Introduction à MPI
Message Passing Interface

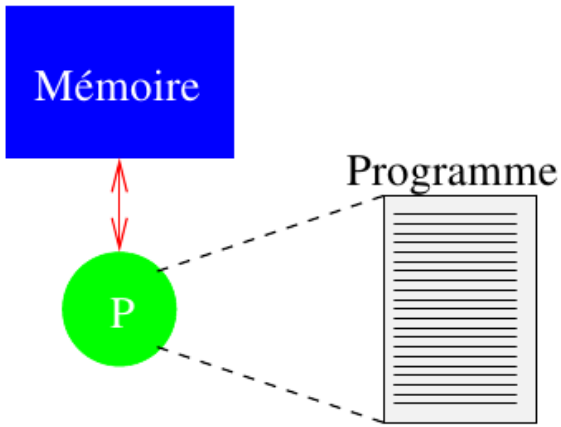
Christophe PERA

Informatique scientifique pour le calcul
Ecole doctorale 2012/2013

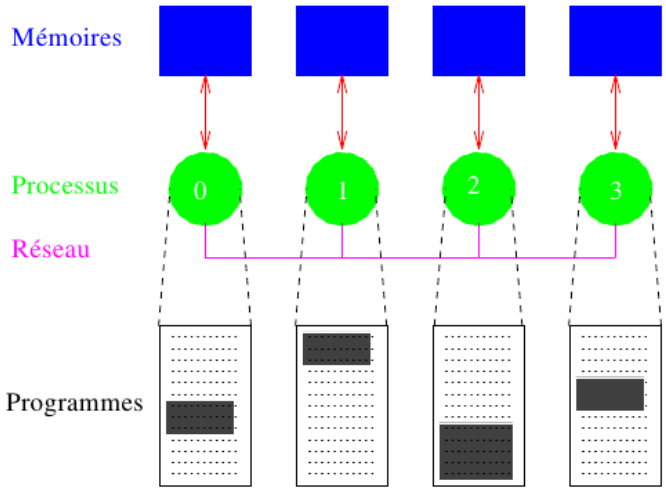
Basé sur les cours de l'IDRIS
http ://www.idris.fr/data/cours/parallel/mpi/

Lignes directrices

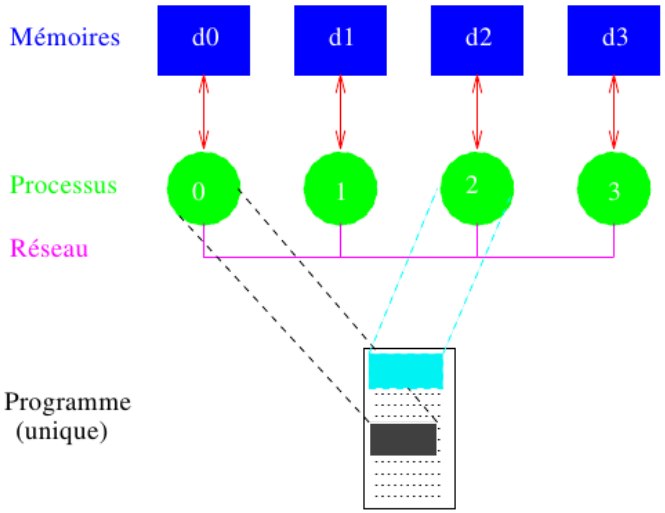
- Le modèle de programmation séquentiel :
  - le programme est exécuté par un et un seul processus ;
  - toutes les variables et constantes du programme sont allouée s dans la mémoire allouée au processus ;
  - un processus s'exécute sur un processeur physique de la machine .



- Dans le modèle de programmation par échange de messages :
  - le programme est écrit dans un langage classique (Fortran , C , C++ , python, ...) ;
  - chaque processus exécute éventuellement des parties différentes d'un programme ;
  - toutes les variables du programme sont privées et résident d ans 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.

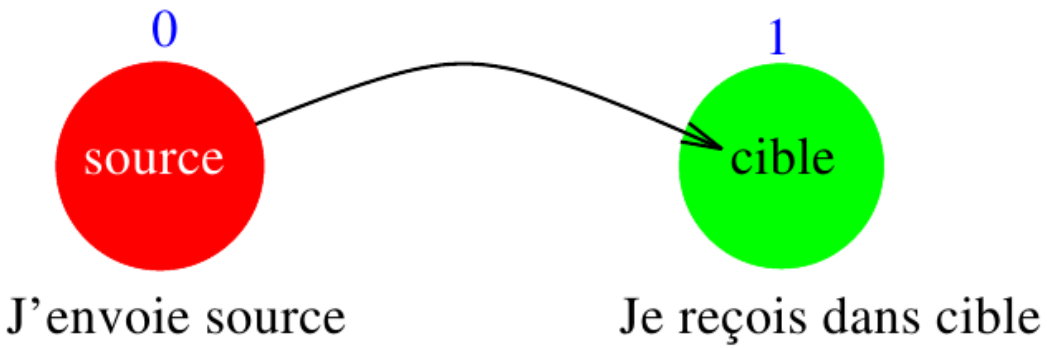


- Le modèle d'exécution SPMD :
  - Single Program , Multiple Data ;
  - le même programme est exécuté par tous les processus ;
  - toutes les machines supportent ce modèle de programmation et certaines ne supportent que celui-là ;
  - c'est un cas particulier du modèle plus général MPMD (Multiple Program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



Lignes directrices

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



- 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.

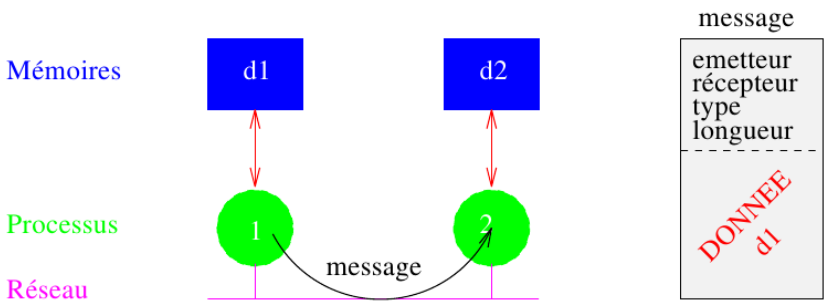


FIGURE 5 – Constitution d'un message

- Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie, à la télécopie, au courrier postal, à la messagerie électronique, etc.
- Le message est envoyé à une adresse déterminée Le processus récepteur doit pouvoir classer et interpréter les messages qui lui ont été adressés.
- L'environnement en question est MPI (Message Passing Interface).
- Une application MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI.

Lignes directrices

- Version 1.0 : en juin 1994, le forum MPI (Message Passing Interface Forum), avec la participation d'une quarantaine d'organisations, abouti à la définition d'un ensemble de sous-programmes concernant la bibliothèque d'échanges de messages MPI.
- Version 1.1 : juin 1995, avec seulement des changements mineurs.
- Version 1.2 : en 1997, avec des changements mineurs pour une meilleure cohérence des dénominations de certains sous-programmes.
- Version 1.3 : septembre 2008, avec des clarifications dans MPI 1.2, en fonction des clarifications elles-mêmes apportées par MPI-2.1.
- Version 2.0 : apparue en juillet 1997, cette version apportait des compléments essentiels volontairement non intégrés dans MPI 1.0 (gestion dynamique de processus, copies mémoire à mémoire, entrées-sorties parallèles, etc.).
- Version 2.1 : juin 2008, avec seulement des clarifications dans MPI 2.0 mais aucun changement.
- Version 2.2 : septembre 2009, avec seulement de « petites » additions.

- Version 3.0
  - changements et ajouts importants par rapport à la version 2.2 ;
  - pour un meilleur support des applications actuelles et futures, notamment sur les machines massivement parallèles et many cores ;
  - principaux changements actuellement envisagés :
    - communications collectives non bloquantes ;
    - révision de l'implémentation des copies mémoire à mémoire ;
    - tolérance aux pannes ;
    - Fortran (2003-2008) bindings ;
    - interfaçage d'outils externes (pour le débogage et les mesures de performance) ;
  - voir :
    - http ://meetings.mpi-forum.org/MPI_3.0_main_page.php.
    - https ://svn.mpi-forum.org/trac/mpi-forum-web/wiki.

Lignes directrices

- Environnement MPI Linux langage C/C++
  - openmpi.
  - mpich.
  - lam.
- Environnement MPI python
  - mpi4py.
  - pympi.

OpenMPI

- Standard MPI version 2, une partie de la version 3.
- Thread safe.
- Gestion dynamique des processus.
- Tolérance aux pannes réseau et système.
- Multiplateformes.

MPI4PY

- Projet initié en 2006 par Lissandro Dalcin.
- Interfaçage de la bibliothèque MPI-1/2 avec Swig (version $<$ 1).
- Interfaçage de la bibliothèque MPI-1/2 avec Cython (version $> = 1$).
- Ce module gère toutes les fonctionnalités que l'on peut trouver dans MPI-1/2.

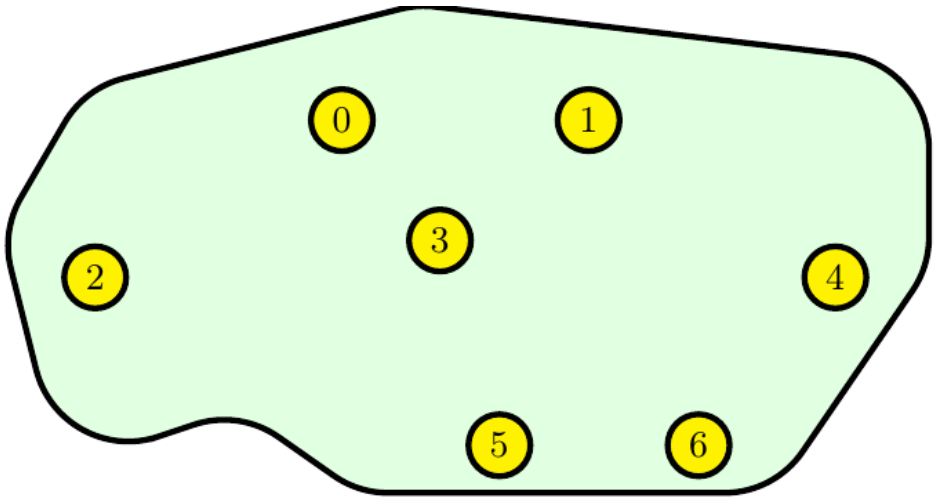
- Tout unité de programme appelant des sous-programmes MPI doit inclure un fichier d'en-têtes. La fonction MPI_INIT() permet d'initialiser l'environnement nécessaire et réciproquement, MPI_FINALIZE() désactive cet environnement :
- En C/C++ il faut utiliser le fichier mpi.h.

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

- En python, l'import du module mpi4py suffit pour initialiser l'environnement

```python
from mpi4py import MPI
# MPI Init
```

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



- On peut connaître le nombre de processus gérés par un communicateur donné par le sous-programme MPI_COMM_SIZE() et 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) :

```c
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[])
{
   int my_rank; /* Rang du processus */
   int p; /* Nombre de processus */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Finalize();
} /* main */
```

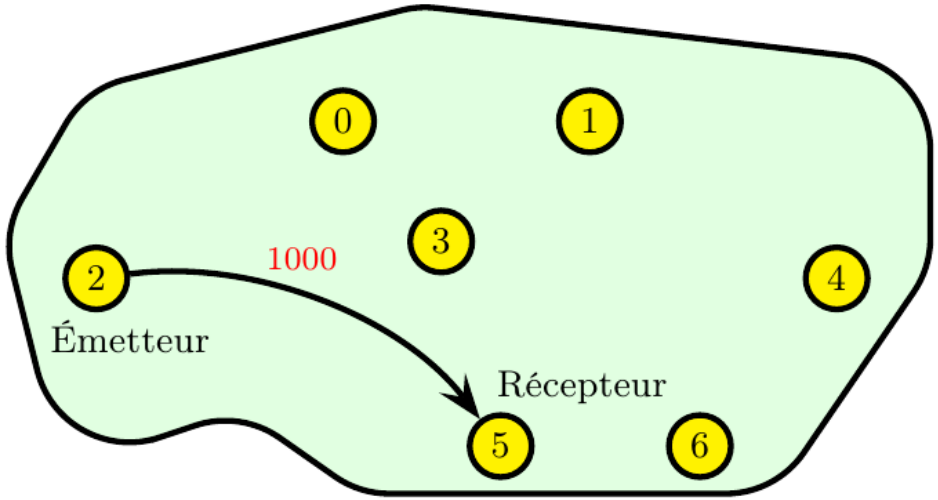
```python
from mpi4py import MPI
# MPI Init
comm = MPI.COMM_WORLD # mon communicateur
mpi_size = comm.Get_size() # le nombre de processus
mpi_rank = comm.Get_rank() # le rang du processus
#
print "Process %d of %d on %s\n" %(mpi_rank, mpi_size, mpi_name)
```

Lignes directrices

- Exemple C
- Exemple python

Lignes directrices

- Une communication dite point à point a lieu entre 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.
- 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 communicateur qui défini le groupe de processus et le contexte de communication.
- 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.

Lignes directrices

- Fonctions d'envoi/réception de messages standard : MPI_Send / MPI_Recv
  - Utile pour les envoie de message de base.
  - Fonction d'envoi bloquante (le retour de l'appel n'est fait que lorsque l'espace mémoire des données est disponible).
- Fonctions non bloquante
  - Initialisent l'envoi / la réception des données et retourne une valeur 'status' immédiatement, on peut ensuite interroger ou attendre la réalisation des opérations.
  - L'espace mémoire des données n'est pas 'accessible' tant que les opérations ne sont pas finies.
  - MPI_Probe() or MPI_Wait() renvoie les infos sur l'état de l'échange non bloquant.
- Fonction d'échange 'Synchronous' et 'Buffered' ...

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

int main(argc,argv){
   int numtasks, rank, dest, source, rc, count, tag=1;
   char inmsg, outmsg='x';
   MPI_Status Stat;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
      dest = 1;
      source = 1;
      rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &
           Stat);
   } else if (rank == 1) {
      dest = 0;
      source = 0;
      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &
           Stat);
      rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   }

   rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
   printf("Task␣%d:␣Received␣%d␣char(s)␣from␣task␣%d␣with␣tag␣%d␣\n",
        rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
   MPI_Finalize();
}
```

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
   data = {'a':7,'b':3.14}
   comm.send(data,dest=1,tag=11)
   print "Message sent, data is: ",data
elif rank == 1:
   data = comm.recv(source=0,tag=11)
   print "Message Received, data is: ",data
```

Lignes directrices

Types C

Types 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

Type python

- Classe Datatype dans le module mpi4py.
- Politique de type implicite python 'mappé' de maniéré transparente en mpi avec le module numpy !
- Dépend de l'implémentation mpi wrappé sur le système (openmpi/C, openmpi/fortran, mpich/C,....).

Types MPI	type python
MPI.BOOL	bool
MPI.CHAR	char
MPI.FLOAT	float
MPI.INT	int
MPI.DOUBLE	double

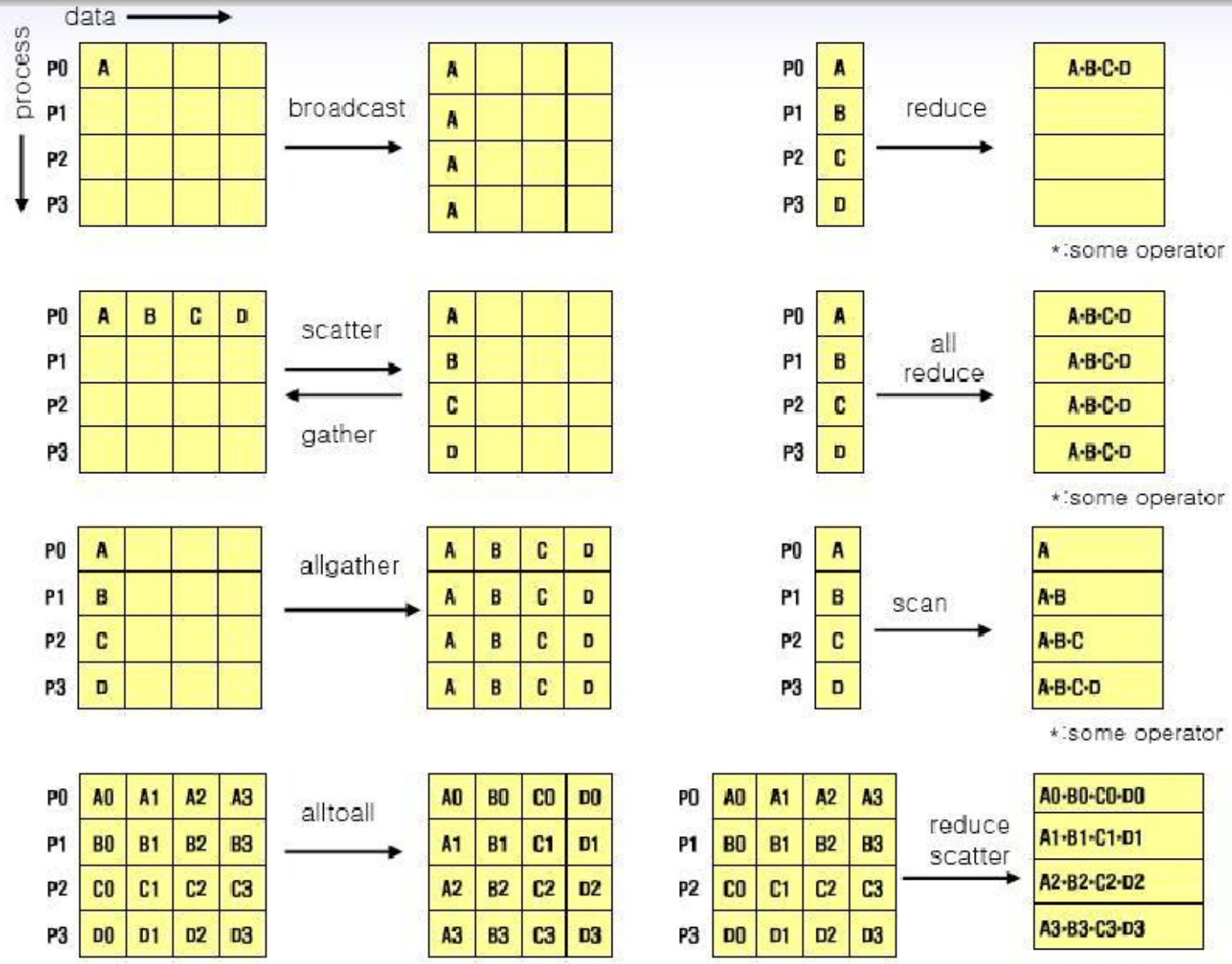
```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
# pass explicit MPI datatypes
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
```

Lignes directrices

- Les communications collectives permettent de faire en une seule opération une série de communications point à point.
- Une communication collective concerne toujours tous les processus du communicateur indiqué.
- Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée, au sens des communications point-à-point (donc quand la zone mémoire concernée peut être modifiée).
- Il est inutile d'ajouter une synchronisation globale (barrière) après une opération collective.
- La gestion des étiquettes dans ces communications est transparente et à la charge du système. Elles ne sont donc jamais définies explicitement lors de l'appel à ces sous-programmes. Cela a entre autres pour avantage que les communications collectives n'interfèrent jamais avec les communications point à point.

- Il y a trois types de sous-programmes (fonctions) :
  1. celui qui assure les synchronisations globales : MPI_BARRIER().
  2. 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().
  3. ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
     - opérations de réduction (somme, produit, maximum, minimum, etc.), qu'elles soient d'un type prédéfini ou d'un type personnel : MPI_REDUCE() ;
     - opérations de réduction avec diffusion du résultat (équivalent à un MPI_REDUCE() suivi d'un MPI_BCAST()) : MPI_ALLREDUCE().


data
process
P0
P1
P2
P3
A
broadcast
reduce
A·B·C·D
*:some operator
B
C
D
scatter
gather
all
reduce
allgather
scan
A·B
A·B·C
alltoall
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
D0
D1
D2
D3
reduce
scatter
A0·B0·C0·D0
A1·B1·C1·D1
A2·B2·C2·D2
A3·B3·C3·D3

exemple bcast C

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
int main(argc,argv){
   int i,myid, numprocs,source,count;
   int buffer[4];
   MPI_Status status; MPI_Request request;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   source=0;  count=4;
   if(myid == source){
      for(i=0;i<count;i++)
         buffer[i]=i;
   }
   MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);
   for(i=0;i<count;i++)
      printf("%d ",buffer[i]);
   printf("\n");
   MPI_Finalize();
}
```

exemple bcast python

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ( 'abc', 'xyz')}
else:
    data = None
data = comm.bcast(data, root=0)
```

Lignes directrices

- Une réduction est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur. Des exemples typiques sont la somme des éléments d'un vecteur SUM(A(:)) ou la recherche de l'élément de valeur maximum dans un vecteur MAX(V(:)).
- MPI propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus. Le résultat est obtenu sur un seul processus (MPI_REDUCE()) ou bien sur tous (MPI_ALLREDUCE(), qui est en fait équivalent à un MPI_REDUCE() suivi d'un MPI_BCAST()).
- Si plusieurs éléments sont concernés par processus, la fonction de réduction est appliquée à chacun d'entre eux.
- Le sous-programme MPI_SCAN() permet en plus d'effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du communicateur et lui-même.
- Les sous-programmes MPI_OP_CREATE() et MPI_OP_FREE() permettent de définir des opérations de réduction personnelles.

Nom	opération
MPI_SUM	Somme des éléments
MPI_PROD	Produit des éléments
MPI_MIN	Recherche du maximum
MPI_MAX	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

Exemple Scatter C

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4
int main(argc,argv){
  int numtasks, rank, sendcount, recvcount, source;
  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();
}
```

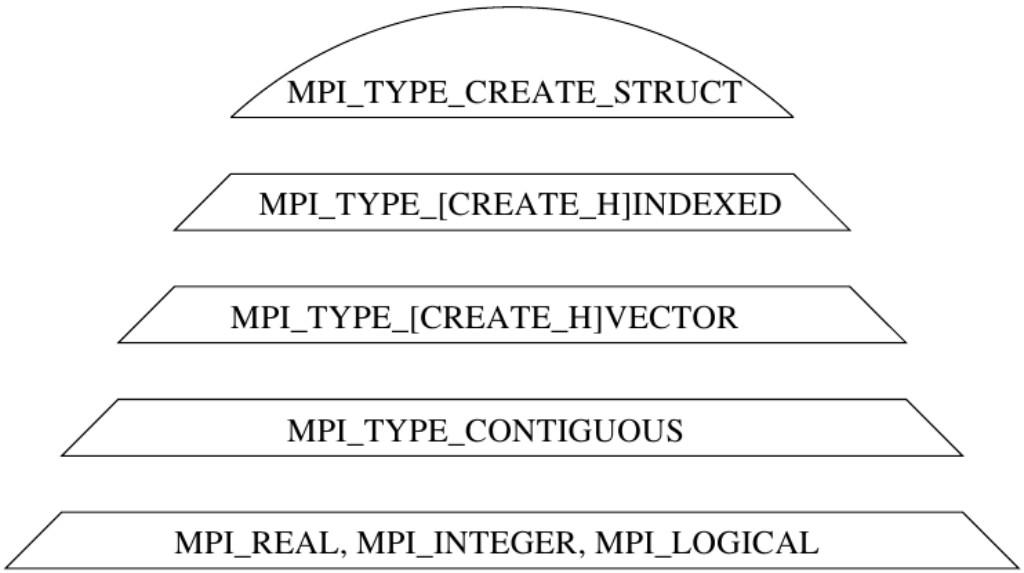
Exemple Scatter python

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```

- Dans les communications, les données échangées sont typées : MPI_INTEGER, MPI_REAL, MPI_COMPLEX, etc.
- On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que MPI_TYPE_CONTIGUOUS(), MPI_TYPE_VECTOR(), MPI_TYPE_CREATE_HVECTOR().
- A chaque fois que l'on crée un type de données, il faut le valider à l'aide du sous-programme MPI_TYPE_COMMIT().
- Si on souhaite réutiliser le même nom pour définir un autre type dérivé, on doit au préalable le libérer avec le sous-programme MPI_TYPE_FREE() .


MPI_TYPE_CREATE_STRUCT
MPI_TYPE_[CREATE_H]INDEXED
MPI_TYPE_[CREATE_H]VECTOR
MPI_TYPE_CONTIGUOUS
MPI_REAL, MPI_INTEGER, MPI_LOGICAL

- Pour compléter :
  - Copies de mémoire à mémoire.
  - Types de données dérivés.
  - Optimisations.
  - Communicateurs.
  - MPI-IO.
  - Gestion dynamique de processus.

Ressources

- Message Passing Interface Forum, MPI : A Message-Passing Interface Standard, Version 2.2, High Performance Computing Center Stuttgart (HLRS), 2009
  https ://fs.hlrs.de/projects/par/mpi/mpi22/
- William Gropp, Ewing Lusk et Anthony Skjellum : Using MPI : Portable Parallel Programming with the Message Passing Interface, second edition, MIT Press, 1999
- William Gropp, Ewing Lusk et Rajeev Thakur : Using MPI-2, MIT Press, 1999
- Peter S. Pacheco : Parallel Programming with MPI, Morgan Kaufman Ed., 1997
- Ian Foster, Designing and Building Parallel Programs : http ://www.mcs.anl.gov/itf/dbpp/

Ressources

- Lawrence Livermore Nationnal Laboratory :
  https ://computing.llnl.gov/tutorials/mpi
- Documentations complémentaires :
  - http ://www.mpi-forum.org/docs/
  - http ://www.mpi-forum.org/mpi2_1/index.htm
  - http ://www.mcs.anl.gov/research/projects/mpi/learning.html
  - http ://www.mpi-forum.org/archives/notes/notes.html
- MPICH2 :
  http ://www.mcs.anl.gov/research/projects/mpich2/
- Open MPI : http ://www.open-mpi.org/