grille peut être périodique ou non ;
- ➡ les processus sont identifiés par leurs coordonnées dans la grille

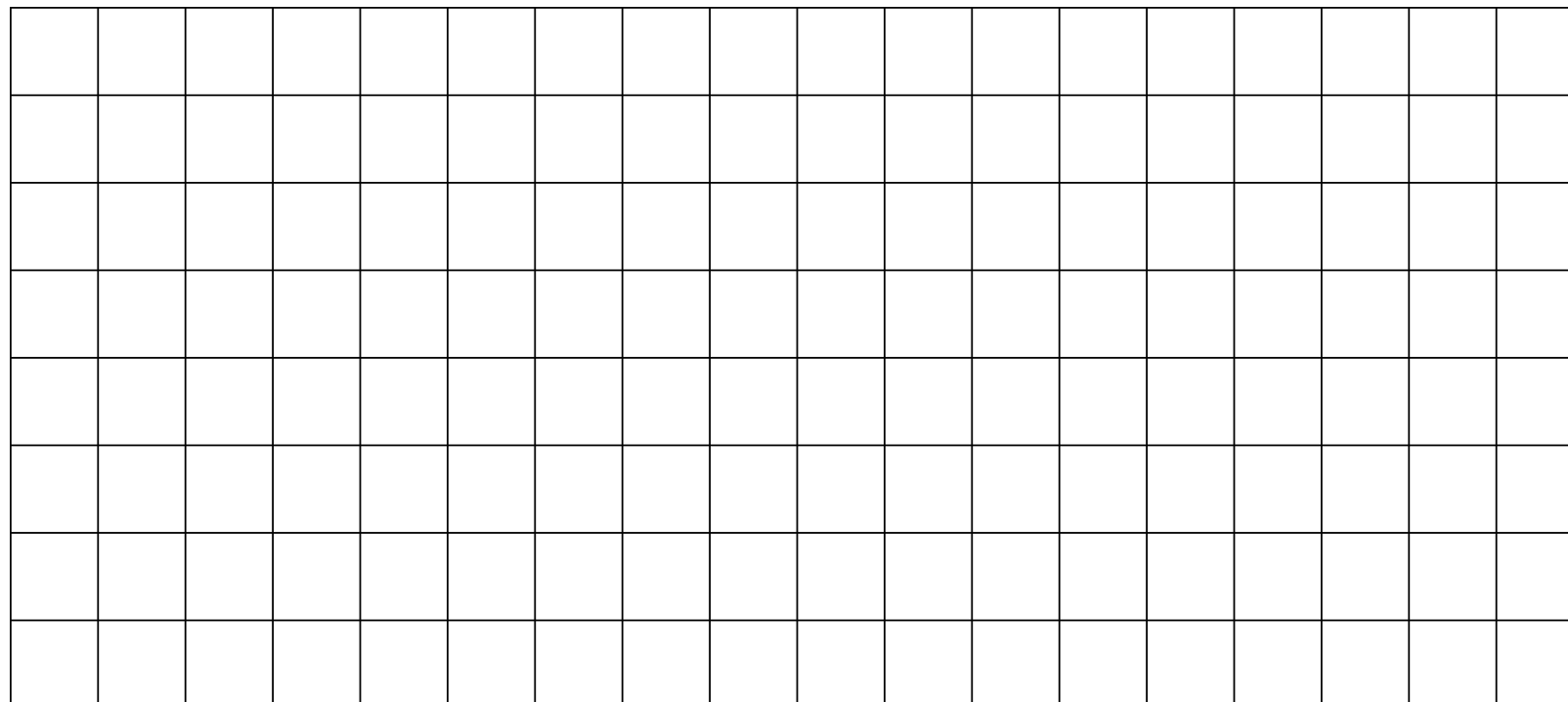
**-Graphe :** Généralisation à des topologies plus complexes



# Topologie MPI

Considérons une grille multi-dimensionnelle: exemple 2D

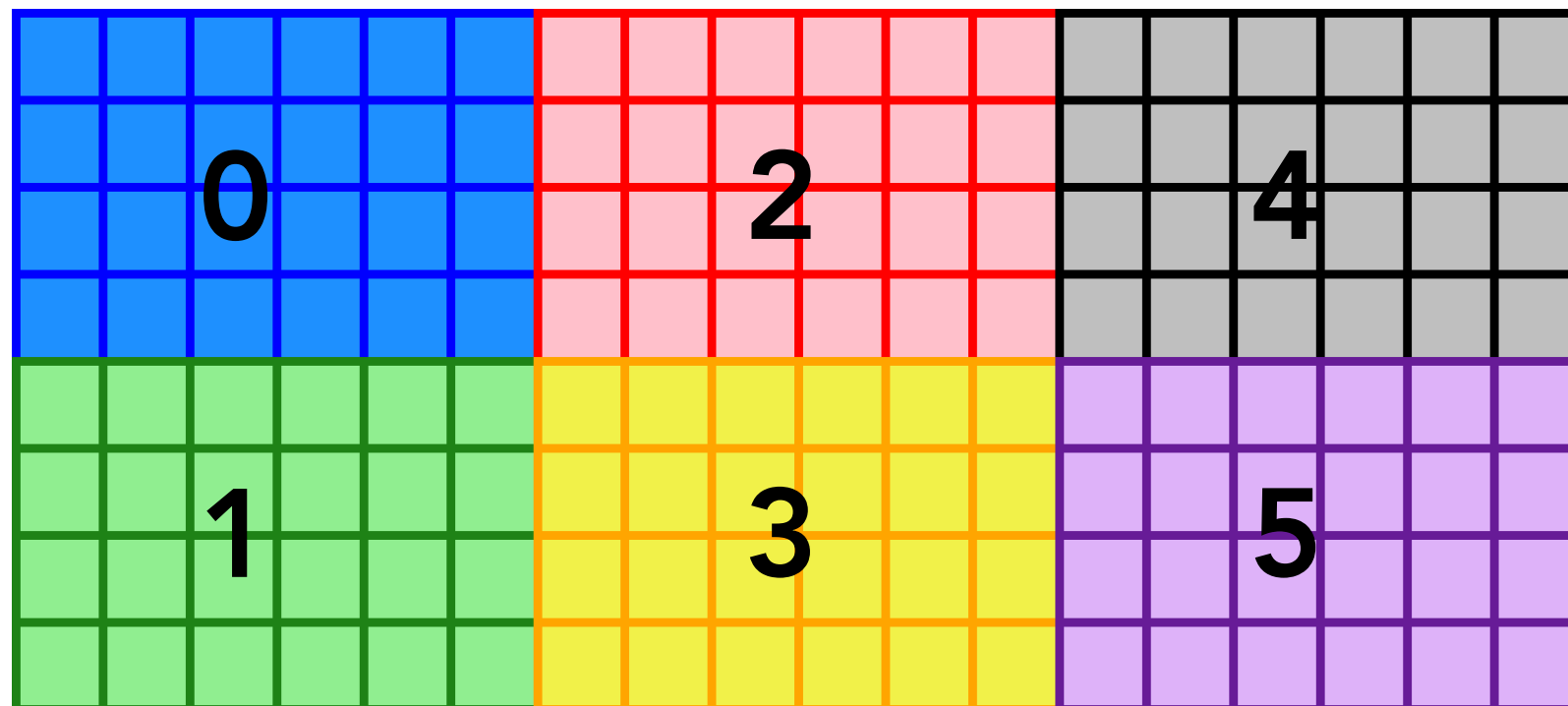
➔ Décomposition en 6 rangs



# Topologie MPI

Considérons une grille multi-dimensionnelle: exemple 2D

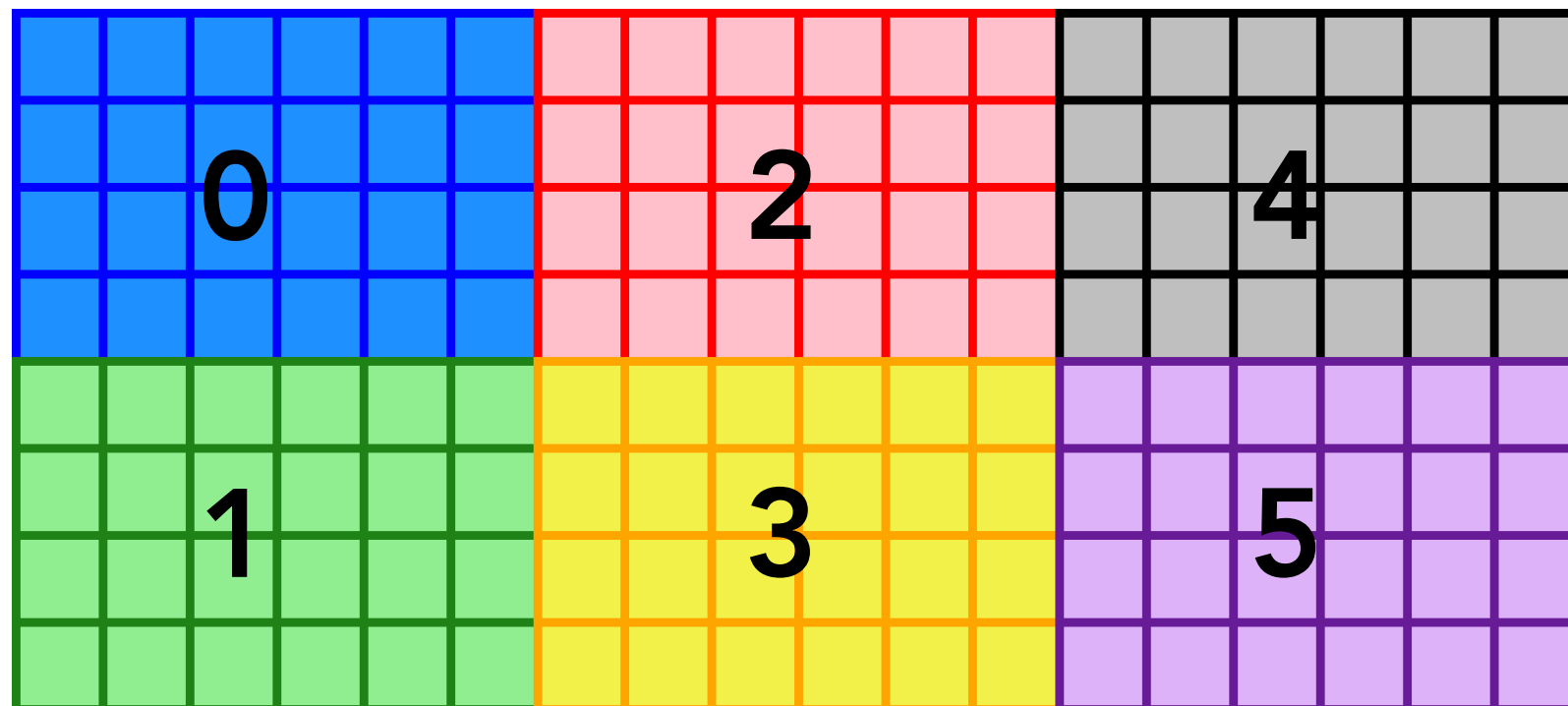
- Décomposition en 6 rangs
- $6 \times 4 = 24$  cellules par rang MPI



# Topologie MPI

Considérons une grille multi-dimensionnelle: exemple 2D

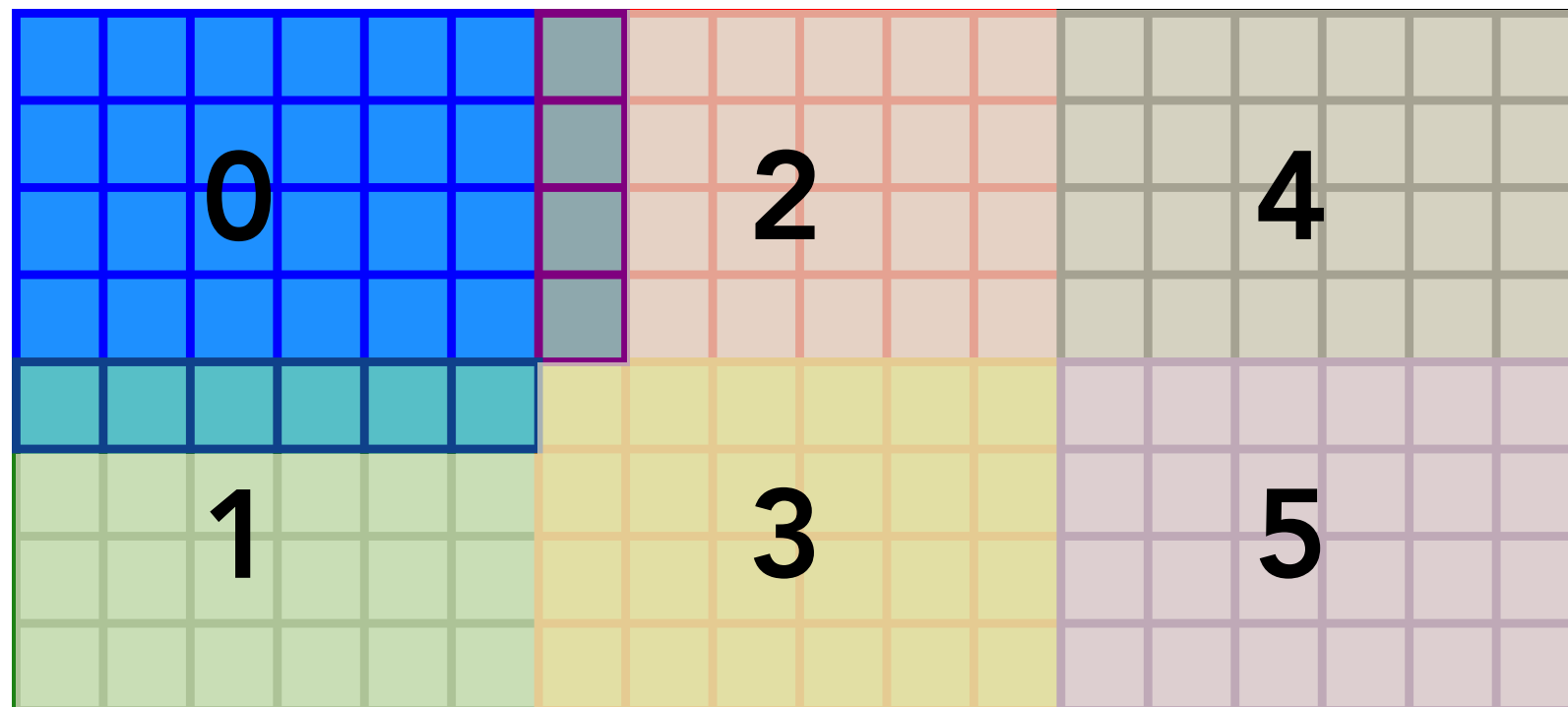
➡ La mise à jour d'une cellule requiert la contribution de ses voisins directs



# Topologie MPI

Considérons une grille multi-dimensionnelle: exemple 2D

➔ La mise à jour d'une cellule requiert la contribution de ses voisins directs



# Topologie cartésienne

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],  
const int periods[], int reorder, MPI_Comm *comm_cart);
```

- ➔ IN comm\_old (communicator)
- ➔ IN ndims (nombre de dimensions de la grille)
- ➔ IN dims (tableau des dimensions de la grille)
- ➔ IN periods (tableau spécifiant la périodicité)
- ➔ IN reorder (si true, renumérotation des processus)
- ➔ OUT comm\_cart (communicateur avec la topologie cartésienne)

Si reorder = false, le rang des processus est le même dans comm et dans comm\_cart

# Fonctions associées à la grille

- MPI\_Cart\_rank retourne le rang du processus associé aux coordonnées dans la grille ;

**MPI\_Cart\_rank**(comm\_cart, coords, rang)

- MPI\_Cart\_coords renvoie les coordonnées d'un processus de rang donné ;

**MPI\_Cart\_coords**(comm\_cart, rank, dim, coords)

- MPI\_Cart\_shift donne le processeur avant et après dans la direction d

**MPI\_Cart\_shift**(comm\_cart, d, pas, rang\_precedent, rang\_suivant)

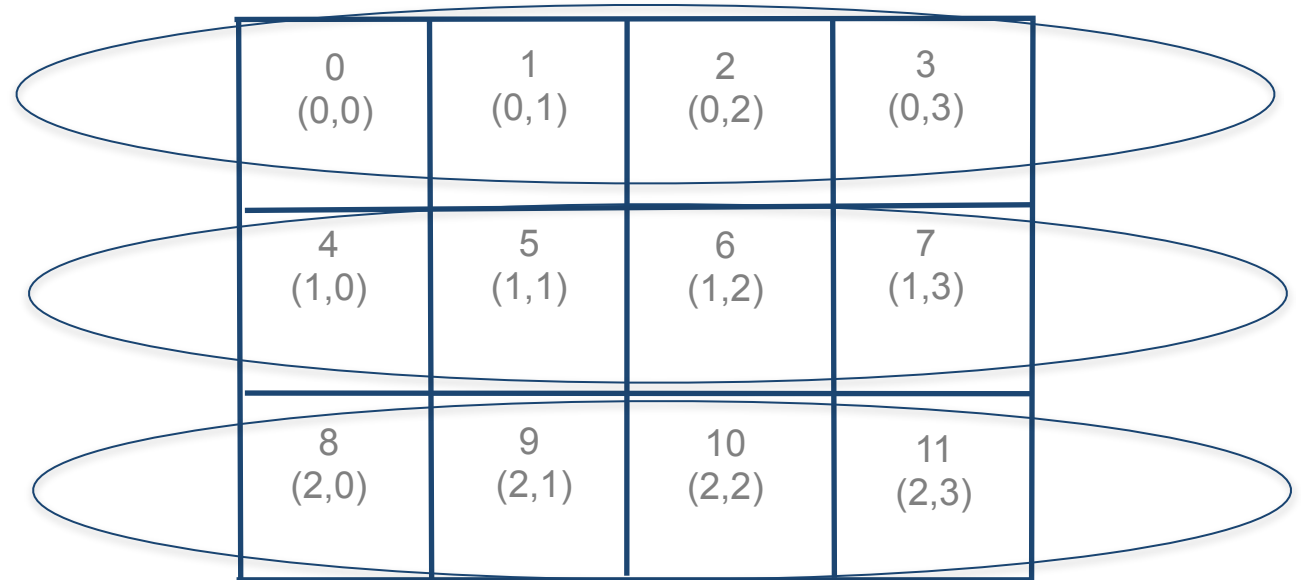
pas : longueur du shift - et +

d : direction du shift

Si on sort de la topologie, MPI\_PROC\_NULL est retourné

# Exemple

```
dims[0]= 3 ;  
dims[1]= 4 ;  
periods[0]= 0 ;  
periods[1]= 1 ;  
reorder    = 1 ;
```



```
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_2D) ;
```

```
MPI_Cart_coords(grid_2D, rank, 2, coords) ;
```

```
MPI_Cart_rank( grid_2D, coords, &rank2d ) ;
```

```
MPI_Cart_shift(grid_2D, 0, 1, &dessous, &dessus) ;
```

```
MPI_Cart_shift(grid_2D, 1, 1, &avant, &apres) ;
```

# Exemple

Rang **5**

Coordonnées (1,1).

Mes voisins:

avant 4 et après 6  
dessus 1 dessous 9

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

Rang **11**

Coordonnées (2,3).

Mes voisins:

avant 10 et après 8  
dessus 7 dessous -2

Dans cette implémentation `MPI_PROC_NULL = -2`



# Partitionnement cartésien

Souvent on souhaite exécuter une opération sur une partie de la topologie :

- les lignes/colonnes (si grille 2D), les plans (si grille 3D)
- Un communicateur par ligne/colonne/plan (autant de communicateurs que de plans)
- Des opérations collectives seulement sur le nouveau communicateur

**`MPI_Cart_sub(comm, remain_dims, subcom)`**

IN	comm	communicateur cartésien
IN	remain_dims	tableaux précisant les dimensions de la grille que l'on garde
OUT	subcomm	communicateur avec la topologie cartésienne

# Graphe de processus

Dans beaucoup d'applications, la décomposition d'un domaine n'est pas cartésienne mais sous forme d'un graphe.

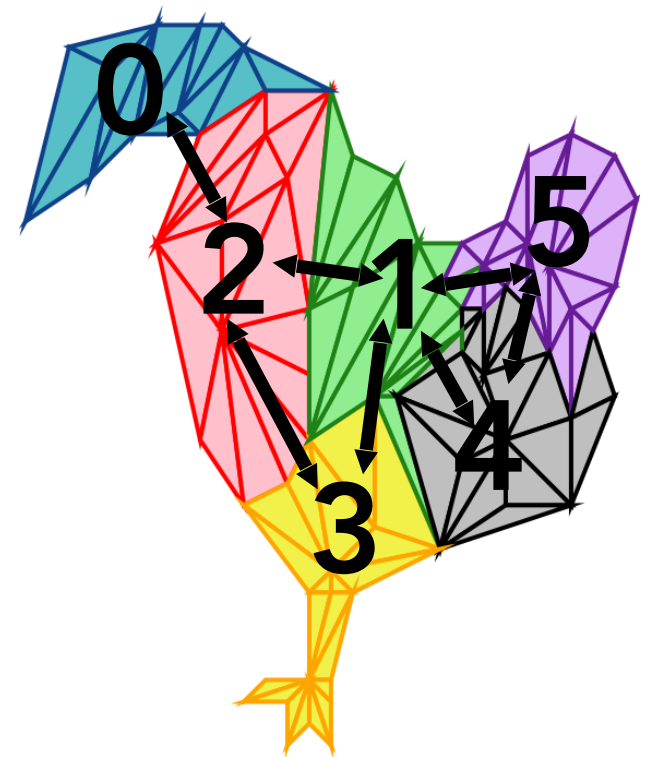
Un rang a besoin d'échanger des données avec ses voisins

$0 \leftrightarrow 2$

$2 \leftrightarrow 1 ; 2 \leftrightarrow 3$

$1 \leftrightarrow 3 ; 1 \leftrightarrow 4 ; 1 \leftrightarrow 5$

$4 \leftrightarrow 5 ;$



# Graphe de processus

Dans beaucoup d'applications, la décomposition d'un domaine n'est pas cartésienne mais sous forme d'un graphe.

MPI\_Graph\_create permet de définir une topologie de type graphe

```
MPI_Graph_create(comm, Nbnodes, index, edges, reorder, comm_graph)
```

IN	comm	communicateur
IN	Nbnodes	nombre de nœuds dans le graphe
IN	index	décrit le degré des nœuds
IN	edges	décrit les arêtes associées au nœud
IN	reorder	true on réordonne les processus ; false on garde l'ordre de comm
OUT	comm_graph	communicateur avec la topologie graphe

# Graphe de processus

Deux autres fonctions sont utiles pour connaître

- Le nombre de voisins, *nneighbors*, pour un processus donné

```
int MPI_Graph_neighbors_count( MPI_Comm comm, int rank,  
int *nneighbors )
```

- La liste des voisins pour un processus donné

```
int MPI_Graph_neighbors( MPI_Comm comm, int rank,  
int maxneighbors, int *neighbors )
```

*maxneighbors* : taille du tableau *neighbors*

*comm* : intracommunicateur avec une topologie de graphe.

# Message Passing Interface

## 1. Manipulation de structures MPI

### 1. Communicateurs

### 2. Opérateurs définis par l'utilisateur

## 2. Topologie MPI

## 3. Types dérivés

# Types prédéfinis

MPI_Datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	CHARACTER
MPI_SHORT	signed short int	INTEGER(2)
MPI_INT	signed int	INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	REAL(4)
MPI_DOUBLE	double	REAL(8)
MPI_LONG_DOUBLE	long double	DOUBLE PRECISION*8

# Types dérivés

## Pour quoi faire ?

Pour le moment, on sait envoyer et recevoir des données qui ont le même type et qui sont contiguës en mémoire.

La construction de types dérivés dans MPI permet d'envoyer et recevoir une partie d'un tableau, une structure, ...

# Types contigus

**MPI\_Type\_contiguous(int count, MPI\_Datatype oldtype, MPI\_Datatype \*newtype)**

IN	count	nombre d'éléments (entier positif)
IN	oldtype	ancien type (MPI_Datatype handle)
OUT	newtype	nouveau type (MPI_Datatype handle)

MPI\_Type\_contiguous crée un nouveau type à partir d'un ensemble homogène de types prédéfinis de données contiguës en mémoire

```
MPI_Type_contiguous( 3, MPI_Float, &newtype )
```





# Manipulation

MPI impose que tous les types de données utilisées pour les communications et les opérations sur les fichiers doivent être enregistrés

➔ Permet d'optimiser la représentation des types

```
int MPI_Type_commit( MPI_Datatype* )
```

```
int MPI_Type_free( MPI_Datatype* )
```

# Exemple

```
/* create a type which describes a line of ghost cells */  
/* buf[1..nxl] set to ghost cells */  
  
int nxl;  
MPI_Datatype ghosts;  
  
MPI_Type_contiguous (nxl, MPI_DOUBLE, &ghosts);  
MPI_Type_commit(&ghosts);  
MPI_Send (buf, 1, ghosts, dest, tag, MPI_COMM_WORLD);  
..  
MPI_Type_free(&ghosts);
```

# Types à pas constants - vecteurs

```
MPI_Type_vector (int count, int blocklength, int stride,  
MPI_Datatype oldtype, MPI_Datatype *newtype);
```

IN	count	nombre de blocs (entier positif)
IN	blocklength	nombre d'éléments dans chaque bloc
IN	stride	nombre d'éléments entre le début de chaque bloc
IN	oldtype	ancien type (MPI_Datatype handle)
OUT	newtype	nouveau type (MPI_Datatype handle)

MPI\_Type\_vector crée un nouveau type basé sur la réplication d'un type de données constitué de blocs régulièrement espacés

# Exemple

```
MPI_Datatype mytype;  
MPI_Type_vector_(3, 4, 5, MPI_DOUBLE, &mytype);  
MPI_Type_commit(&mytype);
```



## Exemple

```
MPI_Datatype mytype;  
MPI_Type_vector_(3, 4, 5, MPI_DOUBLE, &mytype);  
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE 




## Exemple

```
MPI_Datatype mytype;  
MPI_Type_vector_(3, 4, 5, MPI_DOUBLE, &mytype);  
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE 
- blocklength = 4 





# Exemple

```
MPI_Datatype mytype;  
MPI_Type_vector(3, 4, 5, MPI_DOUBLE, &mytype);  
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE 
- blocklength = 4 
- count = 3 

# Exemple






```
MPI_Datatype mytype;  
MPI_Type_vector(3, 4, 5, MPI_DOUBLE, &mytype);  
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE 
- blocklength = 4 
- count = 3 
- stride = 5 



# Exemple

```
MPI_Datatype mytype;  
MPI_Type_vector(3, 4, 5, MPI_DOUBLE, &mytype);  
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE 
- blocklength = 4 
- count = 3 
- stride = 5 
- mytype : 

# Types à pas constants - vecteurs

```
MPI_Type_hvector (int count, int blocklength, MPI_Aint stride,  
MPI_Datatype oldtype, MPI_Datatype *newtype);
```

IN	count	nombre de blocs (entier positif)
IN	blocklength	nombre d'éléments dans chaque bloc
IN	stride	nombre d'octets entre le début de chaque bloc
IN	oldtype	ancien type (MPI_Datatype handle)
OUT	newtype	nouveau type (MPI_Datatype handle)

➔ Même chose que MPI\_Type\_vector sauf que stride est donné en octets

« h » pour heterogeneous

# Types homogène à pas variable

**MPI\_Type\_indexed (int count, int \*array\_of\_blocklengths, int \*array\_of\_displacements, MPI\_Datatype oldtype, MPI\_Datatype \*newtype)**

IN	count	nombre de blocs (entier positif)
IN	a_of_b	nombre d'éléments dans chaque bloc (tableau d'entiers)
IN	a_of_d	espacement entre chaque bloc depuis le début en unité de l'ancien type (tableau d'entiers)
IN	oldtype	ancien type (MPI_Datatype handle)
OUT	newtype	nouveau type (MPI_Datatype handle)




Réplication d'un type de données en une suite de blocs.

Chaque bloc peut

- Contenir un nombre différent d'éléments ;
- Avoir un espacement différent.




# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 




# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0, 




# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0,3 





# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0,3,8 

# Exemple





```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0,3,8 
- mytype : 







# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0,3,8 
- mytype : 





# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0,3,8 
- mytype : 





# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0,3,8 
- mytype : 

# Exemple

```
int blocklength[3] = {2,3,1}
int displacement[3] = {0,3,8}
MPI_Datatype mytype;
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,
&mytype);
MPI_Type_commit(&mytype);
```

- oldtype = MPI\_DOUBLE\_2 
- blocklength = 2,3,1 
- stride = 0,3,8 
- mytype : 

# Types homogène à pas variable

**MPI\_Type\_create\_hindexed** (int count, int \*array\_of\_blocklengths, int \*array\_of\_displacements, MPI\_Datatype oldtype, MPI\_Datatype \*newtype)

IN	count	nombre de blocs (entier positif)
IN	a_of_b	nombre d'éléments dans chaque bloc (tableau d'entiers)
IN	a_of_d	espacement entre chaque bloc depuis le début en unité de l'ancien type
IN	oldtype	ancien type (MPI_Datatype handle)
OUT	newtype	nouveau type (MPI_Datatype handle)

→ Même chose que MPI\_Type\_indexed sauf que a\_of\_d est donné en octets

# Structures quelconques

```
MPI_Type_create_struct (int count, int *array_of_blocklengths,  
MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,  
MPI_Datatype *newtype)
```

IN	count	nombre de blocs (entier positif)
IN	a_of_b	nombre d'éléments dans chaque bloc (tableau d'entiers)
IN	a_of_d	déplacement en <b>octets</b> de chaque bloc (tableau d'Aint)
IN	a_of_t	type des éléments dans chaque bloc (tableau de MPI_Datatype)
OUT	newtype	nouveau type de données (MPI_Datatype handle)

- ➔ Constructeur de type le plus général
- ➔ Permet à chaque bloc d'être la réplique de types différents

## Exemple




```
MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
int blocklength[3] = {2,2,5}
MPI_Aint displacement[3] = {0,14,26}
MPI_Datatype mytype;
MPI_Type_create_struct(3, &blocklength, &displacement, &types, &mytype);
MPI_Type_commit(&mytype);
```

# Exemple

```

MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
int blocklength[3] = {2,2,5}
MPI_Aint displacement[3] = {0,14,26}
MPI_Datatype mytype;
MPI_Type_create_struct(3, &blocklength, &displacement, &types, &mytype);
MPI_Type_commit(&mytype);

```

- oldtype = double, int, short 
- blocklength = 2,2,5 
- stride (bytes)= 0,14,26 



# Exemple

```
MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
```





```
int blocklength[3] = {2,2,5}
```

```
MPI_Aint displacement[3] = {0,14,26}
```

```
MPI_Datatype mytype;
```

```
MPI_Type_create_struct(3, &blocklength, &displacement, &types, &mytype);
```

```
MPI_Type_commit(&mytype);
```

- oldtype = double, int, short 
- blocklength = 2,2,5 
- stride (bytes)= 0,14,26 
- mytype : 

# Exemple

```
MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
```





```
int blocklength[3] = {2,2,5}
```

```
MPI_Aint displacement[3] = {0,14,26}
```

```
MPI_Datatype mytype;
```

```
MPI_Type_create_struct(3, &blocklength, &displacement, &types, &mytype);
```

```
MPI_Type_commit(&mytype);
```

- oldtype = double, int, short 
- blocklength = 2,2,5 
- stride (bytes)= 0,14,26 
- mytype : 

# Exemple

```
MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
```





```
int blocklength[3] = {2,2,5}
```

```
MPI_Aint displacement[3] = {0,14,26}
```

```
MPI_Datatype mytype;
```

```
MPI_Type_create_struct(3, &blocklength, &displacement, &types, &mytype);
```

```
MPI_Type_commit(&mytype);
```

- oldtype = double, int, short 
- blocklength = 2,2,5 
- stride (bytes)= 0,14,26 
- mytype : 

# Exemple

```
MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
```





```
int blocklength[3] = {2,2,5}
```

```
MPI_Aint displacement[3] = {0,14,26}
```

```
MPI_Datatype mytype;
```

```
MPI_Type_create_struct(3, &blocklength, &displacement, &types, &mytype);
```

```
MPI_Type_commit(&mytype);
```

- oldtype = double, int, short 
- blocklength = 2,2,5 
- stride (bytes)= 0,14,26 
- mytype : 

# Exemple

```
MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
```





```
int blocklength[3] = {2,2,5}
```

```
MPI_Aint displacement[3] = {0,14,26}
```

```
MPI_Datatype mytype;
```

```
MPI_Type_create_struct(3, &blocklength, &displacement, &types, &mytype);
```

```
MPI_Type_commit(&mytype);
```

- oldtype = double, int, short 
- blocklength = 2,2,5 
- stride (bytes)= 0,14,26 
- mytype : 

# Exemple

```
struct Partstruct{
    char class;
    double d[6];
    char b[7];
}

struct Partstruct particle[1000];
int dest, tag;
MP_Comm comm;

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_CHAR};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3] = {0, sizeof(char), sizeof(char)+6*sizeof(double)};

MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);

MPI_Send(particle, 1000, Particletype, dest, tag, comm);
```

# Informations sur les types

**MPI\_Get\_address (void \*location, MPI\_Aint \*address)**

IN location (*location in caller memory*)

OUT address (*address of location*)

➔ Pour calculer les pas

**MPI\_Type\_size(MPI\_Datatype datatype, int \*size)**

IN datatype (*datatype*)

OUT size (*datatype size*)

➔ Retourne le nombre d'octets du datatype sans les espacements

**MPI\_Type\_get\_extent(MPI\_Datatype datatype, MPI\_Aint \*lb, MPI\_Aint \*extent)**

IN datatype (*datatype you are querying*)

OUT lb (*lower bound of datatype*)

OUT extent (*extent of datatype*)

➔ Retourne la limite inf et l'étendue du datatype

**A vous de jouer!**