

## Message Passing Interface (MPI)

Jalel Chergui  
Isabelle Dupays  
Denis Girou  
Pierre-François Lavallée  
Dimitri Lecas  
Philippe Wautelet

1 – Introduction	8
1.1 – Définitions	8
1.2 – Concepts de l'échange de messages	11
1.3 – Historique	14
1.4 – Bibliographie	16
2 – Environnement	20
2.1 – Description	20
2.2 – Exemple	23
3 – Communications point à point	24
3.1 – Notions générales	24
3.2 – Types de données de base	27
3.3 – Autres possibilités	29
3.4 – Exemple : anneau de communication	35
4 – Communications collectives	40
4.1 – Notions générales	40
4.2 – Synchronisation globale : MPI_BARRIER()	42
4.3 – Diffusion générale : MPI_BCAST()	43

4.4 – Diffusion sélective : <code>MPI_SCATTER()</code> . . . . .	45
4.5 – Collecte : <code>MPI_GATHER()</code> . . . . .	48
4.6 – Collecte générale : <code>MPI_ALLGATHER()</code> . . . . .	51
4.7 – Collecte : <code>MPI_GATHERV()</code> . . . . .	54
4.8 – Échanges croisés : <code>MPI_ALLTOALL()</code> . . . . .	57
4.9 – Réductions réparties . . . . .	60
4.10 – Compléments . . . . .	67
5 – Copies de mémoire à mémoire . . . . .	68
5.1 – Introduction . . . . .	68
5.1.1 – Rappel : concept de l'échange de messages . . . . .	69
5.1.2 – Concept des copies de mémoire à mémoire . . . . .	70
5.1.3 – Approche RMA de <code>MPI</code> . . . . .	71
5.2 – Notion de fenêtre mémoire . . . . .	72
5.3 – Transfert des données . . . . .	77
5.4 – Achèvement du transfert : la synchronisation . . . . .	81
5.4.1 – Synchronisation de type cible active . . . . .	82
5.4.2 – Synchronisation de type cible passive . . . . .	92

5.5 – Conclusions . . . . .	96
6 – Types de données dérivés . . . . .	97
6.1 – Introduction . . . . .	97
6.2 – Types contigus . . . . .	99
6.3 – Types avec un pas constant . . . . .	100
6.4 – Autres sous-programmes . . . . .	102
6.5 – Exemples . . . . .	103
6.6 – Types homogènes à pas variable . . . . .	109
6.7 – Construction de sous-tableaux . . . . .	116
6.8 – Types hétérogènes . . . . .	122
6.9 – Sous-programmes annexes . . . . .	126
6.10 – Conclusion . . . . .	130
7 – Optimisations . . . . .	131
7.1 – Introduction . . . . .	131
7.2 – Programme modèle . . . . .	132
7.3 – Temps de communication . . . . .	135
7.4 – Quelques définitions . . . . .	136

7.5 – Que fournit MPI ?	140
7.6 – Envoi synchrone bloquant	142
7.7 – Envoi synchrone non-bloquant	144
7.8 – Conseils	148
8 – Communicateurs	152
8.1 – Introduction	152
8.2 – Communicateur par défaut	153
8.3 – Exemple	154
8.4 – Groupes et communicateurs	157
8.5 – Communicateur issu d'un autre	159
8.6 – Topologies	164
8.6.1 – Topologies cartésiennes	166
8.6.2 – Subdiviser une topologie cartésienne	181
9 – MPI-IO	187
9.1 – Introduction	187
9.1.1 – Présentation	187
9.1.2 – Enjeux	189

9.1.3 – Définitions . . . . .	192
9.2 – Gestion de fichiers . . . . .	194
9.3 – Lectures/écritures : généralités . . . . .	197
9.4 – Lectures/écritures individuelles . . . . .	201
9.4.1 – Via des déplacements explicites . . . . .	201
9.4.2 – Via des déplacements implicites individuels . . . . .	206
9.4.3 – Via des déplacements implicites partagés . . . . .	211
9.5 – Lectures/écritures collectives . . . . .	214
9.5.1 – Via des déplacements explicites . . . . .	215
9.5.2 – Via des déplacements implicites individuels . . . . .	217
9.5.3 – Via des déplacements implicites partagés . . . . .	223
9.6 – Positionnement explicite des pointeurs dans un fichier . . . . .	225
9.7 – Définition de vues . . . . .	228
9.8 – Lectures/écritures non bloquantes . . . . .	241
9.8.1 – Via des déplacements explicites . . . . .	242
9.8.2 – Via des déplacements implicites individuels . . . . .	245
9.8.3 – Lectures/écritures collectives et non bloquantes . . . . .	247

9.9 – <b>Conseils</b>	250
10 – <b>Conclusion</b>	251
11 – <b>Annexes</b>	252
11.1 – <b>Communications collectives</b>	253
11.2 – <b>Types de données dérivés</b>	255
11.2.1 – <b>Distribution d’un tableau sur plusieurs processus</b>	255
11.2.2 – <b>Types dérivés numériques</b>	268
11.3 – <b>Optimisations</b>	272
11.4 – <b>Communicateurs</b>	276
11.4.1 – <b>Intra et intercommunicateurs</b>	276
11.4.2 – <b>Graphe de processus</b>	284
11.5 – <b>Gestion de processus</b>	292
11.5.1 – <b>Introduction</b>	292
11.5.2 – <b>Mode maître-ouvriers</b>	294
11.5.3 – <b>Mode client-serveur</b>	309
11.5.4 – <b>Suppression de processus</b>	315
11.5.5 – <b>Compléments</b>	317
11.6 – <b>MPI-IO</b>	318

## 1 – Introduction

### 1.1 – Définitions

- ❶ Le 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 centrale allouée au processus ;
  - ☞ un processus s'exécute sur un processeur physique de la machine.

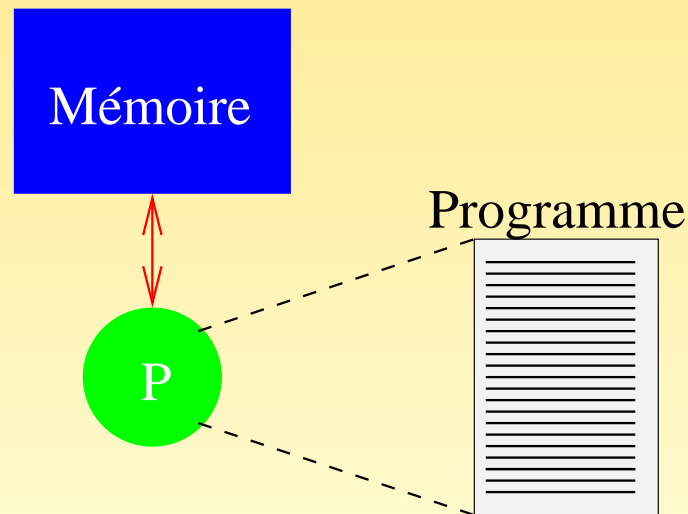


FIG. 1 – Modèle de programmation séquentiel



② Dans le modèle de programmation par **échange de messages** :

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

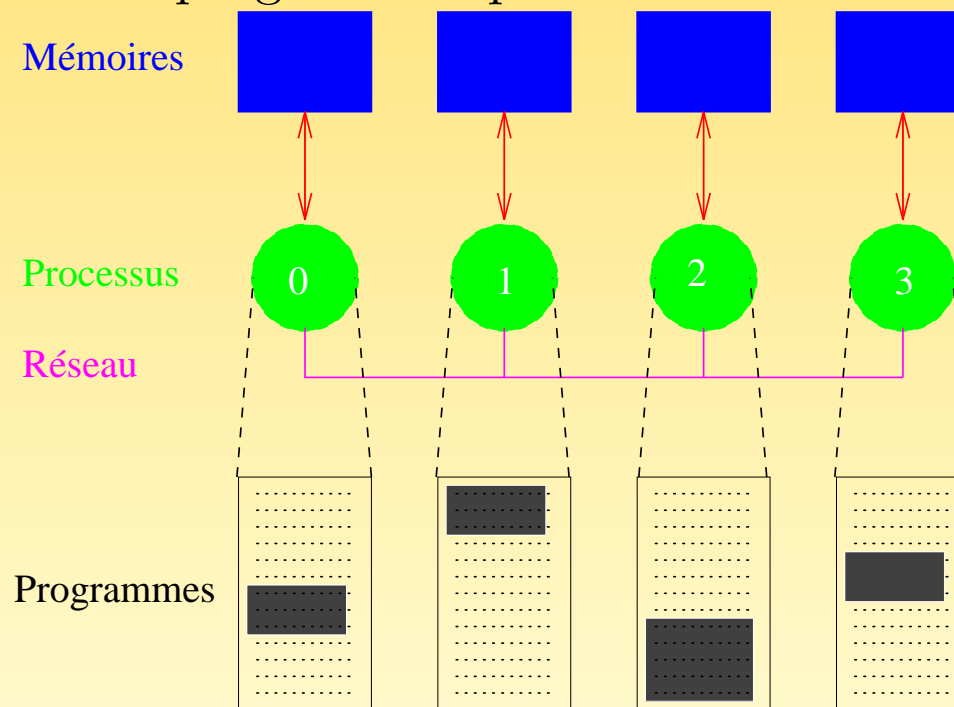


FIG. 2 – Modèle de programmation par échange de messages

## ③ Le modèle d'exécution **SPMD** :

- ➡ **Single Program, Multiple Data** ;
- ➡ le même programme est exécuté par tous les processus ;
- ➡ toutes les machines supportent ce modèle de programmation et certaines ne supportent que celui-là ;
- ➡ c'est un cas particulier du modèle plus général **MPMD** (**M**ultiple **P**rogram, **M**ultiple **D**ata), qu'il peut d'ailleurs émuler.

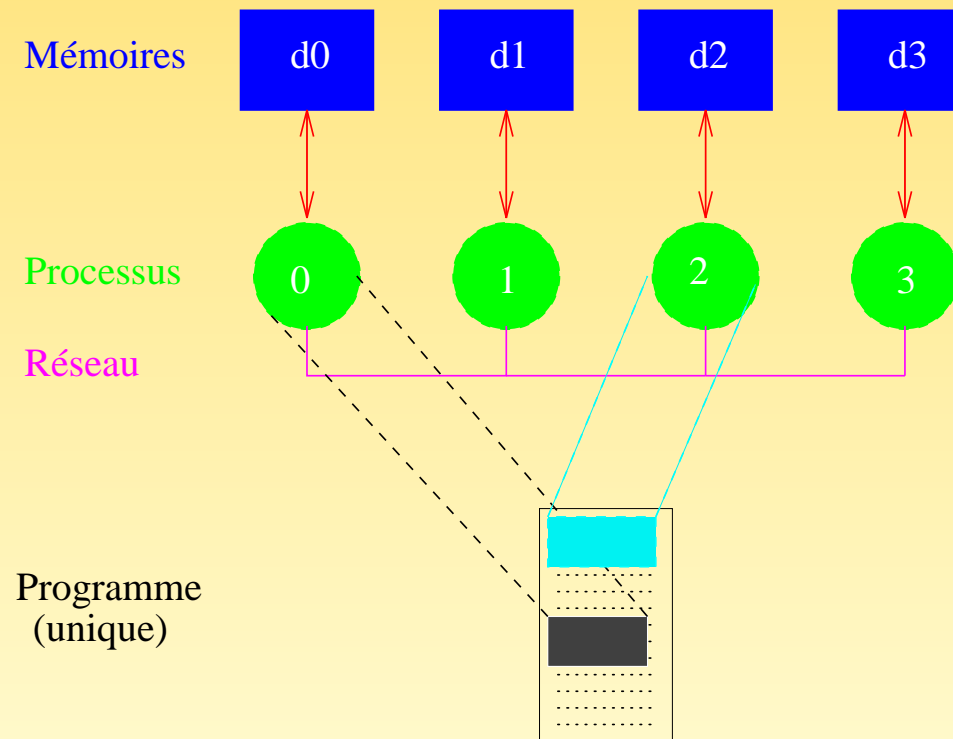


FIG. 3 – *Single Program, Multiple Data*

## 1.2 – Concepts de l'échange de messages

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

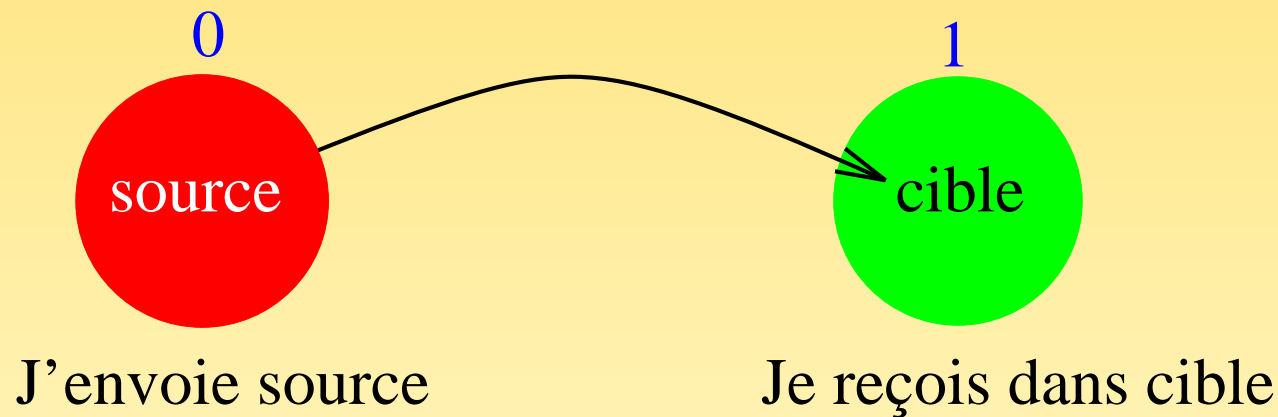


FIG. 4 – Échange d'un message

# 1 – Introduction : concepts de l'échange de messages 12

- ➡ 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.

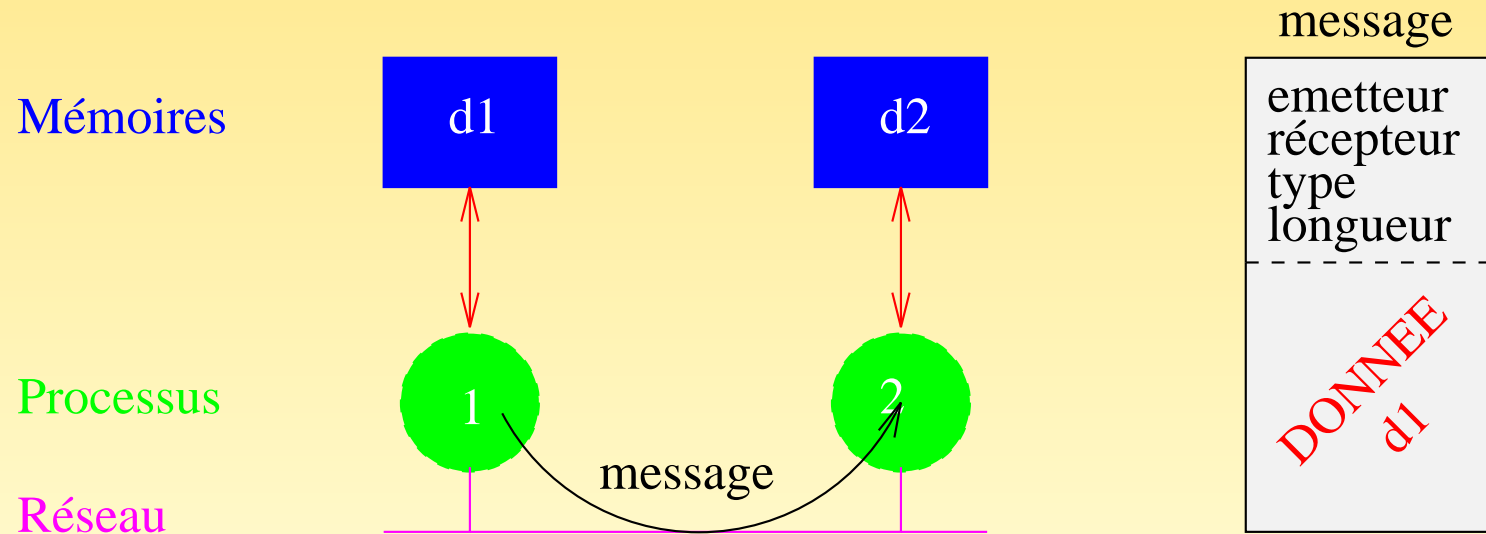


FIG. 5 – Constitution d'un message

- ➡ Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie, à la télécopie, 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**

## 1.3 – Historique

- ➡ **Version 1.0** : en juin 1994, le forum MPI (*Message Passing Interface Forum*), avec la participation d'une quarantaine d'organisations, abouti 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 97, cette version apportait des compléments essentiels 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

- ⇒ changements et ajouts importants par rapport à la version 2.2 ;
- ⇒ pour un meilleur support des applications actuelles et futures, notamment sur les machines massivement parallèles et *many cores* ;
- ⇒ principaux changements actuellement envisagés :
  - ▣► communications collectives non bloquantes ;
  - ▣► révision de l'implémentation des copies mémoire à mémoire ;
  - ▣► tolérance aux pannes ;
  - ▣► Fortran (2003-2008) *bindings* ;
  - ▣► interfaçage d'outils externes (pour le débogage et les mesures de performance) ;
  - ▣► etc.
- ⇒ attendu en 2012 ;
- ⇒ voir [http://meetings.mpi-forum.org/MPI\\_3.0\\_main\\_page.php](http://meetings.mpi-forum.org/MPI_3.0_main_page.php)  
<https://svn.mpi-forum.org/trac/mpi-forum-web/wiki>

## 1.4 – Bibliographie

- Message Passing Interface Forum, *MPI : A Message-Passing Interface Standard, Version 2.2*, High Performance Computing Center Stuttgart (HLRS), 2009  
<https://fs.hlr.de/projects/par/mpi/mpi22/>
- William Gropp, Ewing Lusk et Anthony Skjellum : *Using MPI : Portable Parallel Programming with the Message Passing Interface*, second edition, MIT Press, 1999
- William Gropp, Ewing Lusk et Rajeev Thakur : *Using MPI-2*, MIT Press, 1999
- Peter S. Pacheco : *Parallel Programming with MPI*, Morgan Kaufman Ed., 1997
- Documentations complémentaires :  
<http://www.mpi-forum.org/docs/>  
[http://www.mpi-forum.org/mpi2\\_1/index.htm](http://www.mpi-forum.org/mpi2_1/index.htm)  
<http://www.mcs.anl.gov/research/projects/mpi/learning.html>  
<http://www.mpi-forum.org/archives/notes/notes.html>



☞ 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

1. **MPICH2** : <http://www.mcs.anl.gov/research/projects/mpich2/>
2. **Open MPI** : <http://www.open-mpi.org/>

## ☛ Les Débogueurs

1. **Totalview** <http://www.totalviewtech.com/home/>
2. **DDT** <http://www.allinea.com/products/ddt/>

## ☛ Outils de mesure de performances

1. **MPE** : (MPI Parallel Environment)  
<http://www.mcs.anl.gov/research/projects/perfvis/download/index.htm>
2. **Scalasca** : Scalable Performance Analysis of Large-Scale Applications)  
<http://www.fz-juelich.de/jsc/scalasca/>
3. **Vampir** <http://www.vampir.eu/>

☞ Quelques bibliothèques scientifiques parallèles du *open source* :

1. **ScaLAPACK** : résolution de problèmes d'algèbre linéaire par des méthodes directes. Les sources sont téléchargeables sur le site <http://www.netlib.org/scalapack/>
2. **PETSc** : résolution de problèmes d'algèbre linéaire et non-linéaire par des méthodes itératives. Les sources sont téléchargeables sur le site <http://www.mcs.anl.gov/petsc/petsc-as/>
3. **MUMPS** : résolution de grands systèmes linéaires creux par méthode directe multifrontale parallèle. Les sources sont téléchargeables sur le site <http://graal.ens-lyon.fr/MUMPS/>
4. **FFTW** : transformées de Fourier rapides. Les sources sont téléchargeables sur le site <http://www.fftw.org>

## 2 – Environnement

### 2.1 – Description

- ➡ Tout unité de programme appelant des sous-programmes MPI doit inclure un fichier d'en-têtes. En Fortran, il faut maintenant utiliser le *module* `mpi` introduit dans MPI-2 (dans MPI-1, il s'agissait du fichier `mpif.h`), et en C/C++ le fichier `mpi.h`.
- ➡ Le sous-programme `MPI_INIT()` permet d'initialiser l'environnement nécessaire :

```
integer, intent(out) :: code  
call MPI_INIT(code)
```

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

```
integer, intent(out) :: code  
call MPI_FINALIZE(code)
```

- ☞ Toutes les opérations effectuées par **MPI** portent sur des **communicateurs**. Le communicateur par défaut est **MPI\_COMM\_WORLD** qui comprend tous les processus actifs.

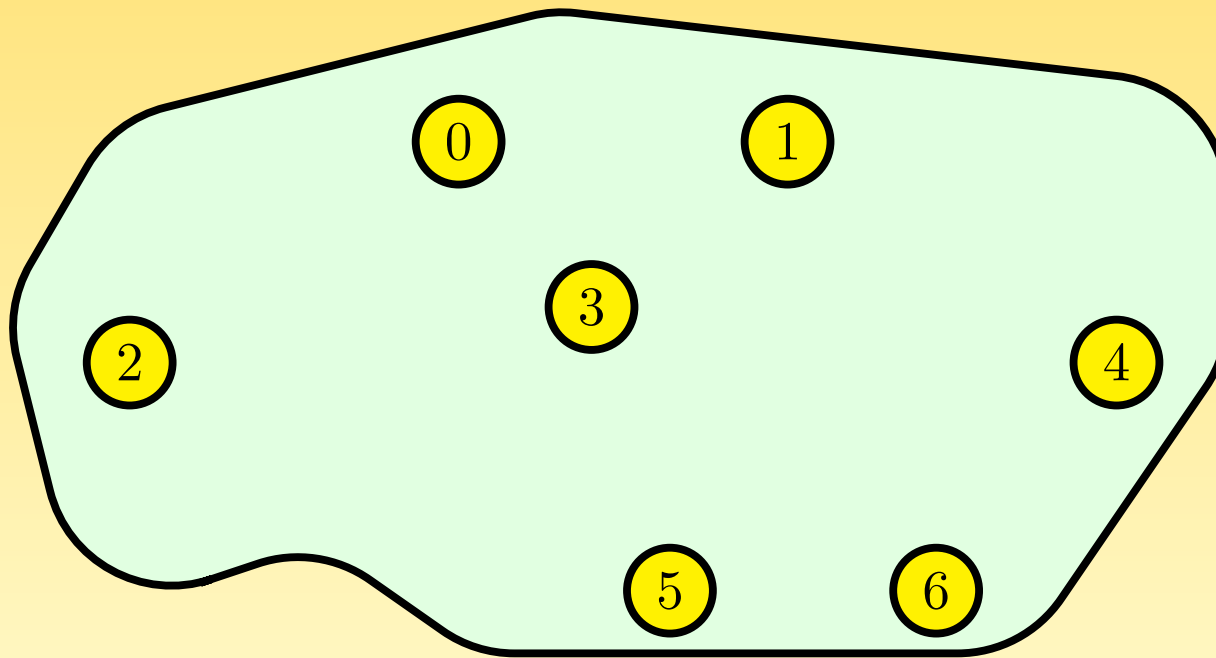


FIG. 6 – Communicateur MPI\_COMM\_WORLD

- ➡ À tout instant, on peut connaître le nombre de processus gérés par un communicateur donné par le sous-programme `MPI_COMM_SIZE()` :

```
integer, intent(out) :: nb_procs,code  
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
```

- ➡ De même, le sous-programme `MPI_COMM_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 `MPI_COMM_SIZE() - 1`) :

```
integer, intent(out) :: rang,code  
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
```

## 2.2 – Exemple

```
1 program qui_je_suis
2   use mpi
3   implicit none
4   integer :: nb_procs,rang,code
5
6   call MPI_INIT(code)
7
8   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
9   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
10
11  print *,'Je suis le processus ',rang,' parmi ',nb_procs
12
13  call MPI_FINALIZE(code)
14 end program qui_je_suis
```

```
> mpiexec -n 7 qui_je_suis
```

```
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
```

## 3 – Communications point à point

### 3.1 – 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**).

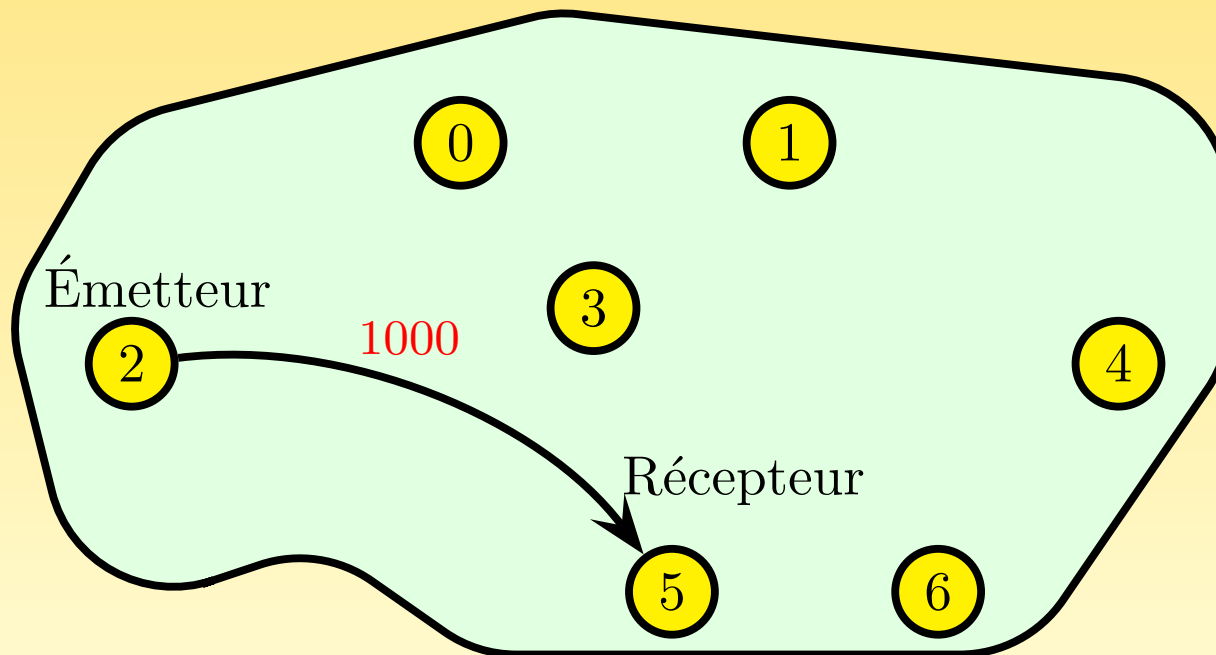


FIG. 7 – Communication point à point



- ☞ L'émetteur et le récepteur sont identifiés par leur **rang** dans le communicateur.
- ☞ Ce que l'on appelle l'**enveloppe d'un message** est constituée :
  - ① du rang du processus émetteur ;
  - ② du rang du processus récepteur ;
  - ③ de l'étiquette (*tag*) du message ;
  - ④ du nom du communicateur qui définira le contexte de communication de l'opération.
- ☞ 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 qui seront vus au chapitre 7.

# 3 – Communications point à point : notions générales 26

```
1 program point_a_point
2   use mpi
3   implicit none
4
5   integer, dimension(MPI_STATUS_SIZE) :: statut
6   integer, parameter      :: etiquette=100
7   integer                  :: rang,valeur,code
8
9   call MPI_INIT(code)
10
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  if (rang == 2) then
14    valeur=1000
15    call MPI_SEND(valeur,1,MPI_INTEGER,5,etiquette,MPI_COMM_WORLD,code)
16  elseif (rang == 5) then
17    call MPI_RECV(valeur,1,MPI_INTEGER,2,etiquette,MPI_COMM_WORLD,statut,code)
18    print *,'Moi, processus 5, j''ai reçu ',valeur,' du processus 2.'
19  end if
20
21  call MPI_FINALIZE(code)
22
23 end program point_a_point
```

```
> mpiexec -n 7 point_a_point
```

```
Moi, processus 5, j'ai reçu 1000 du processus 2
```

## 3.2 – Types de données de base

TAB. 1 – Principaux types de données de base (Fortran)

Type MPI	Type Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER

TAB. 2 – Principaux types de données de base (C)

Type MPI	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

## 3.3 – Autres possibilités

- ➡ À la réception d'un message, le rang du processus et l'étiquette peuvent être des « *jokers* », respectivement `MPI_ANY_SOURCE` et `MPI_ANY_TAG`.
- ➡ Une communication avec le processus « fictif » de rang `MPI_PROC_NULL` n'a aucun effet.
- ➡ `MPI_STATUS_IGNORE` est une constante prédéfinie qui peut être utilisée à la place de la variable prévue pour récupérer en réception le *statut*.
- ➡ `MPI_SUCCESS` est une constante prédéfinie qui permet de tester le code de retour d'une fonction MPI.
- ➡ Il existe des variantes syntaxiques, `MPI_SENDRECV()` et `MPI_SENDRECV_REPLACE()`, qui enchaînent un envoi et une réception (dans le premier cas, la zone de réception doit être forcément différente de la zone d'émission).
- ➡ On peut créer des structures de données plus complexes grâce à ses propres types dérivés (voir le chapitre 6).

# 3 – Communications point à point : autres possibilités<sup>30</sup>

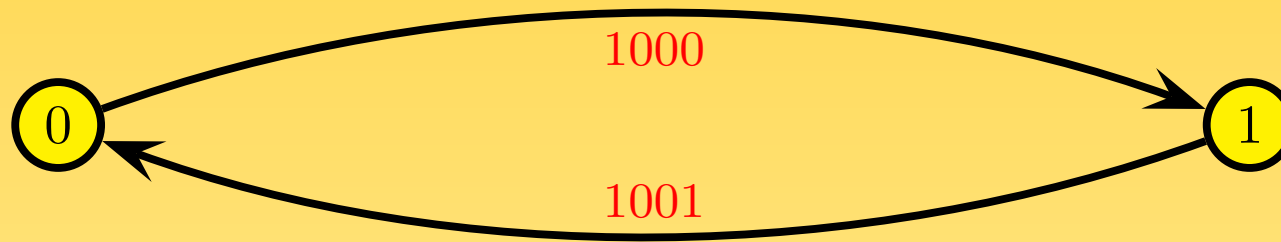


FIG. 8 – Communication sendrecv entre les processus 0 et 1

```
1 program sendrecv
2   use mpi
3   implicit none
4   integer                :: rang,valeur,num_proc,code
5   integer,parameter     :: etiquette=110
6   call MPI_INIT(code)
7   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
8
9   ! On suppose avoir exactement 2 processus
10  num_proc=mod(rang+1,2)
11
12  call MPI_SENDRECV(rang+1000,1,MPI_INTEGER,num_proc,etiquette,valeur,1,MPI_INTEGER, &
13                  num_proc,etiquette,MPI_COMM_WORLD,MPI_STATUS_IGNORE,code)
14  ! Test du code de retour de la fonction MPI_SENDRECV
15  if (code /= MPI_SUCCESS) call MPI_ABORT(MPI_COMM_WORLD,2,code)
16  print *,'Moi, processus ',rang,', j''ai reçu ',valeur,', du processus ',num_proc
17  call MPI_FINALIZE(code)
18 end program sendrecv
```

# 3 – Communications point à point : autres possibilités<sup>31</sup>

```
> mpiexec -n 2 sendrecv
```

```
Moi, processus 1, j'ai reçu 1000 du processus 0  
Moi, processus 0, j'ai reçu 1001 du processus 1
```

Attention ! Il convient de noter que si le sous-programme `MPI_SEND()` est implémenté de façon **synchrone** (voir le chapitre 7) dans la version de la bibliothèque `MPI` mise en œuvre, le code précédent serait en situation de verrouillage si à la place de l'ordre `MPI_SENDRECV()` on utilisait un ordre `MPI_SEND()` suivi d'un ordre `MPI_RECV()`. En effet, chacun des deux processus attendrait un ordre de réception qui ne viendrait jamais, puisque les deux envois resteraient en suspens. Pour des raisons de portabilité, il faut donc absolument éviter ces cas-là.

```
call MPI_SEND(rang+1000, 1, MPI_INTEGER, num_proc, etiquette, MPI_COMM_WORLD, code)  
call MPI_RECV(valeur, 1, MPI_INTEGER, num_proc, etiquette, MPI_COMM_WORLD, statut, code)
```

# 3 – Communications point à point : autres possibilités<sup>32</sup>

```
1 PROGRAM joker
2   USE MPI
3   IMPLICIT NONE
4   INTEGER, PARAMETER           :: m=4,etiquette1=11,etiquette2=22
5   INTEGER, DIMENSION(m,m)     :: A
6   INTEGER                      :: nb_procs,rang,code
7   INTEGER, DIMENSION(MPI_STATUS_SIZE) :: statut
8   INTEGER                      :: nb_elements,i
9   INTEGER, DIMENSION(:), ALLOCATABLE :: C
10
11  CALL MPI_INIT(code)
12  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
13  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15  A(:, :) = 0
```



# 3 – Communications point à point : autres possibilités<sup>33</sup>

```
16 IF (rang == 0) THEN
17   !Initialisation de la matrice a sur le processeur 0
18   A(:, :) = RESHAPE((/ (i,i=1,m*m) /), (/ m,m /))
19   !Envoi de 2 elements de la matrice A au processeur 1
20   CALL MPI_SEND(A(1,1),2,MPI_INTEGER,1,etiquette1,MPI_COMM_WORLD,code)
21   !Envoi de 3 elements de la matrice A au processeur 2
22   CALL MPI_SEND(A(1,2),3,MPI_INTEGER,2,etiquette2,MPI_COMM_WORLD,code)
23
24 ELSE
25   !On teste avant la reception si le message est arrive et de qui
26   CALL MPI_PROBE(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,statut,code)
27   !On regarde combien il y a d'elements a recevoir
28   CALL MPI_GET_ELEMENTS(statut,MPI_INTEGER,nb_elements,code)
29   !On alloue le tableau de reception C sur chaque processeur
30   IF (nb_elements /= 0 ) ALLOCATE (C(1:nb_elements))
31   !On recoit le message
32   CALL MPI_RECV(C,nb_elements,MPI_INTEGER,MPI_ANY_SOURCE,MPI_ANY_TAG,&
33                MPI_COMM_WORLD,statut,code)
34   PRINT *, 'Moi processus ', rang, 'je recois ',nb_elements, &
35           'elements du processus ',&
36           statut(MPI_SOURCE), 'Mon tableau C vaut ', C(:)
37 END IF
38 CALL MPI_FINALIZE(code)
39 END PROGRAM joker
```

# 3 – Communications point à point : autres possibilités<sup>34</sup>

```
> mpiexec -n 3 sendrecv1
```

```
Moi processus 1 je reçois 2 elements du processus 0 Mon tableau C vaut 1 2  
Moi processus 2 je reçois 3 elements du processus 0 Mon tableau C vaut 5 6 7
```

## 3.4 – Exemple : anneau de communication

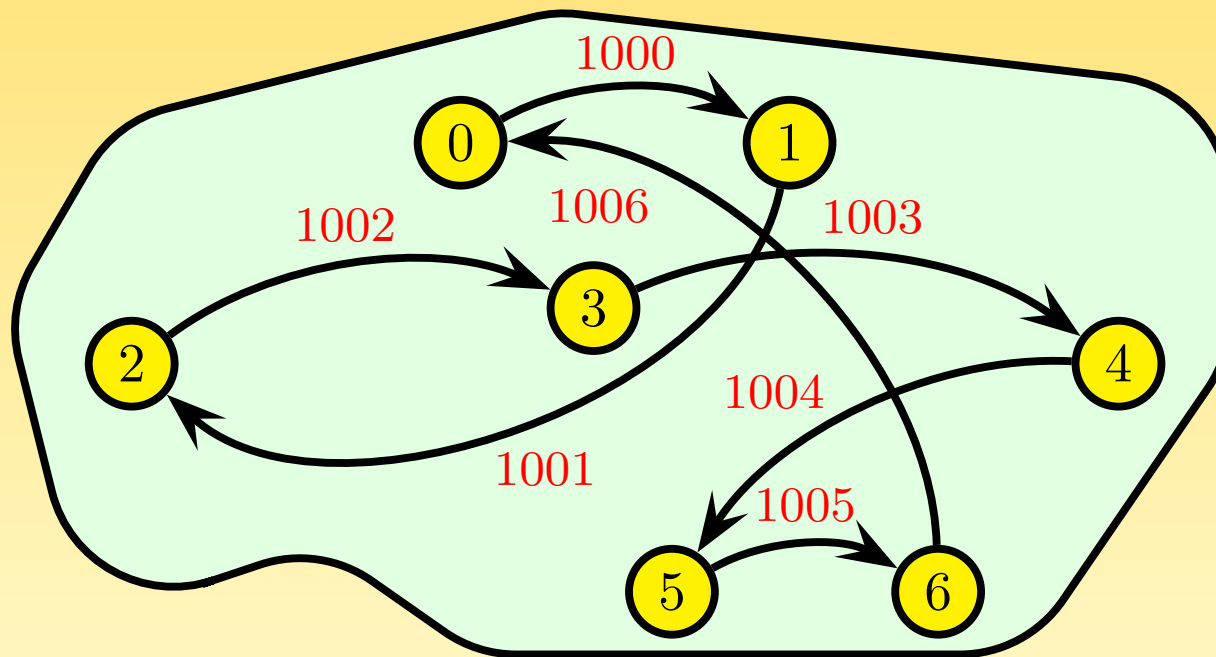


FIG. 9 – Anneau de communication

Si tous les processus font un envoi puis une réception, toutes les communications pourront potentiellement démarrer simultanément et n'auront donc pas lieu en anneau (outre le problème déjà mentionné de portabilité, au cas où l'implémentation du `MPI_SEND()` est faite de façon synchrone dans la version de la bibliothèque `MPI` mise en œuvre) :

```
...  
valeur=rang+1000  
call MPI_SEND(valeur,1,MPI_INTEGER,num_proc_suivant,etiquette,MPI_COMM_WORLD,code)  
call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette,MPI_COMM_WORLD,&  
             statut,code)  
...
```

Pour que les communications se fassent réellement en **anneau**, à l'image d'un passage de **jeton** entre processus, il faut procéder différemment et faire en sorte qu'un processus initie la chaîne :

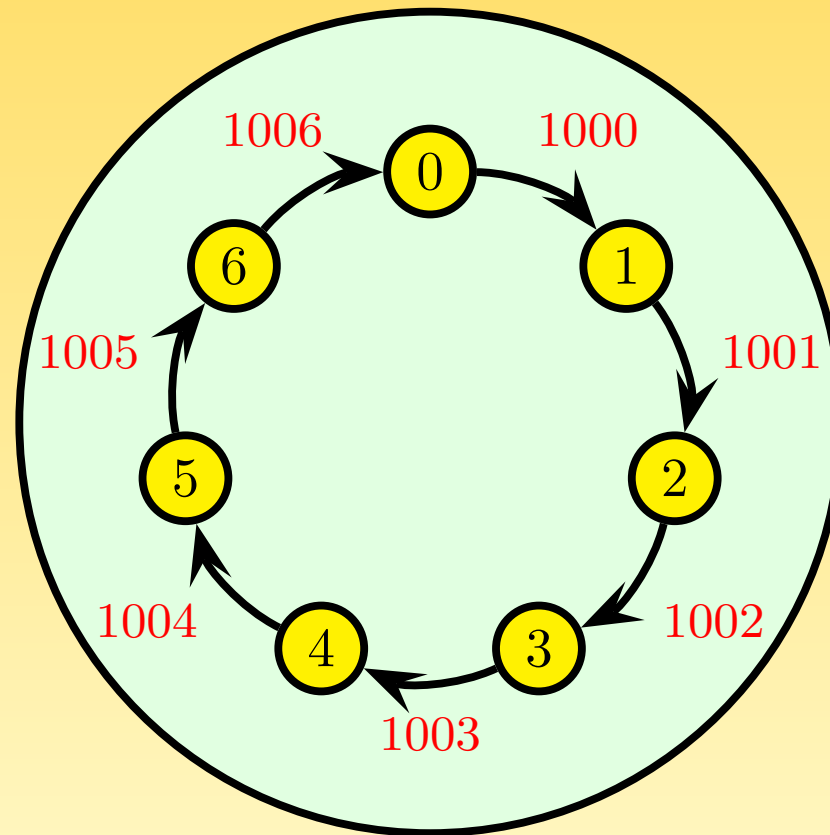


FIG. 10 – Anneau de communication

# 3 – Communications point à point : exemple anneau 38

```
1 program anneau
2   use mpi
3   implicit none
4   integer, dimension(MPI_STATUS_SIZE) :: statut
5   integer, parameter      :: etiquette=100
6   integer                 :: nb_procs,rang,valeur, &
7                           num_proc_precedent,num_proc_suivant,code
8   call MPI_INIT(code)
9   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
10  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
11
12  num_proc_suivant=mod(rang+1,nb_procs)
13  num_proc_precedent=mod(nb_procs+rang-1,nb_procs)
14
15  if (rang == 0) then
16    call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
17                MPI_COMM_WORLD,code)
18    call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
19                MPI_COMM_WORLD,statut,code)
20  else
21    call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
22                MPI_COMM_WORLD,statut,code)
23    call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
24                MPI_COMM_WORLD,code)
25  end if
26
27  print *,'Moi, proc. ',rang,', j''ai reçu ',valeur,' du proc. ',num_proc_precedent
28
29  call MPI_FINALIZE(code)
30 end program anneau
```

```
> mpiexec -n 7 anneau
```

```
Moi, proc. 1, j'ai reçu 1000 du proc. 0  
Moi, proc. 2, j'ai reçu 1001 du proc. 1  
Moi, proc. 3, j'ai reçu 1002 du proc. 2  
Moi, proc. 4, j'ai reçu 1003 du proc. 3  
Moi, proc. 5, j'ai reçu 1004 du proc. 4  
Moi, proc. 6, j'ai reçu 1005 du proc. 5  
Moi, proc. 0, j'ai reçu 1006 du proc. 6
```

## 4 – Communications collectives

### 4.1 – 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).
- ➡ Il est inutile d'ajouter une synchronisation globale (barrière) après une opération collective.
- ➡ 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.



➔ Il y a trois types de sous-programmes :

- ❶ celui qui assure les synchronisations globales : `MPI_BARRIER()`.
- ❷ 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()` ;
  - ❑ diffusion sélective, par tous les processus, de données réparties : `MPI_ALLTOALL()`.
- ❸ ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
  - ❑ opérations de réduction, qu'elles soient d'un type prédéfini (somme, produit, maximum, minimum, etc.) ou d'un type personnel : `MPI_REDUCE()` ;
  - ❑ opérations de réduction avec diffusion du résultat (il s'agit en fait d'un `MPI_REDUCE()` suivi d'un `MPI_BCAST()`) : `MPI_ALLREDUCE()`.

## 4.2 – Synchronisation globale : MPI\_BARRIER()

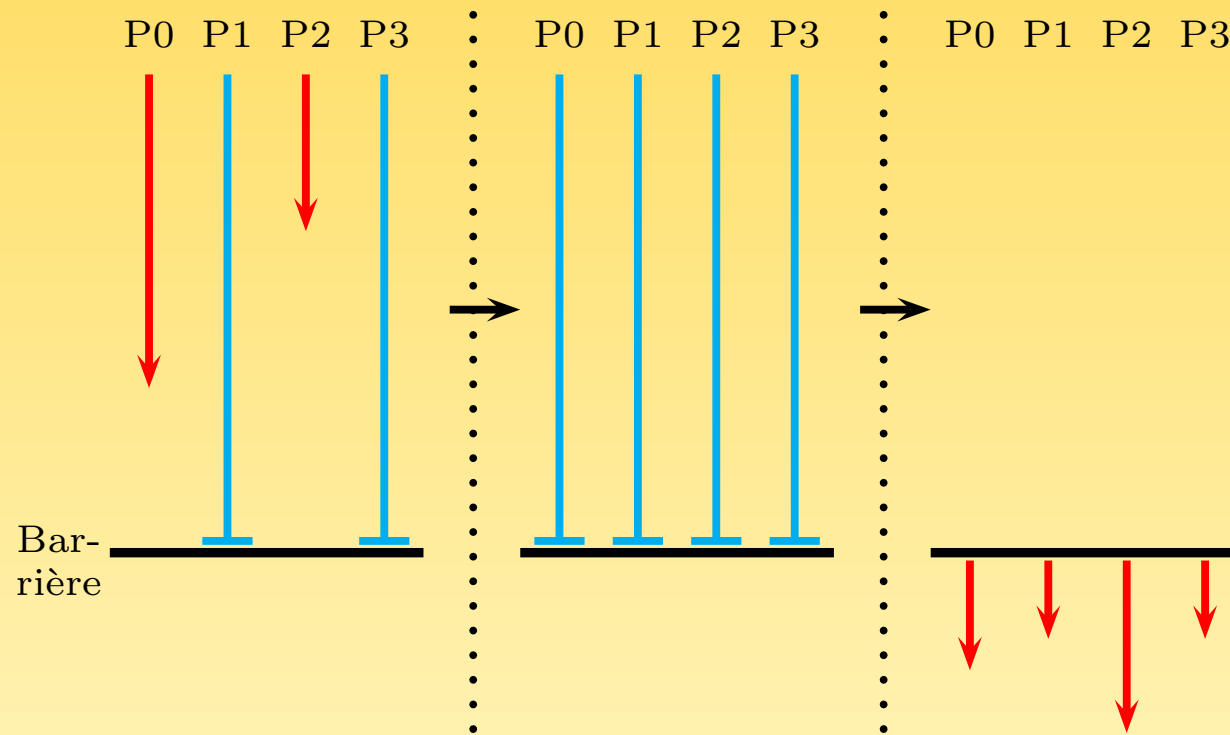


FIG. 11 – Synchronisation globale : MPI\_BARRIER()

```
integer, intent(out) :: code
```

```
call MPI_BARRIER(MPI_COMM_WORLD, code)
```

## 4.3 – Diffusion générale : MPI\_BCAST()

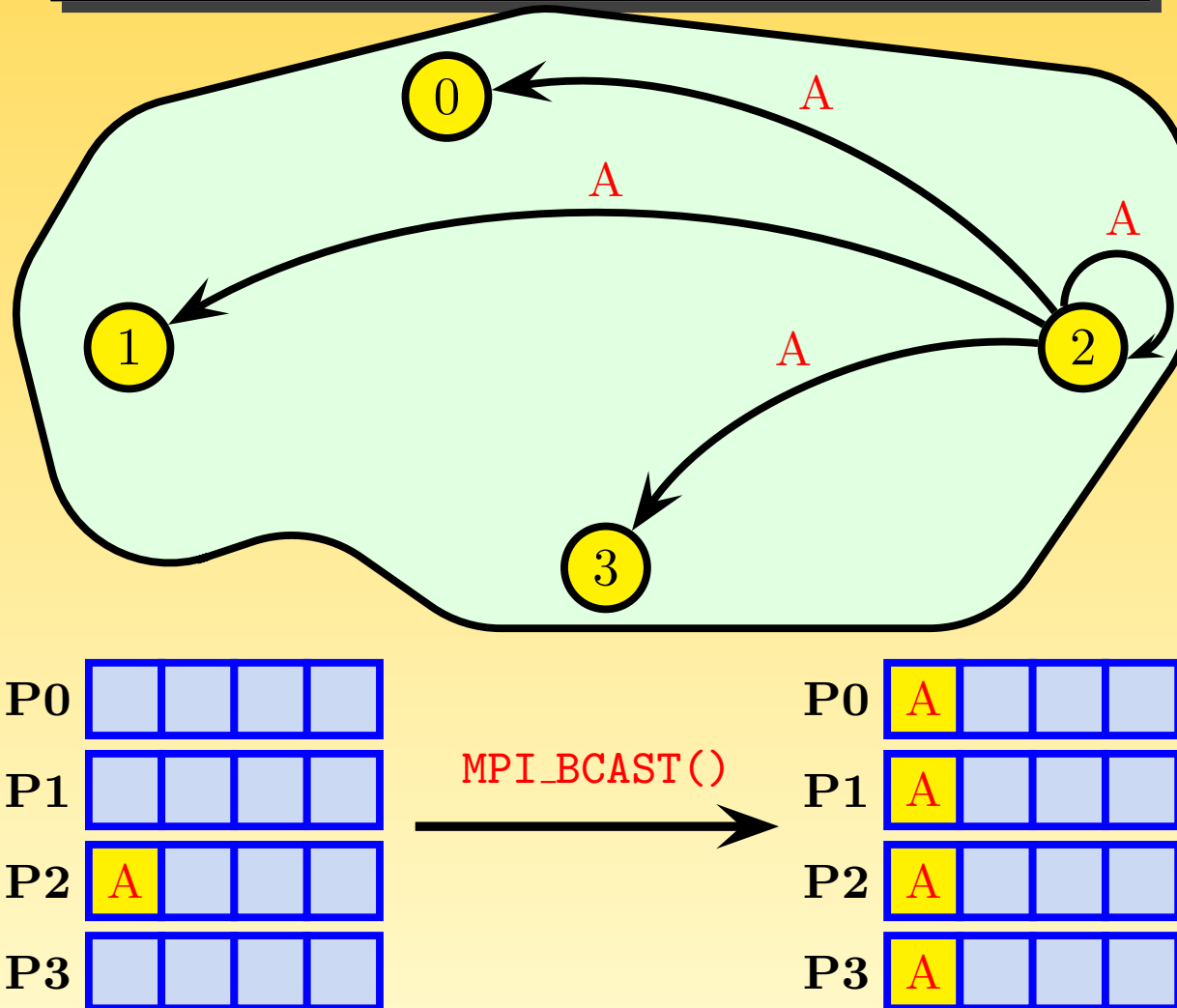


FIG. 12 – Diffusion générale : MPI\_BCAST()

```
1 program bcast
2   use mpi
3   implicit none
4
5   integer :: rang,valeur,code
6
7   call MPI_INIT(code)
8   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
9
10  if (rang == 2) valeur=rang+1000
11
12  call MPI_BCAST(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)
13
14  print *,'Moi, processus ',rang,', j''ai reçu ',valeur,' du processus 2'
15
16  call MPI_FINALIZE(code)
17
18 end program bcast
```

```
> mpiexec -n 4 bcast
```

```
Moi, processus 2, j'ai reçu 1002 du processus 2
Moi, processus 0, j'ai reçu 1002 du processus 2
Moi, processus 1, j'ai reçu 1002 du processus 2
Moi, processus 3, j'ai reçu 1002 du processus 2
```

4.4 – Diffusion sélective : MPI\_SCATTER()

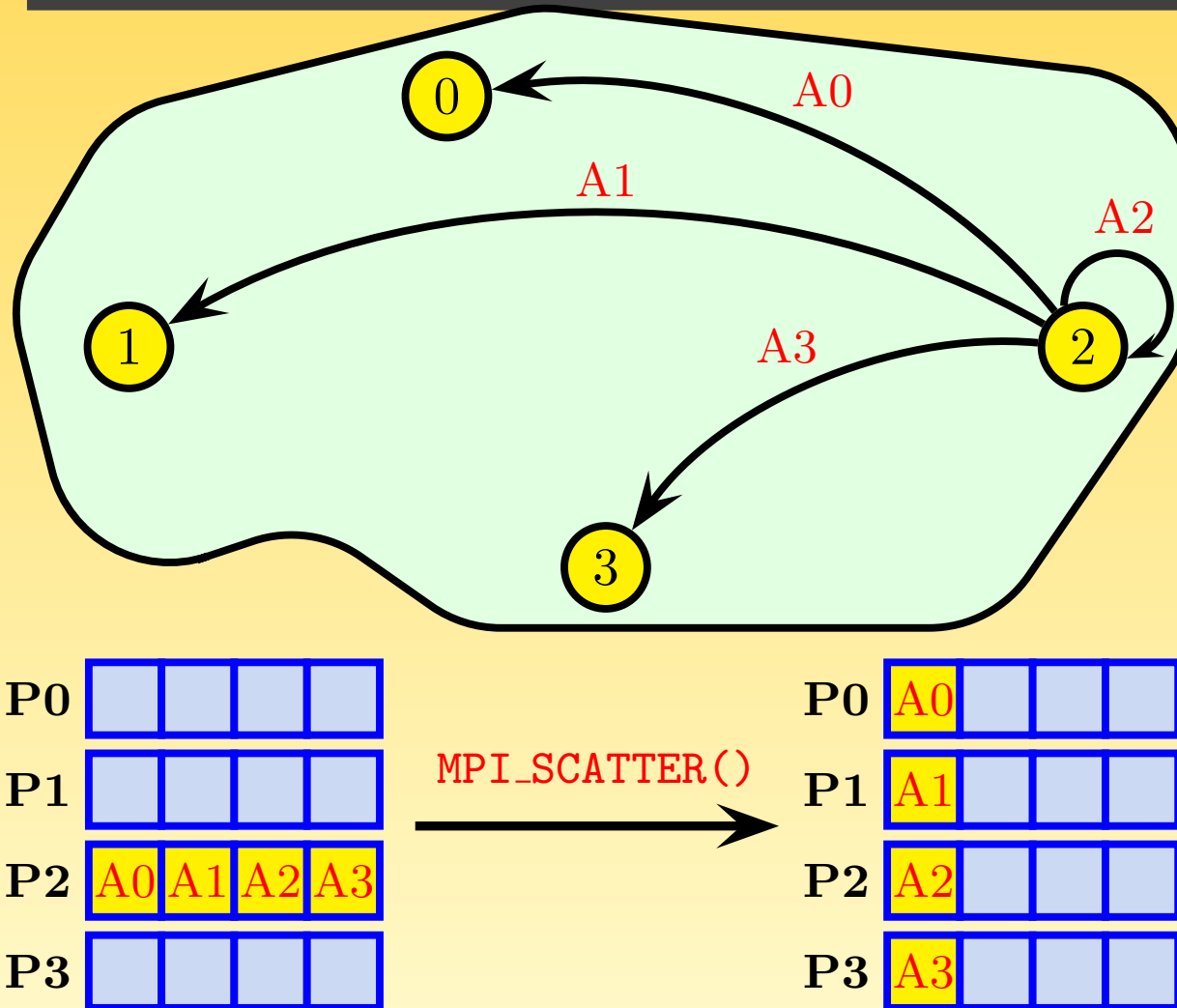


FIG. 13 – Diffusion sélective : MPI\_SCATTER()

```
1 program scatter
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=8
6   integer                    :: nb_procs,rang,longueur_tranche,i,code
7   real, allocatable, dimension(:) :: valeurs,donnees
8
9   call MPI_INIT(code)
10  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12  longueur_tranche=nb_valeurs/nb_procs
13  allocate(donnees(longueur_tranche))
14
15  if (rang == 2) then
16    allocate(valeurs(nb_valeurs))
17    valeurs(:)=(/(1000.+i,i=1,nb_valeurs)/)
18    print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
19           valeurs(1:nb_valeurs)
20  end if
21
22  call MPI_SCATTER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
23                MPI_REAL,2,MPI_COMM_WORLD,code)
24  print *,'Moi, processus ',rang,', j''ai reçu ', donnees(1:longueur_tranche), &
25         ' du processus 2'
26  call MPI_FINALIZE(code)
27
28 end program scatter
```

```
> mpiexec -n 4 scatter
```

```
Moi, processus 2 envoie mon tableau valeurs :
```

```
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

```
Moi, processus 0, j'ai reçu 1001. 1002. du processus 2
```

```
Moi, processus 1, j'ai reçu 1003. 1004. du processus 2
```

```
Moi, processus 3, j'ai reçu 1007. 1008. du processus 2
```

```
Moi, processus 2, j'ai reçu 1005. 1006. du processus 2
```

## 4.5 – Collecte : MPI\_GATHER()

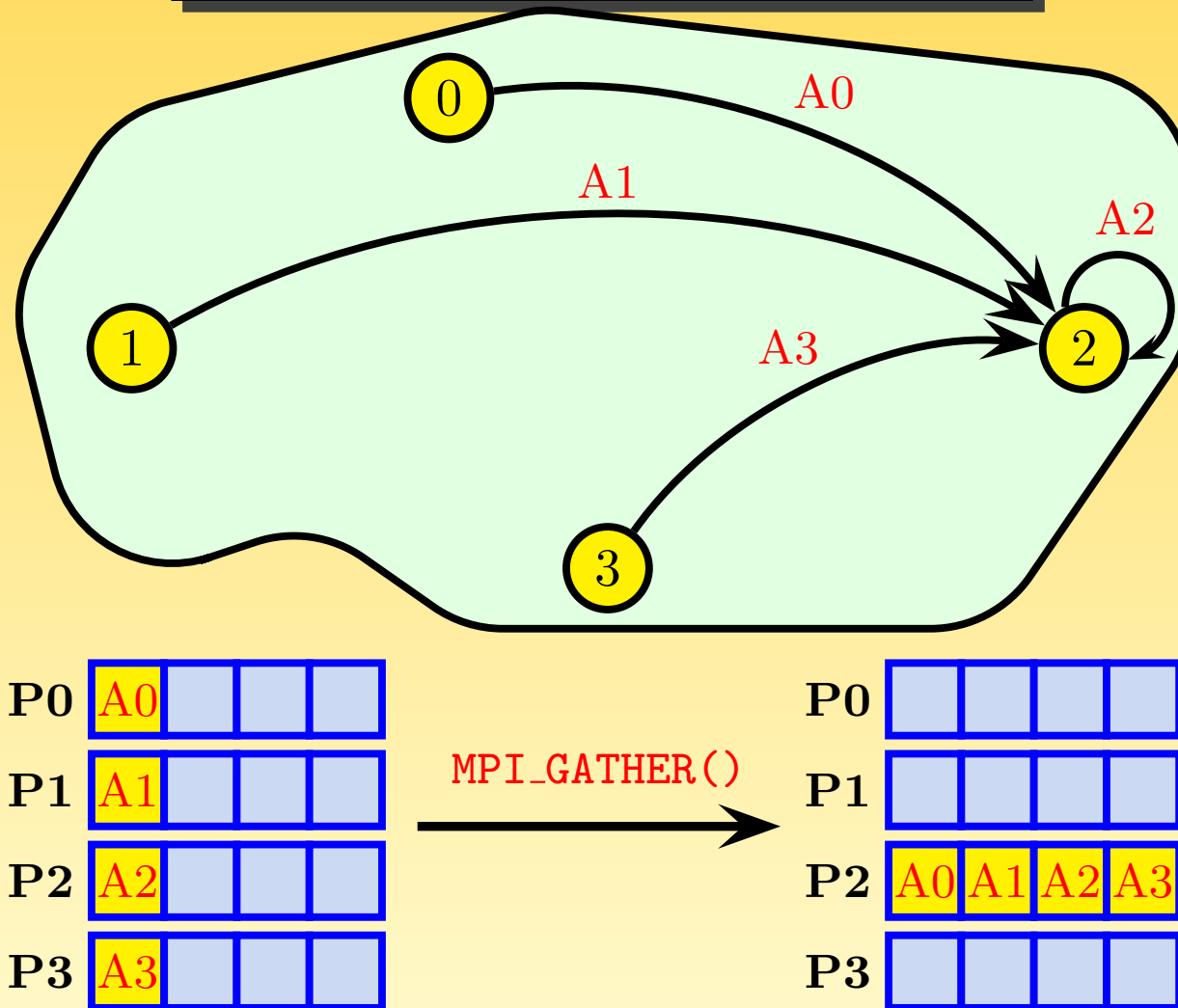


FIG. 14 – Collecte : MPI\_GATHER()



```
1 program gather
2   use mpi
3   implicit none
4   integer, parameter          :: nb_valeurs=8
5   integer                     :: nb_procs,rang,longueur_tranche,i,code
6   real, dimension(nb_valeurs) :: donnees
7   real, allocatable, dimension(:) :: valeurs
8
9   call MPI_INIT(code)
10  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  longueur_tranche=nb_valeurs/nb_procs
14
15  allocate(valeurs(longueur_tranche))
16
17  valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
18  print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
19         valeurs(1:longueur_tranche)
20
21  call MPI_GATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
22                MPI_REAL,2,MPI_COMM_WORLD,code)
23
24  if (rang == 2) print *,'Moi, processus 2', 'j''ai reçu ',donnees(1:nb_valeurs)
25
26  call MPI_FINALIZE(code)
27
28 end program gather
29
```

```
> mpiexec -n 4 gather
```

```
Moi, processus 1 envoie mon tableau valeurs : 1003. 1004.
```

```
Moi, processus 0 envoie mon tableau valeurs : 1001. 1002.
```

```
Moi, processus 2 envoie mon tableau valeurs : 1005. 1006.
```

```
Moi, processus 3 envoie mon tableau valeurs : 1007. 1008.
```

```
Moi, processus 2, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

## 4.6 – Collecte générale : MPI\_ALLGATHER()

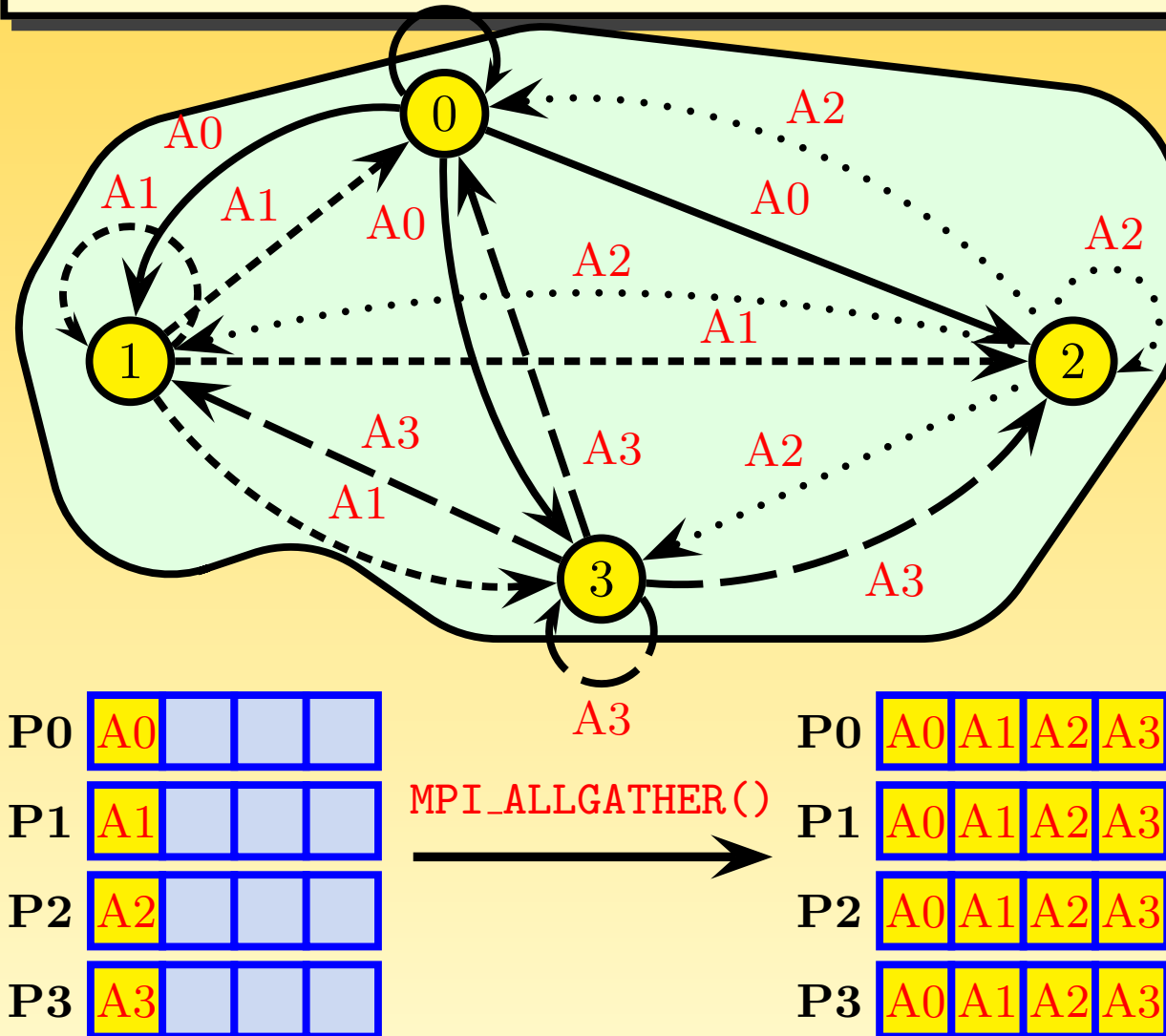


FIG. 15 – Collecte générale : MPI\_ALLGATHER()

```
1 program allgather
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=8
6   integer                    :: nb_procs,rang,longueur_tranche,i,code
7   real, dimension(nb_valeurs) :: donnees
8   real, allocatable, dimension(:) :: valeurs
9
10  call MPI_INIT(code)
11
12  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
13  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15  longueur_tranche=nb_valeurs/nb_procs
16  allocate(valeurs(longueur_tranche))
17
18  valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
19
20  call MPI_ALLGATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
21                   MPI_REAL,MPI_COMM_WORLD,code)
22
23  print *,'Moi, processus ',rang,', j''ai reçu ',donnees(1:nb_valeurs)'
24
25  call MPI_FINALIZE(code)
26
27 end program allgather
```

```
> mpiexec -n 4 allgather
```

```
Moi, processus 1, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.  
Moi, processus 3, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.  
Moi, processus 2, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.  
Moi, processus 0, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

## 4.7 – Collecte : MPI\_GATHERV()

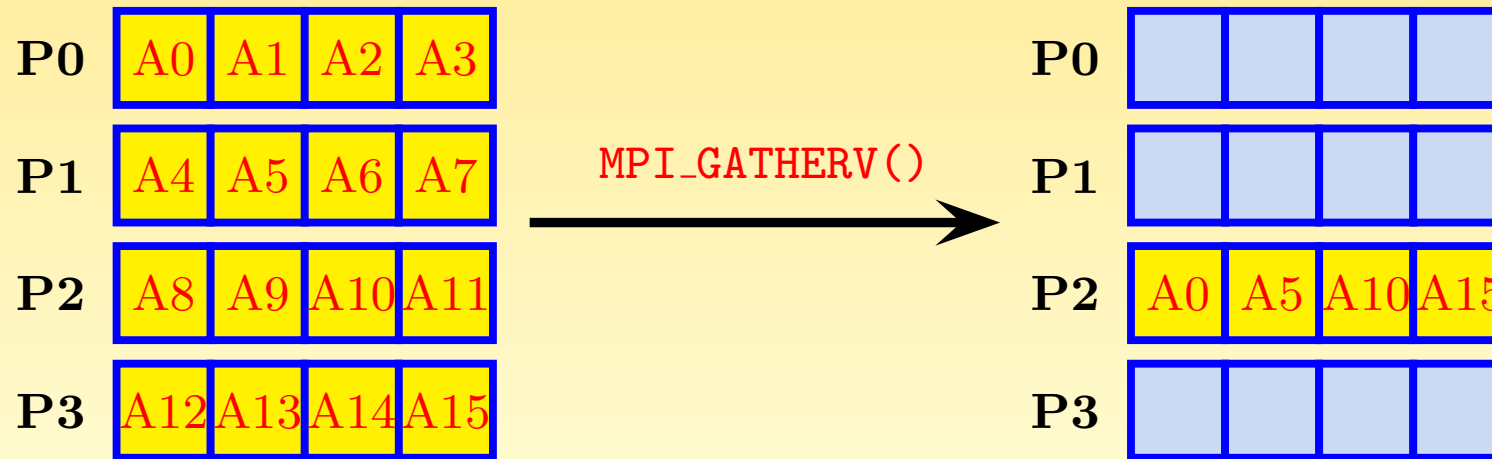
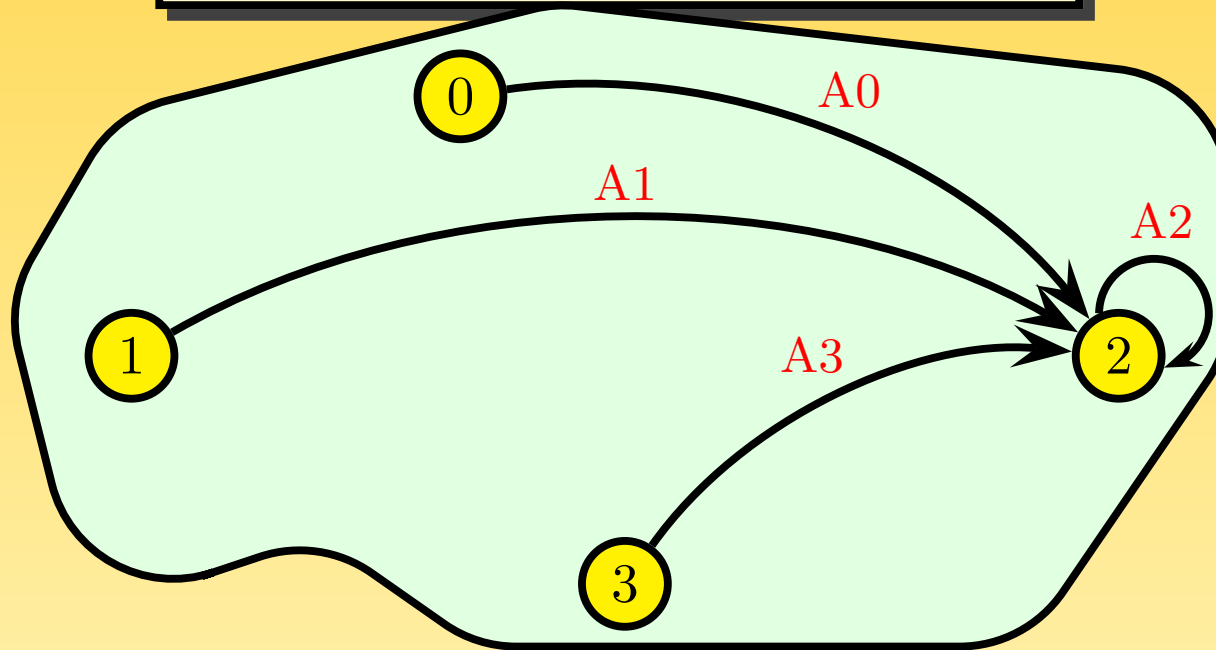


FIG. 16 – Collecte : MPI\_GATHERV()

```
1 program gatherv
2   use mpi
3   implicit none
4   INTEGER, PARAMETER          :: nb_valeurs=10
5   INTEGER                     :: nb_procs, rang, longueur_tranche, i, code
6   REAL, DIMENSION(nb_valeurs) :: donnees
7   REAL, ALLOCATABLE, DIMENSION(:) :: valeurs
8   INTEGER, ALLOCATABLE, DIMENSION(:) :: nb_elements_recus,pas
9
10  CALL MPI_INIT(code)
11  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
12  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14  longueur_tranche=(nb_valeurs*(rang+1))/nb_procs-(nb_valeurs*rang)/nb_procs
15  ALLOCATE(valeurs(longueur_tranche))
16  valeurs(:) = (/ (1000.+(rang*nb_valeurs/nb_procs)+i, i=1,longueur_tranche)/)
17
18  PRINT *, 'Moi, processus ', rang,'envoie mon tableau valeurs : ',&
19         valeurs(1:longueur_tranche)
20
21  IF (rang == 2) THEN
22     ALLOCATE(nb_elements_recus(nb_procs),pas(nb_procs))
23     nb_elements_recus(1) = nb_valeurs/nb_procs
24     pas(1) = 0
25     DO i=2,nb_procs
26         pas(i) = pas(i-1)+nb_elements_recus(i-1)
27         nb_elements_recus(i) = (nb_valeurs*(i))/nb_procs-(nb_valeurs*(i-1))/nb_procs
28     END DO
29  END IF
```

```
30 CALL MPI_GATHERV(valeurs, longueur_tranche, MPI_REAL, donnees, nb_elements_recus, &  
31 pas, MPI_REAL, 2, MPI_COMM_WORLD, code)  
32  
33  
34 IF (rang == 2) PRINT *, 'Moi, processus 2 recoit ', donnees(1:nb_valeurs)  
35  
36 CALL MPI_FINALIZE(code)  
37 end program gatherv
```

```
> mpiexec -n 4 gatherv
```

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

```
Moi, processus 2 recoit 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.  
1009. 1010.
```



## 4.8 – Échanges croisés : MPI\_ALLTOALL()

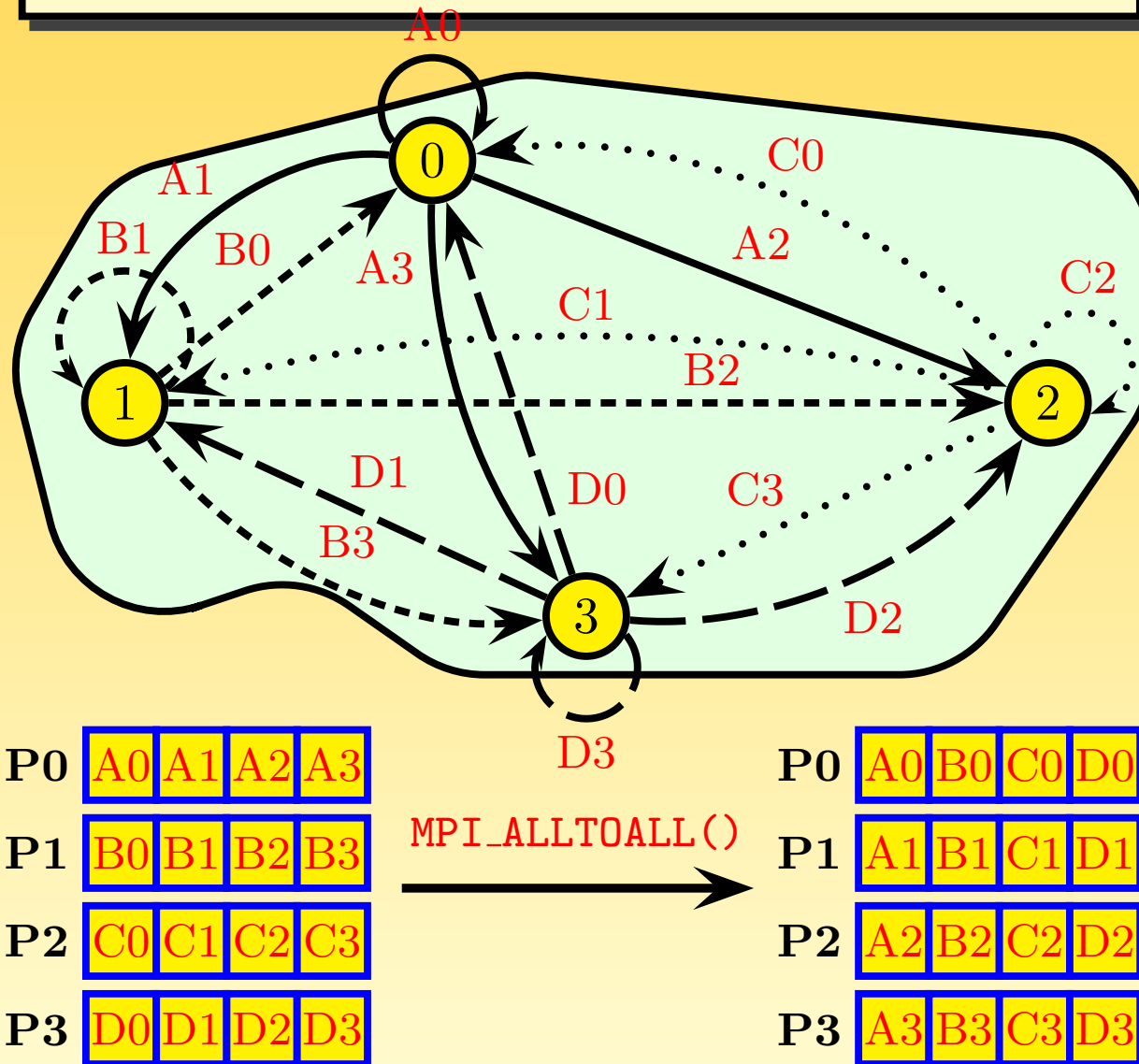


FIG. 17 – Échanges croisés : MPI\_ALLTOALL()

```
1 program alltoall
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=8
6   integer                    :: nb_procs,rang,longueur_tranche,i,code
7   real, dimension(nb_valeurs) :: valeurs,donnees
8
9   call MPI_INIT(code)
10  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  valeurs(:)=(/(1000.+rang*nb_valeurs+i,i=1,nb_valeurs)/)
14  longueur_tranche=nb_valeurs/nb_procs
15
16  print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
17         valeurs(1:nb_valeurs)
18
19  call MPI_ALLTOALL(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
20                 MPI_REAL,MPI_COMM_WORLD,code)
21
22  print *,'Moi, processus ',rang,', j''ai reçu ',donnees(1:nb_valeurs)
23
24  call MPI_FINALIZE(code)
25 end program alltoall
```

```
> mpiexec -n 4 alltoall
```

```
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, j'ai reçu 1001. 1002. 1009. 1010. 1017. 1018. 1025. 1026.
```

```
Moi, processus 2, j'ai reçu 1005. 1006. 1013. 1014. 1021. 1022. 1029. 1030.
```

```
Moi, processus 1, j'ai reçu 1003. 1004. 1011. 1012. 1019. 1020. 1027. 1028.
```

```
Moi, processus 3, j'ai reçu 1007. 1008. 1015. 1016. 1023. 1024. 1031. 1032.
```

## 4.9 – 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  $\text{SUM}(A(:))$  ou la recherche de l'élément de valeur maximum dans un vecteur  $\text{MAX}(V(:))$ .
- ➡ **MPI** propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus, avec récupération du résultat sur un seul processus (**`MPI_REDUCE()`**) ou bien sur tous (**`MPI_ALLREDUCE()`**, qui est en fait seulement 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.
- ➡ Le sous-programme **`MPI_SCAN()`** permet en plus d'effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du groupe.
- ➡ Les sous-programmes **`MPI_OP_CREATE()`** et **`MPI_OP_FREE()`** permettent de définir des opérations de réduction personnelles.

TAB. 3 – Principales opérations de réduction prédéfinies (il existe aussi d'autres opérations logiques)

Nom	Opération
MPI_SUM	Somme des éléments
MPI_PROD	Produit des éléments
MPI_MAX	Recherche du maximum
MPI_MIN	Recherche du minimum
MPI_MAXLOC	Recherche de l'indice du maximum
MPI_MINLOC	Recherche de l'indice du minimum
MPI_LAND	ET logique
MPI_LOR	OU logique
MPI_LXOR	OU exclusif logique

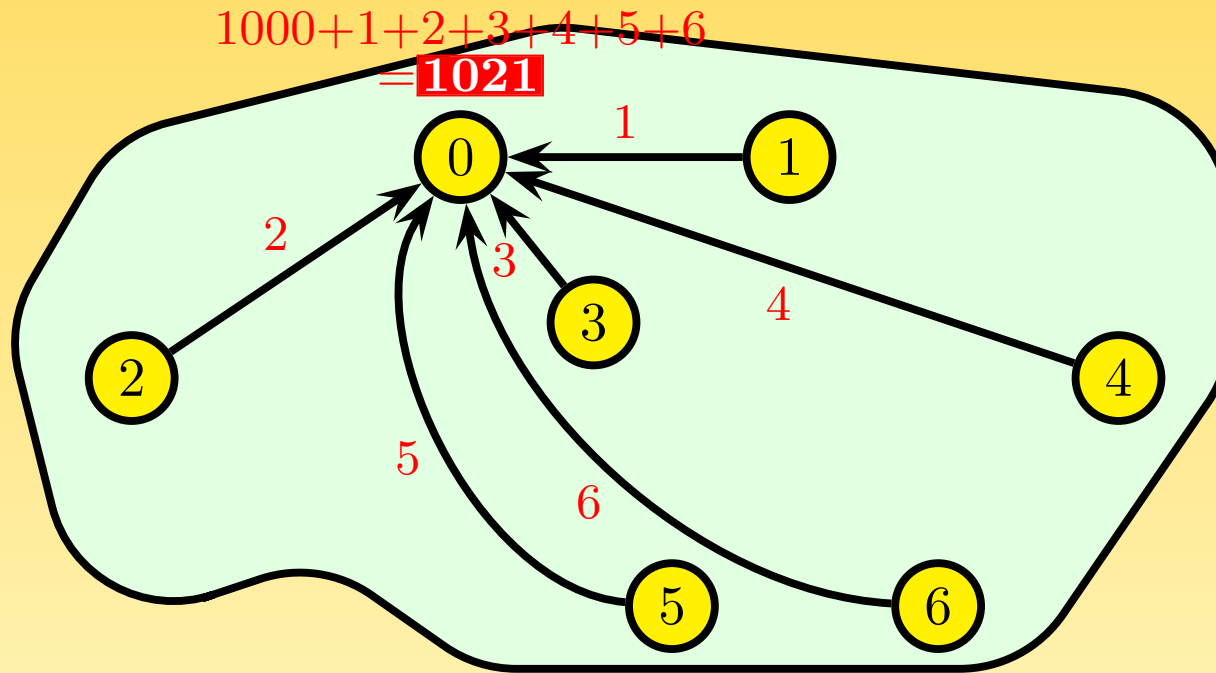


FIG. 18 – Réduction répartie (somme)

```
1 program reduce
2   use mpi
3   implicit none
4   integer :: nb_procs,rang,valeur,somme,code
5
6   call MPI_INIT(code)
7   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
8   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
9
10  if (rang == 0) then
11    valeur=1000
12  else
13    valeur=rang
14  endif
15
16  call MPI_REDUCE(valeur,somme,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,code)
17
18  if (rang == 0) then
19    print *,'Moi, processus 0, j''ai pour valeur de la somme globale ',somme
20  end if
21
22  call MPI_FINALIZE(code)
23 end program reduce
```

```
> mpiexec -n 7 reduce
```

```
Moi, processus 0, j'ai pour valeur de la somme globale 1021
```

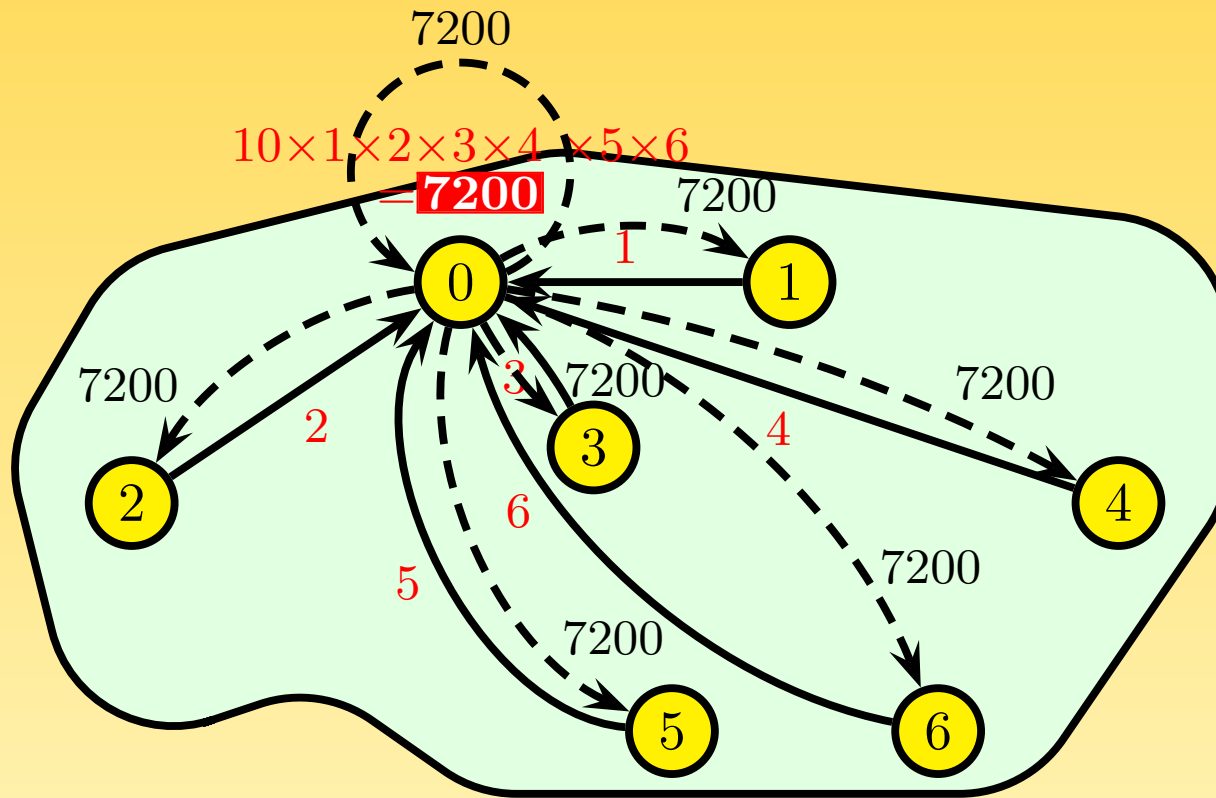


FIG. 19 – Réduction répartie (produit) avec diffusion du résultat



```
1 program allreduce
2
3 use mpi
4 implicit none
5
6 integer :: nb_procs,rang,valeur,produit,code
7
8 call MPI_INIT(code)
9 call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
10 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
11
12 if (rang == 0) then
13     valeur=10
14 else
15     valeur=rang
16 endif
17
18 call MPI_ALLREDUCE(valeur,produit,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD,code)
19
20 print *,'Moi, processus ',rang,', j''ai reçu la valeur du produit global ',produit
21
22 call MPI_FINALIZE(code)
23
24 end program allreduce
```

```
> mpiexec -n 7 allreduce
```

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

## 4.10 – Compléments

- ☞ Les sous-programmes `MPI_SCATTERV()`, `MPI_GATHERV()`, `MPI_ALLGATHERV()` et `MPI_ALLTOALLV()` étendent `MPI_SCATTER()`, `MPI_GATHER()`, `MPI_ALLGATHER()` et `MPI_ALLTOALL()` au cas où le nombre d'éléments à diffuser ou collecter est différent suivant les processus.
- ☞ Deux nouveaux sous-programmes ont été ajoutés pour étendre les possibilités des sous-programmes collectifs dans quelques cas particuliers :
  - ⇒ `MPI_ALLTOALLW()` : version de `MPI_ALLTOALLV()` où les déplacements sont exprimés en octets et non en éléments,
  - ⇒ `MPI_EXSCAN()` : version *exclusive* (processus de 0 à  $rang - 1$ ) de `MPI_SCAN()`, qui elle est *inclusive* (processus de 0 à  $rang$ ).

## 5 – Copies de mémoire à mémoire

### 5.1 – Introduction

Diverses approches existent pour transférer des données entre deux processus distincts.

Parmi les plus utilisées, on trouve :

- ❶ les communications point à point par échange de messages ([MPI](#), etc.) ;
- ❷ les communications par copies de mémoire à mémoire (accès direct à la mémoire d'un processus distant). Appelées RMA pour *Remote Memory Access* ou OSC pour *One Sided Communication*, c'est l'un des apports majeurs de [MPI](#).

## 5.1.1 – Rappel : concept de l'échange de messages

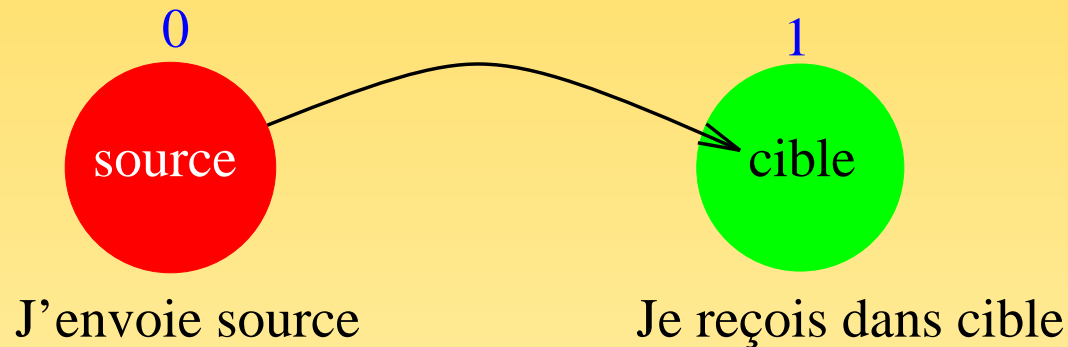


FIG. 20 – L'échange de messages

Dans le concept de l'échange de messages, un émetteur (source) va envoyer un message à un processus destinataire (cible) qui va faire la démarche de recevoir ce message. Cela nécessite que l'émetteur comme le destinataire prennent part à la communication. Ceci peut être contraignant et difficile à mettre en œuvre dans certains algorithmes (par exemple lorsqu'il faut gérer un compteur global).

## 5.1.2 – Concept des copies de mémoire à mémoire

Le concept de communication par copies de mémoire à mémoire n'est pas nouveau, [MPI](#) ayant simplement unifié les solutions constructeurs déjà existantes (telles que shmem (CRAY), lapi (IBM), ...) en proposant ses propres primitives RMA. Via ces sous-programmes, un processus a directement accès (en lecture, écriture ou mise à jour) à la mémoire d'un autre processus distant. Dans cette approche, le processus distant n'a donc pas à intervenir dans la procédure de transfert des données.

Les principaux avantages sont les suivants :

- ☞ des performances améliorées lorsque le matériel le permet,
- ☞ une programmation plus simple de certains algorithmes.

## 5.1.3 – Approche RMA de MPI

L'approche RMA de MPI peut être divisée en trois parties distinctes :

- ❶ définition sur chaque processus d'une zone mémoire (fenêtre mémoire locale) visible et susceptible d'être accédée par des processus distants ;
- ❷ déclenchement du transfert des données directement de la mémoire d'un processus à celle d'un autre processus. Il faut alors spécifier le type, le nombre et la localisation initiale et finale des données.
- ❸ achèvement des transferts en cours par une étape de synchronisation, les données étant alors réellement disponibles pour les calculs.

## 5.2 – Notion de fenêtre mémoire

Tous les processus prenant part à une opération de copie de mémoire à mémoire doivent spécifier quelle partie de leur mémoire va être accessible aux autres processus ; c'est la notion de fenêtre mémoire.

- ➡ Plus précisément, l'opération collective `MPI_WIN_CREATE()` permet la création d'un objet MPI fenêtre. Cet objet est composé, pour chaque processus, d'une zone mémoire spécifique appelée fenêtre mémoire locale. Au niveau de chaque processus, une fenêtre mémoire locale est caractérisée par son adresse de départ, sa taille en octets (qui peut être nulle) et la taille de l'unité de déplacement à l'intérieur de cette fenêtre (en octets). Ces caractéristiques peuvent être différentes sur chacun des processus.



Exemple :

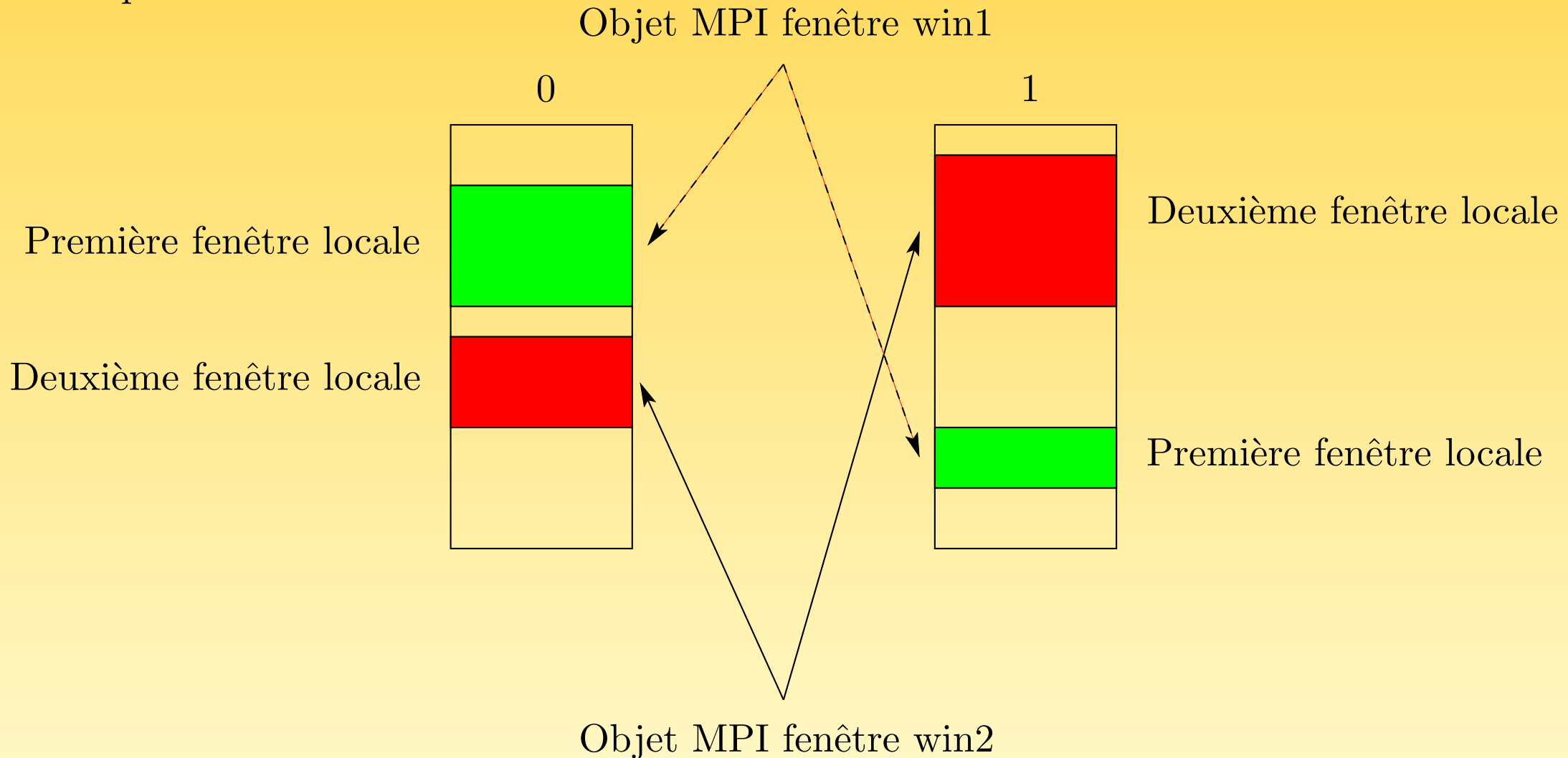


FIG. 21 – Création de deux objets MPI fenêtre, win1 et win2

- ➡ Une fois les transferts terminés, on doit libérer la fenêtre avec le sous-programme `MPI_WIN_FREE()`.
- ➡ `MPI_WIN_GET_ATTR()` permet de connaître les caractéristiques d'une fenêtre mémoire locale en utilisant les mots clés `MPI_WIN_BASE`, `MPI_WIN_SIZE` ou `MPI_WIN_DISP_UNIT`.

Remarque :

- ➡ Le choix de l'unité de déplacement associée à la fenêtre mémoire locale est important (indispensable dans un environnement hétérogène et facilitant le codage dans tous les cas). L'obtention de la taille d'un type MPI prédéfini se fait en appelant le sous-programme `MPI_TYPE_SIZE()`.

```
1 program fenetre
2
3 use mpi
4 implicit none
5
6 integer :: code, rang, taille_reel, win, n=4
7 integer (kind=MPI_ADDRESS_KIND) :: dim_win, taille, base, unite
8 real(kind=kind(1.d0)), dimension(:), allocatable :: win_local
9 logical :: flag
10
11 call MPI_INIT(code)
12 call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
13 call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, taille_reel, code)
14
15 if (rang==0) n=0
16 allocate(win_local(n))
17 dim_win = taille_reel*n
18
19 call MPI_WIN_CREATE(win_local, dim_win, taille_reel, MPI_INFO_NULL, &
20                    MPI_COMM_WORLD, win, code)
```

```
21 call MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, taille, flag, code)
22 call MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, code)
23 call MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, unite, flag, code)
24 call MPI_WIN_FREE(win,code)
25 print *, "processus", rang, "taille, base, unite = " &
26         , taille, base, unite
27 call MPI_FINALIZE(code)
28 end program fenetre
```

```
> mpiexec -n 3 fenetre
```

```
processus 1  taille, base, unite = 32 17248330400 8
processus 0  taille, base, unite = 0 2 8
processus 2  taille, base, unite = 32 17248330400 8
```

## 5.3 – Transfert des données

**MPI** permet à un processus de lire (**MPI\_GET()**), d'écrire (**MPI\_PUT()**) et de mettre à jour (**MPI\_ACCUMULATE()**) des données situées dans la fenêtre mémoire locale d'un processus distant.

On nomme **origine** le processus qui fait l'appel au sous-programme d'initialisation du transfert et **cible** le processus qui possède la fenêtre mémoire locale qui va être utilisée dans la procédure de transfert.

Lors de l'initialisation du transfert, le processus cible n'appelle aucun sous-programme **MPI**. Toutes les informations nécessaires sont spécifiées sous forme de paramètres lors de l'appel au sous-programme **MPI** par l'origine.

En particulier, on trouve :

☞ des paramètres ayant rapport à l'origine :

- le type des éléments ;
- leur nombre ;
- l'adresse mémoire du premier élément.

☞ des paramètres ayant rapport à la cible :

- le rang de la cible ainsi que l'objet MPI fenêtre, ce qui détermine de façon unique une fenêtre mémoire locale ;
- un déplacement dans cette fenêtre locale ;
- le nombre et le type des données à transférer.

```
1 program exemple_put
2 integer :: nb_orig=10, nb_cible=10, cible=1, win, code
3 integer (kind=MPI_ADDRESS_KIND) :: displacement=40
4 integer, dimension(10) :: B
5 ...
6 call MPI_PUT(B,nb_orig,MPI_INTEGER,cible,displacement,nb_cible,MPI_INTEGER,win,code)
7 ...
8 end program exemple_put
```

Déplacement de 40 unités dans la fenêtre locale

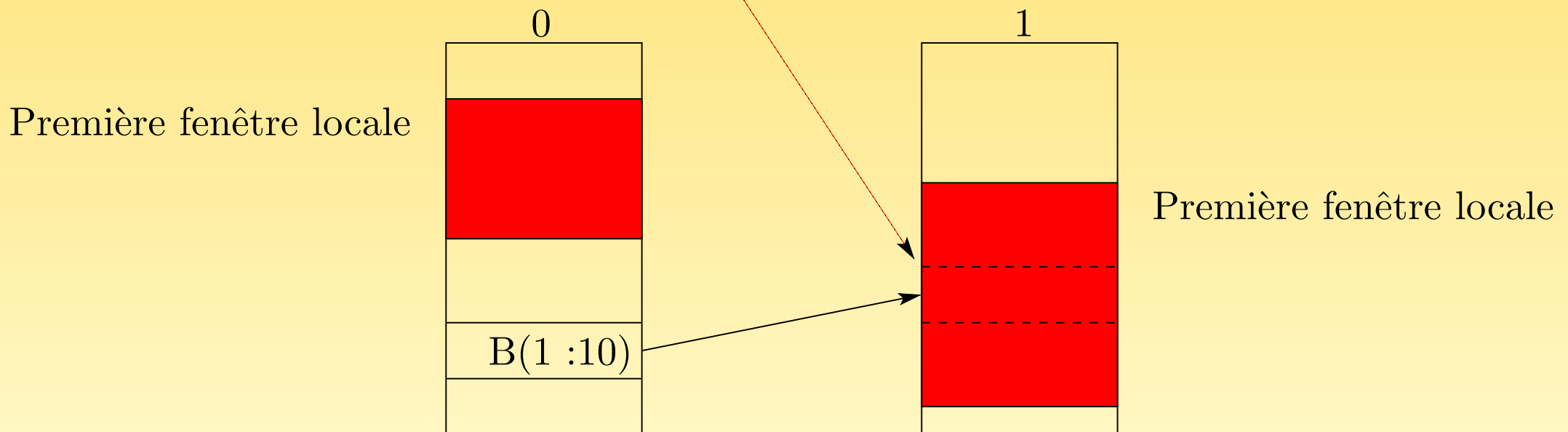


FIG. 22 – Exemple d'un MPI\_PUT

## Remarques :

- ➡ La syntaxe de `MPI_GET` est identique à celle de `MPI_PUT`, seul le sens de transfert des données étant inversé.
- ➡ Les sous-programmes de transfert de données RMA sont des primitives non bloquantes (choix délibéré de `MPI`).
- ➡ Sur le processus cible, les seules données accessibles sont celles contenues dans la fenêtre mémoire locale.
- ➡ `MPI_ACCUMULATE()` admet parmi ses paramètres une opération qui doit être soit du type `MPI_REPLACE`, soit l'une des opérations de réduction prédéfinies : `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, etc. Ce ne peut en aucun cas être une opération définie par l'utilisateur.



## 5.4 – Achèvement du transfert : la synchronisation

Le transfert des données débute après l'appel à l'un des sous-programmes non bloquants (`MPI_PUT()`, ...). Mais quand le transfert est-il terminé et les données réellement disponibles ?



Après une synchronisation qui est à la charge du programmeur.

Ces synchronisations peuvent être classées en deux types :

- ❶ synchronisation de type **cible active** (opération collective, tous les processus associés à la fenêtre prenant part à la synchronisation) ;
- ❷ synchronisation de type **cible passive** (seul le processus origine appelle le sous-programme de synchronisation).

## 5.4.1 – Synchronisation de type cible active

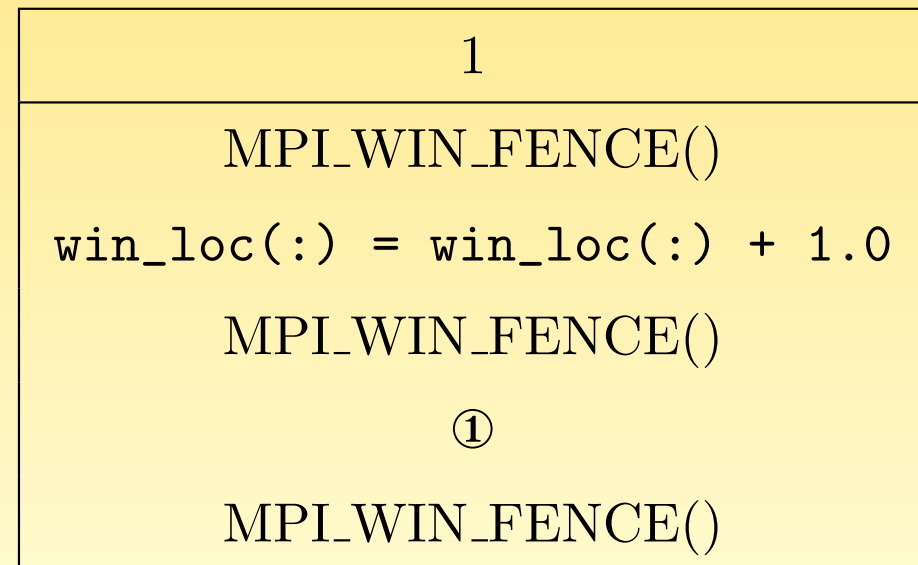
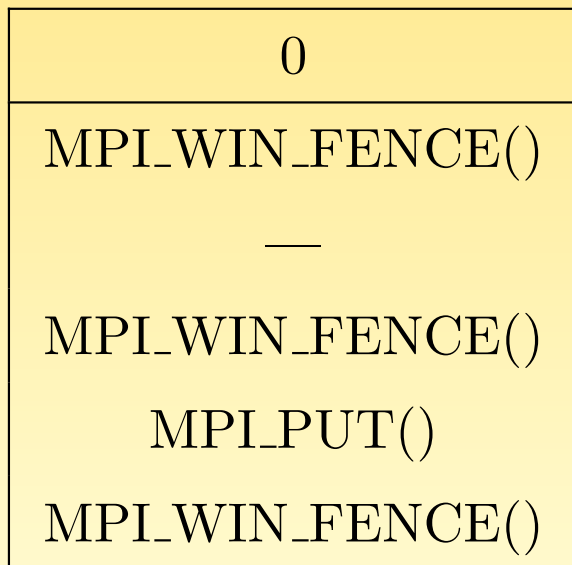
- ➡ Se fait en utilisant le sous-programme `MPI MPI_WIN_FENCE()`.
- ➡ `MPI_WIN_FENCE()` est une opération collective sur tous les processus associés à l'objet MPI fenêtre, argument de la fonction.
- ➡ `MPI_WIN_FENCE()` agit comme une barrière de synchronisation. Elle attend la fin de tous les transferts de données (RMA ou non) utilisant la fenêtre mémoire locale et initiés depuis le dernier appel à `MPI_WIN_FENCE()`.
- ➡ Cette primitive va permettre de séparer les parties calcul du code (où l'on utilise des données de la fenêtre mémoire locale via des *load* ou des *store*) des parties de transfert de données de type RMA.
- ➡ Un argument *assert* de la primitive `MPI_WIN_FENCE()`, de type entier, permet d'affiner son comportement en vue de meilleures performances. Diverses valeurs sont prédéfinies `MPI_MODE_NOSTORE`, `MPI_MODE_NOPUT`, `MPI_MODE_NOPRECEDE`, `MPI_MODE_NOSUCCEED`. Une valeur de zéro pour cet argument est toujours valide.

Remarques :

- ➡ Le fait d'avoir choisi des sous-programmes RMA d'initialisation du transfert non bloquants et une synchronisation pour l'achèvement des transferts en cours autorise l'implémentation à regrouper lors de l'exécution divers transferts vers la même cible en un transfert unique. L'effet de la latence est ainsi réduit et les performances améliorées.
- ➡ Le caractère collectif de la synchronisation a pour conséquence qu'on n'a pas réellement affaire à ce que l'on appelle du « *One Sided Communication* »... En fait tous les processus du communicateur vont devoir prendre part à la synchronisation, ce qui perd de son intérêt !

## Du bon usage de `MPI_WIN_FENCE()`

- ☞ Il faut s'assurer qu'entre deux appels successifs à `MPI_WIN_FENCE()` il n'y a soit que des affectations locales (*load/store*) sur des variables contenues dans la fenêtre mémoire locale du processus, soit que des opérations RMA de type `MPI_PUT()` ou `MPI_ACCUMULATE()`, mais jamais les deux en même temps !



Le programme précédent est-il conforme au bon usage de `MPI_WIN_FENCE()` ?

Tout dépend de la portion de code représentée par ①. Si celle-ci n'engendre pas de *load/store* sur la fenêtre locale (affectation ou utilisation d'une variable stockée dans la fenêtre locale), alors c'est bon ; dans le cas contraire, le résultat est aléatoire.

## Exemple récapitulatif

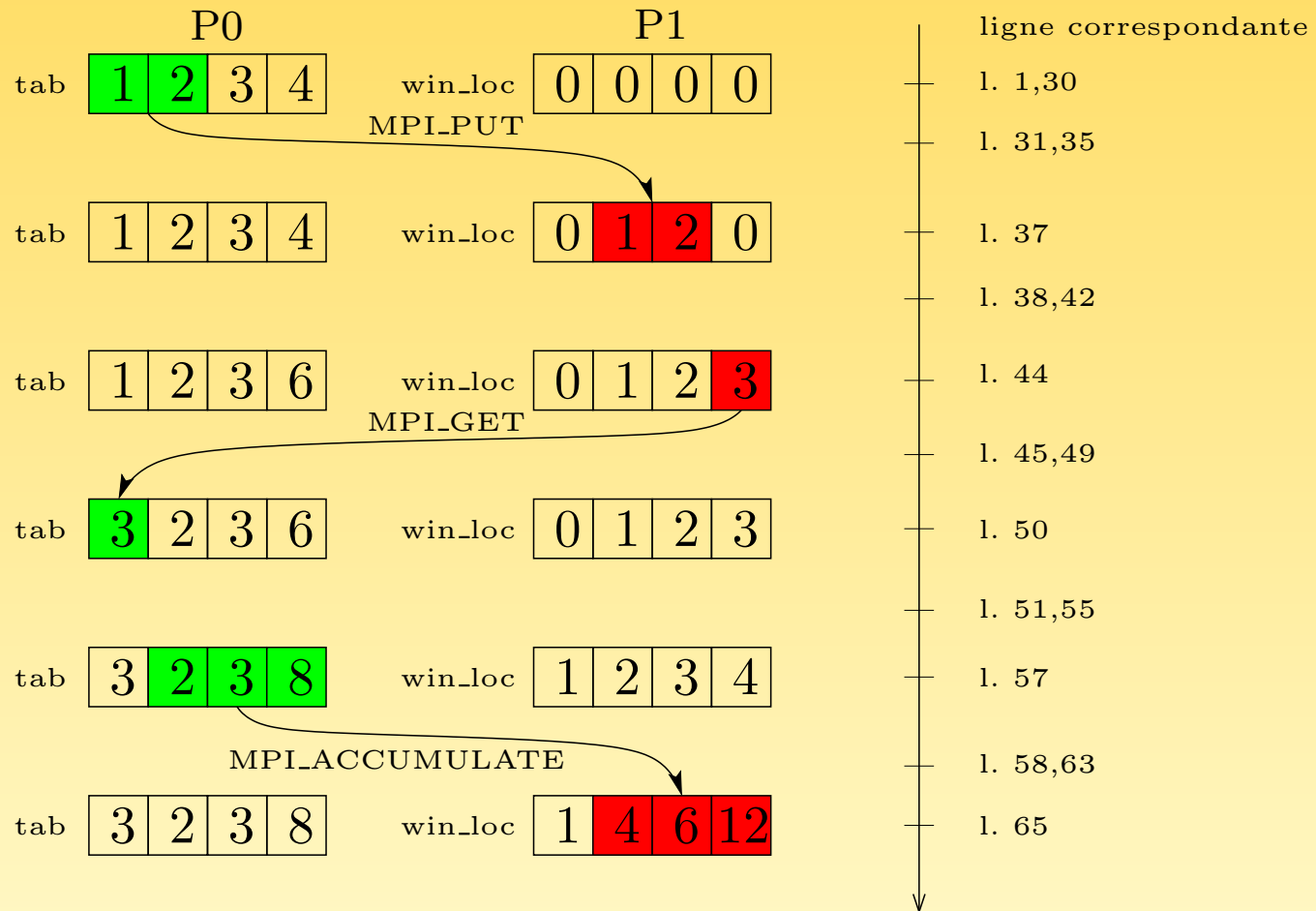


FIG. 23 – Exemple récapitulatif correspondant au code ex\_fence

```
1 program ex_fence
2   use mpi
3   implicit none
4
5   integer, parameter :: assert=0
6   integer :: code, rang, taille_reel, win, i, nb_elements, cible, m=4, n=4
7   integer (kind=MPI_ADDRESS_KIND) :: déplacement, dim_win
8   real(kind=kind(1.d0)), dimension(:), allocatable :: win_local, tab
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
12  call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, taille_reel, code)
13
14  if (rang==0) then
15      n=0
16      allocate(tab(m))
17  endif
18
19  allocate(win_local(n))
20  dim_win = taille_reel*n
21
22  call MPI_WIN_CREATE(win_local, dim_win, taille_reel, MPI_INFO_NULL, &
23                    MPI_COMM_WORLD, win, code)
```

```
24  if (rang==0) then
25      tab(:) = (/ (i, i=1,m) /)
26  else
27      win_local(:) = 0.0
28  end if

29
30  call MPI_WIN_FENCE(assert,win,code)
31  if (rang==0) then
32      cible = 1; nb_elements = 2; déplacement = 1
33      call MPI_PUT(tab, nb_elements, MPI_DOUBLE_PRECISION, cible, déplacement, &
34                  nb_elements, MPI_DOUBLE_PRECISION, win, code)
35  end if

36
37  call MPI_WIN_FENCE(assert,win,code)
38  if (rang==0) then
39      tab(m) = sum(tab(1:m-1))
40  else
41      win_local(n) = sum(win_local(1:n-1))
42  endif

43
44  call MPI_WIN_FENCE(assert,win,code)
45  if (rang==0) then
46      nb_elements = 1; déplacement = m-1
47      call MPI_GET(tab, nb_elements, MPI_DOUBLE_PRECISION, cible, déplacement, &
48                  nb_elements, MPI_DOUBLE_PRECISION, win, code)
49  end if
```



```
50 call MPI_WIN_FENCE(assert,win,code)
51 if (rang==0) then
52     tab(m) = sum(tab(1:m-1))
53 else
54     win_local(:) = win_local(:) + 1
55 endif
56
57 call MPI_WIN_FENCE(assert,win,code)
58 if (rang==0) then
59     nb_elements = m-1; displacement = 1
60     call MPI_ACCUMULATE(tab(2), nb_elements, MPI_DOUBLE_PRECISION, cible, &
61         displacement, nb_elements, MPI_DOUBLE_PRECISION, &
62         MPI_SUM, win, code)
63 end if
64
65 call MPI_WIN_FENCE(assert,win,code)
66 call MPI_WIN_FREE(win,code)
67
68 if (rang==0) then
69     print *, "processus", rang, "tab=", tab(:)
70 else
71     print *, "processus", rang, "win_local=", win_local(:)
72 endif
73
74 call MPI_FINALIZE(code)
75 end program ex_fence
```

## Quelques précisions et restrictions...

- ➡ Il est possible de travailler sur des fenêtres mémoire locales différentes qui se recouvrent, même si cela n'est pas recommandé (une telle utilisation impliquant de trop nombreuses restrictions). Dans la suite on supposera ne pas être dans ce cas.
- ➡ Il faut toujours séparer par un appel à `MPI_WIN_FENCE()` un *store* et l'appel à un sous-programme `MPI_PUT()` ou `MPI_ACCUMULATE()` accédant à la même fenêtre mémoire locale même à des endroits différents ne se recouvrant pas.
- ➡ Entre deux appels successifs au sous-programme `MPI_WIN_FENCE()`, on a les contraintes suivantes :
  - les sous-programmes `MPI_PUT()` n'admettent pas le recouvrement à l'intérieur d'une même fenêtre mémoire locale. En d'autres termes, les zones mémoires mises en jeu lors d'appels à plusieurs sous-programmes `MPI_PUT()` agissant sur la même fenêtre mémoire locale, ne doivent pas se recouvrir ;

- les sous-programmes `MPI_ACCUMULATE()` admettent le recouvrement à l'intérieur d'une même fenêtre mémoire locale, à la condition que les types des données et l'opération de réduction utilisés soient identiques lors de tous ces appels ;
- les sous-programmes `MPI_PUT()` et `MPI_ACCUMULATE()` utilisés consécutivement n'admettent pas le recouvrement à l'intérieur d'une même fenêtre mémoire locale ;
- un *load* et un appel au sous-programme `MPI_GET()` peuvent accéder concurremment à n'importe quelle partie de la fenêtre locale, pourvu qu'elle n'ait pas été mise à jour auparavant soit par un *store*, soit lors de l'appel à un sous-programme de type `MPI_PUT()` ou `MPI_ACCUMULATE()`.

## 5.4.2 – Synchronisation de type cible passive

- ➡ Se fait via les appels aux sous-programmes `MPI MPI_WIN_LOCK()` et `MPI_WIN_UNLOCK()`.
- ➡ Contrairement à la synchronisation par `MPI_WIN_FENCE()` (qui est une opération collective de type barrière), ici seul le processus origine va participer à la synchronisation. De ce fait tous les appels nécessaires au transfert des données (initialisation du transfert, synchronisation) ne font intervenir que le processus origine ; c'est du vrai « *One Sided Communication* ».
- ➡ Les opérations de *lock* et d'*unlock* ne s'appliquent qu'à une fenêtre mémoire locale donnée (i.e. identifiée par un numéro de processus cible et un objet MPI fenêtre). La période qui commence au *lock* et se termine à l'*unlock* est appelée une période d'accès à la fenêtre mémoire locale. Ce n'est que durant cette période que le processus origine va avoir accès à la fenêtre mémoire locale du processus cible.

- ➡ Pour l'utiliser, il suffit pour le processus origine d'entourer l'appel aux primitives RMA d'initialisation de transfert de données par `MPI_WIN_LOCK()` et `MPI_WIN_UNLOCK()`. Pour le processus cible, aucun appel de sous-programmes `MPI` n'est à faire.
- ➡ Lorsque `MPI_WIN_UNLOCK()` rend la main, tous les transferts de données initiés après le `MPI_WIN_LOCK()` sont terminés.
- ➡ Le premier argument de `MPI_WIN_LOCK()` permet de spécifier si le fait de faire plusieurs accès simultanés via des opérations de RMA sur une même fenêtre mémoire locale est autorisé (`MPI_LOCK_SHARED`) ou non (`MPI_LOCK_EXCLUSIVE`).
- ➡ Une utilisation basique des synchronisations de type cible passive consiste à créer des versions bloquantes des RMA (*put*, *get*, *accumulate*) sans que la cible ait besoin de faire appel à des sous-programmes `MPI`.

```
1 subroutine get_bloquant(orig_addr, orig_count, orig_datatype, target_rank, &
2                       target_disp, target_count, target_datatype, win, code)
3 integer, intent(in) :: orig_count, orig_datatype, target_rank, target_count, &
4                       target_datatype, win
5 integer, intent(out) :: code
6 integer(kind=MPI_ADDRESS_KIND), intent(in) :: target_disp
7 real(kind=kind(1.d0)), dimension(:) :: orig_addr
8
9 call MPI_WIN_LOCK(MPI_LOCK_SHARED, target_rank, 0, win, code)
10 call MPI_GET(orig_addr, orig_count, orig_datatype, target_rank, target_disp, &
11             target_count, target_datatype, win, code)
12 call MPI_WIN_UNLOCK(target_rank, win, code)
13 end subroutine get_bloquant
```

## Remarque concernant les codes Fortran

Pour être portable, lors de l'utilisation des synchronisations de type cible passive (`MPI_WIN_LOCK()`, `MPI_WIN_UNLOCK()`), il faut allouer la fenêtre mémoire avec `MPI_ALLOC_MEM()`. Cette fonction admet comme argument des pointeurs de type C (i.e. pointeurs Fortran CRAY, qui ne font pas partie de la norme Fortran95). Dans le cas où ces derniers ne sont pas disponibles, il faut utiliser un programme C pour faire l'allocation de la fenêtre mémoire...

## 5.5 – Conclusions

- ➡ Les concepts RMA de **MPI** sont compliqués à mettre en œuvre sur des applications non triviales. Une connaissance approfondie de la norme est nécessaire pour ne pas tomber dans les nombreux pièges.
- ➡ Les performances peuvent être très variables d'une implémentation à l'autre.
- ➡ L'intérêt du concept RMA de **MPI** réside essentiellement dans l'approche cible passive. C'est seulement dans ce cas que l'utilisation des sous-programmes RMA est réellement indispensable (application nécessitant qu'un processus accède à des données appartenant à un processus distant sans interruption de ce dernier...).



## 6 – Types de données dérivés

### 6.1 – 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_CREATE_HVECTOR()`.

À chaque fois que l'on crée un type de données, il faut le valider à l'aide du sous-programme `MPI_TYPE_COMMIT()`.

Si on souhaite réutiliser le même type, on doit le libérer avec le sous-programme `MPI_TYPE_FREE()`.

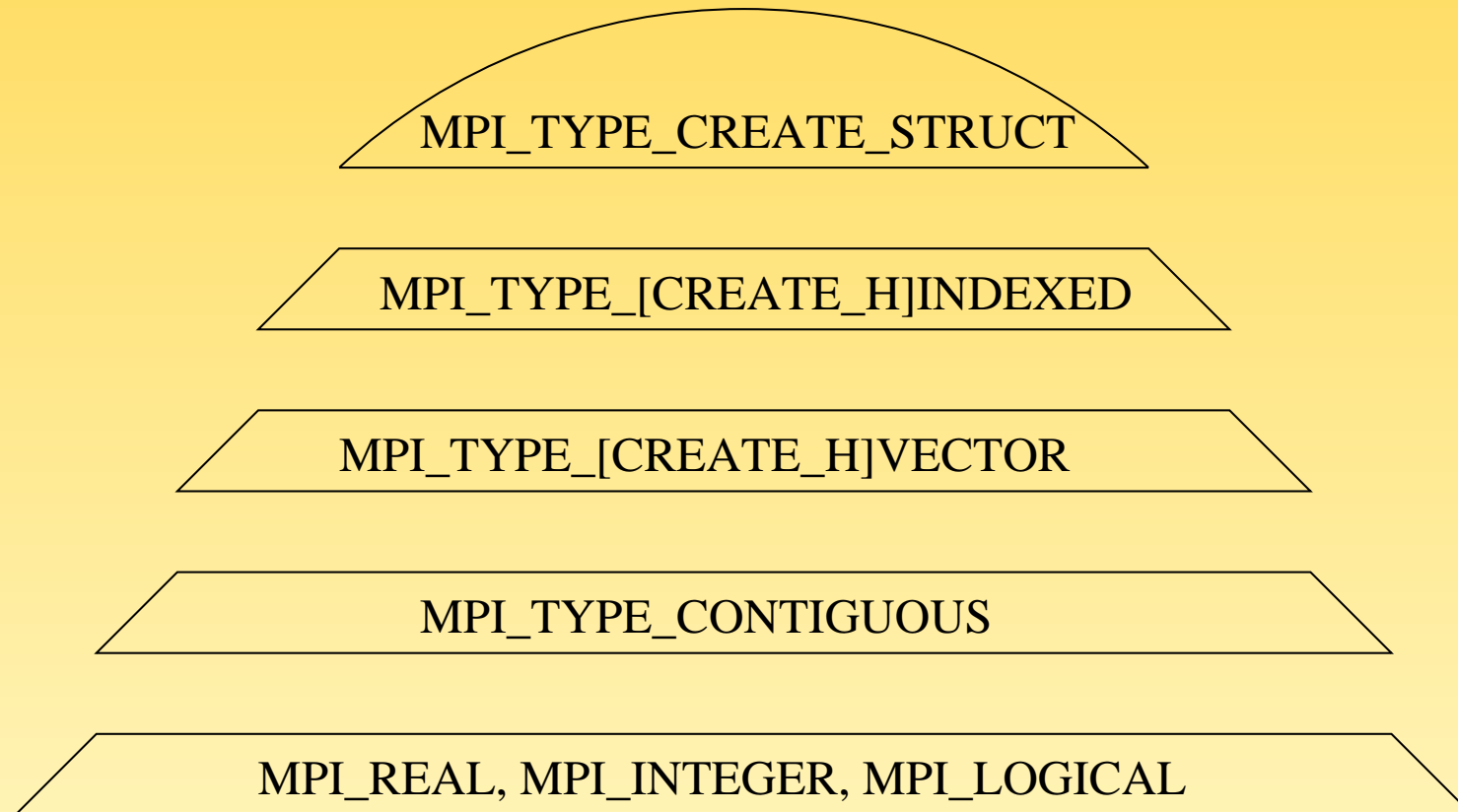


FIG. 24 – Hiérarchie des constructeurs de type MPI

## 6.2 – Types contigus

➔ `MPI_TYPE_CONTIGUOUS()` crée une structure de données à partir d'un ensemble homogène de type prédéfini de données contiguës en mémoire.

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

```
call MPI_TYPE_CONTIGUOUS(5, MPI_REAL, nouveau_type, code)
```

FIG. 25 – Sous-programme `MPI_TYPE_CONTIGUOUS`

```
integer, intent(in)  :: nombre, ancien_type  
integer, intent(out) :: nouveau_type, code  
  
call MPI_TYPE_CONTIGUOUS(nombre, ancien_type, nouveau_type, code)
```

## 6.3 – Types avec un pas constant

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

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

```
call MPI_TYPE_VECTOR(6,1,5,MPI_REAL,nouveau_type,code)
```

FIG. 26 – Sous-programme MPI\_TYPE\_VECTOR

```
integer, intent(in)  :: nombre_bloc, longueur_bloc  
integer, intent(in)  :: pas ! donné en éléments  
integer, intent(in)  :: ancien_type  
integer, intent(out) :: nouveau_type, code
```

```
call MPI_TYPE_VECTOR(nombre_bloc, longueur_bloc, pas, ancien_type, nouveau_type, code)
```

- ➡ `MPI_TYPE_CREATE_HVECTOR()` crée une structure de données à partir d'un ensemble homogène de type prédéfini 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_INTEGER`, `MPI_REAL`,...) mais un type plus complexe construit à l'aide des sous-programmes `MPI` vus précédemment, parce qu'alors le pas ne peut plus être exprimé en nombre d'éléments du type générique.

```
integer, intent(in) :: nombre_bloc, longueur_bloc
integer(kind=MPI_ADDRESS_KIND), intent(in) :: pas ! donné en octets
integer, intent(in) :: ancien_type
integer, intent(out) :: nouveau_type, code

call MPI_TYPE_CREATE_HVECTOR(nombre_bloc, longueur_bloc, pas,
                             ancien_type, nouveau_type, code)
```

## 6.4 – Autres sous-programmes

- ☞ Il est nécessaire de valider tout nouveau type de données dérivé à l'aide du sous-programme `MPI_TYPE_COMMIT()`.

```
integer, intent(inout) :: nouveau_type
integer, intent(out)   :: code

call MPI_TYPE_COMMIT(nouveau_type, code)
```

- ☞ La libération d'un type de données dérivé se fait par le sous-programme `MPI_TYPE_FREE()`.

```
integer, intent(inout) :: nouveau_type
integer, intent(out)   :: code

call MPI_TYPE_FREE(nouveau_type, code)
```

## 6.5 – Exemples

```
1 program colonne
2   use mpi
3   implicit none
4
5   integer, parameter           :: nb_lignes=5,nb_colonnes=6
6   integer, parameter          :: etiquette=100
7   real, dimension(nb_lignes,nb_colonnes) :: a
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9   integer                     :: rang,code,type_colonne
10
11  call MPI_INIT(code)
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14  ! Initialisation de la matrice sur chaque processus
15  a(:,:) = real(rang)
16
17  ! Définition du type type_colonne
18  call MPI_TYPE_CONTIGUOUS(nb_lignes,MPI_REAL,type_colonne,code)
19
20  ! Validation du type type_colonne
21  call MPI_TYPE_COMMIT(type_colonne,code)
```

```
22 ! Envoi de la première colonne
23 if ( rang == 0 ) then
24     call MPI_SEND(a(1,1),1,type_colonne,1,etiquette,MPI_COMM_WORLD,code)
25
26 ! Réception dans la dernière colonne
27 elseif ( rang == 1 ) then
28     call MPI_RECV(a(1,nb_colonnes),1,type_colonne,0,etiquette,&
29                 MPI_COMM_WORLD,statut,code)
30 end if
31
32 ! Libère le type
33 call MPI_TYPE_FREE(type_colonne,code)
34
35 call MPI_FINALIZE(code)
36
37 end program colonne
```



## Le type « ligne d'une matrice »

```
1 program ligne
2   use mpi
3   implicit none
4
5   integer, parameter           :: nb_lignes=5,nb_colonnes=6
6   integer, parameter           :: etiquette=100
7   real, dimension(nb_lignes,nb_colonnes):: a
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9   integer                       :: rang,code,type_ligne
10
11  call MPI_INIT(code)
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14  ! Initialisation de la matrice sur chaque processus
15  a(:,:) = real(rang)
16
17  ! Définition du type type_ligne
18  call MPI_TYPE_VECTOR(nb_colonnes,1,nb_lignes,MPI_REAL,type_ligne,code)
19
20  ! Validation du type type_ligne
21  call MPI_TYPE_COMMIT(type_ligne,code)
```

```
22 ! Envoi de la deuxième ligne
23 if ( rang == 0 ) then
24     call MPI_SEND(a(2,1),1,type_ligne,1,etiquette,MPI_COMM_WORLD,code)
25
26 ! Réception dans l'avant-dernière ligne
27 elseif ( rang == 1 ) then
28     call MPI_RECV(a(nb_lignes-1,1),1,type_ligne,0,etiquette,&
29                 MPI_COMM_WORLD,statut,code)
30 end if
31
32 ! Libère le type type_ligne
33 call MPI_TYPE_FREE(type_ligne,code)
34
35 call MPI_FINALIZE(code)
36
37 end program ligne
```

## Le type « bloc d'une matrice »

```
1 program bloc
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_lignes=5,nb_colonnes=6
6   integer, parameter          :: etiquette=100
7   integer, parameter          :: nb_lignes_bloc=2,nb_colonnes_bloc=3
8   real, dimension(nb_lignes,nb_colonnes):: a
9   integer, dimension(MPI_STATUS_SIZE) :: statut
10  integer                      :: rang,code,type_bloc
11
12  call MPI_INIT(code)
13  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15  ! Initialisation de la matrice sur chaque processus
16  a(:,:) = real(rang)
17
18  ! Création du type type_bloc
19  call MPI_TYPE_VECTOR(nb_colonnes_bloc,nb_lignes_bloc,nb_lignes,&
20                      MPI_REAL,type_bloc,code)
21
22  ! Validation du type type_bloc
23  call MPI_TYPE_COMMIT(type_bloc,code)
```

```
24 ! Envoi d'un bloc
25 if ( rang == 0 ) then
26   call MPI_SEND(a(1,1),1,type_bloc,1,etiquette,MPI_COMM_WORLD,code)
27
28 ! Réception du bloc
29 elseif ( rang == 1 ) then
30   call MPI_RECV(a(nb_lignes-1,nb_colonnes-2),1,type_bloc,0,etiquette,&
31               MPI_COMM_WORLD,statut,code)
32 end if
33
34 ! Libération du type type_bloc
35 call MPI_TYPE_FREE(type_bloc,code)
36
37 call MPI_FINALIZE(code)
38
39 end program bloc
```

## 6.6 – 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_INTEGER`, `MPI_REAL`, ...) 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.

nb=3, longueurs\_blocs=(2,1,3), déplacements=(0,3,7)

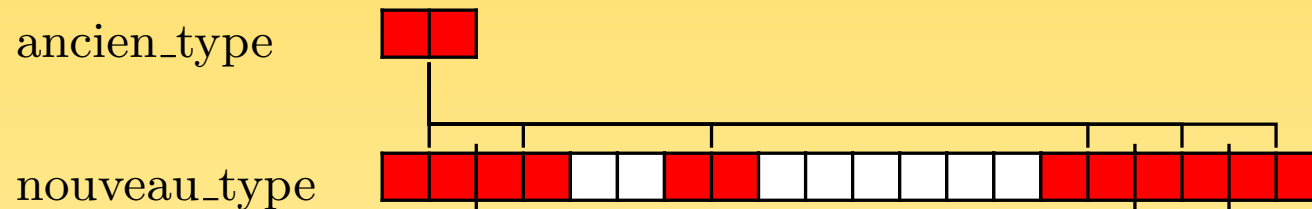


FIG. 27 – Le constructeur MPI\_TYPE\_INDEXED

```
integer,intent(in)                :: nb
integer,intent(in),dimension(nb)  :: longueurs_blocs
! Attention les déplacements sont donnés en éléments
integer,intent(in),dimension(nb)  :: déplacements
integer,intent(in)                :: ancien_type

integer,intent(out)               :: nouveau_type,code

call MPI_TYPE_INDEXED(nb,longueurs_blocs,deplacements,ancien_type,nouveau_type,code)
```

nb=4, longueurs\_blocs=(2,1,2,1), déplacements=(2,10,14,24)

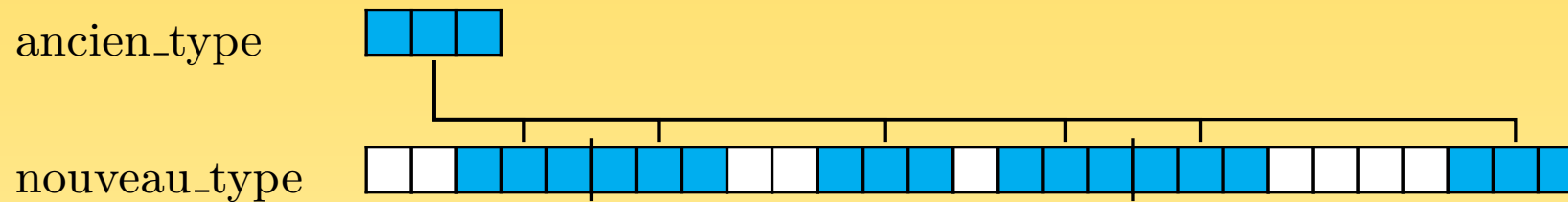


FIG. 28 – Le constructeur MPI\_TYPE\_CREATE\_HINDEXED

```
integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: longueurs_blocs
! Attention les déplacements sont donnés en octets
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb) :: déplacements
integer,intent(in) :: ancien_type

integer,intent(out) :: nouveau_type,code

call MPI_TYPE_CREATE_HINDEXED(nb,longueurs_blocs,deplacements,
ancien_type,nouveau_type,code)
```

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 via l'instruction `MPI_SENDRECV_REPLACE()` ;
- ④ libère ses ressources et quitte `MPI`.



Processus 0

Avant

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Après

1	-2	-3	-5	-8	-14	-22	-32
2	10	-4	-6	-11	-15	-23	-38
3	11	19	-7	-12	-16	-24	-39
4	12	20	28	-13	-20	-29	-40
5	13	21	29	37	-21	-30	-47
6	14	22	30	38	46	-31	-48
7	15	23	31	39	47	55	-56
8	16	24	32	40	48	56	64

Processus 1

-1	-9	-17	-25	-33	-41	-49	-57
-2	-10	-18	-26	-34	-42	-50	-58
-3	-11	-19	-27	-35	-43	-51	-59
-4	-12	-20	-28	-36	-44	-52	-60
-5	-13	-21	-29	-37	-45	-53	-61
-6	-14	-22	-30	-38	-46	-54	-62
-7	-15	-23	-31	-39	-47	-55	-63
-8	-16	-24	-32	-40	-48	-56	-64

-1	-9	-17	-25	-33	-41	-49	-57
9	-10	-18	-26	-34	-42	-50	-58
17	34	-19	-27	-35	-43	-51	-59
18	35	44	-28	-36	-44	-52	-60
25	36	45	52	-37	-45	-53	-61
26	41	49	53	58	-46	-54	-62
27	42	50	54	59	61	-55	-63
33	43	51	57	60	62	63	-64

FIG. 29 – Échanges entre les 2 processus

```
1 program triangle
2   use mpi
3   implicit none
4
5   integer,parameter           :: n=8,etiquette=100
6   real,dimension(n,n)        :: a
7   integer,dimension(MPI_STATUS_SIZE) :: statut
8   integer                     :: i,code
9   integer                     :: rang,type_triangle
10  integer,dimension(n)        :: longueurs_blocs,deplacements
11
12  call MPI_INIT(code)
13  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15  ! Initialisation de la matrice sur chaque processus
16  a(:,:) = reshape( (/ (sign(i,-rang),i=1,n*n) /), (/n,n/))
17
18  ! Création du type matrice triangulaire sup pour le processus 0
19  ! et du type matrice triangulaire inférieure pour le processus1
20  if (rang == 0) then
21    longueurs_blocs(:) = (/ (i-1,i=1,n) /)
22    deplacements(:)    = (/ (n*(i-1),i=1,n) /)
23  else
24    longueurs_blocs(:) = (/ (n-i,i=1,n) /)
25    deplacements(:)    = (/ (n*(i-1)+i,i=1,n) /)
26  endif
```

```
27 call MPI_TYPE_INDEXED(n,longueurs_blocs,deplacements,MPI_REAL,type_triangle,code)
28 call MPI_TYPE_COMMIT(type_triangle,code)
29
30 ! Permutation des matrices triangulaires supérieure et inférieure
31 call MPI_SENDRECV_REPLACE(a,1,type_triangle,mod(rang+1,2),etiquette,mod(rang+1,2), &
32     etiquette,MPI_COMM_WORLD,statut,code)
33
34 ! Libération du type triangle
35 call MPI_TYPE_FREE(type_triangle,code)
36
37 call MPI_FINALIZE(code)
38
39 end program triangle
```

## 6.7 – Construction de sous-tableaux

☞ Le sous-programme `MPI_TYPE_CREATE_SUBARRAY()` permet de créer un sous-tableau à partir d'un tableau.

```
integer, intent(in)                :: nb_dims
integer, dimension(ndims), intent(in) :: profil_tab, profil_sous_tab, coord_debut
integer, intent(in)                :: ordre, ancien_type
integer, intent(out)               :: nouveau_type, code
call MPI_TYPE_CREATE_SUBARRAY(nb_dims, profil_tab, profil_sous_tab, coord_debut,
                              ordre, ancien_type, nouveau_type, code)
```

## Rappels sur le vocabulaire relatif aux tableaux en Fortran 95

- ☞ Le **rang** d'un tableau est son nombre de dimensions.
- ☞ L'**étendue** d'un tableau est son nombre d'éléments dans une dimension.
- ☞ Le **profil** d'un tableau est un vecteur dont chaque dimension est l'**étendue** du tableau dans la dimension correspondante.

Soit par exemple le tableau  $T(10, 0:5, -10:10)$ . Son rang est **3**, son étendue dans la première dimension est **10**, dans la seconde **6** et dans la troisième **21**, son profil est le vecteur **(10,6,21)**.

- ➡ **nb\_dims** : rang du tableau
- ➡ **profil\_tab** : profil du tableau à partir duquel on va extraire un sous-tableau
- ➡ **profil\_sous\_tab** : profil du sous-tableau
- ➡ **coord\_debut** : coordonnées de départ si les indices du tableau commençaient à 0.  
Par exemple, si on veut que les coordonnées de départ du sous-tableau soient `tab(2,3)`, il faut que `coord_debut(:)=( / 1,2 /)`
- ➡ **ordre** : ordre de stockage des éléments
  1. **MPI\_ORDER\_FORTRAN** spécifie le mode de stockage en Fortran, c.-à-d. suivant les colonnes
  2. **MPI\_ORDER\_C** spécifie le mode de stockage en C, c.-à-d. suivant les lignes

AVANT			APRES		
1	5	9	1	5	9
2	6	10	-7	-11	10
3	7	11	-8	-12	11
4	8	12	4	8	12
Processus 0			Processus 0		
-1	-5	-9	-1	-5	-9
-2	-6	-10	-2	-6	-10
-3	-7	-11	-3	2	6
-4	-8	-12	-4	3	7
Processus 1			Processus 1		

FIG. 30 – Échanges entre les 2 processus

```
1 program subarray
2   use mpi
3   implicit none
4
5   integer,parameter                :: nb_lignes=4,nb_colonnes=3,&
6                                   etiquette=1000,nb_dims=2
7   integer                          :: code,rang,type_sous_tab,i
8   integer,dimension(nb_lignes,nb_colonnes) :: tab
9   integer,dimension(nb_dims)       :: profil_tab,profil_sous_tab,coord_debut
10  integer,dimension(MPI_STATUS_SIZE) :: statut
11
12  call MPI_INIT(code)
13  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15  !Initialisation du tableau tab sur chaque processus
16  tab(:,:) = reshape( (/ (sign(i,-rang),i=1,nb_lignes*nb_colonnes) /) , &
17                    (/ nb_lignes,nb_colonnes /) )
18
```



```
19 !Profil du tableau tab à partir duquel on va extraire un sous-tableau
20 profil_tab(:) = shape(tab)
21 !La fonction F95 shape donne le profil du tableau passé en argument.
22 !ATTENTION, si le tableau concerné n'a pas été alloué sur tous les processus,
23 !il faut mettre explicitement le profil du tableau pour qu'il soit connu
24 !sur tous les processus, soit profil_tab(:) = (/ nb_lignes,nb_colonnes /)
25
26 !Profil du sous-tableau
27 profil_sous_tab(:) = (/ 2,2 /)
28
29 !Coordonnées de départ du sous-tableau
30 !Pour le processus 0 on part de l'élément tab(2,1)
31 !Pour le processus 1 on part de l'élément tab(3,2)
32 coord_debut(:) = (/ rang+1,rang /)
33
34 !Création du type dérivé type_sous_tab
35 call MPI_TYPE_CREATE_SUBARRAY(nb_dims,profil_tab,profil_sous_tab,coord_debut,&
36                               MPI_ORDER_FORTRAN,MPI_INTEGER,type_sous_tab,code)
37 call MPI_TYPE_COMMIT(type_sous_tab,code)
38
39 !Permutation du sous-tableau
40 call MPI_SENDRECV_REPLACE(tab,1,type_sous_tab,mod(rang+1,2),etiquette,&
41                           mod(rang+1,2),etiquette,MPI_COMM_WORLD,statut,code)
42 call MPI_TYPE_FREE(type_sous_tab,code)
43 call MPI_FINALIZE(code)
44 end program subarray
```

## 6.8 – Types hétérogènes

- ⇒ Le sous-programme `MPI_TYPE_CREATE_STRUCT()` est le constructeur de types le plus général.
- ⇒ Il a les mêmes fonctionnalités que `MPI_TYPE_INDEXED()` mais permet en plus la réplication de blocs de données de types différents.
- ⇒ Les paramètres de `MPI_TYPE_CREATE_STRUCT()` sont les mêmes que ceux de `MPI_TYPE_INDEXED()` avec en plus :
  - ⇒ le champ *anciens\_types* est maintenant un vecteur de types de données `MPI` ;
  - ⇒ compte tenu de l'hétérogénéité des données et de leur alignement en mémoire, le calcul du déplacement entre deux éléments repose sur la différence de leurs adresses ;
  - ⇒ `MPI`, via `MPI_GET_ADDRESS()`, fournit un sous-programme portable qui permet de retourner l'adresse d'une variable.

nb=5, longueurs\_blocs=(3,1,5,1,1), déplacements=(0,7,11,21,26),

anciens\_types=(type1,type2,type3,type1,type3)

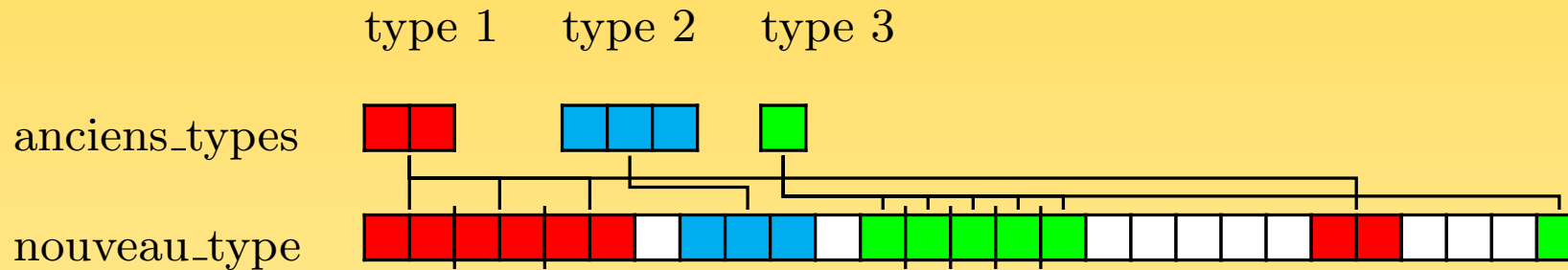


FIG. 31 – Le constructeur MPI\_TYPE\_CREATE\_STRUCT

```
integer, intent(in) :: nb
integer, intent(in), dimension(nb) :: longueurs_blocs
integer(kind=MPI_ADDRESS_KIND), intent(in), dimension(nb) :: déplacements
integer, intent(in), dimension(nb) :: anciens_types

integer, intent(out) :: nouveau_type, code
```

```
call MPI_TYPE_CREATE_STRUCT(nb, longueurs_blocs, déplacements,
                           anciens_types, nouveau_type, code)
```

```
<type>, intent(in) :: variable
integer(kind=MPI_ADDRESS_KIND), intent(out) :: adresse_variable
integer, intent(out) :: code
```

```
call MPI_GET_ADDRESS(variable, adresse_variable, code)
```

```
1 program Interaction_Particules
2   use mpi
3   implicit none
4
5   integer, parameter                :: n=1000, etiquette=100
6   integer, dimension(MPI_STATUS_SIZE) :: statut
7   integer                          :: rang, code, type_particule, i
8   integer, dimension(4)            :: types, longueurs_blocs
9   integer(kind=MPI_ADDRESS_KIND), dimension(4) :: déplacements, adresses
10
11  type Particule
12     character(len=5)                :: categorie
13     integer                        :: masse
14     real, dimension(3)              :: coords
15     logical                        :: classe
16  end type Particule
17  type(Particule), dimension(n)      :: p, temp_p
18
19  call MPI_INIT(code)
20  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
21
22  ! Construction du type de données
23  types = (/MPI_CHARACTER, MPI_INTEGER, MPI_REAL, MPI_LOGICAL/)
24  longueurs_blocs = (/5, 1, 3, 1/)
```

```
25 call MPI_GET_ADDRESS (p(1)%categorie, adresses(1), code)
26 call MPI_GET_ADDRESS (p(1)%masse, adresses(2), code)
27 call MPI_GET_ADDRESS (p(1)%coords, adresses(3), code)
28 call MPI_GET_ADDRESS (p(1)%classe, adresses(4), code)
29
30 ! Calcul des déplacements relatifs à l'adresse de départ
31 do i=1,4
32     déplacements(i)=adresses(i) - adresses(1)
33 end do
34 call MPI_TYPE_CREATE_STRUCT (4, longueurs_blocs, déplacements, types, type_particule, &
35                               code)
36 ! Validation du type structuré
37 call MPI_TYPE_COMMIT (type_particule, code)
38 ! Initialisation des particules pour chaque processus
39 ....
40 ! Envoi des particules de 0 vers 1
41 if (rang == 0) then
42     call MPI_SEND (p(1)%categorie, n, type_particule, 1, etiquette, MPI_COMM_WORLD, code)
43 else
44     call MPI_RECV (temp_p(1)%categorie, n, type_particule, 0, etiquette, MPI_COMM_WORLD, &
45                  statut, code)
46 endif
47
48 ! Libération du type
49 call MPI_TYPE_FREE (type_particule, code)
50 call MPI_FINALIZE (code)
51 end program Interaction_Particules
```

## 6.9 – Sous-programmes annexes

☞ La taille totale d'un type de données : `MPI_TYPE_SIZE()`

```
integer, intent(in)  :: type_donnee
integer, intent(out) :: taille, code

call MPI_TYPE_SIZE(type_donnee,taille,code)
```

☞ La taille ainsi que la borne inférieure d'un type dérivé, en tenant compte des éventuels alignements mémoire : `MPI_TYPE_GET_EXTENT()`

```
integer, intent(in)                :: type_derive
integer(kind=MPI_ADDRESS_KIND), intent(out) :: borne_inf_alignee,taille_alignee
integer, intent(out)                :: code

call MPI_TYPE_GET_EXTENT(type_derive,borne_inf_alignee,taille_alignee,code)
```

☞ On peut modifier la borne inférieure d'un type dérivé (et par voie de conséquence sa taille totale) pour créer un nouveau type adapté du précédent

```
integer, intent(in)                :: ancien_type
integer(kind=MPI_ADDRESS_KIND), intent(in) :: nouvelle_borne_inf,nouvelle_taille
integer, intent(out)                :: nouveau_type,code

call MPI_TYPE_CREATE_RESIZED(ancien_type,nouvelle_borne_inf,nouvelle_taille,
                             nouveau_type,code)
```

```

1 PROGRAM ma_ligne
2   USE mpi
3   IMPLICIT NONE
4   INTEGER, PARAMETER :: nb_lignes=5, nb_colonnes=6, &
5                       demi_ligne=nb_colonnes/2, etiquette=1000
6   INTEGER, DIMENSION(nb_lignes, nb_colonnes) :: A
7   INTEGER :: typeDemiLigne, typeDemiLigne2
8   INTEGER :: code, taille_integer, rang, i
9   INTEGER(kind=MPI_ADDRESS_KIND) :: borneInf=0, tailleDeplacement
10  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: statut

11
12  CALL MPI_INIT(code)
13  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
14  !Initialisation de la matrice A sur chaque processus
15  A(:, :) = RESHAPE( (/ (SIGN(i, -rang), i=1, nb_lignes*nb_colonnes) /), &
16                  (/ nb_lignes, nb_colonnes /) )
17
18  !Construction du type derive typeDemiLigne
19  CALL MPI_TYPE_VECTOR(demi_ligne, 1, nb_lignes, MPI_INTEGER, typeDemiLigne, code)
20  !Connaitre la taille du type de base MPI_INTEGER
21  CALL MPI_TYPE_SIZE(MPI_INTEGER, taille_integer, code)
22  !Construction du type derive typeDemiLigne2
23  tailleDeplacement = taille_integer
24  CALL MPI_TYPE_CREATE_RESIZED(typeDemiLigne, borneInf, tailleDeplacement, &
25                              typeDemiLigne2, code)

```

```
26 !Validation du type typeDemiLigne2
27 CALL MPI_TYPE_COMMIT(typeDemiLigne2,code)
28
29 IF (rang == 0) THEN
30     !Envoi de la matrice A au processus 1 avec le type typeDemiLigne2
31     CALL MPI_SEND(A(1,1), 2, typeDemiLigne2, 1, etiquette, &
32                 MPI_COMM_WORLD, code)
33 ELSE
34     !Reception pour le processus 1 dans la matrice A
35     CALL MPI_RECV(A(1,nb_colonnes-1), 6, MPI_REAL, 0, etiquette,&
36                 MPI_COMM_WORLD,statut, code)
37
38     PRINT *,'Matrice A sur le processus 1'
39     DO i=1,nb_lignes
40         PRINT *,A(i,:)
41     END DO
42
43 END IF
44 CALL MPI_FINALIZE(code)
45 END PROGRAM ma_ligne
```



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

## 6.10 – 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 domaines avec des maillages réguliers ou irréguliers.

## 7 – Optimisations

### 7.1 – Introduction

- ☞ L'optimisation doit être un souci essentiel lorsque la part des communications par rapport aux calculs devient assez importante
- ☞ L'optimisation des communications peut s'accomplir à différents niveaux dont les principaux sont :
  - ① recouvrir les communications par des calculs ;
  - ② éviter la recopie du message dans un espace mémoire temporaire (*buffering*) ou la forcer selon les cas ;
  - ③ minimiser les surcoûts induits par des appels répétitifs aux sous-programmes de communication.

## 7.2 – Programme modèle

```
1 program AOptimiser
2   use mpi
3   implicit none
4
5   integer, parameter           :: na=256,nb=200
6   integer, parameter           :: m=2048,etiquette=1111
7   real, dimension(na,na)       :: a
8   real, dimension(nb,nb)       :: b
9   real, dimension(na)          :: pivota
10  real, dimension(nb)          :: pivotb
11  real, dimension(m,m)         :: c
12  integer                       :: rang,code
13  real(kind=8)                  :: temps_debut,temps_fin,temps_fin_max
14  integer, dimension(MPI_STATUS_SIZE) :: statut
15
16  call MPI_INIT(code)
17  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
18
19  ! Initialisation des tableaux
20  call random_number(a)
21  call random_number(b)
22  call random_number(c)
```

```
23 temps_debut = MPI_WTIME()
24 if (rang == 0) then
25     ! Envoi d'un gros message
26     call MPI_SEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,code)
27     ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
28     call sgetrf(na, na, a, na, pivota, code)
29     ! Calcul modifiant le contenu du tableau C
30     c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
31 elseif (rang == 1) then
32     ! Calcul (factorisation LU avec LAPACK)
33     call sgetrf(na, na, a, na, pivota, code)
34     ! Réception d'un gros message
35     call MPI_RECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,statut,code)
36     ! Calcul dépendant de la réception du message précédent
37     a(:, :) = transpose(c(1:na,1:na)) + a(:, :)
38     ! Calcul indépendant de la réception du message précédent
39     call sgetrf(nb, nb, b, nb, pivotb, code)
40 end if
41 temps_fin = (MPI_WTIME() - temps_debut)
42
43 ! Obtention du temps de restitution maximum
44 call MPI_REDUCE(temps_fin,temps_fin_max,1,MPI_DOUBLE_PRECISION, MPI_MAX,0, &
45               MPI_COMM_WORLD,code)
46 if (rang == 0) print('("Temps : ",f6.3," secondes")'),temps_fin_max
47
48 call MPI_FINALIZE(code)
49 end program AOptimiser
```

```
> mpiexec -n 2 AOptimiser  
Temps : 0.7 secondes
```

Hors communications, le maximum des temps de calcul par processus est de **0,15 secondes**. Ce qui veut dire que les communications prennent environ 78% du temps global!

calcul

communication

## 7.3 – Temps de communication

Que comprend le temps que l'on mesure avec `MPI_WTIME()` ?



- ☞ Latence : temps d'initialisation des paramètres réseaux.
- ☞ Surcoût : temps de préparation du message ; caractéristique liée à l'implémentation MPI et au mode de transfert.

## 7.4 – Quelques définitions

- ❶ **Recopie temporaire d'un message.** C'est la copie du message dans une mémoire tampon locale (*buffer*) avant son envoi. Cette opération est prise en charge par le système MPI dans certains cas.

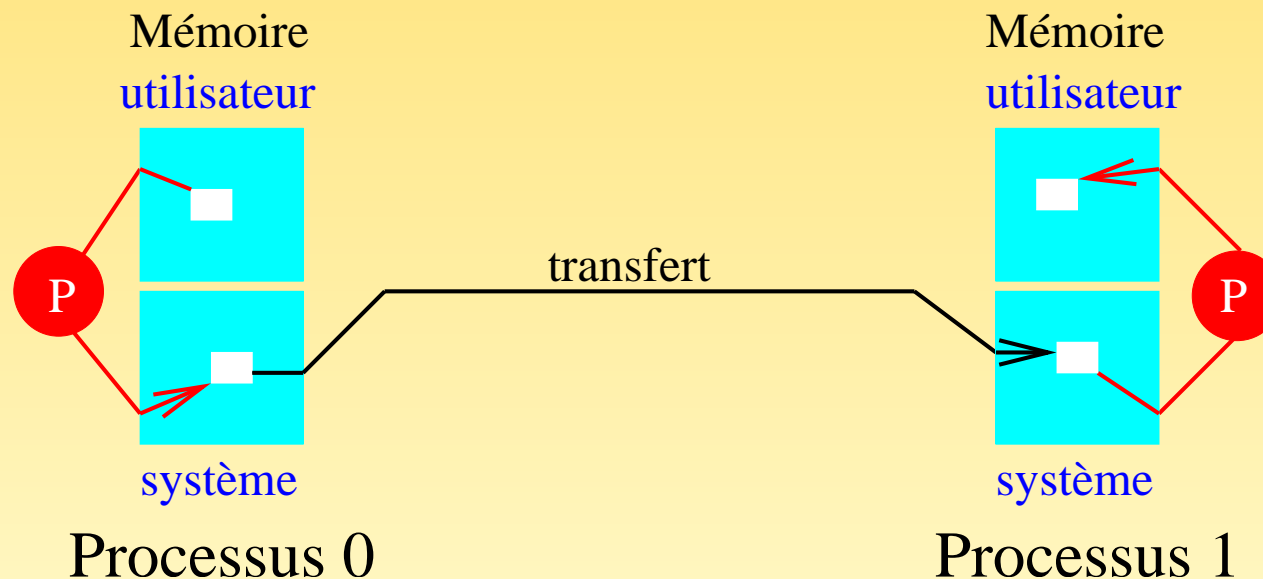


FIG. 32 – Recopie temporaire d'un message



- ② Envoi non bloquant avec recopie temporaire, non couplé avec la réception. L'appel à un sous-programme de ce type retourne au programme appelant même quand la réception n'a pas été postée. La recopie temporaire des messages est l'un des moyens d'implémenter un envoi non bloquant afin de découpler l'envoi de la réception.

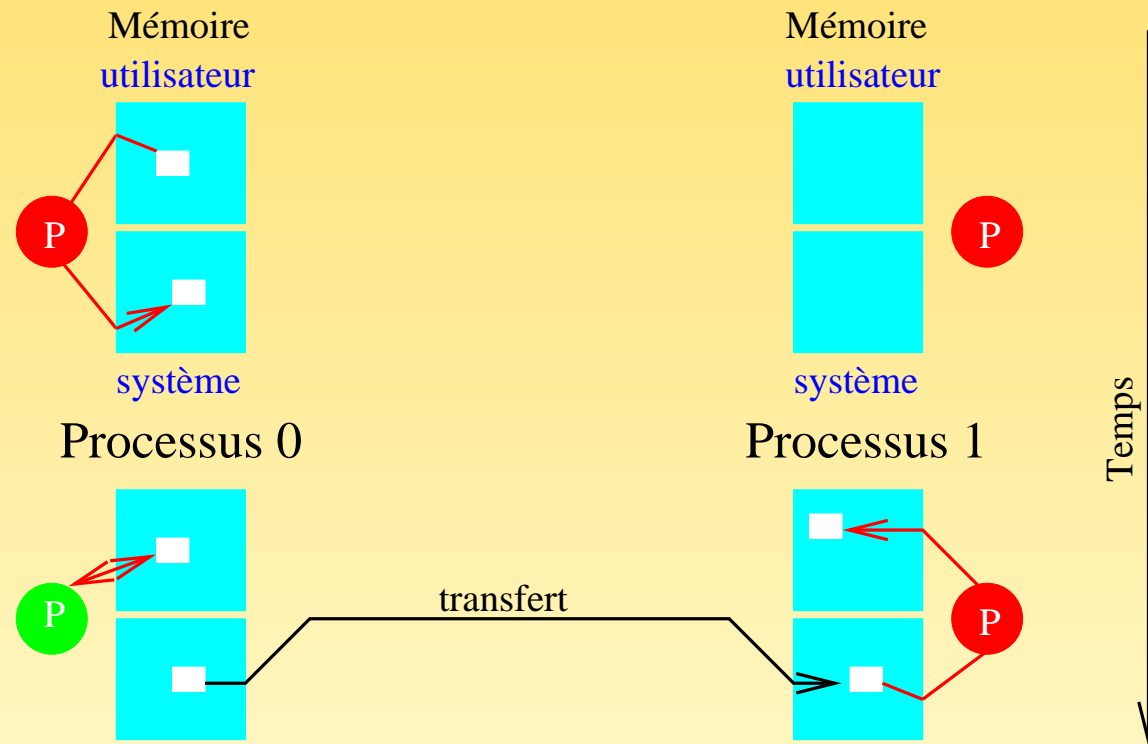


FIG. 33 – Envoi non bloquant avec recopie temporaire du message

- ③ Envoi bloquant sans recopie temporaire, couplé avec la réception. Le message ne quitte le processus émetteur que lorsque le processus récepteur est prêt à le recevoir.

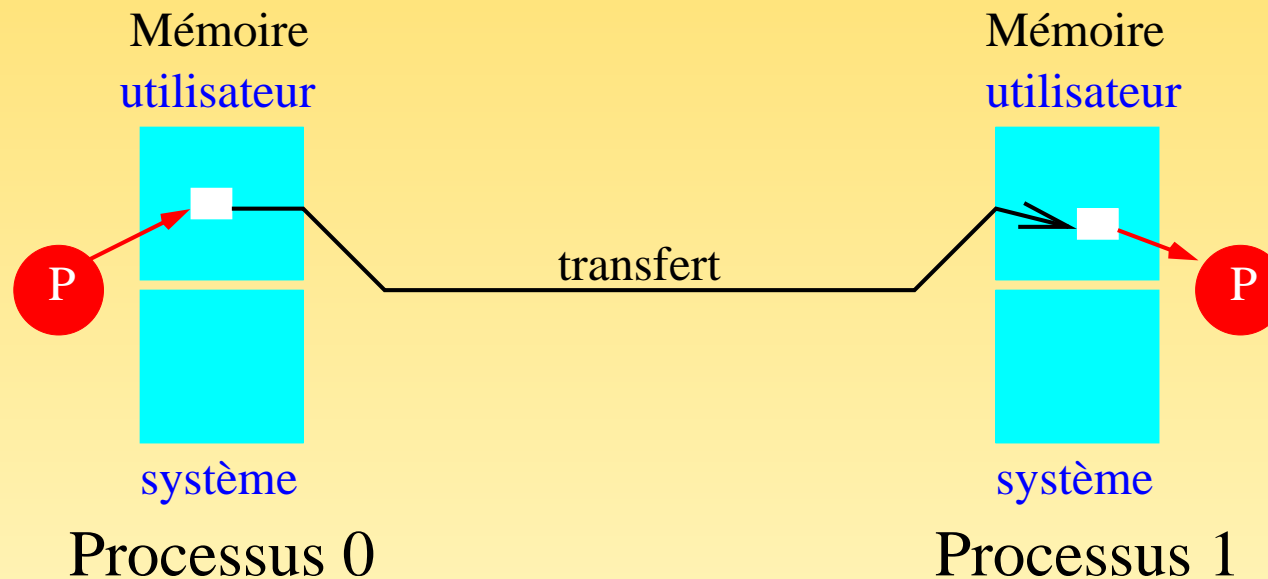


FIG. 34 – Envoi bloquant couplé avec la réception

- ④ Envoi non bloquant sans copie temporaire, couplé avec la réception. L'appel à ce type de sous-programmes retourne immédiatement au programme appelant bien que l'envoi effectif du message reste couplé avec la réception. Il est donc à la charge du programmeur de s'assurer que le message est bien arrivé à sa destination finale avant de pouvoir modifier les données envoyées.

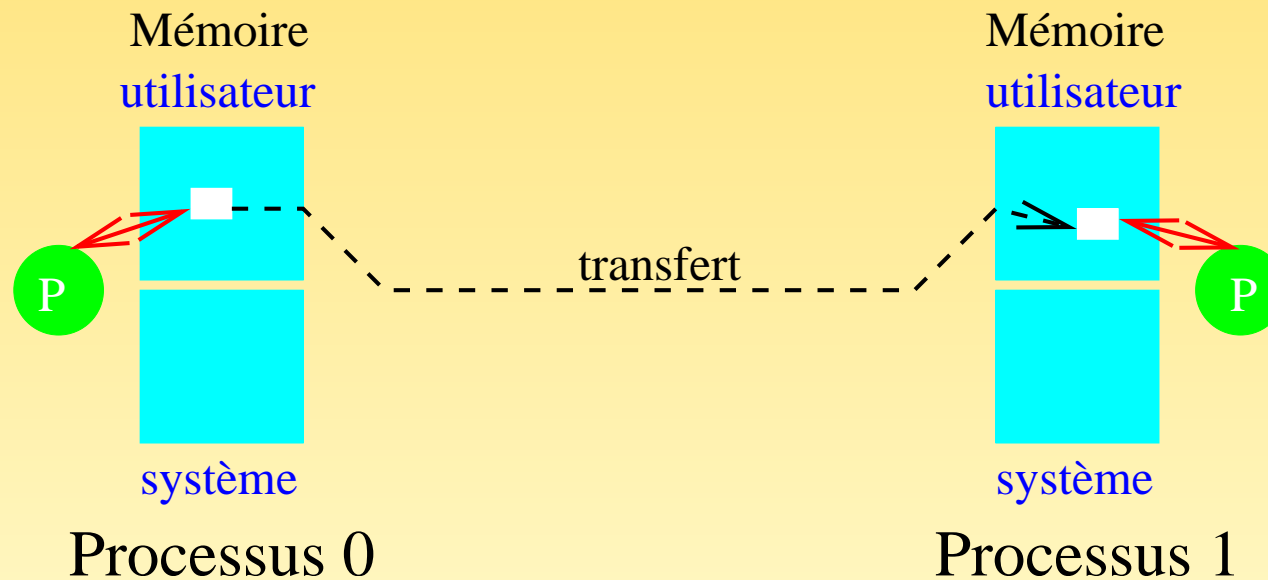


FIG. 35 – Envoi non bloquant couplé avec la réception

## 7.5 – Que fournit MPI ?

- ➡ Avec **MPI** l'envoi d'un message peut se faire suivant différents modes :
- ① *standard* : il est à la charge de **MPI** d'effectuer ou non une copie temporaire du message. Si c'est le cas, l'envoi se termine lorsque la copie temporaire est achevée (l'envoi est ainsi découplé de la réception). Dans le cas contraire, l'envoi se termine quand la réception du message est achevée.
  - ② *synchronous* : l'envoi du message ne se termine que si la réception a été postée et la lecture du message terminée. C'est un envoi couplé avec la réception.
  - ③ *buffered* : il est à la charge du programmeur d'effectuer une copie temporaire du message. L'envoi du message se termine lorsque la copie temporaire est achevée. L'envoi est ainsi découplé de la réception.
  - ④ *ready* : l'envoi du message ne peut commencer que si la réception a été postée auparavant (ce mode est intéressant pour les applications clients-serveurs).

➡ À titre indicatif voici les différents cas envisagés par la norme sachant que les implémentations peuvent être différentes :

modes	bloquant	non-bloquant
envoi <i>standard</i>	MPI_SEND() <sup>a</sup>	MPI_ISEND()
envoi <i>synchronous</i>	MPI_SSEND()	MPI_ISSEND()
envoi <i>buffered</i>	MPI_BSEND()	MPI_IBSEND()
réception	MPI_RECV()	MPI_IRECV()

➡ Remarque : pour une implémentation MPI donnée, un envoi standard peut-être bloquant avec copie temporaire ou bien synchrone avec la réception, ou bien l'un ou l'autre suivant la taille du message à envoyer.

---

<sup>a</sup>Voir la remarque.

## 7.6 – Envoi synchrone bloquant

Ce mode d'envoi (`MPI_SSEND()`) de messages permet d'éviter la copie temporaire des messages et, par conséquent, les surcoûts que cela peut engendrer.

Dans le programme modèle, il suffit de remplacer `MPI_SEND()` par `MPI_SSEND()` pour gagner un **facteur 2!**

```
22 temps_debut = MPI_WTIME()
23 if (rang == 0) then
24     ! Envoi d'un gros message
25     call MPI_SSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,code)
26     ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
27     call sgetrf(na, na, a, na, pivota, code)
28     ! Calcul modifiant le contenu du tableau C
29     c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
30 elseif (rang == 1) then
31     ! Calcul (factorisation LU avec LAPACK)
32     call sgetrf(na, na, a, na, pivota, code)
33     ! Réception d'un gros message
34     call MPI_RECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,statut,code)
35     ! Calcul dépendant de la réception du message précédent
36     a(:, :) = transpose(c(1:na,1:na)) + a(:, :)
37     ! Calcul indépendant de la réception du message précédent
38     call sgetrf(nb, nb, b, nb, pivotb, code)
39 end if
40 temps_fin = (MPI_WTIME() - temps_debut)
```

```
> mpiexec -n 2 AOptimiser
Temps : 0.36 secondes
```

calcul

communication

## 7.7 – Envoi synchrone non-bloquant

L'utilisation des sous-programmes `MPI_ISSEND()` et `MPI_IRecv()` conjointement aux sous-programmes de synchronisation précédents permet principalement de recouvrir les communications par des calculs.

Le programme modèle, une fois modifié, donne des gains en performances atteignant environ un **facteur 3** par rapport à la version initiale !



```
1 program AOptimiser
2   use mpi
3   implicit none
4
5   integer, parameter           :: na=256,nb=200
6   integer, parameter           :: m=2048,etiquette=1111
7   real, dimension(na,na)       :: a
8   real, dimension(nb,nb)      :: b
9   real, dimension(na)          :: pivota
10  real, dimension(nb)          :: pivotb
11  real, dimension(m,m)         :: c
12  integer                       :: rang,code,info,requete0, requete1
13  real(kind=8)                  :: temps_debut,temps_fin,temps_fin_max
14  integer, dimension(MPI_STATUS_SIZE) :: statut
15
16  call MPI_INIT(code)
17  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
18
19  ! Initialisation des tableaux
20  call random_number(a)
21  call random_number(b)
22  call random_number(c)
```

```
23 temps_debut = MPI_WTIME()
24 if (rang == 0) then
25     ! Requête d'envoi d'un gros message
26     call MPI_ISSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
27     ! Calcul (factorisation LU avec LAPACK) modifiant le contenu du tableau A
28     call sgetrf(na, na, a, na, pivota, code)
29     call MPI_WAIT(requete0,statut,code)
30     ! Calcul modifiant le contenu du tableau C
31     c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
32 elseif (rang == 1) then
33     ! Calcul (factorisation LU avec LAPACK)
34     call sgetrf(na, na, a, na, pivota, code)
35     ! Requête de réception d'un gros message
36     call MPI_IRECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
37     ! Calcul indépendant de la réception du message précédent, recouvrant celle-ci
38     call sgetrf(nb, nb, b, nb, pivotb, code)
39     call MPI_WAIT(requete1,statut,code)
40     ! Calcul dépendant de la réception du message précédent
41     a(:, :) = transpose(c(1:na,1:na)) + a(:, :)
42 end if
43 temps_fin = (MPI_WTIME() - temps_debut)
```

```
> mpiexec -n 2 AOptimiser
```

```
Temps : 0.23 secondes
```

calcul

communication

En général, dans le cas d'un envoi (`MPI_IxSEND()`) ou d'une réception (`MPI_Irecv()`) non bloquant, il existe toute une palette de sous-programmes qui permettent :

- ➡ de synchroniser un processus (ex. `MPI_WAIT()`) jusqu'à terminaison de la requête ;
- ➡ ou de vérifier (ex. `MPI_TEST()`) si une requête est bien terminée ;
- ➡ ou encore de contrôler avant réception (ex. `MPI_PROBE()` ou `MPI_Iprobe()`) si un message particulier est bien arrivé.

## 7.8 – Conseils

- ➡ Éviter la recopie temporaire des messages en utilisant le sous-programme `MPI_SSEND()` ou au contraire la forcer (selon les cas).
- ➡ Recouvrir les communications par des calculs tout en évitant la recopie temporaire des messages en utilisant les sous-programmes non bloquants `MPI_ISEND()` et `MPI_IRECV()`.

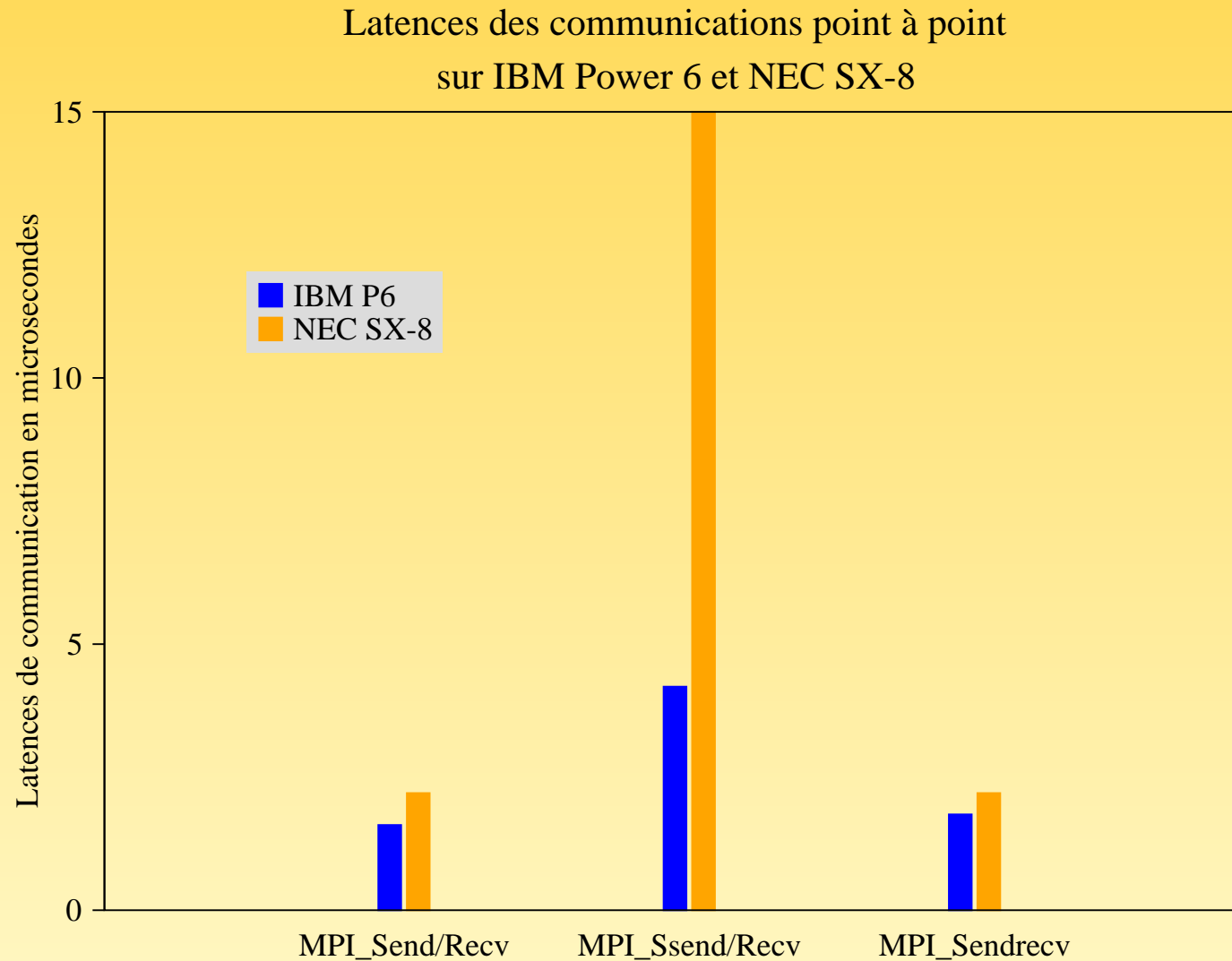
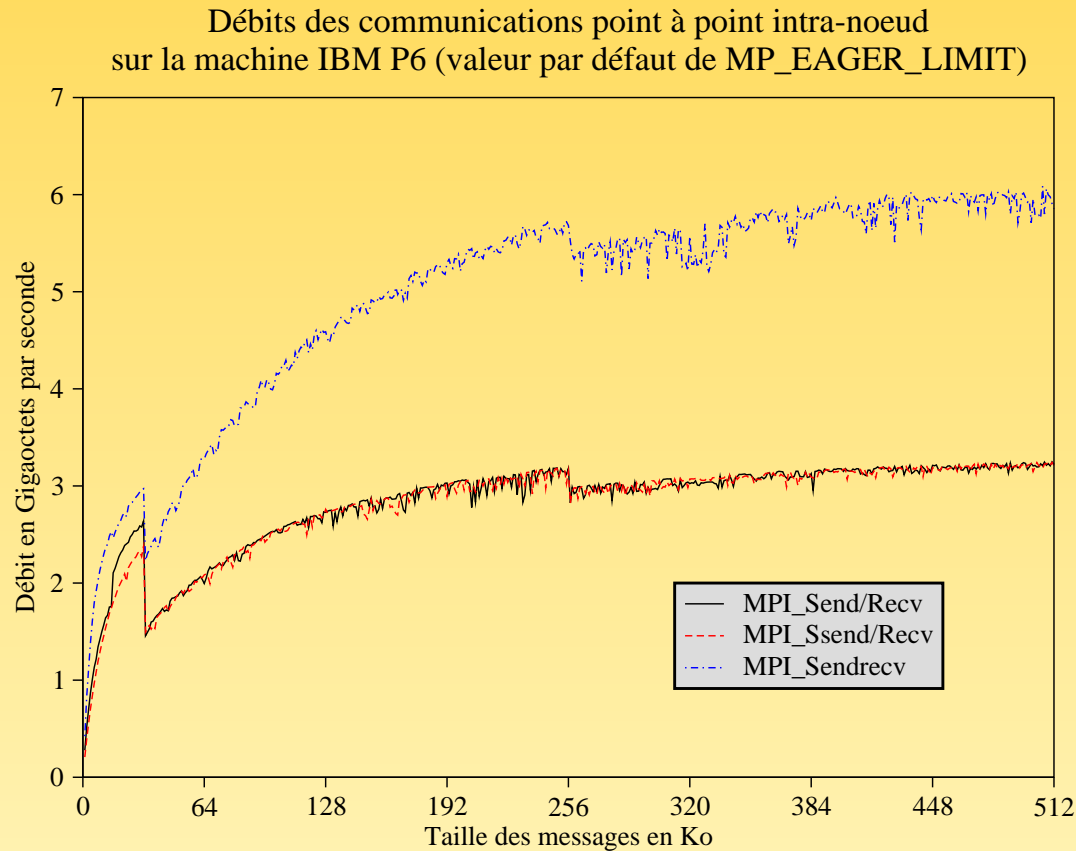
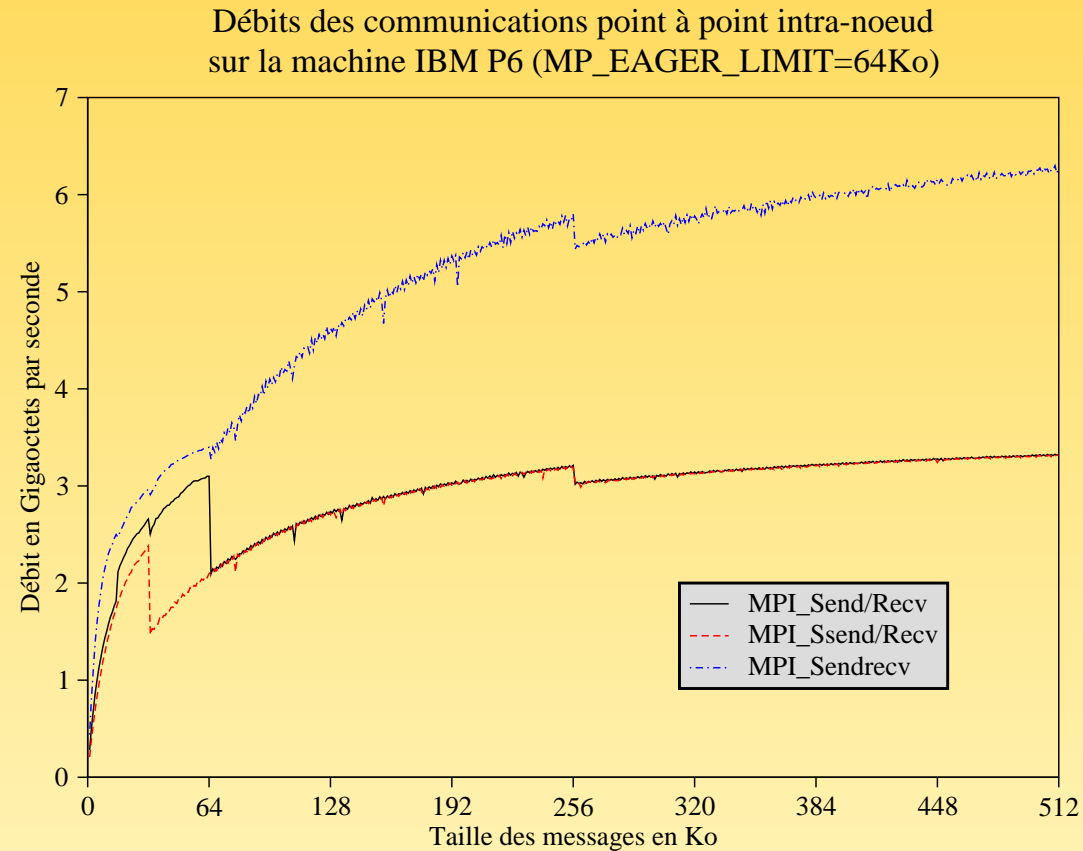


FIG. 36 – Latences des communications sur IBM Power 6 et NEC SX-8 (*valeurs données à titre indicatif, compte-tenu de la relative variabilité des mesures*)



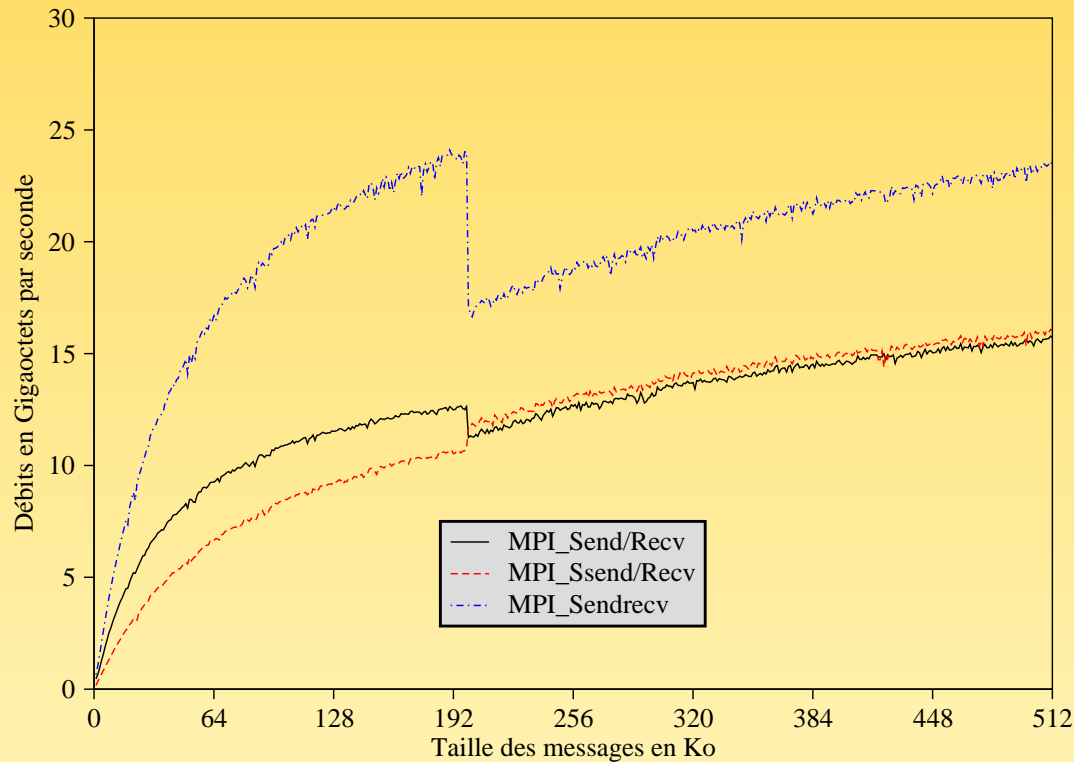
(a) Valeur par défaut de MP\_EAGER\_LIMIT



(b) MP\_EAGER\_LIMIT = 64 Ko

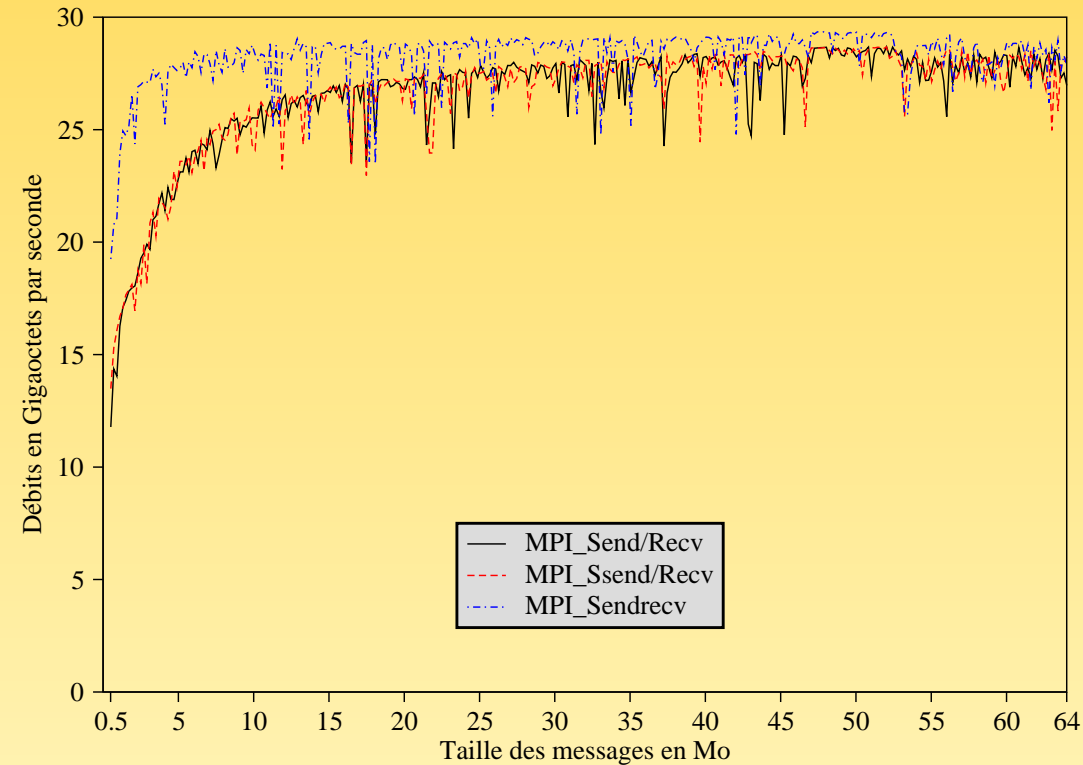
FIG. 37 – Débits intra-noeud sur IBM Power 6 (*valeurs données à titre indicatif, compte-tenu de la relative variabilité des mesures*)

Débits des communications point à point intra-noeud  
sur la machine NEC SX-8 (messages courts < 512 Ko)



(a) Messages courts

Débits des communications point à point intra-noeud  
sur la machine NEC SX-8 (512 Ko < messages longs < 64 Mo)



(b) Messages longs

FIG. 38 – Débits intra-nœud sur NEC SX-8 (*valeurs données à titre indicatif, compte-tenu de la relative variabilité des mesures*)

## 8 – Communicateurs

### 8.1 – Introduction

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

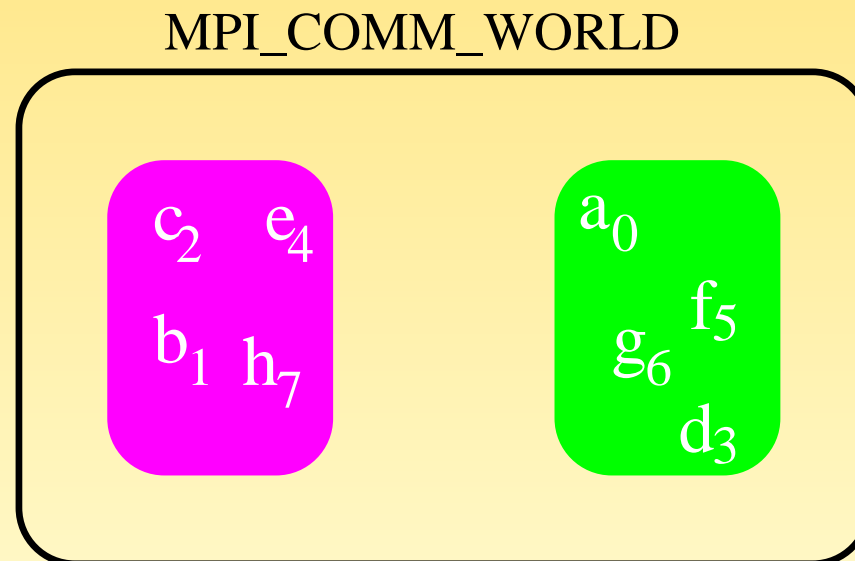


FIG. 39 – Partitionnement d'un communicateur



## 8.2 – Communicateur par défaut

C'est l'histoire de la poule et de l'œuf...

- ➡ On ne peut créer un communicateur qu'à partir d'un autre communicateur
- ➡ Fort heureusement, cela a été résolu en postulant que la poule existait déjà. En effet, un communicateur est fourni par défaut, dont l'identificateur `MPI_COMM_WORLD` est un entier défini dans les fichiers d'en-tête.
- ➡ Ce communicateur initial `MPI_COMM_WORLD` est créé pour toute la durée d'exécution du programme à l'appel du sous-programme `MPI_INIT()`
- ➡ Ce communicateur 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

## 8.3 – Exemple

Dans l'exemple qui suit, nous allons :

- ➡ regrouper d'une part les processus de rang pair et d'autre part les processus de rang impair ;
- ➡ ne diffuser un message collectif qu'aux processus de rang pair et un autre qu'aux processus de rang impair.

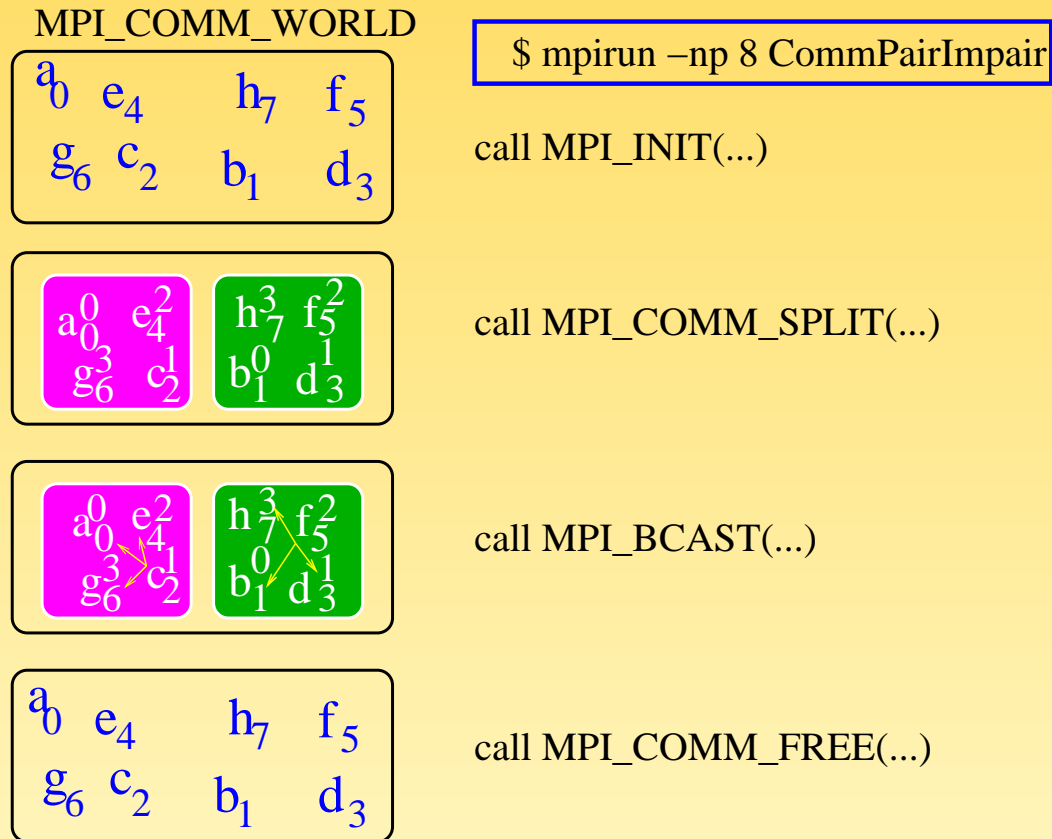


FIG. 40 – Création/destruction d'un communicateur

Que faire pour que le processus 2 diffuse ce message au sous-ensemble de processus de rang pair, par exemple ?

- ➡ 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 2 doit envoyer le message est pair ou impair.
- ➡ La solution est de **créer un communicateur regroupant ces processus** de sorte que le processus 2 diffuse le message à eux seuls

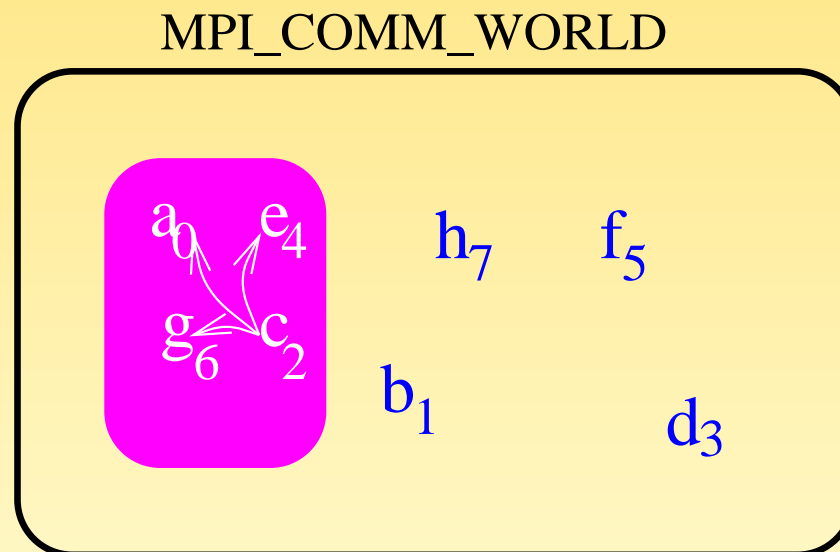


FIG. 41 – Un nouveau communicateur

## 8.4 – Groupes et communicateurs

- ☞ Un communicateur est constitué :
  - ① d'un **groupe**, qui est un ensemble ordonné de processus ;
  - ② 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 »)
  
- ☞ En pratique, pour construire un communicateur, il existe deux façons de procéder :
  - ① par l'intermédiaire d'un groupe de processus ;
  - ② directement à partir d'un autre communicateur.

- ➡ Dans la bibliothèque **MPI**, divers sous-programmes existent pour construire des communicateurs : **MPI\_CART\_CREATE()**, **MPI\_CART\_SUB()**, **MPI\_COMM\_CREATE()**, **MPI\_COMM\_DUP()**, **MPI\_COMM\_SPLIT()**
- ➡ Les **constructeurs de communicateurs** sont des **opérateurs collectifs** (qui engendrent des communications entre les processus)
- ➡ Les communicateurs 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()**

## 8.5 – Communicateur issu d'un autre

L'utilisation directe des groupes présente dans ce cas divers inconvénients, car elle impose de :

- ➡ nommer différemment les deux communicateurs (par exemple `comm_pair` et `comm_impair`);
- ➡ passer par les groupes pour construire ces deux communicateurs;
- ➡ laisser le soin à `MPI` d'ordonner le rang des processus dans ces deux communicateurs;
- ➡ faire des tests conditionnels lors de l'appel au sous-programme `MPI_BCAST()` :

```
if (comm_pair /= MPI_COMM_NULL) then
  ...
  ! Diffusion du message seulement aux processus de rangs pairs
  call MPI_BCAST(a,m,MPI_REAL,rang_ds_pair,comm_pair,code)
elseif (comm_impair /= MPI_COMM_NULL) then
  ...
  ! Diffusion du message seulement aux processus de rangs impairs
  call MPI_BCAST(a,m,MPI_REAL,rang_ds_impair,comm_impair,code)
end if
```

Le sous-programme `MPI_COMM_SPLIT()` permet de partitionner un communicateur donné en autant de communicateurs que l'on veut...

```
integer, intent(in)  :: comm, couleur, clef
integer, intent(out) :: nouveau_comm, code
call MPI_COMM_SPLIT(comm, couleur, clef, nouveau_comm, code)
```

processus	a	b	c	d	e	f	g	h
rang_monde	0	1	2	3	4	5	6	7
couleur	0	2	3	0	3	0	2	3
clef								
	2	15	0	0	1	3	11	1
rang_nv_com	1	1	0	0	1	2	0	2

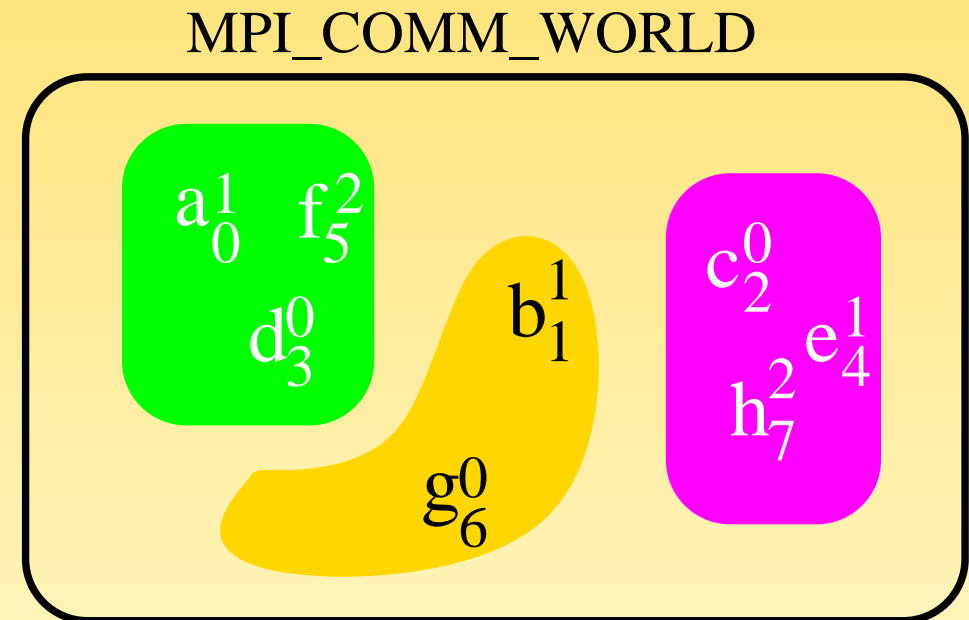


FIG. 42 – Construction de communicateurs avec `MPI_COMM_SPLIT()`

Un processus qui se voit attribuer une couleur égale à la valeur `MPI_UNDEFINED` n'appartiendra qu'à son communicateur initial.



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	1	1	0	3	5	0	7	7
rang_pairs_imp	1	1	0	2	2	0	3	3

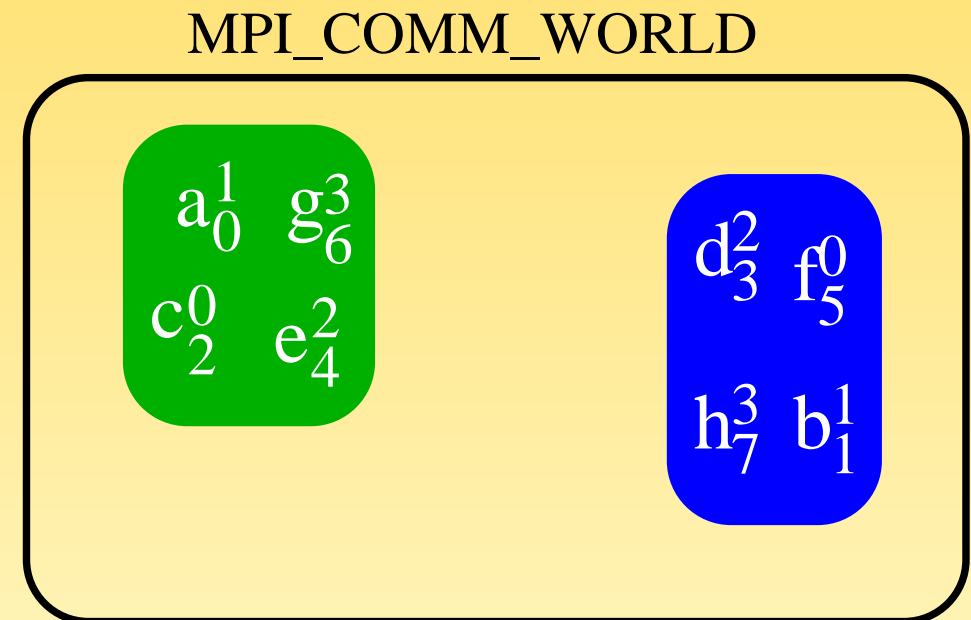


FIG. 43 – Construction du communicateur CommPairsImpairs avec `MPI_COMM_SPLIT()`

En pratique, ceci se met en place très simplement...

```
1 program PairsImpairs
2   use mpi
3   implicit none
4
5   integer, parameter :: m=16
6   integer           :: clef,CommPairsImpairs
7   integer           :: rang_dans_monde,code
8   real, dimension(m) :: a
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang_dans_monde,code)
12
13  ! Initialisation du vecteur A
14  a(:)=0.
15  if(rang_dans_monde == 2) a(:)=2.
16  if(rang_dans_monde == 5) a(:)=5.
```

```
17 clef = rang_dans_monde
18 if (rang_dans_monde == 2 .OR. rang_dans_monde == 5 ) then
19     clef=-1
20 end if
21
22 ! Création des communicateurs pair et impair en leur donnant une même dénomination
23 call MPI_COMM_SPLIT (MPI_COMM_WORLD, mod(rang_dans_monde,2), clef, CommPairsImpairs, code)
24
25 ! Diffusion du message par le processus 0 de chaque communicateur aux processus
26 ! de son groupe
27 call MPI_BCAST (a,m, MPI_REAL, 0, CommPairsImpairs, code)
28
29 ! Destruction des communicateurs
30 call MPI_COMM_FREE (CommPairsImpairs, code)
31 call MPI_FINALIZE (code)
32 end program PairsImpairs
```

## 8.6 – 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 ;
- ⇒ 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.

## 8.6.1 – Topologies cartésiennes

☞ Une topologie cartésienne est définie lorsqu'un ensemble de processus appartenant à un communicateur donné `comm_ancien` appellent le sous-programme

`MPI_CART_CREATE()`.

```
integer, intent(in)           :: comm_ancien, ndims
integer, dimension(ndims), intent(in) :: dims
logical, dimension(ndims), intent(in) :: periods
logical, intent(in)           :: reorganisation

integer, intent(out)          :: comm_nouveau, code

call MPI_CART_CREATE(comm_ancien, ndims, dims, periods, reorganisation, comm_nouveau, code)
```

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

```
use mpi
integer                :: comm_2D, code
integer, parameter     :: ndims = 2
integer, dimension(ndims) :: dims
logical, dimension(ndims) :: periods
logical                :: reorganisation

.....

dims(1) = 4
dims(2) = 2
periods(1) = .false.
periods(2) = .true.
reorganisation = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, dims, periods, reorganisation, comm_2D, code)
```

➡ 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.

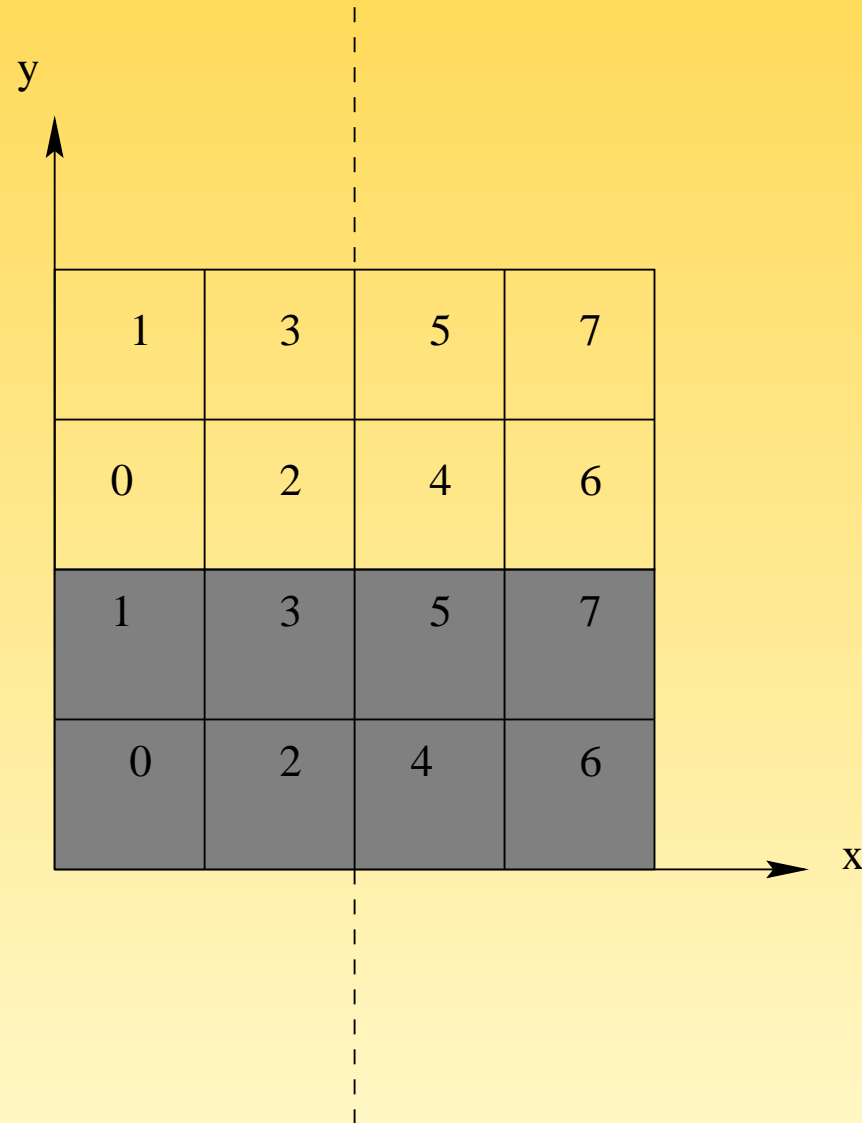


FIG. 44 – Topologie cartésienne 2D périodique en y



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

```
use mpi
integer                :: comm_3D,code
integer, parameter    :: ndims = 3
integer, dimension(ndims) :: dims
logical, dimension(ndims) :: periods
logical                :: reorganisation

.....

dims(1) = 4
dims(2) = 2
dims(3) = 2
periods(:) = .false.
reorganisation = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_3D,code)
```

2	6	10	14
0	4	8	12

 $z = 0$ 

3	7	11	15
1	5	9	13

 $z = 1$

➡ Dans une topologie cartésienne, le sous-programme `MPI_DIMS_CREATE()` retourne le nombre de processus dans chaque dimension de la grille en fonction du nombre total de processus.

```
integer, intent(in)                :: nb_procs, ndims
integer, dimension(ndims), intent(inout) :: dims
integer, intent(out)               :: code

call MPI_DIMS_CREATE(nb_procs, ndims, dims, code)
```

➡ 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	call MPI_DIMS_CREATE	dims en sortie
(0,0)	(8,2,dims,code)	(4,2)
(0,0,0)	(16,3,dims,code)	(4,2,2)
(0,4,0)	(16,3,dims,code)	(2,4,2)
(0,3,0)	(16,3,dims,code)	error

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

```
integer, intent(in)           :: comm_nouveau
integer, dimension(ndims),intent(in) :: coords

integer, intent(out)         :: rang, code

call MPI_CART_RANK(comm_nouveau, coords, rang, code)
```

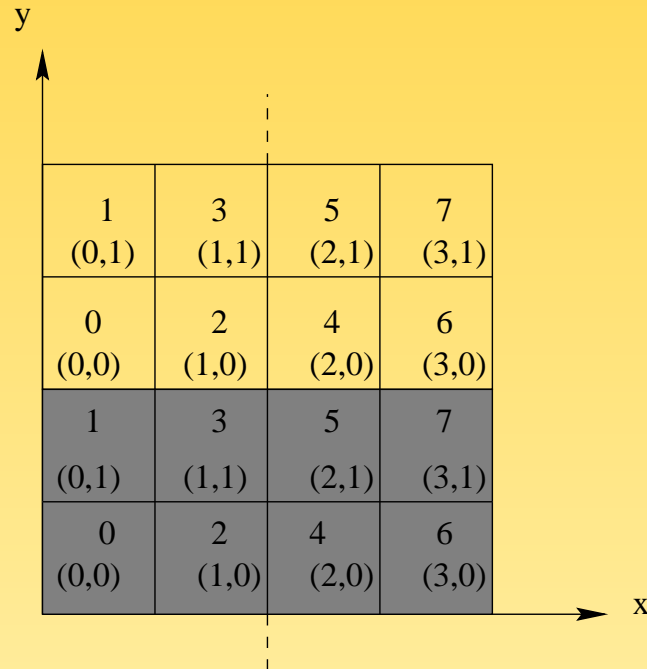


FIG. 45 – Topologie cartésienne 2D périodique en y

```

coords(1)=dims(1)-1
do i=0,dims(2)-1
  coords(2) = i
  call MPI_CART_RANK(comm_2D,coords,rang(i),code)
end do

.....
i=0,en entrée coords=(3,0),en sortie rang(0)=6.
i=1,en entrée coords=(3,1),en sortie rang(1)=7.

```

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

```
integer, intent(in)           :: comm_nouveau, rang, ndims  
  
integer, dimension(ndims), intent(out) :: coords  
integer, intent(out)         :: code  
  
call MPI_CART_COORDS(comm_nouveau, rang, ndims, coords, code)
```

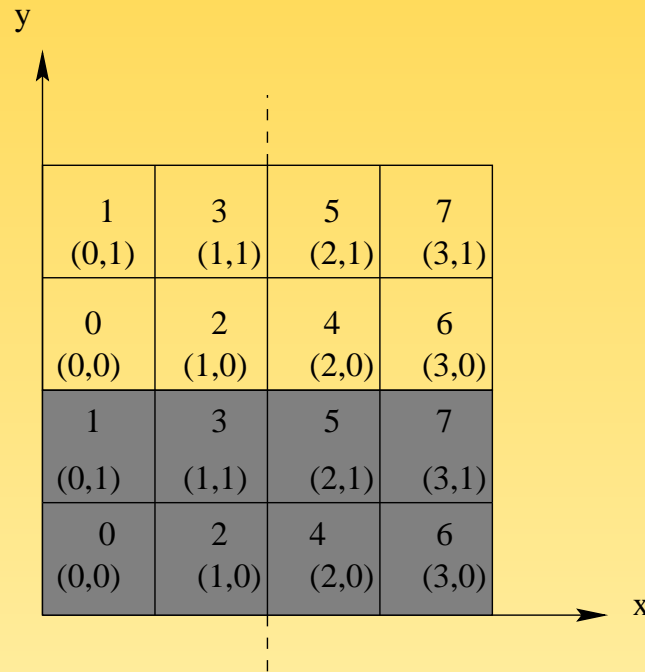


FIG. 46 – Topologie cartésienne 2D périodique en y

```

if (mod(rang,2) == 0) then
  call MPI_CART_COORDS(comm_2D,rang,2,coords,code)
end if

```

.....

En entrée, les valeurs de rang sont : 0,2,4,6.

En sortie, les valeurs de coords sont :

(0,0),(1,0),(2,0),(3,0).

- ➡ 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.

```
integer, intent(in)  :: comm_nouveau, direction, pas  
  
integer, intent(out) :: rang_precedent, rang_suivant  
integer, intent(out) :: code  
  
call MPI_CART_SHIFT(comm_nouveau, direction, pas, rang_precedent, rang_suivant, code)
```

- ➡ Le paramètre **direction** correspond à l'axe du déplacement (xyz).
- ➡ Le paramètre **pas** correspond au pas du déplacement.



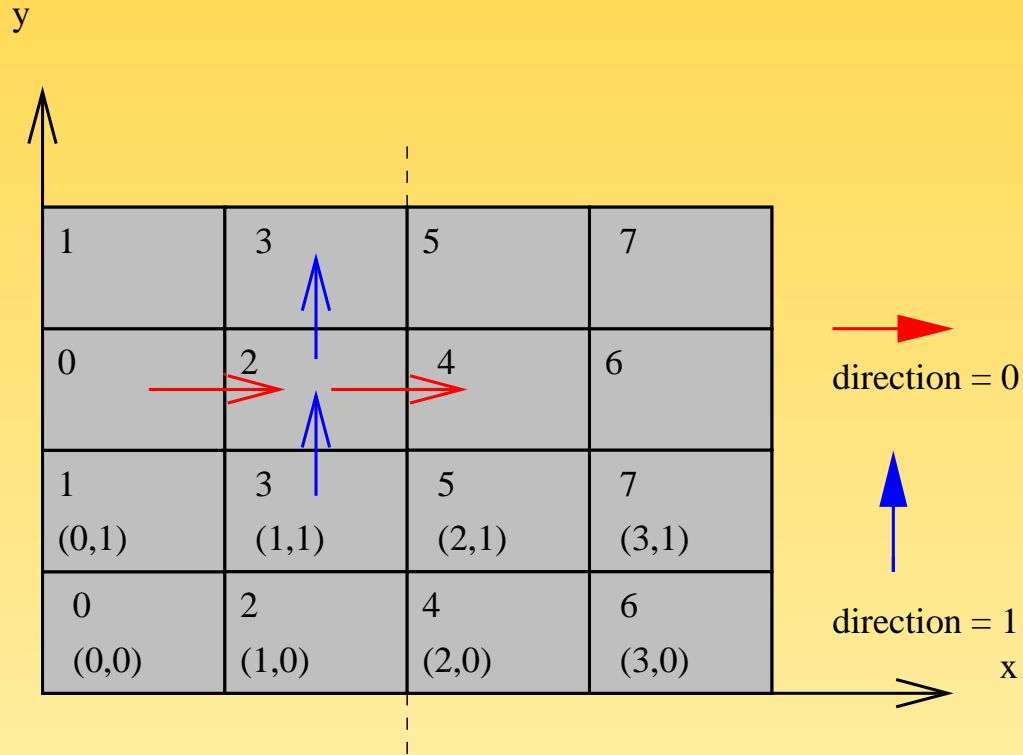


FIG. 47 – Appel du sous-programme MPI\_CART\_SHIFT()

```
call MPI_CART_SHIFT(comm_2D, 0, 1, rang_gauche, rang_droit, code)
```

```
.....
Pour le processus 2, rang_gauche=0, rang_droit=4
```

```
call MPI_CART_SHIFT(comm_2D, 1, 1, rang_bas, rang_haut, code)
```

```
.....
Pour le processus 2, rang_bas=3, rang_haut=3
```

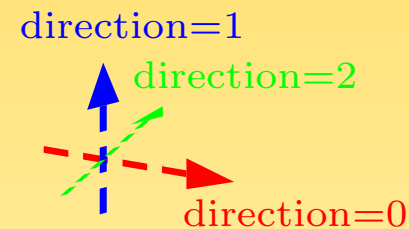


FIG. 48 – Appel du sous-programme MPI\_CART\_SHIFT()

```
call MPI_CART_SHIFT(comm_3D,0,1,rang_gauche,rang_droit,code)
```

```
.....
Pour le processus 0, rang_gauche=-1, rang_droit=4
```

```
call MPI_CART_SHIFT(comm_3D,1,1,rang_bas,rang_haut,code)
```

```
.....
Pour le processus 0, rang_bas=-1, rang_haut=2
```

```
call MPI_CART_SHIFT(comm_3D,2,1,rang_avant,rang_arriere,code)
```

```
.....
Pour le processus 0, rang_avant=-1, rang_arriere=1
```

☞ Exemple de programme :

```
1 program decomposition
2   use mpi
3   implicit none
4
5   integer           :: rang_ds_topo,nb_procs
6   integer           :: code,comm_2D
7   integer, dimension(4) :: voisin
8   integer, parameter :: N=1,E=2,S=3,W=4
9   integer, parameter :: ndims = 2
10  integer, dimension (ndims) :: dims,coords
11  logical, dimension (ndims) :: periods
12  logical             :: reorganisation
13
14  call MPI_INIT(code)
15
16  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
17
18  ! Connaître le nombre de processus suivant x et y
19  dims(:) = 0
20
21  call MPI_DIMS_CREATE(nb_procs,ndims,dims,code)
```

```
22 ! Création grille 2D periodique en y
23 periods(1) = .false.
24 periods(2) = .true.
25 reorganisation = .false.
26
27 call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, dims, periods, reorganisation, comm_2D, code)
28
29 ! Connaître mes coordonnées dans la topologie
30 call MPI_COMM_RANK(comm_2D, rang_ds_topo, code)
31 call MPI_CART_COORDS(comm_2D, rang_ds_topo, ndims, coords, code)
32
33 ! Initialisation du tableau voisin à la valeur MPI_PROC_NULL
34 voisin(:) = MPI_PROC_NULL
35
36 ! Recherche de mes voisins Ouest et Est
37 call MPI_CART_SHIFT(comm_2D, 0, 1, voisin(W), voisin(E), code)
38
39 ! Recherche de mes voisins Sud et Nord
40 call MPI_CART_SHIFT(comm_2D, 1, 1, voisin(S), voisin(N), code)
41
42 call MPI_FINALIZE(code)
43
44 end program decomposition
```

## 8.6.2 – Subdiviser une topologie cartésienne

- ☞ La question est de savoir comment dégénérer une topologie cartésienne 2D ou 3D de processus 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.

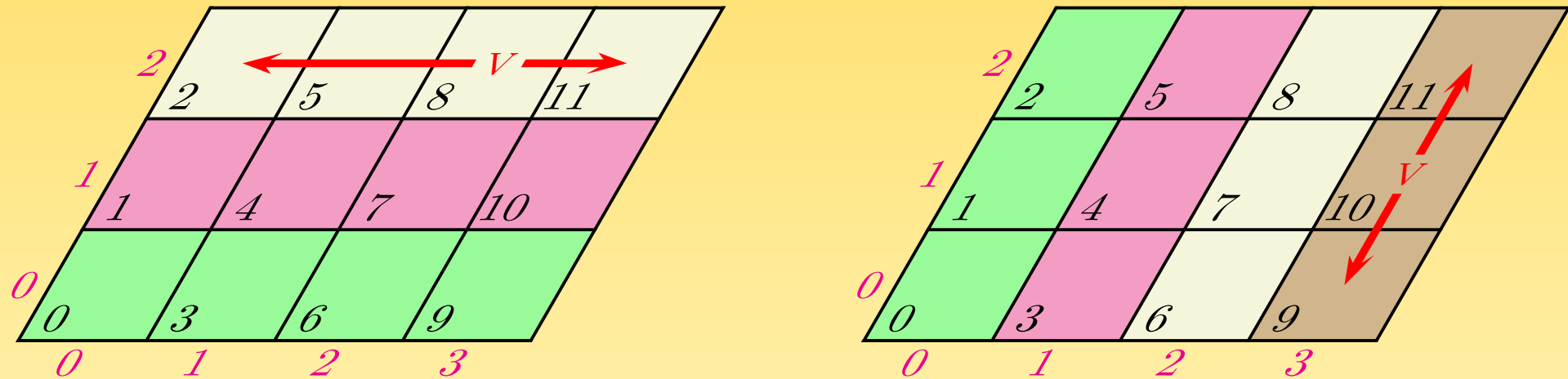


FIG. 49 – Deux exemples de distribution de données dans une topologie 2D dégénérée

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.

```

logical,intent(in),dimension(NDim) :: Subdivision
integer,intent(in)                :: CommCart
integer,intent(out)               :: CommCartD, code
call MPI_CART_SUB(CommCart,Subdivision,CommCartD,code)

```

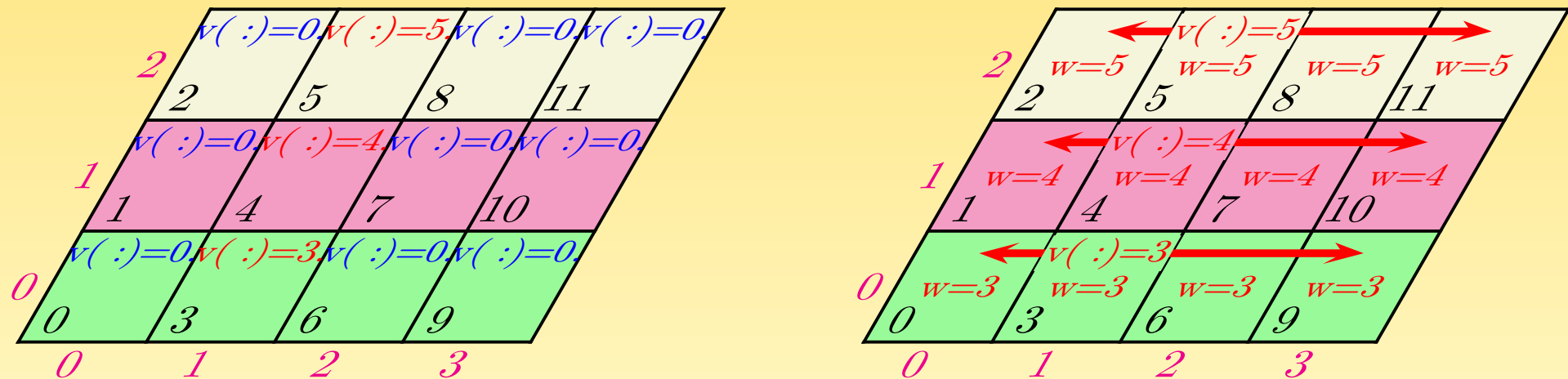


FIG. 50 – Représentation initiale d'un tableau  $V$  dans la grille 2D et représentation finale après la distribution de celui-ci sur la grille 2D dégénérée

```
1 program CommCartSub
2   use mpi
3   implicit none
4
5   integer           :: Comm2D,Comm1D,rang,code
6   integer,parameter :: NDim2D=2
7   integer,dimension(NDim2D) :: Dim2D,Coord2D
8   logical,dimension(NDim2D) :: Periode,Subdivision
9   logical           :: Reordonne
10  integer,parameter  :: m=4
11  real, dimension(m)  :: V(:)=0.
12  real               :: W=0.
```



```
13 call MPI_INIT(code)
14
15 ! Création de la grille 2D initiale
16 Dim2D(1) = 4
17 Dim2D(2) = 3
18 Periode(:) = .false.
19 ReOrdonne = .false.
20 call MPI_CART_CREATE(MPI_COMM_WORLD,NDim2D,Dim2D,Periode,ReOrdonne,Comm2D,code)
21 call MPI_COMM_RANK(Comm2D,rang,code)
22 call MPI_CART_COORDS(Comm2D,rang,NDim2D,Coord2D,code)
23
24 ! Initialisation du vecteur V
25 if (Coord2D(1) == 1) V(:)=real(rang)
26
27 ! Chaque ligne de la grille doit être une topologie cartésienne 1D
28 Subdivision(1) = .true.
29 Subdivision(2) = .false.
30 ! Subdivision de la grille cartésienne 2D
31 call MPI_CART_SUB(Comm2D,Subdivision,Comm1D,code)
32
33 ! Les processus de la colonne 2 distribuent le vecteur V aux processus de leur ligne
34 call MPI_SCATTER(V,1,MPI_REAL,W,1,MPI_REAL,1,Comm1D,code)
35
36 print '("Rang : ",I2," ; Coordonnees : (" ,I1," ,",I1,") ; W = ",F2.0)', &
37     rang,Coord2D(1),Coord2D(2),W
38
39 call MPI_FINALIZE(code)
40 end program CommCartSub
```

```
> 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.
```

## 9 – MPI-IO

### 9.1 – Introduction

#### 9.1.1 – Présentation

- ➡ 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.).
- ☞ Les buts de **MPI-IO**, via l'interface de haut niveau qu'il propose, sont d'offrir **simplicité**, **expressivité** et **souplesse**, tout en autorisant des implémentations **performantes** prenant en compte les spécificités matérielles et logicielles des dispositifs d'entrées-sorties des machines cibles.
- ☞ **MPI-IO** offre une interface calquée sur celle utilisée pour l'échange de messages. La définition des données accédées suivant les processus se fait par l'utilisation de **types de données** (de base ou bien dérivés). Quant aux notions d'**opérations collectives** et de **non-bloquantes**, elles sont gérées de façon similaire à ce que propose **MPI** pour les messages.
- ☞ **MPI-IO** autorise des accès aussi bien **séquentiels** qu'**aléatoires**.

## 9.1.2 – Enjeux

- ☞ C'est une interface de haut niveau, où, comme on l'a dit, certaines techniques d'optimisation sont accessibles aux utilisateurs, mais où beaucoup d'optimisations essentielles peuvent être implémentées de façon transparente. Deux exemples importants en sont :
  - ⇒ le cas d'accès nombreux, par un seul processus, à de petits blocs discontinus : ceci peut être traité par un mécanisme de *passoire* (*data sieving*), après regroupement d'un ensemble de requêtes, lecture d'un grand bloc contigu du disque vers une zone mémoire tampon, puis affectation aux zones mémoire utilisateur des sous-ensembles adéquats de données ;

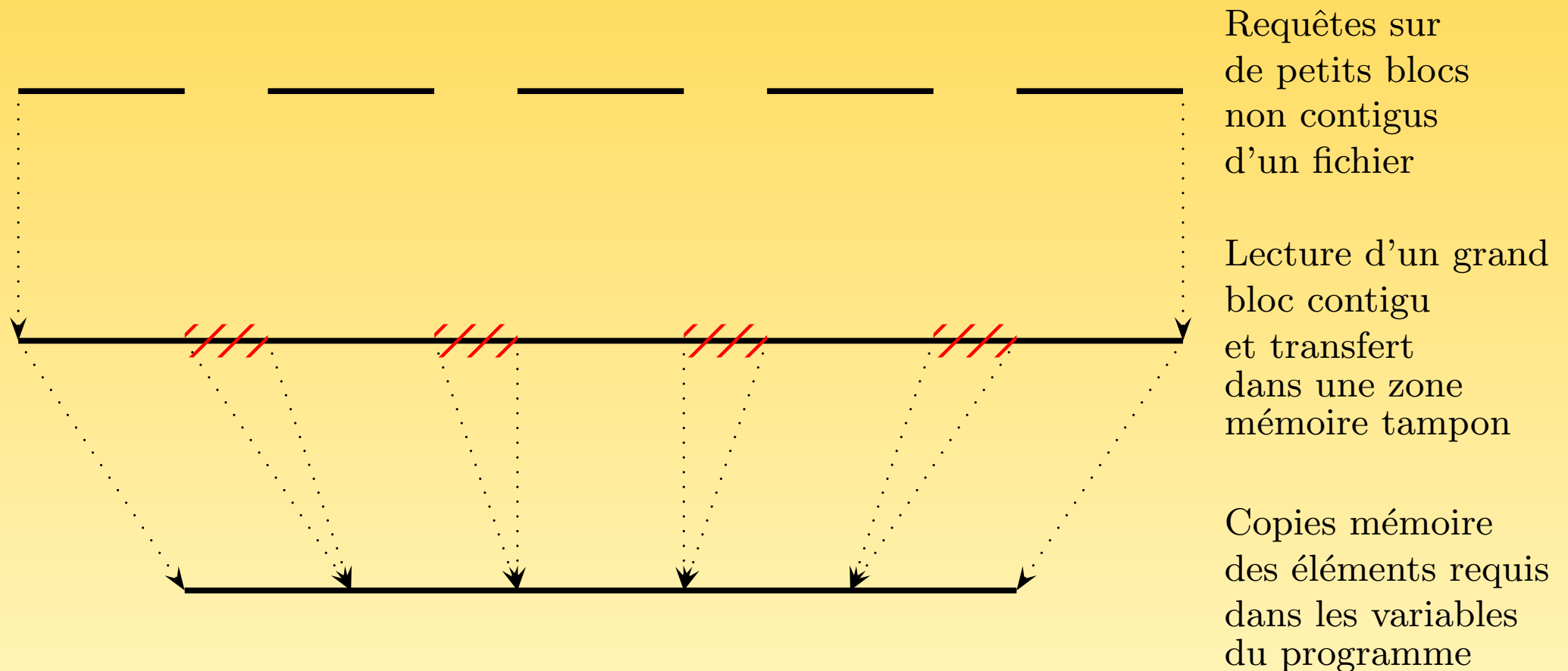


FIG. 51 – Mécanisme de *passoire* dans le cas d'accès nombreux, par un seul processus, à de petits blocs discontinus

⇒ le cas d'accès, par un ensemble de processus, à des blocs discontinus (cas des tableaux distribués, par exemple) : ceci peut être traité par des entrées-sorties collectives en décomposant les opérations en deux phases.

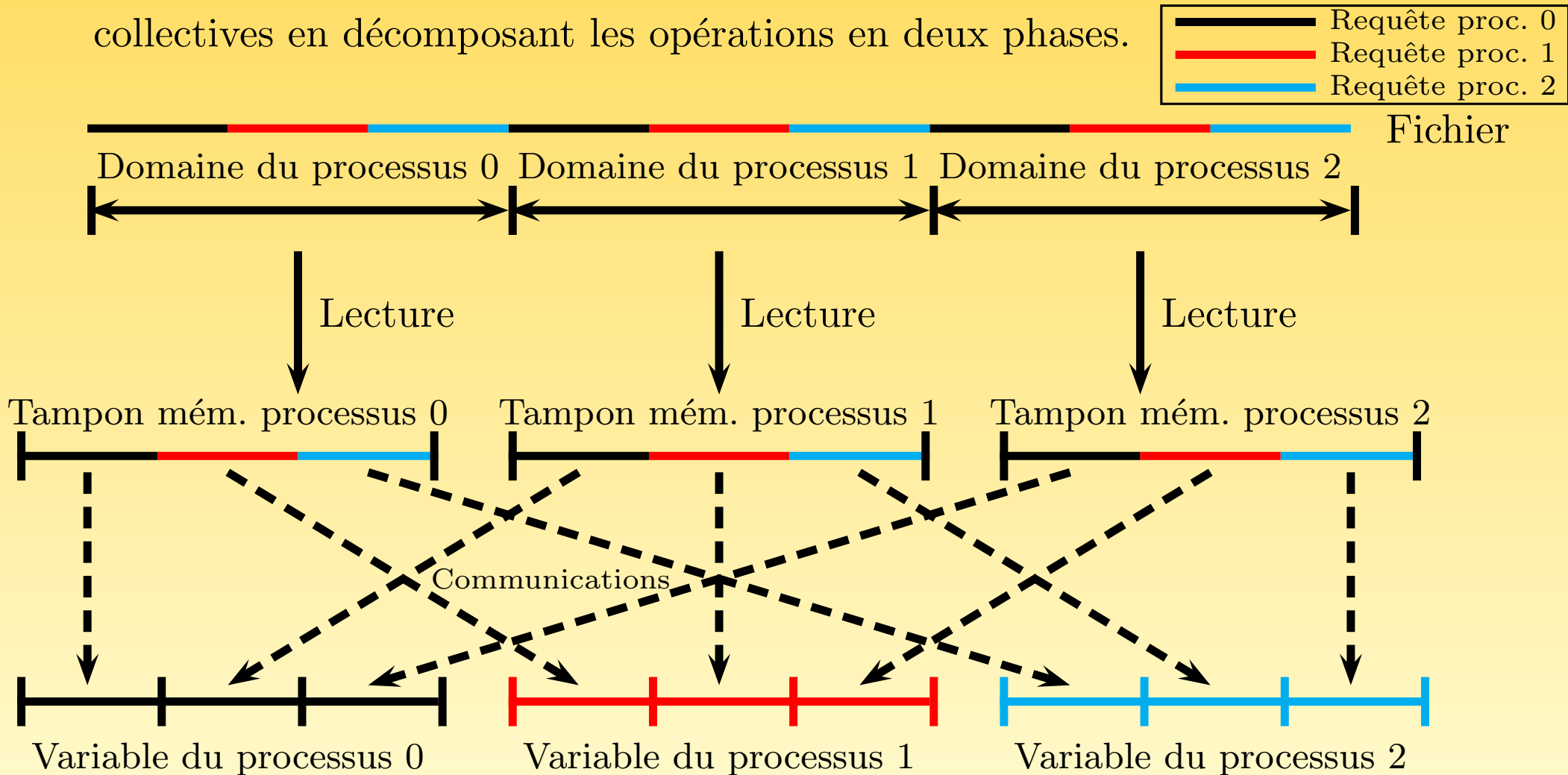


FIG. 52 – Lecture en deux phases, par un ensemble de processus

## 9.1.3 – Définitions

**fichier** (*file*) : un fichier MPI est un ensemble ordonné de données typées.

Un fichier est ouvert collectivement par tous les processus d'un communicateur.

Toutes les opérations d'entrées-sorties collectives ultérieures se feront dans ce cadre.

**déplacement initial** (*displacement*) : c'est une adresse absolue, exprimée en octets, par rapport au début du fichier et à partir de laquelle une **vue** commence.

**type élémentaire de données** — **type\_élém** (*etype*) : c'est l'unité de donnée utilisée pour calculer les positionnements et pour accéder aux données. Ce peut être n'importe quel type de donnée MPI, prédéfini ou bien créé en tant que type dérivé.

**motif** (*filetype*) : c'est un masque qui constitue la base du partitionnement d'un fichier entre processus (si le fichier est vu comme un *pavage* à une dimension, le motif est la *tuile élémentaire* qui sert au pavage). C'est ou bien un type élémentaire de données, ou bien un type dérivé MPI construit comme une répétition d'occurrences d'un tel type élémentaire de données (les *trous* — parties de fichiers non accédées — doivent également être un multiple du *type\_élém* utilisé).



**vue** (*view*) : une vue définit l'ensemble des données *visibles* (et donc accessibles) d'un fichier, une fois que celui-ci a été ouvert. C'est un ensemble ordonné de types élémentaires de données.

**position** (*offset*) : c'est la position dans le fichier, exprimée en nombre de *type\_élément*, relativement à la vue courante (les *trous* définis dans la vue ne sont pas comptabilisés pour calculer les positions).

**descripteur** (*file handle*) : le descripteur est un objet caché créé à l'ouverture et détruit à la fermeture d'un fichier. Toutes les opérations sur un fichier se font en spécifiant comme référence son descripteur.

**pointeur** (*file pointer*) : ils sont tenus à jour automatiquement par MPI et déterminent des positions à l'intérieur du fichier. Il y en a de deux sortes : les pointeurs **individuels** qui sont propres à chaque processus ayant ouvert le fichier ; les pointeurs **partagés** qui sont communs à tous les processus ayant ouvert le fichier.

**taille du fichier** (*file size*) : la taille d'un fichier MPI est mesurée en nombre d'*octets*.

## 9.2 – Gestion de fichiers

- ➡ Les tâches courantes de gestion de fichiers sont des **opérations collectives** faites par tous les processus du communicateur indiqué.
- ➡ Nous ne décrivons ici que les principaux sous-programmes (ouverture, fermeture, obtention des caractéristiques) mais d'autres sont disponibles (suppression, pré-allocation, etc.).
- ➡ Les attributs (décrivant les droits d'accès, le mode d'ouverture, la destruction éventuelle à la fermeture, etc.) doivent être précisés par opérations sur des constantes prédéfinies.
- ➡ Tous les processus du communicateur au sein duquel un fichier est ouvert participeront aux opérations collectives ultérieures d'accès aux données.
- ➡ L'ouverture d'un fichier renvoie un **descripteur**, qui sera ensuite utilisé dans toutes les opérations portant sur ce fichier.
- ➡ Les informations disponibles via le sous-programme **MPI\_FILE\_SET\_INFO()** varient d'une implémentation à l'autre.

TAB. 4 – Attributs pouvant être positionnés lors de l’ouverture des fichiers

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	erreur si le fichier est déjà ouvert par une autre application
MPI_MODE_SEQUENTIAL	accès séquentiel
MPI_MODE_APPEND	pointeurs en fin de fichier (mode ajout)
MPI_MODE_DELETE_ON_CLOSE	destruction après la fermeture

```
1 program open01
2
3 use mpi
4 implicit none
5
6 integer :: descripteur,code
7
8 call MPI_INIT(code)
9
10 call MPI_FILE_OPEN(MPI_COMM_WORLD,"fichier.txt", &
11                   MPI_MODE_RDWR + MPI_MODE_CREATE,MPI_INFO_NULL,descripteur,code)
12
13 call MPI_FILE_CLOSE(descripteur,code)
14 call MPI_FINALIZE(code)
15
16 end program open01
```

```
> ls -l fichier.txt
```

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

## 9.3 – Lectures/écritures : 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 par exemple le nombre voulu d'octets depuis le 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 y a de nombreuses variantes disponibles : nous en décrirons un certain nombre, sans pouvoir être exhaustif.

TAB. 5 – Résumé des types d'accès possibles

Positionnement	Synchronisation	Regroupement	
		<i>individuel</i>	<i>collectif</i>
adresses explicites	bloquantes	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
	non bloquantes	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN
		MPI_FILE_IWRITE_AT	MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END
<i>suite page suivante</i>			

Positionnement	Synchronisation	Regroupement	
		<i>individuel</i>	<i>collectif</i>
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_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
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

- ☞ Il est possible de mélanger les types d'accès effectués à un même fichier au sein d'une application.
- ☞ Les zones mémoire accédées sont décrites par trois quantités :
  - ⇒ l'**adresse initiale** de la zone concernée ;
  - ⇒ le **nombre d'éléments** pris en compte ;
  - ⇒ le **type de données**, qui doit correspondre à une suite de copies contiguës du type élémentaire de donnée (*type\_élem*) de la « vue » courante.



## 9.4 – Lectures/écritures individuelles

### 9.4.1 – Via des déplacements explicites

- ➡ La **position** est exprimée en nombre d'occurrences d'un type de données, lequel doit être un multiple du type élémentaire de donnée de la « vue » courante.
- ➡ Le fichier ne doit pas avoir été ouvert avec l'attribut `MPI_MODE_SEQUENTIAL`.

```
1 program write_at
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=10
6   integer                    :: i,rang,descripteur,code,nb_octets_entier
7   integer(kind=MPI_OFFSET_KIND) :: position_fichier
8   integer, dimension(nb_valeurs) :: valeurs
9   integer, dimension(MPI_STATUS_SIZE) :: statut
10
11  call MPI_INIT(code)
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14  valeurs(:)= ((i+rang*100,i=1,nb_valeurs)/)
15  print *, "Écriture processus",rang, ":",valeurs(:)
16
17  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_WRONLY + MPI_MODE_CREATE, &
18                   MPI_INFO_NULL,descripteur,code)
19
20  call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
21
22  position_fichier=rang*nb_valeurs*nb_octets_entier
23  call MPI_FILE_WRITE_AT(descripteur,position_fichier,valeurs,nb_valeurs,MPI_INTEGER, &
24                       statut,code)
25
26  call MPI_FILE_CLOSE(descripteur,code)
27  call MPI_FINALIZE(code)
28 end program write_at
```

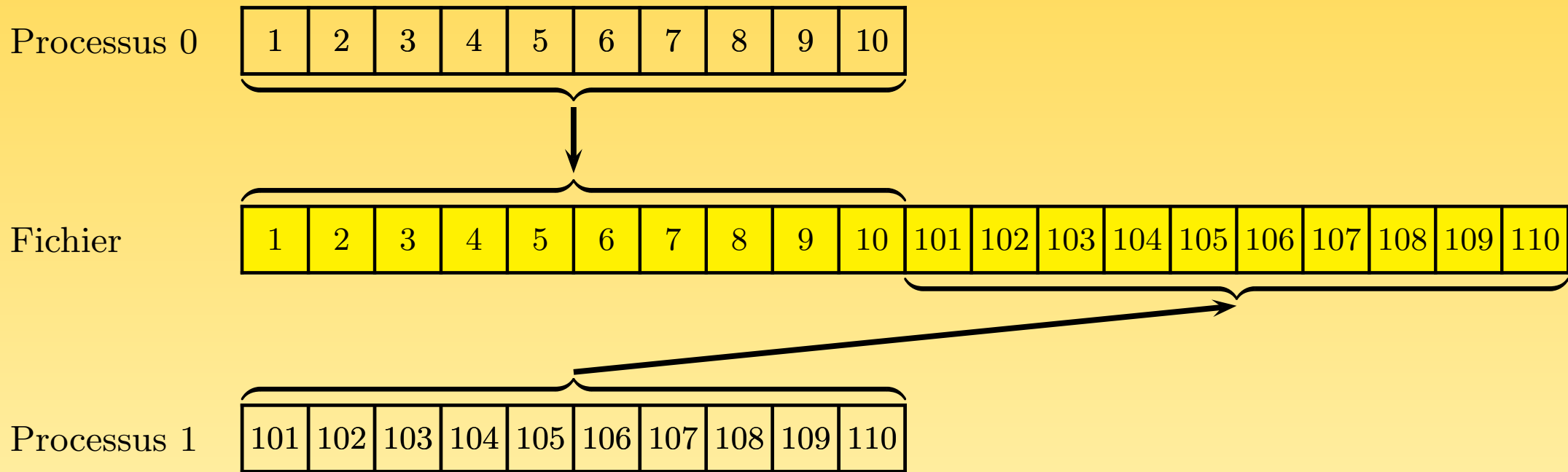
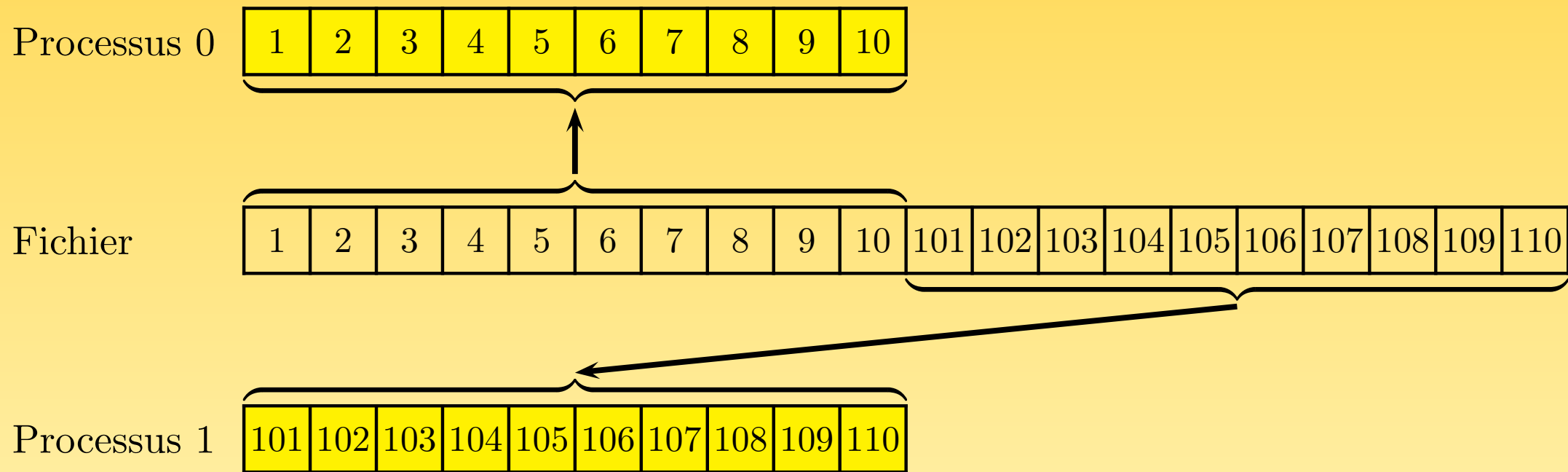


FIG. 53 – Exemple d'utilisation de MPI\_FILE\_WRITE\_AT()

```
> mpiexec -n 2 write_at
```

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

```
1 program read_at
2
3 use mpi
4 implicit none
5
6 integer, parameter          :: nb_valeurs=10
7 integer                    :: rang,descripteur,code,nb_octets_entier
8 integer(kind=MPI_OFFSET_KIND) :: position_fichier
9 integer, dimension(nb_valeurs) :: valeurs
10 integer, dimension(MPI_STATUS_SIZE) :: statut
11
12 call MPI_INIT(code)
13 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                   descripteur,code)
17
18 call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
19
20 position_fichier=rang*nb_valeurs*nb_octets_entier
21 call MPI_FILE_READ_AT(descripteur,position_fichier,valeurs,nb_valeurs,MPI_INTEGER, &
22                      statut,code)
23 print *, "Lecture processus",rang,":",valeurs(:)
24
25 call MPI_FILE_CLOSE(descripteur,code)
26 call MPI_FINALIZE(code)
27
28 end program read_at
```

FIG. 54 – Exemple d'utilisation de `MPI_FILE_READ_AT()`

```
> mpiexec -n 2 read_at
```

```
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
```

## 9.4.2 – Via des déplacements implicites individuels

- ➡ Dans ces cas-ci, un pointeur individuel est géré par le système, et ceci **par fichier** et **par processus**.
- ➡ Pour un processus donné, deux accès successifs au même fichier permettent donc d'accéder automatiquement aux éléments consécutifs de celui-ci.
- ➡ Dans tous ces sous-programmes, les pointeurs partagés ne sont jamais accédés ou modifiés.
- ➡ Après chaque accès, le pointeur est positionné sur le type élémentaire de donnée suivant.
- ➡ Le fichier ne doit pas avoir été ouvert avec l'attribut **MPI\_MODE\_SEQUENTIAL**.

```
1 program read01
2
3 use mpi
4 implicit none
5
6 integer, parameter          :: nb_valeurs=10
7 integer                    :: rang,descripteur,code
8 integer, dimension(nb_valeurs) :: valeurs
9 integer, dimension(MPI_STATUS_SIZE) :: statut
10
11 call MPI_INIT(code)
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                   descripteur,code)
16
17 call MPI_FILE_READ(descripteur,valeurs,6,MPI_INTEGER,statut,code)
18 call MPI_FILE_READ(descripteur,valeurs(7),4,MPI_INTEGER,statut,code)
19
20 print *, "Lecture processus",rang,":",valeurs(:)
21
22 call MPI_FILE_CLOSE(descripteur,code)
23 call MPI_FINALIZE(code)
24
25 end program read01
```

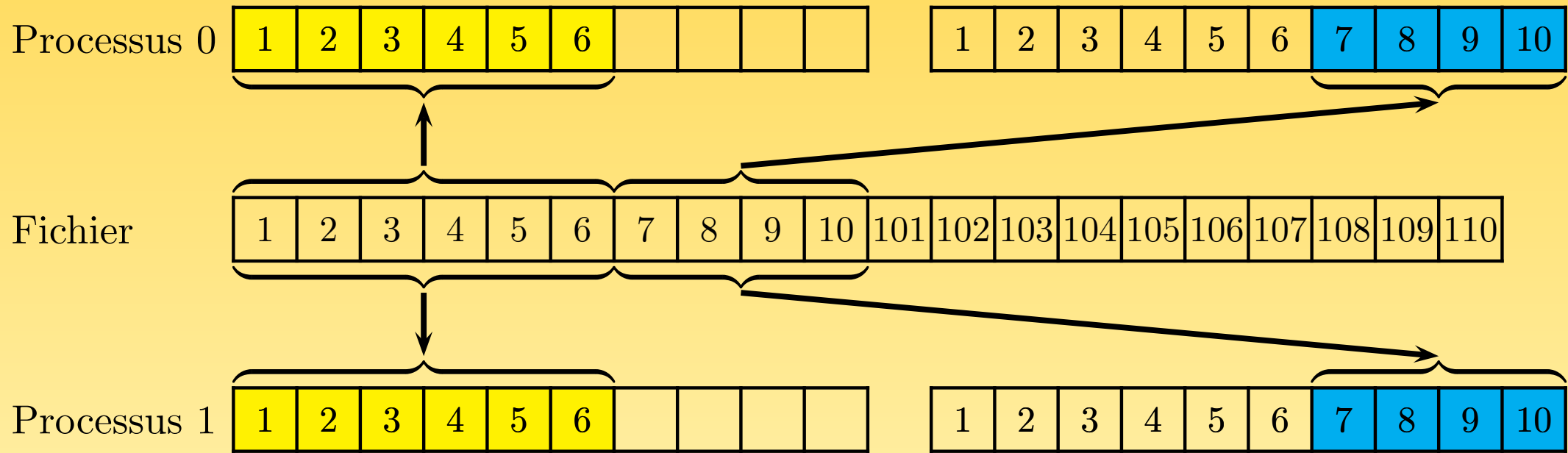


FIG. 55 – Exemple 1 d'utilisation de MPI\_FILE\_READ()

```
> mpiexec -n 2 read01
```

```
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
```



```
1 program read02
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=10
6   integer                    :: rang,descripteur,code
7   integer, dimension(nb_valeurs) :: valeurs=0
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                    descripteur,code)
15
16  if (rang == 0) then
17    call MPI_FILE_READ(descripteur,valeurs,5,MPI_INTEGER,statut,code)
18  else
19    call MPI_FILE_READ(descripteur,valeurs,8,MPI_INTEGER,statut,code)
20    call MPI_FILE_READ(descripteur,valeurs,5,MPI_INTEGER,statut,code)
21  end if
22
23  print *, "Lecture processus",rang,":",valeurs(1:8)
24
25  call MPI_FILE_CLOSE(descripteur,code)
26  call MPI_FINALIZE(code)
27 end program read02
```

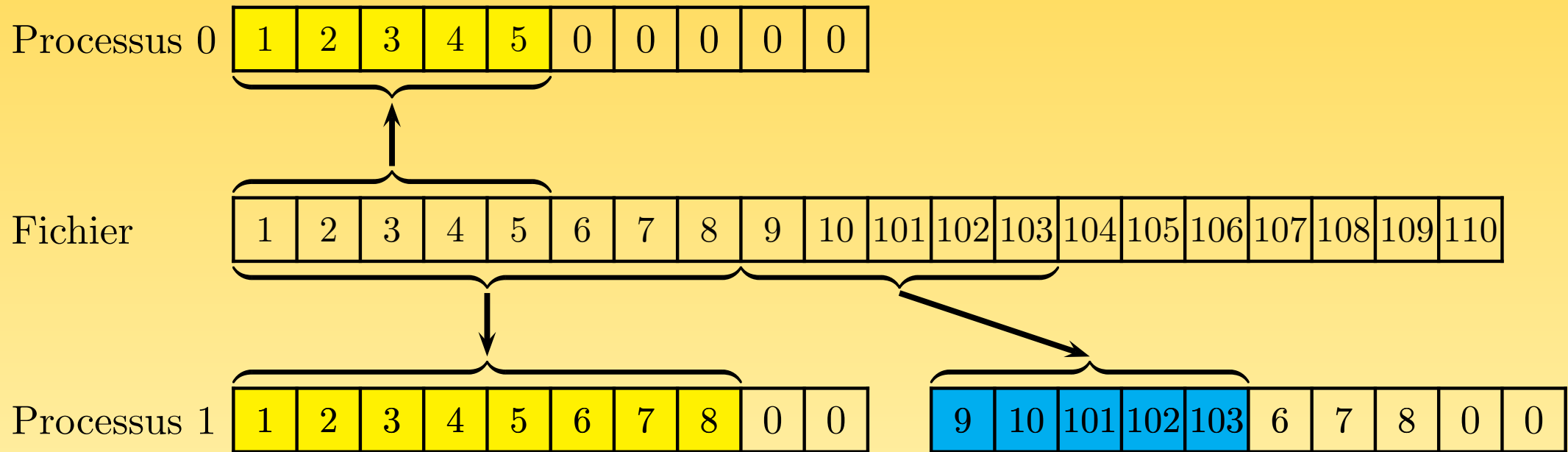


FIG. 56 – Exemple 2 d'utilisation de MPI\_FILE\_READ()

```
> mpiexec -n 2 read02
```

```
Lecture processus 0 : 1, 2, 3, 4, 5, 0, 0, 0
```

```
Lecture processus 1 : 9, 10, 101, 102, 103, 6, 7, 8
```

## 9.4.3 – Via des déplacements implicites partagés

- ➡ 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 pour ce faire **la même vue** du fichier.
- ➡ Si on utilise les variantes non collectives des sous-programmes, l'ordre de lecture **n'est pas déterministe**. Si le traitement doit être déterministe, il faut explicitement gérer l'ordonnancement des processus.
- ➡ Après chaque accès, le pointeur est positionné sur le type élémentaire de donnée suivant.
- ➡ Dans tous ces sous-programmes, les pointeurs individuels ne sont jamais accédés ou modifiés.

```
1 program read_shared01
2
3 use mpi
4 implicit none
5
6 integer                :: rang,descripteur,code
7 integer, parameter    :: nb_valeurs=10
8 integer, dimension(nb_valeurs) :: valeurs
9 integer, dimension(MPI_STATUS_SIZE) :: statut
10
11 call MPI_INIT(code)
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                  descripteur,code)
16
17 call MPI_FILE_READ_SHARED(descripteur,valeurs,4,MPI_INTEGER,statut,code)
18 call MPI_FILE_READ_SHARED(descripteur,valeurs(5),6,MPI_INTEGER,statut,code)
19
20 print *, "Lecture processus",rang,":",valeurs(:)
21
22 call MPI_FILE_CLOSE(descripteur,code)
23 call MPI_FINALIZE(code)
24
25 end program read_shared01
```

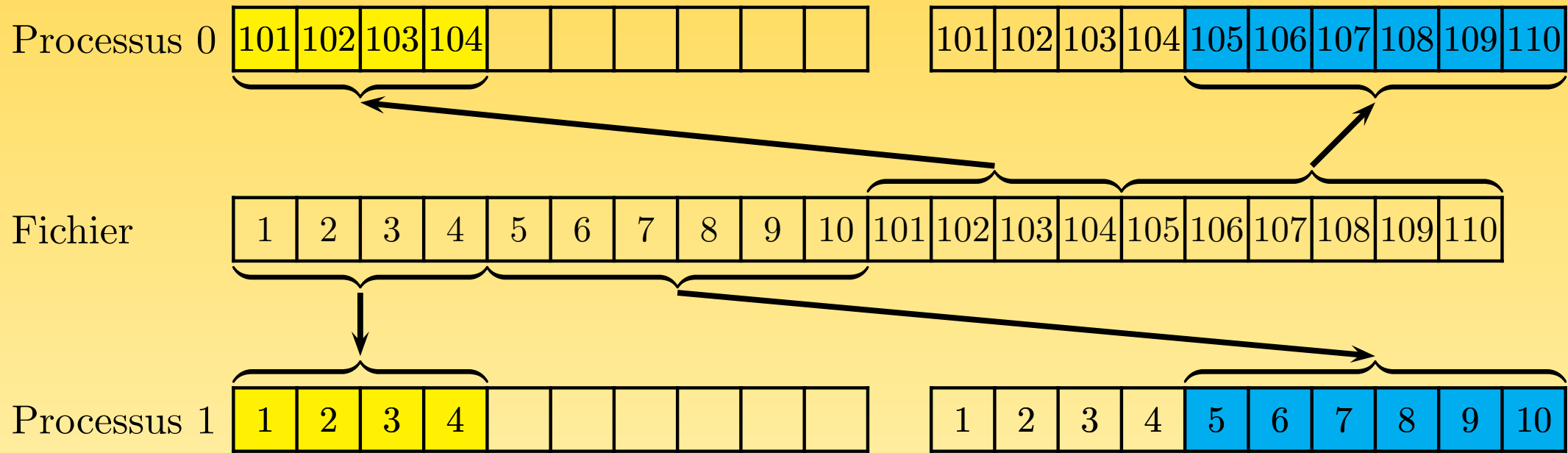


FIG. 57 – Exemple 2 d'utilisation de MPI\_FILE\_READ\_SHARED()

```
> mpiexec -n 2 read_shared01
```

```
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
```

## 9.5 – 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 (comme les accès *en deux phases* — voir le paragraphe **9.1.2**).
- ➡ Dans les opérations collectives, les accès sont effectués **dans l'ordre** des rangs des processus. Le traitement est donc dans ce cas **déterministe**.

## 9.5.1 – Via des déplacements explicites

```
1 program read_at_all
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=10
6   integer                    :: rang,descripteur,code,nb_octets_entier
7   integer(kind=MPI_OFFSET_KIND) :: position_fichier
8   integer, dimension(nb_valeurs) :: valeurs
9   integer, dimension(MPI_STATUS_SIZE) :: statut
10
11  call MPI_INIT(code)
12  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                    descripteur,code)
16
17  call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
18  position_fichier=rang*nb_valeurs*nb_octets_entier
19  call MPI_FILE_READ_AT_ALL(descripteur,position_fichier,valeurs,nb_valeurs, &
20                            MPI_INTEGER,statut,code)
21  print *, "Lecture processus",rang,":",valeurs(:)
22
23  call MPI_FILE_CLOSE(descripteur,code)
24  call MPI_FINALIZE(code)
25 end program read_at_all
```

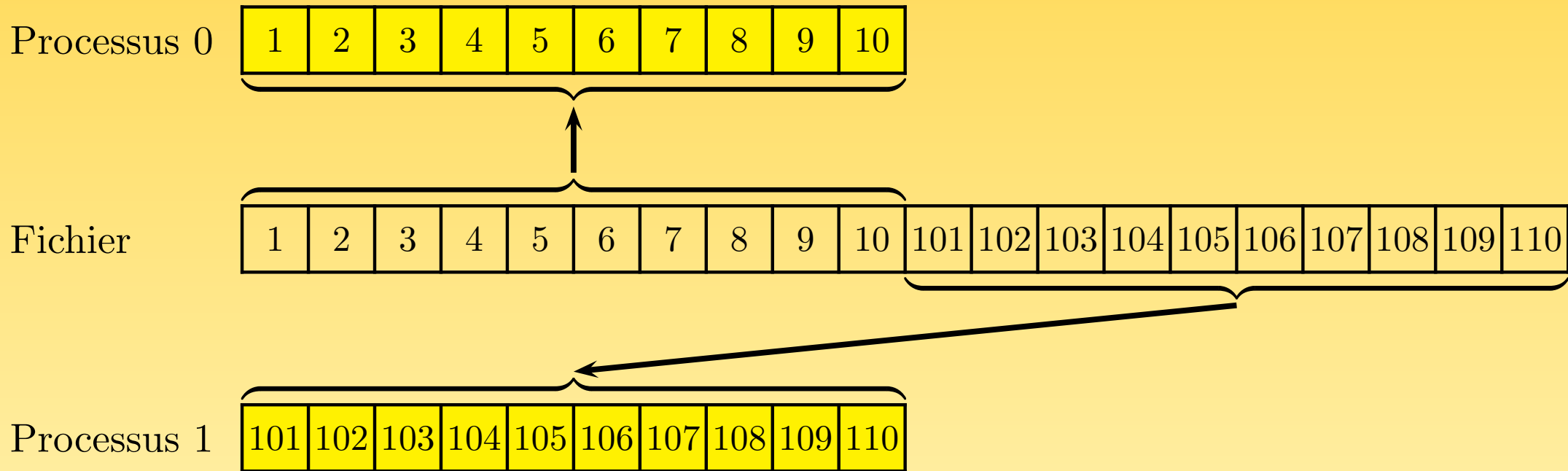


FIG. 58 – Exemple d'utilisation de MPI\_FILE\_READ\_AT\_ALL()

```
> mpiexec -n 2 read_at_all
```

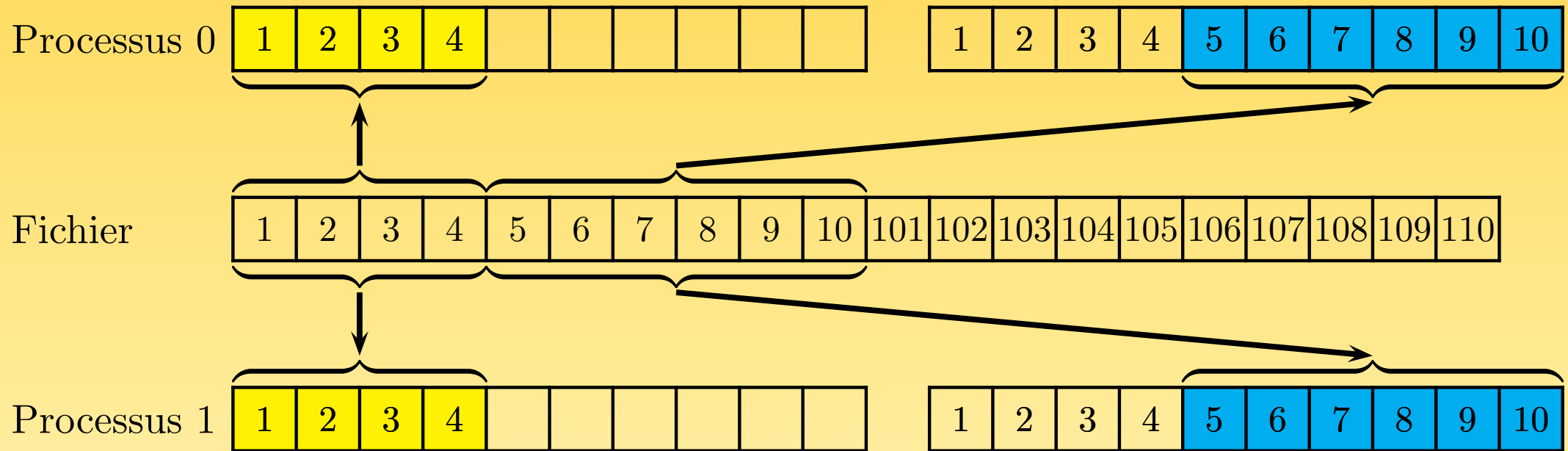
```
Lecture processus 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
Lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```



## 9.5.2 – Via des déplacements implicites individuels

```
1 program read_all01
2   use mpi
3   implicit none
4
5   integer                :: rang,descripteur,code
6   integer, parameter    :: nb_valeurs=10
7   integer, dimension(nb_valeurs) :: valeurs
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                    descripteur,code)
15
16  call MPI_FILE_READ_ALL(descripteur,valeurs,4,MPI_INTEGER,statut,code)
17  call MPI_FILE_READ_ALL(descripteur,valeurs(5),6,MPI_INTEGER,statut,code)
18
19  print *, "Lecture processus ",rang, ":",valeurs(:)
20
21  call MPI_FILE_CLOSE(descripteur,code)
22  call MPI_FINALIZE(code)
23 end program read_all01
```

FIG. 59 – Exemple 1 d'utilisation de `MPI_FILE_READ_ALL()`

```
> mpiexec -n 2 read_all01
```

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

```
Lecture processus 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
1 program read_all02
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=10
6   integer                    :: rang,descripteur,indice1,indice2,code
7   integer, dimension(nb_valeurs) :: valeurs=0
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
13                    descripteur,code)
14
15  if (rang == 0) then
16    indice1=3
17    indice2=6
18  else
19    indice1=5
20    indice2=9
21  end if
22
23  call MPI_FILE_READ_ALL(descripteur,valeurs(indice1),indice2-indice1+1, &
24                        MPI_INTEGER,statut,code)
25  print *, "Lecture processus",rang,":",valeurs(:)
26
27  call MPI_FILE_CLOSE(descripteur,code)
28  call MPI_FINALIZE(code)
29 end program read_all02
```

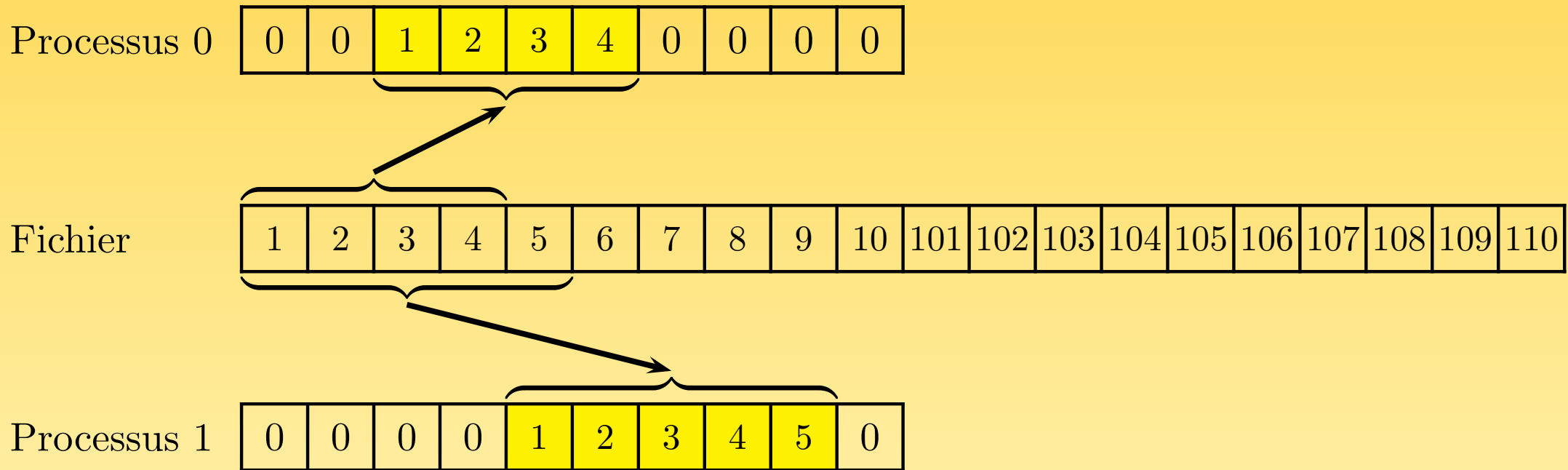


FIG. 60 – Exemple 2 d'utilisation de MPI\_FILE\_READ\_ALL()

```
> mpiexec -n 2 read_all02
```

```
Lecture processus 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
```

```
Lecture processus 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

```
1 program read_all03
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=10
6   integer                    :: rang,descripteur,code
7   integer, dimension(nb_valeurs) :: valeurs=0
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                    descripteur,code)
15
16  if (rang == 0) then
17    call MPI_FILE_READ_ALL(descripteur,valeurs(3),4,MPI_INTEGER,statut,code)
18  else
19    call MPI_FILE_READ_ALL(descripteur,valeurs(5),5,MPI_INTEGER,statut,code)
20  end if
21
22  print *, "Lecture processus",rang,":",valeurs(:)
23
24  call MPI_FILE_CLOSE(descripteur,code)
25  call MPI_FINALIZE(code)
26 end program read_all03
```

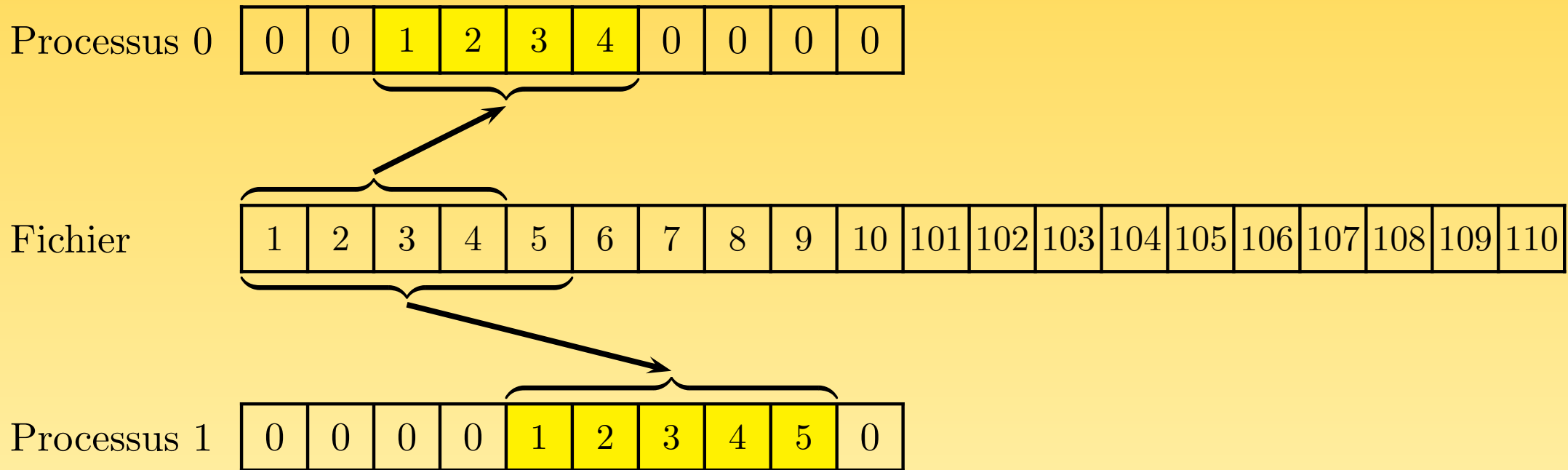


FIG. 61 – Exemple 3 d'utilisation de MPI\_FILE\_READ\_ALL()

```
> mpiexec -n 2 read_all03
```

```
Lecture processus 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
```

```
Lecture processus 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

## 9.5.3 – Via des déplacements implicites partagés

```
1 program read_ordered
2   use mpi
3   implicit none
4
5   integer                :: rang,descripteur,code
6   integer, parameter    :: nb_valeurs=10
7   integer, dimension(nb_valeurs) :: valeurs
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                    descripteur,code)
15
16  call MPI_FILE_READ_ORDERED(descripteur,valeurs,4,MPI_INTEGER,statut,code)
17  call MPI_FILE_READ_ORDERED(descripteur,valeurs(5),6,MPI_INTEGER,statut,code)
18
19  print *, "Lecture processus",rang,":",valeurs(:)
20
21  call MPI_FILE_CLOSE(descripteur,code)
22  call MPI_FINALIZE(code)
23 end program read_ordered
```

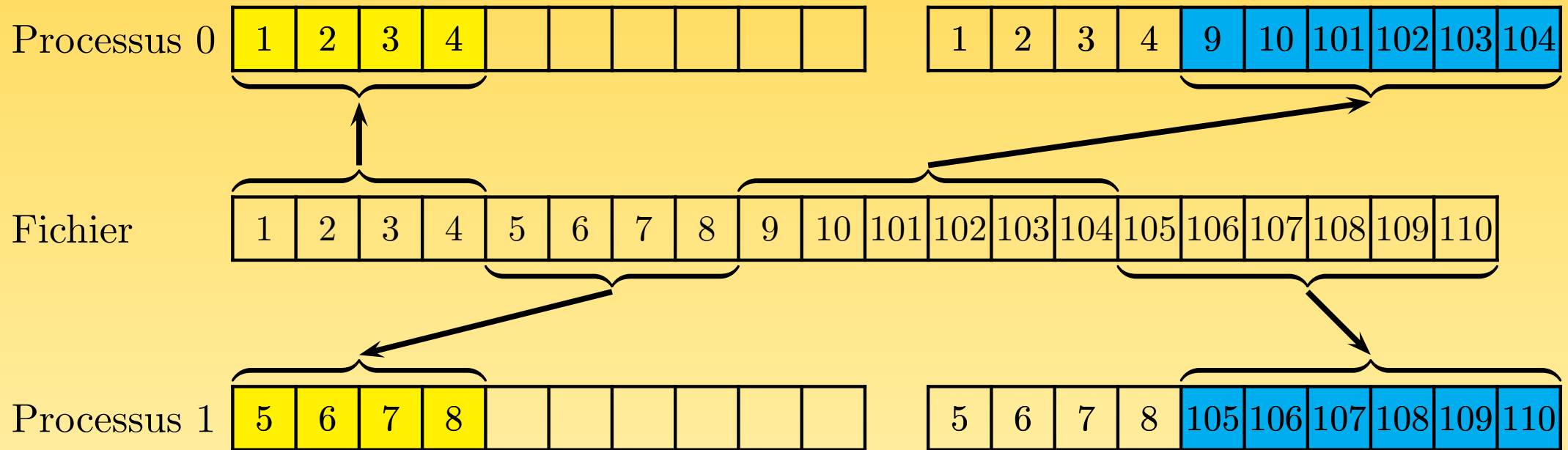


FIG. 62 – Exemple d'utilisation de `MPI_FILE_ORDERED()`

```
> mpiexec -n 2 read_ordered
```

```
Lecture processus 1 : 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
```

```
Lecture processus 0 : 1, 2, 3, 4, 9, 10, 101, 102, 103, 104
```



## 9.6 – Positionnement explicite des pointeurs dans un fichier

- ☞ Les sous-programmes `MPI_FILE_GET_POSITION()` et `MPI_FILE_GET_POSITION_SHARED()` permettent de connaître respectivement la valeur courante des pointeurs individuels et celle du pointeur partagé.
- ☞ 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 fixer la valeur d'un pointeur :
  - ⇒ `MPI_SEEK_SET` fixe une valeur absolue ;
  - ⇒ `MPI_SEEK_CUR` fixe une valeur relative ;
  - ⇒ `MPI_SEEK_END` positionne le pointeur à la fin du fichier, à laquelle un déplacement éventuel est ajouté.
- ☞ Avec `MPI_SEEK_CUR`, on peut spécifier une valeur négative, ce qui permet de revenir **en arrière** dans le fichier.

# 9 – MPI-IO : positionnement explicite des pointeurs 226

```
1 program seek
2   use mpi
3   implicit none
4   integer, parameter          :: nb_valeurs=10
5   integer                    :: rang,descripteur,nb_octets_entier,code
6   integer(kind=MPI_OFFSET_KIND) :: position_fichier
7   integer, dimension(nb_valeurs) :: valeurs
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
13                    descripteur,code)
14
15  call MPI_FILE_READ(descripteur,valeurs,3,MPI_INTEGER,statut,code)
16  call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier,code)
17  position_fichier=8*nb_octets_entier
18  call MPI_FILE_SEEK(descripteur,position_fichier,MPI_SEEK_CUR,code)
19  call MPI_FILE_READ(descripteur,valeurs(4),3,MPI_INTEGER,statut,code)
20  position_fichier=4*nb_octets_entier
21  call MPI_FILE_SEEK(descripteur,position_fichier,MPI_SEEK_SET,code)
22  call MPI_FILE_READ(descripteur,valeurs(7),4,MPI_INTEGER,statut,code)
23
24  print *, "Lecture processus",rang,":",valeurs(:)
25
26  call MPI_FILE_CLOSE(descripteur,code)
27  call MPI_FINALIZE(code)
28 end program seek
```

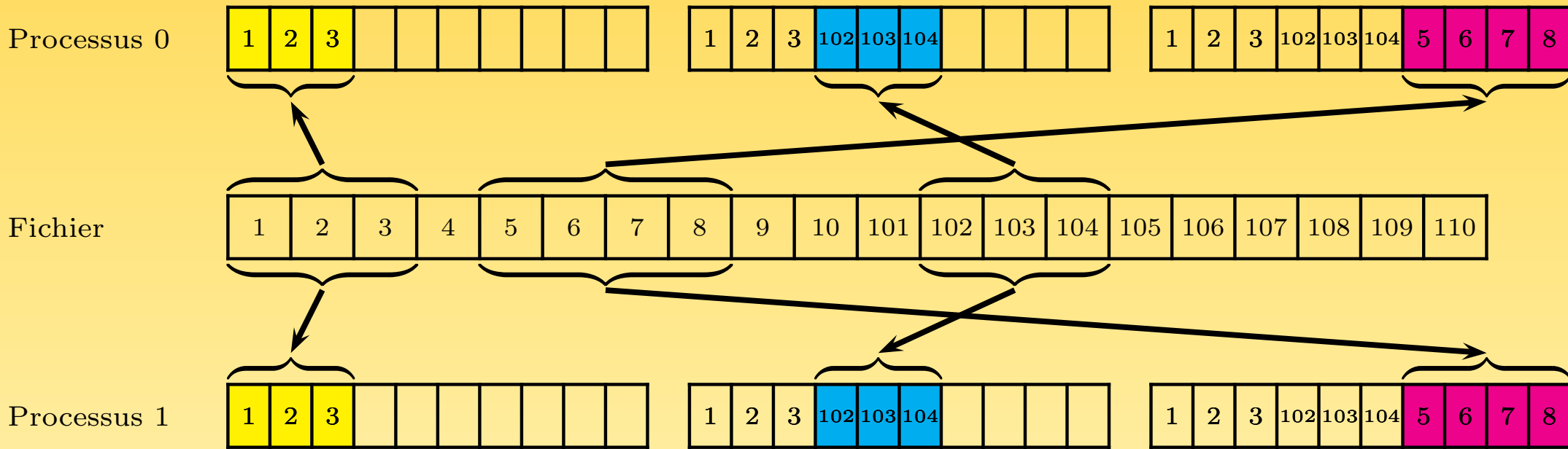


FIG. 63 – Exemple d'utilisation de `MPI_FILE_SEEK()`

```
> mpiexec -n 2 seek
```

```
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
```

## 9.7 – Définition de vues

- ➡ Les **vues** sont un mécanisme souple et puissant pour décrire les zones accédées dans les fichiers.
- ➡ Les vues sont construites à l'aide de **types dérivés** MPI.
- ➡ Chaque processus a sa propre vue (ou ses propres vues) d'un fichier, définie par trois variables : un **déplacement initial**, un **type élémentaire de données** et un **motif**. Une vue est définie comme répétition du motif, une fois le positionnement initial effectué.
- ➡ Il est possible de définir des **trous** dans une vue, de façon à ne pas tenir compte de certaines parties des données.
- ➡ Des processus différents peuvent parfaitement avoir des **vues différentes** du fichier, de façon à accéder à des parties complémentaires de celui-ci.
- ➡ Un processus donné peut définir et utiliser **plusieurs vues** différentes du même fichier.
- ➡ 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 types élémentaires et les motifs ne peuvent se recouvrir, même partiellement.
- ➔ La vue par défaut consiste en une simple suite d'octets (déplacement initial nul, *type\_élém* et motif égaux à `MPI_BYTE`).

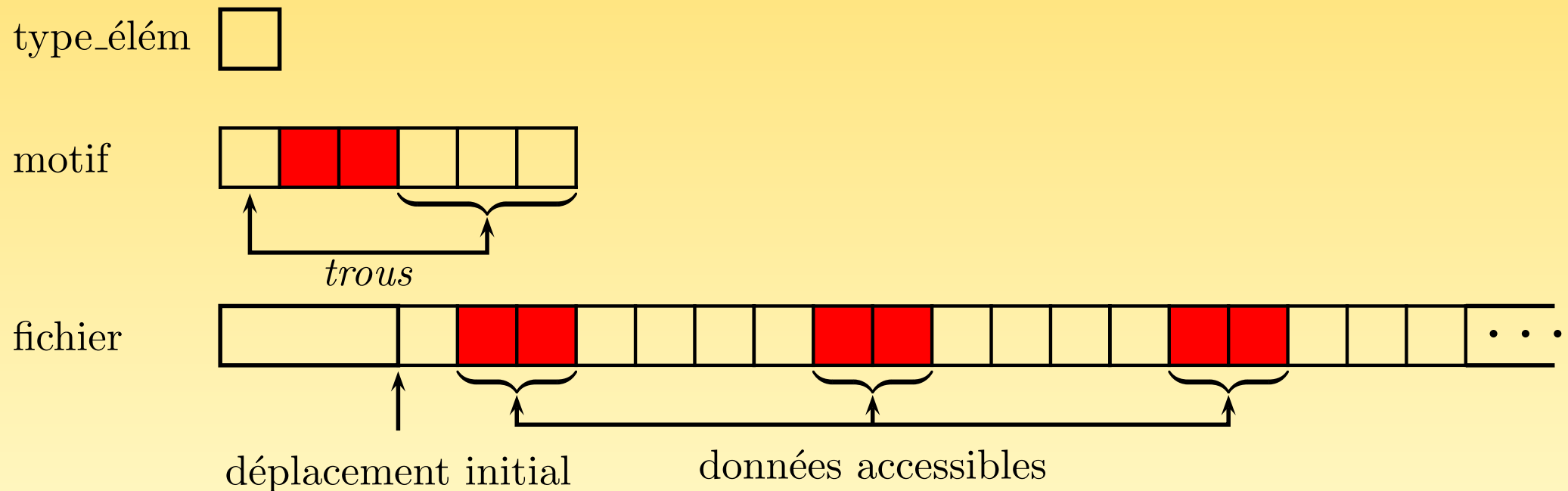


FIG. 64 – Type élémentaire de donnée et motif

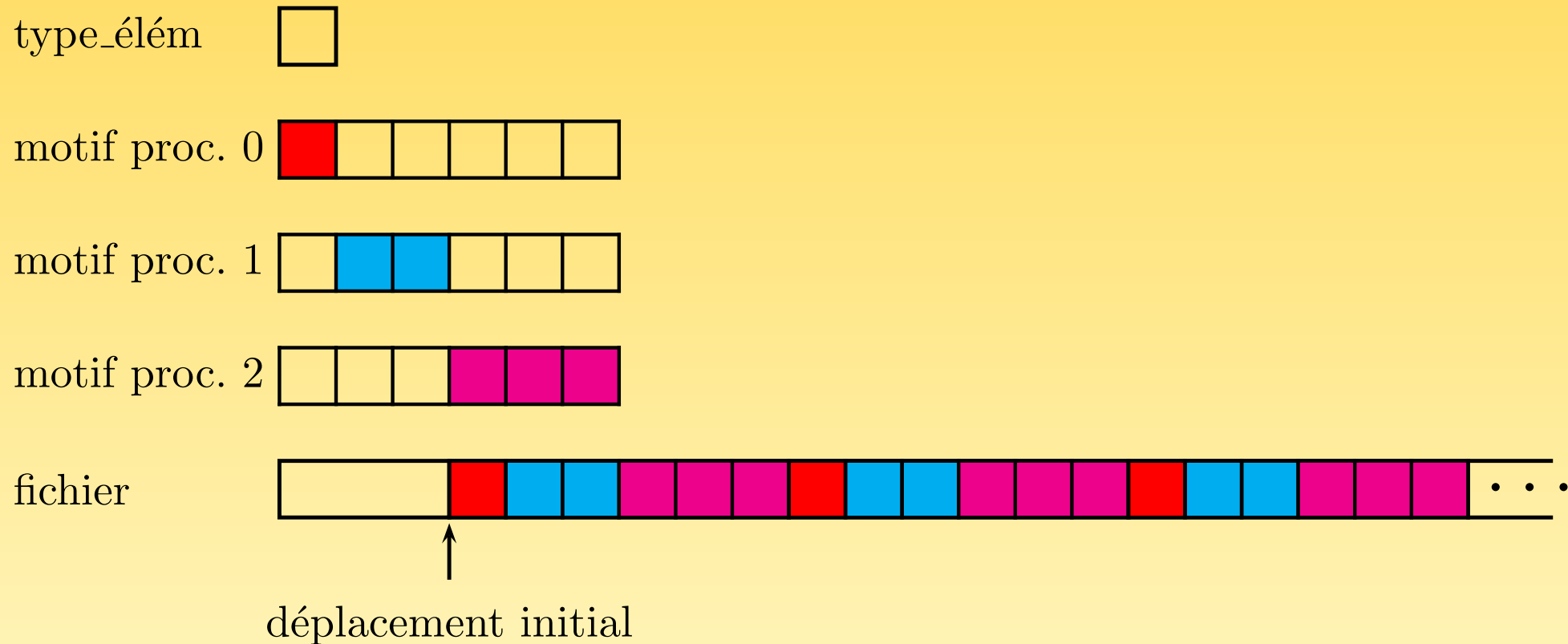


FIG. 65 – Exemple de définition de motifs différents selon les processus

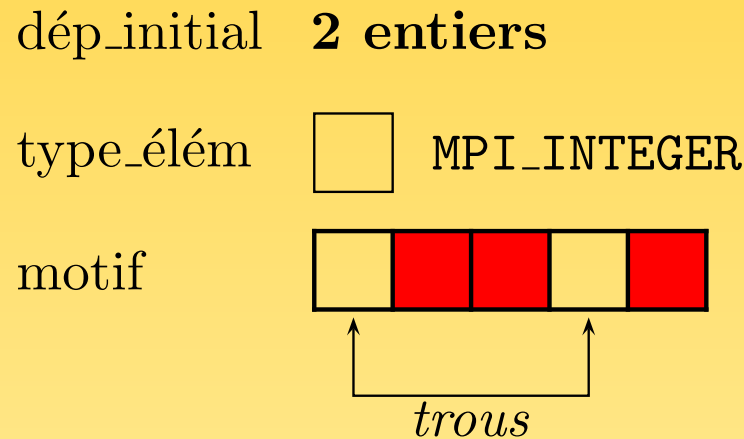


FIG. 66 – Motif employé dans l'exemple 1 d'utilisation de MPI\_FILE\_SET\_VIEW()

```

1 program read_view01
2
3   use mpi
4   implicit none
5
6   integer, parameter                :: nb_valeurs=10
7   integer(kind=MPI_ADDRESS_KIND), parameter :: borne_inf_zero=0
8   integer                          :: rang,descripteur,motif_temp,motif,code
9   integer(kind=MPI_OFFSET_KIND)    :: déplacement_initial
10  integer(kind=MPI_ADDRESS_KIND)    :: borne_inf,taille
11  integer, dimension(2)              :: longueurs,deplacements
12  integer, dimension(nb_valeurs)    :: valeurs
13  integer, dimension(MPI_STATUS_SIZE) :: statut
14
15  call MPI_INIT(code)
16  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

```

```
17
18 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
19                   descripteur,code)
20
21 déplacements(1)=1
22 déplacements(2)=4
23 longueurs(1)=2
24 longueurs(2)=1
25
26 call MPI_TYPE_INDEXED(2,longueurs,déplacements,MPI_INTEGER,motif_temp,code)
27 call MPI_TYPE_GET_EXTENT(motif_temp,borne_inf,taille,code)
28 call MPI_TYPE_CREATE_RESIZED(motif_temp,borne_inf_zero,borne_inf+taille,motif,code)
29 call MPI_TYPE_COMMIT(motif,code)
30
31 ! Ne pas omettre de passer par une variable intermédiaire de représentation
32 ! MPI_OFFSET_KIND, pour des raisons de portabilité
33 déplacement_initial=0
34 call MPI_FILE_SET_VIEW(descripteur,déplacement_initial,MPI_INTEGER,motif, &
35                       "native",MPI_INFO_NULL,code)
36
37 call MPI_FILE_READ(descripteur,valeurs,7,MPI_INTEGER,statut,code)
38 call MPI_FILE_READ(descripteur,valeurs(8),3,MPI_INTEGER,statut,code)
39
40 print *,"Lecture processus",rang,":",valeurs(:)
41
42 call MPI_FILE_CLOSE(descripteur,code)
43 call MPI_FINALIZE(code)
44
45 end program read_view01
```



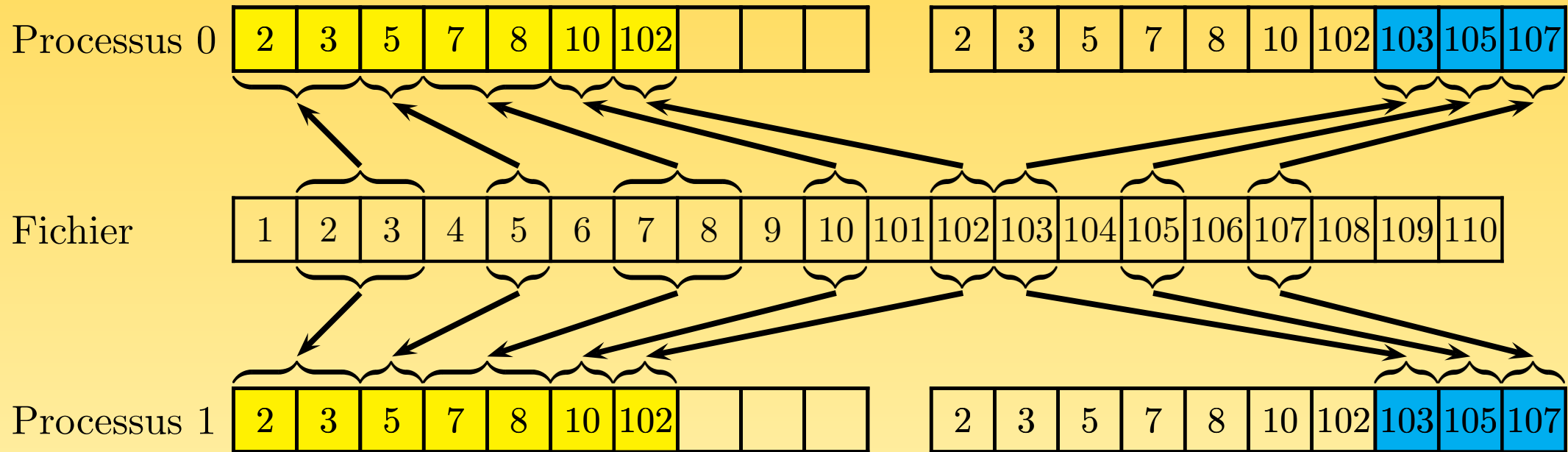


FIG. 67 – Exemple 1 d'utilisation de MPI\_FILE\_SET\_VIEW()

```
> mpiexec -n 2 read_view01
```

```
Lecture processus 1 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
```

```
Lecture processus 0 : 2, 3, 5, 7, 8, 10, 102, 103, 105, 107
```

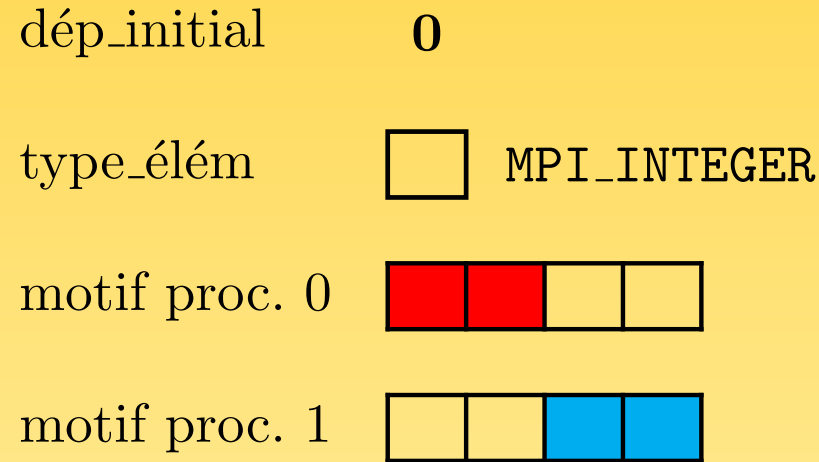


FIG. 68 – Motif employé dans l'exemple 2 d'utilisation de MPI\_FILE\_SET\_VIEW()

```

1 program read_view02
2
3 use mpi
4 implicit none
5
6 integer, parameter      :: nb_valeurs=10
7 integer                 :: rang,descripteur,coord,motif,code
8 integer(kind=MPI_OFFSET_KIND) :: displacement_initial
9 integer, dimension(nb_valeurs) :: valeurs
10 integer, dimension(MPI_STATUS_SIZE) :: statut
11
12 call MPI_INIT(code)
13 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14

```

```
15 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                     descripteur,code)
17
18 if (rang == 0) then
19     coord=1
20 else
21     coord=3
22 end if
23
24 call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/coord - 1/), &
25                               MPI_ORDER_FORTRAN,MPI_INTEGER,motif,code)
26 call MPI_TYPE_COMMIT(motif,code)
27
28 deplacement_initial=0
29 call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif, &
30                       "native",MPI_INFO_NULL,code)
31
32 call MPI_FILE_READ(descripteur,valeurs,nb_valeurs,MPI_INTEGER,statut,code)
33
34 print *, "Lecture processus",rang,":",valeurs(:)
35
36 call MPI_FILE_CLOSE(descripteur,code)
37 call MPI_FINALIZE(code)
38
39 end program read_view02
```

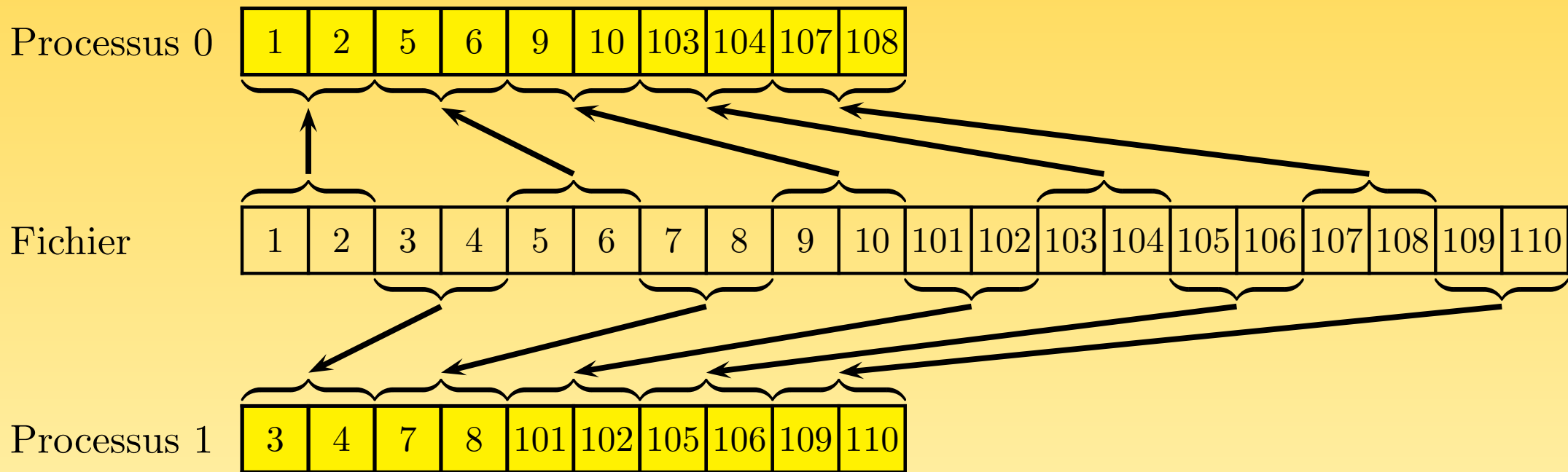


FIG. 69 – Exemple 2 d'utilisation de MPI\_FILE\_SET\_VIEW()

```
> mpiexec -n 2 read_view02
```

```
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
```

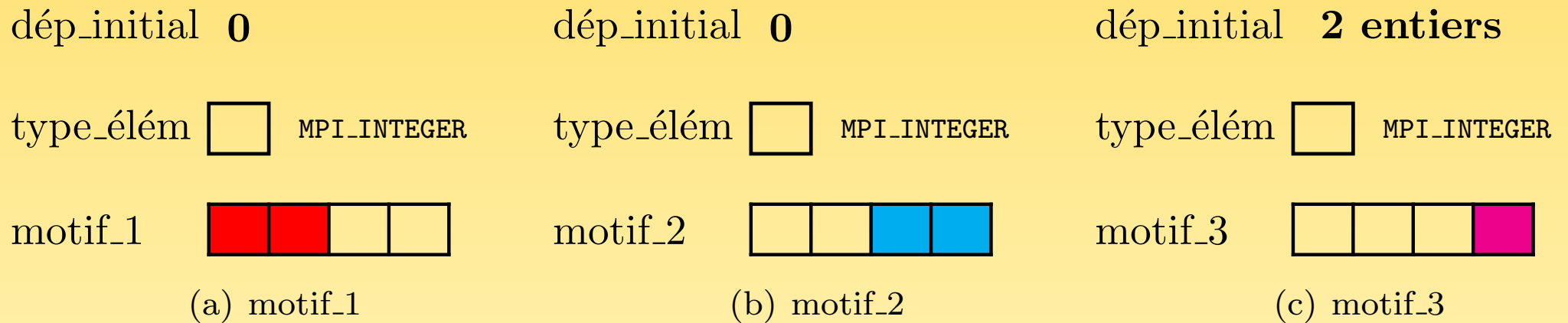


FIG. 70 – Motifs employés dans l'exemple 3 d'utilisation de MPI\_FILE\_SET\_VIEW()

```
1 program read_view03
2
3 use mpi
4 implicit none
5
6 integer, parameter          :: nb_valeurs=10
7 integer                    :: rang,descripteur,code, &
8                            motif_1,motif_2,motif_3,nb_octets_entier
9 integer(kind=MPI_OFFSET_KIND) :: displacement_initial
10 integer, dimension(nb_valeurs) :: valeurs
11 integer, dimension(MPI_STATUS_SIZE) :: statut
12
13 call MPI_INIT(code)
14 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
15 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                   descripteur,code)
17
18 call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/0/), &
19                               MPI_ORDER_FORTRAN,MPI_INTEGER,motif_1,code)
20 call MPI_TYPE_COMMIT(motif_1,code)
21
22 call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/2/), &
23                               MPI_ORDER_FORTRAN,MPI_INTEGER,motif_2,code)
24 call MPI_TYPE_COMMIT(motif_2,code)
25
26 call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/1/),(/3/), &
27                               MPI_ORDER_FORTRAN,MPI_INTEGER,motif_3,code)
28 call MPI_TYPE_COMMIT(motif_3,code)
29
```

```
30  déplacement_initial=0
31  call MPI_FILE_SET_VIEW(descripteur, déplacement_initial, MPI_INTEGER, motif_1, &
32                        "native", MPI_INFO_NULL, code)
33  call MPI_FILE_READ(descripteur, valeurs, 4, MPI_INTEGER, statut, code)
34
35  call MPI_FILE_SET_VIEW(descripteur, déplacement_initial, MPI_INTEGER, motif_2, &
36                        "native", MPI_INFO_NULL, code)
37  call MPI_FILE_READ(descripteur, valeurs(5), 3, MPI_INTEGER, statut, code)
38
39  call MPI_TYPE_SIZE(MPI_INTEGER, nb_octets_entier, code)
40  déplacement_initial=2*nb_octets_entier
41  call MPI_FILE_SET_VIEW(descripteur, déplacement_initial, MPI_INTEGER, motif_3, &
42                        "native", MPI_INFO_NULL, code)
43  call MPI_FILE_READ(descripteur, valeurs(8), 3, MPI_INTEGER, statut, code)
44
45  print *, "Lecture processus", rang, ":", valeurs(:)
46
47  call MPI_FILE_CLOSE(descripteur, code)
48  call MPI_FINALIZE(code)
49  end program read_view03
```

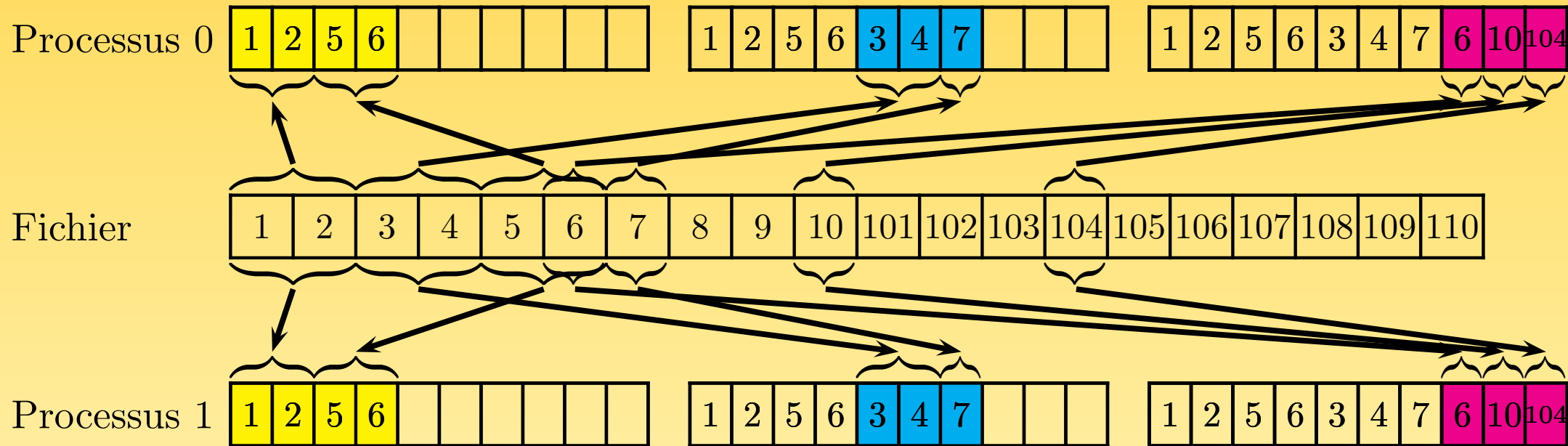


FIG. 71 – Exemple 3 d'utilisation de MPI\_FILE\_SET\_VIEW()

```
> mpiexec -n 2 read_view03
```

```
Lecture processus 1 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```

```
Lecture processus 0 : 1, 2, 5, 6, 3, 4, 7, 6, 10, 104
```



## 9.8 – Lectures/écritures non bloquantes

- ➡ Les entrées-sorties non bloquantes sont implémentées suivant le modèle utilisé pour les communications non bloquantes.
- ➡ 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.), de façon similaire à la gestion des messages non bloquants.
- ➡ L'intérêt est de faire un recouvrement entre les calculs et les entrées-sorties.

## 9.8.1 – Via des déplacements explicites

```
1 program iread_at
2
3 use mpi
4 implicit none
5
6 integer, parameter :: nb_valeurs=10
7 integer :: i,nb_iterations=0,rang,nb_octets_entier, &
8         descripteur,requete,code
9 integer(kind=MPI_OFFSET_KIND) :: position_fichier
10 integer, dimension(nb_valeurs) :: valeurs
11 integer, dimension(MPI_STATUS_SIZE) :: statut
12 logical :: termine
13
14 call MPI_INIT(code)
15 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
16
17 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
18         descripteur,code)
19
```

```
20 call MPI_TYPE_SIZE (MPI_INTEGER, nb_octets_entier, code)
21
22 position_fichier=rang*nb_valeurs*nb_octets_entier
23 call MPI_FILE_IREAD_AT(descriptor, position_fichier, valeurs, nb_valeurs, &
24     MPI_INTEGER, requete, code)
25
26 do while (nb_iterations < 5000)
27     nb_iterations=nb_iterations+1
28     ! Calculs recouvrant le temps demandé par l'opération de lecture
29     ...
30     call MPI_TEST (requete, termine, statut, code)
31     if (termine) exit
32 end do
33 print *, "Après", nb_iterations, "iterations, lecture processus", rang, ":", valeurs
34
35 call MPI_FILE_CLOSE(descriptor, code)
36 call MPI_FINALIZE (code)
37
38 end program iread_at
```

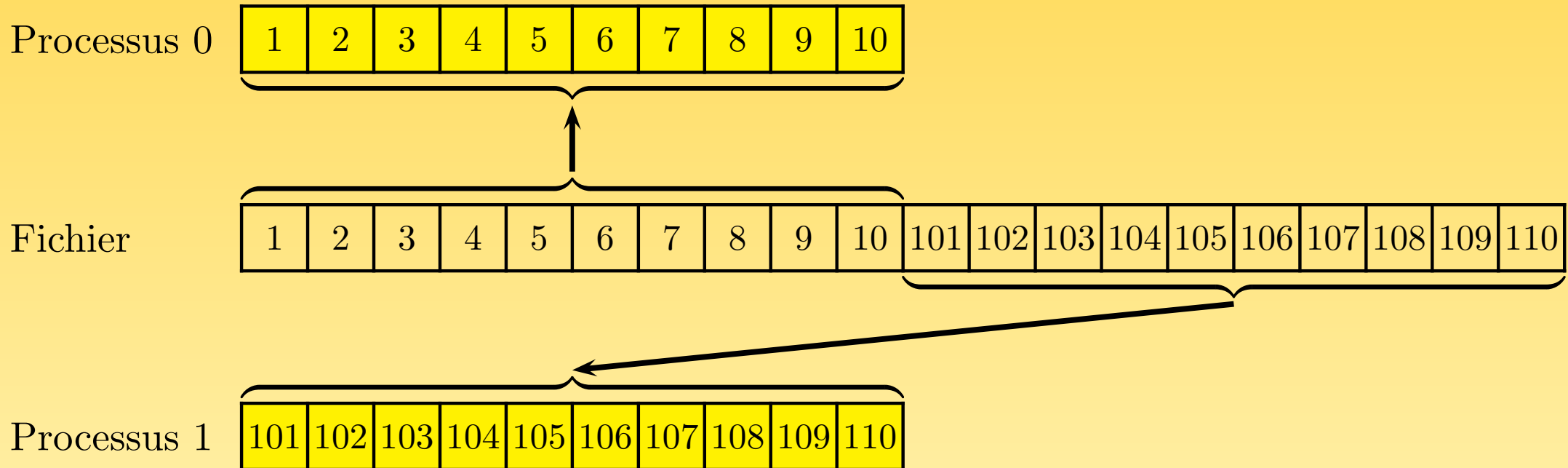


FIG. 72 – Exemple d'utilisation de MPI\_FILE\_IREAD\_AT()

```
> mpiexec -n 2 iread_at
```

```
Après 1 iterations, lecture processus 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
Après 1 iterations, lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

## 9.8.2 – Via des déplacements implicites individuels

```
1 program iread
2   use mpi
3   implicit none
4
5   integer, parameter          :: nb_valeurs=10
6   integer                    :: rang,descripteur,requete,code
7   integer, dimension(nb_valeurs) :: valeurs
8   integer, dimension(MPI_STATUS_SIZE) :: statut
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                    descripteur,code)
15
16  call MPI_FILE_IREAD(descripteur,valeurs,nb_valeurs,MPI_INTEGER,requete,code)
17  ! Calcul recouvrant le temps demandé par l'opération de lecture
18  ...
19  call MPI_WAIT(requete,statut,code)
20  print *, "Lecture processus",rang,":",valeurs(:)
21
22  call MPI_FILE_CLOSE(descripteur,code)
23  call MPI_FINALIZE(code)
24 end program iread
```

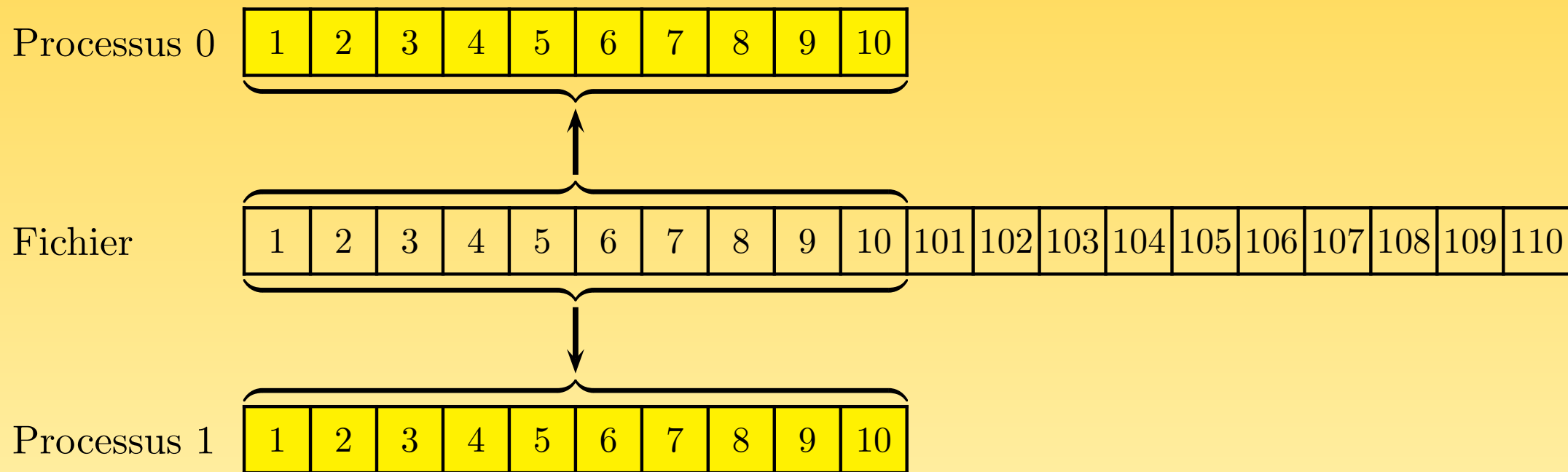


FIG. 73 – Exemple 1 d'utilisation de MPI\_FILE\_IREAD()

```
> mpiexec -n 2 iread
```

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

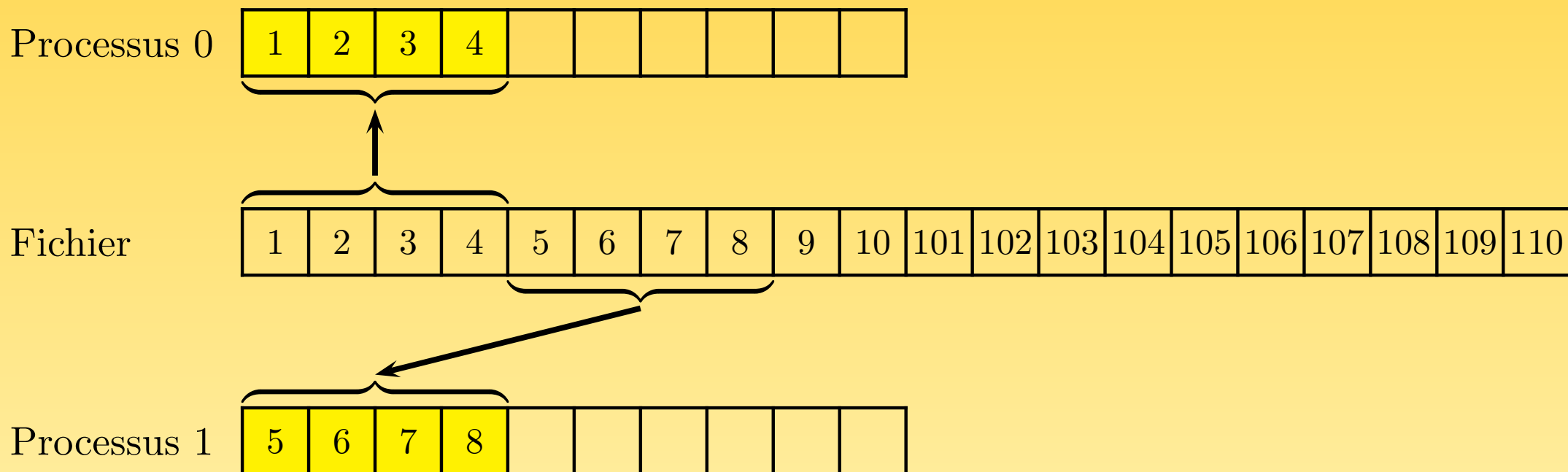
```
Lecture processus 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

## 9.8.3 – Lectures/écritures collectives et non bloquantes

- ➡ Il est possible d'effectuer des opérations qui soient à la fois **collectives** et **non bloquantes**, via une forme particulière d'opération collective non bloquante.
- ➡ Celle-ci nécessite un appel à deux sous-programmes distincts, l'un pour déclencher l'opération et l'autre pour la terminer.
- ➡ On ne peut modifier la zone mémoire concernée entre les deux phases de l'opération.
- ➡ Néanmoins, il est possible pendant ce temps de faire des opérations non collectives sur le fichier.
- ➡ Il ne peut y avoir qu'**une seule** telle opération en cours à la fois par processus.

```
1 program read_ordered_begin_end
2
3 use mpi
4 implicit none
5
6 integer                :: rang,descripteur,code
7 integer, parameter    :: nb_valeurs=10
8 integer, dimension(nb_valeurs) :: valeurs
9 integer, dimension(MPI_STATUS_SIZE) :: statut
10
11 call MPI_INIT(code)
12 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14 call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                   descripteur,code)
16
17 call MPI_FILE_READ_ORDERED_BEGIN(descripteur,valeurs,4,MPI_INTEGER,code)
18 print *, "Processus numéro      :",rang
19 call MPI_FILE_READ_ORDERED_END(descripteur,valeurs,statut,code)
20
21 print *, "Lecture processus",rang,":",valeurs(1:4)
22
23 call MPI_FILE_CLOSE(descripteur,code)
24 call MPI_FINALIZE(code)
25
26 end program read_ordered_begin_end
```



FIG. 74 – Exemple d'utilisation de `MPI_FILE_READ_ORDERED_BEGIN()`

```
> mpiexec -n 2 read_ordered_begin_end
```

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

## 9.9 – Conseils

- ☞ Comme on a pu le voir, MPI-IO offre un ensemble très riche de fonctionnalités, en même temps qu'une interface de haut niveau. Celle-ci, tout en restant portable, permet à la fois de masquer aux utilisateurs des opérations complexes et d'implémenter de façon transparente des optimisations particulières aux machines cibles.
- ☞ Certains choix semblent clairement à conseiller :
  - ⇒ lorsque les opérations font intervenir tous les processus, ou un ensemble d'entre eux qui peuvent être définis dans un communicateur particulier, il faut généralement privilégier la forme **collective** des opérations ;
  - ⇒ l'utilisation des sous-programmes à positionnement explicite dans les fichiers ne sont à employer que dans des cas particuliers, l'utilisation **implicite** de pointeurs individuels ou partagés offrant une interface de plus haut niveau ;
  - ⇒ 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.

## 10 – 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.
- ➡ Ecrire les communications comme si les envois étaient synchrones (`MPI_SSEND()`).
- ➡ Eviter 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, 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/>).

## 11 – Annexes

Il s'agit ici de programmes concernant différentes fonctionnalités de MPI qui sont :

- ➡ moins fréquentes d'utilisation (création de sa propre opération de réduction, type dérivés spécifiques, topologie de type graphe, communications persistantes) ;
- ➡ qui ne sont pas disponibles sur l'ensemble des machines (gestion dynamique de processus, mode client serveur).

**11.1 – Communications collectives**

Dans cet exemple, on se propose de créer sa propre opération de réduction, produit de vecteurs de nombres complexes.

```
1 program ma_reduction
2   use mpi
3   implicit none
4   integer          :: rang,code,i, mon_operation
5   integer, parameter :: n=4
6   complex, dimension(n) :: a,resultat
7   external mon_produit
8
9   call MPI_INIT(code)
10  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
11  ! Initialisation du vecteur A sur chaque processus
12  a(:) = (/ (cmplx(rang+i,rang+i+1),i=1,n) /)
13  ! Création de l'opération commutative mon_operation
14  call MPI_OP_CREATE(mon_produit,.true.,mon_operation,code)
15  ! Collecte sur le processus 0 du produit global
16  call MPI_REDUCE(a,resultat,n,MPI_COMPLEX,mon_operation,0,MPI_COMM_WORLD,code)
17
18  ! Affichage du résultat
19  if (rang == 0) then
20    print *, 'Valeur du produit', resultat
21  end if
22  call MPI_FINALIZE(code)
23 end program ma_reduction
```

```
1 ! Définition du produit terme à terme de deux vecteurs de nombres complexes
2
3 integer function mon_produit(vecteur1,vecteur2,longueur,type_donnee) result(inutilise)
4   implicit none
5
6   complex,dimension(longueur) :: vecteur1,vecteur2
7   integer                       :: longueur,type_donnee,i
8
9   do i=1,longueur
10      vecteur2(i) = cplx(real(vecteur1(i))*real(vecteur2(i)) - &
11                        aimag(vecteur1(i))*aimag(vecteur2(i)), &
12                        real(vecteur1(i))*aimag(vecteur2(i)) + &
13                        aimag(vecteur1(i))*real(vecteur2(i)))
14   end do
15
16   inutilise=0
17
18 end function mon_produit
```

```
> mpiexec -n 5 ma_reduction
```

```
Valeur du produit (155.,-2010.), (-1390.,-8195.), (-7215.,-23420.), (-22000.,-54765.)
```

## 11.2 – Types de données dérivés

### 11.2.1 – Distribution d'un tableau sur plusieurs processus

➡ Le sous-programme `MPI_TYPE_CREATE_DARRAY()` permet de générer un tableau sur un ensemble de processus suivant une distribution par blocs ou cyclique.

```
integer,intent(in)                :: nb_procs,rang,nb_dims
integer,dimension(nb_dims),intent(in) :: profil_tab,mode_distribution
integer,dimension(nb_dims),intent(in) :: profil_sous_tab,distribution_procs
integer,intent(in)                :: ordre,ancien_type
integer,intent(out)               :: nouveau_type,code
call MPI_TYPE_CREATE_DARRAY(nb_procs,rang,nb_dims,profil_tab,mode_distribution,
                           profil_sous_tab,distribution_procs,ordre,ancien_type,
                           nouveau_type,code)
```

- ➡ **nb\_dims** : rang du tableau
- ➡ **nb\_procs** : nombre total de processus
- ➡ **rang** : rang de chaque processus
- ➡ **profil\_tab** : profil du tableau à distribuer
- ➡ **mode\_distribution** : mode de distribution dans chaque dimension du tableau, soit :
  1. **MPI\_DISTRIBUTE\_BLOCK** indique une distribution par blocs
  2. **MPI\_DISTRIBUTE\_CYCLIC** indique une distribution cyclique
  3. **MPI\_DISTRIBUTE\_NONE** indique qu'il n'y a pas de distribution
- ➡ **profil\_sous\_tab** : profil d'un bloc
- ➡ **distribution\_procs** : nombre de processus dans chaque dimension



Quelques remarques :

- ➡ l'ordre des processus est le même que pour les topologies ;
- ➡ pour que l'appel au sous-programme soit correct, on doit avoir
$$\text{nb\_procs} = \prod_{i=1}^{\text{nb\_dims}} \text{distribution\_procs}(i) ;$$
- ➡ lorsqu'une dimension  $i$  est distribuée par blocs, via le paramètre **MPI\_DISTRIBUTE\_BLOCK**, la règle suivante doit être respectée
$$\text{profil\_sous\_tab}(i) * \text{distribution\_procs}(i) \geq \text{profil\_tab}(i) ;$$
- ➡ lorsqu'une dimension  $i$  n'est pas distribuée, via le paramètre **MPI\_DISTRIBUTE\_NONE**, le nombre de processus choisi dans cette dimension doit valoir 1 ( $\text{distribution\_procs}(i) = 1$ ) et le profil du bloc dans cette dimension ( $\text{profil\_sous\_tab}(i)$ ) est ignoré.

## Distribution par blocs d'un tableau suivant 4 processus

➡ `nb_procs = 4, distribution_procs(:) = (/ 2,2 /)`

➡ `profil_tab(:) = (/ 4,6 /)`, `profil_sous_tab(:) = (/ 2,3 /)`

➡ `mode_distribution(:) = (/ MPI_DISTRIBUTE_BLOCK, MPI_DISTRIBUTE_BLOCK /)`

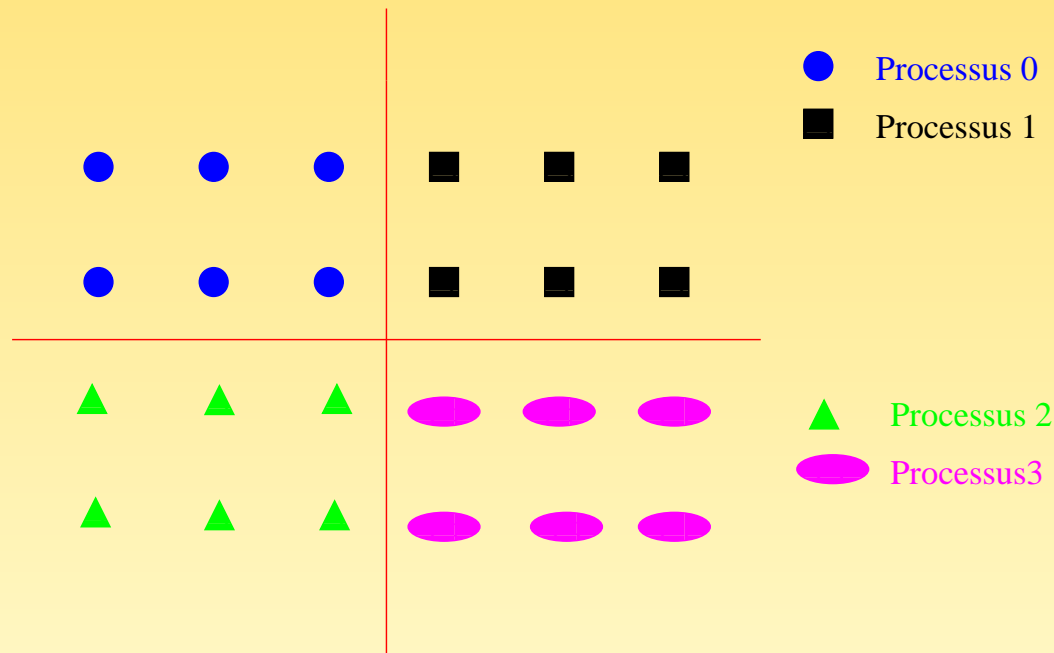


FIG. 75 – Définition du type dérivé sur chaque processus pour une distribution par blocs

```
1 program darray_bloc
2   use mpi
3   implicit none
4   integer,parameter                :: nb_lignes=4,nb_colonnes=6,&
5                                   nb_dims=2,etiquette1=1000,etiquette2=1001
6   integer                          :: nb_procs,code,rang,i,type_bloc
7   integer,dimension(nb_lignes,nb_colonnes) :: tab
8   integer,dimension(nb_dims)       :: profil_tab,mode_distribution,&
9                                   profil_sous_tab,distribution_procs
10  integer,dimension(MPI_STATUS_SIZE) :: statut
11
12  call MPI_INIT(code)
13  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
14  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
15
16  !Initialisation du tableau tab sur chaque processus
17  tab(:,:)=reshape((/(i*(rang+1),i=1,nb_lignes*nb_colonnes)/),(/nb_lignes,nb_colonnes/))
18
19  !Profil du tableau tab
20  profil_tab(:) = shape(tab)
21
22  !Mode de distribution
23  mode_distribution(:) = (/ MPI_DISTRIBUTE_BLOCK,MPI_DISTRIBUTE_BLOCK /)
24
25  !Profil d'un bloc
26  profil_sous_tab(:) = (/ 2,3 /)
```

```
27 !Nombre de processus dans chaque dimension
28 distribution_procs(:) = (/ 2,2 /)
29 !Création du type dérivé type_bloc
30 call MPI_TYPE_CREATE_DARRAY(nb_procs,rang,nb_dims,profil_tab,mode_distribution,&
31                             profil_sous_tab, distribution_procs,MPI_ORDER_FORTRAN,&
32                             MPI_INTEGER,type_bloc,code)
33 call MPI_TYPE_COMMIT(type_bloc,code)
34 select case(rang)
35   case(0)
36     !Le processus 0 envoie son tableau tab au processus 1
37     call MPI_SEND(tab,1,type_bloc,1,etiquette1,MPI_COMM_WORLD,code)
38   case(1)
39     !Le processus 1 reçoit son tableau tab du processeur 0
40     call MPI_RECV(tab,1,type_bloc,0,etiquette1,MPI_COMM_WORLD,statut,code)
41   case(2)
42     !Le processus 2 envoie son tableau tab au processus 3
43     call MPI_SEND(tab,1,type_bloc,3,etiquette2,MPI_COMM_WORLD,code)
44   case(3)
45     !Le processus 3 reçoit son tableau tab du processeur 2
46     call MPI_RECV(tab,1,type_bloc,2,etiquette2,MPI_COMM_WORLD,statut,code)
47 end select
48 !Affichage du tableau tab sur chaque processus
49 .....
50 call MPI_TYPE_FREE(type_bloc,code)
51 call MPI_FINALIZE(code)
52 end program darray_bloc
```

```
> mpiexec -n 4 darray_bloc
```

Tableau tab obtenu sur le processus 0

```
1, 5, 9, 13, 17, 21
2, 6, 10, 14, 18, 22
3, 7, 11, 15, 19, 23
4, 8, 12, 16, 20, 24
```

Tableau tab obtenu sur le processus 1

```
2, 10, 18, 1, 5, 9
4, 12, 20, 2, 6, 10
6, 14, 22, 30, 38, 46
8, 16, 24, 32, 40, 48
```

Tableau tab obtenu sur le processus 2

```
3, 15, 27, 39, 51, 63
6, 18, 30, 42, 54, 66
9, 21, 33, 45, 57, 69
12, 24, 36, 48, 60, 72
```

Tableau tab obtenu sur le processus 3

```
4, 20, 36, 52, 68, 84
8, 24, 40, 56, 72, 88
12, 28, 44, 9, 21, 33
16, 32, 48, 12, 24, 36
```

## Distribution cyclique d'un tableau suivant 4 processus

➡ `nb_procs = 4, distribution_procs(:) = (/ 2,2 /)`

➡ `profil_tab(:) = (/ 4,6 /), profil_sous_tab(:) = (/ 1,2 /)`

➡ `mode_distribution(:) = (/ MPI_DISTRIBUTE_CYCLIC, MPI_DISTRIBUTE_CYCLIC /)`

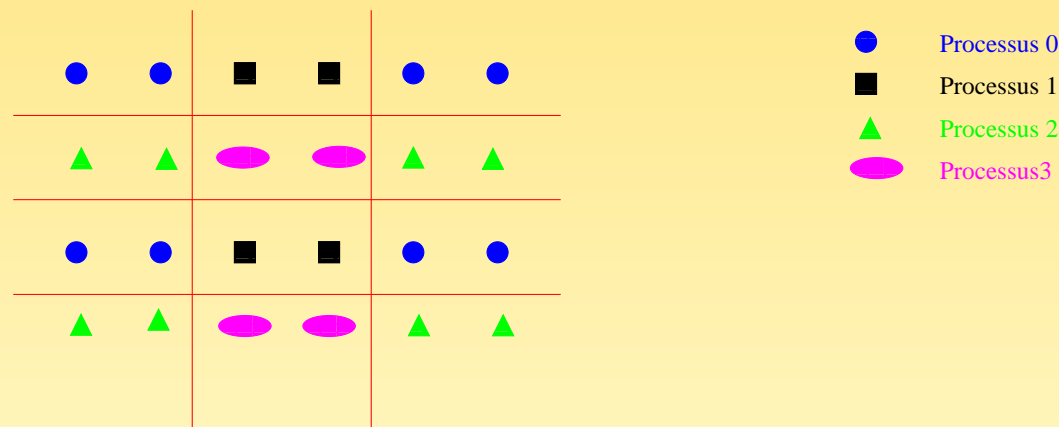


FIG. 76 – Définition du type dérivé sur chaque processus pour une distribution cyclique

```
1 program darray_cyclique
2   use mpi
3   implicit none
4   integer,parameter                :: nb_lignes=4,nb_colonnes=6,&
5                                     nb_dims=2,etiquette1=1000,etiquette2=1001
6   integer                          :: nb_procs,code,rang,i,type_cyclique
7   integer,dimension(nb_lignes,nb_colonnes) :: tab
8   integer,dimension(nb_dims)       :: profil_tab,mode_distribution,&
9                                     profil_sous_tab,distribution_procs
10  integer,dimension(MPI_STATUS_SIZE) :: statut
11
12  call MPI_INIT(code)
13  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
14  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
15
16  !Initialisation du tableau tab sur chaque processus
17  tab(:,:)=reshape((/(i*(rang+1),i=1,nb_lignes*nb_colonnes)/),(/nb_lignes,nb_colonnes/))
18
19  !Profil du tableau tab
20  profil_tab(:) = shape(tab)
21
22  !Mode de distribution
23  mode_distribution(:) = (/ MPI_DISTRIBUTE_CYCLIC,MPI_DISTRIBUTE_CYCLIC /)
24
25  !Profil d'un bloc
26  profil_sous_tab(:) = (/ 1,2 /)
```

```
27 !Nombre de processus dans chaque dimension
28 distribution_procs(:) = (/ 2,2 /)
29 !Création du type dérivé type_cyclique
30 call MPI_TYPE_CREATE_DARRAY(nb_procs,rang,nb_dims,profil_tab,mode_distribution,&
31                             profil_sous_tab, distribution_procs,MPI_ORDER_FORTRAN,&
32                             MPI_INTEGER,type_cyclique,code)
33 call MPI_TYPE_COMMIT(type_cyclique,code)
34 select case(rang)
35   case(0)
36     !Le processus 0 envoie son tableau tab au processus 2
37     call MPI_SEND(tab,1,type_cyclique,2,etiquette1,MPI_COMM_WORLD,code)
38   case(2)
39     !Le processus 2 reçoit son tableau tab du processus 0
40     call MPI_RECV(tab,1,type_cyclique,0,etiquette1,MPI_COMM_WORLD,statut,code)
41   case(1)
42     !Le processus 1 envoie son tableau tab au processus 3
43     call MPI_SEND(tab,1,type_cyclique,3,etiquette2,MPI_COMM_WORLD,code)
44   case(3)
45     !Le processus 3 reçoit son tableau tab du processus 1
46     call MPI_RECV(tab,1,type_cyclique,1,etiquette2,MPI_COMM_WORLD,statut,code)
47 end select
48 !Affichage du tableau tab sur chaque processus
49 .....
50 call MPI_TYPE_FREE(type_cyclique,code)
51 call MPI_FINALIZE(code)
52 end program darray_cyclique
```



```
> mpiexec -n 4 darray_cyclique
```

Tableau tab obtenu sur le processus 0

```
1, 5, 9, 13, 17, 21  
2, 6, 10, 14, 18, 22  
3, 7, 11, 15, 19, 23  
4, 8, 12, 16, 20, 24
```

Tableau tab obtenu sur le processus 1

```
2, 10, 18, 26, 34, 42  
4, 12, 20, 28, 36, 44  
6, 14, 22, 30, 38, 46  
8, 16, 24, 32, 40, 48
```

Tableau tab obtenu sur le processus 2

```
3, 15, 27, 39, 51, 63  
1, 5, 30, 42, 17, 21  
9, 21, 33, 45, 57, 69  
3, 7, 36, 48, 19, 23
```

Tableau tab obtenu sur le processus 3

```
4, 20, 36, 52, 68, 84  
8, 24, 18, 26, 72, 88  
12, 28, 44, 60, 76, 92  
16, 32, 22, 30, 80, 96
```

## Autres exemples

➡ `nb_procs = 4, distribution_procs(:) = (/ 2,2 /)`

➡ `profil_tab(:) = (/ 4,3 /), profil_sous_tab(:) = (/ 2,2 /)`

➡ `mode_distribution(:) = (/ MPI_DISTRIBUTE_BLOCK, MPI_DISTRIBUTE_BLOCK /)`

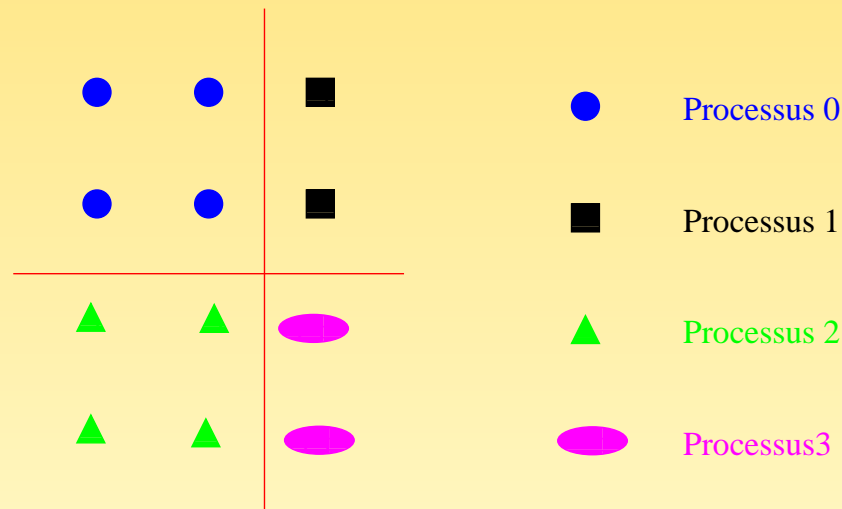


FIG. 77 – Définition du type dérivé sur chaque processus

➡ `nb_procs = 3, distribution_procs(:) = (/ 3,1 /)`

➡ `profil_tab(:) = (/ 5,5 /), profil_sous_tab(:) = (/ 1,5 /)`

➡ `mode_distribution(:) = (/ MPI_DISTRIBUTE_CYCLIC, MPI_DISTRIBUTE_NONE /)`

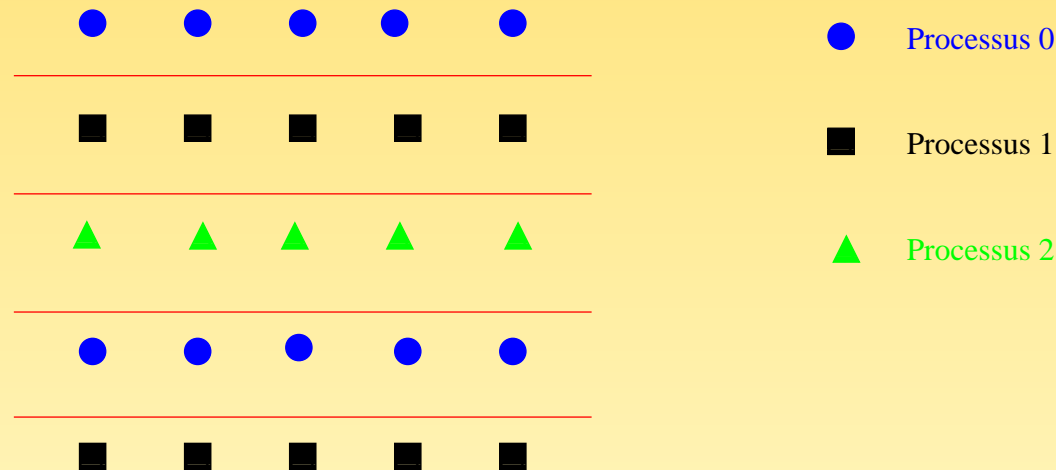


FIG. 78 – Définition du type dérivé sur chaque processus

## 11.2.2 – Types dérivés numériques

- ➡ Le langage Fortran 95 introduit deux fonctions intrinsèques `selected_int_kind()` et `selected_real_kind()` qui permettent de définir la **précision** et/ou l'**étendue** d'un nombre entier, réel ou complexe
- ➡ **MPI** devait donc assurer la portabilité de ces types de données en définissant essentiellement les sous-programmes suivants : `MPI_TYPE_CREATE_F90_INTEGER()`, `MPI_TYPE_CREATE_F90_REAL()` et `MPI_TYPE_CREATE_F90_COMPLEX()`
- ➡ Ces sous-programmes renvoient des types dérivés **MPI**

## Rappels (extrait du cours Fortran 95 de l'IDRIS)

- ➡ La fonction intrinsèque `selected_int_kind(e)` reçoit en argument un nombre entier `e` positif et retourne une valeur qui correspond au sous-type permettant de représenter les entiers `n` tels que  $-10^{+e} < n < 10^{+e}$
- ➡ La fonction intrinsèque `selected_real_kind(p,e)` admet deux arguments optionnels positifs `p` et `e` (toutefois l'un des deux doit obligatoirement être fourni) indiquant respectivement la **précision** (nombre de chiffres décimaux significatifs) et l'**étendue** (la plage des nombres représentables en machine) désirées. Elle retourne un entier correspondant au sous-type permettant de représenter les réels `x` tels que  $10^{-e} < |x| < 10^{+e}$ . (Fin de l'extrait)

**Exemple** : on souhaite représenter le nombre réel 12345.1234568 sur  $p = 8$  chiffres significatifs avec une étendue par défaut. Au mieux, ce nombre aurait la valeur  $x = 12345.123$  en machine. (Fin des rappels)

```
1 program precision
2   use mpi
3   implicit none
4   integer, parameter      :: n=101, preci=12
5   integer                 :: rang, mon_complex, code
6   ! Le sous-type k représentera une précision d'au moins 12 chiffres significatifs
7   integer, parameter      :: k=selected_real_kind(preci)
8   complex(kind=k), dimension(n) :: donnee
9
10  call MPI_INIT(code)
11  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
12
13  ! Construction du type MPI mon_complex associé à la précision demandée
14  call MPI_TYPE_CREATE_F90_COMPLEX(preci, MPI_UNDEFINED, mon_complex, code)
15
16  if (rang == 0) donnee(:) = cmplx(rang, rang, kind=k)
17
18  ! Utilisation du type mon_complex
19  call MPI_BCAST(donnee, n, mon_complex, 0, MPI_COMM_WORLD, code)
20
21  call MPI_FINALIZE(code)
22 end program precision
```

## Remarques

- ➡ En réalité, les types générés par ces sous-programmes sont prédéfinis par `MPI`
- ➡ Par conséquent, ils ne peuvent pas être libérés avec `MPI_TYPE_FREE()`
- ➡ De plus, il n'est pas nécessaire de les valider avec `MPI_TYPE_COMMIT()`

## 11.3 – Optimisations

Dans un programme, il arrive parfois que l'on soit contraint de **boucler** un certain nombre de fois **sur un envoi et une réception de message** où la valeur des données manipulées change mais pas leurs adresses en mémoire ni leurs nombres ni leurs types. En outre, l'appel à un sous-programme de communication à chaque itération peut être très **pénalisant** à la longue d'où l'**intérêt des communications persistantes**.

Elles consistent à :

- ❶ créer un schéma persistant de communication une fois pour toutes (à l'extérieur de la boucle) ;
- ❷ activer réellement la requête d'envoi ou de réception dans la boucle ;
- ❸ libérer, si nécessaire, la requête en fin de boucle.

envoi <i>standard</i>	MPI_SEND_INIT()
envoi <i>synchronous</i>	MPI_SSEND_INIT()
envoi <i>buffered</i>	MPI_BSEND_INIT()
réception <i>standard</i>	MPI_RECV_INIT()



```
23 if (rang == 0) then
24   do k = 1, 1000
25     call MPI_ISSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
26     call sgetrf(na, na, a, na, pivota, code)
27     call MPI_WAIT(requete0,statut,code)
28     c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
29   end do
30 elseif (rang == 1) then
31   do k = 1, 1000
32     call sgetrf(na, na, a, na, pivota, code)
33     call MPI_IRECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
34     call sgetrf(nb, nb, b, nb, pivotb, code)
35     call MPI_WAIT(requete1,statut,code)
36     a(:, :) = transpose(c(1:na,1:na)) + a(:, :)
37   end do
38 end if
```

```
> mpiexec -n 2 AOptimiser
```

```
Temps : 235 secondes
```

L'utilisation d'un schéma persistant de communication permet de cacher la latence et de réduire les surcoûts induits par chaque appel aux sous-programmes de communication dans la boucle. Le gain peut être important lorsque ce mode de communication est réellement implémenté.

```
23 if (rang == 0) then
24   call MPI_SSEND_INIT(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
25   do k = 1, 1000
26     call MPI_START(requete0,code)
27     call sgetrf(na, na, a, na, pivota, code)
28     call MPI_WAIT(requete0,statut,code)
29     c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
30   end do
31 elseif (rang == 1) then
32   call MPI_RECV_INIT(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
33   do k = 1, 1000
34     call sgetrf(na, na, a, na, pivota, code)
35     call MPI_START(requete1,code)
36     call sgetrf(nb, nb, b, nb, pivotb, code)
37     call MPI_WAIT(requete1,statut,code)
38     a(:, :) = transpose(c(1:na,1:na)) + a(:, :)
39   end do
40 end if
```

```
> mpiexec -n 2 AOptimiser
Temps : 235 secondes
```

Ici, l'implémentation MPI et/ou l'infrastructure matérielle de la machine ne permettent malheureusement pas une utilisation efficace du mode persistant.

## Remarques :

- ➔ Une communication activée par `MPI_START()` sur une requête créée par l'un des sous-programmes `MPI_XXXX_INIT()` est équivalente à une communication non bloquante `MPI_IXXXX()`.
- ➔ Pour redéfinir un nouveau schéma persistant avec la même requête, il faut auparavant libérer celle associée à l'ancien schéma en appelant le sous-programme `MPI_REQUEST_FREE(requete,code)`.
- ➔ Ce sous-programme ne libèrera la requête `requete` qu'une fois que la communication associée sera réellement terminée.

## 11.4 – Communicateurs

### 11.4.1 – Intra et intercommunicateurs

- ➡ Les communicateurs que nous avons construits jusqu'à présent sont des **intracommunicateurs** car ils ne permettent pas que des processus appartenant à des communicateurs distincts puissent communiquer entre eux.
- ➡ Des processus appartenant à des intracommunicateurs distincts ne peuvent communiquer que s'il existe un lien de communication entre ces intracommunicateurs.
- ➡ Un **intercommunicateur** est un communicateur qui permet l'établissement de ce lien de communication.
- ➡ Le sous-programme **MPI** `MPI_INTERCOMM_CREATE()` permet de construire des intercommunicateurs.
- ➡ Le couplage des modèles océan/atmosphère illustre bien l'utilité des intra et intercommunicateurs...

## Exemple récapitulatif sur les intra et intercommuniqueurs

## MPI\_COMM\_WORLD

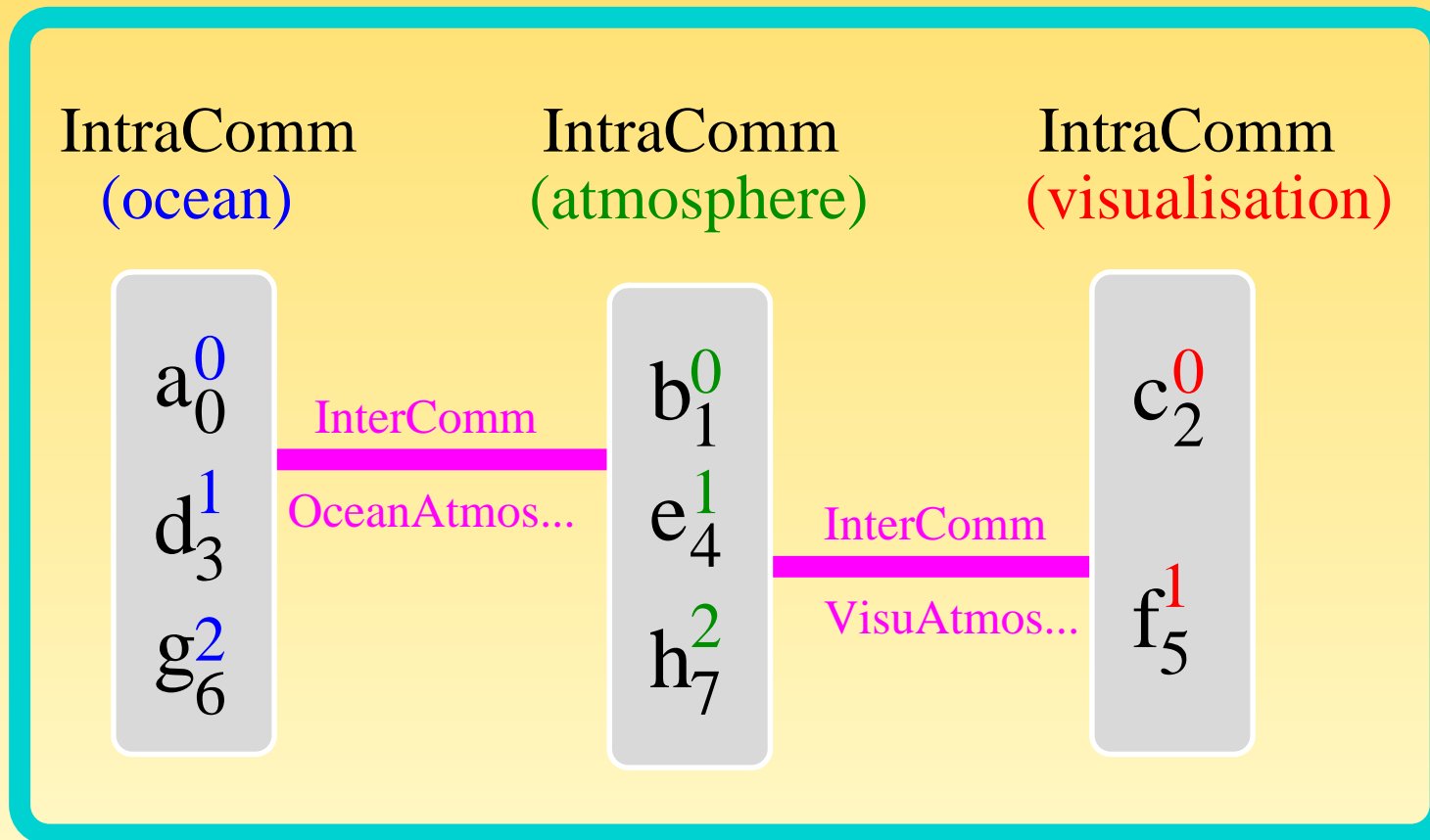


FIG. 79 – Couplage océan/atmosphère

```
1 program OceanAtmosphere
2   use mpi
3   implicit none
4
5   integer,parameter :: tag1=1111, tag2=2222
6   integer           :: RangMonde, NombreIntraComm, couleur, code, &
7                   IntraComm, CommOceanAtmosphere, CommVisuAtmosphere
8
9   call MPI_INIT(code)
10  call MPI_COMM_RANK(MPI_COMM_WORLD,RangMonde,code)
11
12  ! Construction des 3 IntraCommunicateurs
13  NombreIntraComm = 3
14  couleur = mod(RangMonde,NombreIntraComm) ! = 0,1,2
15  call MPI_COMM_SPLIT(MPI_COMM_WORLD,couleur,RangMonde,IntraComm,code)
```

```
16 ! Construction des deux InterCommunicateurs et et appel des sous-programmes de calcul
17 select case(couleur)
18   case(0)
19     ! InterCommunicateur OceanAtmosphere pour que le groupe 0 communique
20     ! avec le groupe 1
21     call MPI_INTERCOMM_CREATE (IntraComm,0,MPI_COMM_WORLD,1,tag1,CommOceanAtmosphere, &
22                               code)
23     call ocean(IntraComm,CommOceanAtmosphere)
24
25   case(1)
26     ! InterCommunicateur OceanAtmosphere pour que le groupe 1 communique
27     ! avec le groupe 0
28     call MPI_INTERCOMM_CREATE (IntraComm,0,MPI_COMM_WORLD,0,tag1,CommOceanAtmosphere, &
29                               code)
30
31     ! InterCommunicateur CommVisuAtmosphere pour que le groupe 1 communique
32     ! avec le groupe 2
33     call MPI_INTERCOMM_CREATE (IntraComm,0,MPI_COMM_WORLD,2,tag2,CommVisuAtmosphere,code)
34     call atmosphere(IntraComm,CommOceanAtmosphere,CommVisuAtmosphere)
35
36   case(2)
37     ! InterCommunicateur CommVisuAtmosphere pour que le groupe 2 communique
38     ! avec le groupe 1
39     call MPI_INTERCOMM_CREATE (IntraComm,0,MPI_COMM_WORLD,1,tag2,CommVisuAtmosphere,code)
40     call visualisation(IntraComm,CommVisuAtmosphere)
41 end select
```

```
42 subroutine ocean(IntraComm,CommOceanAtmosphere)
43   use mpi
44   implicit none
45   integer,parameter          :: n=1024,tag1=3333
46   real,dimension(n)         :: a,b,c
47   integer                    :: rang,code,germe(1),IntraComm,CommOceanAtmosphere
48   integer,dimension(MPI_STATUS_SIZE) :: statut
49   integer,intrinsic          :: irtc
50
51   ! Les processus 0, 3, 6 dédiés au modèle océanographique effectuent un calcul
52   germe(1)=irtc()
53   call random_seed(put=germe)
54   call random_number(a)
55   call random_number(b)
56   call random_number(c)
57   a(:) = b(:) * c(:)
58
59   ! Les processus impliqués dans le modèle océan effectuent une opération collective
60   call MPI_ALLREDUCE(a,c,n,MPI_REAL,MPI_SUM,IntraComm,code)
61
62   ! Rang du processus dans IntraComm
63   call MPI_COMM_RANK(IntraComm,rang,code)
64
65   ! Échange de messages avec les processus associés au modèle atmosphérique
66   call MPI_SENDRECV_REPLACE(c,n,MPI_REAL,rang,tag1,rang,tag1, &
67                             CommOceanAtmosphere,statut,code)
68
69   ! Le modèle océanographique tient compte des valeurs atmosphériques
70   a(:) = b(:) * c(:)
71 end subroutine ocean
```



```
72 subroutine atmosphere(IntraComm,CommOceanAtmosphere,CommVisuAtmosphere)
73   use mpi
74   implicit none
75
76   integer,parameter                :: n=1024,tag1=3333,tag2=4444
77   real,dimension(n)                :: a,b,c
78   integer                          :: rang,code,germe(1),IntraComm, &
79                                   CommOceanAtmosphere,CommVisuAtmosphere
80   integer,dimension(MPI_STATUS_SIZE) :: statut
81   integer,intrinsic                 :: irtc
82
83   ! Les processus 1, 4, 7 dédiés au modèle atmosphérique effectuent un calcul
84   germe(1)=irtc()
85   call random_seed(put=germe)
86
87   call random_number(a)
88   call random_number(b)
89   call random_number(c)
90
91   a(:) = b(:) + c(:)
```

```
92 ! Les processus dédiés au modèle atmosphère effectuent une opération collective
93 call MPI_ALLREDUCE(a,c,n,MPI_REAL,MPI_MAX,IntraComm,code)
94
95 ! Rang du processus dans IntraComm
96 call MPI_COMM_RANK(IntraComm,rang,code)
97
98 ! Échange de messages avec les processus dédiés au modèle océanographique
99 call MPI_SENDRECV_REPLACE(c,n,MPI_REAL,rang,tag1,rang,tag1, &
100                          CommOceanAtmosphere,statut,code)
101
102 ! Le modèle atmosphère tient compte des valeurs océanographiques
103 a(:) = b(:) * c(:)
104
105 ! Envoi des résultats aux processus dédiés à la visualisation
106 if (rang == 0 .or. rang == 1) then
107     call MPI_SSEND(a,n,MPI_REAL,rang,tag2,CommVisuAtmosphere,code)
108 end if
109
110 end subroutine atmosphere
```

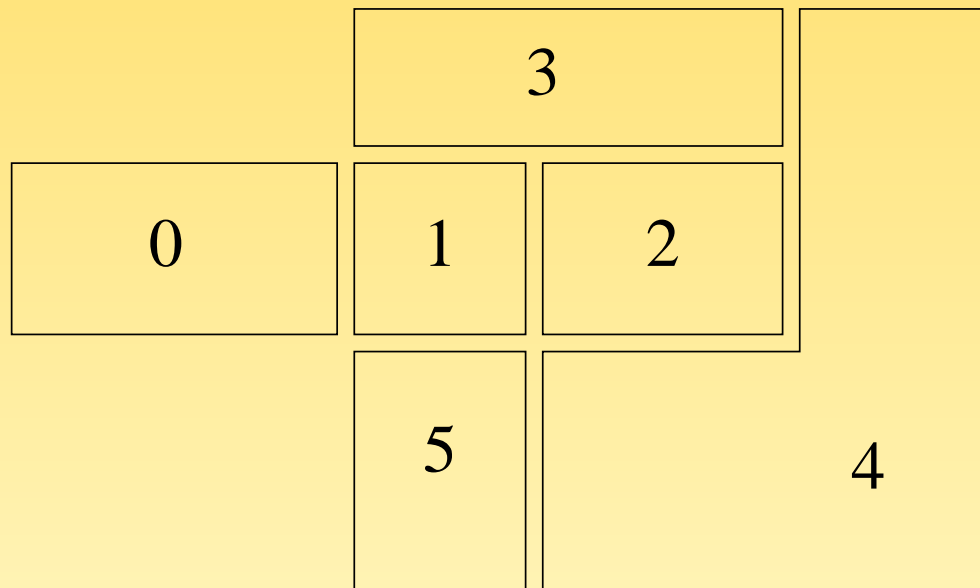
```
111 subroutine visualisation(IntraComm,CommVisuAtmosphere)
112   use mpi
113   implicit none
114
115   integer,parameter                :: n=1024,tag2=4444
116   real,dimension(n)                :: a,b,c
117   integer                           :: rang,code,IntraComm,CommVisuAtmosphere
118   integer,dimension(MPI_STATUS_SIZE) :: statut
119
120   ! Les processus 2 et 5 sont chargés de la visualisation
121   call MPI_COMM_RANK(IntraComm,rang,code)
122
123   ! Réception des valeurs du champ à tracer
124   call MPI_RECV(a,n,MPI_REAL,rang,tag2,CommVisuAtmosphere,statut,code)
125
126   print*, 'Moi, processus ',rang,' je trace mon champ A : ',a(:)
127
128 end subroutine visualisation
```

### 11.4.2 – Graphe de processus

Il arrive cependant que dans certaines applications (géométries complexes), la décomposition de domaine ne soit plus une grille régulière mais un graphe dans lequel un sous-domaine peut avoir un ou plusieurs voisins quelconques. Le sous-programme `MPI_DIST_GRAPH_CREATE()` permet alors de définir une topologie de type graphe.

```
1 integer, intent(in)           :: comm_ancien,n,info
2 integer, dimension(:),intent(in) :: source,degres
3 integer, dimension(nb_voisins_max),intent(in) :: liste_voisins,poids
4 logical, intent(in)          :: reorganisation
5
6 integer, intent(out)         :: comm_nouveau, code
7
8 call MPI_DIST_GRAPH_CREATE(comm_ancien,n,sources,degres,liste_voisins,poids,&
9                             info,reorganisation, comm_nouveau,code)
```

Les tableaux d'entiers `poids` et `liste_voisins` permettent de définir le poids attribué et la liste des voisins ceci pour chacun des nœuds.



Numéro de processus	liste_voisins
0	1
1	0,5,2,3
2	1,3,4
3	1,2,4
4	3,2,5
5	1,4

FIG. 80 – Graphe de processus

```
poids (:) = 1; liste_voisins = (/ 1, 0,5,2,3, 1,3,4, 1,2,4, 3,2,5, 1,4 /)
```

Deux autres fonctions sont utiles pour connaître :

☞ le nombre de voisins pour un processus donné :

```
integer, intent(in)           :: comm_nouveau
integer, intent(in)           :: rang
integer, intent(out)          :: nb_voisins
integer, intent(out)          :: code

call MPI_GRAPH_NEIGHBORS_COUNT(comm_nouveau,rang,nb_voisins,code)
```

☞ la liste des voisins pour un processus donné :

```
integer, intent(in)           :: comm_nouveau
integer, intent(in)           :: rang
integer, intent(in)           :: nb_voisins
integer, dimension(nb_voisins_max), intent(out) :: voisins
integer, intent(out)          :: code

call MPI_GRAPH_NEIGHBORS(comm_nouveau,rang,nb_voisins,voisins,code)
```

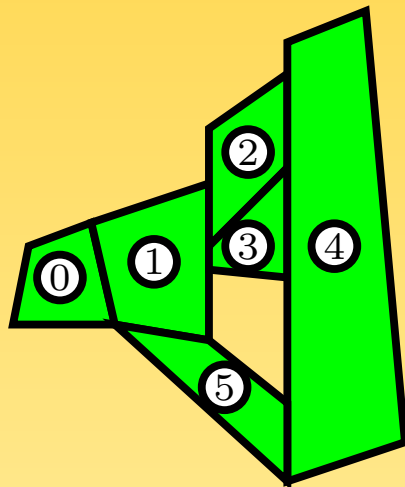
```
1 program graphe
2
3 use mpi
4 implicit none
5
6 integer                :: rang,rang_monde,code,nb_processus,comm_graphe,&
7                        n, nb_voisins,i,iteration=0
8 integer, parameter     :: etiquette=100
9 integer, dimension(16) :: liste_voisins,poids
10 integer, allocatable, dimension(:) :: voisins,sources,degres
11 integer, dimension(MPI_STATUS_SIZE) :: statut
12 real                   :: propagation, & ! Propagation du feu
13                        & ! depuis les voisins
14                        feu=0., & ! Valeur du feu
15                        bois=1., & ! Rien n'a encore brûlé
16                        arret=1. ! Tout a brûlé si arret <= 0.01
17
18 call MPI_INIT(code)
19 call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_processus,code)
20 call MPI_COMM_RANK(MPI_COMM_WORLD,rang_monde,code)
21 allocate(sources(0:nb_processus-1),degres(0:nb_processus-1))
22 if (rang_monde==0) then
23     n=nb_processus
24 else
25     n=0
26 end if
27
```

```
28
29 do i=0,nb_processus-1
30     sources(i)=i
31 end do
32
33 degres(:)= (/ 1,4,3,3,3,2 /)
34
35 liste_voisins(:)= (/ 1,0,5,2,3,1,3,4,1,2,4,3,2,5,1,4 /)
36
37 poids(:) = 1
38
39 call MPI_DIST_GRAPH_CREATE(MPI_COMM_WORLD,n,sources,degres,liste_voisins,poids,MPI_INFO_
40     .false.,comm_graphe,code)
41 call MPI_COMM_RANK(comm_graphe,rang,code)
42
43 if (rang == 2) feu=1.           ! Le feu se déclare arbitrairement sur la parcelle 2
44
45 call MPI_GRAPH_NEIGHBORS_COUNT(comm_graphe,rang,nb_voisins,code)
46
47 allocate(voisins(nb_voisins)) ! Allocation du tableau voisins
48
49 call MPI_GRAPH_NEIGHBORS(comm_graphe,rang,nb_voisins,voisins,code)
50
51
52
53
54
55
```

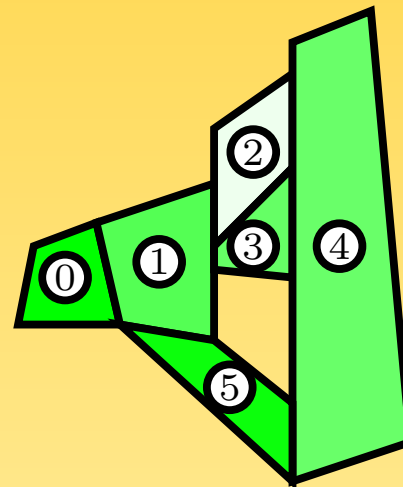


```
56 do while (arret > 0.01)           ! On arrête dès qu'il n'y a plus rien à brûler
57
58     do i=1,nb_voisins             ! On propage le feu aux voisins
59         call MPI_SENDRECV(minval((/1.,feu/)),1,MPI_REAL,voisins(i),etiquette, &
60             propagation,          1,MPI_REAL,voisins(i),etiquette, &
61             comm_graphe,statut,code)
62         ! Le feu se développe en local sous l'influence des voisins
63         feu=1.2*feu + 0.2*propagation*bois
64         bois=bois/(1.+feu)       ! On calcule ce qui reste de bois sur la parcelle
65     end do
66
67     call MPI_ALLREDUCE(bois,arret,1,MPI_REAL,MPI_SUM,comm_graphe,code)
68
69     iteration=iteration+1
70     print '("Itération ",i2," parcelle ",i2," bois=",f5.3)',iteration,rang,bois
71     call MPI_BARRIER(comm_graphe,code)
72     if (rang == 0) print '("--")'
73 end do
74
75 deallocate(voisins)
76
77 call MPI_FINALIZE(code)
78
79 end program graphe
```

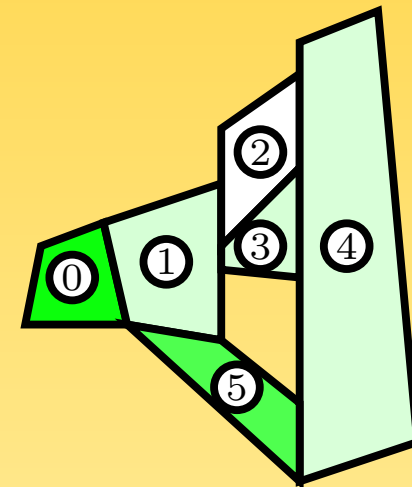
```
> mpiexec -n 6 graphe
Iteration 1 parcelle 0 bois=1.000
Iteration 1 parcelle 3 bois=0.602
Iteration 1 parcelle 5 bois=0.953
Iteration 1 parcelle 4 bois=0.589
Iteration 1 parcelle 1 bois=0.672
Iteration 1 parcelle 2 bois=0.068
--
.....
Iteration 10 parcelle 0 bois=0.008
Iteration 10 parcelle 1 bois=0.000
Iteration 10 parcelle 3 bois=0.000
Iteration 10 parcelle 5 bois=0.000
Iteration 10 parcelle 2 bois=0.000
Iteration 10 parcelle 4 bois=0.000
--
```



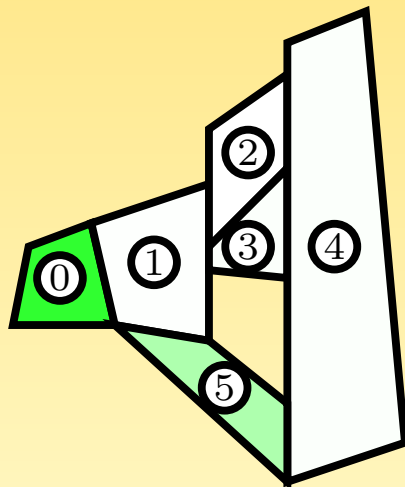
(a) Itération 0



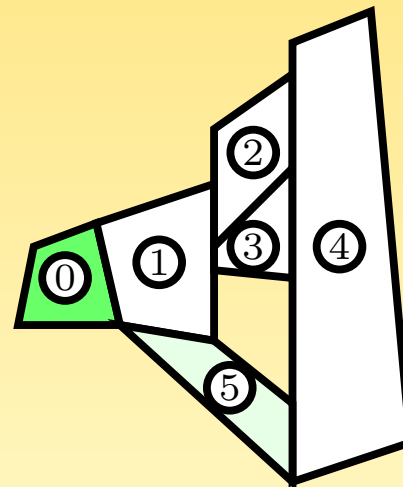
(b) Itération 1



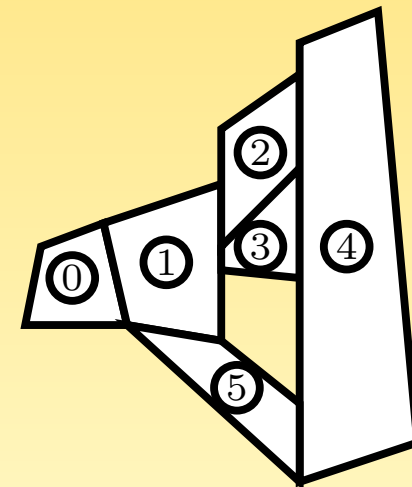
(c) Itération 2



(d) Itération 3



(e) Itération 4



(f) Itération 10

FIG. 81 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt

## 11.5 – Gestion de processus

### 11.5.1 – Introduction

- ➡ La gestion dynamique des processus est l'un des apports majeurs de [MPI-2](#)
- ➡ C'est la possibilité de créer (et dans certaines conditions de supprimer) des processus durant l'exécution de l'application
- ➡ Comme dans [MPI-1](#), le démarrage d'une application reste dépendant de l'environnement d'exécution qui sera défini par le constructeur

Dans [MPI-2](#), l'activation d'un ou plusieurs processus peut se faire selon deux modes bien distincts :

- ❶ Le mode maître-ouvriers : l'un au moins des processus d'une application active un ou plusieurs autres processus. Les processus ouvriers ainsi activés dynamiquement exécutent un code soit identique (modèle SPMD) soit différent (modèle MPMD) du processus maître qui les a générés.
- ❷ Le mode client-serveur : un ou plusieurs processus d'une application serveur (lancée au préalable) sont en attente de connexion d'un ou plusieurs processus d'une application cliente (lancée plus tard). Une fois la connexion effectuée, un lien de communication est établi entre les processus des deux applications.

## 11.5.2 – Mode maître-ouvriers

### Activation d'un programme unique

Dans l'exemple que nous allons décrire, nous suivons le modèle MPMD où un programme parallèle « maître » active, avec le sous-programme `MPI_COMM_SPAWN()`, plusieurs copies d'un programme parallèle unique « ouvriers ». Ce sous-programme est collectif. Il est bloquant pour tous les processus appartenant au communicateur incluant le processus « maître », celui qui active réellement les processus ouvriers.

Nous aurons également besoin du sous-programme `MPI_INTERCOMM_MERGE()` qui permet de fusionner dans un même intracommunicateur deux communicateurs liés par un intercommunicateur donné.

```
NEC SX-5> mpiexec -n 3 -max_np 7 maitre
```

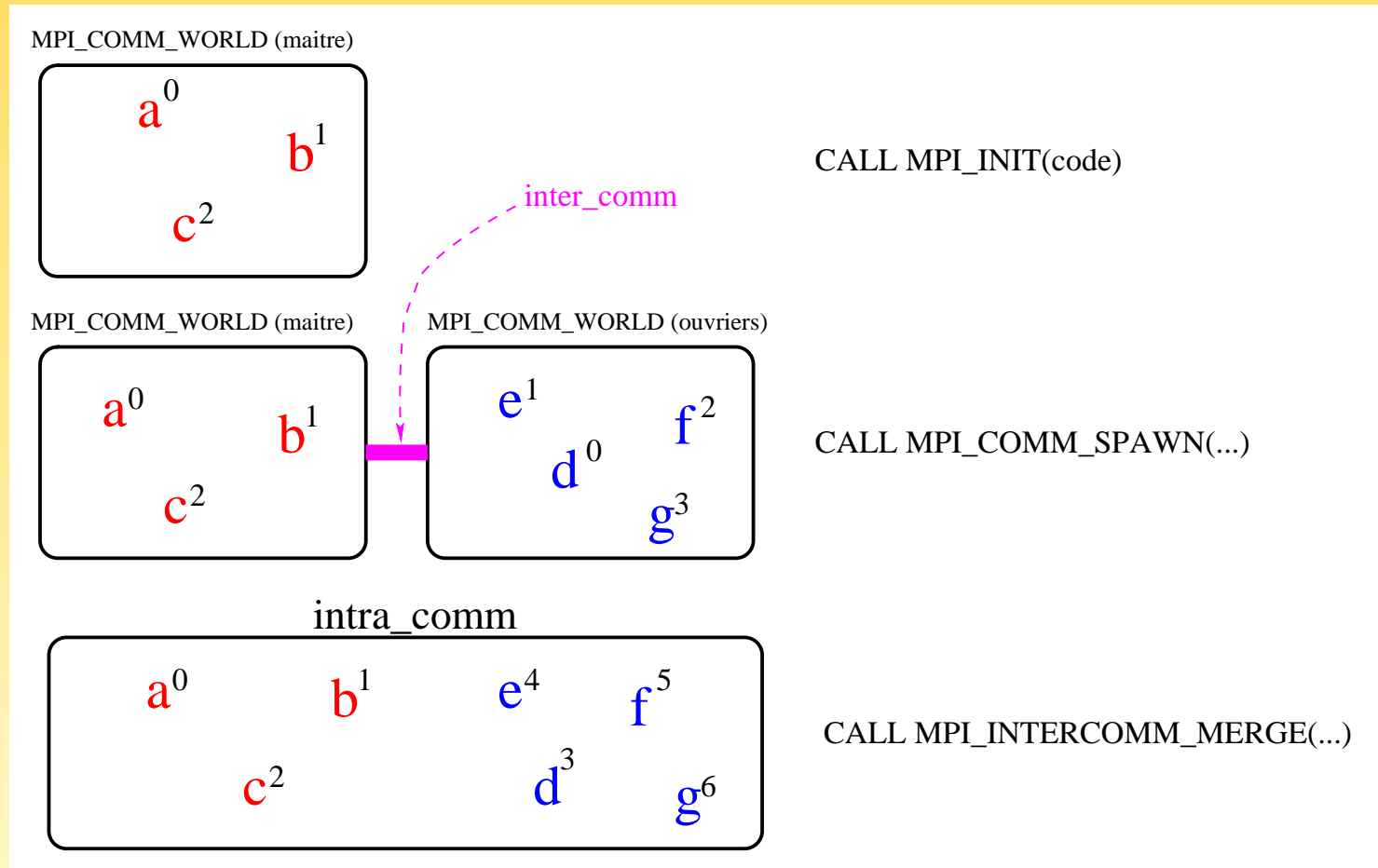


FIG. 82 – Utilisation de `MPI_COMM_SPAWN()`

```
1 program maitre
2   use mpi
3   implicit none
4
5   integer :: nb_procs_maitres,nb_procs_ouvriers=4,nb_procs,rang,code
6   integer :: inter_comm,intra_comm,rang_maitre=1
7   logical :: drapeau=.false.
8
9   call MPI_INIT(code)
10  call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_procs_maitres, code)
11
12  ! Activation des processus ouvriers
13  call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, MPI_INFO_NULL, &
14                    rang_maitre, MPI_COMM_WORLD, inter_comm, MPI_ERRCODES_IGNORE, code)
15
16  ! Fusion des communicateurs associés à inter_comm. Dans intra_comm, les rangs
17  ! des processus seront ordonnés selon la valeur de l'argument drapeau
18  call MPI_INTERCOMM_MERGE(inter_comm, drapeau, intra_comm, code)
19  call MPI_COMM_SIZE(intra_comm, nb_procs, code)
20  call MPI_COMM_RANK(intra_comm, rang, code)
21  print *, "maitre de rang ", rang, "; intra_comm de taille ", nb_procs, &
22          "; mon MPI_COMM_WORLD de taille ", nb_procs_maitres
23  call MPI_FINALIZE(code)
24 end program maitre
```



```
1 program ouvriers
2   use mpi
3   implicit none
4   integer :: nb_procs_ouvriers, nb_procs, rang, code
5   integer :: inter_comm, intra_comm
6   logical :: drapeau=.true.
7
8   call MPI_INIT(code)
9   call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_procs_ouvriers, code)
10
11  ! Ai-je un processus maître ?
12  call MPI_COMM_GET_PARENT(inter_comm, code)
13  if (inter_comm == MPI_COMM_NULL) then
14    print *, 'Pas de processus maître'
15    call MPI_FINALIZE(code)
16    stop
17  end if
18
19  ! Fusion des communicateurs associés à inter_comm. Dans intra_comm, les rangs
20  ! des processus seront ordonnés selon la valeur de l'argument drapeau
21  call MPI_INTERCOMM_MERGE(inter_comm, drapeau, intra_comm, code)
22  call MPI_COMM_SIZE(intra_comm, nb_procs, code)
23  call MPI_COMM_RANK(intra_comm, rang, code)
24  print *, "ouvrier de rang ", rang, "; intra_comm de taille ", nb_procs, &
25    "; mon MPI_COMM_WORLD de taille : ", nb_procs_ouvriers
26  call MPI_FINALIZE(code)
27 end program ouvriers
```

```
NEC SX-5> mpiexec -n 3 -max_np 7 maitre
maitre de rang 0 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 3
maitre de rang 2 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 3
ouvrier de rang 5 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
ouvrier de rang 4 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
ouvrier de rang 6 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
maitre de rang 1 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 3
ouvrier de rang 3 ; intra_comm de taille 7 ; mon MPI_COMM_WORLD de taille 4
```

Noter que, dans ce cas, la fusion des communicateurs ne modifie pas le rang des processus associés au programme maître.

## Signification de MPI\_COMM\_SELF

`MPI_COMM_SELF` est un communicateur prédéfini par `MPI`. À l'appel de `MPI_COMM_SPAWN()`, ce communicateur inclut un et un seul processus. Ce processus est celui qui active les processus ouvriers. `MPI_COMM_SELF` n'inclura donc que le processus maître.

```
...
! Activation des processus ouvriers
rang_maitre=1
nb_procs_ouvriers=4
call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, MPI_INFO_NULL, &
                    rang_maitre, MPI_COMM_SELF, inter_comm, MPI_ERRCODES_IGNORE, code)
...
```

```
NEC SX-5> mpiexec -n 3 -max_np 7 maitre
```

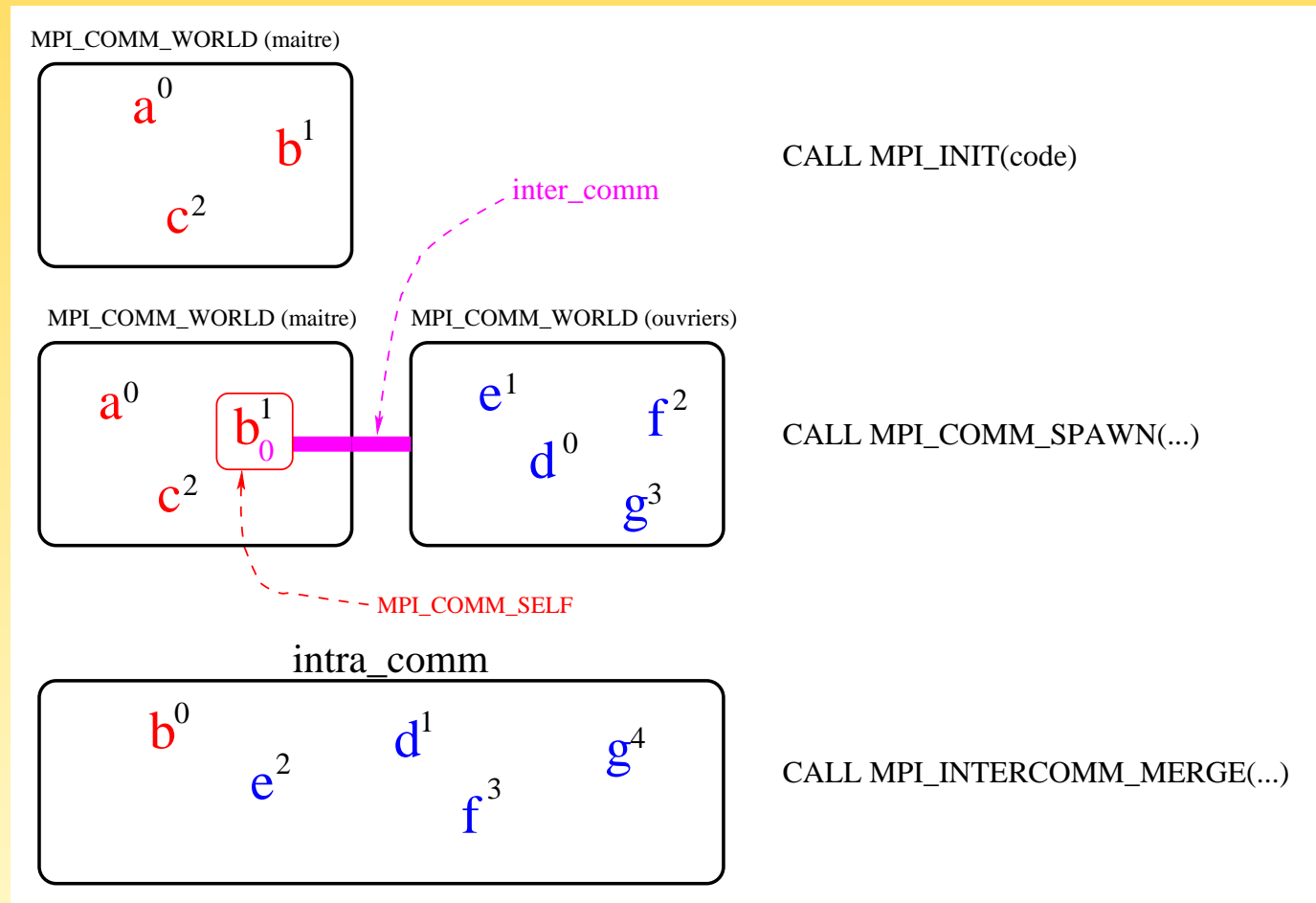


FIG. 83 – Signification de `MPI_COMM_SELF`

## Signification de `MPI_INFO_NULL`

- ➡ Ce paramètre est aussi utilisé dans d'autres contextes, notamment dans les entrées/sorties parallèles avec `MPI-IO` que l'on verra dans le chapitre 9
- ➡ S'il est spécifié à l'appel du sous-programme `MPI_COMM_SPAWN()` (ou bien `MPI_COMM_SPAWN_MULTIPLE()` que l'on introduira par la suite), il indique le mode de recherche **par défaut** des programmes « ouvriers ». Les constructeurs peuvent toutefois définir d'autres valeurs plus spécifiques à leur environnement.
- ➡ Le mode de recherche par défaut voudra dire généralement que les programmes « ouvriers » se trouvent sur la machine locale et dans le même répertoire que le programme « maître »
- ➡ Pour modifier ces valeurs par défaut, il faut utiliser les sous-programmes `MPI_INFO_CREATE()`, `MPI_INFO_SET()` et `MPI_INFO_FREE()`

```
integer :: rang_maitre=1, nb_procs_ouvriers=4, info_spawn
...
! Redéfinition du mode de recherche des programmes ouvriers
call MPI_INFO_CREATE(info_spawn, code)
call MPI_INFO_SET(info_spawn, "host", "aleph.idris.fr", code)
call MPI_INFO_SET(info_spawn, "wdir", "/workdir/idris/rech/rgrp001", code)

! Activation des processus ouvriers
call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, info_spawn, &
                   rang_maitre, MPI_COMM_SELF, inter_comm, MPI_ERRCODES_IGNORE, code)

! Libération du paramètre info_spawn
call MPI_INFO_FREE(info_spawn, code)
...
```

## Signification de MPI\_UNIVERSE\_SIZE

`MPI_UNIVERSE_SIZE` est une clef `MPI-2` dont on peut connaître la valeur grâce au sous-programme `MPI_COMM_GET_ATTR()`. Si la version de `MPI-2` utilisée l'implémente, il est associé au nombre total de processus qu'un utilisateur peut activer.

```
...
! Nombre de processus maximum que l'on peut activer
call MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, nb_procs_total, logique, code)

if (logique) then
  ! Ici nb_procs_ouvriers vaudra 7-3=4
  nb_procs_ouvriers = nb_procs_total - nb_procs_maitres
else
  print *, "MPI_UNIVERSE_SIZE n'est pas supporté"
  nb_procs_ouvriers = 4
end if

! Activation des processus ouvriers
rang_maitre=1
call MPI_COMM_SPAWN("ouvriers", MPI_ARGV_NULL, nb_procs_ouvriers, MPI_INFO_NULL, &
                   rang_maitre, MPI_COMM_WORLD, inter_comm, MPI_ERRCODES_IGNORE, code)
...
```

## Activation de programmes multiples

Dans ce second exemple, nous suivons le modèle MPMD où un programme parallèle « maître » active avec le sous-programme `MPI_COMM_SPAWN_MULTIPLE()` plusieurs copies de 4 programmes parallèles différents « ouvriers1 », ..., « ouvriers4 ». Ce sous-programme est collectif. Il est bloquant pour tous les processus appartenant au communicateur incluant le processus « maître », celui qui active réellement l'ensemble des processus ouvriers.

Dans ce cas, pour des raisons de performance, il est conseillé de ne pas appeler le sous-programme `MPI_COMM_SPAWN()` autant de fois qu'il y a de programmes ouvriers mais plutôt d'appeler le sous-programme `MPI_COMM_SPAWN_MULTIPLE()` une seule fois pour activer l'ensemble des programmes ouvriers.



```
1 program maitre
2   use mpi
3   implicit none
4
5   integer :: inter_comm,intra_comm, rang_maitre=1,code
6   logical :: drapeau=.false.
7   ! On souhaite activer 4 programmes ouvriers
8   integer, parameter :: nb_prog_ouvriers=4
9   character(len=12), dimension(nb_prog_ouvriers) :: ouvriers
10  integer, dimension(nb_prog_ouvriers) :: nb_procs_ouvriers=(/3,2,1,2/),infos
11  ! Un code d'erreur par programme et par processus activé
12  integer, dimension(8) :: codes_erreurs ! 8=3+2+1+2
13
14  call MPI_INIT(code)
15  ouvriers(:) = (/ "ouvriers1", "ouvriers2", "ouvriers3", "ouvriers4" /)
16  infos(:) = MPI_INFO_NULL
17  codes_erreurs(:) = MPI_ERRCODES_IGNORE
18  ! Activation de plusieurs programmes ouvriers
19  call MPI_COMM_SPAWN_MULTIPLE(nb_prog_ouvriers,ouvriers,MPI_ARGVS_NULL, &
20                               nb_procs_ouvriers,infos,rang_maitre,MPI_COMM_WORLD, &
21                               inter_comm,codes_erreurs,code)
22  ! Fusion des communicateurs associés à inter_comm. Dans intra_comm, les rangs
23  ! des processus seront ordonnés selon la valeur de l'argument drapeau
24  call MPI_INTERCOMM_MERGE(inter_comm, drapeau, intra_comm, code)
25
26  ! Inclure ici le code correspondant aux calculs à faire par les processus maîtres
27  ...
28  call MPI_FINALIZE(code)
29 end program maitre
```

```
NEC SX-5> mpiexec -n 3 -max_np 11 maitre
```

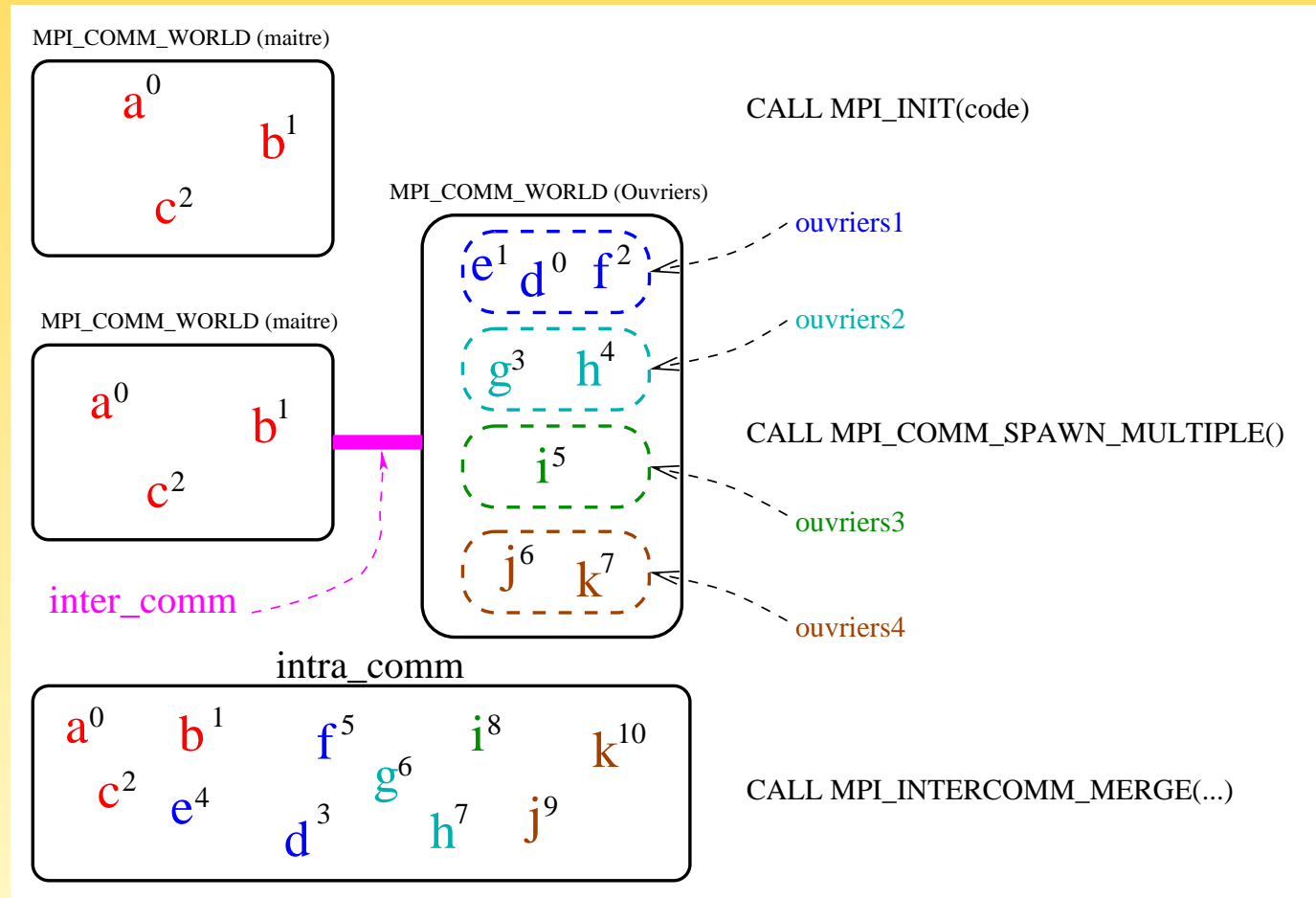


FIG. 84 – Utilisation de `MPI_COMM_SPAWN_MULTIPLE()`

## Remarques

- ➡ `MPI_COMM_SPAWN()` et `MPI_COMM_SPAWN_MULTIPLE()` sont des sous-programmes collectifs qui doivent être appelés par l'ensemble des processus du communicateur incluant le processus maître
- ➡ Attention à l'ordre des processus dans le nouvel intracommunicateur issu de la fusion des deux communicateurs associés à l'intercommunicateur renvoyé par `MPI_COMM_SPAWN()` ou `MPI_COMM_SPAWN_MULTIPLE()`
- ➡ Contrairement à ce que l'on aurait obtenu si `MPI_COMM_SPAWN()` avait été utilisé pour activer plusieurs programmes, `MPI_COMM_SPAWN_MULTIPLE()` inclut tous les processus de tous les programmes ouvriers dans le même communicateur `MPI_COMM_WORLD`

- ➡ Tous les arguments de `MPI_COMM_SPAWN_MULTIPLE()` ont la même signification que ceux de `MPI_COMM_SPAWN()`
- ➡ Dans `MPI_COMM_SPAWN_MULTIPLE()`, certains arguments sont toutefois transformés en tableaux du fait de la multiplicité des programmes ouvriers à activer
- ➡ Avec `MPI_COMM_SPAWN_MULTIPLE()`, les variables `MPI_INFO_NULL`, `MPI_COMM_SELF` et `MPI_UNIVERSE_SIZE` conservent les mêmes caractéristiques que celles que l'on a vues avec `MPI_COMM_SPAWN()`

## 11.5.3 – Mode client-serveur

Deux programmes indépendants peuvent établir entre eux un lien de communication alors que leurs processus ne partagent aucun communicateur. Cette situation peut se produire :

- ➡ lorsque deux parties d'une application démarrent indépendamment l'une de l'autre et veulent, à un moment de leur vie, échanger des informations ;
- ➡ lorsqu'une application parallèle serveur accepte des connexions de plusieurs applications parallèles clientes ;
- ➡ lorsqu'un outil de visualisation veut s'attacher à un processus en cours d'exécution pour extraire certaines informations.

L'environnement (machines, systèmes d'exploitation, etc.) dans lequel s'exécute l'application serveur peut être différent de celui des applications clientes.

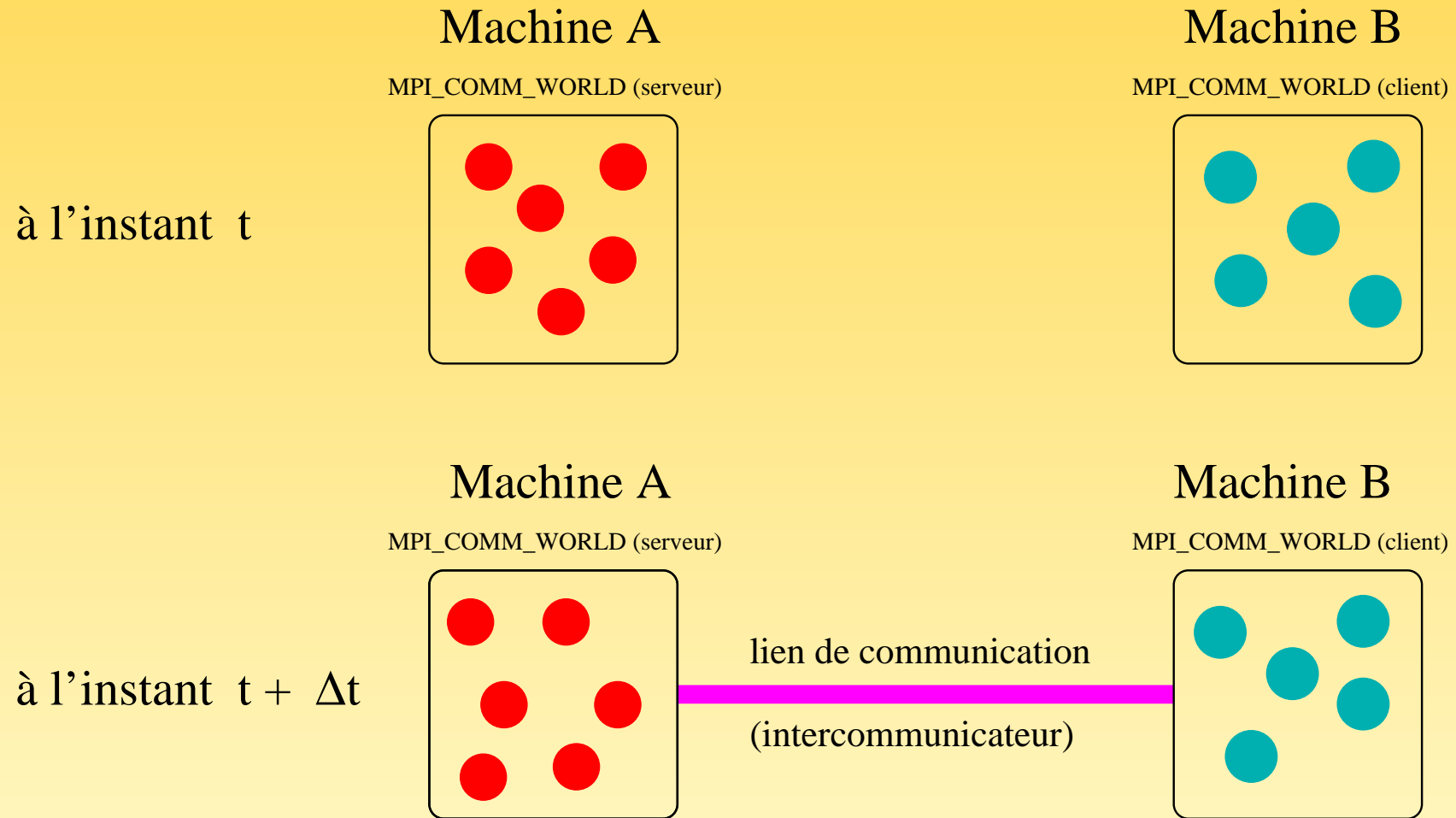


FIG. 85 – Schéma d'application client-serveur

## Processus serveur

Pour accepter un lien de communication avec le processus client, le processus serveur passe par trois étapes :

- ❶ ouverture d'un port de communication : `MPI_OPEN_PORT()` ;
- ❷ publication d'un nom arbitraire de connexion : `MPI_PUBLISH_NAME()` ;
- ❸ acceptation de la connexion : `MPI_COMM_ACCEPT()`

Pour fermer ce lien de communication, de même :

- ❶ fermeture de la connexion avec le processus client : `MPI_COMM_DISCONNECT()` ;
- ❷ retrait du nom de connexion : `MPI_UNPUBLISH_NAME()` ;
- ❸ fermeture du port de communication : `MPI_CLOSE_PORT()`.

Le processus serveur exécutera la séquence de code suivante :

```
...
integer                :: rang_serveur=2, inter_comm, code
character(len=MPI_MAX_PORT_NAME) :: nom_de_port

...
if ( rang == rang_serveur ) then
    call MPI_OPEN_PORT(MPI_INFO_NULL, nom_de_port, code)
    call MPI_PUBLISH_NAME("nom_de_connexion", MPI_INFO_NULL, nom_de_port, code)
end if

call MPI_COMM_ACCEPT(nom_de_port, MPI_INFO_NULL, rang_serveur, MPI_COMM_WORLD, &
                    inter_comm, code)

! Inclure ici le code du serveur
...
call MPI_COMM_DISCONNECT(inter_comm, code)

if ( rang == rang_serveur ) then
    call MPI_UNPUBLISH_NAME("nom_de_connexion", MPI_INFO_NULL, nom_de_port, code)
    call MPI_CLOSE_PORT(nom_de_port, code)
end if
...
```



## Processus client

Le client doit tout d'abord se connecter au port de communication du serveur, ce qui se réalise en deux étapes :

- ① recherche du port de communication associé au nom publié par le serveur :

```
MPI_LOOKUP_NAME();
```

- ② connexion avec le serveur : `MPI_COMM_CONNECT()`.

Ensuite, pour interrompre la connexion avec le serveur, le client devra obligatoirement appeler le sous-programme `MPI_COMM_DISCONNECT()`.

```
...
integer                :: rang_client=1, inter_comm, code
character(len=MPI_MAX_PORT_NAME) :: nom_de_port

...
if ( rang == rang_client ) &
    call MPI_LOOKUP_NAME("nom_de_connexion", MPI_INFO_NULL, nom_de_port, code)

call MPI_COMM_CONNECT(nom_de_port, MPI_INFO_NULL, rang_client, MPI_COMM_WORLD, &
    inter_comm, code)

! Inclure ici le code du client
...

call MPI_COMM_DISCONNECT(inter_comm, code)
...
```

## Remarques

- ➔ `MPI_COMM_CONNECT()`, `MPI_COMM_ACCEPT()` et `MPI_COMM_DISCONNECT()` sont des sous-programmes collectifs (donc bloquants), bien qu'un seul processus participe à la connexion de part et d'autre
- ➔ `MPI_CLOSE_PORT()` libère le port de communication (le serveur devient injoignable) alors que `MPI_COMM_DISCONNECT()` ne fait que rompre le lien de communication entre deux intracommunicateurs pour qu'éventuellement un autre lien puisse s'établir sur le même port
- ➔ `MPI_COMM_SELF` peut être utilisé à la place de `MPI_COMM_WORLD` dans les appels aux sous-programmes `MPI_COMM_ACCEPT()` et `MPI_COMM_CONNECT()`. Dans ce cas, la connexion s'établit entre deux intracommunicateurs ne contenant chacun que le processus appelant l'un ou l'autre sous-programme.
- ➔ Sans le mécanisme des sous-programmes `MPI_PUBLISH_NAME()` et `MPI_LOOKUP_NAME()`, on aurait été amené à préciser explicitement au processus client par un moyen quelconque (sur l'entrée standard ou par l'intermédiaire d'un fichier), le nom du port de communication renvoyé par le processus serveur

## 11.5.4 – Suppression de processus

- ☞ S'il est possible de créer des processus, on devrait pouvoir les supprimer
- ☞ Or, il n'existe pas de sous-programme **MPI** spécifique pour supprimer un processus généré en cours d'exécution
- ☞ En revanche, il est toujours possible de diriger (ex. par échange de messages) l'exécution de ce processus vers une « terminaison normale »
- ☞ Un processus **MPI** se termine normalement à l'appel du sous-programme **MPI\_FINALIZE()** et à la fin de l'exécution du programme principal

☞ Il existe trois contraintes :

- ❶ le nouveau communicateur `MPI_COMM_WORLD` généré ne doit contenir que le processus dont on veut se débarrasser ;
- ❷ il ne doit exister aucun lien de communication (intercommunicateur) entre le communicateur `MPI_COMM_WORLD` contenant le processus père (ou serveur) et celui contenant le processus fils (ou client) à supprimer ;
- ❸ tout intracommunicateur contenant le processus à détruire doit être invalidé avant la terminaison du processus fils (ou client).

☞ Il n'est pas possible de se débarrasser d'un seul processus « ouvrier » si son communicateur `MPI_COMM_WORLD` inclut d'autres processus

☞ Dans ce cas, la terminaison ne s'effectue « proprement » que si tous les processus de `MPI_COMM_WORLD` appellent le sous-programme `MPI_FINALIZE()` et atteignent normalement la fin de l'exécution

## 11.5.5 – Compléments

➡ Dans certains cas, comme celui de `MPI_UNIVERSE_SIZE`, les implémentations ont des clefs spécifiques dont la valeur peut être connue grâce au sous-programme :

```
integer, intent(in)                :: comm, clef
integer(kind=MPI_ADDRESS_KIND), intent(out) :: valeur
logical, intent(out)              :: logique
integer, intent(out)              :: code

call MPI_COMM_GET_ATTR(comm, clef, valeur, logique, code)
```

➡ On peut cependant modifier la valeur associée à une clef définie au préalable, en utilisant le sous-programme :

```
integer, intent(inout)            :: comm
integer, intent(in)              :: clef
integer(kind=MPI_ADDRESS_KIND), intent(in) :: valeur
integer, intent(out)             :: code

call MPI_COMM_SET_ATTR(comm, clef, valeur, code)
```

➡ Plus généralement, on peut définir un couple (clef, valeur) spécifique à son application par l'intermédiaire des sous-programmes `MPI_COMM_CREATE_KEYVAL()` et `MPI_COMM_SET_ATTR()`

## 11.6 – MPI-IO

Récupérer des informations spécifiques sur le fichier.

```
program open02
  use mpi
  implicit none
  integer          :: rang,descripteur,attribut,longueur,code
  character(len=80) :: libelle
  logical          :: defini
  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
  call MPI_FILE_OPEN(MPI_COMM_WORLD,"fichier.txt",MPI_MODE_RDWR + MPI_MODE_CREATE, &
                    MPI_INFO_NULL,descripteur,code)
  call MPI_FILE_GET_INFO(descripteur,attribut,code)
  call MPI_INFO_GET_VALUELEN(attribut,"cb_nodes",longueur,defini,code)
  if (defini) then
    call MPI_INFO_GET(attribut,"cb_nodes",longueur,libelle,defini,code)
    if (rang==0) print *,"Fichier 'fichier.txt' sur ",libelle(1:longueur)," processus"
  end if
  call MPI_INFO_FREE(attribut,code)
  call MPI_FILE_CLOSE(descripteur,code)
  call MPI_FINALIZE(code)
end program open02
```

```
> mpiexec -n 2 open02
```

```
Fichier 'fichier.txt' sur 2 processus
```

barrière .....	42
bloquantes (communications) .....	25, 31, 36, 137, 139, 144, 147, 148
non-bloquantes (communications) .....	275
collectives (communications) .....	13
communicateur .....	13, 21, 22, 25, 40, 152, 153, 156, 158–161, 166, 181
intercommunicateur .....	276
intracommunicateur .....	276
communication .....	13, 131, 134, 135, 137–139, 142, 144, 147–149, 151–153, 158, 272, 275, 276
contexte de communication .....	25, 158
envoi .....	137, 138, 140–142, 147, 272
étiquette .....	25, 40
groupe .....	60, 158, 159
intercommunicateur .....	276
intracommunicateur .....	276
message .....	9, 11–13, 136, 139, 140, 142, 148, 156
MPMD .....	10

optimisation .....	131
performances .....	159
persistantes (communications) .....	272, 275
portabilité .....	109, 130
processeur .....	9
processus ... 9–13, 21, 22, 25, 29, 36, 37, 40, 60, 67, 130, 152, 156, 158–160, 164–166, 171, 172, 174, 176, 276	
rang .....	22, 25, 29, 159, 160, 172, 174, 176
réception .....	137, 138, 140, 141, 147, 272
requête .....	272, 275
SPMD .....	10
surcoût .....	142
topologie .....	13, 130, 164–166, 171, 172, 174, 176, 181, 183
types dérivés .....	130



mpi .....	20
mpi.h .....	20
MPI_ADDRESS_KIND .....	75, 79, 87, 94, 101, 111, 123, 124, 126, 127, 231, 317
MPI_ANY_SOURCE .....	29, 33
MPI_ANY_TAG .....	29, 33
MPI_BYTE .....	229
MPI_CHARACTER .....	124
MPI_COMM_SELF .....	299, 300, 308, 314
MPI_COMM_WORLD ...	21–23, 26, 30–33, 36, 38, 42, 44, 46, 49, 52, 55, 56, 58, 63, 65, 75, 87, 103–108, 114, 115, 120, 121, 124, 125, 127, 128, 132, 133, 143, 145, 146, 153, 162, 163, 167, 169, 179, 180, 185, 196, 202, 204, 207, 209, 212, 215, 217, 219, 221, 223, 226, 231, 232, 234, 235, 238, 242, 245, 248, 253, 259, 260, 263, 264, 270, 273, 274, 278, 279, 287, 288, 296–298, 303, 305, 307, 312–314, 316, 318
MPI_COMPLEX .....	97, 253
MPI_DISTRIBUTE_BLOCK .....	256–258, 266
MPI_DISTRIBUTE_CYCLIC .....	256, 262, 267

MPI_DISTRIBUTE_NONE .....	256, 257, 267
MPI_DOUBLE_PRECISION .....	75, 87–89, 133
MPI_INFO_NULL .....	308
MPI_INTEGER 26, 30, 31, 33, 36, 38, 44, 63, 65, 79, 97, 101, 109, 121, 124, 127, 202, 204, 207, 209, 212, 215, 217, 219, 221, 223, 226, 232, 235, 238, 239, 243, 245, 248, 260, 264	
MPI_LOCK_EXCLUSIVE .....	93
MPI_LOCK_SHARED .....	93
MPI_LOGICAL .....	124
MPI_MAX .....	80, 133, 282
MPI_MODE_NOPRECEDE .....	82
MPI_MODE_NOPUT .....	82
MPI_MODE_NOSTORE .....	82
MPI_MODE_NOSUCCEED .....	82
MPI_MODE_SEQUENTIAL .....	201, 206
MPI_ORDER_C .....	118

MPI_ORDER_FORTRAN .....	118
MPI_PROC_NULL .....	29, 180
MPI_PROD .....	65, 80
MPI_REAL 46, 49, 52, 56, 58, 97, 99–101, 103, 105, 107, 109, 115, 124, 128, 133, 143, 146, 159, 163, 185, 273, 274, 280, 282, 283, 289	
MPI_REPLACE .....	80
MPI_SEEK_CUR .....	225
MPI_SEEK_END .....	225
MPI_SEEK_SET .....	225
MPI_STATUS_IGNORE .....	29
MPI_STATUS_SIZE . 26, 32, 38, 103, 105, 107, 114, 120, 124, 127, 132, 145, 202, 204, 207, 209, 212, 215, 217, 219, 221, 223, 226, 231, 234, 238, 242, 245, 248, 259, 263, 280, 281, 283, 287	
MPI_SUCCESS .....	29
MPI_SUM .....	63, 80, 89, 280, 289

MPL_UNDEFINED .....	160, 270
MPL_UNIVERSE_SIZE .....	303, 308, 317
MPL_WIN_BASE .....	74
MPL_WIN_DISP_UNIT .....	74
MPL_WIN_SIZE .....	74
mpif.h .....	20

MPI_ACCUMULATE .....	77, 80, 84, 90, 91
MPI_ALLGATHER .....	41, 52, 67
MPI_ALLGATHERV .....	67
MPI_ALLOC_MEM .....	95
MPI_ALLREDUCE .....	41, 60, 65, 280, 282, 289
MPI_ALLTOALL .....	41, 58, 67
MPI_ALLTOALLV .....	67
MPI_ALLTOALLW .....	67
MPI_BARRIER .....	41, 42, 289
MPI_BCAST .....	41, 44, 60, 159, 163, 270
MPI_CART_COORDS .....	174, 175, 180, 185
MPI_CART_CREATE .....	158, 166, 167, 169, 180, 185
MPI_CART_RANK .....	172, 173
MPI_CART_SHIFT .....	176–178, 180
MPI_CART_SUB .....	158, 183, 185
MPI_CLOSE_PORT .....	311, 314
MPI_COMM_ACCEPT .....	311, 314
MPI_COMM_CONNECT .....	313, 314

MPI_COMM_CREATE .....	158
MPI_COMM_CREATE_KEYVAL .....	317
MPI_COMM_DISCONNECT .....	311, 313, 314
MPI_COMM_DUP .....	158
MPI_COMM_FREE .....	158, 163
MPI_COMM_GET_ATTR .....	303
MPI_COMM_RANK .22, 23, 26, 30, 32, 38, 44, 46, 49, 52, 55, 58, 63, 65, 75, 87, 103, 105, 107, 114, 120, 124, 127, 132, 145, 162, 180, 185, 202, 204, 207, 209, 212, 215, 217, 219, 221, 223, 226, 231, 234, 238, 242, 245, 248, 253, 259, 263, 270, 278, 280, 282, 283, 287, 288, 296, 297, 318	
MPI_COMM_SET_ATTR .....	317
MPI_COMM_SIZE ..... 22, 23, 32, 38, 46, 49, 52, 55, 58, 63, 65, 179, 259, 263, 287, 296, 297	
MPI_COMM_SPAWN .....	294, 295, 299, 301, 304, 307, 308
MPI_COMM_SPAWN_MULTIPLE .....	301, 304, 306–308
MPI_COMM_SPLIT .....	158, 160, 161, 163, 183, 278
MPI_DIMS_CREATE .....	171, 179
MPI_DIST_GRAPH_CREATE .....	284
MPI_EXSCAN .....	67
MPI_FILE_GET_POSITION .....	225

MPI_FILE_GET_POSITION_SHARED .....	225
MPI_FILE_IREAD .....	199
MPI_FILE_IREAD_AT .....	198
MPI_FILE_IREAD_SHARED .....	199
MPI_FILE_IWRITE .....	199
MPI_FILE_IWRITE_AT .....	198
MPI_FILE_IWRITE_SHARED .....	199
MPI_FILE_READ .....	199
MPI_FILE_READ_ALL .....	199
MPI_FILE_READ_ALL_BEGIN .....	199
MPI_FILE_READ_ALL_END .....	199
MPI_FILE_READ_AT .....	198
MPI_FILE_READ_AT_ALL .....	198
MPI_FILE_READ_AT_ALL_BEGIN .....	198
MPI_FILE_READ_AT_ALL_END .....	198
MPI_FILE_READ_ORDERED .....	199
MPI_FILE_READ_ORDERED_BEGIN .....	199
MPI_FILE_READ_ORDERED_END .....	199

MPI_FILE_READ_SHARED .....	199
MPI_FILE_SEEK .....	225
MPI_FILE_SEEK_SHARED .....	225
MPI_FILE_SET_INFO .....	194
MPI_FILE_WRITE .....	199
MPI_FILE_WRITE_ALL .....	199
MPI_FILE_WRITE_ALL_BEGIN .....	199
MPI_FILE_WRITE_ALL_END .....	199
MPI_FILE_WRITE_AT .....	198
MPI_FILE_WRITE_AT_ALL .....	198
MPI_FILE_WRITE_AT_ALL_BEGIN .....	198
MPI_FILE_WRITE_AT_ALL_END .....	198
MPI_FILE_WRITE_ORDERED .....	199
MPI_FILE_WRITE_ORDERED_BEGIN .....	199
MPI_FILE_WRITE_ORDERED_END .....	199
MPI_FILE_WRITE_SHARED .....	199
MPI_FINALIZE 20, 23, 26, 30, 33, 38, 44, 46, 49, 52, 56, 58, 63, 65, 76, 89, 104, 106, 108, 115, 121, 125, 128, 133, 153, 163, 180, 185, 196, 202, 204, 207, 209, 212, 215, 217, 219, 221, 223, 226, 232, 235, 239, 243, 245, 248, 253, 260, 264, 270, 289, 296, 297, 305, 315, 316, 318	



MPI_GATHER .....	41, 49, 67
MPI_GATHERV .....	56, 67
MPI_GET .....	77, 80, 91
MPI_GET_ADDRESS .....	122, 123, 125
MPI_GRAPH_NEIGHBORS .....	286, 288
MPI_GRAPH_NEIGHBORS_COUNT .....	286, 288
MPI_INFO_CREATE .....	301
MPI_INFO_FREE .....	301
MPI_INFO_SET .....	301
MPI_INIT . 20, 23, 26, 30, 32, 38, 44, 46, 49, 52, 55, 58, 63, 65, 75, 87, 103, 105, 107, 114, 120, 124, 127, 132, 145, 153, 162, 179, 185, 196, 202, 204, 207, 209, 212, 215, 217, 219, 221, 223, 226, 231, 234, 238, 242, 245, 248, 253, 259, 263, 270, 278, 287, 296, 297, 305, 318	
MPI_INTERCOMM_CREATE .....	276, 279
MPI_INTERCOMM_MERGE .....	294
MPI_IPROBE .....	147
MPI_Irecv .....	144, 146–148, 273
MPI_Isend .....	144, 146, 148, 273
MPI_IxSEND .....	147
MPI_LOOKUP_NAME .....	313, 314

MPI_OP_CREATE .....	60, 253
MPI_OP_FREE .....	60
MPI_OPEN_PORT .....	311
MPI_PROBE .....	33, 147
MPI_PUBLISH_NAME .....	311, 314
MPI_PUT .....	77, 80, 81, 84, 90, 91
MPI_RECV .....	26, 31, 33, 36, 38, 104, 106, 108, 125, 128, 133, 143, 260, 264, 283
MPI_RECV_INIT .....	274
MPI_REDUCE .....	41, 60, 63, 133, 253
MPI_REQUEST_FREE .....	275
MPI_SCAN .....	60, 67
MPI_SCATTER .....	41, 46, 67, 185
MPI_SCATTERV .....	67
MPI_SEND .....	26, 31, 33, 36, 38, 104, 106, 108, 125, 128, 133, 142, 260, 264
MPI_SENDRECV .....	29–31, 130, 289
MPI_SENDRECV_REPLACE .....	29, 112, 115, 121, 280, 282
MPI_SSEND .....	142, 143, 148, 282
MPI_SSEND_INIT .....	274

MPI_START .....	274, 275
MPI_TEST .....	147, 241, 243
MPI_TYPE_COMMIT .	97, 102, 103, 105, 107, 115, 121, 125, 128, 232, 235, 238, 260, 264, 271
MPI_TYPE_CONTIGUOUS .....	97, 99, 103
MPI_TYPE_CREATE_DARRAY .....	255
MPI_TYPE_CREATE_F90_COMPLEX .....	268
MPI_TYPE_CREATE_F90_INTEGER .....	268
MPI_TYPE_CREATE_F90_REAL .....	268
MPI_TYPE_CREATE_HINDEXED .....	109, 111
MPI_TYPE_CREATE_HVECTOR .....	97, 101, 109
MPI_TYPE_CREATE_RESIZED .....	126, 127, 232
MPI_TYPE_CREATE_STRUCT .....	122, 123, 125
MPI_TYPE_CREATE_SUBARRAY .....	116
MPI_TYPE_FREE .....	97, 102, 104, 106, 108, 115, 121, 125, 260, 264, 271
MPI_TYPE_GET_EXTENT .....	109, 126, 232
MPI_TYPE_INDEXED .....	109, 110, 115, 122, 232
MPI_TYPE_SIZE .....	74, 75, 87, 109, 126, 127, 202, 204, 215, 226, 239, 243
MPI_TYPE_VECTOR .....	97, 100, 105, 107, 127

MPI_UNPUBLISH_NAME .....	311
MPI_WAIT .....	146, 147, 241, 245, 273, 274
MPI_WIN_CREATE .....	72
MPI_WIN_FENCE .....	82, 84, 85, 90, 92
MPI_WIN_FREE .....	74
MPI_WIN_GET_ATTR .....	74
MPI_WIN_LOCK .....	92, 93, 95
MPI_WIN_UNLOCK .....	92, 93, 95
MPI_WTIME .....	133, 135, 143, 146