Message Passing Interface (MPI-1)

Jalel Chergui
Isabelle Dupays
Denis Girou
Stéphane Requena
<Prénom.Nom@idris.fr>

1 – Introduction

1.1 – Définitions

❶ Le modèle de programmation séquentiel :

- ☞ le programme est exécuté par un et un seul processeur ;
- ☞ toutes les variables et constantes du programme sont allouées dans la mémoire centrale du processeur.

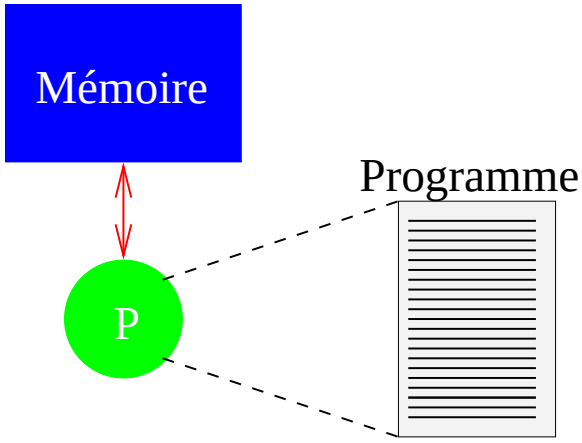


FIG. 1 – *Modèle séquentiel*

❷ Le modèle de programmation par **échange de messages** :

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

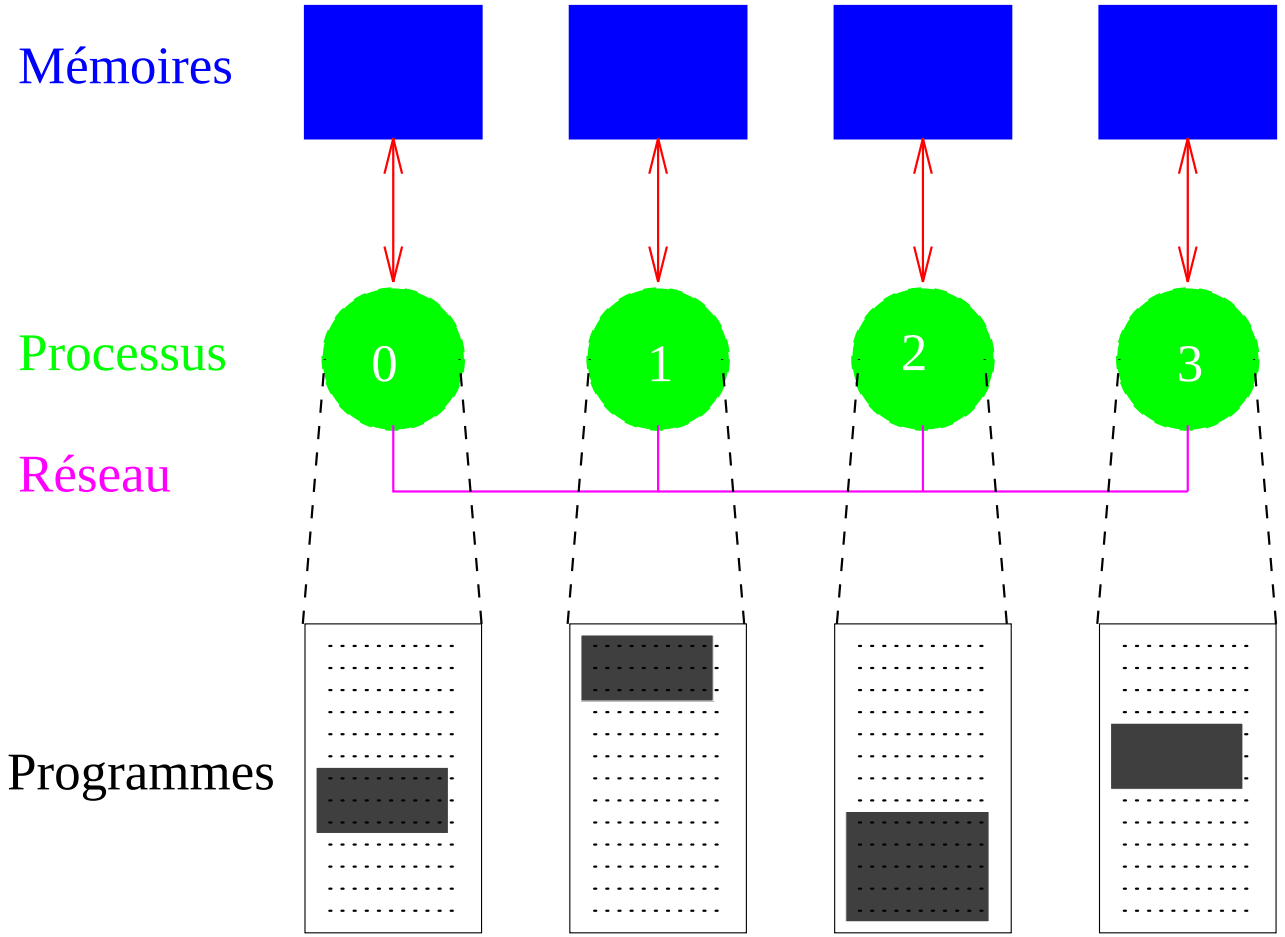


FIG. 2 – *Modèle à échange de messages*

❸ Le modèle d'exécution *SPMD* :

- ☞ *Single Program, Multiple Data* ;
- ☞ le même programme s'exécute pour 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*, qu'il peut en revanche émuler.

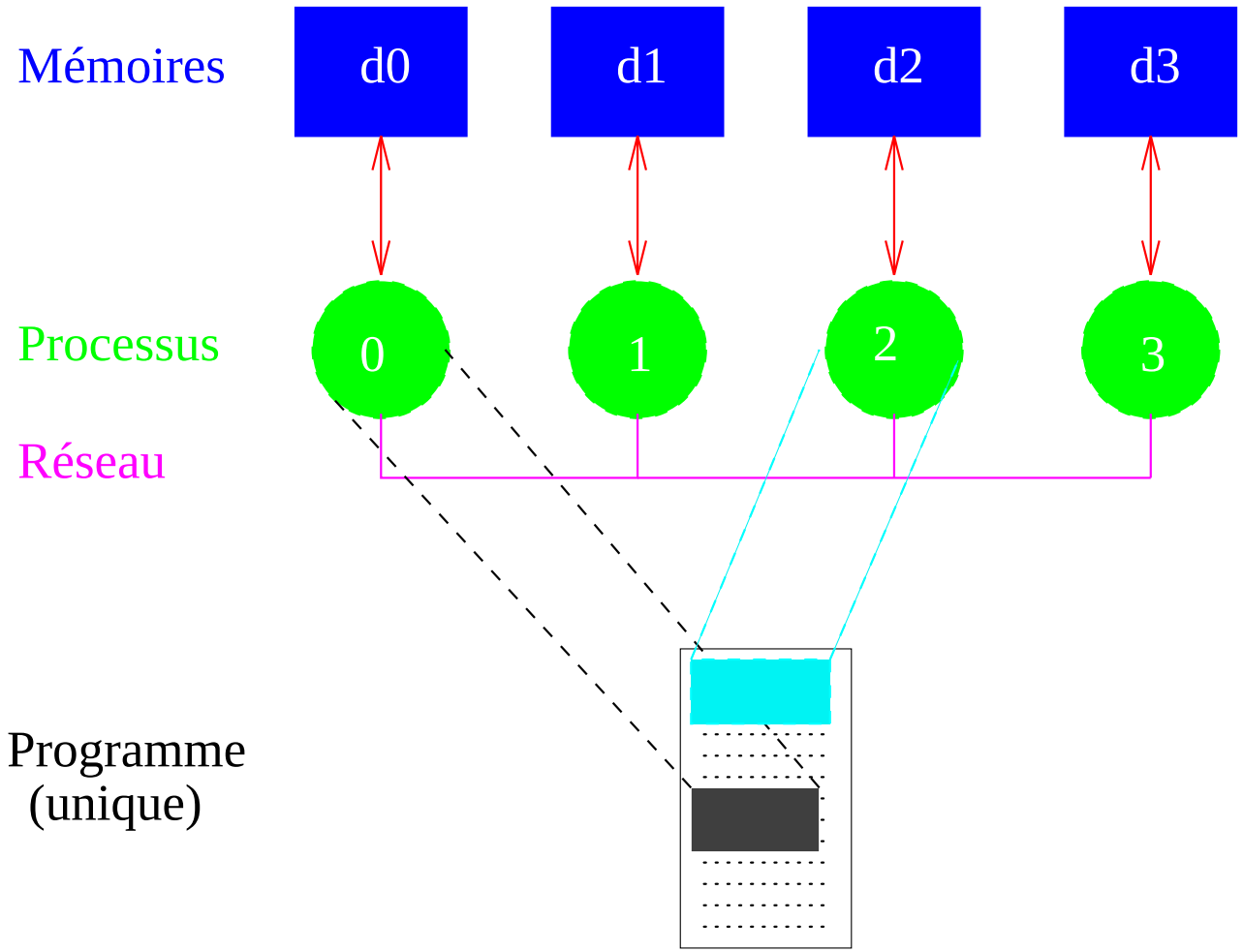


FIG. 3 – *Single Program, Multiple Data*

❹ Exemple en *Fortran* d'émulation *MPMD* en *SPMD*

```
program spmd
  if (ProcessusMaître) then
     call LeChef(Arguments)
  else
     call LesOuvriers(Arguments)
  endif
end program spmd
```

1.2 – Concepts de l'échange de messages

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

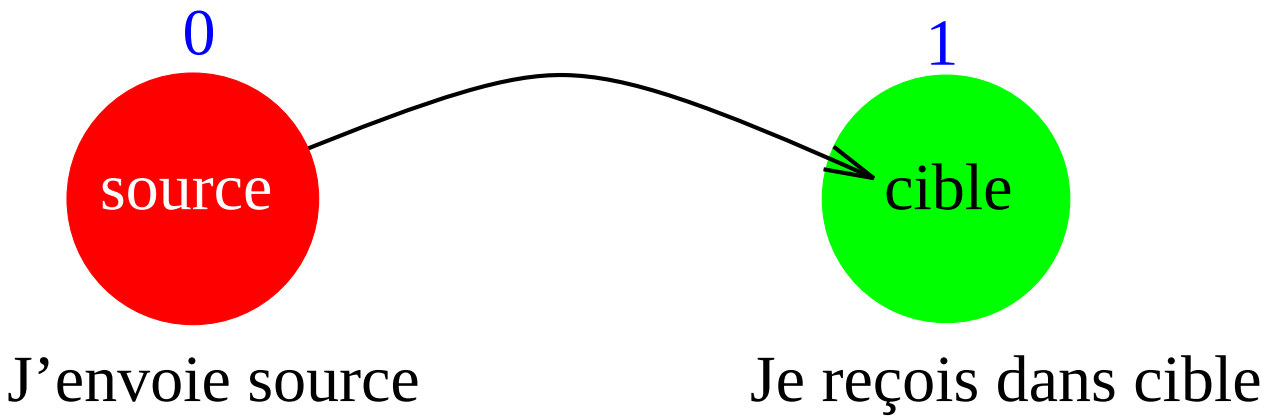


FIG. 4 – *L'échange de messages*

- 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.

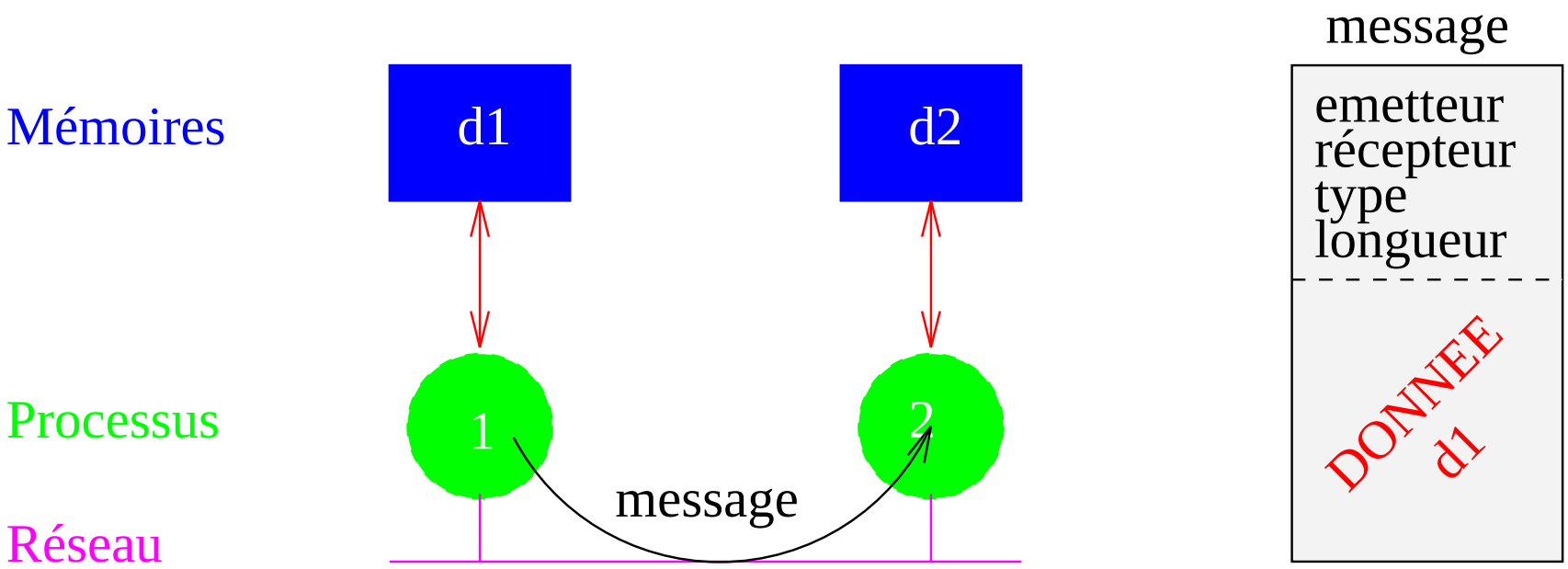


FIG. 5 – *Constitution d'un message*

- Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé :
  - à la téléphonie ;
  - à la télécopie ;
  - au courier postal ;
  - à une 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*-1 (*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*

Ces sous-programmes peuvent être classés dans les grandes catégories suivantes :

① environnement ;
② communications point à point ;
③ communications collectives ;
④ types de données dérivés ;
⑤ topologies ;
⑥ groupes et communicateurs.

1.3 – Historique

- Novembre 92 (*Supercomputing '92*) : « formalisation » d'un groupe de travail créé en avril 92 et décision d'adopter les structures et les méthodes du groupe `HPF` (*High Performance Fortran*).
- Participants, américains (essentiellement) et européens, aussi bien constructeurs que représentants du monde académique.
- « Brouillon » de *MPI*-1 présenté en novembre 93 (*Supercomputing '93*), finalisé en mars 1994.
- Vise à la fois la **portabilité** et la garantie de **bonnes performances**.
- « Brouillon » de *MPI*-2 en novembre 96, suite à des travaux ayant commencés au printemps 95.
- « Standard » non élaboré par les organismes officiels de normalisation (ISO, ANSI, etc.).

1.4 – Bibliographie

- Les spécifications de la norme : *MPI: A Message Passing Interface Standard*, Mars 1994.

  `ftp://ftp.irisa.fr/pub/netlib/mpi/drafts/draft-final.ps`

- Quelques ouvrages :

  1. Marc Snir & al. *MPI : The Complete Reference*. Second edition. MIT Press, 1998. Volume 1, *The MPI core*. Volume 2, *MPI-2*.
  2. William Gropp, Ewing Lusk et Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Second edition. MIT Press 1999 ;
  3. Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman Ed., 1997.

- Une documentation complémentaire et une description des différentes implémentations :

  `http://www.idris.fr/su/Parallele/Page_generale.html`

  `http://www.mcs.anl.gov/mpi/`

2 – Environnement

2.1 – Description

☞ 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.

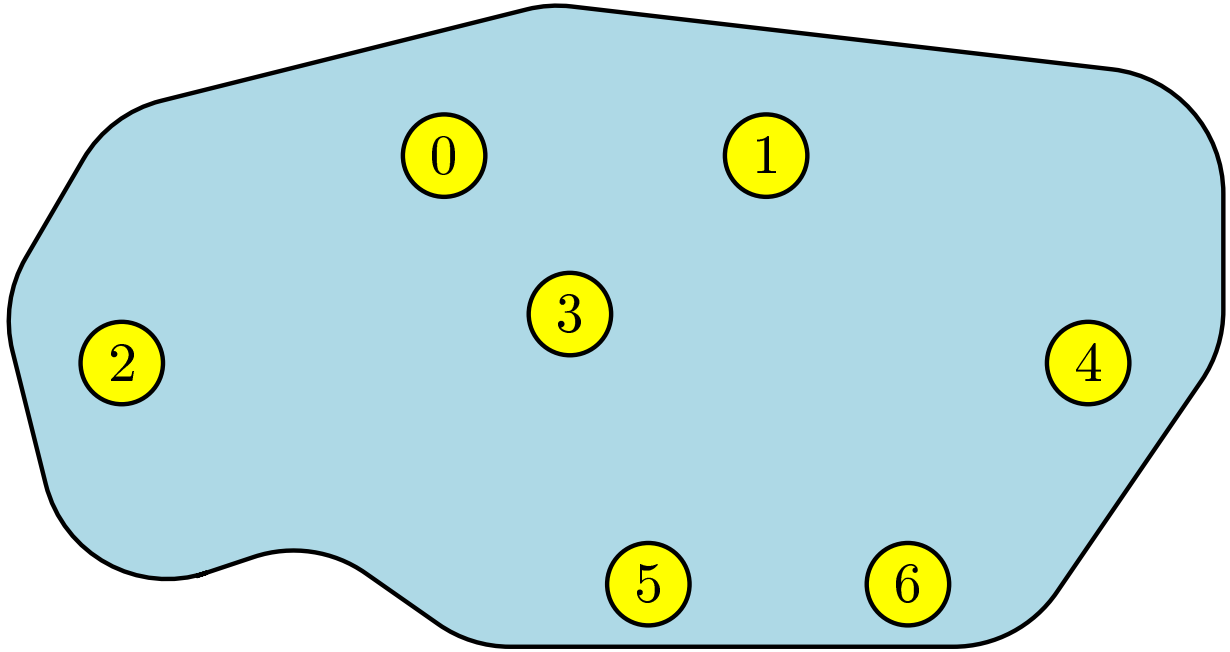


FIG. 6 – *Communicateur MPI_COMM_WORLD*

☞ 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    implicit none
3    include 'mpif.h'
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
```

```
> mpirun -np 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 **destinataire**.

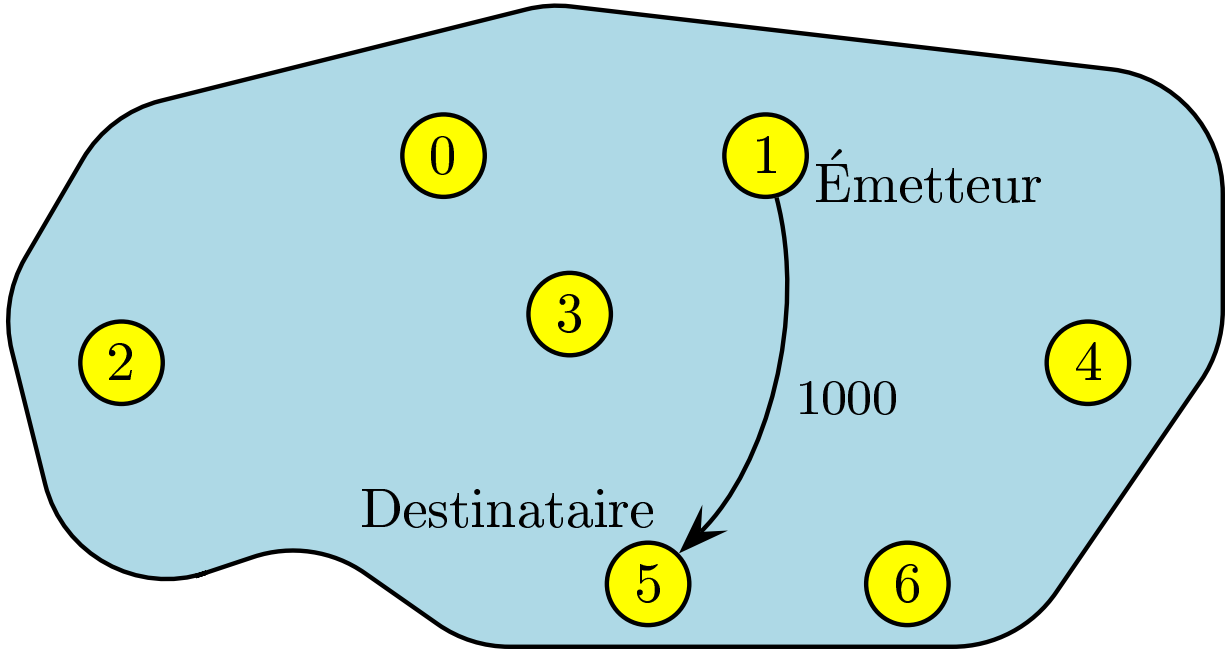


FIG. 7 – *Communication point à point*

- L'émetteur et le destinataire sont identifiés par leur **rang** dans le communicateur (i.e. leur numéro d'instance).
- Ce que l'on appelle l'**enveloppe d'un message** est constituée :
  1. du numéro du processus émetteur ;
  2. du numéro 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, qui font appel à des protocoles différents. Ils seront vus au chapitre 5.

```
1  program point_a_point
2    implicit none
3
4    include 'mpif.h'
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 == 1) 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,1,etiquette,MPI_COMM_WORLD,statut,code)
18      print *,'Moi, processus 5, j''ai reçu ',valeur,' du processus 1.'
19   end if
20
21   call MPI_FINALIZE(code)
22
23 end program point_a_point
```

```
> mpirun -np 7 point_a_point

Moi, processus 5, j'ai reçu 1000 du processus 1
```

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`
MPI_PACKED	Types hétérogènes

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_INT	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_PACKED	Types hétérogènes

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.
- On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que `MPI_TYPE_CONTIGUOUS()`, `MPI_TYPE_VECTOR()`, `MPI_TYPE_INDEXED()` et `MPI_TYPE_STRUCT()` (voir le chapitre 6).

```
1  program sendrecv
2    implicit none
3    include 'mpif.h'
4    integer, dimension(MPI_STATUS_SIZE) :: statut
5    integer, parameter                  :: etiquette=100
6    integer                             :: rang,valeur,num_proc,code
7
8    call MPI_INIT(code)
9    call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
10
11   ! On suppose avoir exactement 2 processus
12   num_proc=mod(rang+1,2)
13
14   call MPI_SENDRECV(rang+1000,1,MPI_INTEGER,num_proc,etiquette,valeur,1,MPI_INTEGER, &
15                     num_proc,etiquette,MPI_COMM_WORLD,statut,code)
16
17   print *,'Moi, processus ',rang,', j''ai reçu ',valeur,' du processus ',num_proc
18
19   call MPI_FINALIZE(code)
20 end program sendrecv
```

```
> mpirun -np 2 sendrecv

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

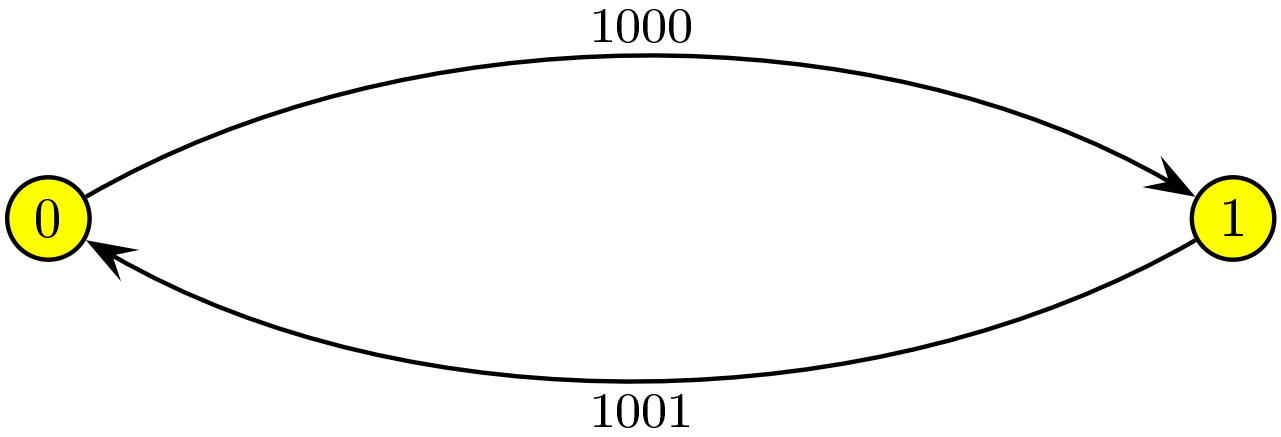


FIG. 8 – *Communication* **`sendrecv`** *entre les processus 0 et 1*

Il convient de noter que si le sous-programme `MPI_SEND` est implémenté de façon **bloquante** (voir le chapitre 5), ce code sera en situation de verrouillage, puisque chacun des deux processus attendra un ordre de réception qui ne viendra jamais.

3.4 – Exemple : anneau de communication

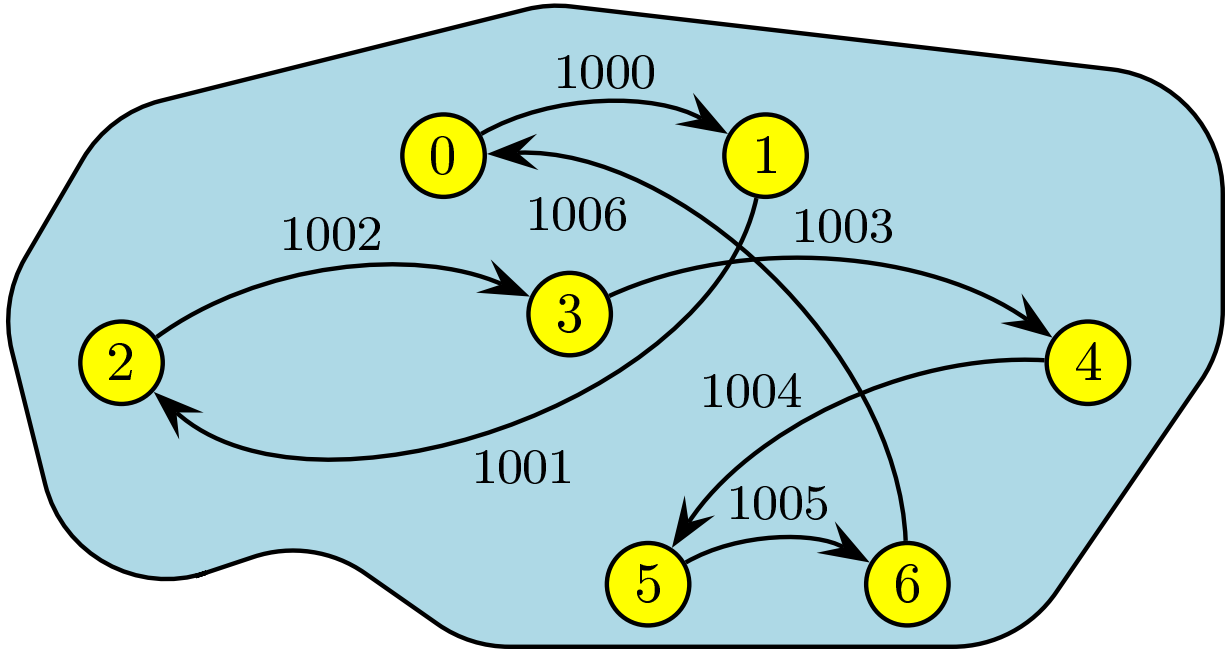


FIG. 9 – *Anneau de communication*

```
1  program anneau_1
2    implicit none
3    include 'mpif.h'
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
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   num_proc_suivant=mod(rang+1,nb_procs)
14   num_proc_precedent=mod(nb_procs+rang-1,nb_procs)
15
16   call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette,MPI_COMM_WORLD,code)
17   call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette,MPI_COMM_WORLD, &
18                 statut,code)
19
20   print *,'Moi, processus ',rang,', j''ai reçu ',valeur,' du processus ', &
21           num_proc_precedent
22
23   call MPI_FINALIZE(code)
24 end program anneau_1
```

```
> mpirun -np 7 anneau_1

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

Cet exemple peut se programmer autrement, en utilisant une seule opération pour faire un envoi et une réception par processus, puisqu'à chaque fois les processus récepteur et émetteur sont définis de façon similaire.

```
1  program anneau_2
2    implicit none
3    include 'mpif.h'
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
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   valeur=rang+1000
14   num_proc_suivant=mod(rang+1,nb_procs)
15   num_proc_precedent=mod(nb_procs+rang-1,nb_procs)
16
17   call MPI_SENDRECV_REPLACE(valeur,1,MPI_INTEGER,num_proc_suivant,etiquette, &
18                             num_proc_precedent,etiquette,MPI_COMM_WORLD,statut,code)
19
20   print *,'Moi, processus ',rang,', j''ai reçu ',valeur,' du processus ',num_proc
21
22   call MPI_FINALIZE(code)
23  end program anneau_2
```

Mais en fait on ne fait pas ici de communications en **anneau**, puisque celles-ci se font toutes simultanément... Il faut procéder différemment si on veut imposer véritablement l'organisation suivante, où les processus représentent du point de vue logique une topologie en **anneau** :

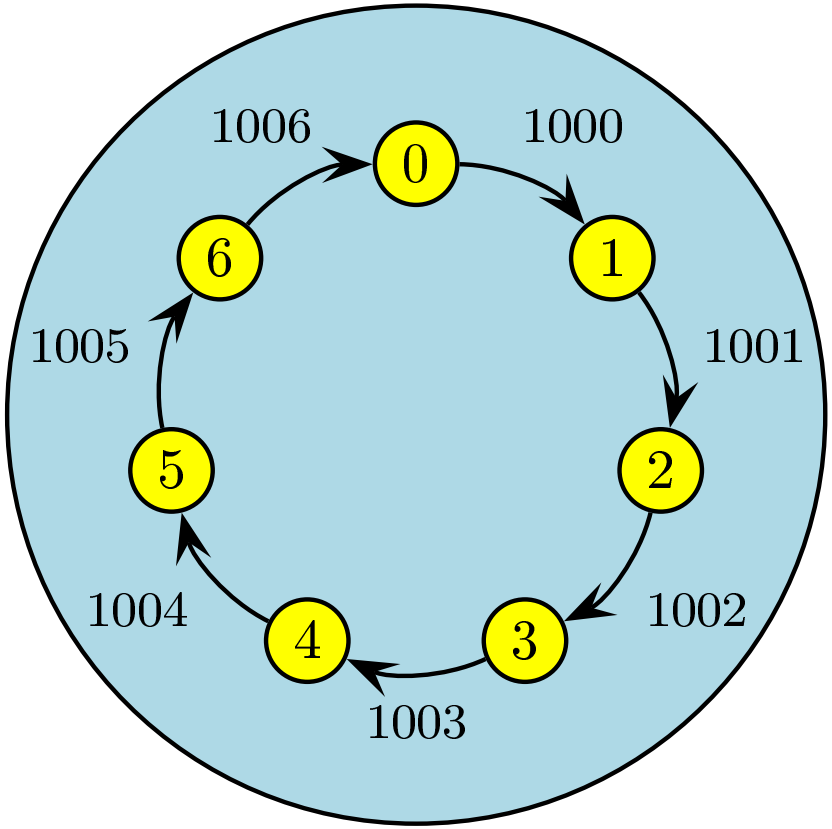


FIG. 10 – *Anneau de communication*

Dans ce cas, il faut donc programmer autrement pour **ordonner** les écritures. Il est alors nécessaire de gérer autrement l'ordre dans lequel vont se faire les communications :

```
17   if (rang == 0) then
18      call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
19                    MPI_COMM_WORLD,code)
20      call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
21                    MPI_COMM_WORLD,statut,code)
22   else
23      call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
24                    MPI_COMM_WORLD,statut,code)
25      call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
26                    MPI_COMM_WORLD,code)
27   end if
```

```
> mpirun -np 7 anneau_3

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

3.5 – Construction et reconstruction de messages

Dans le cas où le processus récepteur ne connaîtra qu'à l'exécution le nombre d'éléments à recevoir, il est possible de n'envoyer qu'un seul message regroupant ces informations. Ceci évite d'envoyer d'abord un message contenant le nombre d'éléments puis un autre contenant les éléments eux-mêmes. Pour ce faire, il faut construire en émission le contenu du message en passant par une zone tampon, via le sous-programme MPI_PACK. En réception, le sous-programme MPI_UNPACK permet de façon réciproque de reconstruire le message.

```
1  program pack_unpack
2    implicit none
3
4    include 'mpif.h'
5    integer, dimension(MPI_STATUS_SIZE)        :: statut
6    integer, parameter                         :: etiquette=999, &
7                                                  longueur_msg_max=100 ! En octets
8    integer                                    :: rang,nb_valeurs,position,code
9    real(kind=8), dimension(:), allocatable    :: valeurs
10   real(kind=8), dimension(longueur_msg_max) :: message
11
12   call MPI_INIT(code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
```

```
14
15    nb_valeurs=rang*3 + 2
16    allocate(valeurs(nb_valeurs))
17    call random_number(valeurs)
18
19    position=1
20    call MPI_PACK(nb_valeurs,1,MPI_INTEGER,message,longueur_msg_max, &
21                  position,MPI_COMM_WORLD,code) ! position est mis à jour en sortie
22    call MPI_PACK(valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,message,longueur_msg_max, &
23                  position,MPI_COMM_WORLD,code) ! position est mis à jour en sortie
24    call MPI_SEND(message,position,MPI_PACKED,mod(rang+1,2),etiquette,MPI_COMM_WORLD, &
25                  code)
26    deallocate(valeurs)
27
28    call MPI_RECV(message,longueur_msg_max,MPI_PACKED,mod(rang+1,2),etiquette, &
29                  MPI_COMM_WORLD,statut,code)
30    position=1
31    call MPI_UNPACK(message,longueur_msg_max,position, & ! position mis à jour en sortie
32                    nb_valeurs,1,MPI_INTEGER,MPI_COMM_WORLD,code)
33    allocate(valeurs(nb_valeurs))
34    call MPI_UNPACK(message,longueur_msg_max,position, & ! position mis à jour en sortie
35                    valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,MPI_COMM_WORLD,code)
36
37    print *,'Moi, processus ',rang,', j''ai reçu ',nb_valeurs,' valeurs du processus ',&
38            mod(rang+1,2),' : ',valeurs(1:nb_valeurs)
39    deallocate(valeurs)
40
41    call MPI_FINALIZE(code)
42  end program pack_unpack
```

```
> mpirun -np 2 pack_unpack

Moi, processus 1, j'ai reçu 2 valeurs du processus 0 : 0.58, 0.95
Moi, processus 0, j'ai reçu 5 valeurs du processus 1 : 0.58, 0.95, 0.78, 0.29, 0.45
```

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 celle qui assure les synchronisations globales : `MPI_BARRIER()`.

❷ celles 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 celles 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()

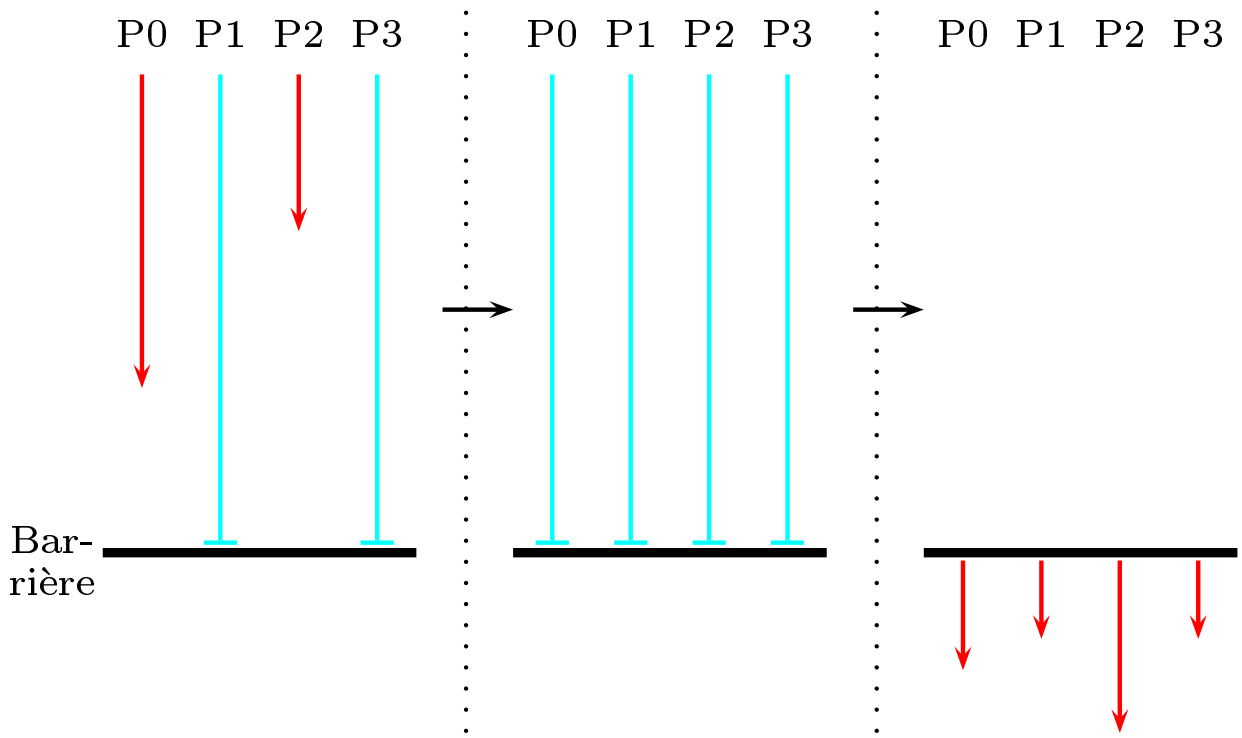


FIG. 11 – *Synchronisation globale : MPI_BARRIER()*

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

call MPI_BARRIER(MPI_COMM_WORLD,code)
```

4.3 – Diffusion générale : MPI_BCAST()

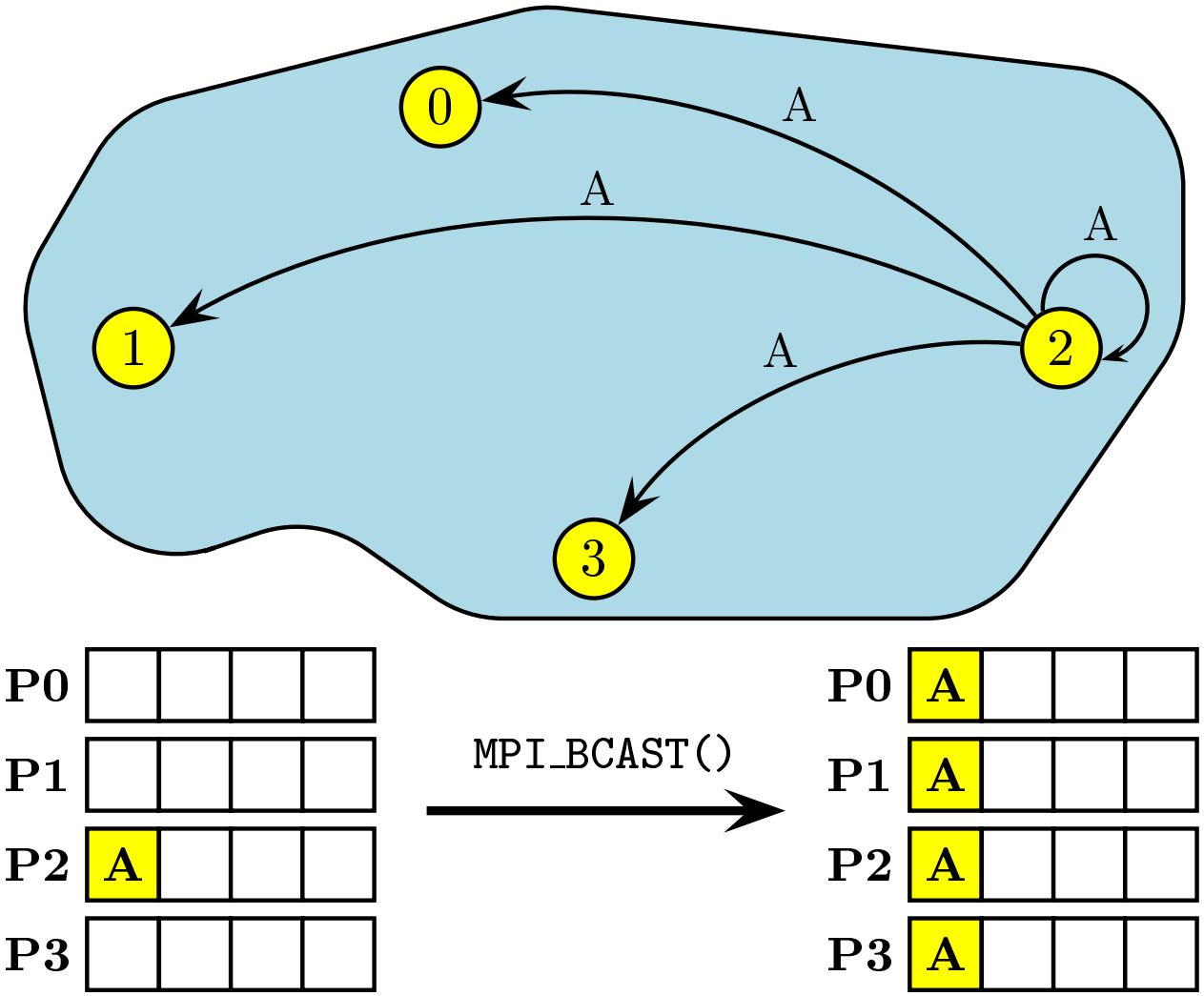


FIG. 12 – *Diffusion générale : MPI_BCAST()*

```
1  program bcast
2    implicit none
3
4    include 'mpif.h'
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
```

```
> mpirun -np 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()

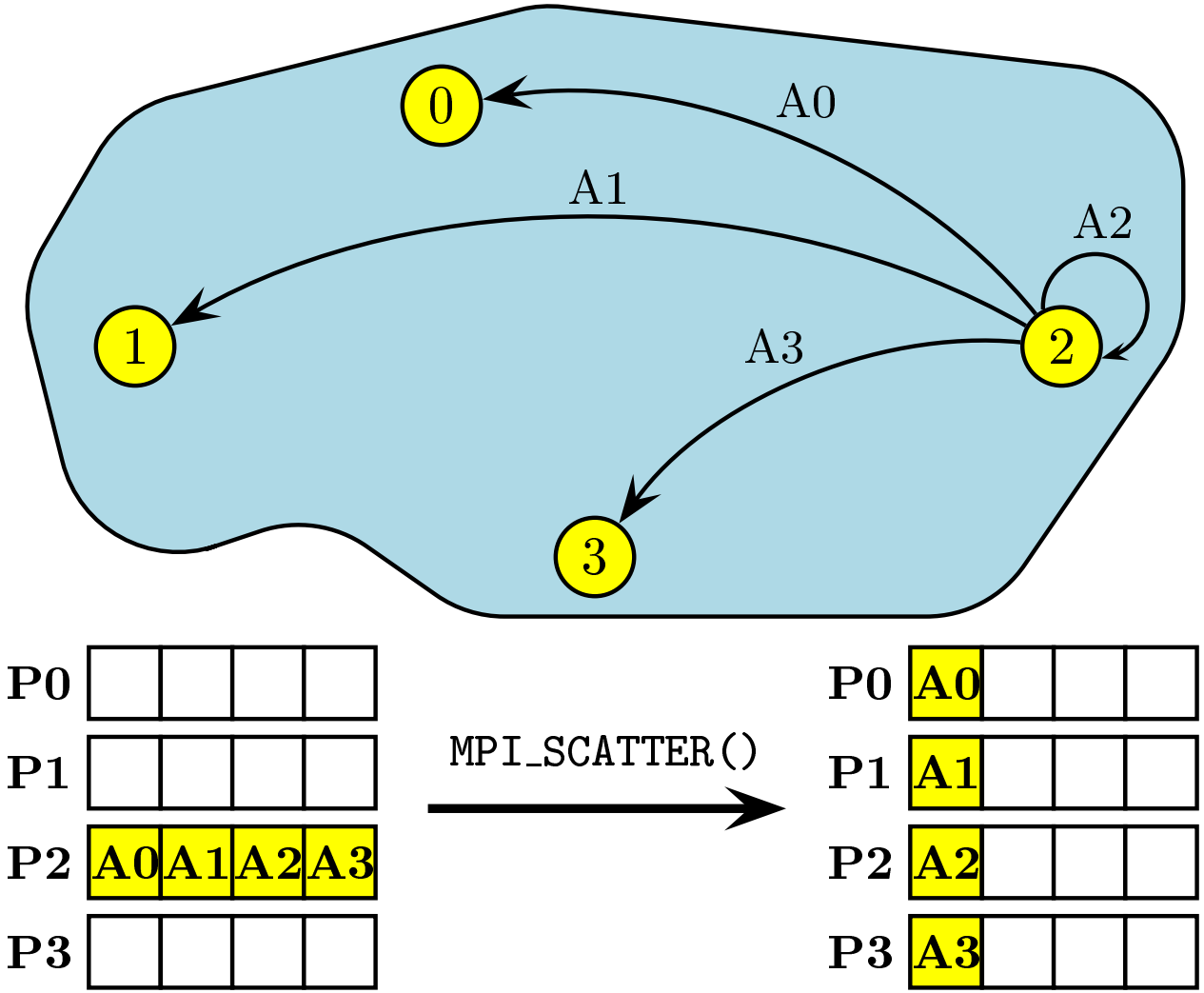


FIG. 13 – *Diffusion sélective : MPI_SCATTER()*

```
1  program scatter
2    implicit none
3
4    include 'mpif.h'
5    integer, parameter                :: nb_valeurs=128
6    integer                           :: nb_procs,rang,longueur_tranche,i,code
7    real, allocatable, dimension(:) :: valeurs,donnees
8
9    call MPI_INIT(code)
10
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/nb_procs
15   allocate(donnees(longueur_tranche))
16
17   if (rang == 2) then
18      allocate(valeurs(nb_valeurs))
19      valeurs(:)=(/(1000.+i,i=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
25   print *,'Moi, processus ',rang,', j''ai reçu ', donnees(1),' à ', &
26           donnees(longueur_tranche),' du processus 2'
27
28   call MPI_FINALIZE(code)
29 end program scatter
```

```
> mpirun -np 4 scatter

Moi, pr. 0, j'ai reçu 1001. à 1032. du pr. 2
Moi, pr. 1, j'ai reçu 1033. à 1064. du pr. 2
Moi, pr. 3, j'ai reçu 1097. à 1128. du pr. 2
Moi, pr. 2, j'ai reçu 1065. à 1096. du pr. 2
```

4.5 – Collecte : MPI_GATHER()

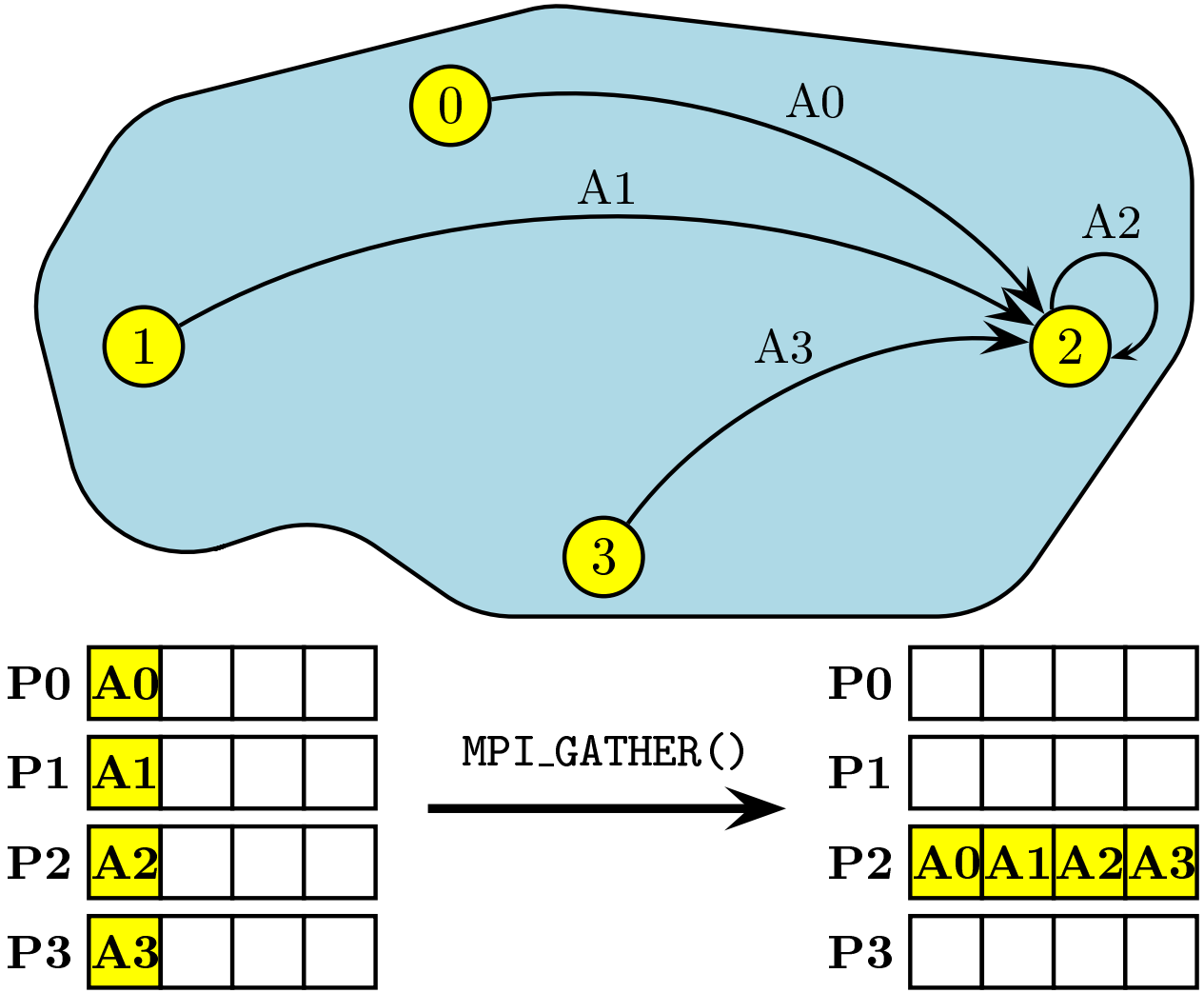


FIG. 14 – *Collecte : MPI_GATHER()*

```
1  program gather
2    implicit none
3    include 'mpif.h'
4    integer, parameter                :: nb_valeurs=128
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   allocate(valeurs(longueur_tranche))
15
16   valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
17
18   call MPI_GATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
19                   MPI_REAL,2,MPI_COMM_WORLD,code)
20
21   if (rang == 2) print *,'Moi, processus 2', j''ai reçu ',donnees(1),' ... ', &
22                          donnees(longueur_tranche+1),' ... ',donnees(nb_valeurs)'
23   call MPI_FINALIZE(code)
24 end program gather
```

```
> mpirun -np 4 gather

Moi, pr. 2, j'ai reçu 1001. ... 1033. ... 1128.
```

4.6 – Collecte générale : MPI_ALLGATHER()

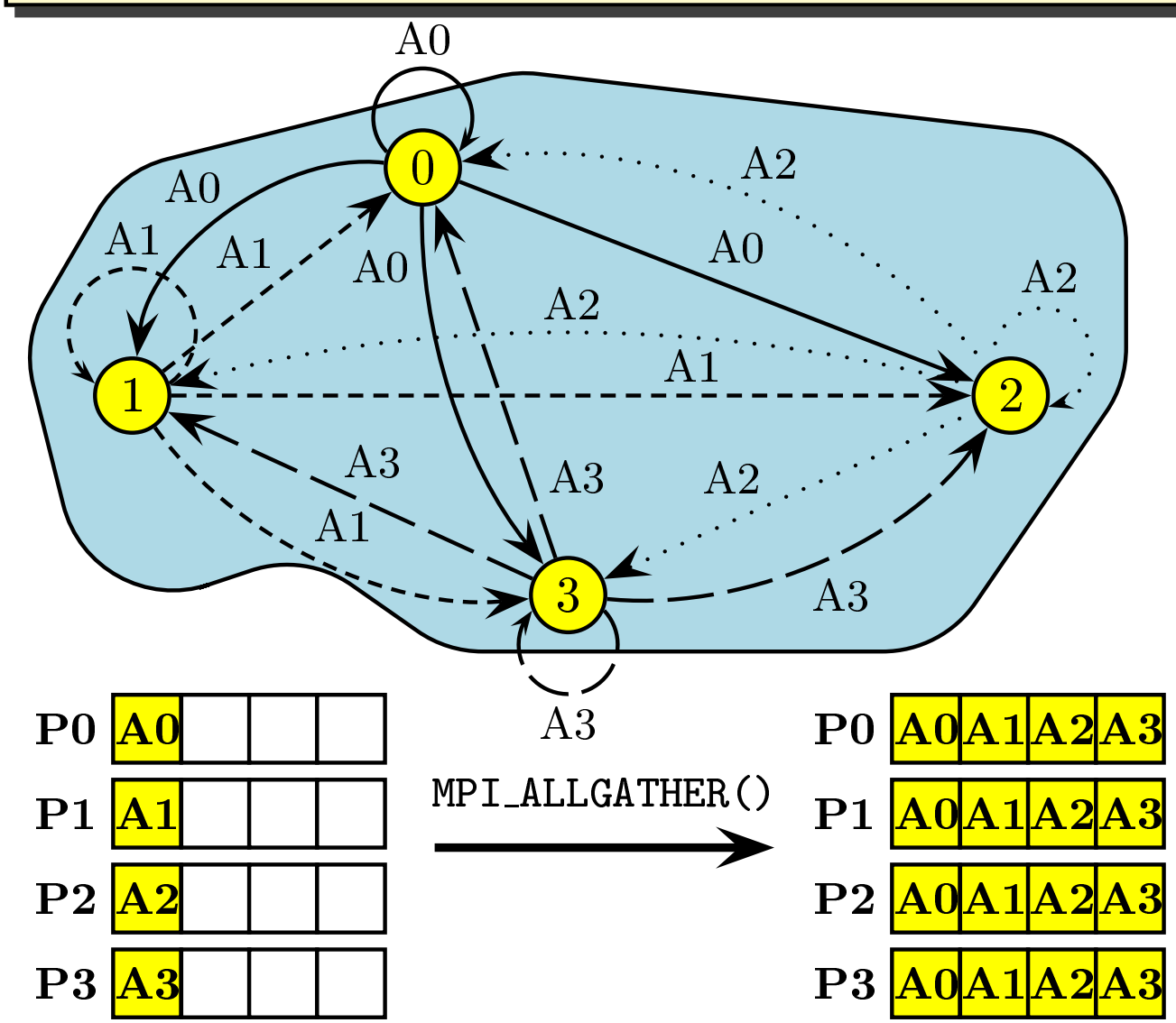


FIG. 15 – *Collecte générale : MPI_ALLGATHER()*

```
1  program allgather
2    implicit none
3
4    include 'mpif.h'
5    integer, parameter                :: nb_valeurs=128
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),' ... ', &
24           donnees(longueur_tranche+1),' ... ',donnees(nb_valeurs)'
25
26   call MPI_FINALIZE(code)
27
28 end program allgather
```

```
> mpirun -np 4 allgather

Moi, pr. 1, j'ai reçu 1001. ... 1033. ... 1128.
Moi, pr. 3, j'ai reçu 1001. ... 1033. ... 1128.
Moi, pr. 2, j'ai reçu 1001. ... 1033. ... 1128.
Moi, pr. 0, j'ai reçu 1001. ... 1033. ... 1128.
```

4.7 – Échanges croisés : MPI_ALLTOALL()

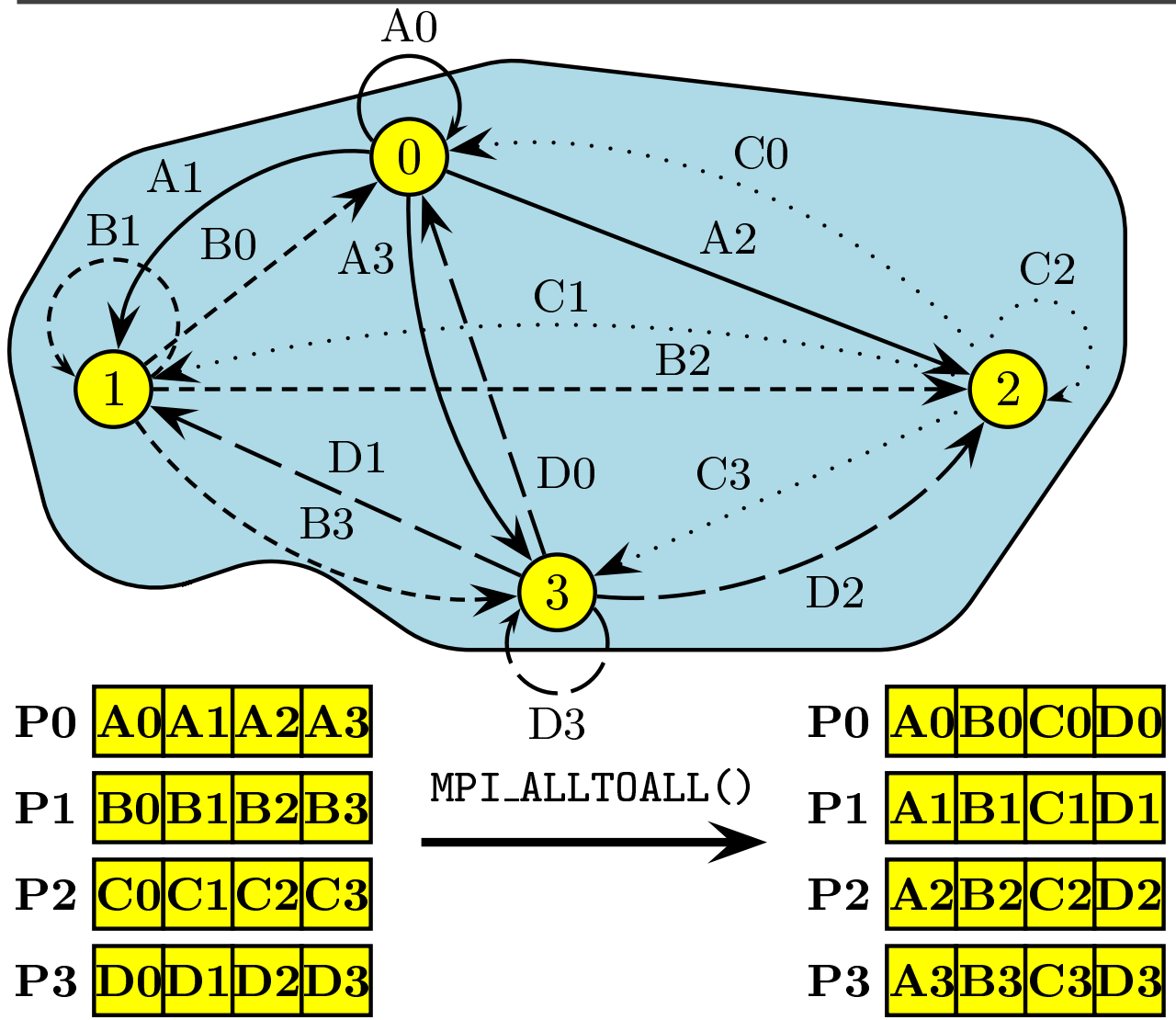


FIG. 16 – *Échanges croisés : MPI_ALLTOALL()*

```
1  program alltoall
2    implicit none
3
4    include 'mpif.h'
5    integer, parameter                :: nb_valeurs=128
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   call MPI_ALLTOALL(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
17                     MPI_REAL,MPI_COMM_WORLD,code)
18
19   print *,'Moi, processus ',rang,', j''ai reçu ',donnees(1),' ... ', &
20           donnees(longueur_tranche+1),' ... ',donnees(nb_valeurs)'
21
22   call MPI_FINALIZE(code)
23
24 end program alltoall
```

```
> mpirun -np 4 alltoall

Moi, processus 0, j'ai reçu 1001. ... 1129. ... 1416.
Moi, processus 2, j'ai reçu 1065. ... 1193. ... 1480.
Moi, processus 1, j'ai reçu 1033. ... 1161. ... 1448.
Moi, processus 3, j'ai reçu 1097. ... 1225. ... 1512.
```

4.8 – Réductions réparties

- Une **réduction** est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur. Des exemples typiques sont la somme des éléments d'un vecteur `SUM(A(:))` ou la recherche de l'élément de valeur maximum dans un vecteur `MAX(V(:))`.
- *MPI* propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus, 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

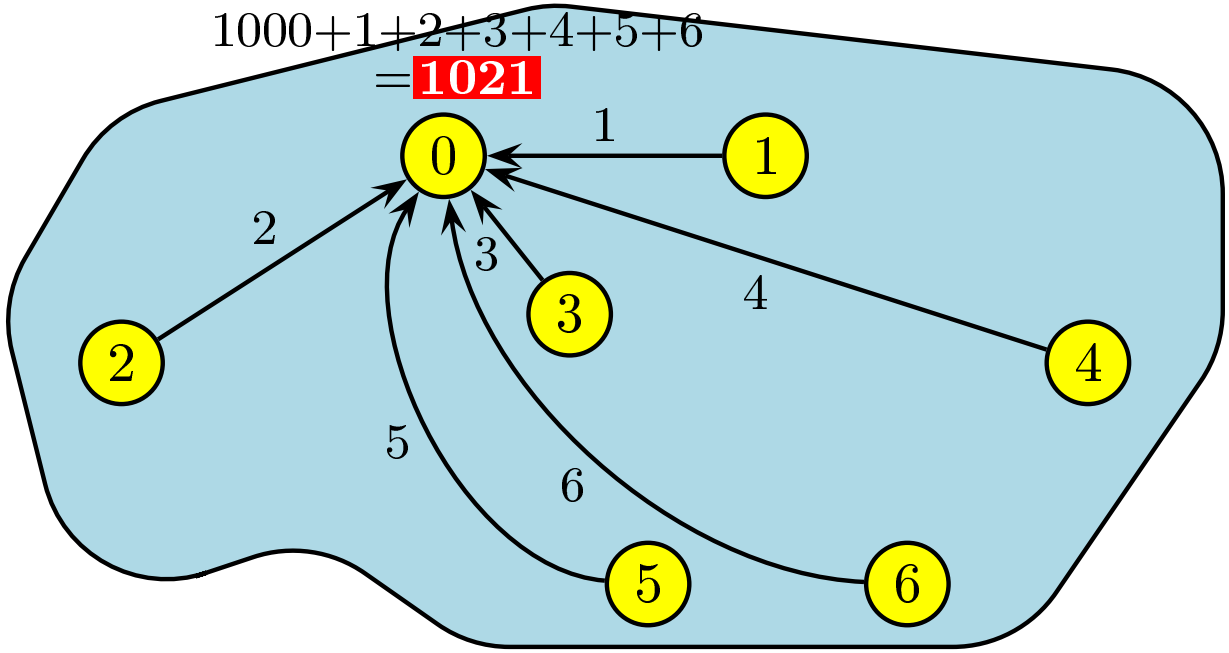


FIG. 17 – *Réduction répartie (somme)*

```
1  program reduce
2    implicit none
3    include 'mpif.h'
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
```

```
> mpirun -np 7 reduce

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

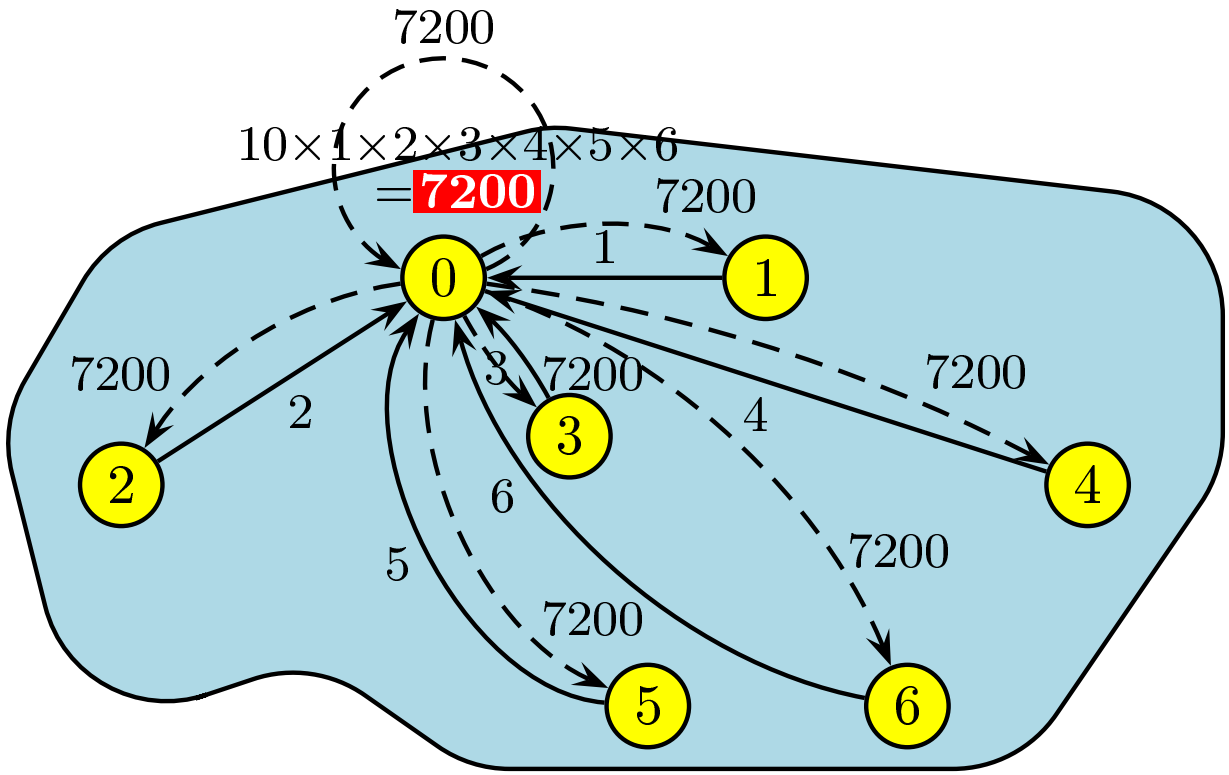


FIG. 18 – *Réduction répartie avec diffusion du résultat (produit)*

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

```
> mpirun -np 7 allreduce

Moi, pr. 6, j'ai reçu val. du produit global 7200
Moi, pr. 2, j'ai reçu val. du produit global 7200
Moi, pr. 0, j'ai reçu val. du produit global 7200
Moi, pr. 4, j'ai reçu val. du produit global 7200
Moi, pr. 5, j'ai reçu val. du produit global 7200
Moi, pr. 3, j'ai reçu val. du produit global 7200
Moi, pr. 1, j'ai reçu val. du produit global 7200
```

```
1  program ma_reduction
2   implicit none
3   include 'mpif.h'
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
12  ! Initialisation du vecteur A sur chaque processus
13  a(:) = (/ (cmplx(rang+i,rang+i+1),i=1,n) /)
14
15  ! Création de l'opération commutative mon_operation
16  call MPI_OP_CREATE(mon_produit,.true.,mon_operation,code)
17
18  ! Collecte sur le processus 0 du produit global
19  call MPI_REDUCE(a,resultat,n,MPI_COMPLEX,mon_operation,0,MPI_COMM_WORLD,code)
20
21  ! Affichage du résultat
22  if (rang == 0) then
23     print *,'Valeur du produit',resultat
24  end if
25
26  call MPI_FINALIZE(code)
27 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) = cmplx(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
```

```
> mpirun -np 5 ma_reduction

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

4.9 – 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.

5 – Optimisations

5.1 – 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 si possible la recopie du message dans un espace mémoire temporaire (*buffering*) ;
  3. minimiser les surcoûts induits par des appels répétitifs aux sous-programmes de communication.

5.2 – Programme modèle

```
1  program AOptimiser
2   implicit none
3   include 'mpif.h'
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                               :: nb_procs,rang,code
13  real(kind=8)                          :: temps_debut,temps_fin,temps_fin_max
14  integer, dimension(MPI_STATUS_SIZE) :: statut
15
16  !*** Initialisation MPI
17  call MPI_INIT(code)
18  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
19  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
20
21  !*** Initialisation des tableaux
22  call random_number(a)
23  call random_number(b)
24  call random_number(c)
```

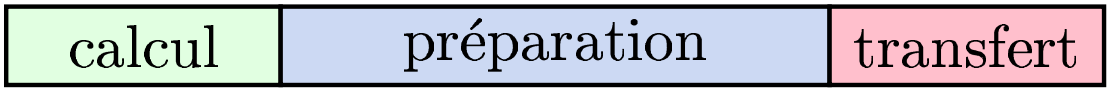
```
25  temps_debut = MPI_WTIME()
26  if (rang == 0) then
27    !*** J'envoie un gros message
28    call MPI_SEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,code)
29    !*** Ce calcul(factorisation LU avec LAPACK) modifie le contenu du tableau A
30    call sgetrf(na, na, a, na, pivota, code)
31    !***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

```
> mpirun -np 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 !

5.3 – 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.
- Ce que l'on mesure en général est la somme des deux.

5.4 – 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 peut parfois être prise en charge par le système *MPI*. C'est le cas par exemple pour les sous-programmes `MPI_SEND()` et `MPI_RECV()` sur *T3E*.

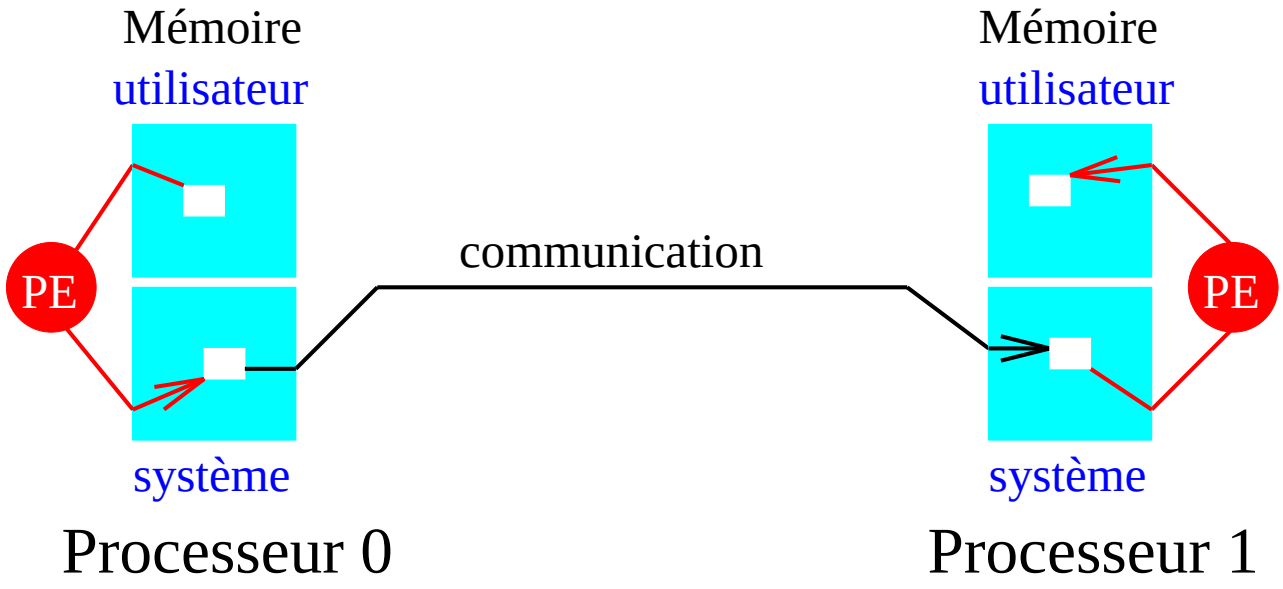


FIG. 19 – *Recopie temporaire d'un message*

❷ *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.

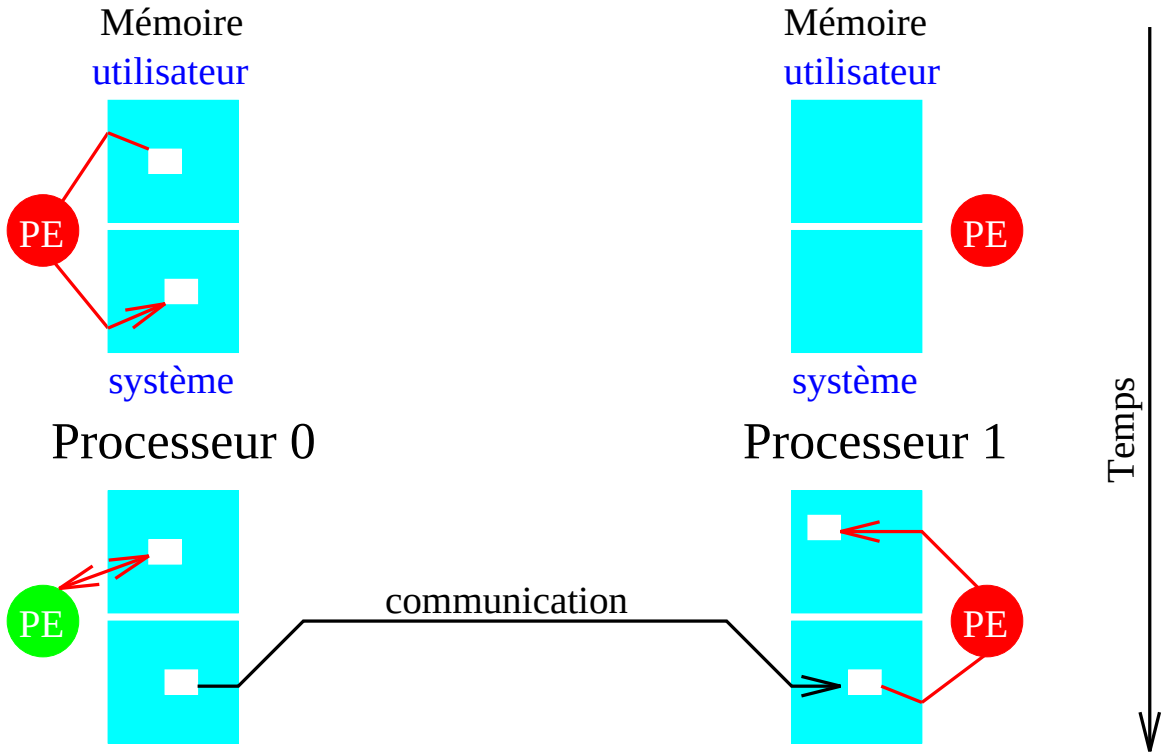


FIG. 20 – *Envoi non bloquant avec recopie temporaire du message*

❸ *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.

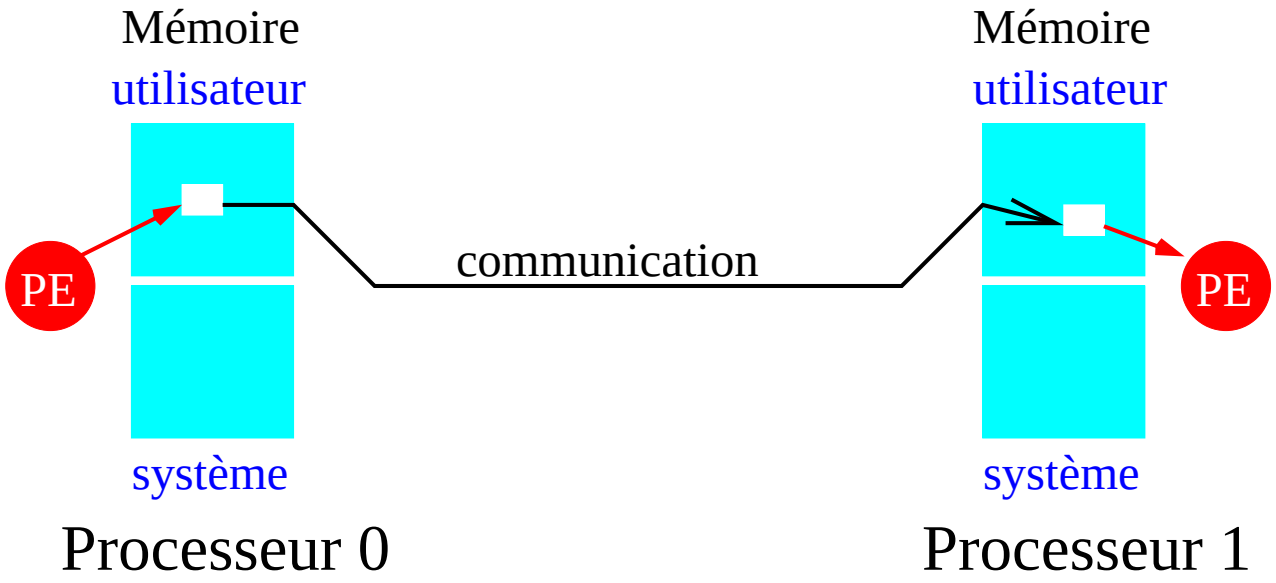


FIG. 21 – *Envoi bloquant couplé avec la réception*

❹ *Envoi non bloquant sans recopie 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.

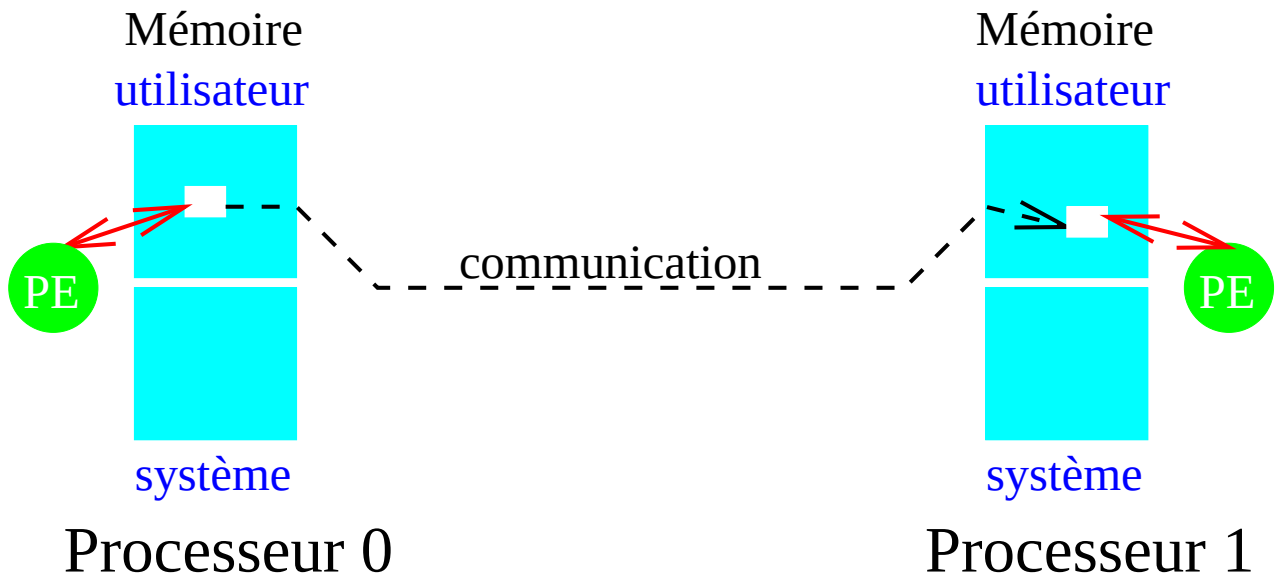


FIG. 22 – *Envoi non bloquant couplé avec la réception*

5.5 – 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 recopie temporaire du message. Si c'est le cas, l'envoi se termine lorsque la recopie 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 ***synchroneous*** : 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 recopie temporaire du message. L'envoi du message se termine lorsque la recopie 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 *standard*	MPI_SEND() [a]	MPI_ISEND()
envoi *synchroneous*	MPI_SSEND()	MPI_ISSEND()
envoi *buffered*	MPI_BSEND()	MPI_IBSEND()
envoi *ready*	MPI_RSEND()	MPI_IRSEND()
réception	MPI_RECV()	MPI_IRECV()

☞ Remarques : pour une implémentation *MPI* donnée, un envoi standard peut-être bloquant avec recopie temporaire ou synchrone avec la réception, selon la taille du message à envoyer.

a. En réalité non bloquant dans certaines implémentations.

5.6 – Envoi synchrone bloquant

Ce mode d'envoi (`MPI_SSEND()`) de messages permet d'éviter la recopie temporaire des messages et, par conséquent, les surcoûts que cela peut engendrer. L'efficacité de ce mode de communication sur *T3E*, particulièrement notable pour les applications **bien équilibrées**, provient surtout du fait qu'il est bâti sur des sous-programmes permettant la copie directe de mémoire à mémoire.

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    !*** J'envoie un gros message
25    call MPI_SSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,code)
26    !*** Je calcule : factorisation LU avec LAPACK
27    call sgetrf(na, na, a, na, pivota, code)
28    !*** Ce calcul modifie le contenu du tableau C
29    c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
30  elseif (rang == 1) then
31    !*** Je calcule
32    call sgetrf(na, na, a, na, pivota, code)
33    !*** Je reçois le gros message
34    call MPI_RECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,statut,code)
35    !*** Ce calcul dépend du message précédent
36    a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
37    !*** Ce calcul est indépendant du message
38    call sgetrf(nb, nb, b, nb, pivotb, code)
39  end if
40  temps_fin = (MPI_WTIME() - temps_debut)
```

```
> mpirun -np 2 AOptimiser
Temps :  0.36 secondes
```

5.7 – Envoi synchrone non-bloquant

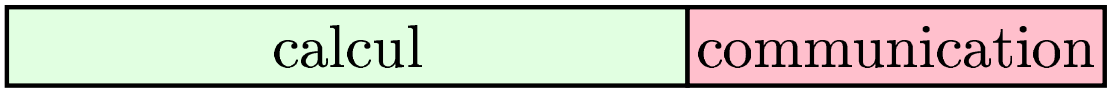
L'utilisation des sous-programmes `MPI_ISSEND()` et `MPI_IRECV()` conjointement aux sous-programmes de synchronisation précédents permet principalement de recouvrir les communications par des calculs.

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   implicit none
3   include 'mpif.h'
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                               :: nb_procs,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  !*** Initialisation MPI
17  call MPI_INIT(code)
18  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
19  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
20
21  !*** Initialisation des tableaux
22  call random_number(a)
23  call random_number(b)
24  call random_number(c)
```

```
25  temps_debut = MPI_WTIME()
26  if (rang == 0) then
27    !*** J'envoie un gros message
28    call MPI_ISSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
29    !*** Je calcule : factorisation LU avec LAPACK
30    call sgetrf(na, na, a, na, pivota, code)
31    !*** Ce calcul modifie le contenu du tableau C
32    call MPI_WAIT(requete0,statut,code)
33    c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
34  elseif (rang == 1) then
35    !*** Je calcule
36    call sgetrf(na, na, a, na, pivota, code)
37    !*** Je reçois le gros message
38    call MPI_IRECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
39    !*** Ce calcul recouvre la réception précédente
40    call sgetrf(nb, nb, b, nb, pivotb, code)
41    !*** Ce calcul dépend du message précédent
42    call MPI_WAIT(requete1,statut,code)
43    a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
44  end if
45  temps_fin = (MPI_WTIME() - temps_debut)
```

```
> mpirun -np 2 AOptimiser
Temps :  0.23 secondes
```



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.

5.8 – Conseils 1

- ☞ Éviter si possible la recopie temporaire des messages en utilisant le sous-programme `MPI_SSEND()`.
- ☞ Recouvrir les communications par des calculs tout en évitant la recopie temporaire des messages en utilisant les sous-programmes non bloquants `MPI_ISSEND()` et `MPI_IRECV()`.

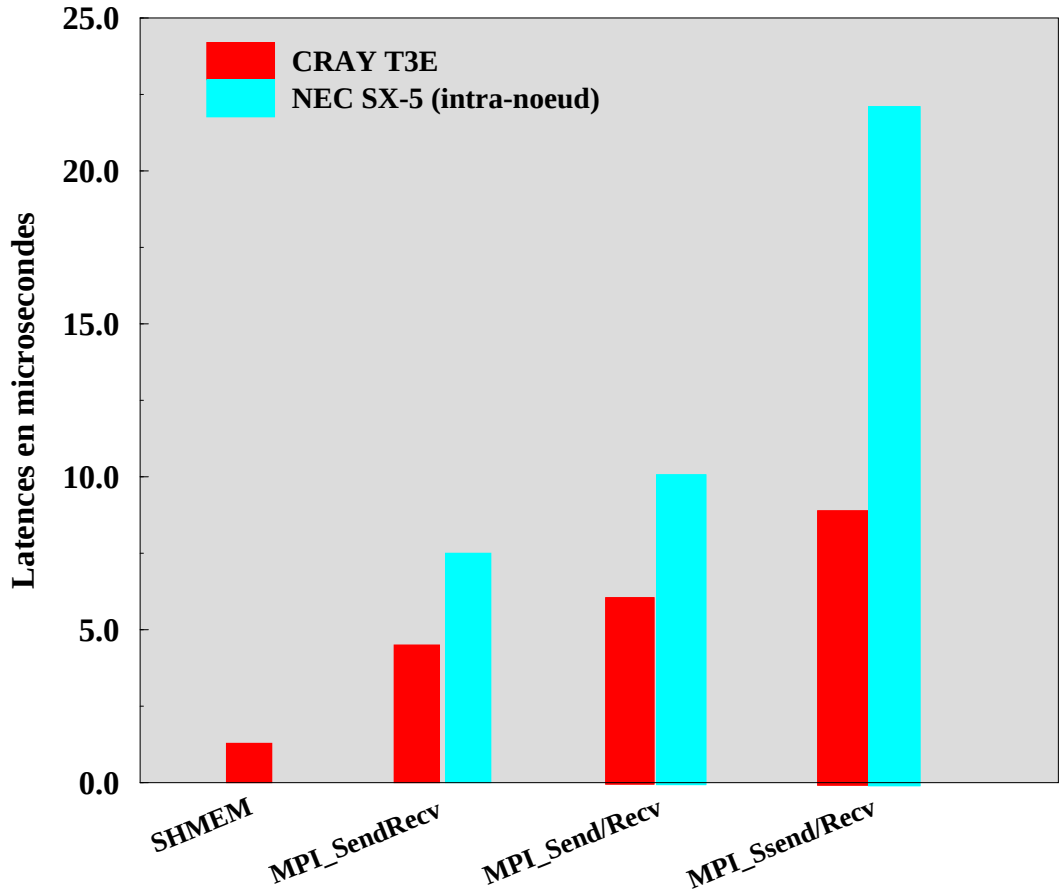
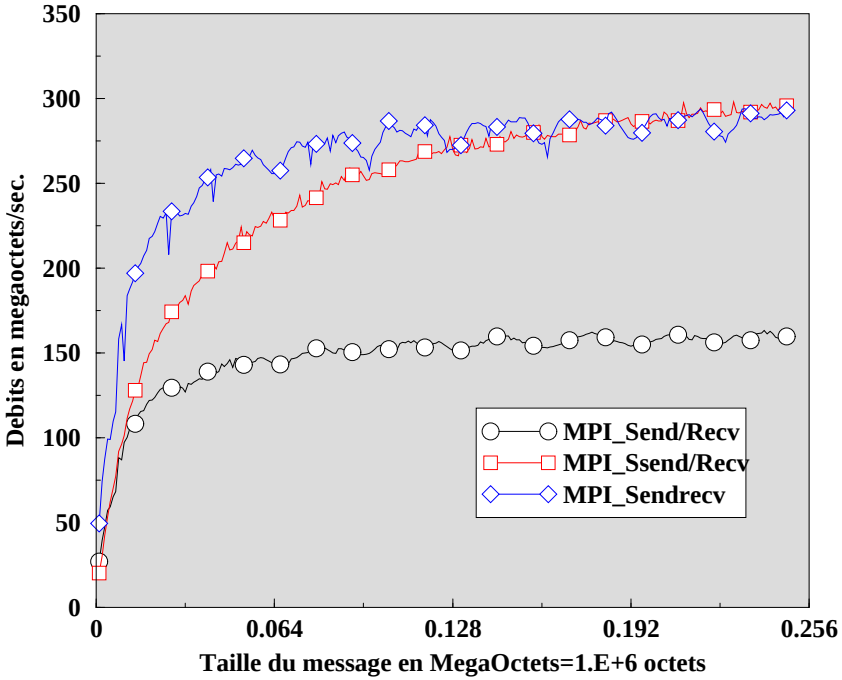
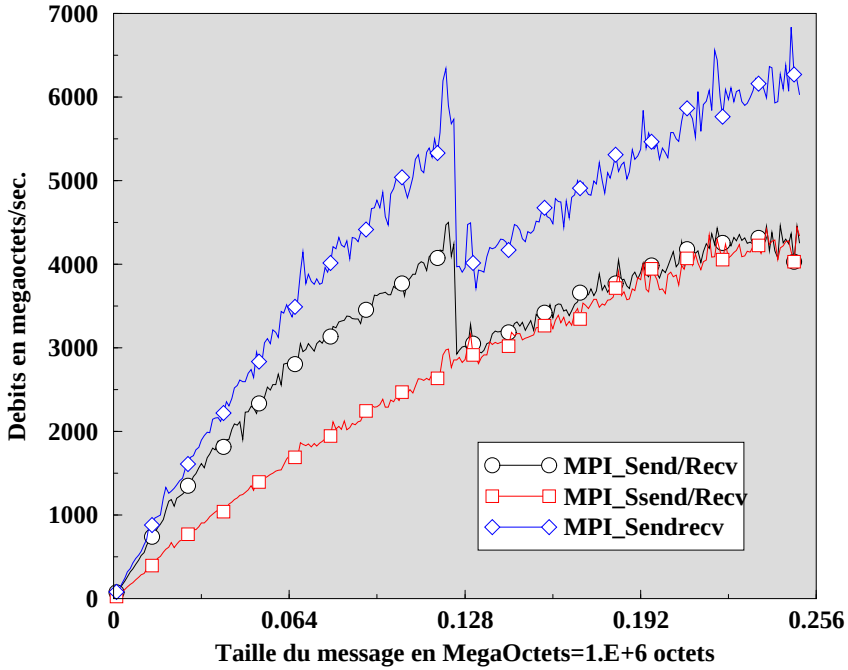


FIG. 23 – *Latences des communications sur* `CRAY T3E-600` *et* `NEC SX-5`

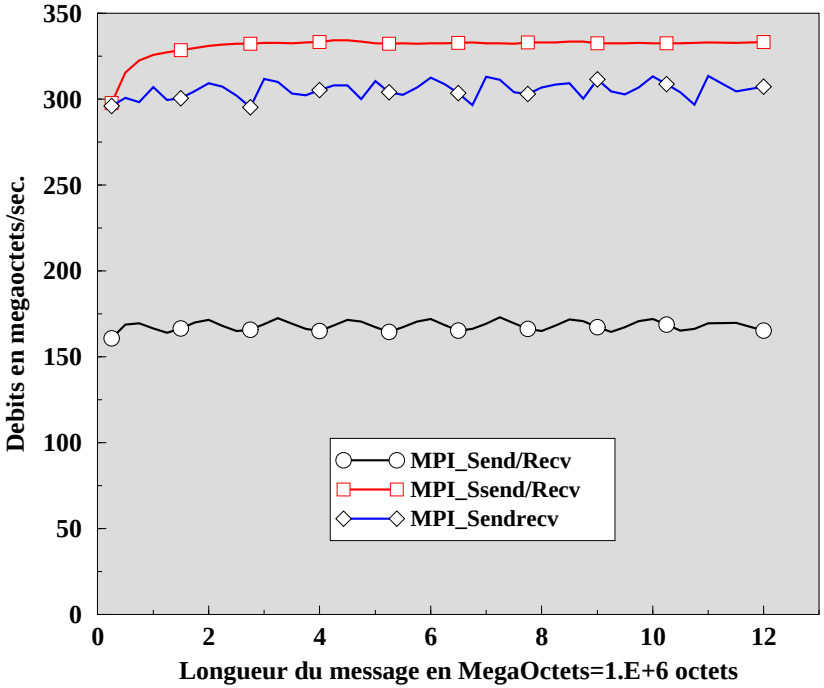


(a) `CRAY T3E-600`

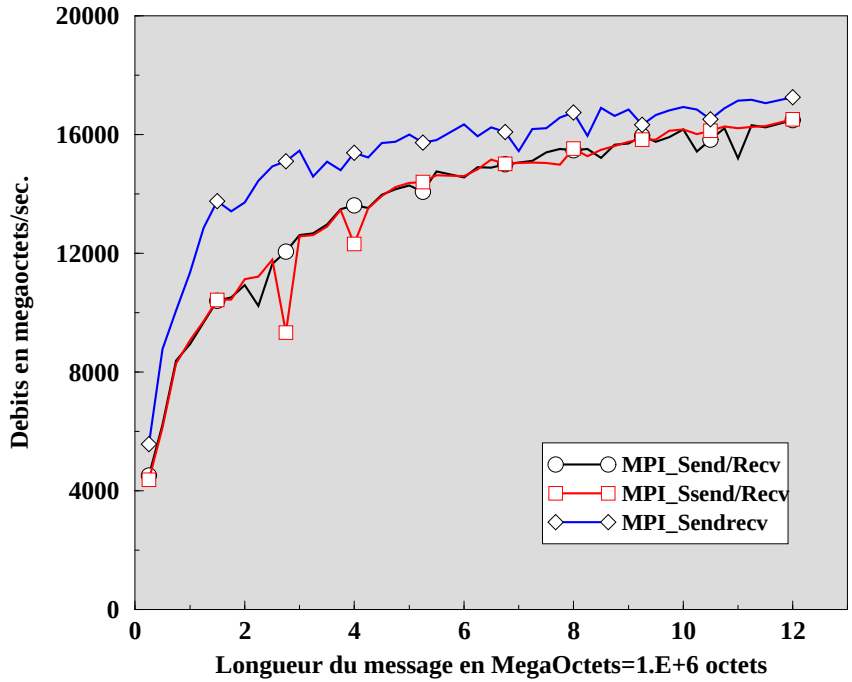


(b) `NEC SX-5` (intra-nœud)

FIG. 24 – *Débits (messages courts)*



(a) CRAY T3E-600



(b) NEC SX-5 (intra-nœud)

FIG. 25 – *Débits (messages longs)*

5.9 – Communications persistantes

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 :

1. créer un schéma persistant de communication une fois pour toute (à l'extérieur de la boucle) ;
2. activer réellement la requête d'envoi ou de réception dans la boucle ;
3. libérer, si nécessaire, la requête en fin de boucle.

envoi *standard*	`MPI_SEND_INIT()`
envoi *synchroneous*	`MPI_SSEND_INIT()`
envoi *buffered*	`MPI_BSEND_INIT()`
réception *standard*	`MPI_RECV_INIT()`

Reprenons le programme modèle...

```
23  if (rang == 0) then
24    do k = 1, 1000
25      !*** J'envoie un gros message
26      call MPI_ISSEND(c,m*m,MPI_REAL,1,etiquette,MPI_COMM_WORLD,requete0,code)
27      !*** Je calcule : factorisation LU avec LAPACK
28      call sgetrf(na, na, a, na, pivota, code)
29      call MPI_WAIT(requete0,statut,code)
30      !*** Ce calcul modifie le contenu du tableau C
31      c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
32    end do
33  elseif (rang == 1) then
34    do k = 1, 1000
35      !*** Je calcule
36      call sgetrf(na, na, a, na, pivota, code)
37      !*** Je reçois le gros message
38      call MPI_IRECV(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,requete1,code)
39      !*** Ce calcul est indépendant du message
40      call sgetrf(nb, nb, b, nb, pivotb, code)
41      call MPI_WAIT(requete1,statut,code)
42      !*** Ce calcul dépend du message précédent
43      a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
44    end do
45  end if
```

```
> mpirun -np 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 considérable 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      !*** J'envoie un gros message
27      call MPI_START(requete0,code)
28      call sgetrf(na, na, a, na, pivota, code)
29      call MPI_WAIT(requete0,statut,code)
30      !*** Ce calcul modifie le contenu du tableau C
31      c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
32    end do
33  elseif (rang == 1) then
34    call MPI_RECV_INIT(c,m*m,MPI_REAL,0,etiquette,MPI_COMM_WORLD,statut,requete1,code)
35    do k = 1, 1000
36      call sgetrf(na, na, a, na, pivota, code)
37      !*** Je reçois le gros message
38      call MPI_START(requete1,code)
39      !*** Ce calcul est indépendant du message
40      call sgetrf(nb, nb, b, nb, pivotb, code)
41      call MPI_WAIT(requete1,statut,code)
42      !*** Ce calcul dépend du message précédent
43      a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
44    end do
45  end if
```

```
> mpirun -np 2 AOptimiser
Temps :  235 secondes
```

L'infrastructure matérielle du *T3E* et du NEC SX-5 ne permet pas une implémentation 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.

5.10 – Conseils 2

- Minimiser les surcoûts induits par des appels répétitifs aux sous-programmes de communication en utilisant une fois pour toute un schéma persistant de communication et activer celui-ci autant de fois qu'il est nécessaire à l'aide du sous-programme `MPI_START()`.
- Recouvrir les communications par des calculs tout en évitant la recopie temporaire des messages car un schéma persistant (ex. `MPI_SSEND_INIT()`) est forcément activé d'une façon **non bloquante** à l'appel du sous-programme `MPI_START()`.

6 – Types de données dérivés

6.1 – Introduction

Dans les communications point à point, 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_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()`.

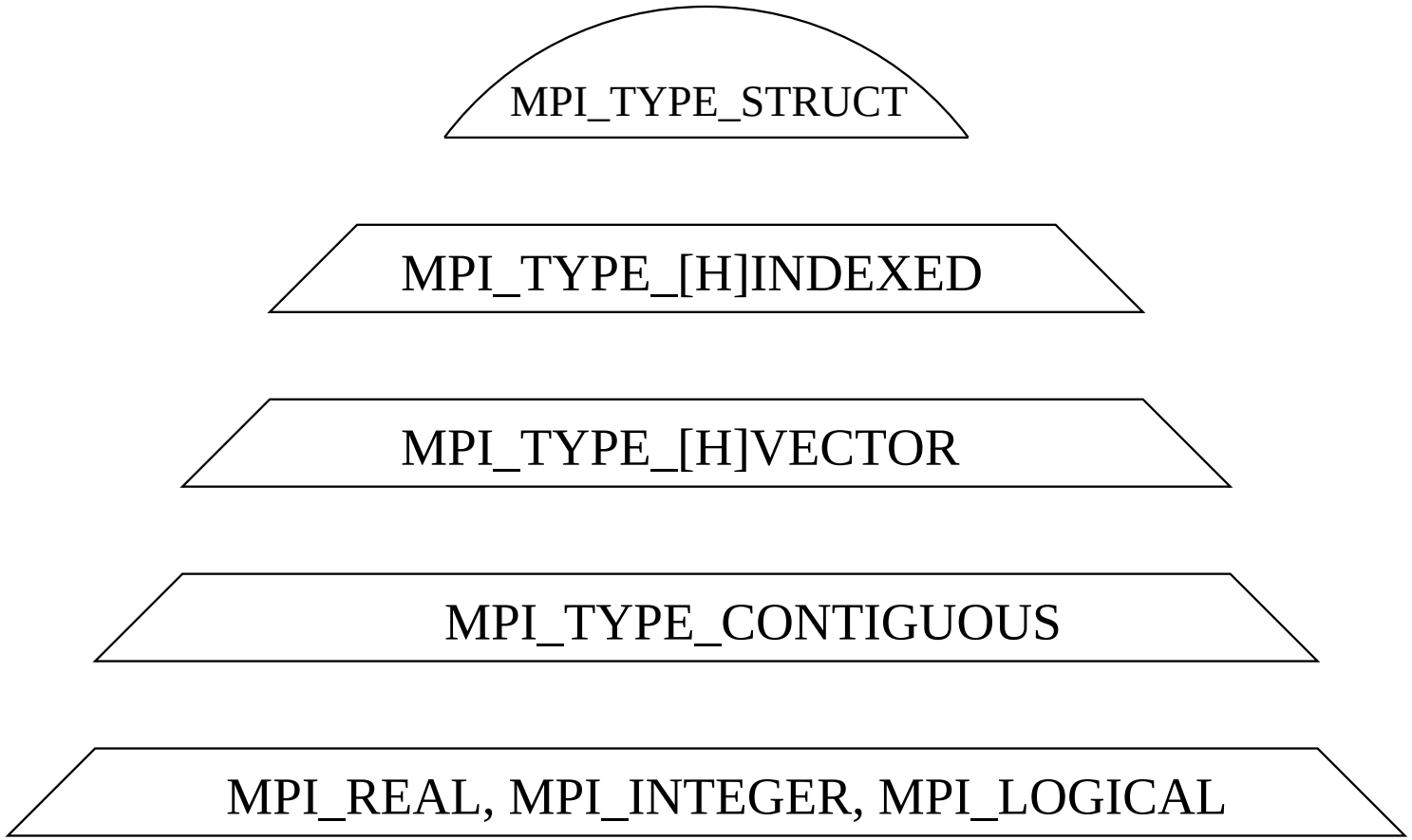


FIG. 26 – *Hiérarchie des constructeurs de type MPI*

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. 27 – *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. 28 – *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_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.
  Le pas ne peut plus alors être exprimé en nombre d'éléments du type générique.

6.4 – Descriptif des sous-programmes

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

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

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

call MPI_TYPE_COMMIT(nouveau_type,code)
```

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

call MPI_TYPE_FREE(nouveau_type,code)
```

6.5 – Exemples

```
1  program colonne
2    implicit none
3    include 'mpif.h'
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    implicit none
3    include 'mpif.h'
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    implicit none
3    include 'mpif.h'
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_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_HINDEXED()`.
- Attention à la **portabilité** avec `MPI_TYPE_HINDEXED()` !

nb=3, longueurs_blocs=(2,1,3), déplacements=(0,3,7)

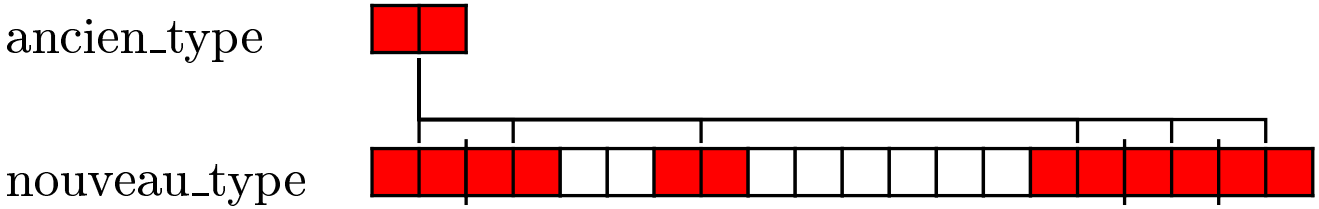


FIG. 29 – *Le constructeur MPI_TYPE_INDEXED*

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

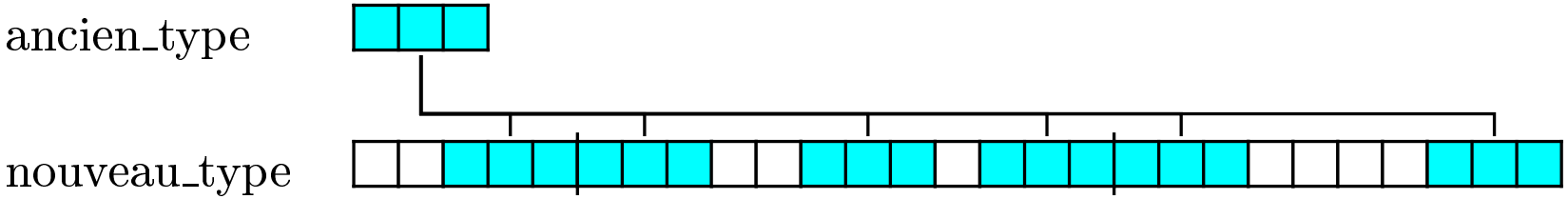


FIG. 30 – *Le constructeur MPI_TYPE_HINDEXED*

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

integer,intent(out)                 :: nouveau_type,code

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

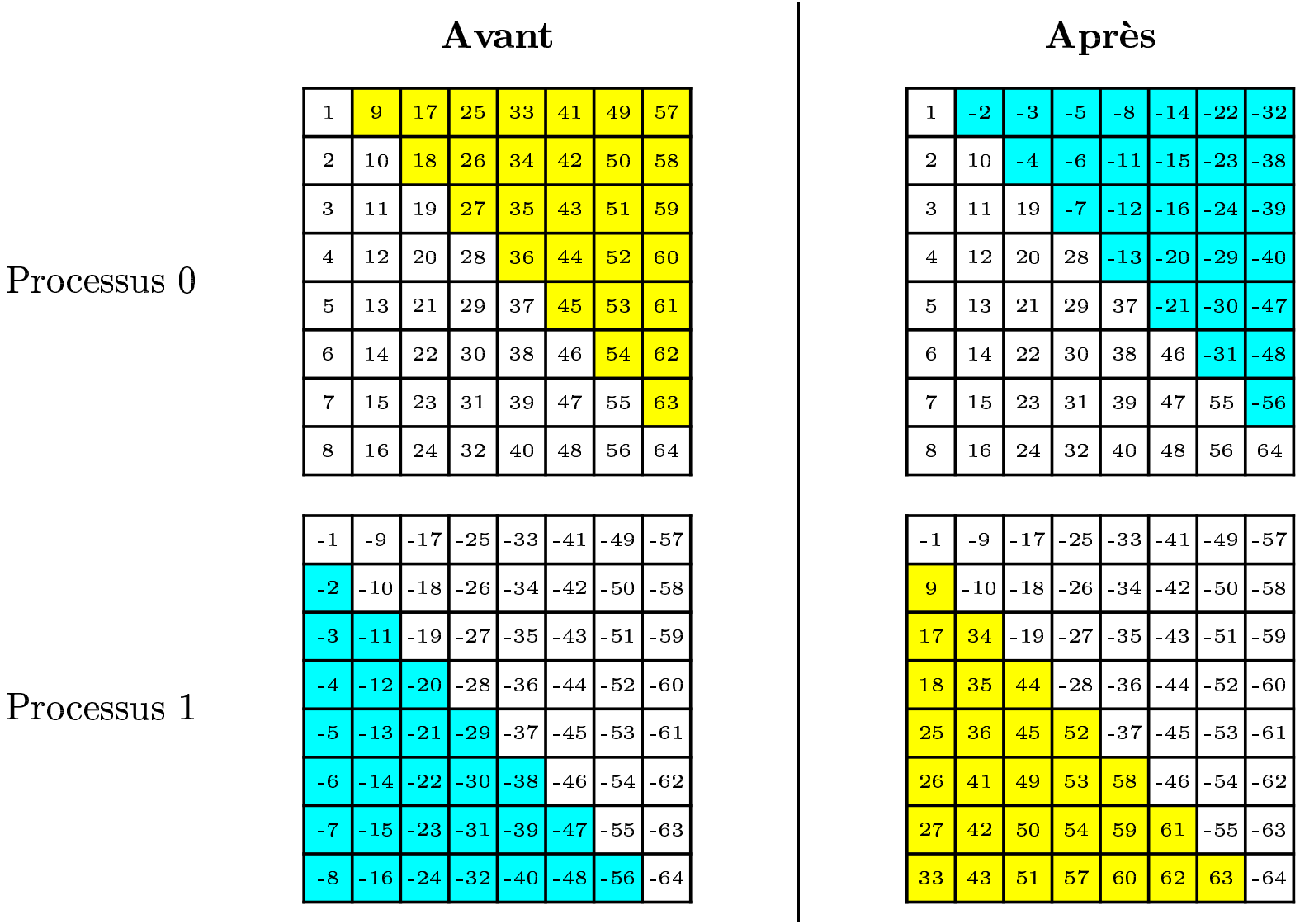
Dans l'exemple suivant, chacun des deux processus :

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

Processus 0 — 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 — 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

Processus 1 — Après							
-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. 31 – *Échanges entre les 2 processus*

```
1  program triangle
2  implicit none
3  include 'mpif.h'
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 – Types hétérogènes

- Le sous-programme `MPI_TYPE_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_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_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)

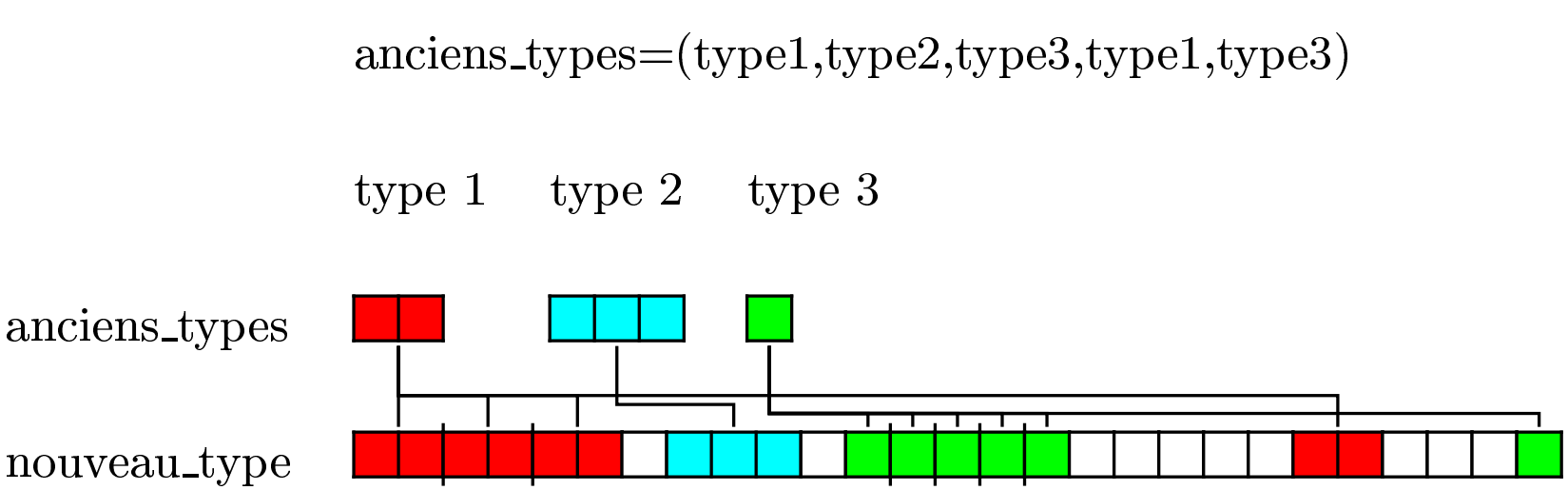


FIG. 32 – *Le constructeur MPI_TYPE_STRUCT*

```
integer,intent(in)                  :: nb
integer,intent(in),dimension(nb) :: longueurs_blocs
integer,intent(in),dimension(nb) :: deplacements
integer,intent(in),dimension(nb) :: anciens_types

integer, intent(out)                :: nouveau_type,code

call MPI_TYPE_STRUCT(nb,longueurs_blocs,deplacements,anciens_types,nouveau_type,code)
```

```
<type>,intent(in)   :: variable

integer,intent(out) :: adresse_variable,code

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

```
1  program Interaction_Particules
2    implicit none
3    include 'mpif.h'
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,adresses,deplacements
9
10   type Particule
11      character(len=5)                  :: type
12      integer                           :: masse
13      real,dimension(3)                 :: coords
14      logical                           :: classe
15   end type Particule
16   type(Particule),dimension(n)         :: p,temp_p
17
18     call MPI_INIT(code)
19     call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
20
21     ! Construction du type de données
22     types = (/MPI_CHARACTER,MPI_INTEGER,MPI_REAL,MPI_LOGICAL/)
23     longueurs_blocs = (/5,1,3,1/)
```

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

6.8 – Sous-programmes annexes

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

  ✓ Attention à ne pas le confondre avec `MPI_TYPE_EXTENT()` qui, lui, renvoie la taille d'un type **aligné** en mémoire.

- Les bornes inférieures et supérieures d'un type de données relativement à l'origine : `MPI_TYPE_LB()` et `MPI_TYPE_UB()`

- `MPI_TYPE_SIZE()` $\leq$ `MPI_TYPE_UB()` − `MPI_TYPE_LB()`

- `MPI_LB`, `MPI_UB` sont des *pseudo* types de données (de taille nulle) permettant un alignement correct en mémoire au sein d'une structure créée avec `MPI_TYPE_STRUCT()`

```
integer,intent(in)  :: type_donnee

integer,intent(out) :: taille,code

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

```
integer, intent(in)  :: type_donnee

integer, intent(out) :: adresse,code

call MPI_TYPE_LB(type_donnee,adresse,code)
call MPI_TYPE_UB(type_donnee,adresse,code)
```

6.9 – 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 au chapitre suivant) fait de *MPI* l'outil idéal pour tous les problèmes de décomposition de domaines avec des maillages réguliers ou irréguliers.

7 – Topologies

7.1 – Introduction

- Dans la plupart des applications, plus particulièrement dans les méthodes de décomposition de domaine, il est intéressant de pouvoir disposer les processus suivant une topologie régulière.
- *MPI* permet de définir des topologies virtuelles du type cartésien ou graphe.

7.2 – Topologies de processus

- 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.

7.3 – 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.

```
include 'mpif.h'
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.`, MPI choisit l'ordre des processus.

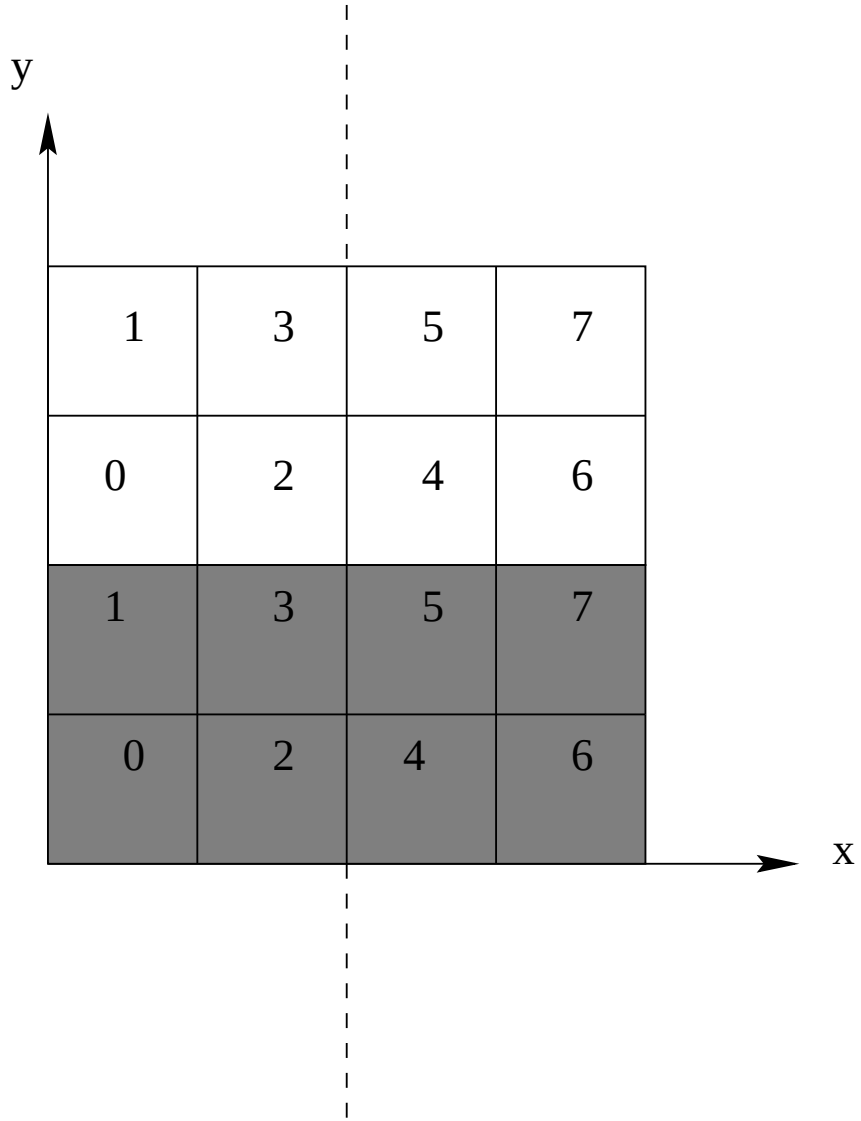


FIG. 33 – *Topologie cartésienne 2D périodique en y*

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

```
include 'mpif.h'
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

$z = 1$

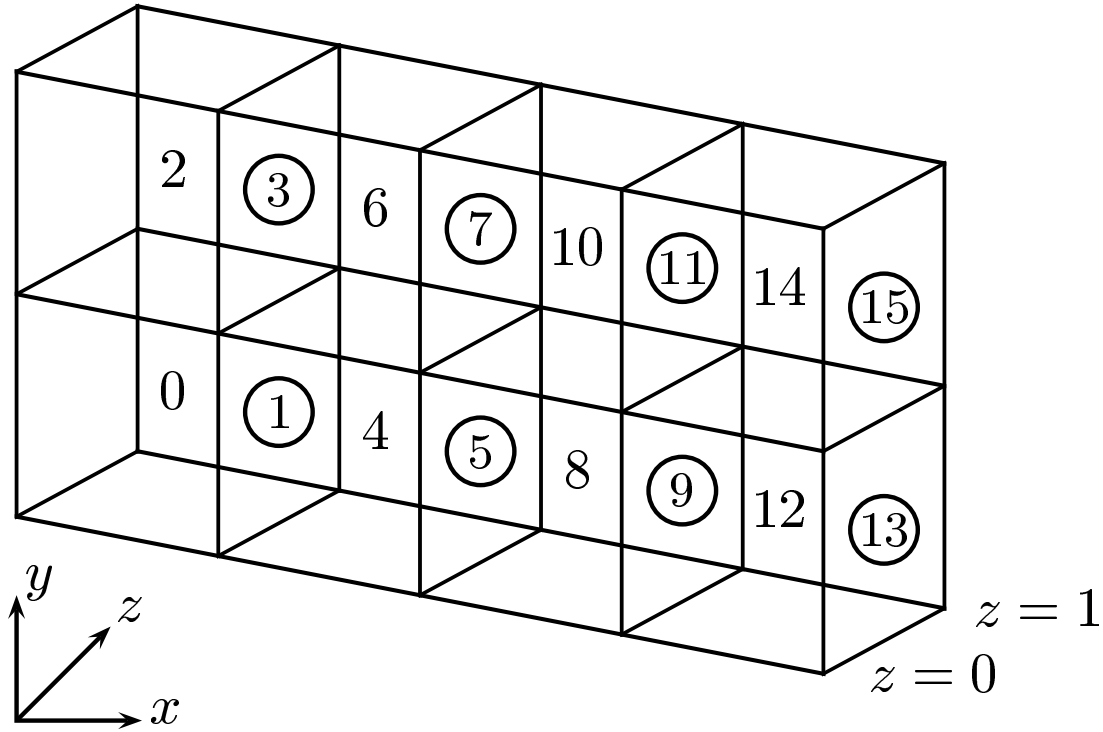


FIG. 34 – *Topologie cartésienne 3D non périodique*

☞ 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)                          :: nnoeuds, ndims
integer, dimension(ndims),intent(inout) :: dims
integer, intent(out)                         :: code

call MPI_DIMS_CREATE(nnoeuds,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)
```

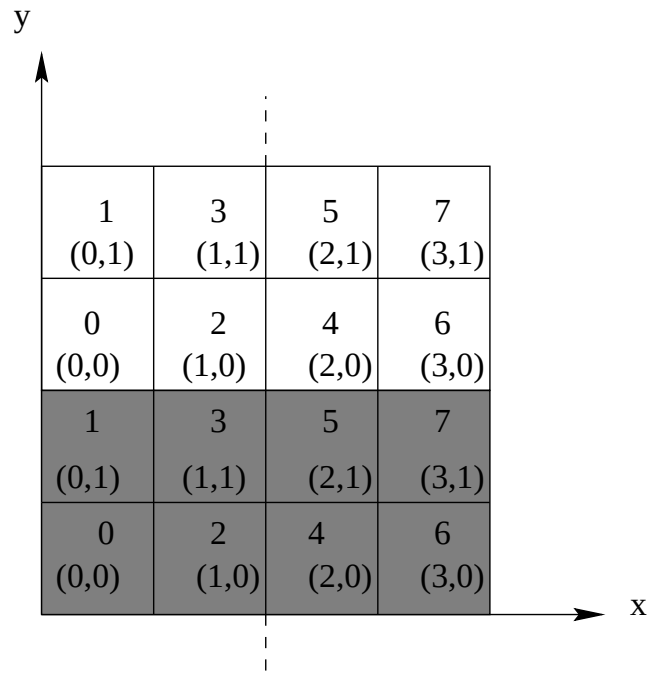


FIG. 35 – *Topologie cartésienne 2D périodique en y*

```
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)
```

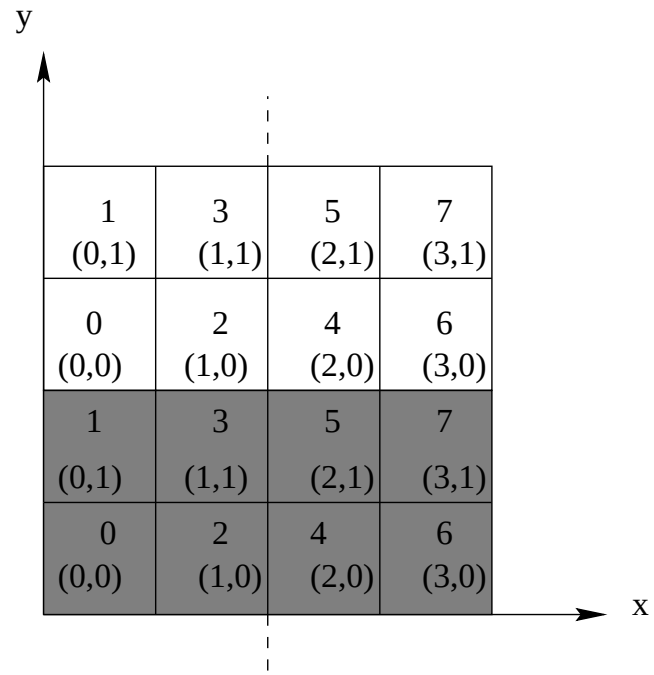


FIG. 36 – *Topologie cartésienne 2D périodique en y*

```
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.

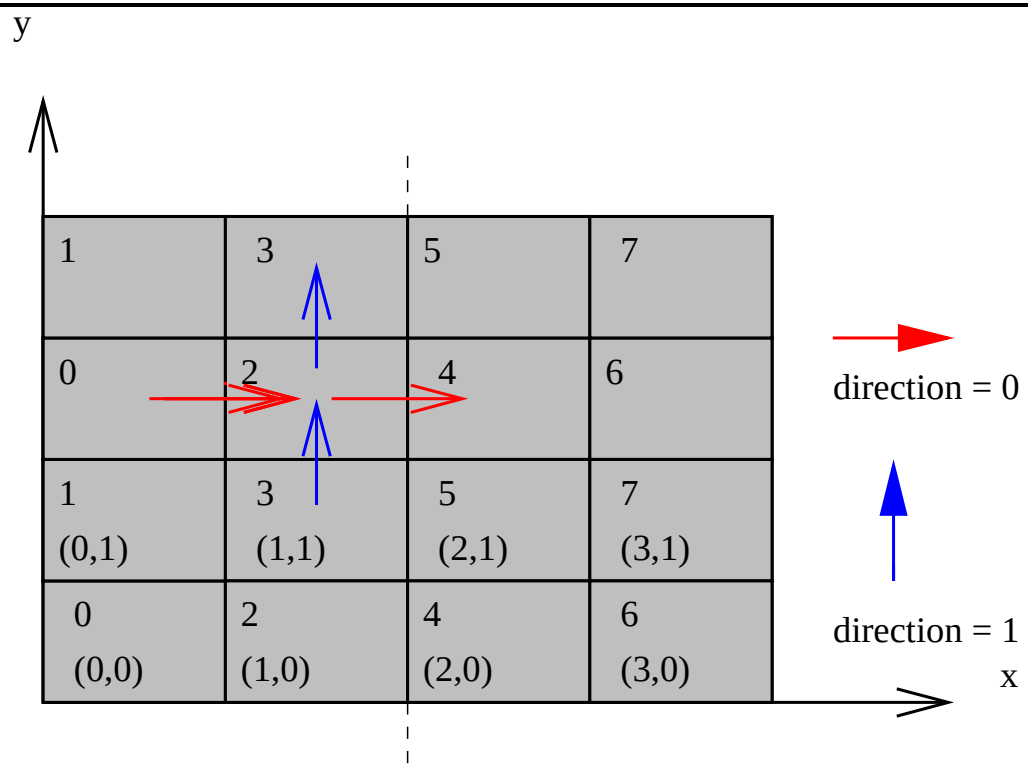


FIG. 37 – *Appel du sous-programme MPI_CART_SHIFT()*

```
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
```

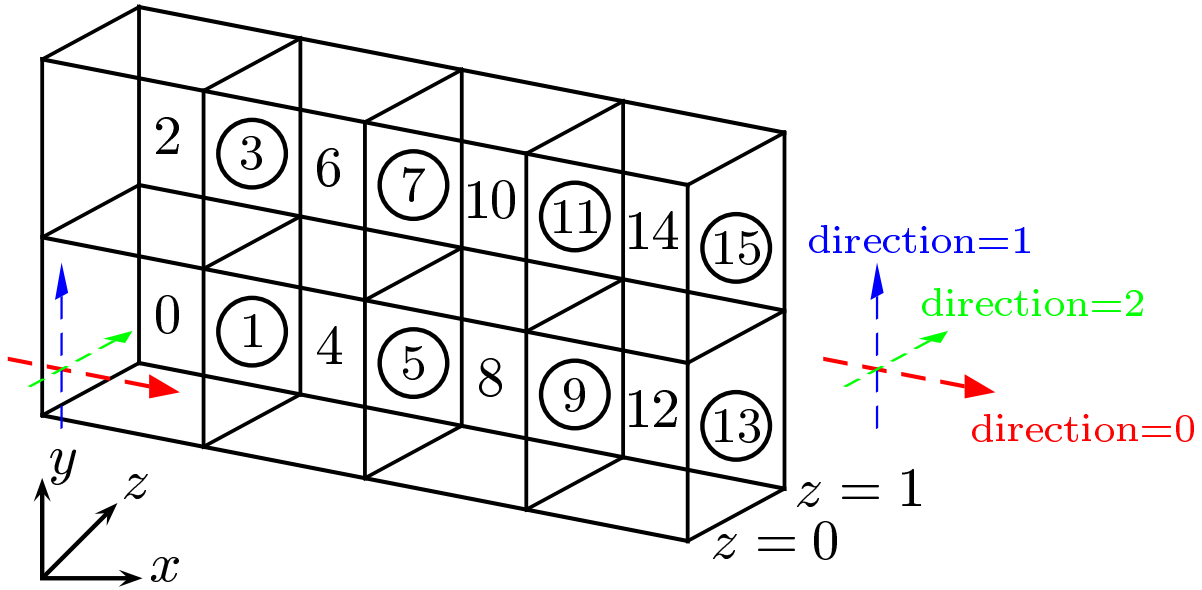


FIG. 38 – *Appel du sous-programme MPI_CART_SHIFT()*

```
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    implicit none
3    include 'mpif.h'
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)
```

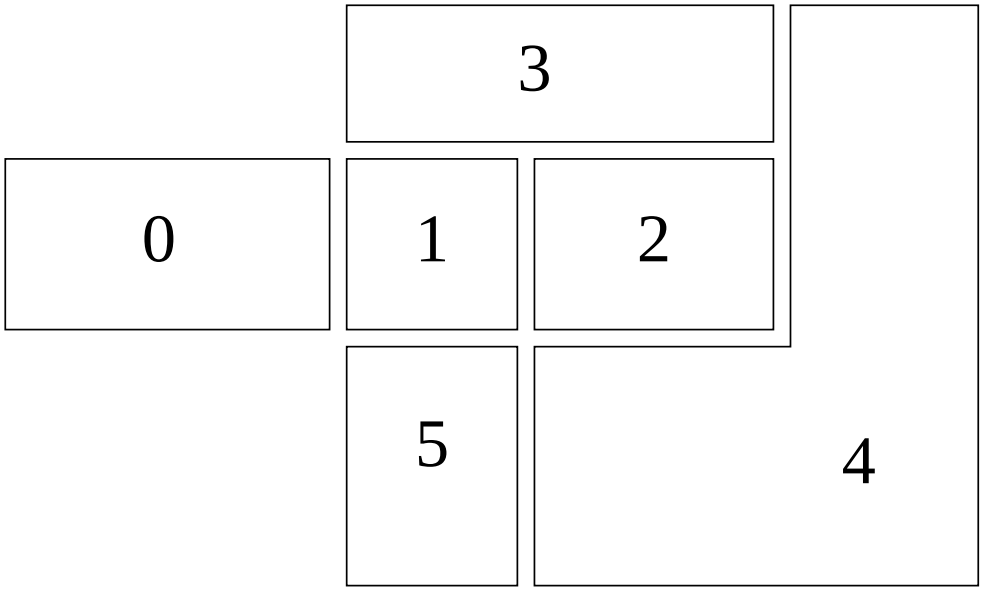
```
22    ! Création grille 2D periodique en y
23    periods(1) = .false.
24    periods(2) = .true.
25    reorganisation = .false.
26
27    call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_2D,code)
28
29    ! Initialisation du tableau voisin à la valeur MPI_PROC_NULL
30    voisin(:)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

7.4 – 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_GRAPH_CREATE() permet alors de définir une topologie de type graphe en indiquant les voisins de chaque sous-domaine.

```
1 integer, intent(in)                                  :: comm_ancien,nb_procs
2 integer, dimension(nb_procs),intent(in)              :: index
3 logical, dimension(nb_voisins_max),intent(in) :: liste_voisins
4 logical, intent(in)                                  :: reorganisation
5
6 integer, intent(out)                                 :: comm_nouveau, code
7
8 call MPI_GRAPH_CREATE(comm_ancien,nb_procs,index,liste_voisins,reorganisation, &
9                       comm_nouveau,code)
```

Les tableaux d'entiers `index` et `liste_voisins` permettent de définir la liste des voisins 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. 39 – *Graphe de processus*

```
index   = (/    1,       5,     8,     11,    14, 16 /)
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
logical, 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 :

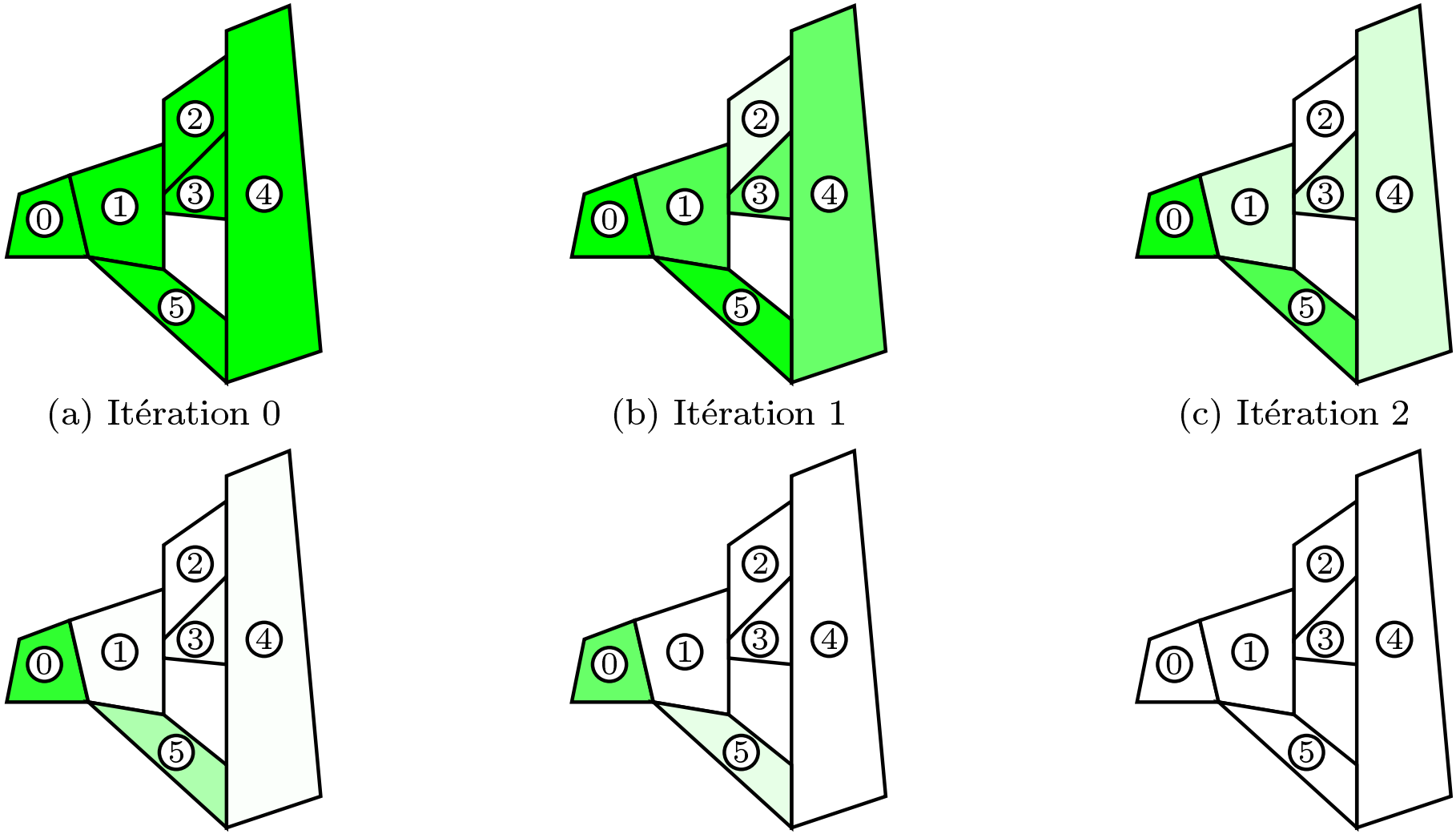
```
integer, intent(in)                                  :: comm_nouveau
integer, intent(in)                                  :: rang
integer, intent(in)                                  :: nb_voisins
logical, dimension(nb_voisins_max), intent(out) :: voisins
integer, intent(out)                    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

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

```
1  program graphe
2
3    implicit none
4    include 'mpif.h'
5
6    integer                                 :: rang,code,comm_graphe,nb_voisins,i,iteration=0
7    integer, parameter                      :: etiquette=100
8    integer, dimension(6)                   :: index
9    integer, dimension(16)                  :: liste_voisins
10   integer, allocatable,dimension(:)       :: voisins
11   integer, dimension(MPI_STATUS_SIZE)     :: statut
12
13   real                                    :: propagation, & ! Propagation du feu
14                                                             ! depuis les voisins
15                                              feu=0.,      & ! Valeur du feu
16                                              bois=1.,     & ! Rien n'a encore brûlé
17                                              arret=1.       ! Tout a brûlé si arret <= 0.01
18
19   call MPI_INIT(code)
20
21   ! On définit les voisins de chacune des parcelles
22   index         =( /1,   5,       8,     11,    14,  16  /)
23   liste_voisins =( /1, 0,5,2,3, 1,3,4, 1,2,4, 3,2,5, 1,4 /)
24
25   call MPI_GRAPH_CREATE(MPI_COMM_WORLD,6,index,liste_voisins,.false.,comm_graphe,code)
26   call MPI_COMM_RANK(comm_graphe,rang,code)
27
```

```
28    if (rang == 2) feu=1.          ! Le feu se déclare arbitrairement sur la parcelle 2
29
30    call MPI_GRAPH_NEIGHBORS_COUNT(comm_graphe,rang,nb_voisins,code)
31    allocatable(voisins(nb_voisins)) ! Allocation du tableau voisins
32    call MPI_GRAPH_NEIGHBORS(comm_graphe,rang,nb_voisins,voisins,code)
33
34    do while (arret > 0.01)        ! On arrête dès qu'il n'y a plus rien à brûler
35
36       do i=1,nb_voisins           ! On propage le feu aux voisins
37          call MPI_SENDRECV(minval((/1.,feu/)),1,MPI_REAL,voisins(i),etiquette, &
38                            propagation,       1,MPI_REAL,voisins(i),etiquette, &
39                            comm_graphe,statut,code)
40          ! Le feu se développe en local sous l'influence des voisins
41          feu=1.2*feu + 0.2*propagation*bois
42          bois=bois/(1.+feu)       ! On calcule ce qui reste de bois sur la parcelle
43       end do
44
45       call MPI_ALLREDUCE(bois,arret,1,MPI_REAL,MPI_SUM,comm_graphe,code)
46
47       iteration=iteration+1
48       print '("Itération ",i2," parcelle ",i2," bois=",f5.3)',iteration,rang,bois
49       call MPI_BARRIER(comm_graphe,code)
50       if (rang  == 0) print '("--")'
51    end do
52    deallocate(voisins)
53    call MPI_FINALIZE(code)
54 end program graphe
```

```
> mpirun -np 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. 40 – *Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt*

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.

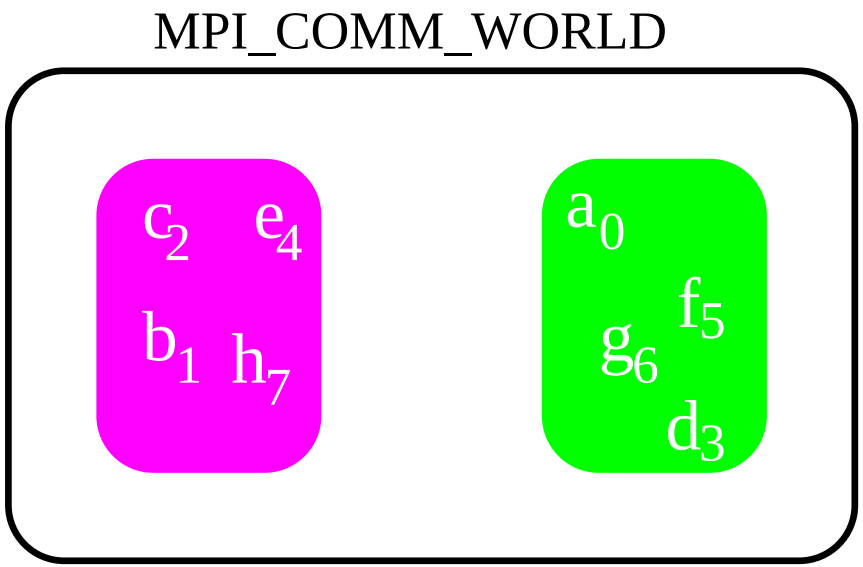


FIG. 41 – *Partitionnement d'un communicateur*

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 le fichier d'en-tête `mpif.h`.
- 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.
- Par défaut, il fixe donc **la portée** des communications point à point et collectives **à tous les processus** de l'application.

Dans cet exemple, le processus 2 diffuse un message contenant un vecteur "a" à tous les processus du communicateur MPI_COMM_WORLD (donc de l'application) :

```
1  program monde
2   implicit none
3   include 'mpif.h'
4
5   integer, parameter :: m=16
6   integer            :: rang,nb_procs,code
7   real, dimension(m) :: a
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  a(:)=0.
13  if(rang == 2) a(:)= 2.
14
15  call MPI_BCAST(a,m,MPI_REAL,2,MPI_COMM_WORLD,code)
16
17  call MPI_FINALIZE(code)
18
19 end program monde
```

```
> mpirun -np 8 monde
```

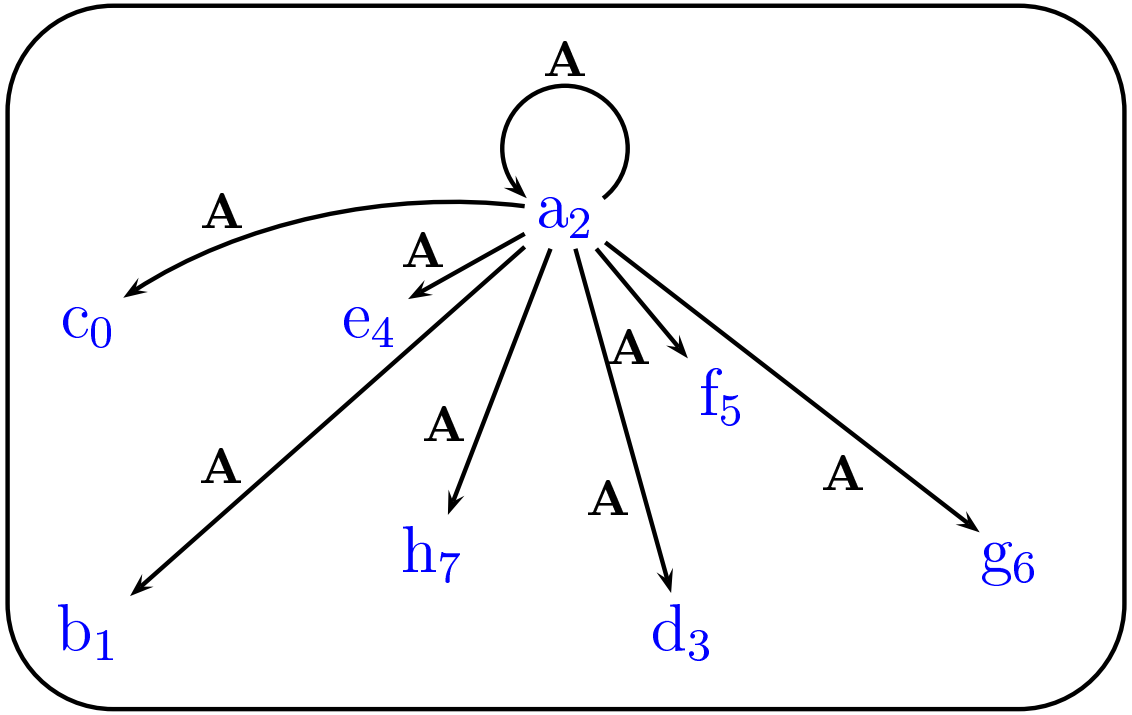


FIG. 42 – *Le communicateur par défaut*

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.

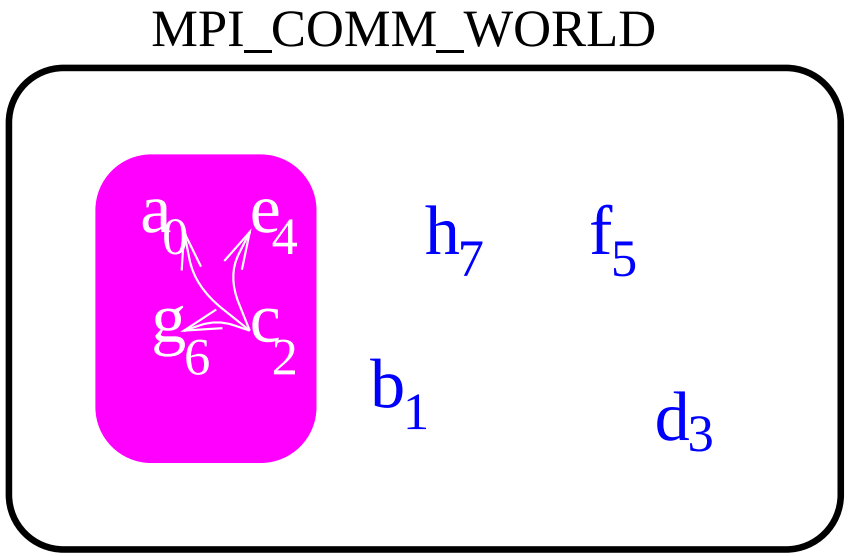


FIG. 43 – *Un nouveau communicateur*

8.3 – Groupes et communicateurs

Dans la bibliothèque *MPI*, il existe des sous-programmes pour :

① construire des groupes de processus ;

- ☞ `MPI_GROUP_INCL()` ;
- ☞ `MPI_GROUP_EXCL()`.

② construire des communicateurs :

- ☞ `MPI_CART_CREATE()` ;
- ☞ `MPI_CART_SUB()` ;
- ☞ `MPI_COMM_CREATE()` ;
- ☞ `MPI_COMM_DUP()` ;
- ☞ `MPI_COMM_SPLIT()`.

- Les **constructeurs de groupes** sont des **opérateurs locaux** aux processus du groupe (qui n'engendrent pas de communications entre les processus).
- Les **constructeurs de communicateurs** sont des **opérateurs collectifs** (qui engendrent des communications entre les processus).
- Les groupes et les communicateurs que le programmeur crée peuvent être gérés dynamiquement. De même qu'il est possible de les créer, il est possible de les détruire : `MPI_GROUP_FREE()`, `MPI_COMM_FREE()`.

Un communicateur est constitué :

① d'un **groupe** de processus ;

② d'un **contexte** de communication mis en place à l'appel du sous-programme de construction du communicateur.

Sachant que :

☞ le **groupe** constitue un ensemble de processus ;

☞ le **contexte** de communication 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 :

❶ par l'intermédiaire d'un groupe de processus ;

❷ directement à partir d'un autre communicateur.

8.4 – Communicateur issu d'un groupe

- Un groupe est un ensemble ordonné de N processus.
- Chaque processus du groupe est identifié par un entier 0, 1, ..., $N-1$ appelé **rang**.
- Initialement, tous les processus sont membres d'un groupe associé au communicateur par défaut `MPI_COMM_WORLD`.
- Des sous-programmes *MPI* (`MPI_GROUP_SIZE()`, `MPI_GROUP_RANK()`, etc.) permettent de connaître les attributs d'un groupe.
- Tout communicateur est associé à un groupe. Le sous-programme `MPI_COMM_GROUP()` permet de connaître le groupe auquel un communicateur est associé.

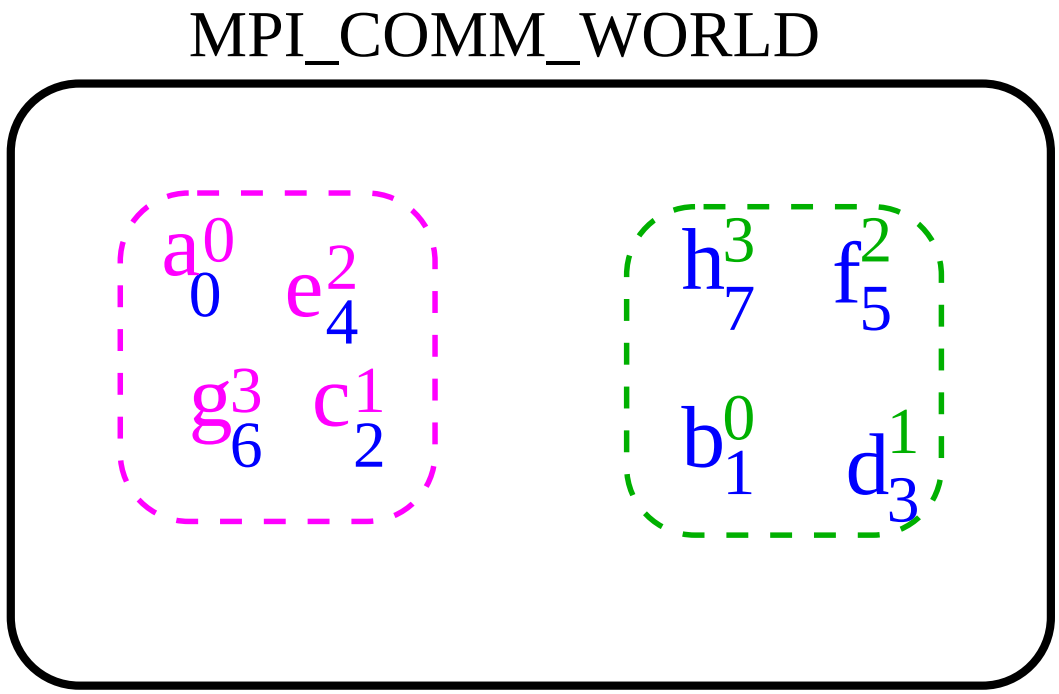


FIG. 44 – *Deux groupes de processus dans un communicateur*

Dans l'exemple qui suit, nous allons :

- ☞ regrouper les processus de rang pair dans un communicateur (comm_pair) et les processus de rang impair dans un autre (comm_impair) ;
- ☞ ne diffuser un message qu'aux processus de chacun de ces deux groupes.

```
1  program GroupePairImpair
2   include 'mpif.h'
3   integer, parameter                  :: m=16
4   integer                             :: i=0,dim_rangs_pair,rang,nb_procs,iproc,code, &
5                                          grp_monde,grp_pair,grp_impair
6   integer                             :: comm_pair,comm_impair
7   integer,dimension(:),allocatable    :: rangs_pair
8   real,dimension(m)                   :: a
9   integer                             :: rang_ds_monde,rang_ds_pair,rang_ds_impair
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  !*** Initialisation du vecteur A
16  a(:)=0.
17  if(rang == 2) a(:)=2.
18  if(rang == 3) a(:)=3.
19
20  !*** Enregistrer le rang des processus pairs
21  dim_rangs_pair = int((nb_procs+1)/2)
22  allocate(rangs_pair(dim_rangs_pair))
23  do iproc = 0, nb_procs-1, 2
24    i = i + 1
25    rangs_pair(i) = iproc
26  end do
```

```
27  !*** Connaître le groupe associé au communicateur MPI_COMM_WORLD
28  call MPI_COMM_GROUP(MPI_COMM_WORLD,grp_monde,code)
29
30  !*** Créer le groupe des processus pairs
31  call MPI_GROUP_INCL(grp_monde,dim_rangs_pair,rangs_pair,grp_pair,code)
32
33  !*** Créer le communicateur des processus pairs
34  call MPI_COMM_CREATE(MPI_COMM_WORLD,grp_pair,comm_pair,code)
35
36  !*** Créer le groupe des processus impairs
37  call MPI_GROUP_EXCL(grp_monde,dim_rangs_pair,rangs_pair,grp_impair,code)
38
39  !*** Créer le communicateur des processus impairs
40  call MPI_COMM_CREATE(MPI_COMM_WORLD,grp_impair,comm_impair,code)
```

```
41  if (mod(rang,2) == 0) then
42    !*** Trouver le rang du processus 2 dans le groupe pair
43    rang_ds_monde=2
44    call MPI_GROUP_TRANSLATE_RANKS(grp_monde,1,rang_ds_monde,grp_pair,rang_ds_pair,code)
45    !*** Diffuser le message seulement aux processus de rangs pairs
46    call MPI_BCAST(a,m,MPI_REAL,rang_ds_pair,comm_pair,code)
47    !*** Destruction du communicateur comm_pair
48    call MPI_COMM_FREE(comm_pair,code)
49  else
50    !*** Trouver le rang du processus 3 dans le groupe impair
51    rang_ds_monde=3
52    call MPI_GROUP_TRANSLATE_RANKS(grp_monde,1,rang_ds_monde,grp_impair,rang_ds_impair,&
53                                   code)
54    !*** Diffuser le message seulement aux processus de rangs impairs
55    call MPI_BCAST(a,m,MPI_REAL,rang_ds_impair,comm_impair,code)
56    !*** Destruction du communicateur comm_impair
57    call MPI_COMM_FREE(comm_impair,code)
58  end if
59
60  call MPI_FINALIZE(code)
61 end program GroupePairImpair
```

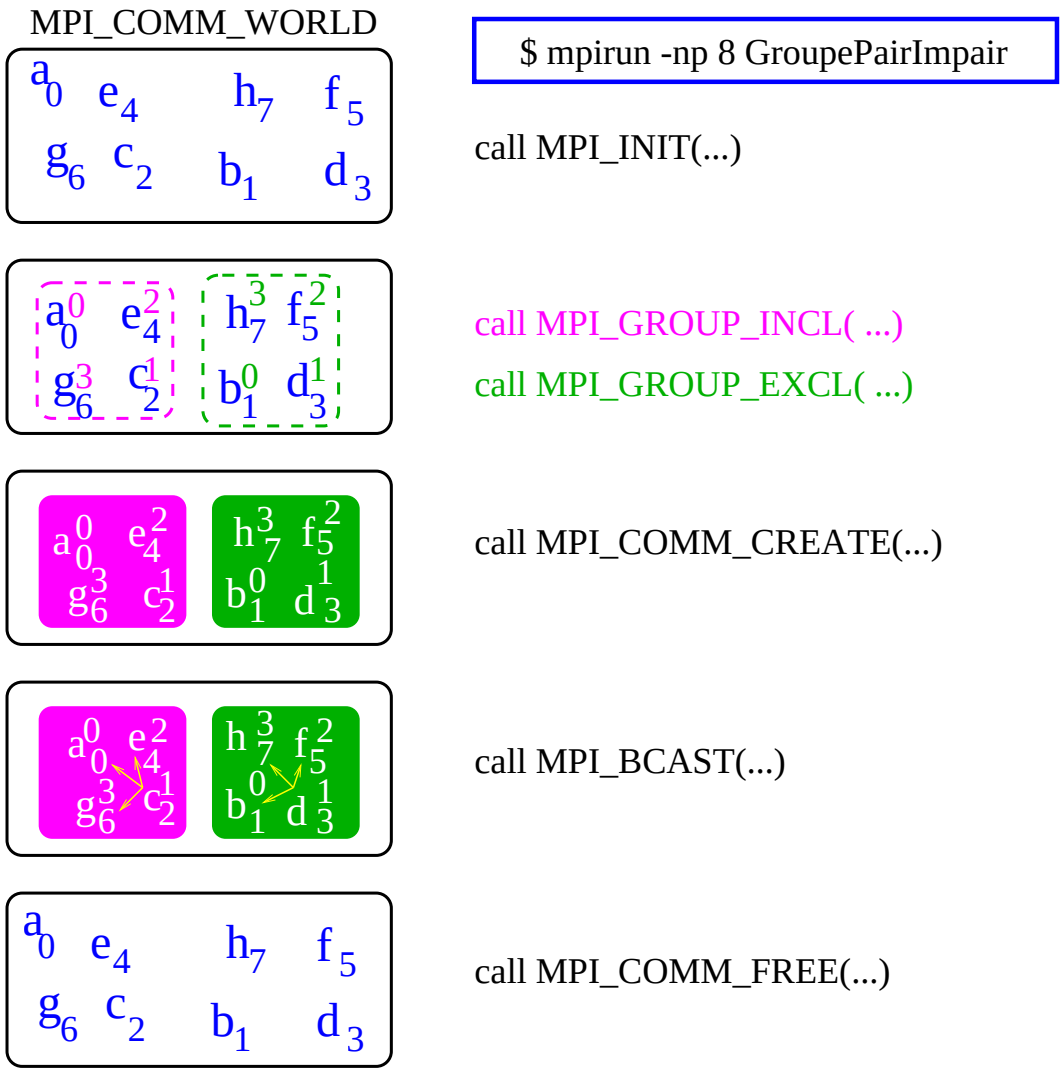


FIG. 45 – *Création/destruction des groupes et des communicateurs* `pair` *et* `impair`

Une fois les groupes constitués, il est possible :

- de les comparer :
  `MPI_GROUP_COMPARE`(group1,group2,`resultat`).
- d'appliquer des opérateurs ensemblistes sur deux groupes :
  `MPI_GROUP_UNION`(group1,group2,`nouveau_groupe`),
  `MPI_GROUP_INTERSECTION`(group1,group2,`nouveau_groupe`),
  `MPI_GROUP_DIFFERENCE`(group1,group2,`nouveau_groupe`)
  où `nouveau_groupe` peut être l'ensemble vide, auquel cas il prend la valeur
  `MPI_GROUP_EMPTY`.

8.5 – Communicateur issu d'un autre

Le programme précédent présente cependant quelques inconvénients. Nous allons le réécrire pour :

- ne pas nommer différemment ces deux communicateurs ;
- ne pas passer par les groupes pour construire les communicateurs `comm_pair` et `comm_impair` ;
- ne pas laisser le soin à *MPI* d'ordonner le rang des processus dans les communicateurs `comm_pair` et `comm_impair` ;
- éviter les tests conditionnels, en particulier lors de l'appel au sous-programme `MPI_BCAST()`.

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)
```

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

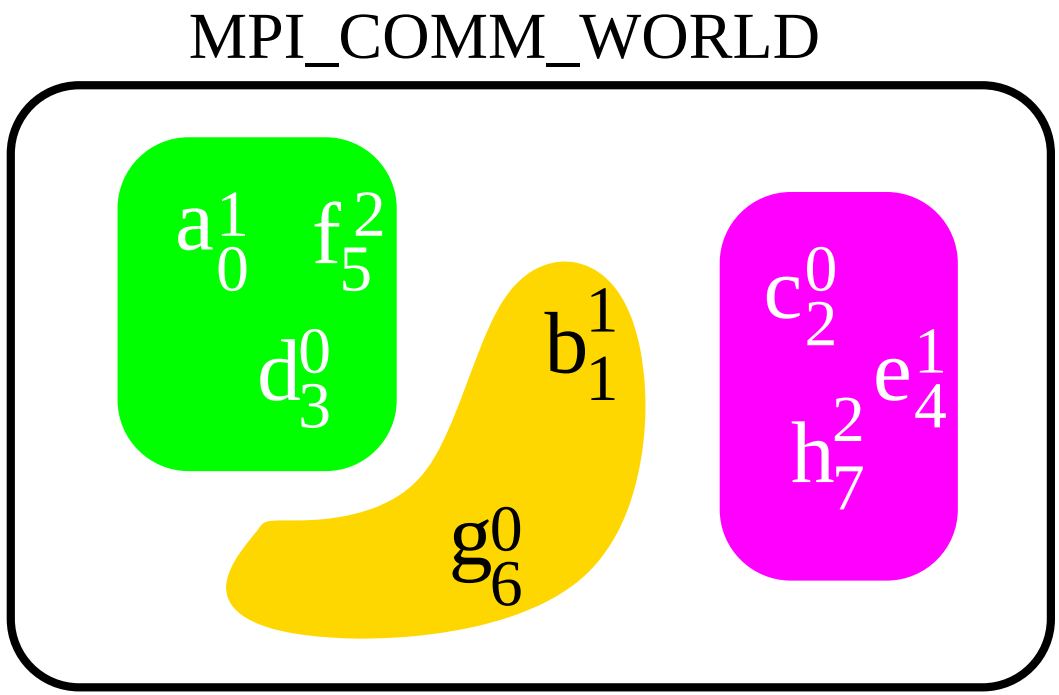


FIG. 46 – *Construction de communicateurs avec `MPI_COMM_SPLIT()`*

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 nos deux communicateurs **pair** et **impair** avec le constructeur `MPI_COMM_SPLIT()`...

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

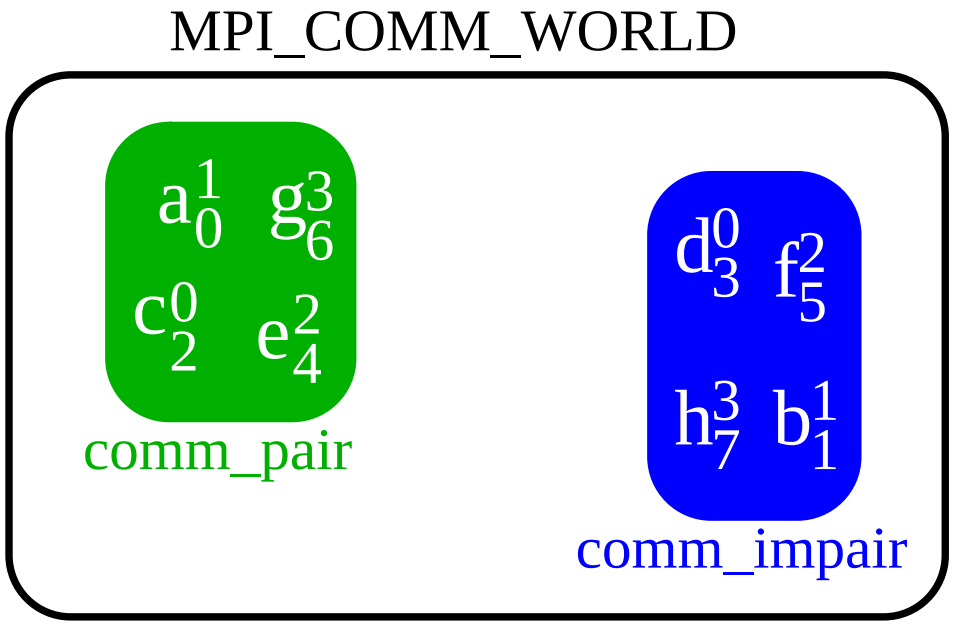


FIG. 47 – *Construction des communicateurs* ***pair*** *et* ***impair*** *avec* *MPI_COMM_SPLIT()*

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

```
1  program CommPairImpair
2   implicit none
3   include 'mpif.h'
4
5   integer, parameter :: m=16
6   integer            :: clef,couleur,comm
7   integer            :: nb_procs,rang,code
8   real, dimension(m) :: a
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  !*** Initialisation du vecteur A
15  a(:)=0.
16  if(rang == 2) a(:)=2.
17  if(rang == 3) a(:)=3.
```

```
18  if ( mod(rang,2) == 0 ) then
19    !*** Couleur et clefs des processus pairs
20    couleur = 0
21    if( rang == 2 ) then
22       clef = 0
23    else if ( rang == 0 ) then
24       clef = 1
25    else
26       clef = rang
27    end if
28  else
29    !*** Couleur et clefs des processus impairs
30    couleur = 1
31    if( rang == 3) then
32       clef = 0
33    else
34       clef = rang
35    end if
36  end if
37
38  !*** Créer les communicateurs pair et impair en leur donnant une même dénomination
39  call MPI_COMM_SPLIT(MPI_COMM_WORLD,couleur,clef,comm,code)
40  !*** Le processus 0 de chaque communicateur diffuse son message aux processus
41  !*** de son groupe
42  call MPI_BCAST(a,m,MPI_REAL,0,comm,code)
43  !*** Destruction des communicateurs
44  call MPI_COMM_FREE(comm,code)
45  call MPI_FINALIZE(code)
46 end program CommPairImpair
```

8.6 – 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.

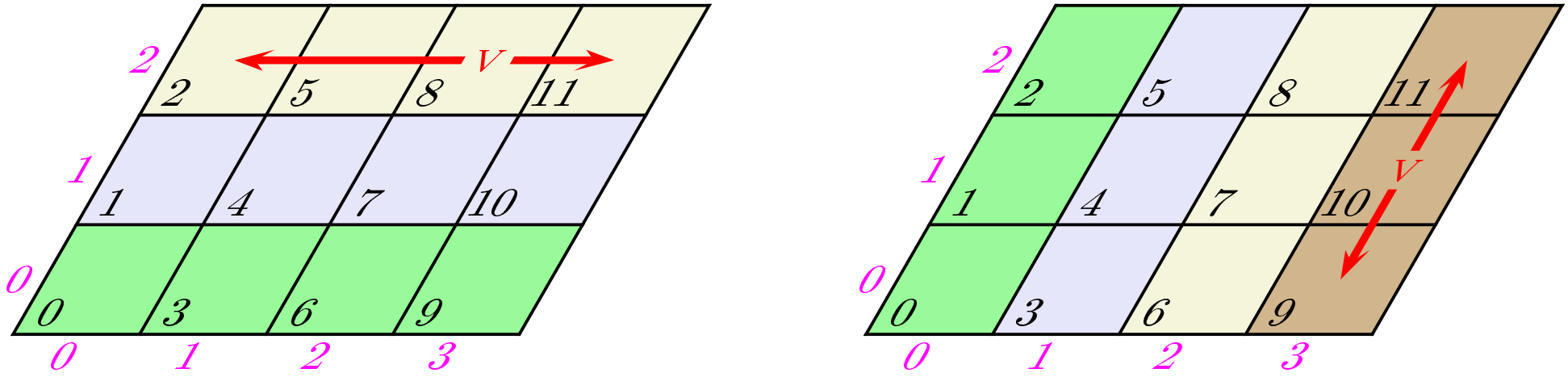


FIG. 48 – *Deux exemples de distribution de données dans une topologie 2D dégénérée*

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)
```

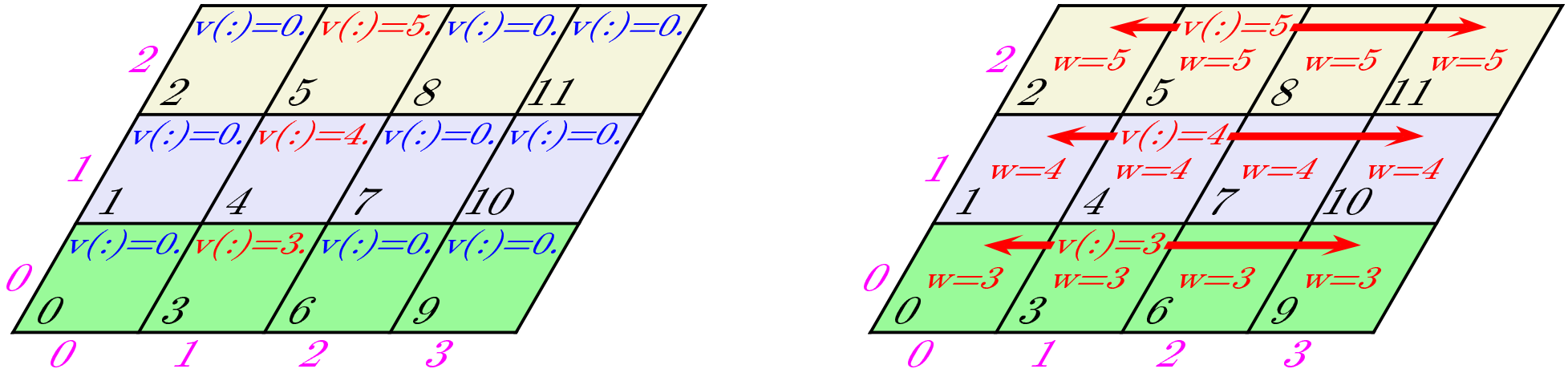


FIG. 49 – *Représentation initiale d'un tableau V dans la grille 2D et représentation finale après la distribution de celui-ci sur la grille 2D dégénérée*

```
1  program CommCartSub
2   implicit none
3   include 'mpif.h'
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  !*** Nous voulons que chaque ligne de la grille soit une topologie cartésienne 1D
27  Subdivision(1) = .true.
28  Subdivision(2) = .false.
29
30  !*** Subdivision de la grille cartésienne 2D.
31  call MPI_CART_SUB(Comm2D,Subdivision,Comm1D,code)
32  !*** Les processus de la colonne 2 distribuent le vecteur V aux processus
33  !*** 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
```

```
> mpirun -np 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.
```

8.7 – Intra et intercommunicateurs

- Les communicateurs que nous avons construits jusqu'à présent sont des **intracommunicateurs** (ex. `comm_pair` et `comm_impair`) 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.
- Une fois ce lien établi, seules sont possibles les communications point à point, les communications collectives au sein d'un **intercommunicateur** n'étant pas permises dans *MPI*-1 (cette limitation a disparu dans *MPI*-2).
- 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...

8.8 – Exemple récapitulatif

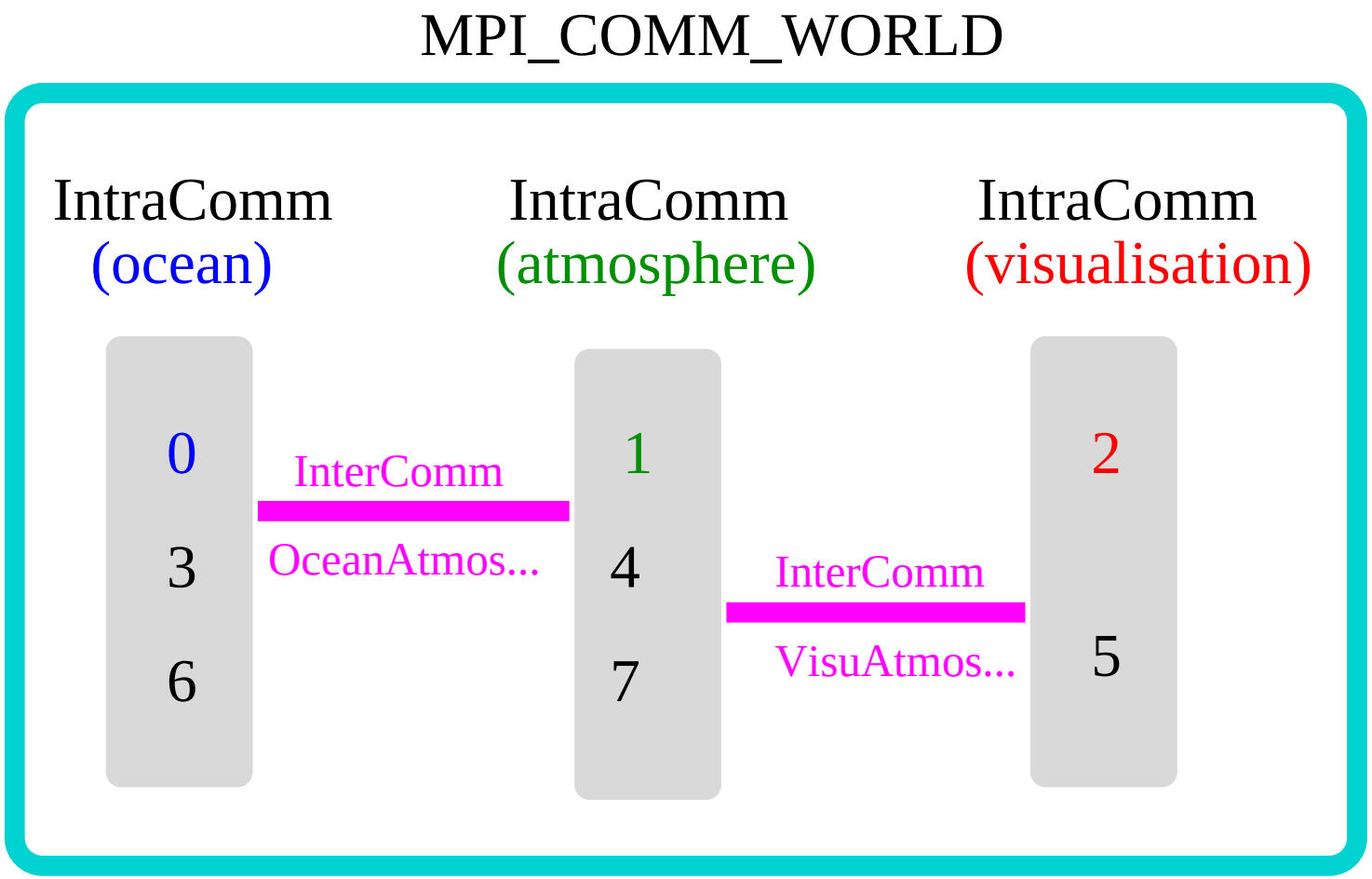


FIG. 50 – *Couplage océan/atmosphère*

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

```
17   !*** Construction des deux InterCommunicateurs et calcul
18   select case(couleur)
19    case(0)
20   !*** InterCommunicateur OceanAtmosphere pour que le groupe 0 communique
21   !*** avec le groupe 1
22     call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,1,tag1,CommOceanAtmosphere, &
23                               code)
24     call ocean(IntraComm,CommOceanAtmosphere)
25
26    case(1)
27    !*** InterCommunicateur OceanAtmosphere pour que le groupe 1 communique
28    !*** avec le groupe 0
29     call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,0,tag1,CommOceanAtmosphere, &
30                               code)
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  implicit none
44  include 'mpif.h'
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(c,n,MPI_REAL,rang,tag1,c,n,MPI_REAL,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  implicit none
74  include 'mpif.h'
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(c,n,MPI_REAL,rang,tag1,c,n,MPI_REAL,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  implicit none
113  include 'mpif.h'
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
```

8.9 – Conclusion

- **Groupes** et **contextes** définissent un objet appelé **communicateur**.
- Les communicateurs définissent la portée des communications.
- Ils sont utilisés pour dissocier les espaces de communication.
- Un communicateur doit être spécifié à l'appel de toute fonction d'échange de messages.
- Ils permettent d'éviter les confusions lors de la sélection des messages, par exemple au moment de l'appel à un sous-programme d'une bibliothèque scientifique qui elle-même effectue des échanges de messages.
- Les communicateurs offrent une programmation modulaire du point de vue de l'espace de communication. Dans le cadre de projets importants, chaque équipe développe son module sans se soucier du choix du communicateur des autres équipes (exemple : modèle couplé océan/atmosphère).

9 – Évolution de MPI : MPI-2

- Historique :
  - début des travaux en mars 1995 ;
  - brouillon présenté pour *SuperComputing 96* ;
  - version « officielle » disponible en juillet 1997 ;
  - voir `http://www.erc.msstate.edu/mpi/mpi2.html`
- Principaux domaines nouveaux :
  - gestion dynamique des processus :
    - possibilité de développer des codes MPMD ;
    - support multi plates-formes ;
    - démarrage et arrêt dynamique de sous-tâches ;
    - gestion de signaux système.
  - communications de mémoire à mémoire ;
  - entrées/sorties parallèles.

- Autres domaines où apparaissent des améliorations :
  - extensions concernant les intracommunicateurs ;
  - extensions concernant les intercommunicateurs ;
  - divers autres apports :
    - inter-opérabilité entre C et Fortran ;
    - interfaçage avec C++ et Fortran 90 (avec des limitations dans ce dernier cas).
- Extensions proposées en dehors de MPI-2 : IMPI (*Interoperable MPI*), MPI-RT (*real time extensions*)

TAB. 4 – *Changements de dénominations dans MPI-2*

Ancien nom	Nouveau nom
MPI_TYPE_HVECTOR()	MPI_TYPE_CREATE_HVECTOR()
MPI_TYPE_HINDEXED()	MPI_TYPE_CREATE_HINDEXED()
MPI_TYPE_STRUCT()	MPI_TYPE_CREATE_STRUCT()
MPI_ADDRESS()	MPI_GET_ADDRESS()
MPI_TYPE_EXTENT() MPI_TYPE_LB() MPI_TYPE_UB()	MPI_TYPE_GET_EXTENT()
MPI_LB MPI_UB	MPI_TYPE_CREATE_RESIZED()

- Il en va de même pour les versions C de ces sous-programmes.
- Les anciens noms sont toujours acceptés pour des raisons de compatibilité.