Cours Calcul Intensif - MPI

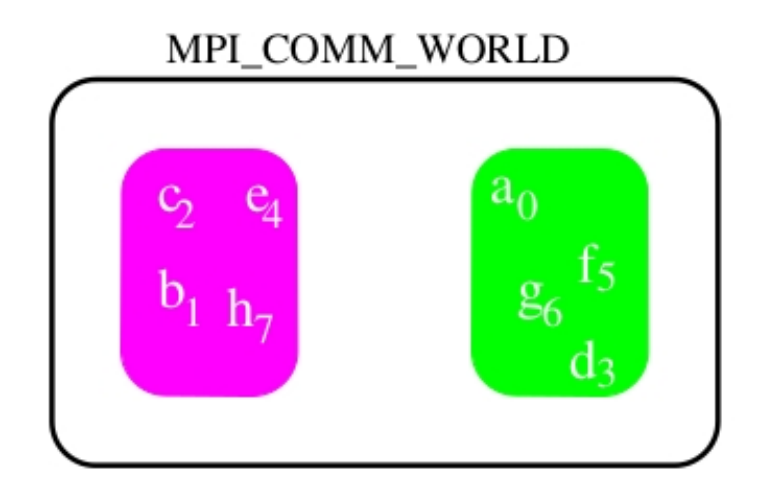
Hélène Coullon , Sophie Robert, Sébastien Limet

21 novembre 2012

Communicateurs

Communicateurs

Il s'agit de partitionner un ensemble de processus MPI 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.

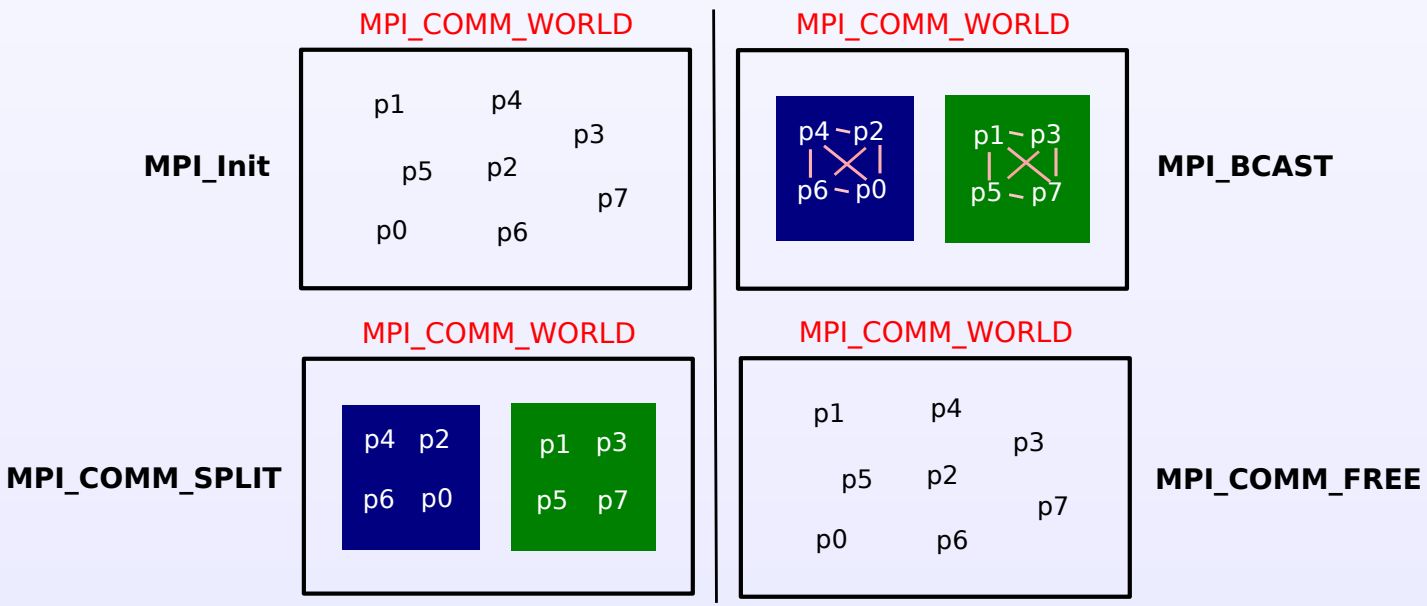


Création des communicateurs

- On ne peut créer un communicateur qu'à partir d'un autre communicateur
- Le communicateur par défaut MPI_COMM_WORLD permet de créer d'autres communicateurs
- MPI_COMM_WORLD est créé à MPI_Init et détruit à MPI_Finalize

Exemple

- 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 message qu'aux processus de rang impair



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
- 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

Routines MPI

- Dans MPI, il existe diverses routines 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 peuvent être gérés dynamiquement et supprimés avec **MPI_Comm_free**

MPI_Comm_split

Le sous-programme MPI_Comm_split permet de partitionner un communicateur donné en autant de communicateurs que l'on veut.

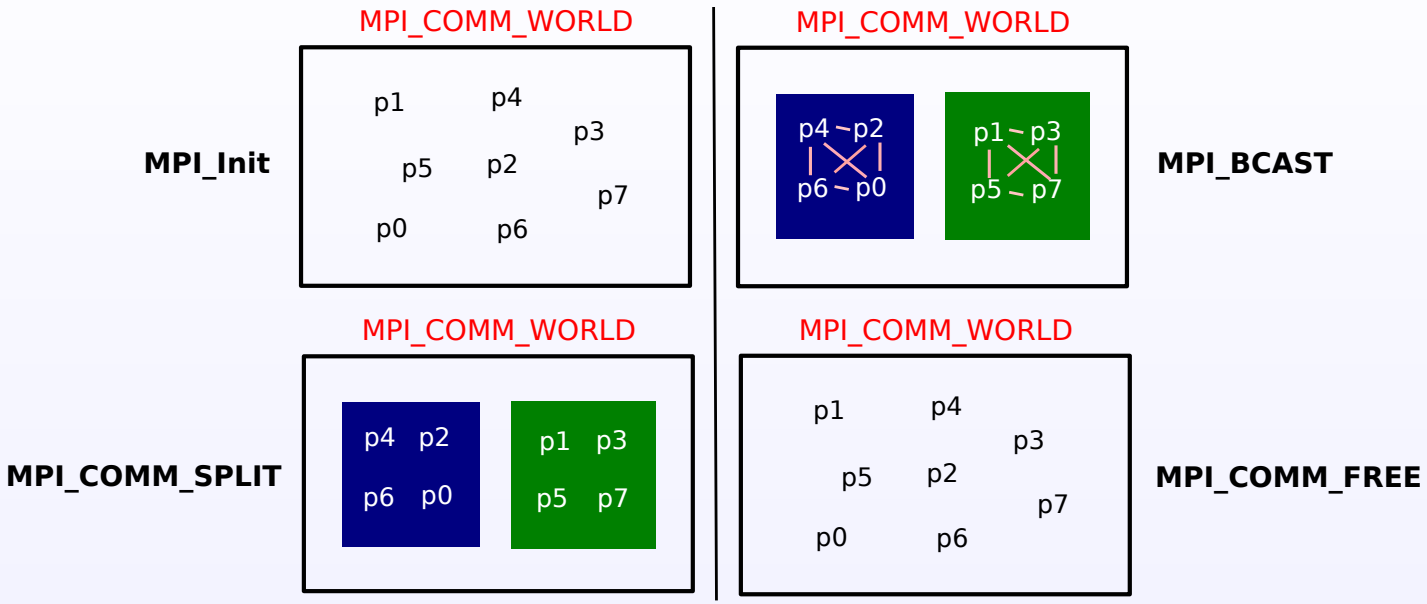
Prototype

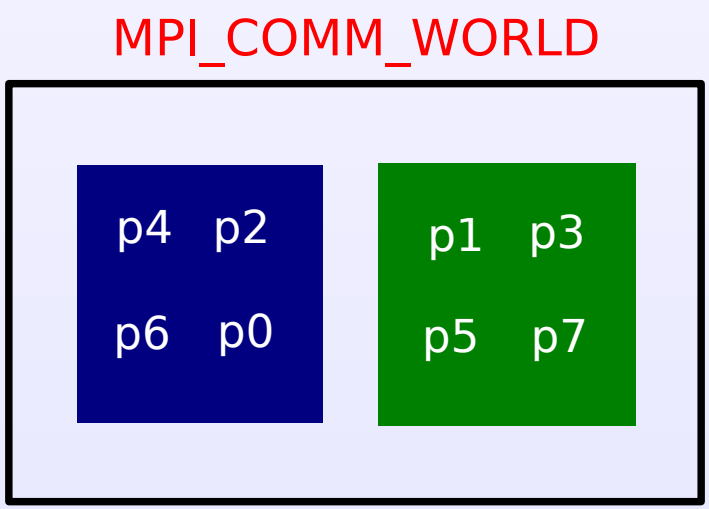
```
int MPI_Comm_split(MPI_Comm comm, int color, int key
                   , MPI_Comm *newcomm)
```

Arguments

- MPI_Comm **comm** : le communicateur à partir duquel on fait le partitionnement de processus
- int **color** : la couleur du processus, les processus de même couleur seront dans le même communicateur
- int **key** : la clé du processus, qui sera utilisée pour obtenir le nouvel identifiant du processus dans le nouveau communicateur
- MPI_Comm ***newcomm** : pointeur sur le nouveau communicateur obtenu

Exemple





- 2 couleurs
- que ce passe-t-il si on met les même clés que dans MPI_COMM_WORLD ?

Exemple

Topologies

Topologies de processus

- Dans la plupart des applications, plus particulièrement dans les méthodes de décomposition de domaine, on fait correspondre le domaine de calcul à la grille de processus. Dans ce cadre, il est intéressant de pouvoir disposer les processus suivant une topologie régulière.
- MPI permet de définir des topologies virtuelles du type cartésien ou graphe
  - Topologies 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
  - Topologies de type graphe : généralisation à des topologies plus complexes

Topologies de type cartésien

Principes

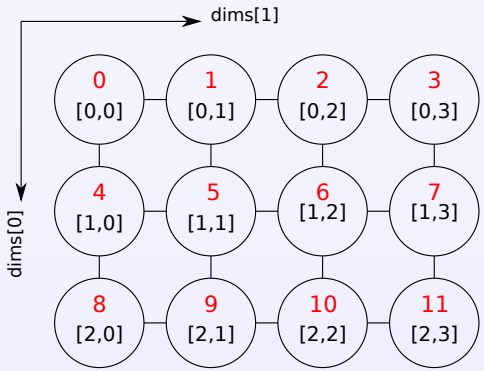
- La grille de processus est définie par
  - Sa dimension
  - Sa périodicité
  - Le nombre de processus dans chaque dimension

Routines

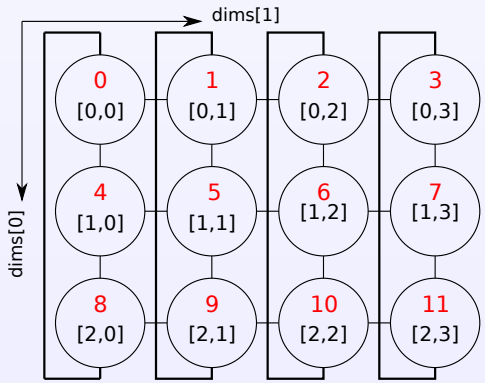
- Création d'une topologie cartésienne
- Création des bonne taille de dimensions suivant le nombre de processus
- Rang d'un processus dans une topologie cartésienne
- Coordonnées (x,y,z) d'un processus dans la topologie
- Partitionner un communicateur de topologie cartésienne en sous-groupes

Routine MPI_Cart_create (1)

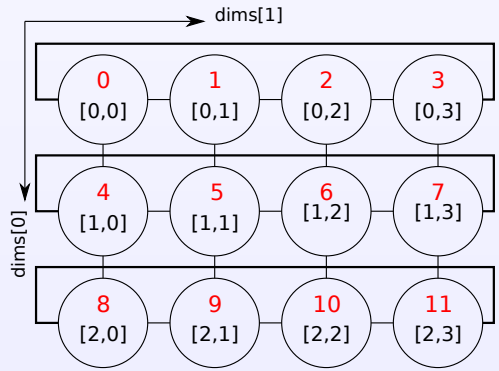
- Cette routine permet de créer une topologie cartésienne
- La routine est collective, elle concerne donc l'ensemble des processus appartenant à l'ancien communicateur



Non périodique



Lignes périodiques



Colonne périodiques

Routine MPI_Cart_create (2)

Prototype

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods,
                    int reorder, MPI_Comm *comm_cart)
```

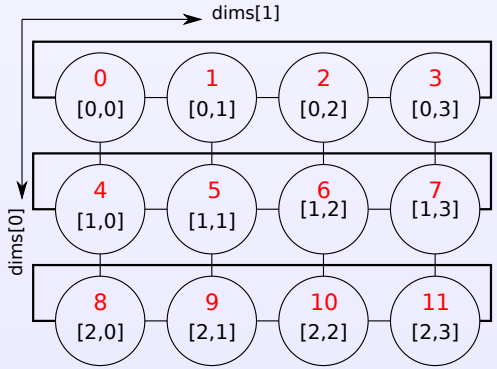
Paramètres

1. MPI_Comm **old_comm** : Ancien communicateur (Le plus simple : MPI_COMM_WORLD)
2. int **ndims** : Nombre de dimensions
3. int ***dim_size** : Nombre de processus dans chaque dimension
4. int ***periods** : Tableau indiquant la périodicité pour chaque dimension
5. int **reorder** : Le rang des processus peut-il être modifié ou non
6. MPI_Comm ***new_comm** : Nouveau communicateur avec la structure cartésienne

Routine MPI_Cart_create (3)

Exemple : Création une topologie cartésienne de 3×4

```
int ndims=2, dims[2], periods[2], reorder;
dims[0]=3; dims[1]=4;
periods[0]=FALSE; periods[1]=TRUE;
reorder=TRUE;
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims,
                periods, reorder, &new_comm);
```



Routine MPI_Dims_create (1)

Cette routine permet d'obtenir une répartition automatique et idéale des processus suivant le nombre de dimensions souhaitées.

Prototype

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

Paramètres

1. int **nnodes** : Nombre de processus dans la grille
2. int **ndims** : Nombre de dimensions souhaitées
3. int ***dims** : Nombre de processus par dimension obtenu

Routine MPI_Dims_create (2)

Exemple : Création une topologie cartésienne avec répartition automatique des processus dans les dimensions

```
int nb_procs;
MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
int ndims=2, periods[2], dims[2];
int reorder=TRUE;
periods[0]=FALSE; periods[1]=FALSE;
dims[0]=dims[1]=0;

MPI_Dims_create(nb_procs,ndims,dims);
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims,
                periods, reorder, &new_comm);
```

Routine MPI_Cart_rank (1)

Cette routine permet de connaître le rang du processus associé aux coordonnées données.

MPI_Cart_rank

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

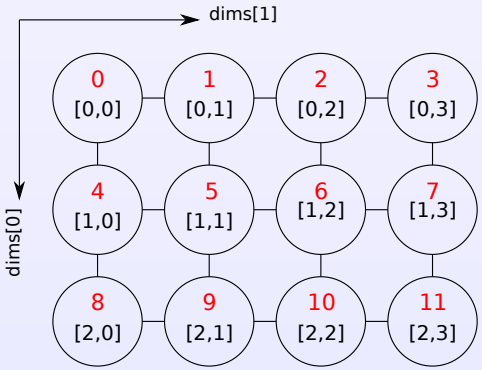
Paramètres

1. MPI_Comm **comm** : Communicateur de la structure cartésienne
2. int ***coords** : Les coordonnées cartésiennes desquelles on souhaite récupérer le numéro de processus
3. int ***rank** : Rang de processus associé aux coordonnées spécifiées

Routine MPI_Cart_rank (2)

Processus 0 calcule le rang d'un processus avec ses coordonnées

```
int coords[2], rank;
coords[0]= 1; coords[1]= 2;
if (pid==0){
  MPI_Cart_rank(new_comm,coords,&rank);
  printf("Proc. à (%d,%d) a le rang %d",
         coords[0], coords[1], rank);
}
```



Résultat

```
Proc. à (1, 2) a le rang 6
```

Routine MPI_Cart_coords (1)

Cette routine fournit les coordonnées d'un processus de rang donné dans la grille.

Prototype

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int maxdims, int *coords)
```

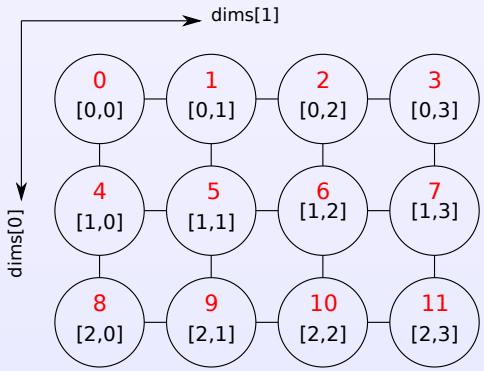
Paramètres

1. MPI_Comm **comm** : Communicateur de la structure cartésienne
2. int **rank** : Rang d'un processus au sein du communicateur
3. int **maxdims** : Taille des coordonnées coords (nombre de dimensions)
4. int ***coords** : Les coordonnées cartésiennes du processus spécifié

Routine MPI_Cart_coords (2)

Proc. 0 calcule les coordonnées de Proc. 10

```
if (pid==0){
  int coords[2], rank=10;
  MPI_Cart_coords(new_comm, rank,
                  ndims, coords);
  printf("Proc. %d aux coords. [%d,%d]",
         rank, coords[0], coords[1]);
}
```



Résultat

```
Proc. 10 aux coords. [2,2]
```

Routine MPI_Cart_shift (1)

Cette routine permet de connaître le rang **des voisins** d'un processus **dans une direction donnée**.

Prototype

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int displ, int *source,
                   int *dest)
```

Paramètres

1. MPI_Comm **comm** : Communicateur de la structure cartésienne
2. int **direction** : Direction de voisinage souhaitée dans les coordonnées cartésiennes, directions $\in$ [0, n-1] pour un maillage cartésien à n dimensions

Routine MPI_Cart_shift (1)

Prototype

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int displ, int *source,
                   int *dest)
```

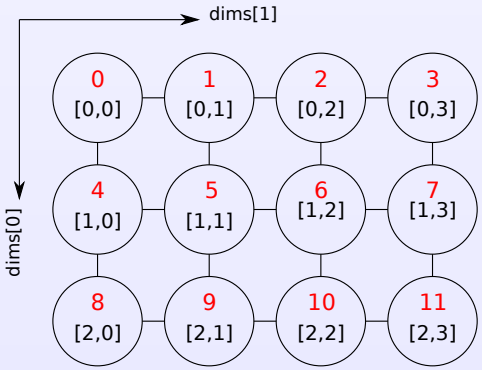
Paramètres

1. int **displ** : Taille du déplacement en voisinage (exemple : voisins à 2 cases horizontalement : droite vers gauche ou gauche vers droite) : >0 upward, <0 downward
2. int ***source** : Rang du processus voisin source dans la direction et le sens indiqués
3. int ***dest** : Rang de processus voisin destination dans la direction et le sens indiqués

Routine MPI_Cart_shift (2)

Proc. 6 cherche ses voisins dans la direction 0

```
int displ = 1;
int nup, nlow;
MPI_Cart_shift(new_comm, 0, displ,
               &nup, &nlow);
printf("Proc. %d a voisin au-dessus %d\n",
       pid, nup);
printf("Proc. %d a voisin au-dessous %d\n",
       pid, nlow);
```



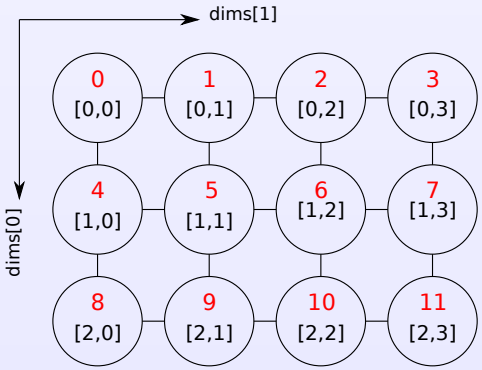
Résultat

```
Proc. 6 a voisin au-dessus 2
Proc. 6 a voisin au-dessous 10
```

Routine MPI_Cart_shift (3)

Proc. 6 cherche ses voisins dans la direction 1

```
int displ = -1;
int nleft, nright;
MPI_Cart_shift(new_comm, 1, displ,
               &nright, &nleft);
printf("Proc. %d a voisin à droite %d\n",
       pid, nright);
printf("Proc. %d a voisin à gauche %d\n",
       pid, nleft);
```



Résultat

```
Proc. 6 a voisin à droite 7
Proc. 6 a voisin à gauche 5
```

Routine MPI_Cart_get

Cette routine donne les informations sur le communicateur donné de dimensions maxdims.

Prototype

```
int MPI_Cart_get(MPI_Comm comm, int maxdims,
                 int *dims, int *periods,
                 int *coords)
```

Paramètres

1. MPI_Comm comm : Communicateur de la structure cartésienne
2. int maxdims : Longueur du vecteur de dimensions (nombre de dimensions)

Routine MPI_Cart_get

Prototype

```
int MPI_Cart_get(MPI_Comm comm, int maxdims,
                 int *dims, int *periods,
                 int *coords)
```

Paramètres

1. int *dims : Tableau d'entier du nombre de processus pour chaque dimension
2. int *periods : Tableau d'entier des périodicités pour chaque dimension
3. int *coords : Coordonnées du processus appelant dans la topologie cartésienne

Routine MPI_Cartdim_get

Cette routine retourne le nombre de dimensions pour le communicateur indiqué dans la structure cartésienne.

Prototype

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

Paramètres

1. MPI_Comm comm : Communicateur de la structure cartésienne
2. int *ndims : Nombre de dimensions de la structure cartésienne du communicateur

Routine MPI_Cart_sub (1)

Principes

- Cette routine partitionne un communicateur en sous-groupes
- Ces sous-groupes forment sous-grilles cartésiennes avec leurs dimension plus petites que l'ancienne
- L'intérêt majeur est de pouvoir effectuer des opérations collectives restreintes à un sous-ensemble de processus appartenant à :
  1. une même ligne (ou colonne), si la topologie initiale est 2D
  2. un même plan, si la topologie initiale est 3D

Routine MPI_Cart_sub (2)

Prototype

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
                 MPI_Comm *comm_new)
```

Paramètres

1. MPI_Comm **comm** : Communicateur de la structure cartésienne
2. int ***remain_dims** : Quelles directions sont conservées dans la sous-grille
3. MPI_Comm ***comm_new** : Retourne un des nouveaux communicateurs créés qui contient le processus appelant

Routine MPI_Cart_sub (3)

Supposons que MPI_Cart_create a défini une grille de (2×3×4).

remain_dims = (true, false, true)

- Nous avons 3 nouvelles sous-grilles
- Chacune a 8 processus avec une topologie cartésienne de 2×4

remain_dims = (false, false, true)

- Nous avons 6 nouvelles sous-grilles
- Chacune a 4 processus avec une topologie cartésienne 1D

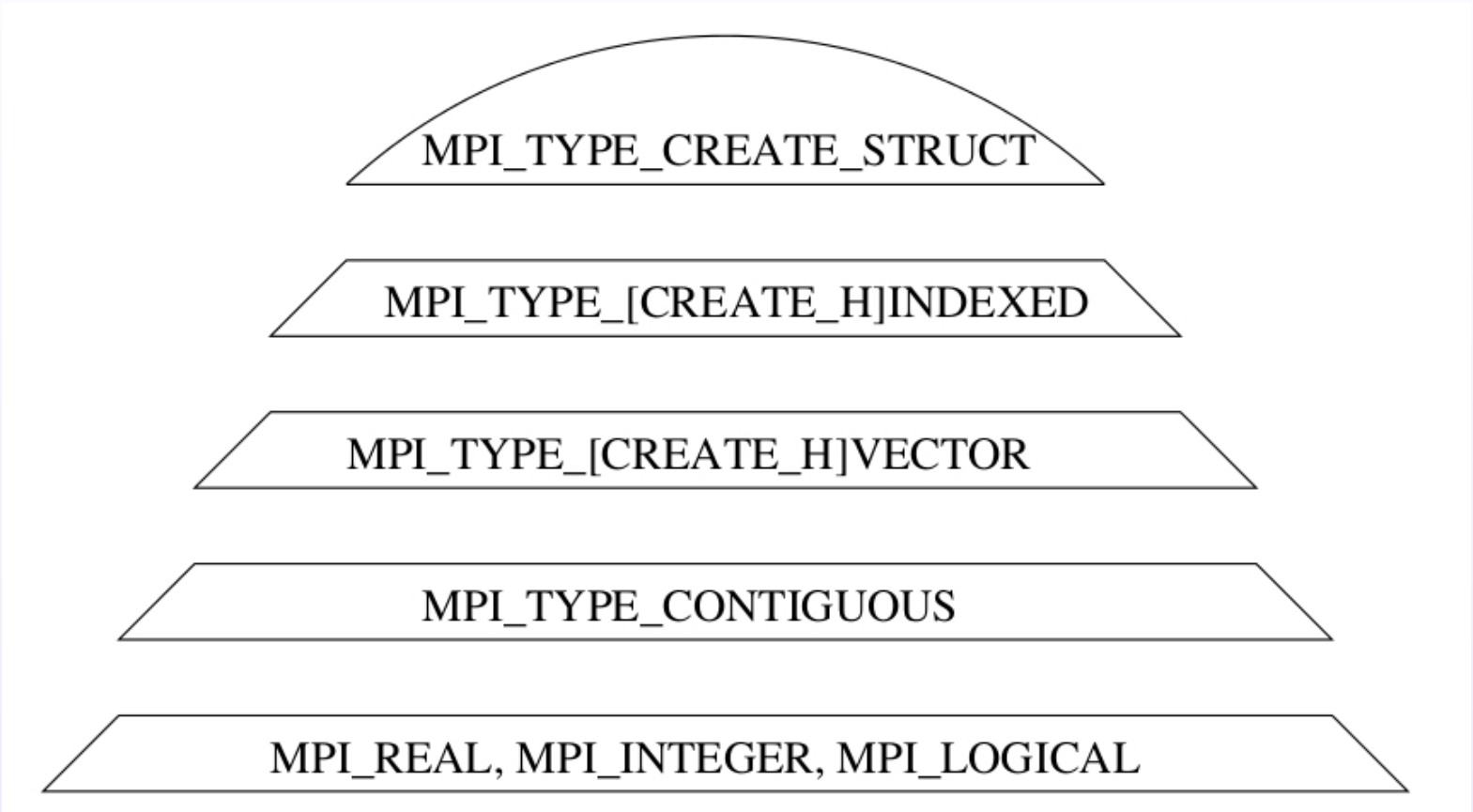
Exemple

Types dérivés

Types de données dérivés

- Dans les communications, les données échangées sont typées : MPI_INT, MPI_FLOAT 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_hvector** etc.
- 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 nom pour définir un autre type dérivé, on doit au préalable le libérer avec le sous-programme **MPI_Type_free**.

Types de données dérivés



MPI_Type_commit

Cette routine permet de commiter dans MPI le nouveau type créer et de pouvoir ensuite l'utiliser dans les communications.

Prototype

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

Paramètres

MPI_Datatype ***datatype*** : nouveau type à commiter dans MPI

MPI_Type_free

Cette routine permet à l'inverse de libérer le type de MPI, après cet appel le type ne peut plus être utilisé.

Prototype

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

Paramètres

MPI_Datatype ***datatype*** : nouveau type à commiter dans MPI

MPI_Type_contiguous

Cette routine crée une structure de données à partir d'un ensemble homogène de type préexistant de données contigües en mémoire.

Prototype

```
int MPI_Type_contiguous(int count,
                        MPI_Datatype old_type,
                        MPI_Datatype *new_type_p)
```

Paramètres

1. int **count** : Nombre d'éléments
2. MPI_Datatype **old_type** : Ancien type de données
3. MPI_Datatype ***new_type_p** : Pointeur sur le nouveau type de données

Structure du code

```
MPI_Datatype new_type;
MPI_Type_contiguous(NB_ELEM,MPI_INT,&new_type);
MPI_Type_commit(&new_type);
...
MPI_Send(&a,1,new_type,nb_proc,ETIQUETTE,
         MPI_COMM_WORLD);
...
MPI_Recv(&a,1,new_type,nb_proc,ETIQUETTE,
         MPI_COMM_WORLD,&statut);
MPI_Type_free(&new_type);
```

Exemple

MPI_Type_vector

Cette routine crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donnée en nombre d'éléments.

Prototype

```
int MPI_Type_vector(int count,
                    int blocklength,
                    int stride,
                    MPI_Datatype old_type,
                    MPI_Datatype *newtype_p)
```

MPI_Type_vector

Prototype

```
int MPI_Type_vector(int count,
                    int blocklength,
                    int stride,
                    MPI_Datatype old_type,
                    MPI_Datatype *newtype_p)
```

Paramètres

1. int **count** : Nombre de blocs d'éléments
2. int **blocklength** : Nombre d'éléments dans chaque bloc de données
3. int **stride** : Nombre d'éléments entre chaque bloc de données
4. MPI_Datatype **old_type** : Ancien type de données
5. MPI_Datatype ***new_type_p** : Pointeur sur le nouveau type de données

Structure du code

```
MPI_Datatype new_type;
MPI_Type_vector(NB_ELEM,1,PAS,MPI_INT,&new_type);
MPI_Type_commit(&new_type);
...
MPI_Send(&a,1,new_type,nb_proc,ETIQUETTE,
         MPI_COMM_WORLD);
...
MPI_Recv(&a,1,new_type,nb_proc,ETIQUETTE,
         MPI_COMM_WORLD,&statut);
```

Exemples

MPI_Type_create_hvector

Cette routine crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donnée en nombre d'octets.

Prototype

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

MPI_Type_create_hvector

Prototype

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

Paramètres

1. int **count** : Nombre de blocs d'éléments
2. int **blocklength** : Nombre d'éléments dans chaque bloc de données
3. MPI_Aint **stride** : Nombre d'octets entre chaque bloc de données
4. MPI_Datatype **old_type** : Ancien type de données
5. MPI_Datatype ***new_type_p** : Pointeur sur le nouveau type de données

Autres types de données dérivés et notes

Autres 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_hindexed a la même fonctionnalité mais le pas séparant deux blocs de données est exprimé en octets.

Notes concernant MPI_Type_hvector et MPI_Type_hindexed

- Ces instructions sont utiles lorsque le type générique n'est pas un type de base MPI mais un type plus complexe construit avec les routines MPI vues précédemment. On ne peut alors pas exprimer le pas en nombre d'éléments du type générique.
- Il faut utiliser les routines **MPI_Type_size** ou **MPI_Type_get_extent** pour obtenir de façon portable la taille du pas en nombre d'octets.

Fin du cours MPI - Next : MPI 2.0 !