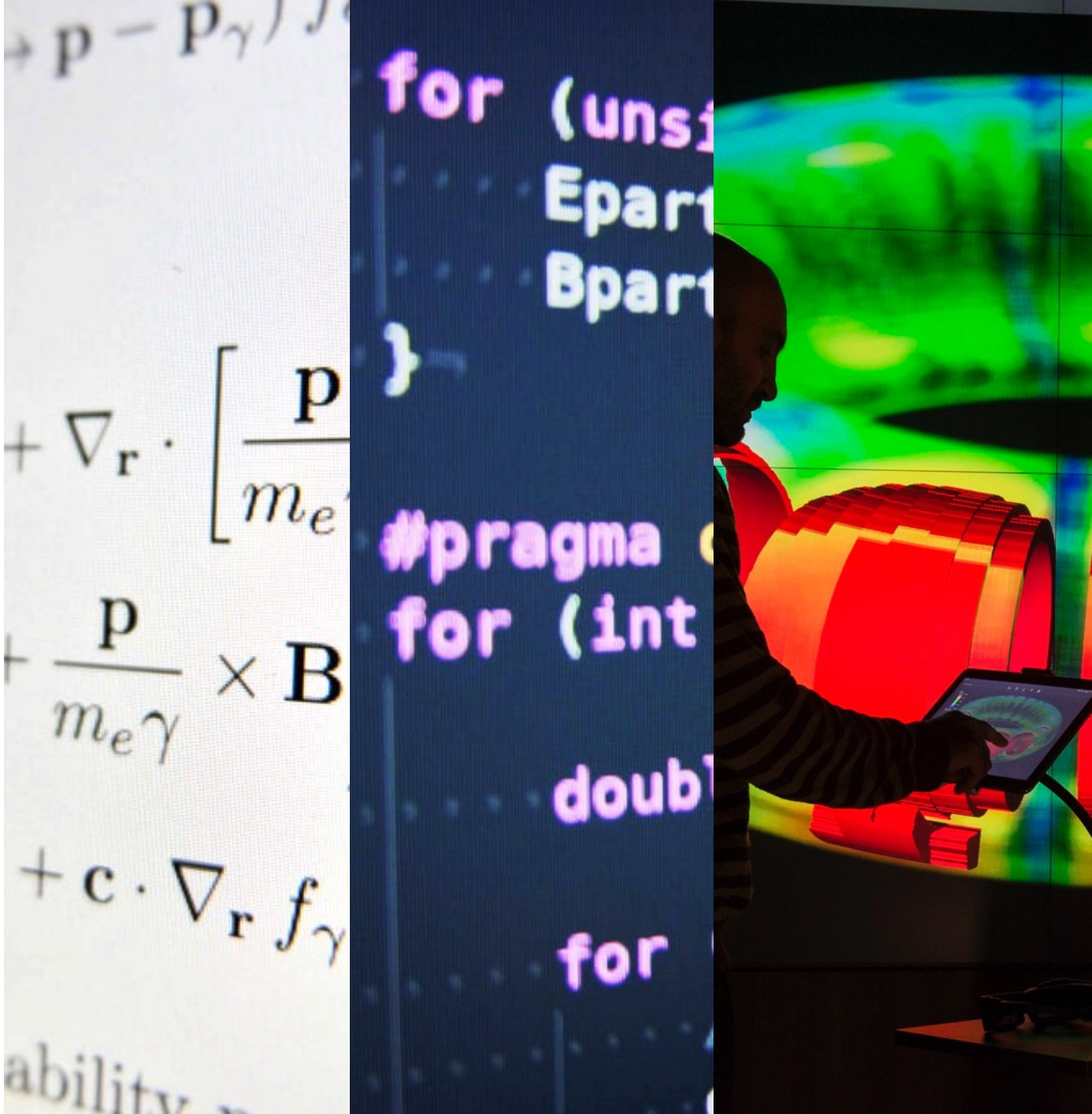
Introduction au parallélisme par échange de messages via MPI

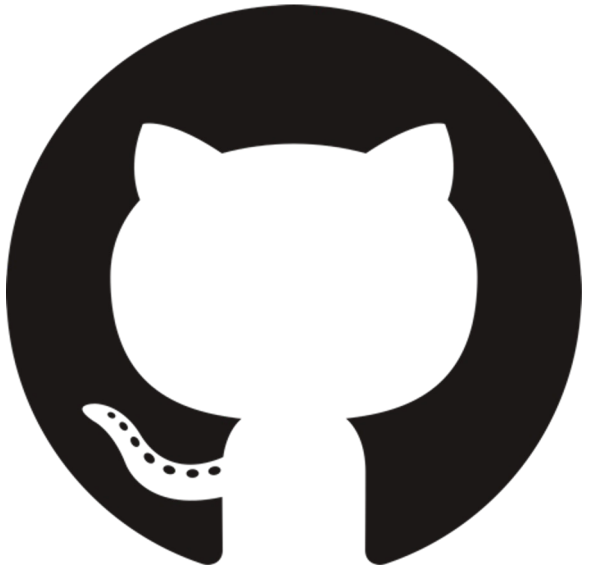
Mathieu LOBET
Maison de la Simulation, CEA Saclay

Mathieu.lobet@cea.fr

Année 2023-2024



Ce cours est en accès libre sur la plateforme GitHub



https://github.com/Maison-de-la-Simulation/HPC-DFE-Paris-Saclay

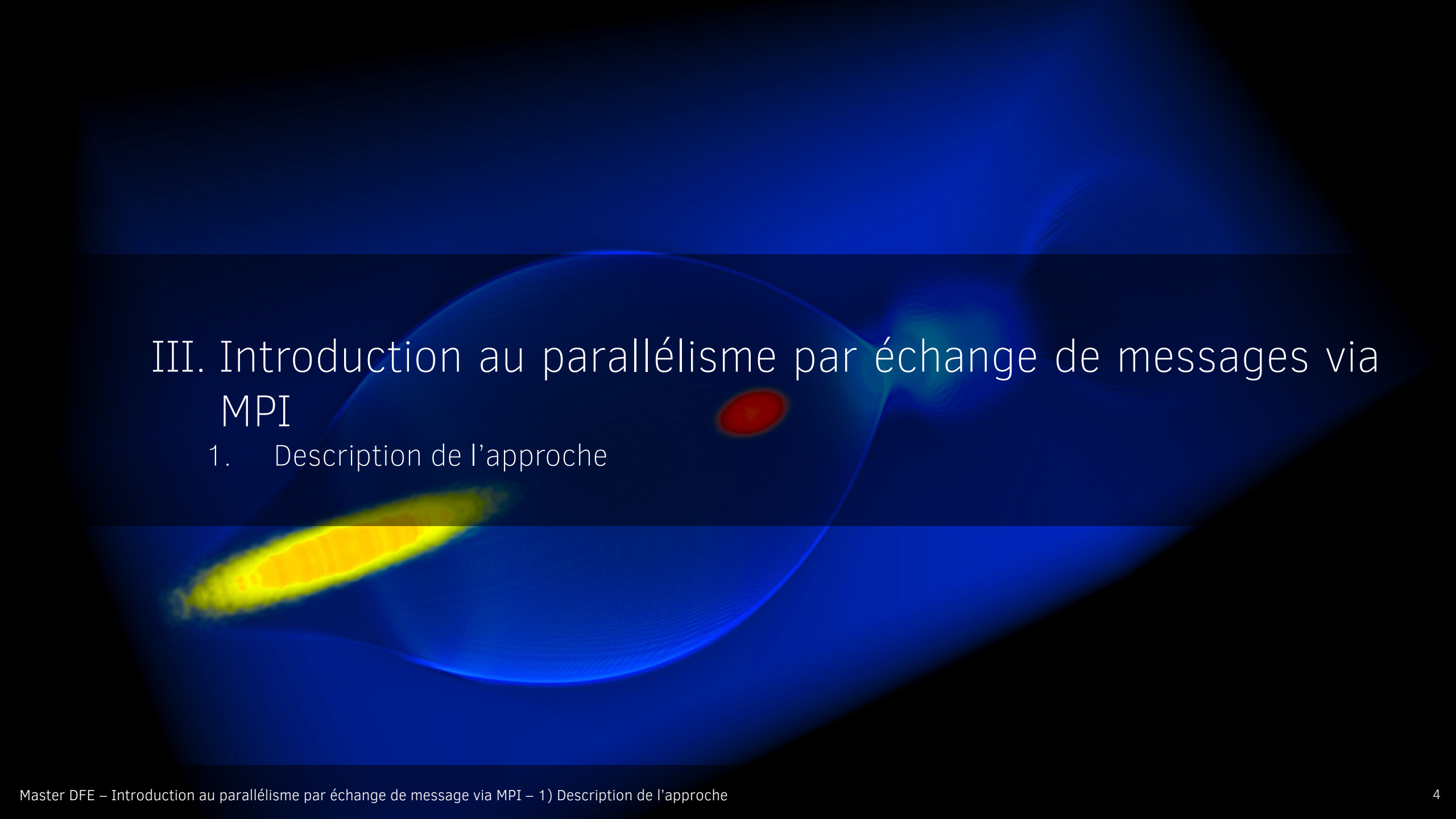
I. Introduction au calcul parallèle

II. Prise en main d'un super-calculateur

III. Introduction au parallélisme par échange de message via MPI

IV. Mesure de la performance

V. Travaux pratiques



III. Introduction au parallélisme par échange de messages via MPI

1. Description de l'approche

Cours et matériel supplémentaires sur internet

- Les codes de simulation sont rarement basés sur une approche Python + MPI, ce sont le plus souvent des codes en C, C++ ou Fortran

- Pour faciliter l'apprentissage, ce cours vous présente MPI au travers de Python. L'API Fortran ou C est un peu plus complexe à utiliser mais les bases sont les mêmes

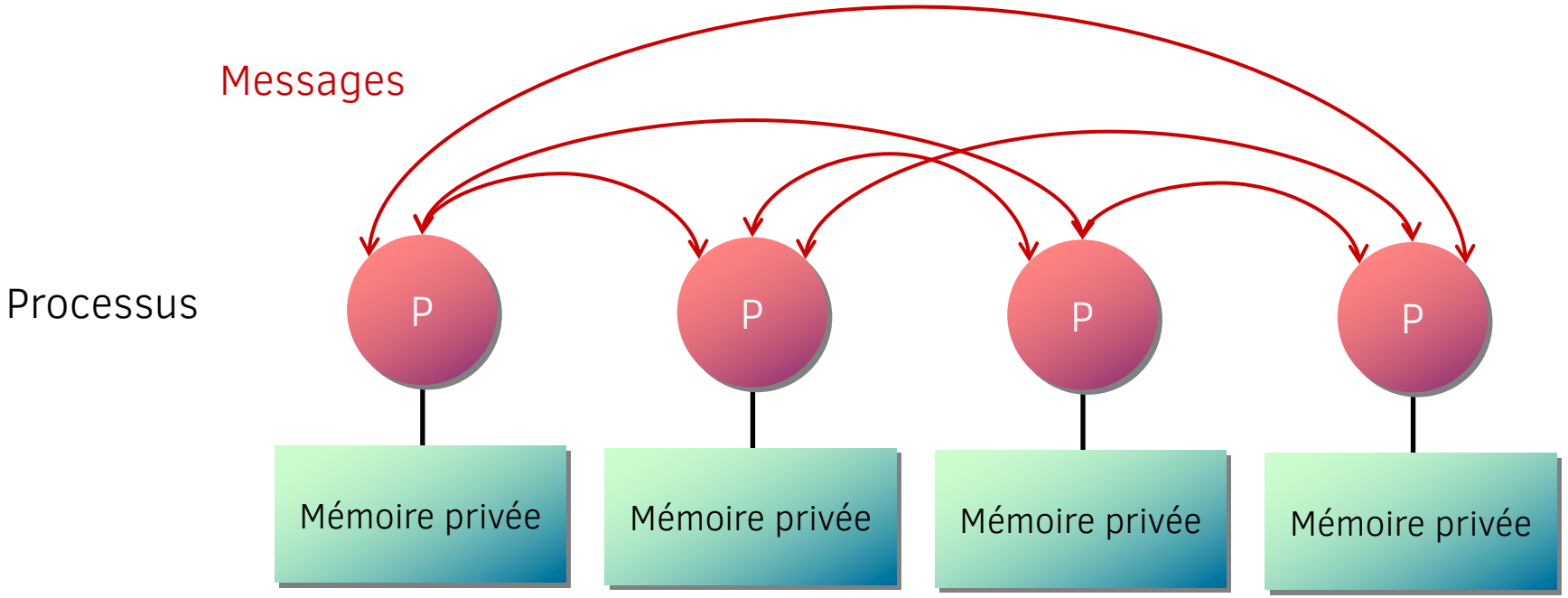
- Selon moi, le cours le plus complet sur MPI en français et anglais: http://www.idris.fr/formations/mpi/

- 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/

- MPI pour Python :
  - https://mpi4py.readthedocs.io/en/stable/tutorial.html

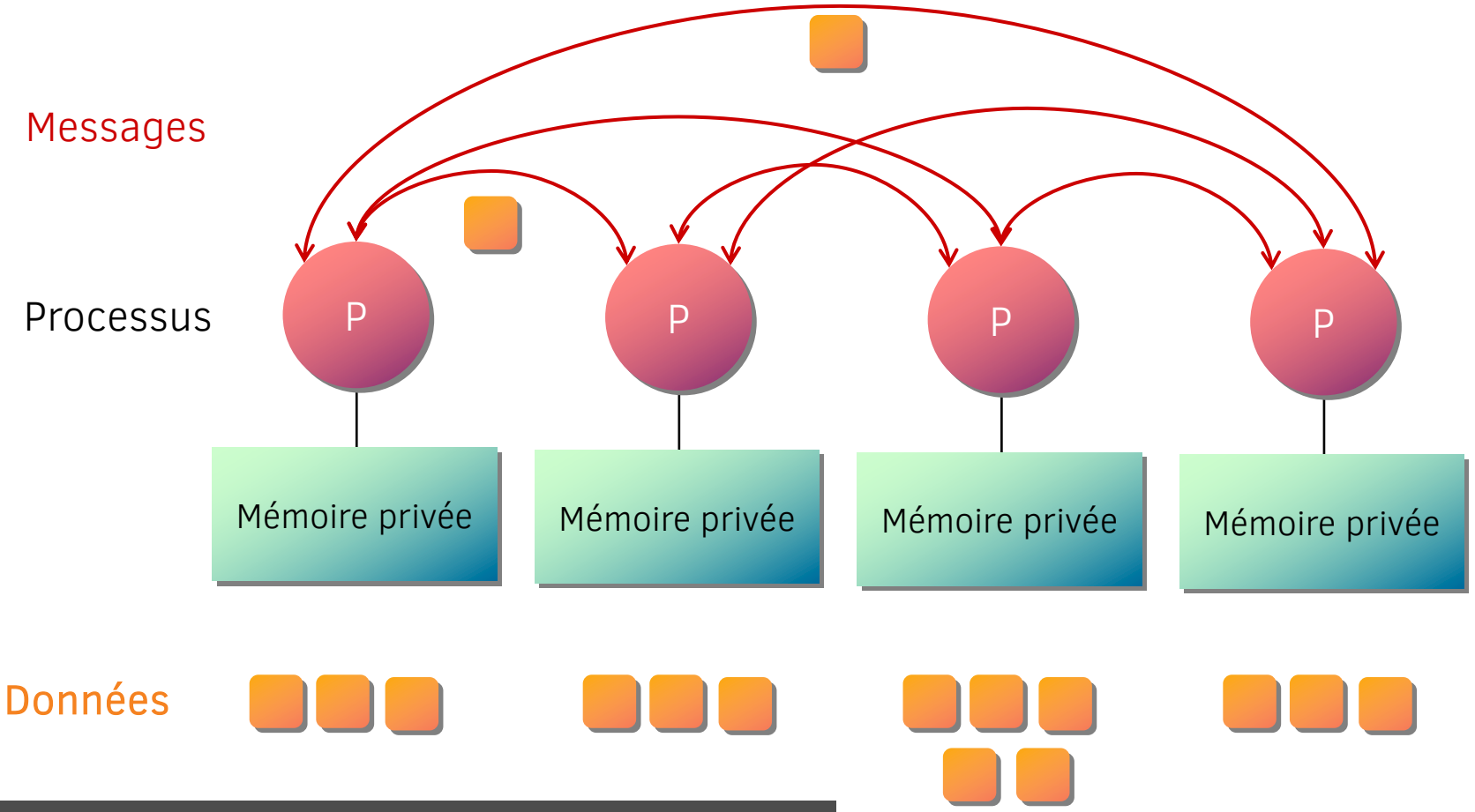
Paradigme par passage de messages : définition

Dans un modèle par échange de messages, les différents exécutants (*process*) ont leurs données propres qu'ils communiquent entre eux par messages. Ces messages peuvent servir à échanger une ou plusieurs données, ou se synchroniser.

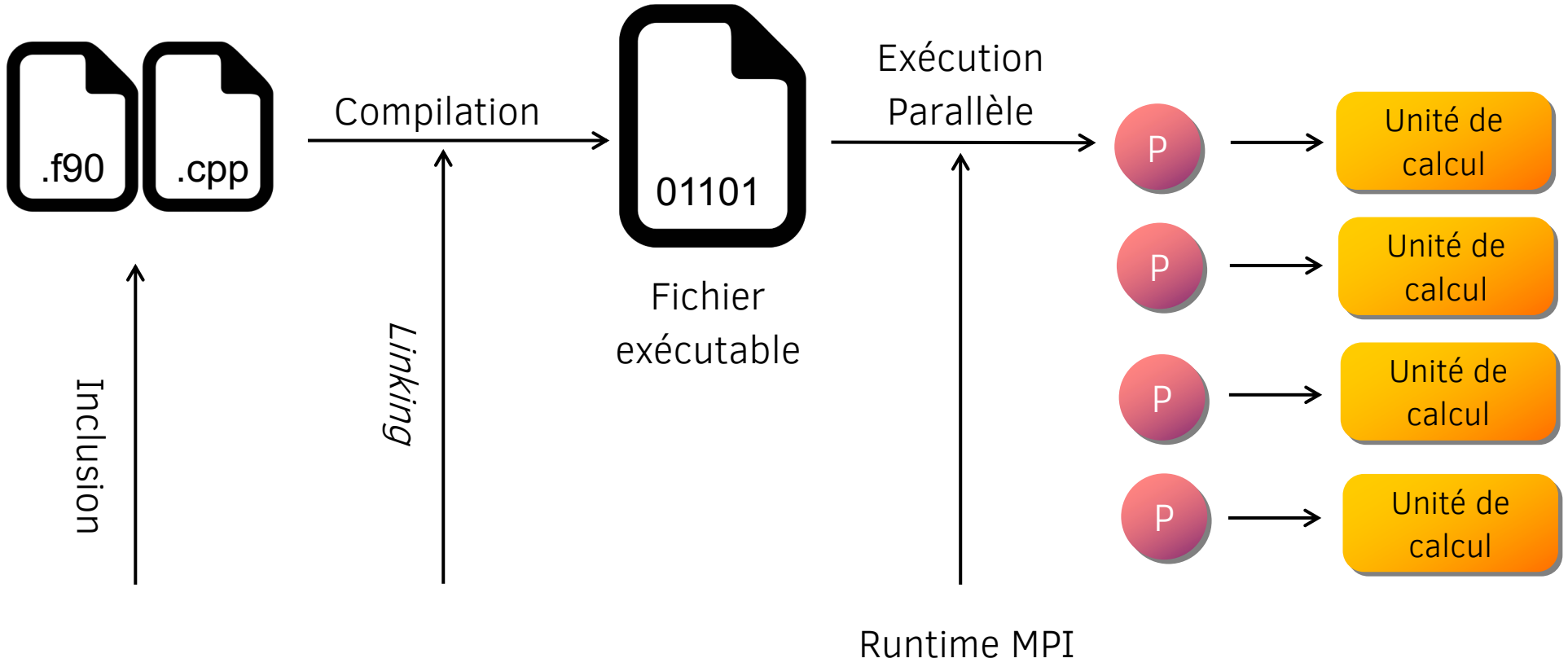


Paradigme par passage de messages : définition

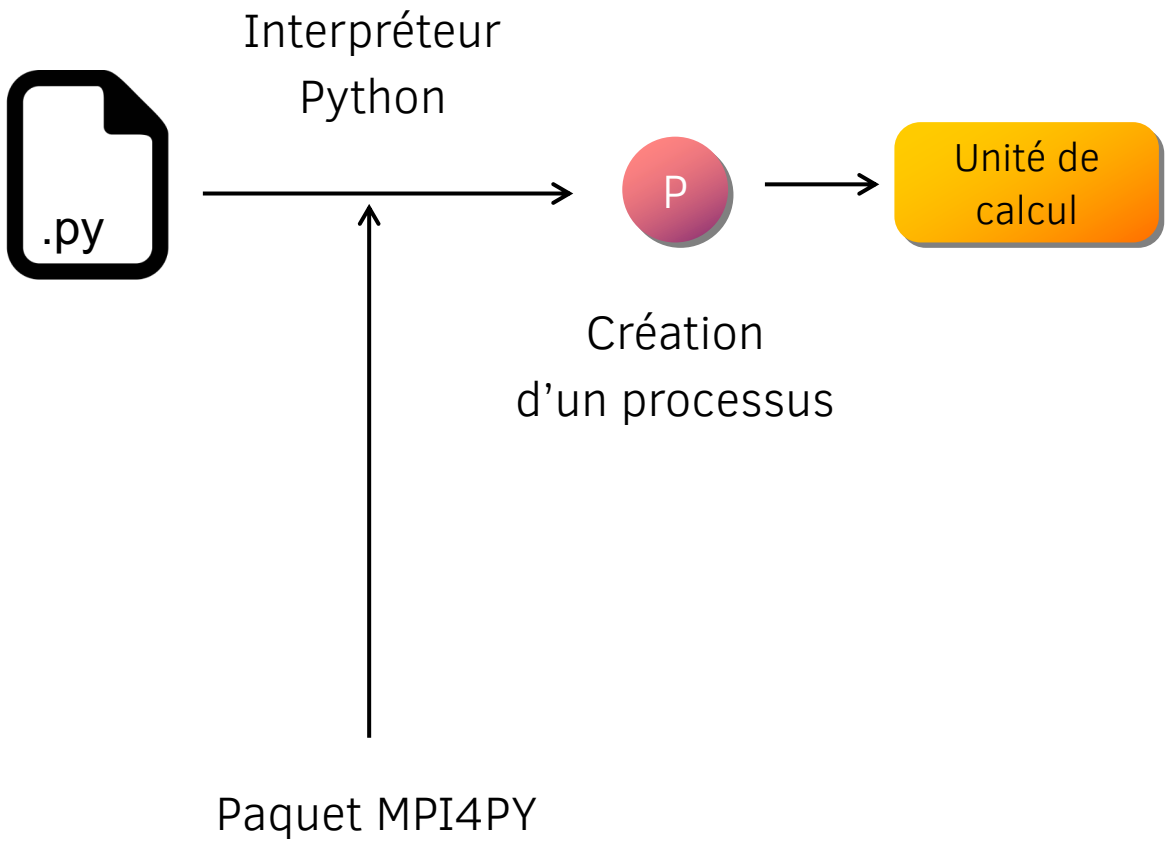
Dans un modèle par échange de messages, les différents exécutants (*process*) ont leurs données propres qu'ils communiquent entre eux par messages. Ces messages peuvent servir à échanger une ou plusieurs données, ou se synchroniser.



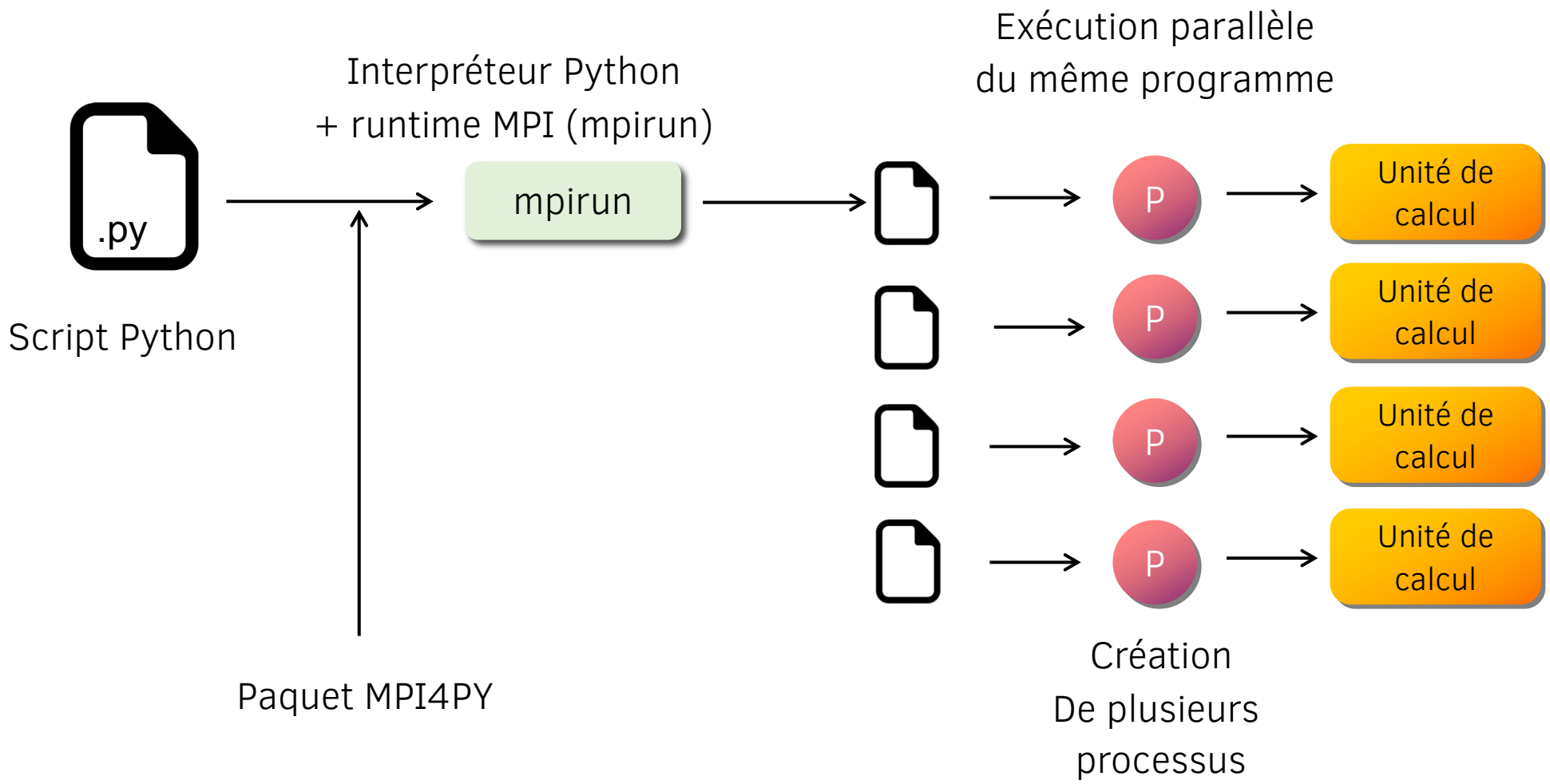
Programme parallèle avec appel aux fonctions MPI



L’approche python séquentiel



L'approche MPI + Python



Compilation et exécution d'un programme MPI avec Python

- Contrairement à des langages compilés comme Fortran ou C, Python ne nécessite pas de compilation
- Pour l'exécution, on utilise la commande mpirun avant l'appel à python:
  - -np représente le nombre de processus MPI à lancer.
  - Si le nombre de processus MPI est inférieur au nombre d'unités de calcul disponibles, 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 avec N processus :

```
> mpirun -np N python mon_script_python.py
```

Démarrage d'un programme MPI : notion de communicateur

- Un communicateur est un ensemble de processus MPI capables de communiquer entre eux.
- Au sein d'un communicateur, chaque processus MPI est représenté par un rang (*rank*) unique sous forme d'un entier, c'est en quelque sorte son adresse.
- Le communicateur par défaut regroupe l'ensemble des processus et se nomme MPI_COMM_WORLD.
- C'est en quelque sorte l'annuaire de tous les processus en jeu

```
> mpirun -np 4 python script.py
```

MPI_COMM_WORLD : Communicateur composé de 4 rangs

Démarrage d'un programme MPI : notion de sous-communicateur

- Nous verrons qu'il est aussi possible de créer des sous-communicateurs

```
> mpirun -np 4 python script.py
```

MPI_COMM_WORLD : Communicateur composé de 4 rangs

Démarrage d'un programme MPI : inclure la bibliothèque Python mpi4py

- Il est nécessaire comme pour chaque bibliothèque d'importer le paquet mpi4py avant son utilisation

script.py

```
# Exemple de script Python

import mpi4py
from mpi4py import MPI

import numpy
…
```

Démarrage d'un programme MPI : initialiser MPI

- MPI est une bibliothèque qui fonctionne par appel à des fonctions fournies par une API
- Pour commencer à utiliser MPI, il faut d'abord appeler la fonction d'initialisation de MPI
- En Python l'étape d'initialisation peut être implicite mais nous la rendons explicite dans ce cours

script.py

```
# Exemple de script Python

import mpi4py
from mpi4py import MPI

# On rend l'init explicite
mpi4py.rc.initialize = False
mpi4py.rc.finalize = False

# On initialise MPI
MPI.Init()

…
```

Démarrage d'un programme MPI : finalisation MPI

- On doit à la fin du programme finaliser MPI proprement
- C'est surtout très important en C, C++ et Fortran

script.py

```
# Exemple de script Python

import mpi4py
from mpi4py import MPI

# On rend l'init explicite
mpi4py.rc.initialize = False
mpi4py.rc.finalize = False

# On initialise MPI
MPI.Init()

…

# On finalise MPI
MPI.Finalize()
```

Démarrage d'un programme MPI : récupération du communicateur global

- Le communicateur global est MPI_COMM_WORLD, on peut le récupérer via MPI.COMM_WORLD en Python

script.py

```
# Exemple de script Python

import mpi4py
from mpi4py import MPI

# On rend l'init explicite
mpi4py.rc.initialize = False
mpi4py.rc.finalize = False

# On initialise MPI
MPI.Init()

# On récupère le communicateur global
comm = MPI.COMM_WORLD

…
```

Démarrage d'un programme MPI : récupération du nombre de rangs

- Le nombre de rangs dans un communicateur se récupère via la fonction MPI_COMM_SIZE
- Pour MPI_COMM_WORLD c'est le nombre total de processus demandé à l'exécution.

script.py

```
# Exemple de script Python

…

# On récupère le communicateur global
comm = MPI.COMM_WORLD

# On récupère le nombre de processus
number_of_ranks = comm.Get_size()


…
```

Démarrage d'un programme MPI : récupérer le rang de chaque processus MPI

- Pour rappel, le rang est un identifiant unique permettant de désigner chaque processus.
- Chaque processus récupère son rang dans le communicateur **MPI_COMM_WORLD** via la fonction MPI_COMM_RANK.

script.py

```
# Exemple de script Python

…

# On récupère le communicateur global
comm = MPI.COMM_WORLD

# On récupère le nombre de processus
number_of_ranks = comm.Get_size()

# Chaque process récupère son numéro de rang dans le communicateur par défaut
rank = comm.Get_rank()


…
```

Exercice 1 : votre premier programme MPI

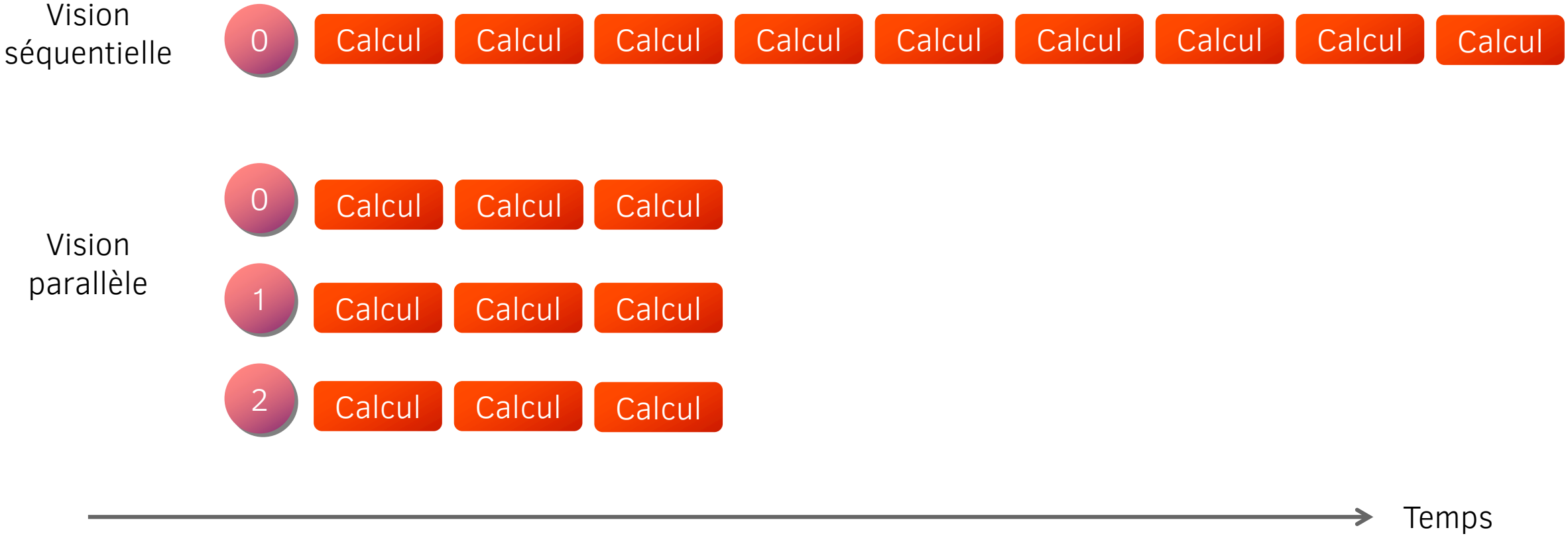
- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/1_initialization.md
```

- Cet exercice a pour but de vous apprendre à concevoir vos premiers programmes MPI et de le lancer

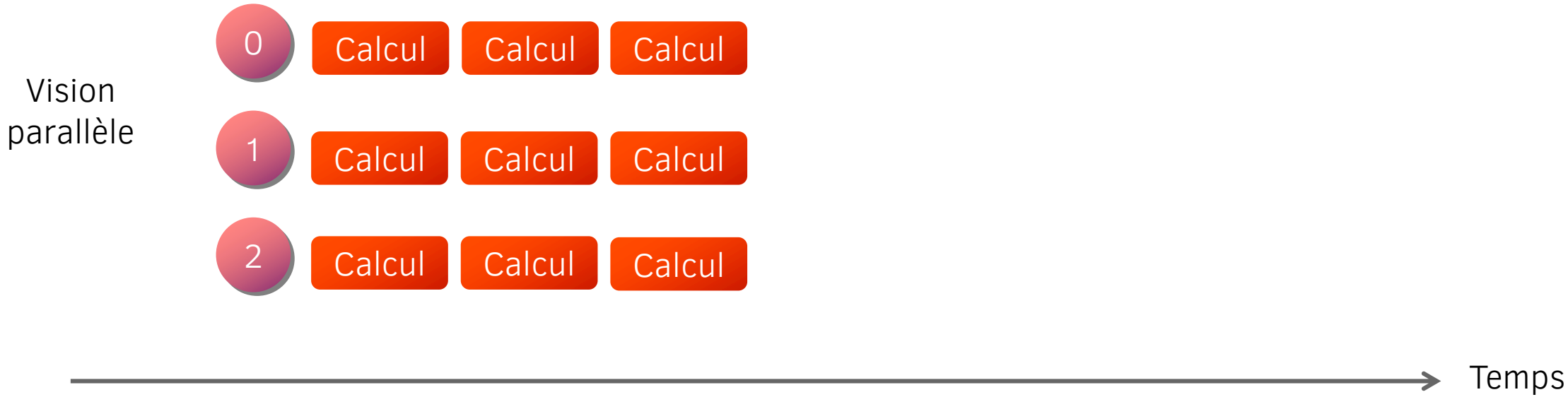
Notion de parallélisme

- On parle de parallélisme lorsque chaque processus travaille sur ses tâches en même temps que les autres
- La base du parallélisme consiste donc à découper une charge de travail en sous-tâches pour les distribuer sur l'ensemble des travailleurs



Notion de parallélisme parfait

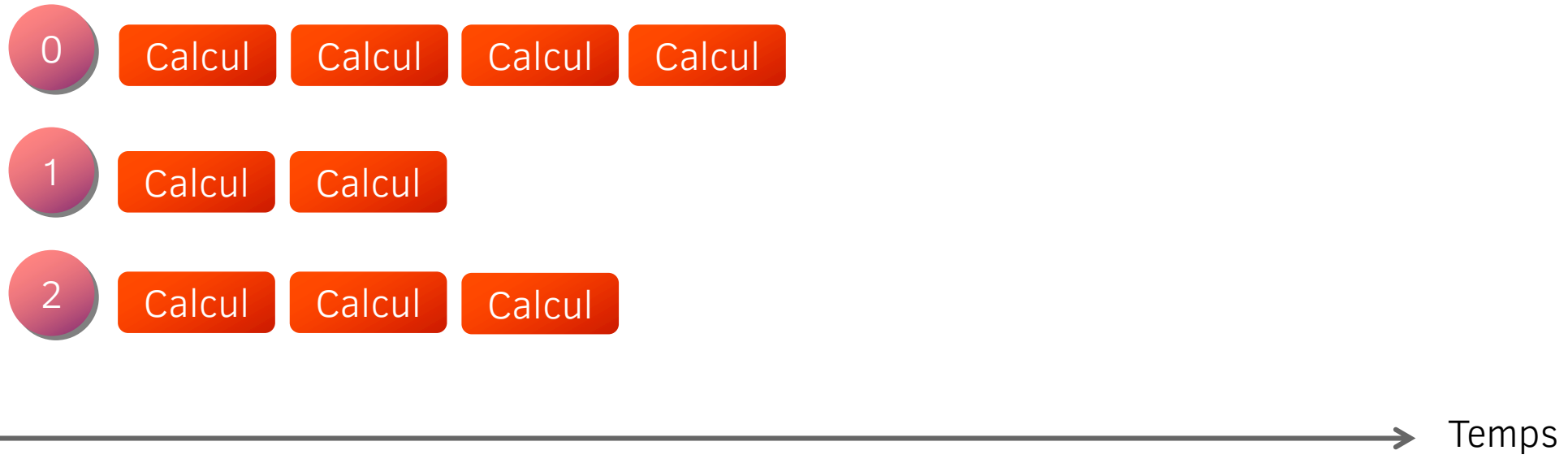
- Le parallélisme est ici parfait : tous les travailleurs ont la même charge de travail



Notion de déséquilibre de charge

- Dans le cas contraire, on parle de déséquilibre de charge

Vision parallèle avec déséquilibre de charge



Mesure du temps sur chaque processus

- MPI.Wtime permet de récupérer le temps écoulé sur le processus courant en secondes

script.py

```
# Exemple de script Python

…

# On récupère un nombre de secondes depuis l'init MPI
Time = MPI.Wtime()

…
```

Mesure du temps sur chaque processus

- Par deux appels et une soustraction, cette fonction permet de déterminer le temps passer dans une section du code
- La mesure du temps permet d'estimer le déséquilibre de charge

script.py

```
# Exemple de script Python

…

# On récupère un nombre de secondes depuis l'init MPI
Time_start = MPI.Wtime()

# Phase de calul
…

Time_stop = MPI.Wtime() - time_start;
```

Différencier du code pour des processus donnés

- Le programme s'exécute simultanément autant de fois qu'il y a de processus en 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.

script.py

```
# Exemple de script Python

# Chaque process récupère son numéro de rang dans le communicateur par défaut
rank = comm.Get_rank()

if (rank == 1):
    # Faire des actions que sur le processus de rang 1
    …
```

Exercice 2 : parallèle parfait

- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

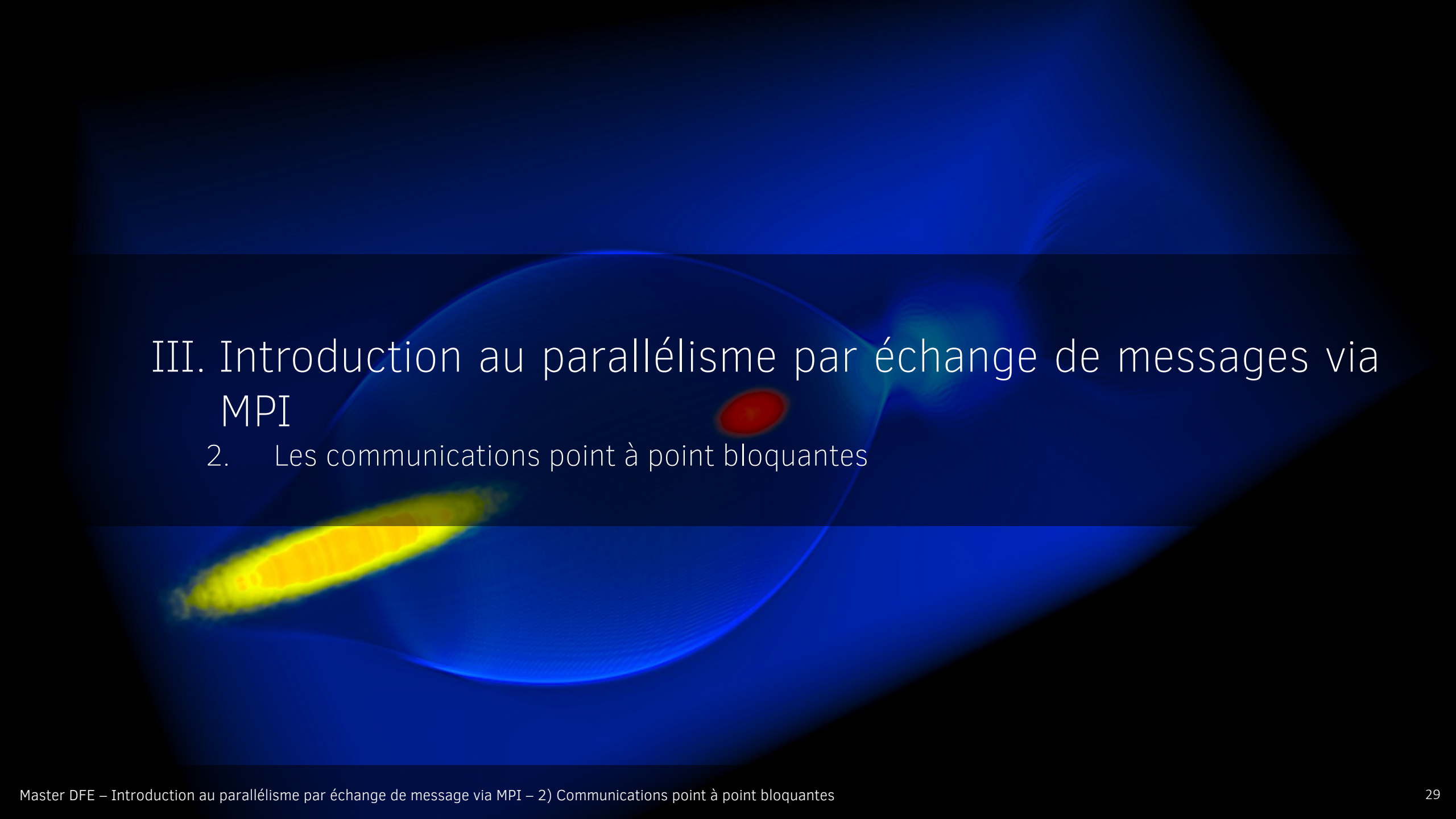
```
> cd exercices/mpi/python/2_embarrassingly_parallel.md
```

- Cet exercice a pour but de vous familiariser avec la notion de parallélisme

Bilan de cette première partie

A ce stade du cours, vous savez maintenant :

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

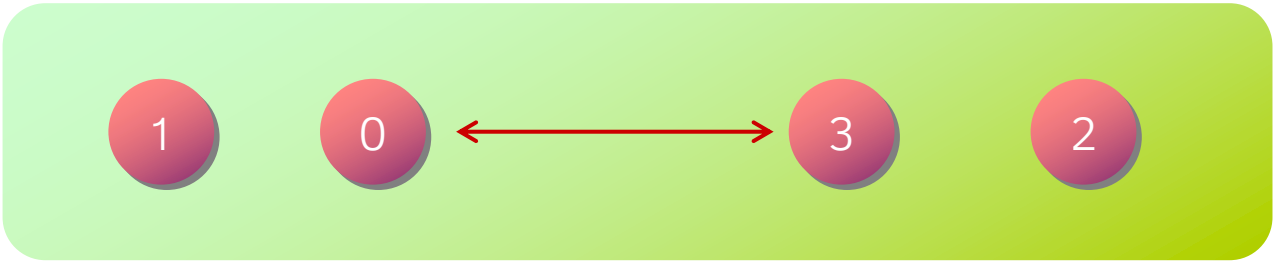


III. Introduction au parallélisme par échange de messages via MPI

2. Les communications point à point bloquantes

Notion de communication point à point

- 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



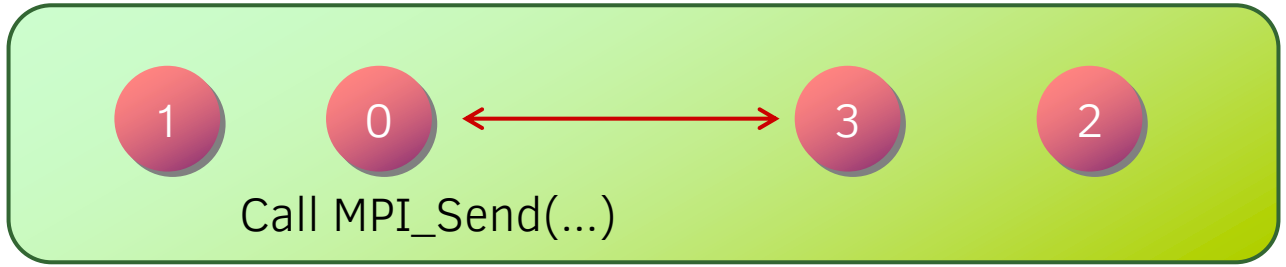
⟵ Le processus de rang 0 envoie un message au processus de rang 3

⟶ 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

- MPI_Send est la fonction appelée par le processus expéditeur

```
# Exemple de script Python
comm.send(message, destination, tag)
```



https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html?highlight=send#mpi4py.MPI.Comm.Send

Notion de communication point à point : Envoi de données via MPI_Send

- MPI_Send est la fonction appelée par le processus expéditeur

```
# Exemple de script Python
comm.send(message, destination, tag)
```

- **Message** : données que vous souhaitez envoyer à un autre processus
- **Destination `(int)`** : rang du processus destinataire
- **`Tag (int) :`** numéro attribué à la communication si plusieurs coms vers le même processus

https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html?highlight=send#mpi4py.MPI.Comm.Send

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

- Envoi d’un message de type double au processus de rang 3 par le processus de rang 2

```
message = 1234.56
tag = 0
destination = 3

if (rank == 2):
    comm.send(message, destination, tag)
```

Exemple d’utilisation de MPI_Send

- Envoi d’un message de type liste d’entiers au processus de rang 5 par le processus de rang 1

```
message = [12,45,67,34,54]
tag = 0
destination = 5

if (rank == 1):
    comm.send(message, destination, tag)
```

Exemple d’utilisation de MPI_Send

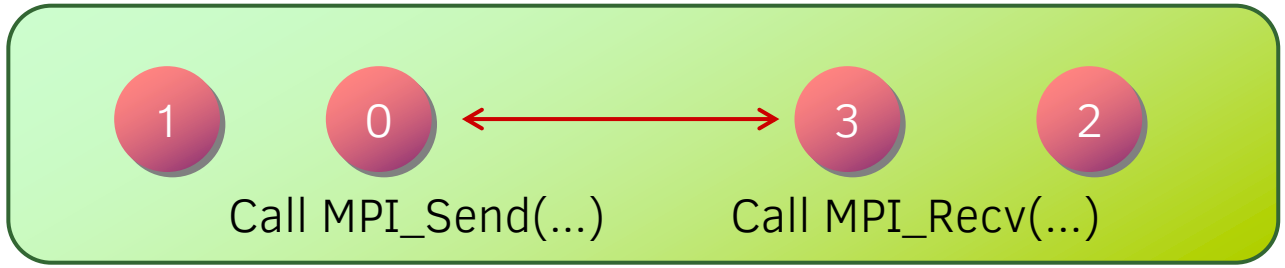
- Envoi d’un message de type dictionnaire au processus de rang 6 par le processus de rang 1

```
message = {‘a’:1, ‘b’:2}
tag = 0
destination = 6

if (rank == 1):
    comm.send(message, destination, tag)
```

Notion de communication point à point : Réception de données via MPI_Recv

- MPI_Recv est la fonction appelée par le processus destinataire

```
message = comm.recv(source, tag=tag)
```



https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Message.html?highlight=recv#mpi4py.MPI.Message.Recv

Notion de communication point à point : Réception de données via MPI_Recv

.MPI_Recv est la fonction appelée par le processus destinataire

```
message = comm.recv(source, tag)
```

- **Message** : la variable contenant le message à recevoir
- **source** : rang du processus expéditeur
- **`Tag (int) :`** numéro attribué à la communication si plusieurs coms vers le même processus

https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Message.html?highlight=recv#mpi4py.MPI.Message.Recv

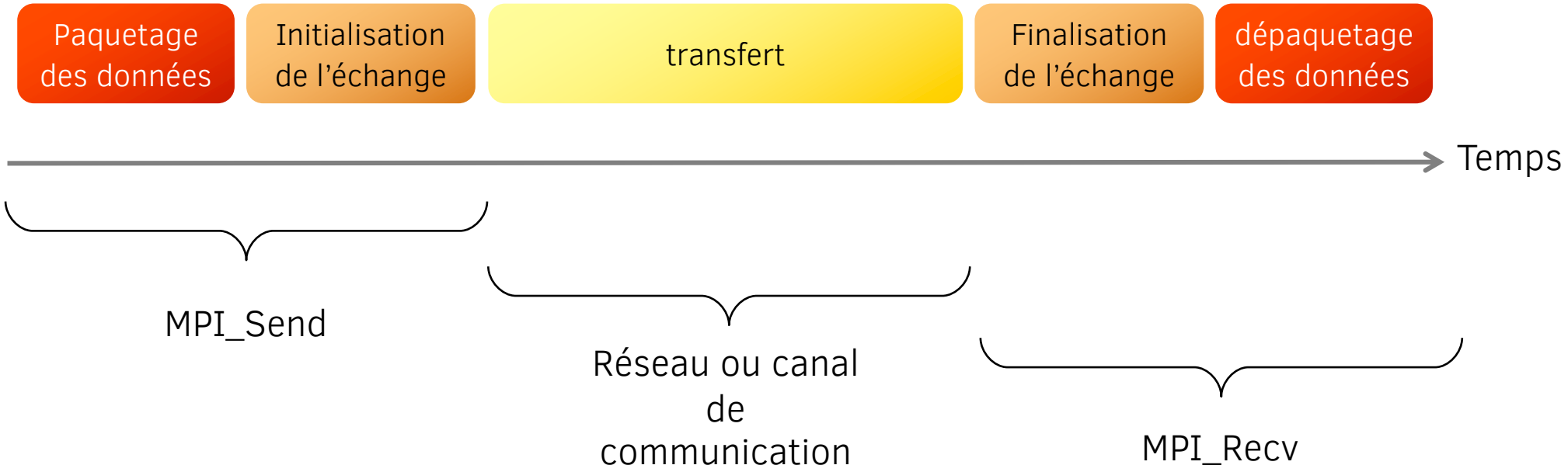
Exemple d'utilisation de MPI_Send and MPI_Recv

- 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

```
Tag = 0
Destination = 3

if (rank == 2):
     message = 1234.56
     comm.send(message, destination, tag)

if (rank == 3):
     message = comm.recv(source, tag)
```

Mieux comprendre une communication

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



Exercice 3 : communication bloquante

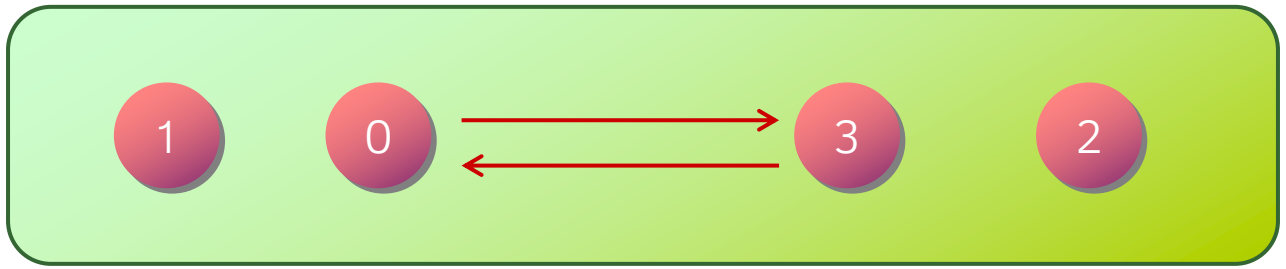
- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/3_blocking_com.md
```

- Cet exercice a pour but de tester les communications bloquantes

Communication point à point : MPI_Sendrecv

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

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

```
recv_message = comm.sendrecv(send_message, destination, sendtag, source, recvtag,
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://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html?highlight=sendrecv#mpi4py.MPI.Comm.Sendrecv

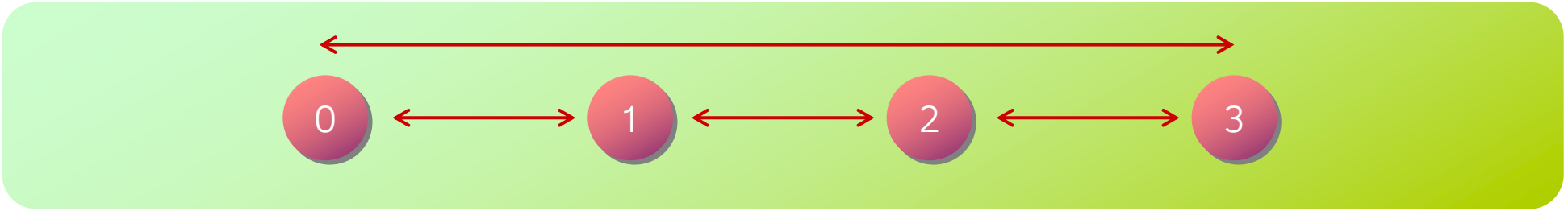
Exemple d'utilisation de MPI_Sendrecv

- Le rang 2 envoie un message au rang 3 et reçoit un message du rang 1

```
tag = 0
send_rank = 3
recv_rank = 1
send_message = 21.78

if rank == 2:
    recv_message = comm.sendrecv(send_message, send_rank, tag, recv_rank, tag, status = None)
```

Communication point à point : MPI_SENDRECV pour les communications chaînées

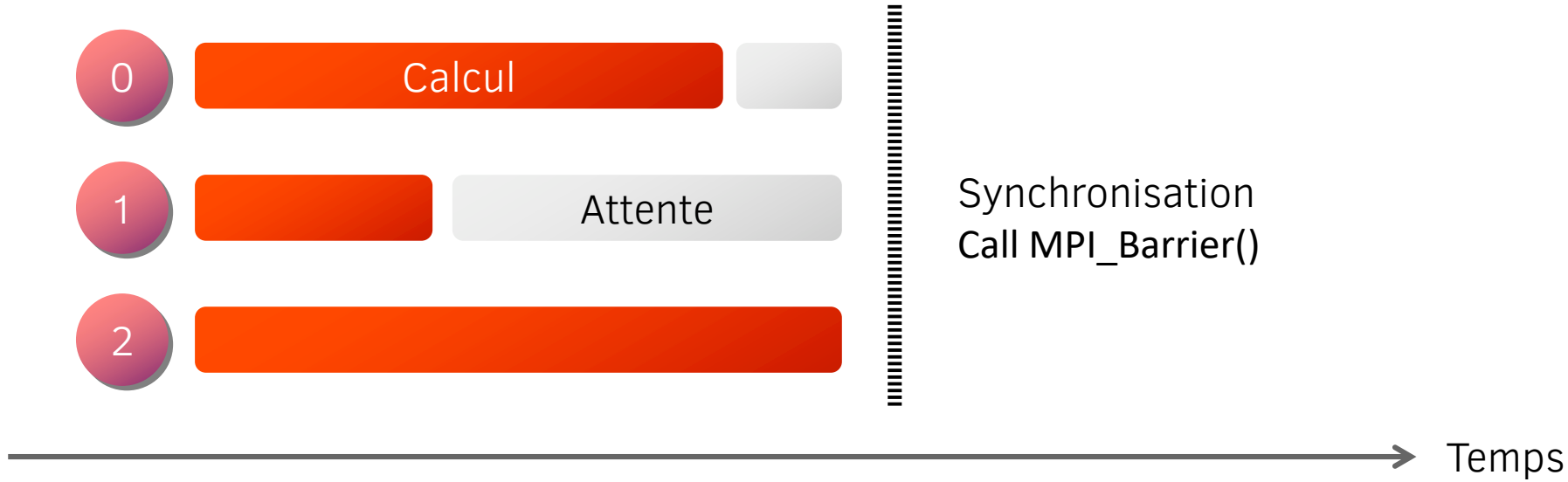
- La fonction **MPI_Sendrecv** est également nécessaire pour effectuer des communications chaînées
- L'utilisation de **MPI_Send** et **MPI_Recv** nécessiterait de gérer manuellement les synchronisations



- 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

- 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



Notion de barrière explicite

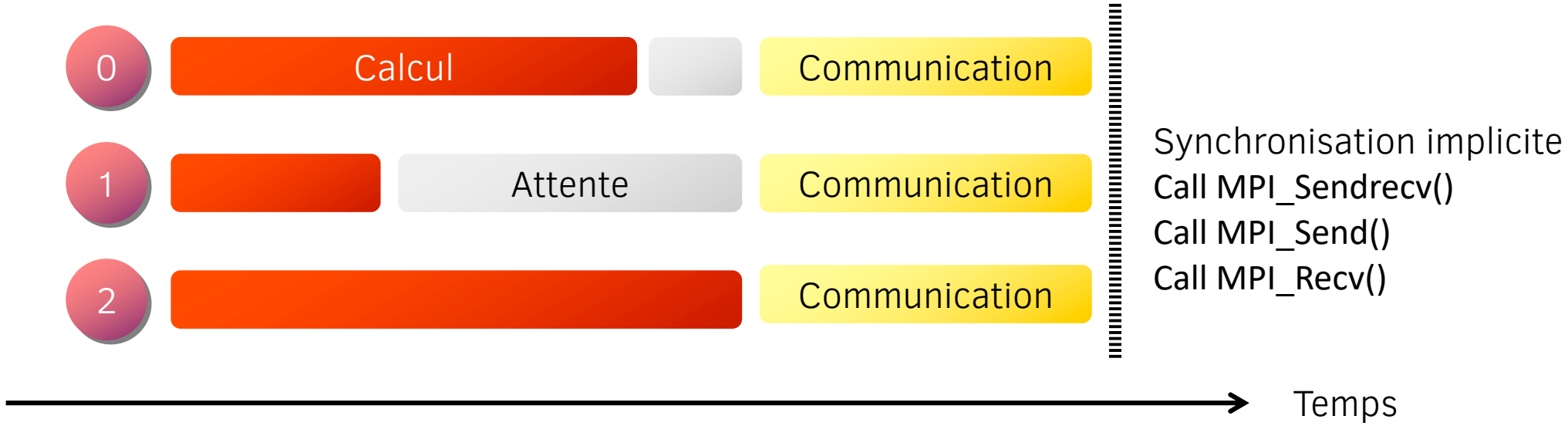
- Il est parfois nécessaire d'imposer un une étape de synchronisation ou barrière qui ne sera pas franchie tant que tous les processus ne seront pas arrivés à ce niveau
- La fonction MPI_Barrier est une manière explicite d'exiger cette synchronisation dans le code

```
comm.Barrier()
```

https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html?highlight=Barrier#mpi4py.MPI.Comm.barrier

Notion de barrière implicite

- 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 4 : communication en anneau

- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/4_sendrecv.md
```

- Cet exercice a pour but de tester les communications sendrecv au travers d'un anneau de communication

Vos premiers échanges MPI

A ce stade du cours, vous savez maintenant :

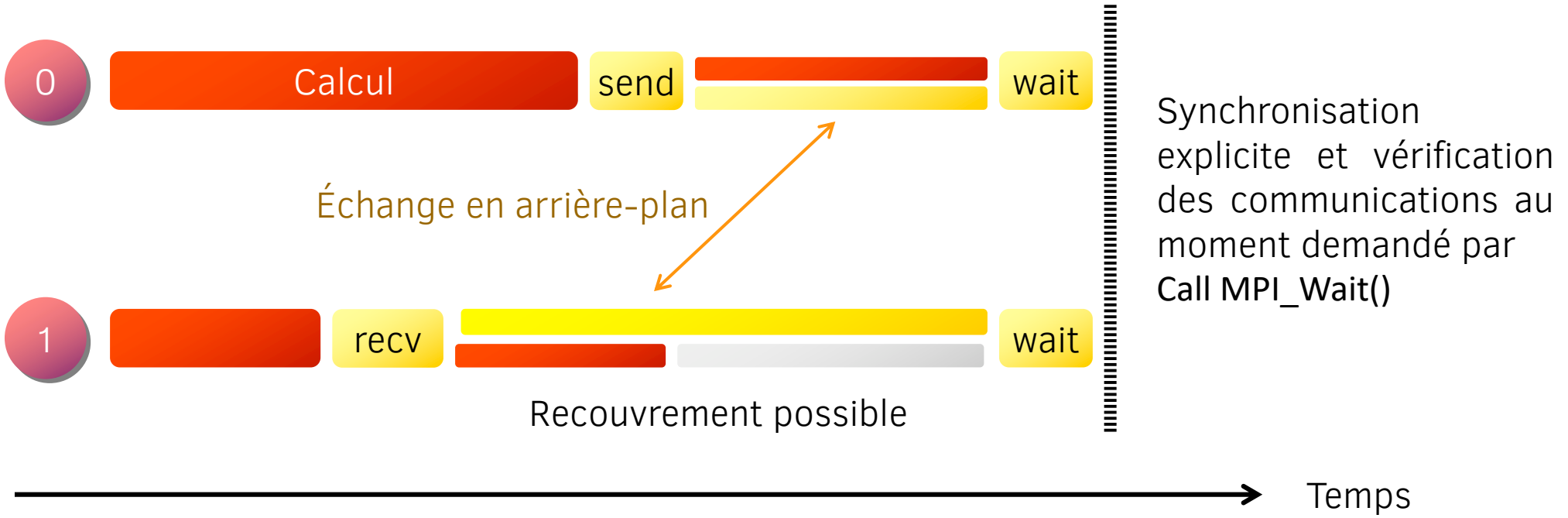
- Faire communiquer différents processus entre eux
- Gérer des chaînes de communication
- Demander une synchronisation explicite

III. Introduction au parallélisme par échanges de message via MPI

3. Les communications point à point non-bloquantes

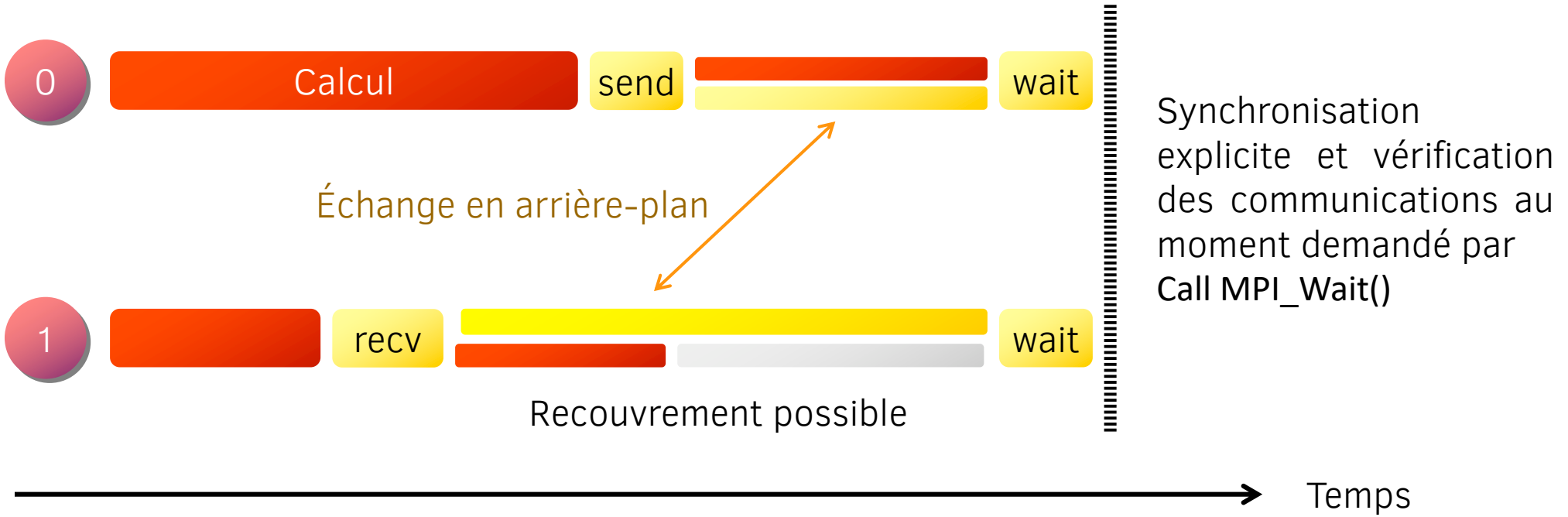
Communication point à point non-bloquante

- 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

- 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`

- MPI.Isend est la fonction appelée par le processus expéditeur

```
req = comm.isend(rank, dest, tag)
```

- MPI.Irecv est la fonction appelée par le processus receveur

```
req = comm.irecv(source, tag)
```

- 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://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html?highlight=isend#mpi4py.MPI.Comm.Isend
https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Message.html?highlight=irecv#mpi4py.MPI.Message.Irecv

La fonction `MPI_Wait`

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

.cpp

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

https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Request.html?highlight=wait#mpi4py.MPI.Request.wait

Exemple d’utilisation des communications non-bloquantes

- 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 3 venant du rang 2

```
Tag = 0

If rank == 2:
    req_send = comm.isend(message, 3, tag=tag)

If rank == 3:
    req_recv = comm.irecv(2, tag=tag)

req_send.wait()
recv_message = req_recv.wait()
```

La fonction `MPI_Waitall`

- `MPI_Waitall` effectue l'action de **`MPI_Wait`** sur un tableau de requêtes.

```
req2 = comm.Irecv(message, dest_rank)
req1 = comm.Isend(dest_message, src_rank)

MPI.Request.Waitall([req1, req2])
```

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 5 : Communications non bloquantes

- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/5_nonblocking_com.md
```

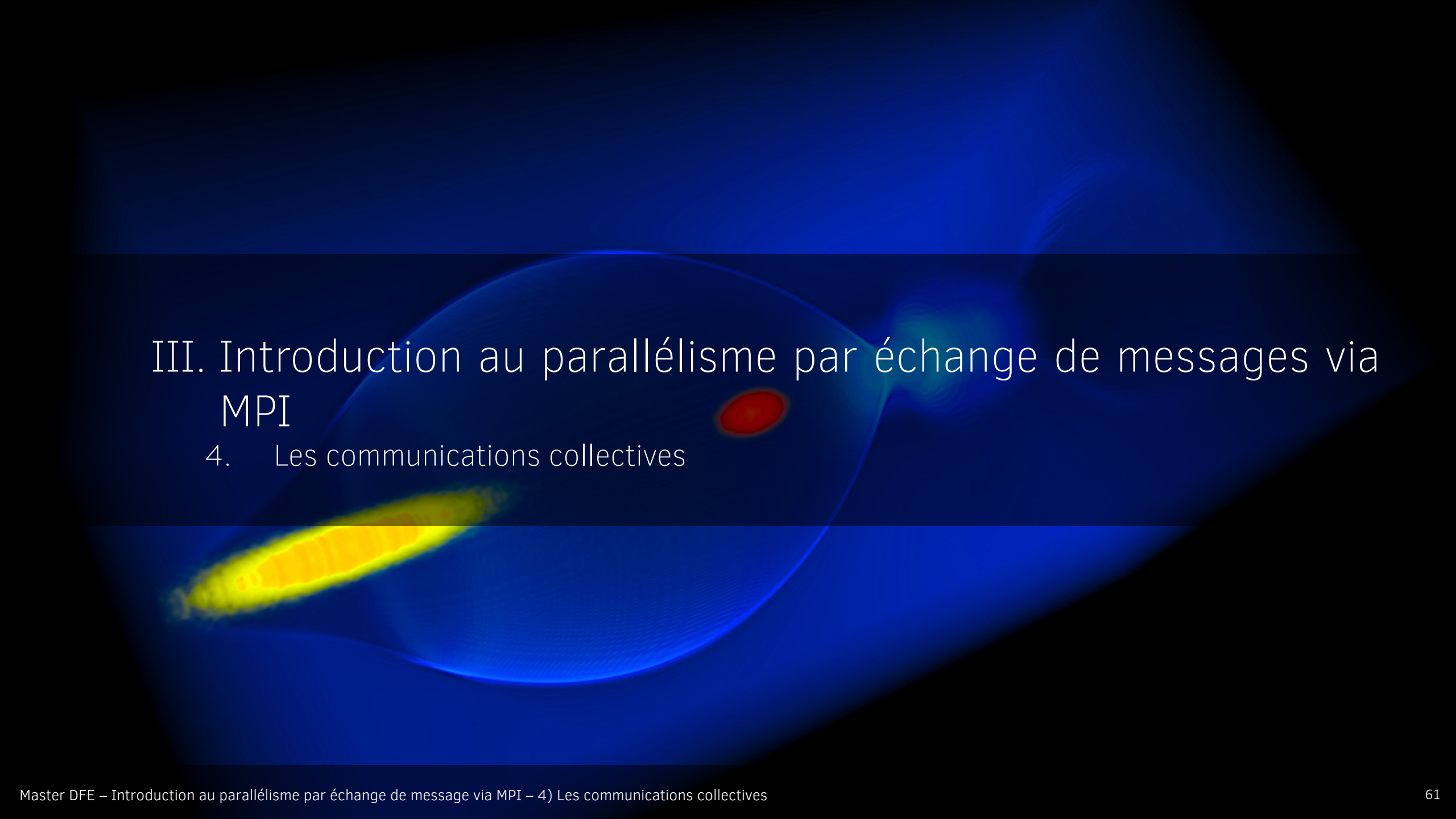
- Cet exercice a pour but de tester les communications non bloquantes

Exercice 6 : Communications de tableaux

- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/6_array_com.md
```

- Cet exercice a pour but de tester les communications avec des tableaux



III. Introduction au parallélisme par échange de messages via MPI

4. Les communications collectives

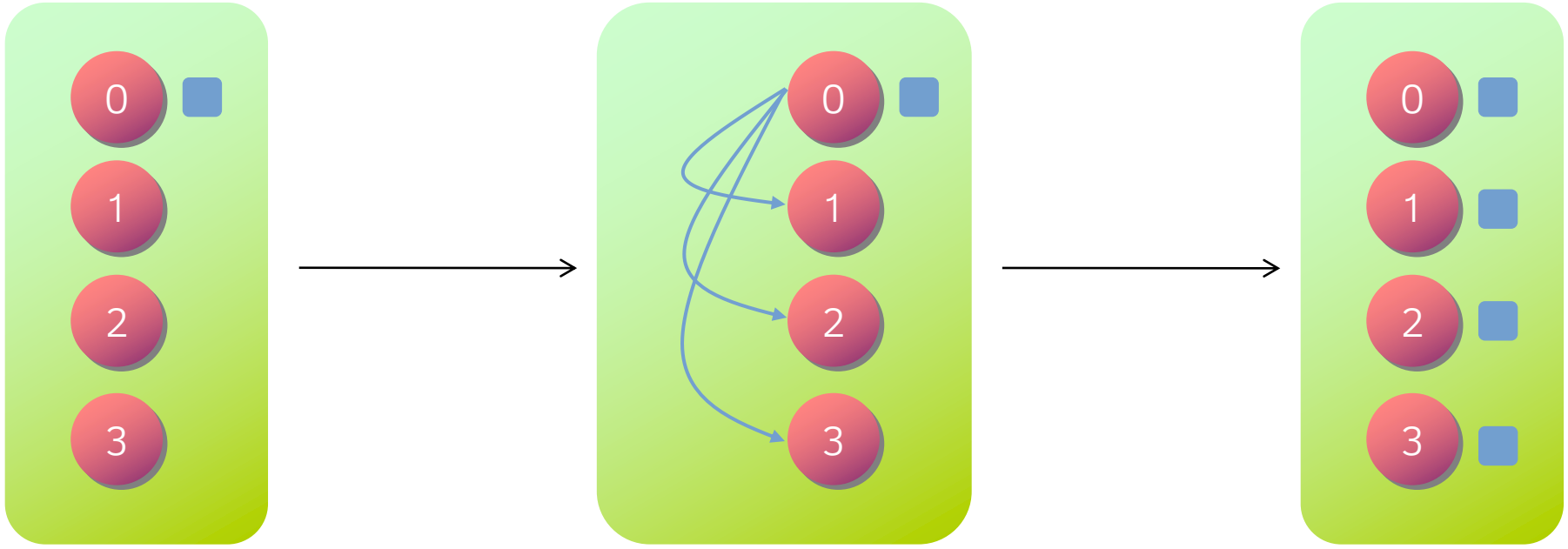
Communication collective : les différents types

- 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

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



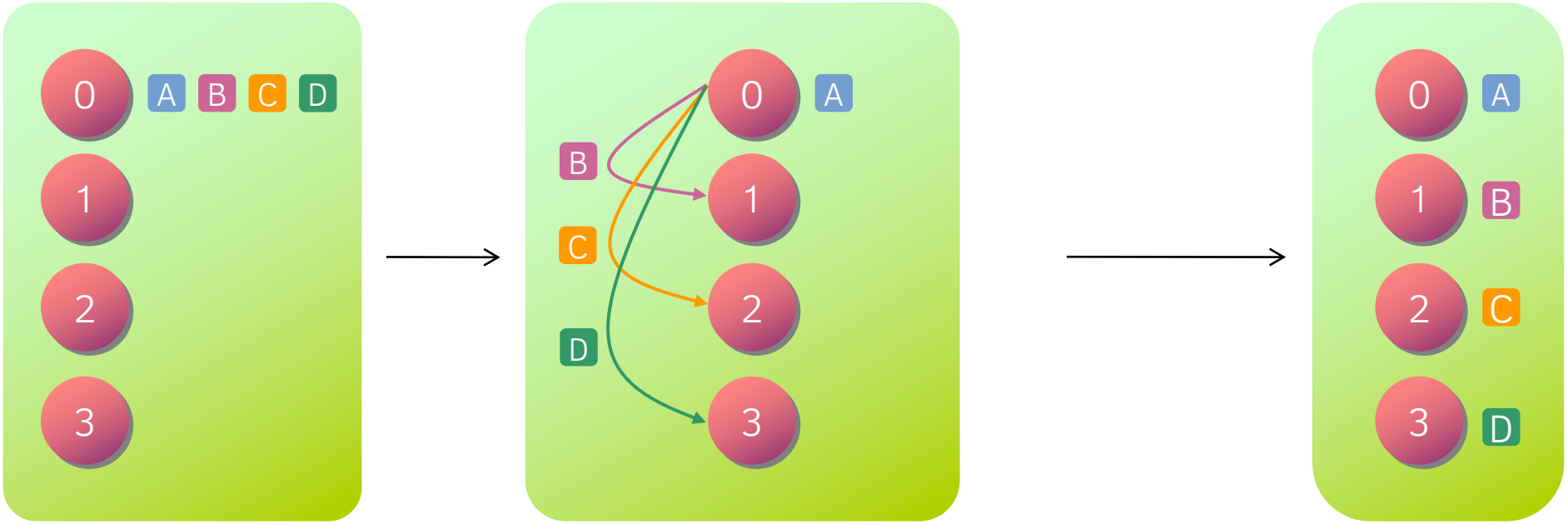
Répartition des données avant l'échange

MPI_Bcast

Nouvelle répartition des données après l'échange

Communication collective : diffusion sélective grâce à MPI_Scatter

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



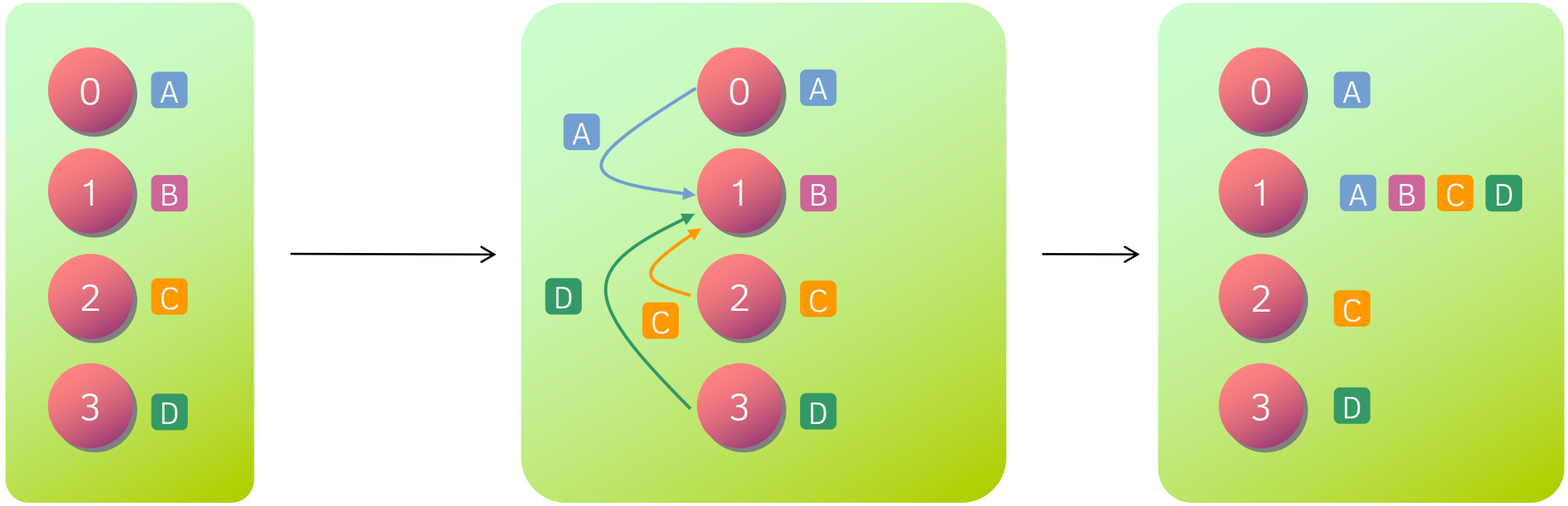
Répartition des données avant l'échange

MPI_SCATTER

Nouvelle répartition des données

Communication collective : collecte grâce à MPI_Gather

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



Répartition des données avant l'échange

MPI_GATHER

Nouvelle répartition des données

Communication collective : collecte grâce à MPI_Gather

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

```
comm.Gather(send_buffer, recv_buffer, root=0)
```

- `send_buffer` : la valeur ou un ensemble de valeurs à envoyer depuis chaque processus
- `recv_buffer` : le tableau réunissant les valeurs reçues
- `root` : le processus qui reçoit les données et reconstruit le tableau final

Exercice 7 : Utilisation de MPI Gather

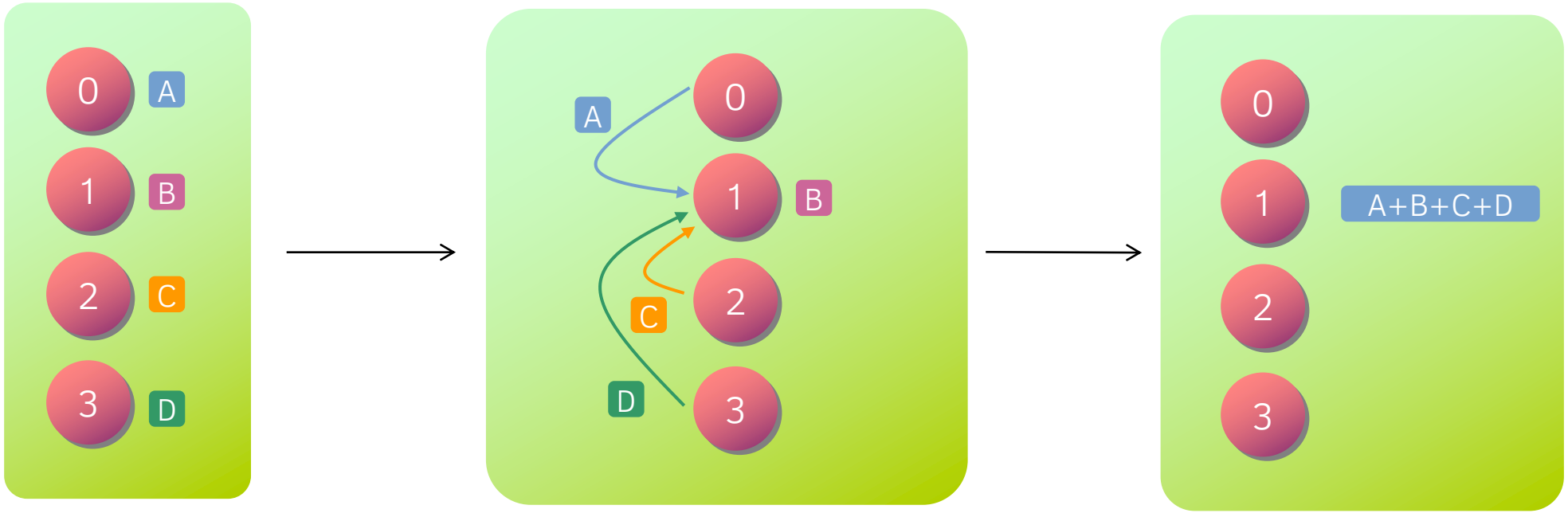
- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/7_gather_com.md
```

- Cet exercice a pour but de tester les communications collectives de typer gather

Communication collective : réduction grâce à MPI_Reduce

- 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://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html?highlight=reduce#mpi4py.MPI.Comm.reduce

Communication collective : réduction grâce à `MPI_Reduce`

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

```
reduced_buffer = comm.reduce(local_buffer, op=MPI.SUM, root=0)
```

- `Local_buffer` : la valeur à envoyer par chaque processus
- `Received_buffer` : la valeur reçue suite aux échanges et à la réduction par le destinataire seulement
- `Op (MPI_Op)` : type d'opération à effectuer pour la réduction (ex : MPI_SUM)
- `root` : le processus qui va recevoir les données réduites

Communication collective : opération de réduction

Il existe de multiples 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

...

Communication collective : exemple d'utilisation de `MPI_Reduce`

- Réduction de type somme d'une variable réelle sur le processus 0

```
local_value = 125.87 * rank

sum = comm.reduce(local_value, op=MPI.SUM, 0)
```

- Réduction de type produit d'une variable entière sur le processus 3

```
local_value = 5 * rank

prod = comm.reduce(local_value, op=MPI.PROD, 3)
```

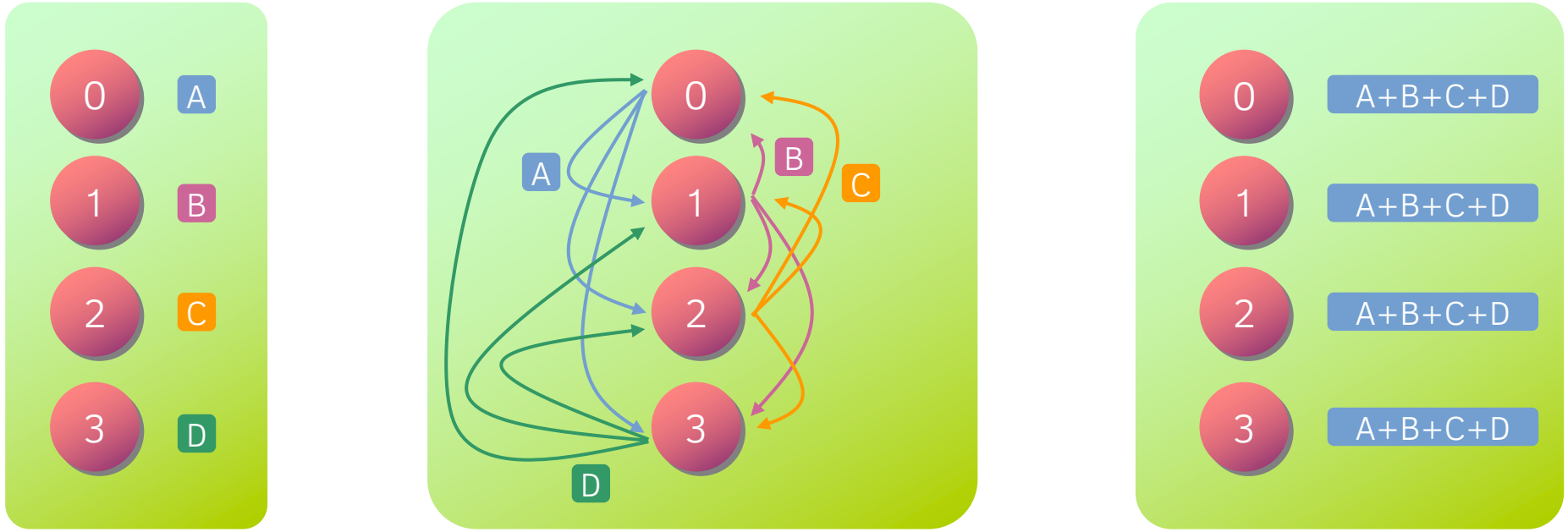
Communication collective : exemple d'utilisation de `MPI_Reduce`

- Réduction de type max d'un tableau de réels

```
# Tableau stocké sur chaque processus
local_array = np.random.rand(5)

# Pour chaque index, on détermine le max de chaque processus
max_array = comm.reduce(local_array , op=MPI.MAX, 0)

# On détermine ensuite le max de tous les index :
max = np.max(max_array)
```

Communication collective : réduction grâce à `MPI_Allreduce`

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



MPI_ALLREDUCE + MPI.SUM

Communication collective : réduction grâce à MPI_Allreduce

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

```
sum = comm.allreduce(local_buffer, op=operation)
```

- **Local_buffer** : la valeur à envoyer par chaque processus
- **sum**: la valeur reçue suite aux échanges et à la réduction par le destinataire seulement
- **Opération** : type d'opération à effectuer pour la réduction (ex : **MPI.SUM**)

https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html?highlight=allreduce#mpi4py.MPI.Comm.allreduce

Exercice 8 : Utilisation de MPI Reduce

- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/8_reduce_com.md
```

- Cet exercice a pour but de tester les communications collectives de typer MPI reduce

Communication collective pour les données de taille variable

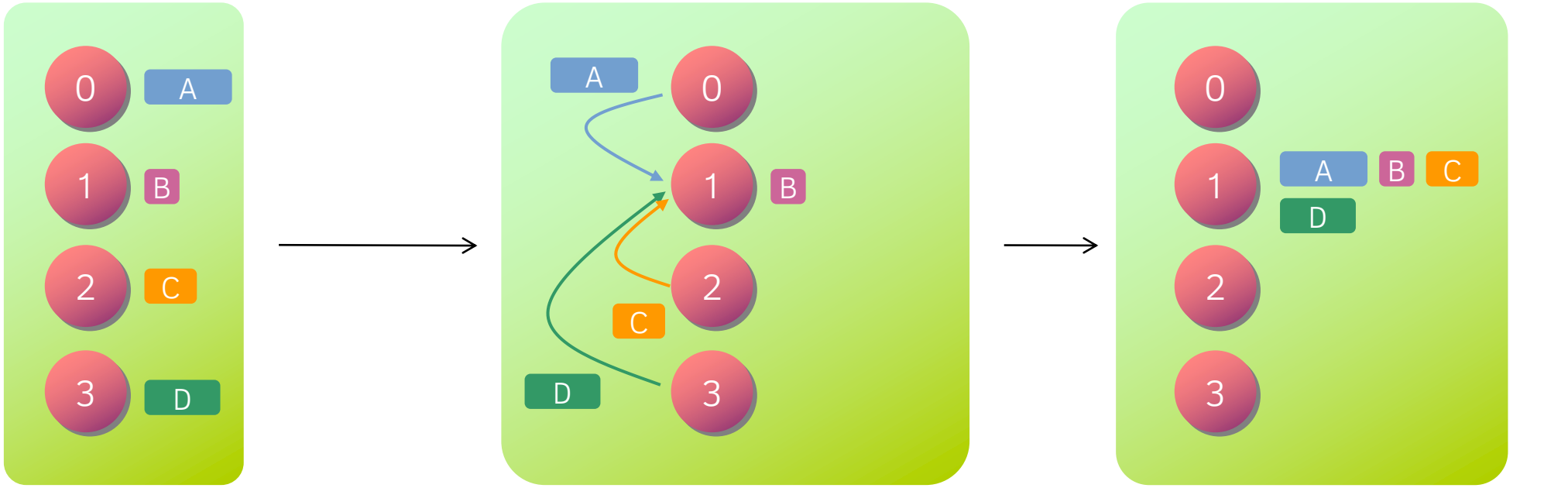
Certaines communications collectives (hors réduction) présentées précédemment imposent que chaque rang échange la même quantité de donnée. Cette limitation peut être levée en utilisant une extension des communications collectives.

Elles possèdent 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

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



MPI_Gatherv

Nouvelle répartition des données

Communication collective pour les données de tailles variables

`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`

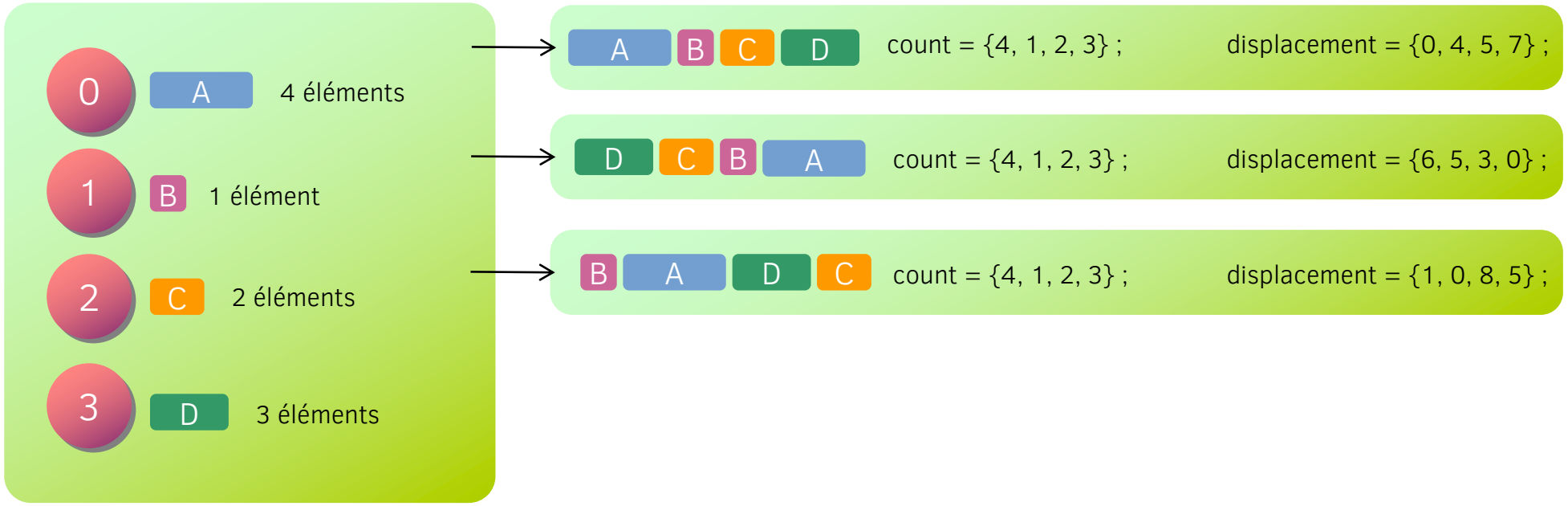
- MPI_Gatherv est appelée par les processus expéditeurs et destinataires en même temps

```
comm.Gatherv(send_buffer, [recv_buffer, count, displacement, recv_type], root=0)
```

- `send_buffer` : le tableau à envoyer par chaque processus
- `count` : liste contenant le nombre de valeurs reçues par chaque processus
- `recv_buffer` : le tableau réunissant toutes les valeurs reçues de chaque processus
- `recv_count` : nombre d'éléments reçus pour chaque rang
- `displacement` : où placer chaque contribution dans `recv_buffer`
- `recv_type (MPI_Datatype)` : le type des données reçues
- `root` : le processus qui reçoit les données

Communication collective : collecte grâce à `MPI_Gatherv`

Le tableau **`displacement`** permet de placer les contributions 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.



Fonctions supplémentaires

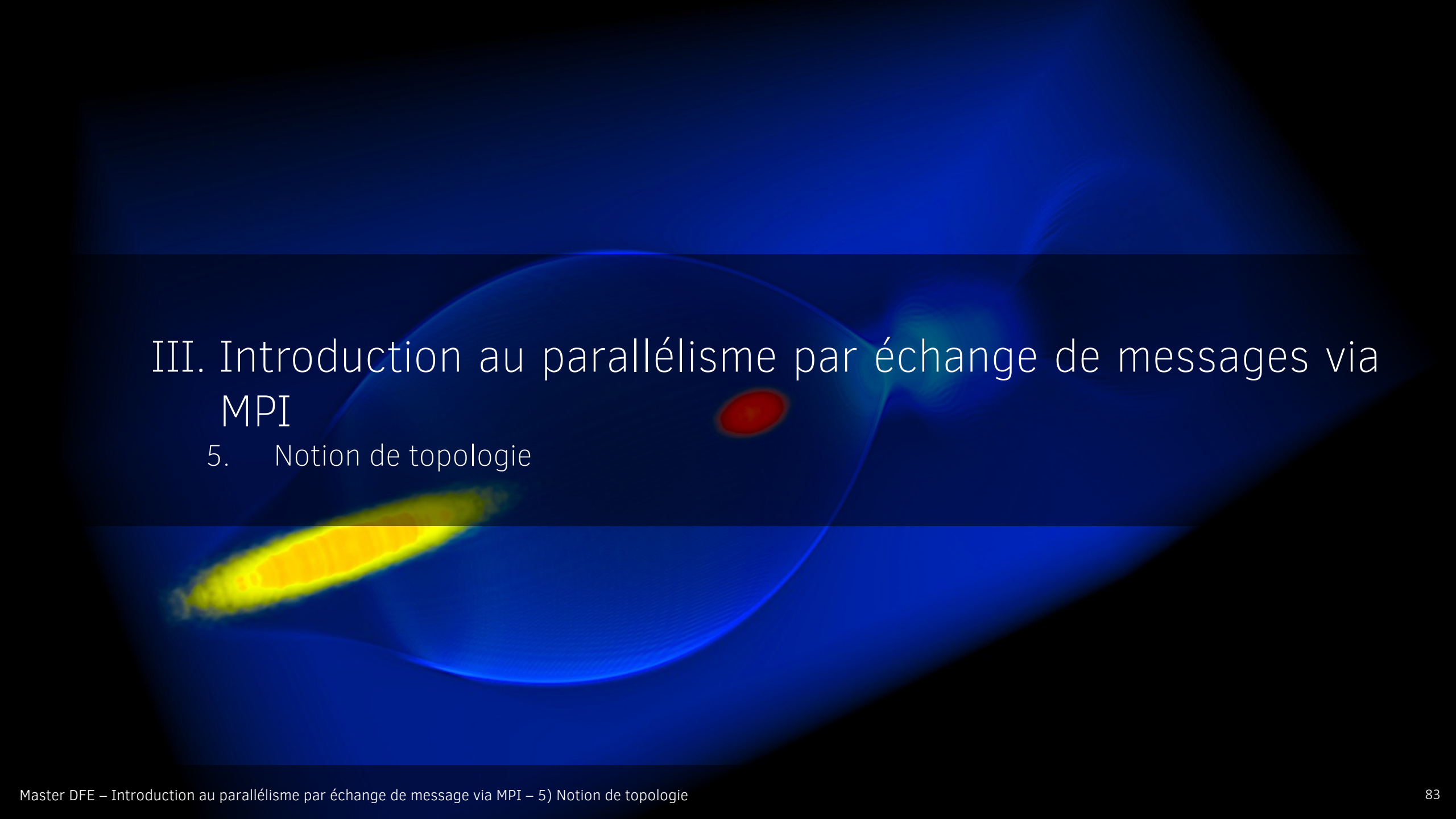
Toutes les communications collectives présentées ont également un équivalent non-bloquant :

- `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

A ce stade du cours, vous savez maintenant :

- Effectuer des communications collectives
- Effectuer des réductions



III. Introduction au parallélisme par échange de messages via MPI

5. Notion de topologie

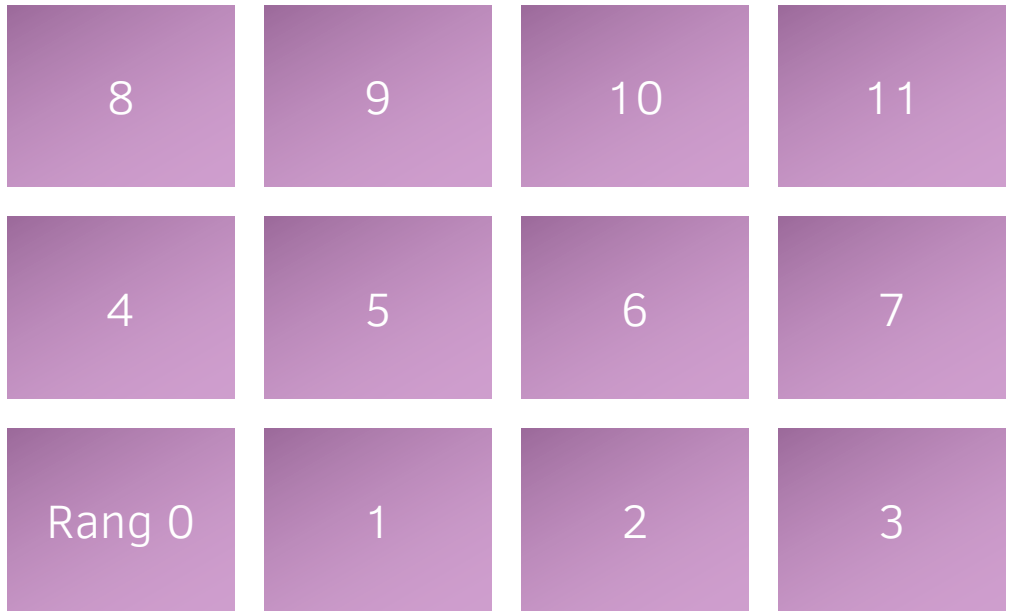
Décomposition de domaine cartésienne

- 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

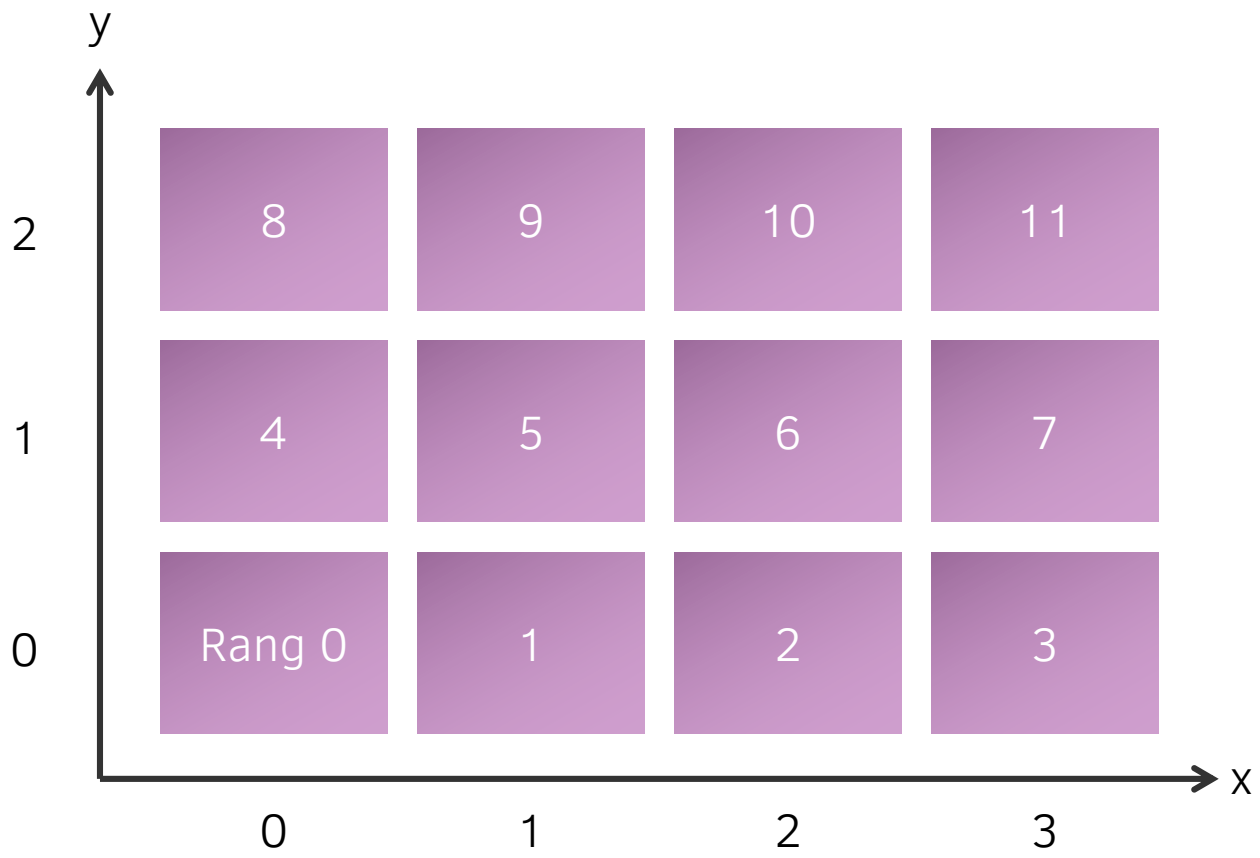


Décomposition de domaine cartésienne

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 Create_cart

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

```
cartesian_comm = comm.Create_cart(dimensions, periodicity, reorder=True)
```

- comm: communicateur à partir duquel on construit la topologie (MPI_COMM_WORLD par exemple)
- dimensions : liste contenant le nombre de processus dans chaque direction. La dimension de la liste détermine la dimension de la topologie.
- Periodicity : liste de booléens permettant de définir si chaque direction est périodique (true) ou non (false)
- Reorder : booléen permettant de réorganiser les rangs pour optimiser les échanges
- cartesian_comm : nouveau communicateur renvoyé par la fonction définissant une topologie cartésienne

Décomposition de domaine cartésienne : création via Create_cart

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 Get_coords

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

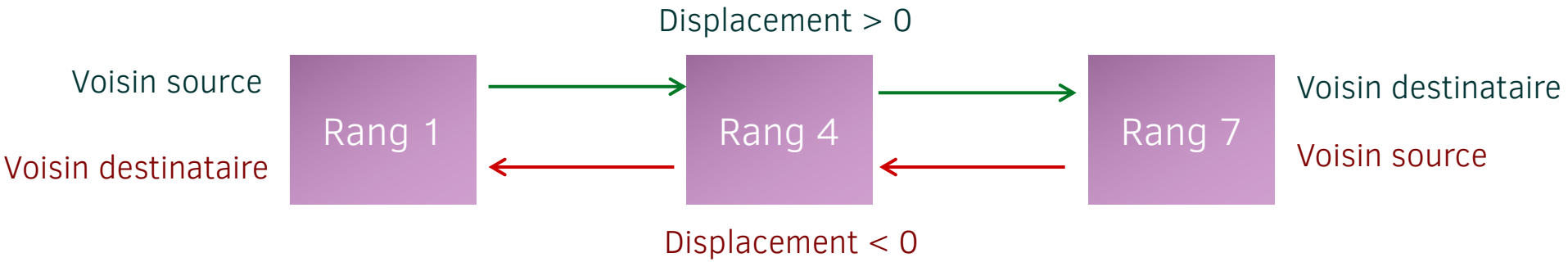
```
cart_coords = cart_comm.Get_coords(rank)
```

- **rank** : rang du processus MPI
- **cart_coords** : les coordonnées du rang **rank** dans le communicateur cartésien **cart_comm**

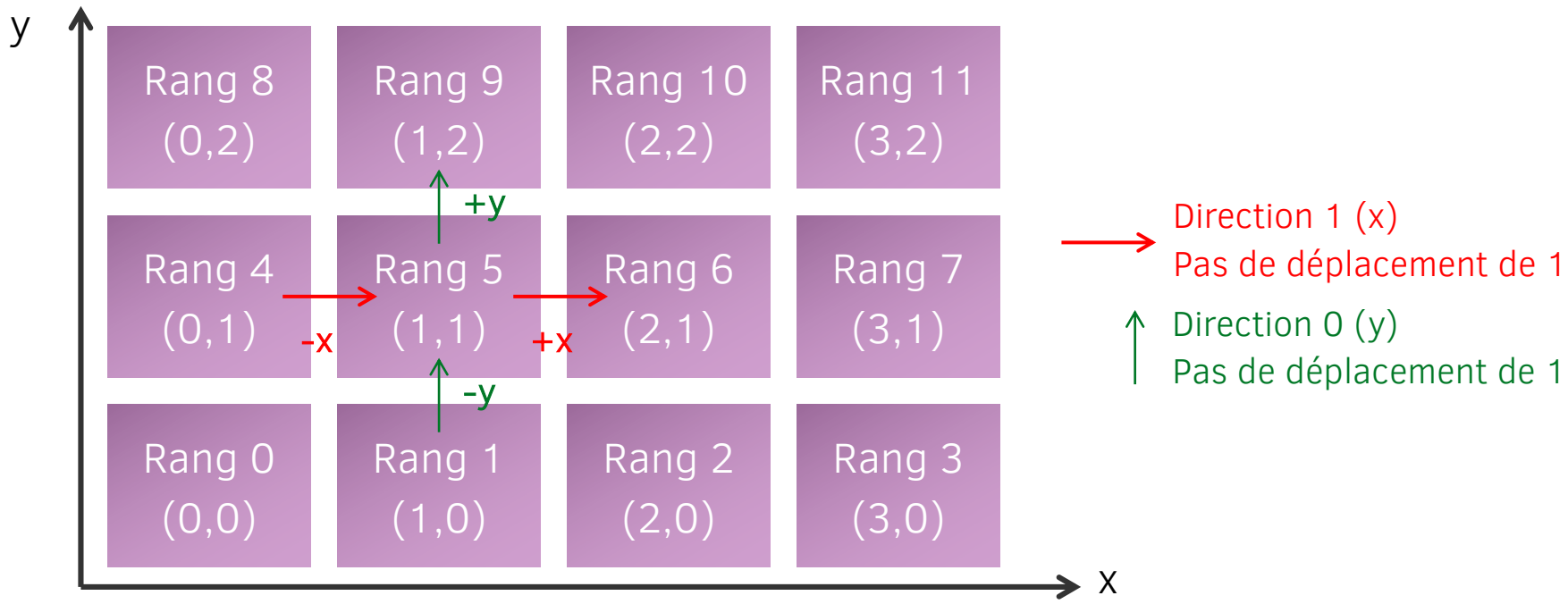
Décomposition de domaine cartésienne : les voisins

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

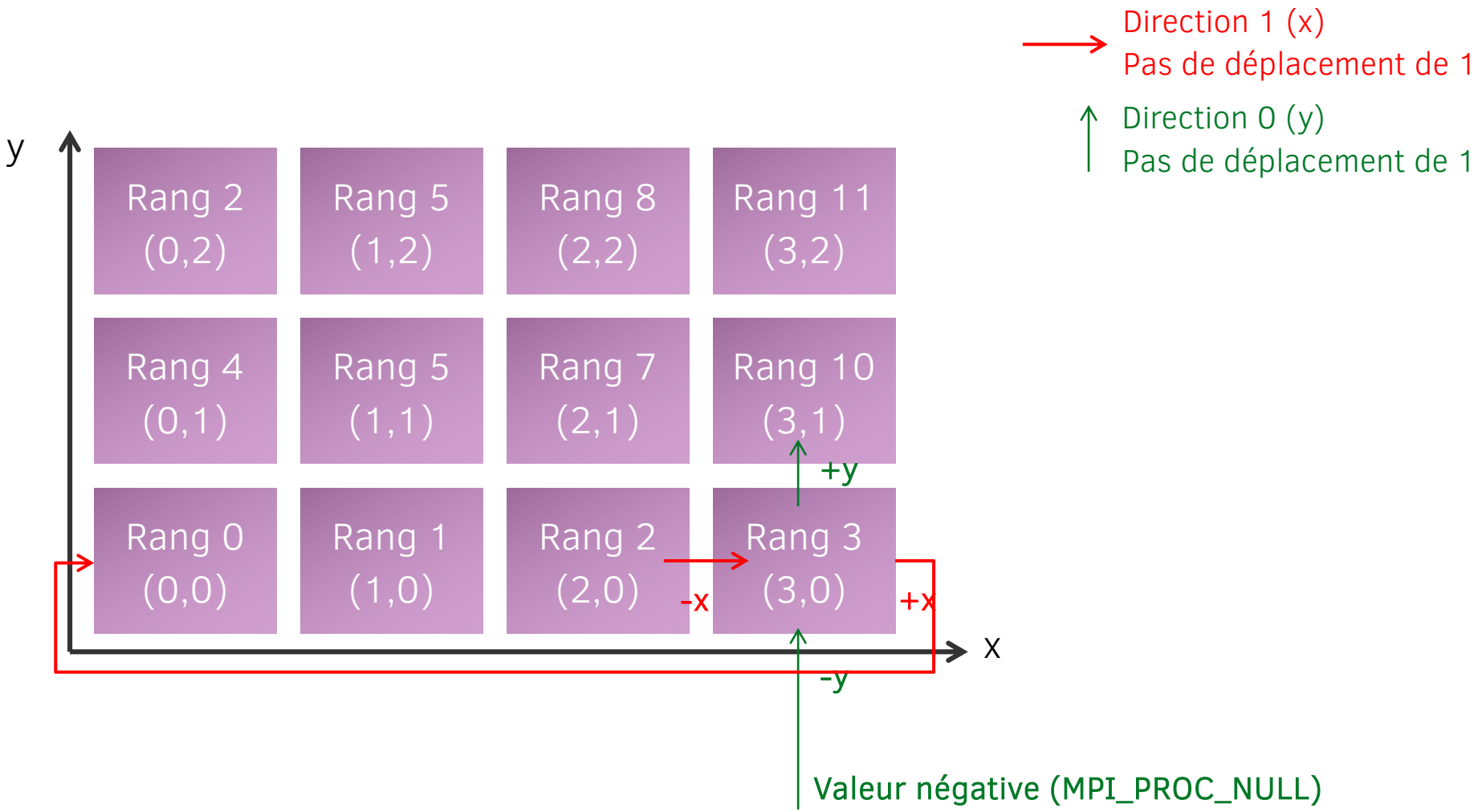
Shift permet de récupérer les rangs voisins d'un rang donné en spécifiant une direction et un sens de déplacement. On récupère 2 voisins dans la philosophie MPI_Sendrecv : un rang source et un rang destinataire.



Exemple de topologie cartésienne 2D



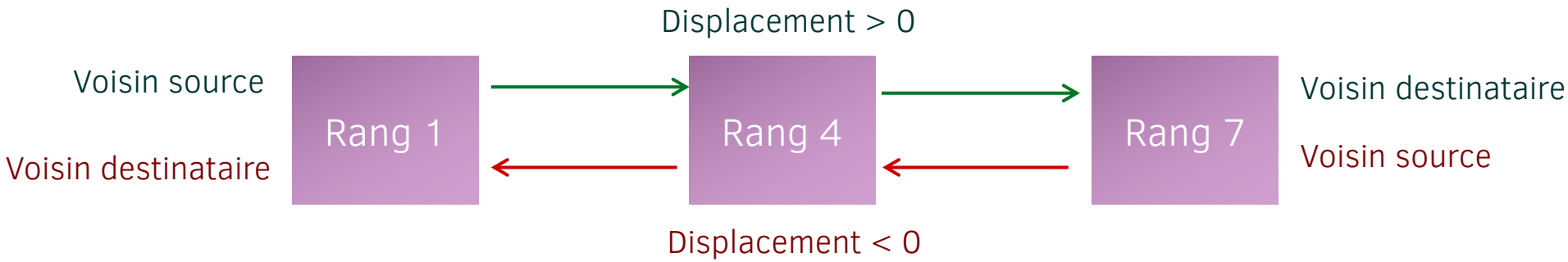
Exemple de topologie cartésienne 2D : notion de périodicité



Shift permet de récupérer les rangs voisins d'un rang donné

```
[left_rank, right_rank] = cart_comm.Shift(direction, displacement)
```

- **direction** (int) : 0 pour la première coordonnée, 1 pour la deuxième coordonnée, etc
- **displacement** (int) : pas de déplacement dans la direction souhaitée, si > 0 déplacement vers les coordonnées supérieures, si < 0 vers les coordonnées inférieures
- **rank_left** : si displacement > 0, correspond au voisin de gauche
- **rank_right** : si displacement > 0, correspond au voisin de droite

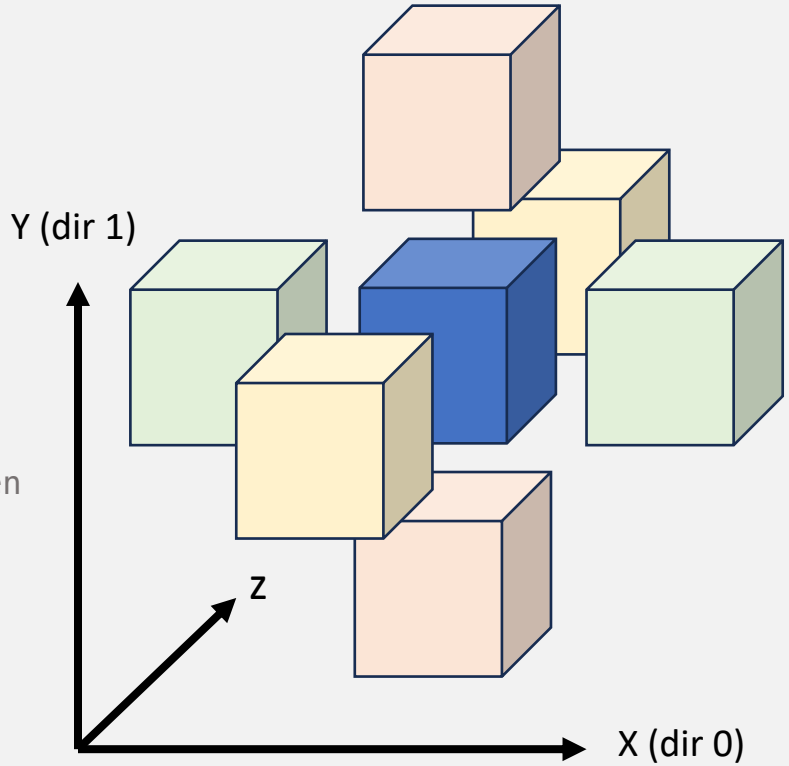


Décomposition de domaine cartésienne : récupérer les rangs des voisins via Shift

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

Exemple de topologie cartésienne 3D

```
# On définit le nombre de dimensions
ndims = 3

# On définit le nombre de processus par dimension
dims = [4, 5, 2]

# On définit si les dimensions sont périodiques ou non
periods = [False, False, True]

# On crée le communicateur cartésien
cart_comm = comm.Create_cart(dims, periods, reorder=True)

# On récupère le rang dans le communicateur cartésien
cart_rank = cart_comm.Get_rank()

# On récupère les coordonnées du rang dans le communicateur cartésien
cart_coords = cart_comm.Get_coords(cart_rank)

# On récupère le rang du processus voisin dans la dimension 0
# avec un décalage de +1
left_rank = cart_comm.Shift(0, 1)[0]
right_rank = cart_comm.Shift(0, 1)[1]
```



Exercice 9 : Utilisation de la topologie cartésienne

- Page de l'exercice :
- Vous pouvez aussi y accéder via votre éditeur en ouvrant le fichier suivant :

```
> cd exercices/mpi/python/9_cartesian_topology.md
```

- Cet exercice a pour but de tester les communications collectives de typer MPI reduce

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