EISTI

Ecole Internationale des Sciences du Traitement de l'Information

Algorithme et programmation parallèle

Message Passing Interface (MPI)

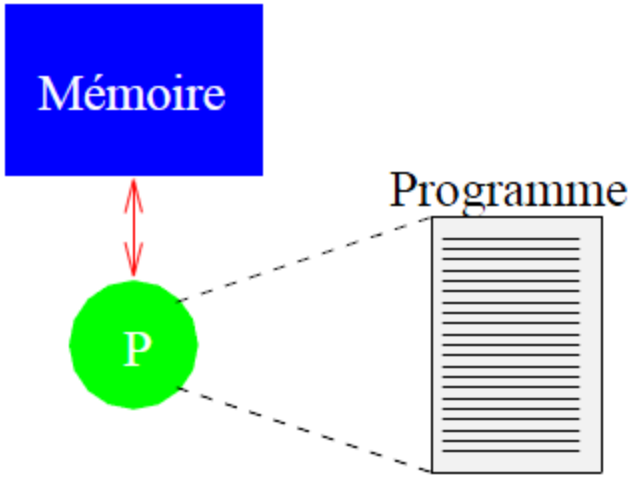
Avant Propos

Message Passing Interface Standard (MPI) est une bibliothèque standard de transmission de messages basée sur le consensus du Forum MPI, qui compte plus de 40 organisations participantes, y compris les fournisseurs, les chercheurs, les développeurs de bibliothèques logiciels, et des utilisateurs. Le but du Message Passing Interface est d'établir un standard portable , efficace et flexible pour la transmission de messages qui seront largement utilisés pour l'écriture de programmes par échange de messages.

Introduction

Définition : Modèle séquentiel

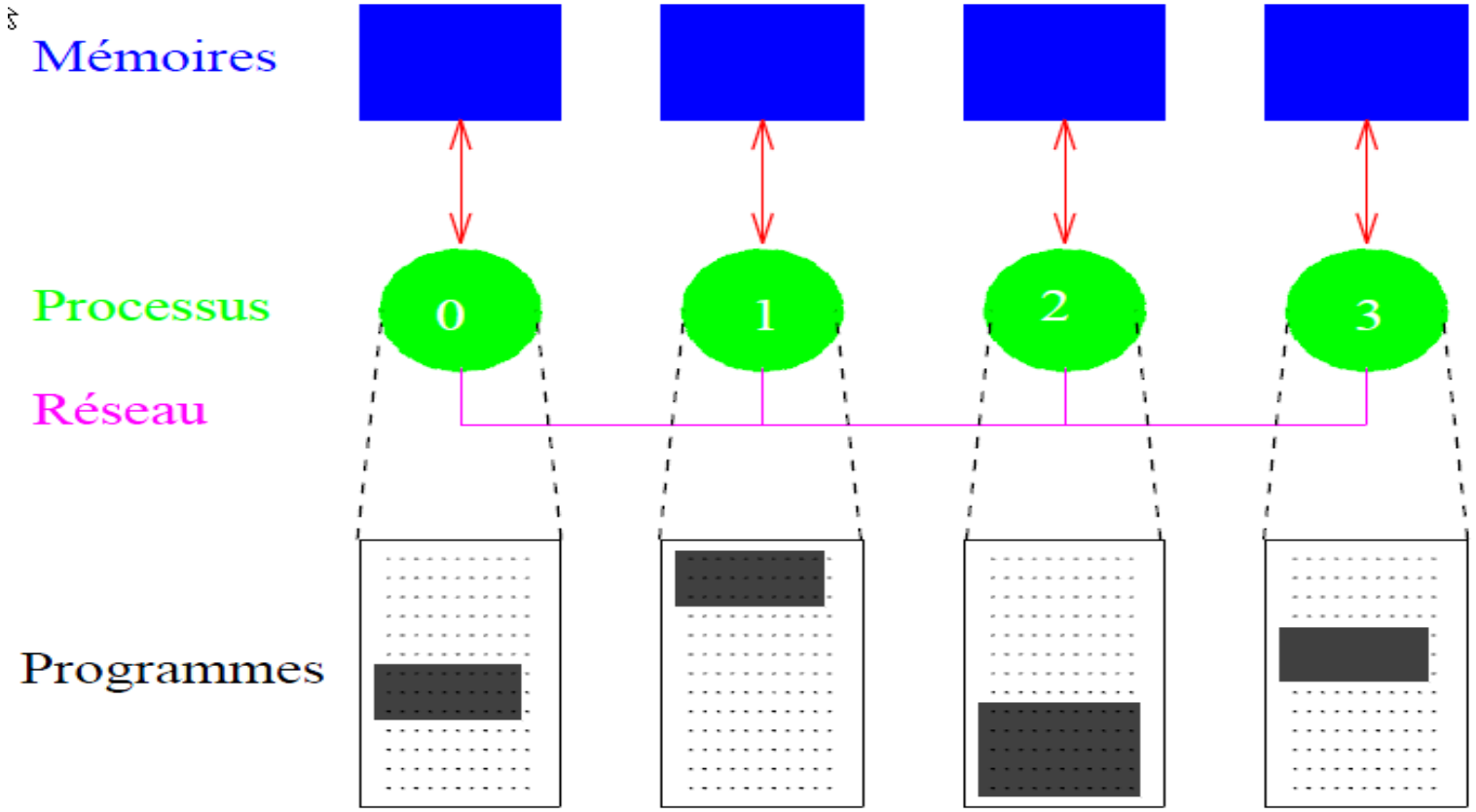
- Le modèle de programmation séquentiel :
  - le programme est exécuté par un et un seul processus ;
  - toutes les variables et constantes du programme sont allouées dans la mémoire centrale allouée au processus ;
  - un processus s'exécute sur un processeur physique de la machine.



Définition

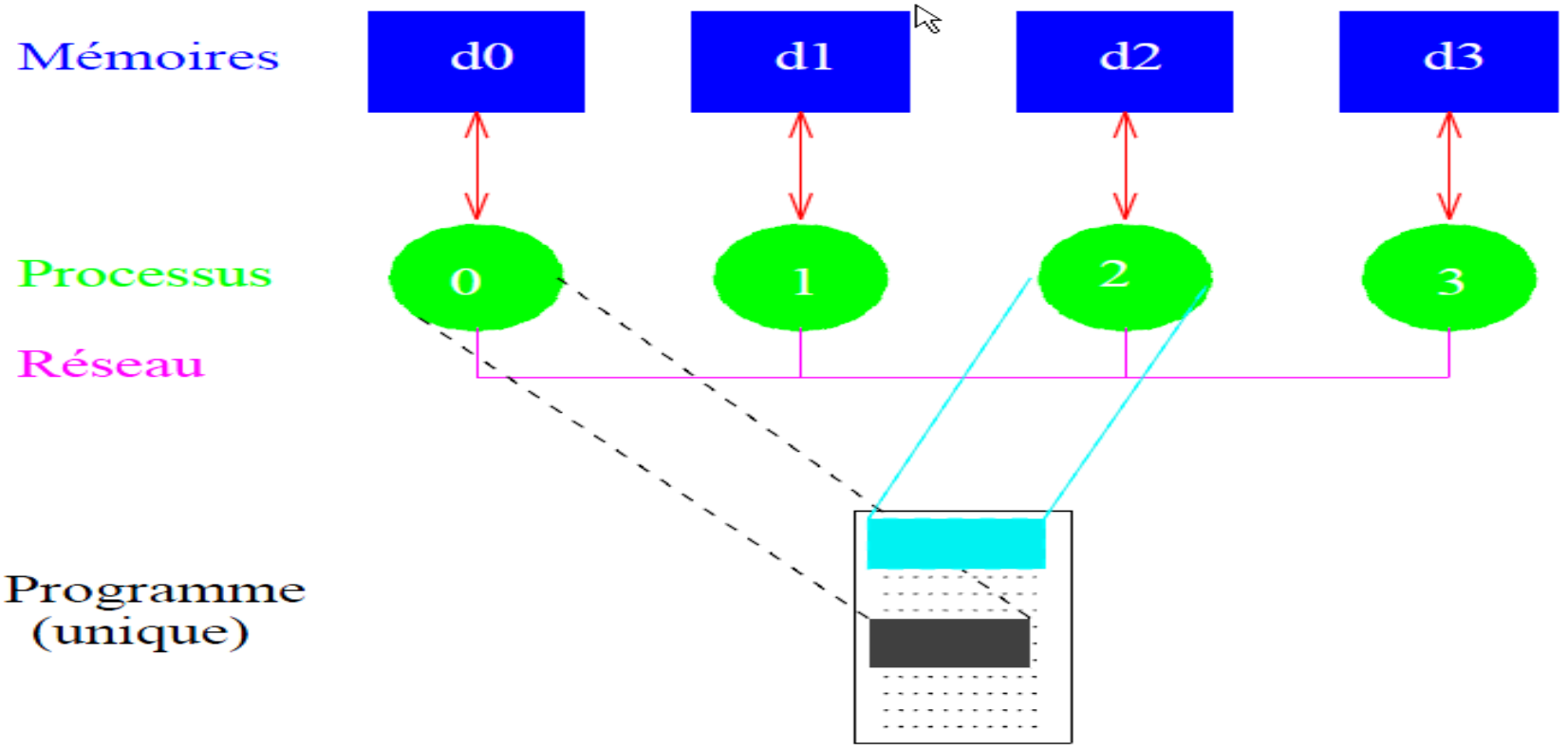
- Le modèle de programmation par échange de messages :
  - le programme est écrit dans un langage classique (Fortran, C ou C++) ;
  - chaque processus exécute éventuellement des parties différentes d'un programme ;
  - toutes les variables du programme sont privées et résident dans la mémoire locale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

Définition : Modèle à échange de messages



Définition : Le modèle d'exécution SPMD

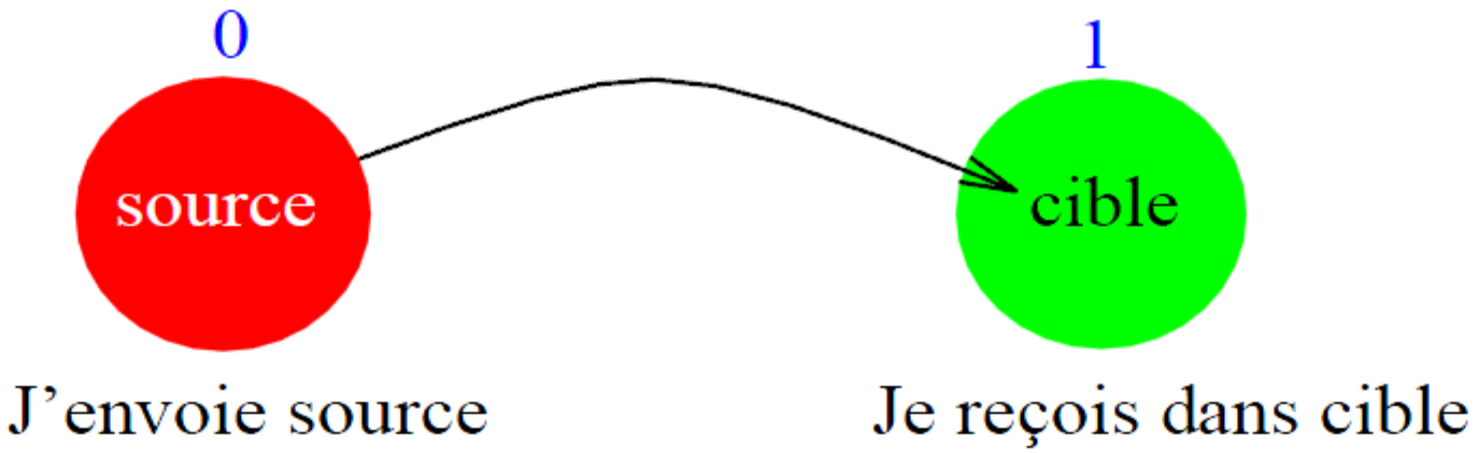
- Single Program, Multiple Data ;
  - le même programme est exécuté par tous les processus



Introduction

- Le but de ce tutoriel est de voir comment élaborer et exécuter des programmes parallèles selon la norme MPI.

- Concepts basés sur l'échange de messages



C’est quoi MPI?

•**M P I** = **M**essage **P**assing **I**nterface

•MPI est une spécification pour les développeurs et les utilisateurs des bibliothèques d’échange de messages. En soi, ce n'est pas une bibliothèque - mais plutôt la spécification de ce que cette bibliothèque devrait être.

•En termes simples, le but de la Message Passing Interface est de fournir un standard largement utilisé pour écrire des programmes d’échange de messages. L'interface se veut

- ✓pratique
- ✓portable
- ✓efficace
- ✓flexible

Raisons de l'utilisation MPI:

- Normalisation - MPI est la seule bibliothèque d'échange de message, qui peut être considérée comme une norme. Elle est supportée sur toutes les plateformes HPC. En pratique, il a remplacé toutes les anciennes bibliothèques d'échange de messages.
- Portabilité - Il n'est pas nécessaire de modifier le code source lorsque vous porter votre application sur une autre plate-forme qui supporte le standard MPI.
- Opportunité de rendement - implémentations des fournisseurs devraient être en mesure d'exploiter les fonctionnalités matérielles natives pour optimiser les performances. Pour plus d'informations sur les performances MPI voir le tutoriel sur les performance MPI.
- Fonctionnalité - Plus de 115 routines sont définies dans MPI-1 seul.
- Disponibilité - Une variété d'implémentations sont disponibles, à la fois commerciale et open source

Programming Model:

- A l'origine, MPI a été conçu pour les architectures à mémoire distribuée, qui devenaient de plus en plus populaire à cette époque.
- Comme les tendances d'architecture ont changé, les mémoires partagées ont été combinées sur de réseaux créant des systèmes hybrides mémoire distribuée / systèmes de mémoire partagée. Les fournisseur de MPI ont adapté leurs bibliothèques pour gérer les deux types d'architectures de mémoire sous-jacents de façon transparente.
- Aujourd'hui, MPI fonctionne sur pratiquement toute plate-forme matérielle:
  - Mémoire distribuée
  - Mémoire partagée
  - hybride

Structure d'un programme MPI

- Utilisation de la bibliothèque MPI :
  - Enrollement dans MPI
  - Quitter MPI proprement

```
...
#include "mpi.h"
   ...
   main (int argc, char *argv []) {
   ...
   MPI_Init (&argc, &argv) ;
   ...
   MPI_Finalize () ;
   ...
}
```

Environnement

Environnement MPI

- L'environnement MPI est initialisé lors de l'appel à la fonction MPI_INIT() et désactivé par l'appel à la fonction MPI_FINALIZE().Une fois cet environnement MPI initialisé, on dispose d'un ensemble de processus actifs et d'un espace de communication au sein duquel on va pouvoir effectuer des opérations MPI.
- Ce couple (processus actifs, espace de communication) est appelé communicateur.
- Le communicateur par défaut est MPI_COMM_WORLD et comprend tous les processus actifs.

Qui (rank)?

- Dans un communicateur, chaque processus a son propre identificateur unique, entier attribué par le système lors de l'initialisation du processus. Un rang est parfois aussi appelé «ID de la tâche". Les rangs sont contiguës et commencent à zéro.

- Utilisé par le programmeur pour spécifier la source et la destination des messages. Souvent utilisé conditionnellement par l'application pour contrôler l'exécution du programme (si rang = 0 faire / si rang = 1 faire).

- Qui suis-je ?

int MPI_Comm_rank (MPI_Comm comm, int *rank) ;

  - Communicateur : collection de processus pouvant communiquer
  - Communicateur MPI_COMM_WORLD prédéfini : tous les processus

Combien(size)?

- Combien sommes-nous ?

On peut connaître le nombre de processus actifs gérés par un communicateur avec la fonction int

int MPI_Comm_size (MPI_Comm comm, int *nb_procs) ;

- comm (< in >) est en entier désignant le communicateur,
- nb_procs (< out >) est en entier donnant le nombre de processus,
- rank (< out >) est un entier indiquant le rang du processus.

Gestion d’erreur

- La plupart des routines MPI retourne un code d'erreur.

- Toutefois, selon la norme MPI, le comportement par défaut de l'appel MPI est d'interrompre en cas d'erreur. Cela signifie que vous ne sera probablement pas en mesure de saisir un code retour d’erreur autre que MPI_SUCCESS (zéro).

- Les types d'erreurs affichés à l'utilisateur sont dépendant de l'implémentation.

Routines de gestion de l'environnement

- Routines gestion de l'environnement MPI sont utilisées pour différentes raisons, telles que l'initialisation et de terminaison de l'environnement MPI, l'interrogation de l'environnement et de l'identité, etc. La plupart de ceux couramment utilisées sont décrites ci-dessous.

- l'initialisation de l'environnement via l'appel à la fonction MPI_INIT(code). Cette fonction retourne une valeur dans la variable code. Si l'initialisation s'est bien passée, la valeur de code est égale à celle dans MPI_SUCCESS.

- La désactivation de l'environnement via l'appel à la fonction MPI_FINALIZE (code). L'oublie de cette fonction provoque une erreur.

Exemple 1 :Hello world distribué

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank;
  int nprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf("Hello, world.  I am %d of %d\n", rank, nprocs);
  fflush(stdout);
  MPI_Finalize();
  return 0;
}
```

Communication point à point

Communication point à point

- La communication point à point est une communication entre deux processus. L'un d'eux envoie un message (c'est l'émetteur), l'autre le reçoit (c'est le récepteur).
- Ce message doit contenir un certain nombre d'informations pour assurer une bonne réception et interprétation par le récepteur, à savoir :

Communication point à point (suite)

- Plusieurs modes de transfert sont possibles pour échanger des messages.
- On décrit ici des fonctions MPI de communication en mode bloquant, qui laisse la main une fois que le message est bien reçu. C'est le mode à utiliser quand on commence à paralléliser un code.
- On peut ensuite passer à un autre mode de transfert pour optimiser le temps de communication (détail plus tard).

Communication point à point (suite)

- 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 :
  - du rang du processus émetteur ;
  - du rang du processus récepteur ;
  - de l'étiquette (tag) du message ;
  - 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

Un message = données + enveloppe

- Fonctions de base d'émission (MPI_Send ()) et de réception (MPI_Recv ()) de messages
- Enveloppe : informations nécessaires
  - Rang du receveur (pour une émission)
  - Rang de l'émetteur (pour une réception)
  - Un tag (int)
    - ➔ Distinguer les messages d'un même couple émetteur/receveur
  - Un communicateur (MPI_Comm)
    - ➔ Distinguer les couples de processus

Envoi de messages

- *Emission d'un message (bloquant ou non bloquant)*

```
int MPI_Send(void* message,
             int count,
             MPI_Datatype datatype,
             int dest,  int tag, MPI_Comm comm);
```

MPI_Send (description)

- Message =
  - Enveloppe
    - *dest* : rang du processus destinataire
    - *tag* : étiquette (numéro de message pour différencier les messages d'un même couple émetteur/receveur).
    - *comm* : handler communicator
  - données :
    - *message* : tableau de type ``datatype ``, bloc de mémoire d'adresse message
    - *count* : taille du message
    - *MPI_Datatype :* type datatype

Réception de messages

Reception d'un message

Int MPI_Recv(void message, int count, MPI_Datatype*

datatype, int source, int tag,

*MPI_Comm comm, MPI_Status *status)*

- Attente de la réception d'un message de "source''. Elle est bloquante.
- "source'' peut avoir la valeur *MPI_ANY_SOURCE* (n'importe quelle source).
- '' tag'' peut avoir la valeur *MPI_ANY_TAG* (n'importe quelle étiquette).

MPI_Recv (description)

- *status* : identificateur de l'émetteur et du message reçu.

  status.MPI_SOURCE : source

  status.MPI_TAG : tag

  status.ERROR : erreur
- "*status*'' contient l'information sur la taille du message.
- Pour cela, on appelle la fonction :

  *int MPI_Get_count(MPI_Status *status,*
  *MPI_Datatype datatype, int *nbre)*

Exemple

```
#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 recu %d du processus 2.\n", valeur);
  }
  MPI_Finalize ();
  return(0);
}
```

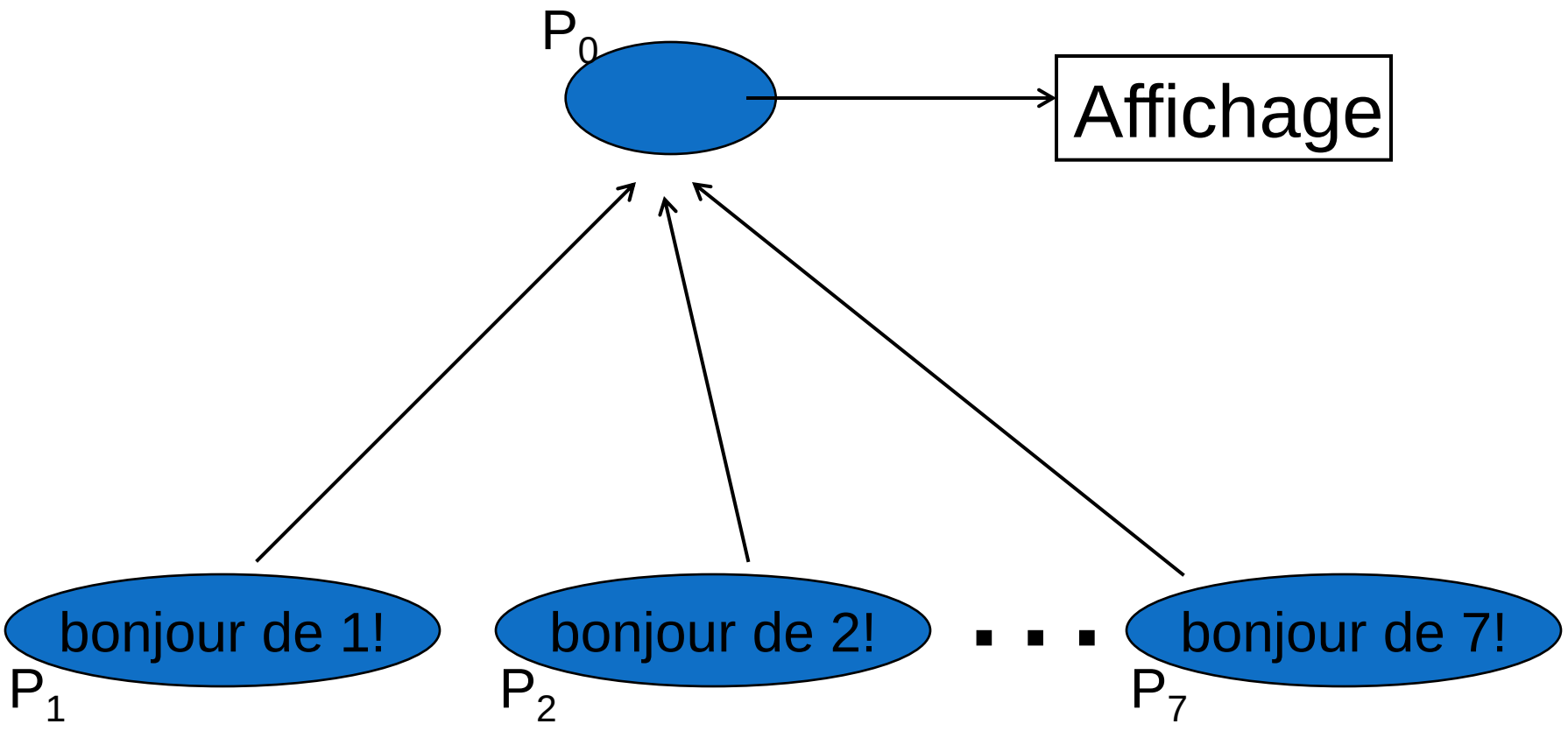
```
> mpirun -np 7 point_a_point
Moi, processus 5 , j'ai recu 1000 du processus 2
```

Ordre de réception des messages

- Si deux processus A et B envoient un message à un autre processus C, l'ordre d'arrivé des messages n'est pas déterminé.

- Si un processus envoie un message m1 à C ensuite envoie un message m2 à C, alors m1 sera reçu avant m2.

Exemple



Hello world (initialisation)

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
void  main(int argc, char* argv[]) {
   int      my_rank;          /* rang du processus          */
   int      p;                /* nombre de processus        */
   int      source;           /* rang de l'émetteur         */
   int      dest;             /* rang du récepteur          */
   int      tag = 0;          /* étiquette                  */
   char     message[100];     /* pour stocker le message */
   MPI_Status  status;        /* status du message          */

    MPI_Init(&argc, &argv);  /* Démarrer MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

Hello world (corps)

```
if (my_rank != 0) { /* si je ne suis pas le processus racine*/
  /Creation du message  */
  sprintf(message, "Bonjour de  %d!", my_rank);
  dest = 0;
   /* Utiliser strlen+1 pour que '\0' soit transmis! */
  MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   }
 else { /* my_rank == 0 */
       for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag,  MPI_COMM_WORLD, &status);
        printf("%s\n", message);
       }
   }
   MPI_Finalize(); /* Quitter MPI */
} /* fin de la fonction main */
```

Constantes : Type MPI_Datatype

- *datatype* : type des données. Les types prédéfinies sont

MPI_CHAR	MPI_UNSIGNED_CHAR	MPI_FLOAT
MPI_SHORT	MPI_UNSIGNED_SHORT	MPI_DOUBLE
MPI_INT	MPI_UNSIGNED	MPI_LONG_DOUBLE
MPI_LONG	MPI_UNSIGNED_LONG	**MPI_PACKED/MPI_BYTE**

- Correspondance des MPI_Datatype avec les types du C :

MPI_CHAR	signed char		MPI_SHORT	signed short int
MPI_INT	signed int		MPI_LONG	signed long int
……			……..	
MPI_FLOAT	float		MPI_DOUBLE	double
MPI_LONG_DOUBLE	Long double		MPI_PACKED	<<struct>>

- Types particuliers :
  - MPI_BYTE Pas de conversion
  - MPI_PACKED Types construits (*cf. infra)*

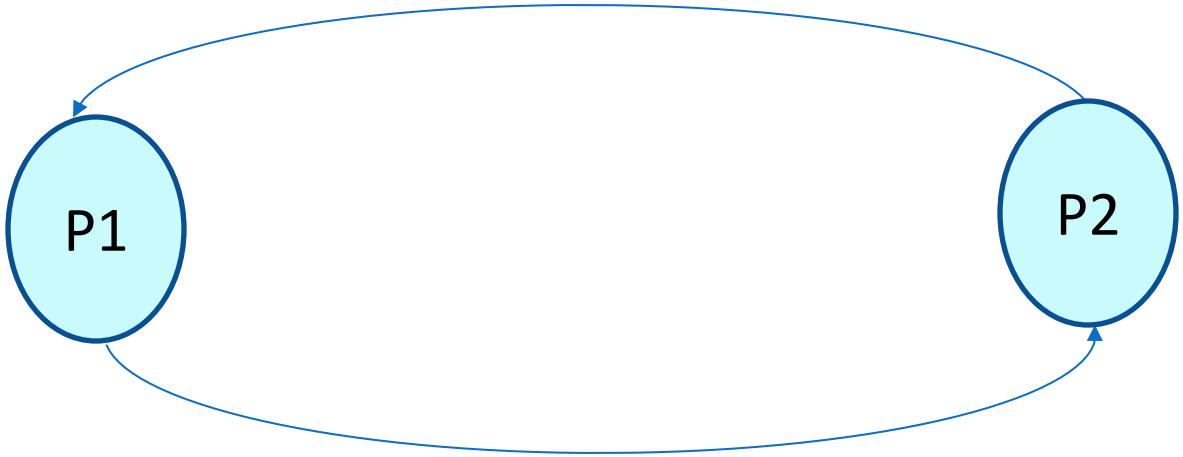
Constantes : Jokers

- Message reçu d'un processus source quelconque
  - Joker pour source : MPI_ANY_SOURCE
- On peut communiquer avec un processus fictif de rang
  - MPI_PROC_NULL.
- Message reçu de tag quelconque
  - Joker : MPI_ANY_TAG
- MPI_STATUS_IGNORE le status ne nous intéresse pas
- Accès au tag et à l'émetteur : status dans mpi.h :

```
typedef struct {
   int MPI_SOURCE;
   int MPI_TAG;
} MPI_Status;
```

Autre possibilité de communication PaP

- Il existe des variantes syntaxiques, MPI_Sendrecv() etMPI_Sendrecv_replace(), qui enchaînent un envoi et une réception.



MPI_Sendrecv

MPI_Sendrecv(val_emis, taille_emis, type_emis, rang_proc_dest, tag_emis, val_recu, taille_recu, type_recu, rang_proc_source, tag_recu, comm, statut)

Pour l'envoi et la réception de messages. Attention, si on utilise la même variable pour l'envoi et la réception (val_emis = val_recu), il y a écrasement.

rang_proc_source (< in >) est un entier indiquant le rang du processus qui a émis le message

MPI_Sendrecv_Replace

MPI_Sendrecv_Replace(val_emis_recu, taille_emis_recu, type_emis_recu, rang_proc_dest, tag_emis, rang_proc_source, tag_recu, comm, statut)

Pour l'envoi et la reception de messages en utilisant le même variable val_emis_recu pour l'envoi et la réception.

Cette fois-ci il n'y a pas d'écrasement.

Exemple de programme sendrecv.c

```
> mpirun -np 2 sendrecv
processus 1 , envoye 1001 au processus 0 et recoit 1000 du processus 0
processus 0 , envoye 1000 au processus 1 et recoit 1001 du processus 1
```

```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#define ETIQUETTE 100
int main(int argc, char *argv[])  {
    int rang, valeur, x, num_proc;
    MPI_Status statut;
    MPI_Init (&argc,&argv);
    MPI_Comm_rank ( MPI_COMM_WORLD , &rang);
    /* On suppose avoir exactement 2 processus */
    num_proc=(rang+1)%2;
    valeur=rang+1000;
    MPI_Sendrecv (&valeur, 1, MPI_INT , num_proc, ETIQUETTE, &x, 1,
                        MPI_INT , num_proc, ETIQUETTE, MPI_COMM_WORLD , &statut);
    /* MPI_Sendrecv_replace(&valeur, 1, MPI_INT, num_proc, ETIQUETTE,
                                    num_proc, ETIQUETTE, MPI_COMM_WORLD, &statut); */
    printf("processus %d, envoye %d au processus % et recoit %d du processus %d.\n", rang,
    valeur, num_proc, x, num_proc);
    MPI_Finalize ();
    return(0);
}
```

Mesure du temps de communication

```
...
//traitement sequentiel par le processus principal
MPI_Init (&argc,&argv);
double temps_debut= MPI_Wtime ();

...///traitement parallèle

double temps_fin= MPI_Wtime ();
MPI_Finalize ();
……///traitement sequentiel processus principal
printf("... en %8.6f secondes.",temps_fin-temps_debut);
...
```

Envoi/réception non bloquant

- *int MPI_Isend(void* buf, int count,*
  MPI_Datatype datatype,
  int dest, int tag,
  MPI_Comm comm,
  *MPI_Request *request)*

- *int MPI_Irecv(void* buf,*
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  *MPI_Request *request)*

Compilation et exécution

- Pour compiler les programmes utilisant la bibliothèque Open MPI, il suffit d'utiliser le compilateur fourni par Open MPI : *mpicc. Un exemple est fourni* ci-dessous . Les paramètres sont similaires à ceux de *gcc.*

mpicc hello_c.c -o hello_exe

- L'exécution de programme MPI est réalisée à l'aide du programme *mpirun. Un* exemple est fourni ci-dessous.

mpirun -hostfile hosts -np 3 hello_exe

- Les principaux paramètres de *mpirun sont présentés ci-dessous.*
  - -np <#> : indique le nombre de processus a utiliser
  - -hostfile <path> : indique le chemin vers le fichier contenant les liste des hotes
  - -nolocal : indique a MPI de ne pas lancer de processus sur la machine locale

Communications collectives

Introduction

- Elles permettent de communiquer en un seul appel avec tous les processus d'un communicateur. Ce sont des fonctions bloquantes, c'est à dire que le système ne rend la main à un processus qu'une fois qu'il a terminé la tache collective.

- Pour ce type de communication, les étiquettes sont automatiquement gérées par le système.

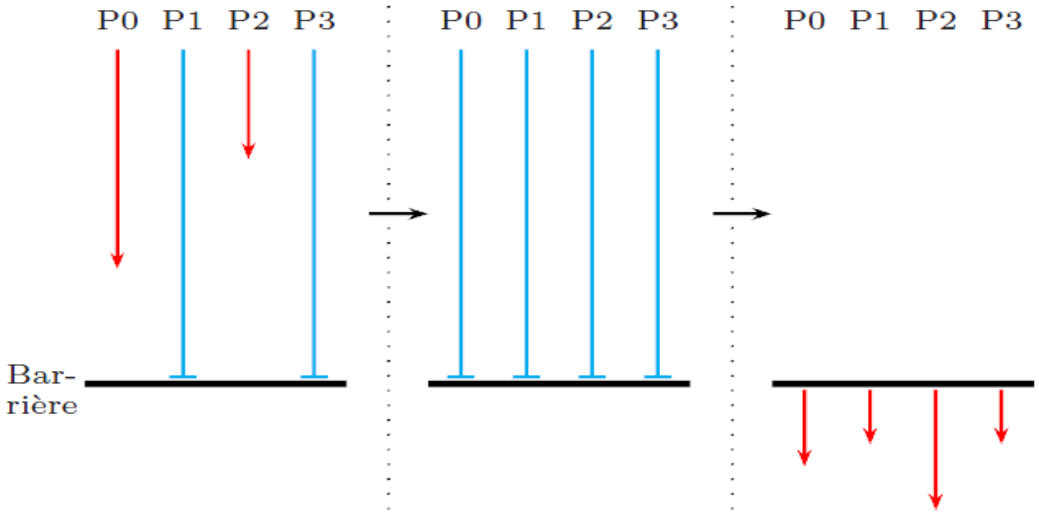
- On détaille quelques fonctions de communication collective.

Notions générales

- Il y a trois types de fonctions :
  - celui qui assure les synchronisations globales : **MPI_Barrier().**
  - 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().**
  - 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()**.

Synchronisation globale : *MPI_Barrier()*

- *int MPI_Barrier(MPI_Comm comm)*




```
int code;
MPI_Comm comm;

code= MPI_Barrier (comm);
```

  - **Synchronisation ou rendez-vous**
  - Pas d'échange d'informations
  - Tous les processus sont assurés que tous ont ralliés le point de synchronisation

Diffusion générale : MPI_Bcast()

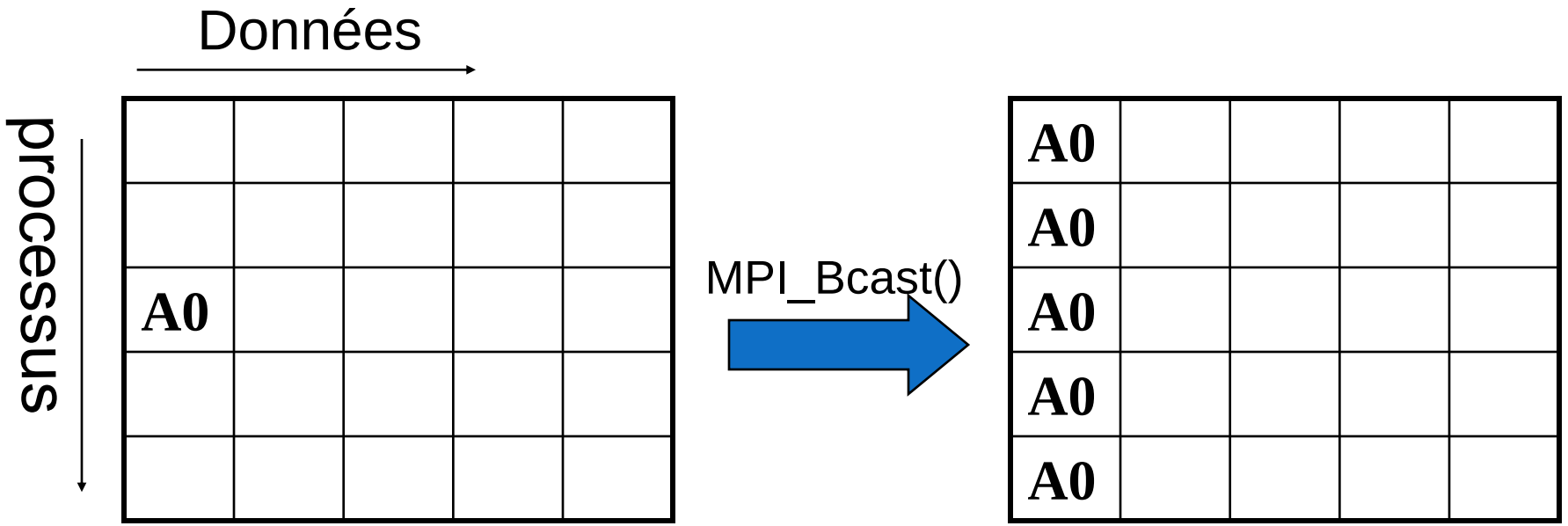
- int MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

- Appel par tous les processus du communicateur *comm* avec la même valeur de *root*, *count*, *datatype*.

- La valeur détenue par le processus root sera émise et rangée chez chacun des autres processus.

Diffusion générale suite: MPI_Bcast()

- Un même processus P2 envoie une même valeur à tous les processus d'un communicateur y compris à lui-même .



Exemple de code source

```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rang, valeur;

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

     if (rang == 2) valeur=rang+1000;
    MPI_Bcast (&valeur,1, MPI_INT , 2, MPI_COMM_WORLD );

    printf("Moi, processus %d, j'ai recu %d du processus 2\n",rang, valeur);
    MPI_Finalize ();
    return(0);
}
```

```
> mpirun -np 4 bcast
Moi, processus 2 , j'ai recu 1002 du processus 2
Moi, processus 0 , j'ai recu 1002 du processus 2
Moi, processus 1 , j'ai recu 1002 du processus 2
Moi, processus 3 , j'ai recu 1002 du processus 2
```

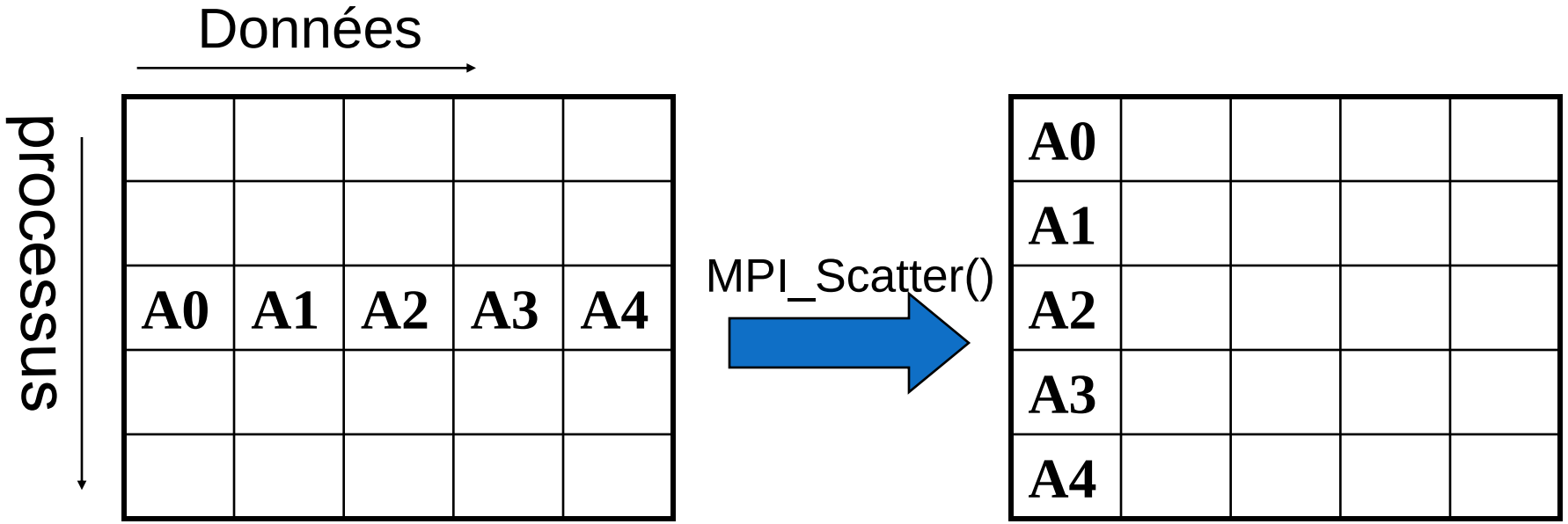
Diffusion sélective : MPI_Scatter()

- int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype , int root, MPI_Comm comm)

- Cette fonction permet à un processus de diffuser des données aux processus du communicateur indiqué de façon sélective. En fait le processus émetteur dispose de données qu'il répartit. Chaque processus (émetteur même compris) reçoit un paquet de données différent

Diffusion sélective suite : MPI_Scatter()

- Le processus P2 diffuse ses données A0,..., A3 (stockées dans sa mémoire) à tous les processus P0 à P4 de façon répartie.
- Distribution d'un message personnalisé aux autres processus (One-to-All)



Exemple de code source

```
#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;
      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;
      donnees=(float*)malloc(longueur_tranche*sizeof(float));
      if (rang == 2) {
            valeurs=(float*) malloc(NB_VALEURS*sizeof(float));
            for (i=0; i<NB_VALEURS; i++) valeurs[i]=(float)(1000+i);
      }
      MPI_Scatter (valeurs,longueur_tranche, MPI_FLOAT ,donnees,longueur_tranche,
      MPI_FLOAT ,2, MPI_COMM_WORLD );
      printf("Moi, processus %d, j'ai recu %.0f à %.0f du processus 2\n",
            rang, donnees[0], donnees[longueur_tranche-1]);
       if (rang == 2) free(valeurs); free(donnees); MPI_Finalize (); return(0);
}
```

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

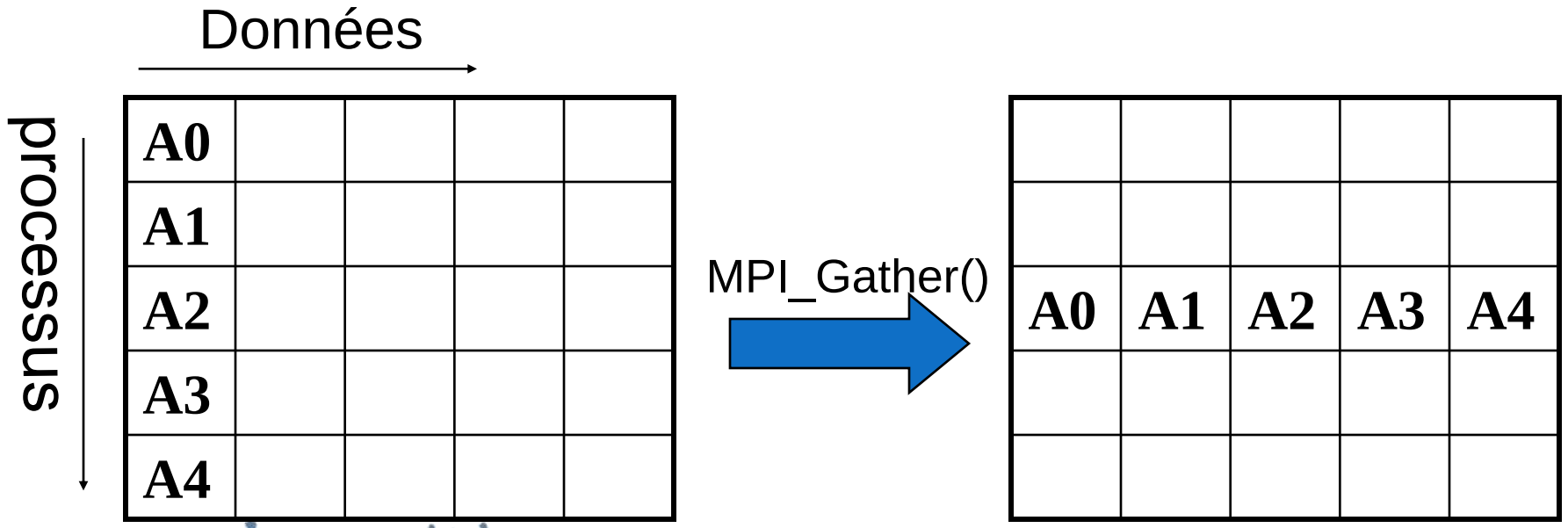
Collecte : MPI_Gather()

- int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,int recvcount, MPI_Datatype recvtype , int root, MPI_Comm comm)

- Cette fonction permet au processus récepteur de collecter les données provenant de tous les processus (lui-même compris). Attention, le résultat n'est connu que par le processus récepteur

Collecte suite : MPI_Gather()

- Le processus P2 collecte les données A0,..., A4 provenant des processus P0 à P4
- La mise bout à bout des messages de chacun des processus (All-to-one).



Exemple de code source

```
#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_Gather (valeurs,longueur_tranche, MPI_FLOAT , donnees,longueur_tranche,
      MPI_FLOAT ,2, MPI_COMM_WORLD );

      if (rang == 2) printf("Moi, processus 2, j'ai recu %.0f, ..., %.0f, ..., %.0f\n",
                        donnees[0], donnees[longueur_tranche], donnees[NB_VALEURS-1]);
      free(valeurs); MPI_Finalize (); return(0);
}
```

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

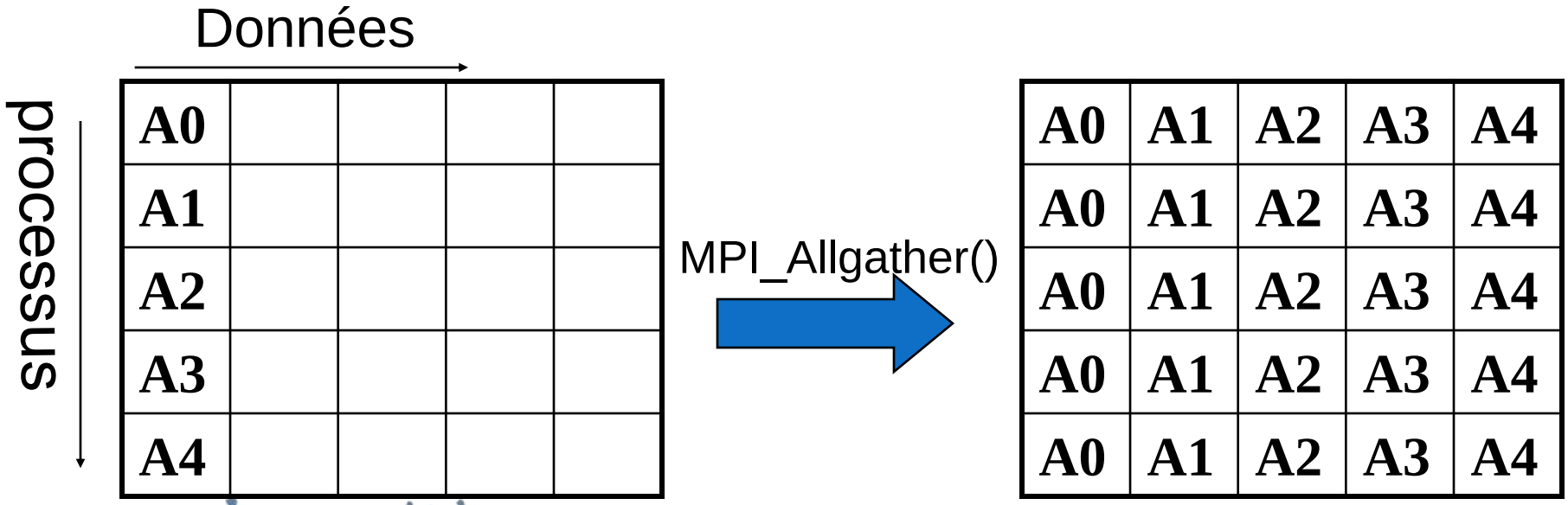
Collecte générale : MPI_Allgather()

- int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype , int root, MPI_Comm comm)

- Cette fonction effectue la même chose que la fonction MPI_Gather, excepté que le résultat de la collecte est connu de tous les processus du communicateur. Ce serait l'équivalent d'un MPI_Gather suivi d'un MPI_Bcast.

Collecte générale suite : MPI_Allgather()

- Le processus P2 collecte les données A0,..., A4 provenant des processus P0 à P4 et les diffuse à tous les processus.
- La mise bout à bout des messages de chacun des processus (All-to-All).



Exemple de code source

```
#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 recu %.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 recu 1000. ... 1032. ... 1127.
Moi, processus 3 , j'ai recu 1000. ... 1032. ... 1127.
Moi, processus 2 , j'ai recu 1000. ... 1032. ... 1127.
Moi, processus 0 , j'ai recu 1000. ... 1032. ... 1127.
```

Echanges croisés : MPI Alltoall()

- int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

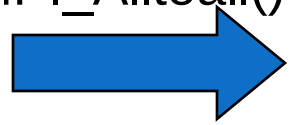
Echanges croisés suite : MPI Alltoall()

Données

processus

A0	A1	A2	A3	A4
B0	B1	B2	B3	B4
C0	C1	C2	C3	C4
D0	D1	D2	D3	D4
E0	E1	E2	E3	E4

MPI_Alltoall()



A0	B0	C0	D0	E0
A1	B1	C1	D1	E1
A2	B2	C2	D2	E2
A3	B3	C3	D3	E3
A4	B4	C4	D4	E4

Exemple de code source

```
#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[NB_VALEURS], donnees[NB_VALEURS];

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

      for (i=0; i<NB_VALEURS; i++)
            valeurs[i]=(float)(1000+rang*NB_VALEURS+i);
      longueur_tranche=NB_VALEURS/nb_procs;

      MPI_Alltoall (valeurs,longueur_tranche, MPI_FLOAT ,donnees,longueur_tranche,
      MPI_FLOAT , MPI_COMM_WORLD );
      printf("Moi, processus %d, j'ai recu %.0f, ..., %.0f, ..., %.0f\n", rang, donnees[0],
            donnees[longueur_tranche], donnees[NB_VALEURS-1]);
      MPI_Finalize (); return(0);
}
```

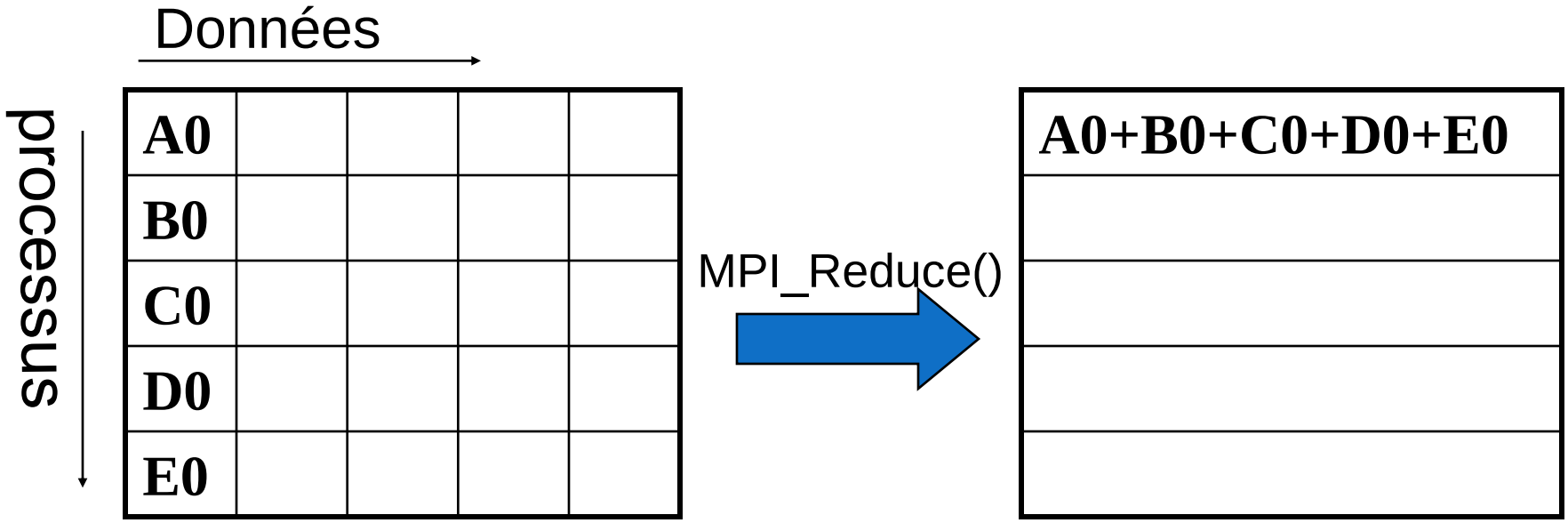
```
> mpirun -np 4 alltoall
Moi, processus 0 , j'ai recu 1000. ... 1128. ... 1415.
Moi, processus 2 , j'ai recu 1064. ... 1192. ... 1479.
Moi, processus 1 , j'ai recu 1032. ... 1160. ... 1447.
Moi, processus 3 , j'ai recu 1096. ... 1224. ... 1511.
```

Reduction : MPI_Reduce()

- int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op operation, int root, MPI_Comm comm)
- Collecte et réduction par un processus d'un ensemble de valeurs détenues par tous les processus.
- Une réduction consiste à appliquer une opération à un ensemble d’éléments pour obtenir un scalaire. Cela peut être la somme des éléments d’un vecteur ou la valeur maximale d’un vecteur

Reduction suite : MPI_Reduce()

- Le processus P1 collecte les données A0,..., A4 provenant des processus P0 à P3 et fait une opération de réduction.



Exemple de code source

```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>

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

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

       if (rang == 0)  valeur=1000;
       else  valeur=rang;

      MPI_Reduce (&valeur,&somme,1, MPI_INT , MPI_SUM ,0, MPI_COMM_WORLD );

      if (rang == 0)
      printf("Moi, processus 0, j'ai pour valeur de la somme globale %d\n",somme);
      MPI_Finalize (); return(0);
}
```

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

Réduction suite

- Appel par tous les processus du communicateur *comm* avec une même valeur de *count*, *datatype*, *MPI_OP*.
- Seul le processus root détient le résultat.

Constantes : les opérations MPI (MPI_OP)

- Les opérations MPI supportées sont les suivantes :
  - MPI_MAX : Maximum
  - MPI_MIN : Minimum
  - MPI_SUM : Somme
  - MPI_PROD : Produit
  - MPI_LAND : ET logique (AND)
  - MPI_BAND : ET bit à bit (AND)
  - MPI_LOR : OU logique (OR)
  - MPI_BOR : OU bit à bit (OR)
  - MPI_LXOR : OU Exclusif logique (XOR)
  - MPI_BXOR : OU Exclusif bit à bit (XOR)

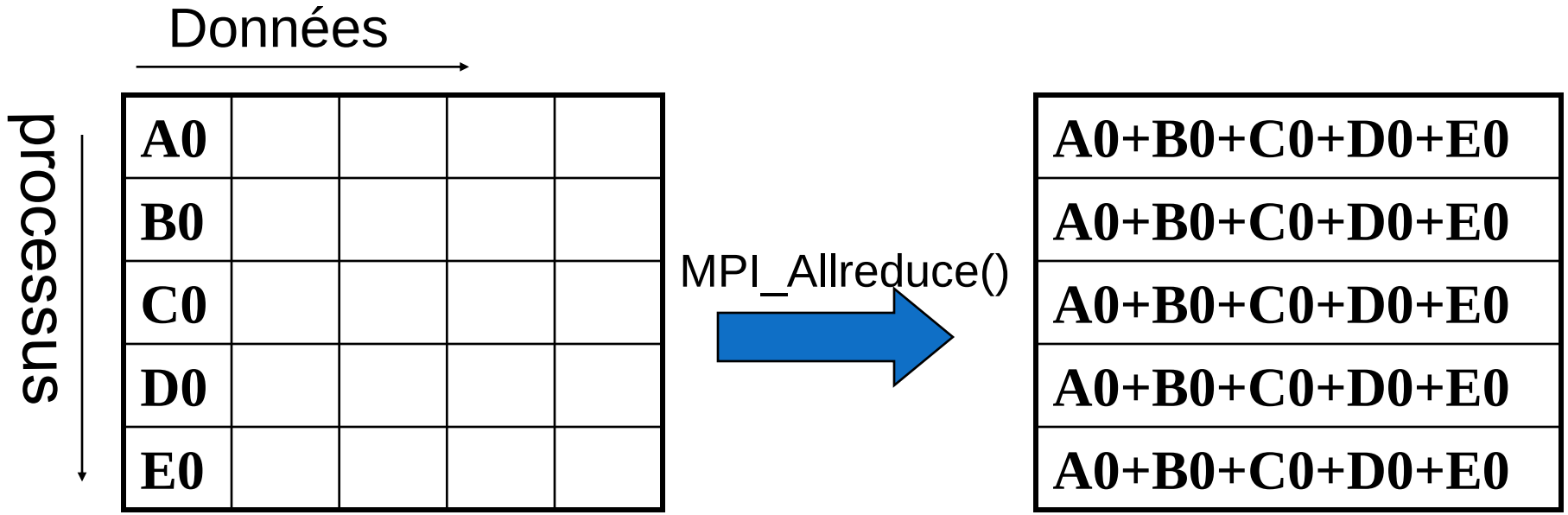
Réduction généralisée : MPI_Allreduce()

- int MPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op operation, MPI_Comm comm)

- Permet de faire les mêmes opérations de réduction que MPI_Reduce(). La différence est que le résultat de l'opération de réduction est connu de tous les processus d'un même communicateur.

Reduction généralisée suite : MPI_Allreduce()

- Opération de réduction et diffusion du résultat à l'ensemble des processus.



Exemple de code source

```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>

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

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

       if (rang == 0)  valeur=1000;
       else  valeur=rang;

      MPI_Allreduce (&valeur,&somme,1, MPI_INT , MPI_SUM , MPI_COMM_WORLD );

      printf("Moi, processus %d, j'ai pour valeur de la somme globale %d\n", rang, somme);
      MPI_Finalize (); return(0);
}
```

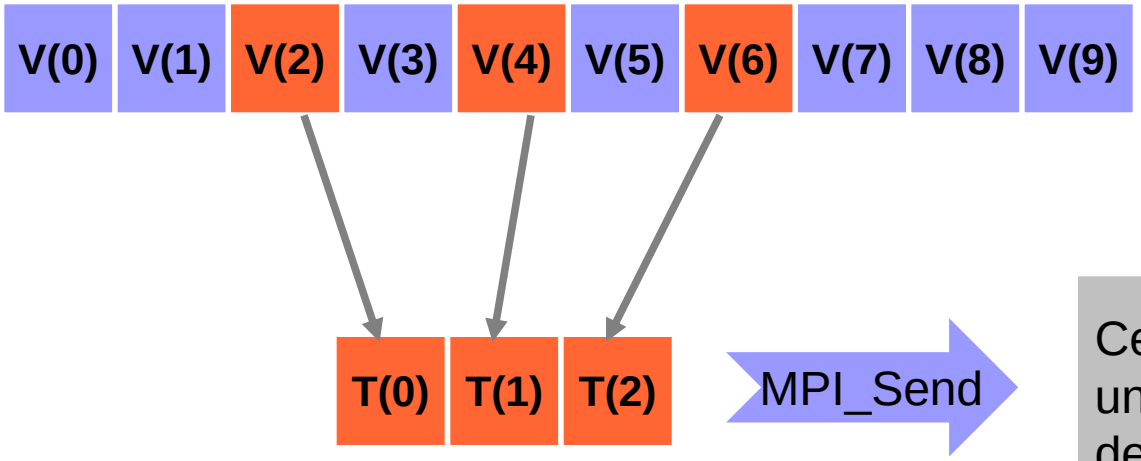
- mpirun -np 4 allreduce

```
Moi, processus  1 , j'ai pour valeur de la somme globale 1006
Moi, processus  0 , j'ai pour valeur de la somme globale 1006
Moi, processus  2, j'ai pour valeur de la somme globale 1006
Moi, processus  3, j'ai pour valeur de la somme globale 1006
```

Datatypes Dérivés

Datatypes dérivés

- Comment envoyer uniquement les éléments rouges de V en une seule communication?



Cette méthode nécessite une copie efficace des données non contiguës

Une possibilité, copier ces éléments dans un tableau temporaire avant de l'envoyer.

Types de données dérivés Non-struct

- Il y a des routines disponibles dans la bibliothèque MPI qui sont plus appropriés pour un tableau ou un vecteur comme des structures de données:
  - **`MPI_Type_contiguous`**
  - **`MPI_Type_vector`**
  - **`MPI_Type_indexed`**

- Toutes les fonctions ci-dessus fonctionnent avec un seul type de données!

MPI_Type_contiguous

- Construit un type consistant à la réplication d'un type de données dans des emplacements continus.

```
int MPI_Type_contiguous( int count,
    MPI_Datatype old_type,
    MPI_Datatype *newtype)
```

Vieux type

Nouveau type

constructeur de données avec **`MPI_Type_contiguous`**

MPI_Type_contiguous

- Si nous créons une matrice avec l'allocation de mémoire statique, nous pouvons dire que les données de la matrice seront en mémoire contiguë.

```
double A[3][3];
```

A (0,0)	A (0,1)	A (0,2)
A (1,0)	A (1,1)	A (1,2)
A (2,0)	A (2,1)	A (2,2)

Dans la mémoire

A (0,0)	A (0,1)	A (0,2)	A (1,0)	A (1,1)	A (1,2)	A (2,0)	A (2,1)	A (2,2)

compteur =3

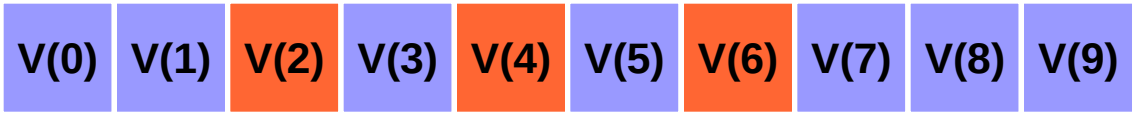
old_type =MPI_DOUBLE

new_type =rowtype

```
MPI_Type_contiguous(int count,
                    MPI_Datatype old_type,
                    MPI_Datatype *newtype);
```

Manipulation de Data Non-contigues

- Comment envoyer uniquement les éléments rouges de V, tout en évitant la copie des données non contiguës dans un tableau temporaire?



- Définir un nouveau type de données, dans ce cas, un vecteur dans la foulée (enjambé) des deux forme de l'original (violet et orange).

Exemple

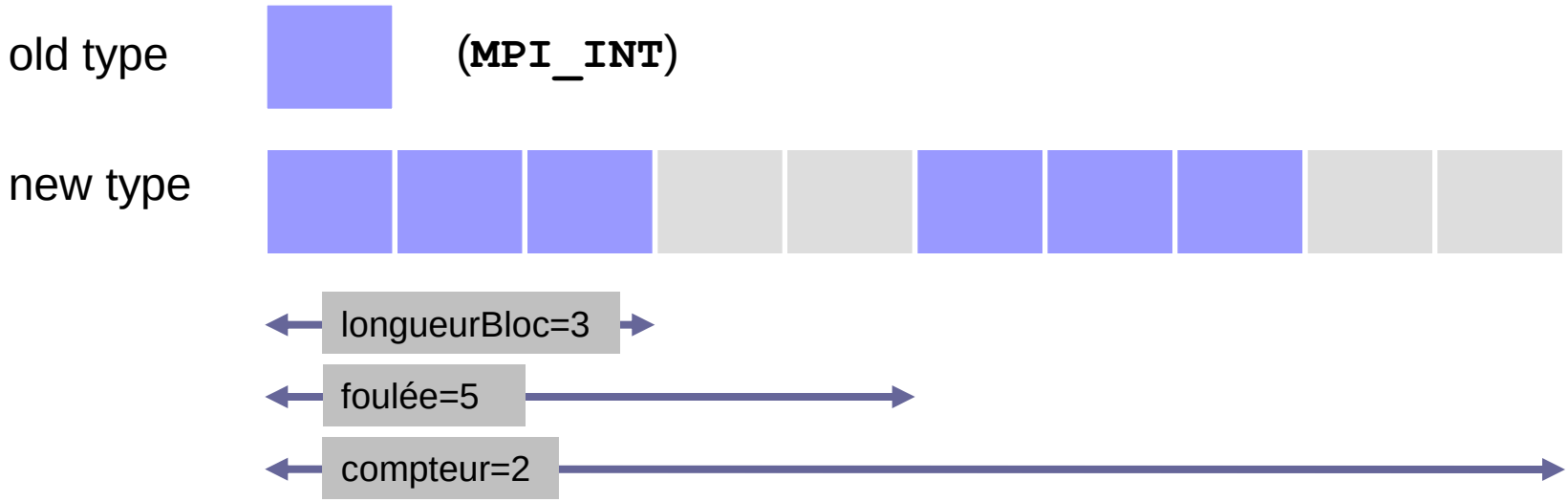
```
#include <stdio.h>
#include "mpi.h"
void main(int argc, char *argv[]) {
    int rank;
    MPI_status status;
    struct{
      int x;
      int y;
      int z;
    }point;
    MPI_Datatype ptype;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Type_contiguous(3,MPI_INT,&ptype);
    MPI_Type_commit(&ptype);
    if(rank==3){
        point.x=15; point.y=23; point.z=6;
        MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
    } else if(rank==1) {
        MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
        printf("P:%d received coords are (%d,%d,%d) \n",
             rank,point.x,point.y,point.z);
    }
    MPI_Finalize();
}
```

P:1 received coords are (15,23,6)

MPI_Type_vector

- Semblable à contigu mais permet des en jambées régulières (foulée) dans les déplacements.



Constructeur de données avec **`MPI_Type_vector.`**

MPI_Type_vector

- Si nous créons une matrice avec l'allocation de mémoire statique, nous pouvons dire que les données de la matrice seront en mémoire contiguë.
- Supposons que nous voulons envoyer des colonnes à la chaque tâche, au lieu de lignes.

```
double A[3][3];
```

A (0,0)	A (0,1)	A (0,2)
A (1,0)	A (1,1)	A (1,2)
A (2,0)	A (2,1)	A (2,2)

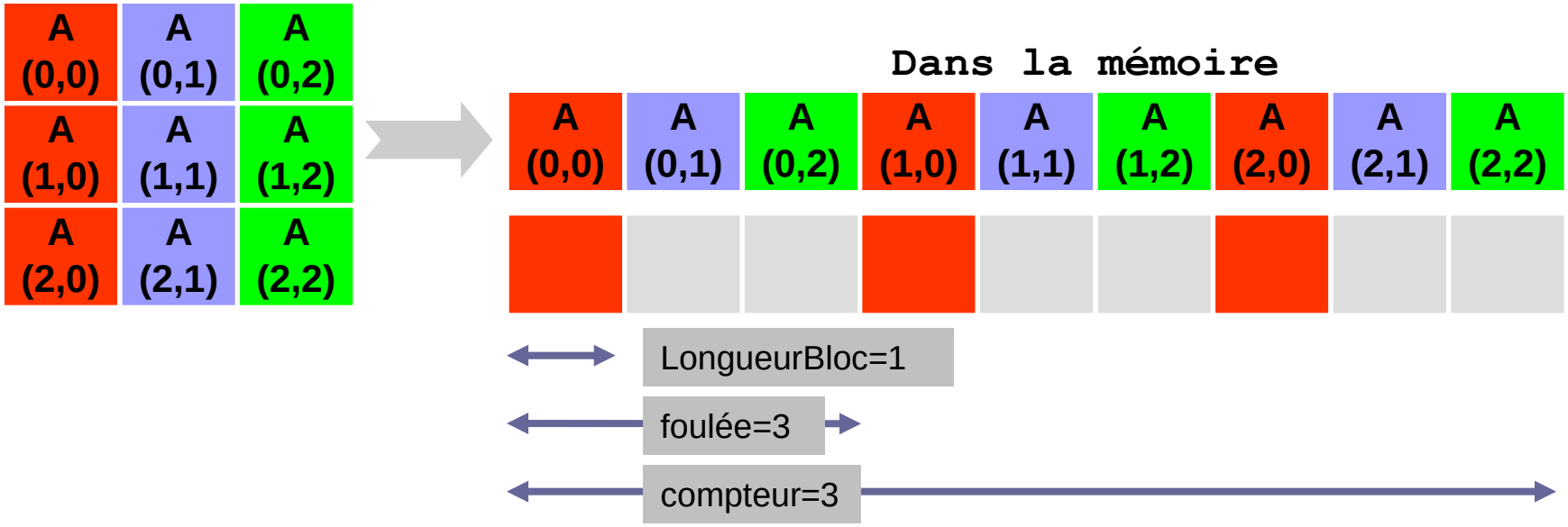
Dans la mémoire

A (0,0)	A (0,1)	A (0,2)	A (1,0)	A (1,1)	A (1,2)	A (2,0)	A (2,1)	A (2,2)

Nous pouvons utiliser, **`MPI_Type_vector`** pour créer le type vecteur de données (foulée).

MPI_Type_vector

```
double A[3][3];
```



MPI_Type_vector

longueurBloc=1

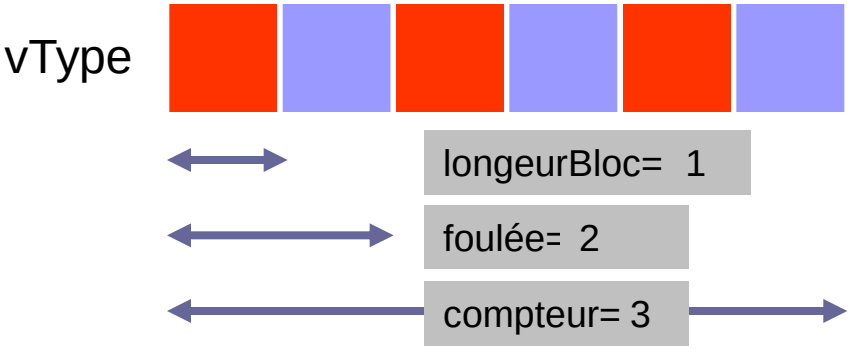
foulée=3

compteur=3

```
MPI_Type_vector(int count,
                int blocklength,
                int stride,
                MPI_Datatype old_type,
                MPI_Datatype *new_type);
```

MPI_Type_vector

- Envoyer les rouges



blocklength = 1
stride = 2
count = 3
old_type = MPI_DOUBLE
new_type = vtype

```
MPI_Type_vector( count, blocklength, stride, old_type, &vType);
MPI_Send(&V[2], 1, vType, dest, tag, MPI_COMM_WORLD);
```

MPI_Type_indexed

- constructeur indexé permet de spécifier un agencement de données non contiguës où les déplacements entre les blocs successifs ne doivent pas être égaux.

double A[4][4];

A (0,0)	A (0,1)	A (0,2)	A (0,3)
A (1,0)	A (1,1)	A (1,2)	A (1,3)
A (2,0)	A (2,1)	A (2,2)	A (2,3)
A (3,0)	A (3,1)	A (3,2)	A (3,3)

MPI_Type_indexed

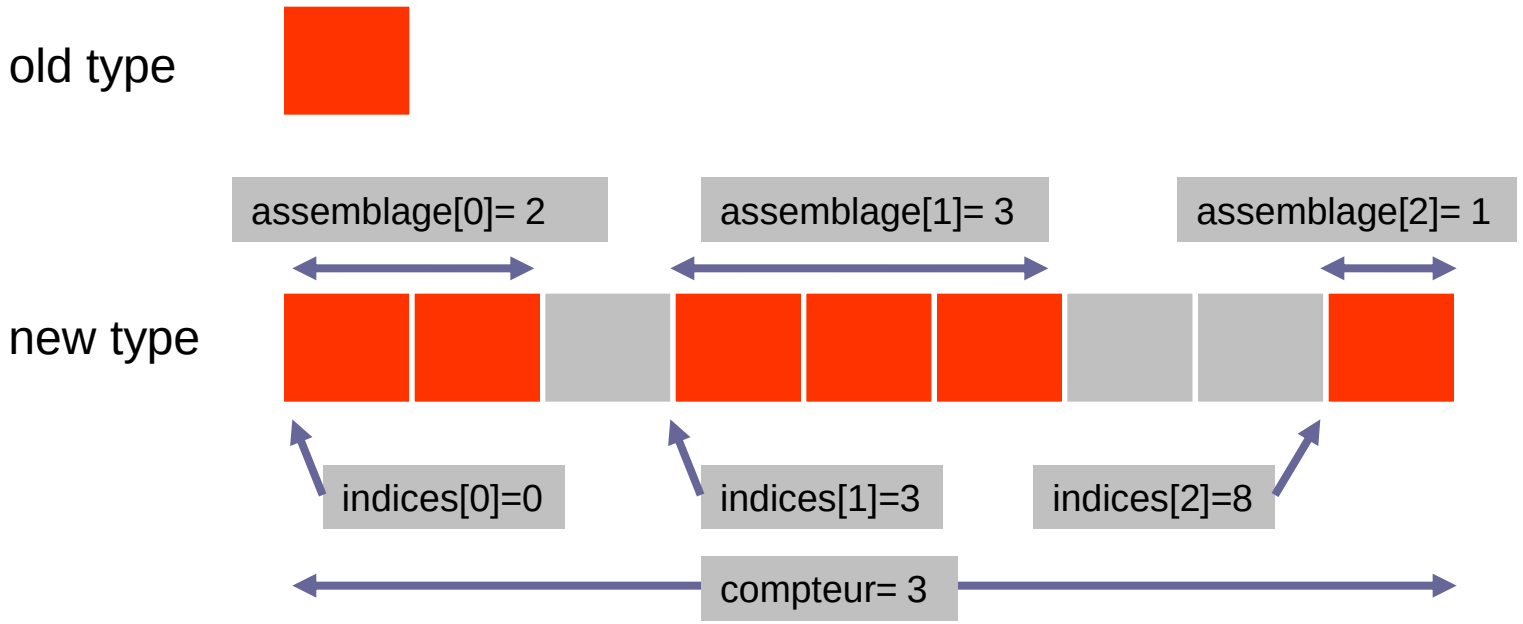
- Ceci permet:
  - Rassembler des entrées arbitraires d'un tableau et de les envoyer dans un message, ou
  - Réception d'un message et la diffusion du message reçu des entrées dans des emplacements arbitraires dans un tableau.

MPI_Type_indexed

```
int MPI_Type_indexed( int count,
                      int blocklength[],
                      int indices[],
                      MPI_Datatype old_type,
                      MPI_Datatype *newtype )
```

- count : nombre de blocs
- blocklength : nombre d'élements dans chaque bloc
- indices : displacement pour chaque bloc, mesuré comme nombre d'élements

MPI_Type_indexed



MPI_Type_indexed

- Sopposons que nous avons une matrice A(4x4)
- Nous voulons envoyer la matrice triangulaire suppérieure

double A[4][4];

A (0,0)	**A (0,1)**	**A (0,2)**	**A (0,3)**
A (1,0)	**A (1,1)**	**A (1,2)**	**A (1,3)**
A (2,0)	**A (2,1)**	**A (2,2)**	**A (2,3)**
A (3,0)	**A (3,1)**	**A (3,2)**	**A (3,3)**

old type= MPI_DOUBLE

new type = upper

count = 4

blocklen[ ] = (4, 3, 2, 1)

indices[ ] = (0, 5, 10, 15)

```
MPI_Type_indexed(count, blocklen, indices, MPI_DOUBLE, upper )
```

Types STRUCT (hétérogènes)

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

MPI_Type_commit

- Engage (commits) nouveau type de données dans le système.
- Obligatoire pour tous types de données (dérivés) construit de l'utilisateur.

```
int MPI_Type_commit( MPI_Datatype *datatype )
```

MPI_Type_free

- Desallouer les objets data type spécifiés.
- L'utilisation de cette routine est particulièrement important pour éviter l'épuisement de mémoire si beaucoup d'objets de data type sont créés , comme dans une boucle

```
int MPI_Type_free( MPI_Datatype *datatype )
```

Exemple

```
int rank,i;MPI_Status status;
struct {
   int num;
   float x;
   double data[4];
  }a;
  int blocklengths[3]={1,1,4};
  MPI_Datatype types[3]={MPI_INT,MPI_FLOAT,MPI_DOUBLE};
  MPI_Aint displacements[3];
  MPI_Datatype restype;
  int intex,floatex;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Type_extent(MPI_INT,&intex);MPI_Type_extent(MPI_FLOAT,&floatex);
  displacements[0]=0; displacemtnts[1]=intex;
  displacements[2]=intex+floatex;
  MPI_Type_struct(3,blocklengths,displacements,types,&restype);
  MPI_Type_commit(&restype);
  if(rank==3){
     a.num=6; a.x=3.14; for(i=0;i 4;++i) a.data[i]=(double) i;
     MPI_Send(&a,1,restype,1,52,MPI_COMM_WORLD);
  } else if(rank==1) {
     MPI_Recv(&a,1,restype,3,52,MPI_COMM_WORLD,&status);
     printf("P:%d my a is %d %f %lf %lf %lf %lf\n",
         rank,a.num,a.x,a.data[0],a.data[1],a.data[2],a.data[3]);
   }
   MPI_Finalize();
}
```

P:1 my a is

6 3.140000 0.000000 1.000000 2.000000 3.000002

Compactage/décompactage

- Les fonctions MPI_Pack et MPI_Unpack, compactent et décompactent différents données destinées à être envoyées.

- *int MPI_Pack(void *buf, int count, MPI_Datatype dtype, void *packbuf, int packsize, int *packpos, MPI_Comm comm)*

- *int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)*

Exemple

```
#define COM  MPI_COMM_WORLD
  if (my_rank == root){
     printf("Donner a, b, et n\ n");
     scanf("% f %f %d", a, b, n);
     /* compactage des données dans le buffer */
     position = 0;
     /* On commence au début du buffer */
     MPI_ Pack( a, 1, MPI_ FLOAT, buffer, 100, &position, COM);
     /* position a été incrémenté de sizeof(float) bytes */
     MPI_ Pack( b, 1, MPI_ FLOAT, buffer, 100, &position,);
     MPI_ Pack( n, 1, MPI_ INT, buffer, 100, &position, COM);
     /* Diffusion du contenu du buffer */
     MPI_Bcast( buffer, 100, MPI_PACKED, root, COM);
}
```

exemple (suite)

```
else {
  MPI_Bcast( buffer, 100, MPI_PACKED, root, COM);
  /* Unpack des données depuis le buffer */
  position = 0;
  MPI_ Unpack( buffer, 100, &position, a, 1, MPI_FLOAT, COM);
  /* De même, position a été incrémenté de sizeof( float) bytes */
  MPI_Unpack( buffer, 100, &position, b, 1, MPI_FLOAT, COM);
  MPI_Unpack( buffer, 100, &position, n, 1, MPI_INT, COM);
}
```