Programmation parallèle par échange de messages pour multiprocesseurs embarqués.

Jalel Chergui

Programmation parallèle par échange de messages pour multi-processeurs embarquées

Jalel Chergui
LIMSI-CNRS, bâtiment 508, BP 133
Université de Paris-Sud F-91403 Orsay Cedex
<Jalel.Chergui@limsi.fr>

Plan

Plan

1 – Introduction

1.1 – Définitions

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

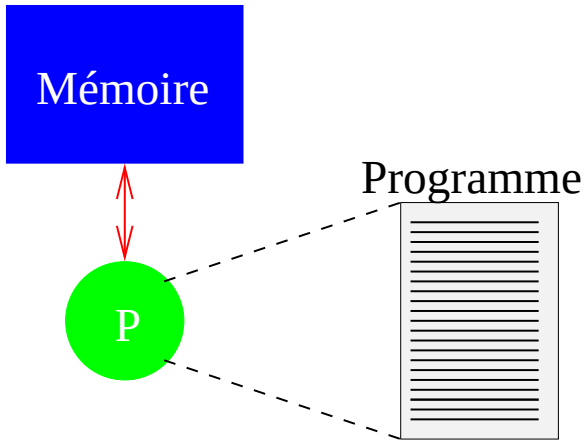


FIGURE 1 – Modèle séquentiel

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 allouée à chaque processus ;
- ☞ une donnée est échangée entre deux ou plusieurs processus via un appel, dans le programme, à des fonctions particulières.

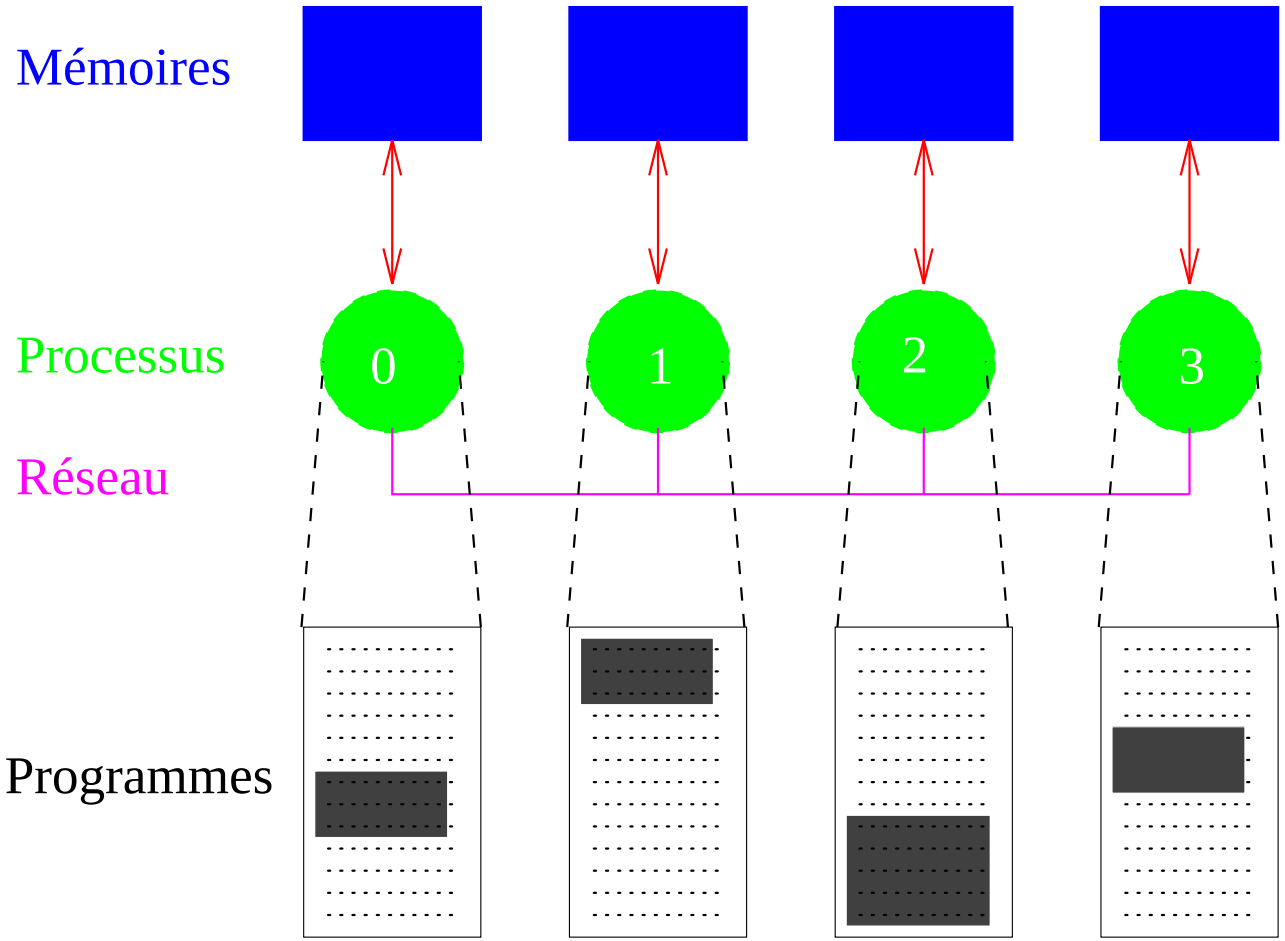


FIGURE 2 – Modèle à échange de messages

❸ Le modèle d'exécution *SPMD* :

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

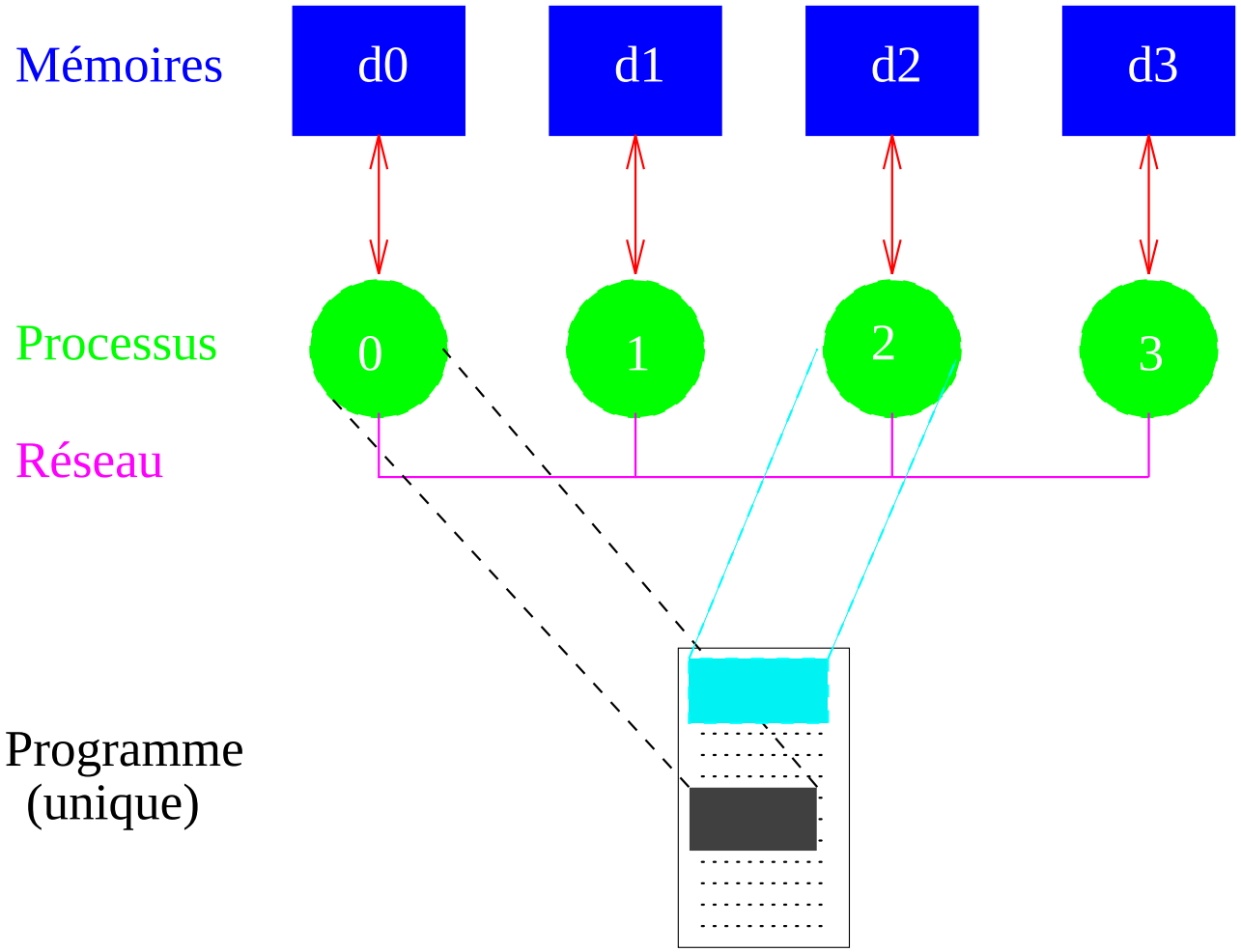


FIGURE 3 – *Single Program, Multiple Data*

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

```
int main(int argc, char *argv[])
{
  if (maître)
     chef(Arguments);
  else
     ouvriers(Arguments);
}
```

1.2 – Concepts de l'échange de messages

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

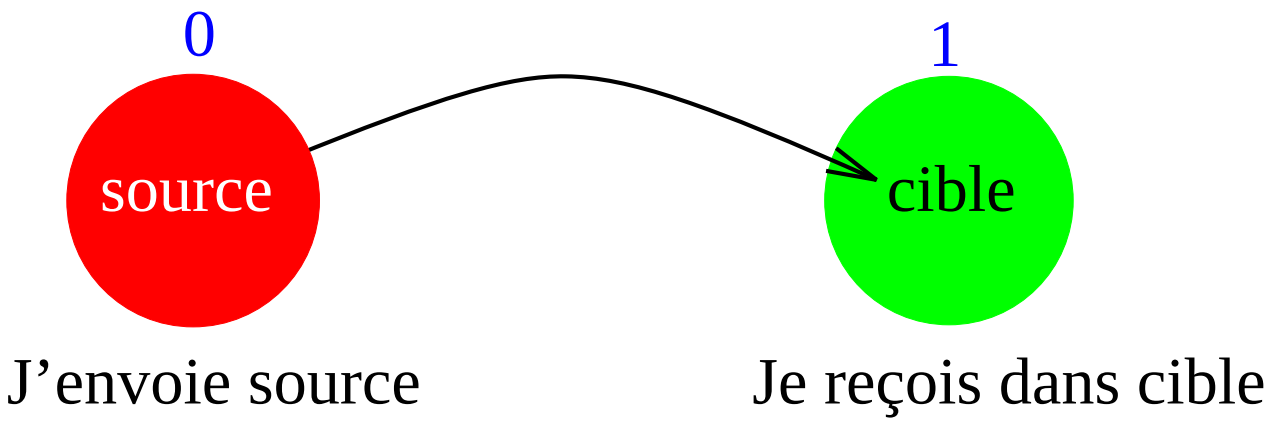


FIGURE 4 – L'échange de messages

- 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.

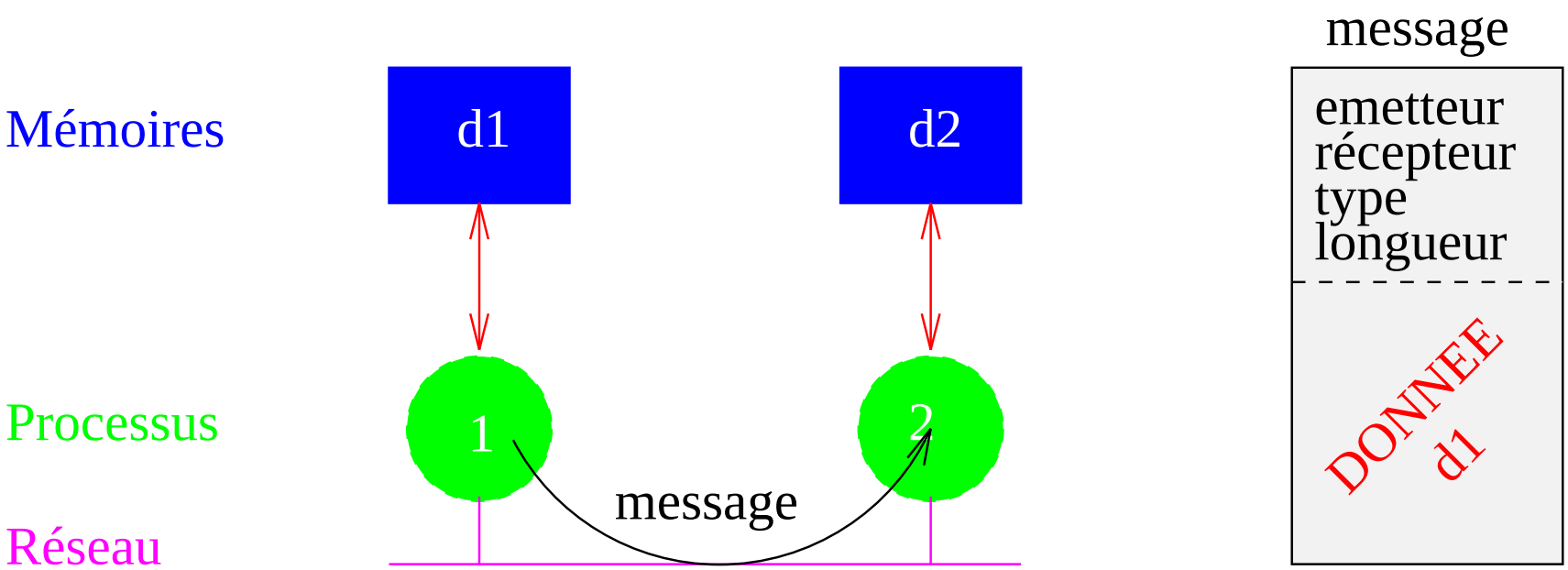


FIGURE 5 – Constitution d'un message

- Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé :
  - à la téléphonie ;
  - à la télécopie ;
  - au courrier postal ;
  - à 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 fonctions de la bibliothèque *MPI*

Ces fonctions 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 de MPI

- 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**.
- *MPI*-2 publié en juillet 97, suite à des travaux ayant commencé au printemps 95.
- « Standard » non élaboré par les organismes officiels de normalisation (ISO, ANSI, etc.).

1.4 – Bibliographie

- Sankalita Saha & al. *A Communication Interface for Multiprocessor Signal Processing Systems*. IEEE Workshop on Embedded Systems for Real-Time Multimedia. Seoul, Korea, Octobre 2006.
  http://www.ece.umd.edu/DSPCAD/papers/saha2006x4.pdf
- Rajagopal Subramaniyan & al. *FEMPI : A Lightweight Fault-tolerant MPI for Embedded Cluster Systems*. http://www.hcs.ufl.edu/pubs/ESA2006a.pdf
- A. Agbaria, K. Dong-in Kang Singh. *LMPI : MPI for Heterogeneous Embedded Distributed Systems*. ICPADS 2006, 12th International Conference on Parallel and Distributed Systems.
- Ahmed A. Jerraya. *Long Term Trends for Embedded System Design*. CEPA 2 Workshop Digital Platforms for Defence. March 15-16, 2005.
  http://tima.imag.fr/SLS/documents/CEPA2_Jerraya.pdf.
- Sidney Cadot & al. *ENSEMBLE : A Communication Layer for Embedded Multi-Processor Systems*. LCTES'2001, June 22-23, 2001.
  http://www.st.ewi.tudelft.nl/~koen/papers/ensemble.ps.gz

- Randal S. Janka, Linda M. Wills. *A Novel Codesign Mathodology for Real-Time Embedded COTS Multiprocessor-Based Signal Processing Systems.*
  http://www.ece.gatech.edu/research/labs/easl/pdf/Janka-Wills.CODES2000.pdf
- Quelques ouvrages sur *MPI* :
  1. Marc Snir & al. *MPI : The Complete Reference*. Second edition. MIT Press, 1998. Volume 1 : *The MPI core* ; Volume 2 : *The MPI-2 extensions*.
  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.
  4. Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. ed. McGraw-Hill. Juin 2003.

- Une documentation complémentaire :

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

- Implémentations *MPI* du domaine public : elles peuvent être installées sur un grand nombre d'architectures mais leurs performances sont en général en dessous de celles des implémentations constructeurs.

  1. `MPICH` : http://www.mcs.anl.gov/mpi/mpich/
  2. `LAM` : http://www.lam-mpi.org/
  3. `OpenMPI` : http://www.open-mpi.org/

2 – Environnement

2.1 – Description

- Tout unité de programme C/C++ appelant des fonctions MPI doit inclure un fichier d'en-têtes `mpi.h`.
- La fonction `MPI_Init()` permet d'initialiser l'environnement nécessaire :

```
#include "mpi.h"
int MPI_Init(int *argc, char ***argv)
```

- Réciproquement, la fonction `MPI_Finalize()` désactive cet environnement :

```
#include "mpi.h"
int MPI_Finalize()
```

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

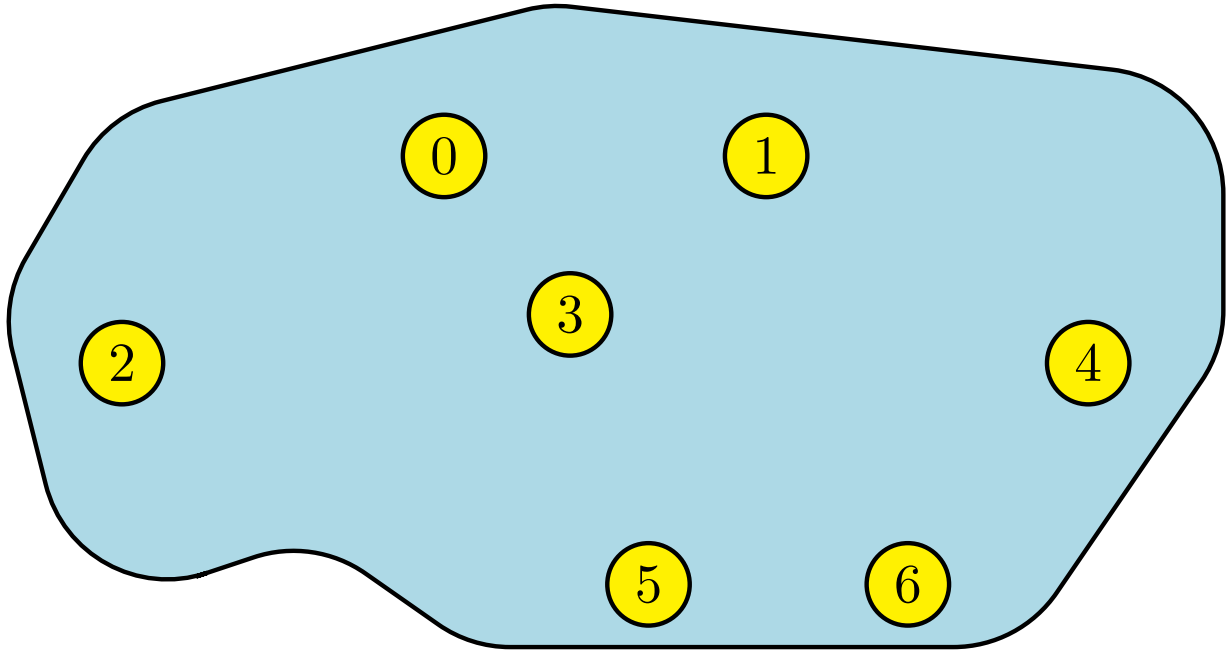


FIGURE 6 – Communicateur MPI_COMM_WORLD

☞ À tout instant, on peut connaître le nombre de processus gérés par un communicateur donné par la fonction `MPI_Comm_size()` :

```
#include "mpi.h"
int MPI_Comm_size ( MPI_Comm comm, int *nb_procs )
```

☞ De même, la fonction `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) :

```
#include "mpi.h"
int MPI_Comm_rank ( MPI_Comm comm, int *rang )
```

2.2 – Exemple

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(int argc, char *argv[])
6  {
7    int rang, nb_procs, code;
8
9    code = MPI_Init(&argc,&argv);
10   code = MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
11   code = MPI_Comm_rank(MPI_COMM_WORLD, &rang);
12   if (code != MPI_SUCCESS)
13     printf ("Erreur :  impossible de connaitre mon rang\n");
14   else
15     printf ("Je suis le processus %d parmi %d\n", rang, nb_procs);
16   code=MPI_Finalize();
17   exit(0);
18 }
```

```
> mpiexec -n 4 qui_je_suis
Je suis le processus 3 parmi 4
Je suis le processus 0 parmi 4
Je suis le processus 1 parmi 4
Je suis le processus 2 parmi 4
```

2.3 – Exercice 1 : environnement MPI

☞ Gestion de l'environnement de MPI : affichage d'un message par chacun des processus, mais **différent** selon qu'ils sont de rang **pair** ou **impair**

3 – Communications point à point

3.1 – Notions générales

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

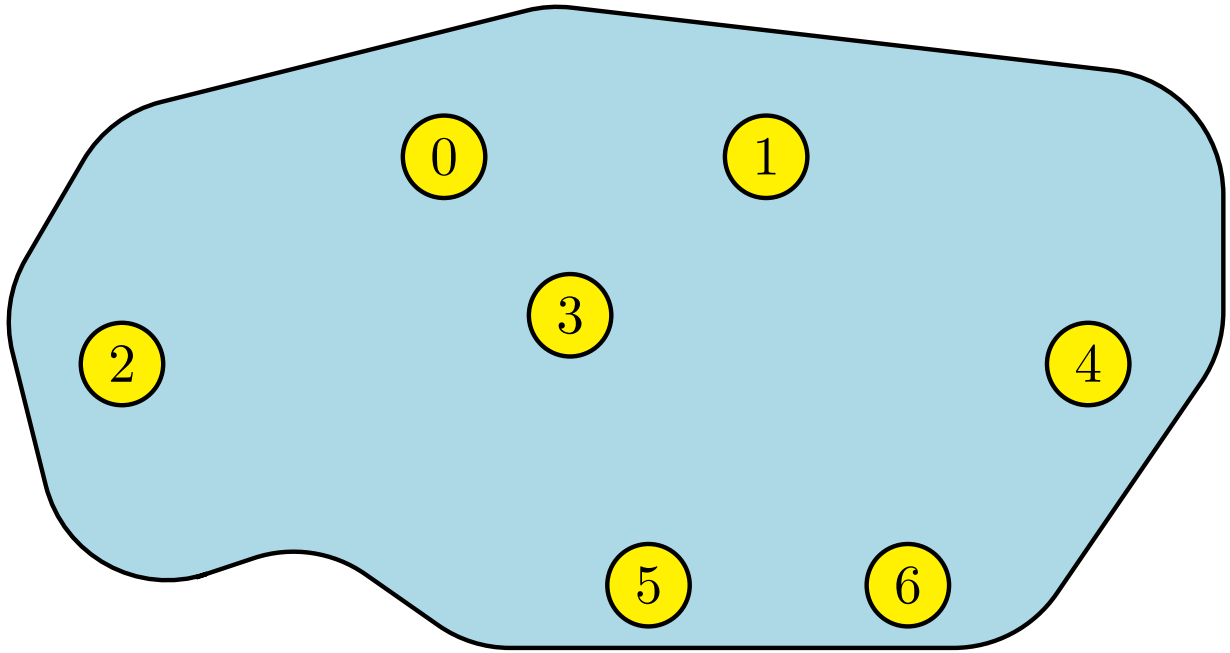


FIGURE 7 – Communication point à point

3 – Communications point à point

3.1 – Notions générales

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

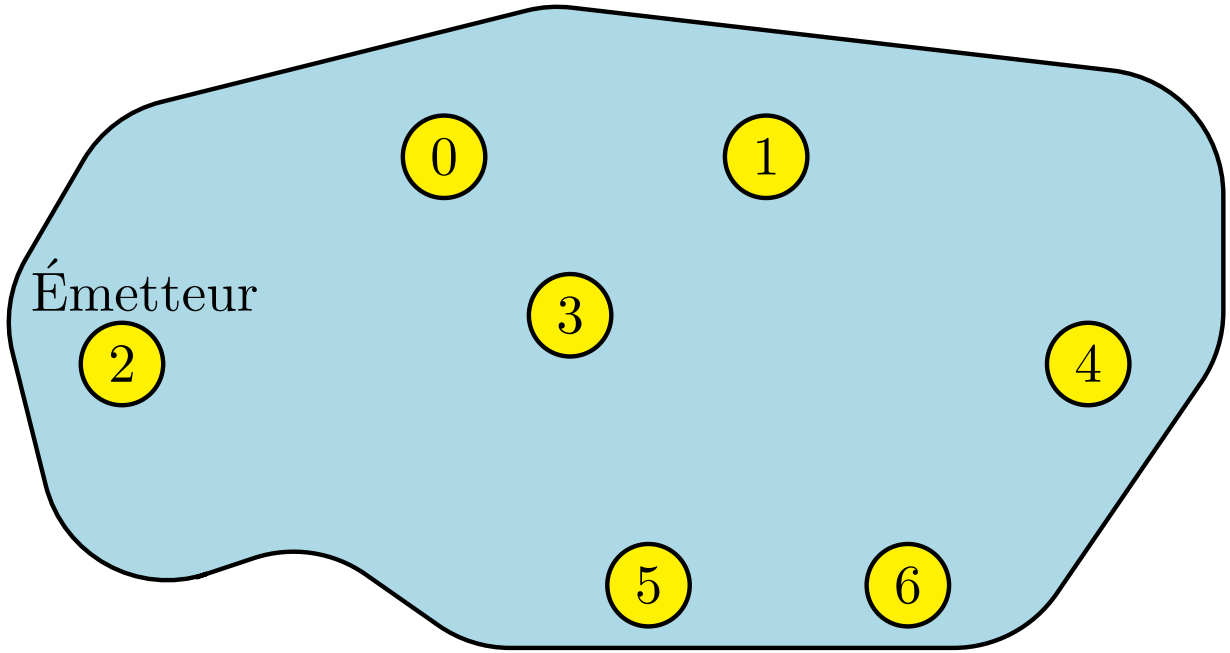


FIGURE 7 – Communication point à point

3 – Communications point à point

3.1 – Notions générales

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

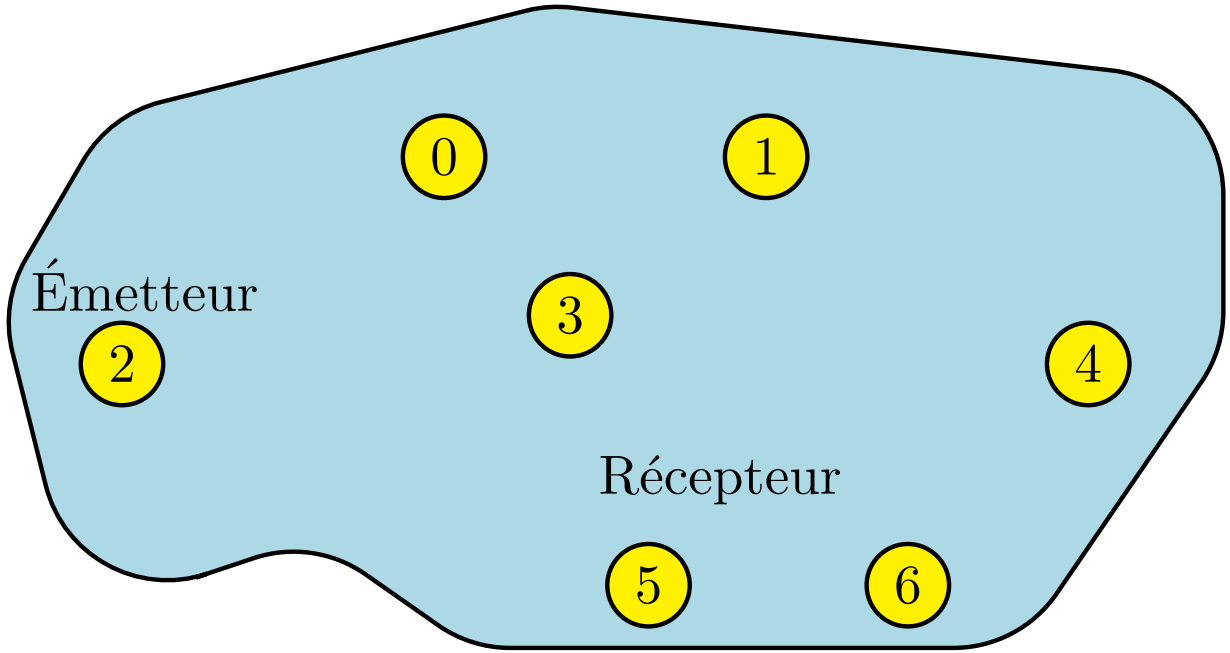


FIGURE 7 – Communication point à point

3 – Communications point à point

3.1 – Notions générales

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

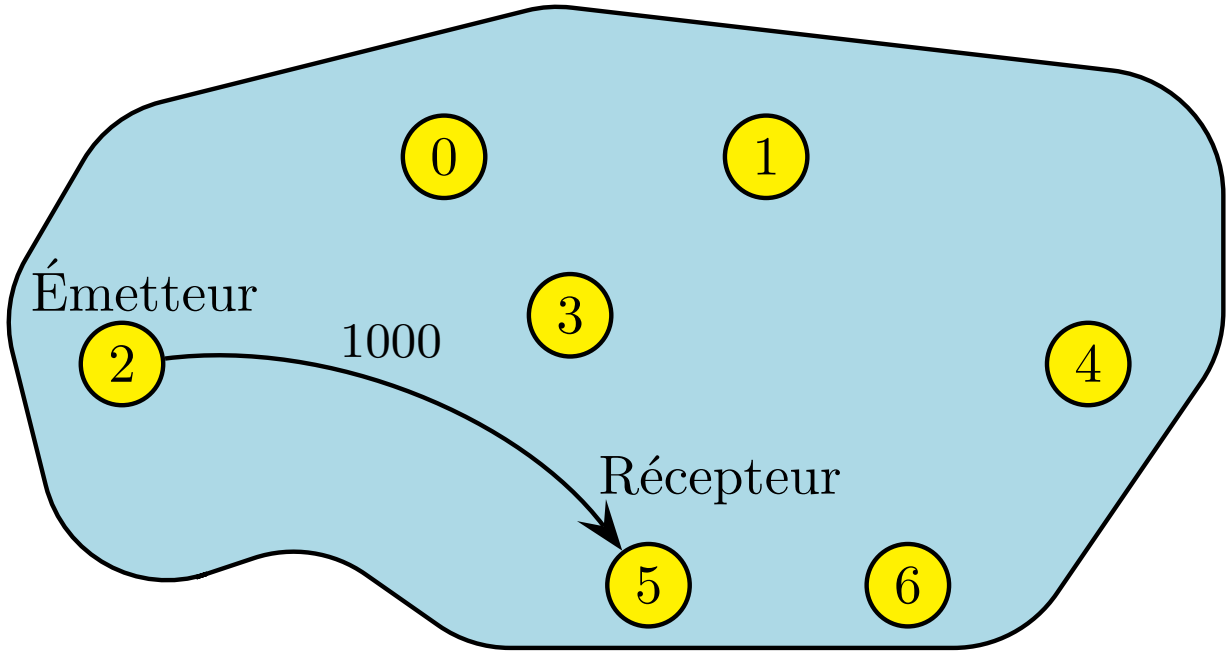


FIGURE 7 – Communication point à point

- ☞ L'émetteur et le récepteur sont identifiés par leur **rang** dans le communicateur.
- ☞ Ce que l'on appelle l'**enveloppe d'un message** est constituée :
  1. du rang du processus émetteur ;
  2. du rang du processus récepteur ;
  3. de l'étiquette (*tag*) du message ;
  4. du nom du communicateur qui définira le contexte de communication de l'opération.
- ☞ Les données échangées sont **typées** (entiers, réels, etc. ou types dérivés personnels).
- ☞ Il existe dans chaque cas plusieurs **modes** de transfert, faisant appel à des protocoles différents qui seront vus au chapitre 5.

```c
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#define ETIQUETTE 100

int main(int argc, char *argv[])
{
  int rang, valeur;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rang);
  if (rang == 2) {
    valeur=1000;
    MPI_Send(&valeur, 1, MPI_INT, 5, ETIQUETTE, MPI_COMM_WORLD);
  } else if (rang == 5) {
    MPI_Recv (&valeur, 1, MPI_INT, 2, ETIQUETTE, MPI_COMM_WORLD, &status);
    printf("Moi, processus 5, j'ai reçu %d du processus 2.\n", valeur);
  }
  MPI_Finalize();
  return(0);
}
```

```
> mpiexec -n 7 point_a_point

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

3.2 – Types de données de base

TABLE 1 – Principaux types de données de base (C)

Type MPI	Type C
MPI_CHAR	`signed char`
MPI_SHORT	`signed short`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_UNSIGNED_CHAR	`unsigned char`
MPI_UNSIGNED_SHORT	`unsigned short`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
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.
- Il existe des variantes syntaxiques, `MPI_Sendrecv()` et `MPI_Sendrecv_replace()`, qui enchaînent un envoi et une réception.
- On peut créer des structures de données plus complexes à l'aide de fonctions telles que `MPI_Type_contiguous()`, `MPI_Type_vector()`, `MPI_Type_indexed()` et `MPI_Type_struct()` (voir le chapitre 6).

FIGURE 8 – Communication `sendrecv` entre les processus 0 et 1

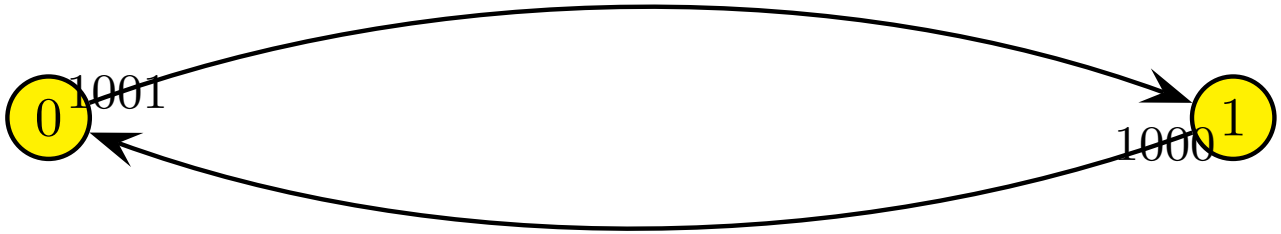


FIGURE 8 – Communication `sendrecv` entre les processus 0 et 1

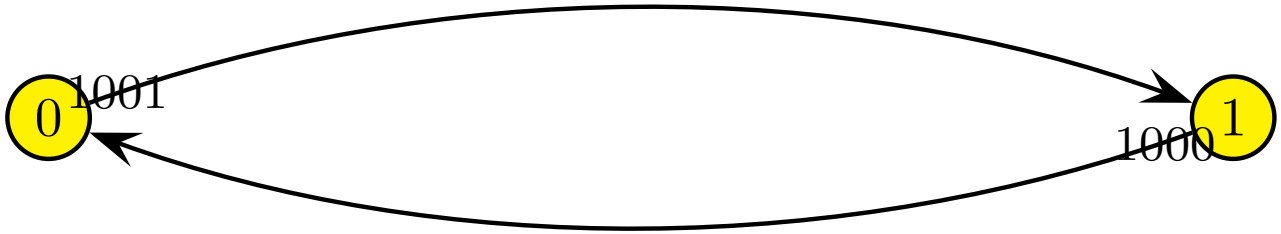


FIGURE 8 – Communication **sendrecv** entre les processus 0 et 1

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define ETIQUETTE 100
5  int main(int argc, char *argv[])
6  {
7    int rang, valeur, x, num_proc;
8    MPI_Status statut;
9    MPI_Init(&argc,&argv);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
11   /* On suppose avoir exactement 2 processus */
12   num_proc=(rang+1)%2;
13   valeur=rang+1000;
14   MPI_Sendrecv(&valeur, 1, MPI_INT, num_proc, ETIQUETTE, &x, 1,
15                MPI_INT, num_proc, ETIQUETTE, MPI_COMM_WORLD, &statut);
16   /* MPI_Sendrecv_replace(&valeur, 1, MPI_INT, num_proc, ETIQUETTE,       */
17   /*                          num_proc, ETIQUETTE, MPI_COMM_WORLD, &statut); */
18   printf("Moi, processus %d, j'ai reçu %d du processus %d.\n", rang,x,num_proc);
19   MPI_Finalize(); return(0);
20 }
```

```
> mpirun -np 2 sendrecv

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

Attention ! Il convient de noter que si la fonction `MPI_Send()` est implémentée de façon **bloquante** (voir le chapitre 5) dans la version de la bibliothèque *MPI* mise en œuvre, le code précédent serait en situation de verrouillage si à la place de l'ordre `MPI_Sendrecv()` on utilisait un ordre `MPI_Send()` suivi d'un ordre `MPI_Recv()`.
En effet, chacun des deux processus attendrait un ordre de réception qui ne viendrait jamais, puisque les deux envois resteraient en suspens. Pour des raisons de portabilité, il faut donc absolument éviter ces cas-là.

```
MPI_Send(&valeur,1,MPI_INT,num_proc,ETIQUETTE,MPI_COMM_WORLD);
MPI_Recv(&x,1,MPI_INT,num_proc,ETIQUETTE,MPI_COMM_WORLD,&statut);
```

3.4 – Exemple : anneau de communication

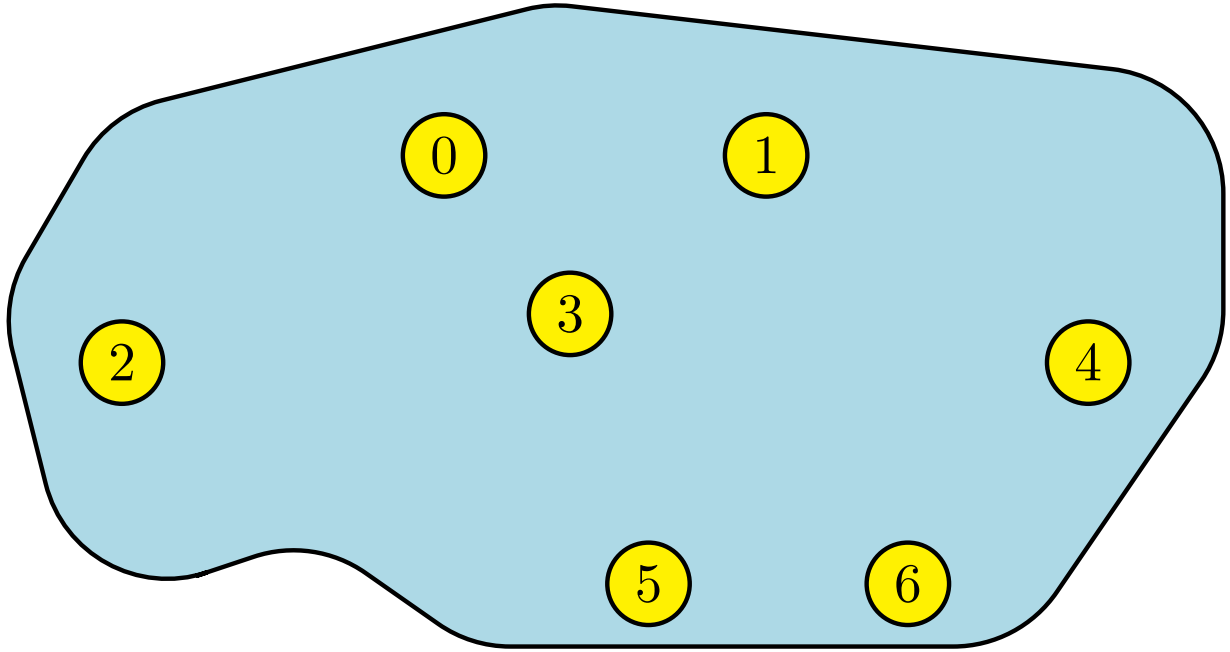


FIGURE 9 – Anneau de communication

3.4 – Exemple : anneau de communication

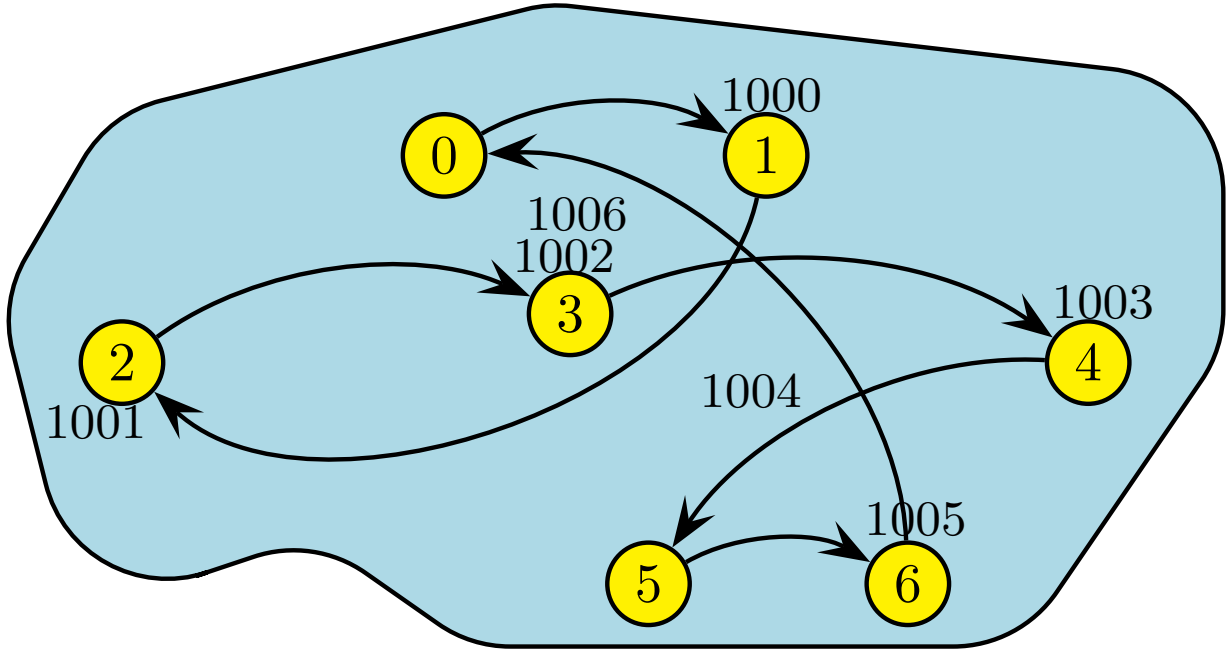


FIGURE 9 – Anneau de communication

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

```
...
valeur=rang+1000;
MPI_Send(&valeur,1,MPI_INT,num_proc_suivant,ETIQUETTE,MPI_COMM_WORLD);
MPI_Recv(&x,1,MPI_INT,num_proc_precedent,ETIQUETTE,MPI_COMM_WORLD,&statut);
...
```

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

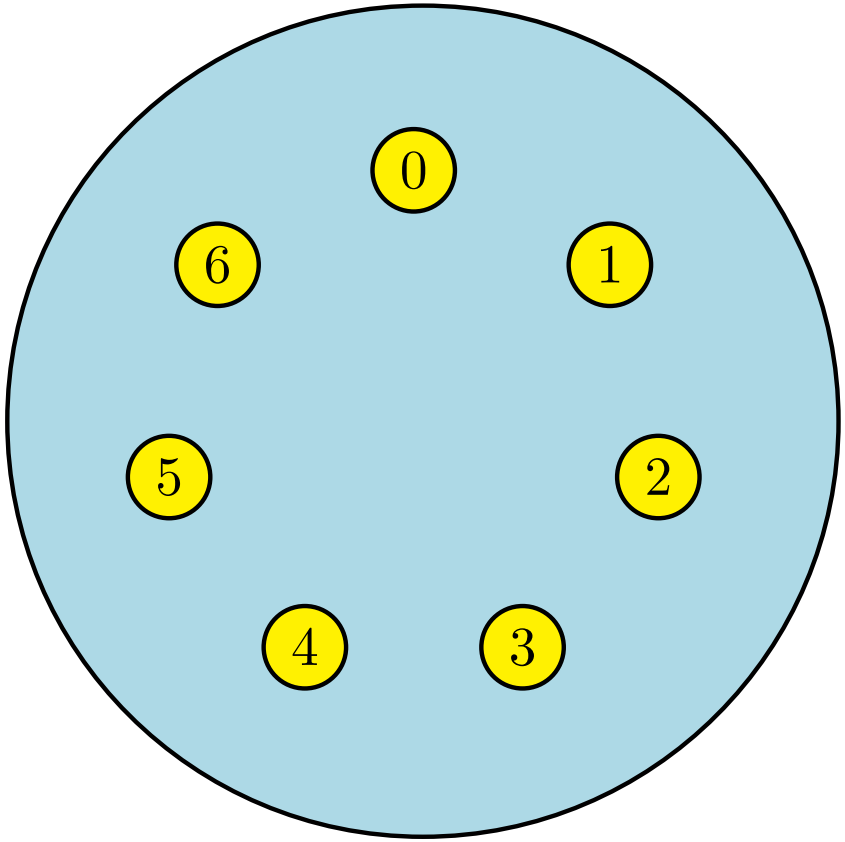


FIGURE 10 – Anneau de communication

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

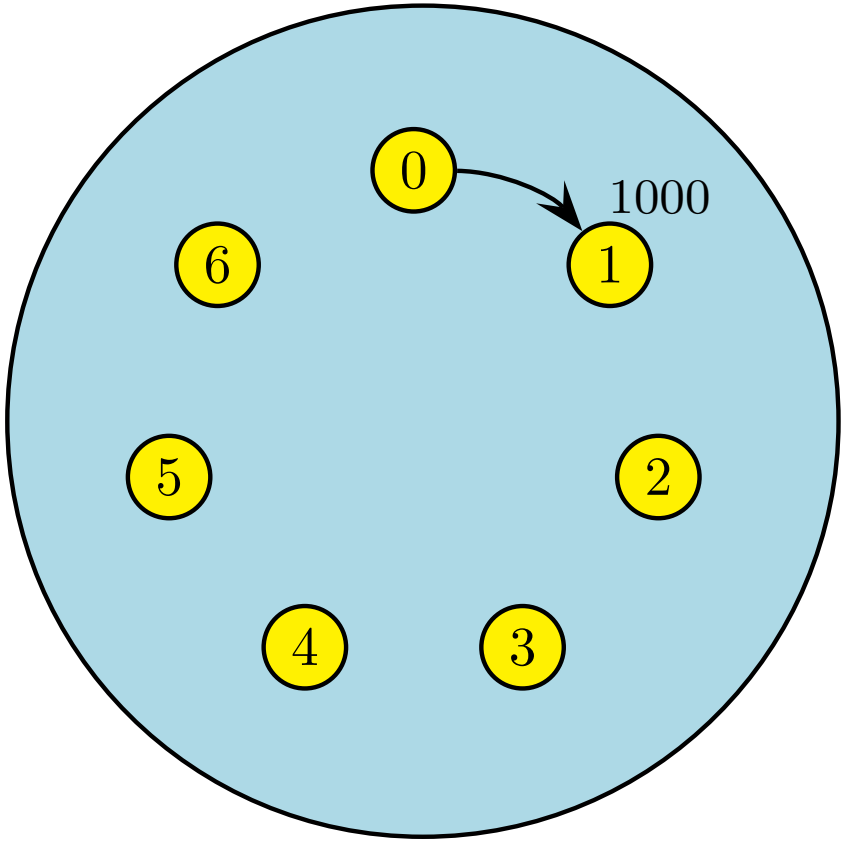


FIGURE 10 – Anneau de communication

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

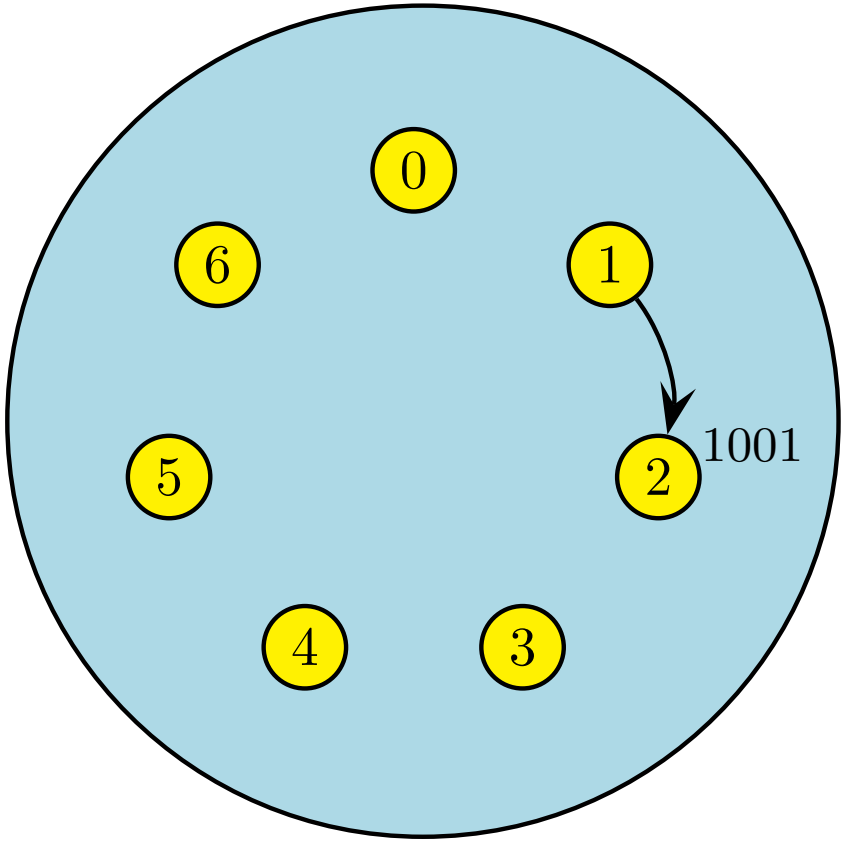


FIGURE 10 – Anneau de communication

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

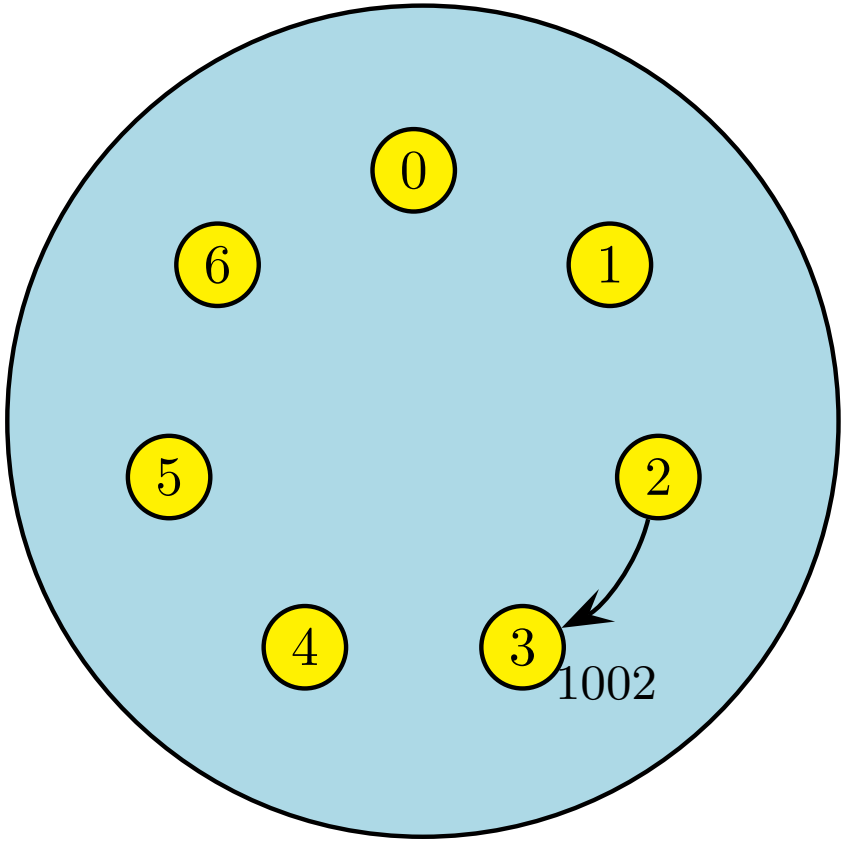


FIGURE 10 – Anneau de communication

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

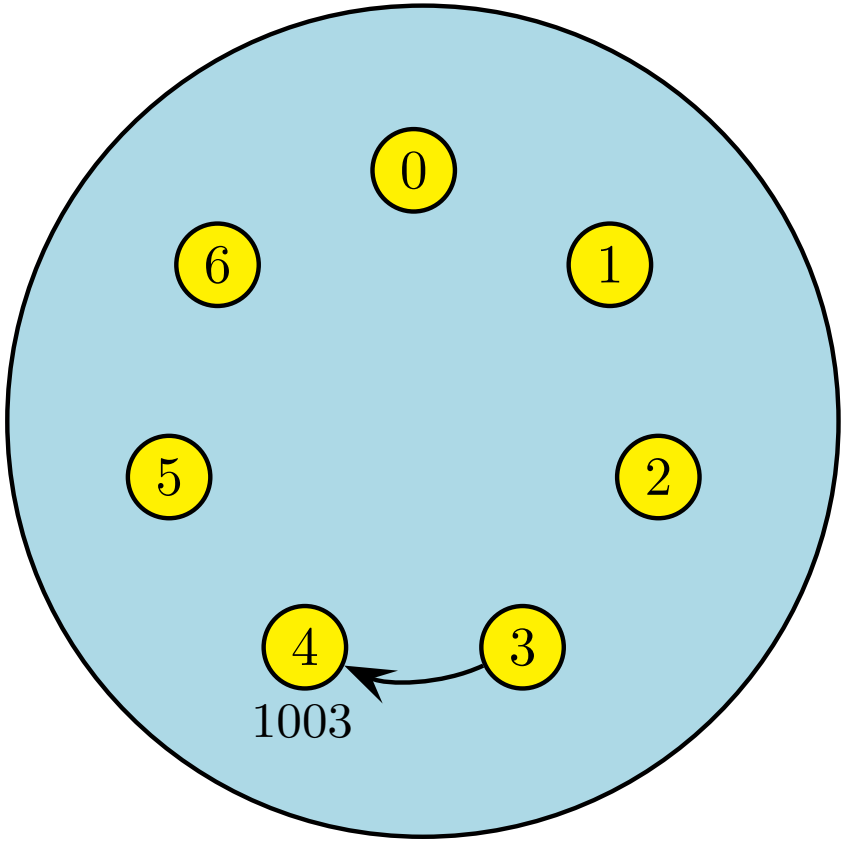


FIGURE 10 – Anneau de communication

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

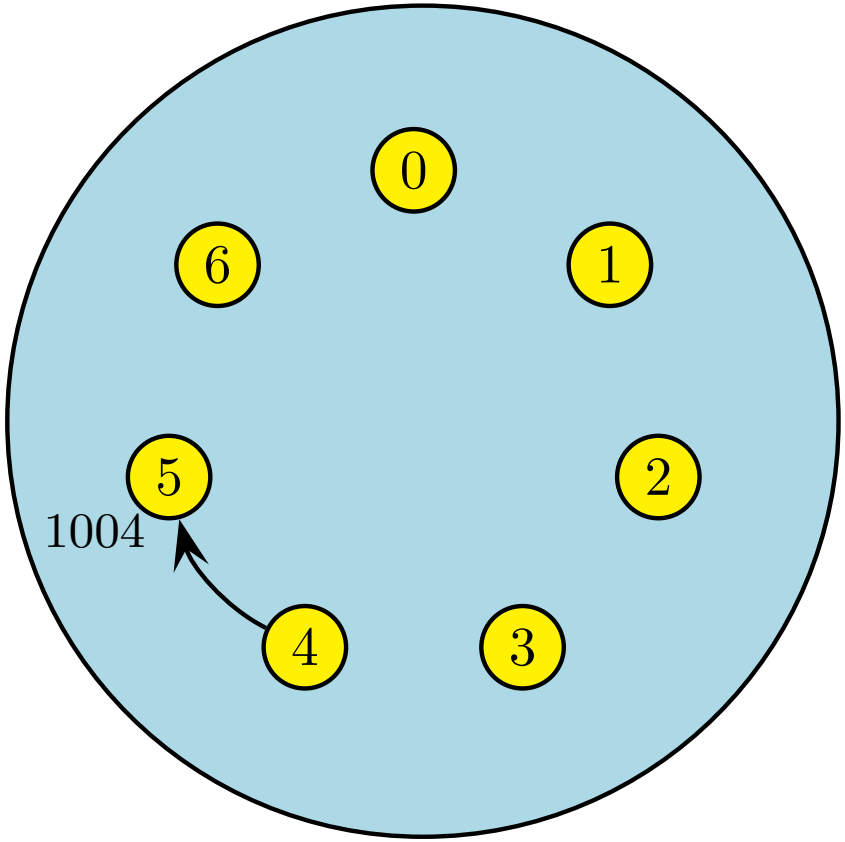


FIGURE 10 – Anneau de communication

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

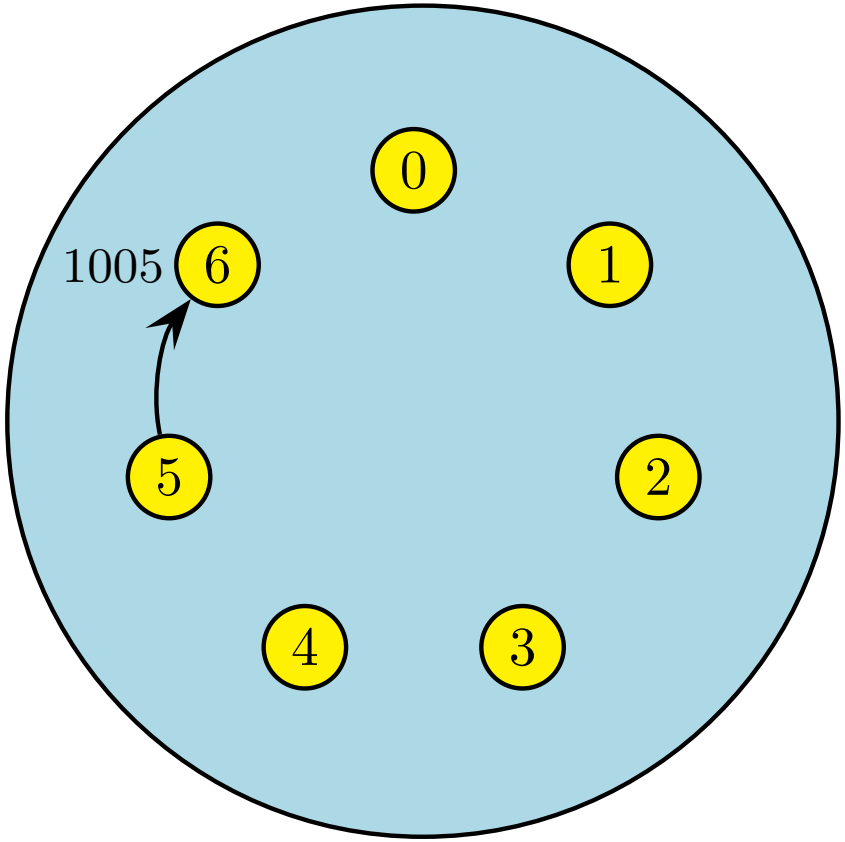


FIGURE 10 – Anneau de communication

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

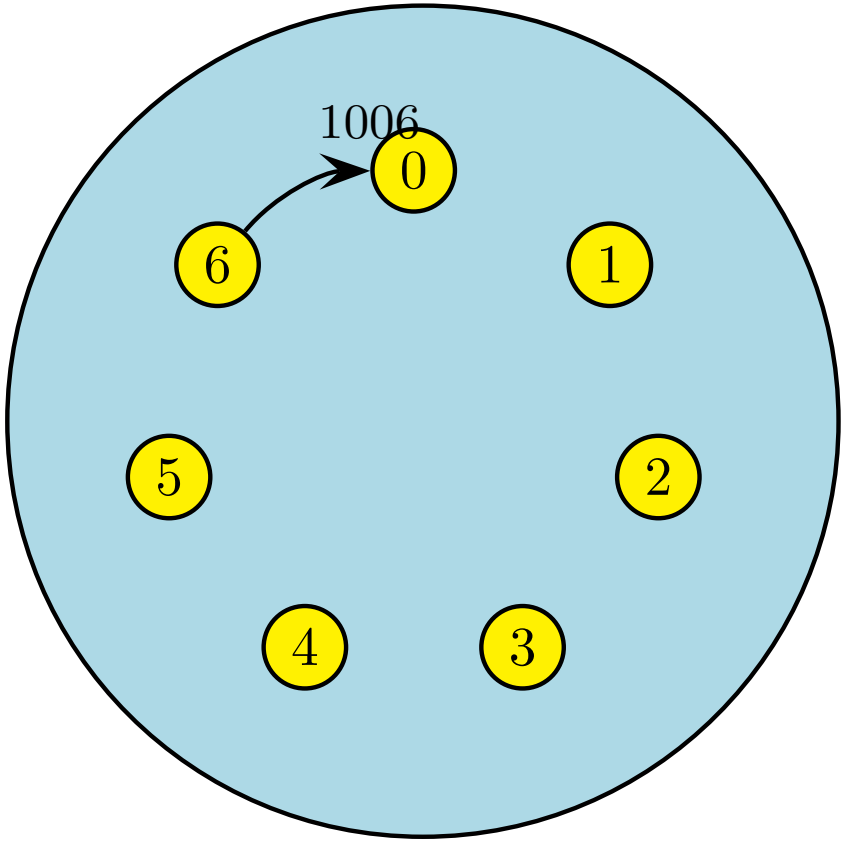


FIGURE 10 – Anneau de communication

```c
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#define ETIQUETTE 100
int main(int argc, char *argv[])
{
  int rang, nb_procs, valeur, x, num_proc_precedent, num_proc_suivant;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rang);

  num_proc_suivant=(rang+1)%nb_procs;
  num_proc_precedent=(nb_procs+rang-1)%nb_procs;
  if (rang == 0) {
    valeur=rang+1000;
    MPI_Send(&valeur, 1, MPI_INT, num_proc_suivant, ETIQUETTE, MPI_COMM_WORLD);
    MPI_Recv(&x, 1, MPI_INT, num_proc_precedent, ETIQUETTE, MPI_COMM_WORLD, &statut);
  } else {
    MPI_Recv(&x, 1, MPI_INT, num_proc_precedent, ETIQUETTE, MPI_COMM_WORLD, &statut);
    valeur=rang+1000;
    MPI_Send(&valeur, 1, MPI_INT, num_proc_suivant, ETIQUETTE, MPI_COMM_WORLD);
  }
  printf("Moi, processus %d, j'ai reçu %d du processus %d.\n", rang, x,
         num_proc_precedent);
  MPI_Finalize();
  return(0);
}
```

```
> mpirun -np 7 anneau

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

Il est possible d'effectuer un seul envoi de message regroupant des données de types différents. Pour ce faire, il faut construire en émission le contenu du message en passant par une zone tampon, via la fonction `MPI_Pack`. En réception, la fonction `MPI_Unpack` permet de façon réciproque de reconstituer les données contenues dans la zone tampon.

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define TAILLE_IMAGE 691476  /* Octets */
5  /* Fonctions externes */
6  extern int loadtiff(char *fileName, unsigned char *image, int *iw, int *ih);
7  extern int dumpTiff(char *fileName, unsigned char *image, int *w, int *h);
8  /* Structure Image */
9  typedef struct {
10   int largeur, hauteur;                /* Dimensions de l'image en pixels */
11   unsigned char donnees[TAILLE_IMAGE]; /* Données associées à l'image */
12 } Image;
13 int main(int argc, char *argv[])
14 {
15   int rang, etiquette=100;
16   int position, taille_membre, taille_max, taille_message;
17   Image img;
18   char *tampon;
19   MPI_Status statut;
```

```
1   MPI_Init(&argc,&argv);
2   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
3
4   /* Le processus 0 lit un fichier contenant l'image */
5   if (rang == 0)
6     loadtiff("Eiffel.tif", img.donnees, &img.largeur, &img.hauteur);
7
8   /* Déterminer la taille globale du message */
9   MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &taille_membre);
10  taille_max = taille_membre;
11  MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &taille_membre);
12  taille_max += taille_membre;
13  MPI_Pack_size(TAILLE_IMAGE, MPI_UNSIGNED_CHAR, MPI_COMM_WORLD, &taille_membre);
14  taille_max += taille_membre;
15  tampon = malloc(taille_max);
```

```
1   /* Envoi de la structure img au processus 1 */
2   if (rang == 0) {
3     position = 0;
4     MPI_Pack(&img.largeur,1,MPI_INT,tampon,taille_max,&position,MPI_COMM_WORLD);
5     MPI_Pack(&img.hauteur,1,MPI_INT,tampon,taille_max,&position,MPI_COMM_WORLD);
6     MPI_Pack(img.donnees,TAILLE_IMAGE,MPI_UNSIGNED_CHAR,tampon,taille_max,
7              &position,MPI_COMM_WORLD);
8     MPI_Send(tampon, position, MPI_PACKED, 1, etiquette, MPI_COMM_WORLD);
9   }
10
11  /* Réception de la structure img du processus 0 */
12  else if ( rang == 1 ) {
13    MPI_Recv(tampon, taille_max, MPI_PACKED, 0, etiquette, MPI_COMM_WORLD, &statut);
14    position = 0;
15    MPI_Get_count(&statut, MPI_PACKED, &taille_message);
16    MPI_Unpack(tampon,taille_message,&position,&img.largeur,1,
17               MPI_INT,MPI_COMM_WORLD);
18    MPI_Unpack(tampon,taille_message,&position,&img.hauteur,1,
19               MPI_INT,MPI_COMM_WORLD);
20    MPI_Unpack(tampon,taille_message,&position,img.donnees,TAILLE_IMAGE,
21               MPI_UNSIGNED_CHAR,MPI_COMM_WORLD);
22
23    /*
24     * dumpTiff("Eiffel_b.tif",img.donnees,&img.largeur,&img.hauteur);
25     */
26  }
27  free(tampon); MPI_Finalize(); return(0);
28 }
```

3.6 – Exercice 2 : ping-pong

☞ Communications point à point : *ping-pong* entre deux processus

① Envoyer un message contenant 1000 réels du processus 0 vers le processus 1 (il s'agit alors seulement d'un *ping*)

② Faire une version *ping-pong* où le processus 1 renvoie le message reçu au processus 0 et mesurer le temps de communication à l'aide de la fonction `MPI_Wtime()`

③ Faire une version où l'on fait varier la taille du message dans une boucle et mesurer les temps de communication respectifs ainsi que les débits

☞ Les mesures de temps peuvent se faire de la façon suivante :

```
...
temps_debut=MPI_Wtime();
...
temps_fin=MPI_Wtime();
printf("... en %8.6f secondes.",temps_fin-temps_debut);
...
```

3.7 – Exercice 3 : distribution d'une image

Reprendre le dernière exemple de ce chapitre et y introduire les modifications appropriées (seules les fonctions `send/recv` seront utilisées) de sorte que le contenu de l'image "`Eiffel.tif`" soit répartie (par bloc de lignes, par exemple) sur un ensemble de 4 processus.

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 fonctions. 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 fonctions :

❶ celui qui assure les synchronisations globales : `MPI_Barrier()`.

❷ ceux qui ne font que transférer des données :

- ❑ diffusion globale de données : `MPI_Bcast()` ;
- ❑ diffusion sélective de données : `MPI_Scatter()` ;
- ❑ collecte de données réparties : `MPI_Gather()` ;
- ❑ collecte par tous les processus de données réparties : `MPI_Allgather()` ;
- ❑ diffusion sélective, par tous les processus, de données réparties : `MPI_Alltoall()`.

❸ ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :

- ❑ opérations de réduction, qu'elles soient d'un type prédéfini (somme, produit, maximum, minimum, etc.) ou d'un type personnel : `MPI_Reduce()` ;
- ❑ opérations de réduction avec diffusion du résultat (il s'agit en fait d'un `MPI_Reduce()` suivi d'un `MPI_Bcast()`) : `MPI_Allreduce()`.

4.2 – Synchronisation globale : MPI_Barrier()

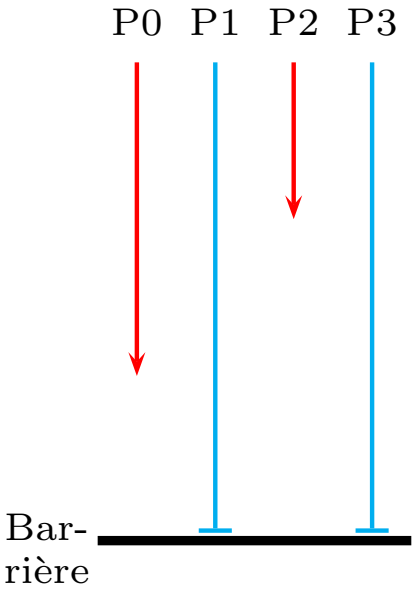


FIGURE 11 – Synchronisation globale : MPI_Barrier()

```
int code;
MPI_Comm comm;

code=MPI_Barrier(comm);
```

4.2 – Synchronisation globale : MPI_Barrier()

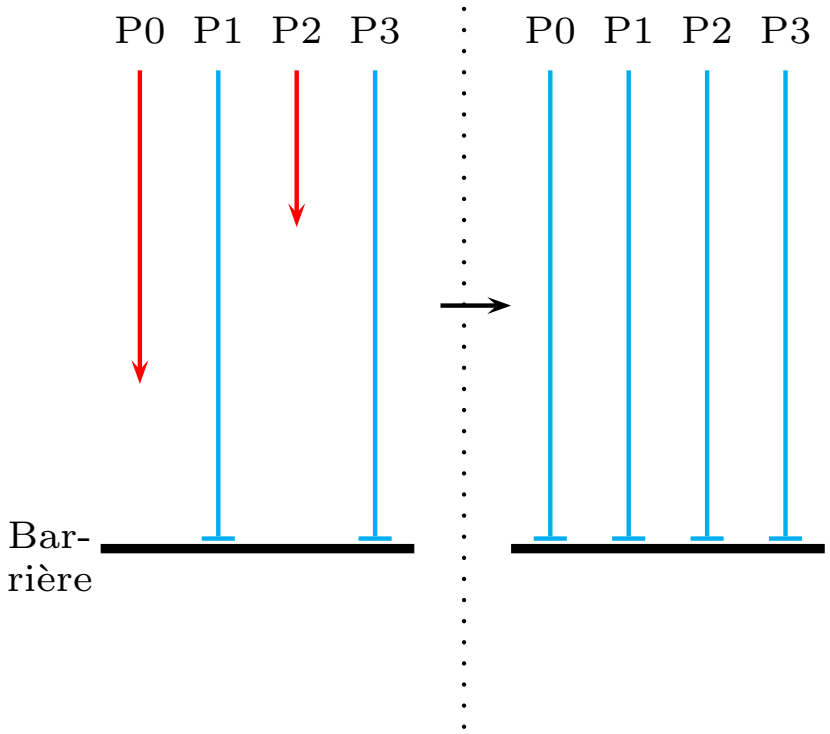


FIGURE 11 – Synchronisation globale : MPI_Barrier()

```
int code;
MPI_Comm comm;

code=MPI_Barrier(comm);
```

4.2 – Synchronisation globale : MPI_Barrier()

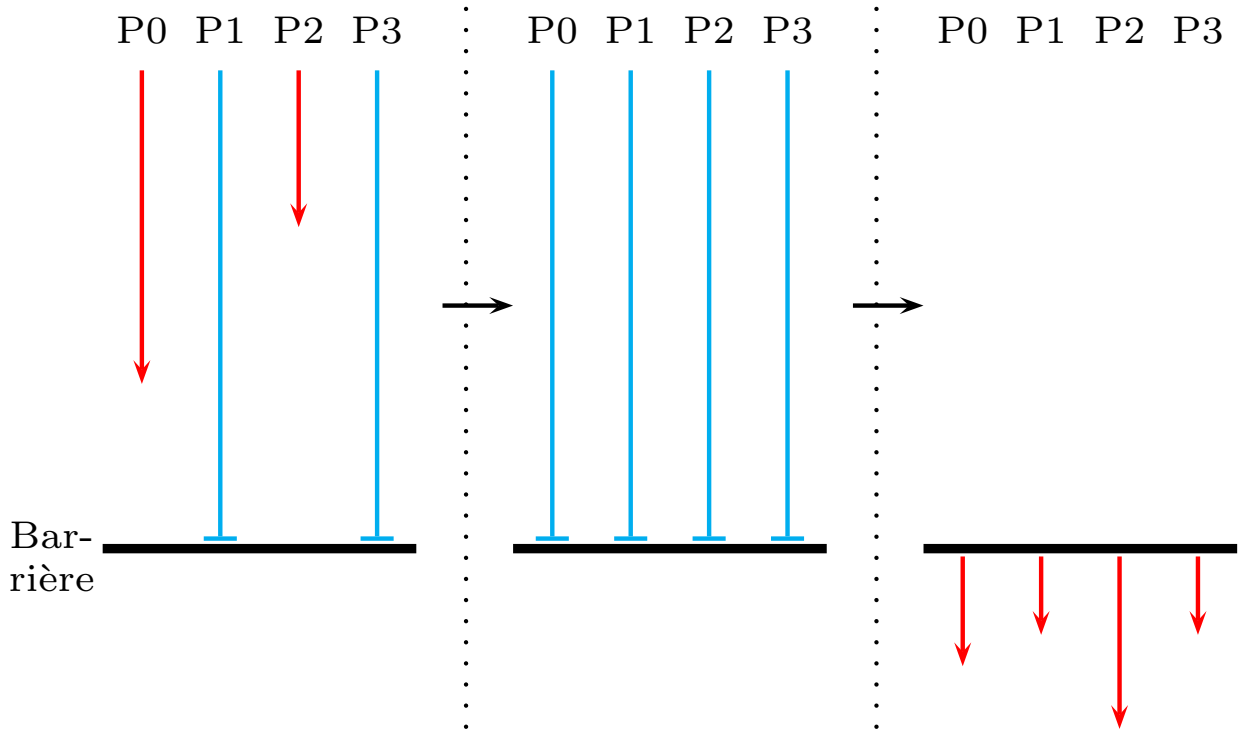


FIGURE 11 – Synchronisation globale : MPI_Barrier()

```
int code;
MPI_Comm comm;

code=MPI_Barrier(comm);
```

4.3 – Diffusion générale : `MPI_Bcast()`

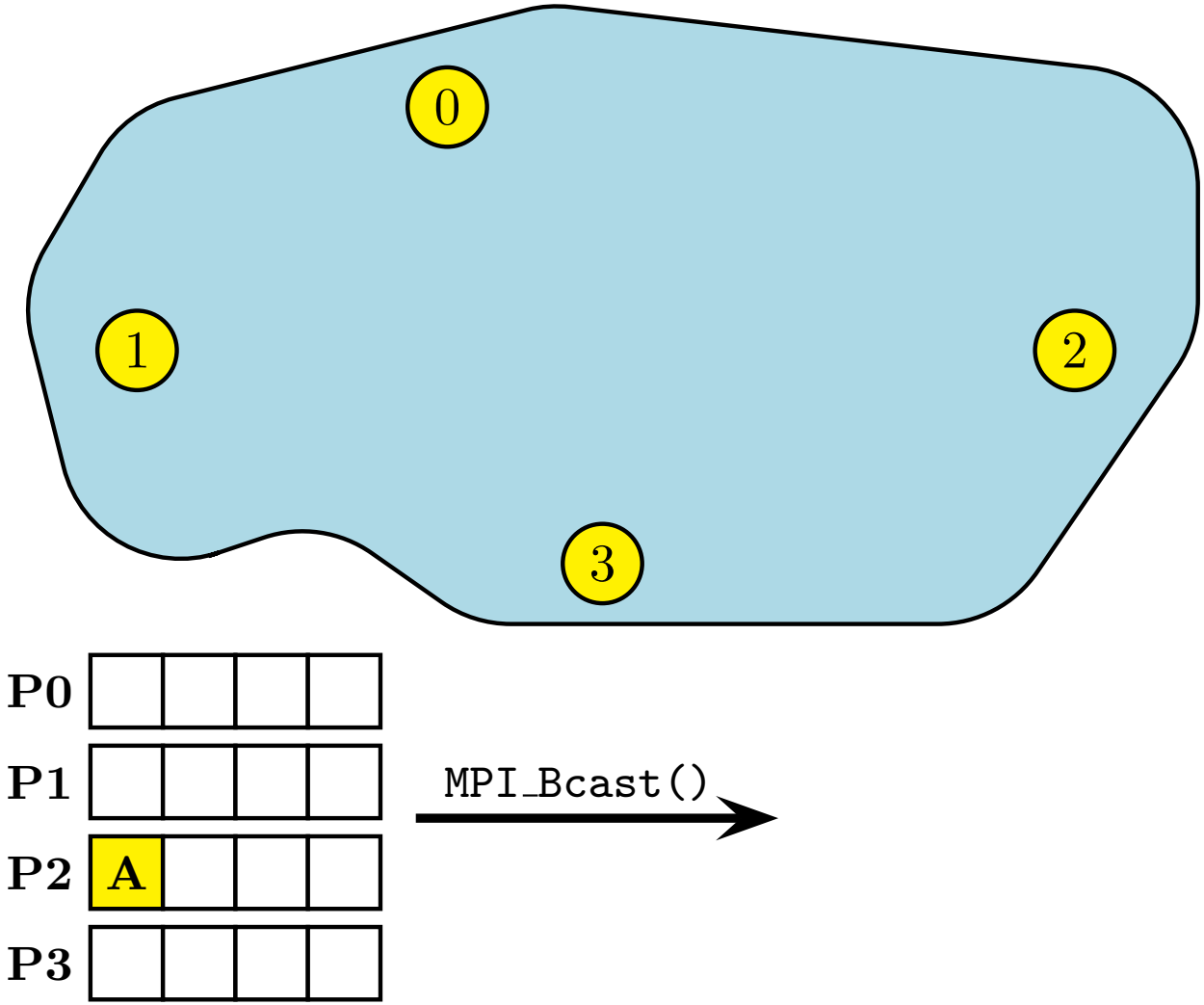


FIGURE 12 – Diffusion générale : `MPI_Bcast()`

4.3 – Diffusion générale : MPI_Bcast()

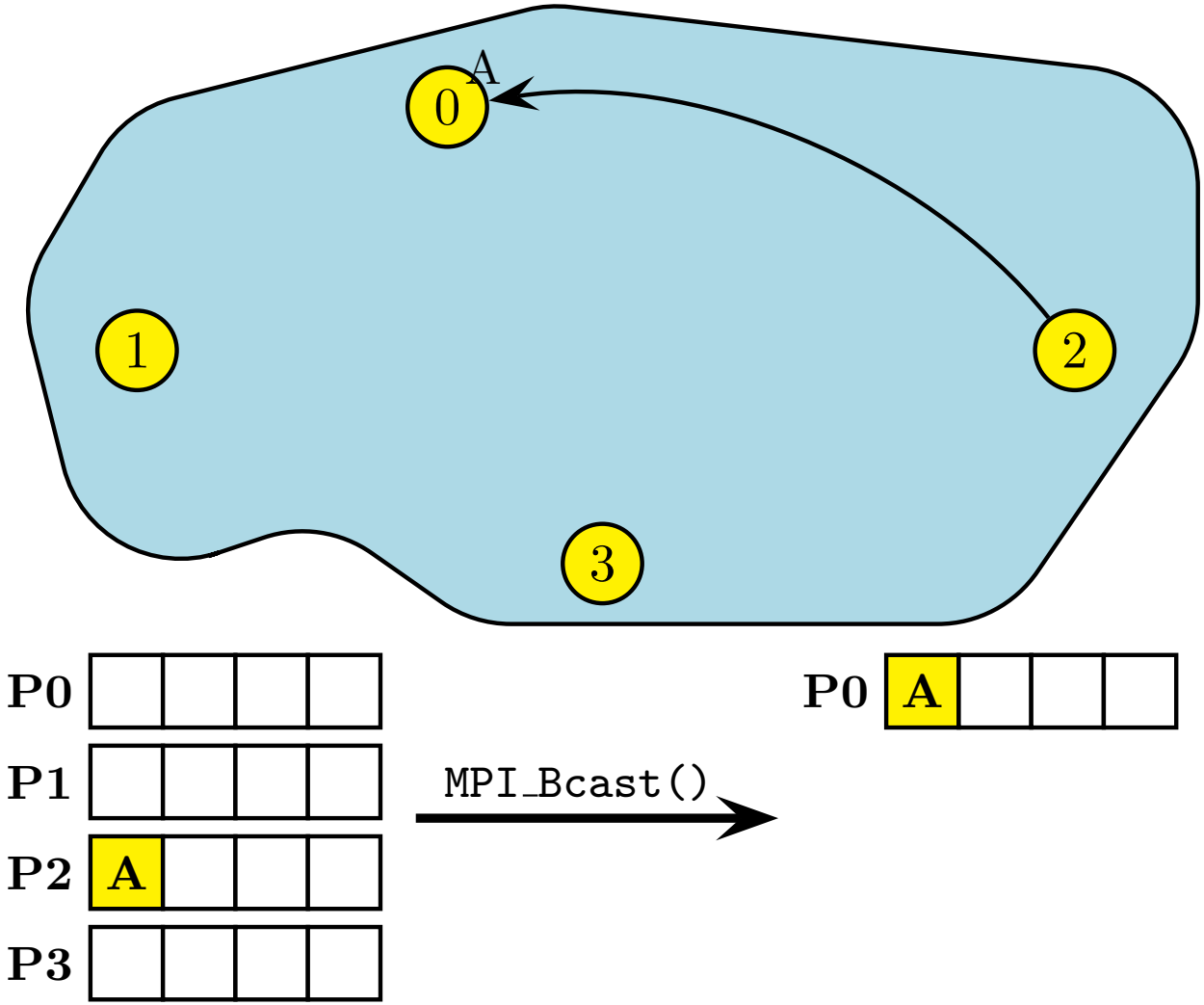


FIGURE 12 – Diffusion générale : MPI_Bcast()

4.3 – Diffusion générale : MPI_Bcast()

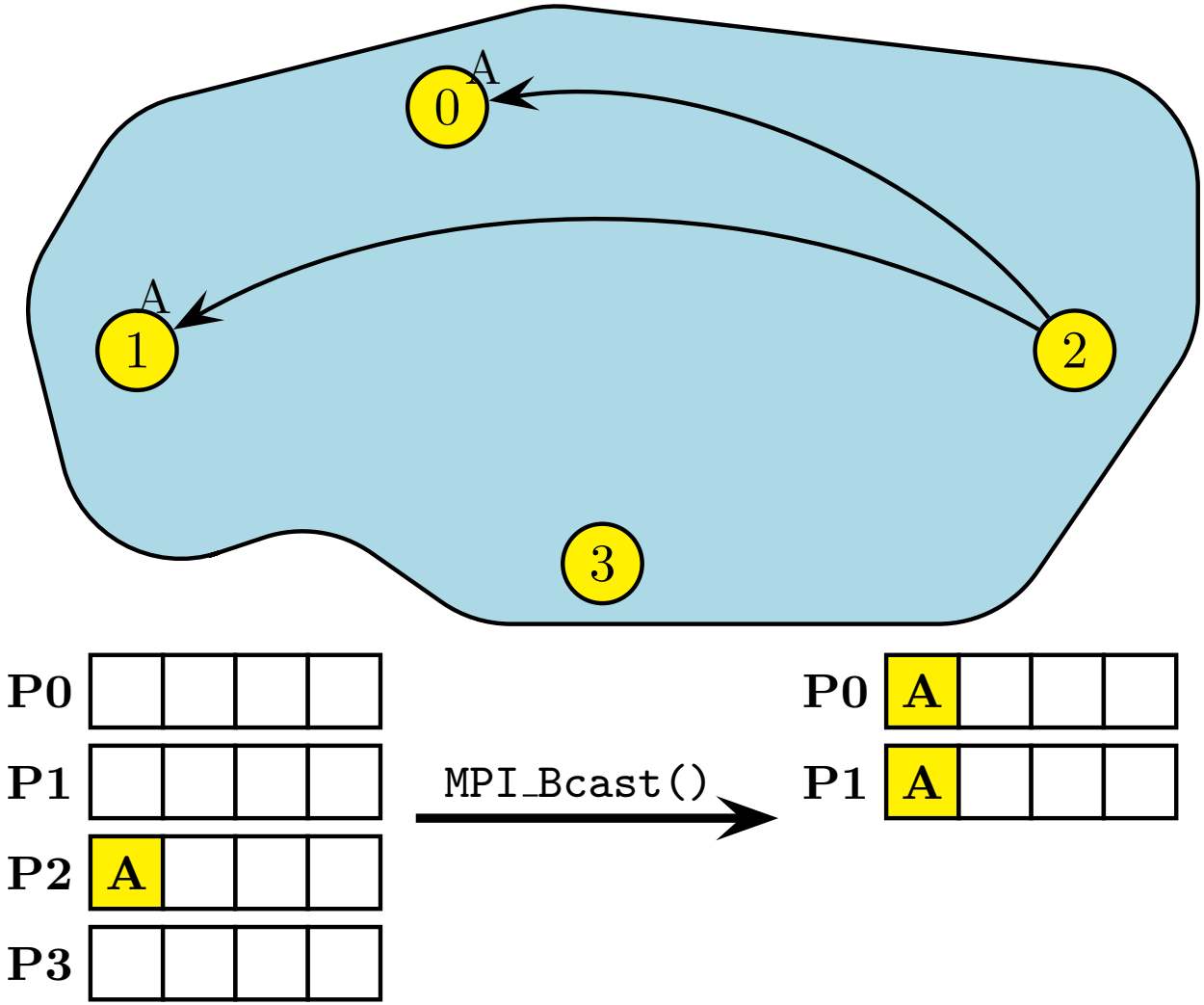


FIGURE 12 – Diffusion générale : MPI_Bcast()

4.3 – Diffusion générale : MPI_Bcast()

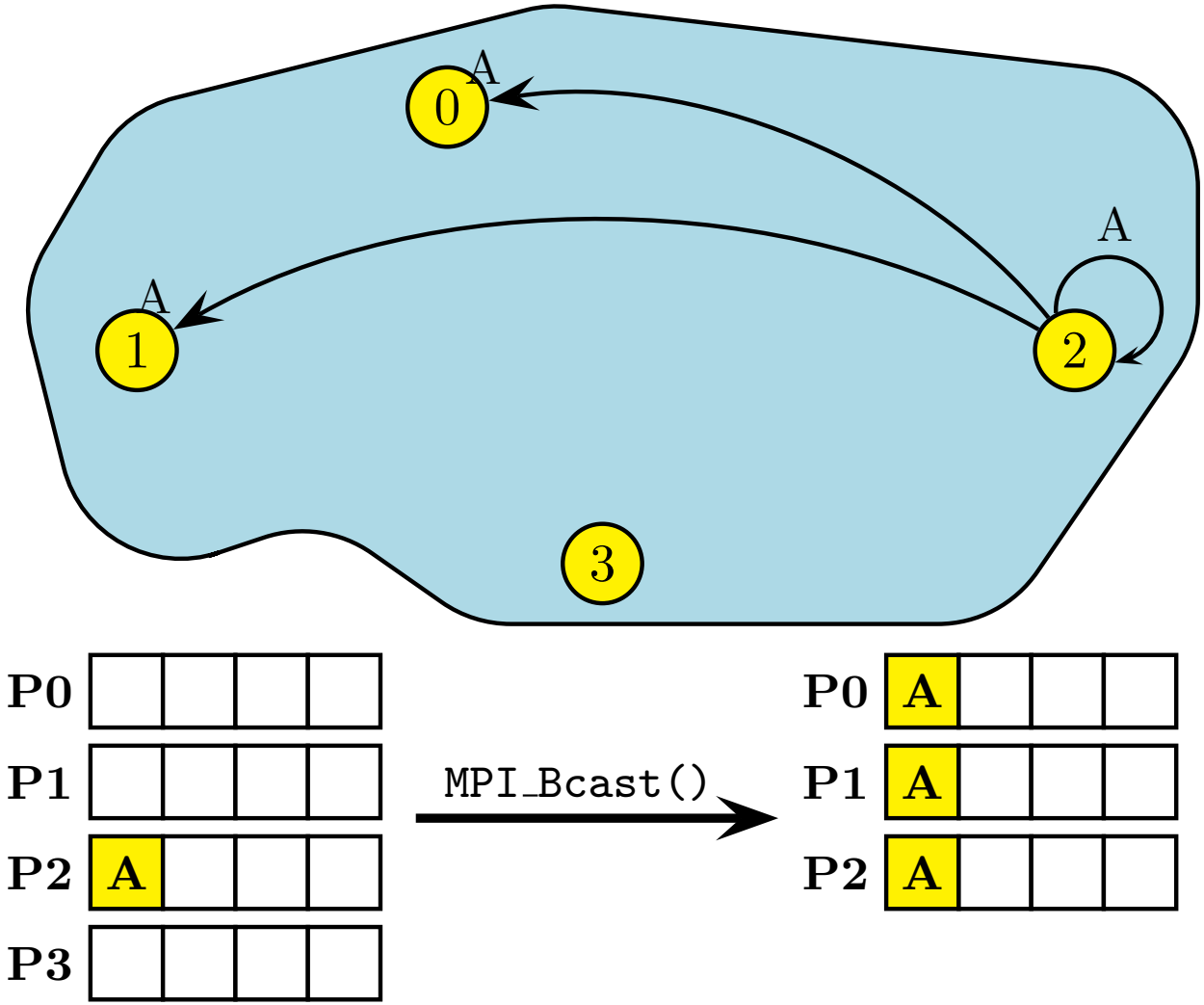


FIGURE 12 – Diffusion générale : MPI_Bcast()

4.3 – Diffusion générale : MPI_Bcast()

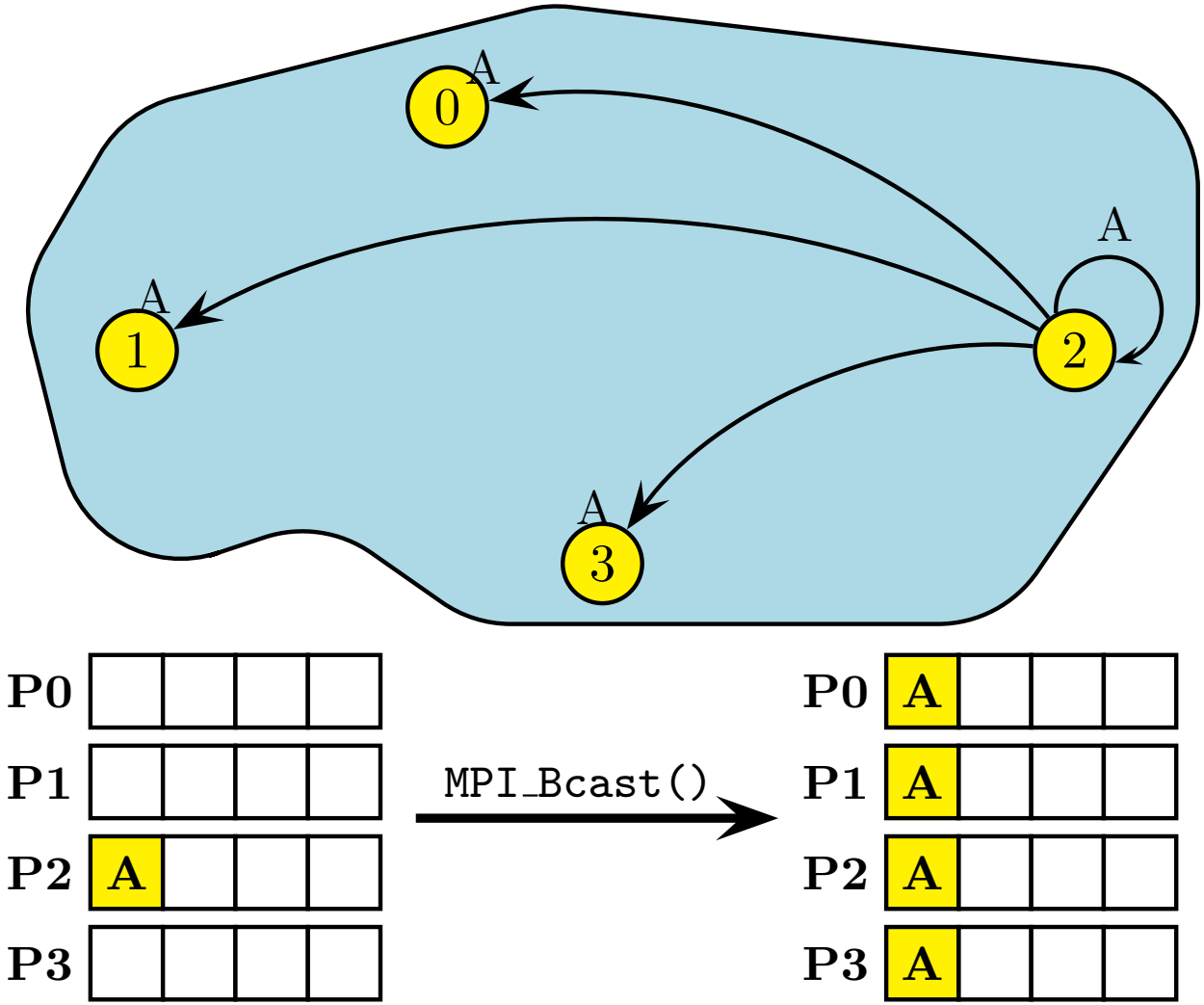


FIGURE 12 – Diffusion générale : MPI_Bcast()

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  int main(int argc, char *argv[])
5  {
6    int rang, valeur;
7
8    MPI_Init(&argc,&argv);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rang);
10
11   if (rang == 2) valeur=rang+1000;
12   MPI_Bcast(&valeur,1,MPI_INT,2,MPI_COMM_WORLD);
13
14   printf("Moi, processus %d, j'ai reçu %d du processus 2\n",rang, valeur);
15   MPI_Finalize(); return(0);
16 }
```

```
> 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()

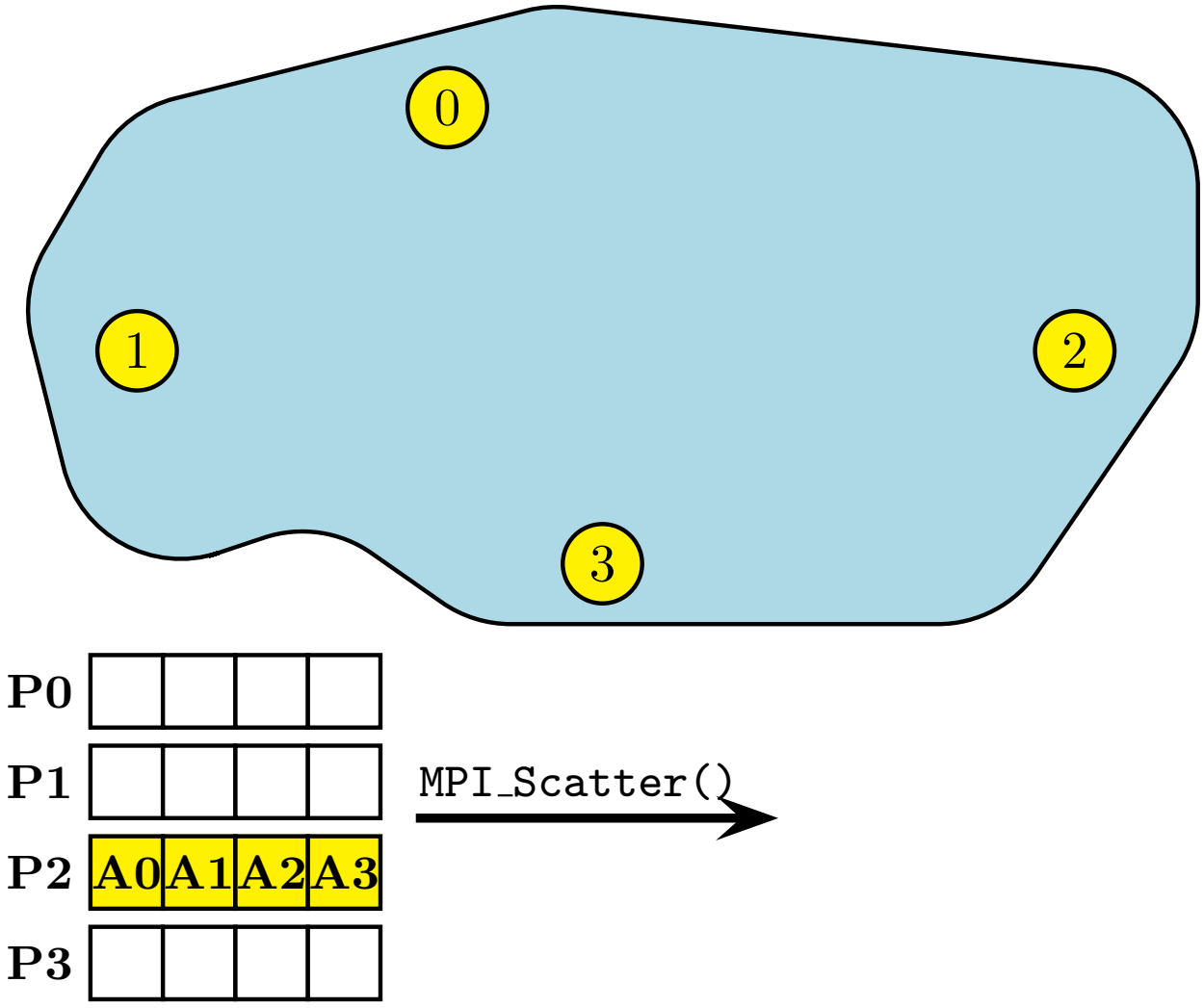


FIGURE 13 – Diffusion sélective : MPI_Scatter()

4.4 – Diffusion sélective : MPI_Scatter()

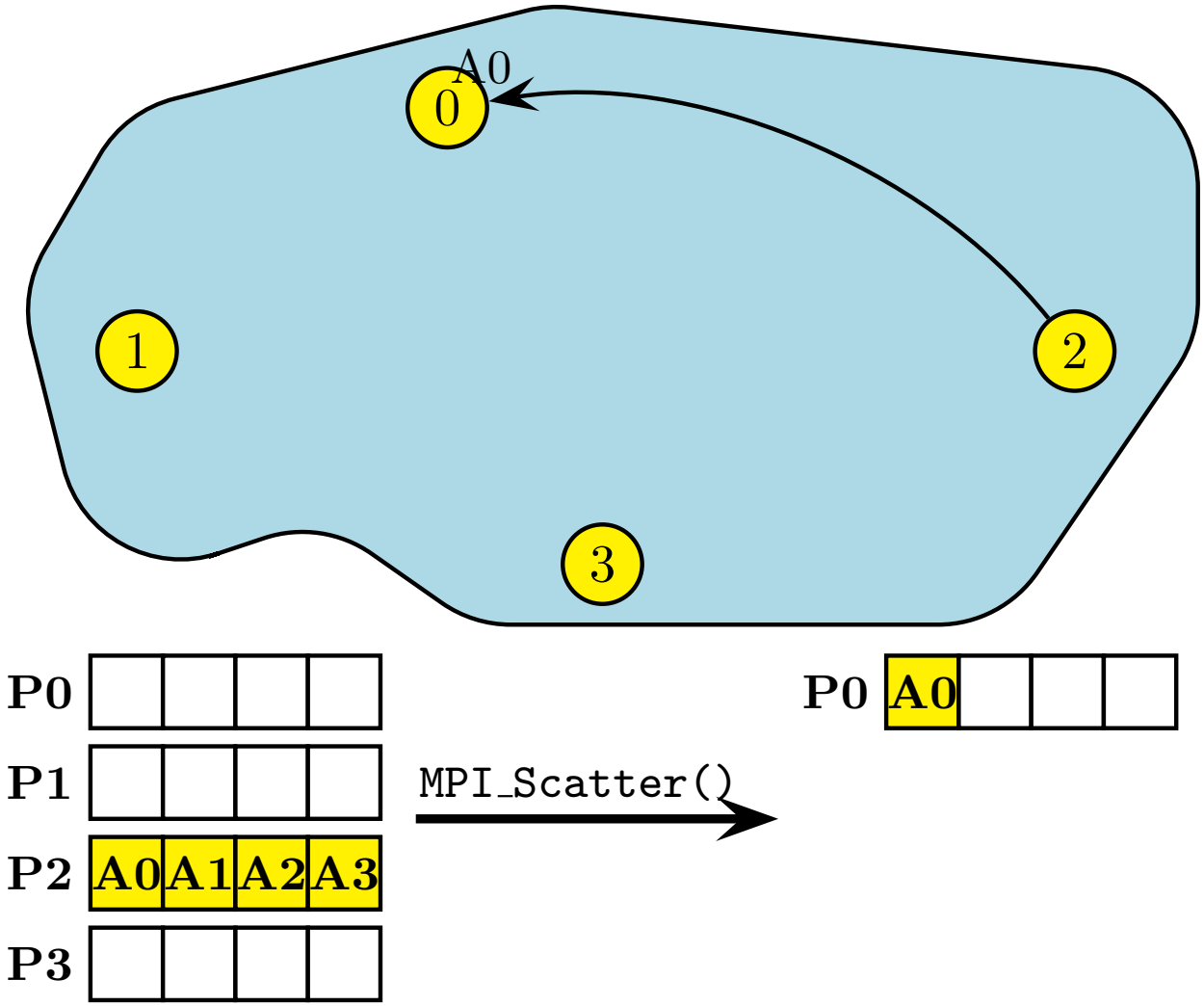


FIGURE 13 – Diffusion sélective : MPI_Scatter()

4.4 – Diffusion sélective : MPI_Scatter()

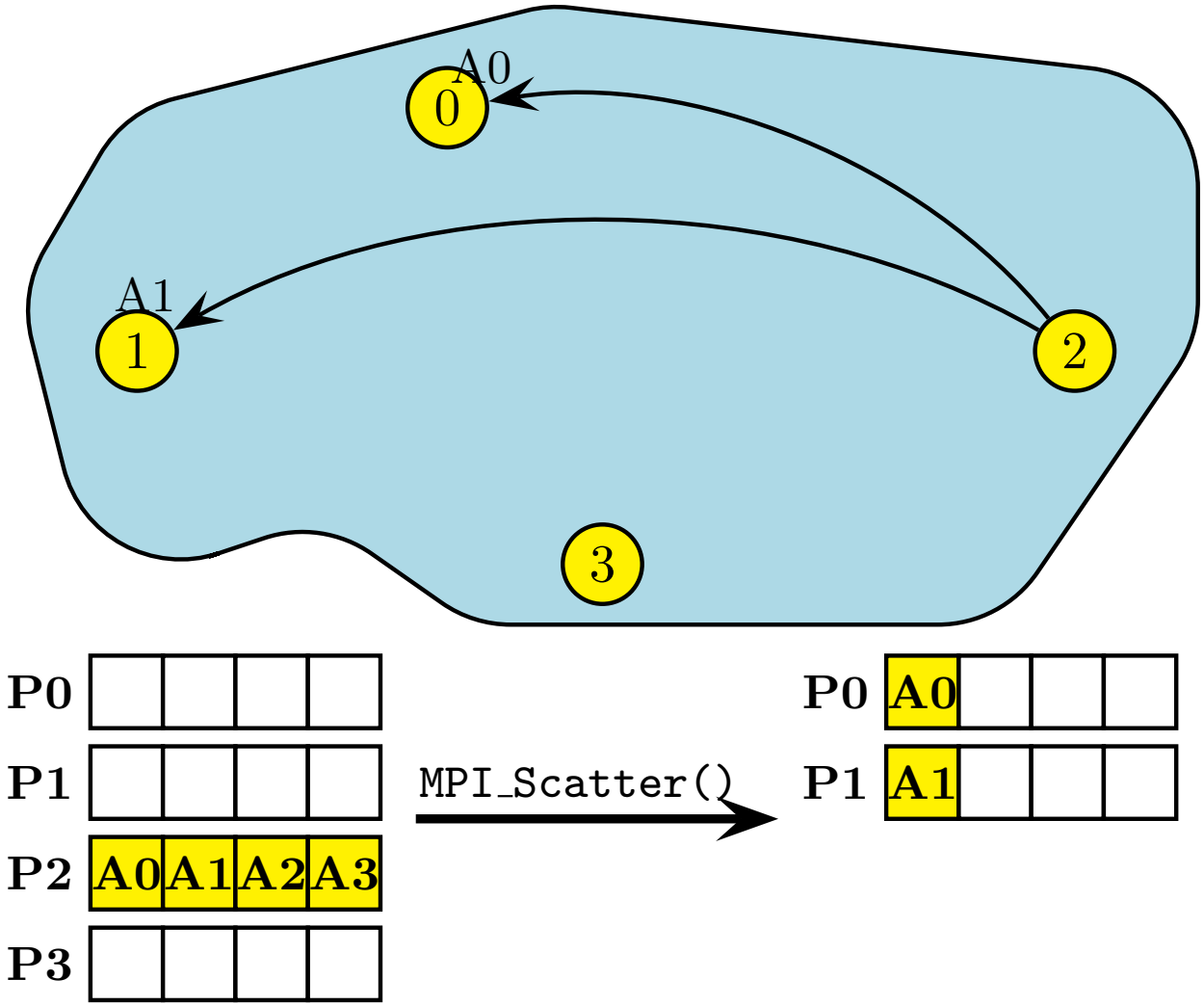


FIGURE 13 – Diffusion sélective : MPI_Scatter()

4.4 – Diffusion sélective : MPI_Scatter()

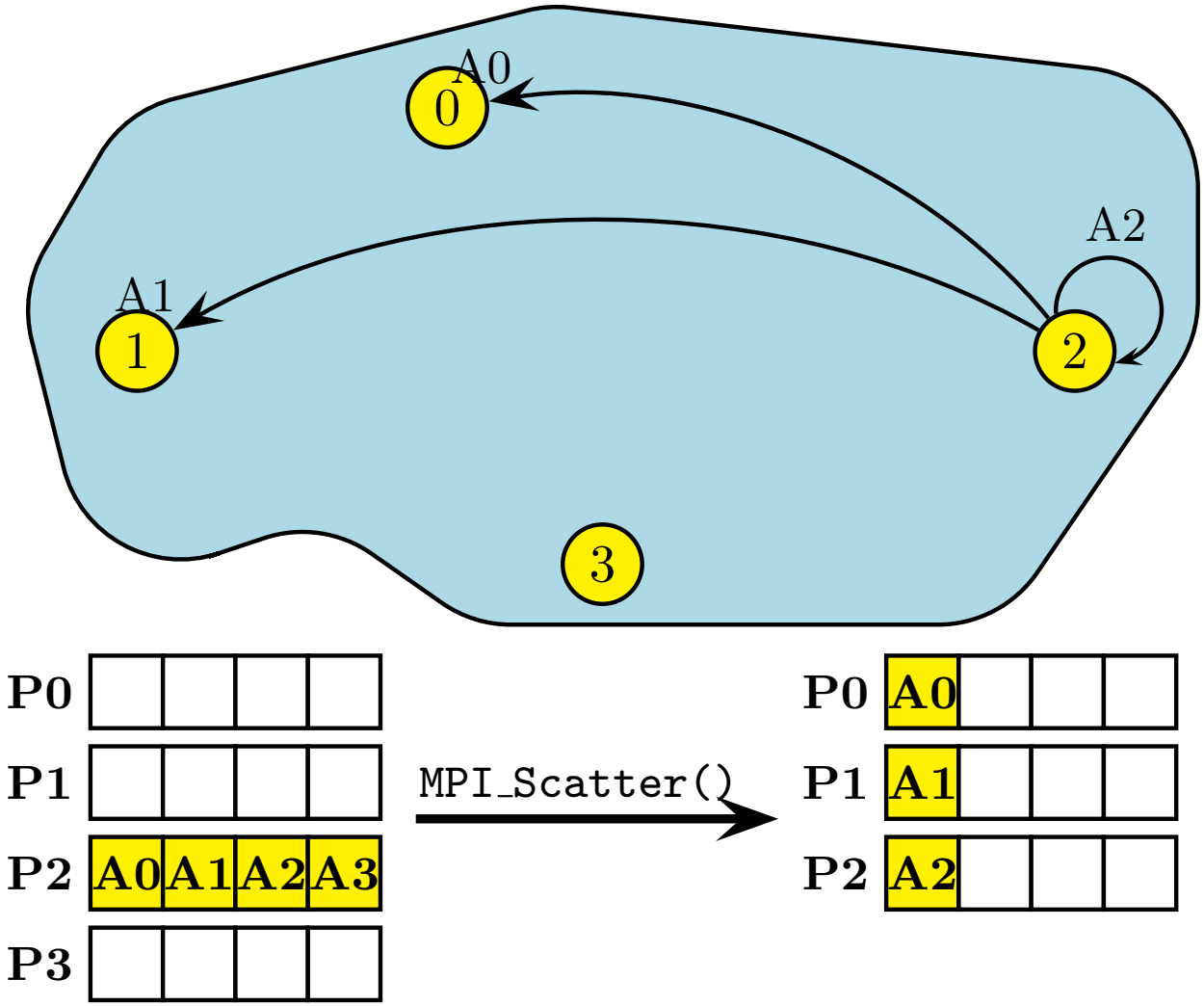


FIGURE 13 – Diffusion sélective : MPI_Scatter()

4.4 – Diffusion sélective : MPI_Scatter()

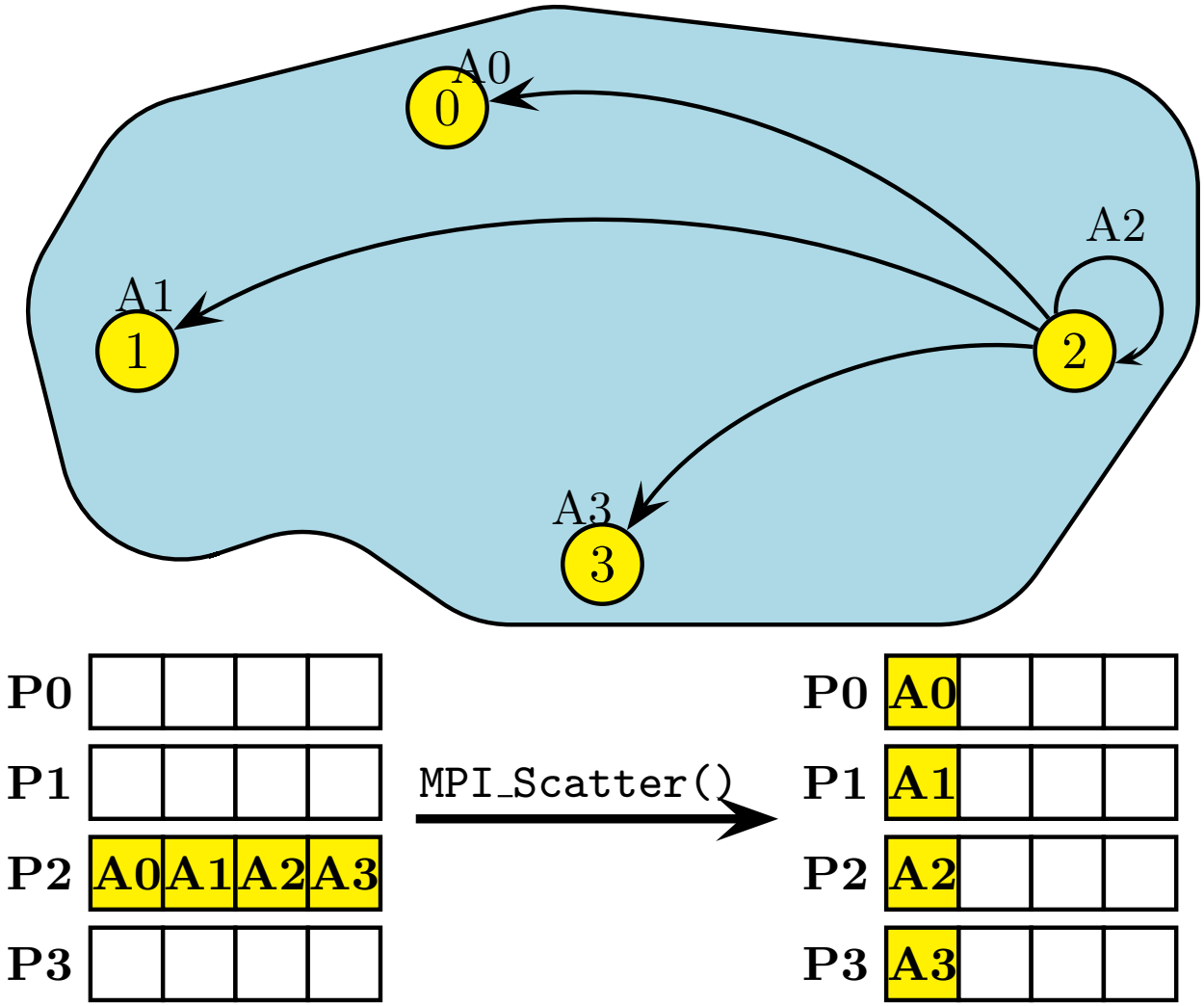


FIGURE 13 – Diffusion sélective : MPI_Scatter()

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define NB_VALEURS 128
5  int main(int argc, char *argv[])
6  {
7    int rang, nb_procs, longueur_tranche, i;
8    float *valeurs, *donnees;
9
10   MPI_Init(&argc,&argv);
11   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
12   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
13
14   longueur_tranche=NB_VALEURS/nb_procs;
15   donnees=(float*)malloc(longueur_tranche*sizeof(float));
16
17   if (rang == 2) {
18     valeurs=(float*) malloc(NB_VALEURS*sizeof(float));
19     for (i=0; i<NB_VALEURS; i++) valeurs[i]=(float)(1000+i);
20   }
21
22   MPI_Scatter(valeurs,longueur_tranche,MPI_FLOAT,donnees,longueur_tranche,
23               MPI_FLOAT,2,MPI_COMM_WORLD);
24   printf("Moi, processus %d, j'ai reçu %.0f à  %.0f du processus 2\n",
25           rang, donnees[0], donnees[longueur_tranche-1]);
26   free(valeurs); free(donnees); MPI_Finalize(); return(0);
27 }
```

```
> mpirun -np 4 scatter
Moi, processus 0, j'ai reçu 1000. à 1031. du processus 2
Moi, processus 1, j'ai reçu 1032. à 1063. du processus 2
Moi, processus 3, j'ai reçu 1096. à 1127. du processus 2
Moi, processus 2, j'ai reçu 1064. à 1095. du processus 2
```

4.5 – Collecte : MPI_Gather()

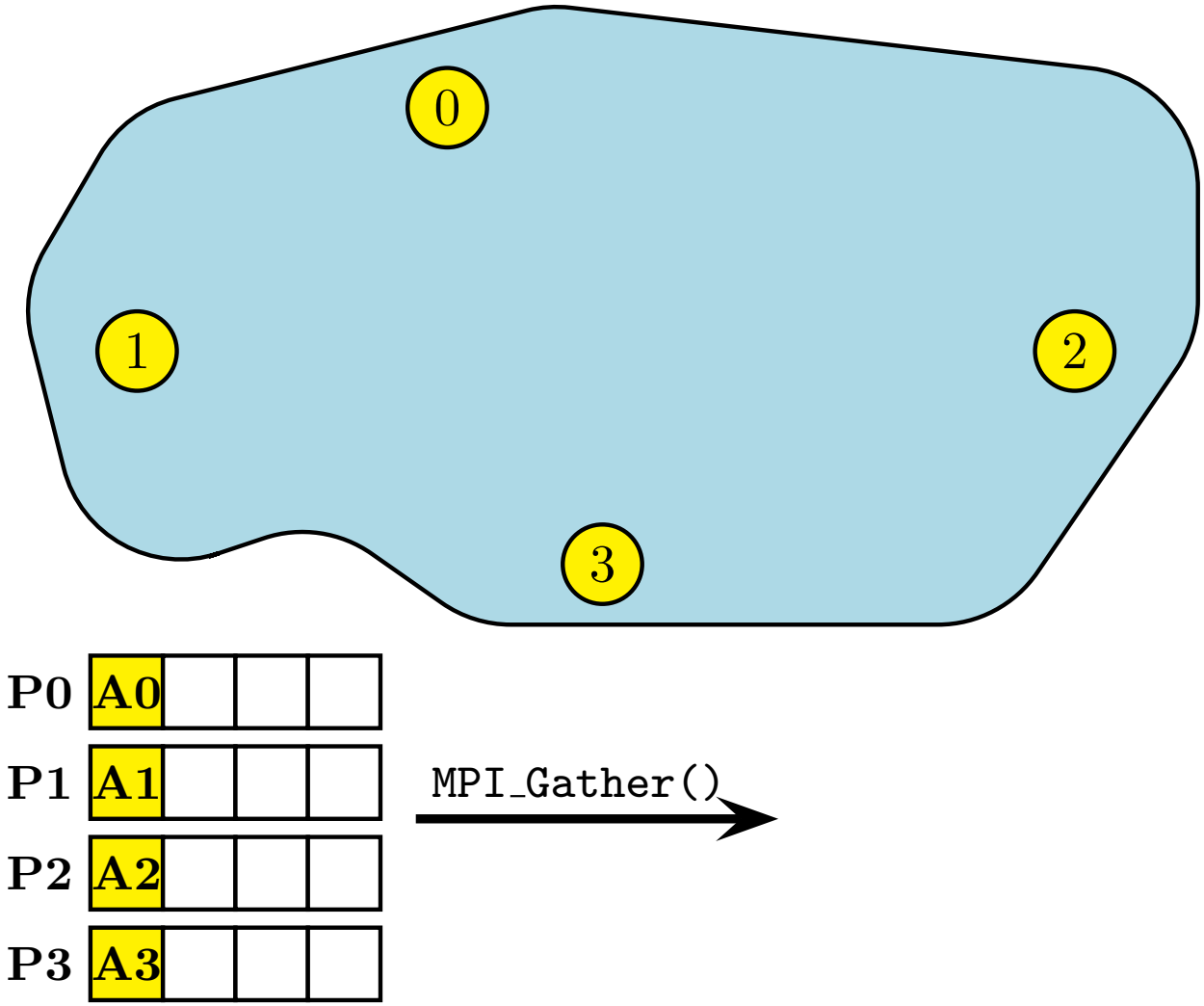


FIGURE 14 – Collecte : MPI_Gather()

4.5 – Collecte : MPI_Gather()

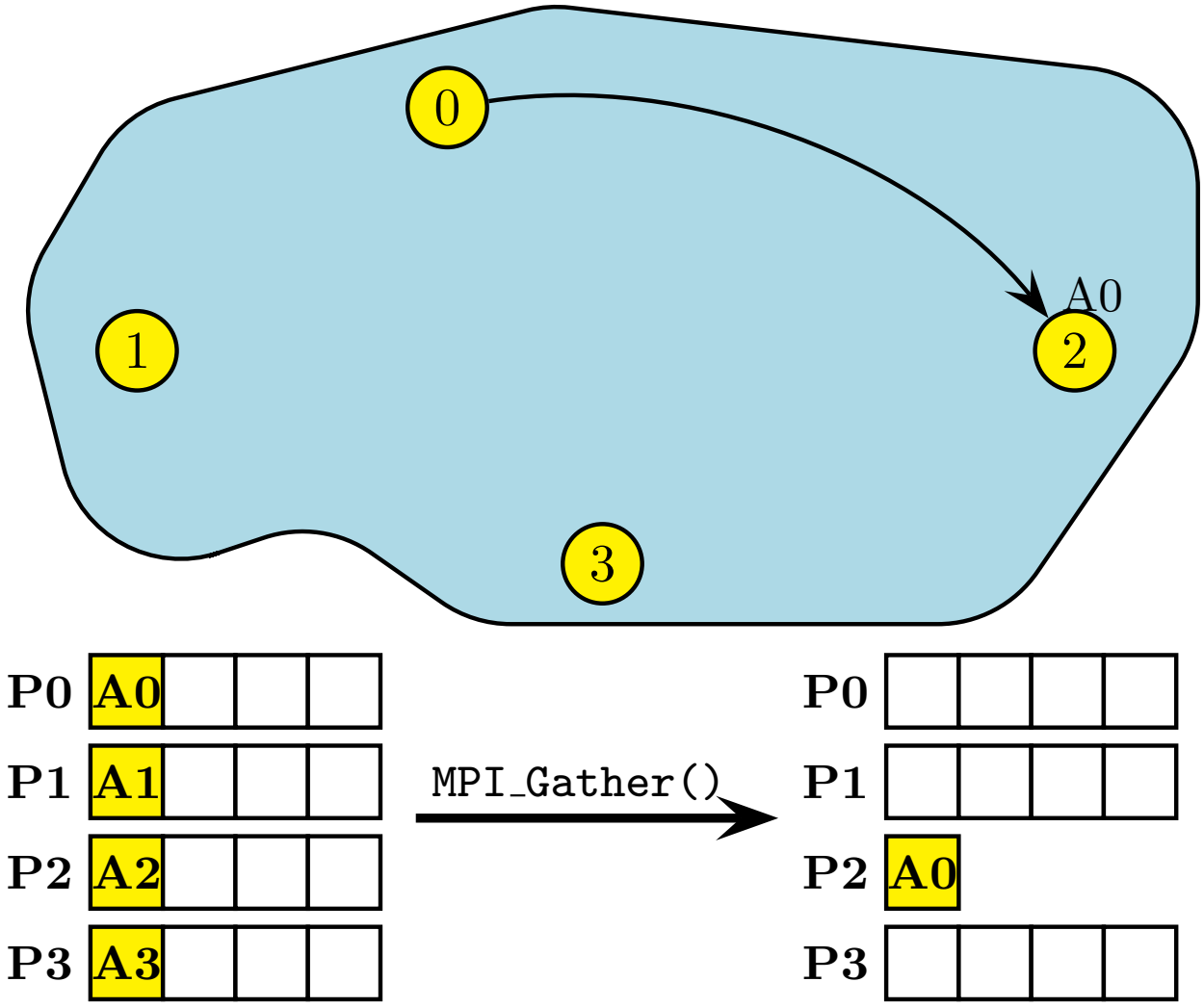


FIGURE 14 – Collecte : MPI_Gather()

4.5 – Collecte : MPI_Gather()

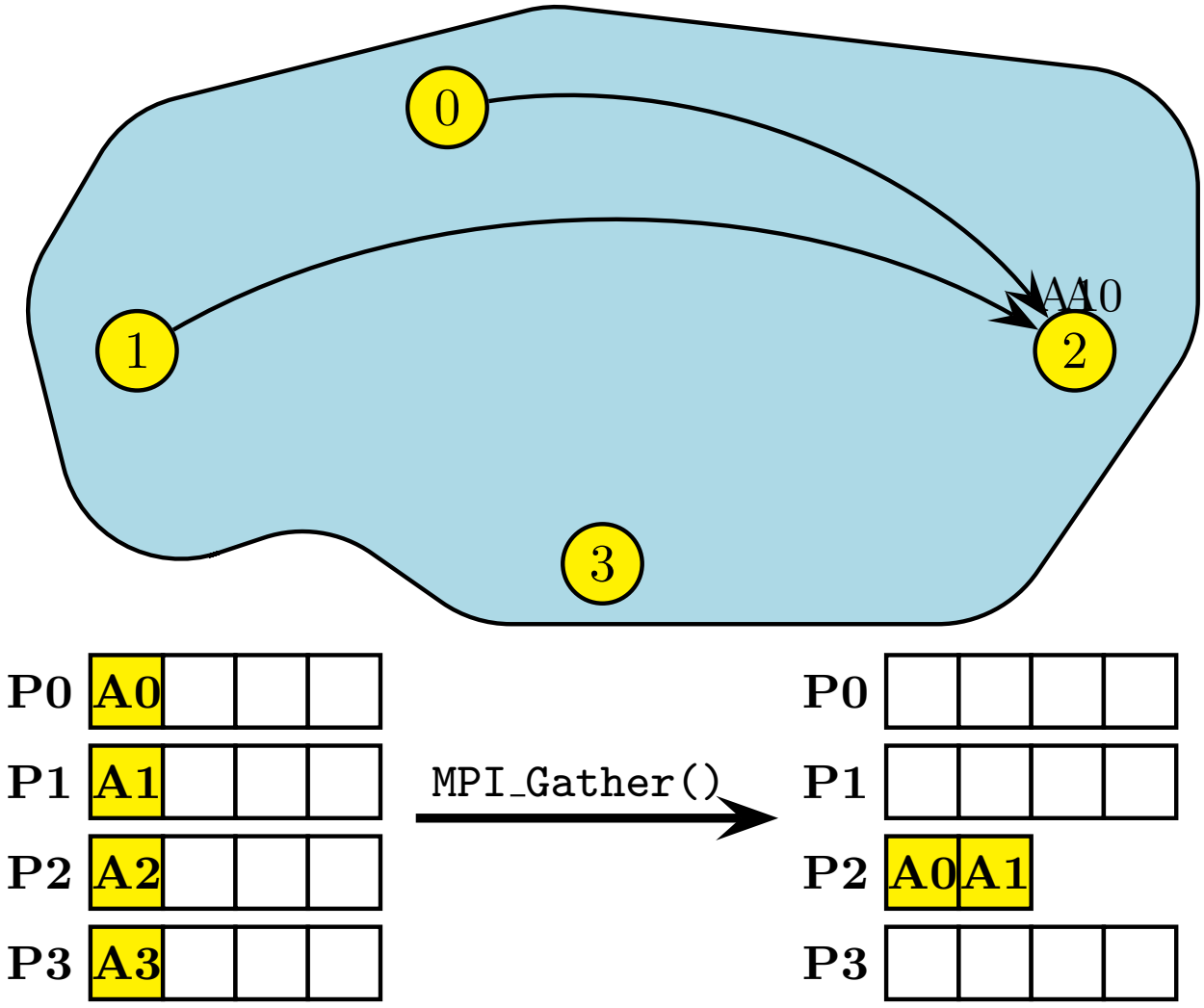


FIGURE 14 – Collecte : MPI_Gather()

4.5 – Collecte : MPI_Gather()

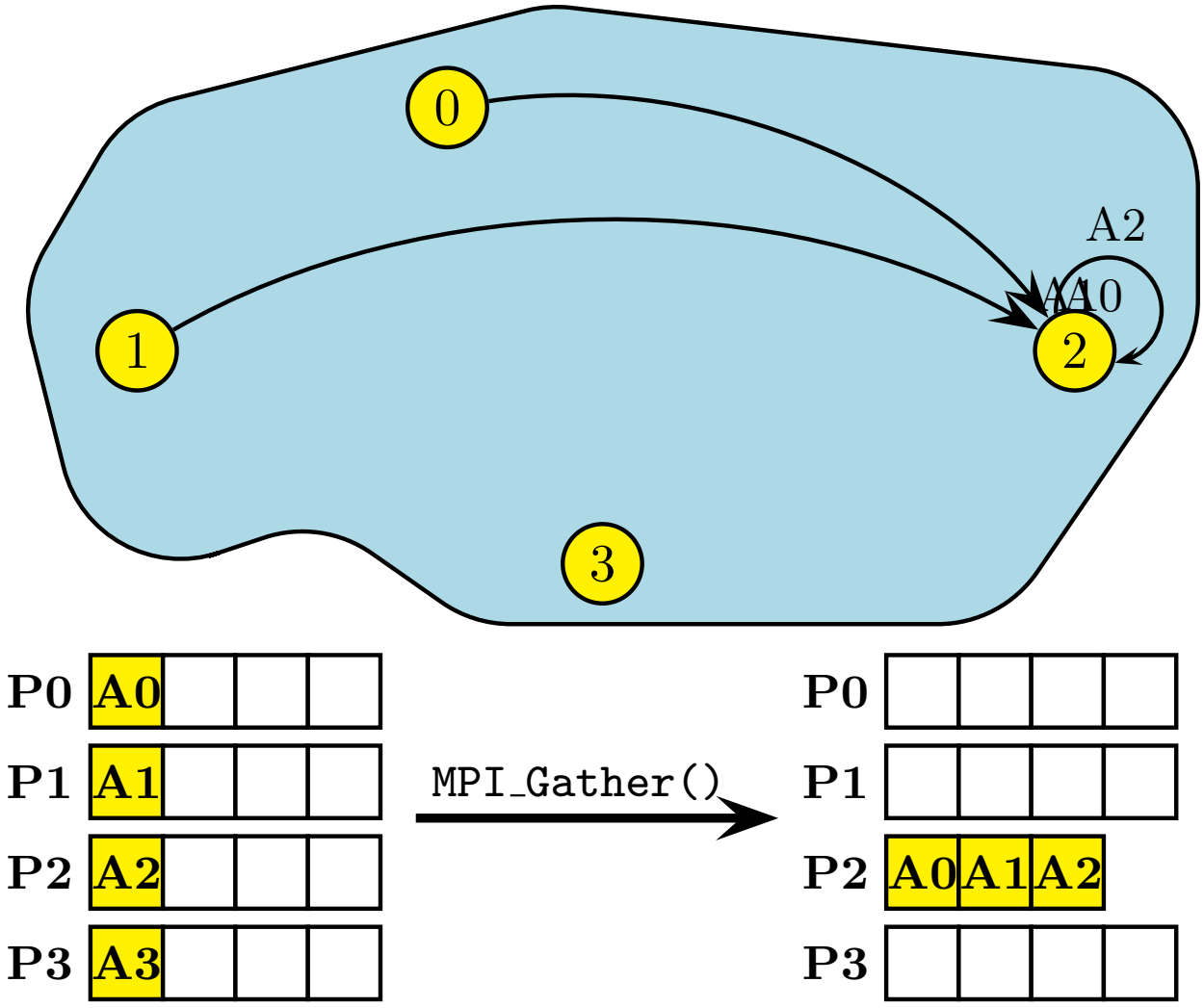


FIGURE 14 – Collecte : MPI_Gather()

4.5 – Collecte : MPI_Gather()

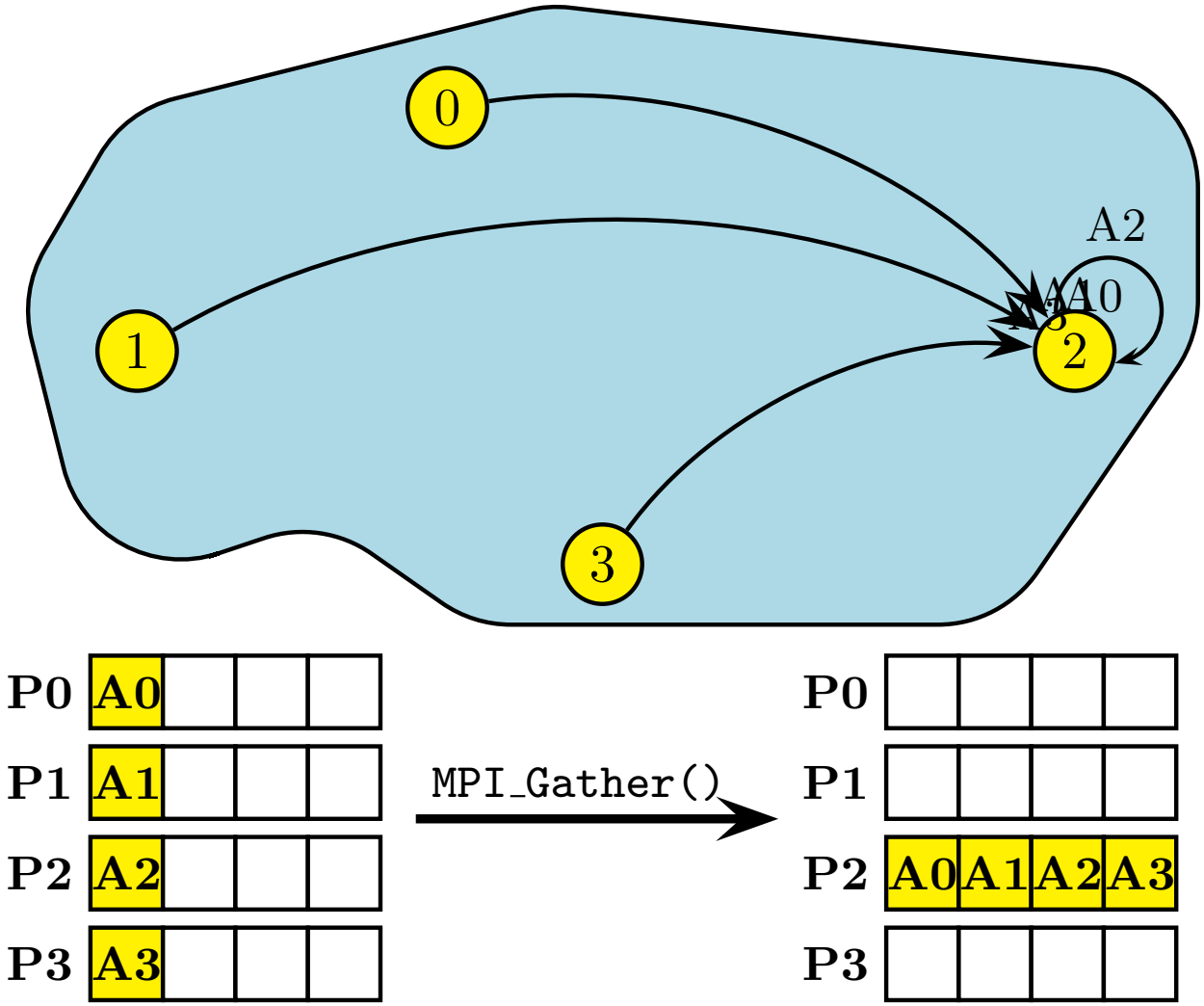


FIGURE 14 – Collecte : MPI_Gather()

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define NB_VALEURS 128
5  int main(int argc, char *argv[])
6  {
7    int rang, nb_procs, longueur_tranche, i;
8    float *valeurs, donnees[NB_VALEURS];
9
10   MPI_Init(&argc,&argv);
11   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
12   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
13
14   longueur_tranche=NB_VALEURS/nb_procs;
15   valeurs=(float *)malloc(longueur_tranche*sizeof(float));
16   for (i=0; i<longueur_tranche; i++)
17   valeurs[i]=(float)(1000+rang*longueur_tranche+i);
18
19   MPI_Gather(valeurs,longueur_tranche,MPI_FLOAT,donnees,longueur_tranche,
20              MPI_FLOAT,2,MPI_COMM_WORLD);
21
22   if (rang == 2) printf("Moi, processus 2, j'ai reçu %.0f, ...,  %.0f, ..., %.0f\n",
23                         donnees[0], donnees[longueur_tranche], donnees[NB_VALEURS-1]);
24   free(valeurs); MPI_Finalize(); return(0);
```

```
> mpirun -np 4 gather
Moi, processus 2, j'ai reçu 1000. ... 1032. ... 1127.
```

4.6 – Collecte générale : MPI_Allgather()

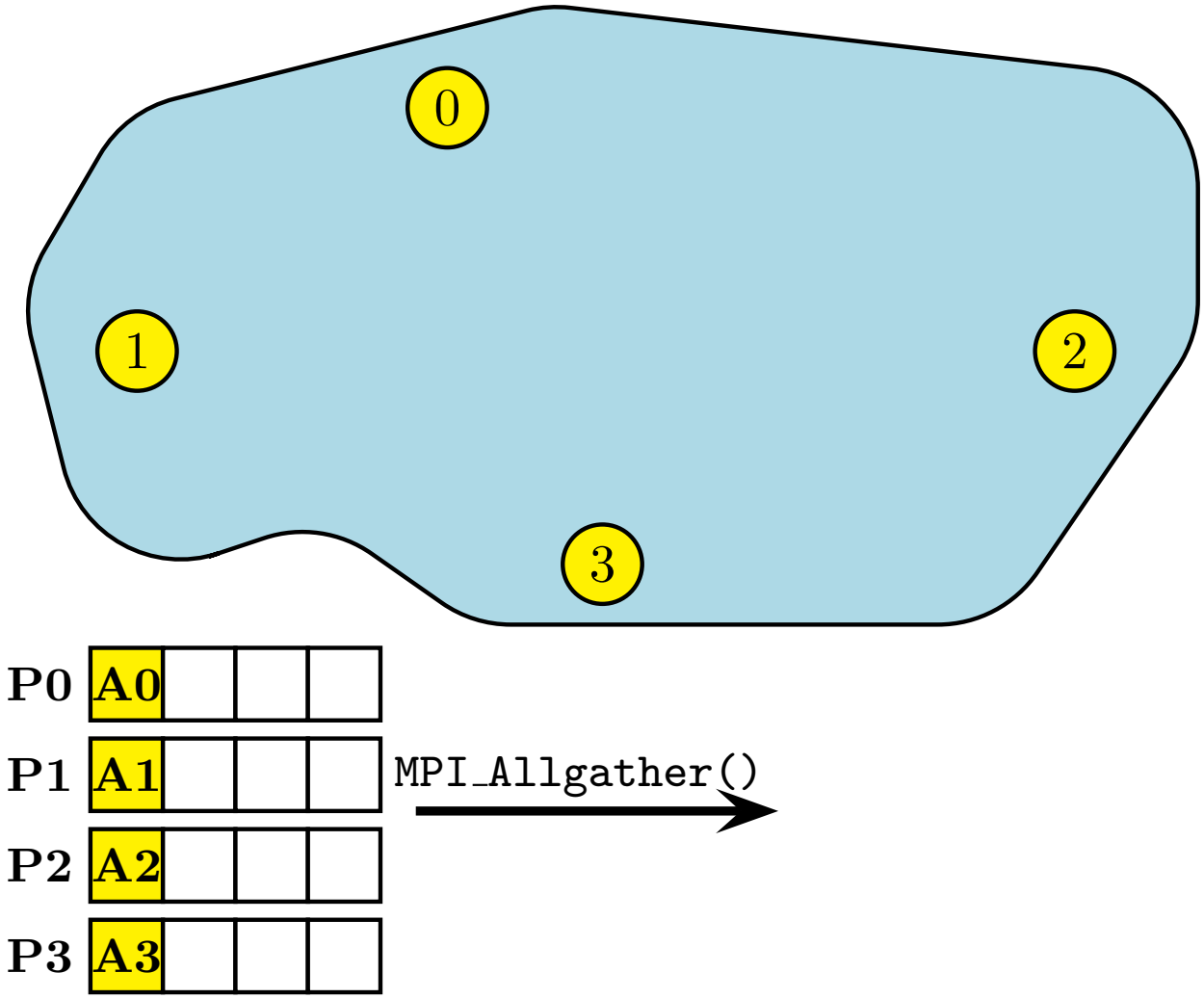


FIGURE 15 – Collecte générale : MPI_Allgather()

4.6 – Collecte générale : MPI_Allgather()

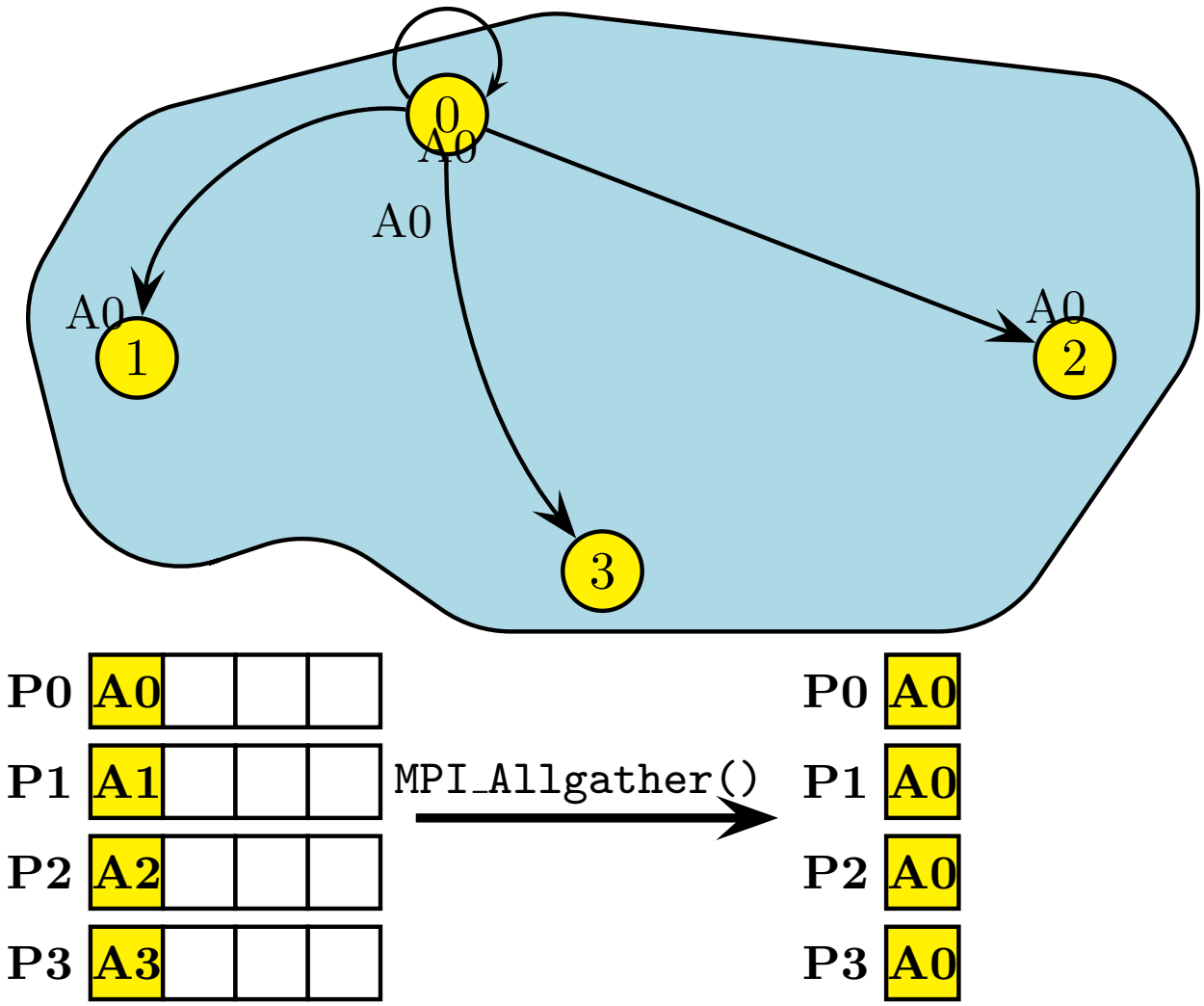


FIGURE 15 – Collecte générale : MPI_Allgather()

4.6 – Collecte générale : MPI_Allgather()

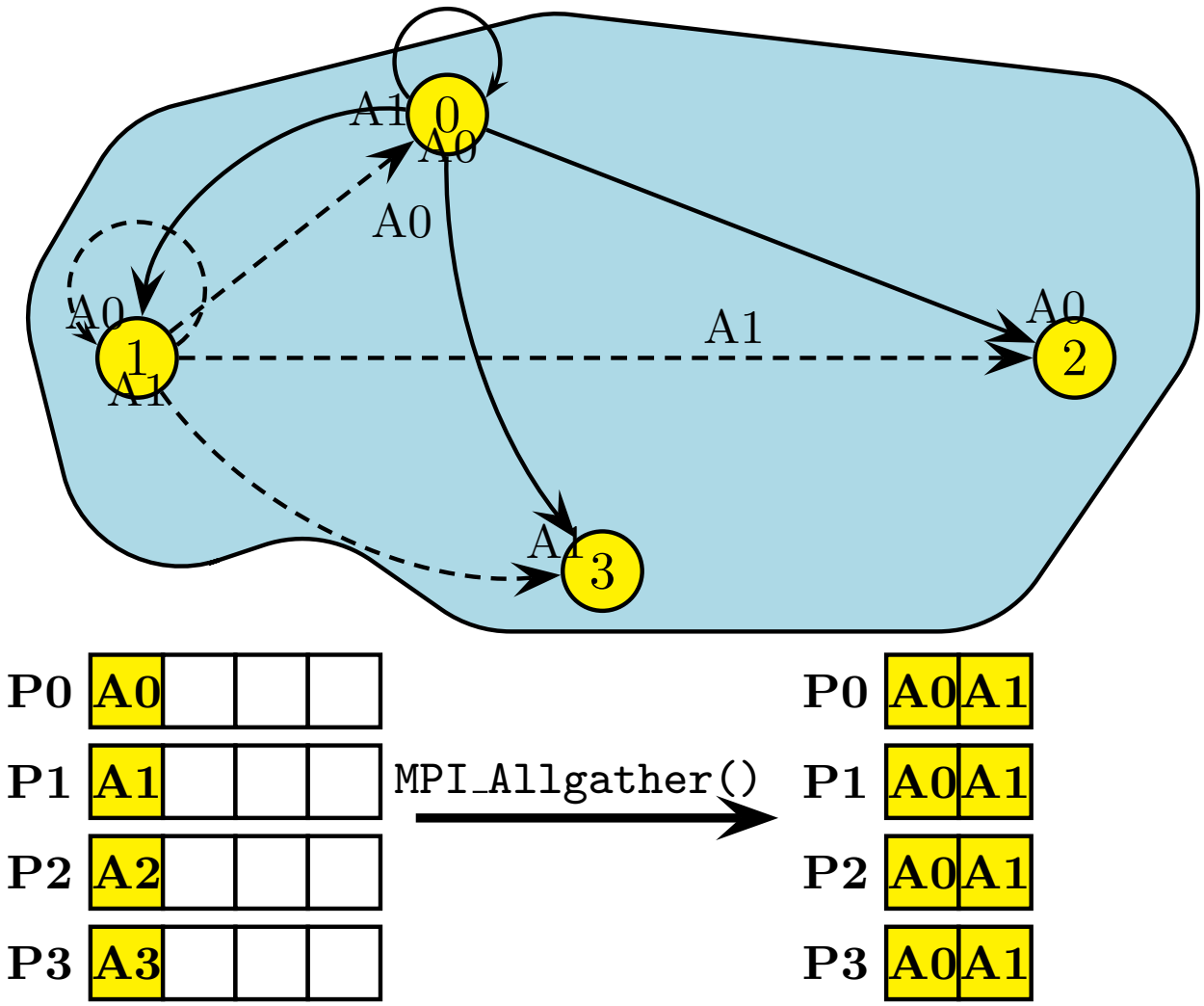


FIGURE 15 – Collecte générale : MPI_Allgather()

4.6 – Collecte générale : MPI_Allgather()

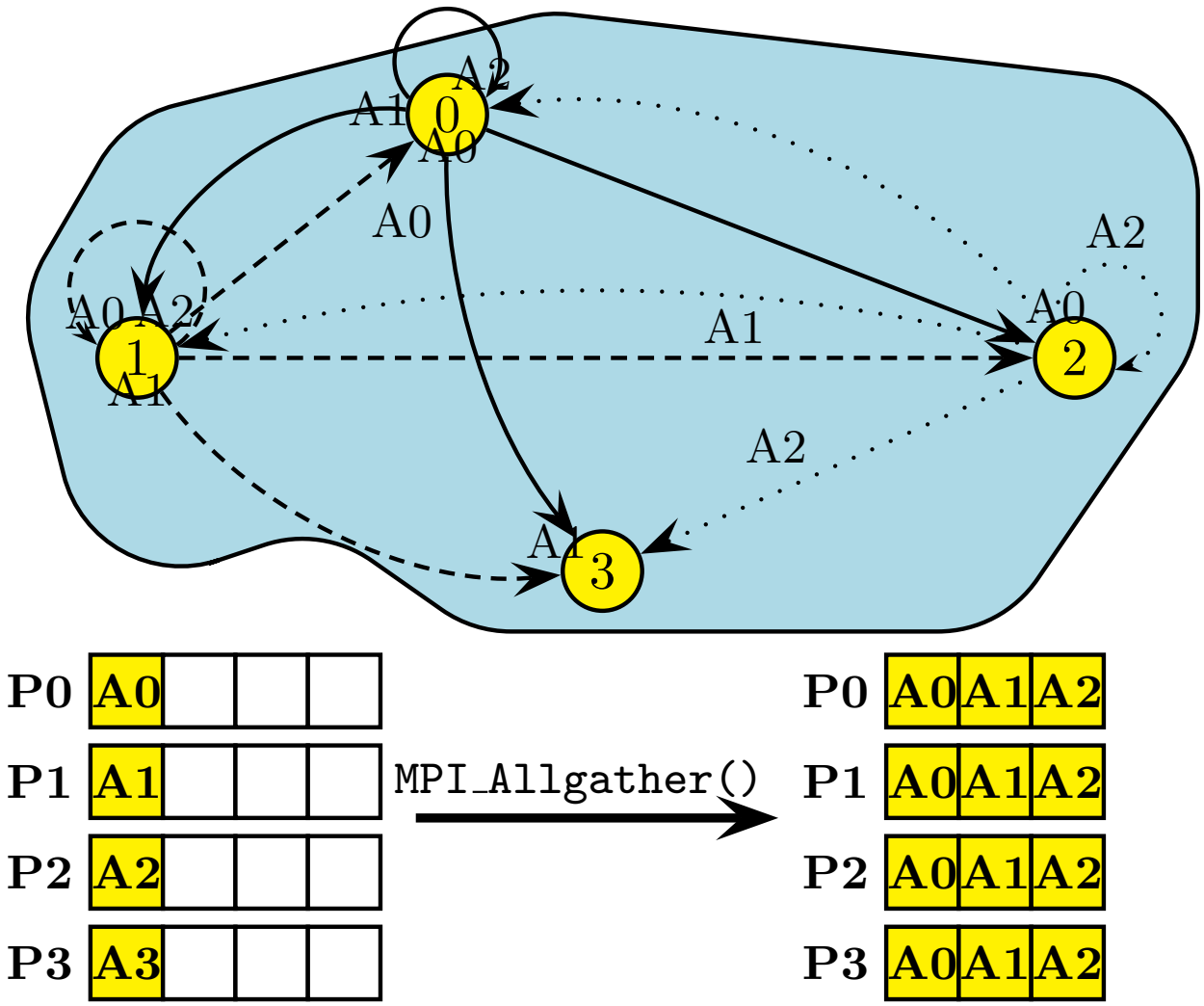


FIGURE 15 – Collecte générale : MPI_Allgather()

4.6 – Collecte générale : MPI_Allgather()

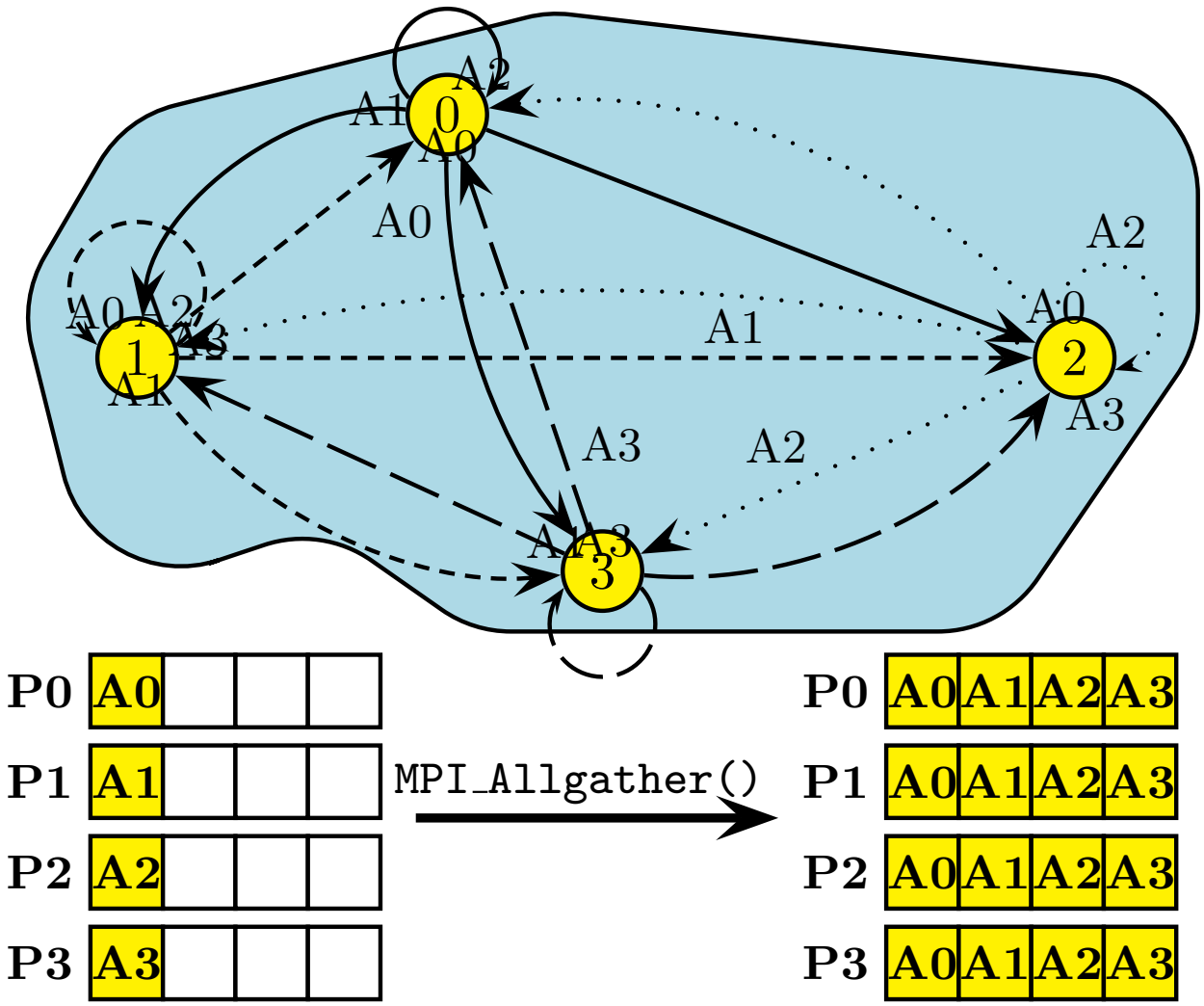


FIGURE 15 – Collecte générale : MPI_Allgather()

```c
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#define NB_VALEURS 128
int main(int argc, char *argv[])
{
  int rang, nb_procs, longueur_tranche, i;
  float *valeurs, donnees[NB_VALEURS];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rang);
  longueur_tranche=NB_VALEURS/nb_procs;
  valeurs=(float*)malloc(longueur_tranche*sizeof(float));
  for (i=0; i<longueur_tranche; i++) valeurs[i]=(float)(1000+rang*longueur_tranche+i);
  MPI_Allgather(valeurs,longueur_tranche,MPI_FLOAT,donnees,longueur_tranche,
                MPI_FLOAT,MPI_COMM_WORLD);
  printf("Moi, processus %d, j'ai reçu %.0f, ...,  %.0f, ..., %.0f\n", rang,
          donnees[0], donnees[longueur_tranche], donnees[NB_VALEURS-1]);
  free(valeurs); MPI_Finalize(); return(0);
}
```

```
> mpirun -np 4 allgather
Moi, processus 1, j'ai reçu 1000. ... 1032. ... 1127.
Moi, processus 3, j'ai reçu 1000. ... 1032. ... 1127.
Moi, processus 2, j'ai reçu 1000. ... 1032. ... 1127.
Moi, processus 0, j'ai reçu 1000. ... 1032. ... 1127.
```

4.7 – Échanges croisés : MPI_Alltoall()

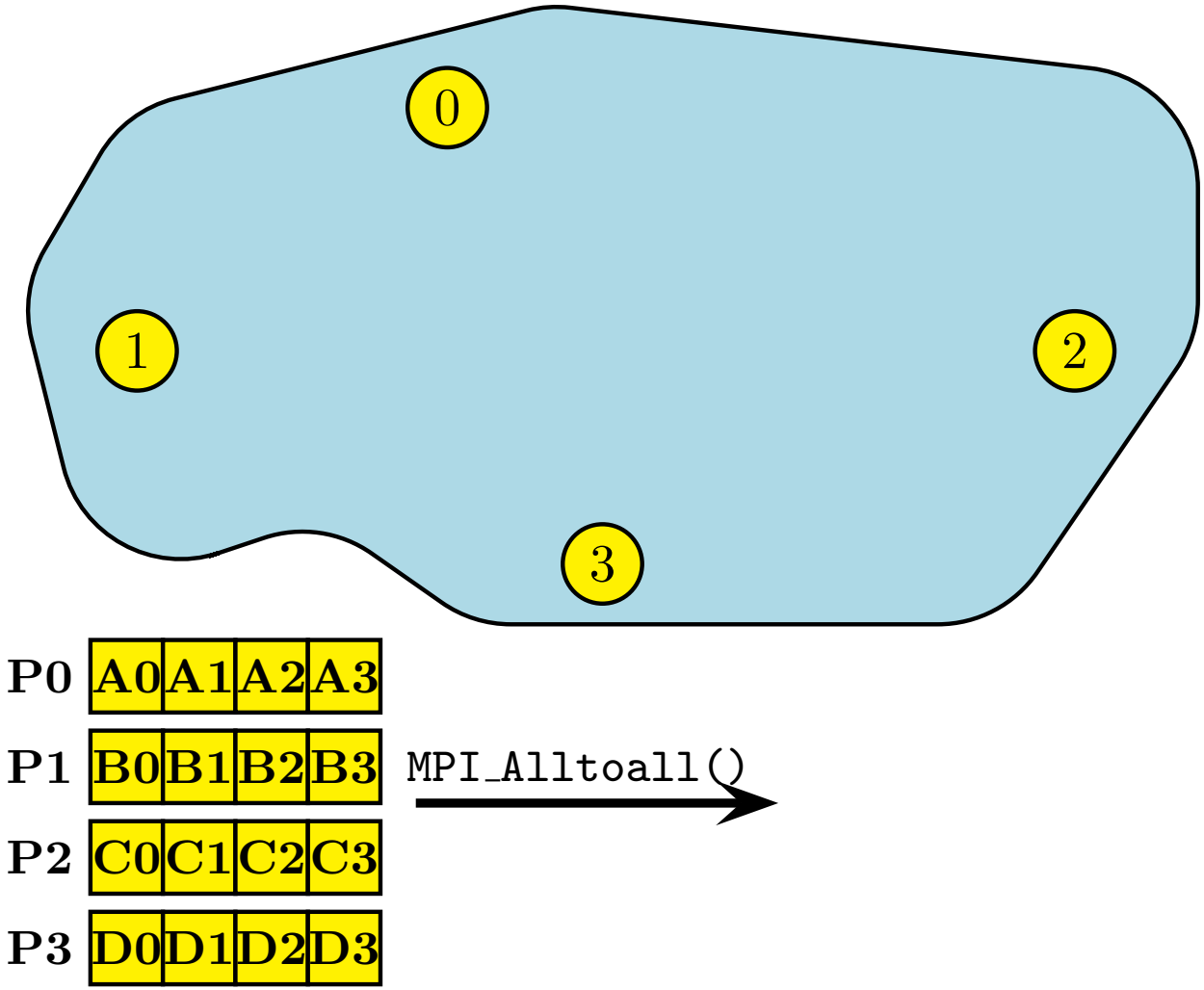


FIGURE 16 – Échanges croisés : MPI_Alltoall()

4.7 – Échanges croisés : `MPI_Alltoall()`

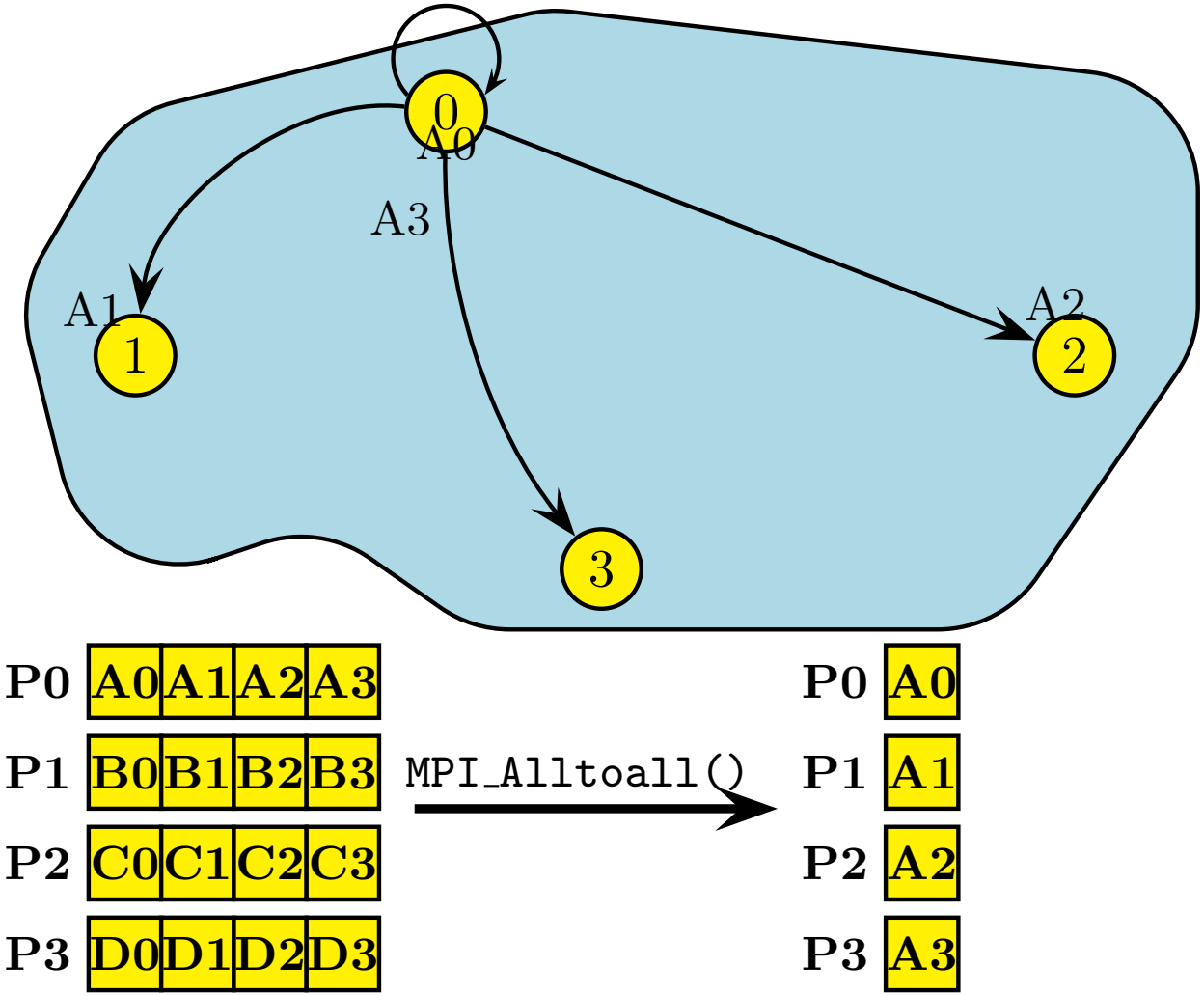


FIGURE 16 – Échanges croisés : `MPI_Alltoall()`

4.7 – Échanges croisés : MPI_Alltoall()

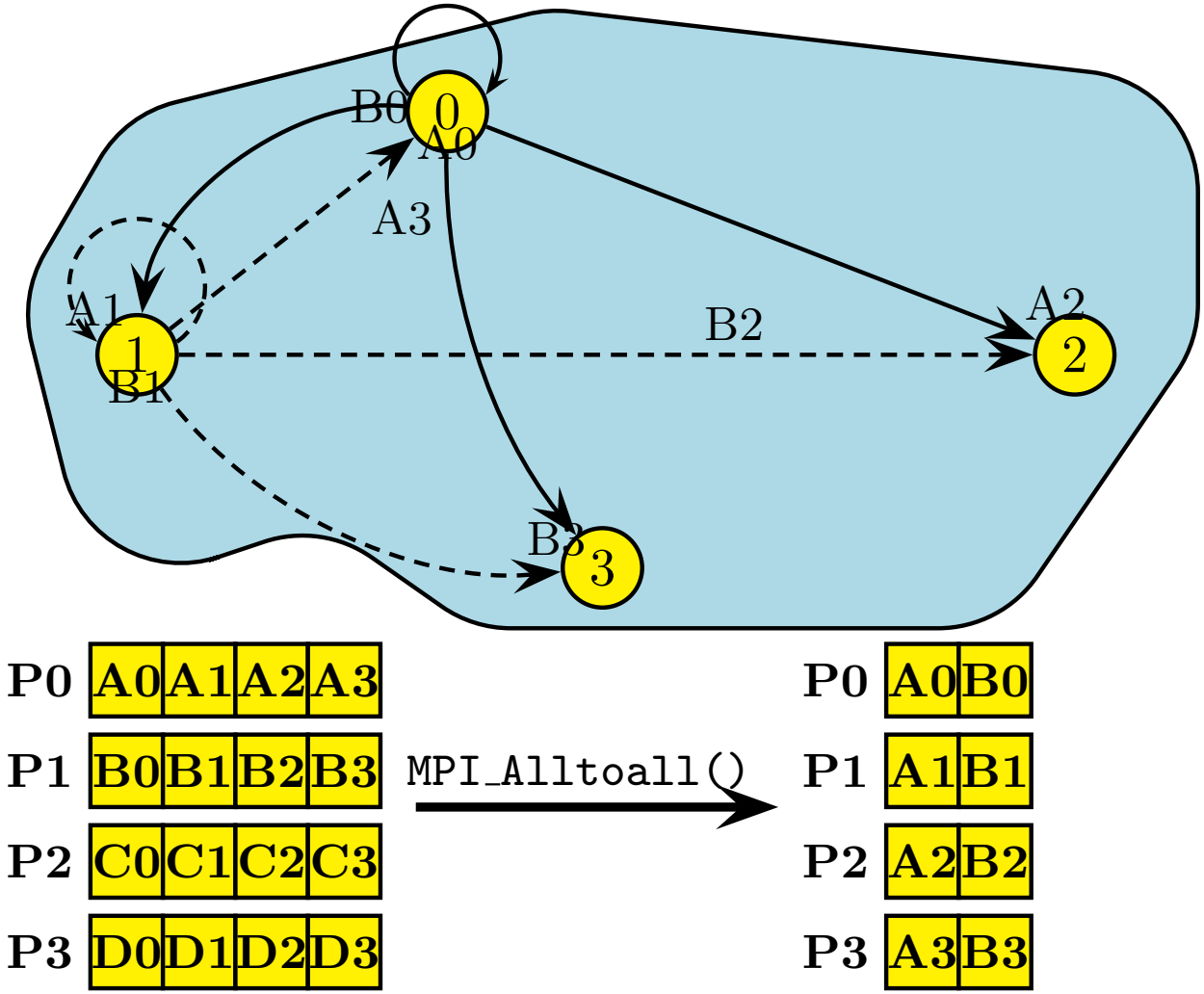
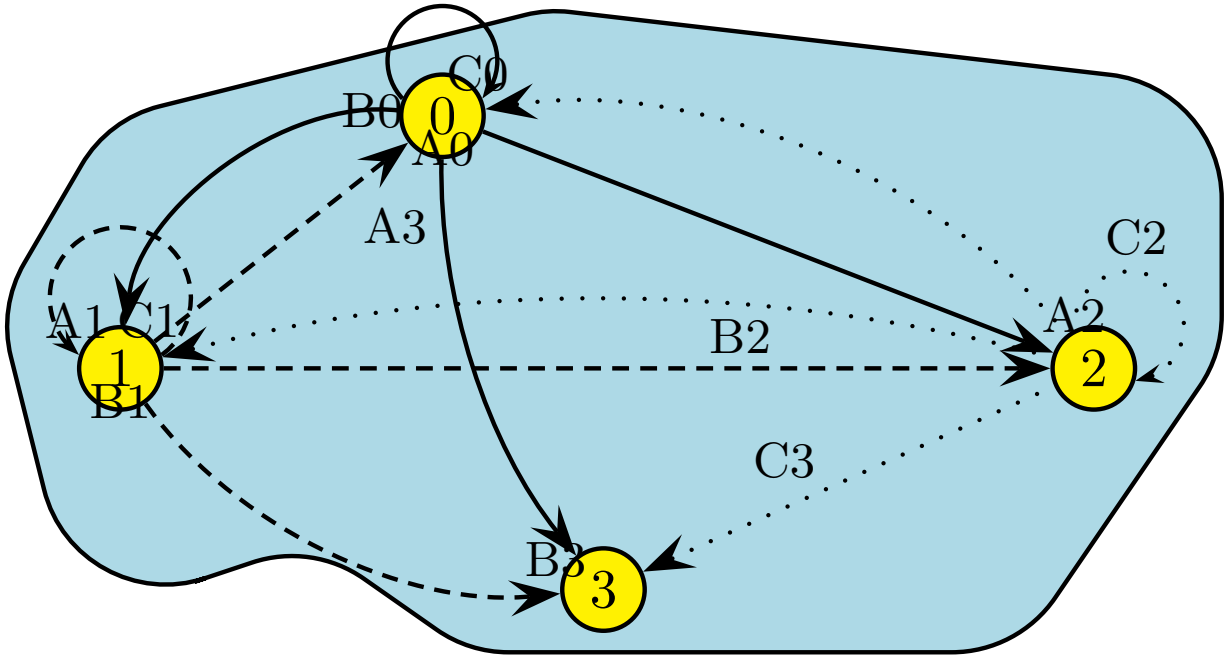


FIGURE 16 – Échanges croisés : MPI_Alltoall()

4.7 – Échanges croisés : MPI_Alltoall()



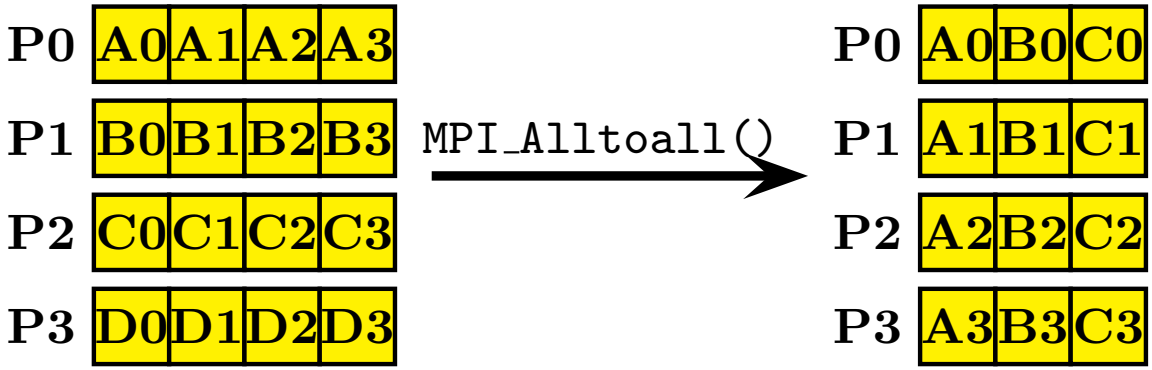
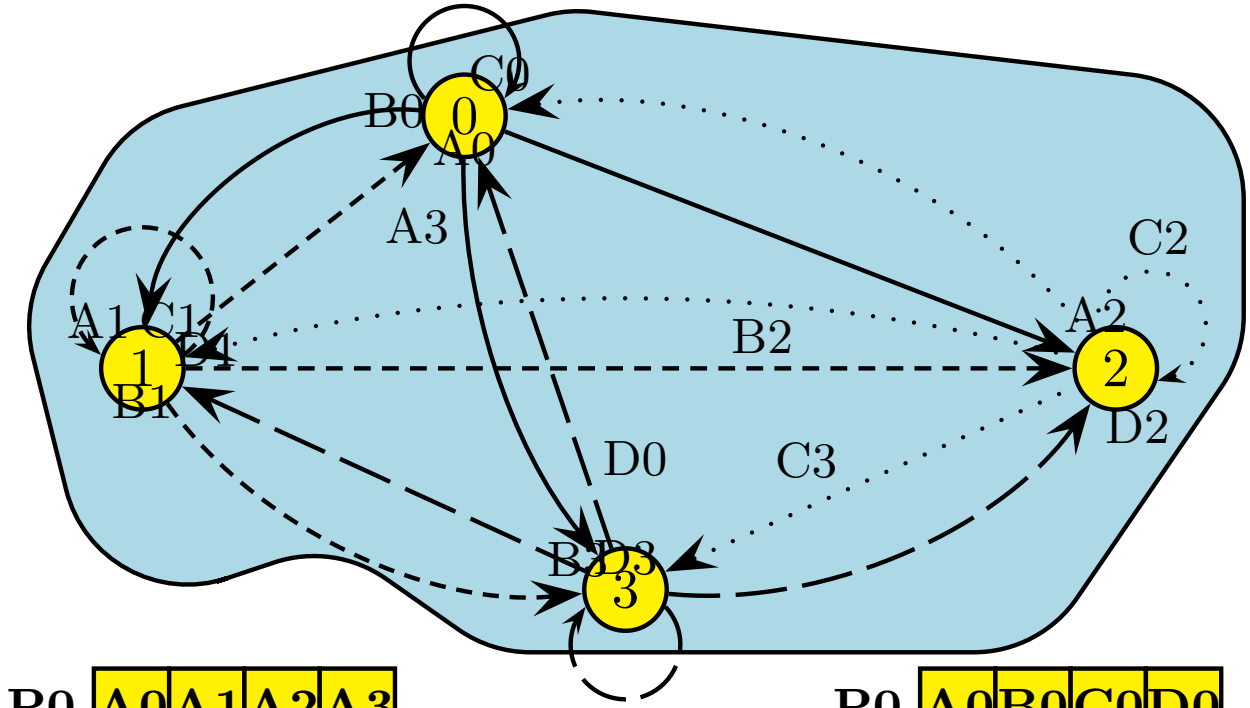


FIGURE 16 – Échanges croisés : MPI_Alltoall()

4.7 – Échanges croisés : MPI_Alltoall()



P0	A0	A1	A2	A3		P0	A0	B0	C0	D0
P1	B0	B1	B2	B3	MPI_Alltoall()	P1	A1	B1	C1	D1
P2	C0	C1	C2	C3		P2	A2	B2	C2	D2
P3	D0	D1	D2	D3		P3	A3	B3	C3	D3

FIGURE 16 – Échanges croisés : MPI_Alltoall()

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define NB_VALEURS 128
5  int main(int argc, char *argv[])
6  {
7    int rang, nb_procs, longueur_tranche, i;
8    float valeurs[NB_VALEURS], donnees[NB_VALEURS];
9
10   MPI_Init(&argc,&argv);
11   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
12   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
13
14   for (i=0; i<NB_VALEURS; i++) valeurs[i]=(float)(1000+rang*NB_VALEURS+i);
15   longueur_tranche=NB_VALEURS/nb_procs;
16
17   MPI_Alltoall(valeurs,longueur_tranche,MPI_FLOAT,donnees,longueur_tranche,
18                MPI_FLOAT,MPI_COMM_WORLD);
19   printf("Moi, processus %d, j'ai reçu %.0f, ...,  %.0f, ..., %.0f\n", rang,
20           donnees[0], donnees[longueur_tranche], donnees[NB_VALEURS-1]);
21   MPI_Finalize(); return(0);
22 }
```

```
> mpirun -np 4 alltoall
Moi, processus 0, j'ai reçu 1000. ... 1128. ... 1415.
Moi, processus 2, j'ai reçu 1064. ... 1192. ... 1479.
Moi, processus 1, j'ai reçu 1032. ... 1160. ... 1447.
Moi, processus 3, j'ai reçu 1096. ... 1224. ... 1511.
```

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 fonctions 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, l'opération de réduction est appliquée à chacun d'entre eux.
- La fonction `MPI_Scan()` permet en plus d'effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du groupe.
- Les fonctions `MPI_Op_create()` et `MPI_Op_free()` permettent de définir des opérations de réduction personnelles.

TABLE 2 – 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

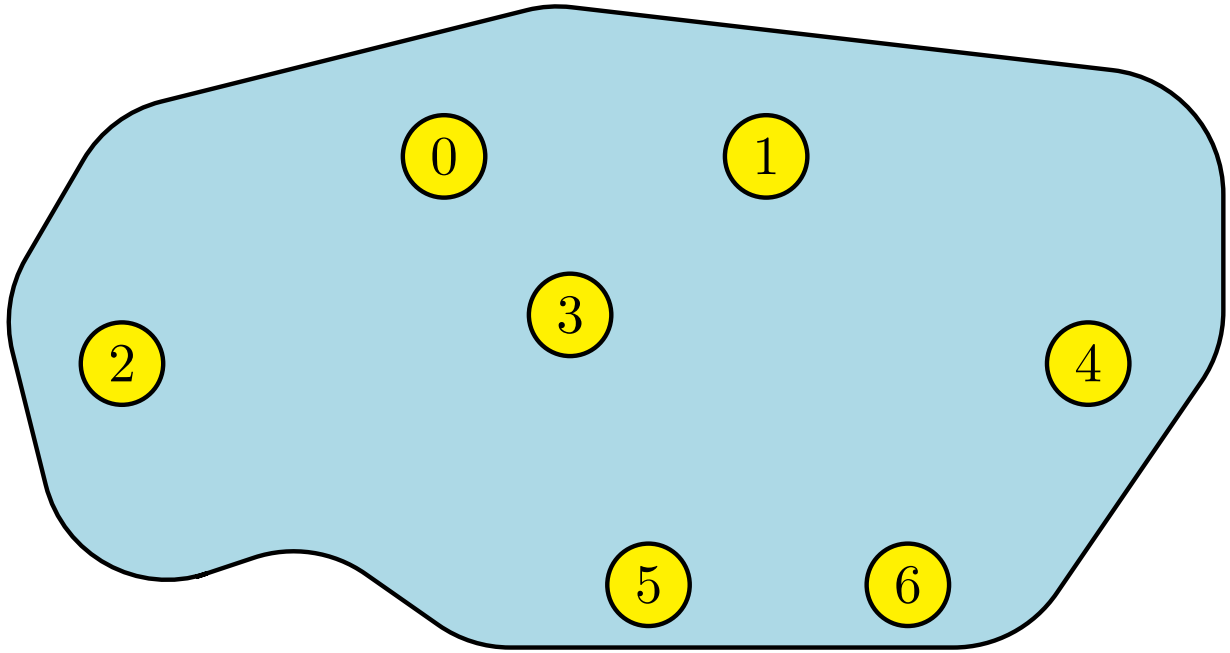


FIGURE 17 – Réduction répartie (somme)

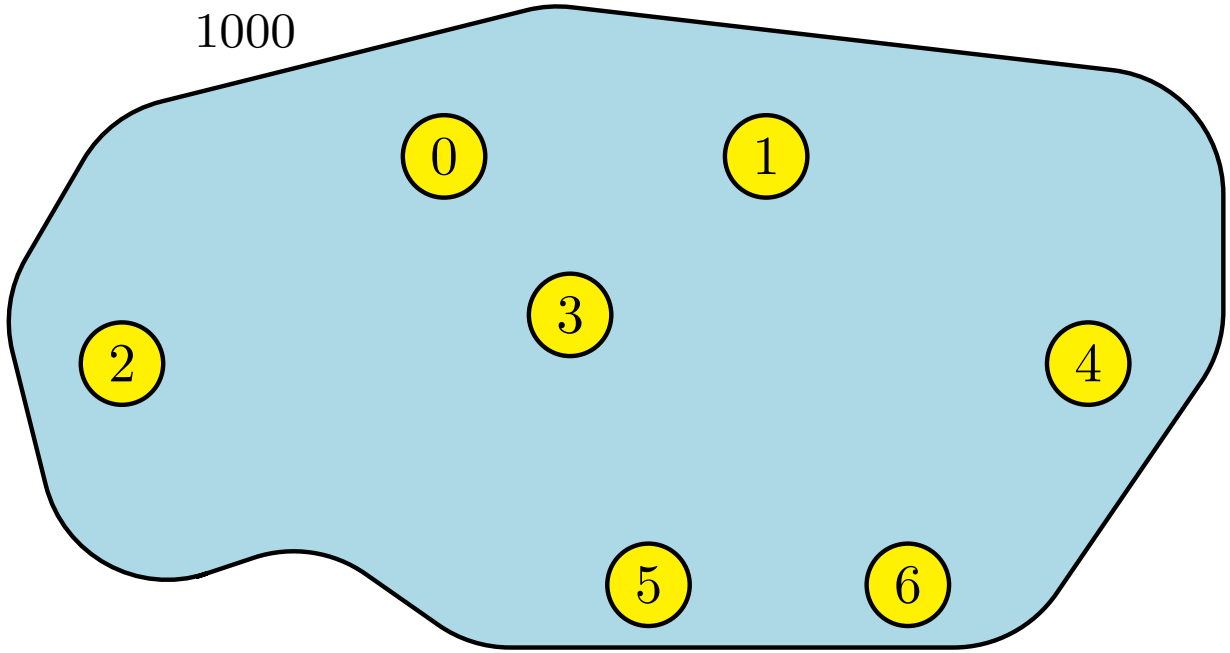


FIGURE 17 – Réduction répartie (somme)

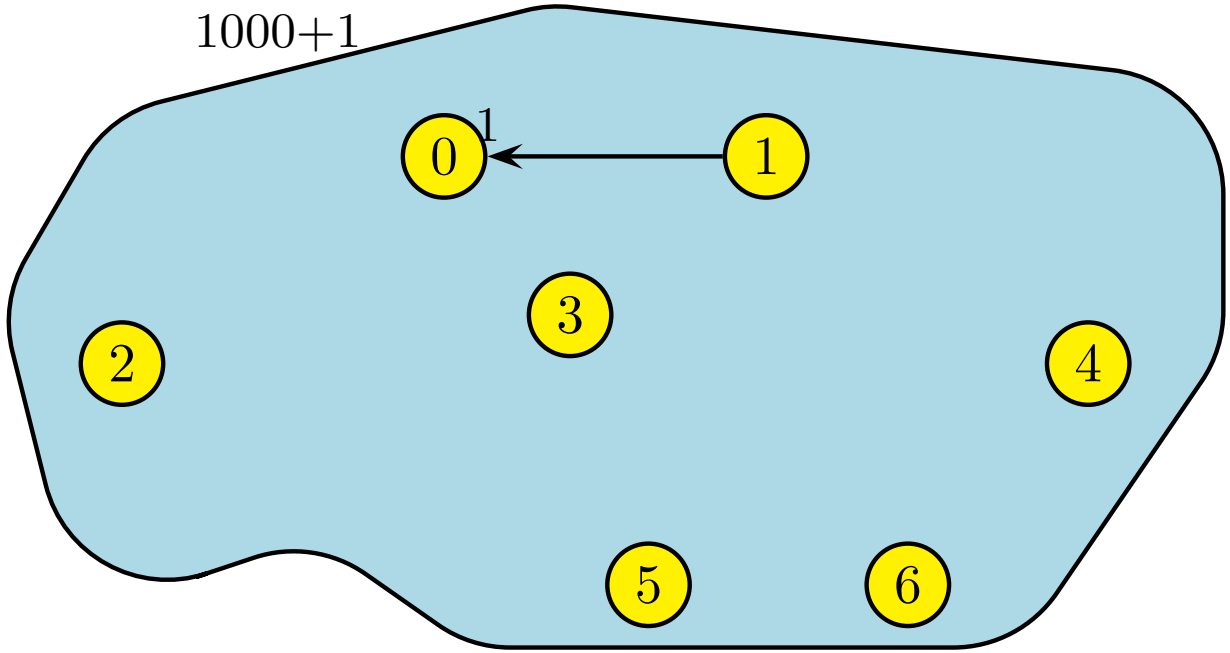


FIGURE 17 – Réduction répartie (somme)

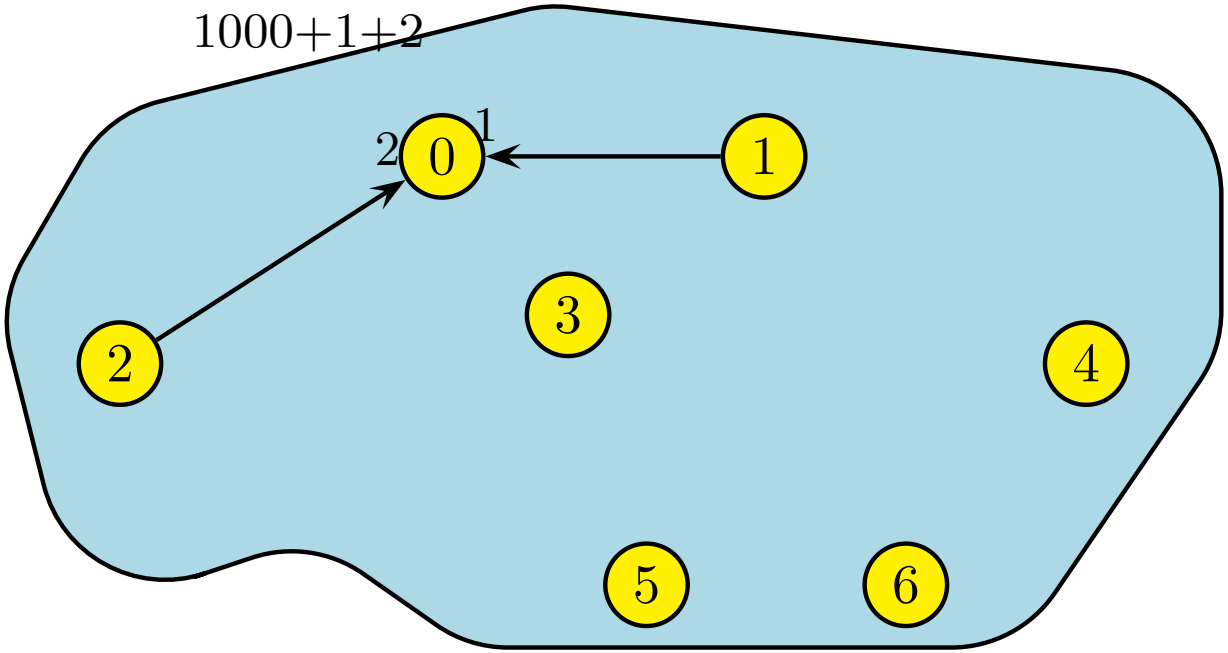


FIGURE 17 – Réduction répartie (somme)

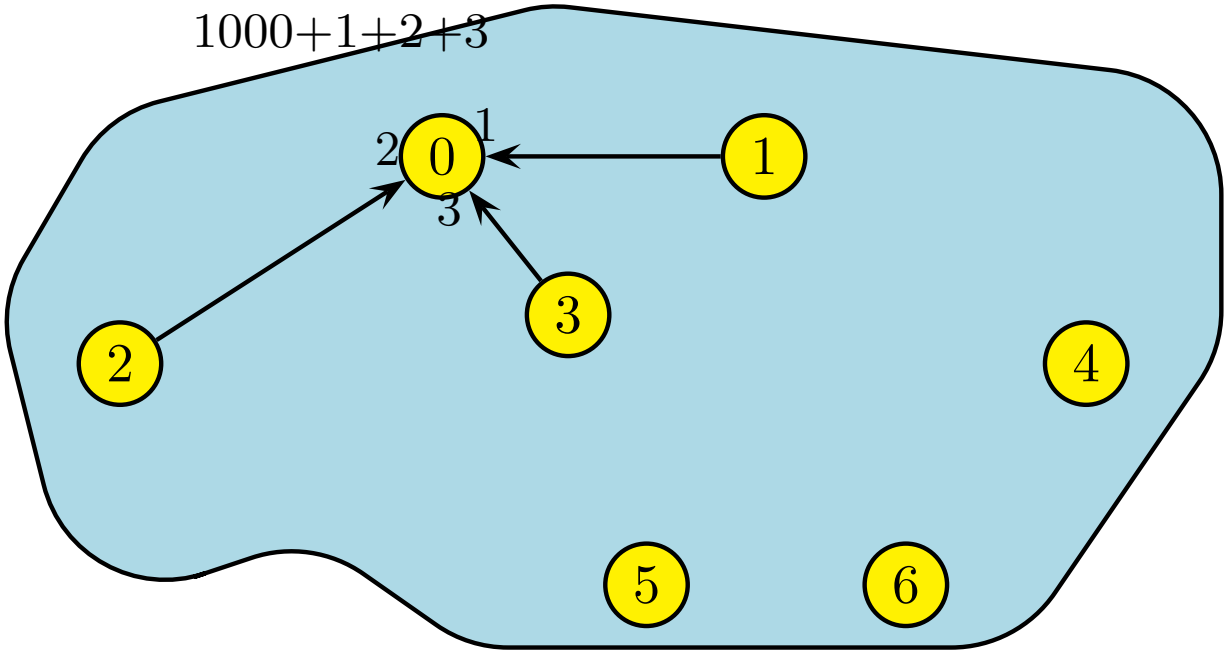


FIGURE 17 – Réduction répartie (somme)

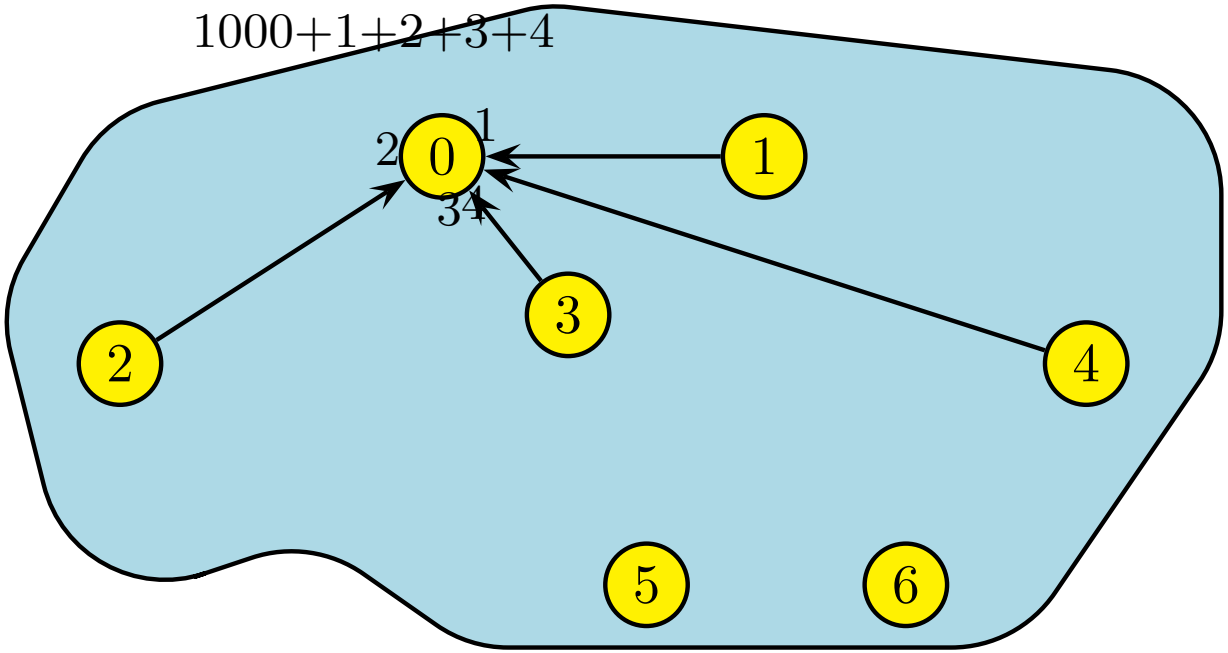


FIGURE 17 – Réduction répartie (somme)

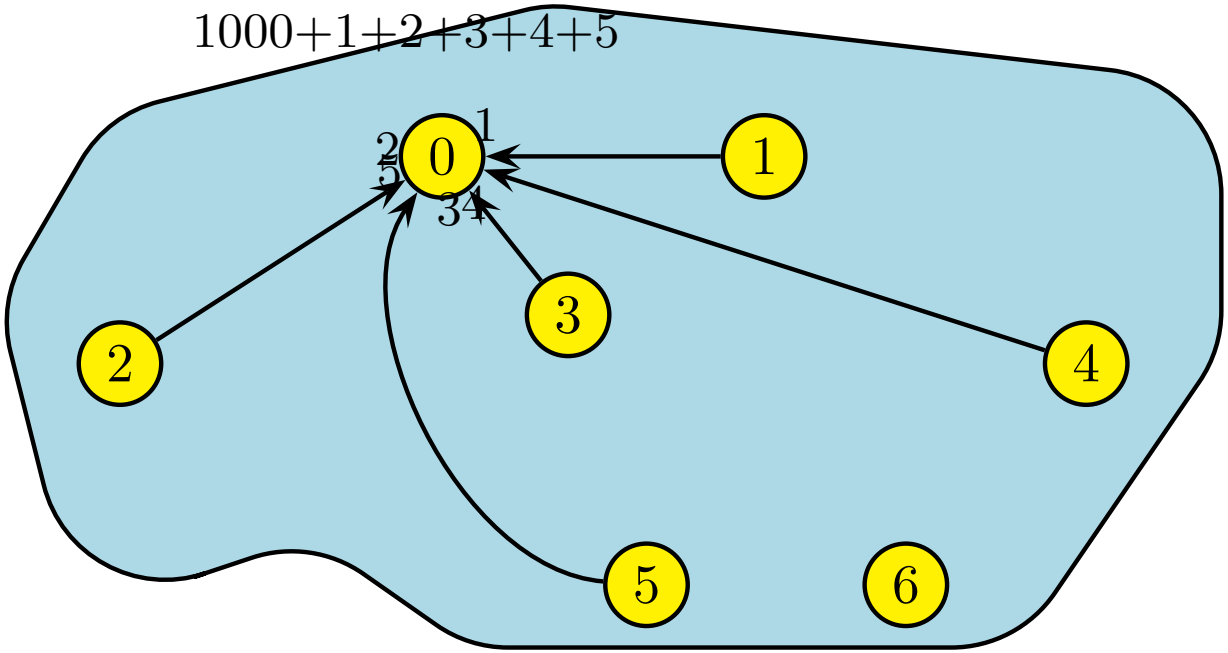


FIGURE 17 – Réduction répartie (somme)

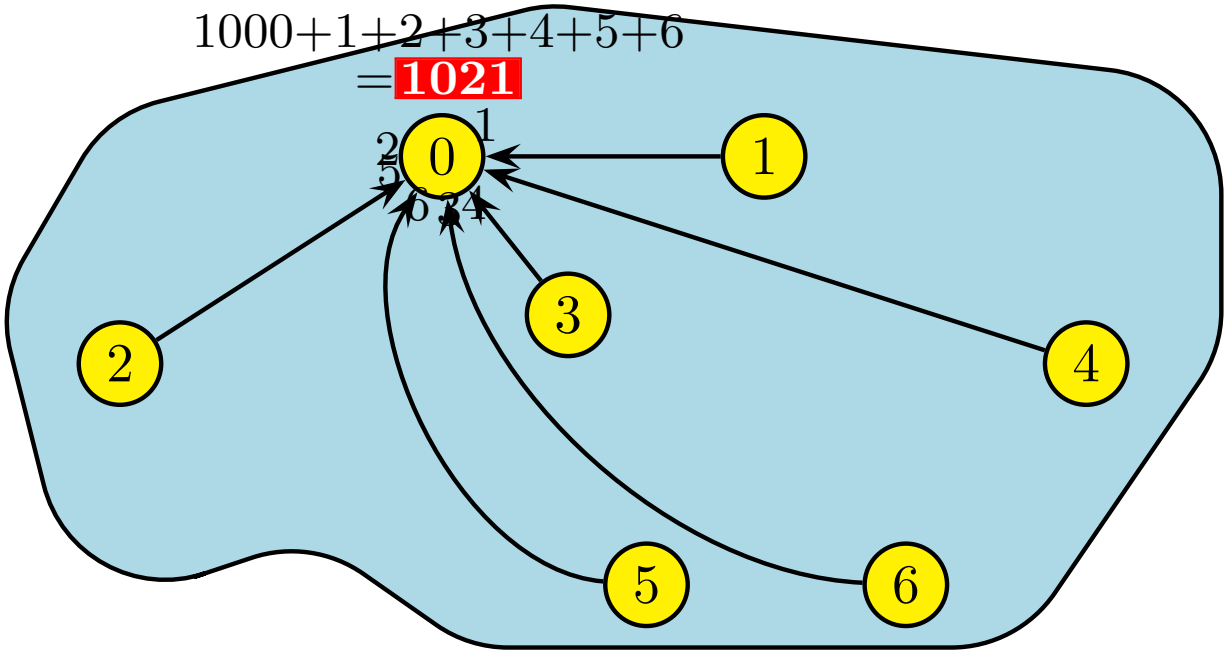


FIGURE 17 – Réduction répartie (somme)

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(int argc, char *argv[])
6  {
7    int rang, valeur, somme;
8
9    MPI_Init(&argc,&argv);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
11
12   if (rang == 0)
13     valeur=1000;
14   else
15     valeur=rang;
16
17   MPI_Reduce(&valeur,&somme,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
18
19   if (rang == 0)
20      printf("Moi, processus 0, j'ai pour valeur de la somme globale %d\n",somme);
21   MPI_Finalize(); return(0);
```

```
> mpirun -np 7 reduce
Moi, processus 0, j'ai pour valeur de la somme globale 1021
```

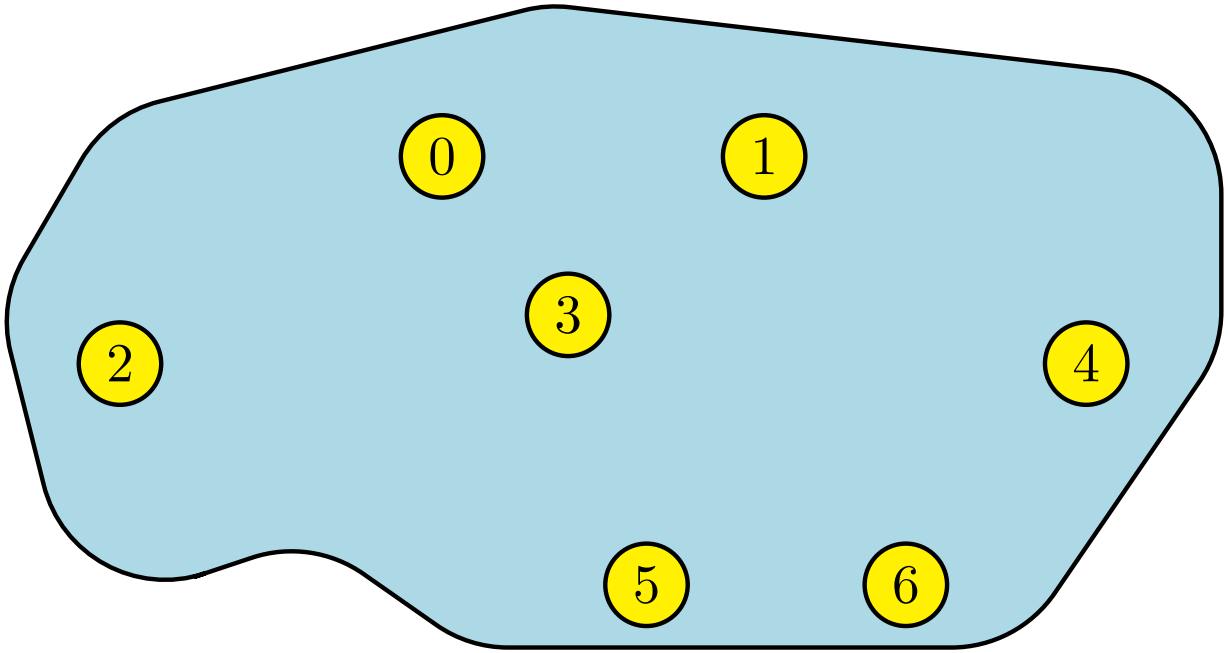


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

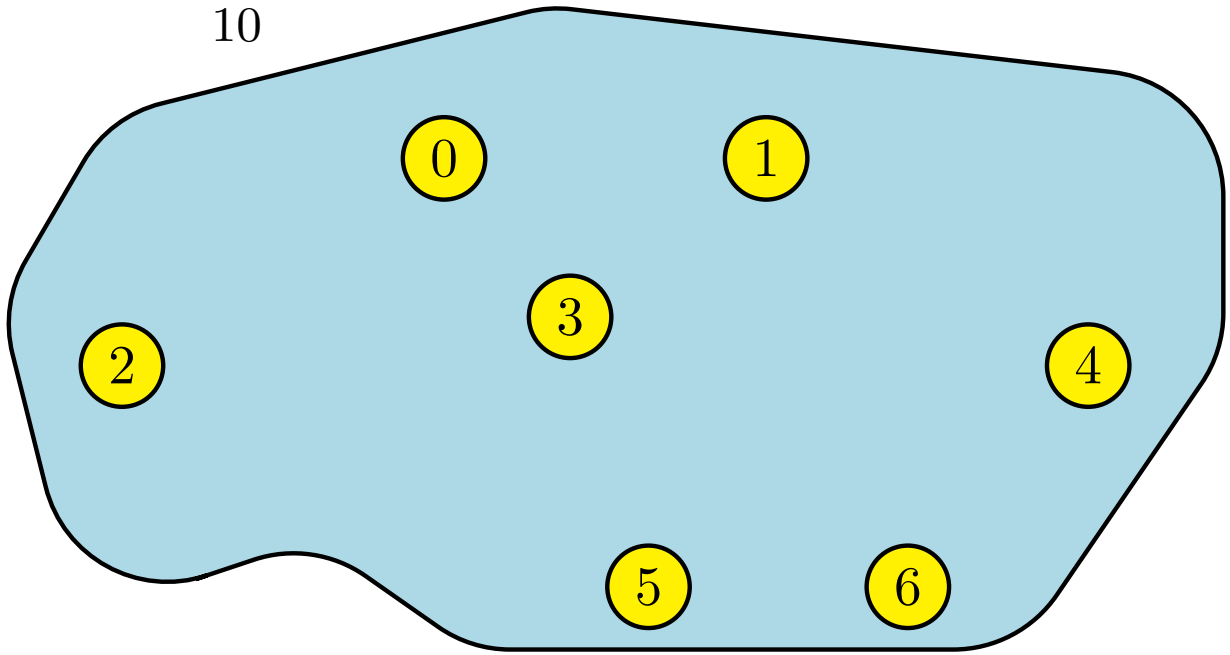


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

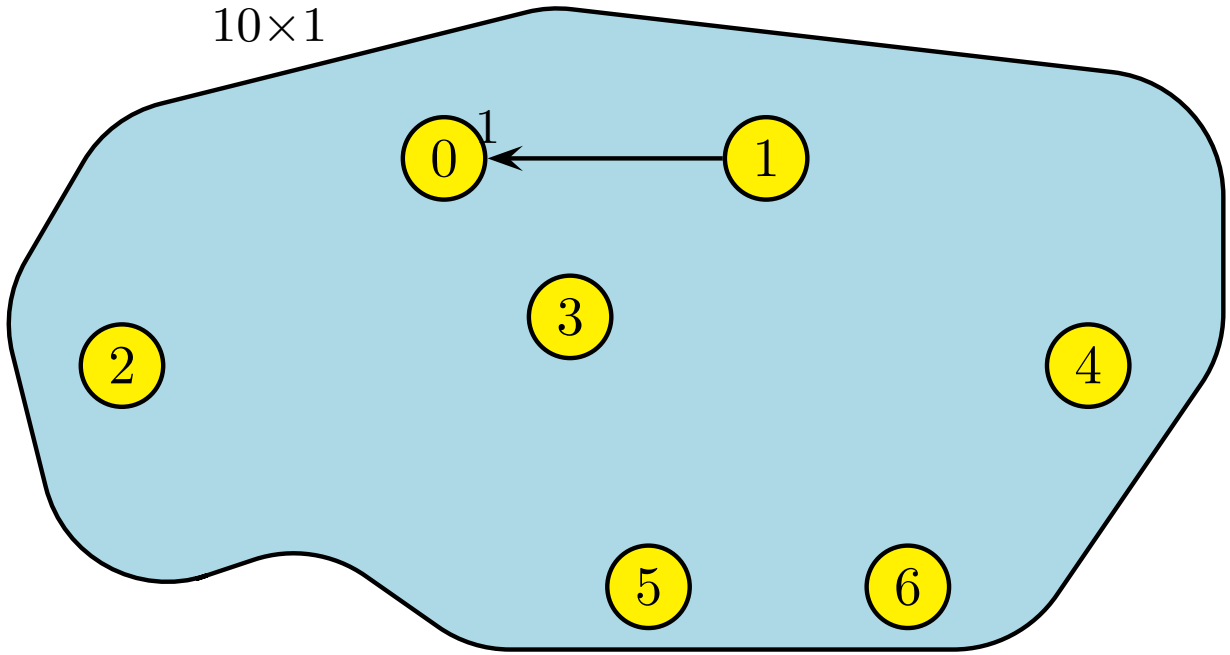


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

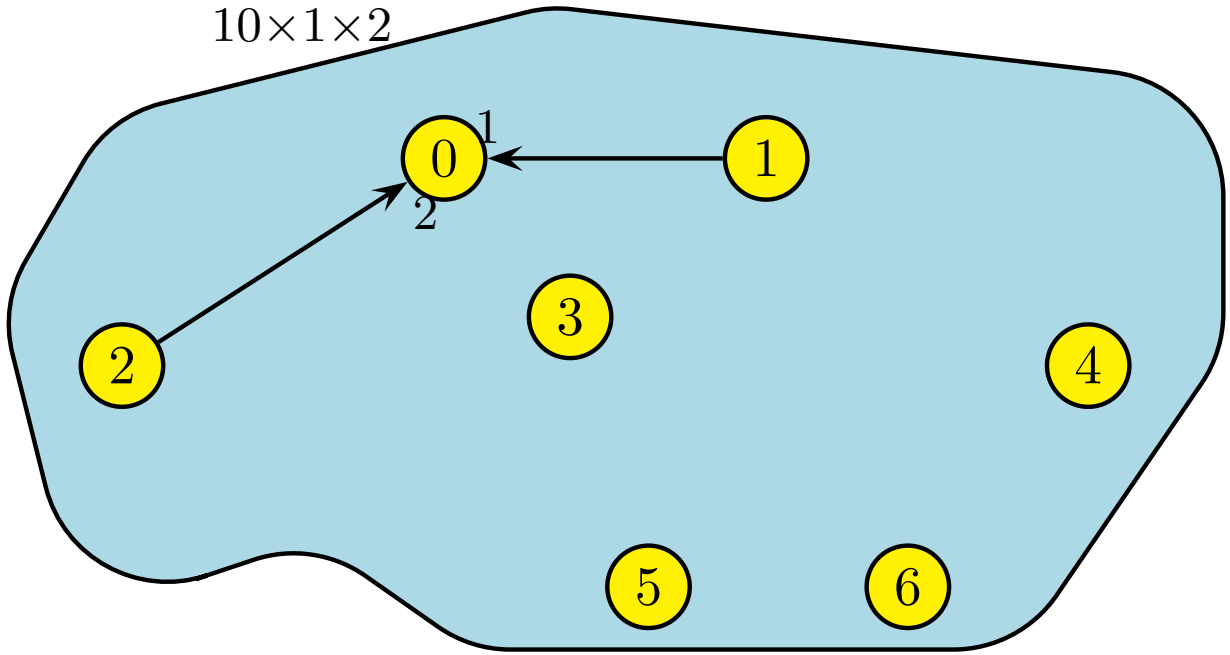


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

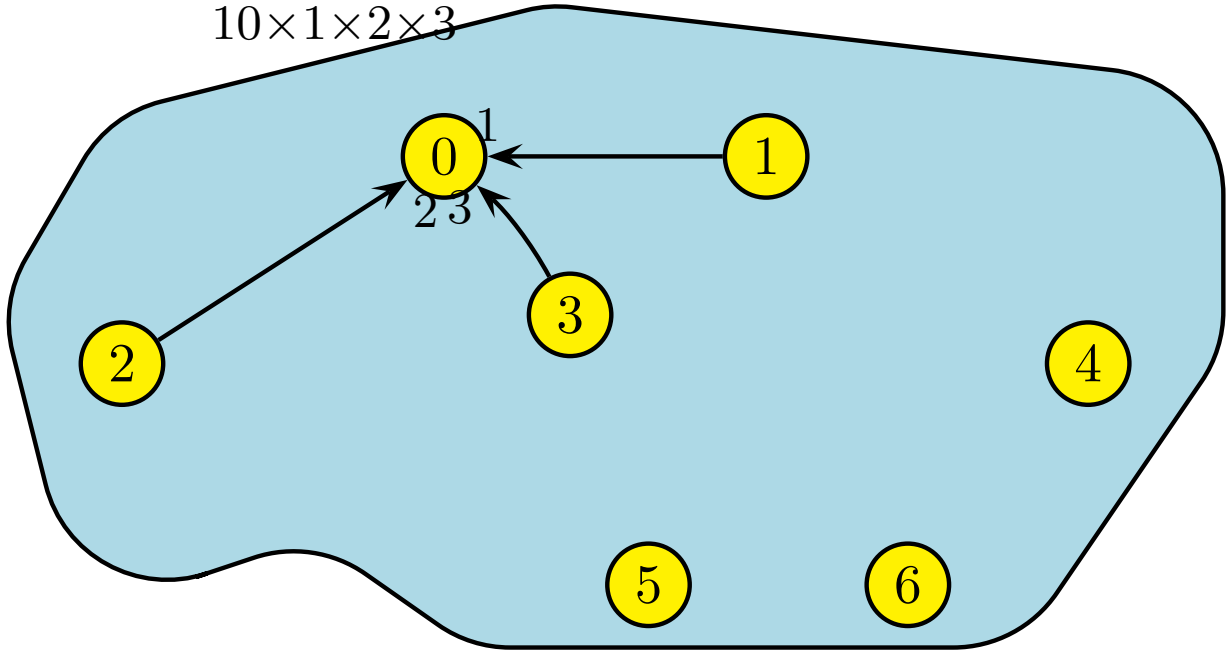


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

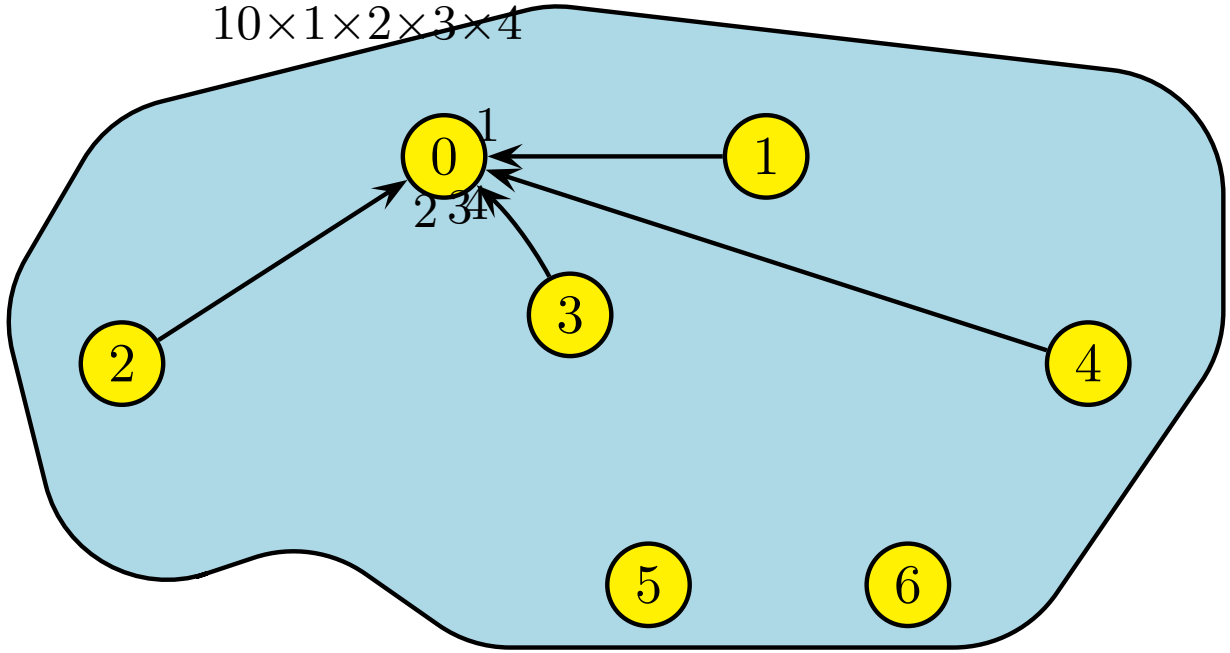


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

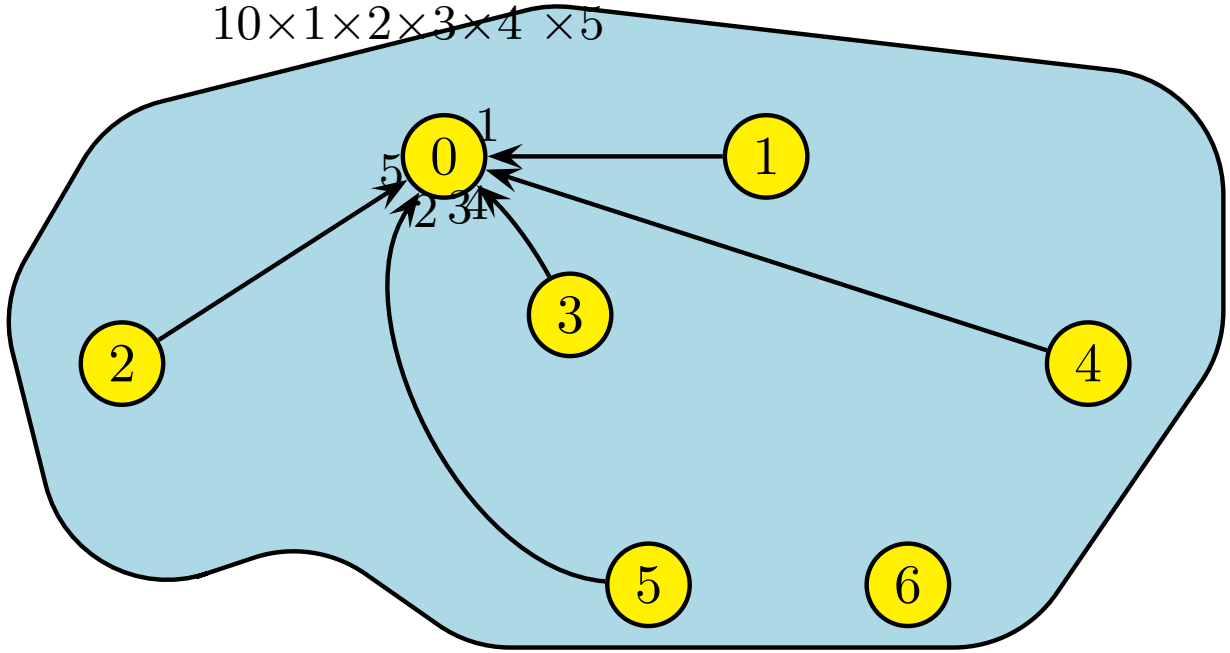


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

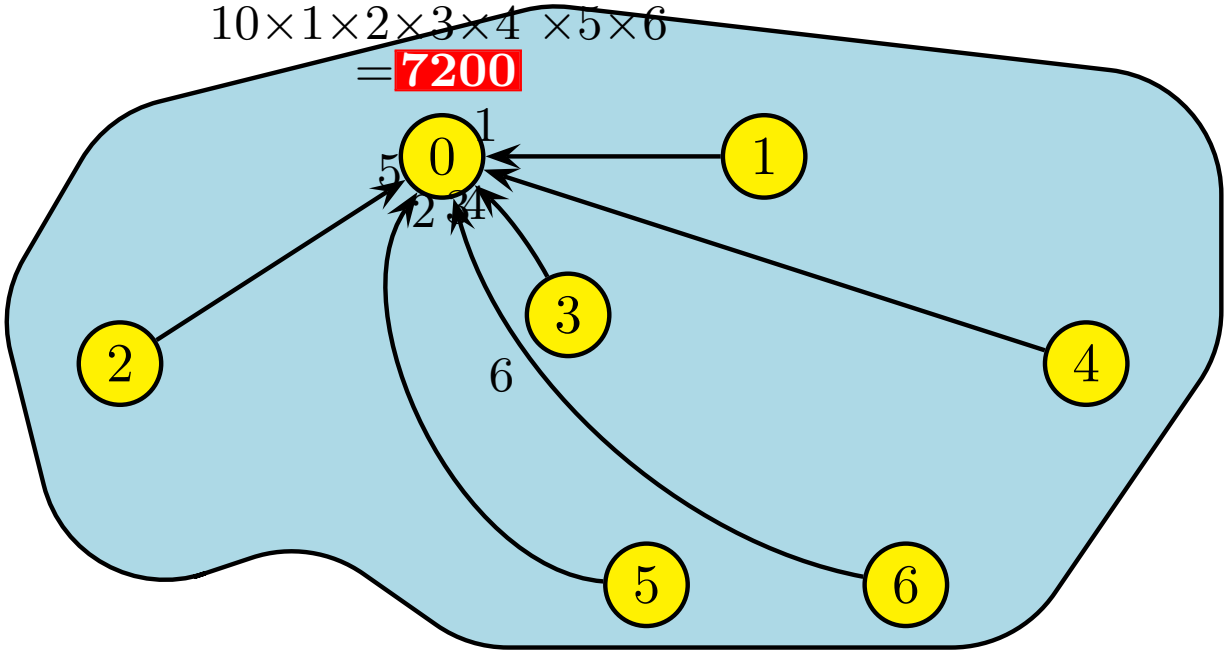


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

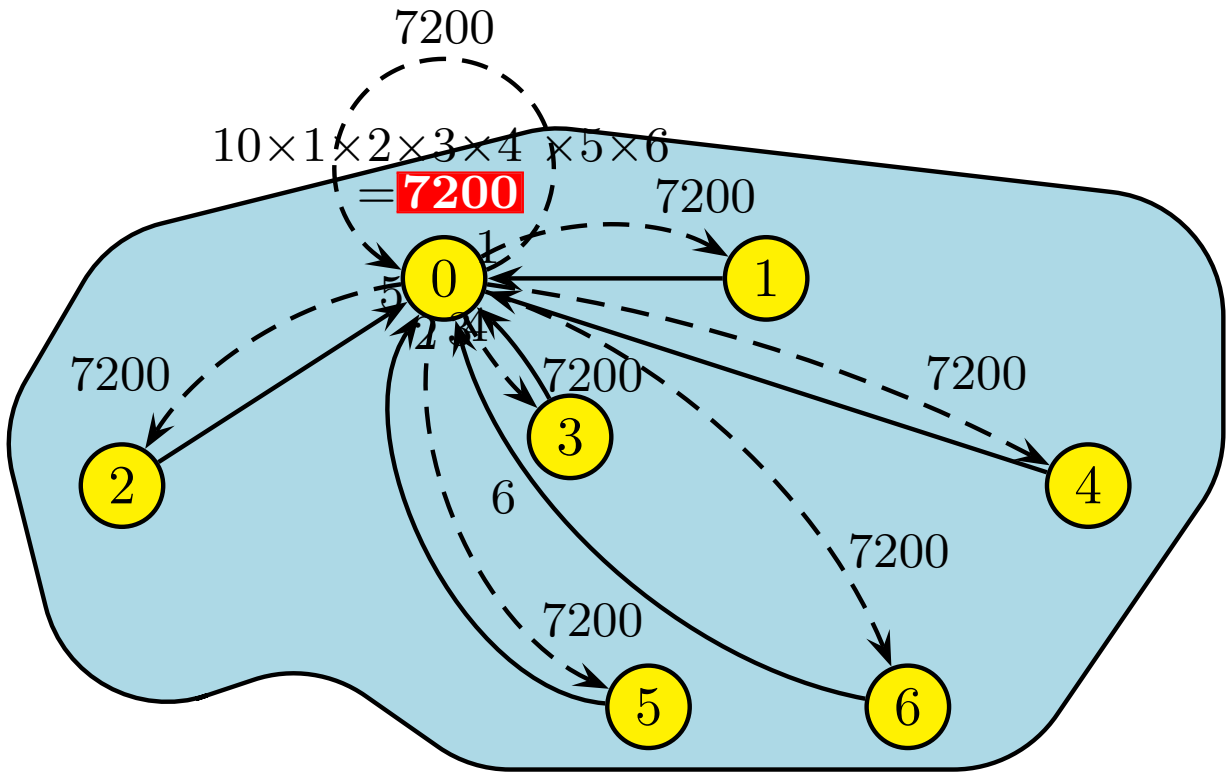


FIGURE 18 – Réduction répartie (produit) avec diffusion du résultat

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(int argc, char *argv[])
6  {
7    int rang, valeur, produit;
8
9    MPI_Init(&argc,&argv);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
11
12   if (rang == 0)
13     valeur=10;
14   else
15     valeur=rang;
16
17   MPI_Allreduce(&valeur,&produit,1,MPI_INT,MPI_PROD,MPI_COMM_WORLD);
18
19   printf("Moi, processus %d, j'ai pour valeur du produit global %d\n",rang,produit);
20   MPI_Finalize(); return(0);
21 }
```

```
> mpirun -np 7 allreduce
Moi, processus 6, j'ai reçu la valeur du produit global 7200
Moi, processus 2, j'ai reçu la valeur du produit global 7200
Moi, processus 0, j'ai reçu la valeur du produit global 7200
Moi, processus 4, j'ai reçu la valeur du produit global 7200
Moi, processus 5, j'ai reçu la valeur du produit global 7200
Moi, processus 3, j'ai reçu la valeur du produit global 7200
Moi, processus 1, j'ai reçu la valeur du produit global 7200
```

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define NB_VALEURS 4
5  typedef struct {double reel,imag;} Complex;
6  void mon_produit(Complex *valeur,Complex *resultat,int *longueur,MPI_Datatype *type);
7  int main(int argc, char *argv[])
8  {
9    int i, rang, commute=0;
10   Complex valeur[NB_VALEURS], resultat[NB_VALEURS];
11   MPI_Op mon_operation;
12   MPI_Datatype type_complexe;
13
14   MPI_Init(&argc,&argv);
15   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
16   MPI_Type_contiguous(2, MPI_DOUBLE, &type_complexe);
17   MPI_Type_commit(&type_complexe);
18   MPI_Op_create(mon_produit, commute, &mon_operation);
19   for (i=0; i<NB_VALEURS; i++) {
20     valeur[i].reel = (double)(rang+i+1); valeur[i].imag = (double)(rang+i+2);}
21   MPI_Reduce(valeur,resultat,NB_VALEURS,type_complexe,mon_operation,0,MPI_COMM_WORLD);
22   if (rang == 0) {
23     printf("Valeur du produit : ");
24     for (i=0; i<NB_VALEURS; i++)
25       printf("(%.0f,%.0f), ",resultat[i].reel,resultat[i].imag);}
26   MPI_Op_free(&mon_operation); MPI_Finalize(); return(0);
27 }
```

```
1  #include "mpi.h"
2  /* Définition du produit terme à terme de deux vecteurs de nombres complexes */
3  void mon_produit(Complex *valeur,Complex *resultat,int *longueur,MPI_Datatype *type);
4  {
5    int i;
6    Complex c;
7
8    for (i=0; i< *longueur; ++i) {
9      c.reel = resultat->reel*valeur->reel - resultat->imag*valeur->imag;
10     c.imag = resultat->reel*valeur->imag + resultat->imag*valeur->reel;
11     *resultat = c;
12     valeur++;
13     resultat++;
14   }
15 }
```

```
> mpirun -np 5 ma_reduction

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

4.9 – Compléments

☞ Les fonctions `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.

4.10 – Exercice 4 : communications collectives et réductions

☞ En tirant à pile ou face sur chacun des processus, boucler jusqu'à ce que tous les processus fassent le même choix ou bien jusqu'à ce qu'on atteigne un nombre, fixé a priori, d'essais

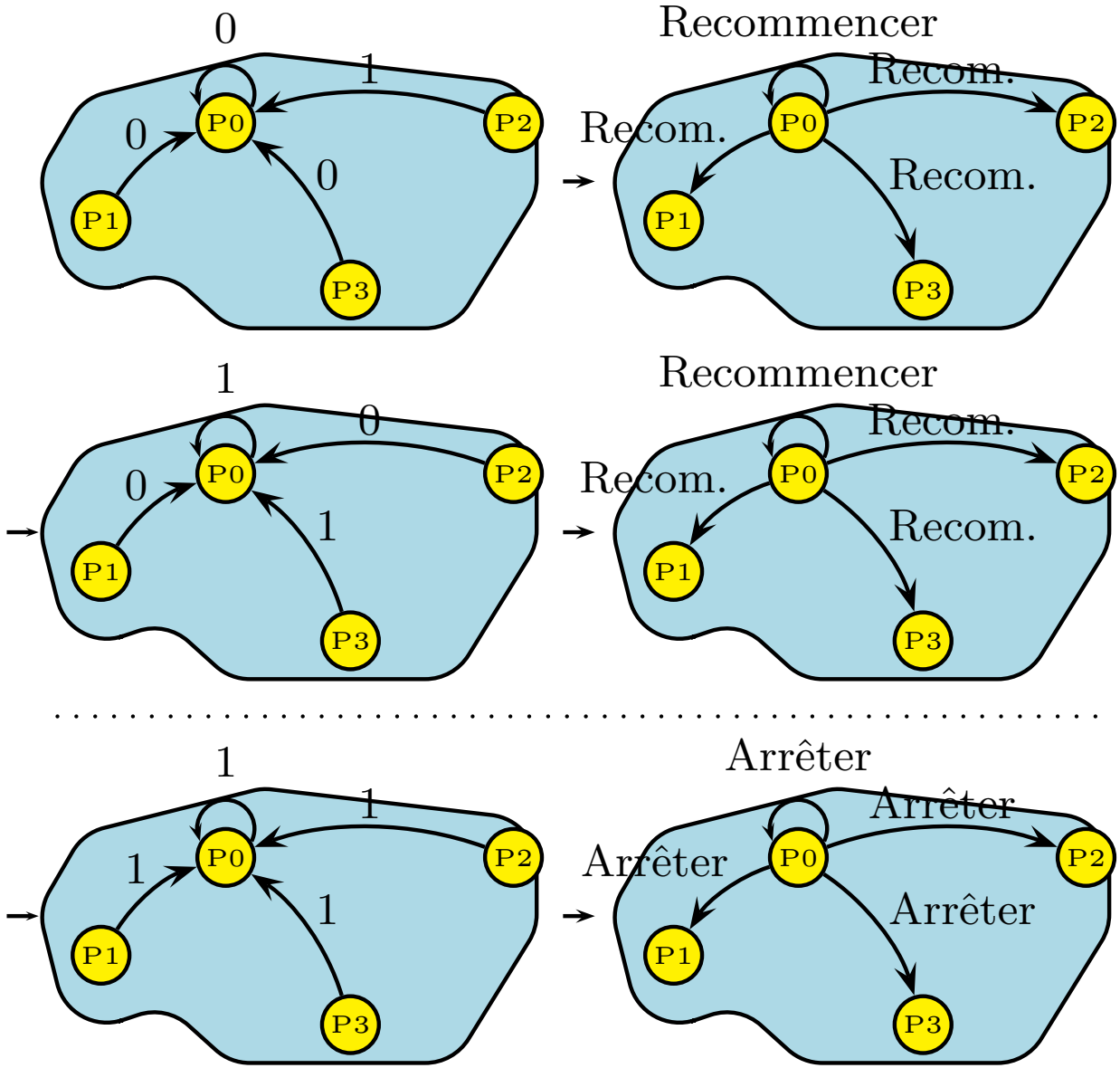


FIGURE 19 – Tirage à pile ou face jusqu'à l'unanimité de tous les processus

4.11 – Exercice 5 : opération de réduction sur une image

Modifiez le programme de l'exercice 3 (chapitre précédent) afin d'introduire les fonctions collectives appropriées pour :

- ☞ répartir l'image sur 4 processus. Chaque partie de l'image sera chargée en mémoire dans un tableau où chaque élément i représente la valeur d'un pixel ;
- ☞ effectuer une opération de réduction pour calculer la valeur globale du RCM (Racine du Carré Moyen) définie par :

$$\mathrm{RCM} = \sqrt{\frac{\sum_{i=0}^{N-1} (\mathrm{pixel}_i)^2}{N}}$$

où N est le nombre total de pixels. Faire imprimer le résultat par le processus 0.

4.12 – Exercice 6 : produit de matrices

☞ Communications collectives et réductions : produit de matrices $C = A \times B$

- ⇨ On se limite au cas de matrices carrées dont l'ordre est un multiple du nombre de processus
- ⇨ Les matrices A et B sont sur le processus 0. Celui-ci distribue une tranche horizontale de la matrice A et une tranche verticale de la matrice B à chacun des processus. Chacun calcule alors un bloc diagonal de la matrice résultante C.
- ⇨ Pour calculer les blocs non diagonaux, chaque processus doit envoyer aux autres processus la tranche de A qu'il possède (voir la figure 20)
- ⇨ Après quoi le processus 0 peut collecter les résultats et vérifier leur validité

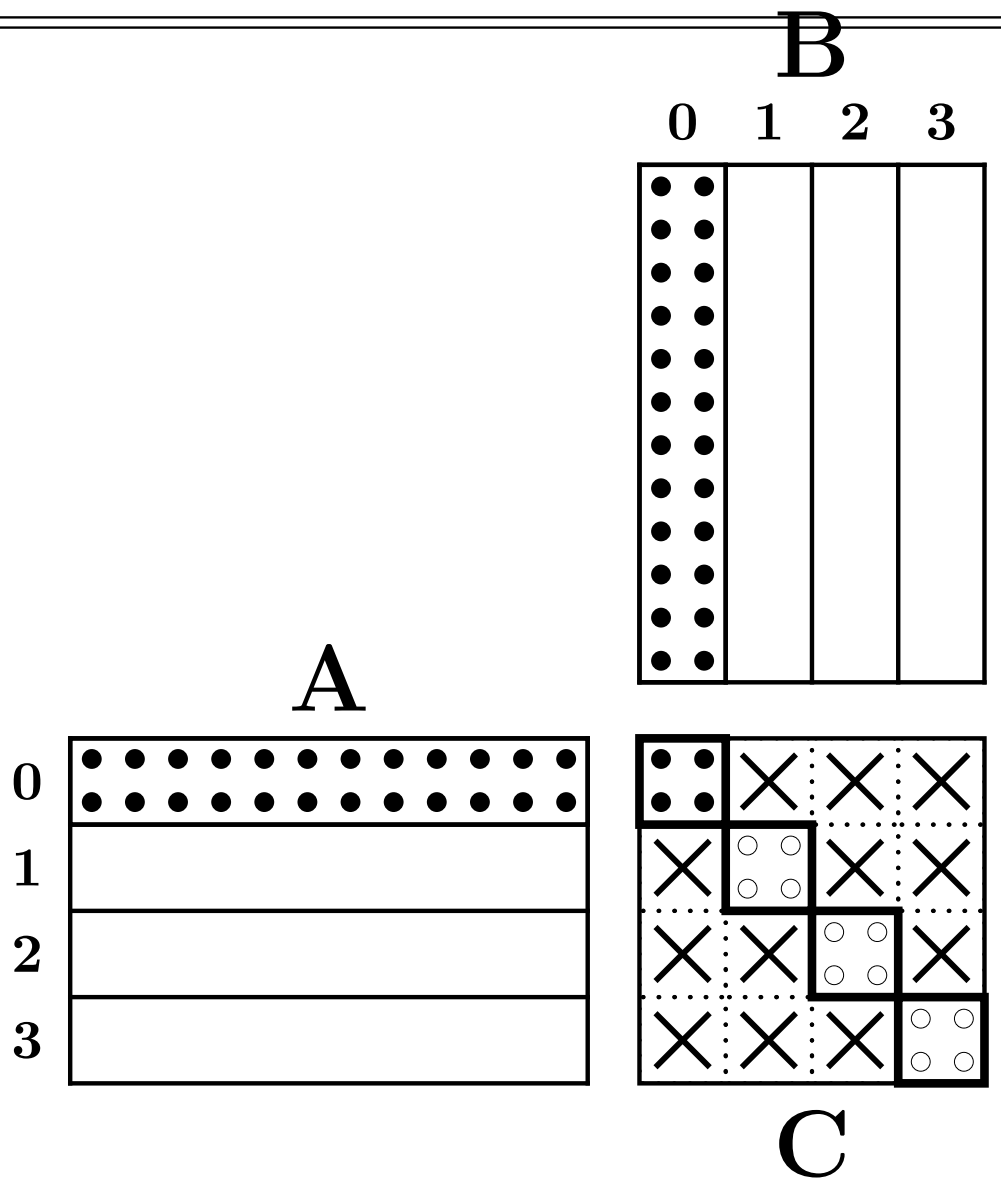


FIGURE 20 – Produit réparti de matrices

5 – Optimisations

5.1 – Introduction

- L'optimisation doit être un souci essentiel lorsque la part de temps des communications par rapport aux calculs devient importante.
- L'optimisation des communications peut s'accomplir à différents niveaux dont les principaux sont :
  1. recouvrir les communications par des calculs (cas d'échanges peu fréquents de gros volumes de données) ;
  2. éviter si possible la recopie du message dans un espace mémoire temporaire (*buffering*) (cas d'échanges d'un volume de données relativement important par rapport à la mémoire disponible) ;
  3. minimiser les surcoûts induits par des appels répétitifs aux fonctions de communication (cas d'échanges fréquents de données).

5.2 – Programme modèle

```
1  #include "mpi.h"
2  ...
3  int main (int argc, char *argv[])
4  {
5   int rang, nb_procs, taille;
6   double temps_debut, temps_fin, temps_fin_max;
7   <type> x, y;
8   MPI_Status statut;
9   ...
10
11  /* Initialisation MPI */
12  MPI_Init(&argc,&argv);
13  MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
14  MPI_Comm_rank(MPI_COMM_WORLD, &rang);
15
16  /* Initialisation divers et variées sans échange de messages */
17  ...
```

```
18  temps_debut = MPI_Wtime();
19  if (rang == 0) {
20    /* Envoi d'un gros message --> cela peut prendre du temps */
21    MPI_Send(x, taille, ..., 1, ..., MPI_COMM_WORLD);
22    /* Traitement séquentiel indépendant de "x" */
23    ...
24    /* Traitement séquentiel impliquant une modification de "x" en mémoire */
25    x = ...
26  }
27  else if (rang == 1) {
28    /* Pré-traitement séquentiel */
29    ...
30    /* Réception du gros message --> cela peut prendre du temps */
31    MPI_Recv(y, taille, ..., 0, ..., MPI_COMM_WORLD, &statut);
32    /* Traitement séquentiel dépendant de "y" */
33    ... = f(y)
34    /* Traitement séquentiel indépendant de "y" */
35    ...
36  };
37  temps_fin = MPI_Wtime() - temps_debut;
38
39  /* Temps de restitution maximum */
40  MPI_Reduce(temps_fin, temps_fin_max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
41  if (rang == 0) printf("Temps : %6.3f secondes\n",temps_fin_max);
42  MPI_Finalize(); exit(0);
43 }
```

Dans ce programme, on suppose que le temps de communication est prépondérant par rapport au temps cumulé relatif au traitement séquentiel.

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.

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`.

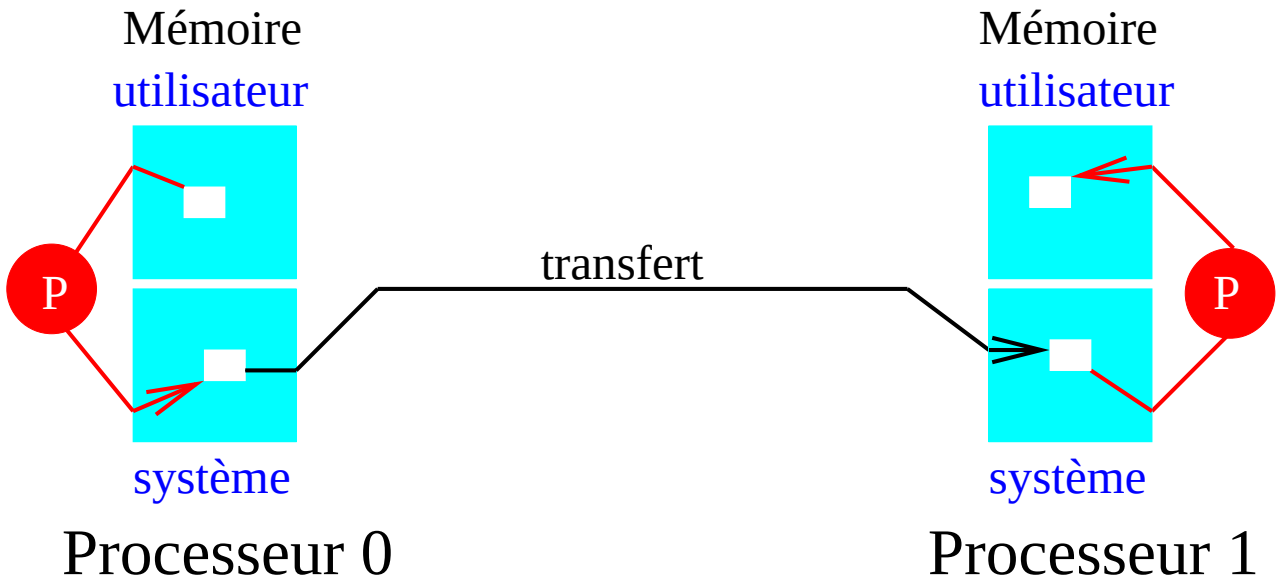


FIGURE 21 – Recopie temporaire d'un message

❷ *Envoi non bloquant avec recopie temporaire, non couplé avec la réception*. L'appel à une fonction 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.

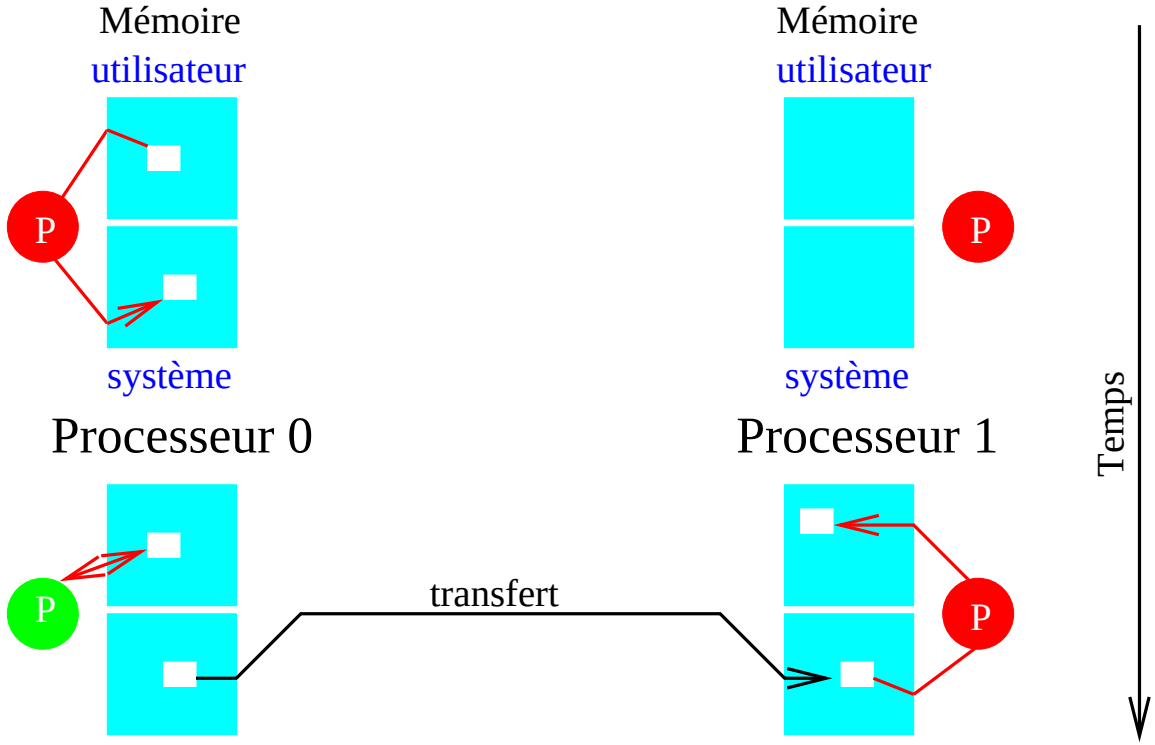


FIGURE 22 – Envoi non bloquant avec recopie temporaire du message

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.

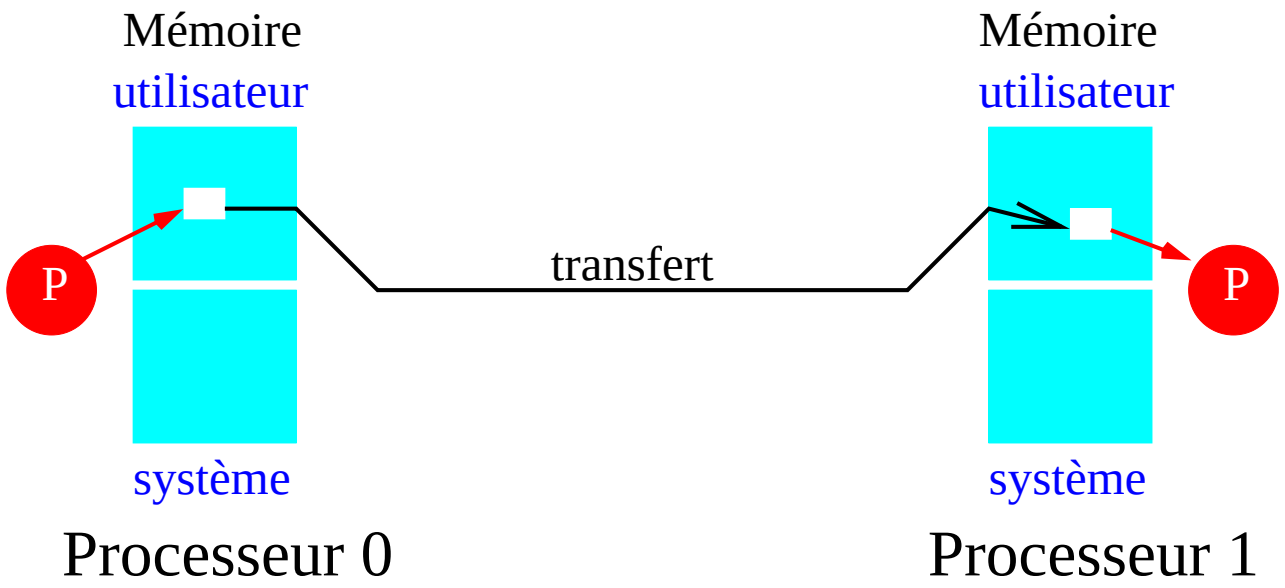


FIGURE 23 – Envoi bloquant couplé avec la réception

❹ *Envoi non bloquant sans recopie temporaire, couplé avec la réception.* L'appel à ce type de fonctions 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.

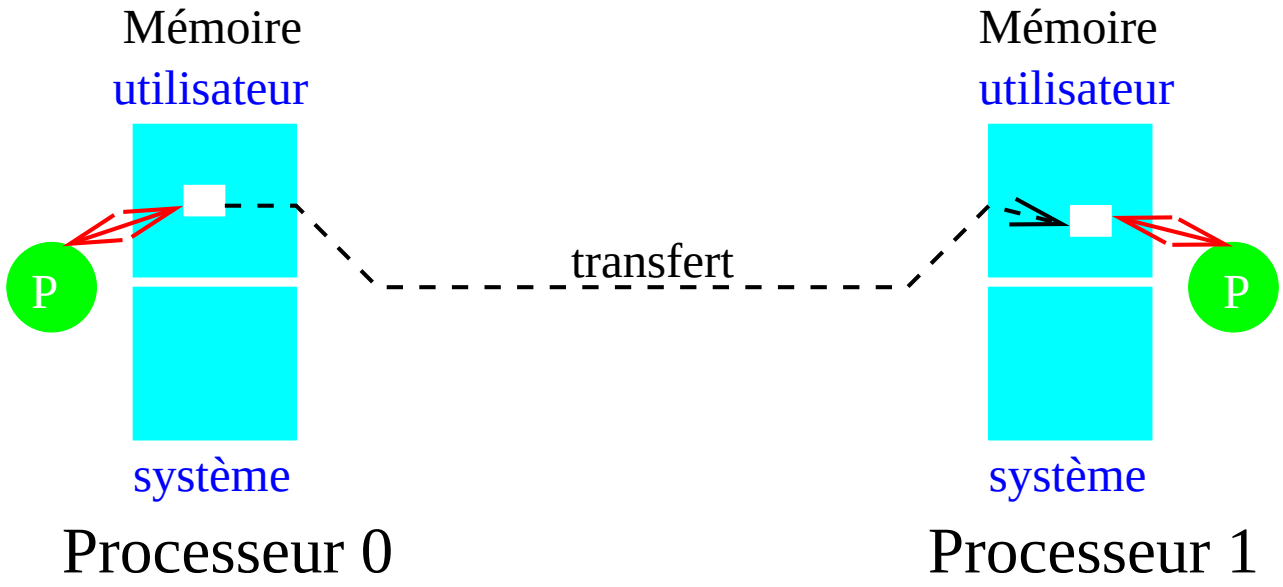


FIGURE 24 – Envoi non bloquant couplé avec la réception

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()`
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.

Dans le programme modèle, il suffit de remplacer `MPI_Send()` par `MPI_Ssend()` pour réduire assez considérablement le temps de restitution.

```
22  temps_debut = MPI_Wtime();
23  if (rang == 0) {
24    /* Envoi d'un gros message --> cela peut prendre du temps */
25    MPI_Ssend(x, taille, ..., 1, ..., MPI_COMM_WORLD);
26    /* Traitement séquentiel indépendant de "x" */
27    ...
28    /* Traitement séquentiel impliquant une modification de "x" en mémoire */
29    x = ...
30  }
31  else if (rang == 1) {
32    /* Pré-traitement séquentiel */
33    ...
34    /* Réception du gros message --> cela peut prendre du temps */
35    MPI_Recv(y, taille, ..., 0, ..., MPI_COMM_WORLD, &statut);
36    /* Traitement séquentiel dépendant de "y" */
37    ... = f(y)
38    /* Traitement séquentiel indépendant de "y" */
39    ...
40  };
41  temps_fin = MPI_Wtime() - temps_debut;
```

5.7 – Envoi synchrone non-bloquant

L'utilisation des fonctions `MPI_Issend()` et `MPI_Irecv()` conjointement à la fonction de synchronisation `MPI_Wait()` permet principalement de recouvrir une communication par un traitement séquentiel des processus.

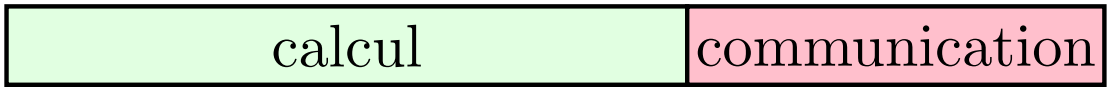
Le programme modèle, une fois modifié, peut voir ses performances sensiblement améliorées ...

```
\fvset{numbers=left}
\begin{ColorVerbatim}[baselinestretch=0.84]
#include "mpi.h"
...
int main (int argc, char *argv[])
{
 int rang, nb_procs, taille;
 double temps_debut, temps_fin, temps_fin_max;
 <type> x, y;
 MPI_Request requete0, requete1;
 MPI_Status statut;
 ...

 /* Initialisation MPI */
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
 MPI_Comm_rank(MPI_COMM_WORLD, &rang);

 /* Initialisation divers et variées sans échange de messages */
 ...
```

```
21  temps_debut = MPI_Wtime();
22  if (rang == 0) {
23    /* Envoi d'un gros message --> cela peut prendre du temps */
24    MPI_Issend(x, taille, ..., 1, ..., MPI_COMM_WORLD, &requete0);
25    /* Traitement séquentiel indépendant de "x" */
26    ...
27    MPI_Wait(requete0,&statut);
28    /* Traitement séquentiel impliquant une modification de "x" en mémoire */
29    x = ...
30  }
31  else if (rang == 1) {
32    /* Pré-traitement séquentiel */
33    ...
34    /* Réception du gros message --> cela peut prendre du temps */
35    MPI_Irecv(y, taille, ..., 0, ..., MPI_COMM_WORLD, &requete1);
36    /* Traitement séquentiel indépendant de "y" */
37    ...
38    MPI_Wait(requete1,&statut);
39    /* Traitement séquentiel dépendant de "y" */
40    ... = f(y)
41  };
42  temps_fin = MPI_Wtime() - temps_debut;
```



En général, dans le cas d'un envoi (`MPI_Ixsend()`) ou d'une réception (`MPI_Irecv()`) non bloquant, il existe toute une palette de fonctions 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 la fonction `MPI_Ssend()`.
- ☞ Recouvrir les communications par des calculs tout en évitant la recopie temporaire des messages en utilisant les fonctions non bloquantes `MPI_Issend()` et `MPI_Irecv()`.

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 une fonction 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 toutes (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  temps_debut = MPI_Wtime();
24  if (rang == 0) {
25    for(k=0; k<1000; k++) {
26      /* Envoi d'un gros message --> cela peut prendre du temps */
27      MPI_Issend(x, taille, ..., 1, ..., MPI_COMM_WORLD, &requete0);
28      /* Traitement séquentiel indépendant de "x" */
29      ...
30      MPI_Wait(requete0,&statut);
31      /* Traitement séquentiel impliquant une modification de "x" en mémoire */
32      x = ...
33    }
34  }
35  else if (rang == 1) {
36    for(k=0; k<1000; k++) {
37      /* Pré-traitement séquentiel */
38      ...
39      /* Réception du gros message --> cela peut prendre du temps */
40      MPI_Irecv(y, taille, ..., 0, ..., MPI_COMM_WORLD, &requete1);
41      /* Traitement séquentiel indépendant de "y" */
42      ...
43      MPI_Wait(requete1,&statut);
44      /* Traitement séquentiel dépendant de "y" */
45      ... = f(y)
46    }
47  };
48  temps_fin = MPI_Wtime() - temps_debut;
```

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

```
23  if (rang == 0) {
24    MPI_Ssend_init(x, taille, ..., 1, ..., MPI_COMM_WORLD, &requete0);
25    for(k=0; k<1000; k++) {
26      /* Envoi d'un gros message --> cela peut prendre du temps */
27      MPI_Start(&requete0);
28      /* Traitement séquentiel indépendant de "x" */
29      ...
30      MPI_Wait(requete0,&statut);
31      /* Traitement séquentiel impliquant une modification de "x" en mémoire */
32      x = ...
33    } ; MPI_Request_free(&requete0);
34  }
35  else if (rang == 1) {
36    MPI_Recv_init(y, taille, ..., 0, ..., MPI_COMM_WORLD, &requete1);
37    for(k=0; k<1000; k++) {
38      /* Pré-traitement séquentiel */
39      ...
40      /* Réception du gros message --> cela peut prendre du temps */
41      MPI_Start(&requete1);
42      /* Traitement séquentiel indépendant de "y" */
43      ...
44      MPI_Wait(requete1,&statut);
45      /* Traitement séquentiel dépendant de "y" */
46      ... = f(y)
47    } ; MPI_Request_free(&requete1);
48  };
```

Remarques

- Une communication activée par `MPI_Start()` sur une requête créée par l'une des fonctions `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 la fonction `MPI_Request_free`(`requete`,code).
- Cette fonction 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 fonctions de communication en utilisant une fois pour toutes un schéma persistant de communication et activer celui-ci autant de fois qu'il est nécessaire à l'aide de la fonction `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 de la fonction `MPI_Start()`.

6 – Types de données dérivés

6.1 – Introduction

Dans les communications, les données échangées sont typées : `MPI_INT`, `MPI_FLOAT`, `MPI_CHAR`, etc.

On peut créer des structures de données plus complexes à l'aide de fonctions telles que `MPI_Type_contiguous()`, `MPI_Type_vector()`, `MPI_Type_create_hvector()`.

À chaque fois que l'on crée un type de données, il faut le valider à l'aide de la fonction `MPI_Type_commit()`.

Si on souhaite réutiliser le même type, on doit le libérer avec la fonction `MPI_Type_free()`.

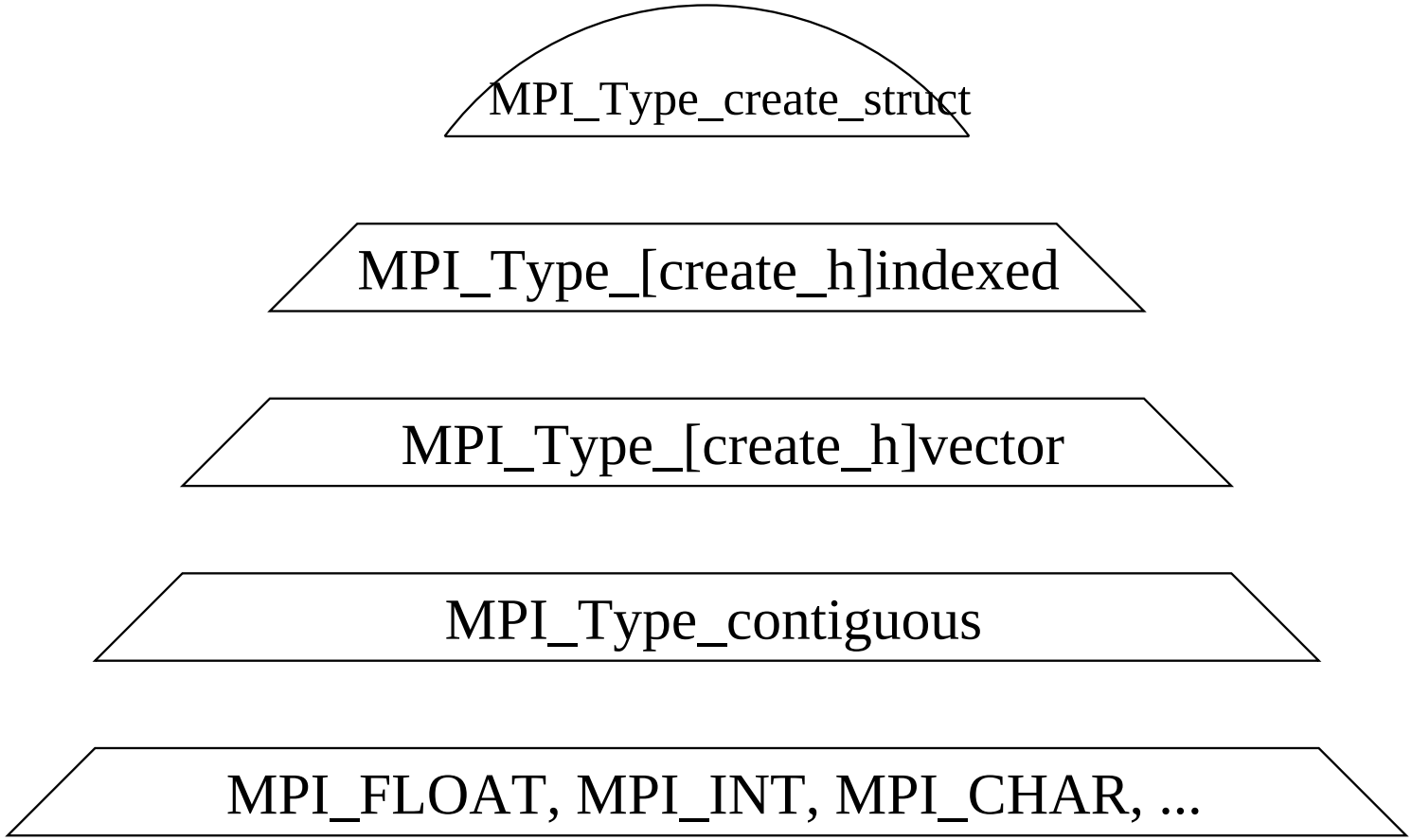


FIGURE 25 – Hiérarchie des constructeurs de type MPI

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

```
#include "mpi.h"
int code;
MPI_Datatype nouveau_type;
code=MPI_Type_contiguous(6,MPI_FLOAT,&nouveau_type);
```

FIGURE 26 – Fonction MPI_Type_contiguous()

```
int nombre, code;
MPI_Datatype ancien_type, nouveau_type;
code=MPI_Type_contiguous(nombre,ancien_type,&nouveau_type)
```

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

```
#include "mpi.h"
int code;
MPI_Datatype nouveau_type;
code=MPI_Type_vector(5,1,6,MPI_FLOAT,&nouveau_type);
```

FIGURE 27 – Fonction MPI_Type_vector()

```
int code, nombre_bloc, longueur_bloc;
int pas; /* donné en nombre d'éléments */
MPI_Datatype ancien_type, nouveau_type;
code=MPI_Type_vector(nombre_bloc,longueur_bloc,pas,ancien_type,&nouveau_type);
```

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

6.4 – Descriptif des fonctions

```
int code, nombre_bloc, longueur_bloc;
int pas; /* donné en octets */
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_create_hvector(nombre_bloc,longueur_bloc,pas,ancien_type,&nouveau_type);
```

```
int code;
MPI_Datatype nouveau_type;

call MPI_Type_commit(&nouveau_type);
```

```
MPI_Datatype nouveau_type;
int code;

code=MPI_Type_free(&nouveau_type);
```

6.5 – Exemples

```
1  #include "mpi.h"
2  #define NB_LIGNES 5
3  #define NB_COLONNES 6
4  #define ETIQUETTE 100
5  int main(int argc, char *argv[])
6  {
7    float a[NB_LIGNES][NB_COLONNES];
8    int i, j, rang;
9    MPI_Statut statut;
10   MPI_Datatype type_ligne;
11
12   MPI_Init(&argc,&argv);
13   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
14   /* Initialisation de la matrice sur chaque processus */
15   for(i=0; i<NB_LIGNES; i++)
16     for (j=0; j<NB_COLONNES; j++)
17       a[i][j] = (float)(rang+1);
18
19   /* Définition du type type_ligne */
20   MPI_Type_contiguous(NB_COLONNES, MPI_FLOAT, &type_ligne);
21
22   /* Validation du type type_ligne
23   MPI_Type_commit(&type_ligne);
```

```
24    if ( rang == 0 )
25      /* Envoi de la première ligne */
26      MPI_Send(&a[0][0], 1, type_ligne, 1, ETIQUETTE, MPI_COMM_WORLD);
27    else if ( rang == 1 )
28        /* Réception dans la dernière ligne */
29        MPI_Recv(&a[NB_LIGNES-1][0], 1, type_ligne, 0, ETIQUETTE,
30                 MPI_COMM_WORLD, &statut);
31
32    /* Libération du type */
33    MPI_Type_free(&type_ligne);
34
35    MPI_Finalize();
36    exit(0);
37  }
```

Le type « colonne d'une matrice »

```
1  #include "mpi.h"
2  #define NB_LIGNES 5
3  #define NB_COLONNES 6
4  #define ETIQUETTE 100
5  int main(int argc, char *argv[])
6  {
7    float a[NB_LIGNES][NB_COLONNES];
8    int i, j, rang;
9    MPI_Status statut;
10   MPI_Datatype type_colonne;
11
12   MPI_Init(&argc,&argv);
13   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
14
15   /* Initialisation de la matrice sur chaque processus */
16   for(i=0; i<NB_LIGNES; i++)
17     for (j=0; j<NB_COLONNES; j++)
18       a[i][j] = (float)(rang+1);
19   /* Définition du type type_colonne */
20   MPI_Type_vector(NB_LIGNES, 1, NB_COLONNES, MPI_FLOAT, &type_colonne);
21
22   /* Validation du type type_ligne */
23   MPI_Type_commit(&type_colonne);
```

```
24    if ( rang == 0 )
25      /* Envoi de la deuxième colonne */
26      MPI_Send(&a[0][1], 1, type_colonne, 1, ETIQUETTE, MPI_COMM_WORLD);
27
28    else if ( rang == 1 )
29        /* Réception dans lavant-dernière colonne */
30        MPI_Recv(&a[0][NB_COLONNES-2], 1, type_colonne, 0, ETIQUETTE,
31                 MPI_COMM_WORLD, &statut);
32
33    /* Libération du type */
34    MPI_Type_free(&type_colonne);
35
36    MPI_Finalize();
37    exit(0);
38  }
```

Le type « bloc d'une matrice »

```
1  #include "mpi.h"
2  #define NB_LIGNES 5
3  #define NB_COLONNES 6
4  #define ETIQUETTE 100
5  int main(int argc, char *argv[])
6  {
7    float a[NB_LIGNES][NB_COLONNES];
8    int i, j, rang, nb_lignes_bloc=2, nb_colonnes_bloc=3;
9    MPI_Status statut;
10   MPI_Datatype type_bloc;
11
12   MPI_Init(&argc,&argv);
13   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
14   /* Initialisation de la matrice sur chaque processus */
15   for(i=0; i<NB_LIGNES; i++)
16     for (j=0; j<NB_COLONNES; j++)
17       a[i][j] = (float)(rang+1);
18
19   /* Définition du type type_bloc */
20   MPI_Type_vector(nb_lignes_bloc,nb_colonnes_bloc,NB_COLONNES,MPI_FLOAT,&type_bloc);
21
22   /* Validation du type type_bloc */
23   MPI_Type_commit(&type_bloc);
```

```
24    if ( rang == 0 )
25      /* Envoi d'un bloc */
26      MPI_Send(&a[0][0],1,type_bloc,1,ETIQUETTE,MPI_COMM_WORLD);
27    else if ( rang == 1 )
28        /* Réception du bloc */
29        MPI_Recv(&a[NB_LIGNES-2][NB_COLONNES-3],1,type_bloc,0,ETIQUETTE,
30                 MPI_COMM_WORLD,&statut);
31
32    /* Libération du type */
33    MPI_Type_free(&type_bloc);
34
35    MPI_Finalize();
36    exit(0);
37  }
```

6.6 – Types homogènes à pas variable

- `MPI_Type_indexed()` permet de créer une structure de données composée d'une séquence de blocs contenant un nombre variable d'éléments et séparés par un pas variable en mémoire. Ce dernier est exprimé en **éléments**.
- `MPI_Type_create_hindexed()` a la même fonctionnalité que `MPI_Type_indexed()` sauf que le pas séparant deux blocs de données est exprimé en **octets**.

  Cette instruction est utile lorsque le type générique n'est pas un type de base *MPI* (`MPI_INT`, `MPI_FLOAT`, ...) mais un type plus complexe construit avec les fonctions *MPI* vues précédemment. On ne peut exprimer alors le pas en nombre d'éléments du type générique d'où le recours à `MPI_Type_create_hindexed()`.
- Attention à la **portabilité** avec `MPI_Type_create_hindexed()` !

nb=3, longueurs_blocs=( ), déplacements=( )

ancien_type 

nouveau_type

FIGURE 28 – Le constructeur MPI_Type_indexed

```
int code, nb;
/* Attention les déplacements sont donnés en éléments */
int *longueurs_blocs, *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_indexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

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

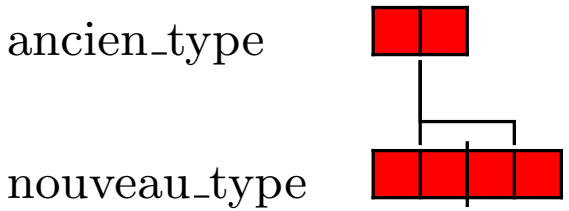


FIGURE 28 – Le constructeur MPI_Type_indexed

```
int code, nb;
/* Attention les déplacements sont donnés en éléments */
int *longueurs_blocs, *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_indexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

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

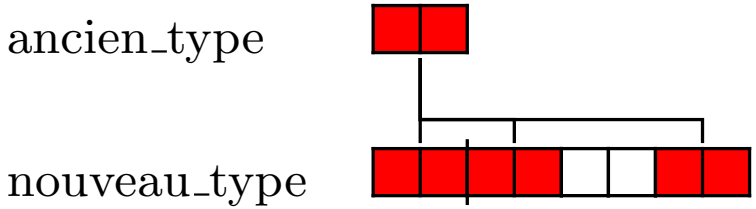


FIGURE 28 – Le constructeur MPI_Type_indexed

```
int code, nb;
/* Attention les déplacements sont donnés en éléments */
int *longueurs_blocs, *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_indexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

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

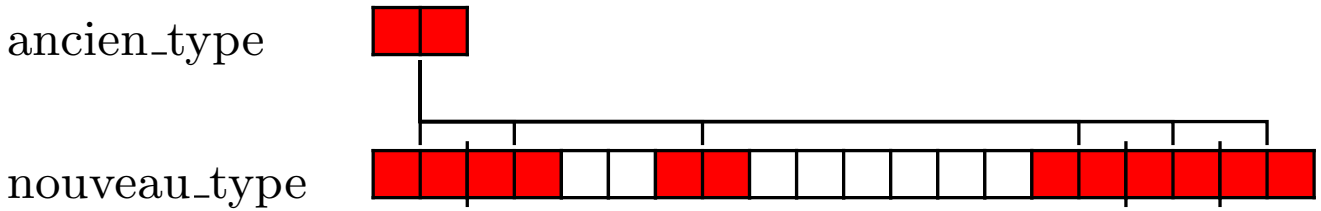


FIGURE 28 – Le constructeur MPI_Type_indexed

```
int code, nb;
/* Attention les déplacements sont donnés en éléments */
int *longueurs_blocs, *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_indexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

nb=4, longueurs_blocs=( ), déplacements=( )

ancien_type 

nouveau_type

FIGURE 29 – Le constructeur MPI_Type_create_hindexed

```
int code, nb;
int *longueurs_blocs;
/* Attention les déplacements sont donnés en octets */
MPI_Aint *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_create_hindexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

nb=4, longueurs_blocs=(2,), déplacements=(2,)

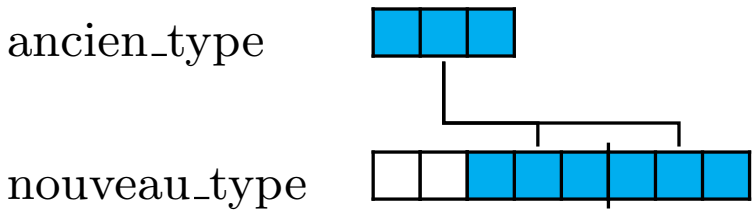


FIGURE 29 – Le constructeur MPI_Type_create_hindexed

```
int code, nb;
int *longueurs_blocs;
/* Attention les déplacements sont donnés en octets */
MPI_Aint *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_create_hindexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

nb=4, longueurs_blocs=(2,1,), déplacements=(2,10,)

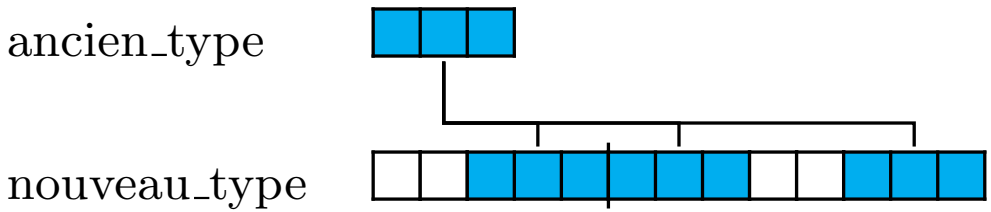


FIGURE 29 – Le constructeur MPI_Type_create_hindexed

```
int code, nb;
int *longueurs_blocs;
/* Attention les déplacements sont donnés en octets */
MPI_Aint *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_create_hindexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

nb=4, longueurs_blocs=(2,1,2,), déplacements=(2,10,14,)

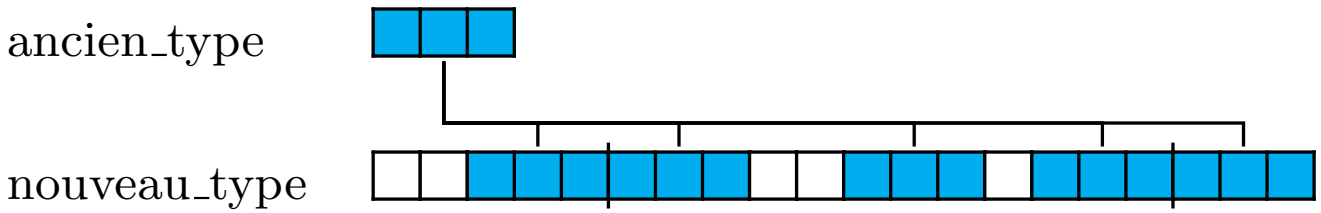


FIGURE 29 – Le constructeur MPI_Type_create_hindexed

```
int code, nb;
int *longueurs_blocs;
/* Attention les déplacements sont donnés en octets */
MPI_Aint *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_create_hindexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

nb=4, longueurs_blocs=(2,1,2,1), déplacements=(2,10,14,24)

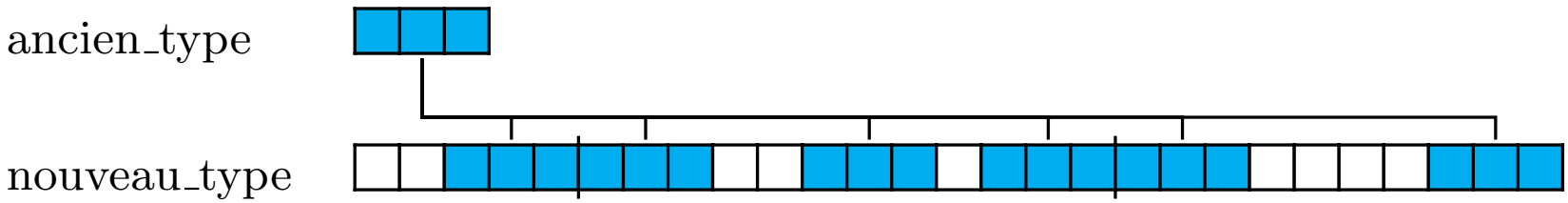


FIGURE 29 – Le constructeur MPI_Type_create_hindexed

```
int code, nb;
int *longueurs_blocs;
/* Attention les déplacements sont donnés en octets */
MPI_Aint *deplacements;
MPI_Datatype ancien_type, nouveau_type;

code=MPI_Type_create_hindexed(nb,longueurs_blocs,deplacements,ancien_type,&nouveau_type);
```

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)

0	1	2	3	4	5	6	7
8	**9**	10	11	12	13	14	15
16	17	**18**	19	20	21	22	23
24	25	26	**27**	28	29	30	31
32	33	34	35	**36**	37	38	39
40	41	42	43	44	**45**	46	47
48	49	50	51	52	53	**54**	55
56	57	58	59	60	61	62	**63**

Processus 1 (avant)

0	-1	-2	-3	-4	-5	-6	-7
-8	**-9**	-10	-11	-12	-13	-14	-15
-16	-17	**-18**	-19	-20	-21	-22	-23
-24	-25	-26	**-27**	-28	-29	-30	-31
-32	-33	-34	-35	**-36**	-37	-38	-39
-40	-41	-42	-43	-44	**-45**	-46	-47
-48	-49	-50	-51	-52	-53	**-54**	-55
-56	-57	-58	-59	-60	-61	-62	**-63**

Processus 0 (après)

0	1	2	3	4	5	6	7
-1	**9**	10	11	12	13	14	15
-2	-3	**18**	19	20	21	22	23
-4	-5	-6	**27**	28	29	30	31
-7	-10	-11	-12	**36**	37	38	39
-13	-14	-15	-19	-20	**45**	46	47
-21	-22	-23	-28	-29	-30	**54**	55
-31	-37	-38	-39	-46	-47	-55	**63**

Processus 1 (après)

0	8	16	17	24	25	26	32
-8	**-9**	33	34	35	40	41	42
-16	-17	**-18**	43	44	48	49	50
-24	-25	-26	**-27**	51	52	53	56
-32	-33	-34	-35	**-36**	57	58	59
-40	-41	-42	-43	-44	**-45**	60	61
-48	-49	-50	-51	-52	-53	**-54**	62
-56	-57	-58	-59	-60	-61	-62	**-63**

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define N 8
5  #define SIGN(a,b) ((b) == 0) ? abs(a) : ((b)/abs(b))*abs(a)
6  int main(int argc, char *argv[])
7  {
8    int i, j, rang, etiquette=100, a[N][N];
9    MPI_Status statut;
10   MPI_Datatype type_triangle;
11   int *longueurs_blocs, *deplacements;
12
13   MPI_Init(&argc,&argv);
14   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
15
16   /* Initialisation de la matrice sur chaque processus */
17   for (i=0; i<N; i++)
18     for (j=0; j<N; j++)
19       a[i][j] = SIGN(i*N + j,-rang);
20
21   /* Création du type matrice triangulaire sup pour le processus 0
22      et du type matrice triangulaire inférieure pour le processus 1 */
23   longueurs_blocs = (int *)malloc(sizeof(int)*N);
24   deplacements    = (int *)malloc(sizeof(int)*N);
```

```
25    if (rang == 0) {
26      for (i=0; i<N; i++) {
27        longueurs_blocs[i] = i;
28        deplacements[i]    = N*i;
29      }
30    }
31    else {
32      for (i=0; i<N; i++) {
33        longueurs_blocs[i] = N-i-1;
34        deplacements[i]    = i*(N+1)+1;
35      }
36    }
37    MPI_Type_indexed(N, longueurs_blocs, deplacements, MPI_INT, &type_triangle);
38
39    /* Validation du type type_triangle */
40    free(longueurs_blocs); free(deplacements); MPI_Type_commit(&type_triangle);
41
42    /* Permutation des matrices triangulaires supérieure et inférieure */
43    MPI_Sendrecv_replace(a, 1, type_triangle, (rang+1)%2, etiquette, (rang+1)%2,
44                         etiquette, MPI_COMM_WORLD, &statut);
45
46    /* Libération du type */
47    MPI_Type_free(&type_triangle);
48    MPI_Finalize(); exit(0);
49  }
```

6.7 – Types hétérogènes

- La fonction `MPI_Type_create_struct()` est le constructeur de types le plus général.
- Il a les mêmes fonctionnalités que `MPI_Type_indexed()` mais permet en plus la réplication de blocs de données de types différents.
- Les paramètres de `MPI_Type_create_struct()` sont les mêmes que ceux de `MPI_Type_indexed()` avec en plus :
  - le champ *anciens_types* est maintenant un vecteur de types de données *MPI* ;
  - compte tenu de l'hétérogénéité des données et de leur alignement en mémoire, le calcul du déplacement entre deux éléments repose sur la différence de leurs adresses.

    MPI, via `MPI_Get_address()`, fournit une fonction qui permet de retourner l'adresse d'une variable.

nb=5, longueurs_blocs=( ), déplacements=( ),

anciens_types=( )

type 1 type 2 type 3

anciens_types

nouveau_type

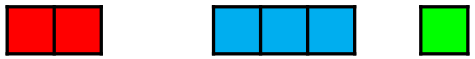


FIGURE 30 – Le constructeur MPI_Type_create_struct

```
int nb, code, *longueurs_blocs;
MPI_Aint *deplacements;
MPI_Datatype *anciens_types, nouveau_type;

code=MPI_Type_create_struct(nb,longueurs_blocs,deplacements,anciens_types,&nouveau_type);
```

nb=5, longueurs_blocs=(3,), déplacements=(0,),

anciens_types=(type1,)

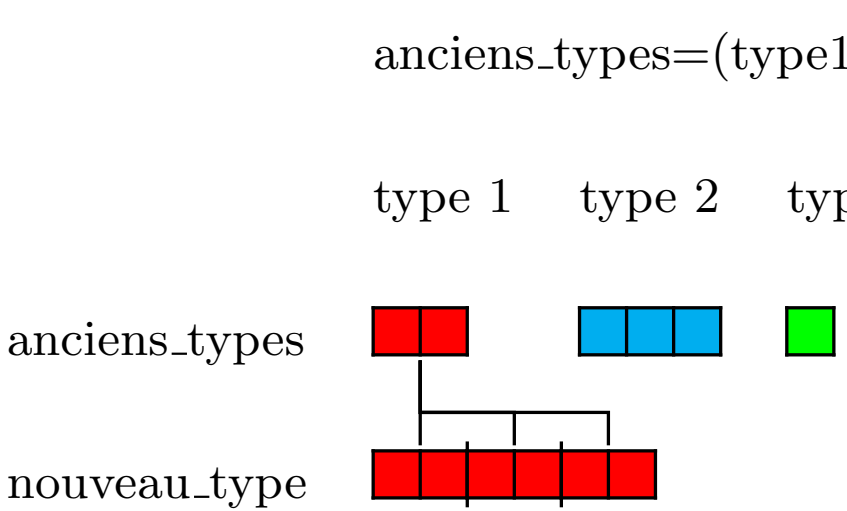


FIGURE 30 – Le constructeur MPI_Type_create_struct

```
int nb, code, *longueurs_blocs;
MPI_Aint *deplacements;
MPI_Datatype *anciens_types, nouveau_type;

code=MPI_Type_create_struct(nb,longueurs_blocs,deplacements,anciens_types,&nouveau_type);
```

nb=5, longueurs_blocs=(3,1,), déplacements=(0,7,),

anciens_types=(type1,type2,)

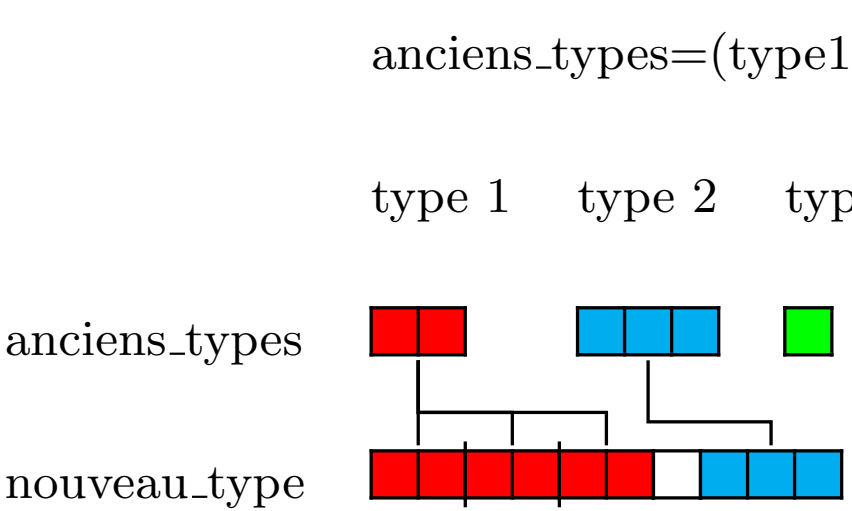


FIGURE 30 – Le constructeur MPI_Type_create_struct

```
int nb, code, *longueurs_blocs;
MPI_Aint *deplacements;
MPI_Datatype *anciens_types, nouveau_type;

code=MPI_Type_create_struct(nb,longueurs_blocs,deplacements,anciens_types,&nouveau_type);
```

nb=5, longueurs_blocs=(3,1,5,), déplacements=(0,7,11,),

anciens_types=(type1,type2,type3,)

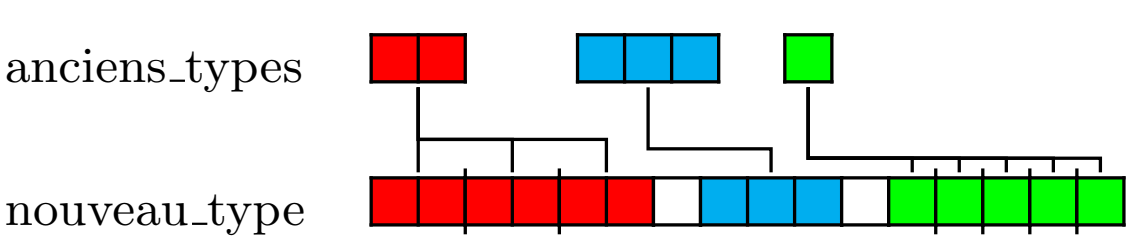


FIGURE 30 – Le constructeur MPI_Type_create_struct

```
int nb, code, *longueurs_blocs;
MPI_Aint *deplacements;
MPI_Datatype *anciens_types, nouveau_type;

code=MPI_Type_create_struct(nb,longueurs_blocs,deplacements,anciens_types,&nouveau_type);
```

nb=5, longueurs_blocs=(3,1,5,1,), déplacements=(0,7,11,21,),

anciens_types=(type1,type2,type3,type1,)

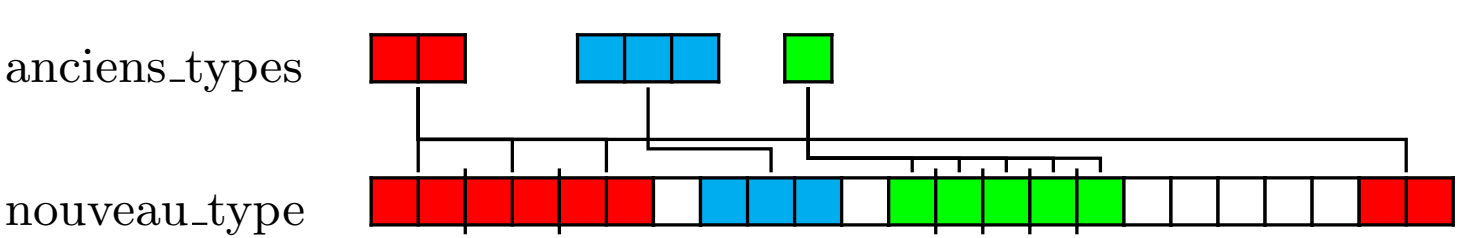


FIGURE 30 – Le constructeur MPI_Type_create_struct

```
int nb, code, *longueurs_blocs;
MPI_Aint *deplacements;
MPI_Datatype *anciens_types, nouveau_type;

code=MPI_Type_create_struct(nb,longueurs_blocs,deplacements,anciens_types,&nouveau_type);
```

nb=5, longueurs_blocs=(3,1,5,1,1), déplacements=(0,7,11,21,26),

anciens_types=(type1,type2,type3,type1,type3)

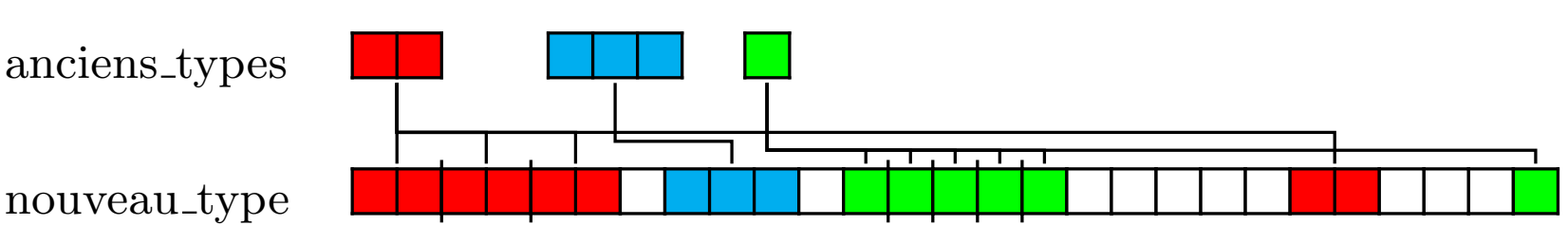


FIGURE 30 – Le constructeur MPI_Type_create_struct

```
int nb, code, *longueurs_blocs;
MPI_Aint *deplacements;
MPI_Datatype *anciens_types, nouveau_type;

code=MPI_Type_create_struct(nb,longueurs_blocs,deplacements,anciens_types,&nouveau_type);
```

```
int code;
void *variable;
MPI_Aint *adresse;

code=MPI_Get_address(&variable, &adresse);
```

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #define TAILLE_IMAGE 691476  /* octets */
4
5  /* Fonctions externes */
6  extern int loadtiff(char *fileName, unsigned char *image, int *iw, int *ih);
7  extern int dumpTiff(char *fileName, unsigned char *image, int *w, int *h);
8
9  typedef struct {
10   int largeur, hauteur;                /* Dimensions de l'image en pixels */
11   unsigned char donnees[TAILLE_IMAGE]; /* Données associées à l'image     */
12 } Image;
13
14 int main(int argc, char *argv[])
15 {
16   int i, rang, nb_pixels;
17   Image img;
18   int longueurs_blocs[3];
19   MPI_Aint deplacements[3];
20   MPI_Datatype types[3]={ MPI_INT, MPI_INT, MPI_UNSIGNED_CHAR }, type_img;
```

```
21    MPI_Init(&argc,&argv);
22    MPI_Comm_rank(MPI_COMM_WORLD, &rang);
23
24    /* Le processus 0 lit le fichier image "Eiffel.tif" */
25    if (rang == 0) {
26        loadtiff("Eiffel.tif", img.donnees, &img.largeur, &img.hauteur);
27        nb_pixels=img.largeur*img.hauteur;
28    }
29    MPI_Bcast(&nb_pixels, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
30    /* Création du type_img pour chacun des processus */
31    longueurs_blocs[0]=1;
32    longueurs_blocs[1]=1;
33    longueurs_blocs[2]=nb_pixels;
34    MPI_Get_address(&img.largeur, &deplacements[0]);
35    MPI_Get_address(&img.hauteur, &deplacements[1]);
36    MPI_Get_address(&img.donnees[0], &deplacements[2]);
37    for (i=2; i>0; i--) deplacements[i] -= deplacements[0];
38
39    MPI_Type_create_struct(3, longueurs_blocs, deplacements, types, &type_img);
40
41    /* Validation du type type_img */
42    MPI_Type_commit(&type_img);
43
44    /* Diffusion de l'image à l'ensemble des processus */
45    MPI_Bcast(&img, 1, type_img, 0, MPI_COMM_WORLD);
46
47    /* Libération du type */
48    MPI_Type_free(&type_img);
49
50    /*
51     * if(rang==1) dumpTiff("Eiffel_b.tif",img.donnees,&img.largeur,&img.hauteur);
52     */
53    MPI_Finalize(); return(0);
54  }
```

6.8 – Fonctions annexes

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

  ✓ Attention à ne pas la confondre avec `MPI_Type_get_extent()` qui, elle, renvoie, en particulier, la taille d'un type en tenant compte des éventuels alignements en mémoire

- L'adresse de la borne inférieure (lb) d'un type de données relativement à l'origine ainsi que son étendue : `MPI_Type_get_extent()`.

  ✓ L'adresse de la borne supérieure (ub) est obtenue par : ub = lb + etendue

- `MPI_Type_size()` $\leq$ ub − 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 par exemple avec `MPI_Type_create_struct()`

```
int code;
int taille;
MPI_Datatype type;

code=MPI_Type_size(type, &taille);
```

```
int code;
MPI_Datatype type;
MPI_Aint lb, etendue;

code=MPI_Type_get_extent(type, &lb, &etendue);
```

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 procédures 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 partitionnement de domaines.

6.10 – Exercice 7 : type colonne d'une matrice

- On se donne une matrice A initialisée sur chacun des processus
- Il s'agit pour le processus 0 d'envoyer les *seconde* et *troisième* colonnes de sa matrice au processeur 1 et pour celui-ci de les recevoir dans ses *avant-dernière* et *dernière* colonnes

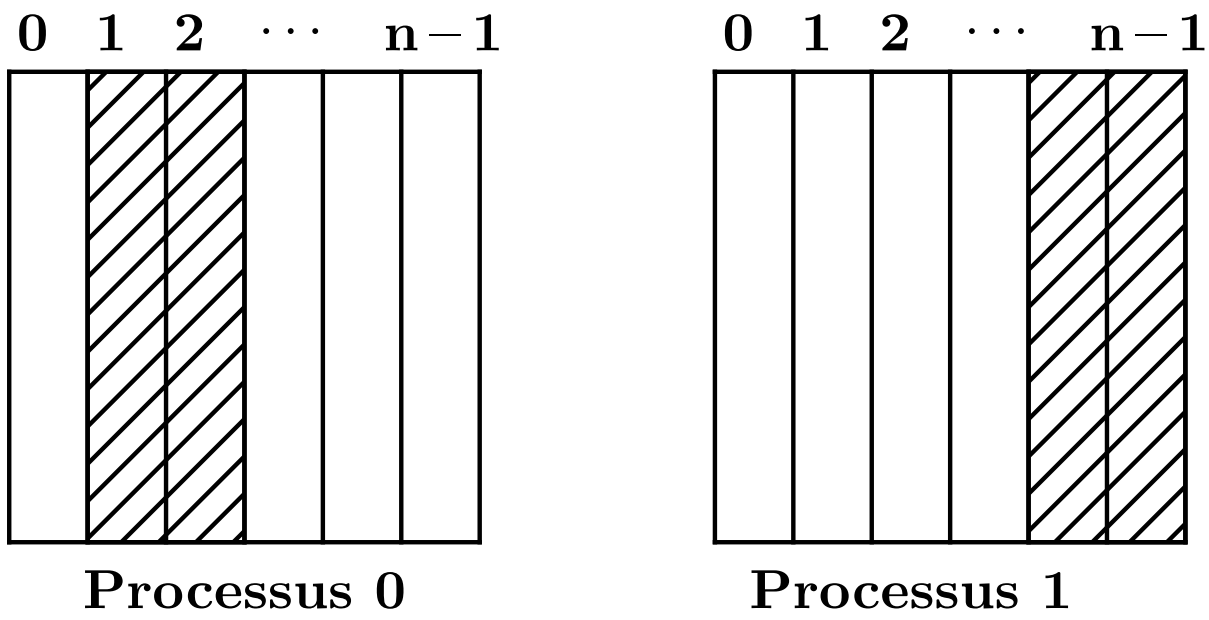


FIGURE 31 – Transfert de deux colonnes de matrice

☞ Pour ce faire, on va devoir se construire un type `type_colonne` qui décrira deux colonnes de la matrice

7 – Topologies

7.1 – Introduction

- Dans la plupart des applications où l'on fait correspondre le domaine de traitement à la grille de processus, il est intéressant de pouvoir disposer ces 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 la fonction `MPI_Cart_create()`.

```
int code, ndims, periods, reorganisation, *dims;
MPI_Comm comm_ancien, comm_nouveau;

code=MPI_Cart_create(comm_ancien, ndims,dims,periods,reorganisation,&comm_nouveau);
```

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

```
#include 'mpi.h'
#define NDIMS 2
int reorganisation;
int dims[NDIMS], periods[NDIMS];
MPI_Comm comm_2D;
...
dims[0] = 4, dims[1] = 2;
periods[0] = 0, periods[1] = 1;
reorganisation = 0;
...
MPI_Cart_create(MPI_COMM_WORLD,NDIMS,dims,periods,reorganisation,&comm_2D);
...
```

☞ Si `reorganisation = 0 (faux)` alors le rang des processus dans le nouveau communicateur (**comm_2D**) est le même que dans l'ancien communicateur (`MPI_COMM_WORLD`). Si `reorganisation = 1 (vrai)`, l'implémentation MPI choisit l'ordre des processus.

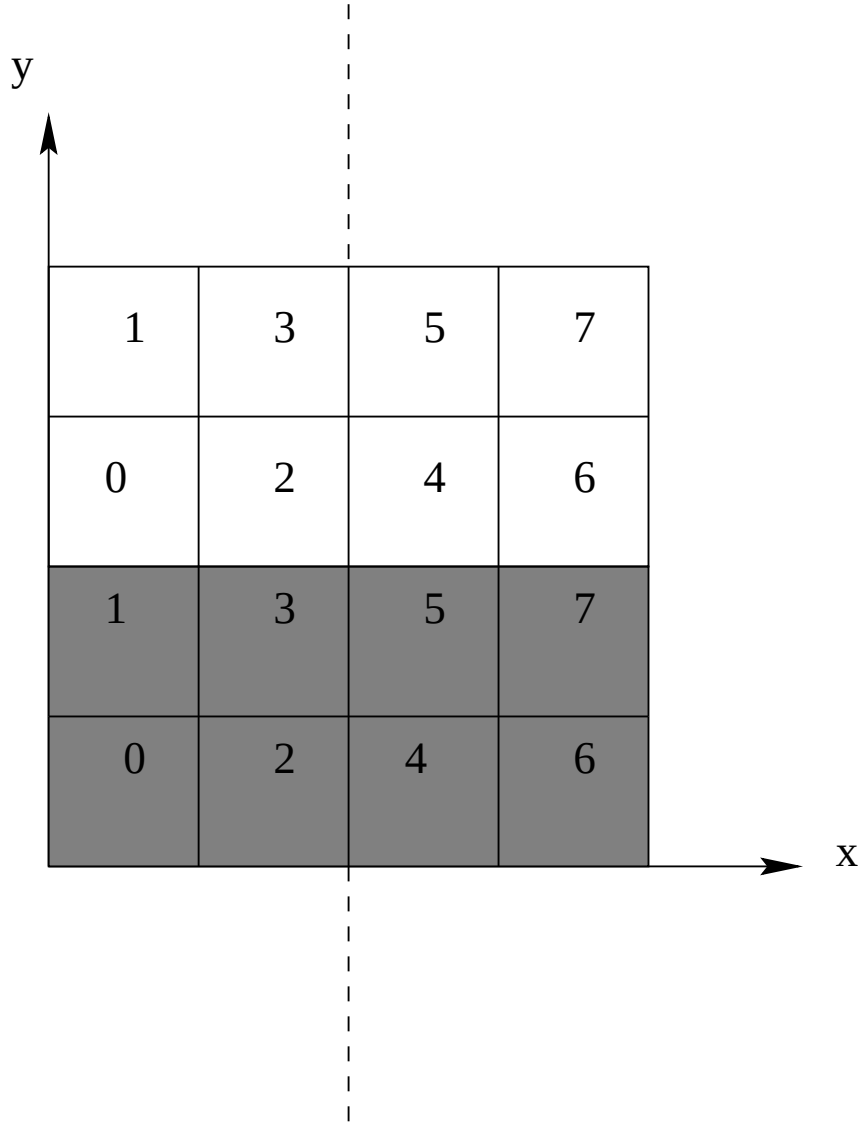


FIGURE 32 – Topologie cartésienne 2D périodique en y

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

```
#include 'mpi.h'
#define NDIMS 3
int reorganisation;
int periods[NDIMS]={ 0, 0, 0 };
int dims[NDIMS]={ 4, 2, 2 };
MPI_Comm comm_3D;
...
reorganisation = 0;
MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,&comm_3D);
...
```

2	6	10	14
0	4	8	12

$z = 0$

3	7	11	15
1	5	9	13

$z = 1$

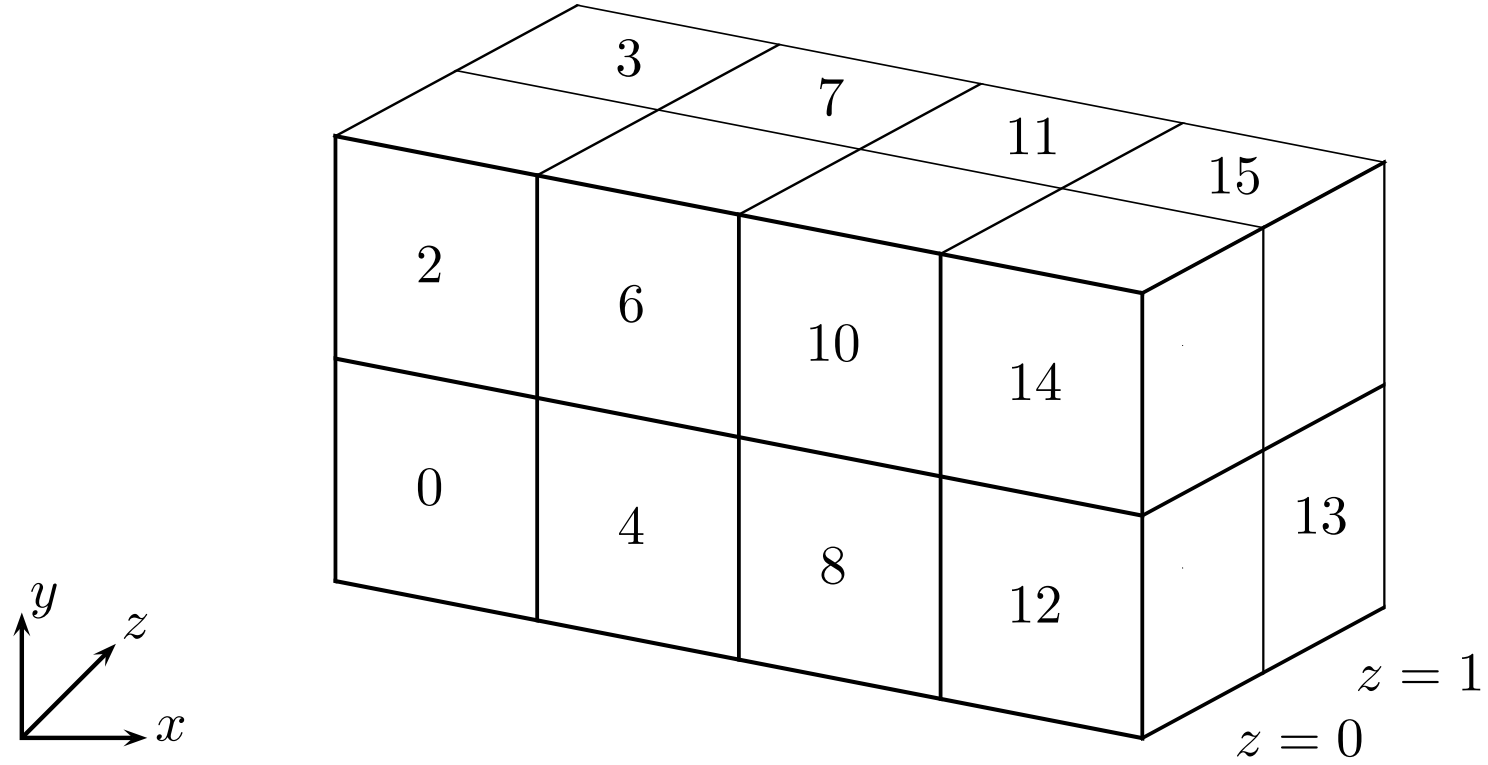


FIGURE 33 – Topologie cartésienne 3D non périodique

☞ Dans une topologie cartésienne, la fonction `MPI_Dims_create()` retourne le nombre de processus dans chaque dimension de la grille en fonction du nombre total de processus.

```
#include "mpi.h"
int code, nb_procs, ndims;
int *dims;

code=MPI_Dims_create(nb_procs,ndims,dims);
```

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

`dims` en entrée	`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, la fonction `MPI_Cart_rank()` retourne le rang du processus associé aux coordonnées dans la grille.

```
int rang, code;
int *coords;
MPI_Comm comm_nouveau;

code=MPI_Cart_rank(comm_nouveau,coords,&rang);
```

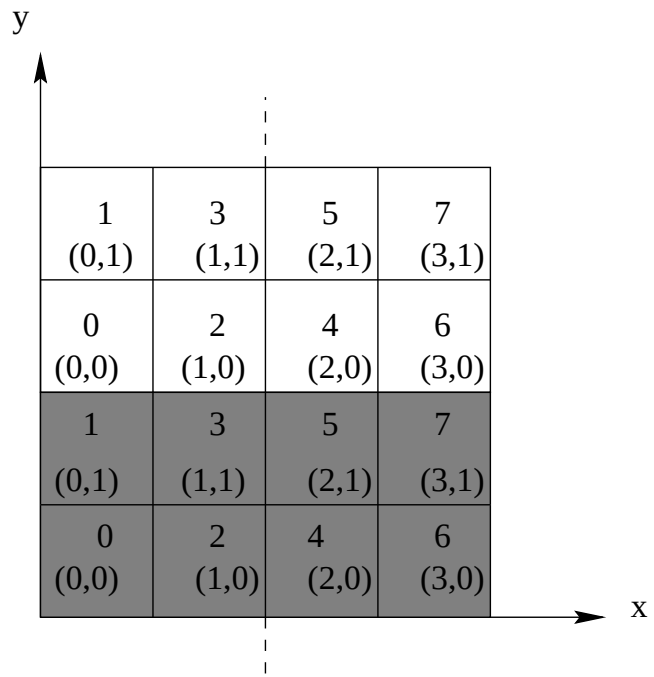


FIGURE 34 – Topologie cartésienne 2D périodique en y

```
coords[0]=dims[0]-1;
for (i=0; i<dims[1]-1; i++) {
  coords[1] = i;
  MPI_Cart_rank(comm_2D,coords,&rang[i]);
}
...
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, la fonctions `MPI_Cart_coords()` retourne les coordonnées d'un processus de rang donné dans la grille.

```
int code, rang, ndims;
int *coords;
MPI_Comm comm_nouveau;

code=MPI_Cart_coords(comm_nouveau, rang, ndims, coords);
```

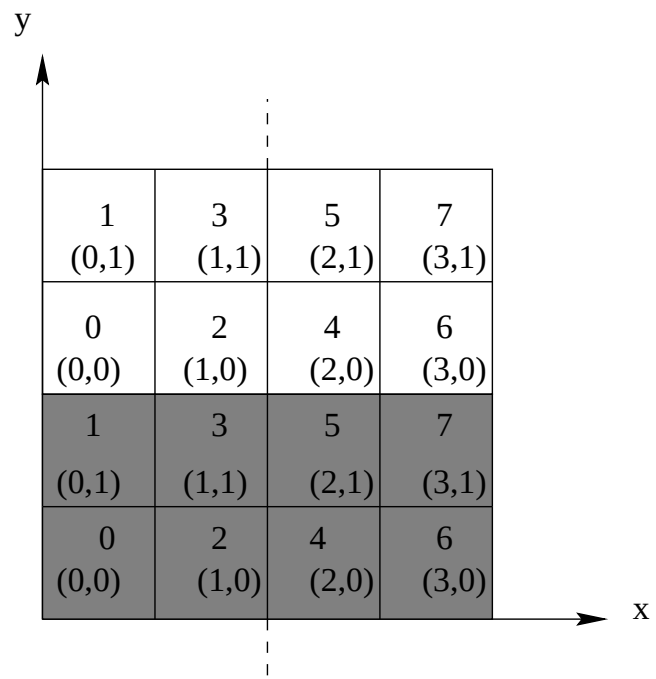


FIGURE 35 – Topologie cartésienne 2D périodique en y

```
...
if (rang%2 == 0)
  MPI_Cart_coords(comm_2D,rang,2,coords);
...
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 la fonction `MPI_Cart_shift()` se voit retourner le rang de ses processus voisins dans une direction donnée.

```
int code, direction, pas;
int rang_precedent, rang_suivant;
MPI_Comm comm_nouveau;

code=MPI_Cart_shift(comm_nouveau, direction, pas, &rang_precedent, &rang_suivant);
```

☞ Le paramètre **direction** correspond à l'axe du déplacement (xyz).

☞ Le paramètre **pas** correspond au pas du déplacement.

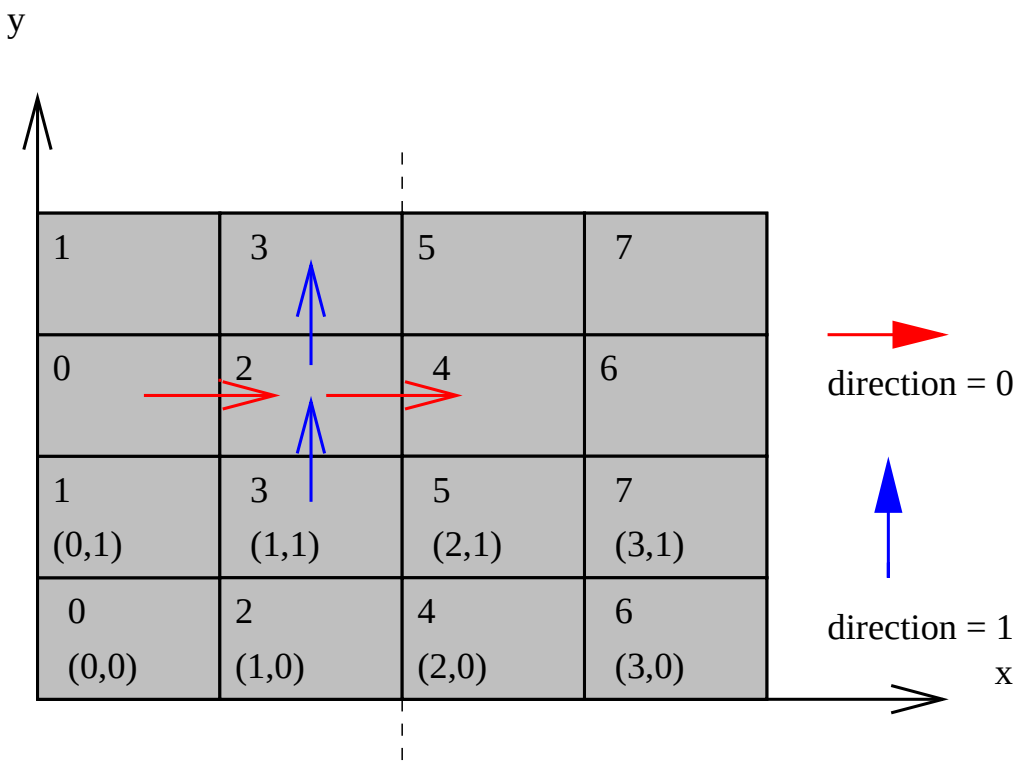


FIGURE 36 – Appel de la fonction MPI_Cart_shift()

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

```
MPI_Cart_shift(comm_2D,1,1,&rang_bas,&rang_haut);
...
Pour le processus 2, rang_bas=3, rang_haut=3
```

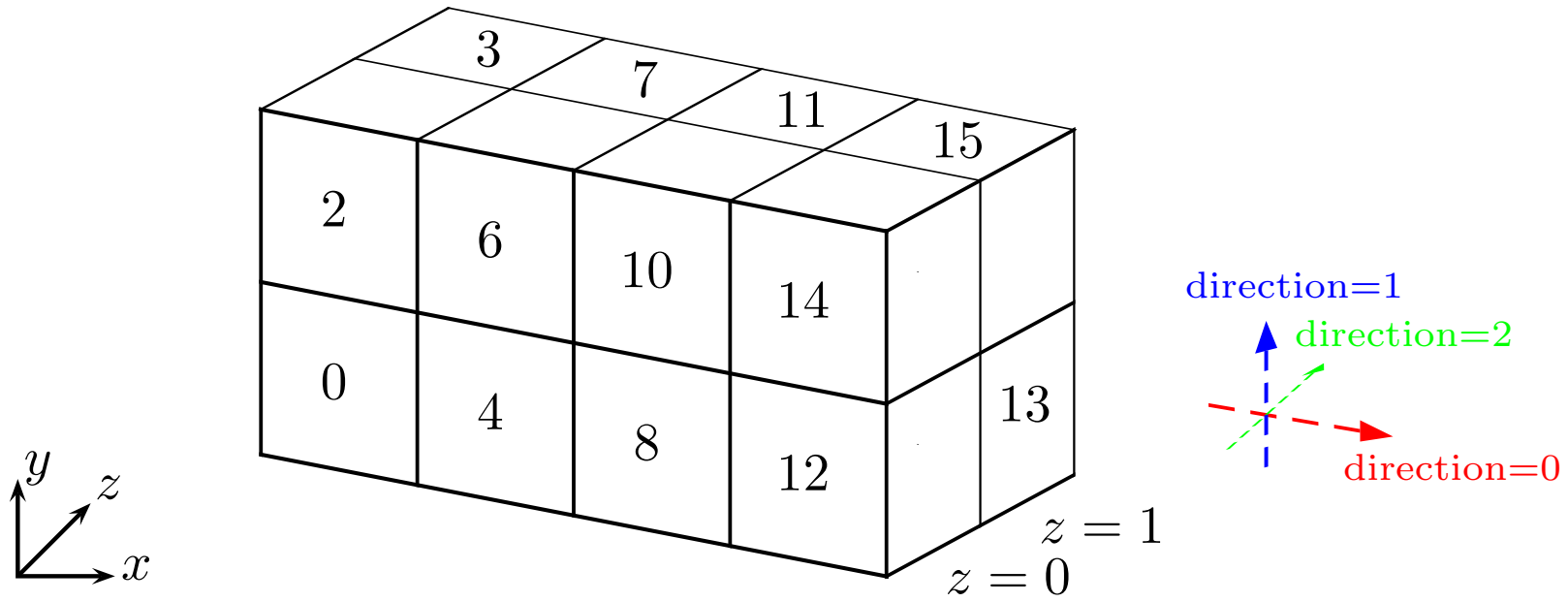


FIGURE 37 – Appel de la fonction MPI_Cart_shift()

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

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

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

☞ Exemple de programme :

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define NDIMS 2
5  int main(int argc, char *argv[])
6  {
7    int i, rang_ds_topo, nb_procs, reorganisation, N=1, E=2, S=3, W=4;
8    int voisin[2*NDIMS], dims[NDIMS], coords[NDIMS] periods[NDIMS];
9    MPI_Comm comm_2D;
10
11   MPI_Init(&argc,&argv);
12   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
13
14   /* Connaître le nombre de processus suivant x et y */
15   dims[0]=0, dims[1]=0;
16   MPI_Dims_create(nb_procs, NDIMS, dims);
```

```
17    /* Création grille 2D periodique en y */
18    periods[0] = 0, periods[1] = 1;
19    reorganisation = 0;
20    MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,&comm_2D);
21
22    /* Connaître mes coordonnées dans la topologie */
23    MPI_Comm_rank(comm_2D,&rang_ds_topo);
24    MPI_Cart_coords(comm_2D,rang_ds_topo,NDIMS,coords);
25
26    /* Initialisation du tableau voisin à la valeur MPI_PROC_NULL */
27    for (i=0, i<2*NDIMS; i++) voisin[i] = MPI_PROC_NULL;
28
29    /* Recherche de mes voisins Ouest et Est */
30    MPI_Cart_shift(comm_2D, 0, 1, &voisin[W], &voisin[E]);
31
32    /* Recherche de mes voisins Sud et Nord */
33    MPI_Cart_shift(comm_2D, 1, 1, &voisin[S], &voisin[N]);
34
35    MPI_Finalize()
36    return(0);
37 }
```

7.4 – 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.

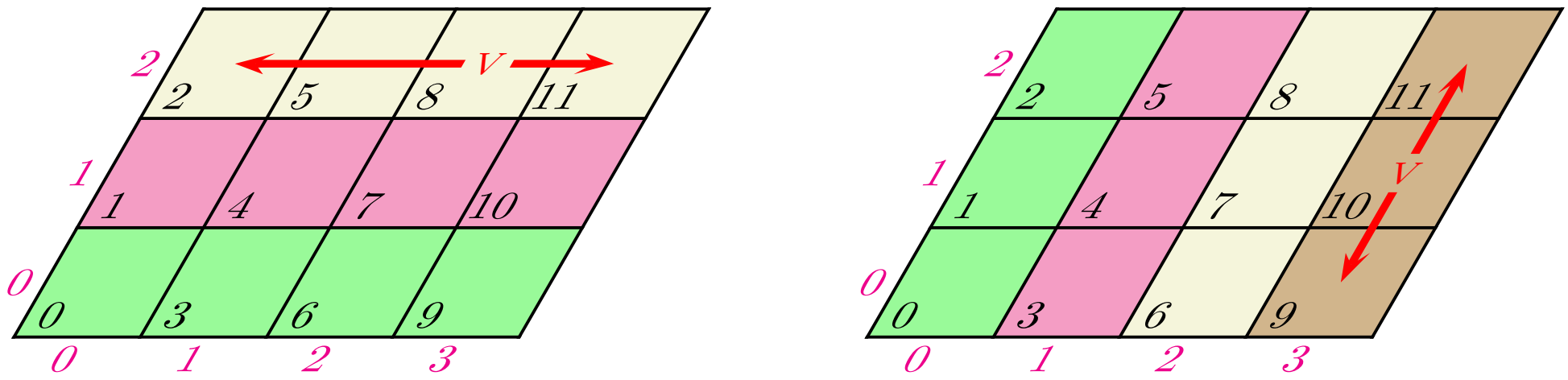


FIGURE 38 – Deux exemples de distribution de données dans une topologie 2D dégénérée

Pour se faire, une seule fonction à connaître : `MPI_cart_sub()`.

```
int code;
int *Subdivision; /* Pointeur sur un tableau de booleens */
MPI_Comm CommCart, CommCartD;
code=MPI_Cart_sub(CommCart, Subdivision, &CommCartD);
```

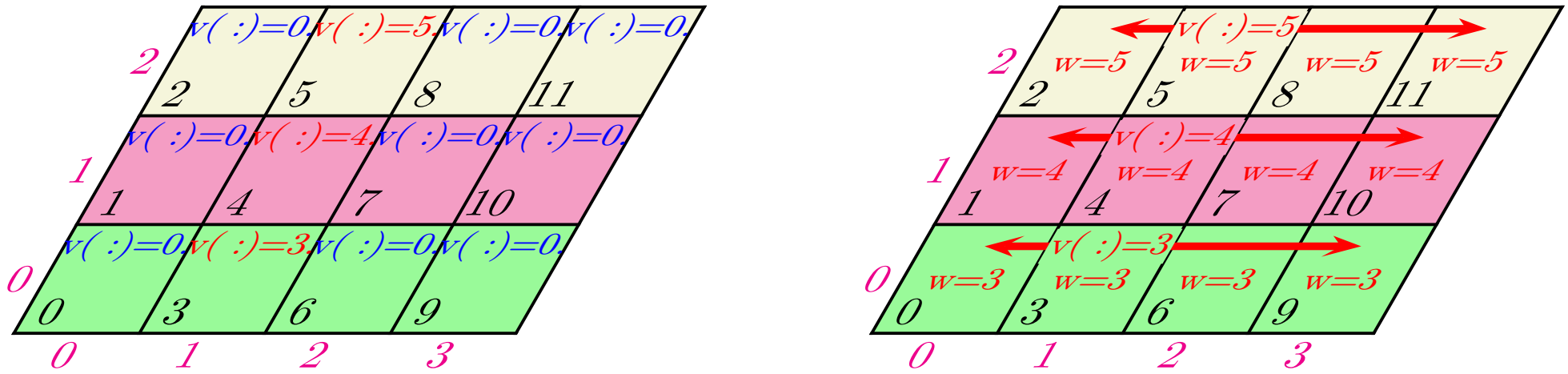


FIGURE 39 – Représentation initiale d'un tableau V dans la grille 2D et représentation finale après la distribution de celui-ci sur la grille 2D dégénérée

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #define NDIM2d 2
4  #define M 4
5  int main( int argc, char *argv[])
6  {
7   int i, rang, reordonne;
8   int dim2d[NDIM2d], coord2d[NDIM2d], periode[NDIM2d] ,subdivision[NDIM2d];
9   MPI_Comm comm2d, comm1d;
10  float v[M], w=0;
11
12  MPI_Init(&argc,&argv);
13
14  /* Création de la grille 2d initiale */
15  dim2d[0]=4,   dim2d[1]=3;  /* Exécution sur 4*3=12 processus */
16  periode[0]=0, periode[1]=0;
17  reordonne=0;
18  MPI_Cart_create(MPI_COMM_WORLD, NDIM2d, dim2d, periode, reordonne, &comm2d);
19
20  MPI_Comm_rank(comm2d, &rang);
21  MPI_Cart_coords(comm2d, rang, NDIM2d, coord2d);
```

```
22  /* Initialisation du vecteur v */
23  if (coord2d[0] == 1) for (i=0; i<M; i++) v[i]=(float)rang;
24
25  /* Nous voulons que chaque ligne de la grille soit une topologie cartésienne 1D */
26  subdivision[0]=1, subdivision[1]=0;
27
28  /* Subdivision de la grille cartésienne 2d. */
29  MPI_Cart_sub(comm2d, subdivision, &comm1d);
30  /* Les processus de la colonne 2 distribuent le vecteur V  */
31  /* aux processus de leur ligne.                            */
32  MPI_Scatter(v, 1, MPI_FLOAT, &w, 1, MPI_FLOAT, 1, comm1d);
33
34  printf("Rang : %d ; coordonnees : (%d,%d) ; w = %2.0f\n",rang,coord2d[0],coord2d[1],w);
35  MPI_Finalize(); return(0);
36 }
```

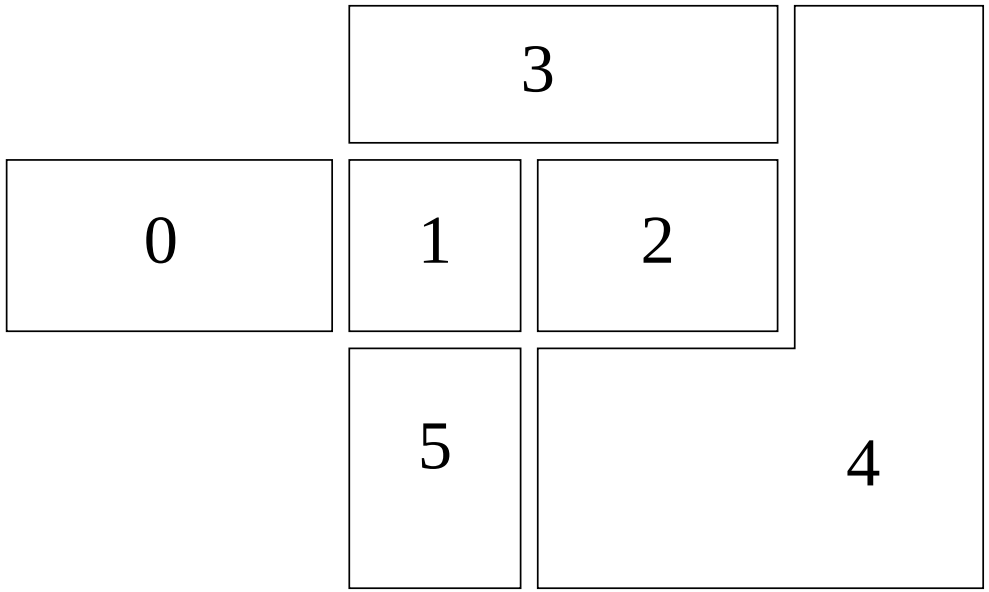
```
> mpirun -np 12 CommCartSub
Rang :  0 ; Coordonnees : (0,0) ; W = 3.
Rang :  1 ; Coordonnees : (0,1) ; W = 4.
Rang :  3 ; Coordonnees : (1,0) ; W = 3.
Rang :  8 ; Coordonnees : (2,2) ; W = 5.
Rang :  4 ; Coordonnees : (1,1) ; W = 4.
Rang :  5 ; Coordonnees : (1,2) ; W = 5.
Rang :  6 ; Coordonnees : (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.
```

7.5 – Graphe de processus

Il arrive cependant que dans certaines applications (géométries complexes), la décomposition ne soit plus une grille régulière mais un graphe dans lequel chaque partie peut avoir un ou plusieurs voisins quelconques. La fonction `MPI_Graph_create()` permet alors de définir une topologie de type graphe en indiquant les voisins de chaque partie du domaine.

```
1 int code, nb_procs, reorganisation;
2 int *index, *liste_voisins;
3 MPI_Comm comm_ancien, comm_nouveau;
4
5 code=MPI_Graph_create(comm_ancien, nb_procs,index, liste_voisins, reorganisation,
6                        &comm_nouveau);
```

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

FIGURE 40 – Graphe de processus

```
index[6]          = { 1,       5,     8,    11,    14,  16 };
liste_voisins[16] = { 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 :

```
int code, rang, nb_voisins;
MPI_Comm comm_nouveau;

code=MPI_Graph_neighbors_count(comm_nouveau, rang, &nb_voisins);
```

- la liste des voisins pour un processus donné :

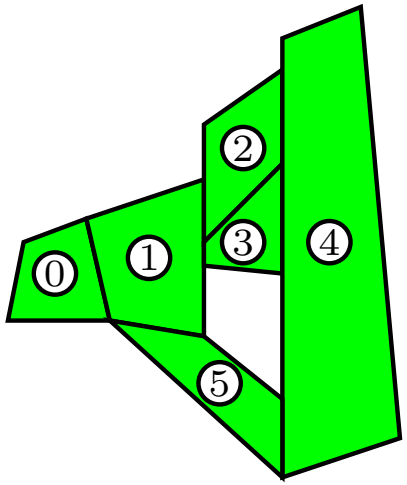
```
int code, rang, nb_voisins;
int *voisins;
MPI_Comm comm_nouveau;

code=MPI_Graph_neighbors(comm_nouveau, rang, nb_voisins, voisins);
```

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define MIN(a,b) (a)<(b)?(a):(b)
5  int main(int argc, char *argv[])
6  {
7    int i, code, rang, nb_procs, nb_voisins, iteration=0, etiquette=100;
8
9    /* On définit les voisins de chacune des parcelles */
10   int index[]         ={ 1,       5,     8,    11,   14,   16 };
11   int liste_voisins[]={ 1, 0,5,2,3, 1,3,4, 1,2,4, 3,2,5, 1,4 };
12   int *voisins;
13   MPI_Status statut;
14   MPI_Comm comm_graphe;
15
16   float p, propagation;      /* Propagation du feu              */
17   float feu=0.;              /*  Valeur du feu                  */
18   float bois=1.;             /* Rien n'a encore brûlé           */
19   float arret=1.;            /* Tout a brûlé si arret <= 0.01 */
20
21   MPI_Init(&argc,&argv);
22   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
23   if (nb_procs != 6) MPI_Abort(MPI_COMM_WORLD, code), exit(1);
24
25   MPI_Graph_create(MPI_COMM_WORLD, nb_procs, index, liste_voisins, 0, &comm_graphe);
26   MPI_Comm_rank(comm_graphe, &rang);
```

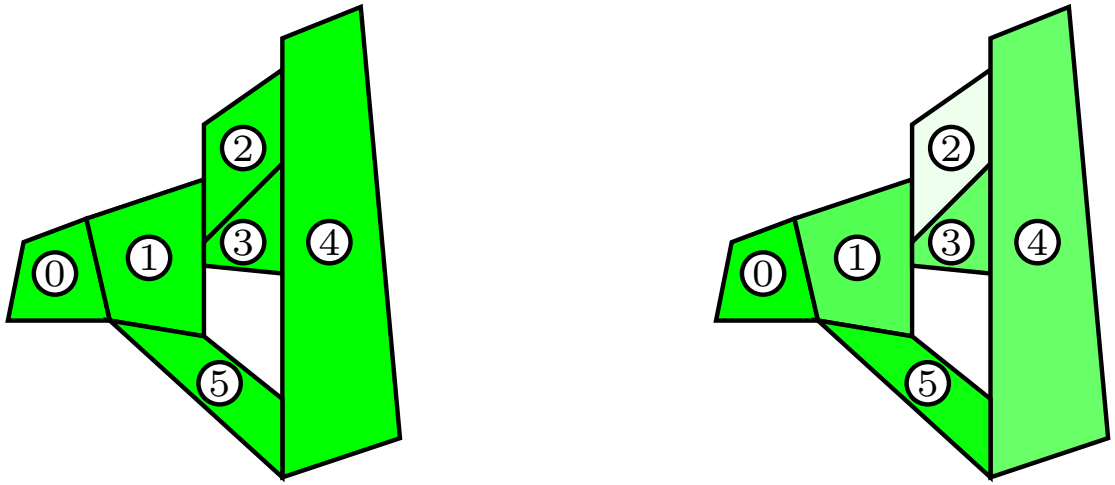
```
1   if (rang == 2) feu=1.;  /* Le feu se déclare arbitrairement sur la parcelle 2 */
2   MPI_Graph_neighbors_count(comm_graphe, rang, &nb_voisins);
3   voisins = (int *) malloc(nb_voisins*sizeof(int));
4   MPI_Graph_neighbors(comm_graphe, rang, nb_voisins, voisins);
5
6   while (arret > 0.01) {  /* On arrête dès qu'il n'y a plus rien à brûler */
7      for (i=0; i<nb_voisins; i++)  { /* On propage le feu aux voisins */
8        p=MIN(1.,feu);
9        MPI_Sendrecv(&p, 1, MPI_FLOAT, voisins[i], etiquette,
10                    &propagation, 1, MPI_FLOAT, voisins[i], etiquette,
11                    comm_graphe, &statut);
12        /* Le feu se développe en local sous l'influence des voisins */
13        feu=1.2*feu + 0.2*propagation*bois;
14        bois=bois/(1.+feu); /* On calcule ce qui reste de bois sur la parcelle */
15     };
16     MPI_Allreduce(&bois, &arret, 1, MPI_FLOAT, MPI_SUM, comm_graphe);
17
18     iteration +=1;
19     printf("Iteration %d parcelle %d bois=%5.3f\n", iteration, rang, bois);
20     MPI_Barrier(comm_graphe);
21     if (rang  == 0) printf("--\n");
22  }
23  free(voisins); MPI_Finalize(); return(0);
24 }
```

```
> 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
--
```



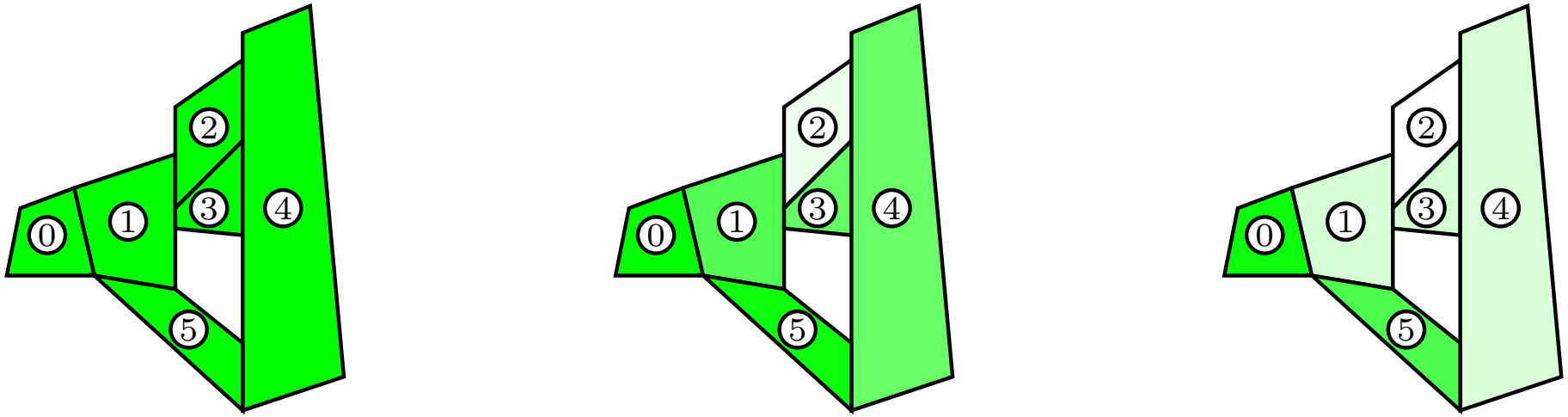
(a) Itération 0

FIGURE 41 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt



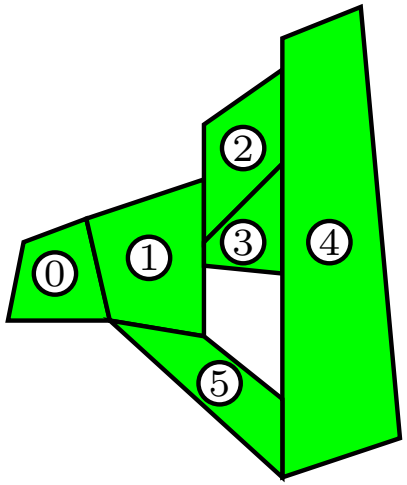
(a) Itération 0 (b) Itération 1

FIGURE 41 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt

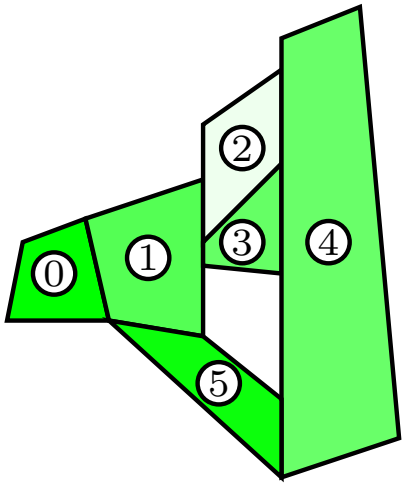


(a) Itération 0 (b) Itération 1 (c) Itération 2

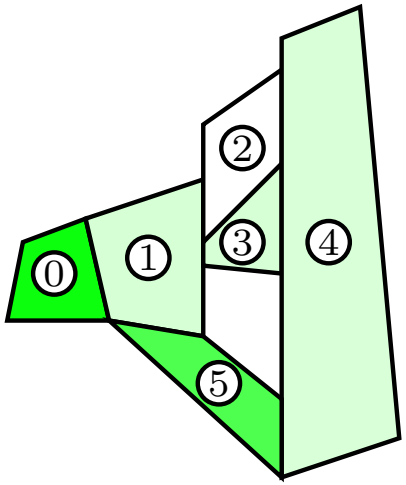
FIGURE 41 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt



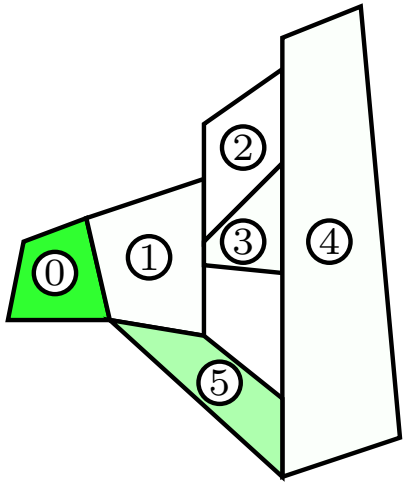
(a) Itération 0



(b) Itération 1

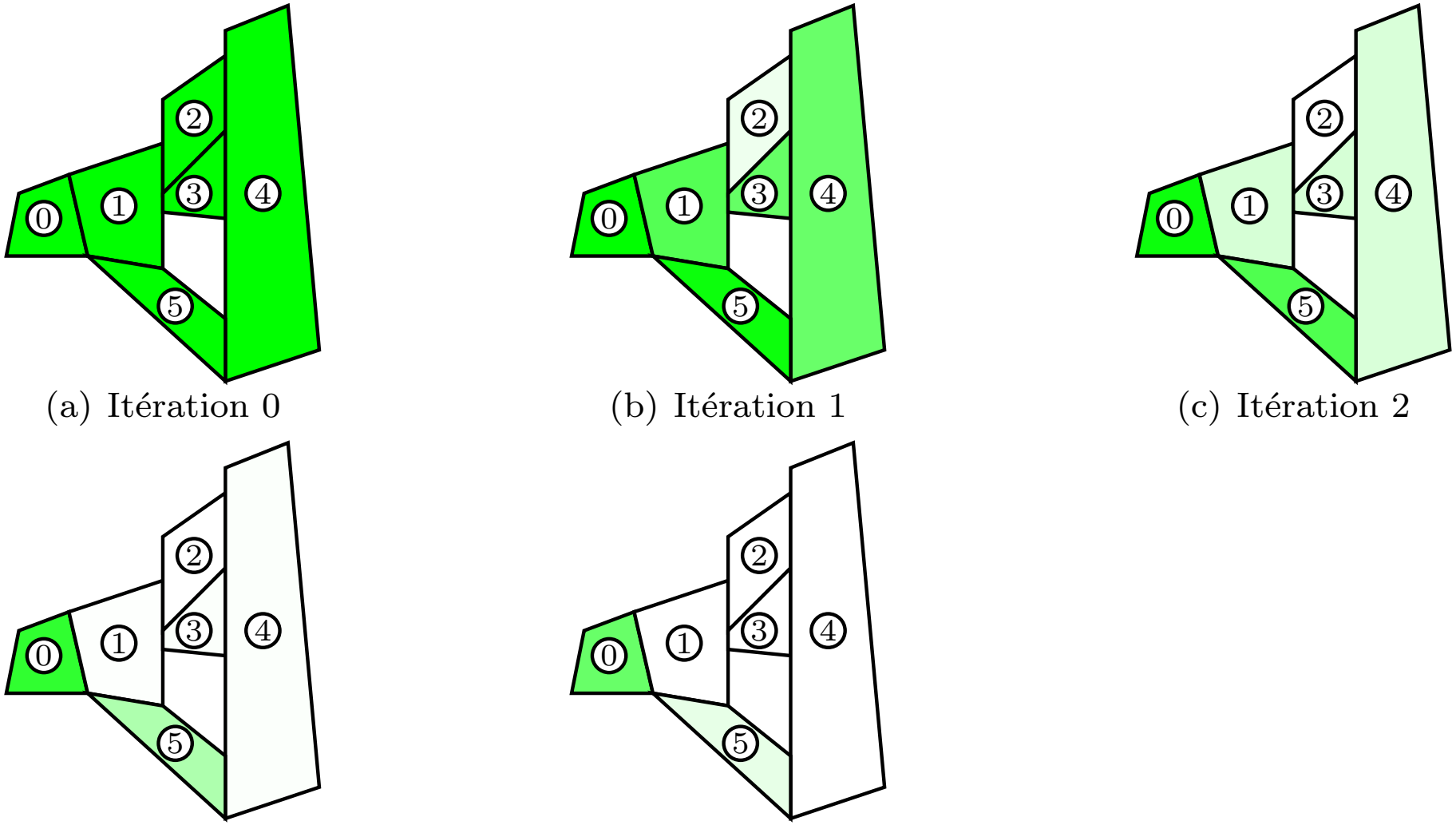


(c) Itération 2



(d) Itération 3

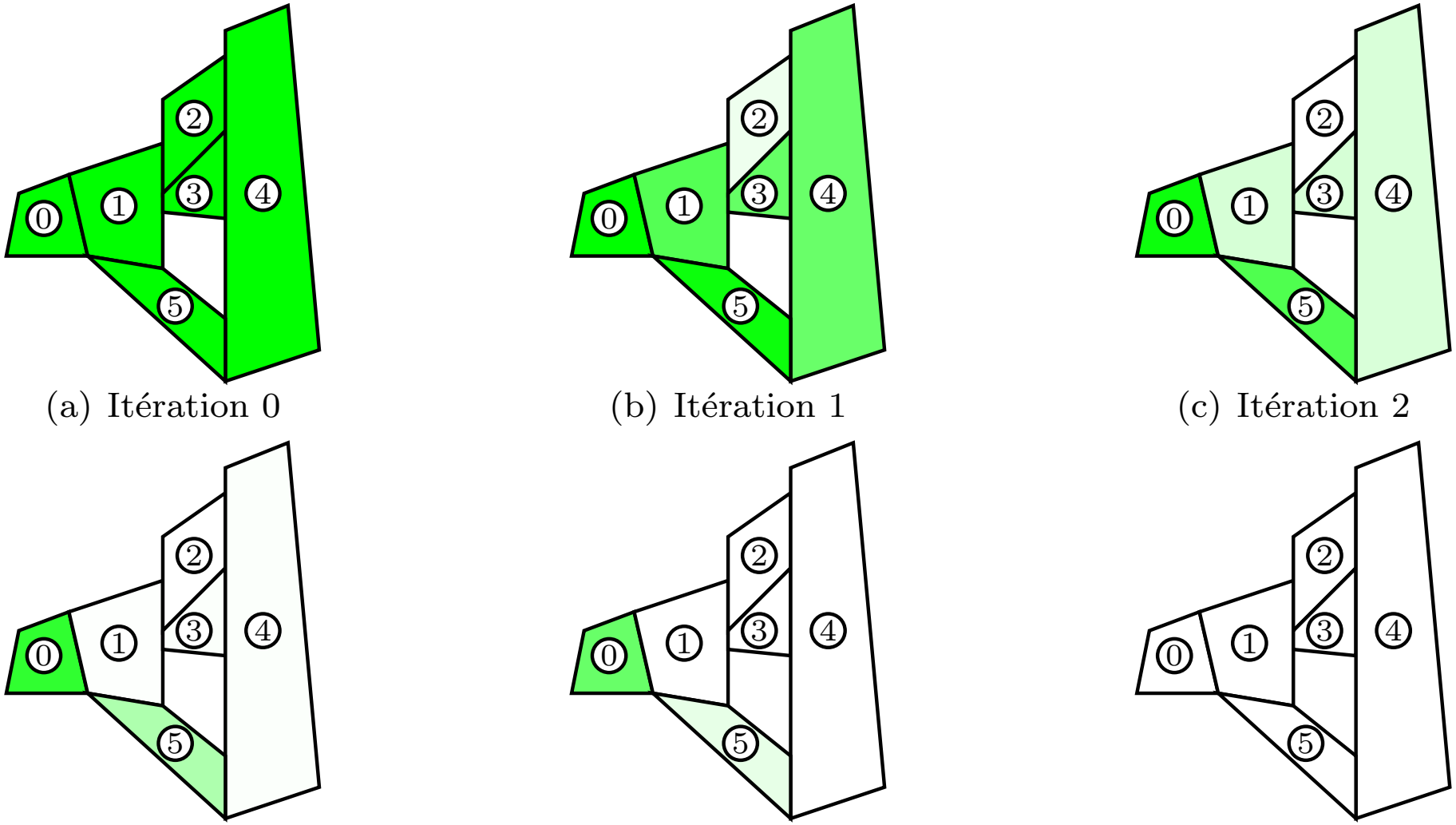
FIGURE 41 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt



(a) Itération 0 (b) Itération 1 (c) Itération 2

(d) Itération 3 (e) Itération 4

FIGURE 41 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt



(a) Itération 0 (b) Itération 1 (c) Itération 2

(d) Itération 3 (e) Itération 4 (f) Itération 10

FIGURE 41 – Définition d'une topologie quelconque via un graphe — Exemple de la propagation d'un feu de forêt

7.6 – Exercice 8 : image et grille de processus

- Disposant de 4 processus, répartir l'image "Eiffel.tif" sur une grille (2x2) de processus (utiliser le type bloc défini dans le chapitre précédent).
- Déterminer la valeur minimale globale d'un pixel sur chaque rangée de la grille de processus.

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.

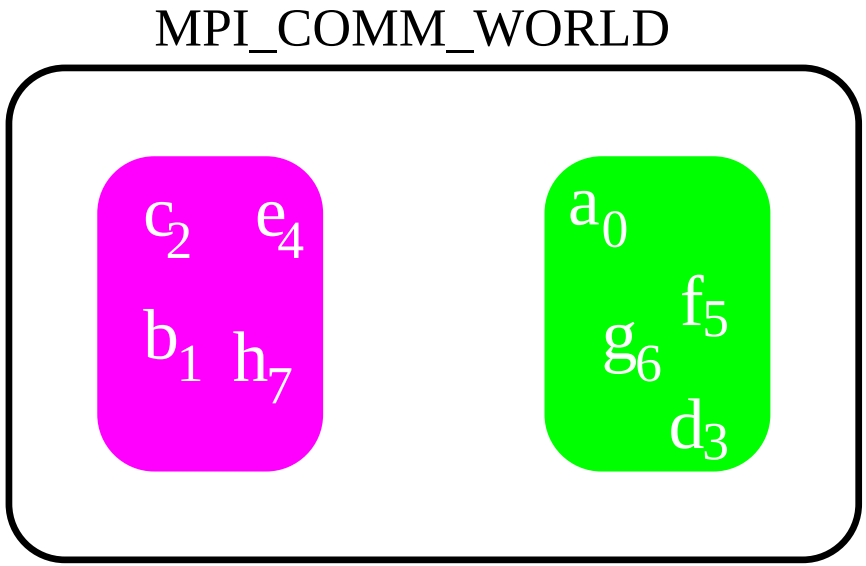


FIGURE 42 – Partitionnement d'un communicateur

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 `mpi.h`.
- Ce communicateur initial `MPI_COMM_WORLD` est créé pour toute la durée d'exécution du programme à l'appel de la fonction `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  #include "mpi.h"
2  #include <stdlib.h>
3  #define M 16
4  int main( int argc, char *argv[])
5  {
6   int i, rang;
7   float a[M];
8
9   MPI_Init(&argc,&argv);
10  MPI_Comm_rank(MPI_COMM_WORLD, &rang);
11
12  for (i=0; i<M; i++) a[i]=0.;
13
14  if(rang == 2)
15    for (i=0; i<M; i++) a[i]=(float)rang;
16
17  MPI_Bcast(a, M, MPI_FLOAT, 2, MPI_COMM_WORLD);
18
19  MPI_Finalize(); return(0);
20 }
```

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

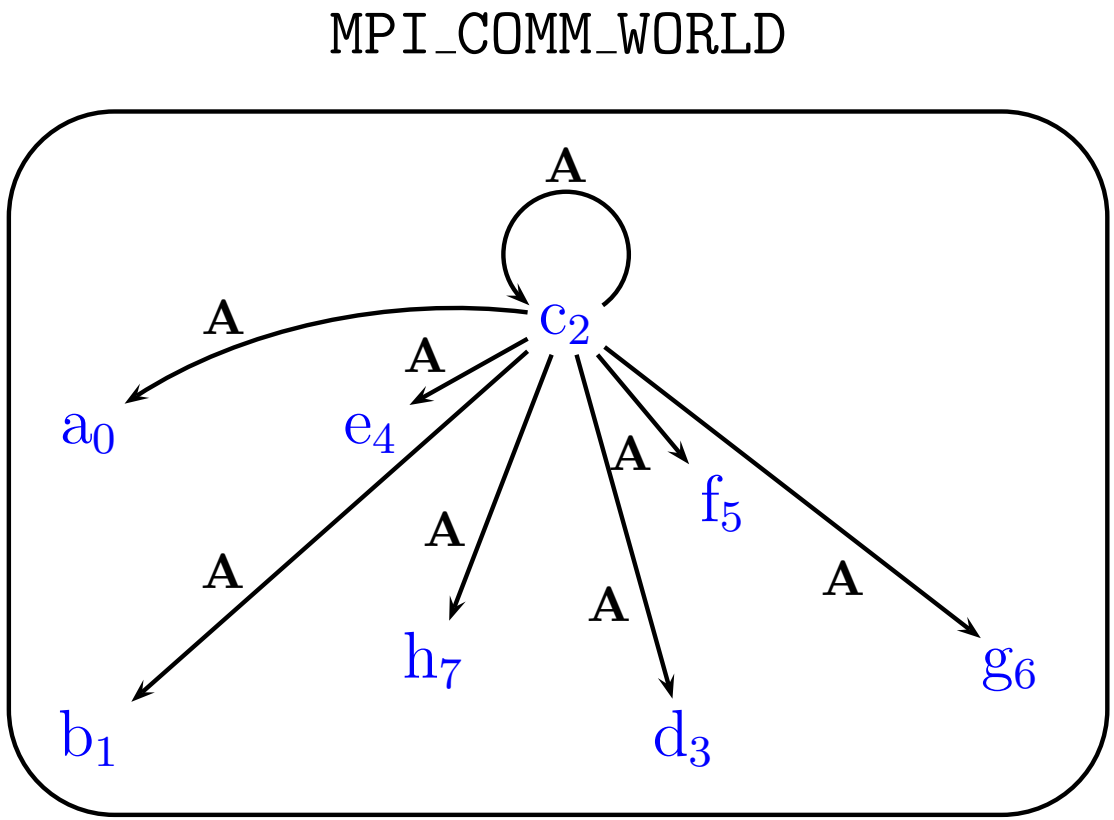


FIGURE 43 – Le communicateur par défaut

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.

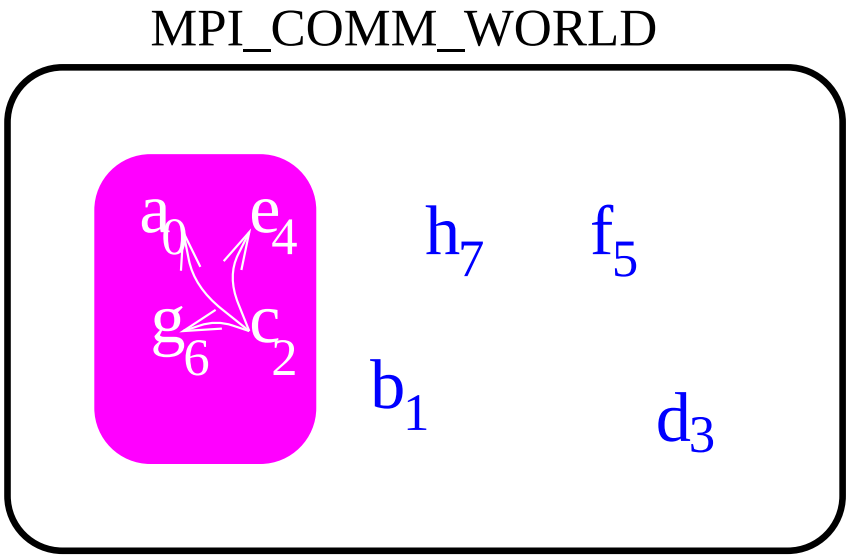


FIGURE 44 – Un nouveau communicateur

8.3 – Groupes et communicateurs

Dans la bibliothèque *MPI*, il existe des fonctions 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 de la procédure 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 $0,\ 1,\ \ldots,\ N-1$ appelé **rang**.
- Initialement, tous les processus sont membres d'un groupe associé au communicateur par défaut `MPI_COMM_WORLD`.
- Des fonctions *MPI* (`MPI_Group_size()`, `MPI_Group_rank()`, etc.) permettent de connaître les attributs d'un groupe.
- Tout communicateur est associé à un groupe. La fonction `MPI_Comm_group()` permet de connaître le groupe auquel un communicateur est associé.

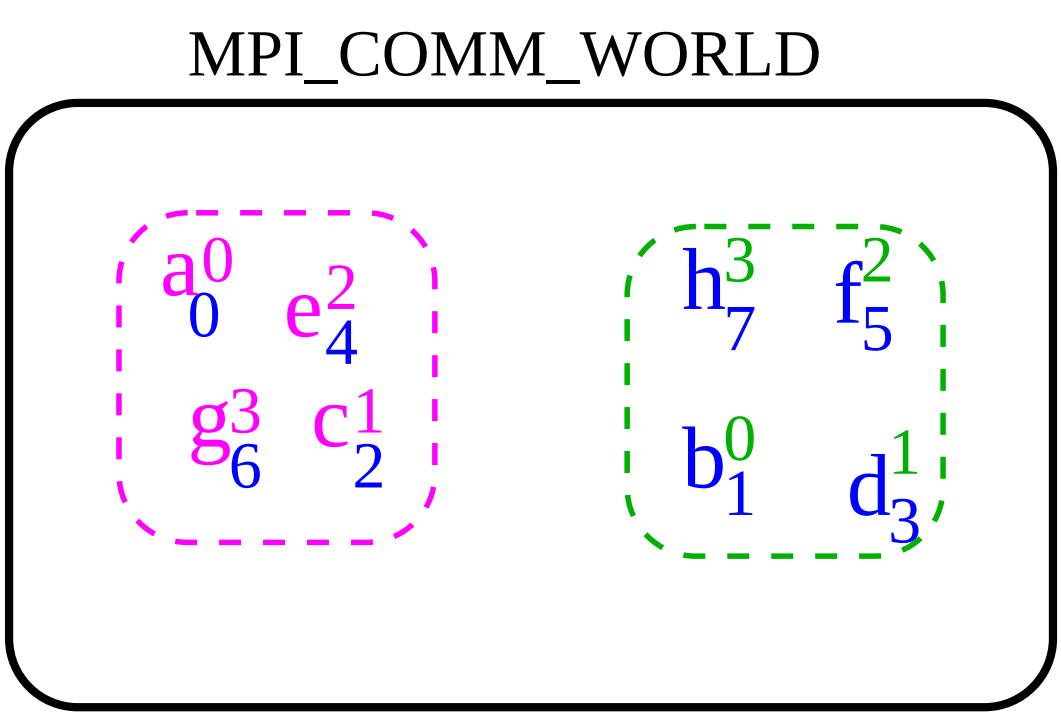


FIGURE 45 – Deux groupes de processus dans un communicateur

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.

```c
#include "mpi.h"
#include <stdlib.h>
#define M 16
int main( int argc, char *argv[])
{
 int i, rang, nb_procs, dim_rangs_pair, iproc;
 int rang_ds_monde, rang_ds_pair, rang_ds_impair;
 MPI_Group grp_monde, grp_pair, grp_impair;
 MPI_Comm comm_pair, comm_impair;
 int *rangs_pair;
 float a[M];

 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rang);
 MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);

 /* Initialisation du vecteur A */
 for (i=0; i<M; i++) a[i]=0.;
 if(rang == 2) for (i=0; i<M; i++) a[i]=2.;
 if(rang == 3) for (i=0; i<M; i++) a[i]=3.;

 /* Enregistrer le rang des processus pairs */
 dim_rangs_pair = (nb_procs+2)/2;
 rangs_pair=(int *) malloc(dim_rangs_pair*sizeof(int));
 for (iproc=0, i=0; iproc<nb_procs; iproc+=2)
   rangs_pair[i++] = iproc, i++;
```

```
27  /* Connaître le groupe associé au communicateur MPI_COMM_WORLD */
28  MPI_Comm_group(MPI_COMM_WORLD, &grp_monde);
29
30  /* Créer le groupe des processus pairs */
31  MPI_Group_incl(grp_monde, dim_rangs_pair, rangs_pair, &grp_pair);
32
33  /* Créer le communicateur des processus pairs */
34  MPI_Comm_create(MPI_COMM_WORLD, grp_pair, &comm_pair);
35  MPI_Group_free(&grp_pair);
36
37  /* Créer le groupe des processus impairs */
38  MPI_Group_excl(grp_monde, dim_rangs_pair, rangs_pair, &grp_impair);
39  MPI_Group_free(&grp_monde); free(rang_pair);
40
41  /* Créer le communicateur des processus impairs */
42  MPI_Comm_create(MPI_COMM_WORLD, grp_impair, &comm_impair);
43  MPI_Group_free(&grp_impair);
```

```
44  if (comm_pair != MPI_COMM_NULL) {
45    /* Trouver le rang du processus 2 dans le groupe pair */
46    rang_ds_monde=2;
47    MPI_Group_translate_ranks(grp_monde, 1, &rang_ds_monde, grp_pair, &rang_ds_pair);
48    /* Diffuser le message seulement aux processus de rangs pairs */
49    MPI_Bcast(a, M, MPI_FLOAT, rang_ds_pair, &comm_pair);
50    /* Destruction du communicateur comm_pair */
51    MPI_Comm_free(&comm_pair);
52  }
53  else if (comm_impair != MPI_COMM_NULL) {
54    /* Trouver le rang du processus 3 dans le groupe impair */
55    rang_ds_monde=3;
56    MPI_Group_translate_ranks(grp_monde, 1, &rang_ds_monde, grp_impair, &rang_ds_impair);
57    /* Diffuser le message seulement aux processus de rangs impairs */
58    MPI_Bcast(a, M, MPI_FLOAT, rang_ds_impair, comm_impair);
59    /* Destruction du communicateur comm_impair */
60    MPI_Comm_free(&comm_impair);
61  };
62  MPI_Finalize(); return(0);
63 }
```

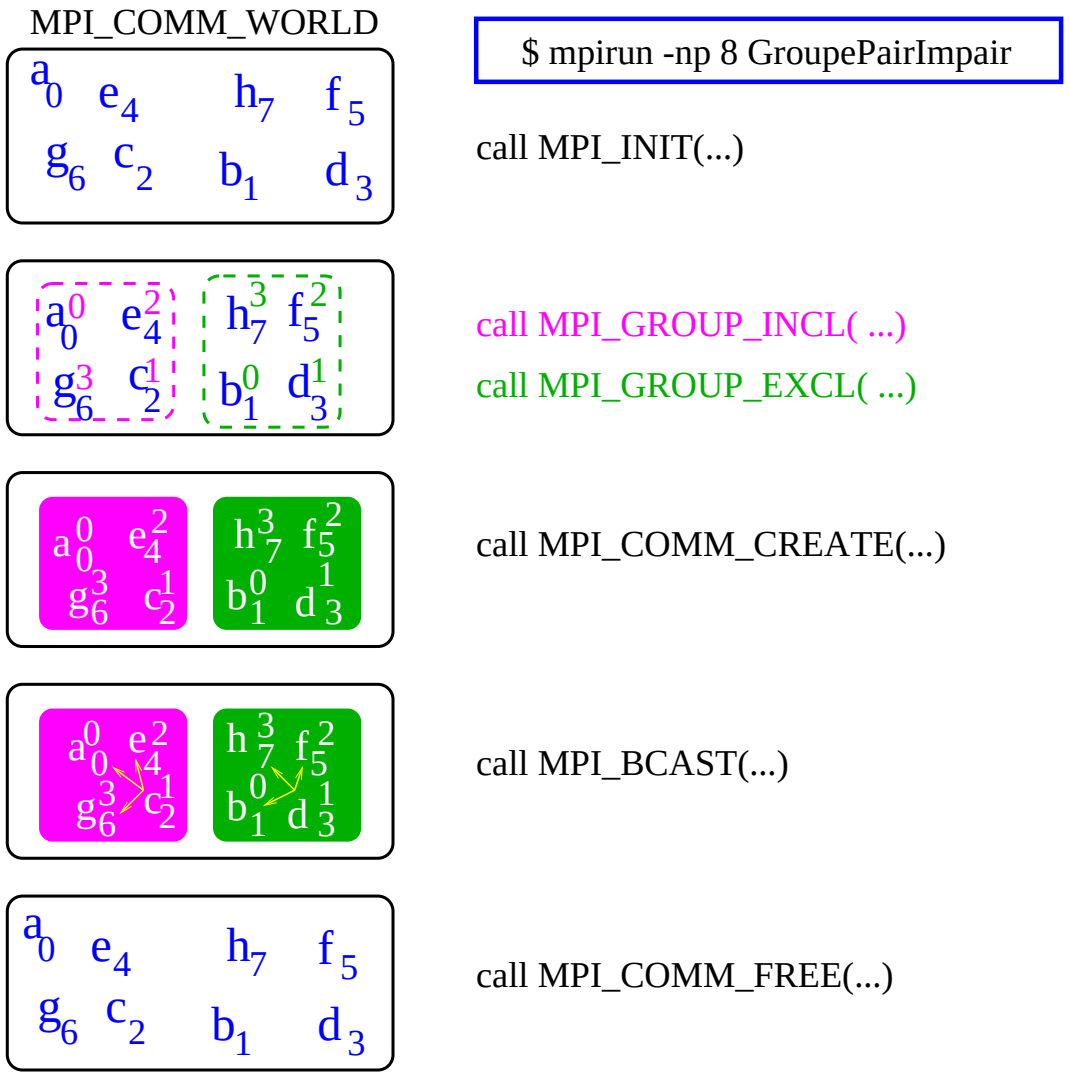


FIGURE 46 – Création/destruction des groupes et des communicateurs `pair` et `impair`

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 à la fonction `MPI_Bcast()`.

La fonction `MPI_Comm_split()` permet de partitionner un communicateur donné en autant de communicateurs que l'on veut...

```
int code, couleur, clef;
MPI_Comm comm, nouveau_comm;

code=MPI_Comm_split(comm, couleur, clef, &nouveau_comm);
```

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

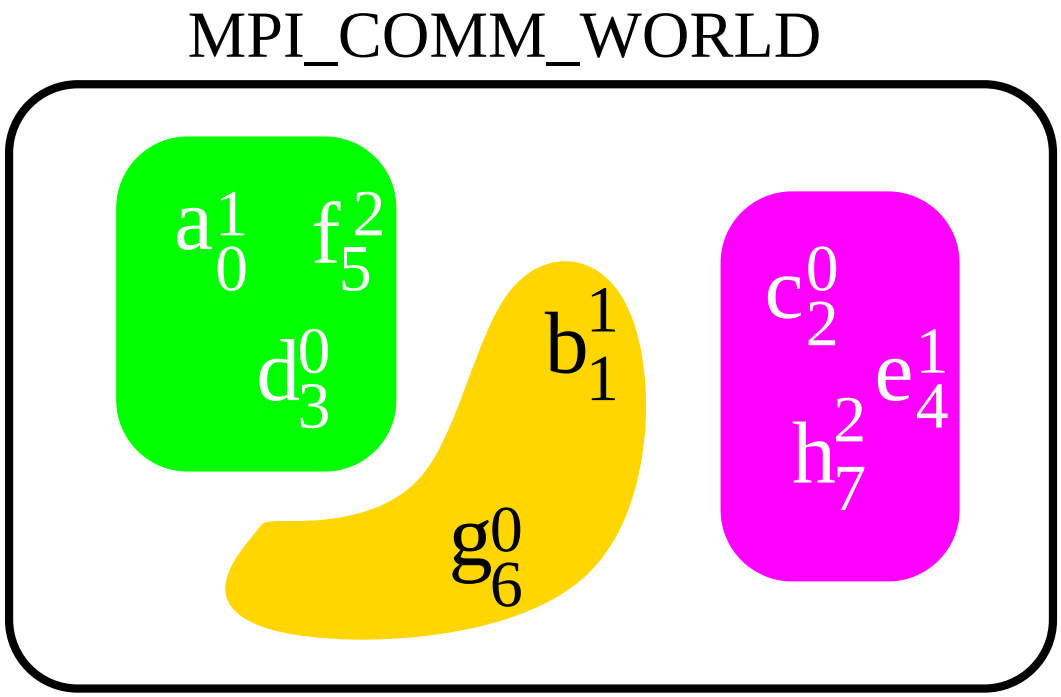


FIGURE 47 – Construction de communicateurs avec `MPI_Comm_split()`

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

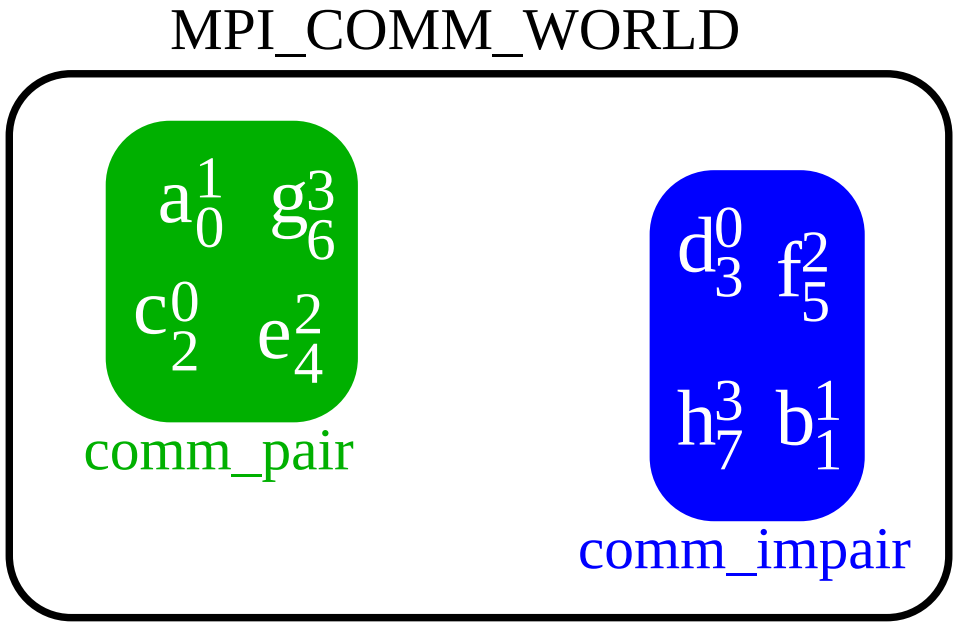


FIGURE 48 – Construction des communicateurs `pair` et `impair` avec `MPI_Comm_split()`

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

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #define M 16
4  int main( int argc, char *argv[])
5  {
6   int i, rang, nb_procs, clef, couleur;
7   MPI_Comm comm;
8   float a[M];
9
10  MPI_Init(&argc,&argv);
11  MPI_Comm_rank(MPI_COMM_WORLD, &rang);
12  MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
13
14  /* Initialisation du vecteur A */
15  for (i=0; i<M; i++) a[i]=0.;
16  if(rang == 2) for (i=0; i<M; i++) a[i]=2.;
17  if(rang == 3) for (i=0; i<M; i++) a[i]=3.;
```

```
18  if ( rang%2 == 0 ) {
19    /* Couleur et clefs des processus pairs */
20    couleur = 0;
21    if( rang == 2 ) clef = 0;
22    else if ( rang == 0 ) clef = 1;
23    else clef = rang;
24  }
25  else {
26    /* Couleur et clefs des processus impairs */
27    couleur = 1;
28    if( rang == 3) clef = 0;
29    else clef = rang;
30  }
31  /* Créer les communicateurs pair et impair en leur donnant une même dénomination */
32  MPI_Comm_split(MPI_COMM_WORLD, couleur, clef, &comm);
33  /* Chaque processus 0 d'un communicateur diffuse */
34  /* son message aux processus de son groupe       */
35   MPI_Bcast(a, M, MPI_FLOAT, 0, comm);
36  /* Destruction des communicateurs */
37  MPI_Comm_free(&comm);
38  MPI_Finalize(); return(0);
39 }
```

8.6 – 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).
- La fonction *MPI* `MPI_intercomm_create()` permet de construire des intercommunicateurs.
- Le couplage de procédures indépendantes illustre bien l'utilité des intra et intercommunicateurs...

8.7 – 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 une procédure d'une bibliothèque qui elle-même effectue des échanges de messages.
- Les communicateurs offrent une programmation modulaire du point de vue de l'espace de communication (exemple : couplage de procédures indépendantes).

8.8 – Exercice 9 : communicateurs

En partant de l'exercice 8 du chapitre précédent,

- ☞ construire un communicateur regroupant les processus situés sur la diagonale de la topologie cartésienne de profile carré (2x2) (utiliser la fonction `MPI_Comm_split()`) ;
- ☞ déterminer la valeur maximum globale des pixels situés sur la diagonale de l'`Image` distribuée ;
- ☞ pour une image dont le rapport de forme est toujours égal à 1, généralisez la distribution et l'opération précédentes au cas d'une topologie cartésienne dont le profile et la taille (nombre de processus) du communicateur associé sont quelconques.

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*)

TABLE 3 – Changements de dénominations dans *MPI*-1.2

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.

Index des constantes MPI