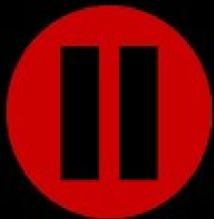
Introduction à MPI

Mésocentre de calcul de Franche-Comté

Kamel Mazouzi

01/06/2010



mésocentredecalculdefranche-comté

Plan

- Introduction
- Le standard MPI
- Communications point à point
- Communications collectives
- Performances

Contexte : parallélisme de tâches

- Décomposition d'une tâche en sous-tâches
  - variables partagées entre les tâches
  - communications par messages entre les tâches

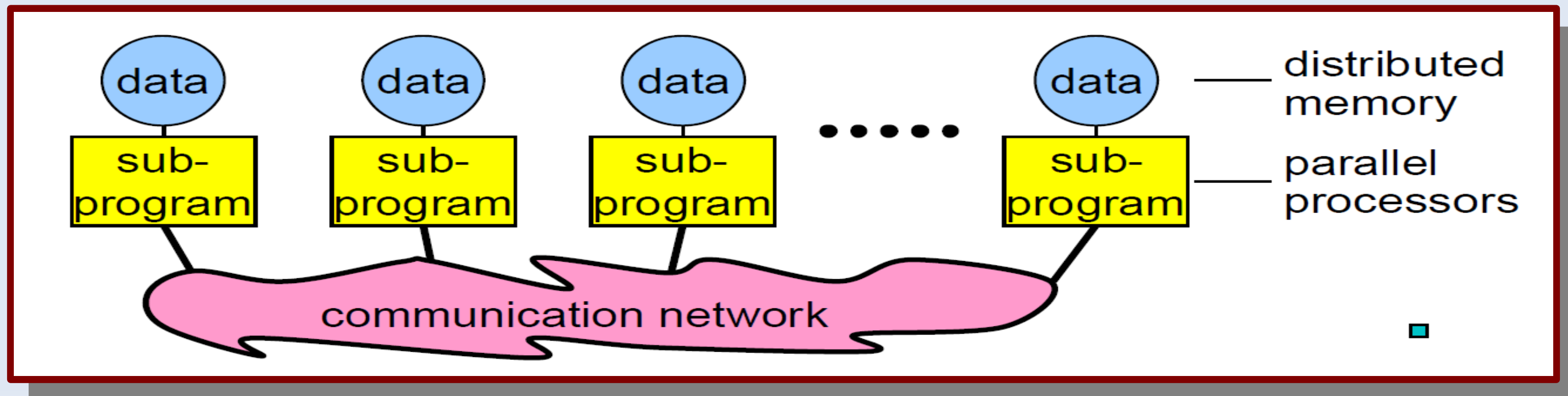
Parallélisme = réalisation simultanée de différents traitement sur les données

- Exemple : **MPI** sur architectures à mémoire distribuée

Introduction : définition

Modèle de programmation par échange de messages :

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



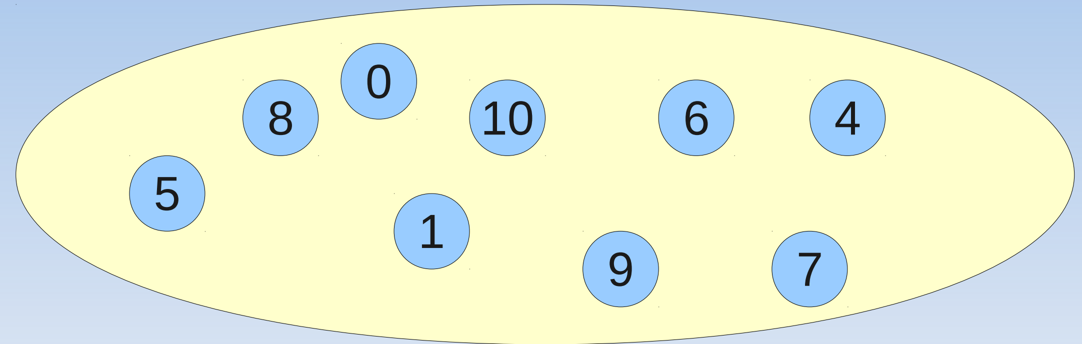
Introduction : définition

- **Le modèle d'exécution SPMD :**
  - **S**ingle **P**rogram, **M**ultiple **D**ata
  - le même programme est exécuté par tous les processus
  - c'est un cas particulier du modèle plus général MPMD (**M**ultiple **P**rogram, **M**ultiple **D**ata)

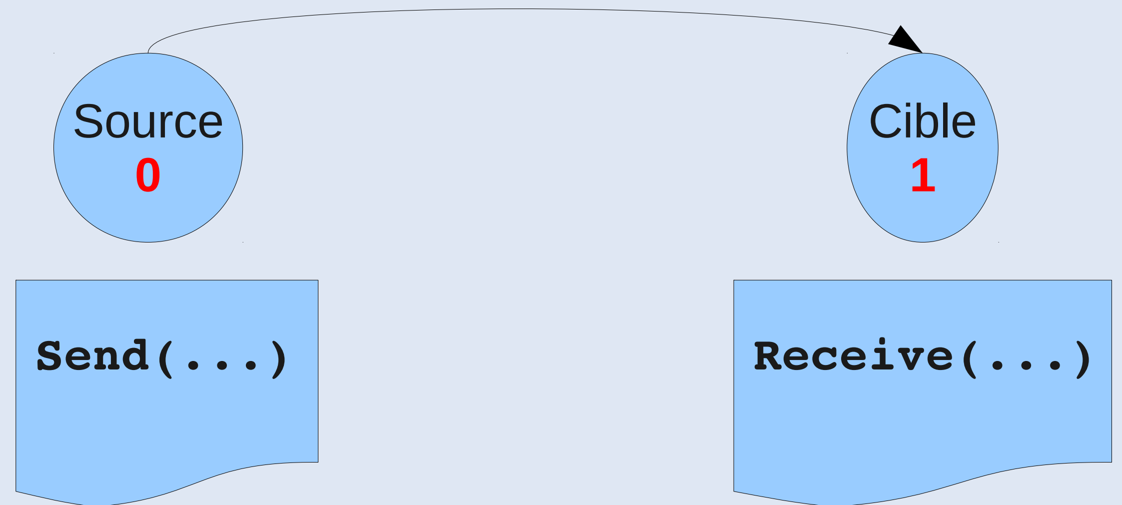
```
program spmd
if (ProcessusMaître) then
   call LeMaitre(Arguments)
else
   call LesEsclaves(Arguments)
endif
end program spmd
```

Introduction : concepts de l'échange de messages

- Les processus sont numérotés de `0` à `N-1`

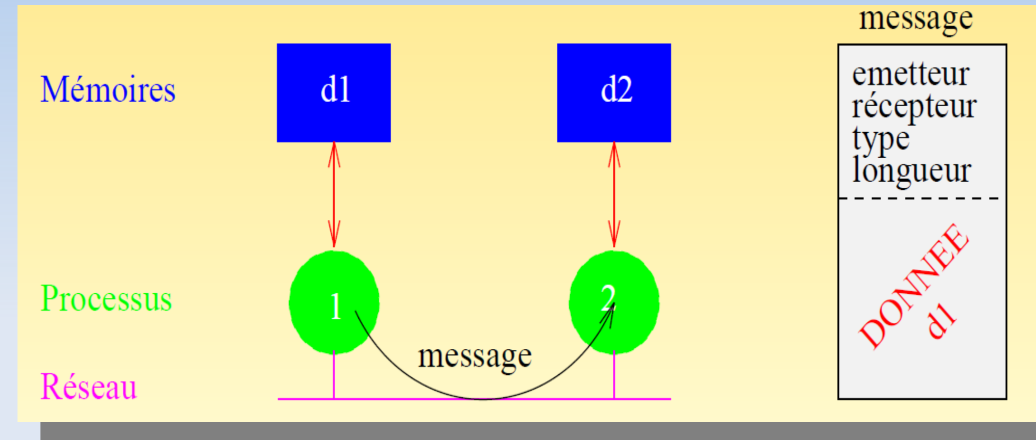


- Envoi et réception explicite de messages
- Si un message est envoyé à un processus, celui-ci doit ensuite le recevoir



Introduction : concepts de 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
  - étiquette (tag)




```
send(tampon,taille,destinaitre,type)
```

```
recv(tampon,taille,expediteur,type)
```

Le standard MPI

MPI

- MPI : **M**essage **P**assing **I**ntreface
- Standard pour le modèle de passage de message (1994)
  - Performance
  - Portabilité
  - Fonctionnalité : plus de 100 routines dans MPI-1
- Plusieurs implémentations
  - **MPICH** (Intel MPI, HP MPI, Bull-MPI, ...)
  - **OpenMPI**
  - LAM-MPI

MPI : application

- Une application MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI
- Ces sous-programmes peuvent être classés dans les grandes catégories suivantes :
  - **environnement**
  - **communications point à point**
  - **communications collectives**
  - **types de données dérivées**
  - topologies
  - groupes et communicateurs

MPI : compilation & exécution

Compilation

```
mpicc program.c -o program

mpif90 program.f90 -o program
```

Exécution

```
mpirun -np <N> -machinefile <machine> program param1 param2 ...
```

np : nombre de processus
machinefile : fichier contenant la liste des noeuds

MPI : environnement

- Tout unité de programme appelant des sous-programmes MPI doit inclure un fichier d'entêtes :
  - C/C++ : **`mpi.h`**
  - Fortran : **`mpif.h`**
- Le sous-programme **`MPI_INIT()`** permet d'initialiser l'environnement nécessaire
- Le sous-programme **`MPI_FINALIZE()`** désactive cet environnement
- Le sous-programme **`MPI_ABORT()`** termine tous les processus MPI

```
#include mpi.h
...
MPI_Init (&argc,&argv);
/*CODE PARALLEL*/
 …

MPI_Finalize ();
```

```
include 'mpif.h'
...
integer err
CALL MPI_INIT(err);
/*CODE PARALLEL*/
 …

MPI_FINALIZE (err);
```

MPI : environnement

- Pour connaître le nombre de processus gérés par un communicateur :

MPI_COMM_SIZE()

- Pour obtenir le rang d'un processus dans un communicateur :

MPI_COMM_RANK()

```
#include mpi.h
…
int rank,numtasks;
MPI_Init (&argc,&argv);

MPI_Comm_rank
    (MPI_COMM_WORLD, &rank) ;

MPI_Comm_size
(MPI_COMM_WORLD, &numtasks);

MPI_Finalize ();
```

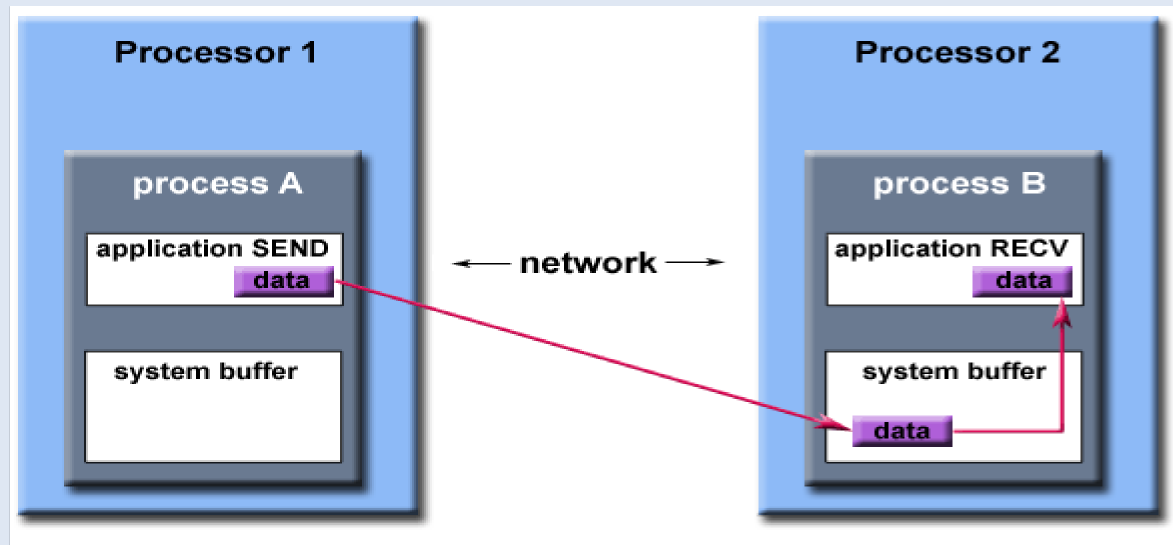
```
 include 'mpif.h'
 ...
 integer err, nb_procs, rang
   CALL MPI_INIT(err);

   call MPI_COMM_SIZE
       (MPI_COMM_WORLD ,nb_procs,code)

   call MPI_COMM_RANK
       (MPI_COMM_WORLD ,rang,code)

MPI_FINALIZE (err);
```

MPI : exemple « HelloWorld »

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

int main(argc,argv)
int argc;
char *argv[]; {
int  numtasks, rank, rc;

rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  printf ("Error starting MPI
           program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf ("Number of tasks=
  %d My rank= %d\n", numtasks,rank);

/*******  do some work *******/

MPI_Finalize();
}
```

```fortran
   program hello
   include 'mpif.h'

   integer numtasks, rank, ierr, rc

   call MPI_INIT(ierr)
   if (ierr .ne. MPI_SUCCESS) then
      print *,'Error starting
        MPI program. Terminating.'
     call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
   end if

   call MPI_COMM_RANK
                (MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE
          (MPI_COMM_WORLD, numtasks, ierr)
   print *, 'Number of tasks
        =',numtasks,' My rank=',rank

C ****** do some work ******

   call MPI_FINALIZE(ierr)

   end
```

Communications point à point



Communications point à point

- Une communication est dite point à point si elle implique uniquement **deux processus,** l'un appelé processus émetteur et l'autre processus récepteur (ou destinataire)
- L'émetteur et le récepteur sont identifiés par leur **rang** dans le communicateur
- Les données échangées sont **typées** (entiers, réels, etc. ou types dérivés personnels)
- Les données sont encapsulées dans une l'**enveloppe** qui est constituée :
  - du **rang** du processus émetteur
  - du **rang** du processus récepteur
  - de l'étiquette (**tag**) du message
  - du **nom du communicateur**
- Plusieurs type de communications : **bloquantes /non-bloquantes**

Type de Communications

SEND Mode	bloquant	non-bloquant
Standard	**MPI_SEND**	**MPI_ISEND**
Synchrone	**MPI_SSEND**	**MPI_ISSEND**
Ready	**MPI_RSEND**	**MPI_IRSEND**
Bufferisé	**MPI_BSEND**	**MPI_IBSEND**

Type de données de base

- Principaux types de données de base (Fortran)

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

Type de données de base

- Principaux types de données de base (C/C++)

Type MPI	Type C/C++
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	Type hétérogène

Communications bloquantes

MPI_Send

Envoie le message et bloque jusqu'à ce que le buffer d'envoi soit réutilisable

```
MPI_Send (&buf,count,datatype,dest,tag,comm)

MPI_SEND (buf,count,datatype,dest,tag,comm,ierr)
```

MPI_Recv

Reçoit un message et bloque jusqu'à ce que les données demandées soient disponibles dans le buffer de l'application

```
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)

MPI_RECV (buf,count,datatype,source,tag,comm,status,ierr)
```

Communications bloquantes

MPI_Sendrecv

Envoie un message et attend (reçoit) un autre message

```
MPI_Sendrecv (&sendbuf,sendcount,sendtype,dest,sendtag,
     &recvbuf,recvcount,recvtype,source,recvtag,
                    comm,&status)

MPI_SENDRECV (sendbuf,sendcount,sendtype,dest,sendtag,
         recvbuf,recvcount,recvtype,source,recvtag,
                  comm,status,ierr)
```

MPI : exemple send/recv

```c
char inmsg, outmsg='x';int rank,
size, source, dest, tag=100 ;
MPI_Status Stat;

 if(rank==0){
   dest=1 ;
MPI_Send(&outmsg, 1,
 MPI_CHAR, dest, tag, MPI_COMM_WORLD);
 }if(rank==1){
   source=0 ;
MPI_Recv(&inmsg,1,MPI_CHAR,
, tag, MPI_COMM_WORLD, &Stat);

 }
```

```fortran
integer numtasks, rank,
dest, source, count, tag, ierr
&integer stat(MPI_STATUS_SIZE)
character inmsg, outmsg
outmsg = 'x'
tag = 100
 if (rank .eq. 0) then
      dest = 1
call MPI_SEND(outmsg, 1,
   &MPI_CHARACTER, dest, tag,
   &MPI_COMM_WORLD, ierr)
 else if (rank .eq. 1) then
     source = 0
 call MPI_RECV(inmsg, 1,
  MPI_CHARACTER, source, tag,
 & MPI_COMM_WORLD, stat, err)
```

Communications non-bloquantes

Les fonctions de communications retournent avant que la communication ne soit terminée

MPI_Isend

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
MPI_ISEND (buf,count,datatype,dest,tag,comm,request,ierr)
```

MPI_Irecv

```
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
MPI_IRECV (buf,count,datatype,source,tag,comm,request,ierr)
```

Communications non-bloquantes

Pour vérifier / contrôler l'état des communications

Bloque jusqu'à ce que l'opération non bloquante soit terminée

```
MPI_Wait (&request,&status);
MPI_WAIT (request,status, err);
```

Effectue un test de blocage sur l'arrivé d'un message

```
MPI_Probe (source,tag,comm,&status)
MPI_PROBE (source,tag,comm,status,ierr)
```

Les jokers : **MPI_ANY_SOURCE**
MPI_ANY_TAG
Peuvent être utilisés

Communications non-bloquantes

Pour vérifier / contrôler le l'état des communications

Teste l'état de la communication (send / receive)

```
MPI_Test (&request,&flag,&status)
MPI_TEST (request,flag,status,err)
```

Teste sur la réception d'un message

```
MPI_Iprobe (source,tag,comm,&flag,&status)
MPI_IPROBE (source,tag,comm,flag,status,ierr)
```

Les jokers : **MPI_ANY_SOURCE**
MPI_ANY_TAG
Peuvent être utilisés

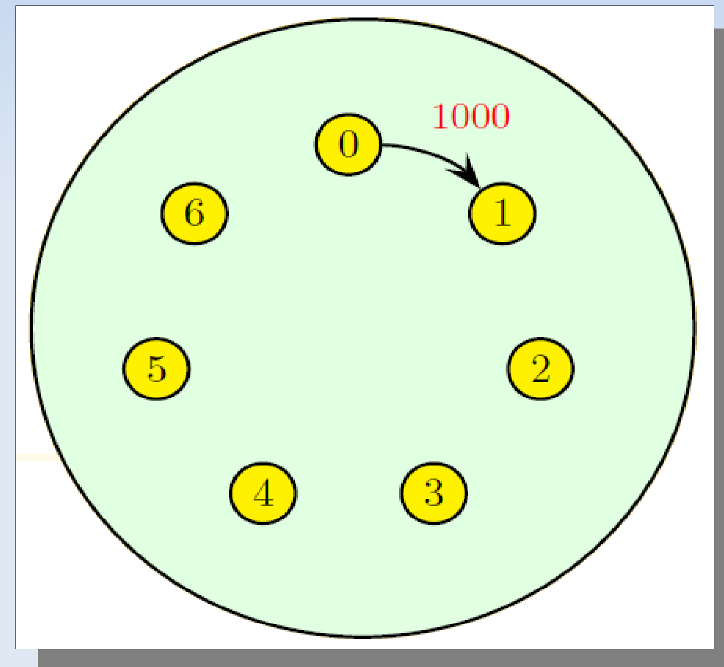
MPI : exemple Isend/Irecv

```
MPI_Request reqs[4];
MPI_Status stats[4];
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]) ;

   {  do some work  }

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
```

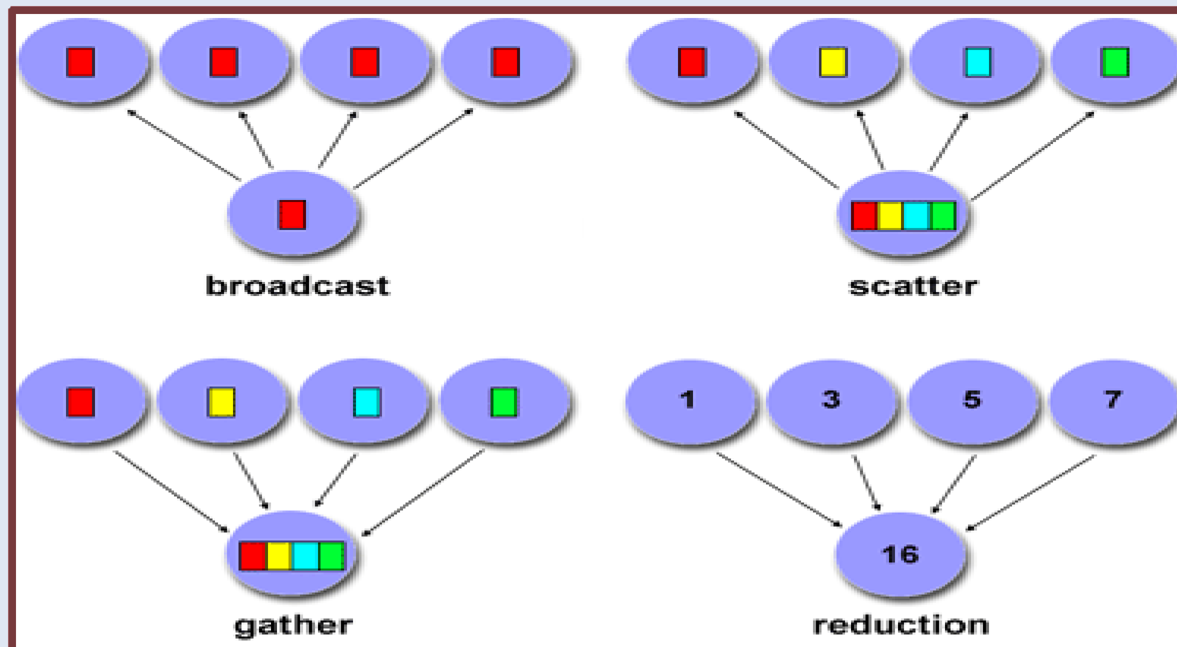
Recouvrement calcul/communication

MPI : Exemple anneau

```
call MPI_INIT (code)
call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,code)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)

num_proc_suivant=mod(rang+1,nb_procs)
num_proc_precedent=mod(nb_procs+rang-1,nb_procs)

if (rang == 0) then

call MPI_SEND (rang+1000,1, MPI_INTEGER &
  ,num_proc_suivant,etiquette, MPI_COMM_WORLD ,code)

call MPI_RECV (valeur,1, MPI_INTEGER &
    ,num_proc_precedent,etiquette,&
        MPI_COMM_WORLD ,statut,code)

 else
call MPI_RECV (valeur,1, MPI_INTEGER &
   ,num_proc_precedent,etiquette,&
    MPI_COMM_WORLD ,statut,code)

call MPI_SEND (rang+1000,1, MPI_INTEGER &
  ,num_proc_suivant,etiquette, MPI_COMM_WORLD ,code)
 end if
```



Communications collectives



Communications collectives

- 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 même communicateur
- Il est inutile d'ajouter une synchronisation globale (barrière) après une opération
- La gestion des étiquettes dans ces communications est transparente et à la charge du système
- Il existe trois types de communications :
  - Synchronisations globales
  - Transfert de données
  - Opérations de réduction

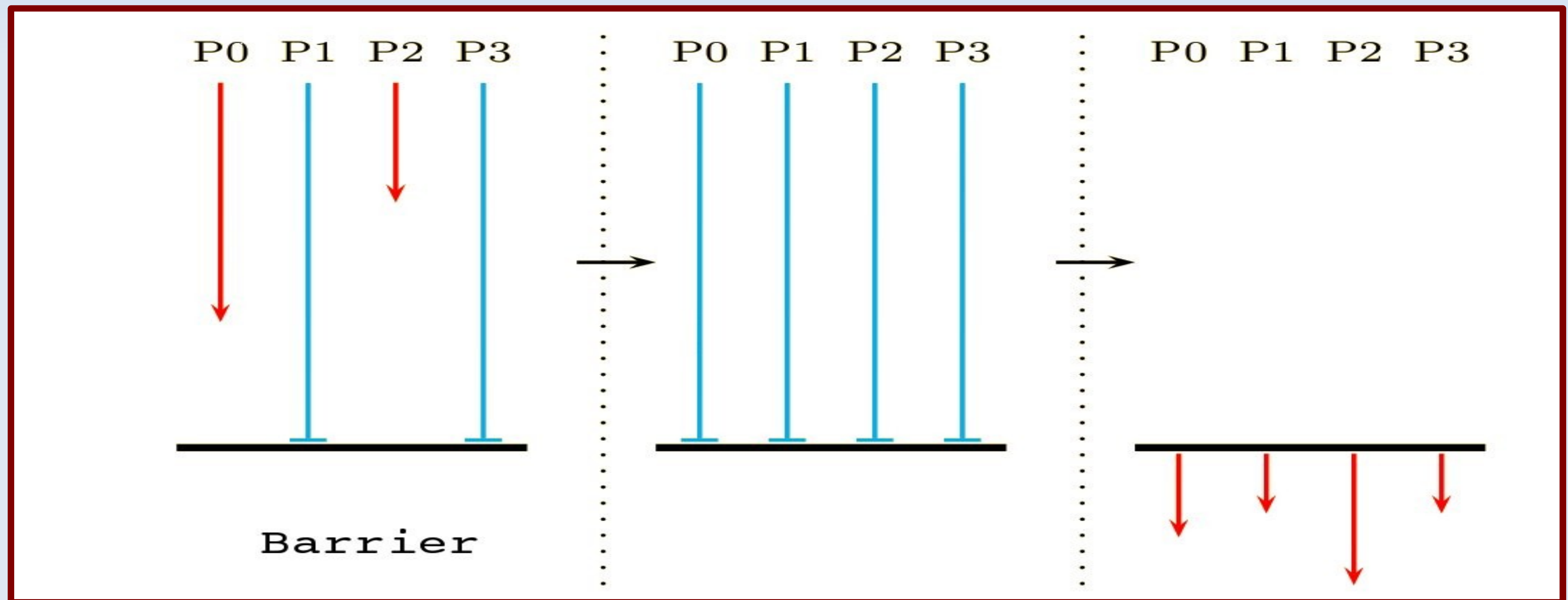
Synchronisation

Transfert de données

Réductions

Synchronisation : MPI BARRIER()

```
MPI_Barrier (comm)
MPI_BARRIER (comm,ierr)
```



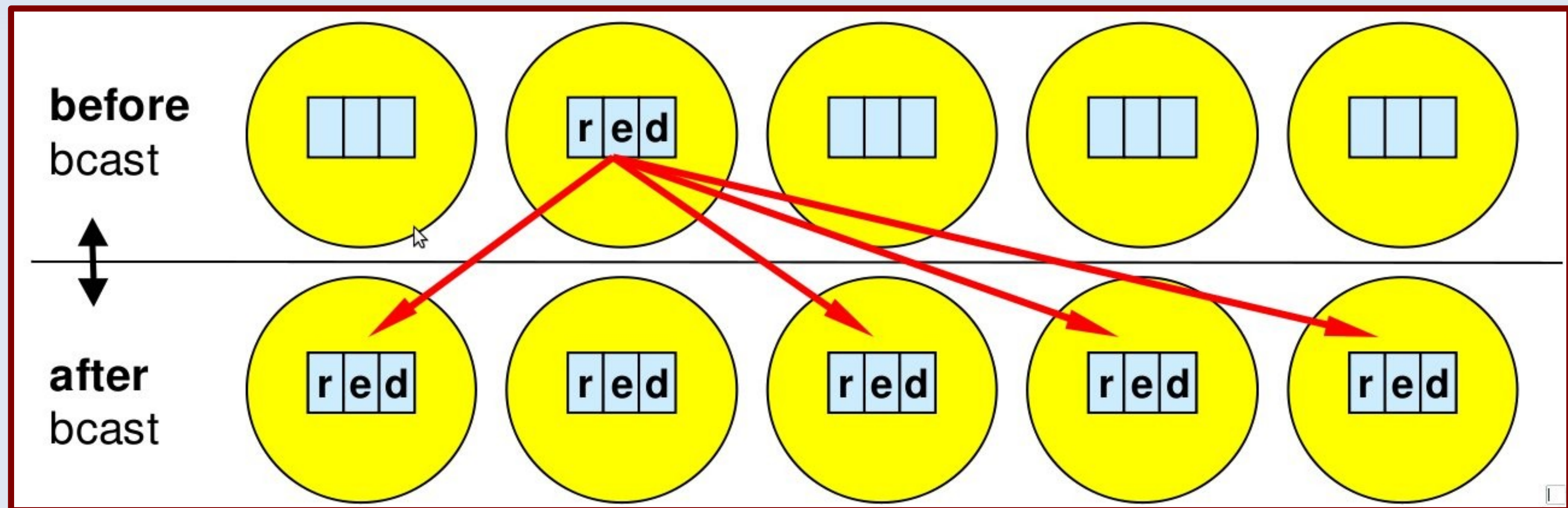
Synchronisation

Transfert de données

Réductions

Diffusion générale : MPI_BCAST()

```
MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int root, MPI_Comm co)

MPI_Bcast(BUF, COUNT, DATATYPE, ROOT, COMM, IERROR)
```



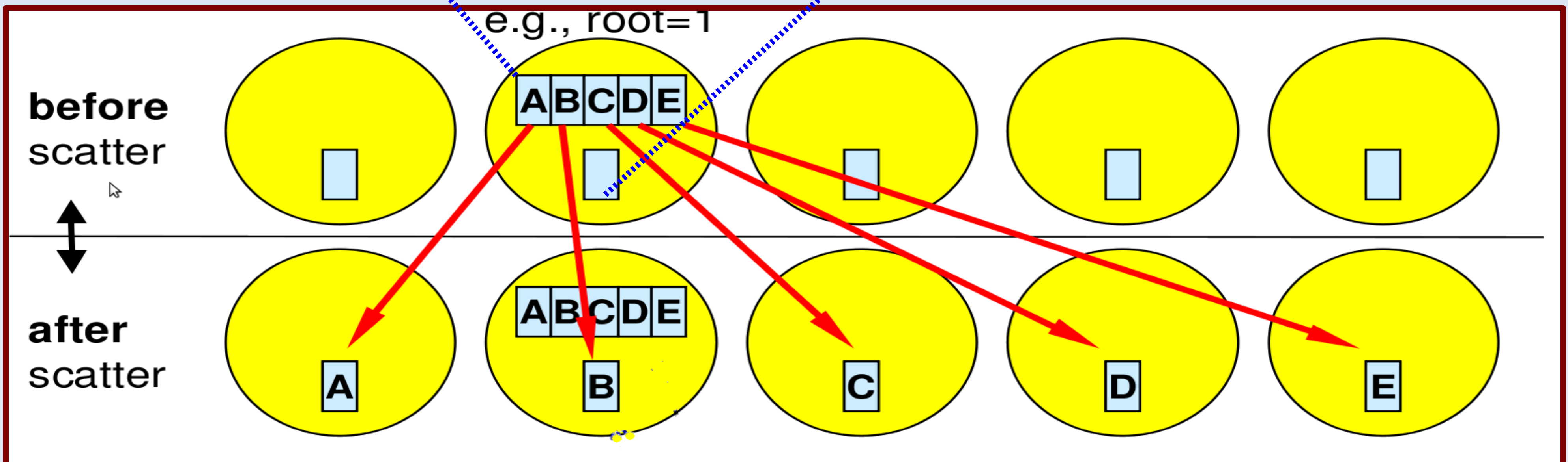
- Exemple : root=1
- ROOT : le rang du processus émetteur
- Tous les processus doivent donner la même valeur

Exemple BCAST

```
program bcast
 use mpi
  implicit none
  integer :: rang,valeur,code
  call MPI_INIT (code)
  call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)

    if (rang == 2) valeur=rang+1000

  call MPI_BCAST (valeur,1, MPI_INTEGER ,2, MPI_COMM_WORLD ,code)

   print *,'processus ',rang,', 'receive',valeur,' from rocessus 2'

  call MPI_FINALIZE (code)

end program bcast
```

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)

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                                     RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
```



Exemple :

→ MPI_Scatter(sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD)

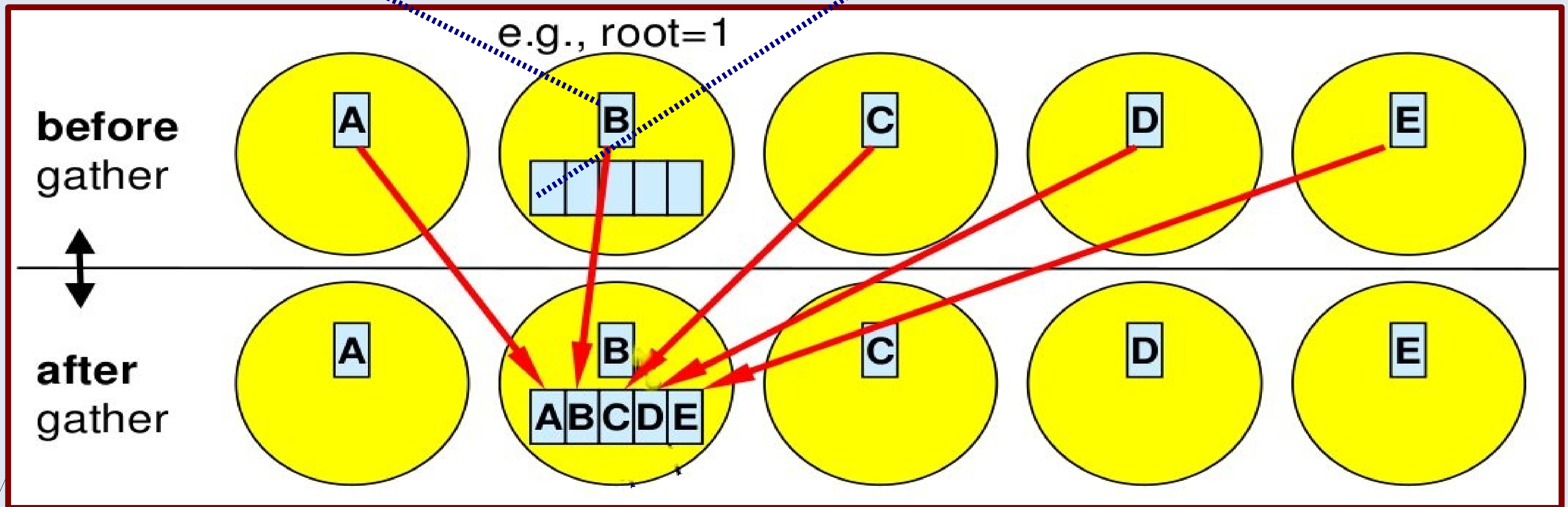
→

Exemple Scatter

```
int numtasks, rank, sendcount, recvcount, source,SIZE ;
 SIZE=4;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
             MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Collecte générale : MPI_GATHER()

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

 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
               RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
```



Exemple :

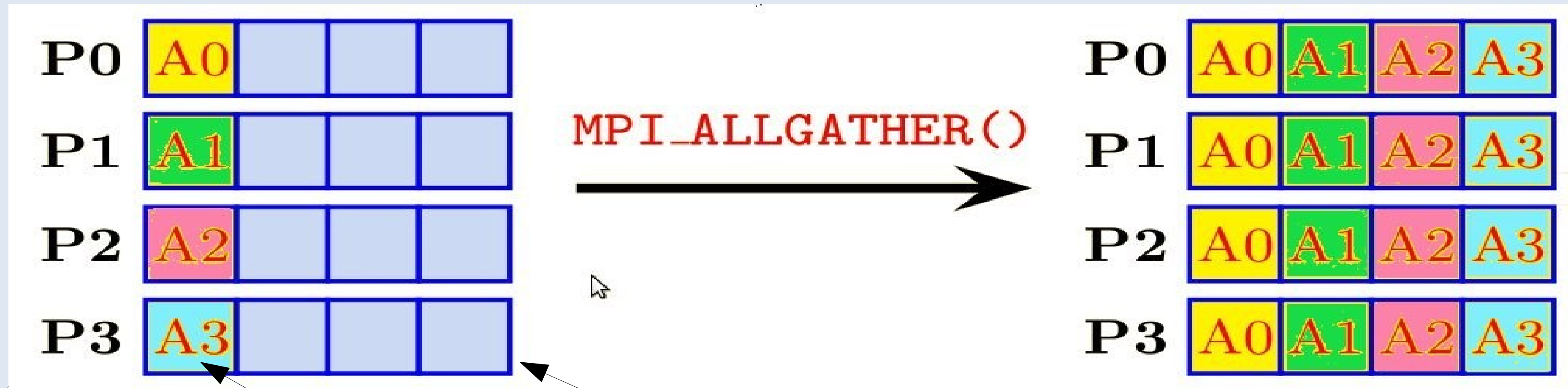
- MPI_Gather(sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD)

Collecte générale : MPI_ALLGATHER()

Concaténation des données de toutes les tâches

Il n'y a pas de processus ROOT

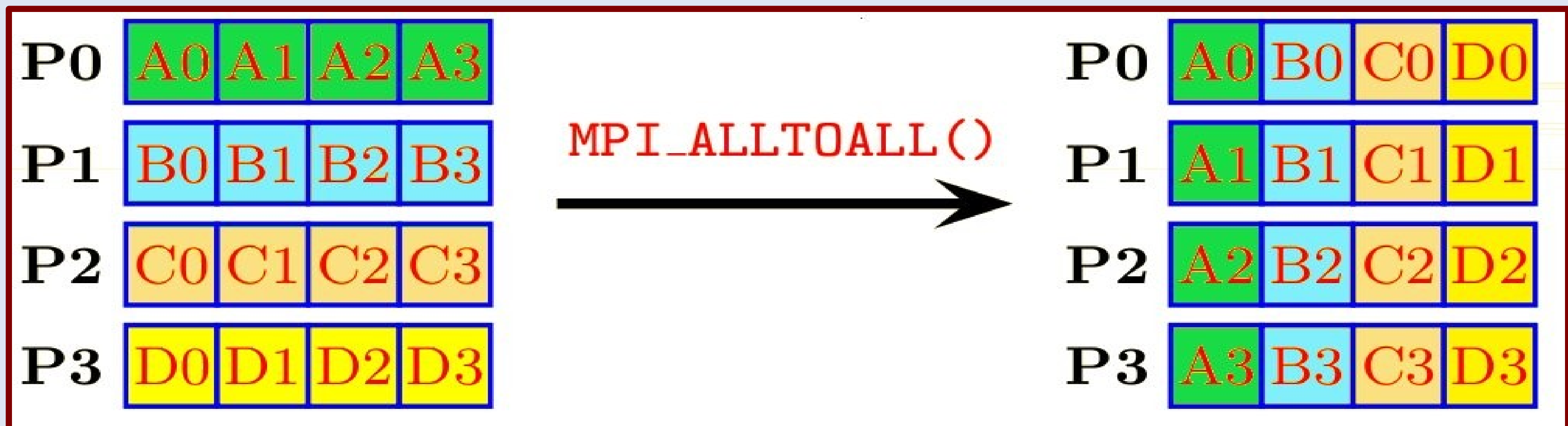
```
MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf,
        recvcount,recvtype,comm)

MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf,
        recvcount,recvtype,comm,info)
```



- MPI_AllGather(sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, MPI_COMM_WORLD)

Échange croisé : MPI_ALLTOALL

```
MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf,
              recvcnt,recvtype,comm)
MPI_ALLTOALL (sendbuf,sendcount,sendtype,recvbuf,
              recvcnt,recvtype,comm,ierr)
```



- MPI_Alltoall(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD)

Synchronisation

Transfert de données

Réductions

Réduction répartie

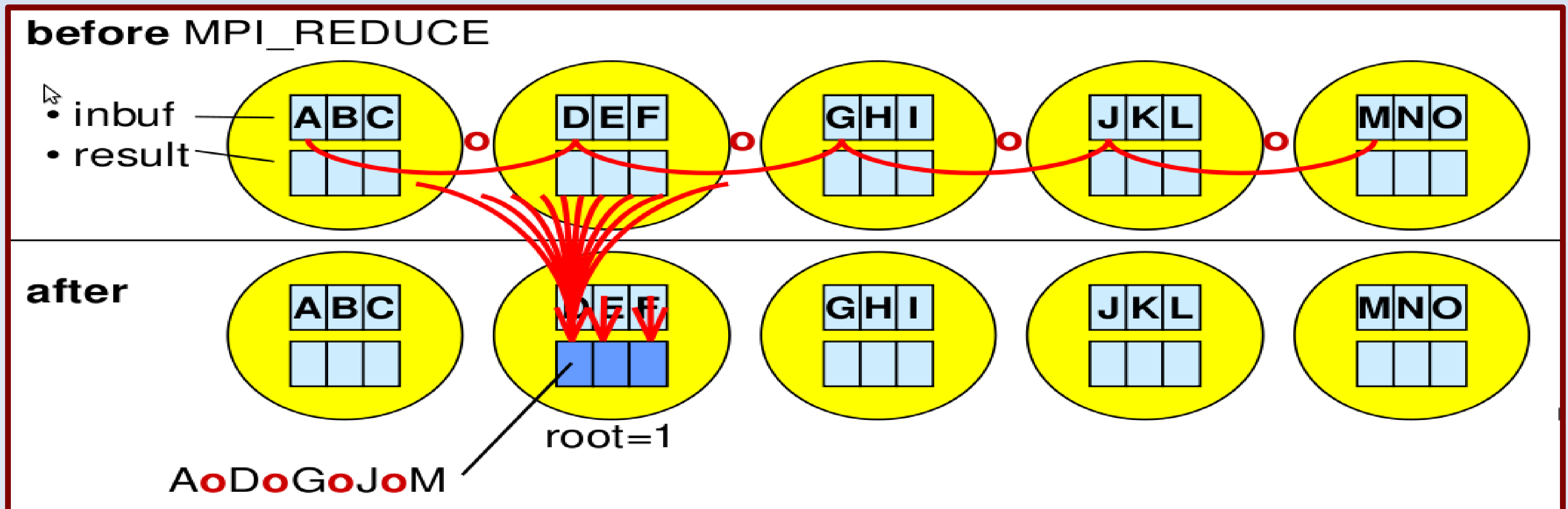
- Une réduction est une opération appliquée à un ensemble d’éléments pour en obtenir une seule valeur, par exemple :
  - Somme des élément d'un vecteur
  - Recherche du Maximum dans un vecteur
- **MPI** propose des sous-programmes de haut-niveau pour effectuer des réductions sur des données réparties avec récupération du résultat :
  - sur un seul processus **MPI_REDUCE()**
  - sur tous les processus **MPI_ ALLREDUCE()**
- Le sous-programme **MPI SCAN()** permet en plus d’effectuer des réductions partielles
- Les sous-programmes **MPI_OP_CREATE()** et **MPI_OP_FREE()** permettent de définir des opérations de réduction personnelles

Réduction répartie

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

Réduction répartie : MPI_REDUCE

```
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)

MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```



- MPI_Reduce(sbuf, rbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD)

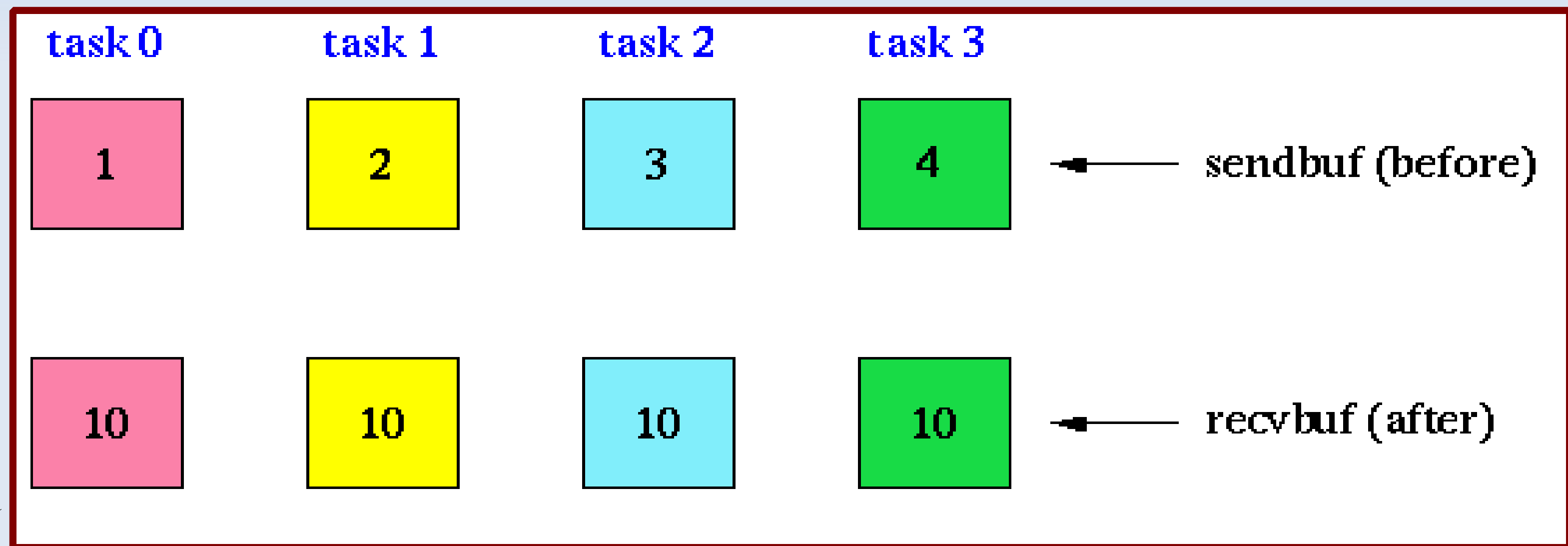
Réduction répartie : MPI_REDUCE

```
integer :: nb_procs,rang,valeur,somme,code

if (rang == 0) then
     valeur=1000
  else
     valeur=rang
   endif

 call MPI_REDUCE (valeur,somme,1, MPI_INTEGER , MPI_SUM ,0, MPI_COMM_WORLD ,code)

   if (rang == 0) then
   print *,'processus 0, valeur de la somme globale ',somme

   endif
```

```
Somme= 1000 + (1+2+...+ nb_process-1)
```

Réduction répartie : MPI_ALLREDUCE

```
MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)

MPI_ALLREDUCE (sendbuf,recvbuf,count,datatype,op,comm,ierr)
```

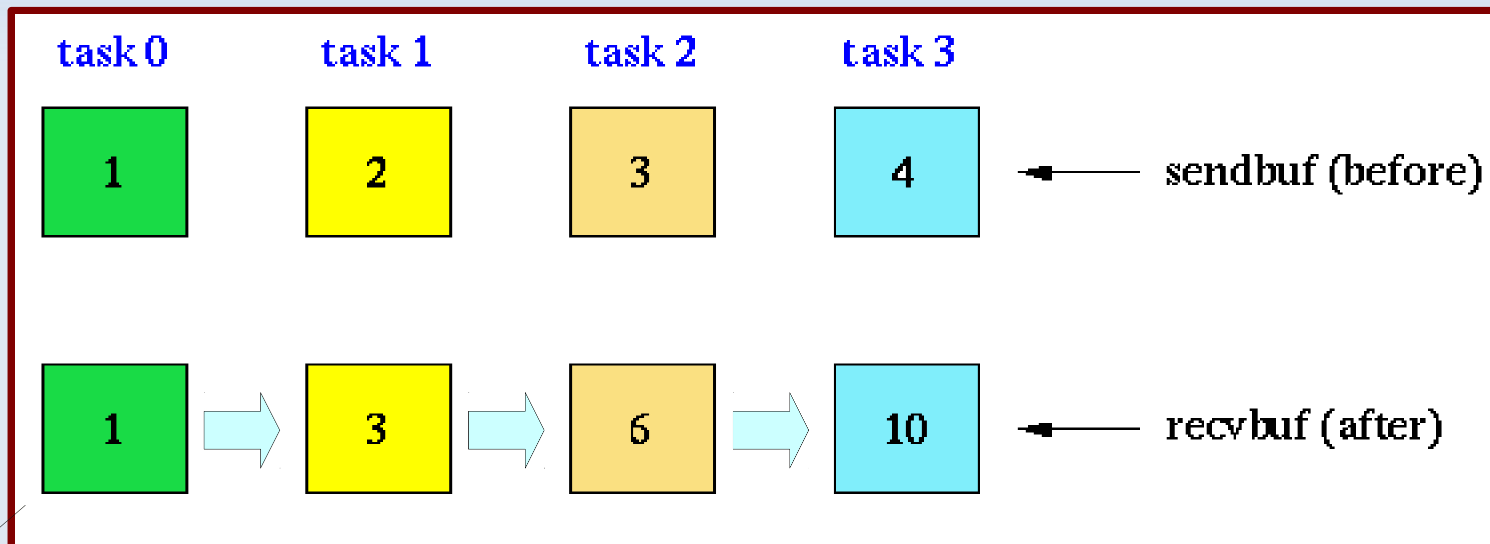


count=1

```
MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,MPI_SUM,comm)
```

Réduction répartie : MPI_SCAN

```
MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)

MPI_SCAN (sendbuf,recvbuf,count,datatype,op,comm,ierr)
```



count=1

```
MPI_Scan (&sendbuf,&recvbuf,count,datatype,MPI_SUM,comm)
```

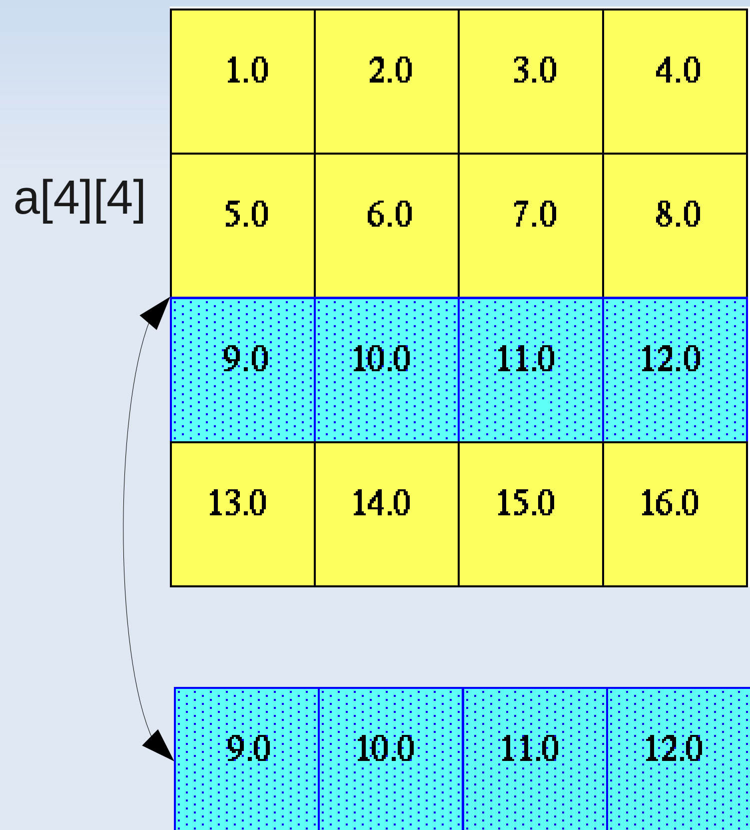
Types de données dérivés

Types de données dérivés

- MPI utilise par défaut les types de données de base (MPI_INT,...) pour les communications, les types de données primitifs sont contigus (tableaux)
- MPI fournit également des techniques pour définir des structures de données dérivées basées sur des séquences de types MPI de base
- MPI fournit plusieurs méthodes pour construire des types de données dérivés
  - **Contiguous**
  - **Vector**
  - **Indexed**
  - **Struct**

Type contiguous

```
MPI_Type_contiguous (count,oldtype,&newtype)
MPI_TYPE_CONTIGUOUS (count,oldtype,newtype,ierr)
```




```
count=4
MPI_Type_contigous(count,MPI_FLOAT,&rowtype)
```

```
MPI_Send(&a[2][0],1,rowtype,dest, tag, com)
```

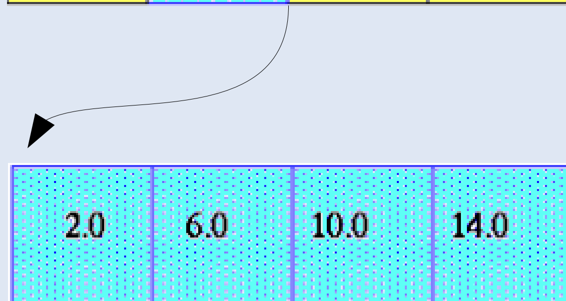
Type vector

```
MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)
MPI_TYPE_VECTOR (count,blocklength,stride,oldtype,newtype,ierr)
```

a[4][4]

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

```
count=4 ;blocklenght=1;stride=4

MPI_Type_vector (count,blocklenght, stride,
        MPI_FLOAT,&vecttype)
```

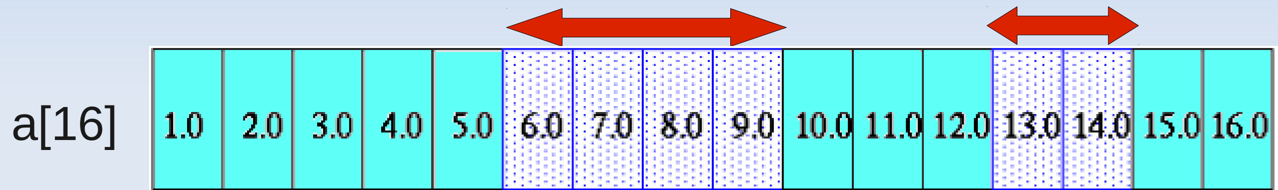



```
MPI_Send(&a[0][1],1,vecttype,dest, tag, com)
```

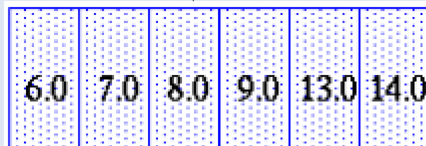
Type indexed

```
MPI_Type_indexed (count,blocklens[],offsets[],old_type,&newtype)
MPI_TYPE_INDEXED (count,blocklens(),offsets(),old_type,newtype,ierr)
```

a[16]




```
count=2 ;
blocklens[0]=4;
offsets[0]=5 ;

blocklens[1]=2;
offsets[1]=12;

MPI_Type_indexed(count,blocklens, offsets,
        MPI_FLOAT,&indexedtype)
```

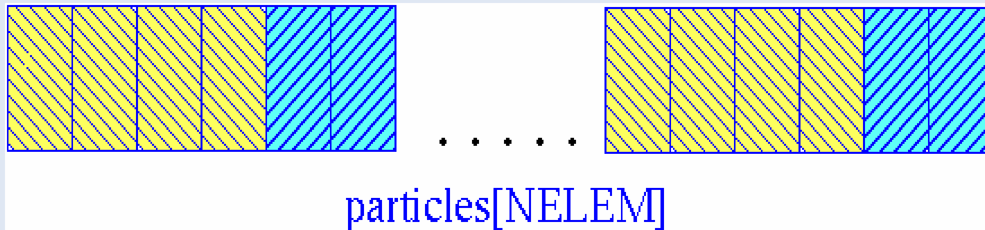



```
MPI_Send(&a,1,indexedtype,dest, tag, com)
```

Type struct

```
MPI_Type_struct (count,blocklens[],offsets[],old_types[],&newtype)
MPI_TYPE_STRUCT (count,blocklens(),offsets(),old_types(),newtype,ierr)
```

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];
```




```
count=2 ;
MPI_Type_Extent(MPI_FLOAT,&extlen);
old_type[0]=MPI_FLOAT;
offsets[0]=0 ;
blocklens[0]=4;

old_type[1]=MPI_INT;
offsets[1]=4*extlen ;
blocklens[1]=2;

MPI_Type_struct(count,blocklens,offsets,
oldtype,&structtype)
```

- Envoie le tableau de particules
- Chaque particule est composé de 4 float et de deux int

```
MPI_Send(particules,NELEM,structtype,dest, tag, com)
```

Types de données dérivés

- Il faut ***valider*** les nouveaux types avant de les utiliser dans les communications

```
int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(MPI_Datatype datatype, err)
```

- Marque un type de données pour la désallocation

```
int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE(MPI_Datatype *datatype, err)
```

Exemple

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] =
  {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;

MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
MPI_Type_commit(&indextype);

if (rank == 0) {
  for (i=0; i<numtasks; i++)
     MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
     rank,b[0],b[1],b[2],b[3],b[4],b[5]);

MPI_Type_free(&indextype);
MPI_Finalize();
}
```

B[6] =

6.0	7.0	8.0	9.0	13.0	14.0

Performance

Performance

- En général, les performances dépendent de l'architecture (processeur, mémoire) et de l'implémentation MPI utilisé
- Communications :

Temps global= Temps de calcul + temps des communications

- Il existe, néanmoins, quelques règles de bonne conduite indépendantes de l'architecture

Performances

Algorithmique : le rapport communication sur calcul doit être aussi faible que possible

- **Communications**: Particulièrement important lorsque le part des communications est importante par rapport au calcul
  - recouvrir les communications par les calculs
  - éviter la recopie des messages dans un espace mémoire temporaire
  - minimiser les surcoûts occasionner par les appels répétitifs aus sous-programmes de communication

Évolution de MPI : MPI-2

- Début des travaux en mars 1995
- Brouillon présenté pour SuperComputing 96 **version officielle** disponible en juillet 1997 *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**

Questions ?