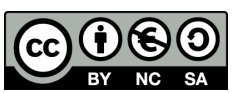
IDRIS

MPI

Dimitri Lecas - Rémi Lacroix - Serge Van Criekingen - Myriam Peyrounette

CNRS — IDRIS

v5.5 19 juin 2026



Plan I

Plan II

Plan III

Plan IV

Introduction

Introduction

A propos

Ce document est mis à jour régulièrement. La version la plus récente est disponible sur le site Web de l'IDRIS : http://www.idris.fr/formations/mpi/

- IDRIS
  Institut du développement et des ressources en informatique scientifique
  Rue John Von Neumann
  Bâtiment 506
  BP 167
  91403 ORSAY CEDEX
  France
  http://www.idris.fr

Introduction

Parallélisme

L'intérêt de faire de la programmation parallèle est :

- De réduire le temps de restitution ;
- D'effectuer de plus gros calculs ;
- D'exploiter le parallélisme des processeurs modernes (multi-coeurs, multithreading).

Mais pour travailler à plusieurs, la coordination est nécessaire. MPI est une bibliothèque permettant de coordonner des processus en utilisant le paradigme de l'échange de messages.

Modèle de programmation séquentiel

- le programme est exécuté par un et un seul processus ;
- toutes les variables et constantes du programme sont allouées dans la mémoire allouée au processus ;
- un processus s'exécute sur un processeur physique de la machine.

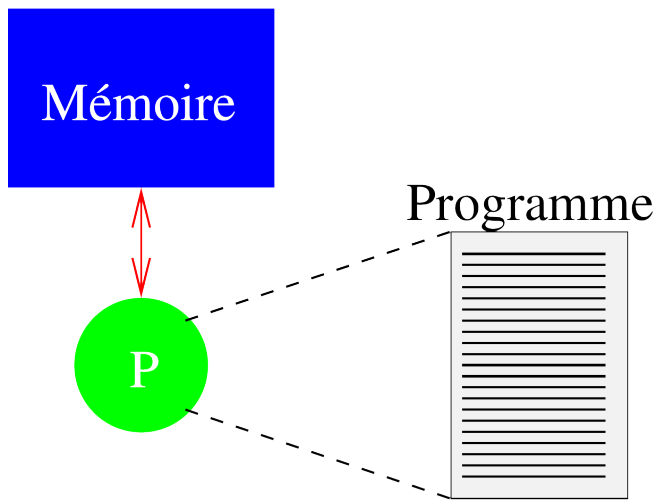


Figure 1 – Modèle de programmation séquentiel

Introduction

Modèle de programmation par échange de messages

- le programme est écrit dans un langage classique (Fortran, C, C++, etc.) ;
- toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus ;
- chaque processus exécute éventuellement des parties différentes d'un programme ;
- une donnée est échangée entre deux ou plusieurs processus via un appel, dans le programme, à des sous-programmes particuliers.

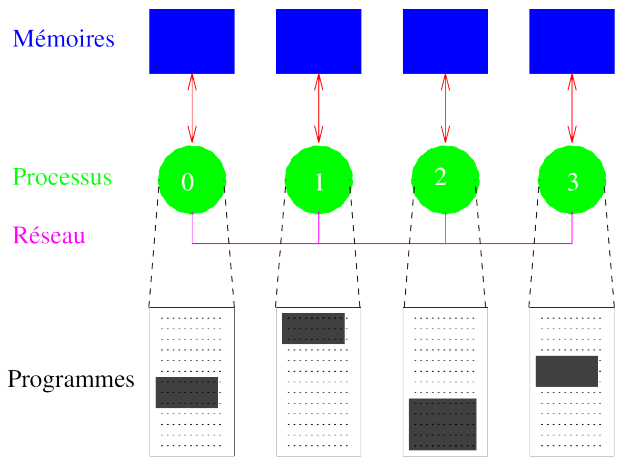


Figure 2 – Modèle de programmation par échange de messages

Concepts de l'échange de messages

Si un message est envoyé à un processus, celui-ci doit ensuite le recevoir

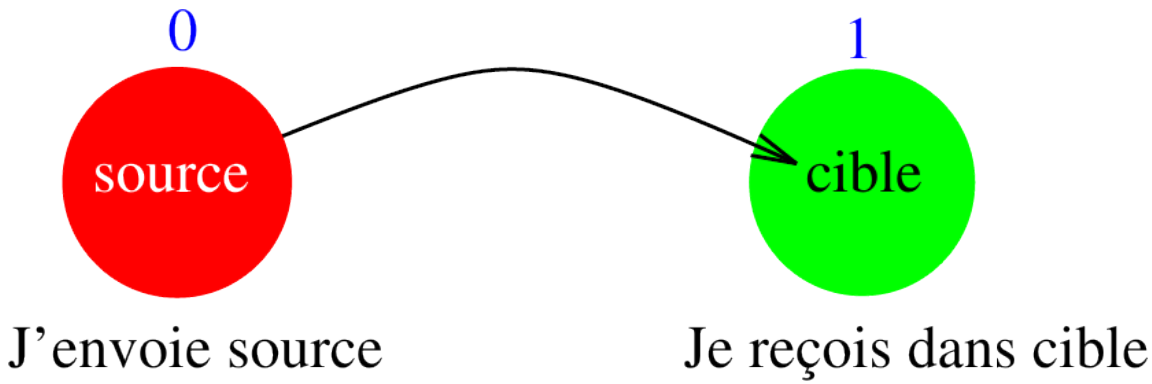


Figure 3 – échange d'un message

Introduction

Constitution d'un message

- Un message est constitué de paquets de données transitant du processus émetteur au(x) processus récepteur(s)
- En plus des données (variables scalaires, tableaux, etc.) à transmettre, un message doit contenir les informations suivantes :
  - l'identificateur du processus émetteur ;
  - le type de la donnée ;
  - sa longueur ;
  - l'identificateur du processus récepteur.

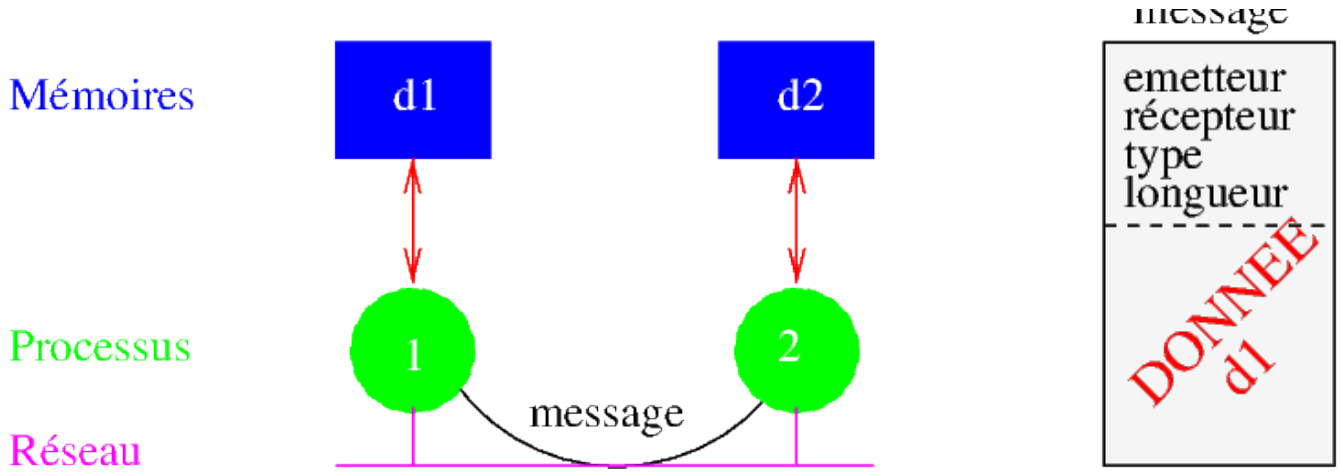


Figure 4 – Constitution d'un message

Introduction

Environnement

- Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie, au courrier postal, à la messagerie électronique, etc.
- Le message est envoyé à une adresse déterminée
- Le processus récepteur doit pouvoir classer et interpréter les messages qui lui ont été adressés
- L'environnement en question est MPI (Message Passing Interface). Une application MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI.

Architecture des supercalculateurs

La plupart des supercalculateurs sont des machines à mémoire distribuée. Ils sont composés d'un ensemble de nœud, à l'intérieur d'un nœud la mémoire est partagée.

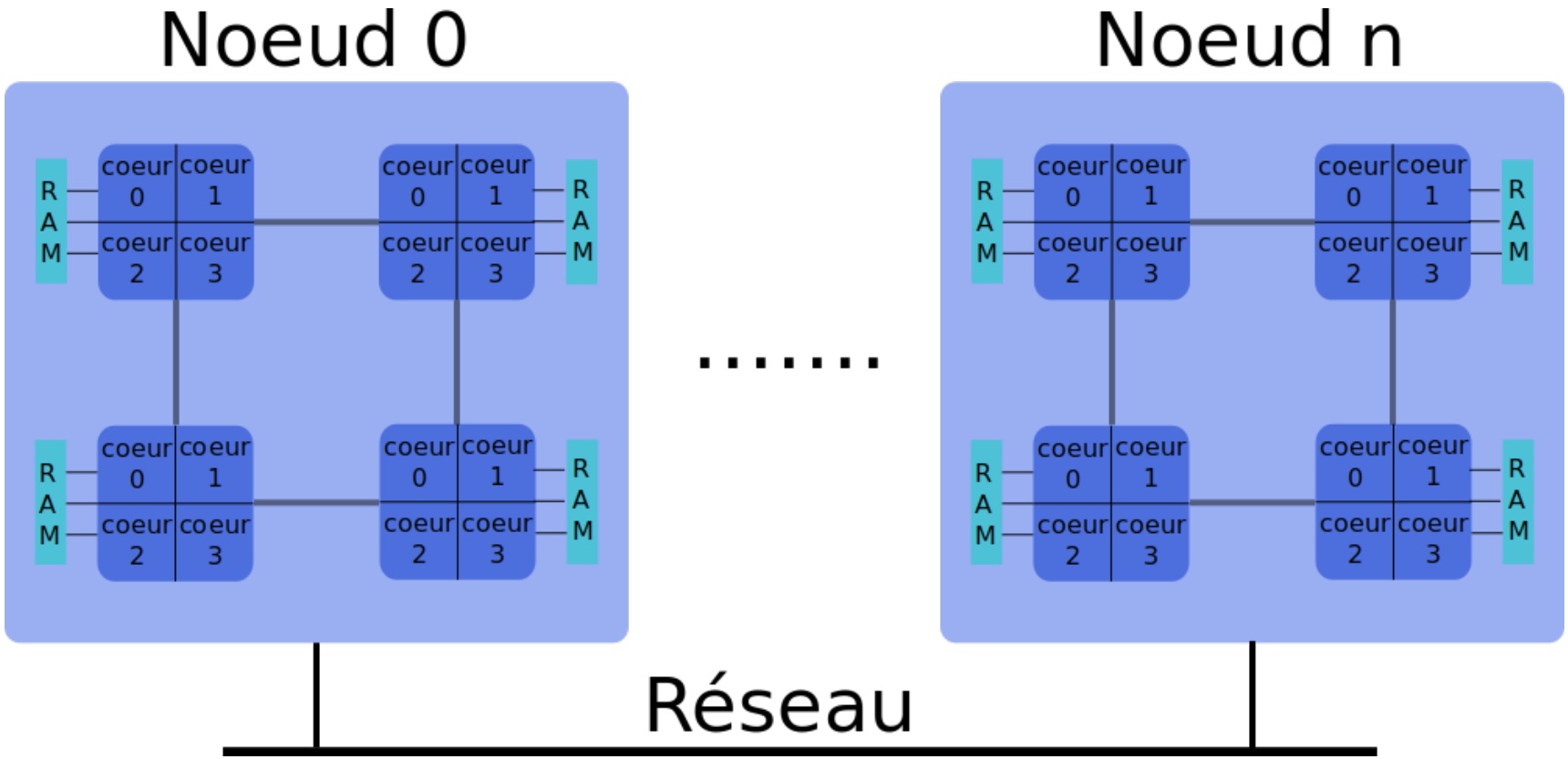


Figure 5 – Architecture des supercalculateurs

Introduction

Jean Zay

720 nœuds

- 2 processeurs Intel Cascade Lake (20 cœurs) à 2,5 Ghz par nœud
- 192 Go par nœud

396 nœuds

- 2 processeurs Intel Cascade Lake (20 cœurs) à 2,5 Ghz par nœud
- 192 Go par nœud
- 126 nœuds avec 4 GPU Nvidia Tesla V100 SXM2 16 Go
- 270 nœuds avec 4 GPU Nvidia Tesla V100 SXM2 32 Go

31 nœuds

- 2 processeurs Intel Cascade Lake 6226 (12 cœurs à 2,7 GHz), soit 24 cœurs par nœud
- 20 nœuds à 384 Go de mémoire
- 11 nœuds à 768 Go de mémoire
- 8 GPU Nvidia Tesla V100 SXM2 32Go

Jean Zay

52 nœuds

- 2 processeurs AMD Milan EPYC 7543 (32 cœurs à 2,80 GHz), soit 64 cœurs par nœud
- 512 Go de mémoire par nœud
- 8 GPU Nvidia A100 SXM4 80 Go

364 nœuds

- 2 processeurs Intel Xeon Platinum 8468 (48 cœurs à 2,10 GHz), soit 96 cœurs par nœud
- 512 Go de mémoire par nœud
- 4 GPU Nvidia H100 SXM5 80 Go

MPI vs OpenMP

OpenMP utilise un schéma à mémoire partagée, tandis que pour MPI la mémoire est distribuée.

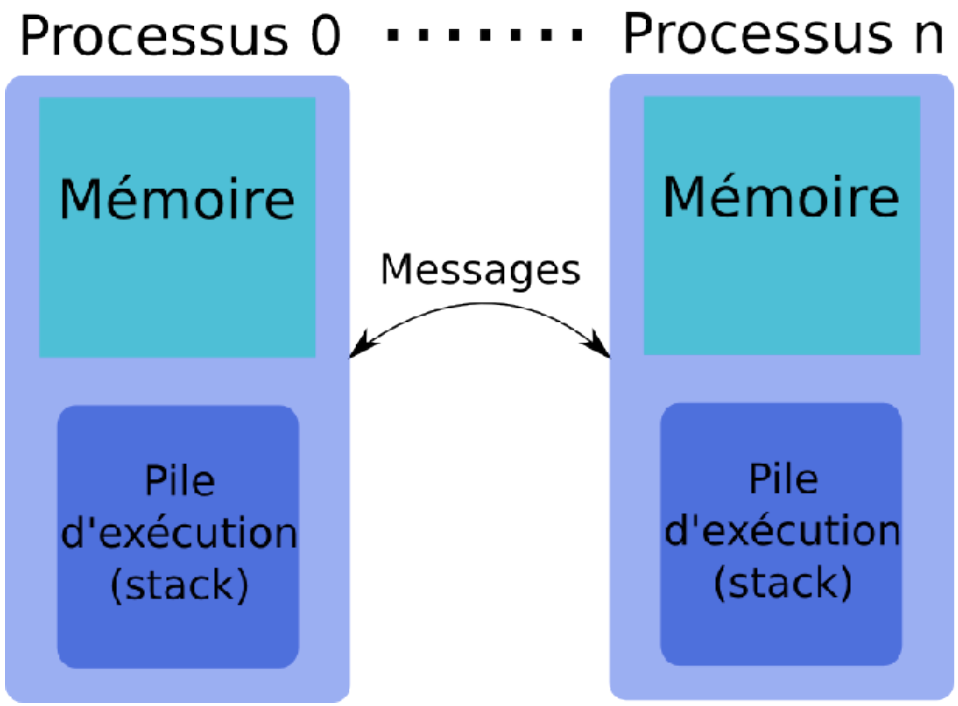


Figure 6 – Schéma MPI

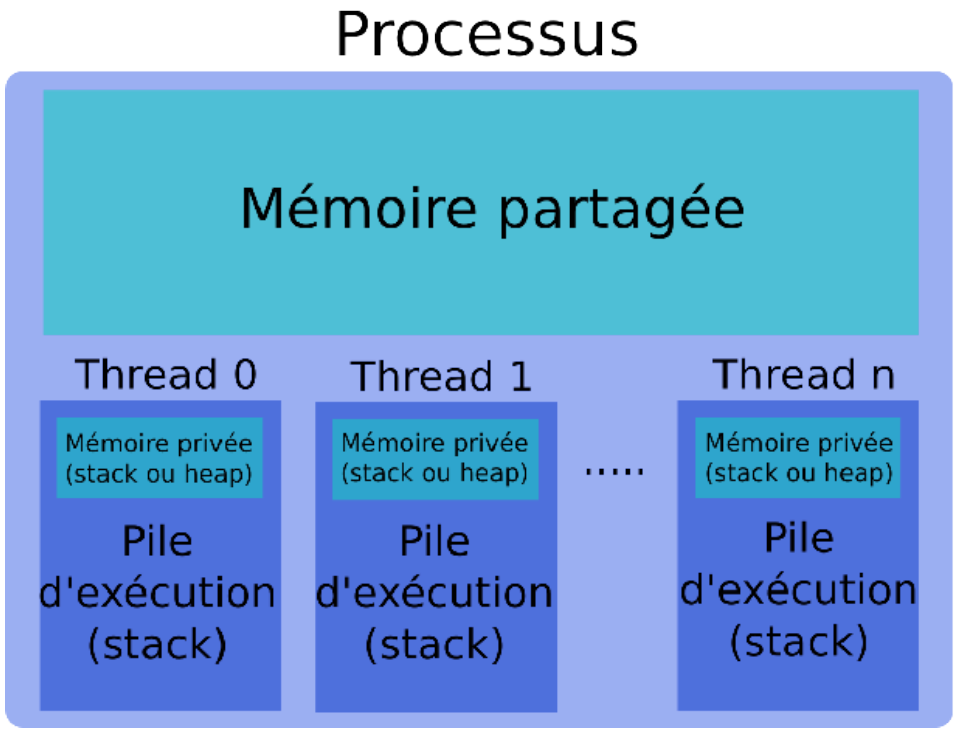


Figure 7 – Schéma OpenMP

Introduction

Décomposition de domaine

Un schéma que l'on rencontre très souvent avec MPI est la décomposition de domaine. Chaque processus possède une partie du domaine global, et effectue principalement des échanges avec ses processus voisins.

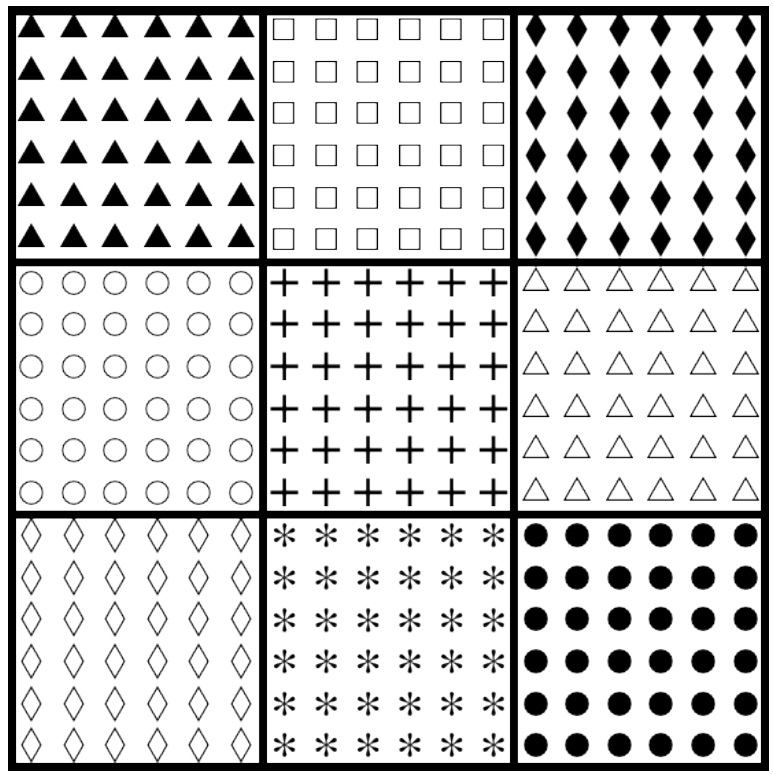


Figure 8 – Découpage en sous-domaines

Introduction

Historique

- Version 1.0 : en juin 1994, le forum MPI, avec la participation d'une quarantaine d'organisations, aboutit à la définition d'un ensemble de sous-programmes concernant la bibliothèque d'échanges de messages `MPI`
- Version 1.1 : juin 1995, avec seulement des changements mineurs
- Version 1.2 : en 1997, avec des changements mineurs pour une meilleure cohérence des dénominations de certains sous-programmes
- Version 1.3 : septembre 2008, avec des clarifications dans MPI 1.2, en fonction des clarifications elles-mêmes apportées par MPI-2.1
- Version 2.0 : apparue en juillet 1997, cette version apportait des compléments importants volontairement non intégrés dans MPI 1.0 (gestion dynamique de processus, copies mémoire à mémoire, entrées-sorties parallèles, etc.)
- Version 2.1 : juin 2008, avec seulement des clarifications dans MPI 2.0 mais aucun changement
- Version 2.2 : septembre 2009, avec seulement de petites additions
- Version 3.0 : septembre 2012, cette version apportait les communications collectives non bloquantes, nouvelle interface Fortran, etc.
- Version 3.1 : juin 2015, avec des corrections et des petites additions

Introduction

MPI 4.0

Version 4.0 : juin 2021

- Grand nombre
- Communication par morceaux
- MPI Session

Version 4.1 : novembre 2023

MPI 5.0

Version 5.0 : juin 2025

Définition d'une ABI (interface binaire-programme)

Introduction

Bibliographie

- Site du MPI Forum http://www.mpi-forum.org
- Normes disponible en PDF sur http://www.mpi-forum.org/docs/
- William Gropp, Ewing Lusk et Anthony Skjellum : *Using MPI, third edition Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 2014.
- William Gropp, Torsten Hoefler, Rajeev Thakur et Erwing Lusk : *Using Advanced MPI Modern Features of the Message-Passing Interface*, MIT Press, 2014.
- Victor Eijkhout : The Art of HPC http://theartofhpc.com

Implémentations MPI *open source*

Elles peuvent être installées sur un grand nombre d'architectures mais leurs performances sont en général en dessous de celles des implémentations constructeurs.

- MPICH : `http://www.mpich.org`
- Open MPI : `http://www.open-mpi.org`

Introduction

Outils

- Débogueurs
  - Totalview
    https://totalview.io
  - DDT
    https://www.linaroforge.com/linaro-ddt
- Outils de mesure de performances
  - FPMPI : *FPMPI*
    http://www.mcs.anl.gov/research/projects/fpmpi/WWW/
  - Scalasca : *Scalable Performance Analysis of Large-Scale Applications*
    http://www.scalasca.org
  - MUST : *MPI Runtime Correctness Analysis*
    https://itc.rwth-aachen.de/must/

Introduction

Bibliothèques scientifiques parallèles *open source*

- ScaLAPACK : résolution de problèmes d'algèbre linéaire par des méthodes directes.
  http://www.netlib.org/scalapack/
- PETSc : résolution de problèmes d'algèbre linéaire et non-linéaire par des méthodes itératives.
  https://petsc.org/release/
- PaStiX : résolution de grands systèmes linéaires creux.
  https://solverstack.gitlabpages.inria.fr/pastix/
- FFTW : transformées de Fourier rapides.
  http://www.fftw.org
- HDF5 : Lecture et écriture sur fichiers.
  https://www.hdfgroup.org/solutions/hdf5/

Environnement

Environnement

Description

- Toute unité de programme appelant des fonctions MPI doit inclure le module `mpi4py`.
- Le sous-programme `MPI_Init()` permet d'initialiser l'environnement nécessaire :

```
mpi4py.MPI.Init()
```

  Par défaut, `mpi4py` initialise l'environnement MPI au moment de l'importation du module.

- Réciproquement, le sous-programme `MPI_Finalize()` désactive cet environnement :

```
mpi4py.MPI.Finalize()
```

  Par défaut, `mpi4py` désactive l'environnement MPI au moment de la fin de l'exécution du programme.

Environnement

Communicateurs

- Toutes les opérations effectuées par MPI portent sur des communicateurs. Le communicateur par défaut est `MPI_COMM_WORLD` qui comprend tous les processus actifs.

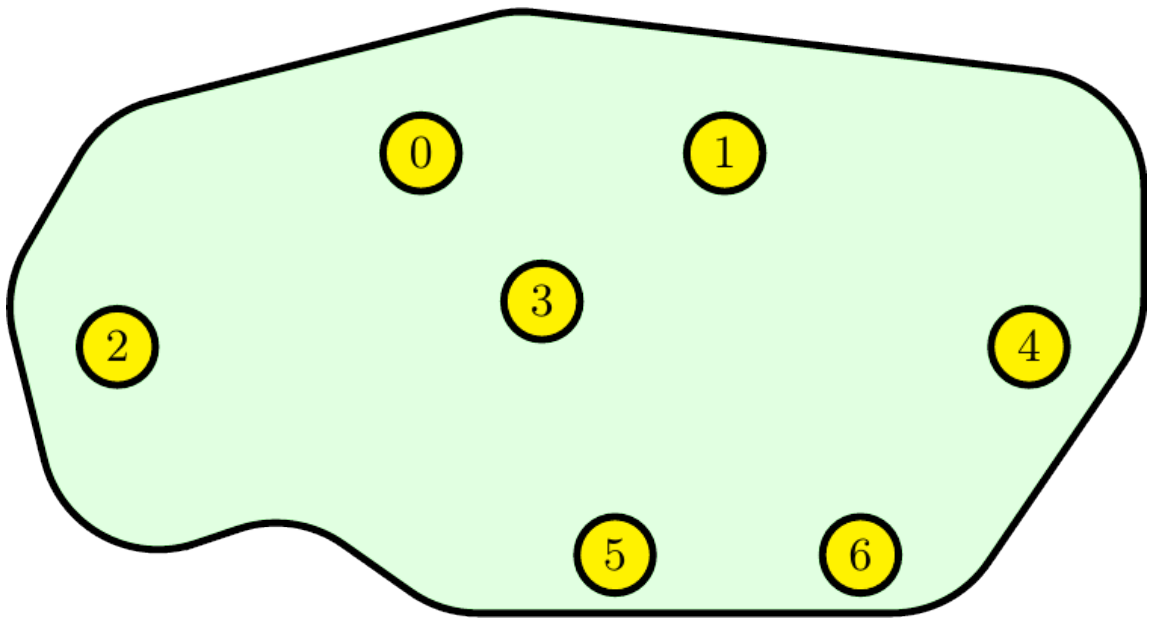


Figure 9 – Communicateur MPI_COMM_WORLD

Environnement

Arrêt d'un programme

Parfois un programme se trouve dans une situation où il doit s'arrêter sans attendre la fin normale. C'est typiquement le cas si un des processus ne peut pas allouer la mémoire nécessaire à son calcul. Dans ce cas, il faut utiliser le sous-programme `MPI_Abort()` et non l'instruction Python exit.

```
mpi4py.MPI.Comm.Abort(errorcode=0)
```

- `comm` (instance de la classe `mpi4py.MPI.Comm`) : tous les processus appartenant à ce communicateur seront stoppés, il est donc conseillé d'utiliser MPI.`COMM_WORLD` ;
- `errorcode` : numéro d'erreur retourné à l'environnement UNIX.

La gestion des erreurs dans `mpi4py` est faite via des exceptions. Les erreurs retournées par les appels MPI dans le code Python lèveront une instance de la classe MPI.Exception, qui est une sous-classe de l'exception standard Python `RuntimeError`. La non-capture de l'exception provoque l'arrêt uniquement du processus sauf à utiliser l'option `-m mpi4py` avec l'interpréteur Python.

Environnement

Rang et nombre de processus

- À tout instant, on peut connaître le nombre de processus gérés par un communicateur en appelant le sous-programme `Get_size()` :

```
# Retourne le nombre de processus
mpi4py.MPI.Comm.Get_size()
```

- De même, le sous-programme `Get_rank()` permet d'obtenir le rang d'un processus (i.e. son numéro d'instance, qui est un nombre compris entre 0 et la valeur renvoyée par `Get_size()` – 1) :

```
# Retourne le rang
mpi4py.MPI.Comm.Get_rank()
```

Environnement

Exemple

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
nb_procs = comm.Get_size()
rang = comm.Get_rank()

print(f"Je suis le processus {rang} parmi {nb_procs}")
```

```
> mpiexec -n 7 python -m mpi4py qui_je_suis.py

Je suis le processus 3 parmi 7
Je suis le processus 0 parmi 7
Je suis le processus 4 parmi 7
Je suis le processus 1 parmi 7
Je suis le processus 5 parmi 7
Je suis le processus 2 parmi 7
Je suis le processus 6 parmi 7
```

Environnement

Compilation et exécution d'un code MPI

- Pour exécuter un code MPI, on utilise un lanceur d'application MPI qui ordonne le lancement de l'exécution sur un nombre de processus choisi.
- Le lanceur défini par la norme MPI est `mpiexec`. Il existe également des lanceurs non standards, comme `mpirun`.

```
> mpiexec -n <nombre de processus> python -m mpi4py mon_script.py
```

Travaux pratiques MPI – Exercice 1 : Environnement MPI

- Implémenter un programme MPI dans lequel chaque processus affiche un message indiquant si son rang est pair ou impair. Par exemple :

```
> mpiexec -n 4 python -m mpi4py ./pair_impair
Moi, processus 0, je suis de rang pair
Moi, processus 2, je suis de rang pair
Moi, processus 3, je suis de rang impair
Moi, processus 1, je suis de rang impair
```

- Pour tester la parité, l'opérateur *modulo* est `%` : `a%b`

- Pour exécuter votre programme, utilisez la commande `make exe`

- Pour être reconnu par le Makefile, le programme doit se nommer `pair_impair.py`

Communications point à point

Communications point à point

Notions générales

Une communication dite point à point a lieu entre deux processus, l'un appelé processus émetteur et l'autre processus récepteur (ou destinataire).

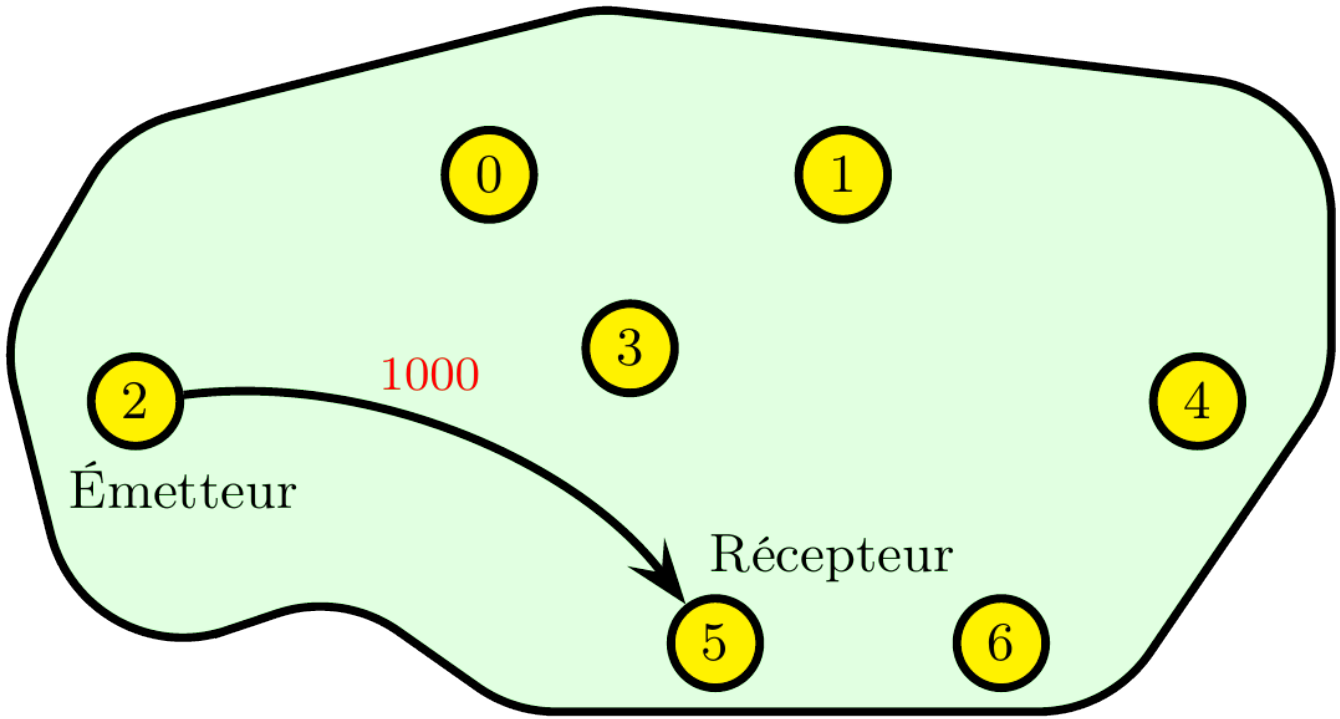


Figure 10 – Communication point à point

Communications point à point

Notions générales

- L'émetteur et le récepteur sont identifiés par leur rang dans le communicateur.
- L'entité transmise entre deux processus est appelée message.
- Un message est caractérisé par son enveloppe. Celle-ci est constituée :
  - du rang du processus émetteur ;
  - du rang du processus récepteur ;
  - de l'étiquette (*tag*) du message ;
  - du communicateur qui définit le groupe de processus et le contexte de communication.
- Les données échangées sont typées (entiers, réels, etc ou types dérivés personnels).
- Il existe dans chaque cas plusieurs modes de transfert, faisant appel à des protocoles différents.
- Si deux messages sont envoyés avec la même enveloppe, l'ordre de réception et d'envoi sont les mêmes.

Communications point à point

MPI4PY : Deux types de messages

- Il y a deux types de messages avec mpi4py ;
- Un type pour les objets Python, avec un nom de fonction de communication en minuscule ;
- Ce type de message utilise la sérialisation pour les communications et est moins performant ;
- Il n'est possible de communiquer qu'un seul objet Python avec ce type de message ;
- L'objet reçu est la valeur de retour de la fonction recevant le message ;
- L'autre type est pour les tableaux contigus comme avec NumPy, avec un nom de fonction de communication avec la première lettre en majuscule ;
- Pour ce type de message il faut fournir un triplet (buffer, longueur, type) pour la communication ;
- La longueur et le type sont optionnels ;
- L'objet reçu est dans les arguments de la fonction ;
- Ce type de message est plus performant.

Communications point à point

Opération d'envoi `MPI_Send`

```
mpi4py.MPI.Comm.Send([message, longueur, type_message], dest, tag=0)
mpi4py.MPI.Comm.send(obj, dest, tag=0)
```

Envoi, à partir de l'adresse message, d'un message de taille longueur, de type type_message, étiqueté tag, au processus dest dans le communicateur comm.

Remarque :
Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message puisse être réécrit sans risque d'écraser la valeur qui devait être envoyée.

Communications point à point

Opération de réception `MPI_Recv`

```
mpi4py.MPI.Comm.Recv([message, longueur, type_message],
                     source=ANY_SOURCE, tag=ANY_TAG, status=None)
# Retourne l'objet recu
mpi4py.MPI.Comm.recv(source=ANY_SOURCE, tag=ANY_TAG, status=None)
```

Réception, à partir de l'adresse message, d'un message de taille longueur, de type type_message, étiqueté tag, du processus source.

Remarques :

- statut stocke des informations sur la communication : source, tag, code,...
- L'appel `MPI_Recv` ne pourra fonctionner avec une opération `MPI_Send` que si ces deux appels ont la même enveloppe (source, dest, tag, Comm).
- Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message corresponde au message reçu.

Communications point à point

Exemple (voir Fig. 10)

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
etiquette = 100

if rang == 2:
    valeur = 1000
    comm.send(valeur, dest=5, tag=etiquette)
elif rang == 5:
    valeur = comm.recv(source=2, tag=etiquette)
    print(f"Moi, processus 5, ai recu {valeur} du processus 2.")
```

```
> mpiexec -n 7 python -m mpi4py point_a_point.py

Moi, processus 5, ai recu 1000 du processus 2
```

Exemple (voir Fig. 10)

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
etiquette = 100

valeur = np.zeros(1, dtype=np.int32)

if rang == 2:
    valeur[0] = 1000
    comm.Send(valeur, dest=5, tag=etiquette)
elif rang == 5:
    comm.Recv(valeur, source=2, tag=etiquette)
    print(f"Moi, processus 5, ai recu {valeur[0]} du processus 2.")
```

Communications point à point

Types de données de base MPI4PY

Type MPI4PY	dtype Numpy
mpi4py.MPI.INT	numpy.int32
mpi4py.MPI.LONG	numpy.int64
mpi4py.MPI.FLOAT	numpy.float32
mpi4py.MPI.DOUBLE	numpy.float64
mpi4py.MPI.BYTE	numpy.byte

Il existe des fonctions pour convertir les types numpy et les types mpi4py

```
# Retourne un type mpi4py
mpi4py.util.dtlib.from_numpy_dtype(dtype)
# Retourne un dtype
mpi4py.util.dtlib.to_numpy_dtype(datatype)
```

Autres possibilités

- À la réception d'un message, le rang de l'émetteur et l'étiquette peuvent être des « *jokers* », respectivement `MPI.ANY_SOURCE` et `MPI.ANY_TAG`.
- Une communication impliquant le processus « fictif » de rang `MPI.PROC_NULL` n'a aucun effet.
- On peut communiquer des structures de données plus complexes en créant ses propres types dérivés.
- Il existe d'autres opérations qui effectuent simultanément un envoi et une réception : `MPI_Sendrecv()` et `MPI_Sendrecv_replace()`.

Communications point à point

Opération d'envoi et de réception simultanés `MPI_Sendrecv`

```
mpi4py.MPI.Comm.Sendrecv([message_emis, longueur_message_emis, type_message_emis],
                         dest, sendtag=0,
                         recvbuf=[message_recu, longueur_message_recu, type_message_recu],
                         source=ANY_SOURCE, recvtag=ANY_TAG, status=None)
# Retourne l'objet recu
mpi4py.MPI.Comm.sendrecv(sendobj, dest, sendtag=0,
                         recvbuf=None, source=ANY_SOURCE, recvtag=ANY_TAG,
                         status=None)
```

- Envoi, à partir de l'adresse message_emis, d'un message de taille longueur_message_emis, de type type_message_emis, étiqueté sendtag, au processus dest dans le communicateur Comm ;
- Réception, à partir de l'adresse message_recu, d'un message de taille longueur_message_recu, de type type_message_recu, étiqueté recvtag, du processus source dans le communicateur Comm.

Remarque :

- La zone de réception message_recu doit différer de la zone d'envoi message_emis.

Opération d'envoi et de réception simultanés `MPI_Sendrecv`

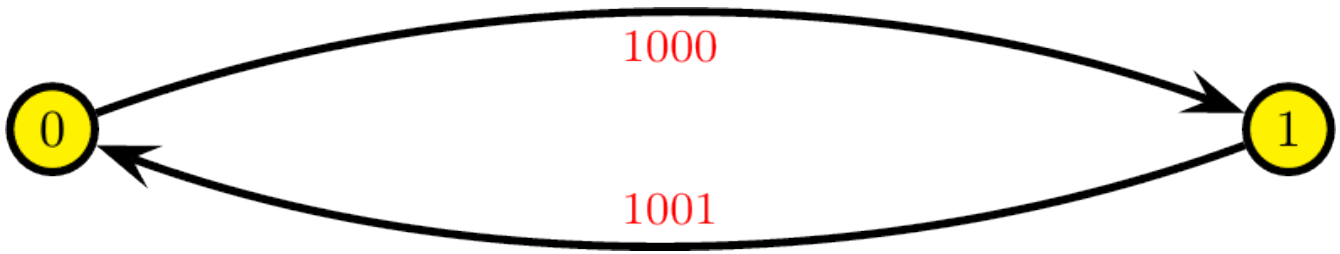


Figure 11 – Communication `sendrecv` entre les processus 0 et 1

Communications point à point

Exemple (voir Fig. 11)

```
from mpi4py import MPI

etiquette = 110

comm = MPI.COMM_WORLD
rang = comm.Get_rank()

num_proc = (rang + 1) % 2
message = np.zeros(1, dtype=np.int32)
valeur = np.zeros(1, dtype=np.int32)
message[0] = rang + 1000

comm.Sendrecv(message, dest=num_proc, sendtag=etiquette,
              recvbuf=valeur, source=num_proc, recvtag=etiquette)

print(f"Moi, processus {rang}, ai recu {valeur[0]} du processus {num_proc}.")
```

```
> mpiexec -n 2 python -m mpi4py sendrecv.py

Moi, processus 1, ai recu 1000 du processus 0
Moi, processus 0, ai recu 1001 du processus 1
```

Communications point à point

Attention !
Dans le cas d'une implémentation synchrone de `MPI_Send()`, l'exemple précédent serait en situation de verrouillage si l'appel à `MPI_Sendrecv()` était remplacé par un `MPI_Send()` suivi d'un `MPI_Recv()`. En effet, chacun des deux processus attendrait un ordre de réception qui ne viendrait jamais, puisque les deux envois resteraient en suspens.

```
val[0] = rang + 1000
comm.Send(val, dest=num_proc, tag=etiquette)
comm.Recv(val, source=num_proc, tag=etiquette)
```

Communications point à point

Opération d'envoi et de réception simultanés `MPI_Sendrecv_replace`

```
mpi4py.MPI.Comm.Sendrecv_replace([message, longueur, type_message],
                                 dest, sendtag=0,
                                 source=ANY_SOURCE, recvtag=ANY_TAG,
                                 status=None)
```

- Envoi, à partir de l'adresse message, d'un message de taille longueur, de type type_message, étiqueté sendtag, au processus dest dans le communicateur Comm ;
- Réception d'un message à la même adresse, d'une taille et d'un type identique, étiqueté recvtag, du processus source dans le communicateur Comm.

Remarque :

- Contrairement à l'usage imposée par `MPI_Sendrecv()` , la zone de réception coincide ici avec la zone d'envoi message.

Communications point à point

Exemple

```
from mpi4py import MPI
import numpy as np

m = 4
etiquette = 11

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
intnp = np.int32
A = np.zeros((m, m), dtype=intnp)

if rang == 0:
    A = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 16]], dtype=intnp)
    comm.Send([A, 3], dest=1, tag=etiquette)
else:
    statut = MPI.Status()
    comm.Recv([A[0, 1:], 3], status=statut)
    print(f"Moi processus {rang}, ai recu 3 elements du processus "
          f"{statut.source} avec comme etiquette "
          f"{statut.tag} les elements sont {A[0, 1]} {A[0, 2]} {A[0, 3]}.")
```

Communications point à point

```
> mpiexec -n 2 python -m mpi4py joker.py
Moi processus 1, ai recu 3 elements du processus 0
 avec comme etiquette 11 les elements sont 1 2 3.
```

Travaux pratiques MPI – Exercice 2 : Ping-pong

- Communications point à point : *Ping-Pong* entre deux processus

- L'exercice 2 est décomposé en 3 étapes :

  1. *Ping* : compléter le script `ping_pong_1.py` de manière à ce que le processus de rang 0 envoie un message contenant une série aléatoire de 1000 réels au rang 1.

  2. *Ping-Pong* : compléter le script `ping_pong_2.py` de manière à ce que le processus de rang 1 renvoie le message vers le processus de rang 0, et mesurer le temps pris par la communication à l'aide de la fonction `MPI.Wtime()`.

  3. *Match de Ping-Pong* : compléter le script `ping_pong_3.py` de manière à enchainer 9 *Ping-Pong*, en faisant varier la taille du message, et mesurer les temps pris par chaque échange. Les débits correspondants seront affichés.

Travaux pratiques MPI – Exercice 2 : Ping-pong

Remarques :

- Pour exécuter la première étape : `make exe1`
- Pour exécuter la seconde étape : `make exe2`
- Pour exécuter la dernière étape : `make exe3`

- La génération de nombres réels pseudo-aléatoires uniformément répartis dans l'intervalle [0,1[se fait en Python par un appel au sous-programme de numpy `random.rand`

- Les mesures de temps peuvent s'effectuer de la façon suivante :

```
temps_debut = MPI.Wtime()
.................................................
temps_fin = MPI.Wtime();
print(f"... en {temps_fin-temps_debut} secondes.\n")
```

Communications collectives

Communications collectives

Notions générales

- Les communications collectives permettent de faire en une seule opération une série de communications point à point.
- Une communication collective concerne toujours tous les processus du communicateur indiqué.
- Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée, au sens des communications point-à-point (donc quand la zone mémoire concernée peut être modifiée).
- La gestion des étiquettes dans ces communications est transparente et à la charge du système. Elles ne sont donc jamais définies explicitement lors de l'appel à ces sous-programmes. Cela a entre autres pour avantage que les communications collectives n'interfèrent jamais avec les communications point à point.

Communications collectives

Types de communications collectives

Il y a trois types de sous-programmes :

1. celui qui assure les synchronisations globales : `MPI_Barrier()`.
2. ceux qui ne font que transférer des données :
   - diffusion globale de données : `MPI_Bcast()` ;
   - diffusion sélective de données : `MPI_Scatter()` ;
   - collecte de données réparties : `MPI_Gather()` ;
   - collecte par tous les processus de données réparties : `MPI_Allgather()` ;
   - collecte et diffusion sélective, par tous les processus, de données réparties : `MPI_Alltoall()`.
3. ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
   - opérations de réduction (somme, produit, maximum, minimum, etc.), qu'elles soient d'un type prédéfini ou d'un type personnel : `MPI_Reduce()` ;
   - opérations de réduction avec diffusion du résultat (équivalent à un `MPI_Reduce()` suivi d'un `MPI_Bcast()`) : `MPI_Allreduce()`.

Communications collectives

Synchronisation globale : `MPI_Barrier()`

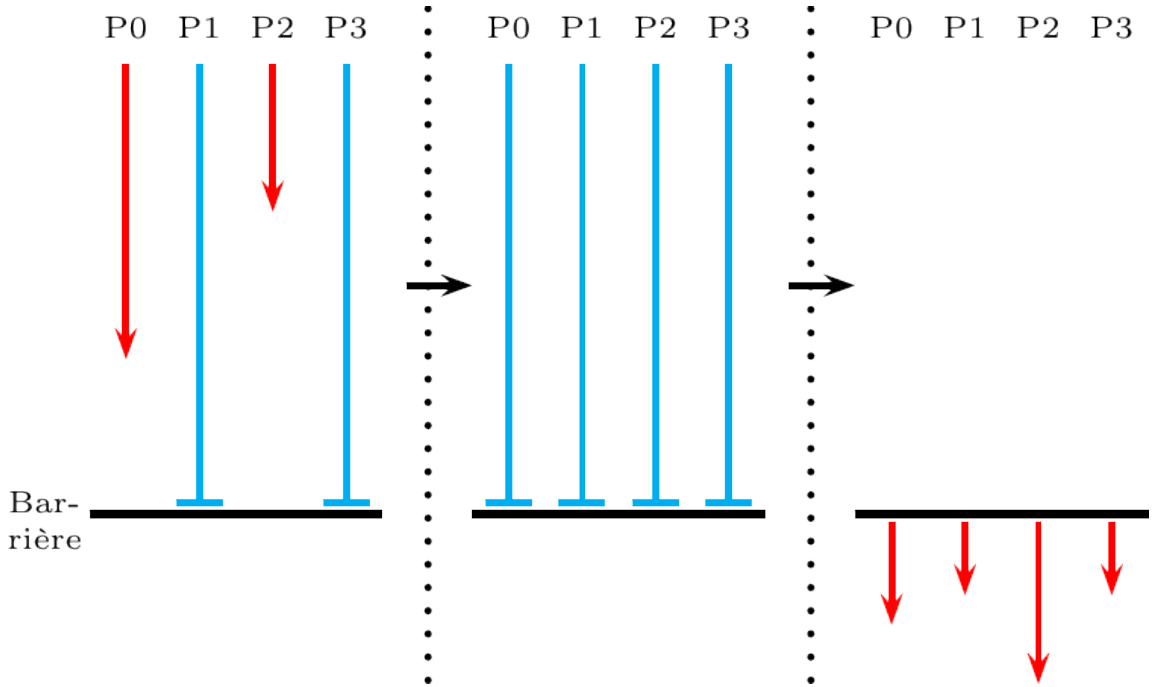


Figure 12 – Synchronisation globale : `MPI_Barrier()`

```
mpi4py.MPI.Comm.Barrier()
mpi4py.MPI.Comm.barrier()
```

Communications collectives

Diffusion générale : `MPI_Bcast()`

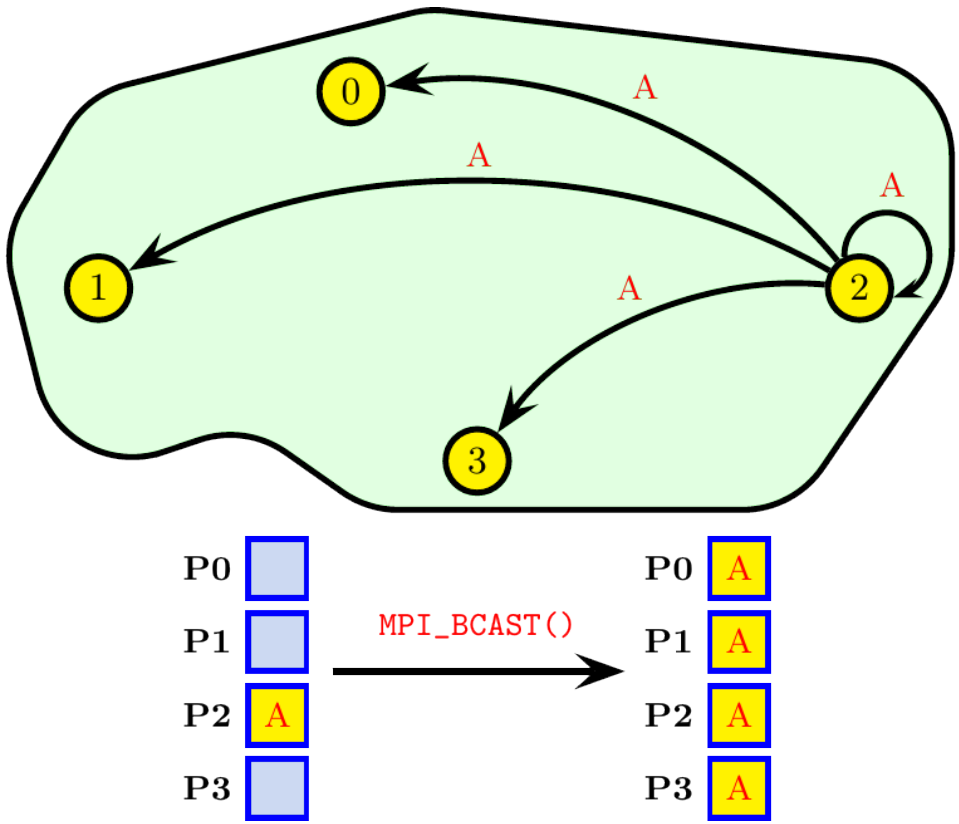


Figure 13 – Diffusion générale : `MPI_Bcast()`

Communications collectives

Diffusion générale : `MPI_Bcast()`

```
mpi4py.MPI.Comm.Bcast([message, longueur, type_message], root=0)
# Retourne l'objet diffuse
mpi4py.MPI.Comm.bcast(obj, root=0)
```

1. Envoi, à partir de l'adresse message, d'un message constitué de longueur élément de type type_message, par le processus root , à tous les autres processus du communicateur comm.
2. Réception de ce message à l'adresse message pour les processus autre que root

Communications collectives

Exemple de `MPI_Bcast()`

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rang = comm.Get_rank()

valeur = np.zeros(1, dtype=np.int32)

if rang == 2:
    valeur[0] = 1002

comm.Bcast(valeur, root=2)

print(f"Moi, processus {rang}, ai recu {valeur[0]} du processus 2")
```

```
> mpiexec -n 4 python -m mpi4py bcast.py

Moi, processus 2, ai recu 1002 du processus 2
Moi, processus 0, ai recu 1002 du processus 2
Moi, processus 1, ai recu 1002 du processus 2
Moi, processus 3, ai recu 1002 du processus 2
```

Communications collectives

Diffusion sélective : `MPI_Scatter()`

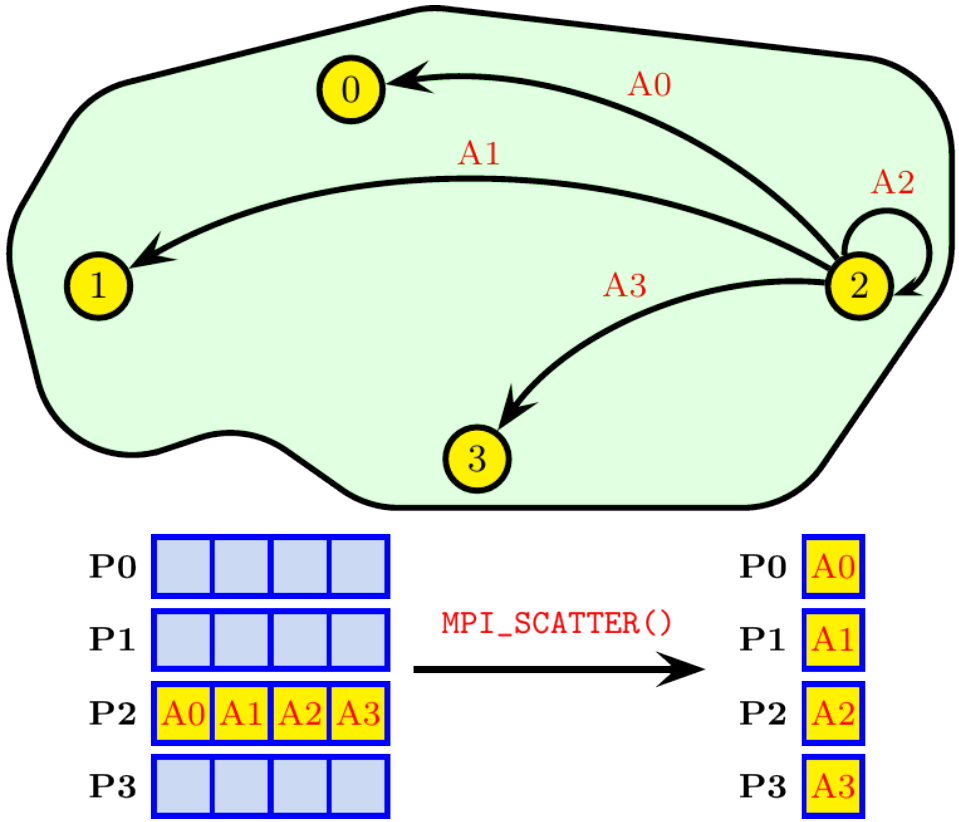


Figure 14 – Diffusion sélective : `MPI_Scatter()`

Communications collectives

Diffusion sélective : `MPI_Scatter()`

```
mpi4py.MPI.Comm.Scatter([message_a_repartir, longueur_message_emis, type_message_emis],
                        [message_recu, longueur_message_recu, type_message_recu],
                        root=0)
# Retourne l'objet
mpi4py.MPI.Comm.scatter(sendobj, root=0)
```

1. Distribution, par le processus root , à partir de l'adresse message_a_repartir, d'un message de taille longueur_message_emis, de type type_message_emis, à tous les processus du communicateur comm ;
2. réception du message à l'adresse message_recu, de longueur longueur_message_recu et de type type_message_recu par tous les processus du communicateur comm.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont distribuées en tranches égales, une tranche étant constituée de longueur_message_emis éléments du type type_message_emis.
- La ième tranche est envoyée au ième processus.

Communications collectives

Exemple de `MPI_Scatter()`

```python
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
nb_procs = comm.Get_size()

nb_valeurs = 8
longueur_tranche = nb_valeurs // nb_procs

if rang == 2:
    valeurs = np.arange(1001, 1001 + nb_valeurs, dtype=np.float32)
    print(f"Moi, processus {rang}, envoie mon tableau de valeurs :{valeurs}")
else:
    valeurs = None

donnees = np.empty(longueur_tranche, dtype=np.float32)
comm.Scatter(valeurs, donnees, root=2)
print(f"Moi, processus {rang}, ai recu {donnees} du processus 2")
```

```
> mpiexec -n 4 python -m mpi4py scatter.py
Moi, processus 2 envoie mon tableau valeurs :
[1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.]

Moi, processus 0, ai recu [1001. 1002.] du processus 2
Moi, processus 1, ai recu [1003. 1004.] du processus 2
Moi, processus 3, ai recu [1007. 1008.] du processus 2
Moi, processus 2, ai recu [1005. 1006.] du processus 2
```

Communications collectives

Collecte : `MPI_Gather()`

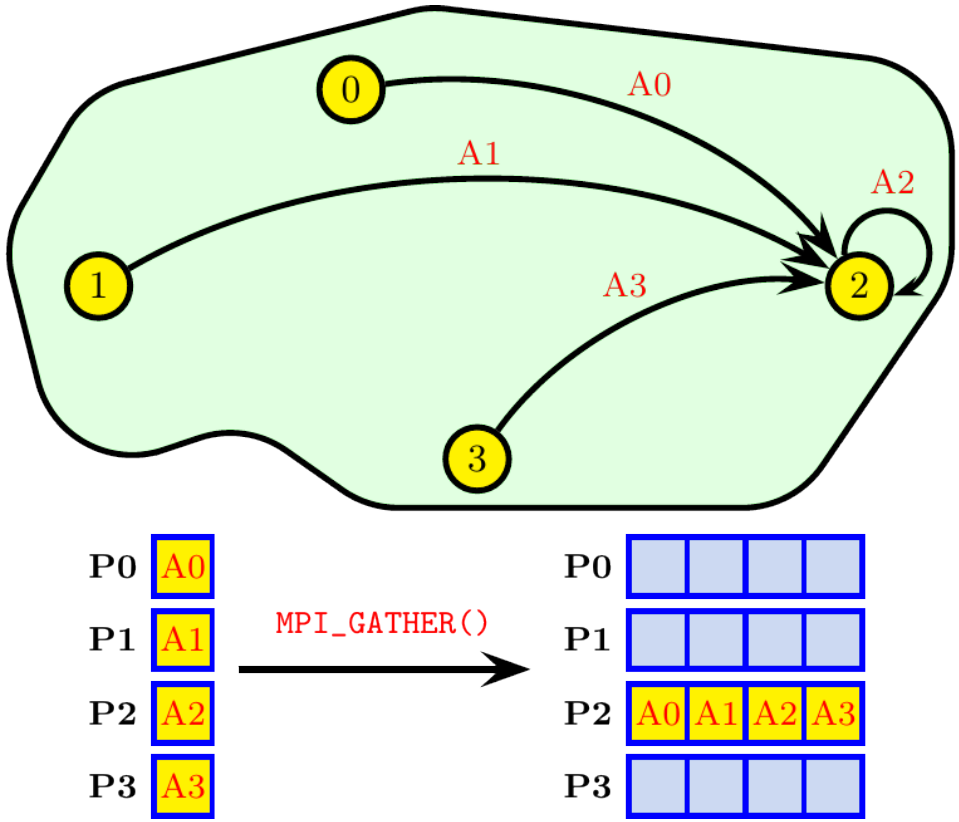


Figure 15 – Collecte : `MPI_Gather()`

Communications collectives

Collecte : `MPI_Gather()`

```
mpi4py.MPI.Comm.Gather([message_emis, longueur_message_emis, type_message_emis],
                       [message_recu, longueur_message_recu, type_message_recu],
                       root=0)
# Pour root retourne une liste d'objets
mpi4py.MPI.Comm.gather(sendobj, root=0)
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par le processus root, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

Communications collectives

Collecte : `MPI_Gather()`

```python
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
nb_procs = comm.Get_size()

nb_valeurs = 8
longueur_tranche = nb_valeurs // nb_procs

valeurs = np.arange(1001+rang*longueur_tranche,
                    1001+(rang+1)*longueur_tranche,
                    dtype=np.float32)
print(f"Moi, processus {rang}, envoie mon tableau de valeurs : {valeurs}")

if rang == 2:
    donnees = np.empty(nb_valeurs, dtype=np.float32)
else:
    donnees = None
comm.Gather(valeurs, donnees, root=2)
if rang == 2:
    print(f"Moi, processus {rang}, ai recu {donnees}")
```

```
> mpiexec -n 4 python -m mpi4py ./0403-gather.py
Moi, processus 1, envoie mon tableau de valeurs : [1003. 1004.]
Moi, processus 2, envoie mon tableau de valeurs : [1005. 1006.]
Moi, processus 3, envoie mon tableau de valeurs : [1007. 1008.]
Moi, processus 0, envoie mon tableau de valeurs : [1001. 1002.]

Moi, processus 2, ai recu [1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.]
```

Communications collectives

Collecte générale : `MPI_Allgather()`

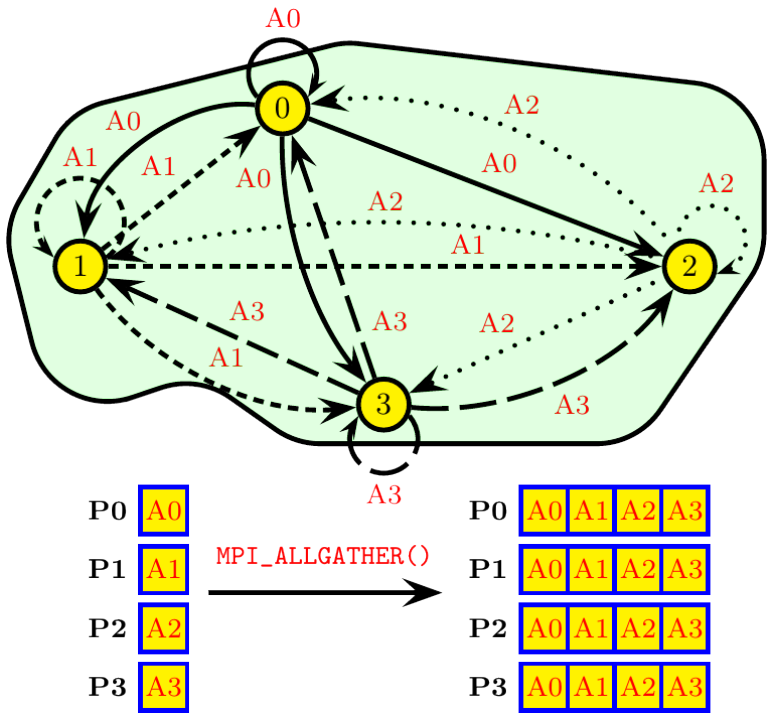


Figure 16 – Collecte générale : `MPI_Allgather()`

Communications collectives

Collecte générale : `MPI_Allgather()`

Correspond à un `MPI_Gather()` suivi d'un `MPI_Bcast()` :

```
mpi4py.MPI.Comm.Allgather([message_emis, longueur_message_emis, type_message_emis],
                          [message_recu, longueur_message_recu, type_message_recu])
# Retourne une liste d'objets
mpi4py.MPI.Comm.allgather(sendobj)
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par tous les processus, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

Communications collectives

Exemple de `MPI_Allgather()`

```python
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
nb_procs = comm.Get_size()

nb_valeurs = 8
longueur_tranche = nb_valeurs // nb_procs

valeurs = np.arange(1001+rang*longueur_tranche,
                    1001+(rang+1)*longueur_tranche,
                    dtype=np.float32)
donnees = np.empty(nb_valeurs, dtype=np.float32)

comm.Allgather(valeurs, donnees)

print(f"Moi, processus {rang}, ai recu {donnees}")
```

```
> mpiexec -n 4 python -m mpi4py allgather.py

Moi, processus 1, ai recu [1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.]
Moi, processus 3, ai recu [1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.]
Moi, processus 2, ai recu [1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.]
Moi, processus 0, ai recu [1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.]
```

Collecte "variable" : MPI_Gatherv()

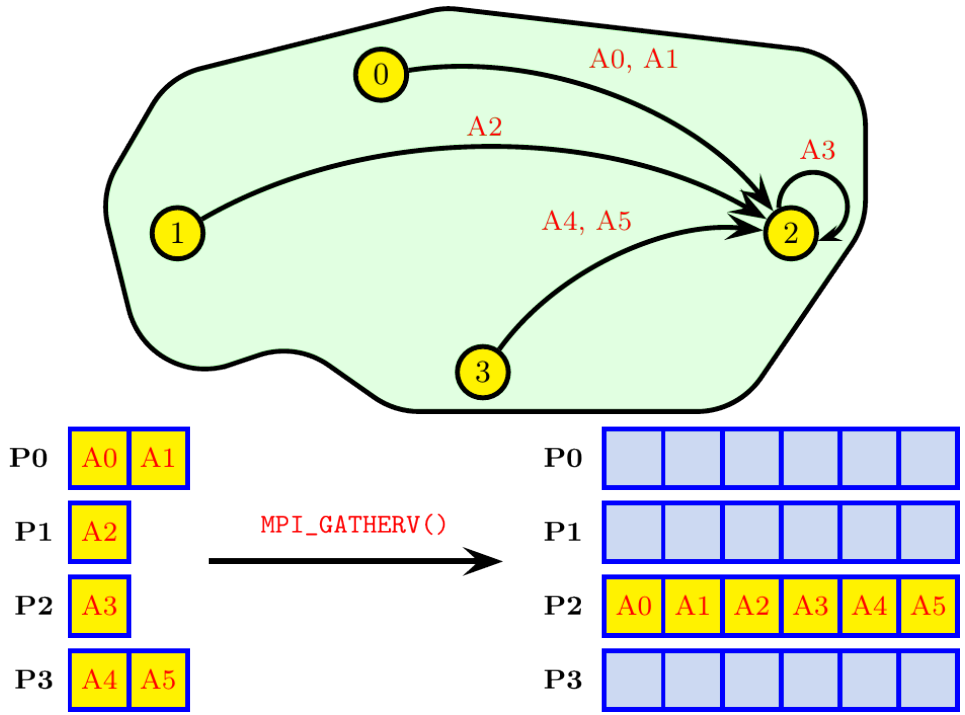


Figure 17 – Collecte : MPI_Gatherv()

Communications collectives

Collecte "variable" : `MPI_Gatherv()`

Correspond à un `MPI_Gather()` pour lequel la taille des messages varie :

```
mpi4py.MPI.Comm.Gatherv([message_emis, longueur_message_emis, type_message_emis],
                        [message_recu, nb_elts_recus, deplts, type_message_recu],
                        root=0)
```

Le ième processus du communicateur comm envoie au processus root, un message depuis l'adresse message_emis, de taille longueur_message_emis, de type type_message_emis, avec réception du message à l'adresse message_recu, de type type_message_recu, de taille nb_elts_recus(i) avec un déplacement de deplts(i).

Remarques :

- Les couples (longueur_message_emis, type_message_emis) du ième processus et (nb_elts_recus(i), type_message_recu) du processus rang_dest doivent être tels que les quantités de données envoyées et reçues soient égales.

Communications collectives

Exemple de `MPI_Gatherv()`

```
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
nb_procs = comm.Get_size()

nb_valeurs = 10
longueur_tranche = nb_valeurs // nb_procs
reste = nb_valeurs % nb_procs
if rang < reste:
    longueur_tranche += 1

valeurs = np.empty(longueur_tranche, dtype=np.float32)
debut = 1001+rang*(nb_valeurs // nb_procs)+(rang if rang < reste else reste)
valeurs = np.arange(debut, debut+longueur_tranche, dtype=np.float32)
print(f"Moi, processus {rang} envoie mon tableau valeurs : {valeurs}")

if rang == 2:
    nb_elements_recus = np.empty(nb_procs, dtype=np.int32)
    deplacements = np.empty(nb_procs, dtype=np.int32)
    nb_elements_recus[0] = nb_valeurs // nb_procs
    if reste > 0:
        nb_elements_recus[0] += 1
    deplacements[0] = 0
    for i in range(1, nb_procs):
        deplacements[i] = deplacements[i-1] + nb_elements_recus[i-1]
        nb_elements_recus[i] = nb_valeurs // nb_procs
        if i < reste:
            nb_elements_recus[i] += 1
    donnees = np.empty(nb_valeurs, dtype=np.float32)
else :
    nb_elements_recus = None
    deplacements = None
    donnees = None
```

Communications collectives

Exemple de `MPI_Gatherv()` (suite)

```
comm.Gatherv(valeurs, [donnees, nb_elements_recus, deplacements,MPI.FLOAT],
             root=2)
if rang == 2:
    print(f"Moi, processus {rang}, ai recu {donnees}")
```

```
> mpiexec -n 4 gatherv

Moi, processus  0 envoie mon tableau valeurs :  [1001. 1002. 1003.]
Moi, processus  2 envoie mon tableau valeurs :  [1007. 1008.]
Moi, processus  3 envoie mon tableau valeurs :  [1009. 1010.]
Moi, processus  1 envoie mon tableau valeurs :  [1004. 1005. 1006.]

Moi, processus 2 ai recu  [1001. 1002. 1003.  1004. 1005. 1006.  1007. 1008.  1009. 1010.]
```

Communications collectives

Collectes et diffusions sélectives : `MPI_Alltoall()`

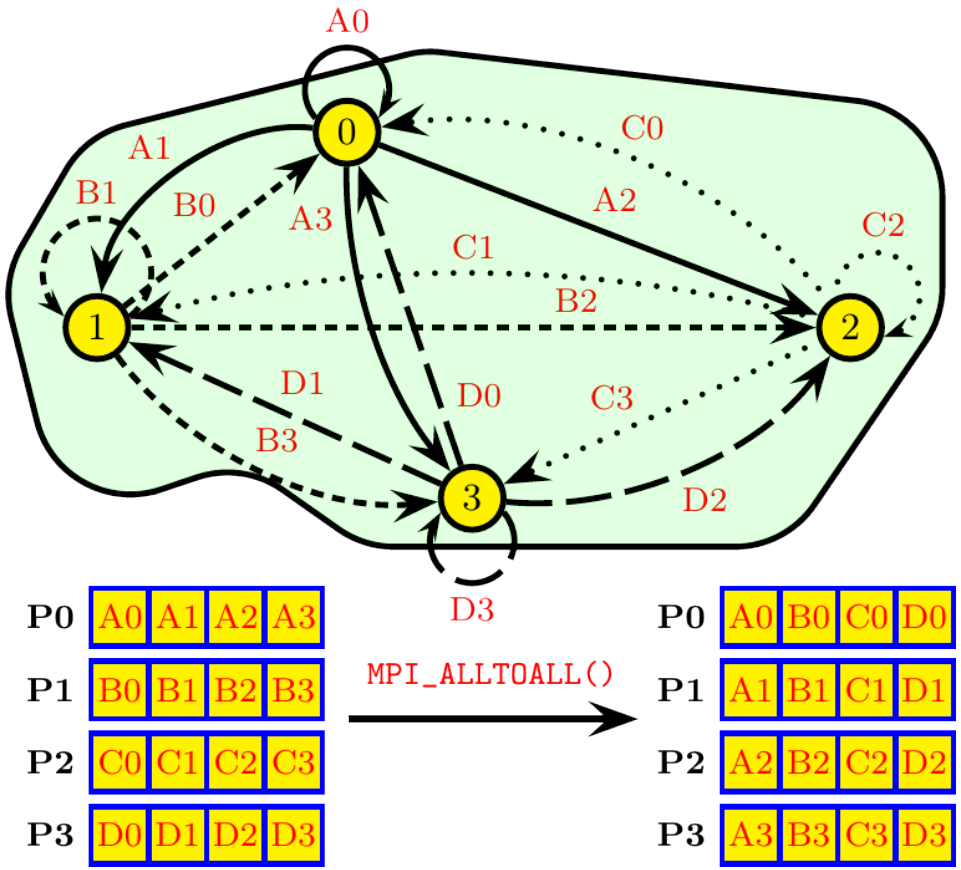


Figure 18 – Collecte et diffusion sélectives : `MPI_Alltoall()`

Communications collectives

Collectes et diffusions sélectives : `MPI_Alltoall()`

```
mpi4py.MPI.Comm.Alltoall([message_emis, longueur_message_emis, type_message_emis],
                         [message_recu, longueur_message_recu, type_message_recu])
# Retourne une liste d'objets
mpi4py.MPI.Comm.alltoall(sendobj)
```

Ici, le ième processus envoie la jème tranche au jème processus qui le place à l'emplacement de la ième tranche.

Remarque :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.

Communications collectives

Exemple de `MPI_Alltoall()`

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
nb_procs = comm.Get_size()

nb_valeurs = 8
debut = 1001+rang*nb_valeurs
valeurs = np.arange(debut, debut+nb_valeurs, dtype=np.float32)
print(f"Moi, processus {rang} envoie mon tableau valeurs : {valeurs}")

longueur_tranche = nb_valeurs // nb_procs
donnees = np.empty(nb_valeurs, dtype=np.float32)
comm.Alltoall(valeurs, donnees)

print(f"Moi, processus {rang} ai recu : {donnees}")
```

Communications collectives

Exemple de `MPI_Alltoall()` (suite)

```
> mpiexec -n 4 python -m mpi4py alltoall.py
Moi, processus 1 envoie mon tableau valeurs :
[1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.]
Moi, processus 0 envoie mon tableau valeurs :
[1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.]
Moi, processus 2 envoie mon tableau valeurs :
[1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.]
Moi, processus 3 envoie mon tableau valeurs :
[1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.]

Moi, processus 0, ai recu [1001. 1002.  1009. 1010.  1017. 1018.  1025. 1026.]
Moi, processus 2, ai recu [1005. 1006.  1013. 1014.  1021. 1022.  1029. 1030.]
Moi, processus 1, ai recu [1003. 1004.  1011. 1012.  1019. 1020.  1027. 1028.]
Moi, processus 3, ai recu [1007. 1008.  1015. 1016.  1023. 1024.  1031. 1032.]
```

Réductions réparties

- Une réduction est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur. Des exemples typiques sont la somme des éléments d'un vecteur `SUM(A(:))` ou la recherche de l'élément de valeur maximum dans un vecteur `MAX(V(:))`.
- MPI propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus. Le résultat est obtenu sur un seul processus (`MPI_Reduce()`) ou bien sur tous (`MPI_Allreduce()`, qui est en fait équivalent à un `MPI_Reduce()` suivi d'un `MPI_Bcast()`).
- Si plusieurs éléments sont concernés par processus, la fonction de réduction est appliquée à chacun d'entre eux (par exemple à tous les éléments d'un vecteur).

Réductions réparties : `MPI_Reduce()`

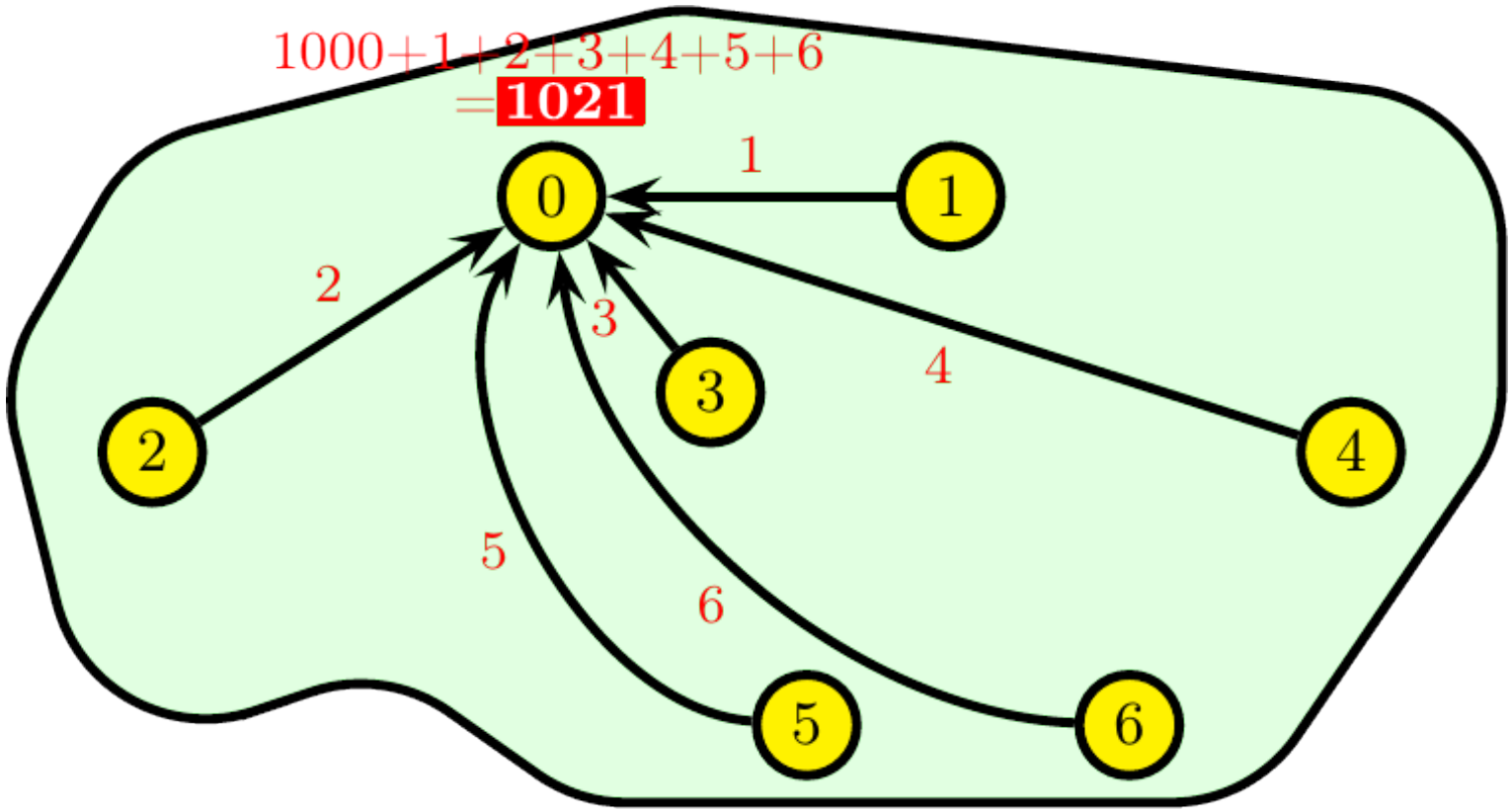


Figure 19 – Réduction répartie : `MPI_Reduce()` avec l'opérateur somme

Communications collectives

Opérations pour réductions réparties

Nom	Opération
`mpi4py.MPI.SUM`	Somme des éléments
`mpi4py.MPI.PROD`	Produit des éléments
`mpi4py.MPI.MAX`	Recherche du maximum
`mpi4py.MPI.MIN`	Recherche du minimum
`mpi4py.MPI.MAXLOC`	Recherche de l'indice du maximum
`mpi4py.MPI.MINLOC`	Recherche de l'indice du minimum
`mpi4py.MPI.LAND`	ET logique
`mpi4py.MPI.LOR`	OU logique
`mpi4py.MPI.LXOR`	OU exclusif logique

Communications collectives

Réductions réparties : `MPI_Reduce()`

```
mpi4py.MPI.Reduce([message_emis, longueur, type_message],
                  message_recu, op=SUM, root=0)
# Retourne le resultat de la reduction pour root seulement
mpi4py.MPI.reduce(sendobj, op=SUM, root=0)
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type_message, pour les processus du communicateur comm,
2. Ecrit le résultat à l'adresse message_recu pour le processus de rang root.

Communications collectives

Exemple de `MPI_Reduce()` (voir Fig.19)

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rang = comm.Get_rank()
nb_procs = comm.Get_size()

if rang == 0:
    valeur = np.array([1000], np.int32)
else:
    valeur = np.array([rang], np.int32)
somme = np.zeros(1, np.int32)

comm.Reduce(valeur, somme, op=MPI.SUM, root=0)

if rang == 0:
    print(f"Moi, processus 0, ma valeur de la somme globale est {somme[0]}")
```

```
> mpiexec -n 7 python -m mpi4py reduce.py

Moi, processus 0, ma valeur de la somme globale est 1021
```

Communications collectives

Réductions réparties avec diffusion du résultat : `MPI_Allreduce()`

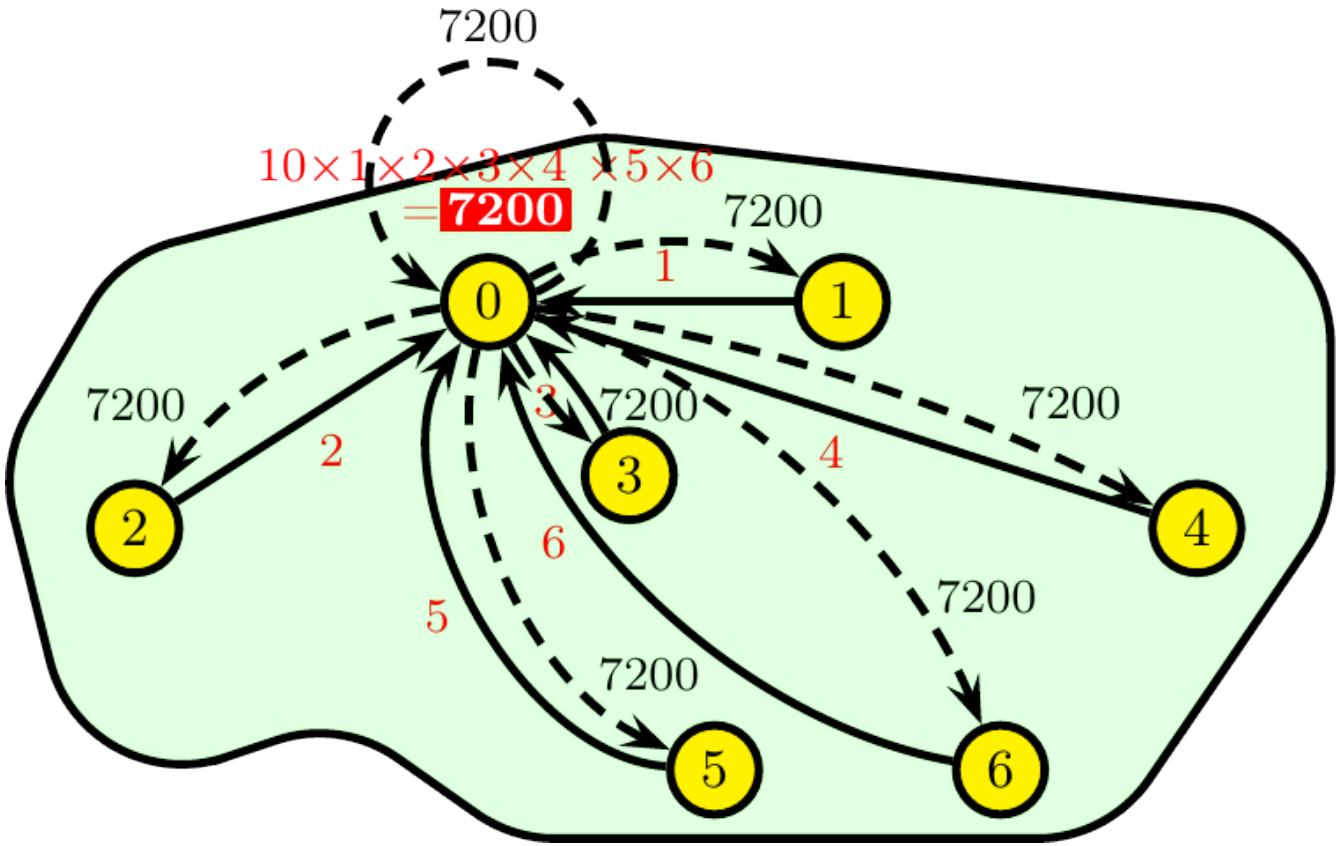


Figure 20 – Réduction répartie avec diffusion du résultat : `MPI_Allreduce` (utilisation de l'opérateur produit)

Communications collectives

Réductions réparties avec diffusion du résultat : `MPI_Allreduce()`

```
mpi4py.MPI.Comm.Allreduce([message_emis, longueur, type_message], message_recu, op=SUM)
# Retourne le resultat de la reduction
mpi4py.MPI.Comm.allreduce(sendobj, op=SUM)
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type_message, pour les processus du communicateur comm,
2. Ecrit le résultat à l'adresse message_recu pour tous les processus du communicateur comm.

Communications collectives

Exemple de `MPI_Allreduce()` (voir Fig.20)

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rang = comm.Get_rank()
6  nb_procs = comm.Get_size()
7
8  if rang == 0:
9      valeur = np.array([10], np.int32)
10 else:
11     valeur = np.array([rang], np.int32)
12 produit = np.zeros(1, np.int32)
13
14 comm.Allreduce(valeur, produit, op=MPI.PROD)
15
16 print(f"Moi, processus {rang}, ai recu la valeur du produit globale"
17       f" {produit[0]}")
```

Communications collectives

Exemple de `MPI_Allreduce()` (voir Fig.20) (suite)

```
> mpiexec -n 7 python -m mpi4py allreduce.py

Moi, processus 6, ai recu la valeur du produit global 7200
Moi, processus 2, ai recu la valeur du produit global 7200
Moi, processus 0, ai recu la valeur du produit global 7200
Moi, processus 4, ai recu la valeur du produit global 7200
Moi, processus 5, ai recu la valeur du produit global 7200
Moi, processus 3, ai recu la valeur du produit global 7200
Moi, processus 1, ai recu la valeur du produit global 7200
```

Compléments

- Le sous-programme `MPI_Scan()` permet d'effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du communicateur et lui-même. `MPI_Exscan()` est la version *exclusive* de `MPI_Scan()` , qui elle est inclusive.
- Les sous-programmes `MPI_Op_create()` et `MPI_Op_free()` permettent de définir des opérations de réduction personnelles.
- Pour toutes les opérations de reduction, le mot-clé `MPI.IN_PLACE` peut être utilisé pour que les données et résultats de l'opération soient stockés au même endroit (mais uniquement pour le ou les processus qui reçoivent les résultats). Exemple : `Allreduce(MPI.IN_PLACE,[message_emis_et_recu,...);`

Compléments

- De même que ce qui a été vu pour `MPI_Gatherv()` vis-à-vis de `MPI_Gather()`, les sous-programmes `MPI_Scatterv()`, `MPI_Allgatherv()` et `MPI_Alltoallv()` étendent `MPI_Scatter()`, `MPI_Allgather()` et `MPI_Alltoall()` au cas où le nombre d'éléments à diffuser ou collecter est différent suivant les processus.
- `MPI_Alltoallw()` est la version de `MPI_Alltoallv()` permettant de traiter des éléments hétérogènes (en exprimant les déplacements en octets et non en éléments).

T.P. MPI – Exercice 3 : Communications collectives et réductions

- Il s'agit de calculer π par intégration numérique $\pi = \int_0^1 \frac{4}{1+x^2} dx$.
- On utilise la méthode des rectangles (point milieu).
- La fonction à intégrer est $f(x) = \frac{4}{1+x^2}$.
- *nbbloc* est le nombre de rectangles.
- $largeur = \frac{1}{nbbloc}$ est le pas de discrétisation et la largeur de chaque rectangle.
- La version séquentielle est disponible dans le fichier `pi.py`.
- Il vous faut écrire la version parallélisée avec MPI dans ce fichier.

Modèles de communication

Modèles de communication

Modes d'envoi point à point

Mode	*Bloquant*	*Non bloquant*
Envoi standard	`MPI_Send()`	`MPI_Isend()`
Envoi synchrone	`MPI_Ssend()`	`MPI_Issend()`
Envoi *bufferisé*	`MPI_Bsend()`	`MPI_Ibsend()`
Réception	`MPI_Recv()`	`MPI_Irecv()`

Modèles de communication

Appels bloquants

- Un appel est bloquant si l'espace mémoire servant à la communication peut être réutilisé immédiatement après la sortie de l'appel.
- Les données envoyées peuvent être modifiées après l'appel bloquant.
- Les données reçues peuvent être lues après l'appel bloquant.

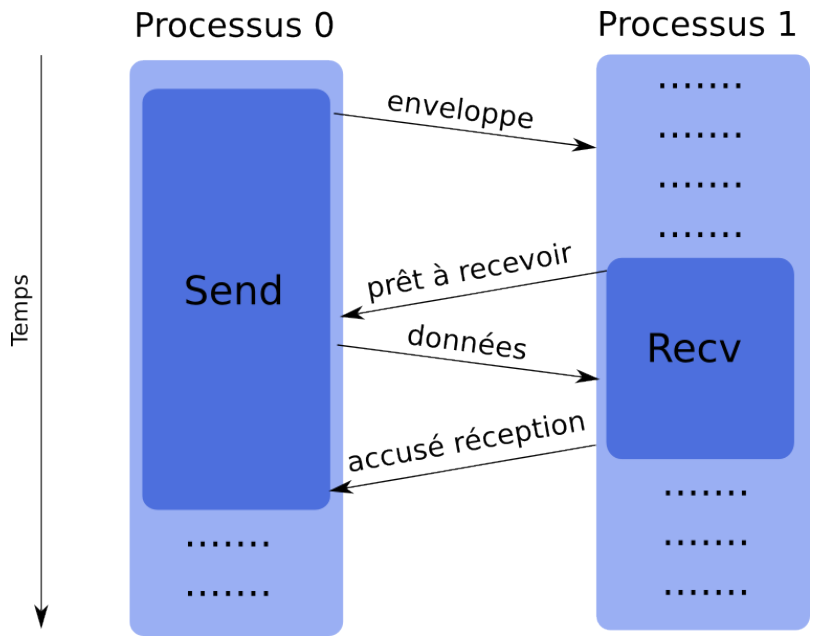
Modèles de communication

Envois synchrones

Un envoi synchrone implique une synchronisation entre les processus concernés. Un envoi ne pourra commencer que lorsque sa réception sera postée. Il ne peut y avoir de communication que si les deux processus sont prêts à communiquer.

Protocole de *rendez-vous*

Le protocole de *rendez-vous* est généralement celui employé pour les envois en mode synchrone (dépend de l'implémentation). L'accusé de réception est optionnel.



Modèles de communication

Interface de `MPI_Ssend()`

```
mpi4py.MPI.Comm.Ssend([valeurs, taille, type_message], dest, tag=0)
mpi4py.MPI.Comm.ssend(obj, dest, tag=0)
```

Avantages

- Consomment peu de ressources (pas de *buffer*)
- Rapides si le récepteur est prêt (pas de recopie dans un *buffer*)
- Connaissance de la réception grâce à la synchronisation

Inconvénients

- Temps d'attente si le récepteur n'est pas là/pas prêt
- Risques d' inter-blocage

Modèles de communication

Exemple d'inter-blocage

Dans l'exemple suivant, on a un inter-blocage, car on est en mode synchrone, les deux processus sont bloqués sur le `MPI_Ssend()` car ils attendent le `MPI_Recv()` de l'autre processus. Or ce `MPI_Recv()` ne pourra se faire qu'après le déblocage du `MPI_Ssend()`.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rang = comm.Get_rank()

etiquette = 110

num_proc = (rang + 1) % 2

tmp = np.array([rang + 1000], dtype=np.int32)
comm.Ssend(tmp, dest=num_proc, tag=etiquette)
valeur = np.zeros(1, dtype=np.int32)
comm.Recv(valeur, source=num_proc, tag=etiquette)

print(f"Moi, processus {rang} ai recu {valeur} du processus {num_proc}")
```

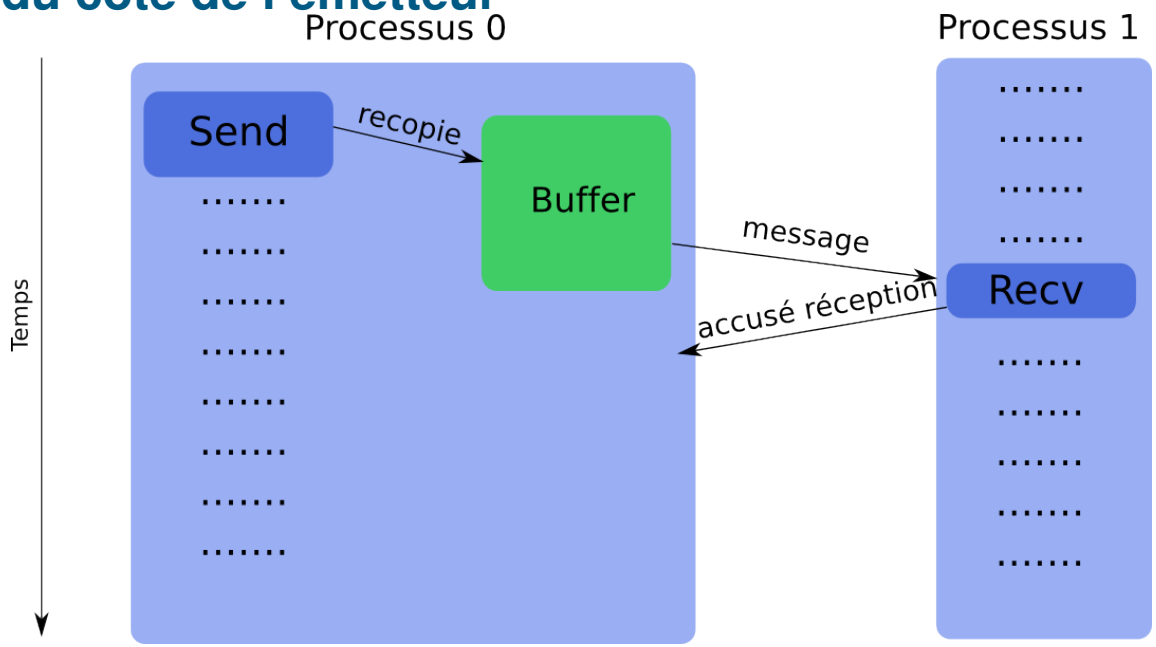
Modèles de communication

Envois *bufferisés*

Un envoi bufferisé implique la recopie des données dans un espace mémoire intermédiaire. Il n'y a alors pas de couplage entre les deux processus de la communication. La sortie de ce type d'envoi ne signifie donc pas que la réception a eu lieu.

Protocole avec *buffer* utilisateur du côté de l'émetteur

Dans cette approche, le *buffer* se trouve du côté de l'émetteur et est géré explicitement par l'application. Un *buffer* géré par MPI peut exister du côté du récepteur. De nombreuses variantes sont possibles. L'accusé de réception est optionnel.



Modèles de communication

Buffers

Les *buffers* doivent être gérés manuellement (avec appels à `MPI_Buffer_attach()` et `MPI_Buffer_detach()`). Ils doivent être alloués en tenant compte des surcoûts mémoire des messages (en ajoutant la constante `MPI_BSEND_OVERHEAD` pour chaque instance de message).

Interfaces

```
mpi4py.MPI.Attach_buffer(buf)
mpi4py.MPI.Detach_buffer()
mpi4py.MPI.Comm.Bsend([valeurs, taille, type_message], dest, tag=0)
mpi4py.MPI.Comm.bsend(obj, dest, tag=0)
```

Modèles de communication

Avantages du mode bufferisé

- Pas besoin d'attendre le récepteur (recopie dans un *buffer*)
- Pas de risque de blocage (*deadlocks*)

Inconvénients du mode bufferisé

- Consomment plus de ressources (occupation mémoire par les *buffers* avec risques de saturation)
- Les *buffers* d'envoi doivent être gérés manuellement (souvent délicat de choisir une taille adaptée)
- Un peu plus lent que les envois synchrones si le récepteur est prêt
- Pas de connaissance de la réception (découplage envoi-réception)
- Risque de gaspillage d'espace mémoire si les *buffers* sont trop sur-dimensionnés
- L'application plante si les *buffers* sont trop petits
- Il y a aussi souvent des *buffers* cachés géré par l'implémentation MPI du côté de l'expéditeur et/ou du récepteur (et consommant des ressources mémoires)

Modèles de communication

Absence d'inter-blocage

Dans l'exemple suivant, on a pas d'inter-blocage, car on est en mode bufferisé. Une fois la copie faite dans le *buffer*, l'appel `MPI_Bsend()` retourne et on passe à l'appel `MPI_Recv()`.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rang = comm.Get_rank()

etiquette = 110
nb_elt = 1
nb_msg = 1

taille = MPI.INT.Get_size()
# Calculer le surcout en nombre d'entiers
surcout = int(1 + (MPI.BSEND_OVERHEAD / taille))
taille_buf = nb_msg * (nb_elt + surcout)
buff = np.empty(taille_buf, dtype=np.int32)
MPI.Attach_buffer(buff)
# On suppose avoir exactement 2 processus
num_proc = (rang + 1) % 2
tmp = np.array([rang + 1000], dtype=np.int32)
comm.Bsend(tmp, dest=num_proc, tag=etiquette)
valeur = np.zeros(1, dtype=np.int32)
comm.Recv(valeur, source=num_proc, tag=etiquette)

print(f"Moi, processus {rang} ai recu {valeur} du processus {num_proc}")
MPI.Detach_buffer()
```

Modèles de communication

Envois standards

Un envoi standard se fait en appelant le sous-programme `MPI_Send()`. Dans la plupart des implémentations, ce mode passe d'un mode *bufferisé* (*eager*) à un mode synchrone lorsque la taille des messages croît.

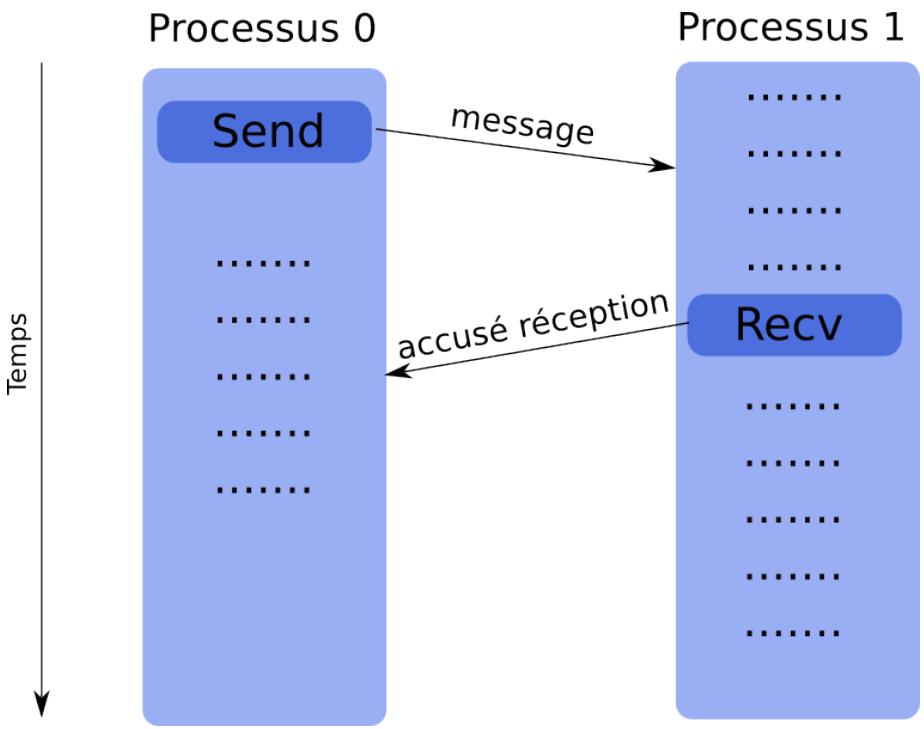
Interfaces

```
mpi4py.MPI.Comm.Send([valeurs, taille, type_message], dest, tag=0)
mpi4py.MPI.Comm.send(obj, dest, tag=0)
```

Modèles de communication

Protocole *eager*

Le protocole *eager* est souvent employé pour les envois en mode standard (`MPI_Send()`) pour les messages de petites tailles. Il peut aussi être utilisé pour les envois avec `MPI_Bsend()` avec des petits messages (dépend de l'implémentation) et en court-circuitant le *buffer* utilisateur du côté de l'émetteur. Dans cette approche, le *buffer* se trouve du côté du récepteur. L'accusé de réception est optionnel.



Modèles de communication

Avantages du mode standard

- Souvent le plus performant (choix du mode le plus adapté par le constructeur)

Inconvénients du mode standard

- Peu de contrôle sur le mode réellement utilisé (souvent accessible via des variables d'environnement)
- Risque de *deadlock* selon le mode réel
- Comportement pouvant varier selon l'architecture et la taille du problème

Modèles de communication

Nombre d'éléments reçus

```
mpi4py.MPI.Comm.Recv([message, longueur, type_message],
                     source=ANY_SOURCE, tag=ANY_TAG, status=None)
# Retourne l'objet recu
mpi4py.MPI.Comm.recv(source=ANY_SOURCE, tag=ANY_TAG, status=None)
```

- Dans l'appel à `MPI_Recv()`, l'argument longueur correspond dans la norme au nombre d'éléments dans le buffer message.
- Ce nombre doit être supérieur au nombre d'éléments à recevoir.
- Quand c'est possible, pour des raisons de lisibilité, il est conseillé de mettre le nombre d'éléments à recevoir.
- On peut connaitre le nombre d'éléments reçus avec `MPI_Get_count()` et à l'aide de l'argument statut retourné par l'appel à `MPI_Recv()`.

```
# Retourne le nombre d'element
mpi4py.MPI.Status.Get_count(datatype=BYTE)
```

Modèles de communication

Nombre d'éléments reçus

`MPI_Probe` permet de vérifier les messages entrants sans les recevoir.

```
mpi4py.MPI.Comm.Probe(source=ANY_SOURCE, tag=ANY_TAG, status=None)
```

Une utilisation courante de `MPI_Probe` consiste à allouer de l'espace pour un message avant de le recevoir.

```
comm.Probe(status=status)
msgsize = status.Get_count(MPI.INT)
buf = np.empty(msgsize, dtype=int32)
comm.Recv(buf, source=status.source, tag=status.tag)
```

Modèles de communication

Présentation

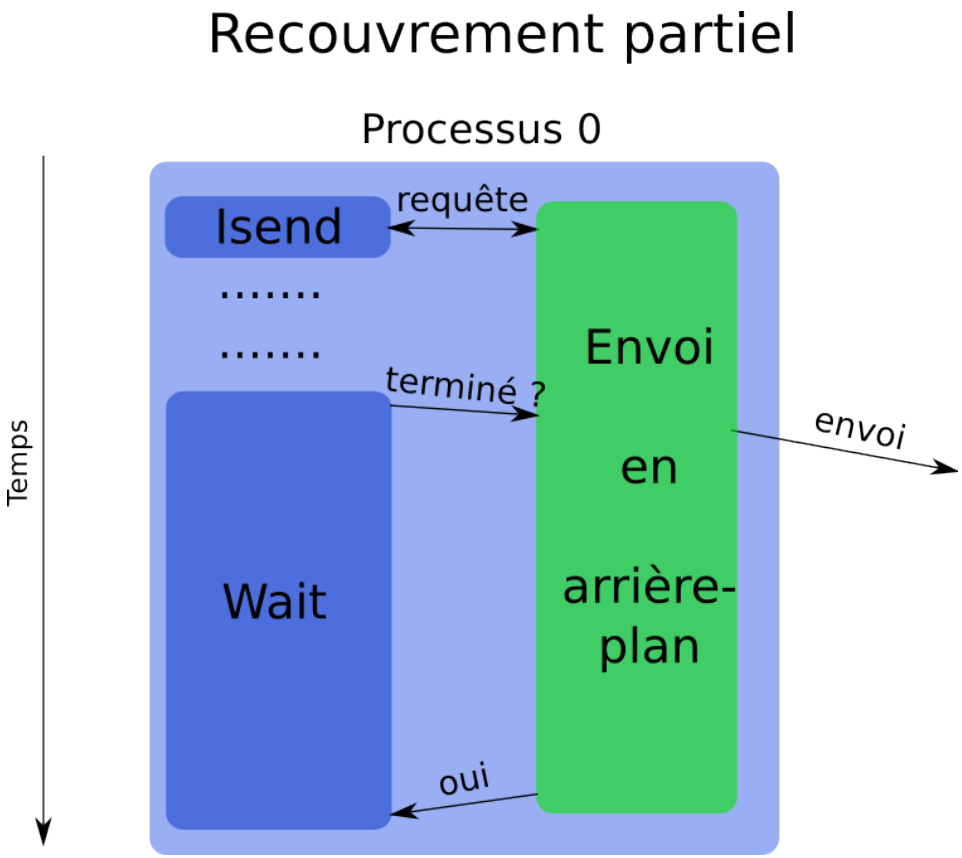
Le recouvrement des communications par des calculs est une méthode permettant de réaliser des opérations de communications en arrière-plan pendant que le programme continue de s'exécuter. Sur Jean Zay, la latence d'une communication inter-nœud est de 1μs soit 2500 cycles processeur.

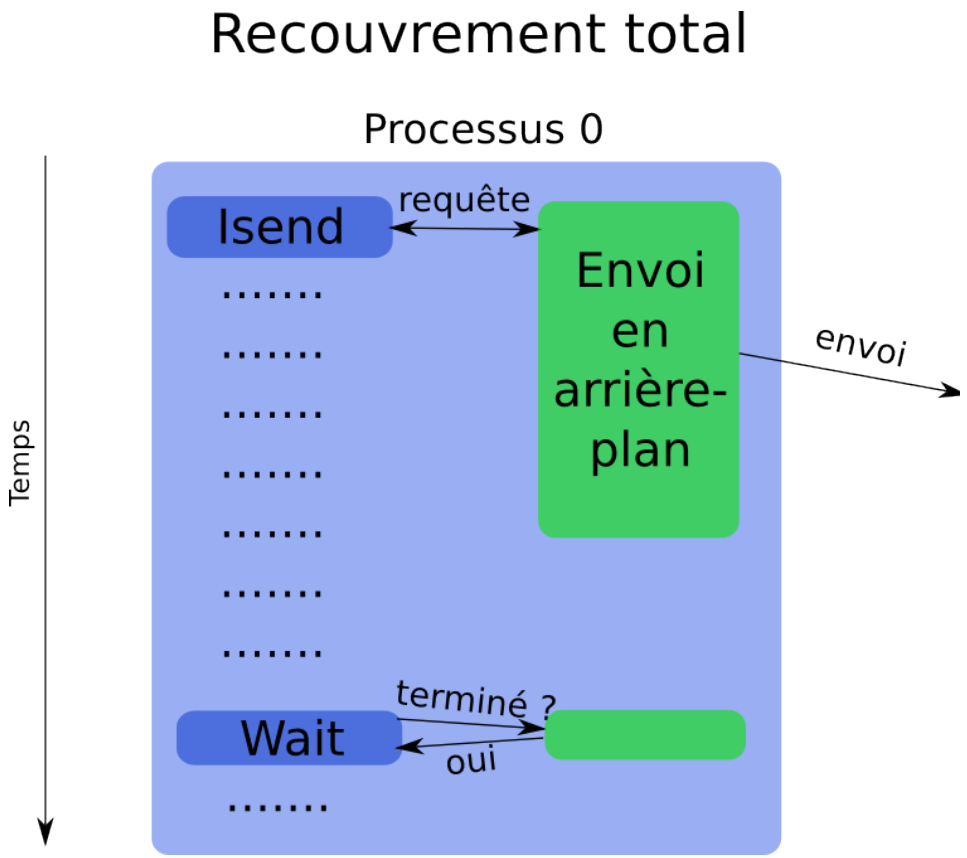
- Il est ainsi possible, si l'architecture matérielle et logicielle le permet, de masquer tout ou une partie des coûts de communications.
- Le recouvrement calculs-communications peut être vu comme un niveau supplémentaire de parallélisme.
- Cette approche s'utilise dans MPI par l'utilisation de sous-programmes non-bloquants (i.e. `MPI_Isend()`, `MPI_Irecv()` et `MPI_Wait()`).

Appels non bloquants

Un appel non bloquant rend la main très rapidement, mais n'autorise pas la réutilisation immédiate de l'espace mémoire utilisé dans la communication. Il est nécessaire de s'assurer que la communication est bien terminée (avec `MPI_Wait()` par exemple) avant de l'utiliser à nouveau.

Modèles de communication





Modèles de communication

Avantages des appels non bloquants

- Possibilité de masquer tout ou une partie des coûts des communications (si l'architecture le permet)
- Pas de risques de *deadlock*

Inconvénients des appels non bloquants

- Surcoûts plus importants (plusieurs appels pour un seul envoi ou réception, gestion des requêtes)
- Complexité plus élevée et maintenance plus compliquée
- Peu performant sur certaines machines (par exemple avec transfert commençant seulement à l'appel de `MPI_Wait()`)
- Risque de perte de performance sur les noyaux de calcul (par exemple gestion différenciée entre la zone proche de la frontière d'un domaine et la zone intérieure entraînant une moins bonne utilisation des caches mémoires)
- Limité aux communications point à point (a été étendu aux collectives dans MPI 3.0)

Modèles de communication

Interfaces

`MPI_Isend()` `MPI_Issend()` et `MPI_Ibsend()` pour les envois non bloquants

```
# Ces fonctions renvoient une instance de la classe request
mpi4py.MPI.Comm.Isend([valeurs, taille, type_message], dest=dest, tag=etiquette)
mpi4py.MPI.Comm.Issend([valeurs, taille, type_message], dest=dest, tag=etiquette)
mpi4py.MPI.Comm.Ibsend([valeurs, taille, type_message], dest=dest, tag=etiquette)
mpi4py.MPI.Comm.isend(obj, dest, tag=0)
mpi4py.MPI.Comm.issend(obj, dest, tag=0)
mpi4py.MPI.Comm.ibsend(obj, dest, tag=0)
```

`MPI_Irecv()` pour les réceptions non bloquantes.

```
# Ces fonctions renvoient une instance de la classe request
mpi4py.MPI.Comm.Irecv([valeurs, taille, type_message], source=ANY_SOURCE, tag=ANY_TAG)
mpi4py.MPI.Comm.irecv(buf=None, source=ANY_SOURCE, tag=ANY_TAG)
```

Modèles de communication

Interfaces

`MPI_Wait()` attend la fin d'une communication. `MPI_Test()` est la version non bloquante.

```
mpi4py.MPI.Request.Wait(status=None)
# Retourne l'objet recu (uniquement pour les requetes avec irecv)
mpi4py.MPI.Request.wait(status=None)
# Retourne un bool
mpi4py.MPI.Request.Test(status=None)
# Retourne un tuple avec un bool et l'objet recu eventuel
mpi4py.MPI.Request.test(status=None)
```

`MPI_Waitall()` attend la fin de toutes les communications. `MPI_Testall()` est la version non bloquante.

```
mpi4py.MPI.Request.Waitall(requests, statuses=None)
# Retourne la liste des objets recus
mpi4py.MPI.Request.waitall(requests, statuses=None)
# Retourne un bool indiquant si toutes les communications sont terminees.
mpi4py.MPI.Request.Testall(requests, statuses=None)
# Retourne un tuple avec un bool et une liste d'objets recus
mpi4py.MPI.Request.testall(requests, statuses=None)
```

Modèles de communication

Interfaces

`MPI_Waitany()` attend la fin d'une communication parmi plusieurs. `MPI_Testany()` est la version non bloquante.

```
# Retourne l'indice de la requete effectuee
mpi4py.MPI.Request.Waitany(requests, status=None)
# Retourne un tuple avec l'indice et l'objet recu eventuellement
mpi4py.MPI.Request.waitany(requests, status=None)
# Retourne un tuple avec un bool indiquant si une communication est effectues et
# l'indice de cette communication
mpi4py.MPI.Request.Testany(requests, status=None)
# Retourne un tuple avec un bool, l'indice et l'objet recu eventuel
mpi4py.MPI.Request.testany(requests, status=None)
```

`MPI_Waitsome()` attend la fin d'une ou plusieurs communications. `MPI_Testsome()` est la version non bloquante.

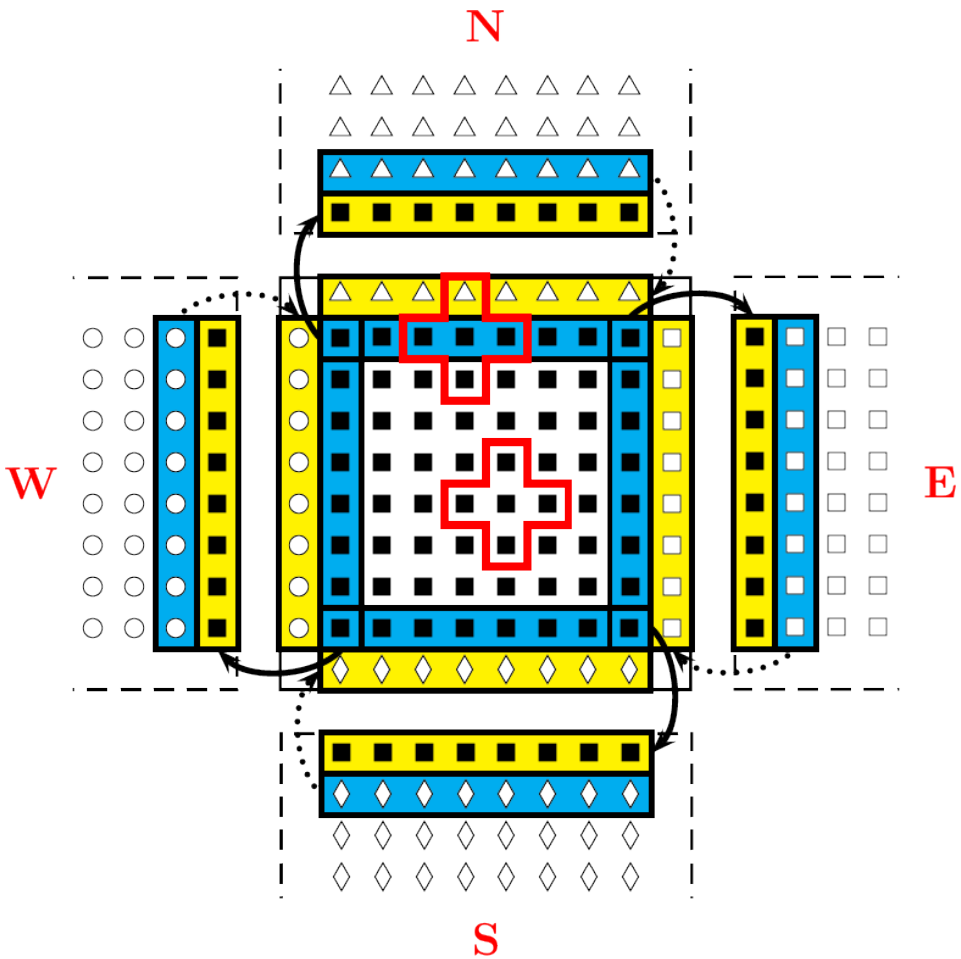
```
# Retourne les indices des requetes effectuees
mpi4py.MPI.Request.Waitsome(requests, statuses=None)
# Reourne un tuple avec une liste d'indice et une liste objet recu eventuellement
mpi4py.MPI.Request.waitsome(requests, statuses=None)
# Retourne une liste d'indice de communication ou None
mpi4py.MPI.Request.Testsome(requests, statuses=None)
# Retourne un tuple avec une liste d'indice et une liste d'objet recu eventuel
# ou un tuple de deux None
mpi4py.MPI.Request.testsome(requests, statuses=None)
```

Modèles de communication

Gestion des requêtes

- Après un appel aux fonctions bloquantes d'attente (`MPI_Wait()` , `MPI_Waitall()`,...), la requête vaut `MPI_REQUEST_NULL`.
- De même après un appel aux fonctions non bloquantes d'attente lorsque le *flag* est à vrai.
- Une attente avec une requête qui vaut `MPI_REQUEST_NULL` ne fait rien.

Modèles de communication



Modèles de communication

```
1  def debut_communication():
2    # Envoi au voisin N et reception du voisin S
3    req[0] = comm2d.Isend(sendbuf=[u[], 1, type_ligne], dest=voisin[N])
4    req[1] = comm2d.Irecv(recvbuf=[u[], 1, type_ligne], source=voisin[S])
5    # Envoi au voisin S et reception du voisin N
6    req[2] = comm2d.Isend(sendbuf=[u[], 1, type_ligne], dest=voisin[S])
7    req[3] = comm2d.Irecv(recvbuf=[u[], 1, type_ligne], source=voisin[N])
8    # Envoi au voisin W et reception du voisin E
9    req[4] = comm2d.Isend(sendbuf=[u[], 1, type_colonne], dest=voisin[W])
10   req[5] = comm2d.Irecv(recvbuf=[u[], 1, type_colonne], source=voisin[E])
11   # Envoi au voisin E et  reception du voisin W
12   req[6] = comm2d.Isend(sendbuf=[u[], 1, type_colonne], dest=voisin[E])
13   req[7] = comm2d.Irecv(recvbuf=[u[], 1, type_colonne], source=voisin[W])
14
15 def fin_communication():
16   MPI.Request.Waitall(req)
```

Modèles de communication

```
1  while not convergence and it< d.it_max :
2    it = it+1
3    # Swap u and u_new
4    u, u_new = u_new, u
5    debut_communication()
6    computation()
7    fin_communication()
8    bordure_comptutation()
9    diffnorm = global_error()
10   convergence = diffnorm < 2e-16
```

Modèles de communication

Niveau de recouvrement sur différentes machines

Machine	*Niveau*
Jean Zay (Intel MPI)	43%
Jean Zay (Intel MPI) I_MPI_ASYNC_PROGRESS=yes	95%

Mesures faites en recouvrant un noyau de calcul et un noyau de communication de mêmes durées.

Un recouvrement de 0% signifie que la durée totale d'exécution vaut 2x la durée d'un noyau de calcul (ou communication).
Un recouvrement de 100% signifie que la durée totale vaut 1x la durée d'un noyau de calcul (ou communication).

Modèles de communication

Communications collectives non bloquantes

- Version non bloquante des communications collectives
- Avec un I (immediate) devant : `MPI_Ireduce()` , `MPI_Ibcast()`, ...
- Attente avec les appels `MPI_Wait()`, `MPI_Test()` et leurs variantes
- Pas de correspondance bloquant et non bloquant
- Le *status* récupéré par `MPI_Wait()` contient une valeur non définie pour `MPI_SOURCE` et `MPI_TAG`
- Pour les processus d'un communicateur donné, l'ordre des appels doit être le même (comme en version bloquante)

```
# Retourne une requete
Ibarrier()
```

Modèles de communication

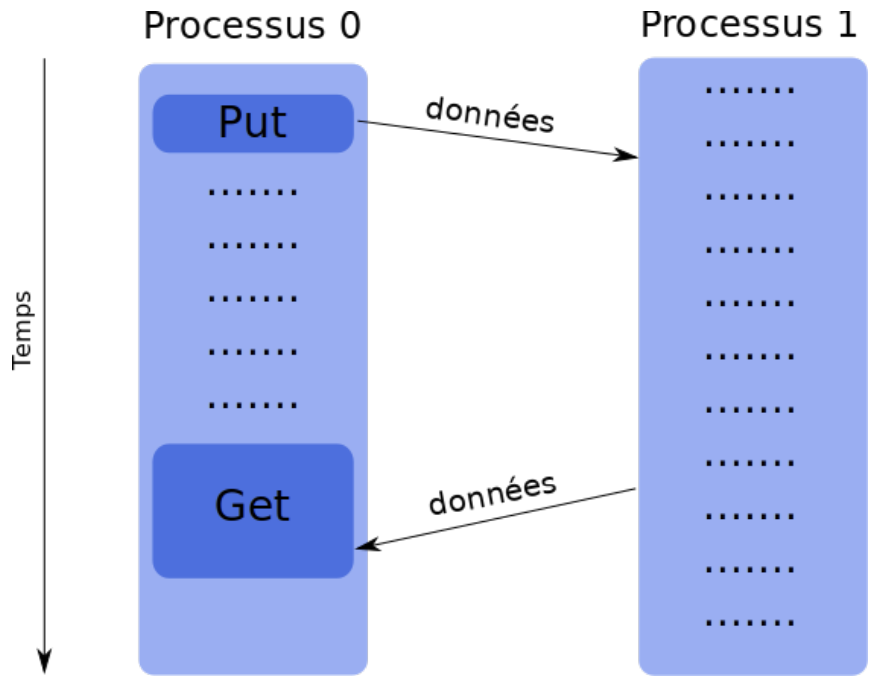
Exemple d'utilisation du `MPI_Ibarrier`
Comment gérer les communications quand on ne sait pas à chaque itération si nos voisins vont envoyer un message.

```
isAllFinish = False
isMySendFinish = False
reqs = []

# Envoi synchrone de tous les messages
for i in range(m):
  reqs.append(comm.Issend(sbuf[i], dest=dst[i], tag))

while not isAllFinish:
  # Verifie si un message est pret a etre recu
  if (comm.Iprobe(source=MPI.ANY_SOURCE, tag=tag, status=astat)):
    # Recoit le message
    comm.Recv(rbuf, source=astat.source, tag=tag)

  if not isMySendFinish:
    # Verifie si tous les Isend sont termines
    if MPI.Request.Testall(reqs):
      # Si c'est le cas, on demarre la ibarrier
      reqb = comm.Ibarrier(comm)
      isMySendFinish = True
  else:
    # Test si tout le monde a fait la ibarrier
    isAllFinish = reqb.Test()
```

Modèles de communication

Communications mémoire à mémoire (RMA)

Les communications mémoire à mémoire (ou RMA pour *Remote Memory Access* ou *one sided communications*) consistent à accéder en écriture ou en lecture à la mémoire d'un processus distant sans que ce dernier doive gérer cet accès explicitement. Le processus cible n'intervient donc pas lors du transfert.



Modèles de communication

RMA - Approche générale

- Création d'une fenêtre mémoire avec `MPI_Win_create()` pour autoriser les transferts RMA dans cette zone.
- Accès distants en lecture ou écriture en appellant `MPI_Put()`, `MPI_Get()`, `MPI_Accumulate()`, `MPI_Fetch_and_op()`, `MPI_Get_accumulate()` et `MPI_Compare_and_swap()` .
- Libération de la fenêtre mémoire avec `MPI_Win_free()`.

RMA - Méthodes de synchronisation

Pour s'assurer d'un fonctionnement correct, il est obligatoire de réaliser certaines synchronisations. 3 méthodes sont disponibles :

- Communication à cible active avec synchronisation globale (`MPI_Win_fence()`) ;
- Communication à cible active avec synchronisation par paire (`MPI_Win_start()` et `MPI_Win_complete()` pour le processus origine ; `MPI_Win_post()` et `MPI_Win_wait()` pour le processus cible) ;
- Communication à cible passive sans intervention de la cible (`MPI_Win_lock()` et `MPI_Win_unlock()`).

Modèles de communication

```
1  from mpi4py import MPI
2  import numpy as np
3
4  # Initialisation de la communication MPI
5  comm = MPI.COMM_WORLD
6  rang = comm.Get_rank()
7
8  # Parametres globaux
9  n = 4
10 m = 4
11
12 # Taille des donnees reelles en octets
13 taille_reel = MPI.DOUBLE.Get_size()
14
15 # Initialisation des buffers
16 if rang == 0:
17     tab = np.zeros(m)
18
19 win_local = np.zeros(n)
20 # Creation de la fenetre memoire partagee
21 dim_win = taille_reel * n
22 win = MPI.Win.Create(win_local, disp_unit=taille_reel, comm=comm)
```

Modèles de communication

```
23  if rang == 0:
24      for i in range(m):
25          tab[i] = i + 1
26  else:
27      for i in range(n):
28          win_local[i] = 0.0
29
30  win.Fence()
31
32  # Operations sur la fenetre memoire partagee
33  if rang == 0:
34      cible = 1
35      nb_elements = 2
36      deplacement = 1
37      win.Put([tab, nb_elements, MPI.DOUBLE], cible,
38              [deplacement, nb_elements, MPI.DOUBLE])
39
40  win.Fence()
41
42  asum = 0.0
43  if rang == 0:
44      for i in range(m - 1):
45          asum += tab[i]
46      tab[m - 1] = asum
47  else:
48      for i in range(n - 1):
49          asum += win_local[i]
50      win_local[n - 1] = asum
51
52  win.Fence()
53
54  if rang == 0:
55      nb_elements = 1
56      deplacement = m-1
57      win.Get([tab, nb_elements, MPI.DOUBLE], cible,
58              [deplacement, nb_elements, MPI.DOUBLE])
```

Modèles de communication

Avantages des RMA

- Permet de mettre en place plus efficacement certains algorithmes.
- Plus performant que les communications point à point sur certaines machines (utilisation de matériels spécialisés tels que moteur DMA, coprocesseur, mémoire spécialisée...).
- Possibilité pour l'implémentation de regrouper plusieurs opérations.

Inconvénients des RMA

- La gestion des synchronisations est délicate.
- Complexité et risques d'erreurs élevés.
- Moins performant que les communications point à point sur certaines machines.

Types de données dérivés

Types de données dérivés

Introduction

- Dans les communications, les données échangées sont typées : `MPI_INTEGER`, `MPI_REAL`, `MPI_COMPLEX`, etc .
- On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que `MPI_Type_contiguous()`, `MPI_Type_vector()`, `MPI_Type_indexed()` ou `MPI_Type_create_struct()`.
- Les types dérivés permettent notamment l'échange de données non contiguës ou non homogènes en mémoire et de limiter le nombre d'appels aux sous-programmes de communications.

Types de données dérivés

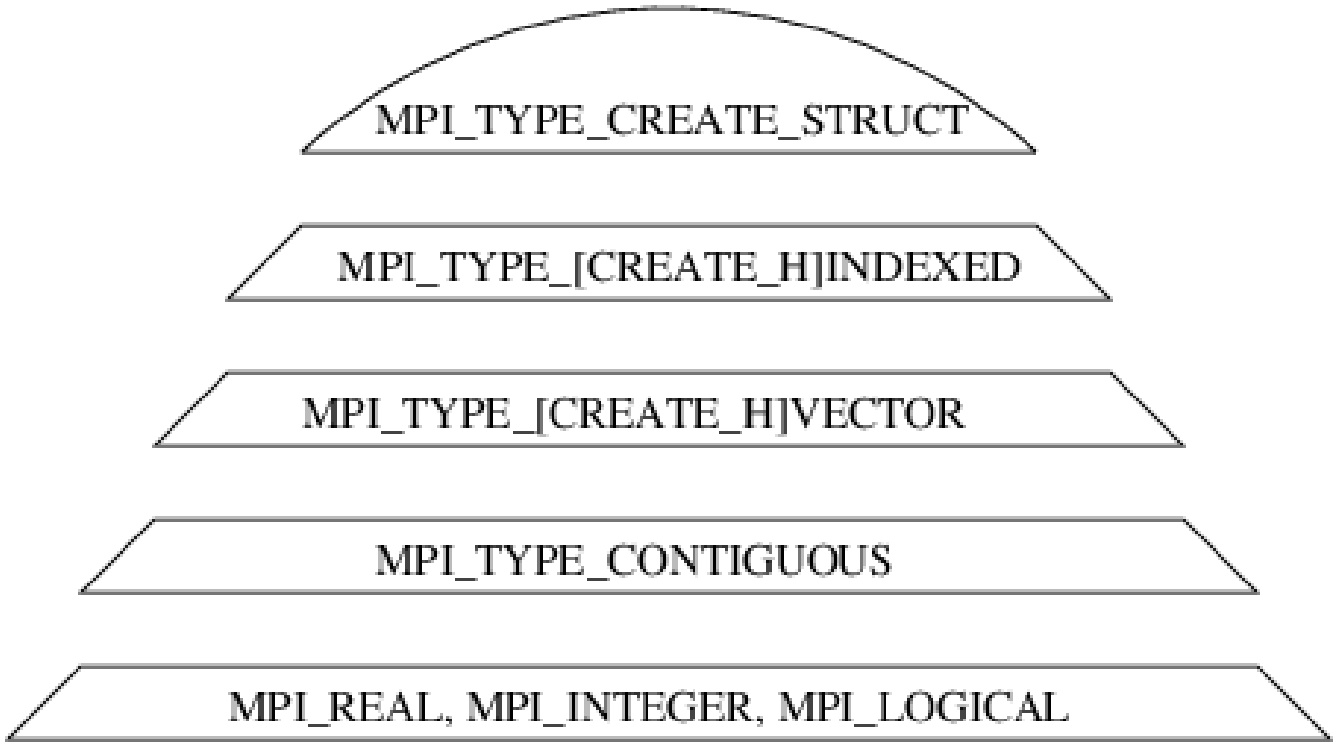


Figure 21 – Hiérarchie des constructeurs de type MPI

Types de données dérivés

Types contigus

- `MPI_Type_contiguous()` crée une structure de données à partir d'un ensemble homogène de type préexistant de données contiguës en mémoire.

1.	2.	3.	4.	5.
6.	7.	8.	9.	10.
11.	12.	13.	14.	15.
16.	17.	18.	19.	20.
21.	22.	23.	24.	25.
26.	27.	28.	29.	30.

```
nouveau_type = MPI.FLOAT.Create_contiguous(5)
```

Figure 22 – Sous-programme MPI_Type_contiguous

```
# Retourne un type
mpi4py.MPI.Datatype.Create_contiguous(count)
```

Types de données dérivés

Types avec un pas constant

- `MPI_Type_vector()` crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'éléments.

1.	2.	3.	4.	5.
6.	7.	8.	9.	10.
11.	12.	13.	14.	15.
16.	17.	18.	19.	20.
21.	22.	23.	24.	25.
26.	27.	28.	29.	30.

```
nouveau_type = MPI.FLOAT.Create_vector(6, 1, 5)
```

Figure 23 – Sous-programme MPI_Type_vector

```
# Retourne un type
mpi4py.MPI.Datatype.Create_vector(count, blocklength, stride)
```

Types de données dérivés

Types avec un pas constant

- `MPI_Type_create_hvector()` crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'octets.
- Cette instruction est utile lorsque le type générique n'est plus un type de base (`MPI_INT`, `MPI_FLOAT`,...) mais un type plus complexe construit à l'aide des sous-programmes MPI, parce qu'alors le pas ne peut plus être exprimé en nombre d'éléments du type générique.

```
# Retourne un type
mpi4py.MPI.Datatype.Create_hvector(count, blocklength, stride)
```

Types de données dérivés

Validation des types de données dérivés

- Les types dérivés doivent être validés avant d'être utilisés dans une communication. La validation s'effectue à l'aide du sous-programme `MPI_Type_commit()`.

```
mpi4py.MPI.Datatype.Commit()
```

- Si on souhaite réutiliser le même nom pour définir un autre type dérivé, on doit au préalable le libérer en utilisant le sous-programme `MPI_Type_free()`.

```
mpi4py.MPI.Datatype.Free()
```

Types de données dérivés

```
1  from mpi4py import MPI
2  import numpy as np
3
4  # Initialisation de la communication MPI
5  comm = MPI.COMM_WORLD
6  rang = comm.Get_rank()
7
8  # Definitions des parametres
9  nb_lignes = 6
10 nb_colonnes = 5
11 etiquette = 100
12
13 # Initialisation de la matrice sur chaque processus
14 a = np.full((nb_lignes, nb_colonnes), rang,dtype=np.float32)
15
16 # Definition du type de donnees type_ligne
17 type_ligne = MPI.FLOAT.Create_contiguous(nb_colonnes)
18 type_ligne.Commit()
```

Types de données dérivés

```
19  if rang == 0:
20      # Envoi de la premiere ligne
21      comm.Send([a, 1, type_ligne], 1, etiquette)
22  else:
23      # Reception dans la derniere ligne
24      comm.Recv([a[nb_lignes-1, 0:], nb_colonnes, MPI.FLOAT], 0, etiquette)
25
26  # Liberation du type de donnees
27  type_ligne.Free()
```

Types de données dérivés

```
1  from mpi4py import MPI
2  import numpy as np
3
4  # Initialisation environnement MPI
5  comm = MPI.COMM_WORLD
6  rang = comm.Get_rank()
7
8  # Definitions des parametres
9  nb_lignes = 6
10 nb_colonnes = 5
11 etiquette = 100
12
13 # Initialisation de la matrice sur chaque processus
14 a = np.full((nb_lignes, nb_colonnes), rang, dtype=np.float32)
15
16 # Definition du type de donnees type_colonne
17 type_colonne = MPI.FLOAT.Create_vector(nb_lignes, 1, nb_colonnes)
18 type_colonne.Commit()
```

Types de données dérivés

```
19  if rang == 0:
20      # Envoi
21      comm.Send([a[0, 0:], nb_lignes, MPI.FLOAT], 1, etiquette)
22  else:
23      # Reception dans l'avant-derniere colonne
24      comm.Recv([a[0, nb_colonnes-2:], 1, type_colonne], 0, etiquette)
25
26  # Liberation du type de donnees
27  type_colonne.Free()
```

Types de données dérivés

```
1  from mpi4py import MPI
2  import numpy as np
3
4  # Initialisation de l'environnement MPI
5  comm = MPI.COMM_WORLD
6  rang = comm.Get_rank()
7
8  # Definitions des parametres
9  nb_lignes = 6
10 nb_colonnes = 5
11 etiquette = 100
12 nb_lignes_bloc = 3
13 nb_colonnes_bloc = 2
14
15 # Initialisation de la matrice sur chaque processus
16 a = np.full((nb_lignes, nb_colonnes), rang, dtype=np.float32)
17
18 # Definition du type de donnees type_bloc
19 type_bloc = MPI.FLOAT.Create_vector(nb_lignes_bloc, nb_colonnes_bloc,
20                                     nb_colonnes)
21 type_bloc.Commit()
```

Types de données dérivés

```
22  if rang == 0:
23      # Envoi d'un bloc
24      comm.Send([a, 1, type_bloc], 1, etiquette)
25  else:
26      # Reception du bloc
27      comm.Recv([a[nb_lignes-3, nb_colonnes-2:], 1, type_bloc], 0, etiquette)
28
29  # Liberation du type de donnees
30  type_bloc.Free()
```

Types de données dérivés

Types homogènes à pas variable

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

Types de données dérivés

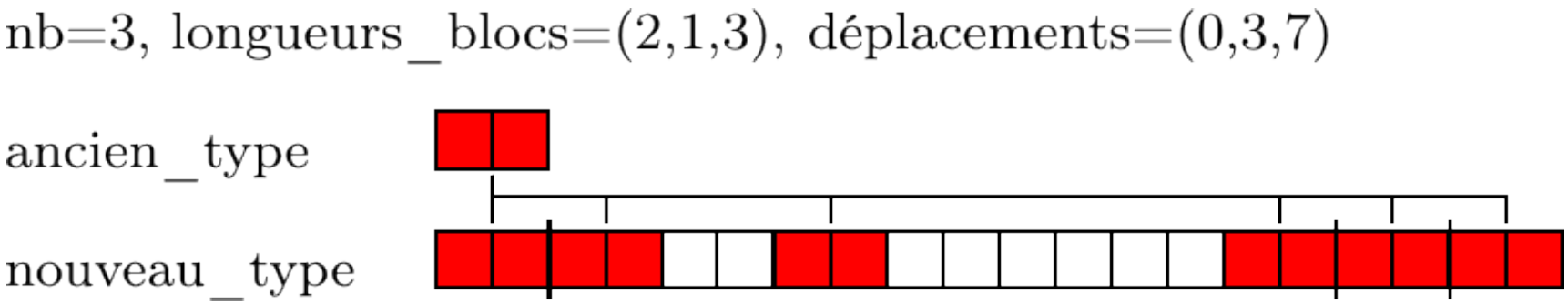


Figure 24 – Le constructeur `MPI_Type_indexed`

```
# Retourne un type
mpi4py.MPI.Datatype.Create_indexed(blocklengths, displacements)
```

Types de données dérivés

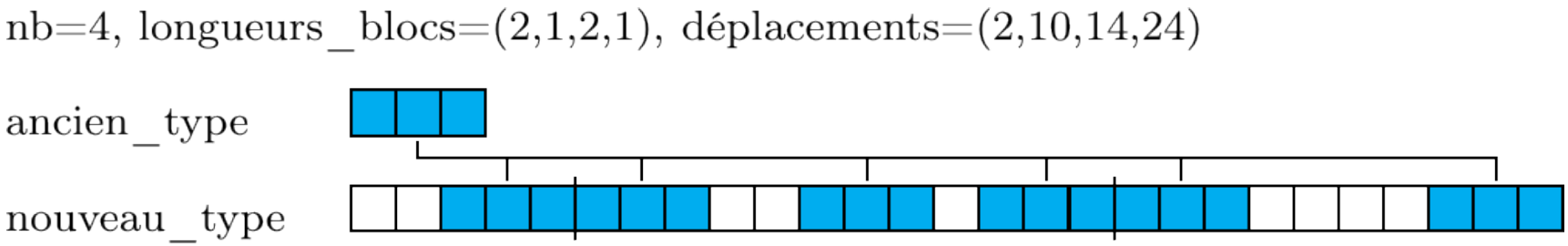


Figure 25 – Le constructeur `MPI_Type_create_hindexed`

```
# Retourne un type
mpi4py.MPI.Datatype.Create_hindexed(blocklengths, displacements)
```

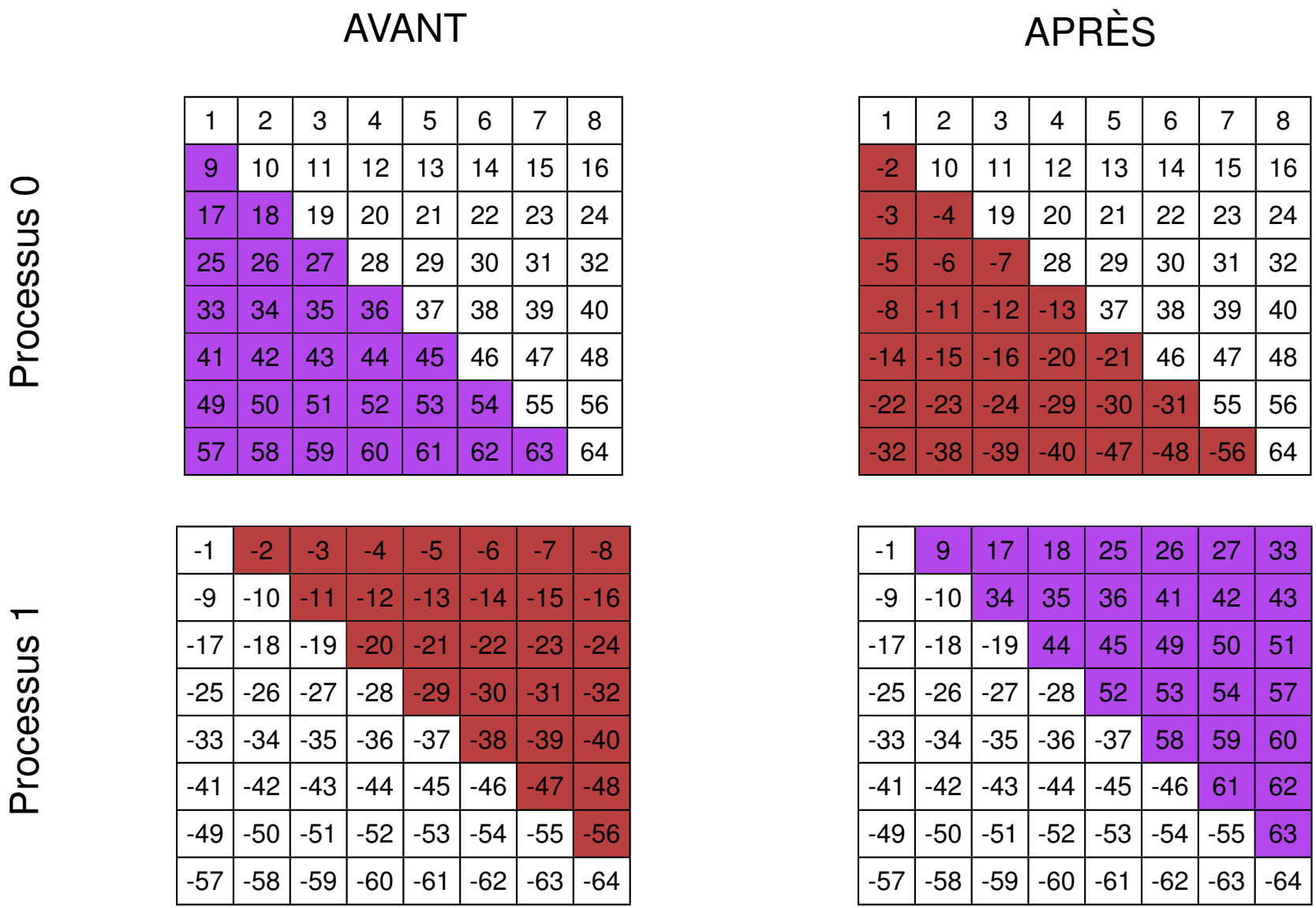
Types de données dérivés

Exemple : matrice triangulaire

Dans l'exemple suivant, chacun des deux processus :

1. initialise sa matrice (nombres croissants positifs sur le processus 0 et négatifs décroissants sur le processus 1) ;
2. construit son type de données (*datatype*) : matrice triangulaire (supérieure pour le processus 0 et inférieure pour le processus 1) ;
3. envoie sa matrice triangulaire à l'autre et reçoit une matrice triangulaire qu'il stocke à la place de celle qu'il a envoyée. Cela se fait avec l'instruction `MPI_Sendrecv_replace()` ;
4. libère ses ressources et quitte MPI.

Types de données dérivés



Types de données dérivés

```python
from mpi4py import MPI
import numpy as np

# Parametres
n = 8
etiquette = 100

# Creation de la matrice a
a = np.zeros((n, n), dtype=np.float32)

# Environnement MPI
comm = MPI.COMM_WORLD
rang = comm.Get_rank()

sign = 1
if rang == 1:
    sign = -1

# Initialisation de la matrice sur chaque processus
for i in range(n):
    for j in range(n):
        a[i, j] = sign * (1 + i * n + j)

# Creation du type matrice triangulaire inf pour le processus 0
# et du type matrice triangulaire sup pour le processus 1
longueurs_blocs = [i if rang == 0 else n - i - 1 for i in range(n)]
deplacements = [n * i if rang == 0 else (n + 1) * i + 1 for i in range(n)]

# Definition du type matrice triangulaire
type_triangle = MPI.FLOAT.Create_indexed(longueurs_blocs, deplacements)
type_triangle.Commit()

# Permutation des matrices triangulaires inferieure et superieure
comm.Sendrecv_replace([a, 1, type_triangle],
                      source=(rang + 1) % 2, sendtag=etiquette,
                      dest=(rang + 1) % 2, recvtag=etiquette)

# Liberation du type triangle
type_triangle.Free()
```

Types de données dérivés

Taille des types de données

- `MPI_Type_size()` retourne le nombre d'octets nécessaire pour envoyer un type de données. Cette valeur ne tient pas compte des *trous* présents dans le type de données.

```
# Retourne la taille en octets d'un type de donnees
mpi4py.MPI.Datatype.Get_size()
```

- L'étendue d'un type est l'espace mémoire occupé par le type (en octets). Cette valeur intervient directement pour calculer la position du prochain élément en mémoire (c'est-à-dire le pas entre des éléments successifs).

```
# Retourne un tuple avec la borne inferieur et l'etendue
mpi4py.MPI.Datatype.Get_extent()
```

Types de données dérivés

Exemple 1 : `MPI_Type_indexed(2,{2,1},{1,4},MPI_INT,&type)`

Type dérivé :

Deux éléments successifs :

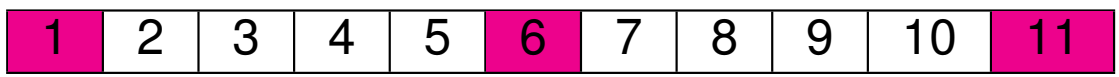
`taille = 12 (3 entiers); borne_inf = 4 (1 entier); etendue = 16 (4 entiers)`

Exemple 2 : `MPI_Type_vector(3,1,nb_colonnes,MPI_INT,&type_demi_colonne)`

Vue 2D :

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30

Vue 1D :



`taille = 12 (3 entiers); borne_inf = 0; etendue = 44 (11 entiers)`

Types de données dérivés

Changer l'étendue

- L'étendue est un paramètre du type de données. Par défaut, c'est généralement l'intervalle en mémoire entre le premier et le dernier composant du type (bornes incluses et en tenant compte de l'alignement mémoire). On peut modifier l'étendue d'un type pour créer un nouveau type adapté du précédent avec `MPI_Type_create_resized()`. Cela permet de choisir le pas entre des éléments successifs.

```
# Retourne un type
mpi4py.MPI.Datatype.Create_resized(lb, extent)
```

Types de données dérivés

```python
from mpi4py import MPI
import numpy as np

# Parametres
nb_lignes = 6
nb_colonnes = 5
etiquette = 100

# Determination du rang du processus et de son signe
comm = MPI.COMM_WORLD
rang = comm.Get_rank()

sign = 1
if rang == 1:
    sign = -1

# Initialisation de la matrice sur chaque processus
a = np.zeros((nb_lignes, nb_colonnes), dtype=np.int32)
for i in range(nb_lignes):
    for j in range(nb_colonnes):
        a[i, j] = sign * (1 + nb_colonnes * i + j)

# Construction du type derive type_demi_colonne1
taille_demi_colonne = nb_lignes // 2
type_demi_colonne1 = MPI.INT.Create_vector(taille_demi_colonne, 1, nb_colonnes)
type_demi_colonne1.Commit()

# Connaissance de la taille du type de base MPI_INT
taille_integer = MPI.INT.Get_size()

# Informations sur le type type_demi_colonne1
borne_inf1, etendue1 = type_demi_colonne1.Get_extent()
if rang == 0:
    print(f"Type_demi_colonne1: borne_inf={borne_inf1}, etendue={etendue1}")
```

Types de données dérivés

```
35  # Construction du type type_demi_colonne2
36  borne_inf2 = 0
37  etendue2 = taille_integer
38  type_demi_colonne2 = type_demi_colonne1.Create_resized(borne_inf2, etendue2)
39  type_demi_colonne2.Commit()
40
41  # Informations sur le type type_demi_colonne2
42  borne_inf2, etendue2 = type_demi_colonne2.Get_extent()
43  if rang == 0:
44      print(f"Type_demi_colonne2: borne_inf={borne_inf2}, etendue={etendue2}")
45
46  # Envoi de la matrice a au processus 1 avec le type demi_colonne2
47  if rang == 0:
48      comm.Send([a, 2, type_demi_colonne2], dest=1, tag=etiquette)
49  else:
50      # Reception pour le processus 1 dans la matrice a
51      comm.Recv([a[nb_lignes-2, 0:], 6, MPI.INT], source=0, tag=etiquette)
```

```
> mpiexec -n 2 python -m mpi4py demi_ligne.py
 type_demi_colonne1: borne_inf=0, etendue=44
 type_demi_colonne2: borne_inf=0, etendue=4
```

```
Matrice A sur le processus 1
  -1  -2  -3  -4  -5
  -6  -7  -8  -9 -10
 -11 -12 -13 -14 -15
 -16 -17 -18 -19 -20
   1   6  11   2   7
  12 -27 -28 -29 -30
```

Types de données dérivés

Conclusion

- Les types dérivés MPI sont de puissants mécanismes portables de description de données.
- Ils permettent, lorsqu'ils sont associés à des instructions comme `MPI_Sendrecv()`, de simplifier l'écriture de sous-programmes d'échanges interprocessus.
- L'association des types dérivés et des topologies (décrites dans l'un des prochains chapitres) fait de MPI l'outil idéal pour tous les problèmes de décomposition de domaine avec des maillages réguliers ou irréguliers.

Types de données dérivés

Memento

Sous-routines	longueurs_blocs	pas	types_anciens
MPI_Type_Contiguous()	constant*	constant*	constant
MPI_Type_[Create_H]Vector()	constant	constant	constant
MPI_Type_[Create_H]Indexed()	*variable*	*variable*	constant
MPI_Type_Create_Struct()	*variable*	*variable*	*variable*

(*) paramètre caché, égal à 1

Travaux pratiques MPI – Exercice 4 : Transposée d'une matrice

- Dans cet exercice, on se propose de se familiariser avec les types dérivés
- On se donne une matrice A de 4 lignes et 5 colonnes sur le processus 0
- Il s'agit pour le processus 0 d'envoyer au processus 1 cette matrice mais d'en faire automatiquement la transposition au cours de l'envoi

1.	2.	3.	4.	5.
6.	7.	8.	9.	10.
11.	12.	13.	14.	15.
16.	17.	18.	19.	20.

Processus 0

1.	6.	11.	16.
2.	7.	12.	17.
3.	8.	13.	18.
4.	9.	14.	19.
5.	10.	15.	20.

Processus 1

- Pour ce faire, on va devoir se construire deux types dérivés, un type type_colonne et un type type_transpose

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

- Communications collectives et réductions : produit de matrices $C = A \times B$
  - On se limite au cas de matrices carrées dont l'ordre est un multiple du nombre de processus
  - Les matrices A et B sont sur le processus 0. Celui-ci distribue une tranche horizontale de la matrice A et une tranche verticale de la matrice B à chacun des processus. Chacun calcule alors un bloc diagonal de la matrice résultante C.
  - Pour calculer les blocs non diagonaux, chaque processus doit envoyer aux autres processus la tranche de A qu'il possède
  - Après quoi le processus 0 peut collecter les résultats et vérifier les résultats

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

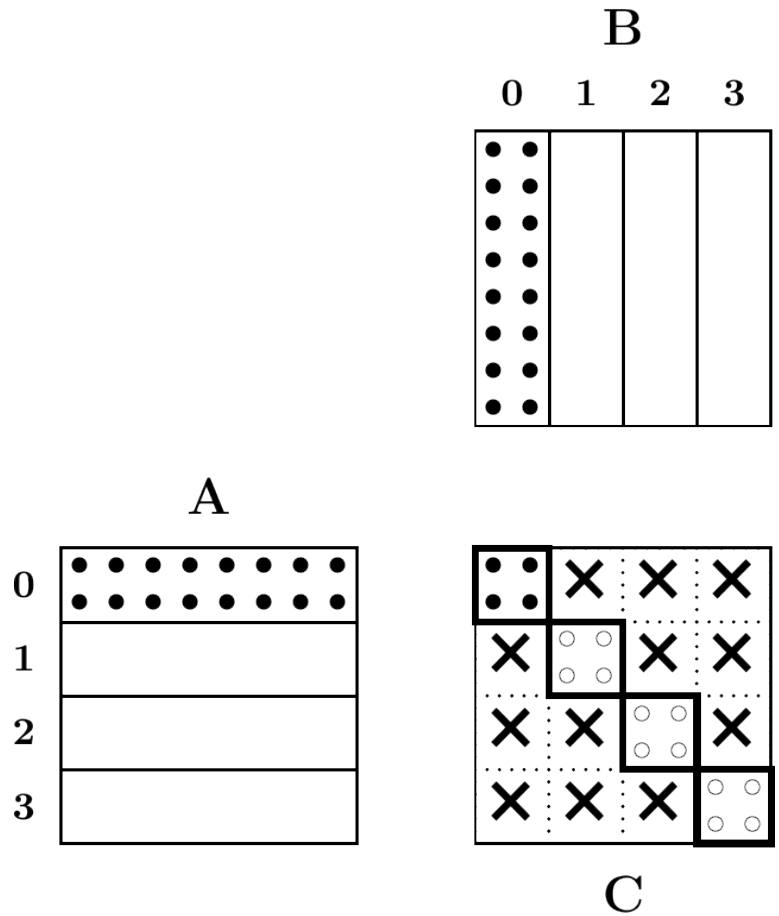


Figure 26 – Produit parallèle de matrices

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

- Toutefois, l'algorithme qui peut sembler le plus immédiat, et qui est le plus simple à programmer, consistant à faire envoyer par chaque processus sa tranche de la matrice A à chacun des autres, n'est pas performant parce que le schéma de communication n'est pas du tout équilibré. C'est très facile à voir en faisant des mesures de performances et en représentant graphiquement les traces collectées.

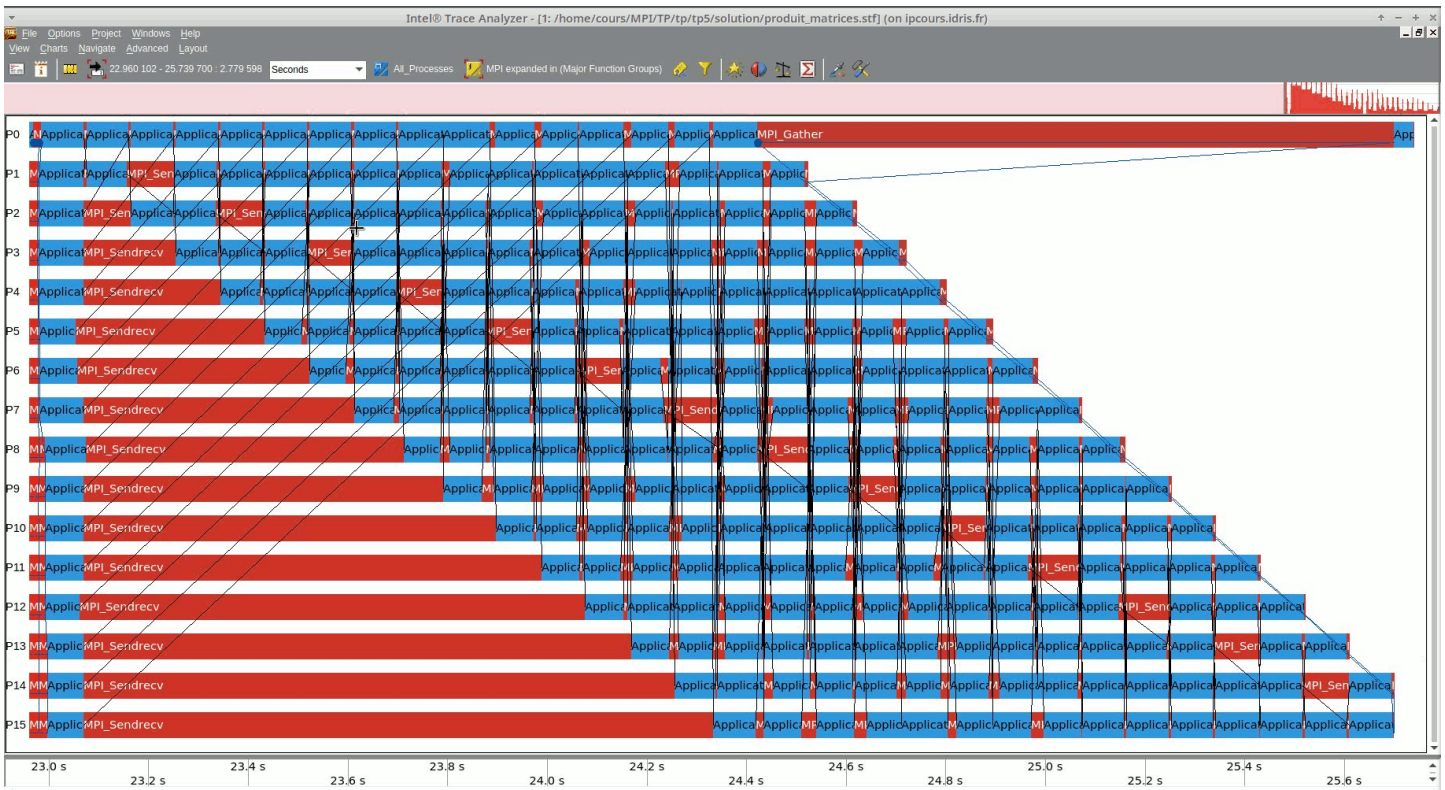


Figure 27 – Produit parallèle de matrices sur 16 processus, pour une taille de matrice de 1024 (premier algorithme)

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

- Mais en changeant l'algorithme pour faire *glisser* le contenu des tranches de processus à processus, on peut obtenir un équilibre parfait des calculs et des communications, et gagner ainsi un facteur 2.

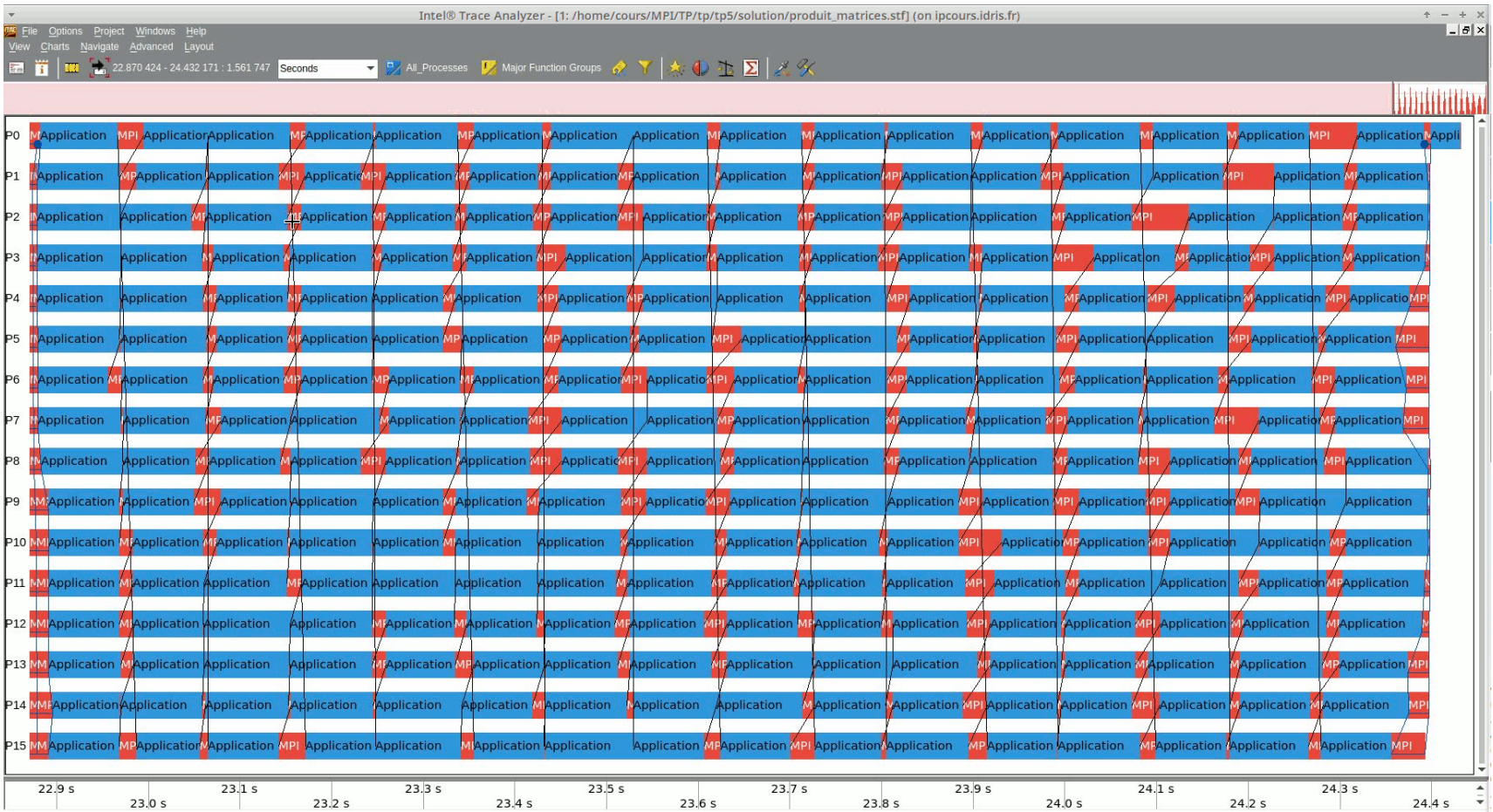


Figure 28 – Produit parallèle de matrices sur 16 processus, pour une taille de matrice de 1024 (second algorithme)

Communicateurs

Communicateurs

Introduction

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

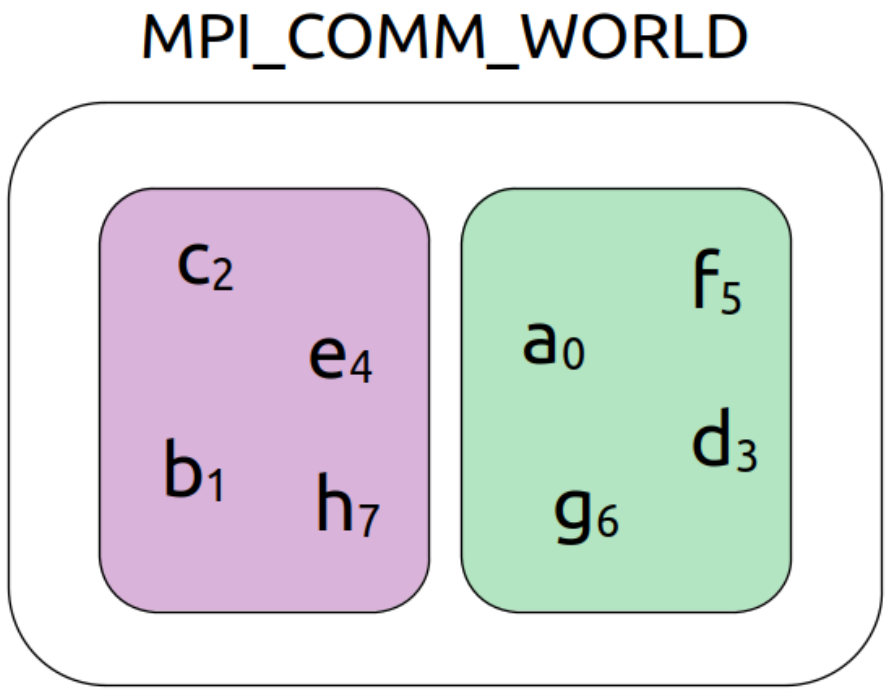


Figure 29 – Partitionnement d'un communicateur

Communicateurs

Exemple

Par exemple, on veut diffuser un message collectif aux processus de rang pair et un autre aux processus de rang impair.

- Boucler sur des *send/recv* peut être très pénalisant surtout si le nombre de processus est élevé. De plus un test serait obligatoire dans la boucle pour savoir si le rang du processus auquel le processus émetteur doit envoyer le message est pair ou impair.
- Une solution est de créer un communicateur regroupant les processus pairs et un autre regroupant les processus impairs, puis d'initier les communications collectives à l'intérieur de ces groupes.

Communicateurs

Communicateur par défaut

- On ne peut créer un communicateur qu'à partir d'un autre communicateur. Le premier sera créé à partir de `MPI_COMM_WORLD`.
- En effet, suite à l'appel à `MPI_Init()`, un communicateur est créé pour toute la durée d'exécution du programme.
- Son identificateur `MPI_COMM_WORLD` est une variable définie dans les fichiers d'en-tête.
- Il ne peut être détruit que via l'appel à `MPI_Finalize()`
- Par défaut, il fixe donc la portée des communications point à point et collectives à tous les processus de l'application

Communicateurs

Groupes et communicateurs

- Un communicateur est constitué :
  - d'un groupe, qui est un ensemble ordonné de processus ;
  - d'un contexte de communication mis en place à l'appel du sous-programme de construction du communicateur, qui permet de délimiter l'espace de communication.
- Les contextes de communication sont gérés par MPI (le programmeur n'a aucune action sur eux) : c'est un attribut « caché »
- Dans la bibliothèque MPI, divers sous-programmes existent pour construire des communicateurs : `MPI_Comm_create()`, `MPI_Comm_dup()`, `MPI_Comm_split()`
- Les constructeurs de communicateurs sont des opérateurs collectifs (qui engendrent des communications entre les processus)
- Les communicateurs que le programmeur crée peuvent être gérés dynamiquement et, de même qu'il est possible d'en créer, il est possible d'en détruire en utilisant le sous-programme `MPI_Comm_free()`

Communicateurs

Partitionnement d'un communicateur

Pour résoudre le problème de l'exemple, nous allons :

- partitionner le communicateur en processus de rang pair et d'autre part en processus de rang impair ;
- ne diffuser un message collectif qu'aux processus de rang pair et un autre qu'aux processus de rang impair.

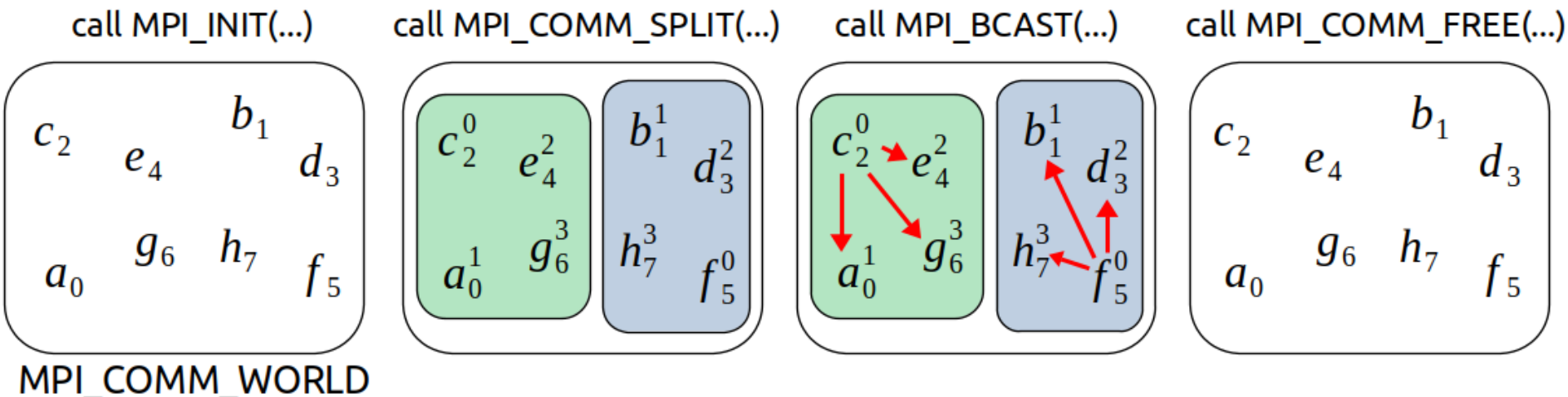


Figure 30 – Création/destruction d'un communicateur

Communicateurs

Partitionnement d'un communicateur avec `MPI_Comm_split()`

Le sous-programme `MPI_Comm_split()` permet de :

- partitionner un communicateur donné en autant de communicateurs que l'on veut
- donner le même nom à tous ces communicateurs : il aura la valeur du communicateur dans lequel se trouve le processus courant
- Méthode :
  1. définir une valeur couleur associant à chaque processus le numéro du communicateur auquel il appartiendra
  2. définir une valeur clef permettant de numéroter les processus dans chaque communicateur
  3. créer la partition où chaque communicateur s'appelle nouveau_comm

```
# Retourne un communicateur
mpi4py.MPI.Comm.Split(color=0, key=0)
```

Un processus qui s'attribue une couleur égale à la valeur `MPI_UNDEFINED` aura pour nouveau_com le communicateur invalide `MPI_COMM_NULL`.

Communicateurs

Exemple

Voyons comment procéder pour construire le communicateur qui va subdiviser l'espace de communication entre processus de rangs pairs et impairs, via le constructeur `MPI_Comm_split()`.

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

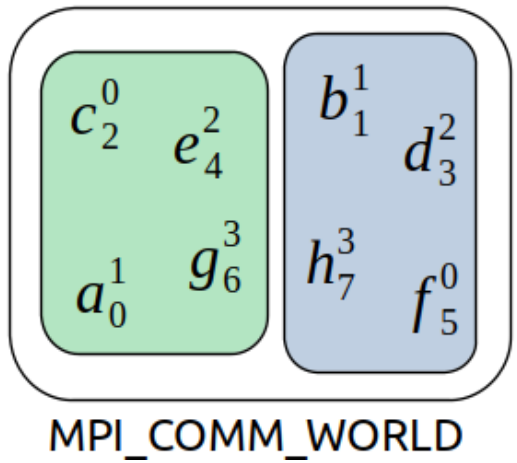
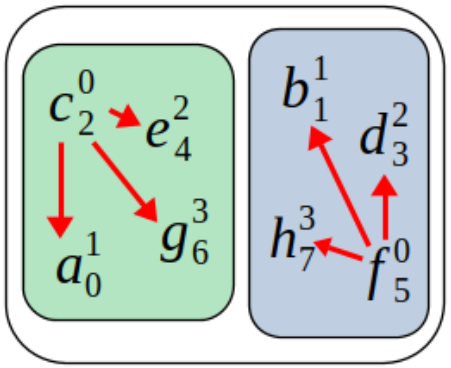


Figure 31 – Construction du communicateur `CommPairsImpairs` avec `MPI_Comm_split()`

Communicateurs

```
1  from mpi4py import MPI
2  import numpy as np
3
4  # Initialisation de la matrice et des parametres
5  m = 16
6  a = np.zeros(m, dtype=np.float32)
7
8  # Determination du rang du processus dans le monde
9  comm = MPI.COMM_WORLD
10 rang_dans_monde = comm.Get_rank()
11
12 # Initialisation de la matrice a
13 for i in range(m):
14     a[i] = 0.
15 if rang_dans_monde == 2:
16     for i in range(m):
17         a[i] = 2.
18 if rang_dans_monde == 5:
19     for i in range(m):
20         a[i] = 5.
21
22 # Determination de la cle pour la separation en pairs et impairs
23 clef = rang_dans_monde
24 if (rang_dans_monde == 2) or (rang_dans_monde == 5):
25     clef = -1
26
27 # Creation des communicateurs pair et impair
28 commPairsImpairs = comm.Split(rang_dans_monde % 2, clef)
29
30 # Diffusion du message par le processus 0 de chaque communicateur
31 # aux processus de son groupe
32 commPairsImpairs.Bcast([a, m, MPI.FLOAT], root=0)
33
34 # Destruction des communicateurs
35 commPairsImpairs.Free()
```



Communicateurs

Topologies

- Dans la plupart des applications, plus particulièrement dans les méthodes de décomposition de domaine où l'on fait correspondre le domaine de calcul à la grille de processus, il est intéressant de pouvoir disposer les processus suivant une topologie régulière
- MPI permet de définir des topologies virtuelles du type cartésien ou graphe
  - Topologies de type cartésien
    - chaque processus est défini dans une grille de processus ;
    - chaque processus a un voisin dans la grille ;
    - la grille peut être périodique ou non ;
    - les processus sont identifiés par leurs coordonnées dans la grille.
  - Topologies de type graphe
    - généralisation à des topologies plus complexes.

Figure 32 – Topologie cartésienne 2D (gauche) et topologie de type graphe (droite)

Communicateurs

Topologies cartésiennes

- Une topologie cartésienne est définie à partir d'un communicateur donné comm_ancien, en appelant le sous-programme `MPI_Cart_create()`.
- On définit :
  - un entier ndims représentant le nombre de dimensions de la grille
  - un tableau d'entiers dims de dimension ndims indiquant le nombre de processus dans chaque dimension
  - un tableau de logiques de dimension ndims indiquant la périodicité dans chaque dimension
  - un logique reorganisation indiquant si la numérotation des processus peut être changé par MPI

```
mpi4py.MPI.Intracomm.Create_cart(dims, periods=None, reorder=False)
```

Communicateurs

Exemple

Exemple sur une grille comportant 4 domaines suivant x et 2 suivant y, périodique en y.

```
dims = [4,2]
periods = [False, True]
comm_2D = MPI.COMM_WORLD.Create_cart(dims, periods)
```

Si `reorganisation = false` alors le rang des processus dans le nouveau communicateur (comm_2D) est le même que dans l'ancien communicateur (`MPI_COMM_WORLD`).
Si `reorganisation = true`, l'implémentation MPI choisit l'ordre des processus.

Communicateurs

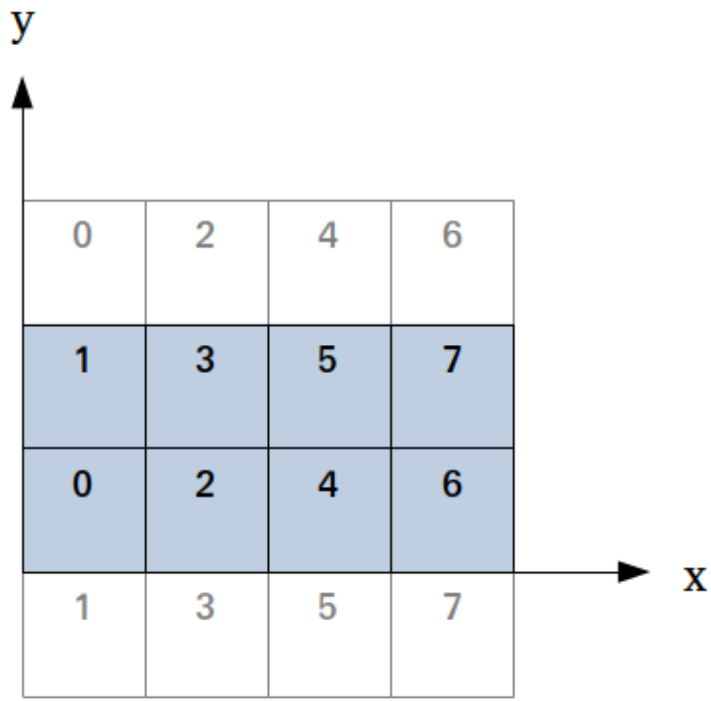


Figure 33 – Topologie cartésienne 2D périodique en y

Communicateurs

Exemple 3D

Exemple sur une grille 3D comportant 4 domaines suivant x, 2 suivant y et 2 suivant z, non périodique.

```
dims = [4,2,2]
comm_3D = MPI.COMM_WORLD.Create_cart(dims)
```

Communicateurs

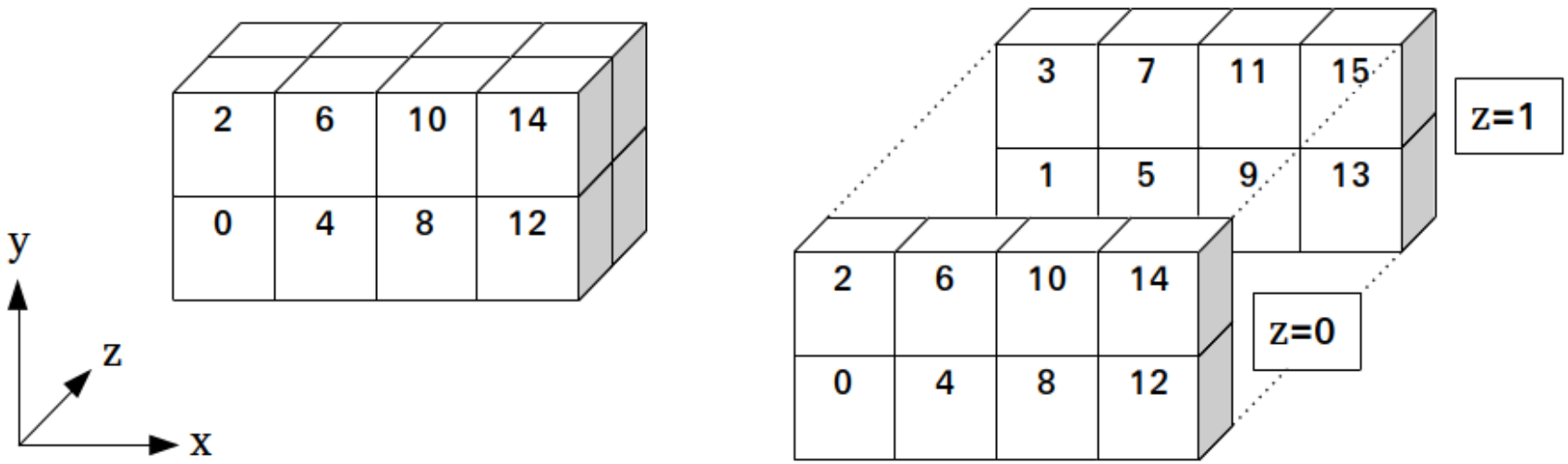


Figure 34 – Topologie cartésienne 3D non périodique

Communicateurs

Distribution des processus

Le sous-programme `MPI_Dims_create()` retourne le nombre de processus dans chaque dimension de la grille en fonction du nombre total de processus.

```
# Retourne une distribution equilibree des processus
mpi4py.MPI.Compute_dims(nnodes, dims)
```

Remarque : si les valeurs de dims en entrée valent toutes 0, cela signifie qu'on laisse à MPI le choix du nombre de processus dans chaque direction en fonction du nombre total de processus.

`dims` en entrée	`Compute_dims`	`dims` en sortie
(0,0)	(8,dims)	(4,2)
(0,0,0)	(16,dims)	(4,2,2)
(0,4,0)	(16,dims)	(2,4,2)
(0,3,0)	(16,dims)	`error`

Rang et coordonnées d'un processus

Dans une topologie cartésienne, le rang de chaque processus est associé à ses coordonnées dans la grille.

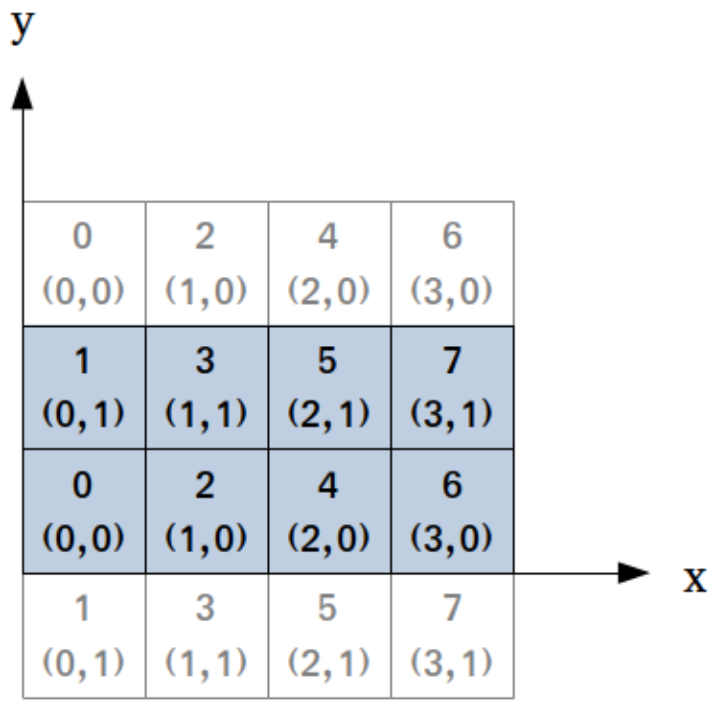


Figure 35 – Topologie cartésienne 2D périodique en y

Communicateurs

Rang d'un processus connaissant ses coordonnées

Dans une topologie cartésienne, le sous-programme `MPI_Cart_rank()` retourne le rang du processus associé aux coordonnées dans la grille.

```
# Retourne le rang
mpi4py.MPI.Cartcomm.Get_cart_rank(coords)
```

Communicateurs

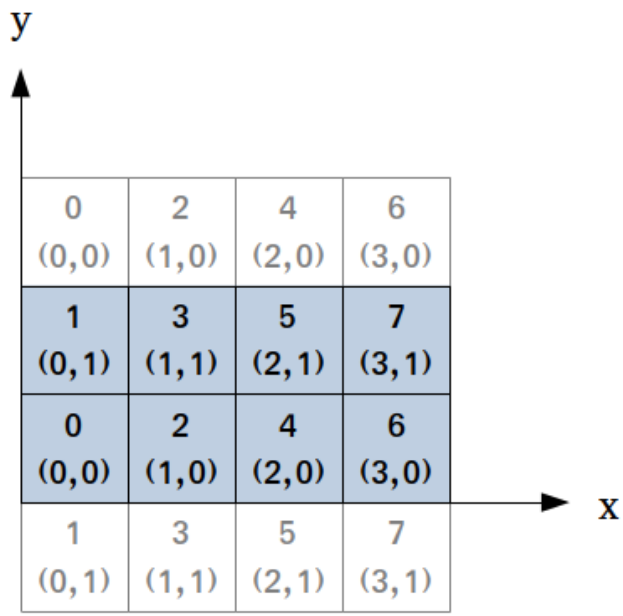


Figure 36 – Topologie cartésienne 2D périodique en y

```
coords[0] = dims[0]-1
for i in range(dims[1]):
    coords[1]=i
    rang[i] = comm_2D.Get_cart_rank(coords)
.........................................
i=0,en entree coords=[3,0],en sortie rang[0]=6.
i=1,en entree coords=[3,1],en sortie rang[1]=7.
```

Communicateurs

Coordonnées d'un processus connaissant son rang

Dans une topologie cartésienne, le sous-programme `MPI_Cart_coords()` retourne les coordonnées d'un processus de rang donné dans la grille.

```
# Retourne les coordonnees
mpi4py.MPI.Cartcomm.Get_coords(rank)
```

Communicateurs

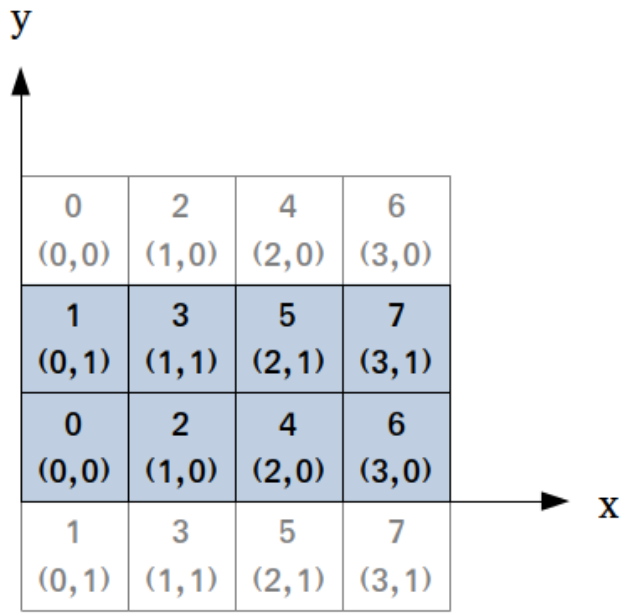


Figure 37 – Topologie cartésienne 2D périodique en y

```
if rang%2 == 0:
    coords = comm_2D.Get_coords(rang)
.........................................
En entree, les valeurs de rang sont : 0,2,4,6.
En sortie, les valeurs de coords sont :
(0,0),(1,0),(2,0),(3,0).
```

Rang des voisins

Dans une topologie cartésienne, un processus appelant le sous-programme `MPI_Cart_Shift()` se voit retourner le rang de ses processus voisins dans une direction donnée.

```
# Retourne un tuple (rang_precedent, rang_suivant)
mpi4py.MPI.Cartcomm.Shift(direction, disp)
```

- Le paramètre direction correspond à l'axe du déplacement (xyz).
- Le paramètre pas correspond au pas du déplacement.
- Si un rang n'a pas de voisin précédent (resp. suivant) dans la direction demandée, alors la valeur du rang précédent (resp. suivant) sera `MPI_PROC_NULL`.

Communicateurs

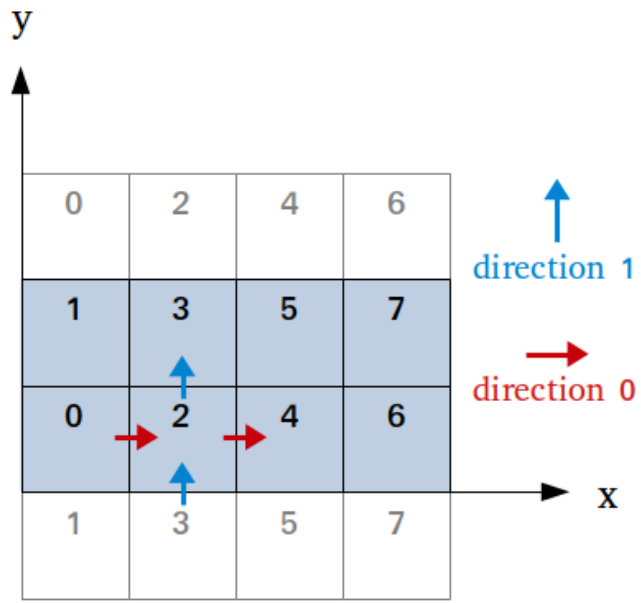


Figure 38 – Appel du sous-programme MPI_Cart_shift()

```
rang_gauche, rang_droit = comm_2D.Shift(0, 1)
.........................................
Pour le processus 2, rang_gauche=0, rang_droit=4
```

```
rang_bas, rang_haut = comm_2D.Shift(1, 1)
.........................................
Pour le processus 2, rang_bas=3, rang_haut=3
```

Communicateurs

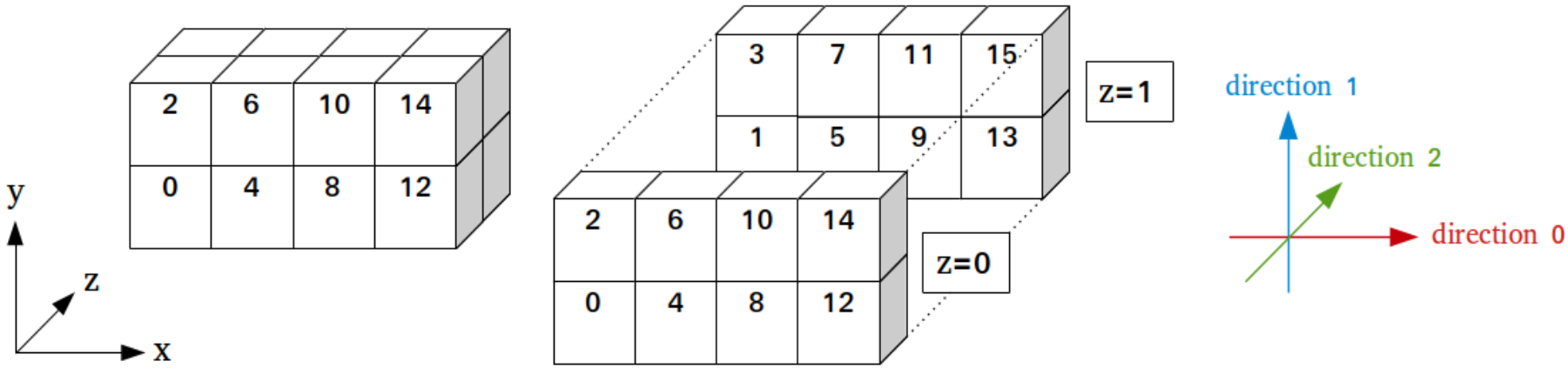


Figure 39 – Appel du sous-programme MPI_Cart_shift()

```
rang_gauche, rang_droit = comm_3D.Shift(0, 1)
.....................................
Pour le processus 0, rang_gauche=-1, rang_droit=4
```

```
rang_bas, rang_haut = comm_3D.Shift(1, 1)
.....................................
Pour le processus 0, rang_bas=-1, rang_haut=2
```

```
rang_avant, rang_arriere = comm_3D.Shift(2, 1)
.....................................
Pour le processus 0, rang_avant=-1, rang_arriere=1
```

Communicateurs

Exemple

- création d'une grille cartésienne 2D périodique en y
- récupération des coordonnées de chaque processus
- récupération des rangs voisins pour chaque processus

```
1  from mpi4py import MPI
2
3  # Initialisation de la matrice et des parametres
4  ndims = 2
5  N = 1
6  E = 2
7  S = 3
8  W = 4
9
10 comm = MPI.COMM_WORLD
11 nb_procs = comm.Get_size()
12
13 # Determination du nombre de processus suivant x et y
14 dims = [0, 0]
15 dims = MPI.Compute_dims(nb_procs, dims)
```

Communicateurs

```
16  # Creation d'une grille 2D periodique en y
17  periods = [False, True]
18  comm_2D = comm.Create_cart(dims, periods)
19
20  # Determination des coordonnees du processus dans la topologie
21  rang_ds_topo = comm_2D.Get_rank()
22  coords = comm_2D.Get_coords(rang_ds_topo)
23
24  # Recherche des voisins Ouest et Est
25  voisin_W, voisin_E = comm_2D.Shift(0, 1)
26
27  # Recherche des voisins Sud et Nord
28  voisin_S, voisin_N = comm_2D.Shift(1, 1)
```

Communicateurs

Subdiviser une topologie cartésienne

- La question est de savoir comment dégénérer une topologie cartésienne de processus 2D ou 3D en une topologie cartésienne respectivement 1D ou 2D.
- Pour MPI, dégénérer une topologie cartésienne 2D (ou 3D) revient à créer autant de communicateurs qu'il y a de lignes ou de colonnes (resp. de plans) dans la grille cartésienne initiale.
- L'intérêt majeur est de pouvoir effectuer des opérations collectives restreintes à un sous-ensemble de processus appartenant à :
  - une même ligne (ou colonne), si la topologie initiale est 2D ;
  - un même plan, si la topologie initiale est 3D.

Communicateurs

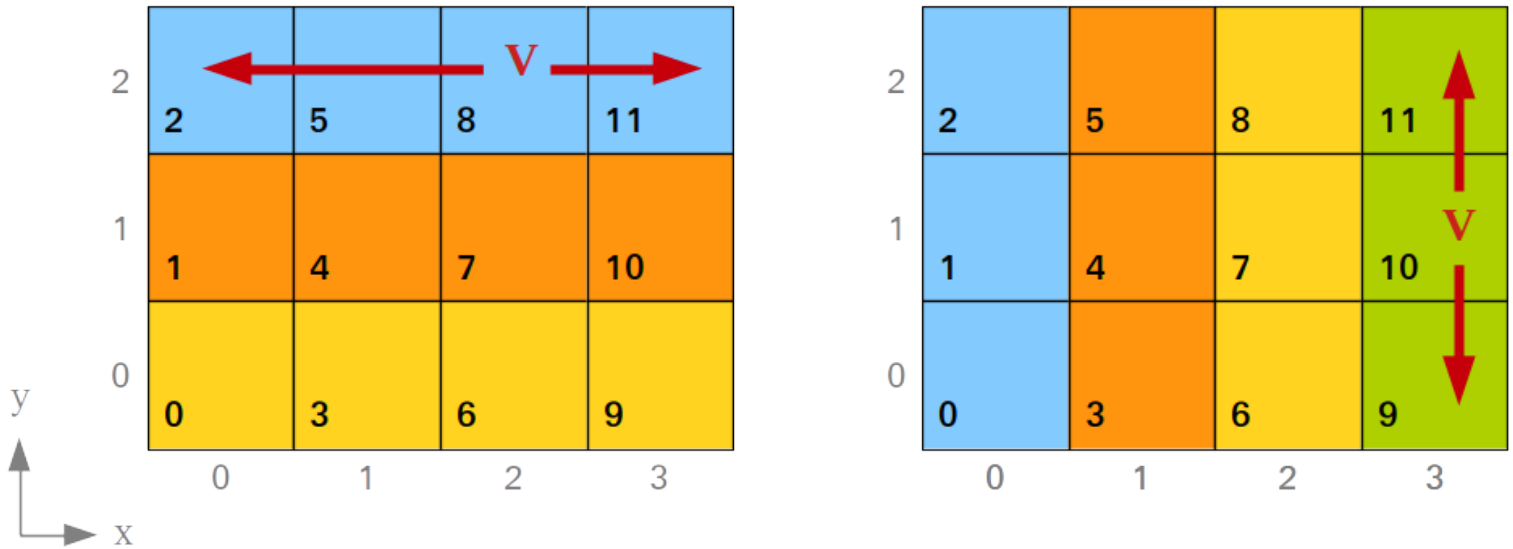


Figure 40 – Deux exemples de distribution de données dans une topologie 2D dégénérée

Communicateurs

Subdiviser une topologie cartésienne

Il existe deux façons de faire pour dégénérer une topologie :

- en utilisant le sous-programme général `MPI_Comm_split()` ;
- en utilisant le sous-programme `MPI_Cart_sub()` prévu à cet effet.

```
# Retourne une topologie
mpi4py.MPI.Cartcomm.Sub(remain_dims)
```

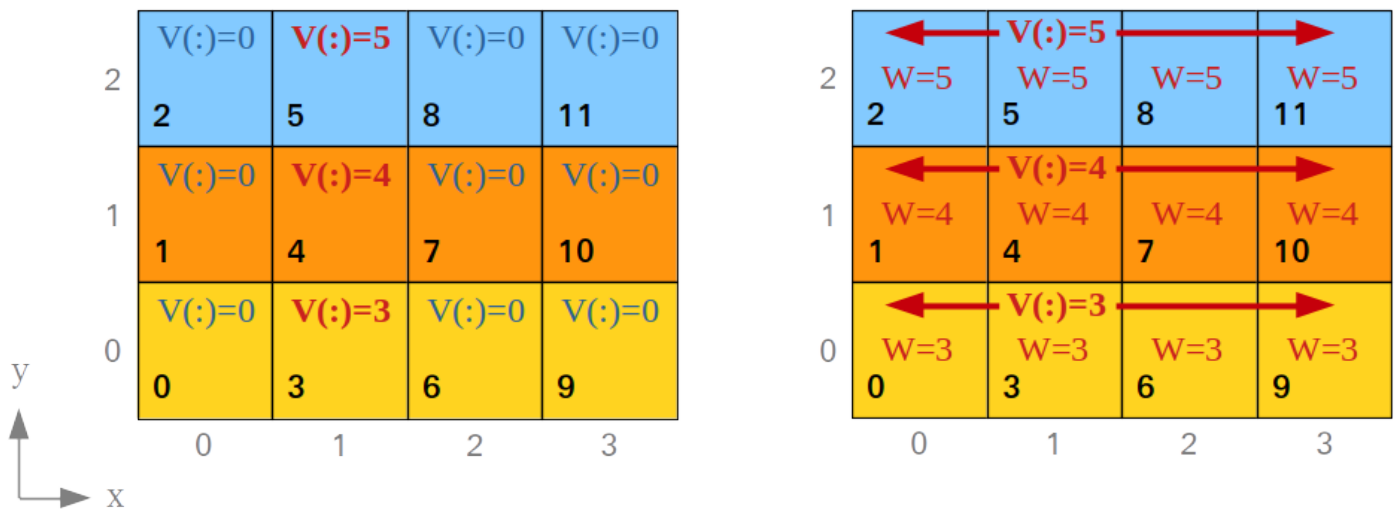


Figure 41 – Distribution d'un tableau *V* sur la grille 2D dégénérée

Communicateurs

```
1  from mpi4py import MPI
2  import numpy as np
3
4  # Parametres
5  NDim2D = 2
6  m = 4
7  Dim2D = [4, 3]
8
9  # Creation de la grille 2D initiale
10 comm = MPI.COMM_WORLD
11 comm_2D = comm.Create_cart(Dim2D)
12
13 # Determination du rang et des coordonnees du processus
14 rang = comm_2D.Get_rank()
15 Coord2D = comm_2D.Get_coords(rang)
```

Communicateurs

```
16  # Initialisation du vecteur V
17  if Coord2D[0] == 1:
18      V = np.full(m, rang, dtype=np.float32)
19  else:
20      V = np.zeros(m, dtype=np.float32)
21
22  # Chaque ligne de la grille doit etre une topologie cartesienne 1D
23  conserve_dims = [1, 0]
24
25  # Subdivision de la grille cartesienne 2D
26  comm_1D = comm_2D.Sub(conserve_dims)
27
28  # Les processus de la colonne 2 distribuent le vecteur V
29  # aux processus de leur ligne
30  W = np.zeros(1, dtype=np.float32)
31  comm_1D.Scatter([V, 1, MPI.FLOAT], [W, 1, MPI.FLOAT], root=1)
32
33  # Affichage des resultats
34  print(f"Rang : {rang} ; Coordonnees : ({Coord2D[0]}, {Coord2D[1]});"
35        f" W = {W[0]}")
```

Communicateurs

```
> mpiexec -n 12 CommCartSub
Rang :  0 ; Coordonnees : (0,0) ; W = 3.
Rang :  1 ; Coordonnees : (0,1) ; W = 4.
Rang :  3 ; Coordonnees : (1,0) ; W = 3.
Rang :  8 ; Coordonnees : (2,2) ; W = 5.
Rang :  4 ; Coordonnees : (1,1) ; W = 4.
Rang :  5 ; Coordonnees : (1,2) ; W = 5.
Rang :  6 ; Coordonnees : (2,0) ; W = 3.
Rang : 10 ; Coordonnees : (3,1) ; W = 4.
Rang : 11 ; Coordonnees : (3,2) ; W = 5.
Rang :  9 ; Coordonnees : (3,0) ; W = 3.
Rang :  2 ; Coordonnees : (0,2) ; W = 5.
Rang :  7 ; Coordonnees : (2,1) ; W = 4.
```

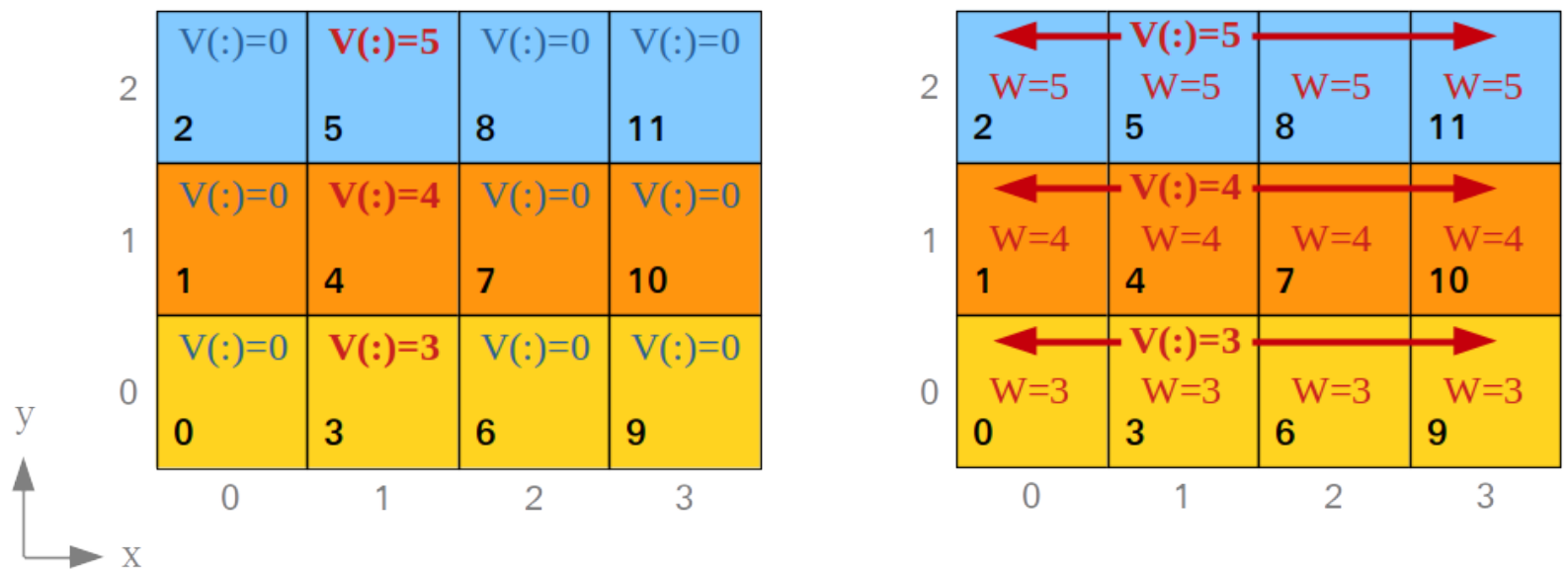


Figure 42 – Distribution d'un tableau *V* sur la grille 2D dégénérée

Travaux pratiques MPI – Exercice 6 : Communicateurs

- En partant de la topologie cartésienne définie ci-dessous, subdiviser en 2 communicateurs suivant les lignes via `MPI_Comm_split()`. Ensuite, faire en sorte que les processus de la 2eme colonne diffusent sélectivement le vecteur V aux processus de leur ligne.

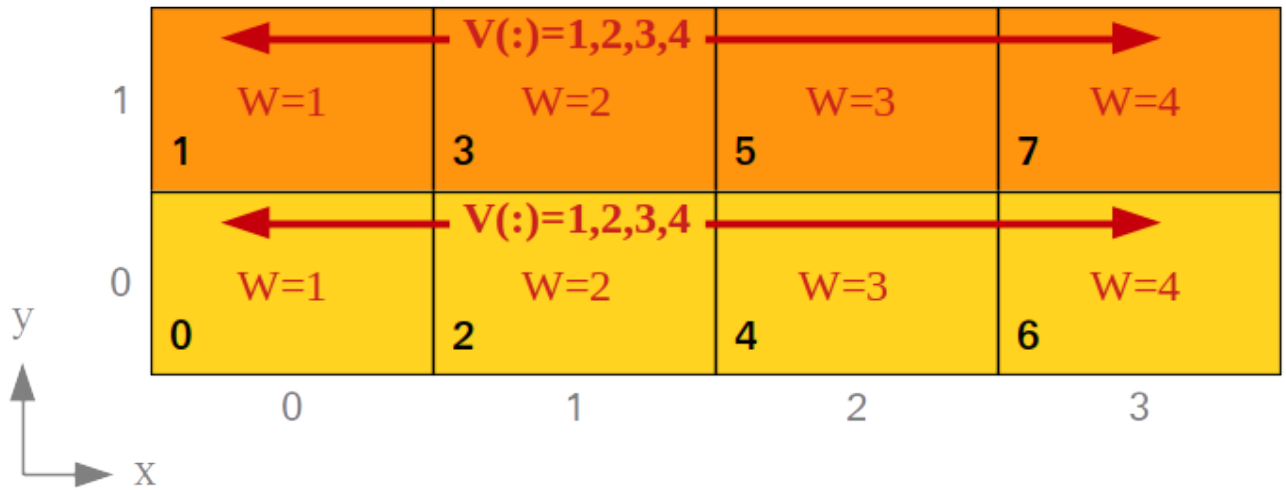


Figure 43 – Subdivision d'une topologie 2D et communication suivant la topologie 1D obtenue

- Contrainte : définir les couleurs de chaque processus sans utiliser l'opération *modulo*.

MPI-IO

Optimisation des entrées-sorties

- Très logiquement, les applications qui font des calculs volumineux manipulent également des quantités importantes de données externes, et génèrent donc un nombre conséquent d'entrées-sorties.
- Le traitement efficace de celles-ci influe donc parfois très fortement sur les performances globales des applications.
- L'optimisation des entrées-sorties de codes parallèles se fait par la combinaison :
  - de leur parallélisation, pour éviter de créer un goulet d'étranglement en raison de leur sérialisation ;
  - de techniques mises en œuvre explicitement au niveau de la programmation (lectures / écritures non-bloquantes) ;
  - d'opérations spécifiques prises en charge par le système d'exploitation (regroupement des requêtes, gestion des tampons d'entrées-sorties, etc.).
- L'utilisation d'une bibliothèque facilite l'optimisation des entrées-sorties.

L'interface MPI-IO

- La norme MPI-2 définit un ensemble de fonctions permettant de réaliser des entrées-sorties parallèles.
- L'interface est calquée sur celle utilisée pour l'échange de messages MPI. Par exemple, les opérations collectives et non-bloquantes sur les fichiers sont gérées de façon similaire à ce que propose MPI pour les messages entre processus. La définition des données accédées suivant les processus se fait par l'utilisation de types de données (de base ou bien dérivés).
- Bien sûr, de nombreux éléments (descripteurs de fichiers, attributs . . .) rappellent les interfaces d'entrées-sorties natives des langages de programmation.

MPI-IO

Exemple d'optimisation séquentielle implémentée par les bibliothèques

- Pour obtenir de bonnes performances, il est préférable de limiter le nombre de requêtes (latence) et de lire de larges blocs de données.
- Lorsqu'un seul processus accède à de nombreux petits blocs discontinus, il est possible de regrouper les requêtes pour plus de performances.
- Une bibliothèque MPI-IO peut implémenter cette optimisation de manière transparente.

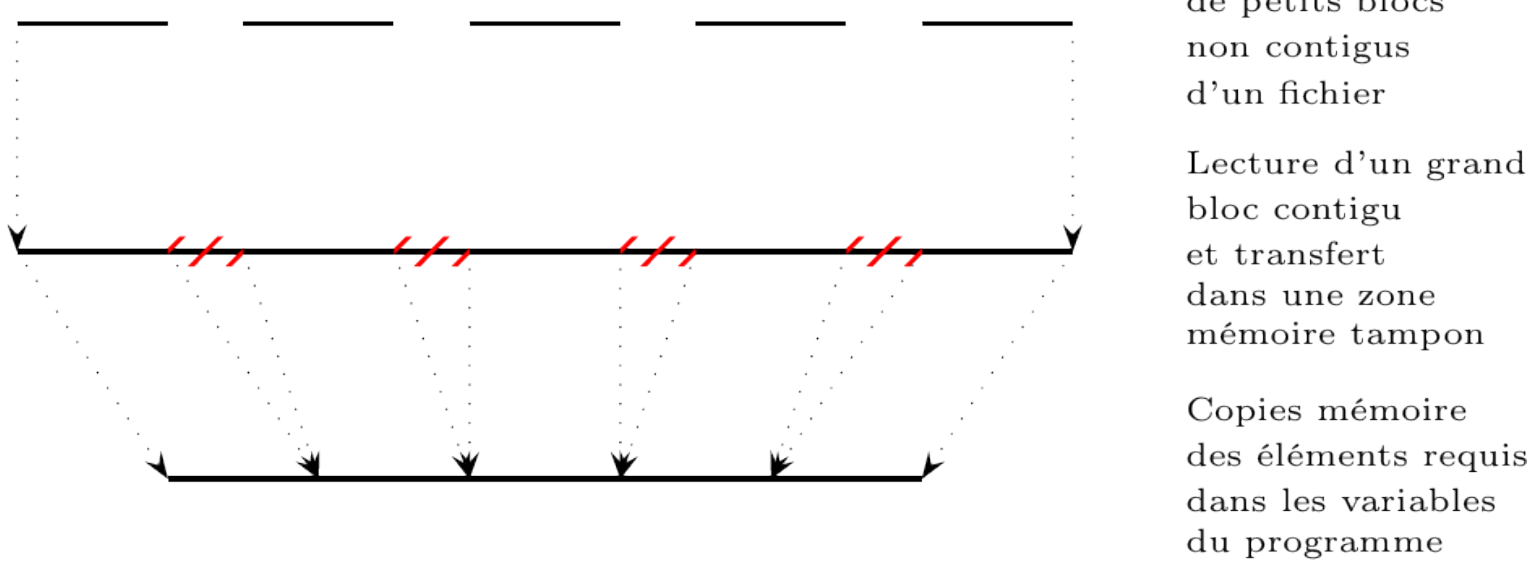


Figure 44 – Mécanisme de *passoire* (*data sieving*) dans le cas d'accès nombreux, par un seul processus, à de petits blocs discontinus

Exemple d'optimisation parallèle

Lorsqu'un ensemble de processus accède à des blocs discontinus (cas des tableaux distribués, par exemple), la bibliothèque d'I/O peut optimiser l'opération en maximisant l'accès aux données contiguës et en utilisant des communications collectives de redistribution.

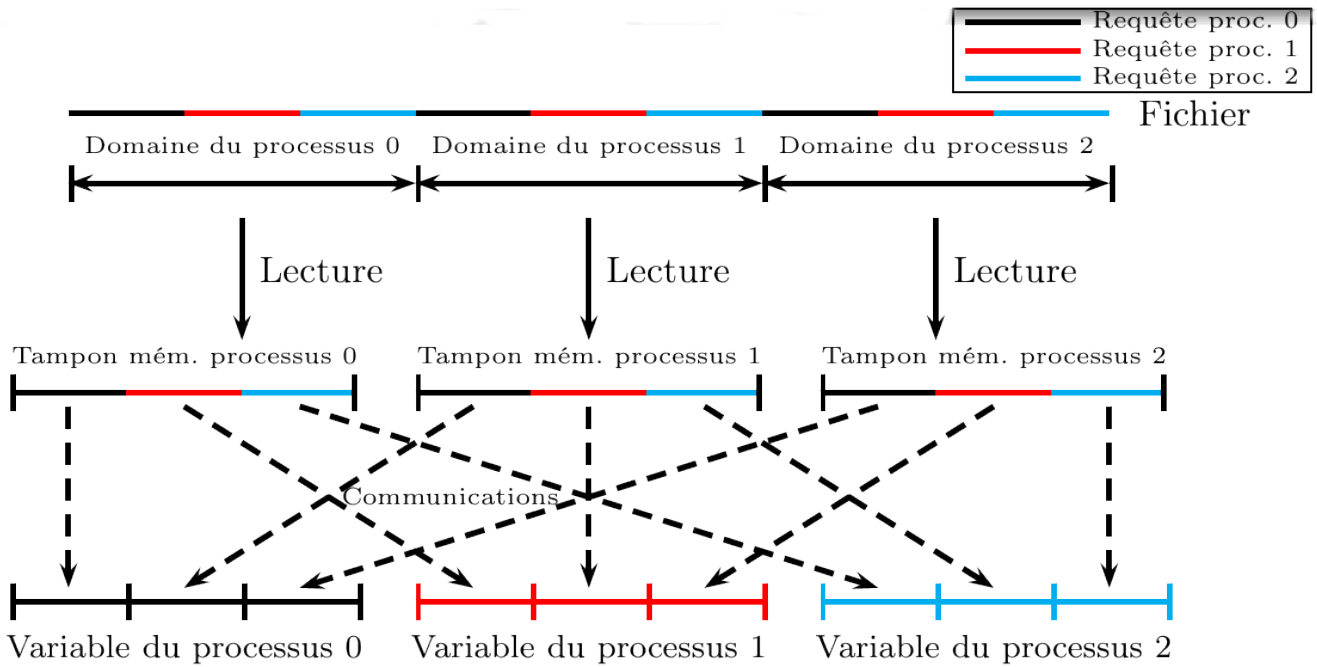


Figure 45 – Lecture en deux phases, par un ensemble de processus

Ouverture d'un fichier

```
# Retourne un descripteur de fichier
mpi4py.MPI.File.Open(comm, filename, amode=MODE_RDONLY, info=INFO_NULL)
```

- Ouvre le fichier filename avec les attributs amode ;
- descripteur est un objet opaque qui est ensuite utilisé comme référence dans toutes les opérations portant sur le fichier ;
- L'ouverture est une opération *collective* ;
- filename et amode doivent être identiques sur tous les rangs du communicateur comm ;
- Un objet de type MPI_Info est une base de donnée de type clé-valeur utile pour l'optimisation. `MPI_INFO_NULL` permet d'utiliser une valeur par défaut.

Attributs

Attribut	Signification
MPI.MODE_RDONLY	seulement en lecture
MPI.MODE_RDWR	en lecture et écriture
MPI.MODE_WRONLY	seulement en écriture
MPI.MODE_CREATE	création du fichier s'il n'existe pas
MPI.MODE_EXCL	erreur si le fichier existe
MPI.MODE_UNIQUE_OPEN	le fichier n'est pas ouvert ailleurs
MPI.MODE_SEQUENTIAL	accès séquentiel
MPI.MODE_APPEND	pointeurs en fin de fichier (mode ajout)
MPI.MODE_DELETE_ON_CLOSE	destruction après la fermeture

Les attributs peuvent être combinés en utilisant l'opérateur |.

Fermeture d'un fichier

```
mpi4py.MPI.File.Close()
```

- Ferme le fichier ;
- La fermeture est une opération *collective*.

MPI-IO

```python
from mpi4py import MPI

# Tentative d'ouverture du fichier
descripteur = MPI.File.Open(MPI.COMM_WORLD, "fichier.txt",
                            MPI.MODE_RDWR | MPI.MODE_CREATE)

# En python pas besoin de verifier

# Fermeture du fichier
descripteur.Close()
```

```
> ls -l fichier.txt

-rw-------   1 nom       grp    0 Feb 08 12:13 fichier.txt
```

Généralités

- Les transferts de données entre fichiers et zones mémoire des processus se font via des appels explicites à des sous-programmes de lecture et d'écriture.
- On distingue trois propriétés des accès aux fichiers :
  - le positionnement, qui peut être explicite (en spécifiant un déplacement par rapport au début du fichier) ou implicite, via des pointeurs gérés par le système (ces pointeurs peuvent être de deux types : soit individuels à chaque processus, soit partagés par tous les processus) ;
  - la synchronisation, les accès pouvant être de type bloquants ou non bloquants ;
  - le regroupement, les accès pouvant être collectifs (c'est-à-dire effectués par tous les processus du communicateur au sein duquel le fichier a été ouvert) ou propres seulement à un ou plusieurs processus.
- Il est possible de mélanger les types d'accès effectués à un même fichier au sein d'une application.

MPI-IO

Positionnement	Synchronisation	individuel	collectif
adresses explicites	bloquantes	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	non bloquantes	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
pointeurs implicites individuels	bloquantes	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	non bloquantes	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
pointeurs implicites partagés	bloquantes	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	non bloquantes	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

Le mécanisme de vue

- Par défaut, les fichiers sont lus comme une simple suite d'octets mais MPI-IO dispose d'un mécanisme permettant une abstraction de plus haut niveau du contenu des fichiers : il est possible de décrire des structures de données complexes et de s'en servir comme gabarit lors de l'accès aux fichiers.
- Pour l'instant, il faut seulement savoir qu'un type élémentaire de données sert d'unité de base à ces constructions et que, par défaut, le type élémentaire est l'octet.
- Ce mécanisme de vue sera décrit en détail plus tard.

Déplacements explicites

```
mpi4py.MPI.File.Read_at(offset, [buf, count, datatype], status=None)
mpi4py.MPI.File.Write_at(offset, [buf, count, datatype], status=None)
```

- Ecrit/lit à la position offset dans le fichier descripteur, count élément de type datatype depuis l'adresse buf ;
- La position dans le fichier s'exprime toujours comme un multiple du type élémentaire de la vue courante. Par défaut, la position s'exprime donc en octets ;
- La taille du datatype doit être un multiple du type élémentaire.

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

for i in range(nb_valeurs):
    valeurs[i] = i + 1 + rank * 100
print(f"Ecriture processus {rank} : {valeurs}")

# Ouvrir le fichier
descripteur = MPI.File.Open(comm, "donnees.dat",
                            MPI.MODE_WRONLY | MPI.MODE_CREATE)

# Calculer la position du fichier
nb_octets_entier = MPI.INT.Get_size()
position_fichier = rank * nb_valeurs * nb_octets_entier

# Ecrire les donnees
descripteur.Write_at(position_fichier, valeurs)

# Fermer le fichier
descripteur.Close()
```

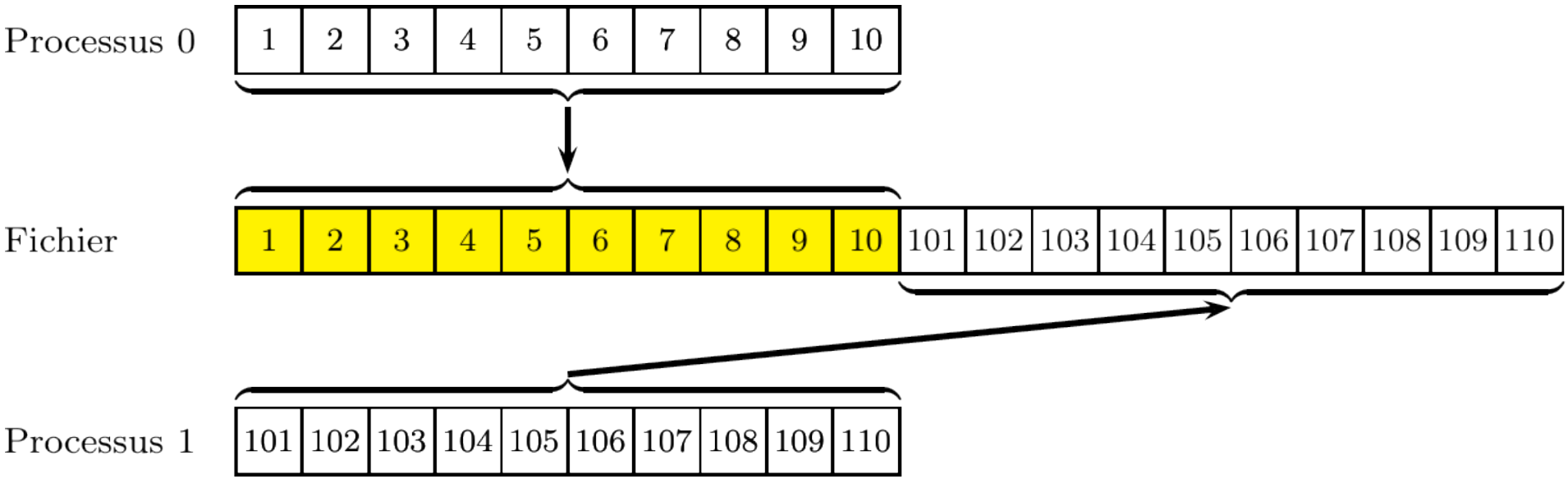


Figure 46 – Exemple d'utilisation de `MPI_File_write_at()`

```
> mpiexec -n 2 python -m mpi4py write_at.py

Ecriture processus 0 :  [ 1  2  3  4  5  6  7  8  9  10]
Ecriture processus 1 :  [101 102 103 104 105 106 107 108 109 110]
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

# Ouvrir le fichier
descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Calculer la position du fichier
nb_octets_entier = MPI.INT.Get_size()
position_fichier = rank * nb_valeurs * nb_octets_entier

# Lecture des donnees
descripteur.Read_at(position_fichier, valeurs)
print(f"Lecture processus {rank} : {valeurs}")

# Fermer le fichier
descripteur.Close()
```

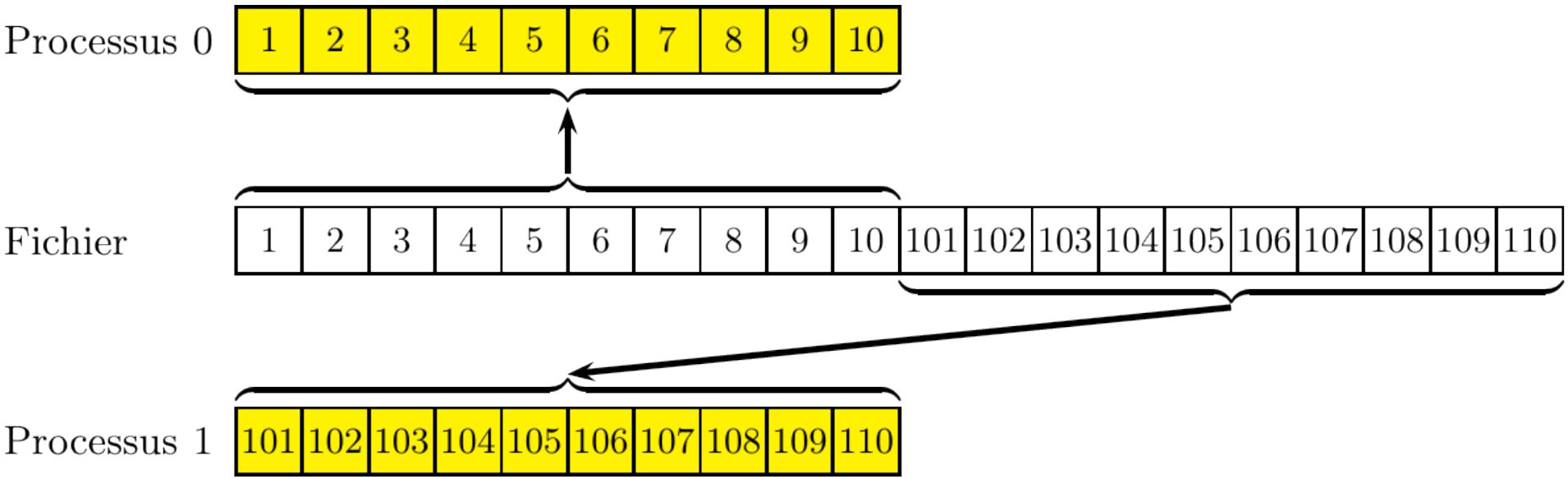


Figure 47 – Exemple d'utilisation de `MPI_File_read_at()`

```
> mpiexec -n 2 python -m mpi4py read_at.py

Lecture processus 0 : [  1  2  3  4  5  6  7  8  9  10]
Lecture processus 1 : [101 102 103 104 105 106 107 108 109 110]
```

Déplacements implicites individuels

```
mpi4py.MPI.File.Read([buf, count, datatype], status=None)
mpi4py.MPI.File.Write([buf, count, datatype], status=None)
```

- Ecrit/lit dans le fichier descripteur, count élément de type datatype depuis l'adresse buf ;
- Un pointeur individuel est géré par MPI, et ceci par fichier et par processus.
- Après chaque accès, le pointeur est positionné sur l'élément suivant.
- Dans tous ces sous-programmes, les pointeurs partagés ne sont jamais accédés ou modifiés explicitement.

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

# Ouvrir le fichier
descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Lire les donnees
descripteur.Read([valeurs, 6])
descripteur.Read([valeurs[6:], 4])
print(f"Lecture processus {rank} : {valeurs}")

# Fermer le fichier
descripteur.Close()
```

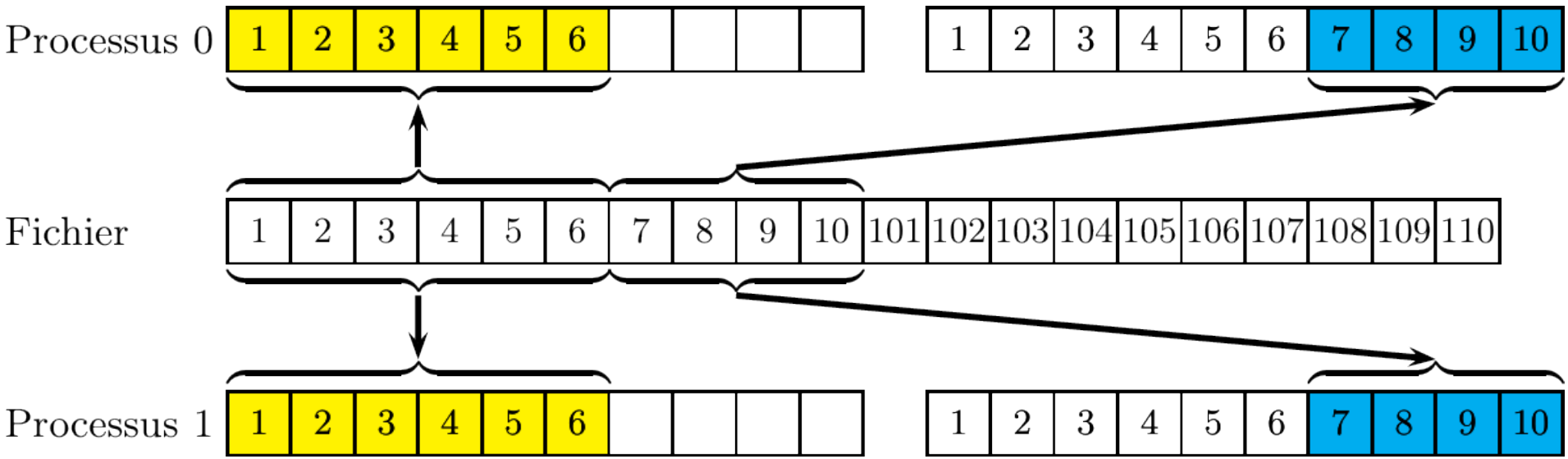


Figure 48 – Exemple 1 d'utilisation de `MPI_File_read()`

```
> mpiexec -n 2 python -m mpi4py read01.py

Lecture processus 1 :  [1  2  3  4  5  6  7  8  9  10]
Lecture processus 0 :  [1  2  3  4  5  6  7  8  9  10]
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

# Ouvrir le fichier
descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Lire les donnees
if rank == 0:
    descripteur.Read([valeurs, 5])
else:
    descripteur.Read([valeurs, 8])
    descripteur.Read([valeurs, 5])
print(f"Lecture processus {rank} : {valeurs[:8]}")

# Fermer le fichier
descripteur.Close()
```

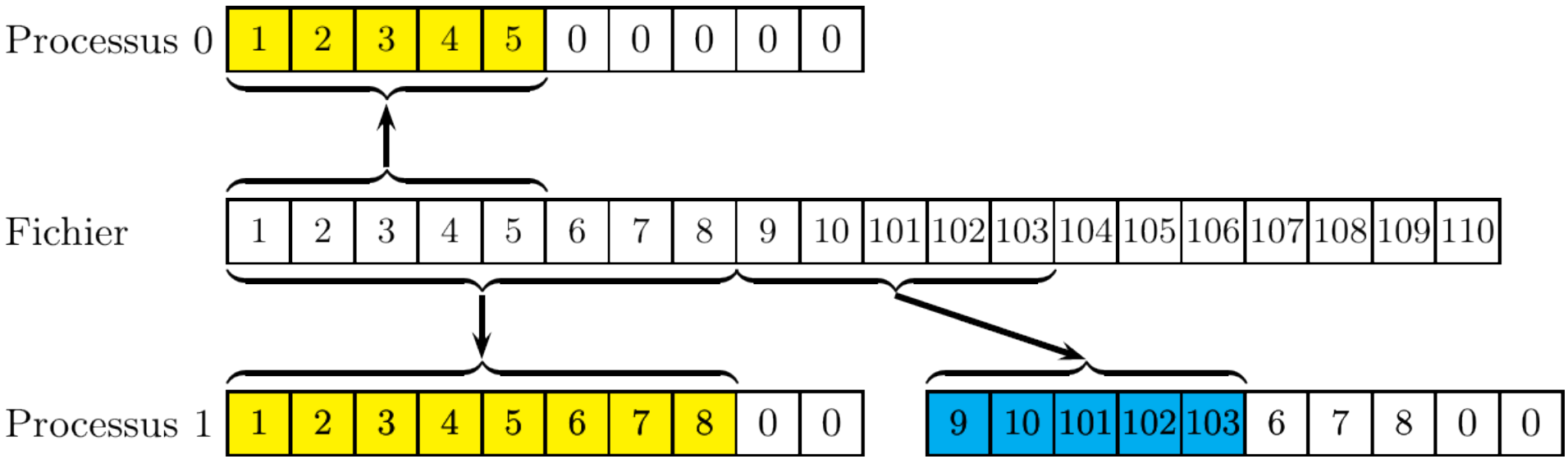


Figure 49 – Exemple 2 d'utilisation de `MPI_File_read()`

```
> mpiexec -n 2 python -m mpi4py read02.py

Lecture processus 0 : [1 2 3 4 5 0 0 0]
Lecture processus 1 : [  9  10 101 102 103   6   7   8]
```

Déplacements implicites partagés

```
mpi4py.MPI.File.Read_shared([buf, count, datatype], status=None)
mpi4py.MPI.File.Write_shared([buf, count, datatype], status=None)
```

- Ecrit/lit dans le fichier descripteur, count élément de type datatype depuis l'adresse buf ;
- Il existe un et un seul pointeur partagé par fichier, commun à tous les processus du communicateur dans lequel le fichier a été ouvert.
- Tous les processus qui font une opération d'entrée-sortie utilisant le pointeur partagé doivent employer la même vue du fichier.
- Si on utilise les variantes non collectives des sous-programmes, l'ordre n'est pas déterministe. Si le traitement doit être déterministe, il faut explicitement gérer l'ordonnancement des processus ou utiliser les variantes collectives.
- Après chaque accès, le pointeur est positionné sur l'élément suivant.
- Dans tous ces sous-programmes, les pointeurs individuels ne sont jamais accédés ou modifiés.

MPI-IO

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6
7  nb_valeurs = 10
8  valeurs = np.zeros(nb_valeurs, dtype=np.int32)
9
10 # Ouvrir le fichier
11 descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
12
13 # Lire les donnees avec lecture partagee
14 descripteur.Read_shared([valeurs, 4, MPI.INT])
15 descripteur.Read_shared([valeurs[4:], 6, MPI.INT])
16
17 print(f"Lecture processus {rank} : {valeurs}")
18
19 # Fermer le fichier
20 descripteur.Close()
```

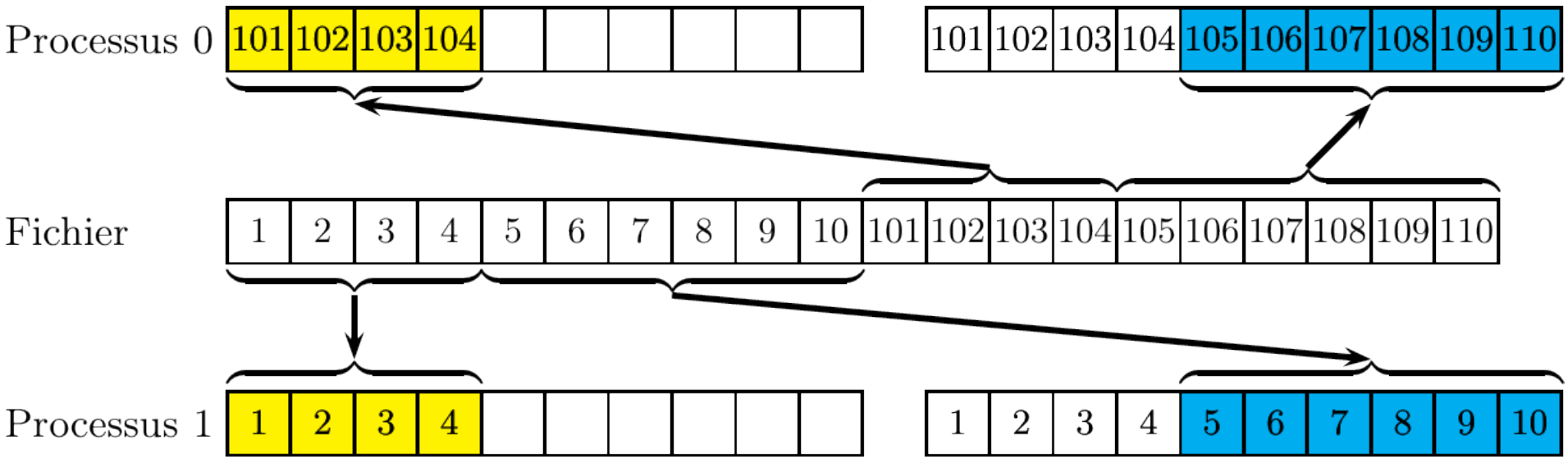


Figure 50 – Exemple 2 d'utilisation de `MPI_File_read_shared()`

```
> mpiexec -n 2 python -m mpi4py read_shared01.py

Lecture processus  1 : [  1  2  3  4  5  6  7  8  9  10]
Lecture processus  0 : [101 102 103 104 105 106 107 108 109 110]
```

Lectures/écritures collectives

- Tous les processus du communicateur au sein duquel un fichier est ouvert participent aux opérations collectives d'accès aux données.
- Les opérations collectives sont généralement plus performantes que les opérations individuelles, parce qu'elles autorisent davantage de techniques d'optimisation mises en œuvre automatiquement ;
- Les accès sont effectués dans l'ordre des rangs des processus : le traitement est donc ici déterministe.

MPI-IO

Interfaces

```
mpi4py.MPI.File.Read_at_all(offset, [buf, count, datatype], status=None)
mpi4py.MPI.File.Write_at_all(offset, [buf, count, datatype], status=None)
mpi4py.MPI.File.Read_all([buf, count, datatype], status=None)
mpi4py.MPI.File.Write_all([buf, count, datatype], status=None)
mpi4py.MPI.File.Read_ordered([buf, count, datatype], status=None)
mpi4py.MPI.File.Write_ordered([buf, count, datatype], status=None)
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

# Ouvrir le fichier
descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Calculer la position du fichier
nb_octets_entier = MPI.INT.Get_size()
position_fichier = rank * nb_valeurs * nb_octets_entier

# Lecture des donnees
descripteur.Read_at_all(position_fichier, valeurs)
print(f"Lecture processus {rank} : {valeurs}")

# Fermer le fichier
descripteur.Close()
```

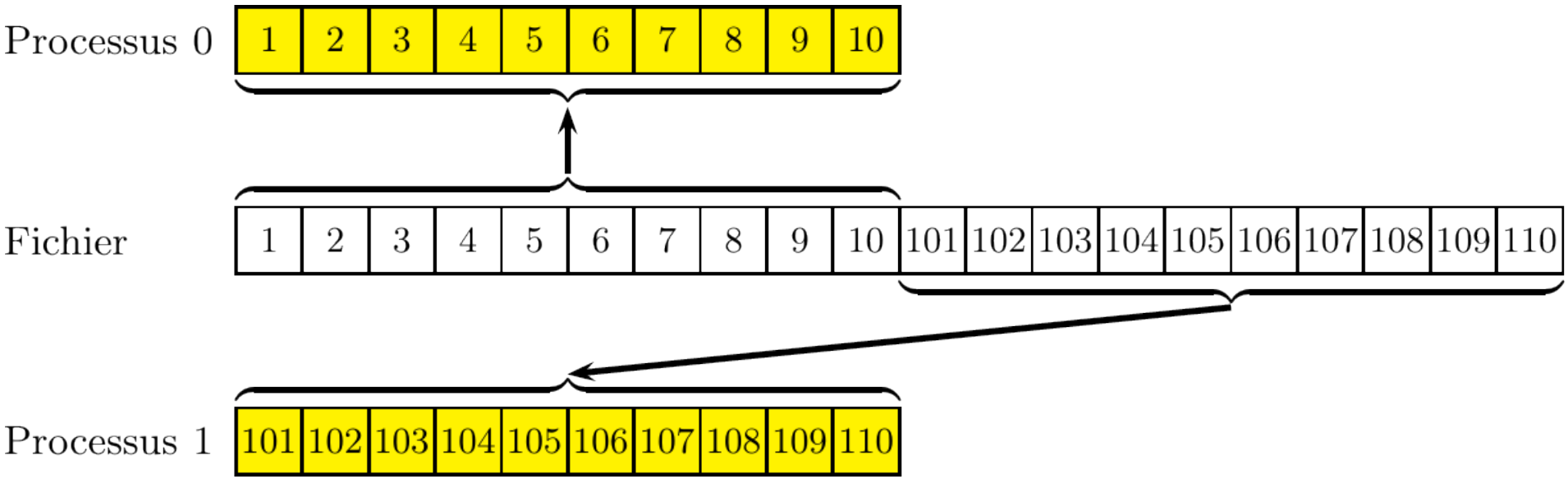


Figure 51 – Exemple d'utilisation de `MPI_File_read_at_all()`

```
> mpiexec -n 2 python -m mpi4py read_at_all.py

Lecture processus 0 : [  1  2  3  4  5  6  7  8  9  10]
Lecture processus 1 : [101 102 103 104 105 106 107 108 109 110]
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

# Ouvrir le fichier
descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Lire les donnees
descripteur.Read_all([valeurs, 6])
descripteur.Read_all([valeurs[6:], 4])
print(f"Lecture processus {rank} : {valeurs}")

# Fermer le fichier
descripteur.Close()
```

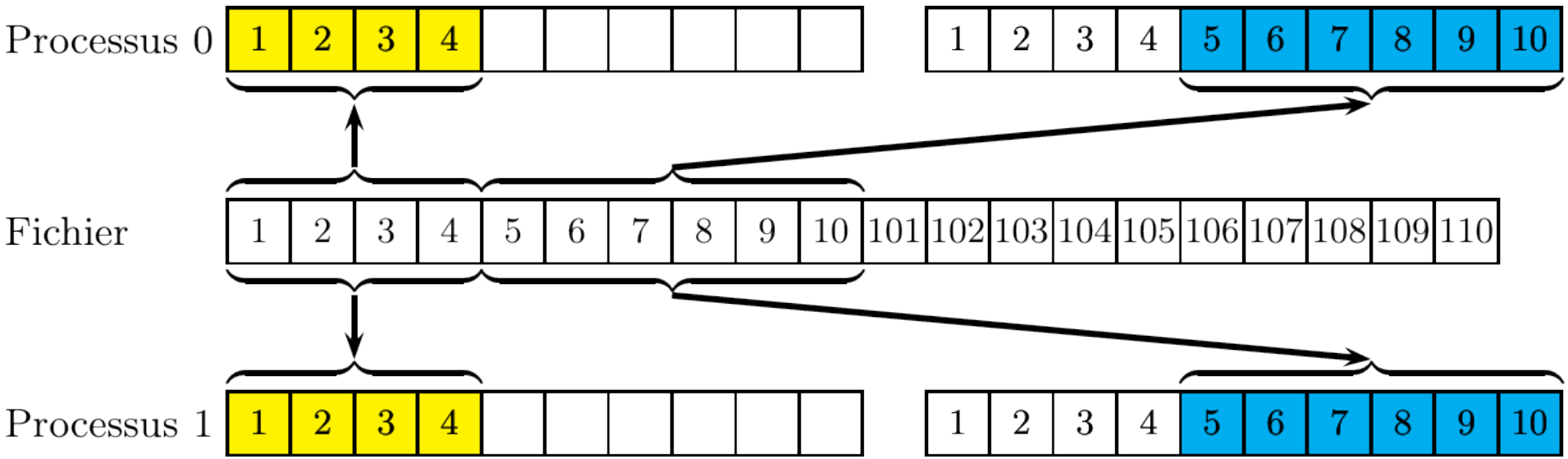


Figure 52 – Exemple 1 d'utilisation de `MPI_File_read_all()`

```
> mpiexec -n 2 python -m mpi4py readall01.py

Lecture processus 1 :  [1  2  3  4  5  6  7  8  9  10]
Lecture processus 0 :  [1  2  3  4  5  6  7  8  9  10]
```

MPI-IO

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6
7  nb_valeurs = 10
8  valeurs = np.zeros(nb_valeurs, dtype=np.int32)
9
10 # Ouvrir le fichier
11 descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
12
13 # Lire les donnees
14 if rank == 0:
15     indice1 = 2
16     indice2 = 5
17 else:
18     indice1 = 4
19     indice2 = 8
20 descripteur.Read_all([valeurs[indice1:], indice2-indice1+1])
21 print(f"Lecture processus {rank} : {valeurs}")
22
23 # Fermer le fichier
24 descripteur.Close()
```

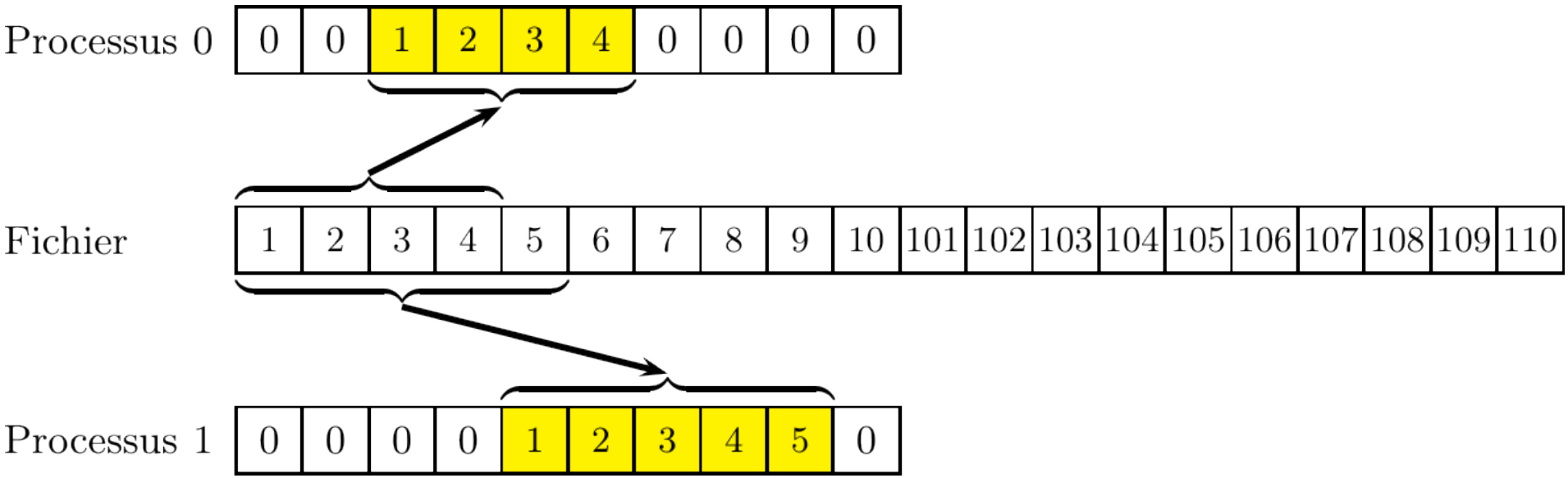


Figure 53 – Exemple 2 d'utilisation de `MPI_File_read_all()`

```
> mpiexec -n 2 python -m mpi4py ./read_all02.py

Lecture processus 0 : [0 0 1 2 3 4 0 0 0 0]
Lecture processus 1 : [0 0 0 0 1 2 3 4 5 0]
```

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

# Ouvrir le fichier
descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)

# Lire les donnees
if rank == 0:
    descripteur.Read_all([valeurs[2:], 4])
else:
    descripteur.Read_all([valeurs[4:], 5])
print(f"Lecture processus {rank} : {valeurs}")

# Fermer le fichier
descripteur.Close()
```

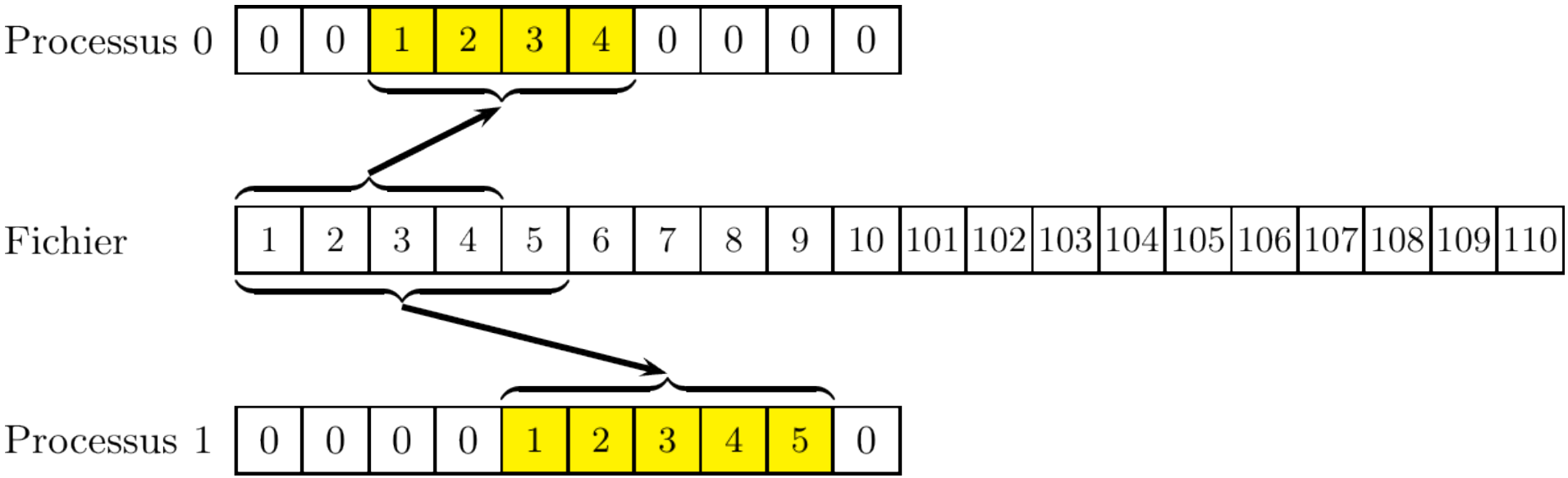


Figure 54 – Exemple 3 d'utilisation de `MPI_File_read_all()`

```
> mpiexec -n 2 python -m mpi4py ./read_all02.py

Lecture processus 0 : [0 0 1 2 3 4 0 0 0 0]
Lecture processus 1 : [0 0 0 0 1 2 3 4 5 0]
```

MPI-IO

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6
7  nb_valeurs = 10
8  valeurs = np.zeros(nb_valeurs, dtype=np.int32)
9
10 # Ouvrir le fichier
11 descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
12
13 # Lire les donnees avec lecture partagee
14 descripteur.Read_ordered([valeurs, 4, MPI.INT])
15 descripteur.Read_ordered([valeurs[4:], 6, MPI.INT])
16
17 print(f"Lecture processus {rank} : {valeurs}")
18
19 # Fermer le fichier
20 descripteur.Close()
```

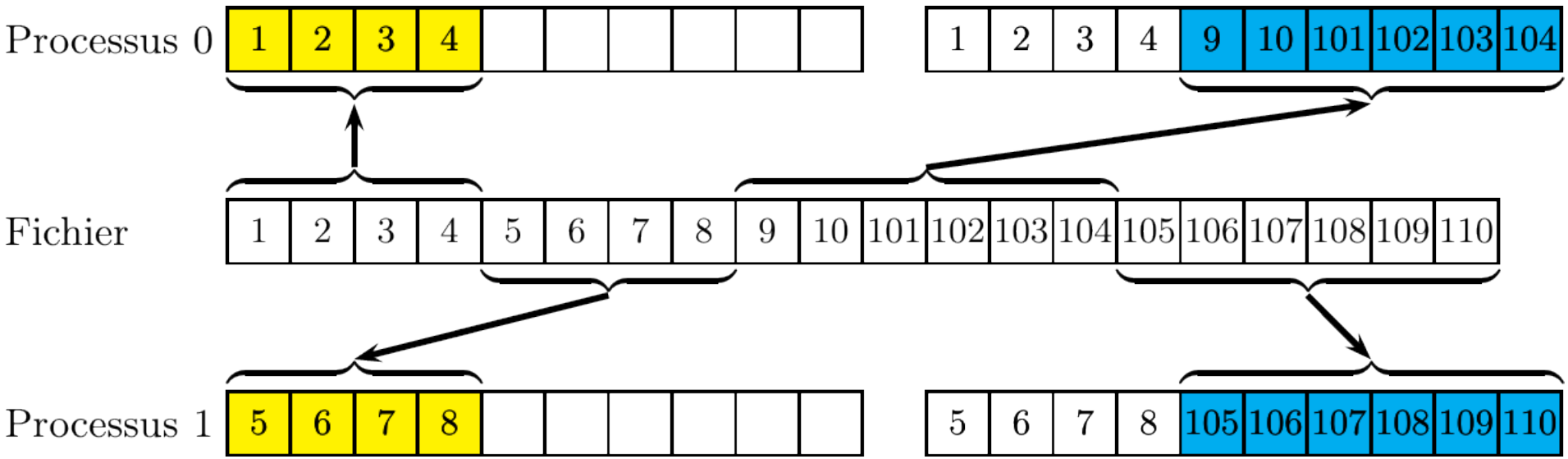


Figure 55 – Exemple d'utilisation de `MPI_File_ordered()`

```
> mpiexec -n 2 python -m mpi4py ./readOrdered.py

Lecture processus 0 : [  1   2   3   4   9  10 101 102 103 104]
Lecture processus 1 : [  5   6   7   8 105 106 107 108 109 110]
```

Positionnement explicite des pointeurs dans un fichier

```
mpi4py.MPI.File.Seek(offset, whence=SEEK_SET)
mpi4py.MPI.File.Seek_shared(offset, whence=SEEK_SET)
```

- Il est possible de positionner explicitement les pointeurs individuels à l'aide du sous-programme `MPI_File_seek()`, et de même le pointeur partagé avec le sous-programme `MPI_File_seek_shared()`.
- Il y a trois modes possibles pour modifier la valeur d'un pointeur :
  - `MPI_SEEK_SET` permet de définir un déplacement absolu ;
  - `MPI_SEEK_CUR` permet un déplacement relativement à la position courante ;
  - `MPI_SEEK_END` positionne le pointeur à la fin du fichier, à laquelle un déplacement éventuel est ajouté.
- Avec `MPI_SEEK_CUR` et `MPI_SEEK_END`, on peut spécifier une valeur négative, ce qui permet de revenir en arrière dans le fichier.

MPI-IO

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nb_valeurs = 10
valeurs = np.zeros(nb_valeurs, dtype=np.int32)

descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
descripteur.Read([valeurs, 3, MPI.INT])
nb_octets_entier = MPI.INT.Get_size()
position_fichier = 8 * nb_octets_entier
descripteur.Seek(position_fichier, MPI.SEEK_CUR)
descripteur.Read([valeurs[3:], 3, MPI.INT])
position_fichier = 4 * nb_octets_entier
descripteur.Seek(position_fichier, MPI.SEEK_SET)
descripteur.Read([valeurs[6:], 4, MPI.INT])
print(f"Lecture processus {rank} : {valeurs}")
descripteur.Close()
```

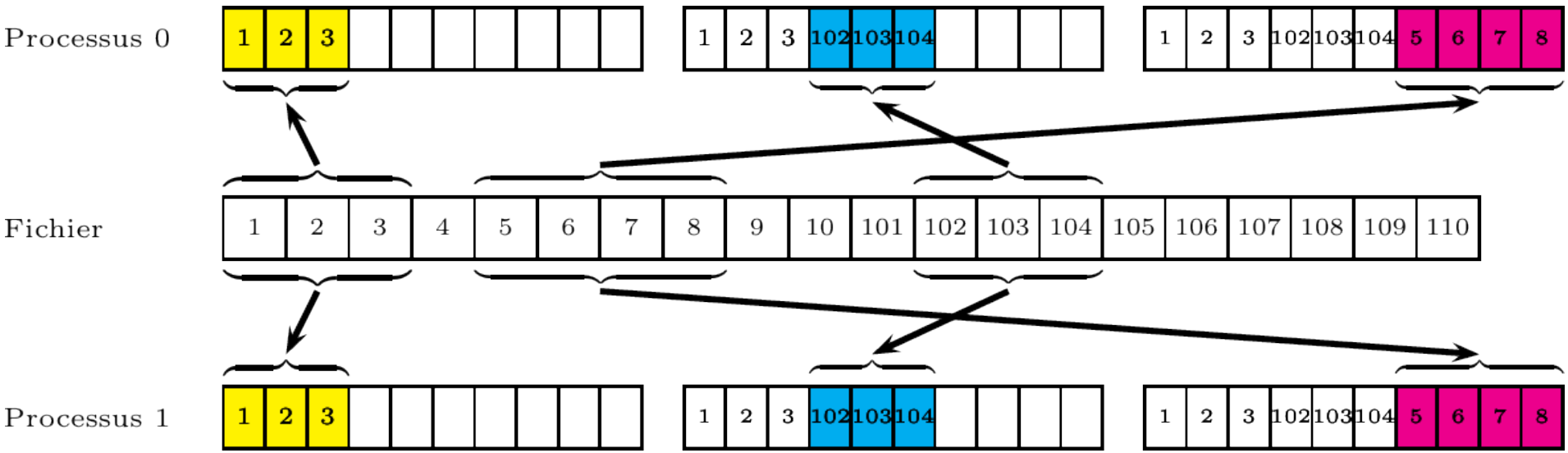


Figure 56 – Exemple d'utilisation de `MPI_File_seek()`

```
> mpiexec -n 2 python -m mpi4py ./seek.py

Lecture processus 1 : [  1   2   3 102 103 104   5   6   7   8]
Lecture processus 0 : [  1   2   3 102 103 104   5   6   7   8]
```

Accès à la fin du fichier

- Écrire à la fin du fichier augmente la taille du fichier.
- Lire à partir de la fin du fichier ne récupère aucune donnée. Lors d'une lecture, l'utilisation de `MPI_Get_count()` permet de connaître le nombre d'éléments réellement lus.

Lectures/écritures non bloquantes

- L'intérêt est de faire un recouvrement entre les calculs et les entrées-sorties.
- Les entrées-sorties non bloquantes sont implémentées suivant le modèle utilisé pour les communications non bloquantes entre processus.
- Un accès non-bloquant doit donner lieu ultérieurement à un test explicite de complétude ou à une mise en attente (via `MPI_Test()`, `MPI_Wait()`, etc.)

MPI-IO

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  nb_valeurs = 10
7  valeurs = np.zeros(nb_valeurs, dtype=np.int32)
```

MPI-IO

```
 9  descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
10  nb_octets_entier = MPI.INT.Get_size()
11  position_fichier = rank * nb_valeurs * nb_octets_entier
12  requete = descripteur.Iread_at(position_fichier, valeurs)
13
14  nb_iterations = 0
15  termine = False
16  while nb_iterations < 5000 and not termine:
17      nb_iterations += 1
18      # Calculs recouvrant le temps demande par l'operation de lecture
19      # ...
20      termine = requete.Test()
21  if not termine:
22      requete.Wait()
23  print(f"Apres {nb_iterations} iterations, lecture processus "
24        f"{rank} : {valeurs}")
25  descripteur.Close()
```

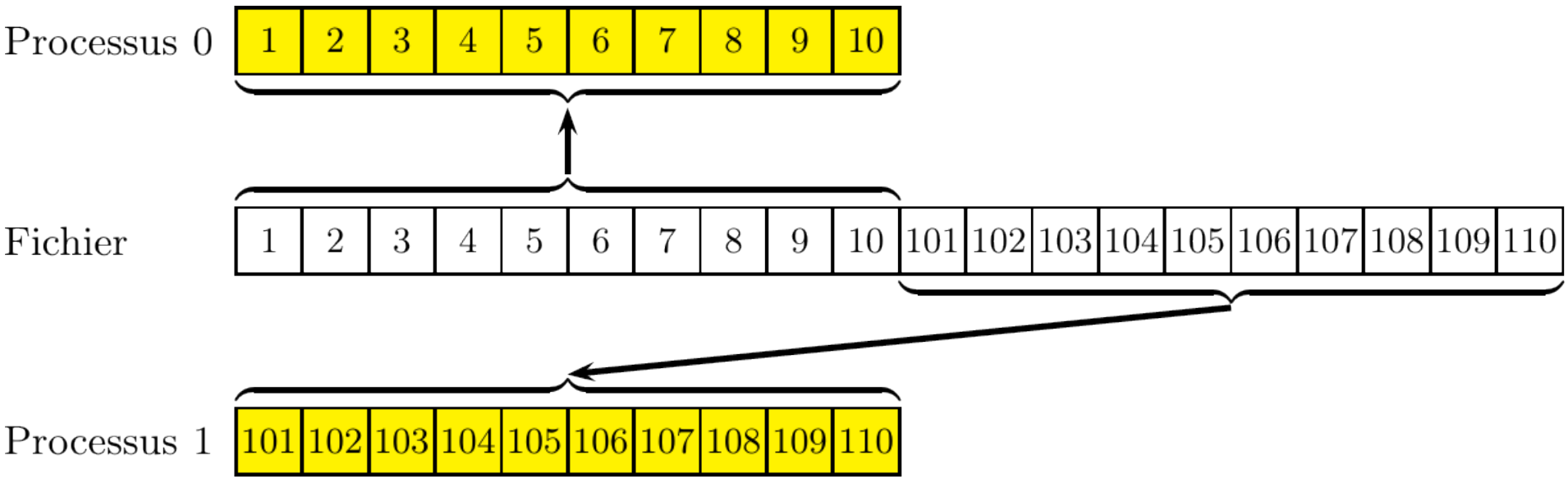


Figure 57 – Exemple d'utilisation de `MPI_File_iread_at()`

```
> mpiexec -n 2 python -m mpi4py iread_at.py

Apres 1 iterations, lecture processus 0 : [1  2  3  4  5  6  7  8  9  10]
Apres 1 iterations, lecture processus 1 : [101 102 103 104 105 106 107 108 109 110]
```

MPI-IO

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  nb_valeurs = 10
7  valeurs = np.zeros(nb_valeurs, dtype=np.int32)
8  temp = np.zeros(nb_valeurs, dtype=np.int32)
9
10 descripteur = MPI.File.Open(comm, "donnees.dat",
11                             MPI.MODE_WRONLY | MPI.MODE_CREATE)
12 temp = valeurs
13 nb_iterations = 0
14 requete = descripteur.Iwrite(temp)
15 while nb_iterations < 5000:
16     nb_iterations += 1
17     termine = requete.Test()
18     if termine:
19         temp = valeurs
20         descripteur.Seek(offset, MPI.SEEK_SET)
21         requete = descripteur.Iwrite(temp)
22 requete.Wait()
23 descripteur.Close()
```

Lectures/écritures collectives et non bloquantes

- Il est possible d'effectuer des opérations qui soient à la fois collectives et non bloquantes.
- L'accès est initié par `MPI_File_iread(_at)_all` ou `MPI_File_iwrite(_at)_all` et clôt par un test de complétude ou une mise en attente, mais pour les pointeurs partagés on a les opérations `MPI_File_read_ordered_begin` / `MPI_File_read_ordered_end` ou `MPI_File_write_ordered_begin` / `MPI_File_write_ordered_end`.
  Il ne peut y avoir qu'une seule opération collective non bloquante de ce type ("split") en cours à la fois par descripteur de fichier.
- Entre les deux phases de l'opération collective non-bloquante, il est possible de faire des opérations sur le fichier, mais la zone mémoire concernée par l'opération collective ne peut être modifiée.

MPI-IO

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  nb_valeurs = 10
7  valeurs = np.zeros(nb_valeurs, dtype=np.int32)
8  descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
9  descripteur.Read_ordered_begin([valeurs,4,MPI.INT])
10 print(f"Processus numero {rank}")
11 descripteur.Read_ordered_end([valeurs,4,MPI.INT])
12 print(f"Lecture processus {rank} : {valeurs[0]} {valeurs[1]} "
13       f"{valeurs[2]} {valeurs[3]}")
14 descripteur.Close()
```

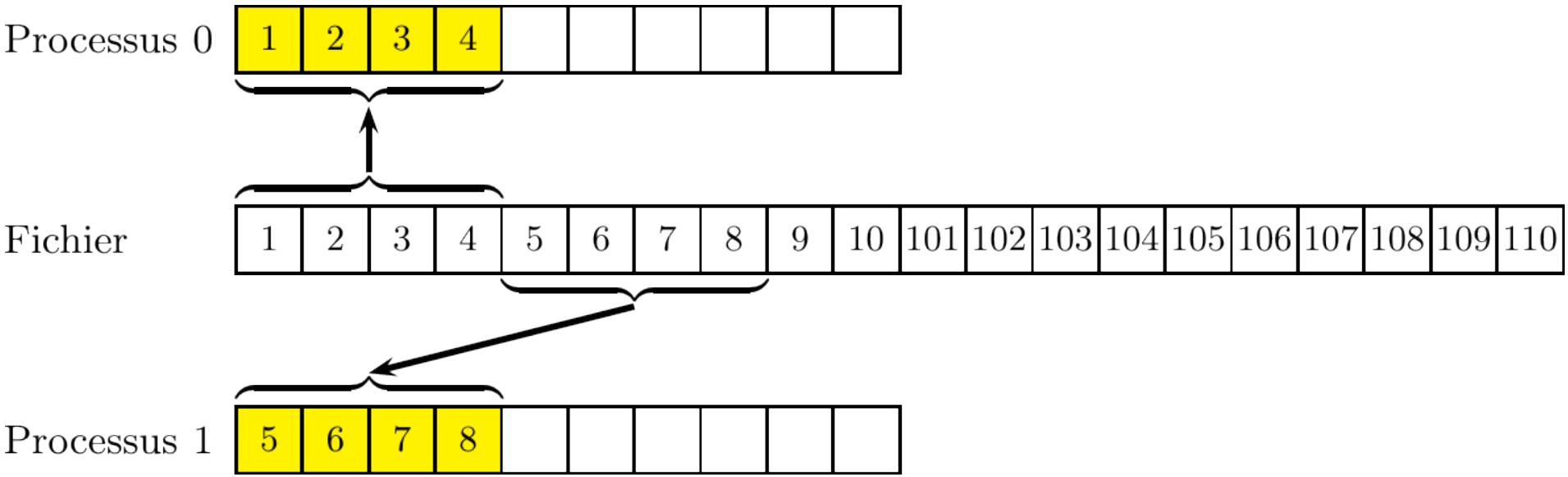


Figure 58 – Exemple d'utilisation de `MPI_File_read_ordered_begin/end()`

```
> mpiexec -n 2 python -m mpi4py ./readOrderedBeginEnd.py

Processus numero 0
Processus numero 1
Lecture processus 1 : 1 2 3 4
Lecture processus 0 : 5 6 7 8
```

T.P. MPI – Exercice 7 : Lecture d'un fichier en mode parallèle

- On dispose du fichier binaire donnees.dat, constitué d'une suite de 484 valeurs entières
- En considérant un programme parallèle mettant en œuvre 4 processus, il s'agit de lire les 121 premières valeurs sur le processus 0, les 121 suivantes sur le processus 1, etc. et d'écrire celles-ci dans quatre fois quatre fichiers appelés fichier_XXX0.dat ··· fichier_XXX3.dat
- On emploiera pour ce faire 4 méthodes différentes, parmi celles présentées :
  - lecture via des déplacements explicites, en mode individuel ;
  - lecture via les pointeurs partagés, en mode collectif ;
  - lecture via les pointeurs individuels, en mode individuel ;
  - lecture via les pointeurs partagés, en mode individuel.
- Pour compiler utilisez la commande make, pour exécuter le code utilisez la commande make exe et pour vérifier les résultats utilisez la commande make verification qui génère des fichiers images correspondant aux quatre cas à traiter.

Version MPI

Version MPI

Il est possible de connaître la version de MPI utilisée. La version est représentée par deux entiers `MPI.VERSION` et `MPI.SUBVERSION`.

```
from mpi4py import MPI

print(f"Version MPI : {MPI.VERSION}.{MPI.SUBVERSION}")
```

```
> mpiexec -n 1 python -m mpi4py version.py

Version MPI : 3.1
```

MPI 4.0

Ajout

- Grand nombre
- Communication par morceaux
- MPI Session
- Autres

Grand nombre

- Les nombres d'éléments étaient de type `integer` ou `int`.
- MPI 4.0 ajoute des fonctions avec le type `MPI_Count` à la place.
- En *C* ces nouvelles fonctions contiennent en plus `_c` à la fin de la fonction.

```
int MPI_Send(const void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Send_c(const void * buf, MPI_Count count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
```

- En *Fortran* les nombres de type `integer` peuvent être remplacés par `integer(kind=MPI_COUNT_KIND)`.
- Uniquement disponible avec le module `mpi_f08`.
- Pas de changement de nom grâce au polymorphisme.

```
MPI_Send(buf,count,datatype,dest,tag,comm,ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN)                :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN)     :: datatype
TYPE(MPI_Comm), INTENT(IN)         :: comm
INTEGER, OPTIONAL, INTENT(OUT)     :: ierror

MPI_Send(buf,count,datatype,dest,tag,comm,ierror)
TYPE(*), DIMENSION(..), INTENT(IN)        :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN)            :: datatype
INTEGER, INTENT(IN)                       :: dest, tag
TYPE(MPI_Comm), INTENT(IN)                :: comm
INTEGER, OPTIONAL, INTENT(OUT)            :: ierror
```

Communication par morceaux

- Contributions multiples à une communication
- Utile pour l'hybride
- Initialisation avec `MPI_Psend_init()` ou `MPI_Precv_init()` en fournissant le nombre de partitions et le nombre d'éléments par partition
- `MPI_Start()` pour démarrer la communication
- `MPI_Pready()` pour signaler qu'une partition est prête à être envoyée.
- `MPI_Parrived()` permet de savoir si une partition a été reçue
- `MPI_Wait()` pour terminer la communication
- Il n'est pas possible de faire un `MPI_Recv()` d'un `MPI_Psend_init()`

Sessions

- Permet de faire plusieurs `MPI_Init()`/`MPI_Finalize()`.
- `MPI_Session_init()` pour lancer une session.
- `MPI_Session_finalize()` pour terminer la session.
- Plus de notion de `MPI_COMM_WORLD`.
- Notion de *Process Sets* : `mpi://WORLD` et `mpi://SELF`.
- `MPI_Group_from_session_pset()` pour faire un groupe à partir d'un *pset*.
- `MPI_Comm_create_from_group()` pour faire un communicateur à partir d'un groupe.
- `MPI_Session_get_num_psets()` permet d'obtenir le nombre de *pset* disponibles.
- `MPI_Session_get_nth_pset()` permet d'avoir le nom d'un *pset* disponible.

Autres

- Ajout de `MPI_Isendrecv` et `MPI_Isendrecv_replace`.
- Ajout de `MPI_ERRORS_ABORT`
- Ajout de l'option `mpi_initial_errhandler` pour *mpiexec* afin de spécifier le gestionnaire d'erreur par défaut.

MPI 4.1

Ajout

- Le fichier mpif.h devient obsolète

MPI 5.0

Ajout

- Définition d'une ABI. L'intérêt d'une ABI est de pouvoir exécuter un programme en utilisant une bibliothèque MPI différente de celle utilisée lors de la compilation.

MPI-IO Vues

MPI-IO Vues

Définition des vues

- C'est un mécanisme permettant de décrire un schéma d'accès aux fichiers.
- Une vue est définie par trois variables : un déplacement initial, un type élémentaire de données et un motif.
- L'accès aux fichiers s'effectue par répétition du motif, une fois le positionnement initial effectué.

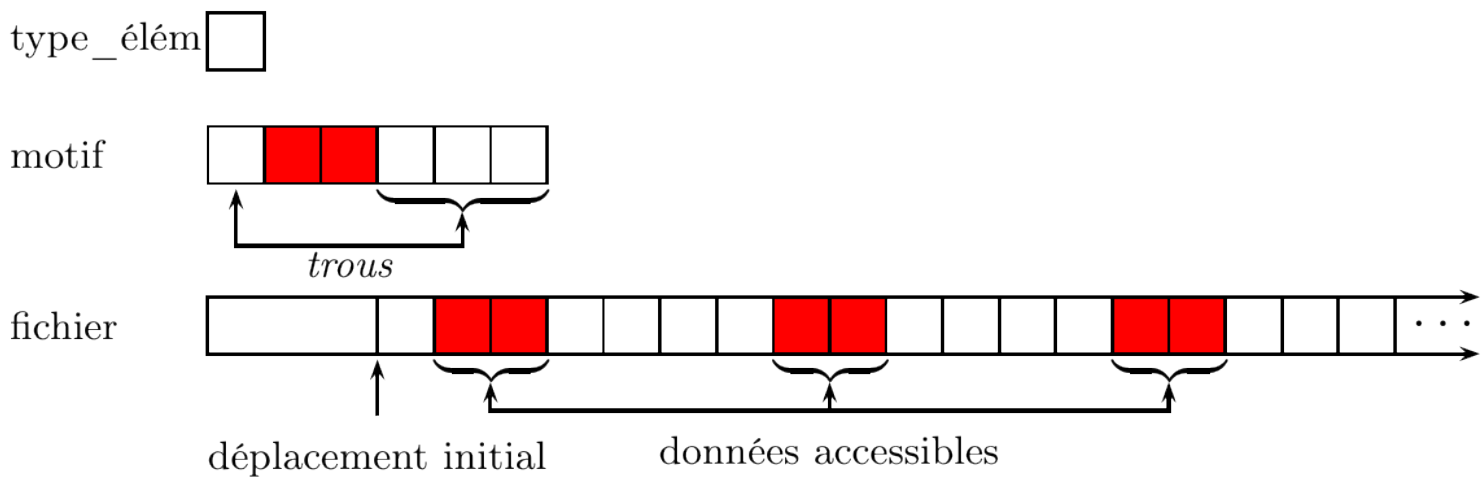


Figure 59 – Type élémentaire de donnée et motif

MPI-IO Vues

Définition des vues

- Les vues sont construites à l'aide de types dérivés MPI.
- Il est possible de définir des trous dans une vue, de façon à ne pas tenir compte de certaines parties des données.
- La vue par défaut consiste en une simple suite d'octets (déplacement initial nul, type_élém et motif égaux à `MPI_BYTE`).

Vues multiples

- Un processus donné peut définir et utiliser successivement plusieurs vues d'un même fichier.
- Les processus peuvent avoir des vues différentes du fichier, de façon à accéder à des parties complémentaires de celui-ci.

MPI-IO Vues

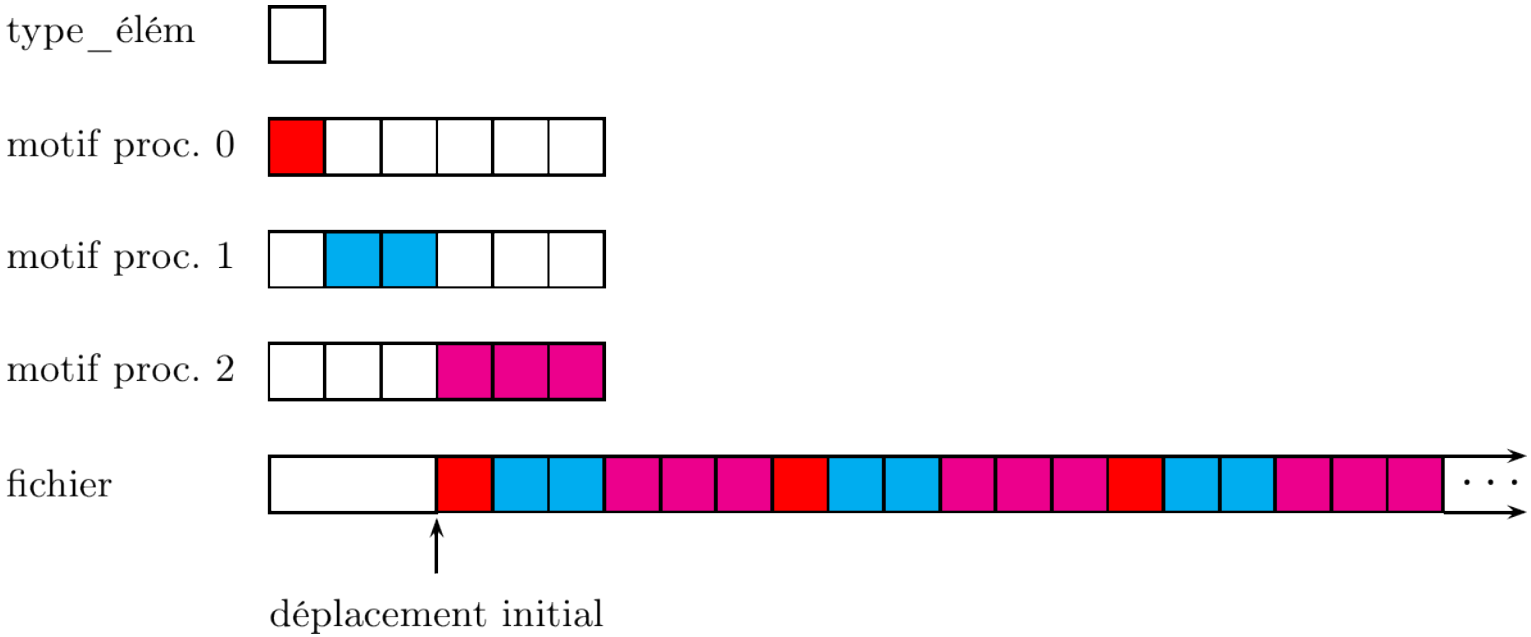


Figure 60 – Exemple de définition de motifs différents selon les processus

Remarques :

- Un pointeur partagé n'est utilisable avec une vue que si tous les processus ont la même vue.
- Si le fichier est ouvert en écriture, les zones décrites par les différentes vues ne peuvent se recouvrir, même partiellement.

MPI-IO Vues

Changement de la vue sur un fichier : `MPI_File_set_view()`

```
mpi4py.MPI.File.Set_view(disp=0, etype=BYTE, filetype=None, datarep='native',
                         info=INFO_NULL)
```

- C'est une opération collective à l'ensemble des processus impliqués dans l'accès au fichier. Chaque processus peut définir un disp et un filetype différent. L'étendue du type etype doit être identique.
- Les pointeurs individuels et le pointeur partagé sont réinitialisés au début de la vue, en tenant compte du déplacement initial.

Notes :

- Les types dérivés utilisés dans la vue doivent avoir été validés au préalable à l'aide du sous-programme `MPI_Type_commit()`.
- Il y a trois représentations possibles des données (mode) : "native", "internal" ou "external32".

Construction de sous-tableaux

Un type dérivé utile pour créer un motif est le type “subarray”, qu’on introduit ici. Ce type permet de créer un sous-tableau à partir d’un tableau et se définit via le sous-programme `MPI_Type_create_subarray()`.

Le profil d’un tableau est un vecteur dont chaque élément est le nombre d’éléments dans chaque dimension. Soit par exemple le tableau `T(10,0:5,-10:10)` (ou `T[10][6][21]`), son profil est le vecteur (10,6,21).

MPI-IO Vues / Types de données dérivés

```
# Retourne un type
mpi4py.MPI.Datatype.Create_subarray(sizes, subsizes, starts, order=ORDER_C)
```

Description des arguments

- `sizes` : profil du tableau à partir duquel on va extraire un sous-tableau
- `subsizes` : profil du sous-tableau
- `starts` : coordonnées de départ du sous-tableau dans le tableau, les indices du tableau commençant à 0
- `order` : ordre de stockage des éléments
  - `MPI.ORDER_FORTRAN` spécifie le mode de stockage en Fortran, c.-à-d. suivant les colonnes
  - `MPI.ORDER_C` spécifie le mode de stockage en C, c.-à-d. suivant les lignes

MPI-IO Vues / Types de données dérivés

Échanges entre 2 processus avec subarray

AVANT

1	2	3	4
5	6	7	8
9	10	11	12

Processus 0

-1	-2	-3	-4
-5	-6	-7	-8
-9	-10	-11	-12

Processus 1

APRÈS

1	-7	-8	4
5	-11	-12	8
9	10	11	12

Processus 0

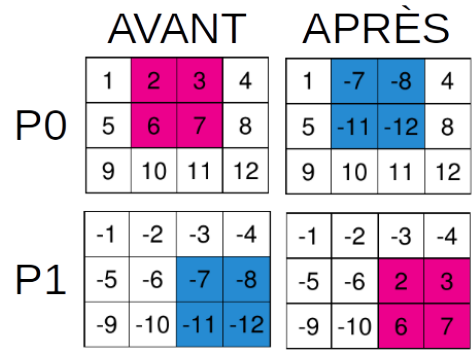
-1	-2	-3	-4
-5	-6	2	3
-9	-10	6	7

Processus 1

MPI-IO Vues / Types de données dérivés

Échanges entre 2 processus avec subarray : code

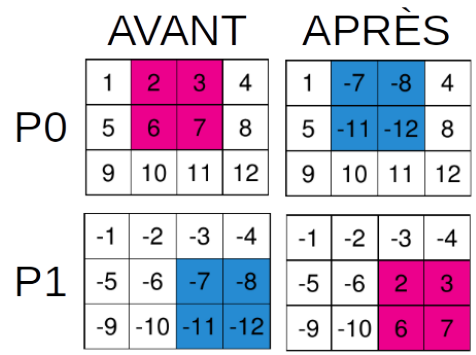
```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6
7  nb_lignes = 3
8  nb_colonnes = 4
9  sign = 1
10 etiquette = 1000
11
12 # Initialisation du tableau tab sur chaque processus
13 if rank == 1:
14     sign = -1
15 tab = np.zeros((nb_lignes, nb_colonnes), dtype=np.int32)
16 for i in range(nb_lignes):
17     for j in range(nb_colonnes):
18         tab[i, j] = sign * (1 + i * nb_colonnes + j)
```



MPI-IO Vues / Types de données dérivés

Échanges entre 2 processus avec subarray : code (suite)

```
19  # Profil du tableau tab a partir duquel on va extraire un sous tableau
20  profil_tab = [nb_lignes, nb_colonnes]
21  # Profil du sous tableau
22  profil_sous_tab = [2, 2]
23  # Coordonnees de debut du sous tableau
24  coord_debut = [rank, rank + 1]
25  # Creation du type type_sous_tab
26  type_sous_tab = MPI.INT.Create_subarray(profil_tab, profil_sous_tab,
27                                          coord_debut)
28  type_sous_tab.Commit()
29  # Permutation du sous-tableau
30  comm.Sendrecv_replace([tab, 1, type_sous_tab], (rank + 1) % 2, etiquette,
31                        (rank + 1) % 2, etiquette)
32  type_sous_tab.Free()
```



Exemple 1 : lecture d'un fichier par blocs de deux éléments

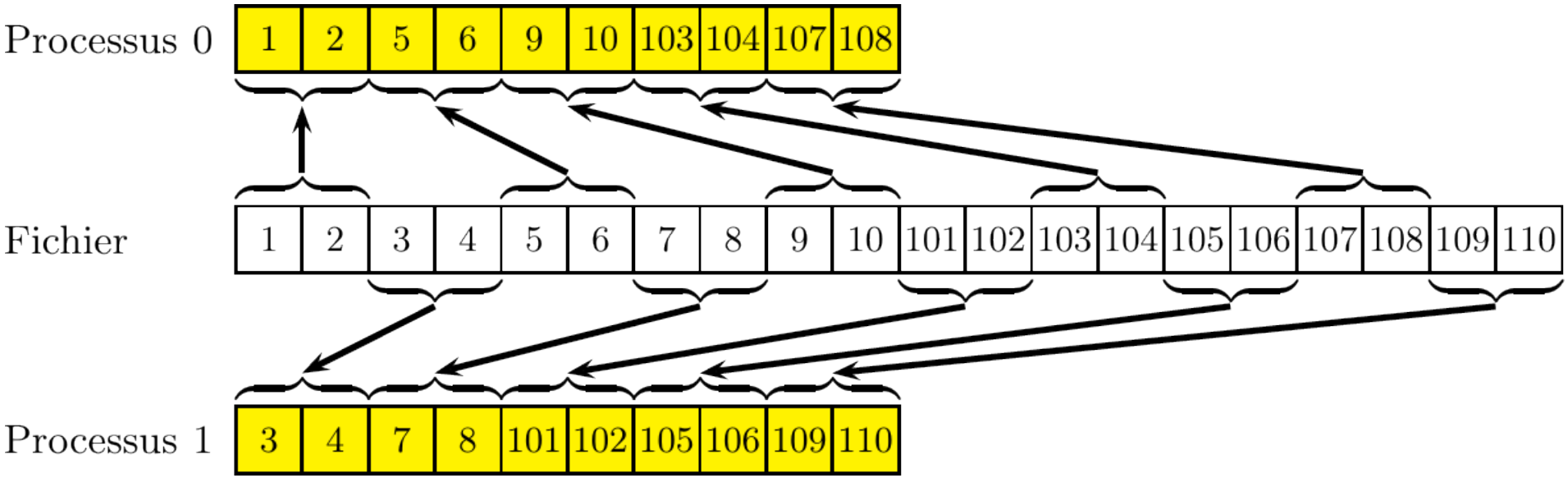


Figure 61 – Exemple 1 : lecture d'un fichier par blocs de deux éléments

```
> mpiexec -n 2 python -m mpi4py read_view01.py

Lecture processus 1 : [3 4 7 8 101 102 105 106 109 110]
Lecture processus 0 : [1 2 5 6   9  10 103 104 107 108]
```

Exemple 1 (suite)

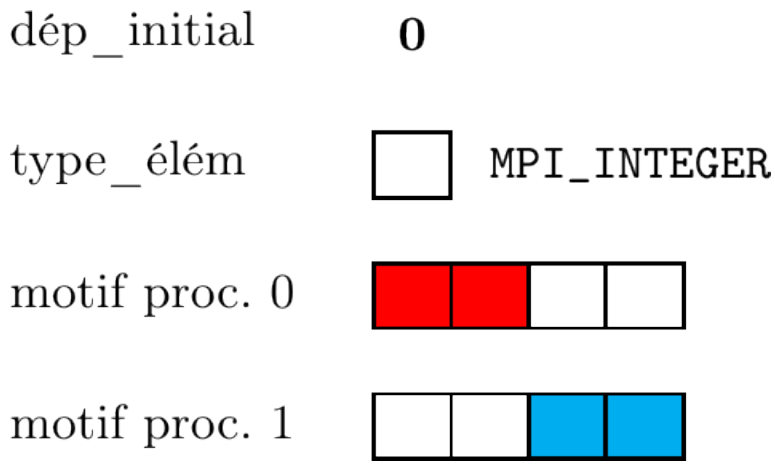
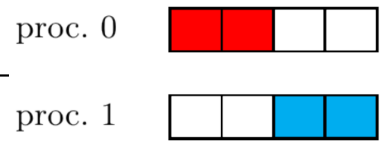


Figure 62 – Exemple 1 (suite) : création des vues sur le fichier

```
1  if rank == 0:
2      coord = 0
3  elif rank == 1:
4      coord = 2
5  motif = MPI.INT.Create_subarray([4],[2],[coord])
6  motif.Commit()
7  descripteur.Set_view(0,MPI.INT,motif)
```

MPI-IO Vues

Exemple 1 : code complet




```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6
7  if rank == 0:
8      coord = 0
9  elif rank == 1:
10     coord = 2
11 motif = MPI.INT.Create_subarray([4], [2], [coord])
12 motif.Commit()
13
14 descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
15 descripteur.Set_view(0, MPI.INT, motif)
16 valeurs = np.empty(10, dtype=np.int32)
17 descripteur.Read(valeurs)
18 print(f"Lecture processus {rank} : {valeurs}")
19 descripteur.Close()
```

MPI-IO Vues

Exemple 2 : utilisation successive de plusieurs vues

Figure 63 – Exemple 2 : utilisation successive de plusieurs vues

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  valeurs = np.empty(10, dtype=np.int32)
```

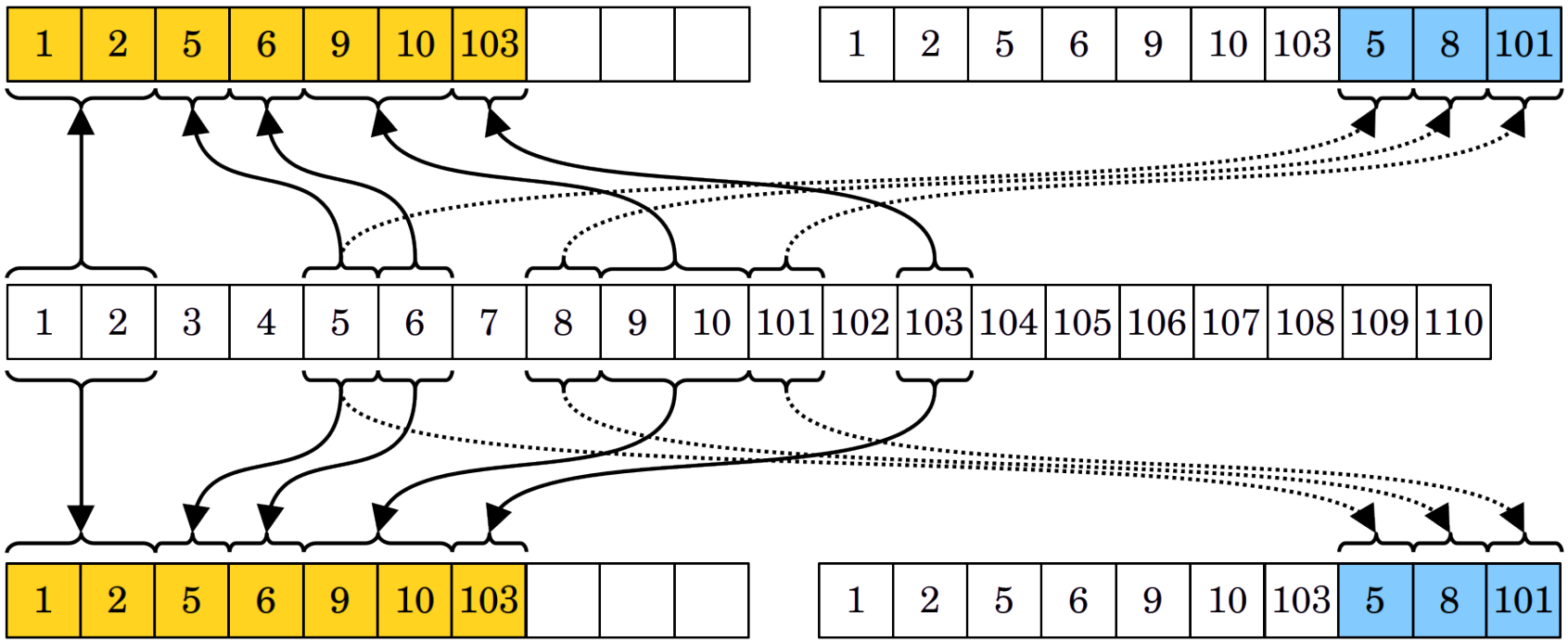
MPI-IO Vues

Exemple 2 (suite du code)

```
 7  motif_1 = MPI.INT.Create_subarray([4], [2], [0])
 8  motif_1.Commit()
 9  motif_2 = MPI.INT.Create_subarray([3], [1], [2])
10  motif_2.Commit()
11
12  descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
13  descripteur.Set_view(0, MPI.INT, motif_1)
14  descripteur.Read([valeurs, 3, MPI.INT])
15  descripteur.Read([valeurs[3:], 4, MPI.INT])
16  nb_octets_entier = MPI.INT.Get_size()
17  descripteur.Set_view(2*nb_octets_entier, MPI.INT, motif_2)
18  descripteur.Read([valeurs[7:], 3, MPI.INT])
19  print(f"Lecture processus {rank} : {valeurs}")
20  descripteur.Close()
```

MPI-IO Vues

Exemple 2 : illustration




```
> mpiexec -n 2 python -m mpi4py read_view02.py

Lecture processus 1 : [1 2 5 6 9 10 103 5 8 101]
Lecture processus 0 : [1 2 5 6 9 10 103 5 8 101]
```

Exemple 3 : gestion des trous dans les types de données

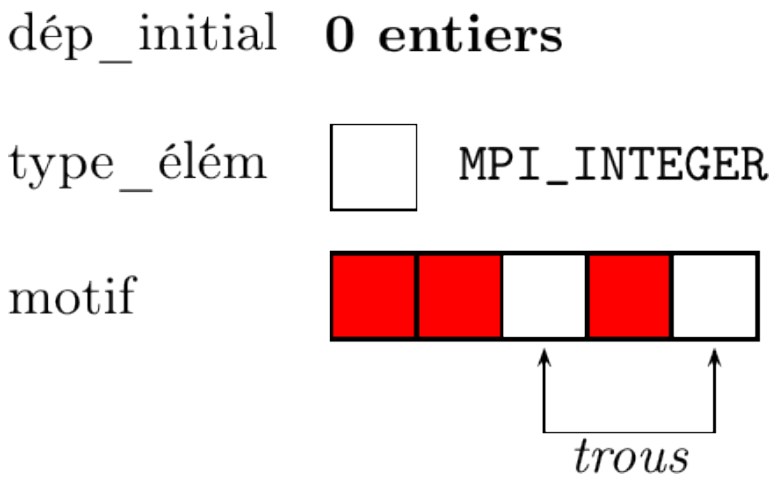


Figure 64 – Exemple 3 : gestion des trous dans les types de données

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6
7  valeurs = np.empty(9, dtype=np.int32)
```

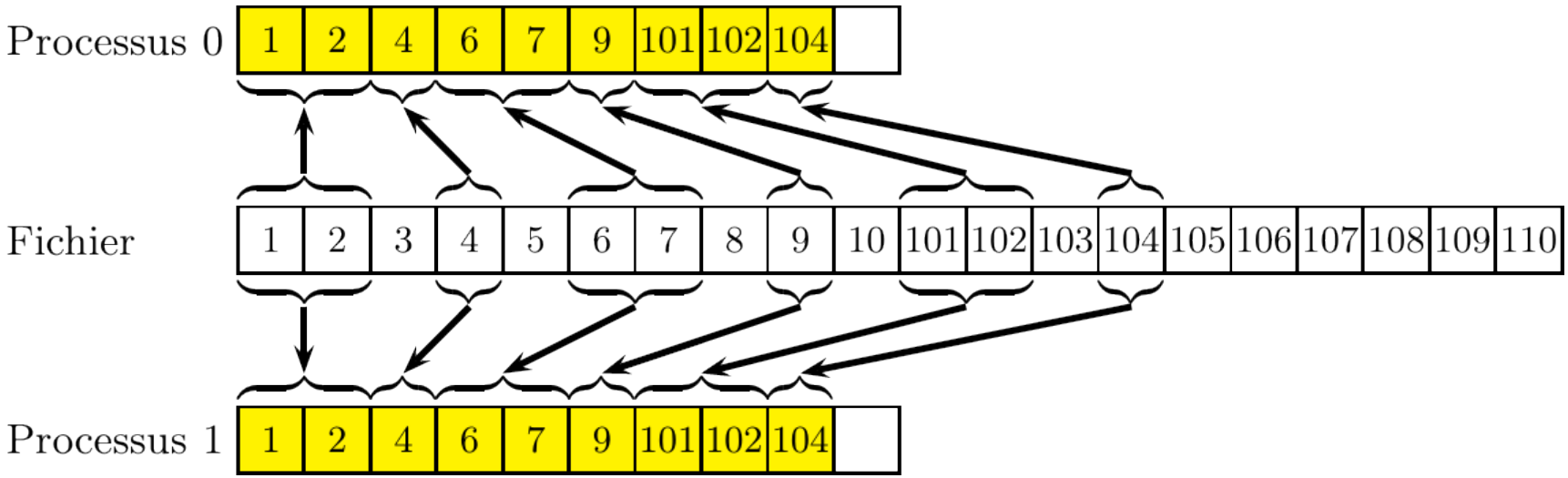
MPI-IO Vues

Exemple 3 (suite du code)

```
motif = MPI.INT.Create_indexed([2, 1], [0, 3])
nb_octets_entier = MPI.INT.Get_size()
_, extent = motif.Get_extent()
motif = motif.Create_resized(0, extent + nb_octets_entier)
motif.Commit()

descripteur = MPI.File.Open(comm, "donnees.dat", MPI.MODE_RDONLY)
descripteur.Set_view(0, MPI.INT, motif)
descripteur.Read(valeurs)
print(f"Lecture processus {rank} : {valeurs}")
descripteur.Close()
```

MPI-IO Vues

Exemple 3 : illustration




```
> mpiexec -n 2 python -m mpi4py read_view03.py

Lecture processus 0 : [  1   2   4   6   7   9 101 102 104]
Lecture processus 1 : [  1   2   4   6   7   9 101 102 104]
```

MPI-IO Vues

Conclusion sur les MPI-IO et les vues

MPI-IO offre une interface de haut niveau et un ensemble de fonctionnalités très riche. Il est possible de réaliser des opérations complexes et de tirer parti des optimisations implémentées dans la bibliothèque. MPI-IO offre aussi une bonne portabilité.

Conseils

- L'utilisation des sous-programmes à positionnement explicite dans les fichiers doit être réservée à des cas particuliers, l'utilisation implicite de pointeurs individuels avec des vues offrant une interface de plus haut niveau.
- Lorsque les opérations font intervenir l'ensemble des processus (ou un sous-ensemble identifiable par un sous-communicateur MPI), il faut généralement privilégier la forme collective des opérations.
- Exactement comme pour le traitement des messages lorsque ceux-ci représentent une part importante de l'application, le non-bloquant est une voie privilégiée d'optimisation à mettre en œuvre par les programmeurs, mais ceci ne doit être implémenté qu'après qu'on se soit assuré du comportement correct de l'application en mode bloquant.

Conclusion

Conclusion

- Utiliser les communications point-à-point bloquantes, ceci avant de passer aux communications non-bloquantes. Il faudra alors essayer de faire du recouvrement calcul/communications.
- Utiliser les fonctions d'entrées-sorties bloquantes, ceci avant de passer aux entrées-sorties non-bloquantes. De même, il faudra alors faire du recouvrement calcul/entrées-sorties.
- Écrire les communications comme si les envois étaient synchrones (`MPI_Ssend()`).
- Éviter les barrières de synchronisation (`MPI_Barrier()`), surtout sur les fonctions collectives qui sont bloquantes.
- La programmation mixte MPI/OpenMP peut apporter des gains d'extensibilité, mais pour que cette approche fonctionne bien, il est évidemment nécessaire d'avoir de bonnes performances OpenMP à l'intérieur de chaque processus MPI. Un cours est dispensé à l'IDRIS (`https://cours.idris.fr`).

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

On considère l'équation de Poisson suivante :

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} & = f(x,y) \quad \text{dans } [0,1]\text{x}[0,1] \\ u(x,y) & = 0. \quad \text{sur les frontières} \\ f(x,y) & = 2.\left(x^2 - x + y^2 - y\right) \end{cases}$$

La solution exacte est connue : $u_{exacte}(x,y) = xy\,(x-1)\,(y-1)$

On va résoudre cette équation avec une méthode de décomposition de domaine :

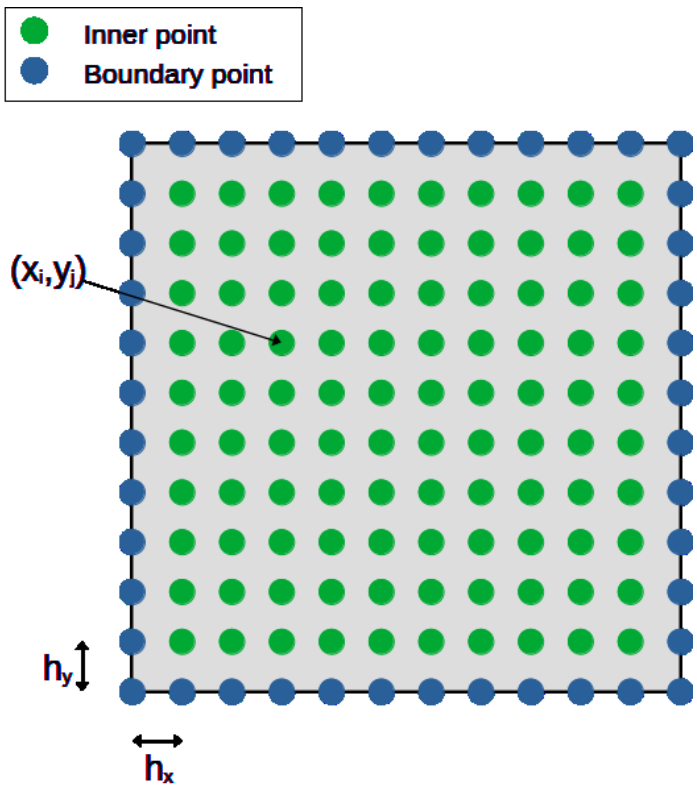
- L'équation est discretisée sur le domaine via la méthode des différences finies.
- Le système obtenu est résolu avec un solveur suivant la méthode de Jacobi.
- Le domaine global est découpé en sous domaines. Chaque processus résout l'équation sur un des sous-domaines et stocke la solution dans un fichier.

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

On discrétise le domaine d'étude selon une grille régulière constituée d'un ensemble de points de coordonnées (x_i, y_j) sur lesquels la solution sera approximée. On définit :

- *ntx* le nombre de points intérieurs suivant *x*
- *nty* le nombre de points intérieurs suivant *y*

Remarque : il y a au total $(ntx + 2) \times (nty + 2)$ points en comptant les points au bord.



- $h_x = \frac{1}{ntx+1}$ le pas suivant *x*
- $h_y = \frac{1}{nty+1}$ le pas suivant *y*
- $x_i = ih_x$ pour $i \in \{0, ..., ntx + 1\}$
  les coordonnées des points selon *x*
- $y_j = jh_y$ pour $j \in \{0, ..., nty + 1\}$
  les coordonnées des points selon *y*
- $u_{i,j} = u(x_i, y_j)$ et $f_{i,j} = f(x_i, y_j)$
  pour $(i, j) \in \{0, ..., ntx+1\} \times \{0, ..., nty+1\}$

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

En utilisant la méthode des différences finies, les dérivées partielles $\frac{\partial^2 u}{\partial x^2}$ et $\frac{\partial^2 u}{\partial y^2}$ en un point (x_i, y_j) peuvent être approximées en fonction des valeurs de u dans un voisinage proche (c.-à-d. pour h_x et h_y petits) :

$$\underbrace{\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}}_{f_{i,j}} \simeq \frac{\frac{u_{i+1,j}-u_{i,j}}{h_x} - \frac{u_{i,j}-u_{i-1,j}}{h_x}}{h_x} + \frac{\frac{u_{i,j+1}-u_{i,j}}{h_y} - \frac{u_{i,j}-u_{i,j-1}}{h_y}}{h_y}.$$

Un peu d'algèbre mène à

$$u_{i,j} \simeq \frac{h_x^2 h_y^2}{2(h_x^2+h_y^2)} \left[\frac{1}{h_x^2}\left(u_{i+1,j}+u_{i-1,j}\right) + \frac{1}{h_y^2}\left(u_{i,j+1}+u_{i,j-1}\right) - f_{i,j} \right].$$

La méthode de Jacobi est une méthode iterative qui revient à faire évoluer $u_{i,j}$ à chaque itération selon la formule

$$u_{i,j}^{n+1} = \frac{h_x^2 h_y^2}{2(h_x^2+h_y^2)} \left[\frac{1}{h_x^2}\left(u_{i+1,j}^{n}+u_{i-1,j}^{n}\right) + \frac{1}{h_y^2}\left(u_{i,j+1}^{n}+u_{i,j-1}^{n}\right) - f_{i,j} \right].$$

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

- En parallèle, chaque processus résout l'équation sur un des sous-domaines.
- Les valeurs aux interfaces des sous-domaines doivent être échangées entre les processus voisins.
- On utilise des cellules fantômes, ces cellules servent de buffer de réception pour les échanges entre voisins.

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

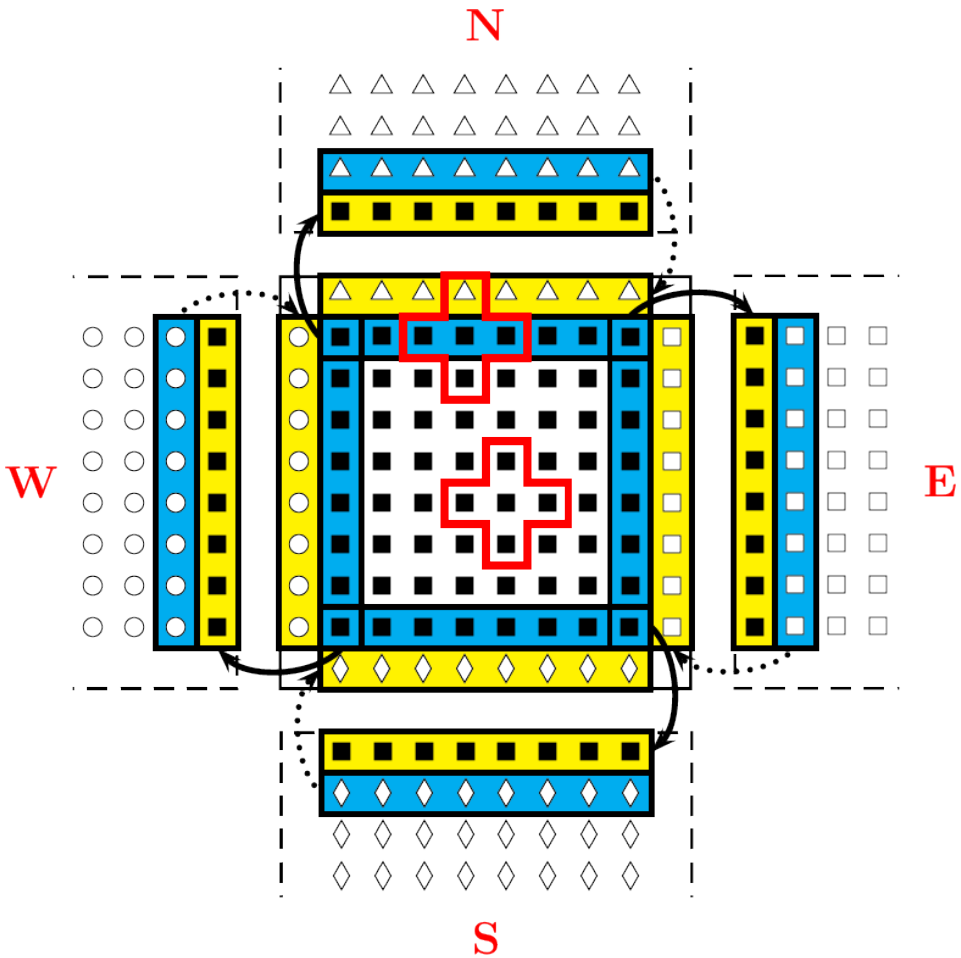


Figure 65 – Échange de points aux interfaces

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

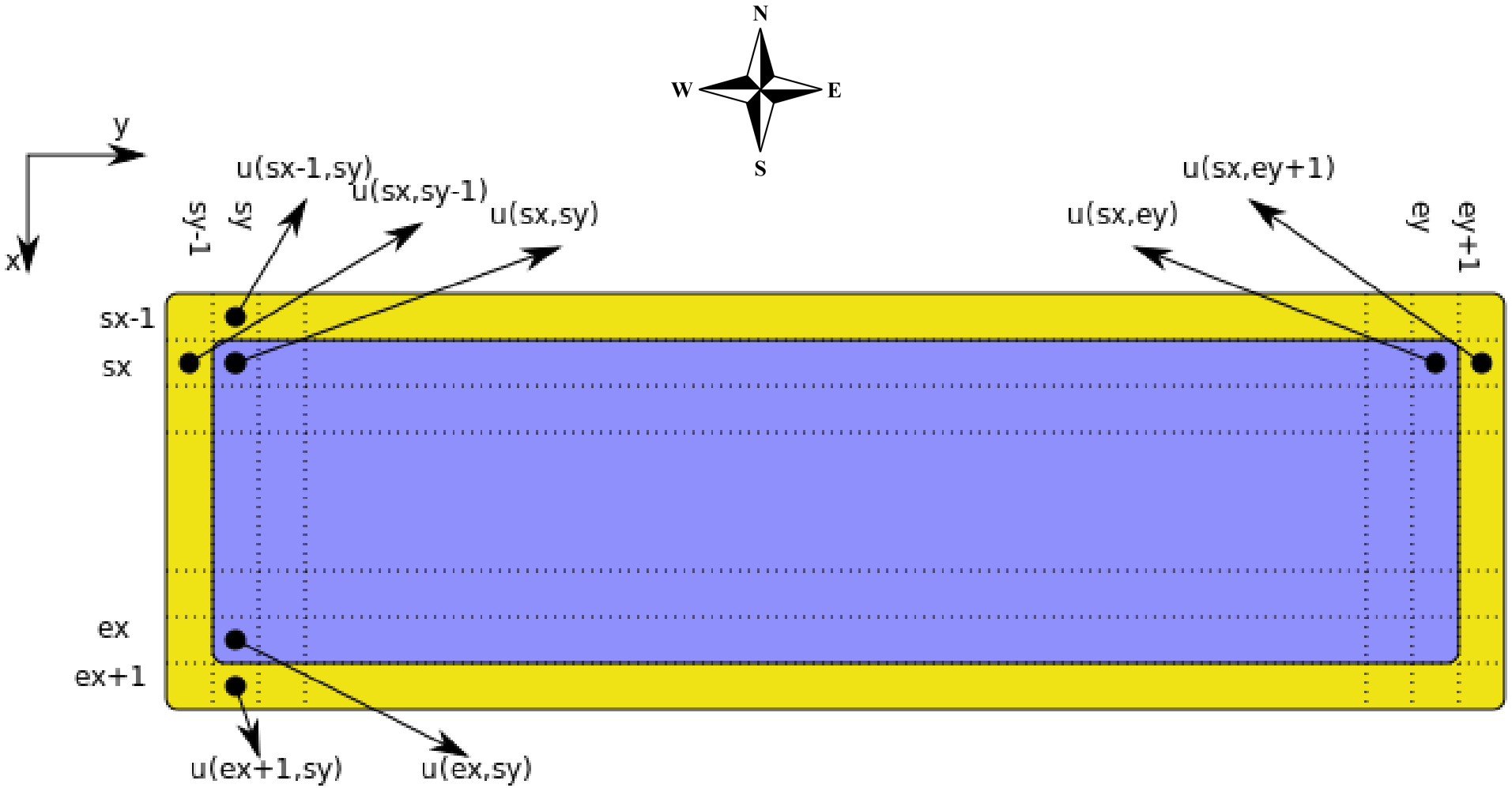


Figure 66 – Numérotation des points dans les différents sous-domaines

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

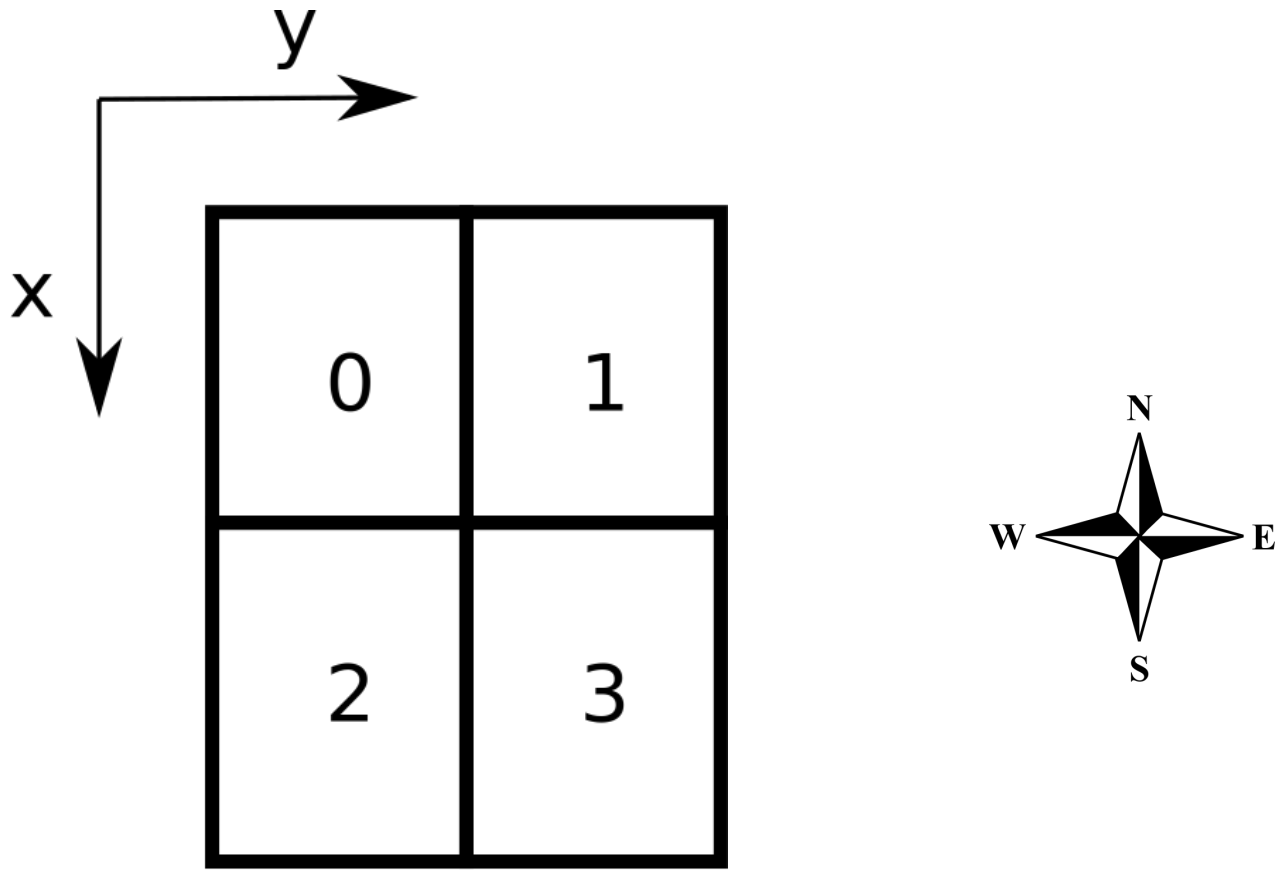


Figure 67 – Rang correspondant aux différents sous-domaines

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

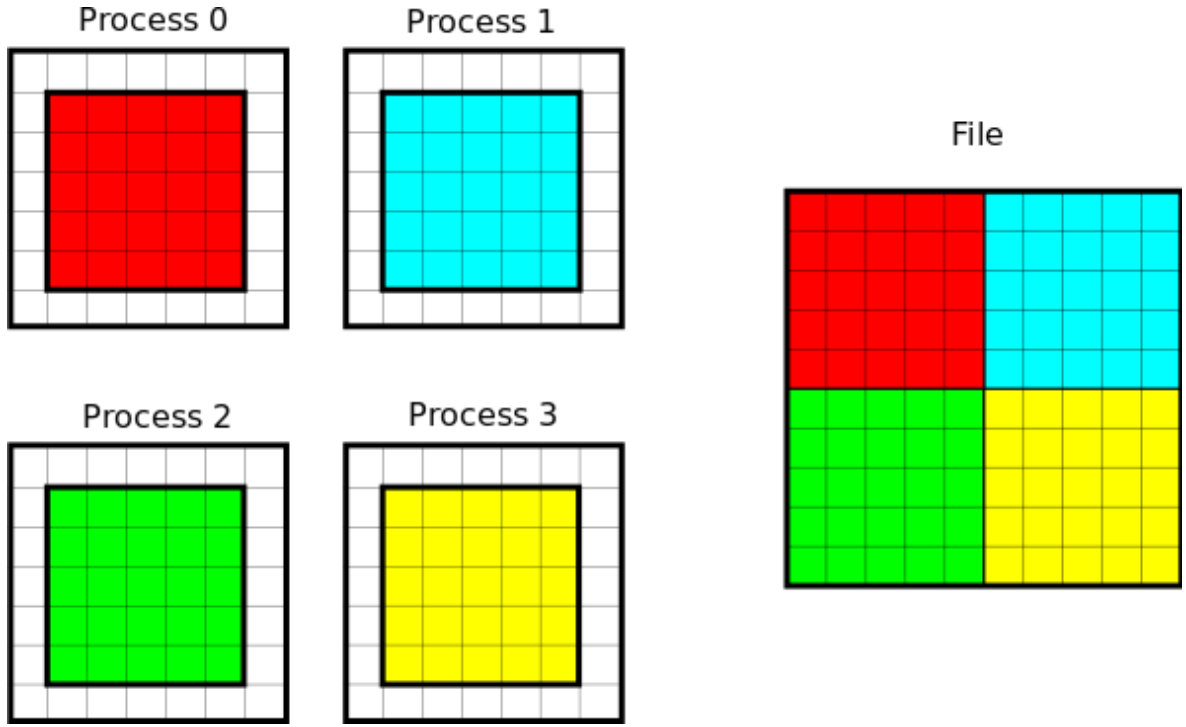


Figure 68 – Ecriture de la matrice u globale dans un fichier

Les processus écrivent la solution globale dans un fichier. Pour cela il faut :

- créer une vue, pour ne voir dans le fichier que la partie de la matrice u globale que l'on possède ;
- définir un type afin d'écrire la matrice u locale (sans les cellules fantômes) ;
- appliquer la vue au fichier ;
- faire l'écriture en une fois.

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

- Un squelette de la version parallèle est proposé : il s'agit d'un programme principal (`poisson.py`) et de plusieurs sous-programmes. Les actions suivantes sont à implémenter **dans le fichier `parallel.py`**.

  - Initialiser l'environnement MPI ;
  - Créer la topologie cartésienne 2D ;
  - Déterminer les indices de tableau pour chaque sous-domaine ;
  - Déterminer les 4 processus voisins d'un processus traitant un sous-domaine donné ;
  - Créer deux types dérivés *type_ligne* et *type_colonne* ;
  - échanger les valeurs aux interfaces avec les autres sous-domaines ;
  - Calculer l'erreur globale. Lorsque l'erreur globale sera inférieure à une valeur donnée (précision machine par exemple), alors on considérera qu'on a atteint la solution ;
  - Reformer la matrice u globale (identique à celle obtenue avec la version monoprocesseur) dans un fichier `donnees.dat`.

- Pour compiler utilisez la commande make, pour exécuter le code utilisez la commande make exe. Pour vérifier les résultats, utilisez la commande make verification qui exécute un programme de relecture du fichier `donnees.dat` puis le compare avec la version monoprocesseur.