# Introduction au parallélisme par échange de message via MPI

## Master DFE – année 2020/2021

Mathieu Lobet, Maison de la Simulation
Mathieu.lobet@cea.fr

# Introduction au parallélisme par échange de message via MPI

## 1) Description de l'approche

# Cours et matériel supplémentaires sur internet

Selon moi, le cours le plus complet sur MPI en français et anglais:
http://www.idris.fr/formations/mpi/

Les implémentations fournissent en général une documentation en ligne complète :
https://www.open-mpi.org/
https://www.mcs.anl.gov/research/projects/mpi/learning.html
http://mpi.deino.net/

# Information concernant le cours

Il existe des implémentations de MPI pour Fortran95, Fortran08, C, C++ et python.

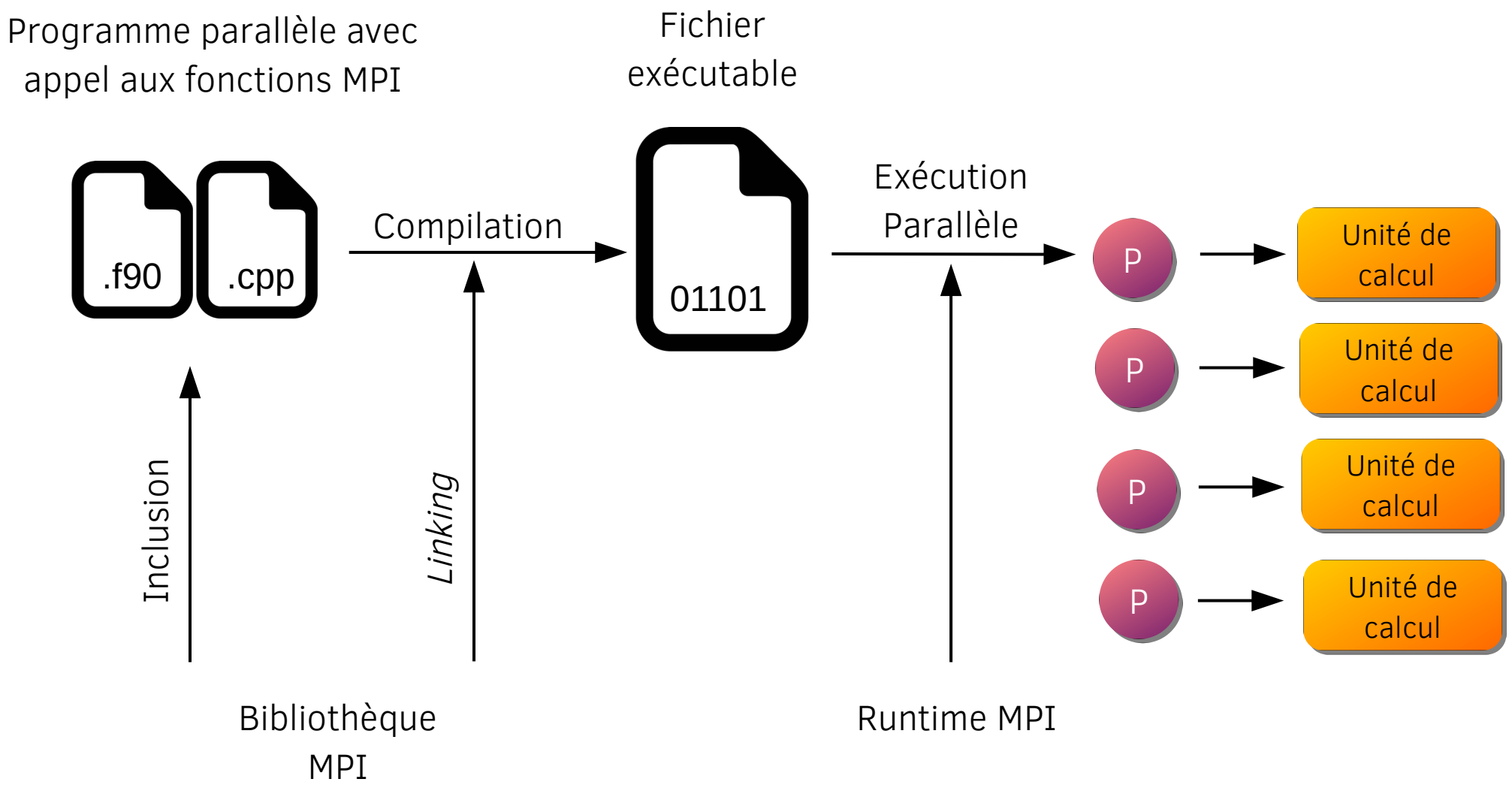
La syntaxe diffère pour chaque langage. Le cours s'axe sur la version **Fortran95 et C**.

On programmera **en C++ en utilisant l'API C**.

Il existe également une version C++ avec classe pour les communicateurs ainsi qu'une version adaptée au Fortran moderne.

La philosophie reste la même peut importe le langage choisi.

# La chaîne de compilation et d'exécution d'un programme MPI



# Implémentations MPI

Implémentations libres que vous pouvez vous procurer pour vos ordinateurs :

- OpenMPI
- MPICH
- Deino (Windows)

Implémentation propriétaire :

- IntelMPI

Disponible en Fortran, C, C++, Python

L'installation peut se faire par les sources ou via `apt-get install`

# Compilation d'un programme MPI (Fortran)

La compilation fait appel au wrapper MPI mpif90. Le wrapper utilise le compilateur Fortran par défaut (par exemple gfortran) en y ajoutant des options de compilation supplémentaires et les chemins vers la bibliothèque MPI.

Pour connaître le contenu du wrapper :

```
> mpif90 -show
```

Pour compiler :

```
> mpif90 program.f90 -o executable
```

# Compilation d'un programme MPI (C++)

La compilation fait appel au wrapper MPI mpic++. Le wrapper utilise le compilateur C++ par défaut (par exemple g++) en y ajoutant des options de compilation supplémentaires et les chemins vers la bibliothèque MPI.

Pour connaître le contenu du wrapper :

```
> mpic++ -show
```

Pour compiler :

```
> mpic++ program.cpp -o executable
```

# Exécution d'un programme MPI

- L'exécution se fait par l'intermédiaire de la commande mpirun.
- `-np` représente le nombre de processus MPI à lancer.
- Si le nombre de processus MPI est inférieur au nombre d'unités de calcul disponible, chaque processus est exécuté par une unité indépendante
- Il est possible de lancer plus de processus MPI que d'unités de calcul, dans ce cas, les ressources sont partagées.
- En OpenMPI, il faut spécifier « --oversubscribe » pour activer cette possibilité

Pour exécuter votre code en ligne de commande :

```
> mpirun -np 6 ./executable
```

# Démarrage d'un programme MPI : notion de communicateur

*Communicator*

- Un communicateur est un ensemble de processus MPI capable de communiquer entre eux.
- Au sein d'un communicateur, chaque processus MPI est représenté par un rang (*rank*) unique sous forme d'un entier.
- Le communicateur par défaut regroupe l'ensemble des processus et se nomme MPI_COMM_WORLD.

```
> mpirun -np 4 ./executable
```


MPI_COMM_WORLD : Communicateur composé de 4 rangs

# Démarrage d'un programme MPI (Fortran) : inclure MPI

La première étape consiste à ne pas oublier d'inclure le module MPI (header en C/C++)

.f90

```
Program test

Use mpi

...

End program
```

.cpp

```
#include <mpi.h>

int main( int argc, char *argv[] )
{

}
```

# Démarrage d'un programme MPI (Fortran) : initialiser et finaliser MPI

- MPI est une bibliothèque qui fonctionne par appel à des fonctions (subroutine en Fortran)
- La deuxième étape importante est l'initialisation de MPI
- Il ne faut pas oublier de finaliser pour finir son programme proprement

.f90

```
Program test

Use mpi

Implicit none
Integer :: ierror

Call MPI_INIT(ierror)

...

Call MPI_FINALIZE(ierror)

End program
```

# Démarrage d'un programme MPI (C++) : initialiser et finaliser MPI

- MPI est une bibliothèque qui fonctionne par appel à des fonctions (subroutine en Fortran)
- La deuxième étape importante est l'initialisation de MPI
- Il ne faut pas oublier de finaliser pour finir son programme proprement

.cpp

```
#include <mpi.h>

int main( int argc, char *argv[] )
{
    int ierror ;

    ierror = MPI_Init() ;

    ...

    ierror = MPI_Finalize() ;

}
```

# Démarrage d'un programme MPI : initialiser MPI

- L'initialisation se fait avec la fonction `MPI_INIT`.
- Toute fonction MPI renvoie en dernier argument un code d'erreur noté ici `ierror`.

.f90

```
Call MPI_INIT(ierror)
```

.cpp

```
Ierror = MPI_INIT() ;
```

- Le code d'erreur permet si besoin de vérifier qu'un appel s'est bien déroulé

https://www.open-mpi.org/doc/v1.8/man3/MPI_Init.3.php

# Démarrage d'un programme MPI (Fortran) : finaliser MPI

- La finalisation se fait avec la fonction `MPI_FINALIZE`.
- Elle est appelée à la toute fin du programme

.f90

```
Call MPI_FINALIZE(ierror)
```

.cpp

```
Ierror = MPI_Finalize() ;
```

# Démarrage d'un programme MPI (Fortran) : récupérer le nombre de rangs

- Le nombre de rangs dans le communicateur **MPI_COMM_WORLD** se récupère via la fonction **MPI_COMM_SIZE** : c'est le nombre total de processus demandé.

.f90

```
Integer :: number_of_ranks

Call MPI_COMM_SIZE(MPI_COMM_WORLD, number_of_ranks, ierror)
```

.cpp

```
Int number_of_ranks ;

Ierror = MPI_Comm_size(MPI_COMM_WORLD, &number_of_ranks) ;
```

- **MPI_COMM_WORLD** : communicateur (ici celui par défaut)
- **Number_of_ranks** : entier renvoyé contenant le nombre de rangs MPI

https://www.open-mpi.org/doc/current/man3/MPI_Comm_size.3.php

# Démarrage d'un programme MPI (Fortran) : récupérer le rang de chaque processus MPI

- Chaque processus récupère son rang dans le communicateur `MPI_COMM_WORLD` via la fonction `MPI_COMM_RANK`.

.f90

```
Integer :: rank

Call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
```

.cpp

```
int rank ;

Ierror = MPI_Comm_rank(MPI_COMM_WORLD, &rank) ;
```

- `MPI_COMM_WORLD` : communicateur (ici celui par défaut)
- `rank` : entier renvoyé désignant le rang du processus qui appelle la fonction

https://www.open-mpi.org/doc/v3.0/man3/MPI_Comm_rank.3.php

# Mesure du temps : MPI_WTIME (fortran)

- `MPI_WTIME` permet de récupérer le temps écoulé sur le processus courant en seconde

.f90

```
Real :: time
Time = MPI_WTIME()
```

- Par deux appels et une soustraction, cette fonction permet de déterminer le temps passer dans une section du code

.f90

```
Real :: time
time = MPI_WTIME()

! Des calculs...
…

! Ce temps est le temps passé entre les deux appels à MPI_WTIME
time = MPI_WTIME() - time
```

https://www.open-mpi.org/doc/v4.0/man3/MPI_Wtime.3.php

# Mesure du temps : MPI_Wtime (C++)

- `MPI_Wtime` permet de récupérer le temps écoulé sur le processus courant en seconde


.cpp

```
double time ;
Time = MPI_Wtime() ;
```

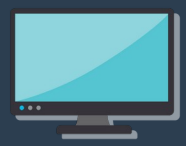
- Par deux appels et une soustraction, cette fonction permet de déterminer le temps passer dans une section du code


.cpp

```
double time ;
time = MPI_Wtime() ;

! Des calculs...
…

! Ce temps est le temps passé entre les deux appels à MPI_Wtime
time = MPI_Wtime() - time ;
```


https://www.open-mpi.org/doc/v4.0/man3/MPI_Wtime.3.php

# Exercice n°1 : votre premier programme MPI



- Rendez vous sur le Gitlab des exercices : https://gitlab.maisondelasimulation.fr/mlobet/cours-hpc-m2-dfe
- Télécharger les exercices sur votre session de travail
- Décompressez l'archive en ligne de commande

```
> tar xvf archivedossier.tar
```

- Rendez vous dans le dossier de l'exercice n°1 appelé `1_initialization`

```
> cd exercices/mpi/1_initialization
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit...)

Vous pouvez lire le README directement depuis le Gitlab et c'est plus confortable comme ça.

# Différencier du code pour des processus donnés

Le programme s'exécute simultanément autant de fois qu'il y a de processus parallèle : chaque ligne de code est appelée par chaque processus.

Pour faire en sorte que certaines portions de code soient réservées à certains processus, on utilise des conditions **`if`** avec le numéro de rang comme condition.

.f90

```
If (rank == 1) then
  ! Cette portion de code ne sera exécutée que par le rang 1
  ...
End if
```

.cpp

```
if (rank == 1) {
  ! Cette portion de code ne sera exécutée que par le rang 1
  ...
}
```

# Votre premier programme MPI

A ce stade du cours, vous savez maintenant :

- Écrire un programme parallèle simple
- Compiler un programme MPI
- Exécuter un programme MPI
- Récupérer le nombre de rangs et le rang de chaque processus

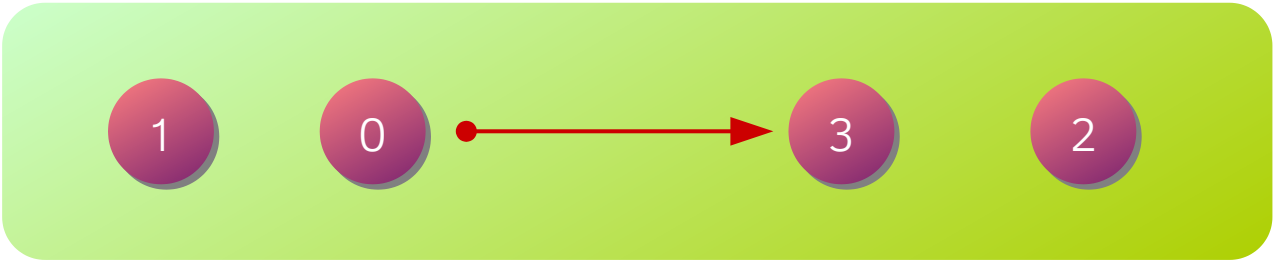
# Introduction au parallélisme par échange de message via MPI

## 2) Les communications point à point bloquantes

# Notion de communication point à point

*Point to point communication*

- L'échange de message constitue la base du concept MPI
- L'échange de message se décompose toujours en deux étapes :
  - Envoi : Un processus envoie un message à un processus destinataire en spécifiant le rang
  - Réception : Un processus doit explicitement recevoir le message en connaissant le rang de l'expéditeur




Le processus de rang 0 envoie un message au processus de rang 3

Le processus de rang 3 reçoit un message du processus de rang 0

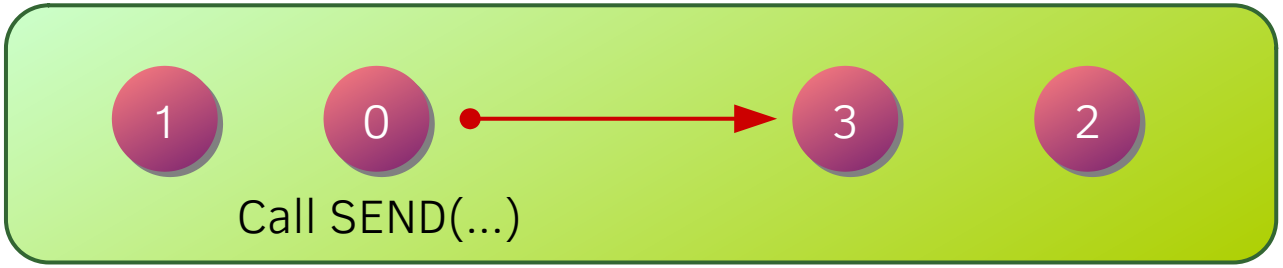
# Notion de communication point à point : Envoi de données via MPI_SEND

*Point to point communication : MPI_SEND*

- MPI_SEND est la fonction appelée par le processus expéditeur

.f90

```
MPI_SEND(message, size, data_type, destination_rank, tag,
communicator, ierror)
```




https://www.open-mpi.org/doc/v1.8/man3/MPI_Send.3.php

# Notion de communication point à point : Envoi de données via MPI_SEND (Fortran)

*Point to point communication : SEND*

- MPI_SEND est la fonction appelée par le processus expéditeur


```
MPI_SEND(message, size, data_type, destination_rank, tag,
communicator, ierror)
```

- `Message` : la variable contenant le message à envoyer (booléen, entier, double, caractère, chaîne, tableau, structure plus complexe…)
- `Size` : nombre d'éléments constituant le message (> 1 uniquement pour une chaîne ou un tableau)
- `data_type` : type de variable utilisée pour le message (`MPI_INTEGER` pour les integer, `MPI_DOUBLE_PRECISION` pour les `real(8)`, `MPI_REAL` pour les real(4)…)
- `Tag` : numéro attribué à la communication si plusieurs coms vers le même processus
- `destination_rank` : rang du processus destinataire

https://linux.die.net/man/3/mpi_real

# Notion de communication point à point : Envoi de données via MPI_Send (C/C++)

*Point to point communication : MPI_Send*

- MPI_Send est la fonction appelée par le processus expéditeur


```
MPI_Send(message, size, data_type, destination_rank, tag,
communicator)
```

- `Message (const void *)` : la variable contenant le message à envoyer (booléen, entier, double, caractère, chaîne, tableau, structure plus complexe…)
- `Size (int)` : nombre d'éléments constituant le message (> 1 uniquement pour une chaîne ou un tableau)
- `data_type` : type de variable utilisée pour le message (`MPI_INT` pour les integer, `MPI_DOUBLE` pour les `double`, `MPI_FLOAT` pour les `float`…)
- `Tag (int) :` numéro attribué à la communication si plusieurs coms vers le même processus
- `destination_rank (int)` : rang du processus destinataire

https://www.open-mpi.org/doc/v1.8/man3/MPI_Send.3.php

# Tag MPI

La notion de tag permet de différencier des communications mais cet aspect ne sera pas exploité dans ce cours.

Une valeur de tag par défaut peu être donnée ou l'utilisation du paramètre `MPI_ANY_TAG` permet d'ignorer l'utilisation de ce dernier.

# Exemple d'utilisation de MPI_SEND

*Point to point communication : MPI_SEND*

- Envoi d'un message de type real(8) au processus de rang 3 par le processus de rang 2

.f90

```
Real(8) :: message
Integer :: tag = 0
Integer :: ierror

Message = 1245.76

If (rank == 2) then
  Call MPI_SEND(message, 1, MPI_DOUBLE_PRECISION, 3, tag, &
                MPI_COMM_WORLD, ierror)
End if
```

.cpp

```
double message ;
int tag = 0 ;
ierror ;

message = 1245.76 ;

if (rank == 2) {
  ierror = MPI_Send(&message, 1, MPI_DOUBLE, 3, tag, &
                MPI_COMM_WORLD)
}
```

# Exemple d'utilisation de MPI_SEND

*Point to point communication : MPI_SEND*

- Envoi d'un message de type tableau contenant 5 entiers au processus de rang 6 par le processus de rang 1

.f90

```
Integer, dimension(5) :: message
Integer :: tag
Integer :: ierror

Message = (/ 12,45,37,43,59 /)

If (rank == 1) then
  Call MPI_SEND(message, 5, MPI_INTEGER, 6, tag, &
                MPI_COMM_WORLD, ierror)
End if
```

.cpp

```
int message[5] ;
int tag, ierror ;

message = { 12,45,37,43,59 } ;

if (rank == 1) {
  Ierror = MPI_Send(message, 5, MPI_INT, 6, tag, &
                MPI_COMM_WORLD) ;
}
```

# Exemple d'utilisation de MPI_SEND

*Point to point communication : MPI_SEND*

- Envoi d'un message au processus de rang 6 par le processus de rang 1 de type chaîne de caractère contenant 4 caractères et commençant au deuxième élément du message

.f90

```
character(len=10) :: message
Integer :: tag
Integer :: ierror

Message = « abcdefghij »

If (rank == 1) then
  ! Seulement « bcde » est envoyé
  Call MPI_SEND(message(2), 4, MPI_CHARACTER, 6, tag, &
                MPI_COMM_WORLD, ierror)
End if
```

.cpp

```
char message[10] = « abcdefghij » ;
int tag, ierror ;

if (rank == 1) {
  ! Seulement « bcde » est envoyé
  Ierror = MPI_Send(&message[1], 4, MPI_CHAR, 6, tag, MPI_COMM_WORLD) ;
}
```

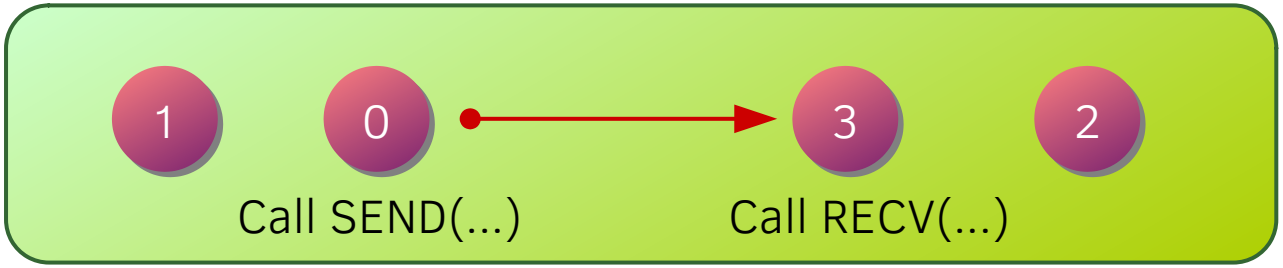
# Notion de communication point à point : Réception de données via MPI_RECV

*Point to point communication : MPI_RECV*

- MPI_RECV est la fonction appelée par le processus destinataire

.f90

```
MPI_RECV(message, size, data_type, source_rank, tag,
communicator, status, ierror)
```




https://www.open-mpi.org/doc/v1.8/man3/MPI_Recv.3.php

# Notion de communication point à point : Réception de données via MPI_RECV (Fortran)

*Point to point communication : MPI_RECV*

- `MPI_RECV` est la fonction appelée par le processus destinataire


```
MPI_RECV(message, size, data_type, source_rank, tag,
communicator, status, ierror)
```

- `Message` : la variable contenant le message à recevoir (booléen, entier, double, caractère, chaîne, tableau, structure plus complexe…)
- `Size` : nombre d'éléments constituant le message (> 1 uniquement pour une chaîne ou un tableau)
- `data_type` : type de variable utilisée pour le message (`MPI_INTEGER` pour les integer, `MPI_DOUBLE_PRECISION` pour les `real(8)`, `MPI_REAL` pour les real(4)…)
- `source_rank` : rang du processus expéditeur
- `Status :` état de la communication (en dehors de la portée de ce cours)

# Notion de communication point à point : Réception de données via MPI_Recv (C/C++)

*Point to point communication : MPI_Recv*

- RECV est la fonction appelée par le processus destinataire

.cpp

```
MPI_Recv(message, size, data_type, source_rank, tag,
communicator, status, ierror)
```

- `Message (void *)` : la variable contenant le message à recevoir (booléen, entier, double, caractère, chaîne, tableau, structure plus complexe…)
- `Size (int)` : nombre d'éléments constituant le message (> 1 uniquement pour une chaîne ou un tableau)
- `data_type (MPI_Datatype)` : type de variable utilisée pour le message (`MPI_INT` pour les int, `MPI_DOUBLE` pour les `double`, `MPI_FLOAT` pour les `float`…)
- `source_rank (int)` : rang du processus expéditeur
- `Status (MPI_Status *) :` état de la communication (en dehors de la portée de ce cours)

https://www.open-mpi.org/doc/v1.8/man3/MPI_Recv.3.php

# Exemple d'utilisation de MPI_SEND and MPI_RECV (Fortran95)

*Point to point communication : MPI_SEND and MPI_RECV*

- Envoi d'un message de type real(8) au processus de rang 3 par le processus de rang 2
- Réception d'un message de type real(8) par le rang 2 venant du rang 3

.f90

```
Real(8) :: message
Integer :: tag
Integer :: ierror

If (rank == 2) then

  Message = 1245.76

  Call MPI_SEND(message, 1, MPI_DOUBLE_PRECISION, 3, tag, &
                MPI_COMM_WORLD, ierror)

End if

If (rank == 3) then

  Call MPI_RECV(message, 1, MPI_DOUBLE_PRECISION, 2, tag, &
                MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierror)

End if
```

# Exemple d'utilisation de MPI_Send and MPI_Recv (C/C++)

*Point to point communication : MPI_Send and MPI_Recv*

- Envoi d'un message de type **double** au processus de rang 3 par le processus de rang 2
- Réception d'un message de type **double** par le rang 2 venant du rang 3

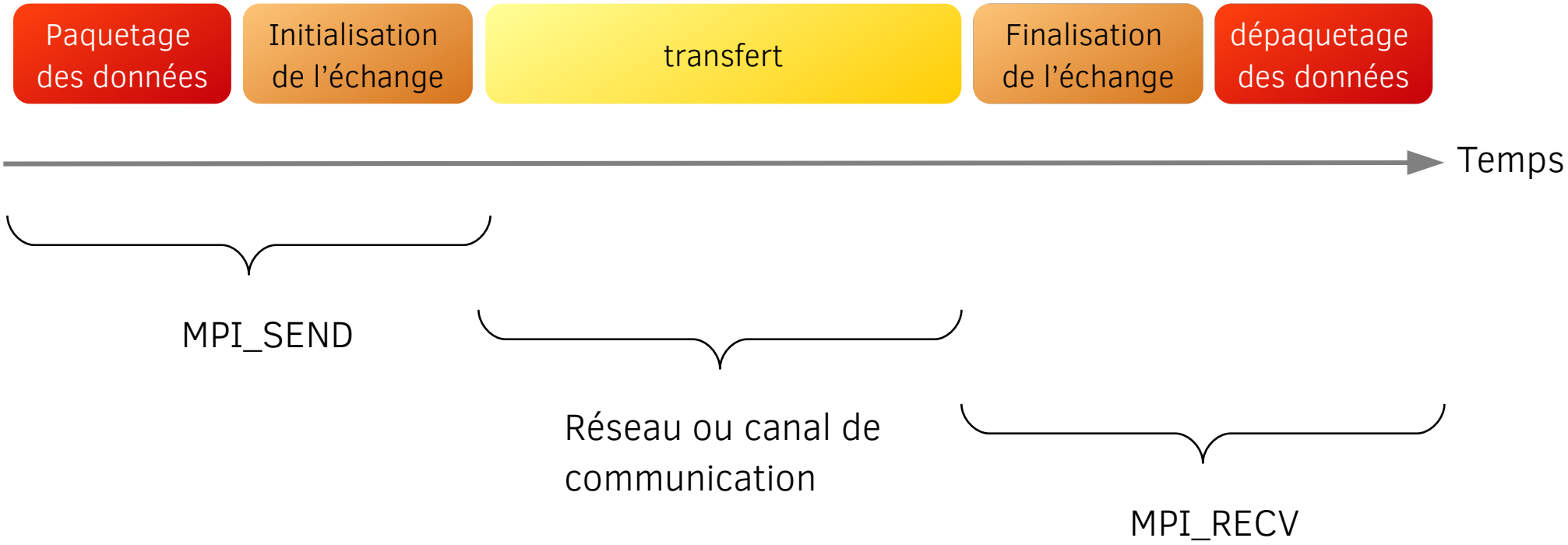
.cpp

```
double message ;
int tag, ierror ;

if (rank == 2) {
  Message = 1245.76 ;
  MPI_Send(&message, 1, MPI_DOUBLE, 3, tag, MPI_COMM_WORLD) ;
}

If (rank == 3) {
  MPI_Recv(&message, 1, MPI_DOUBLE, 2, tag,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;

}
```

# Mieux comprendre une communication

*Point to point communication : SEND and RECV*

- Une communication se compose d'un ensemble de sous-étapes :



# Exercice n°2 : Utilisation des communications point à point

- Rendez vous dans le dossier de l'exercice n°2 appelé `2_blocking_com`

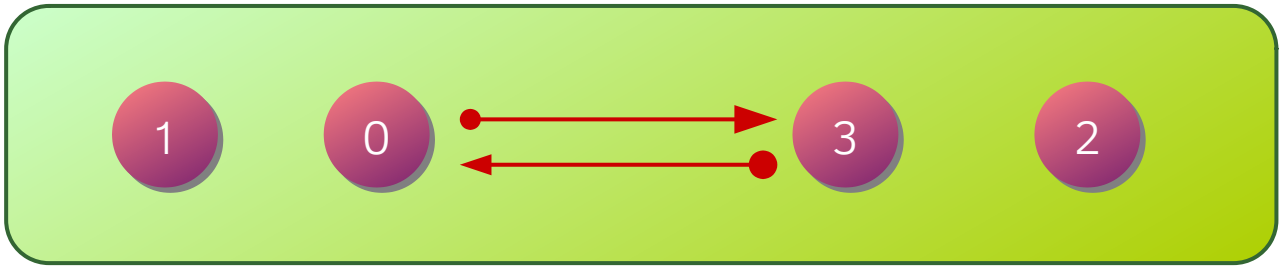
```
> cd exercises/mpi/2_blocking_com
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit…) ou visualisez directement les instructions sur le GitLab.

# Communication point à point : MPI_SENDRECV

*Point to point communication*

- Il est parfois nécessaire de faire un échange mutuel de données. Pour ce faire, il existe une fonction qui allie l'envoie et la réception : MPI_SENDRECV.




Le processus de rang 0 envoie et reçoit un message au processus de rang 3

Le processus de rang 3 envoie et reçoit un message du processus de rang 0

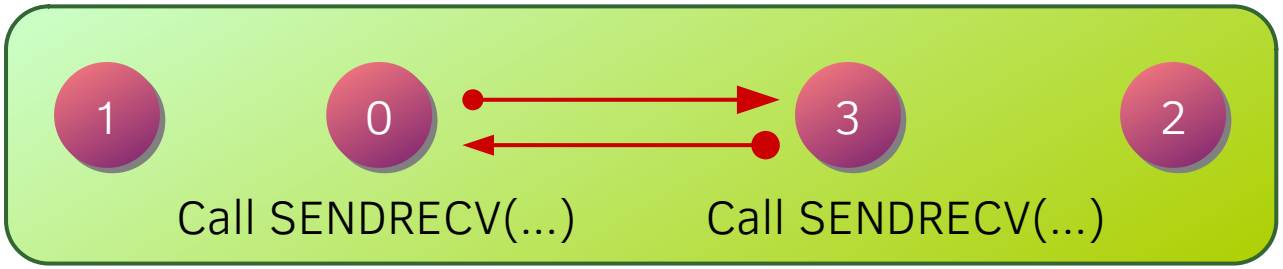
# Communication point à point : MPI_SENDRECV (Fortran95)

*Point to point communication : MPI_SENDRECV*

- SENDRECV est appelée par les processus expéditeur et destinataire en même temps

.f90

```
MPI_SENDRECV(                                                    &
send_message, send_size, send_type, destination, send_tag,       &
recv_message, recv_size, recv_type, source, recv_tag,            &
communicator, status, ierror)
```




https://www.open-mpi.org/doc/v1.8/man3/MPI_Sendrecv.3.php

# Communication point à point : MPI_Sendrecv (C/C++)

*Point to point communication : MPI_Sendrecv*

- MPI_Sendrecv est appelée par les processus expéditeur et destinataire en même temps

.cpp

```
ierror = MPI_Sendrecv(
send_message, send_size, send_type, destination, send_tag,
recv_message, recv_size, recv_type, source, recv_tag,
communicator, status) ;
```

- `send_message (const void *)` : données envoyées
- `recv_message (void *)` : données reçues
- Les autres paramètres sont les mêmes que pour `MPI_Send` et `MPI_Recv`

https://www.open-mpi.org/doc/v1.8/man3/MPI_Sendrecv.3.php

# Exemple d'utilisation de MPI_SENDRECV (Fortran 95)

*Point to point communication : MPI_SENDRECV*

- Envoi et réception d'un message de type real(8) au processus de rang 3 par le processus de rang 2
- Envoi et réception d'un message de type real(8) par le rang 2 venant du rang 3


```
Real(8) :: send_message
Real(8) :: recv_message
Integer :: send_tag
Integer :: recv_tag
Integer :: ierror

If (rank == 2) then

  send_message = 1245.76

  Call MPI_SENDRECV(send_message, 1, MPI_DOUBLE_PRECISION, 3, send_tag, &
                    recv_message, 1, MPI_DOUBLE_PRECISION, 3, recv_tag, &
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierror)

End if

If (rank == 3) then

  send_message = 4567.32

  Call MPI_SENDRECV(send_message, 1, MPI_DOUBLE_PRECISION, 2, send_tag, &
                    recv_message, 1, MPI_DOUBLE_PRECISION, 2, recv_tag, &
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierror)

End if
```

# Exemple d'utilisation de MPI_Sendrecv

*Point to point communication : MPI_Sendrecv*

- Envoi et réception d'un message de type **double** au processus de rang 3 par le processus de rang 2
- Envoi et réception d'un message de type **double** par le rang 2 venant du rang 3

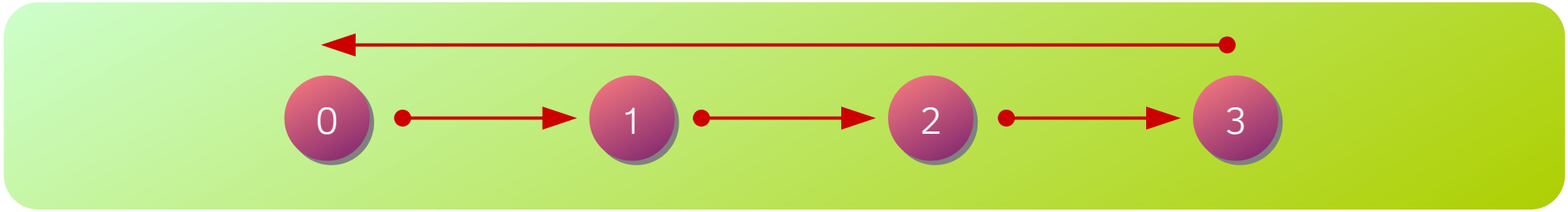
.cpp

```
double send_message, recv_message ;
int send_tag, recv_tag, ierror ;

if (rank == 2) {

  send_message = 1245.76 ;

  ierror = MPI_Sendrecv(&send_message, 1, MPI_DOUBLE, 3, send_tag,
                  &recv_message, 1, MPI_DOUBLE, 3, recv_tag,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;

}

If (rank == 3)  {

  send_message = 4567.32 ;

  ierror = MPI_Sendrecv(&send_message, 1, MPI_DOUBLE, 2, send_tag, &
                  &recv_message, 1, MPI_DOUBLE, 2, recv_tag, &
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;

}
```

# Communication point à point : MPI_SENDRECV pour les communications chaînées

*Point to point communication*

- La fonction **MPI_SENDRECV** est également nécessaire pour effectuer des communications chaînées
- L'utilisation de **MPI_SEND** et **MPI_RECV** nécessiterait de gérer manuellement les synchronisations



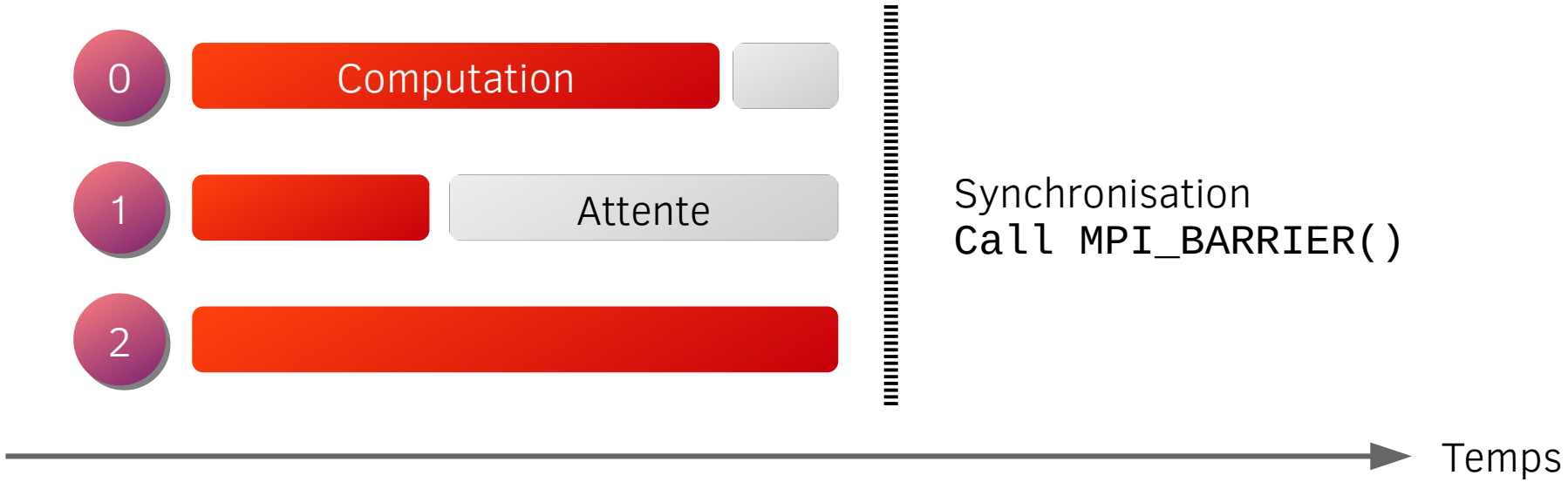

Chaque processus reçoit un élément d'un processus A et envoie des données à un processus B distinct.

Lorsqu'une communication n'arrive pas à son terme, le programme attend et peut rester figé.

# Notion de barrière explicite

*Explicit barrier*

- Il est parfois nécessaire d'imposer un une étape de synchronisation ou barrière qui ne sera pas franchie tant que tous les processus ne seront pas arrivés à ce niveau
- La fonction `MPI_BARRIER` est une manière explicite d'exiger cette synchronisation dans le code

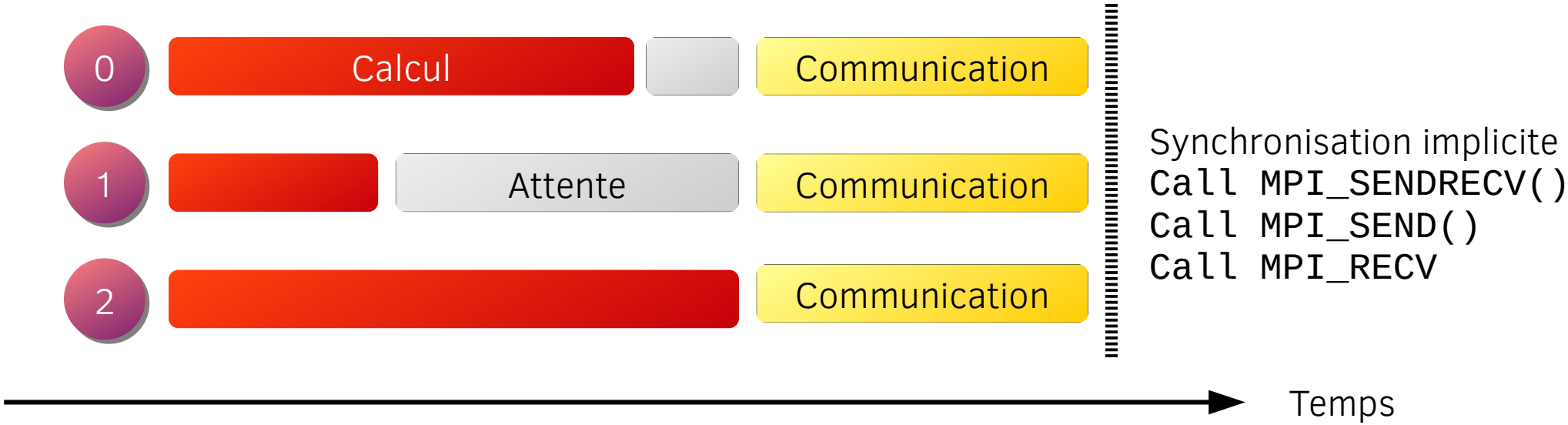



https://www.open-mpi.org/doc/v1.5/man3/MPI_Barrier.3.php

# Notion de barrière implicite

*Implicit barrier*

- Certaines fonctions d'échange induisent des barrières implicites au niveau des processus concernés
- C'est le cas de `MPI_SEND`, `MPI_RECV`, `MPI_SENDRECV` d'où l'appellation de communication bloquante



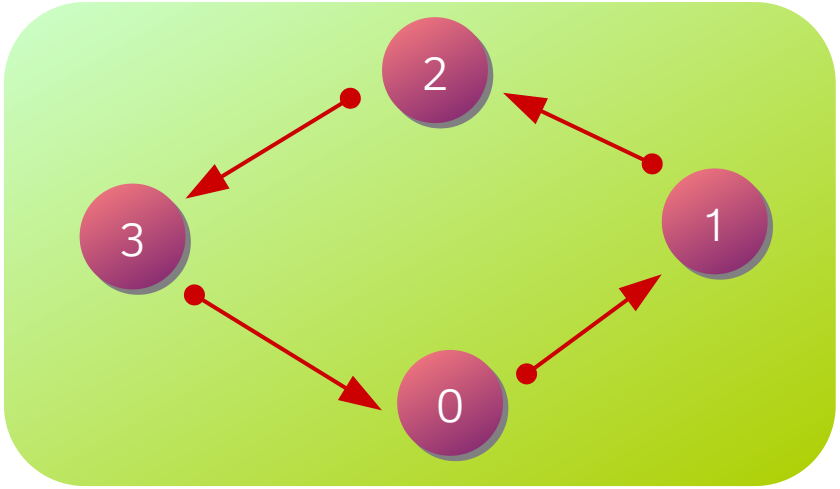

Si certains processus sont en avance, ils effectuent une attente active ou passive. Cette attente est vue comme une perte de ressource.

# Exercice n°3 : Chaîne ou anneau de communication

- Rendez vous dans le dossier de l'exercice n°3 appelé `3_sendrecv`

```
> cd exercises/mpi/3_sendrecv
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit…) ou visualisez directement les instructions sur le GitLab.




Anneau de communication

# Premiers échanges MPI

A ce stade du cours, vous savez maintenant :

- Faire communiquer différents processus entre eux
- Gérer des chaînes de communication
- Demander une synchronisation explicite

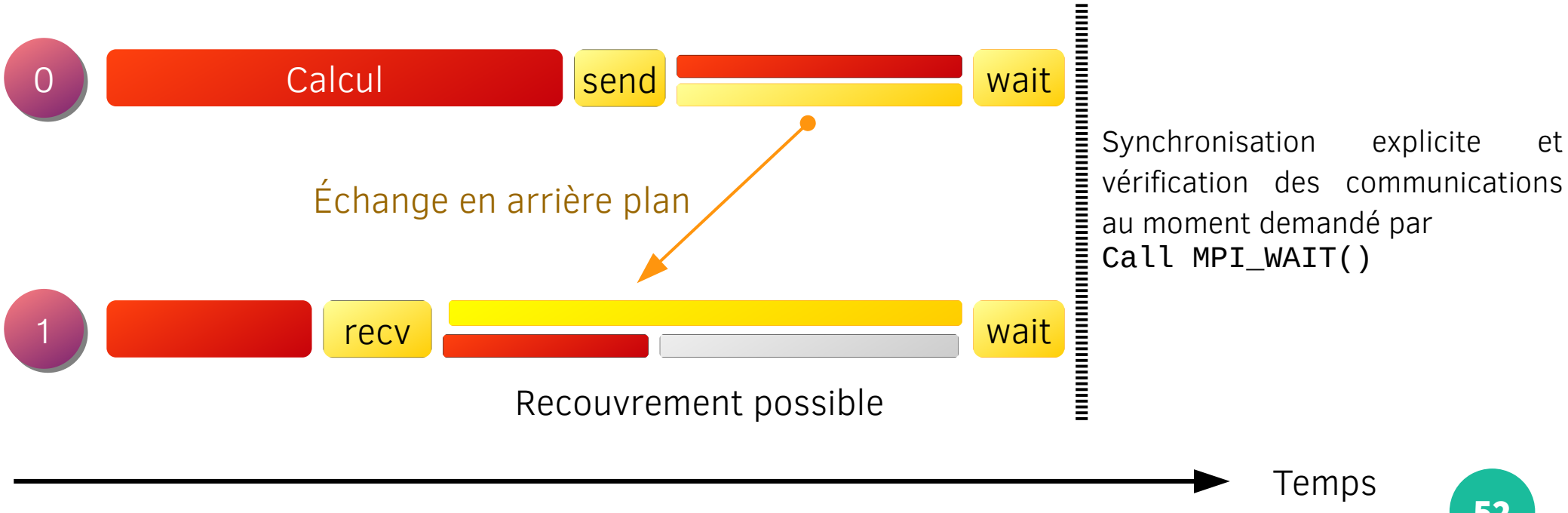
# Introduction au parallélisme par échange de message via MPI

## 3) Les communications point à point non-bloquantes

# Communication point à point non-bloquante

*Non-blocking communication*

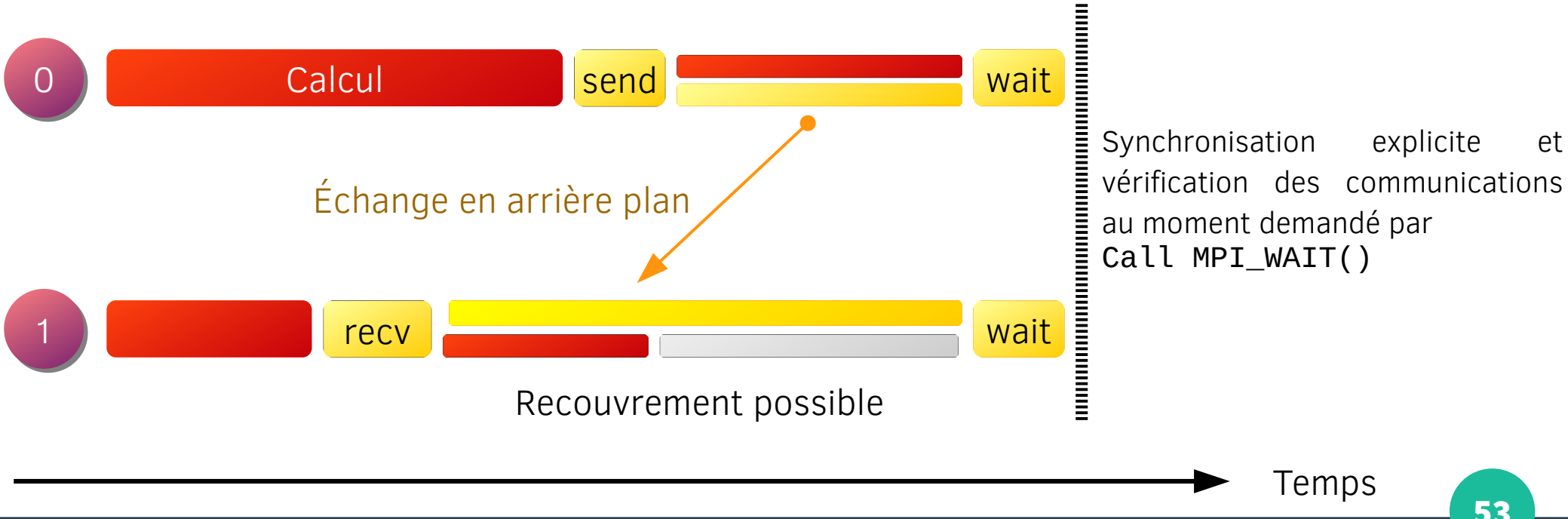
- Les communications non-bloquantes permettent d'éviter la synchronisation implicite des processus.
- Une synchronisation explicite est nécessaire pour s'assurer que les communications ont eu lieu avant d'utiliser les données échangées
- Ce type de communication permet de recouvrir communication et calcul : envoi et réception en arrière plan



# Communication point à point non-bloquante

*Non-blocking communication*

- Les communications se font via les fonctions **`MPI_ISEND`** et **`MPI_IRECV`**.
- Les communications se voient attribuées un identifiant unique
- A un moment donné, il est nécessaire de vérifier que ces communications ont eu lieu, c'est le rôle de **`MPI_WAIT`**. Cette fonction analyse les identifiants fournis.
- **`MPI_WAIT`** impose une barrière



# Les fonctions `MPI_Isend` et `MPI_Irecv` (Fortran95)

*MPI_Isend and MPI_Irecv*

- MPI_ISEND est la fonction appelée par le processus expéditeur


```
MPI_ISEND(message, size, data_type, destination_rank, tag,
communicator, request, ierror)
```

- MPI_IRECV est la fonction appelée par le processus receveur


```
MPI_IRECV(message, size, data_type, source_rank, tag,
communicator, request, ierror)
```

- Les paramètres sont les mêmes pour que pour les communications bloquantes.
- S'ajoute la variable **`request (int)`** utilisée par **`MPI_WAIT`** pour vérifier l' état de la communication

https://www.open-mpi.org/doc/v4.0/man3/MPI_Isend.3.php
https://www.open-mpi.org/doc/v4.0/man3/MPI_Irecv.3.php

# Les fonctions `MPI_Isend` et `MPI_Irecv` (C/C++)

*MPI_Isend and MPI_Irecv*

- MPI_Isend est la fonction appelée par le processus expéditeur


```
MPI_Isend(&message, size, data_type, destination_rank, tag,
communicator, request) ;
```

- MPI_Irecv est la fonction appelée par le processus receveur


```
MPI_Irecv(&message, size, data_type, source_rank, tag,
communicator, request) ;
```

- Les paramètres sont les mêmes pour que pour les communications bloquantes.
- S'ajoute la variable **`request`** (de type **`MPI_Request *`**) utilisé par **`MPI_WAIT`** pour vérifier l' état de la communication

https://www.open-mpi.org/doc/v4.0/man3/MPI_Isend.3.php
https://www.open-mpi.org/doc/v4.0/man3/MPI_Irecv.3.php

# La fonction `MPI_Wait`

*MPI_Wait*

- `MPI_Wait` est la fonction appelée pour vérifier et attendre que la communication a bien été effectuée

.f90

```
MPI_WAIT(request, status, ierror)
```

.cpp

```
MPI_Wait(request, status) ;
```

https://www.open-mpi.org/doc/v4.0/man3/MPI_Wait.3.php

# Exemple d'utilisation des communications non-bloquantes (Fortran 95)

*Non-blocking communication*

- Envoi d'un message de type real(8) au processus de rang 3 par le processus de rang 2
- Réception d'un message de type real(8) par le rang 2 venant du rang 3

.f90

```
Real(8) :: send_message, recv_message
Integer :: send_tag, recv_tag, ierror
Integer :: request

If (rank == 2) then

  send_message = 1245.76

  Call MPI_ISEND(send_message, 1, MPI_DOUBLE_PRECISION, 3, send_tag, &
                 MPI_COMM_WORLD, request, ierror)

End if

If (rank == 3) then

  Call MPI_IRECV(recv_message, 1, MPI_DOUBLE_PRECISION, 2, recv_tag, &
                 MPI_COMM_WORLD, request, ierror)

End if

… Calcul ...

Call MPI_WAIT(request,MPI_COMM_WORLD,ierror) ;

… calcul suivant ...
```

# Exemple d'utilisation des communications non-bloquantes (C/C++)

*Non-blocking communication*

- Envoi d'un message de type double au processus de rang 3 par le processus de rang 2
- Réception d'un message de type double par le rang 2 venant du rang 3

.cpp

```
double send_message, recv_message ;
int send_tag, recv_tag, ierror ;
MPI_Request request ;

If (rank == 2) {

  send_message = 1245.76

  Call MPI_Isend(send_message, 1, MPI_DOUBLE, 3, send_tag, &
                 MPI_COMM_WORLD, request) ;

}

If (rank == 3) {

  Call MPI_Irecv(&recv_message, 1, MPI_DOUBLE, 2, recv_tag, &
                 MPI_COMM_WORLD, request) ;

}

MPI_Wait(request,MPI_COMM_WORLD) ;
```

# La fonction MPI_WAITALL

*MPI_Wait*

- MPI_WAITALL effectue l'action de MPI_WAIT sur un tableau de requêtes.

.f90

```
MPI_WAITALL(number_of_request, request, status, ierror)
```

.cpp

```
MPI_Waitall(number_of_request, request, status) ;
```

- request et status sont alors des tableaux (d'entiers en Fortran95, MPI_Request et MPI_Status * en C)

https://www.open-mpi.org/doc/v4.0/man3/MPI_Waitall.3.php

# Mélange des types de communication

Il est tout à fait possible de mélanger des appels bloquants à des appels non-bloquants.

Lorsque la requête est terminée, elle devient `MPI_REQUEST_NULL`. Il est également possible d'initialiser certaines requêtes ainsi pour les ignorer lors du `MPI_Wait` et `MPI_Waitall`.

# Exercice n°4 : Utilisation des communications non bloquantes

- Rendez vous dans le dossier de l'exercice n°4 appelé `4_nonblocking_com`

```
> cd exercises/mpi/4_nonblocking_com
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit…) ou visualisez directement les instructions sur le GitLab.

# Introduction au parallélisme par échange de message via MPI

## 4) Les communications collectives

### 4.1) Communications collectives de base

# Communication collective : les différents types

*Collective communication*

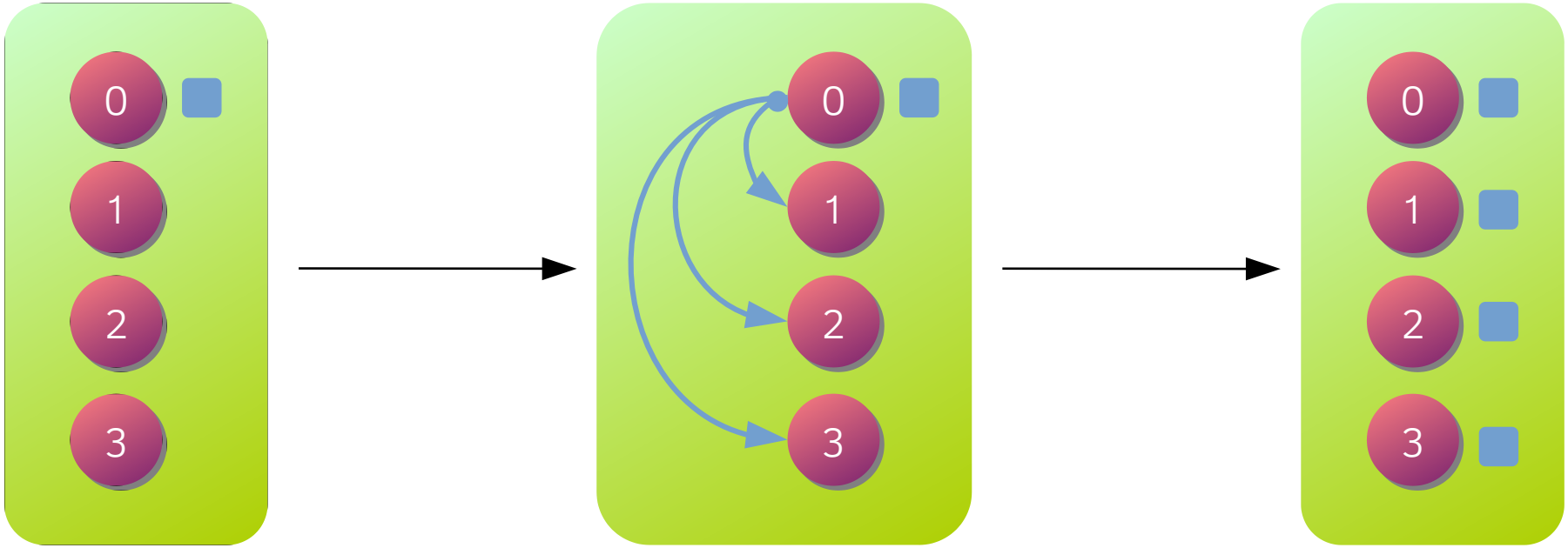
- Les communications collectives sont des communications qui font intervenir plusieurs processus (voir tout le communicateur) dans le but de propager ou de rassembler de l'information.
- 3 types de communication collective :
  - Synchronisation : c'est le `MPI_BARRIER`
  - Transfert de données (diffusion, collecte)
  - Transfert et opérations sur les données (opération de réduction)

Les versions standards induisent des barrières implicites pour les processus concernés. Pour chaque processus, la barrière est relâchée dès la participation terminée.

# Communication collective : diffusion générale grâce à `MPI_Bcast`

*Collective communication*

- Envoi d'une donnée depuis un processus vers tous les processus du communicateur




Répartition des données avant l'échange

`MPI_Bcast`

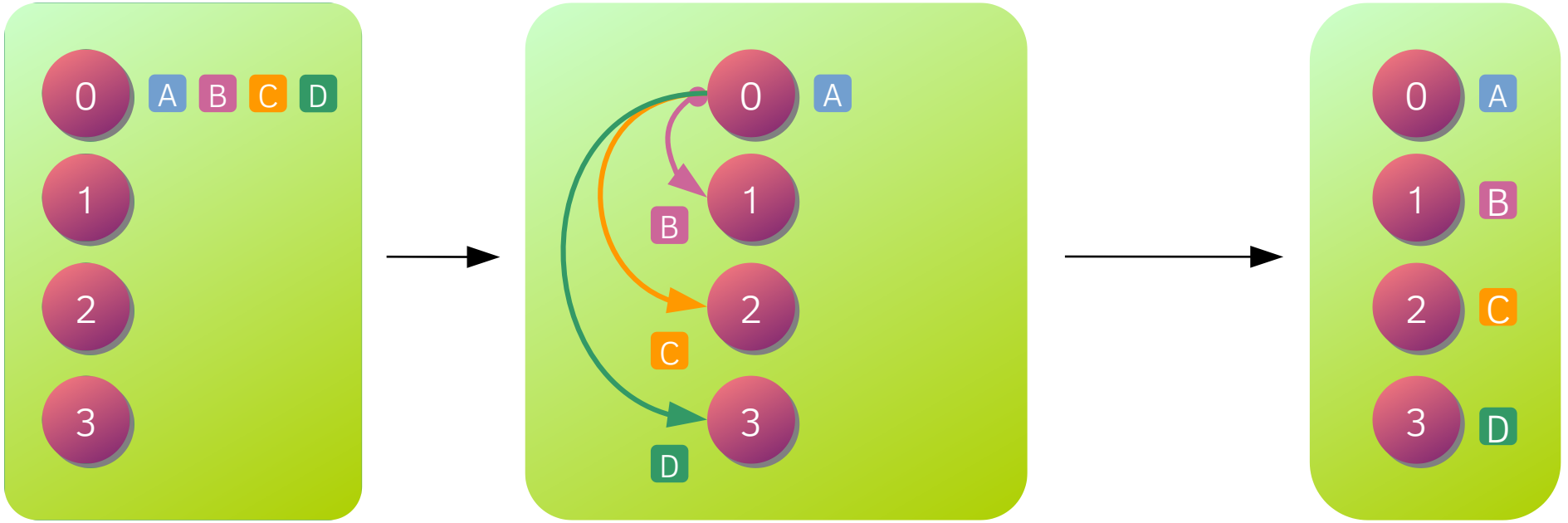
Nouvelle répartition des données après l'échange

https://www.open-mpi.org/doc/v4.0/man3/MPI_Bcast.3.php

# Communication collective : diffusion sélective grâce à `MPI_Scatter`

*Collective communication*

- Partage de données sélectif (selon les critères du développeur) depuis un processus vers tous les processus du communicateur




Répartition des données avant l'échange

`MPI_SCATTER`

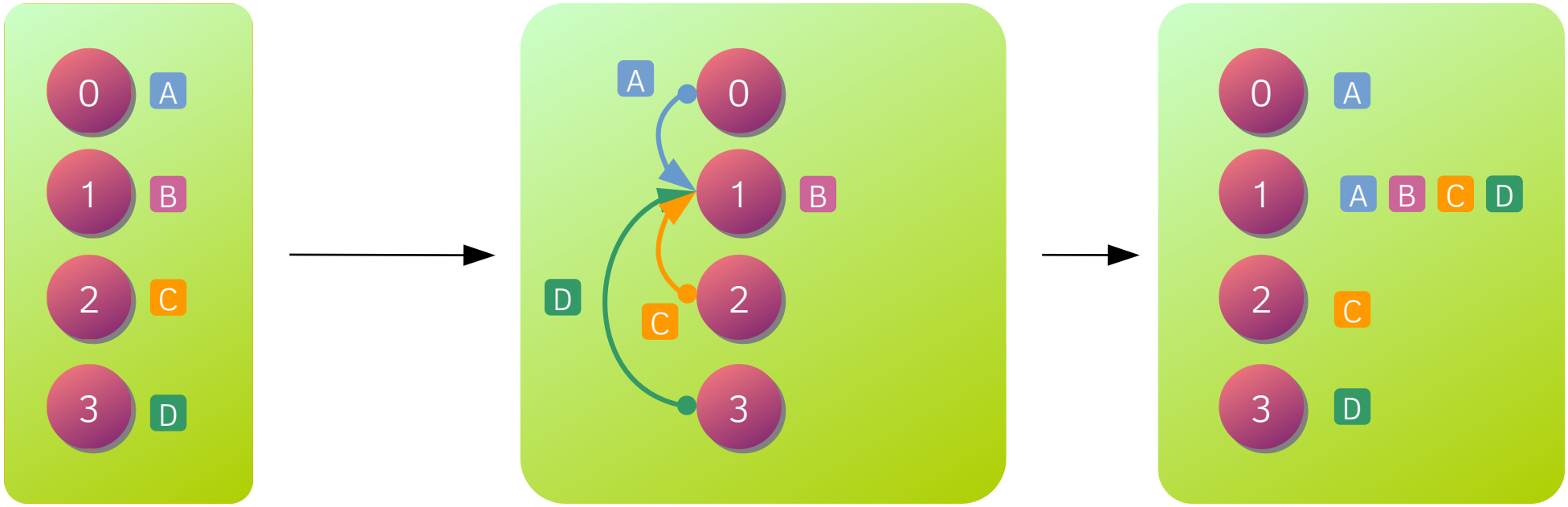
Nouvelle répartition des données

https://www.open-mpi.org/doc/v4.0/man3/MPI_Scatter.3.php

# Communication collective : collecte grâce à `MPI_Gather`

*Collective communication*

- Envoi de données réparties sur plusieurs processus vers un processus unique




Répartition des données avant l'échange

`MPI_GATHER`

Nouvelle répartition des données

https://www.open-mpi.org/doc/v4.0/man3/MPI_Gather.3.php

# Communication collective : collecte grâce à `MPI_Gather` (Fortran95)

*Collective communication*

- `MPI_Gather` est appelée par les processus expéditeurs et destinataires en même temps

.f90

```
MPI_GATHER(send_buf, send_count, send_type, &
           recv_buf, recv_count, recv_type, &
           destination, communicator, ierror)
```

- **`send_buf`** : la valeur ou un ensemble de valeur
- **`send_count`** : le nombre de valeur à envoyer
- **`send_type`** : type MPI des valeurs envoyées
- **`recv_buf`** : le tableau réunissant les valeurs reçues
- **`recv_count`** : nombre d'éléments reçus
- **`recv_type`** : le type des données reçues
- **`destination`** : le processus qui reçoit les données

https://www.open-mpi.org/doc/v4.0/man3/MPI_Gather.3.php

# Communication collective : collecte grâce à `MPI_Gather` (C/C++)

*Collective communication*

- `MPI_Gather` est appelée par les processus expéditeurs et destinataires en même temps

.cpp

```
MPI_Gather(send_buf, send_count, send_type,
        recv_buf, recv_count, recv_type,
        destination, communicator) ;
```

- `send_buf (const void *)` : la valeur ou un ensemble de valeur
- `send_count (int)` : le nombre de valeur à envoyer
- `send_type (MPI_Datatype)` : type MPI des valeurs envoyées
- `recv_buf (void *)` : le tableau réunissant les valeurs reçues
- `recv_count (int)` : nombre d'éléments reçus
- `recv_type (MPI_Datatype)` : le type des données reçues
- `Destination (int)` : le processus qui reçoit les données

https://www.open-mpi.org/doc/v4.0/man3/MPI_Gather.3.php

# Introduction au parallélisme par échange de message via MPI

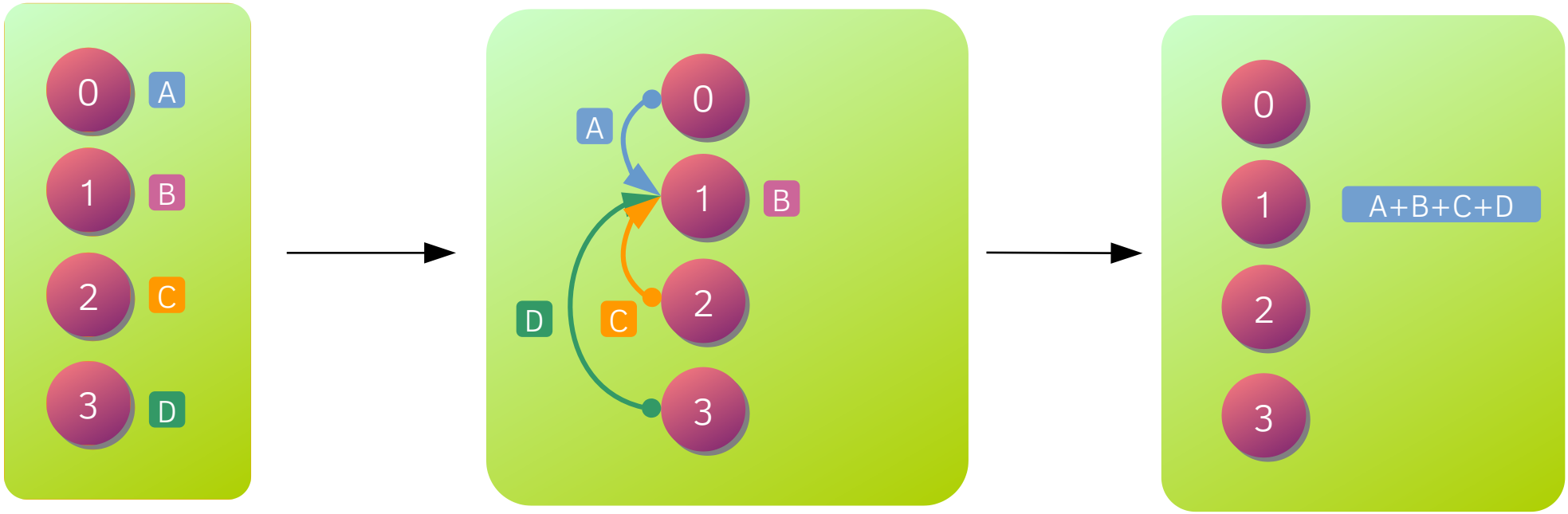
## 4) Les communications collectives

### 4.2) Les réductions

# Communication collective : réduction grâce à `MPI_Reduce`

*Collective communication*

- Envoi de données réparties sur plusieurs processus vers un seul processus avec une opération de réduction réalisée simultanément




Répartition des données avant l'échange

`MPI_REDUCE` + `MPI_SUM`

Nouvelle répartition des données et addition

https://www.open-mpi.org/doc/v4.0/man3/MPI_Reduce.3.php

# Communication collective : réduction grâce à `MPI_Reduce` (Fortran95)

*Collective communication*

- `MPI_Reduce` est appelée par les processus expéditeurs et destinataires en même temps

.f90

```
MPI_REDUCE(send_value, recv_value, size, MPI_data_type,
MPI_reduction_operation, destination, communicator, ierror)
```

- `send_value` : la valeur à envoyer par chaque processus
- `recv_value` : la valeur reçue suite aux échanges et à la réduction par le destinataire seulement
- `Size` : nombre d'éléments (>1 si tableau)
- `MPI_data_type` : le type de donnée (ex : `MPI_INTEGER`)
- `MPI_reduction_operation` : type d'opération à effectuer pour la réduction (ex : `MPI_SUM`)
- `Destination` : le processus qui va recevoir les données réduites

https://www.open-mpi.org/doc/v4.0/man3/MPI_Reduce.3.php

# Communication collective : réduction grâce à `MPI_Reduce` (C/C++)

*Collective communication*

- `MPI_Reduce` est appelée par les processus expéditeurs et destinataires en même temps

.cpp

```
MPI_Reduce(send_value, &recv_value, size, MPI_data_type,
MPI_reduction_operation, destination, communicator) ;
```

- **`send_value (const void *)`** : la valeur à envoyer par chaque processus
- **`recv_value (void *)`** : la valeur reçue suite aux échanges et à la réduction par le destinataire seulement
- **`Size`** : nombre d'éléments (>1 si tableau)
- **`MPI_data_type (MPI_Datatype)`** : le type de donnée (ex : **`MPI_INT`**)
- **`MPI_reduction_operation (MPI_Op)`** : type d'opération à effectuer pour la réduction (ex : **`MPI_SUM`**)
- **`Destination`** : le processus qui va recevoir les données réduites

https://www.open-mpi.org/doc/v4.0/man3/MPI_Reduce.3.php

# Communication collective : opération de réduction

*Collective communication*

Il existe de multiple opérations de réduction disponibles (`MPI_reduction_operation`) :

- `MPI_SUM` : Somme l'ensemble des données
- `MPI_PROD` : multiplication des données
- `MPI_MAX` : maximum des valeurs
- `MPI_MIN` : minimum des valeurs

...

https://www.open-mpi.org/doc/v4.0/man3/MPI_Reduce.3.php

# Communication collective : exemple d'utilisation de MPI_REDUCE

*Collective communication*

- Réduction d'une simple variable réelle double précision


```
Real(8) :: rank_value
Real(8) :: reduction_value
Integer :: ierror

rank_value = rank

! Addition de l'ensemble des rank_value dans le processus 0

Call MPI_REDUCE(rank_value, reduction_value, 1    &
                MPI_DOUBLE_PRECISION, MPI_SUM, 0, &
                MPI_COMM_WORLD, ierror)
```

# Communication collective : exemple d'utilisation de `MPI_Reduce` (C/C++)

*Collective communication*

- Réduction d'une simple variable réelle double précision

.cpp

```
double rank_value, reduction_value ;
int ierror ;

rank_value = rank ;

// Addition de l'ensemble des rank_value dans le processus 0
ierror = MPI_Reduce(rank_value, &reduction_value, 1
                    MPI_DOUBLE, MPI_SUM, 0,
                    MPI_COMM_WORLD) ;
```

# Communication collective : exemple d'utilisation de MPI_REDUCE

*Collective communication*

- Réduction d'un tableau d'entiers avec multiplication de toutes les valeurs

.f90

```
integer, dimension(4) :: rank_value
integer, dimension(4) :: reduction_value
Integer :: ierror

rank_value = (/ 18, 22, 43, 52/)

! Addition de l'ensemble des rank_value dans le processus 0

Call MPI_REDUCE(rank_value, reduction_value, 4    &
                MPI_INTEGER, MPI_PROD, 0,         &
                MPI_COMM_WORLD, ierror)
```

# Communication collective : exemple d'utilisation de `MPI_Reduce` (C/C++)

*Collective communication*

- Réduction d'un tableau d'entiers avec multiplication de toutes les valeurs

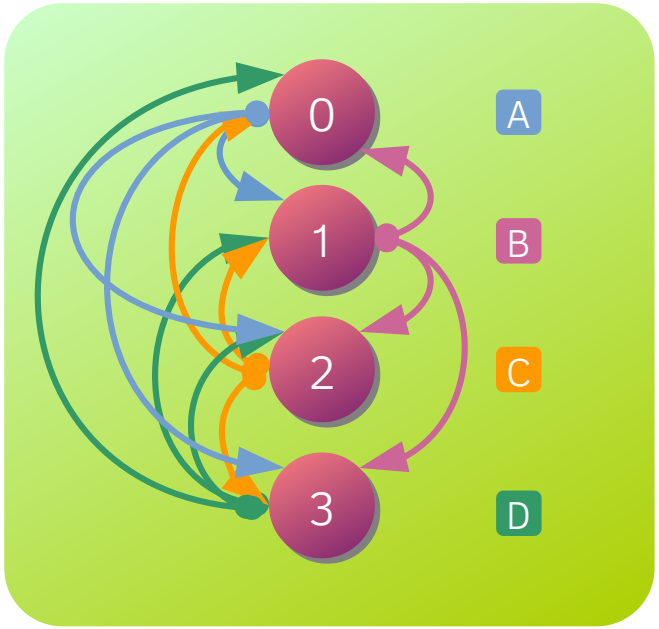
.cpp

```
int rank_value[4] ;
int reduction_value[4] ;
int ierror ;

rank_value = { 18, 22, 43, 52 } ;

// Addition de l'ensemble des rank_value dans le processus 0

ierror = MPI_Reduce(rank_value, &reduction_value[0], 4
                    MPI_INT, MPI_PROD, 0,
                    MPI_COMM_WORLD) ;
```

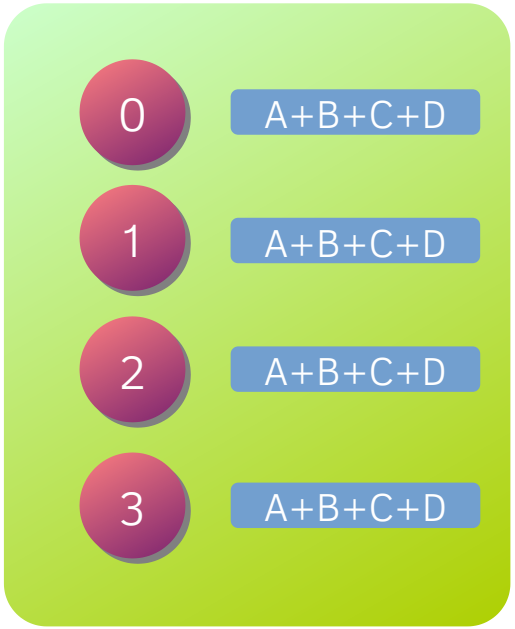
# Communication collective : réduction grâce à `MPI_Allreduce`

*Collective communication*

- Envoi de données réparties sur plusieurs processus vers tous les processus avec une opération de réduction réalisée simultanément




`MPI_ALLREDUCE` + `MPI_SUM`




Nouvelle répartition des données et addition

https://www.open-mpi.org/doc/v4.0/man3/MPI_Allreduce.3.php

# Communication collective : réduction grâce à MPI_ALLREDUCE (Fortran95)

*Collective communication*

- MPI_ALLREDUCE est appelée par les processus expéditeurs et destinataires en même temps

.f90

```
MPI_ALLREDUCE(send_value, recv_value, size, MPI_data_type,
MPI_reduction_operation, communicator, ierror)
```

- send_value : la valeur à envoyer par chaque processus
- recv_value : la valeur reçue suite aux échanges et à la réduction par le destinataire seulement
- Size : nombre d'éléments (>1 si tableau)
- MPI_data_type : le type de donnée (ex : MPI_INTEGER)
- MPI_reduction_operation : type d'opération à effectuer pour la réduction (ex : MPI_SUM)

https://www.open-mpi.org/doc/v4.0/man3/MPI_Allreduce.3.php

# Communication collective : réduction grâce à `MPI_Allreduce` (C/C++)

*Collective communication*

- `MPI_Allreduce` est appelée par les processus expéditeurs et destinataires en même temps


.f90

```
MPI_Allreduce(send_value, &recv_value, size, MPI_data_type,
MPI_reduction_operation, communicator) ;
```

- `send_value` : la valeur à envoyer par chaque processus
- `recv_value` : la valeur reçue suite aux échanges et à la réduction par le destinataire seulement
- `Size` : nombre d'éléments (>1 si tableau)
- `MPI_data_type` : le type de donnée (ex : `MPI_INTEGER`)
- `MPI_reduction_operation` : type d'opération à effectuer pour la réduction (ex : `MPI_SUM`)


https://www.open-mpi.org/doc/v4.0/man3/MPI_Allreduce.3.php

# Exercice n°5 : Utilisation de la communication collective `MPI_REDUCE`

- Rendez vous dans le dossier de l'exercice n°5 appelé `5_reduce_com`

```
> cd exercises/mpi/5_reduce_com
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit…) ou visualisez directement les instructions sur le GitLab.

# Introduction au parallélisme par échange de message via MPI

## 4) Les communications collectives

### 4.2) Les collectives avec des données de taille variable

# Communication collective pour les données de taille variable

Certaines communications collectives (hors réduction) présentées précédemment imposent que chaque rang échange la même quantité de donnée. Cette limitation peut être levée en utilisant une extension des communications collectives.

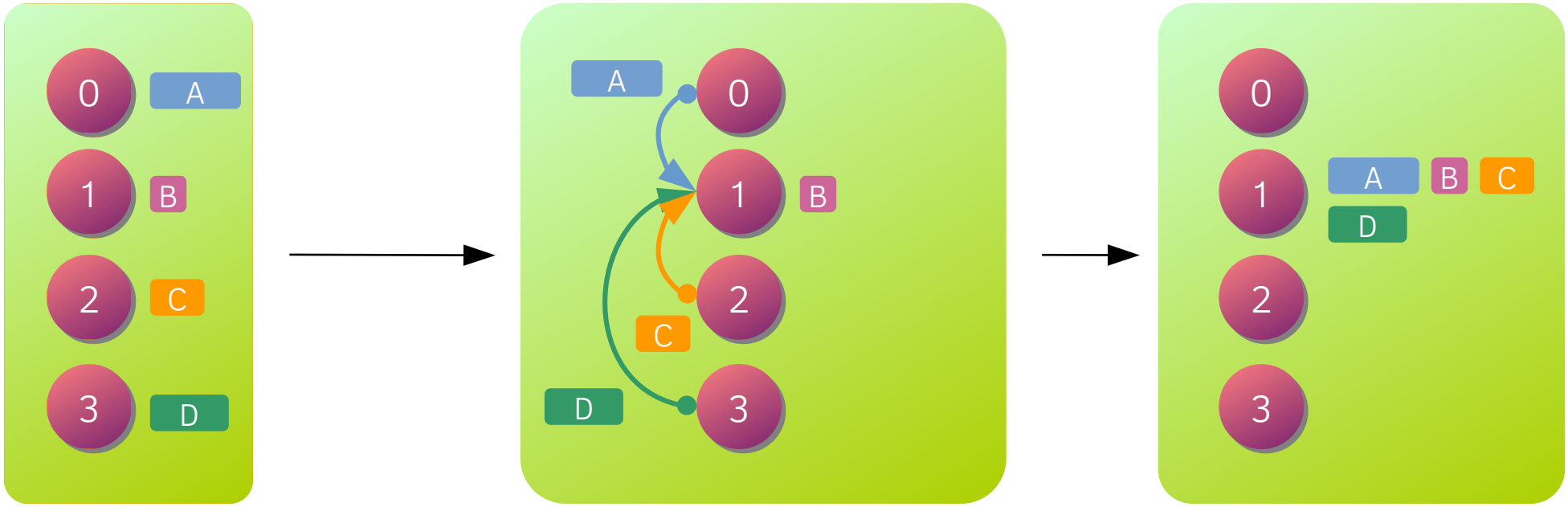
Elles possède le même nom suivi d'un v à la fin :

- MPI_Gatherv
- MPI_Allgatherv
- MPI_Alltoallv
- MPI_Scatterv

# Communication collective : collecte grâce à `MPI_Gatherv`

*Collective communication*

- Envoi de données de taille différente réparties sur plusieurs processus vers un processus unique




Répartition des données avant l'échange

`MPI_Gatherv`

Nouvelle répartition des données

https://www.open-mpi.org/doc/v4.0/man3/MPI_Gatherv.3.php

# Communication collective pour les données de taille variable

`MPI_Gatherv` peut également être utilisée pour changer l'ordre des données une fois ramenées sur le rang cible contrairement à `MPI_Gather` qui utilise l'ordre des rangs.

# Communication collective : collecte grâce à `MPI_Gatherv` (Fortran95)

*Collective communication*

- `MPI_Gatherv` est appelée par les processus expéditeurs et destinataires en même temps

.f90

```
MPI_GATHERV(send_array, send_count, send_type, &
            recv_array, recv_count, displacement, recv_type, &
            destination, communicator, ierror)
```

- **`send_array`** : la valeur ou un ensemble de valeur
- **`send_count`** : le nombre de valeur à envoyer, ce nombre peut être différent sur chaque processus
- **`send_type`** : type MPI des valeurs envoyées
- **`recv_array`** : le tableau réunissant les valeurs reçues
- **`recv_count`** (tableau) : nombre d'éléments reçus de chaque rang
- **`displacement`** (tableau) : où placer chaque contribution dans **`recv_array`**
- **`recv_type`** : le type des données reçues
- **`destination`** : le processus qui reçoit les données

https://www.open-mpi.org/doc/v4.0/man3/MPI_Gatherv.3.php

# Communication collective : collecte grâce à `MPI_Gatherv` (C/C++)

*Collective communication*

- `MPI_Gatherv` est appelée par les processus expéditeurs et destinataires en même temps

.cpp

```
MPI_Gatherv(send_array, send_count, send_type,
        recv_array, recv_count, displacement, recv_type,
        destination, communicator) ;
```

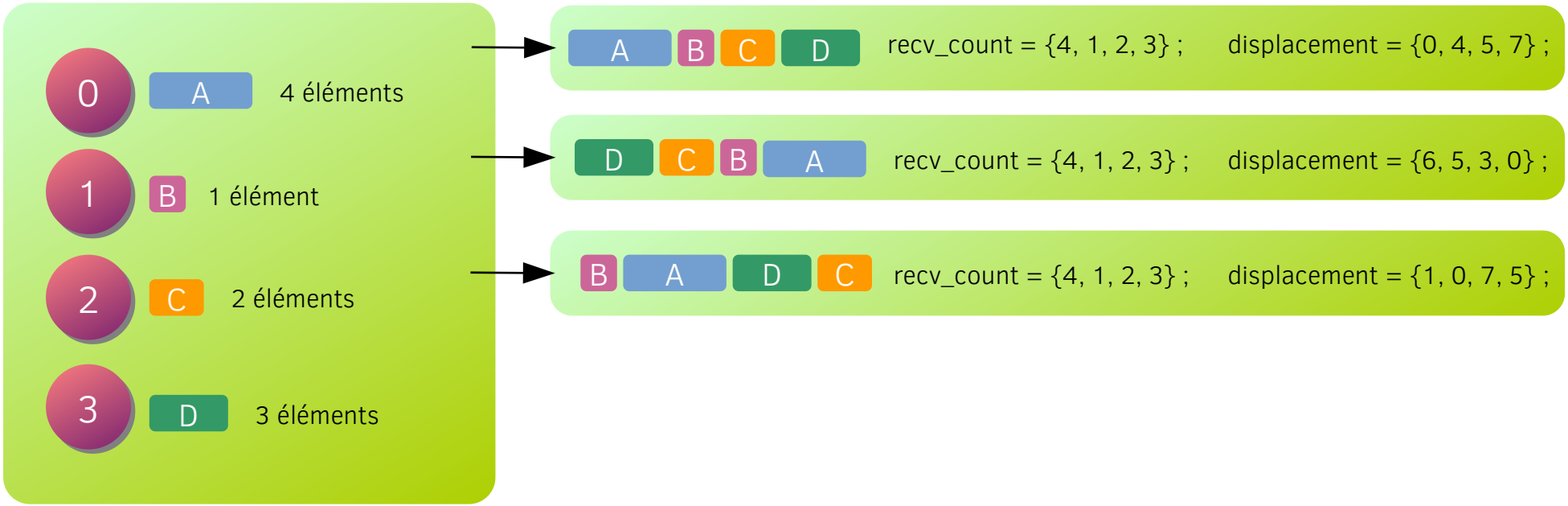
- `send_array (const void *)` : la valeur ou un ensemble de valeur
- `send_count (int)` : le nombre de valeur à envoyer
- `send_type (MPI_Datatype)` : type MPI des valeurs envoyées
- `recv_array (void *)` : le tableau réunissant les valeurs reçues
- `recv_count (const int *)` : nombre d'éléments reçus pour chaque rang
- `displacement (const int *)` : où placer chaque contribution dans `recv_array`
- `recv_type (MPI_Datatype)` : le type des données reçues
- `Destination (int)` : le processus qui reçoit les données

https://www.open-mpi.org/doc/v4.0/man3/MPI_Gatherv.3.php

# Communication collective : notion de déplacement

*Collective communication*

- Le tabeau **`displacement`** permet de placer les contribution de chaque rang dans le tableau qui reçoit les données. Il s'agit de l'emplacement de la première donnée venant de chaque rang. Ce tableau doit avoir pour taille le nombre de rang dans le communicateur.




https://www.open-mpi.org/doc/v4.0/man3/MPI_Gatherv.3.php

# Fonctions supplémentaires

Toutes les communications collectives présentées ont également un équivalent non-bloquant :

- `MPI_Igather`
- `MPI_Igatherv`
- `MPI_Iscatter`
- `MPI_Ibcast`
- `MPI_Ialltoall`
- `MPI_Ireduce`
- `...`

D'autres variantes de communications collectives sont à découvrir dans le cours de l'IDRIS et la documentation MPI

# Exercice n°6 : Utilisation de la communication collective `MPI_GATHER`

- Rendez vous dans le dossier de l'exercice n°6 appelé `6_gather_com`

```
> cd exercises/mpi/6_gather_com
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit…) ou visualisez directement les instructions sur le GitLab.

# Communications collectives MPI

A ce stade du cours, vous savez maintenant :

- Effectuer des communications collectives
- Effectuer des réductions

# Introduction au parallélisme par échange de message via MPI

## 5) Topologie cartésienne

# Décomposition de domaine cartésienne

En calcul scientifique, il est courant de décomposer le domaine d'étude (grille, matrice) en sous-domaines, chaque sous-domaine étant alors géré par un processus MPI unique.

Sur grille régulière et structurée, une approche simple et classique consiste à diviser le domaine en blocs réguliers possédant alors dans sa mémoire locale une partie unique de la grille globale.

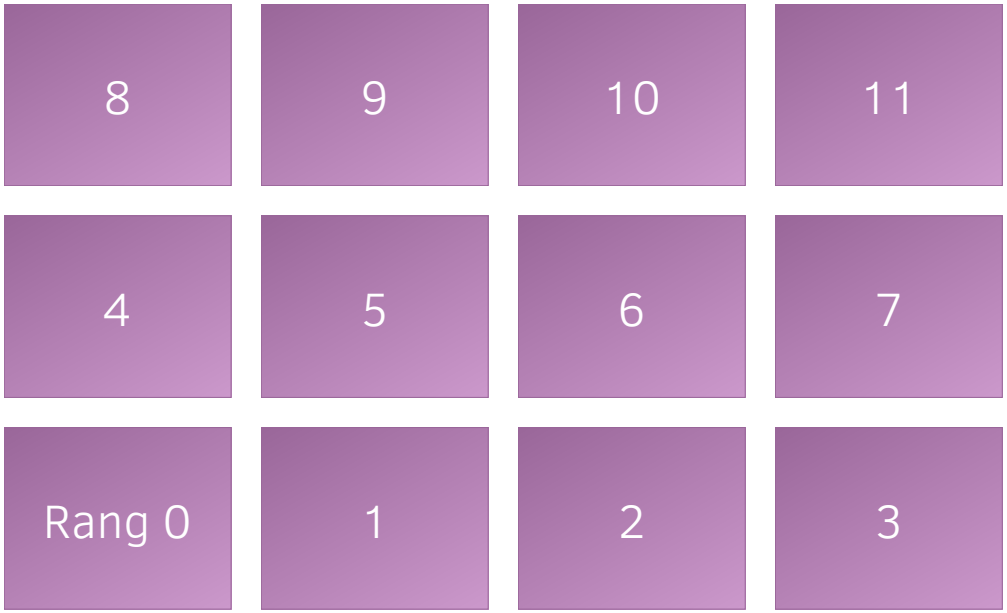
- Méthode de décomposition cartésienne

Il s'agit d'un exemple typique de parallélisme de donnée

# Décomposition de domaine cartésienne

Décomposition 1D


Décomposition 2D

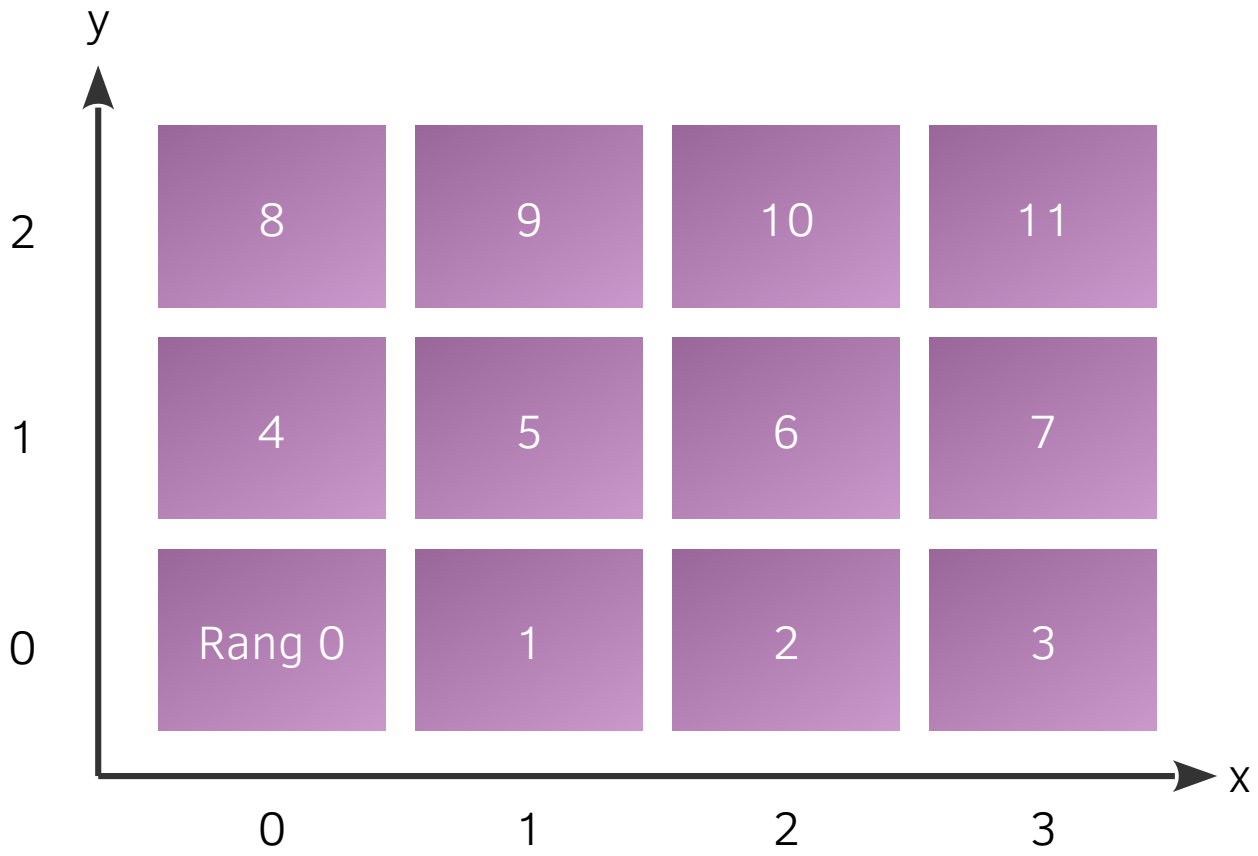


# Décomposition de domaine cartésienne : coordonnées

Une topologie cartésienne a besoin de :

- Coordonnées pour situer les processus (bloc) dans l'espace cartésien
- De rangs pour chaque processus en adéquation avec la topologie cartésienne
- exemple : le rang 5 a pour coordonnées (1,1)

Topologie cartésienne 2D



# Décomposition de domaine cartésienne : création

Deux solutions pour mettre en place une topologie cartésienne :

- Le faire à la main
- Faire appel aux fonctions MPI conçues pour ça

# Décomposition de domaine cartésienne : création via `MPI_Cart_create` (Fortran95)

`MPI_CART_CREATE` permet de définir une topologie cartésienne à partir d'un ancien communicateur (celui par défaut par exemple **`MPI_COMM_WORLD`**)


.f90

```
MPI_CART_CREATE(old_communicator, dimension,
                ranks_per_direction, periodicity,
                reorganisation,cartesian_communicator,ierror)
```

- **`old_communicator`** : ancien communicateur (**`MPI_COMM_WORLD`** par exemple)
- **`dimension`** (entier) : dimension de la topologie (2 pour 2D par exemple)
- **`ranks_per_direction`** (tableau d'entier) : le nombre de rangs dans chaque dimension
- **`Periodicity`** (tableau de booléens) : permet de définir les directions périodiques (true)
- **`Reorganisation`** (booléen) : réorganisation des rangs pour optimiser les échanges (true)
- **`cartesian_communicator`** (entier) : nouveau communicateur renvoyé par la fonction qui vient remplacer l'ancien communicateur.


https://www.open-mpi.org/doc/v4.0/man3/MPI_Cart_create.3.php

# Décomposition de domaine cartésienne : création via `MPI_Cart_create` (C/C++)

`MPI_Cart_create` permet de définir une topologie cartésienne à partir d'un ancien communicateur (celui par défaut par exemple **`MPI_COMM_WORLD`**)


.cpp

```
MPI_Cart_create(old_communicator, dimension,
                ranks_per_adirection, periodicity,
                reorganisation,cartesian_communicator) ;
```

- **`old_communicator (MPI_Comm)`** : ancien communicateur (**`MPI_COMM_WORLD`** par exemple)
- **`dimension`** (int) : dimension de la topologie (2 pour 2D par exemple)
- **`ranks_per_direction`** (int *) : le nombre de rangs dans chaque dimension
- **`Periodicity`** (int *) : permet de définir les directions périodiques
- **`Reorganisation`** (int) : réorganisation des rangs pour optimiser les échanges
- **`cartesian_communicator`** (MPI_Comm *) : nouveau communicateur renvoyé par la fonction qui vient remplacer l'ancien communicateur.


https://www.open-mpi.org/doc/v4.0/man3/MPI_Cart_create.3.php

# Décomposition de domaine cartésienne : création

Comme pour n'importe quel communicateur, on peut récupérer les rangs dans le communicateur cartésien (`cartesian_communicator`) avec `MPI_COMM_RANK`

# Décomposition de domaine cartésienne : récupérer les coordonnées d'un rang via `MPI_CART_COORDS` (Fortran95)

- `MPI_CART_COORDS` permet de récupérer les coordonnées d'un rang donné dans la topologie cartésienne.


```
CALL MPI_CART_COORDS(cartesian_communicator, rank, dimension, &
                                    rank_coordinates, ierror)
```

- `Dimension` (entier) : dimension de la topologie (2 pour 2D par exemple)
- `ranks_coordinates` (tableau d'entier) : les coordonnées du rang `rank` dans `cartesian_communicator`

https://www.open-mpi.org/doc/v2.0/man3/MPI_Cart_coords.3.php

# Décomposition de domaine cartésienne : récupérer les coordonnées d'un rang via `MPI_CART_COORDS` (C/C++)

- `MPI_Cart_coords` permet de récupérer les coordonnées d'un rang donné dans la topologie cartésienne.


.cpp

```
MPI_Cart_coords(cartesian_communicator, rank, dimension,
                                rank_coordinates) ;
```

- `Rank` (int) : rang du processus MPI
- `Dimension` (int) : dimension de la topologie (2 pour 2D par exemple)
- `ranks_coordinates` (int *) : les coordonnées du rang `rank` dans `cartesian_communicator`

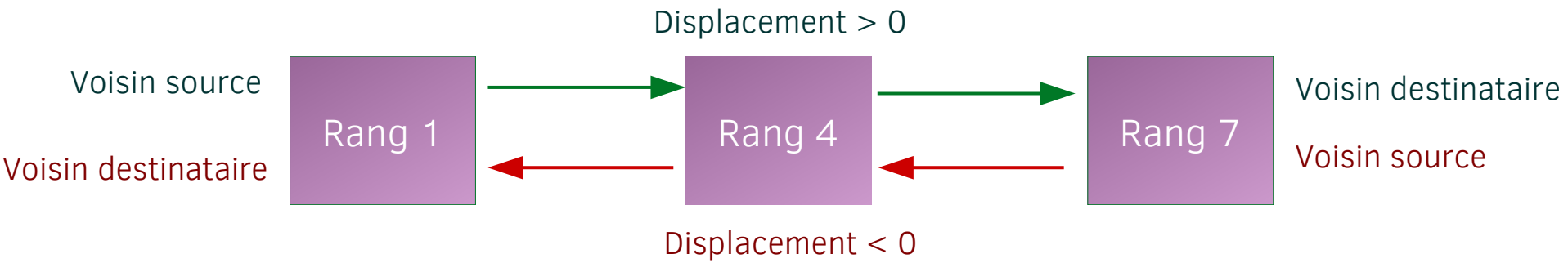
https://www.open-mpi.org/doc/v2.0/man3/MPI_Cart_coords.3.php

# Décomposition de domaine cartésienne : les voisins

- Chaque processus doit être en mesure de récupérer le rang de ses voisins dans la topologie cartésienne pour d'éventuelles communications.

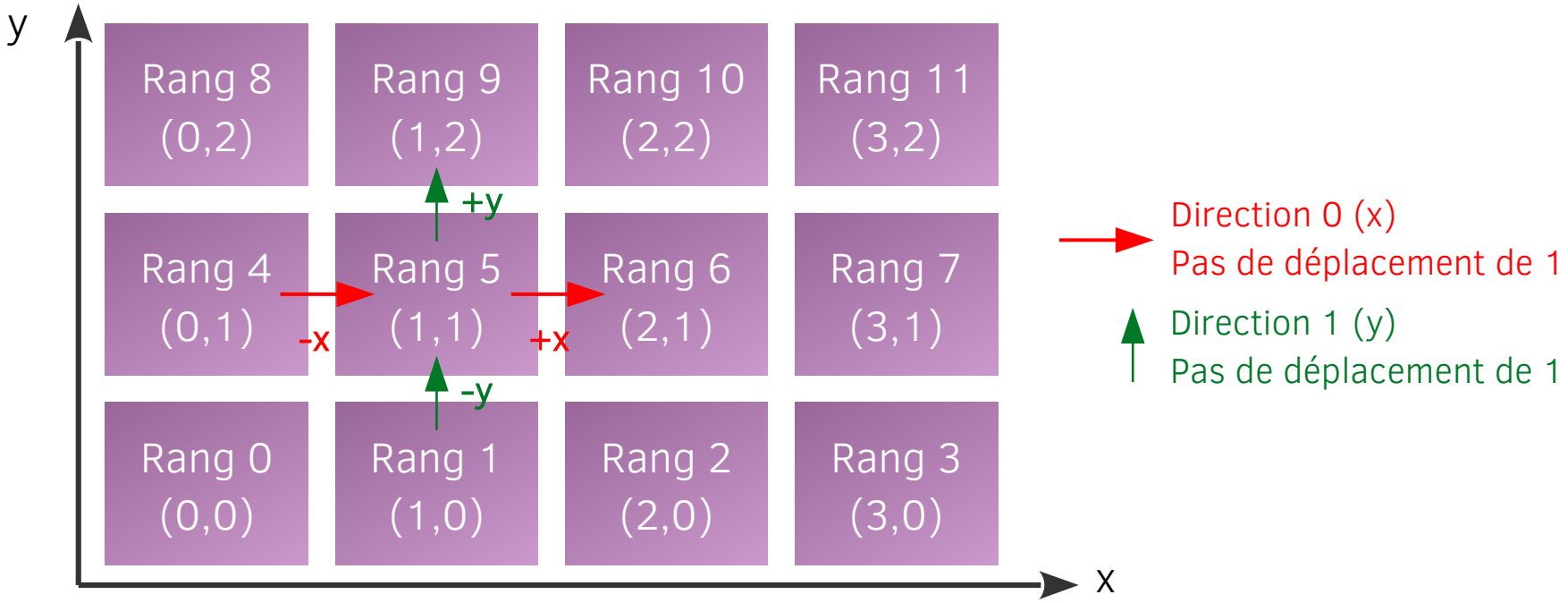
# Décomposition de domaine cartésienne : récupérer les rangs des voisins via `MPI_CART_SHIFT`

`MPI_CART_SHIFT` permet de récupérer les rangs voisins d'un rang donné en spécifiant une direction et un sens de déplacement. On récupère 2 voisins dans la philosophie MPI_Sendrecv : un rang source et un rang destinataire.

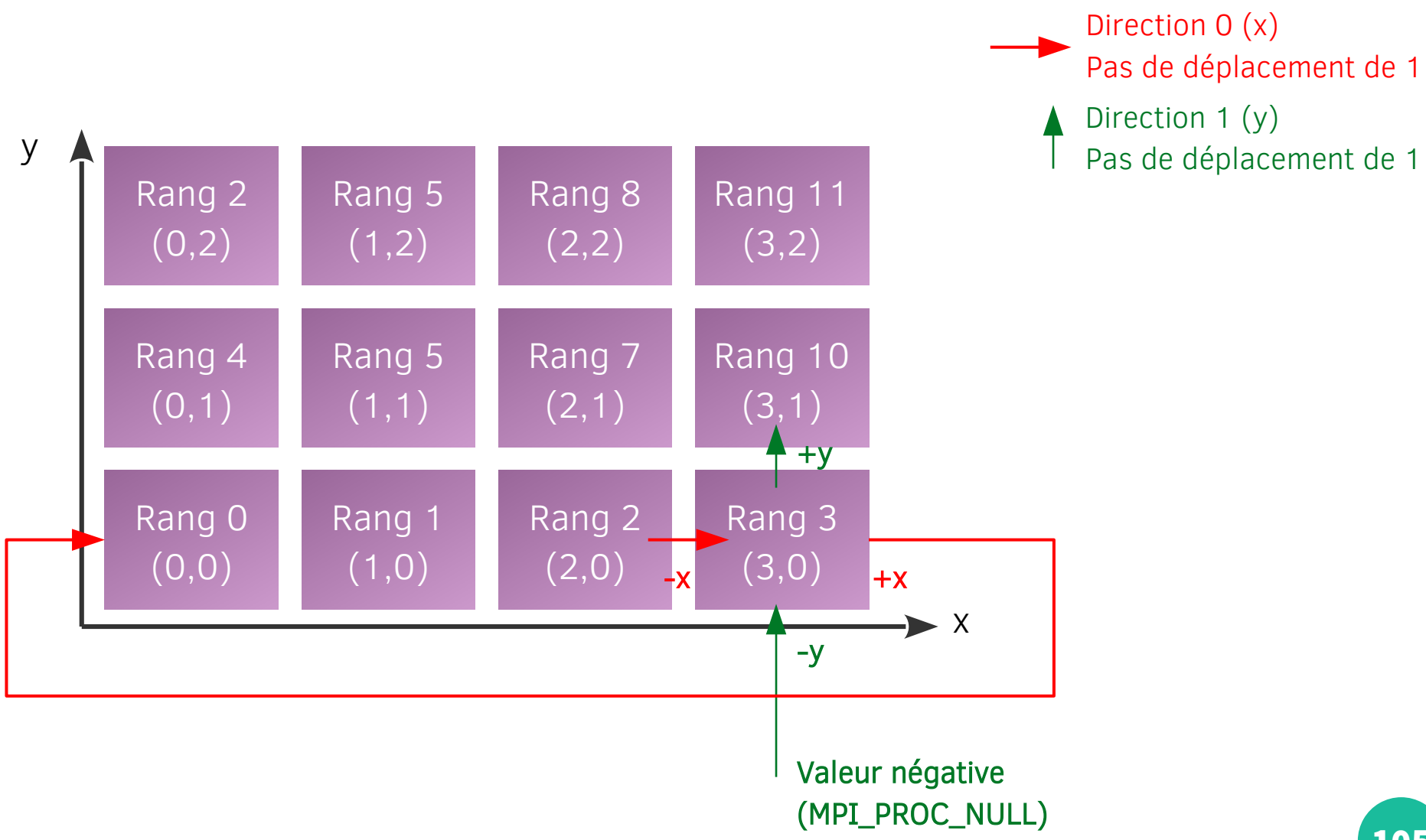



https://www.open-mpi.org/doc/v2.0/man3/MPI_Cart_shift.3.php

# Exemple de topologie cartésienne 2D



# Exemple de topologie cartésienne 2D : notion de périodicité



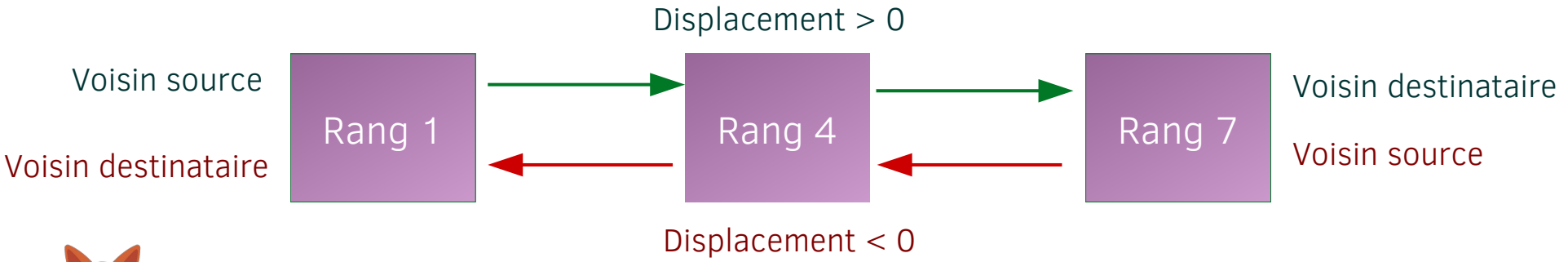
# Décomposition de domaine cartésienne : récupérer les rangs des voisins via `MPI_CART_SHIFT`

`MPI_CART_SHIFT` permet de récupérer les rangs voisins d'un rang donné


.f90

```
MPI_CART_SHIFT(cartesian_communicator, direction,
               displacement,
               rank_neighbors_src, rank_neighbors_dest, ierror)
```

- `direction` (integer) : 1 pour la première coordonnée, 2 pour la deuxième coordonnée
- `Displacement` (integer) : pas de déplacement dans la direction souhaitée, si > 0 déplacement vers les coordonnées supérieures, si < 0 vers les coordonnées inférieures
- `rank_neighbord_source` : si displacement > 0, correspond au voisin source
- `rank_neighbord_dest` : si displacement > 0, correspond au voisin destinataire




https://www.open-mpi.org/doc/v2.0/man3/MPI_Cart_shift.3.php

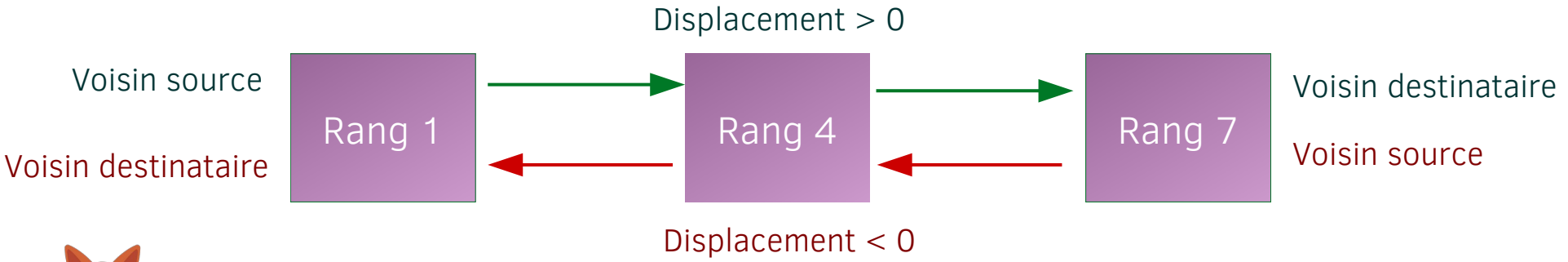
# Décomposition de domaine cartésienne : récupérer les rangs des voisins via `MPI_CART_SHIFT` (C/C++)

`MPI_Cart_shift` permet de récupérer les rangs voisins d'un rang donné

.cpp

```
MPI_Cart_shift(cartesian_communicator, direction,
               displacement,
               rank_neighbors_src, rank_neighbors_dest) ;
```

- `direction` (int) : 1 pour la première coordonnée, 2 pour la deuxième coordonnée
- `Displacement` (int) : pas de déplacement dans la direction souhaitée, si > 0 déplacement vers les coordonnées supérieures, si < 0 vers les coordonnées inférieures
- `rank_neighbord_source` : si displacement > 0, correspond au voisin source
- `rank_neighbord_dest` : si displacement > 0, correspond au voisin destinataire




https://www.open-mpi.org/doc/v2.0/man3/MPI_Cart_shift.3.php

# Décomposition de domaine cartésienne : récupérer les rangs des voisins via `MPI_CART_SHIFT` (C/C++)

Lorsqu'un rang n'a pas de voisin (par exemple en non-périodique), `MPI_CART_SHIFT` renvoie `MPI_PROC_NULL`.

Lorsqu'une communication a pour destinataire ou expéditeur MPI_PROC_NULL, elle peut être écrite mais la communication est simplement ignorée. Cela permet de gérer la périodicité sans avoir à multiplier les conditions par exemple.

https://www.open-mpi.org/doc/v2.0/man3/MPI_Cart_shift.3.php

# Exemple de topologie cartésienne 2D (Fortran95)


```
Integer, dimension(2) :: ranks_per_direction = (/4, 3/)
Logical, dimension(2) :: periodicity = (/ .true., .true. /)
Logical               :: reorganisation = .true.
Integer               :: cartesian_communicator
Integer, dimension(2) :: rank_coordinates
Integer               :: rank_neighbors_mx, rank_neighbors_px
Integer               :: rank_neighbors_my, rank_neighbors_py

Call MPI_INIT(ierror)

Call MPI_CART_CREATE(MPI_COMM_WORLD, 2,ranks_per_direction, periodicity,
                     reorganisation,cartesian_communicator,ierror)

Call MPI_COMM_RANK(cartesian_communicator, rank, ierror)

Call MPI_CART_COORDS(cartesian_communicator, rank, 2, rank_coordinates, ierror)

CALL MPI_CART_SHIFT(cartesian_communicator, 1, 1, &
                    rank_neighbors_my, rank_neighbors_py, ierror)

CALL MPI_CART_SHIFT(cartesian_communicator, 0, 1, &
                    rank_neighbors_mx, rank_neighbors_px, ierror)
```

# Exemple de topologie cartésienne 2D (C/C++)

.cpp

```
int ranks_per_direction[2] = {4, 3} ;
int periodicity[2] = {1, 1} ;
int reorganisation = 1 ;
MPI_Comm cartesian_communicator ;
int rank_coordinates[2] ;
int rank_neighbors_mx, rank_neighbors_px ;
int rank_neighbors_my, rank_neighbors_py ;

ierror = MPI_Init(ierror)

Ierror = MPI_Cart_create(MPI_COMM_WORLD, 2,ranks_per_direction, periodicity,
                         reorganisation,&cartesian_communicator) ;

ierror = MPI_Comm_rank(cartesian_communicator, rank) ;

ierror = MPI_Cart_coords(cartesian_communicator, rank, 2, &rank_coordinates) ;

ierror = MPI_Cart_shift(cartesian_communicator, 1, 1,
                        &rank_neighbors_my, &rank_neighbors_py) ;

ierror = MPI_Cart_shift(cartesian_communicator, 0, 1,
                        &rank_neighbors_mx, &rank_neighbors_px) ;
```

# Exercice n°7 : Création d'une topologie cartésienne

- Rendez vous dans le dossier de l'exercice n°7 appelé `7_cartesian_com`

```
> cd exercises/mpi/7_cartesian_com
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit…) ou visualisez directement les instructions sur le GitLab.

# Décomposition de domaine cartésienne

A ce stade du cours, vous savez maintenant :

- Créer un communicateur cartésien pour décomposer vos données
- Utiliser les fonctions liées à la décomposition cartésienne

# Introduction au parallélisme par échange de message via MPI

## 6) Types dérivés

# Types dérivés

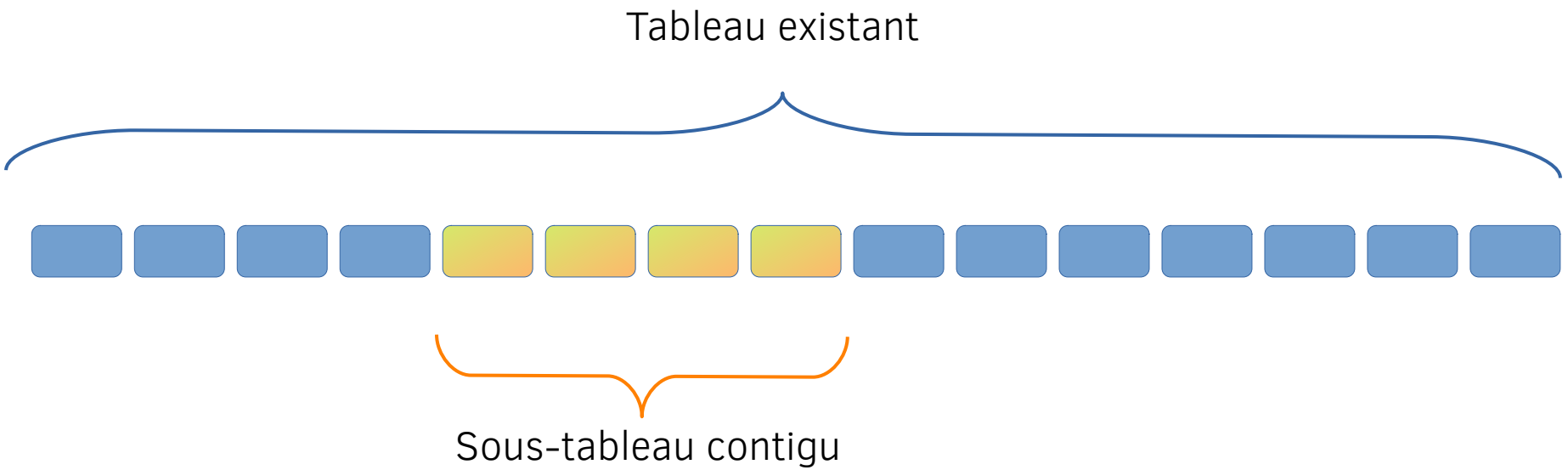
Les types dérivés permettent de décrire des données plus complexes que les types classiques (MPI_INT, MPI_DOUBLE…)

- **`MPI_Type_contiguous`** : permet de sélectionner une portion contiguë d'un tableau
- **`MPI_Type_vector`** : permet de créer un sous-tableau à partir de sous-ensemble d'éléments constant séparés par un déplacement constant (stride)
- **MPI_Type_indexed** : permet de créer un sous-tableau à partir de sous-ensemble d'éléments variables séparés par un pas variable
- **MPI_Type_create_struct** : permet de créer l'équivalent d'une structure C en mélangeant les types de base

Les types dérivés sont ensuite utilisés dans les communications à la place des types classiques. Ils permettent au développeur de ne pas avoir à créer de structure intermédiaire à la main et de minimiser le nombre d'appel MPI.

# Types dérivés : MPI_Type_contiguous

MPI_Type_contiguous : permet de sélectionner une portion contiguë d'un tableau existant



# Types dérivés : MPI_Type_contiguous

La fonction `MPI_Type_contiguous` contient les arguments suivants :

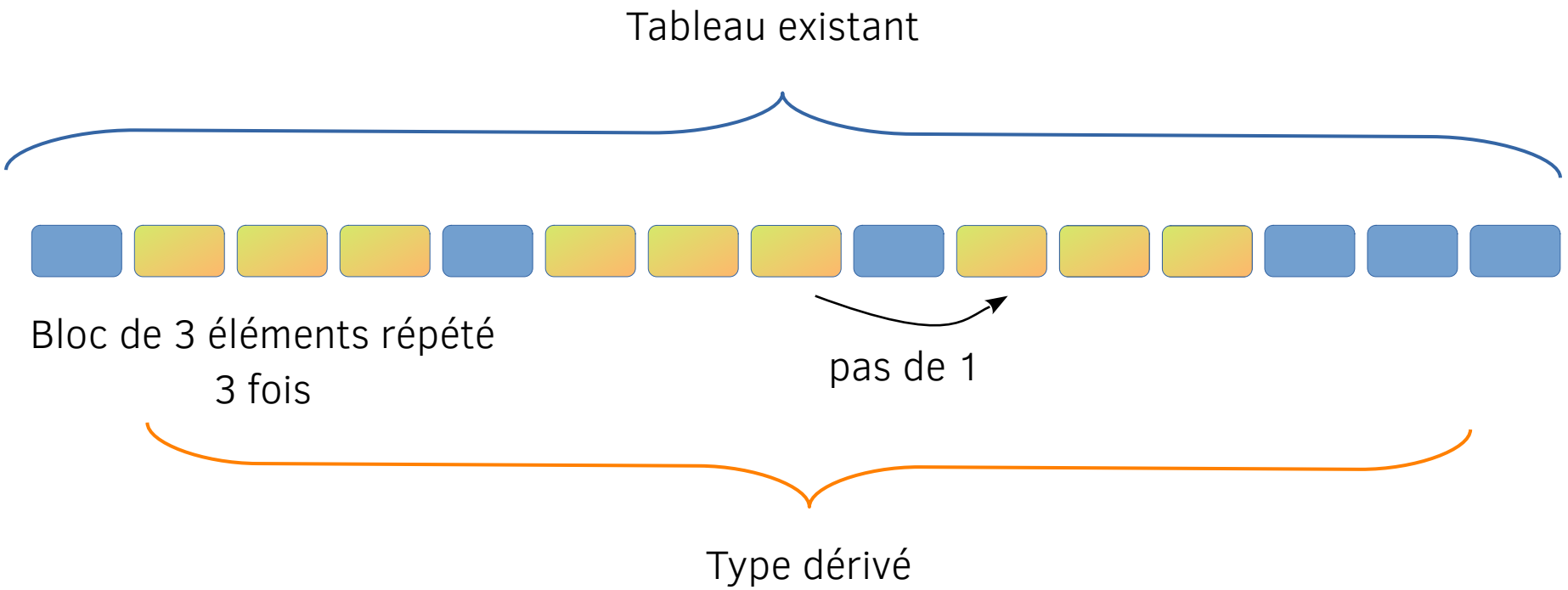

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

- `count (int)` : nombre d'éléments qui compose le nouveau type
- `oldtype (MPI_Datatype)` : le type de donnée qui compose le tableau initial
- `newtype (MPI_Datatype)` : notre nouveau type dérivé

https://www.open-mpi.org/doc/current/man3/MPI_Type_contiguous.3.php

# Types dérivés : MPI_Type_vector

MPI_Type_vector : permet de créer un sous-tableau à partir de sous-ensemble d'éléments constant séparés par un déplacement constant (stride)



# Types dérivés : MPI_Type_vector

La fonction `MPI_Type_vector` contient les arguments suivants :


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

- `count (int)` : nombre de blocs
- `Blocklength (int)` : la taille en nombre d'éléments de chaque bloc
- `Stride (int)` : distance (pas) entre chaque bloc
- `oldtype (MPI_Datatype)` : le type de donnée qui compose le tableau initial
- `newtype (MPI_Datatype)` : notre nouveau type dérivé

https://www.rookiehpc.com/mpi/docs/mpi_type_vector.php

# Création d'un type dérivé

La fonction `MPI_Type_commit` permet officialiser la création du type :

Bloc de 3 éléments répété 3 fois

pas de 1

.cpp

```
MPI_Datatype column_type;

MPI_Type_vector(3, 3, 1, MPI_INT, &column_type);

MPI_Type_commit(&column_type);
```

https://www.rookiehpc.com/mpi/docs/mpi_type_commit.php

# Exemple complet d'utilisation d'un type dérivé

Dans cette exemple, le rang 0 envoie de l'information au rang 1 à partir du tableau buffer et d'un type vector.

.cpp

```
 If (rank == 0) {
            MPI_Datatype vector_type;
            MPI_Type_vector(3, 3, 1, MPI_INT, &vector_type);
            MPI_Type_commit(&vector_type);

            int buffer[12];

            MPI_Send(&buffer[1], 1, vector_type, 1, 0, MPI_COMM_WORLD);
}

If (rank == 1) {
            int received[9];
            MPI_Recv(&received, 9, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

# Exercice n°8 : Utilisation du type dérivé vector

- Rendez vous dans le dossier de l'exercice n°8 appelé `8_type_vector`

```
> cd exercises/mpi/8_type_vector
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit…) ou visualisez directement les instructions sur le GitLab.