Calcul parallèle – Librairie MPI

Xavier Juvigny

Calcul Haute Performance
ONERA

Mai-Juin 2007

Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d'envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d'envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d'envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d’envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Initialisation

Il s'agit de préciser que le programme qui s'éxécute n'est pas une tâche isolée mais un processus d'une application parallèle

- Initialisation (des processus parallèles) ;
- Terminaison (des processus parallèles) ;

Cela a un impact sur l'environnement d'exécution du processus.
On dispose de commandes spéciales pour lancer l'éxécution :

```
mpirun -np <nombre de processus>
       -machinefile <nœuds où les processus sont exécutés>
       nom du programme
```

Initialisation (programmation)

- Toute unité doit include l'entète MPI (`mpif.h` en fortran, `mpi.h` en C/C++) ;
- La routine `MPI_Init` permet d'initialiser l'environnement parallèle d'un processus :

En fortran

```
integer, intent(out) : : code_erreur
call MPI_INIT(code_erreur)
```

En C/C++

```
int code_erreur;
code_erreur=MPI_Init(&argc,&argv);
```

Terminaison

- Réciproquement, la routine `mpi_finalize` désactive l'environnement parallèle :

En fortran

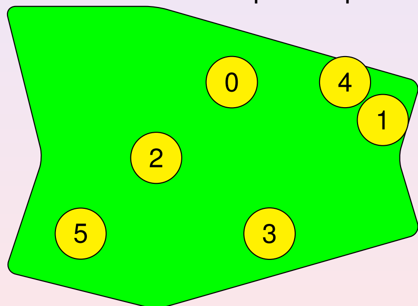
```
integer, intent(out) : : code_erreur
call MPI_FINALIZE(code_erreur)
```

En C/C++

```
int code_erreur;
code_erreur=MPI_Finalize();
```

Communicateurs

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



Communicateur MPI_COMM_WORLD

Interrogation de l'environnement

A tout instant, un processus doit pouvoir connaître le nombre de processus gérés par un communicateur. On utilise pour celà la routine `mpi_comm_size` :

En fortran

```
integer, intent(out) : : nb_procs, code_erreur
call
MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code_erreur)
```

En C/C++

```
int nb_procs, code_erreur;
code_erreur=MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
```

Interrogation de l'environnement (suite)

De même, il est utile que le processus puisse obtenir son rang dans l'environnement parallèle. On utilise pour celà la routine `mpi_comm_rank` :

En fortran

```
integer, intent(out) :: rang, code_erreur
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code_erreur)
```

En C/C++

```
int rang, code_erreur;
code_erreur=MPI_Comm_rank(MPI_COMM_WORLD,&rang);
```

Attention, le rang est comprise entre 0 et `nb_procs`-1

Exemple

Programme

```
program qui_je_suis

   implicit none
   include 'mpif.h'
   integer : : nb_procs,rang,code

   call MPI_INIT(code)
   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
   print *,'Je suis le processus ', rang,' parmi ', nb_procs
   call MPI_FINALIZE(code)

end program qui_je_suis
```

Exécution

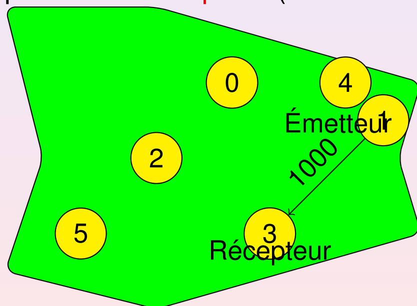
```
> mpirun -np 5 qui_je_suis

Je suis le processus 3 parmi 5
Je suis le processus 0 parmi 5
Je suis le processus 2 parmi 5
Je suis le processus 4 parmi 5
Je suis le processus 1 parmi 5
```

Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d'envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Définition

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



Genèse d'un message

- L'émetteur et le récepteur sont identifiés par leur rang dans le communicateur ;
- l'enveloppe d'un message est constituée :
  1. du rang de l'émetteur ;
  2. du rang du récepteur ;
  3. de l'étiquette (tag) du message ;
  4. du nom du communicateur qui définit le contexte de communication
- les données échangées sont typées ;
- Il existe plusieurs modes de transfert (protocoles)

Exemple de communications point à point

Programme

```
program point_a_point
   implicit none
   include 'mpif.h'
   integer status(MPI_STATUS_SIZE)
   integer : : rang,valeur,code,etiquette
   call MPI_INIT(code)
   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
   etiquette = 100
   if ( rang .EQ. 1 ) then
      valeur = 1000
      call MPI_SEND(valeur,1,MPI_INTEGER,3,etiquette, MPI_COMM_WORLD,code)
   else if ( rang .EQ. 3 ) then
      call MPI_RECV(valeur,1,MPI_INTEGER,1,etiquette, MPI_COMM_WORLD,status,code)
      print *,'Moi, processus 3, j'ai reçu ', valeur,' du processus 1.'
   end if
   call MPI_FINALIZE(code)
end program point_a_point
```

Exécution

```
> mpirun -np 5 point_a_point

Moi, processus 3, j'ai reçu 1000 du processus 1.
```

Même exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int etiquette = 100;
 int rang, valeur, code;
 MPI_Status status;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 if rang == 1
 {
    valeur = 1000;
    code = MPI_Send(&valeur,1,MPI_INT,3,etiquette,MPI_COMM_WORLD);
 }
 else if rang == 3
 {
    code = MPI_Recv(&valeur,1,MPI_INT,1,etiquette,MPI_COMM_WORLD,&status);
    printf("Moi, processus 3, j'ai reçu %d du processus 1",valeur);
 }
 code = MPI_Finalize();
}
```

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

Types de données de base (C/C++)

Type MPI	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	Type hétérogènes

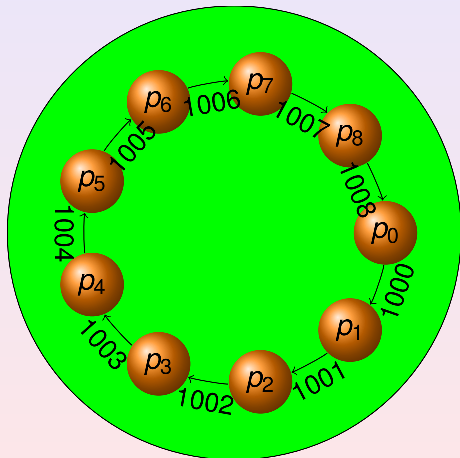
Autres possibilités

- À la réception d'un message, le rang et l'identificateur 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 est possible de créer des structures de données plus complexes à l'aide de sous-programmes spécifiques ;
- Il y a de multiples protocoles de transfert détaillés plus loin

Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d'envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Anneau de communication



Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d'envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Vision software

Il y a plusieurs protocoles d’envoi et de réception de messages. On peut les classer selon l’impact des requêtes (`send, receive`) sur le déroulement de l’application (vision *software*)

- non bloquant ;
- localement bloquant ;
- bloquant.

Vision hardware

La vision *hardware* conduit à distinguer plusieurs modes de fonctionnement du calculateur :

- mode asynchrone (chaque processeur est indépendant) ;
- mode synchrone (notion de contrôle centralisé).

Certains protocoles de communication sont rendus possibles par les détails de l'architecture matérielle.

Envoi synchrone localement bloquant

Le plus “standard” est l’asynchrone, localement bloquant :

- L’envoi ou la réception d’un message n’a pas d’impact sur le déroulement des autres processus (concernés ou non) ;
- il n’y a pas de notion de rendez-vous ;
- l’ordre n’est pas forcément respecté.

Notion de boîte aux lettres.

Principe de l'envoi asynchrone

Ce mécanisme suppose l'existence de zones mémoire "tampon" de taille a priori illimités (ce qui n'est jamais le cas en pratique).
Le principe est le suivant :

- Le sous-programme `MPI_Send` rend la main lorsque les données à émettre sont sauvegardées dans cette zone tampon.
- Le sous-programme `MPI_Recv` rend la main lorsque les données à recevoir sont recopiées de la zone tampon dans la zone "utilisateur".

Cela fonctionne bien comme un système de *boîtes aux lettres* dans lesquelles sont stockés les messages en attente d'émission ou de réception.

On peut interroger le système (`MPI_Probe` par exemple) pour savoir si un message avec une certaine étiquette est arrivé :

```
int MPI_Probe( int source, int tag, MPI_Comm comm, MPI_Status *status )
```

Programme modèle

Programme

```
# include "mpi.h"
# include <stdio.h>
# include <lapack.h>
void main(int narg, char** argv)
{
 const int na=256, nb=200, m=2048, etiquette=1111;
 float a[na][na], b[nb][nb], c[m][m], pivota[na], pivotb[nb];
 int nprocs, rang, ierr, info;
 int i,j;
 double tps_deb, tps_fin, tps_fin_max;
 MPI_Status status;
 // Initialisation MPI
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 // Initialisation des tableaux
 for ( i = 0; i < na; i++ )
     for ( j = 0; j < na; j++ )
         a[i][j] = 1+(int) (10.0*rand()/(RAND_MAX+1.0));
 for ( i = 0; i < nb; i++ )
     for ( j = 0; j < nb; j++ )
         b[i][j] = 1+(int) (10.0*rand()/(RAND_MAX+1.0));
 for ( i = 0; i < m; i++ )
     for ( j = 0; j < m; j++ )
         c[i][j] = 1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

Programme modèle (suite)

Programme (suite)

```
tps_deb = MPI_WTime();
if rang == 0
{
   code = MPI_Send(c,m*m,MPI_FLOAT,1,etiquette,
                   MPI_COMM_WORLD );
   // Je calcule : factorisation LU avec LAPACK
   sgetrf_(&na, &na, a, &na, pivota, &info );
   // On appelle une routine de produit matrice-matrice
   matmul(nb,a,b,c);
} else if ( rang == 1 )
{
   // Je calcule
   sgetrf_(&na, &na, a, &na, pivota, &info );
   // Je reçois le gros message
   code = MPI_Recv(c,m*m,MPI_FLOAT,0,etiquette,MPI_COMM_WORLD,&statut);
   // Ce calcul dépend du message précédent
   transpose(na, c, a );
   // Ce calcul est indépendant du message
   sgetrf_(&nb, &nb, b, &nb, pivotb, &info );
}
tps_fin = MPI_WTime() - tps_deb;
printf("Temps = %lg secondes",tps_fin);
code = MPI_Finalize();
}
```

Mesure performance programme modèle

Le temps de calcul mesuré est de 0.7 secondes.
Hors communications, le temps de calcul est de 0.15 secondes ce qui veut dire que les communications prennent environ 78% du temps global !
Le temps pris dans la communication provient principalement de la recopie dans la mémoire tampon des données.

Envoi bloquant synchrone

Ce protocole suppose un rendez–vous entre processus. Celà a un impact sur l'ensemble de l'application.
On distingue deux variantes :

- Rendez–vous "simple" : `MPI_Ssend` ne peut commencer à s'éxécuter que si le `MPI_Recv` correspondant a commencé.
- `MPI_Issend` permet d'initier un message synchrone et rendre la main à l'utilisateur pour continuer ses calculs... De même `MPI_Irecv` permet d'initier la réception d'un message dans un tableau ou variable et rend la main pour continuer les calculs.

Modification programme modèle avec rendez–vous simple

Programme (suite)

```
tps_deb = MPI_WTime();
if rang == 0
{
   code = MPI_Ssend(c,m*m,MPI_FLOAT,1,etiquette,
                    MPI_COMM_WORLD );
   // Je calcule : factorisation LU avec LAPACK
   sgetrf_(&na, &na, a, &na, pivota, &info );
   // Ce calcul modifie le contenu du tableau c
   matmul(nb,a,b,c);
} else if ( rang == 1 )
{
   // Je calcule
   sgetrf_(&na, &na, a, &na, pivota, &info );
   // Je reçois le gros message
   code = MPI_Recv(c,m*m,MPI_FLOAT,0,etiquette,MPI_COMM_WORLD,&statut);
   // Ce calcul dépend du message précédent
   transpose(na, c, a );
   // Ce calcul est indépendant du message
   sgetrf_(&nb, &nb, b, &nb, pivotb, &info );
}
tps_fin = MPI_WTime() - tps_deb;
printf("Temps = %lg secondes",tps_fin);
code = MPI_Finalize();
```

Performance pour rendez–vous simple

Le temps mesuré ici est alors de 0.36 secondes.
Sur certaines machines comme celle utilisée pour le test, on peut gagner par rapport au programme modèle un facteur 2 sur le temps pris par les échanges de message.
Cela est dû principalement au fait qu'il n'y a pas de recopie des données dans un buffer.

Modification programme modèle avec rendez–vous non simple

Programme (suite)

```
MPI_Request request;
...
if rang == 0 {
   code = MPI_Issend(c,m*m,MPI_FLOAT,1,etiquette,
                     MPI_COMM_WORLD );
   // Je calcule : factorisation LU avec LAPACK
   sgetrf_(&na, &na, a, &na, pivota, &info );
   // Ce calcul modifie le contenu du tableau c
   code = MPI_Wait(&request,&statut);
   matmul(nb,a,b,c);
} else if ( rang == 1 )
{ // Je calcule
   sgetrf_(&na, &na, a, &na, pivota, &info );
   // Je reçois le gros message
   code = MPI_Irecv(c,m*m,MPI_FLOAT,0,etiquette,MPI_COMM_WORLD,&statut);
   // Ce calcul est indépendant du message
   sgetrf_(&nb, &nb, b, &nb, pivotb, &info );
   // Ce calcul dépend du message précédent
   code = MPI_Wait(&request,&statut);
   transpose(na, c, a ); }
tps_fin = MPI_WTime() - tps_deb;
```

Performance pour rendez–vous non simple

Le temps mesuré ici est alors de 0.28 secondes.
On gagne ici un facteur 3 par rapport à la version initiale

Outils pour les messages synchrones

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`) si un message particulier est bien arrivé.

Conseils de programmation

- Éviter si possible la recopie temporaire des message en utilisant la fonction `MPI_Ssend` ;
- Chevaucher les communications avec les calculs en évitant la recopie temporaire des messages en utilisant les fonctions non bloquantes `MPI_Issend` et `MPI_Irecv`.

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 pénalisante à la longue d'où l'intérêt des communications persistantes

Définition d'une communication persistante

Elle consiste à :

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

envoi *standard*	`MPI_Send_init`
envoi *synchrone*	`MPI_Ssend_init`
envoi *bufférisé*	`MPI_Bsend_init`
réception *standard*	`MPI_Recv_init`

Rappel programme stantard avec rendez–vous non simple

Programme standard

```
if rang == 0
{
   for( k = 1; k < 1000; k++ )
   {
      code = MPI_Issend(c,m*m,MPI_FLOAT,1,etiquette,
                        MPI_COMM_WORLD,&request );
      sgetrf_(&na, &na, a, &na, pivota, &info );
      code = MPI_Wait(&request,&statut);
      matmul(nb,a,b,c);
   }
} else if ( rang == 1 )
{
   for( k = 1; k < 1000; k++ )
   {
      sgetrf_(&na, &na, a, &na, pivota, &info );
      code = MPI_Irecv(c,m*m,MPI_FLOAT,0,etiquette,MPI_COMM_WORLD,&request );
      sgetrf_(&nb, &nb, b, &nb, pivotb, &info );
      code = MPI_Wait(&request,&statut);
      transpose(na, c, a );
   }
}
```

Mesure du temps pour le programme exemple

Exécution

```
> mpirun -np 2 NonBloquant2
```

Temps : 235 secondes

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

Programme stantard avec communication persistante

Programme standard

```
if rang == 0
{
   code = MPI_Ssend_init(c,m*m,MPI_FLOAT,1,etiquette,
                        MPI_COMM_WORLD,&request );
   for( k = 1; k < 1000; k++ )
   {
      code = MPI_Start(&request);
      sgetrf_(&na, &na, a, &na, pivota, &info );
      code = MPI_Wait(&request,&statut);
      matmul(nb,a,b,c);
   }
} else if ( rang == 1 )
{
   code = MPI_Recv_init(c,m*m,MPI_FLOAT,0,etiquette,MPI_COMM_WORLD,&statut,&request);
   for( k = 1; k < 1000; k++ )
   {
      sgetrf_(&na, &na, a, &na, pivota, &info );
      // Je reçois le gros message
      code = MPI_Start(&request);
      sgetrf_(&nb, &nb, b, &nb, pivotb, &info );
      code = MPI_Wait(&request,&statut);
      transpose(na, c, a );
   }
}
```

Mesure du temps

Exécution

```
> mpirun -np 2 persistant
```

Temps : 195 secondes

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, il faut auparavant libérer la requête `request` associée à l'ancien schéma en appelant la fonction `MPI_Request_free(&request)` ;
- Cette fonctions ne libérera la requête `request` qu'une fois que la communication associée sera réellement terminée

Conseils 2

- Minimiser les surcoûts induits par des appels répétitifs aux fonctions de communication en utilisant une fois pour toute un schéma persistant de communication et activer celui-ci autant de fois qu'il est nécessaire à l'aide de la fonction `MPI_Start` ;
- Chevaucher les communications avec les calculs tout en évitant la recopie temporaire des messages car un schéma persistante (ex. `MPI_Ssend_init`) est forcément activé d'une façon non bloquante à l'appel de la fonction `MPI_Start`.

Plan

1. Environnement
2. Communications point à point
   - Notions générales
   - Exemple : Anneau de communication
   - Protocoles d'envoi et de réception
3. Communications collectives
   - Notions générales
4. Les communicateurs

Définition

- Les communications collectives permettent de faire en une seule opération une série de communications point à point ;
- Une communication collective concerne toujours les processus d'un communicateur donné ;
- La gestion des étiquettes est transparente ;

Types de communications collectives

Elles sont de trois types :

1. les synchronizations globales
2. les transferts de données :
   - Diffusion globale de données ;
   - Diffusion sélective de données ;
   - Collecte de données réparties ;
   - collecte par tous de données réparties ;
   - diffusion sélective par tous de données réparties
3. les opérations sur les transferts de données
   - Opérations de réductions
   - Opérations de réductions et diffusion

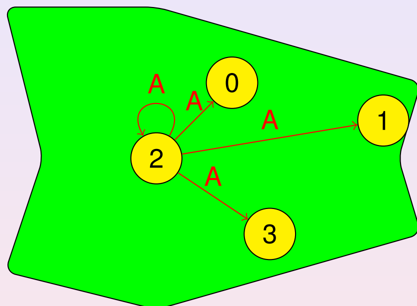
Synchronisation

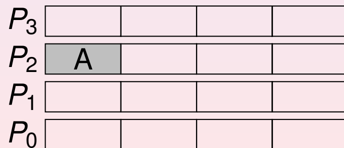
Donne un point de rendez-vous à tous les processeur du communicateur

Exemple en C

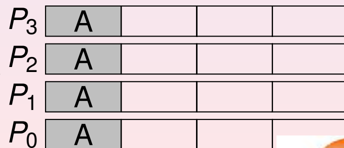
```
int code;
code = MPI_Barrier(MPI_COMM_WORLD);
```

Diffusion général : MPI_BCAST





MPI_BCAST



Exemple en fortran

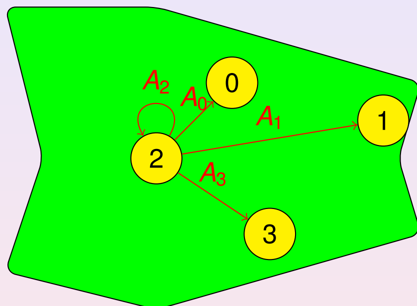
Programme

```
program bcast
   implicit none
   include 'mpif.h'
   integer : : rang,valeur,code,nb_proc
   call MPI_INIT(code)
   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
   if ( rang .EQ. 2 ) valeur = rang+1000
   call MPI_BCAST(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)
   print *,'Moi, processus ', rang,' , j''ai reçu ', valeur, 'du processus 2'
   call MPI_FINALIZE(code)
end program bcast
```

Exécution

```
> 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.
```

Diffusion sélective : MPI_SCATTER



P_3				
P_2	A_0	A_1	A_2	A_3
P_1				
P_0				

→ MPI_SCATTER

P_3	A_3			
P_2	A_2			
P_1	A_1			
P_0	A_0			

Exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int nb_valeurs = 128;
 int rang, longueur_tranche, i, code, nb_procs;
 float* valeurs, donnees;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 longueur_tranche = nb_valeurs/nb_procs;
 donnees = malloc(longueur_tranche*sizeof(float));
 if rang == 2
 {
    valeurs = malloc(nb_valeurs*sizeof(float));
    for ( i = 0; i < nb_valeurs; i++ )
        valeurs[i] = 1000+i;
 }
 code = MPI_Scatter(valeurs,longueur_tranche,MPI_FLOAT,
                    donnees,longueur_tranche,MPI_FLOAT,2,MPI_COMM_WORLD );
 printf("Moi, processus %d, j'ai reçu %f à %f du processus 2",rang,
        donnees[0],donnees[longueur_tranche-1]);
 code = MPI_Finalize();
}
```

Sortie de l'exemple

Exécution

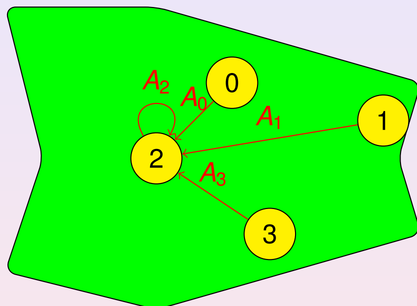
```
> mpirun -np 4 ex_scatter

Moi, processus 0, j'ai reçu xxxx à xxxx du processus 2.
Moi, processus 1, j'ai reçu xxxx à xxxx du processus 2.
Moi, processus 3, j'ai reçu xxxx à xxxx du processus 2.
Moi, processus 2, j'ai reçu xxxx à xxxx du processus 2.
```

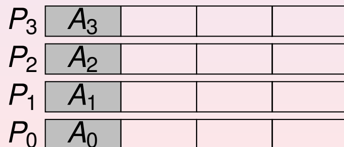
Sortie de l'exemple

Exécution

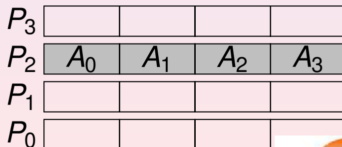
```
> mpirun -np 4 ex_scatter

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.
```

Collecte : MPI_GATHER





MPI_GATHER



Exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int nb_valeurs = 128;
 int rang, longueur_tranche, i, code, nb_procs;
 float* valeurs, donnees;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 longueur_tranche = nb_valeurs/nb_procs;
 donnees = malloc(nb_valeurs*sizeof(float));
 valeurs = malloc(longueur_tranche*sizeof(float));
 for ( i = 0; i < longueur_tranche; i++ )
     valeurs[i] = 1000+rang*longueur_tranche+i;
 code = MPI_Gather(valeurs,longueur_tranche,MPI_FLOAT,
                   donnees,longueur_tranche,MPI_FLOAT,2,MPI_COMM_WORLD );
 if rang == 2
    printf("Moi, processus %d, j'ai reçu %f ... %f ... %f",
           donnees[0],donnees[longueur_tranche],donnees[nb_valeurs]);
 code = MPI_Finalize();
}
```

Sortie de l'exemple

Exécution

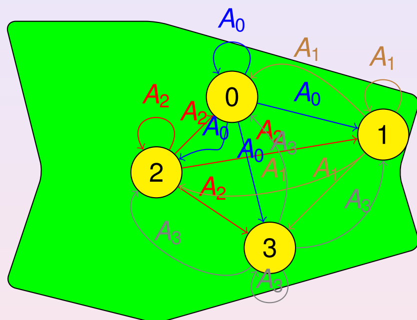
```
> mpirun -np 4 ex_gather

Moi, processus 2, j'ai reçu xxxx ... xxxx ... xxxx
```

Sortie de l'exemple

Exécution

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

Collecte générale : MPI_ALLGATHER



P_3	A_3			
P_2	A_2			
P_1	A_1			
P_0	A_0			

$\xrightarrow{\text{MPI_ALLGATHER}}$

P_3	A_0	A_1	A_2	A_3
P_2	A_0	A_1	A_2	A_3
P_1	A_0	A_1	A_2	A_3
P_0	A_0	A_1	A_2	A_3

Exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int nb_valeurs = 128;
 int rang, longueur_tranche, i, code, nb_procs;
 float* valeurs, donnees;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 longueur_tranche = nb_valeurs/nb_procs;
 donnees = malloc(nb_valeurs*sizeof(float));
 valeurs = malloc(longueur_tranche*sizeof(float));
 for ( i = 0; i < longueur_tranche; i++ )
     valeurs[i] = 1000+rang*longueur_tranche+i;
 code = MPI_Allgather(valeurs,longueur_tranche,MPI_FLOAT,
                      donnees,longueur_tranche,MPI_FLOAT,MPI_COMM_WORLD );
 printf("Moi, processus %d, j'ai reçu %f ... %f ... %f",
        donnees[0],donnees[longueur_tranche],donnees[nb_valeurs]);
 code = MPI_Finalize();
}
```

Sortie de l'exemple

Exécution

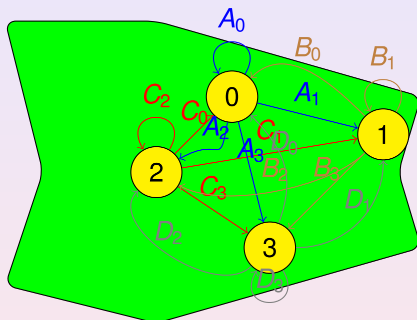
```
> mpirun -np 4 ex_allgather

Moi, processus 1, j'ai reçu xxxx ... xxxx ... xxxx
Moi, processus 3, j'ai reçu xxxx ... xxxx ... xxxx
Moi, processus 2, j'ai reçu xxxx ... xxxx ... xxxx
Moi, processus 0, j'ai reçu xxxx ... xxxx ... xxxx
```

Sortie de l'exemple

Exécution

```
> mpirun -np 4 ex_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
```

Échanges croisés : MPI_ALLTOALL



P_3	D_0	D_1	D_2	D_3
P_2	C_0	C_1	C_2	C_3
P_1	B_0	B_1	B_2	B_3
P_0	A_0	A_1	A_2	A_3

$\xrightarrow{\texttt{MPI_ALLTOALL}}$

P_3	A_3	B_3	C_3	D_3
P_2	A_2	B_2	C_2	D_2
P_1	A_1	B_1	C_1	D_1
P_0	A_0	B_0	C_0	D_0

Exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int nb_valeurs = 128;
 int rang, longueur_tranche, i, code, nb_procs;
 float* valeurs, donnees;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 longueur_tranche = nb_valeurs/nb_procs;
 donnees = malloc(nb_valeurs*sizeof(float));
 valeurs = malloc(longueur_tranche*sizeof(float));
 for ( i = 0; i < longueur_tranche; i++ )
     valeurs[i] = 1000+rang*longueur_tranche+i;
 code = MPI_Alltoall(valeurs,longueur_tranche,MPI_FLOAT,
                       donnees,longueur_tranche,MPI_FLOAT,MPI_COMM_WORLD );
 printf("Moi, processus %d, j'ai reçu %f ... %f ... %f",
        donnees[0],donnees[longueur_tranche],donnees[nb_valeurs]);
 code = MPI_Finalize();
}
```

Sortie de l'exemple

Exécution

```
> mpirun -np 4 ex_mpialltoall

Moi, processus 1, j'ai reçu xxxx ... xxxx ... xxxx
Moi, processus 3, j'ai reçu xxxx ... xxxx ... xxxx
Moi, processus 2, j'ai reçu xxxx ... xxxx ... xxxx
Moi, processus 0, j'ai reçu xxxx ... xxxx ... xxxx
```

Sortie de l'exemple

Exécution

```
> mpirun -np 4 ex_mpialltoall

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

Réductions réparties

- Une réduction est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur (par exemple la somme des éléments ou le maximum) ;
- *MPI* propose des sous–programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus avec récupération du résultat sur un seul processus (MPI_REDUCE) ou bien sur tous (MPI_ALLREDUCE)

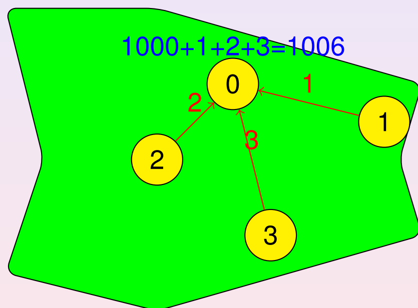
Réductions réparties (suite)

- Si plusieurs éléments sont concernés par processus, la fonction de réduction est d'abord appliquée à chacun d'entre eux en faisant les réductions localement sur chaque processus ;
- Le sous–programme `MPI_SCAN` permet d'effectuer en plus des réductions partielles en considérant pour chaque processus les processus précédents du groupe ;
- Les sous–programe `MPI_OP_CREATE` et `MPI_OP_FREE` permettent de définir des opérations de réduction personelles.

Principales opérations de réduction

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_XOR	Ou exclusif logique

Réduction répartie (Somme)



Exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 int rang, valeur, somme, code, nb_procs;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 if rang == 0
    valeur = 1000;
 else
    valeur = rang;
 code = MPI_Reduce(valeurs,somme,1,MPI_INT,
                   MPI_SUM,0,MPI_COMM_WORLD );
 if rang == 0
    printf("Moi, processus %d, j'ai pour valeur de somme globale %d",
           rang, somme);
 code = MPI_Finalize();
}
```

Sortie de l'exemple

Exécution

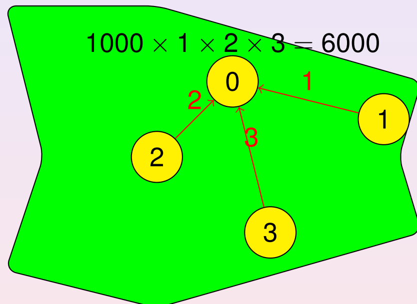
```
> mpirun -np 7 ex_mpireduce
Moi, processus 0, j'ai pour valeur de somme globale xxxx
```

Sortie de l'exemple

Exécution

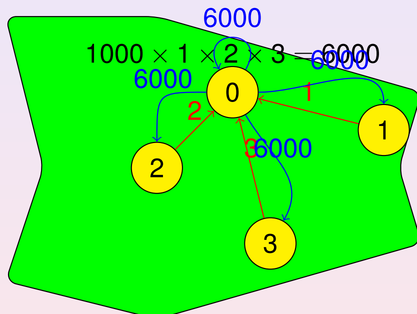
```
> mpirun -np 7 ex_mpireduce
Moi, processus 0, j'ai pour valeur de somme globale
1021
```

Réduction répartie avec diffusion(Produit)



Équivalent à un `MPI_Gather` suivit d'un `MPI_Bcast`.

Réduction répartie avec diffusion(Produit)



Équivalent à un `MPI_Gather` suivit d'un `MPI_Bcast`.

Exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 int rang, valeur, produit, code, nb_procs;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 if rang == 0
    valeur = 10;
 else
    valeur = rang;
 code = MPI_Allreduce(valeurs,produit,1,MPI_INT,
                      MPI_PROD,MPI_COMM_WORLD );
 printf("Moi, processus %d, j'ai pour valeur le produit global %d",
        rang, produit);
 code = MPI_Finalize();
}
```

Sortie de l'exemple

Exécution

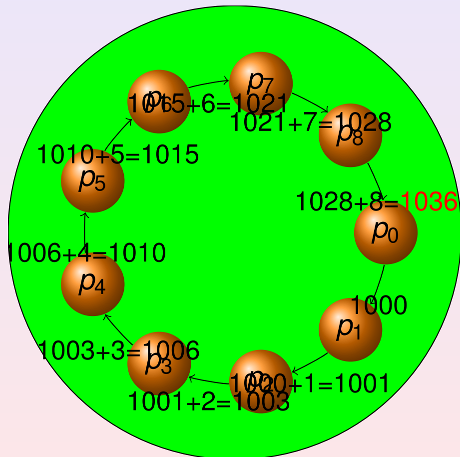
```
> mpirun -np 7 ex_mpiallreduce

Moi, processus 6, j'ai pour valeur de somme globale xxxx
Moi, processus 2, j'ai pour valeur de somme globale xxxx
Moi, processus 0, j'ai pour valeur de somme globale xxxx
Moi, processus 4, j'ai pour valeur de somme globale xxxx
Moi, processus 5, j'ai pour valeur de somme globale xxxx
Moi, processus 3, j'ai pour valeur de somme globale xxxx
Moi, processus 1, j'ai pour valeur de somme globale xxxx
```

Sortie de l'exemple

Exécution

```
> mpirun -np 7 ex_mpiallreduce

Moi, processus 6, j'ai pour valeur de somme globale 7200
Moi, processus 2, j'ai pour valeur de somme globale 7200
Moi, processus 0, j'ai pour valeur de somme globale 7200
Moi, processus 4, j'ai pour valeur de somme globale 7200
Moi, processus 5, j'ai pour valeur de somme globale 7200
Moi, processus 3, j'ai pour valeur de somme globale 7200
Moi, processus 1, j'ai pour valeur de somme globale 7200
```

Réduction répartie avec calculs partiels



Exemple en C

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 int rang, valeur, somme, code, nb_procs;
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 if rang == 0
    valeur = 1000;
 else
    valeur = rang;
 code = MPI_Scan(valeurs,somme,1,MPI_INT,
                 MPI_SUM,0,MPI_COMM_WORLD );
 printf("Moi, processus %d, j'ai pour valeur de la somme globale %d",
        rang, somme);
 code = MPI_Finalize();
}
```

Sortie de l'exemple

Exécution

```
> mpirun -np 7 allgather

Moi, processus 6, j'ai pour valeur de somme globale xxxx
Moi, processus 2, j'ai pour valeur de somme globale xxxx
Moi, processus 0, j'ai pour valeur de somme globale xxxx
Moi, processus 4, j'ai pour valeur de somme globale xxxx
Moi, processus 5, j'ai pour valeur de somme globale xxxx
Moi, processus 3, j'ai pour valeur de somme globale xxxx
Moi, processus 1, j'ai pour valeur de somme globale xxxx
```

Sortie de l'exemple

Exécution

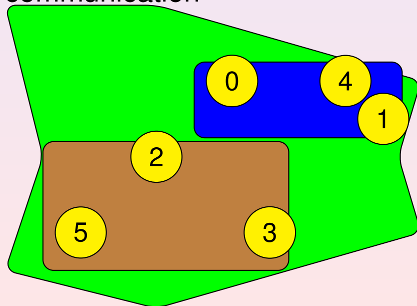
```
> mpirun -np 7 allgather

Moi, processus 6, j'ai pour valeur de somme globale 1021
Moi, processus 2, j'ai pour valeur de somme globale 1003
Moi, processus 0, j'ai pour valeur de somme globale 1000
Moi, processus 4, j'ai pour valeur de somme globale 1010
Moi, processus 5, j'ai pour valeur de somme globale 1015
Moi, processus 3, j'ai pour valeur de somme globale 1006
Moi, processus 1, j'ai pour valeur de somme globale 1001
```

Compléments

Les sous–programmes `MPI_Scatterv`, `MPI_Gatherv`, `MPI_Allgatherv` et `MPI_Alltoallv` étendent respectivement `MPI_Scatter`,`MPI_Gather`,`MPI_Allgather`,`MPI_Alltoall` au cas où le nombre d'éléments à diffuser ou collecter est différent suivant les processus.

Principe

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



Création d'un communicateur

- On ne peut créer un communicateur qu'à partir d'un autre communicateur,
- Fort heureusement, un communicateur est fourni par défaut : l'identificateur `MPI_COMM_WORLD` défini dans le fichier d'en-tête ;
- 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 la portée des communications point à point et collectives à tous les processus de l'application

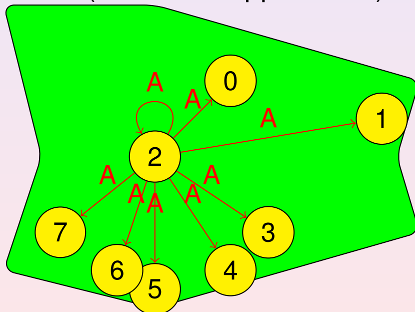
Exemple de diffusion via MPI_COMM_WORLD

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int m = 16;
 int i, rang, code, nb_procs;
 float a[m];
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 for ( i = 0; i < m; i++ ) a[i] = 0.;
 if rang == 2
    for ( i = 0; i < m; i++ ) a[i] = 2.;
 code = MPI_Bcast(a,n,MPI_FLOAT,2,MPI_COMM_WORLD);
 code = MPI_Finalize();
}
```

Exécution de l'exemple

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).

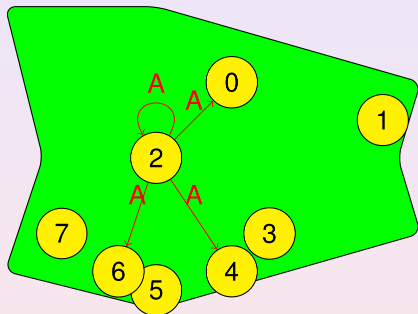



```
> mpirun -np 8 monde
```

Création d'un sous-groupe

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 leur diffuse à eux seuls le message.


0
1
2
3
4
5
6
7
A
A
A
A

Groupes et communicateurs

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

1. Construire des groupes de processus :
   - `MPI_Group_incl`;
   - `MPI_Groupe_excl`.
2. Construire des communicateurs :
   - `MPI_Cart_create`;
   - `MPI_Cart_sub`;
   - `MPI_Comm_create`;
   - `MPI_Comm_dup`;
   - `MPI_Comm_merge`;
   - `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 est 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`.

En pratique, pour construire un communicateur, il existe deux façons de procéder :

1. par l'intermédiaire d'un groupe de processus ;
2. directement à partir d'un autre communicateur.

Communicateur issu d'un groupe

- Un groupe est un ensemble ordonné de *N* processus ;
- Chaque processus du groupe est identifié par un entier 0,1,. . . ,N-1 appelé rang ;
- Initialement, tous les processus sont membre d'un groupe associé au communicateur par défaut `MPI_COMM_WORLD` ;
- Des sous–programmes *MPI* (`MPI_Group_size`, `MPI_Group_rank`, etc.) permettent de connaître les attributs d'un groupe ;
- Tout communicateur est associé à un groupe. Le sous–programme `MPI_Comm_group` permet de connaître le groupe auquel un communicateur est associé ;
- La construction d'un groupe est une étape intermédiaire pour créer un nouveau communicateur.

Dans l'exemple suivant, on va :

- regrouper les processus de rang pair dans un communicateur ;
- ne diffuser un message qu'aux processus de ce groupe.

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int m = 16;
 int i, rang, code, nb_procs, iproc,n;
 int rang_ds_monde, rang_ds_pair;
 MPI_Group grp_monde,grp_pair;
 int* rangs_pair;
 MPI_Comm comm_pair;
 float a[m];
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 for ( i = 0; i < m; i++ ) a[i] = 0.;
 if rang == 2
    for ( i = 0; i < m; i++ ) a[i] = 2.;
 // enregistrer le rang des processus pairs
 n = nprocs/2;
 rangs_pair = malloc(n*sizeof(int));
 i = 0;
 for( rang = 0; rang < nb_procs; rang += 2 )
 {
    i = i + 1;
    rangs_pair[i] = rang;
 }
```

Programme (suite)

```
    // Connaître le groupe associé à MPI_COMM_WORLD
    code = MPI_Comm_group(MPI_COMM_WORLD,&grp_monde);
    // Créer le groupe des processus pairs
    code = MPI_Group_incl(grp_monde,n,rangs_pair, &grp_pair);
    // Trouver le rang du processus 2 dans le groupe pair.
    rang_ds_monde = 2;
    code = MPI_Group_translate_ranks(grp_monde, 1, &rang_ds_monde,grp_pair,&rang_ds_pair);
    // Créer le communicateur des processus pairs
    code = MPI_Comm_create(MPI_COMM_WORLD,grp_pair,&comm_pair);
    // Une fois le communicateur créé, on détruit le groupe qui lui a donné naissance
    code = MPI_Group_free(&grp_pair);
    // Diffuser le message seulement aux processus de rang pair
    ifrang%2 == 0
    {
      code = MPI_Bcast(a,n,MPI_FLOAT,rang_ds_pair,MPI_COMM_WORLD);
      // Destruction du communicateur
      code = MPI_Comm_free(&comm_pair);
    }
    code = MPI_Finalize();
}
```

Contexte de communication

Un communicateur est constitué :

1. d'un groupe de processus ;
2. d'un contexte de communication mis en place à l'appel de la fonction de construction du communicateur.

- Le groupe constitue un sous–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 : attribut "caché").

Essayons de reprendre le programme en créant un groupe pour les processus impairs et un groupe pour les processus pairs.

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int m = 16;
 int i, rang, code, nb_procs, iproc,n;
 int rang_ds_monde, rang_ds_pair, rang_ds_impair;
 MPI_Group grp_monde,grp_pair, grp_impair;
 int* rangs_pair;
 MPI_Comm comm_pair, comm_impair;
 float a[m];
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 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] = 5.;
 // enregistrer le rang des processus pairs
 n = nprocs/2;
 rangs_pair = malloc(n*sizeof(int));
 i = 0;
 for( rang = 0; rang < nb_procs; rang += 2 )
 {  i = i + 1;
    rangs_pair[i] = rang; }
```

Programme (suite)

```
// Connaître le groupe associé à MPI_COMM_WORLD
code = MPI_Comm_group(MPI_COMM_WORLD,&grp_monde);
// Créer le groupe des processus pairs
code = MPI_Group_incl(grp_monde,n,rangs_pair, &grp_pair);
// Trouver le rang du processus 2 dans le groupe pair.
rang_ds_monde = 2;
code = MPI_Group_translate_ranks(grp_monde, 1, &rang_ds_monde,grp_pair,&rang_ds_pair);
// Créer le communicateur des processus pairs
code = MPI_Comm_create(MPI_COMM_WORLD,grp_pair,&comm_pair);
// Créer le groupe des processus impairs.
code = MPI_Group_excl(grp_monde,n,rangs_ds_pair,&grp_impair);
// Trouver le rang du processus 3 dans le groupe impair.
rang_ds_monde = 3;
MPI_Group_translate_ranks(grp_monde, 1, &rang_ds_monde,grp_impair,&rang_ds_impair);
// Créer le communicateur des processus impairs
code = MPI_Comm_create(MPI_COMM_WORLD,grp_impair,&comm_impair);
// Diffuser le message seulement aux processus de rang pair
ifrang % 2 == 0
{
  code = MPI_Bcast(a,n,MPI_FLOAT,rang_ds_pair,MPI_COMM_WORLD);
  // Destruction du communicateur pair
  code = MPI_Comm_free(&comm_pair);
} else {
  code = MPI_Bcast(a,n,MPI_FLOAT,rang_ds_impair,MPI_COMM_WORLD);
  // Destruction du communicateur impair
  code = MPI_Comm_free(&comm_impair);
}
code = MPI_Finalize();
}
```

Une fois les groupes (ou les communicateurs) constitués, il est possible :

- De les comparer :
  `MPI_Group_compare`(group1,group2,&resultat) ;
  `MPI_Comm_compare`(comm1,comm2,&resultat) ;
  Le résultat (un entier) peut être : `MPI_IDENT` ou `MPI_CONGRUENT` ou `MPI_SIMILAR` sinon `MPI_UNEQUAL`.
- 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`.

Exemple : Appliquer ces opérations aux groupes pair et impair.

Le programme précédent présente cependant quelques inconvénients. Nous allons voir en réalité que :

- nous n'étions pas contraint de nommer différemment ces deux communicateurs ;
- nous n'étions pas obligés de passer par des groupes pour construire les communicateurs ;
- nous n'étions pas non plus contraint de laisser le soin à MPI d'ordonner le rang des processus dans les communicateurs ;
- pour des raisons de lisibilité et de performances, nous aurions pu éviter les tests conditionnels, en particuliers lors de l'appel au sous–programme `MPI_Bcast`.

Le sous–programme `MPI_Comm_split` permet de partionner un communicateur donné en autant de communicateurs que l'on veut...

Syntaxe

```
const int comm, couleur, clef;
int nouveau_comm, ierr;
ierr = MPI_Comm_split(comm, couleur, clef,
&nouveau_comm);
```

rang	0	1	2	3	4	5	6	7
couleur	0	*	3	0	3	0	2	3
clef	2	1	0	0	1	3	0	1

$* \equiv$ MPI_UNDEFINED.

Exemple avec les communicateurs pair et impair

rang	0	1	2	3	4	5	6	7
couleur	0	1	0	1	0	1	0	1
clef	1	1	0	0	4	5	6	7

Programme

```
# include "mpi.h"
# include <stdio.h>
void main(int narg, char** argv)
{
 const int m = 16;
 int clef, couleur;
 int rang, code, nb_procs;
 MPI_Comm comm;
 float a[m];
 code = MPI_Init(&narg, &argv);
 code = MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
 code = MPI_Comm_rank(MPI_COMM_WORLD,&rang);
 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] = 5.;
 if rang%2 == 0
 {
    // Couleur et rang des processus pairs
    couleur = 0;
    clef = ( rang == 2 ? 0 : rang == 0 ? 1 : rang );
 }
 else
 {
    // Couleur et rang des processus impairs
    couleur = 1;
    clef = ( rang == 3 ? 0 : rang );
 }
```

Programme (suite)

```
    // Créer les communicateurs pair et impair
    code = MPI_Comm_split(MPI_COMM_WORLD,couleur,clef,&comm);
    // Le processus 0 de chaque communicateur diffuse
    // son message aux processus de son groupe
    code = MPI_Bcast(a,n,MPI_FLOAT,0,MPI_COMM_WORLD);
    code = MPI_Comm_free(&comm);
    code = MPI_Finalize();
}
```

Conclusions

- 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 message ;
- Ils permettent d'éviter les confusions lors de la sélection des messages, par exemple, au moment de l'appel d'une fonction d'une bibliothèque scientifique qui elle-même effectue des échanges de message ;
- Les communicateurs offrent une programmation modulaire du point de vue de l'espace de communication.