

Langages du parallélisme

Message Passing Interface

1. Rappels

2. Communications collectives

1. Communications bloquantes

2. Communications non bloquantes

Rappels

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

➔ Retourne la taille du communicateur comm dans size

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

➔ Retourne le rang dans le communicateur comm dans rank

Communications point à point bloquantes

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

```
int MPI_Recv( void* buf, int count, MPI_Datatype datatype,  
              int source, int tag, MPI_Comm comm, MPI_Status *status )
```

Rappels

Communications point à point non bloquantes

```
int MPI_Isend( void* buf, int count, MPI_Datatype datatype,  
              int dest, int tag, MPI_Comm comm, MPI_Request *request )  
int MPI_Irecv( void* buf, int count, MPI_Datatype datatype,  
              int source, int tag, MPI_Comm comm, MPI_Request *request )  
int MPI_Wait ( MPI_Request *req, MPI_Status *status )
```

```
int MPI_Waitany/Waitsome/Waitall(...)  
int MPI_Test/Testany/Testsome/Testall(...)
```

Rappels

Communications persistantes

```
int MPI_Send_init( void* buf, int count, MPI_Datatype datatype,  
                  int dest, int tag, MPI_Comm comm, MPI_Request *request )  
int MPI_Recv_init( void* buf, int count, MPI_Datatype datatype,  
                  int source, int tag, MPI_Comm comm, MPI_Request *request )
```

`MPI_Start(...), MPI_Startall(...)`

`MPI_Wait(...)`

`MPI_Request_free(...)`

Message Passing Interface

1. Rappels

2. Communications collectives

1. Communications bloquantes
2. Communications non bloquantes

Message Passing Interface

1. Rappels

2. Communications collectives

1. Communications bloquantes

2. Communications non bloquantes

Communications collectives

- ➔ Elles permettent de faire en une opération une série de communications point à point.
- ➔ Elles impliquent **tous les processus d'un communicateur** indiqué.

- Point à point:

One-to-One

- Collective:

One-to-All (MPI_Bcast, MPI_Scatter, MPI_Scatterv)

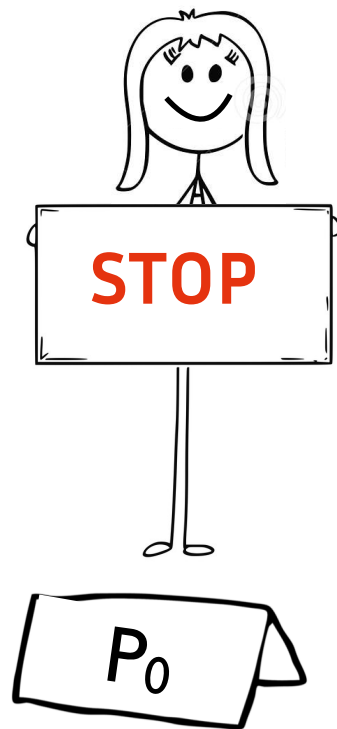
All-to-One (MPI_Gather, MPI_Gatherv, MPI_Reduce)

All-to-All (MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Alltoallv,
MPI_Allreduce, MPI_Reduce_scatter)

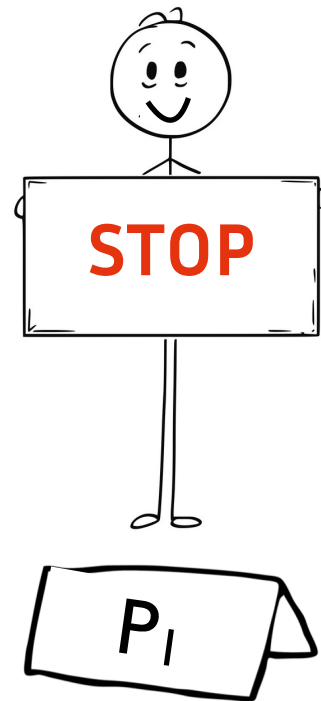
Barrière

Synchronise tous les processus dans un communicateur

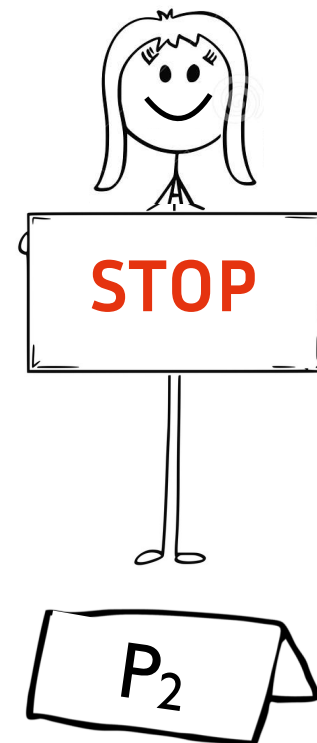
`MPI_Barrier`



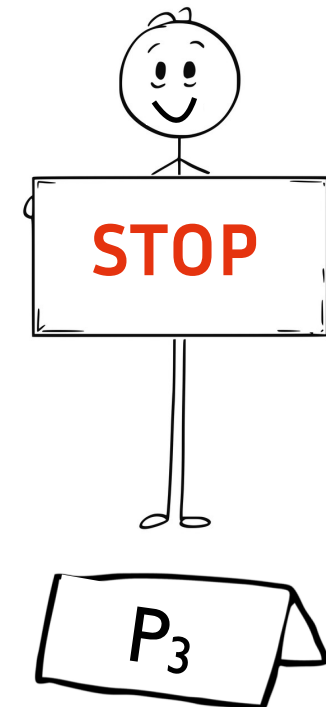
`MPI_Barrier`



`MPI_Barrier`



`MPI_Barrier`

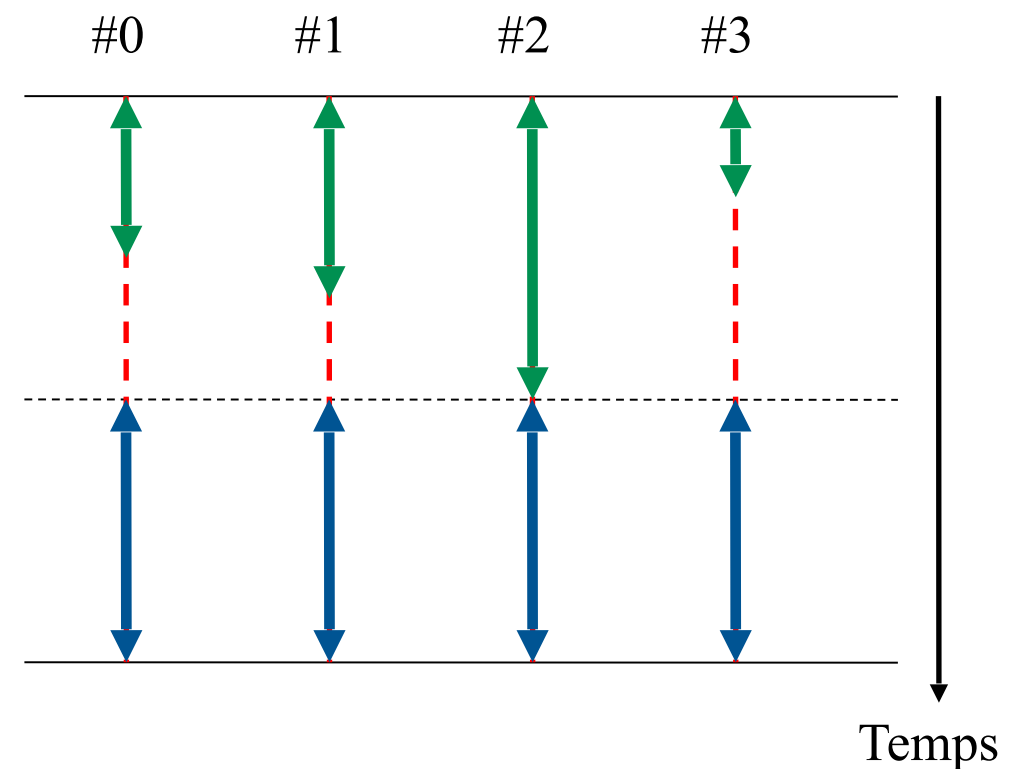


Barrière

Synchronise tous les processus dans un communicateur

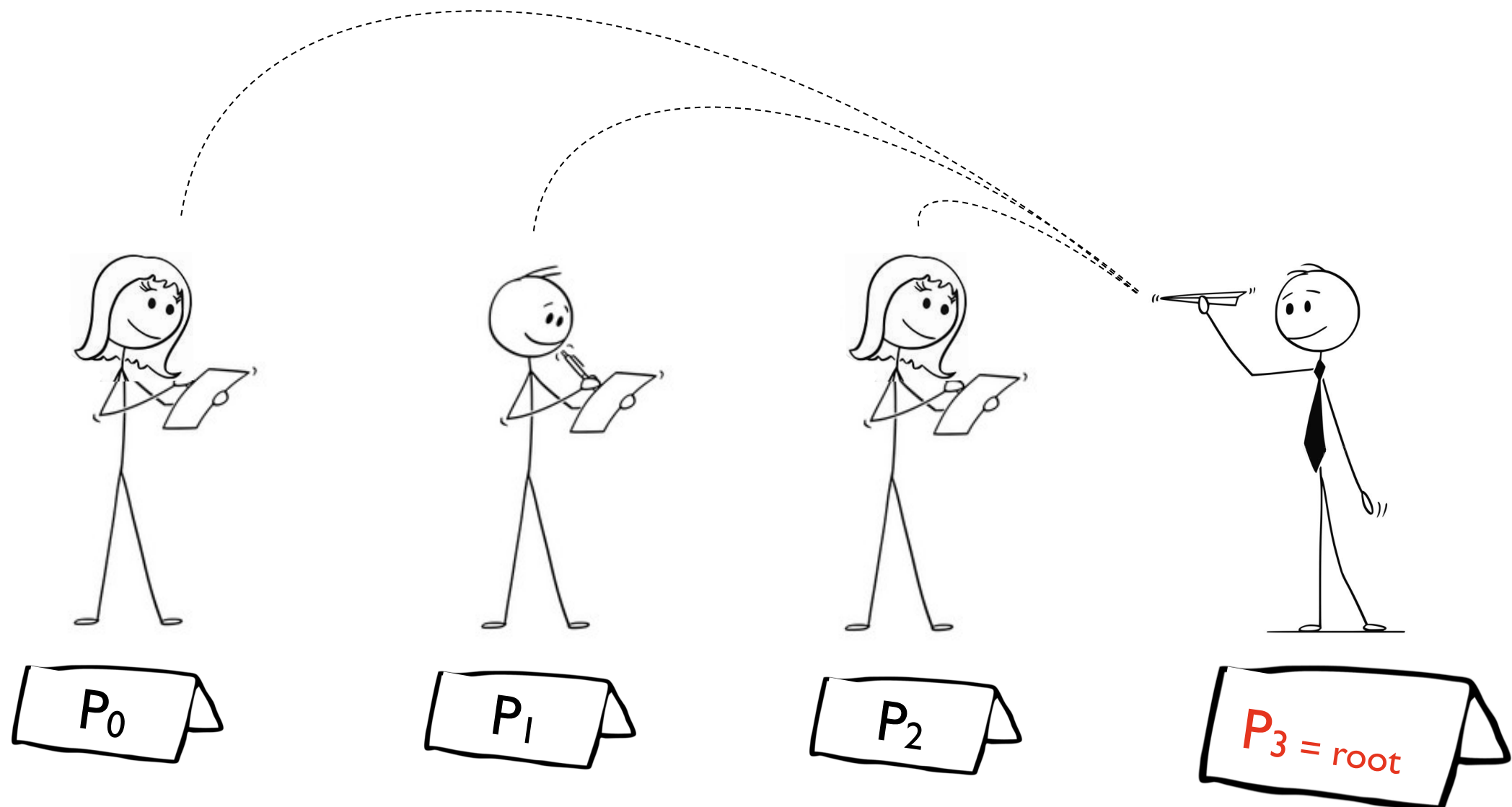
```
int MPI_Barrier( MPI_Comm comm ) ;
```

```
MPI_Init(&argc, &argv);  
  
/* Work 1 */  
MPI_Barrier(MPI_COMM_WORLD);  
  
/* Work 2 */  
MPI_Finalize();
```



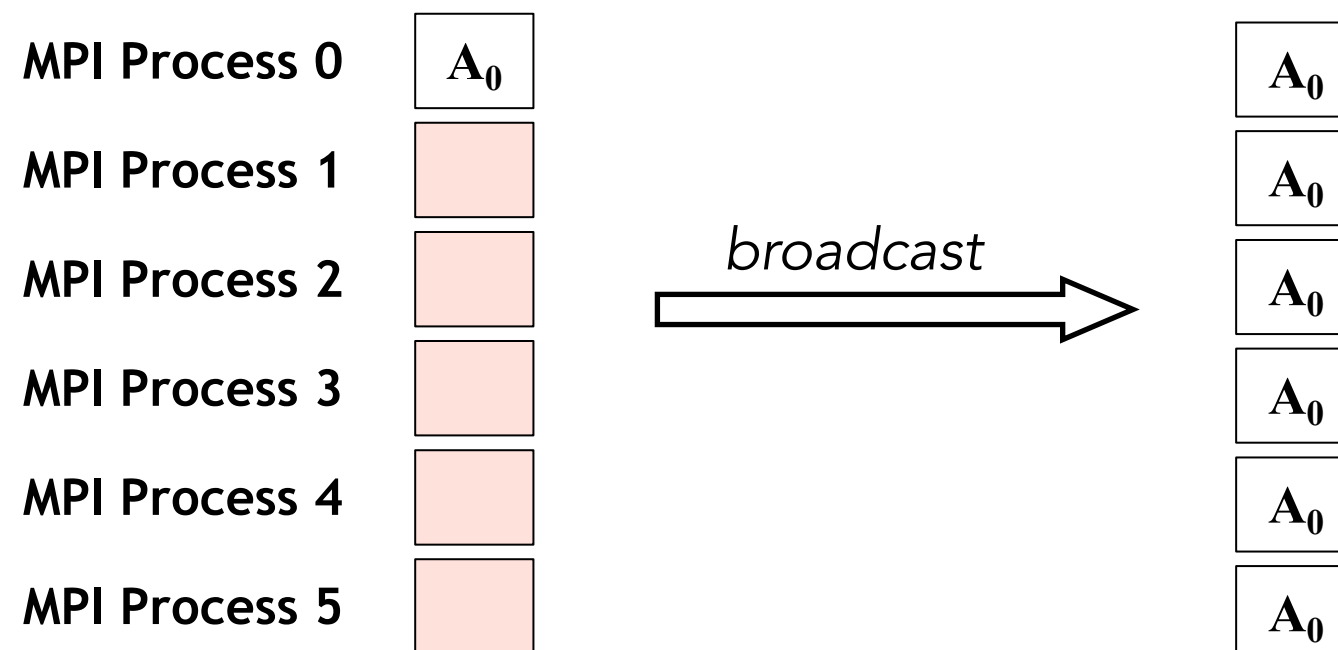
Broadcast

- Envoie les données d'un processus à tous les autres processus du communicateur
- Le **processus émetteur** est appelé **root**



Broadcast

- Collective **One-to-all**



Broadcast

```
int MPI_Bcast (  
    void *buf(inout),  
  
    int count(in),  
  
    MPI_Datatype datatype(in),  
  
    int root(in),  
  
    MPI_Comm comm(in),  
);
```

- **rank == root** -> adresse de la zone mémoire à envoyer
 - **rank != root** -> adresse où stocker les données diffusées
- Le segment mémoire de sortie doit être alloué par l'utilisateur

- Le rang **root** doit être valide dans **comm**
- Tous les processus appelant la fonction doivent avoir le même **root**

Broadcast

```
int me, root;
float pi = 0.0 ;

root = 0; /* Process 0 is the root */
...
MPI_Comm_rank(MPI_COMM_WORLD, &me);
...
if (me == root)
    pi = 3.14; /* Only root has the right initial value */

/* All processes have to call MPI_Bcast */
MPI_Bcast(&pi, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

printf("P%d: pi = %f\n", me, pi);
```

```
$ mpirun -np 4 ./a.out
```

```
P0: pi = 3.14
```

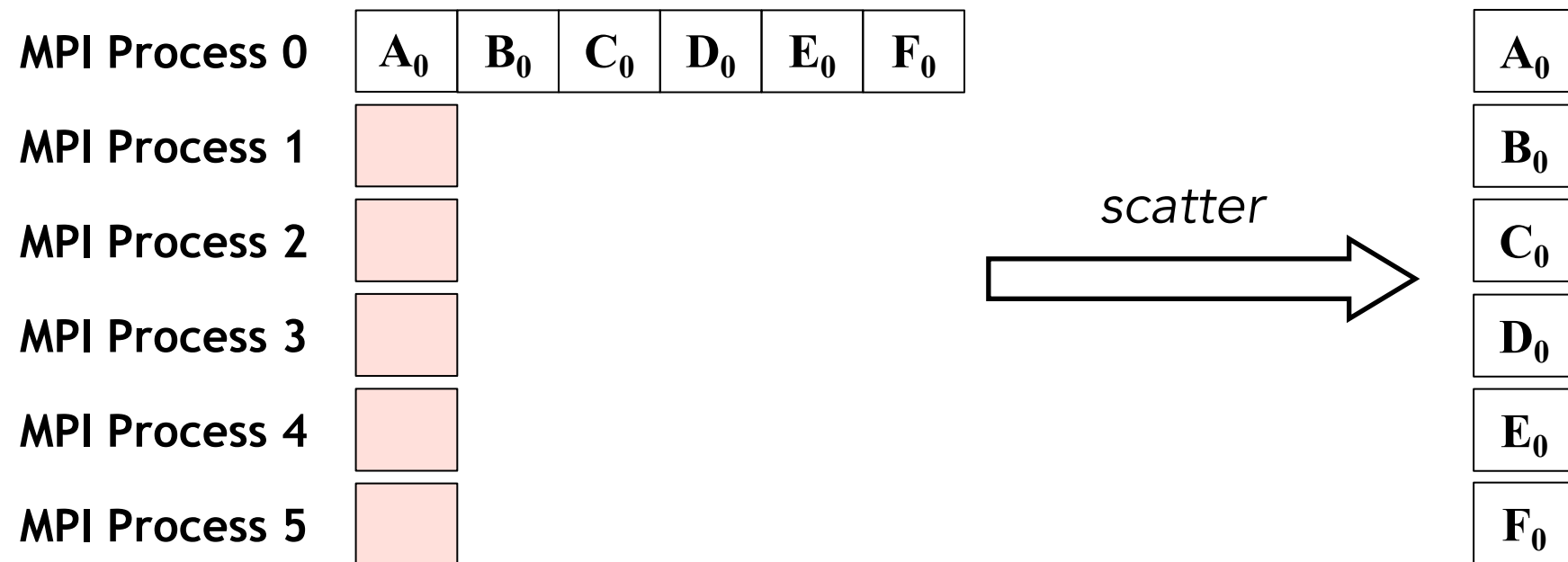
```
P3: pi = 3.14
```

```
P1: pi = 3.14
```

```
P2: pi = 3.14
```

Scatter

- Un processus **root** envoie différentes données à tous les autres processus du communicateur
- Données envoyées: même taille et même type
- Collective **One-to-all**



Scatter

```
int MPI_Scatter (
```

```
    void *sendbuf(in),
```

```
    int sendcount(in),
```

```
    MPI_Datatype sendtype(in),
```

```
    void *recvbuf(out),
```

```
    int recvcount(in),
```

```
    MPI_Datatype recvtype(in),
```

```
    int root(in),
```

```
    MPI_Comm comm(in)
```

```
);
```

- Segment mémoire à envoyer à chaque processus dans le communicateur **comm**
- Le message est de taille **sendcount** et de type **sendtype**
- **sendbuf** n'est valide que pour le processus de rang **root**

Scatter

```
int MPI_Scatter (  
    void *sendbuf(in),  
    int sendcount(in),  
    MPI_Datatype sendtype(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype recvtype(in),  
    int root(in),  
    MPI_Comm comm(in)  
);
```

- Adresse où stocker les données
- La taille est en nombre d'éléments de type rcvtyp
- Le segment mémoire doit être alloué avant l'appel

Scatter

```
int MPI_Scatter (  
    void *sendbuf(in),  
    int sendcount(in),  
    MPI_Datatype sendtype(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype recvtype(in),  
    int root(in),  
    MPI_Comm comm(in)  
);
```

- Le rang **root** doit être valide dans **comm**
- Tous les processus appelant la fonction doivent avoir le même **root**

Scatter

```

int me, v, root, P;
int *sdbuf;
root = 0; /* Process 0 is the root */

MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &P);

if (me == root) {

    sdbuf = (int*)malloc(P*sizeof(int));
    fd = fopen("my_data", "r");

    for( p = 0 ; p < P ; p++ )
        sdbuf[p] = read_file_value(fd, p);

    fclose( fd );

} else { sdbuf = NULL; }

MPI_Scatter(sdbuf, 1, MPI_INT, &v, 1, MPI_INT, root, MPI_COMM_WORLD);

printf("P%d: received value = %d\n", me, v);

```

```

$ cat my_data
18
25
6
3

```

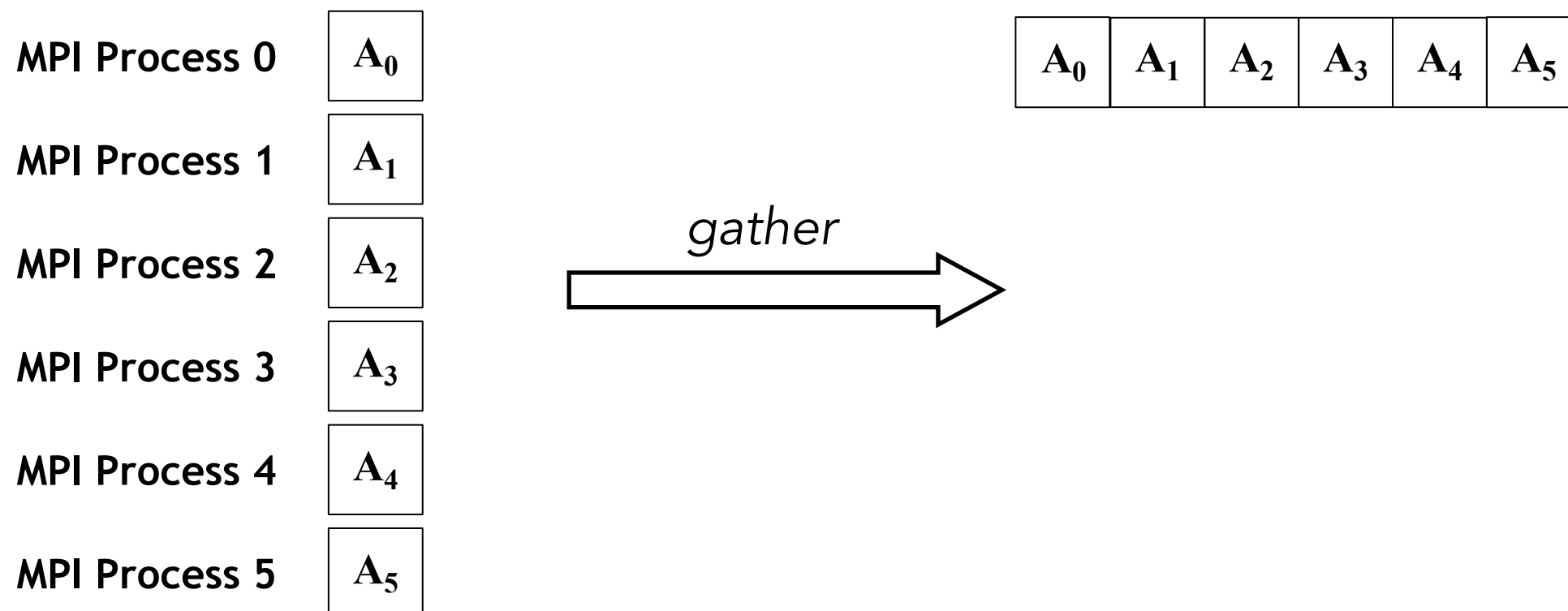
```

$ mpirun -np 4 ./a.out
P0: received value = 18
P3: received value = 3
P1: received value = 25
P2: received valud = 6

```

Gather

- Un processus **root** récupère des données de tous les autres processus (inverse de scatter)
- Données envoyées: même taille et même type
- Collective **All-to-one**



Gather

```
int MPI_Gather (
```

```
    void *sendbuf(in),
```

```
    int sendcount(in),
```

```
    MPI_Datatype sendtype(in),
```

```
    void *recvbuf(out),
```

```
    int recvcount(in),
```

```
    MPI_Datatype recvtype(in),
```

```
    int root(in),
```

```
    MPI_Comm comm(in)
```

```
);
```

- Adresse du segment mémoire contenant les données à envoyer au processus de rang root
- Le message est de taille sendcount

Gather

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

- Adresse où stocker les données à recevoir des processus
- Adresse valide que pour le processus root
- recvcount: taille des données à recevoir par le processus

Gather

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

- Le rang **root** doit être valide dans **comm**
- Tous les processus appelant la fonction doivent avoir le même **root**

Gather

```
int me, v, root, P; int *rcvbuf;
root = 0; /* Process 0 is the root */

MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &P);
...
if (me == root)
    rcvbuf = (int*)malloc(P*sizeof(int));
else
    rcvbuf = NULL;

v = ... ;
MPI_Gather(&v, 1, MPI_INT, rcvbuf, 1, MPI_INT, root, MPI_COMM_WORLD);

if (me == root) {

    fd = fopen("results", "w");

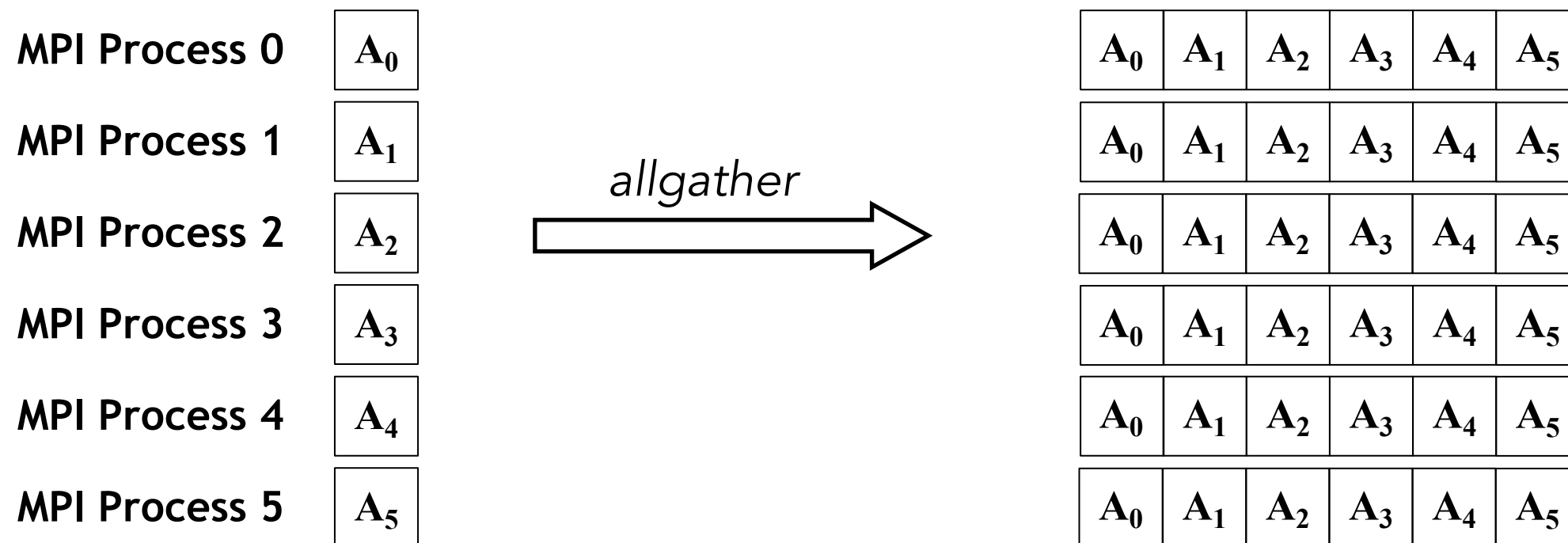
    for( p = 0 ; p < P ; p++ )
        write_file_value(fd, p, rcvbuf[p]);

    fclose( fd );
}
```

```
$ mpirun -np 4 a.out
$ cat results
30
-100
23
19
```


Allgather

- Equivalent à une collective Gather sauf que tous les processus reçoivent le résultat
- Allgather = Gather + Broadcast
- Collective **All-to-all**



Allgather

```
int MPI_Allgather (
```

```
    void *sendbuf(in),
```

```
    int sendcount(in),
```

```
    MPI_Datatype sendtype(in),
```

```
    void *recvbuf(out),
```

```
    int recvcount(in),
```

```
    MPI_Datatype recvtype(in),
```

```
    MPI_Comm comm(in)
```

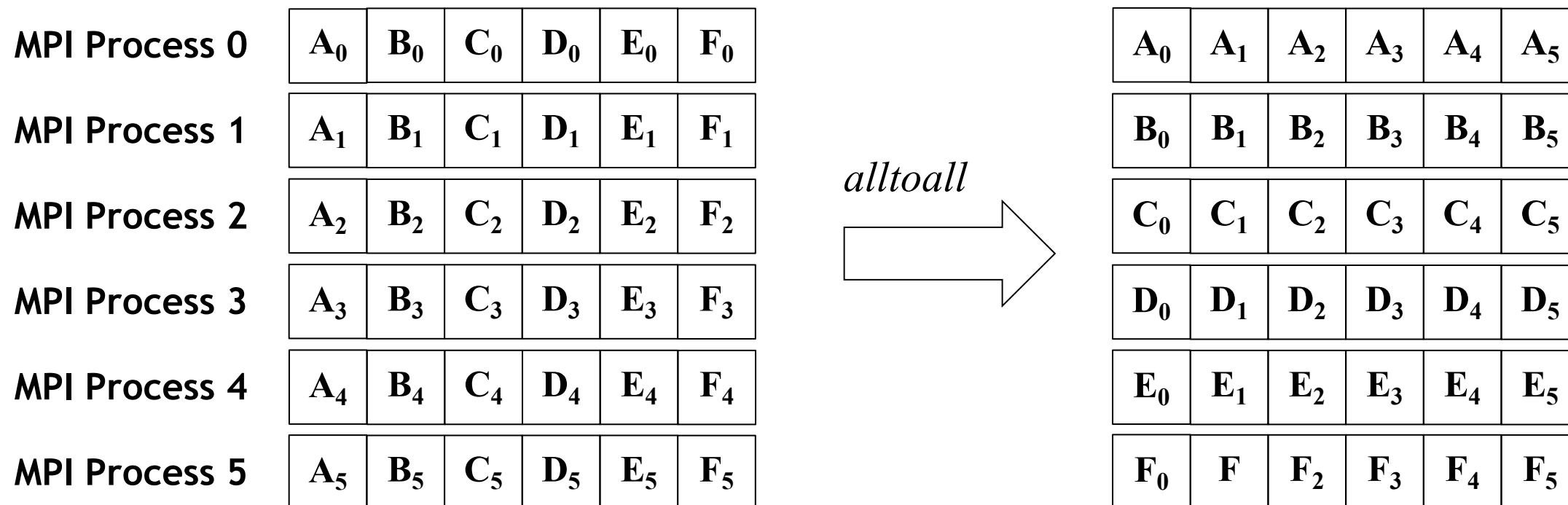
```
);
```

Arguments correspondant
aux données à envoyer

Arguments correspondant
à la réception des données

Alltoall

- Chaque processus envoie et reçoit des données
- Chaque processus envoie son ième paquet de données au rang i
- C'est une sorte de transposée de matrice
- Collective **All-to-all**



Alltoall

```
int MPI_Alltoall (
```

```
    void *sendbuf(in),
```

```
    int sendcount(in),
```

```
    MPI_Datatype sendtype(in),
```

```
    void *recvbuf(out),
```

```
    int recvcount(in),
```

```
    MPI_Datatype recvtype(in),
```

```
    MPI_Comm comm(in)
```

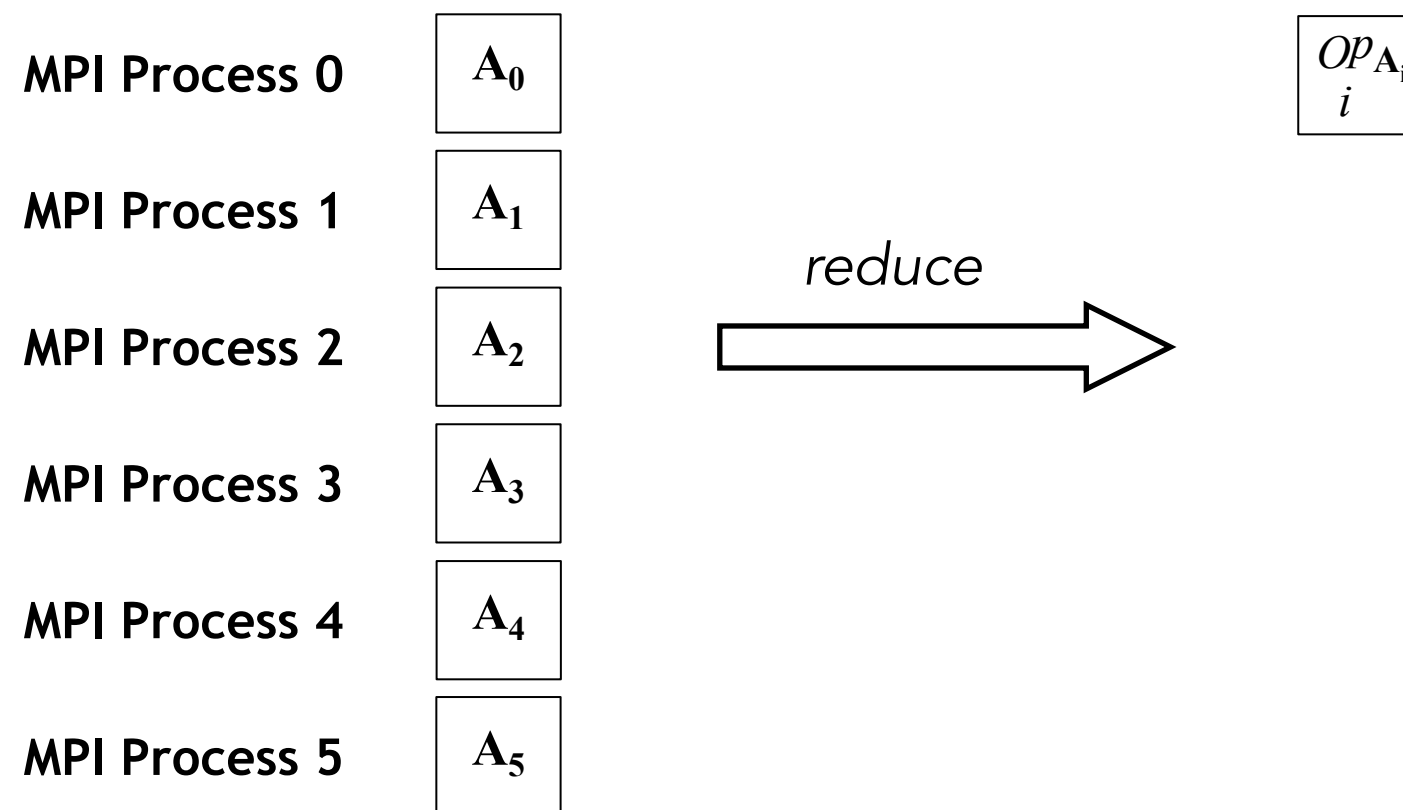
```
);
```

Arguments correspondant
aux données à envoyer

Arguments correspondant à
la réception des données

Reduce

- Le processus de rang **root** collecte les données des autres processus et applique une opération
- MPI autorise plusieurs réductions en même temps
- Collective **All-to-one**



Reduce

```
int MPI_Reduce (
```

```
void *sendbuf(in),
```

- Données à envoyer au processus root
- Tableau de count éléments de types datatype

```
void *recvbuf(out),
```

- Adresse du résultat
- Valide uniquement pour le processus de rang root
- Tableau de count éléments de types datatype

```
int count(in),
```

```
MPI_Datatype datatype(in),
```

```
MPI_Op op(in),
```

```
int root(in),
```

```
MPI_Comm comm(in),
```

```
);
```

Reduce

```
int MPI_Reduce (
```

```
    void *sendbuf(in),
```

```
    void *recvbuf(out),
```

```
    int count(in),
```


```
    MPI_Datatype datatype(in),
```

```
    MPI_Op op(in),
```

```
    int root(in),
```

```
    MPI_Comm comm(in),
```

```
);
```

- 
- Opération à faire sur chaque élément de chaque processus
 - Au retour de la fonction, le processus root aura:
$$\text{recvbuf}[i] = \text{op}_{0 \leq p < P} \text{sendbuf}_p[i]$$
avec $0 \leq i < \text{count}$
et P la taille de comm

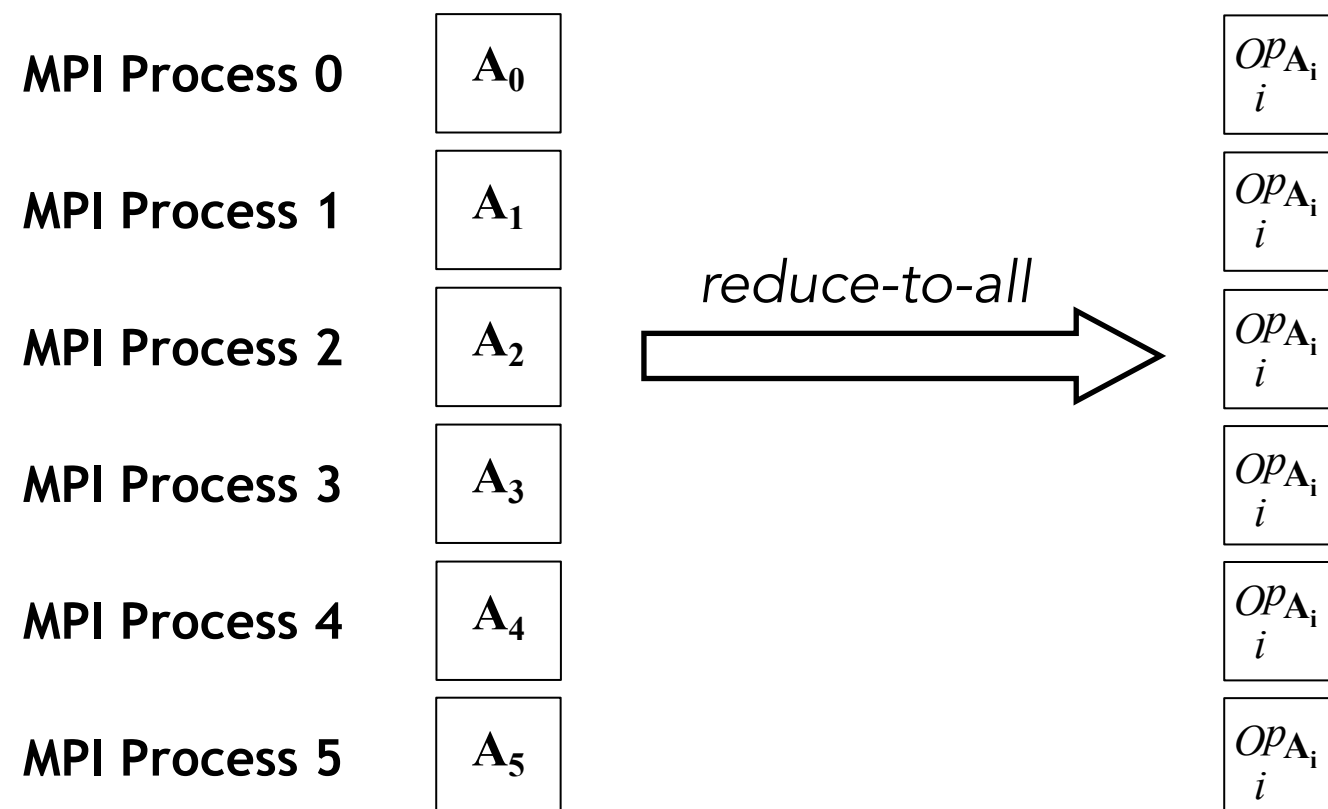
Reduce

- Les opérations de réduction sont de type `MPI_Op`
- Il est possible de créer des opérateurs: `MPI_Op_create()`

MPI Operation	Meaning	Corresponding Type
MPI_MAX	Maximum	Integers and real
MPI_MIN	Minimum	Integers and real
MPI_SUM	Sum	Integer and real
MPI_PROD	Product	Integer and real
MPI_LAND	Logical AND	Integer
MPI_BAND	Binary AND	Integer and MPI_BYTE
MPI_LOR	Logical OR	Integer
MPI_BOR	Binary OR	integer MPI_BYTE
MPI_LXOR	Logical XOR	Integer
MPI_BXOR	Binary XOR	Integer and MPI_BYTE
MPI_MAXLOC	Maximum w/ index	Structures w/ 2 integers
MPI_MINLOC	Minimum w/ index	Structures w/ 2 integers

Allreduce

- Equivalent à une réduction sauf que tous les processus ont le résultat final
- Pas besoin de spécifier un processus root
- Allreduce \approx Reduce + Broadcast
- Collective **All-to-all**



Allreduce

```
int MPI_Allreduce (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int count(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    MPI_Comm comm(in),  
);
```

Allreduce

```
int P, N = 100;
int me, i, sum_glob = 0, sum_loc = 0;

MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &P);
...
/* Works if N%P == 0 */
for( i = 1 + me*N/P ; i <= (me+1)*N/P ; i++ )
    sum_loc += i;

MPI_Allreduce(&sum_loc, &sum_glob, 1, MPI_INT,
             MPI_SUM, MPI_COMM_WORLD);

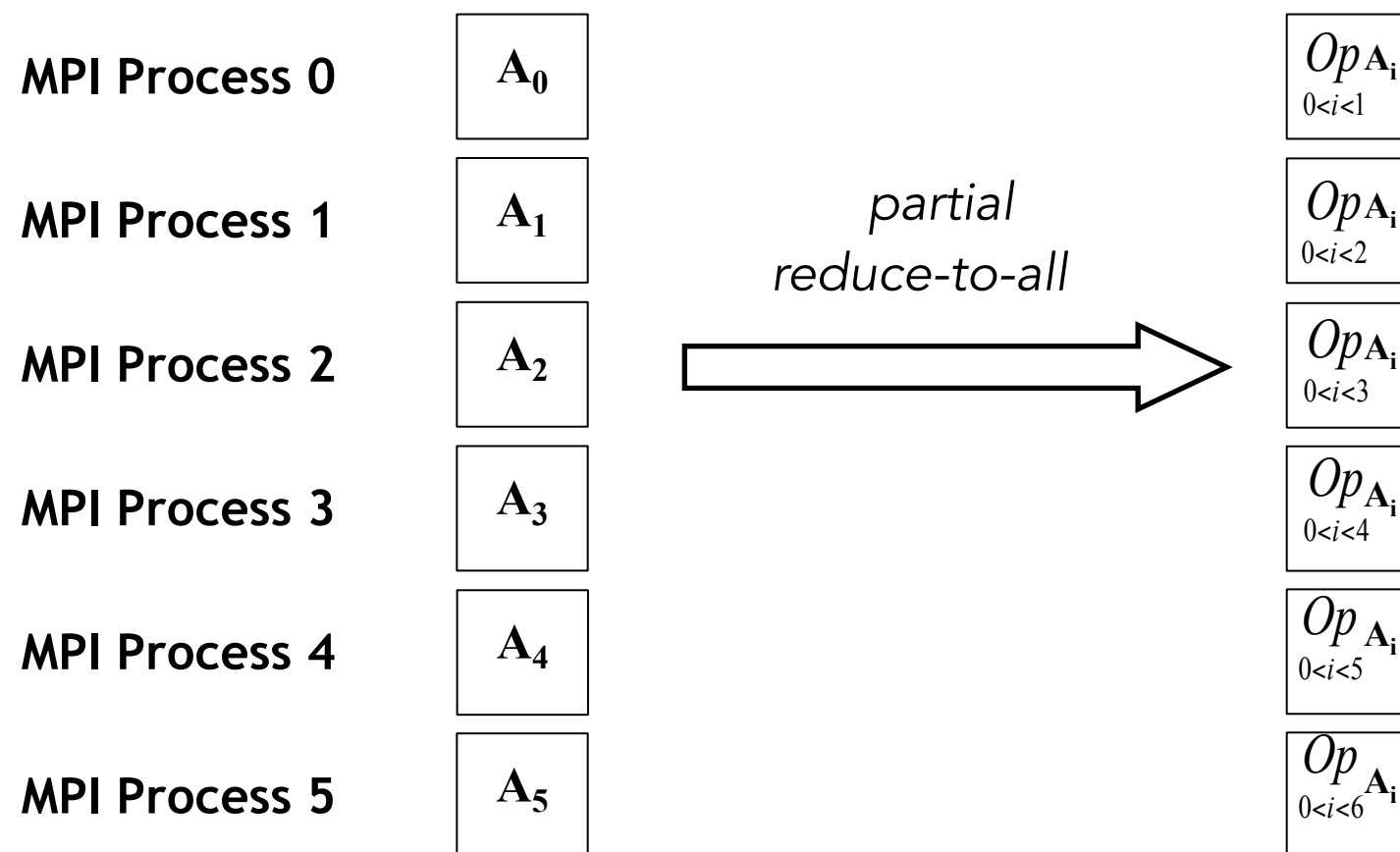
/* After reduction, all processes have, in sum_glob, the sum of
N first integers */

printf("1+...+%d = %d\n", N, sum_glob);
```

```
$ mpirun -np 4 ./a.out
1+...+100 = 5050
1+...+100 = 5050
1+...+100 = 5050
1+...+100 = 5050
```

Réduction partielle avec Scan

- Tous les processus collectent des données des autres processus avec un rang plus petit et appliquent une opération spécifique
- MPI autorise plusieurs réductions à la fois



Réduction partielle avec Scan

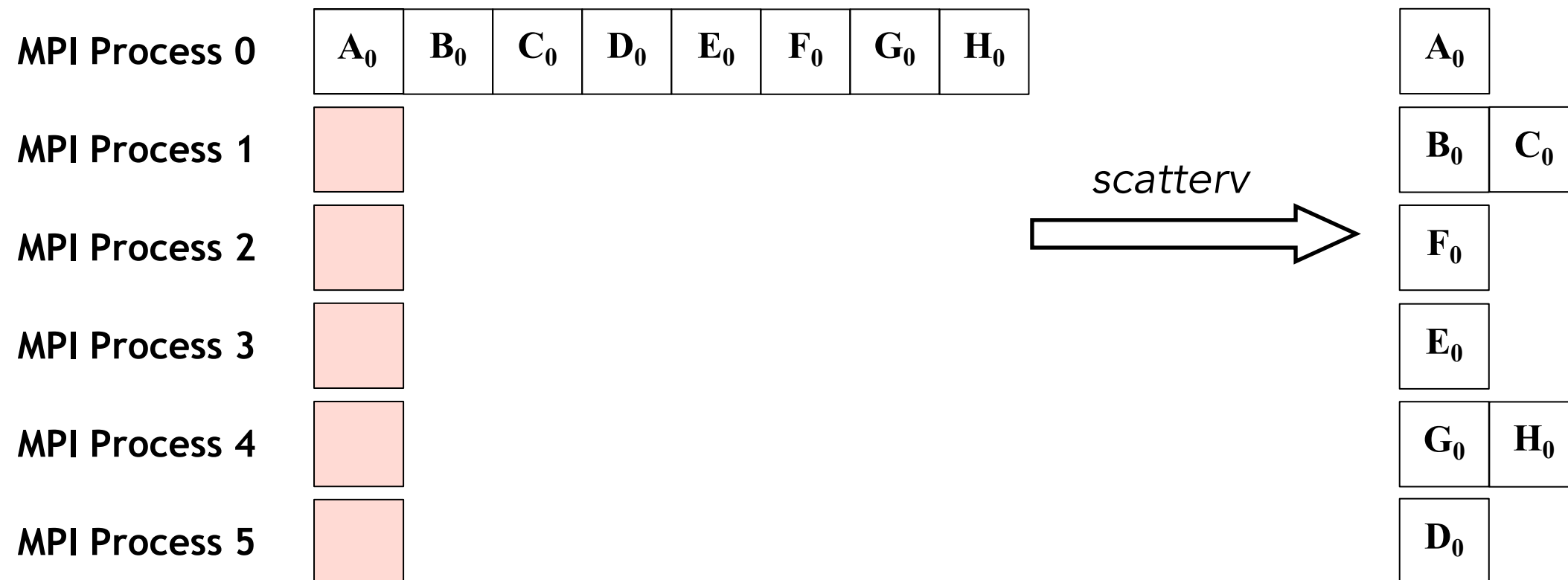
```
int MPI_Scan(  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int count(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    MPI_Comm comm(in),  
);
```

Une taille des données par rang

- Certaines collectives proposent une gestion des données de différentes tailles par processus
 - Par ex., Broadcast, Gather
- Le nom correspondant a le suffixe v
 - v = variation ou vecteur
- Exemples
 - `MPI_Gather` → `MPI_Gatherv`
 - `MPI_Allgather` → `MPI_Allgatherv`
 - `MPI_Scatter` → `MPI_Scatterv`
 - `MPI_Alltoall` → `MPI_Alltoallv`

MPI_Scatterv

- Les données A_i sont de même type
- A l'inverse de MPI_Scatter, A_0, A_1, \dots peuvent être de tailles différentes
 - On a besoin d'un nouvel argument pour spécifier la taille de chaque donnée
- A l'inverse de MPI_Scatter, les données A_i peuvent ne pas être contiguës en mémoire
 - On a besoin d'un nouvel argument pour spécifier l'adresse de chaque donnée



MPI_Scatterv

```
int MPI_Scatterv (  
    void *sendbuf(in),  
    int *sendcounts(in),  
    int *displs(in),  
    MPI_Datatype sendtype(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype recvtype(in),  
    int root(in), MPI_Comm comm(in)  
);
```

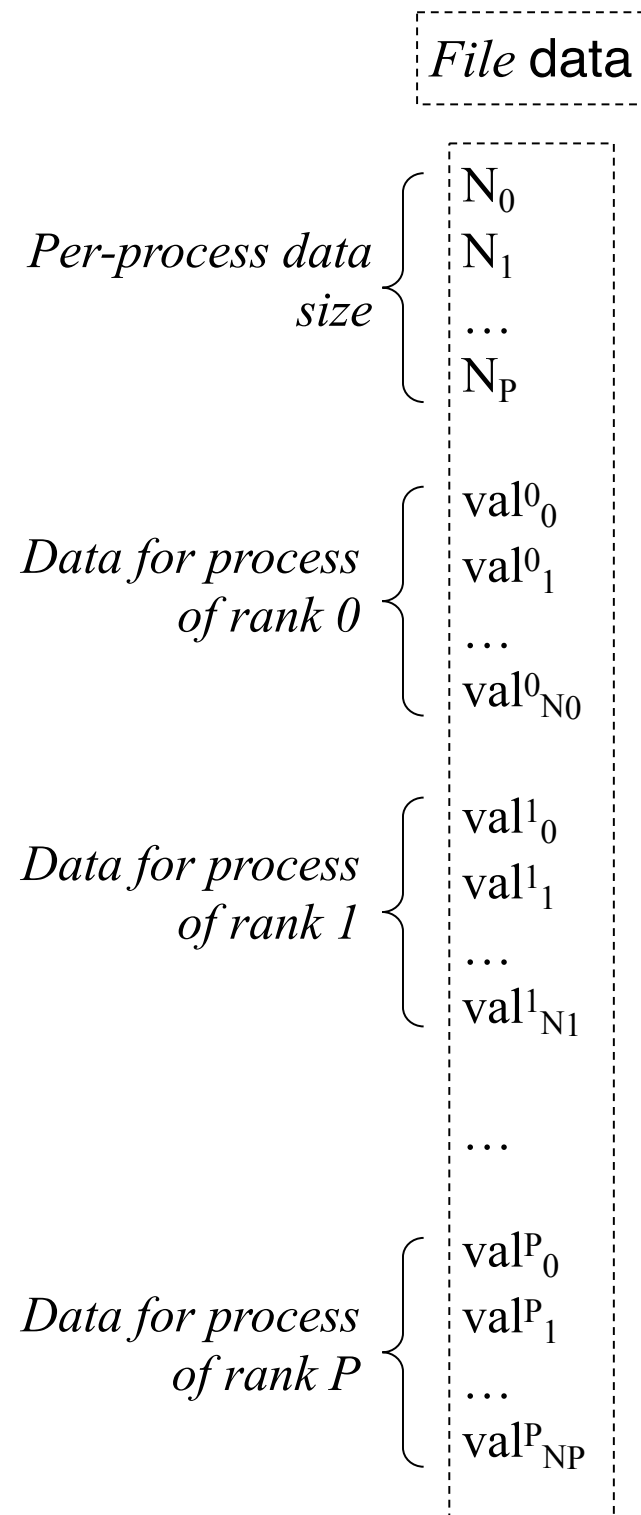
- **sendbuf** = données à distribuer à chaque processus dans le communicateur **comm**
- **sendcounts**[p] éléments doivent être envoyés au processus p ($0 \leq p < P$)
- Ces éléments sont stockés à l'adresse **sendbuf**+**displs**[p]
- Tous les éléments sont de type **sendtype**
- Ces arguments ne sont valides que pour le processus **root**

MPI_Scatterv

```
int MPI_Scatterv (  
    void *sendbuf(in),  
    int *sendcounts(in),  
    int *displs(in),  
    MPI_Datatype sendtype(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype recvtype(in),  
    int root(in), MPI_Comm comm(in)  
);
```

- `recvbuf` = données à recevoir
- `recvcount` éléments de types `recvtype`

MPI_Scatterv



- Un processus lit un fichier de nom data
- Ce fichier contient les données pour chaque processus
- Le format est décrit sur la gauche
 - Taille des données par processus
 - Données (ex., float values) stockées **in a per-process manner**
- Après l'étape de parsing, tous les processus ont leur propre partie des données

MPI_Scatterv

```
int sz; int *szbuf, *displs;
double *sdbuf, *rcvbuf;

if (me == root) {
    szbuf = (int*)malloc(P*sizeof(int));
    displs = (int*)malloc((P+1)*sizeof(int));
    fd = fopen("data", "r");
    displs[0] = 0;
    for( p = 0 ; p < P ; p++ ) {
        szbuf[p] = read_int_value(fd, p);
        displs[p+1] = displs[p] + szbuf[p];
    }
    sdbuf = (double*)malloc(displs[P]*sizeof(double));
    for( p = 0 ; p < P ; p++ )
        for( i = 0 ; i < szbuf[p] ; i++ )
            sdbuf[displs[p]+i] = read_float_value(fd, p);
    fclose( fd );
}
```

/* Read data sizes for each process (szbuf array)
Compute corresponding offset (displs array) */

/* Read per-block data (sdbuf array).
Blocks will be distributed among ranks
*/

MPI_Scatterv

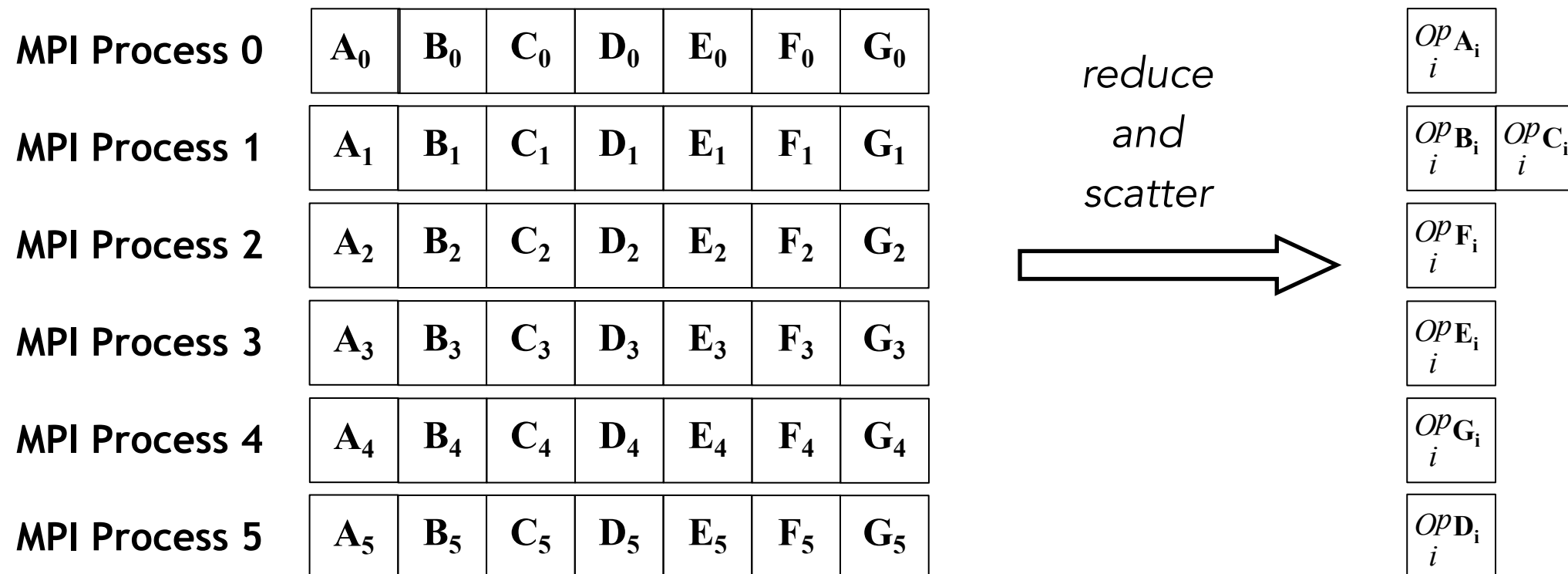
```
else {
    szbuf = displs = NULL;
    sdbuf = NULL;
}

/* All processes must call MPI_Scatter
   Corresponding data size is stored into sz
*/
MPI_Scatter(szbuf, 1, MPI_INT, &sz, 1, MPI_INT,
            root, MPI_COMM_WORLD);

rcvbuf = (double*)malloc(sz*sizeof(double));
/* All processes must call MPI_Scatterv
   Corresponding data are stored in rcvbuf
*/
MPI_Scatterv(sdbuf, szbuf, displs, MPI_DOUBLE, rcvbuf,
             sz, MPI_DOUBLE, root, MPI_COMM_WORLD);
```

Reduce-Scatter

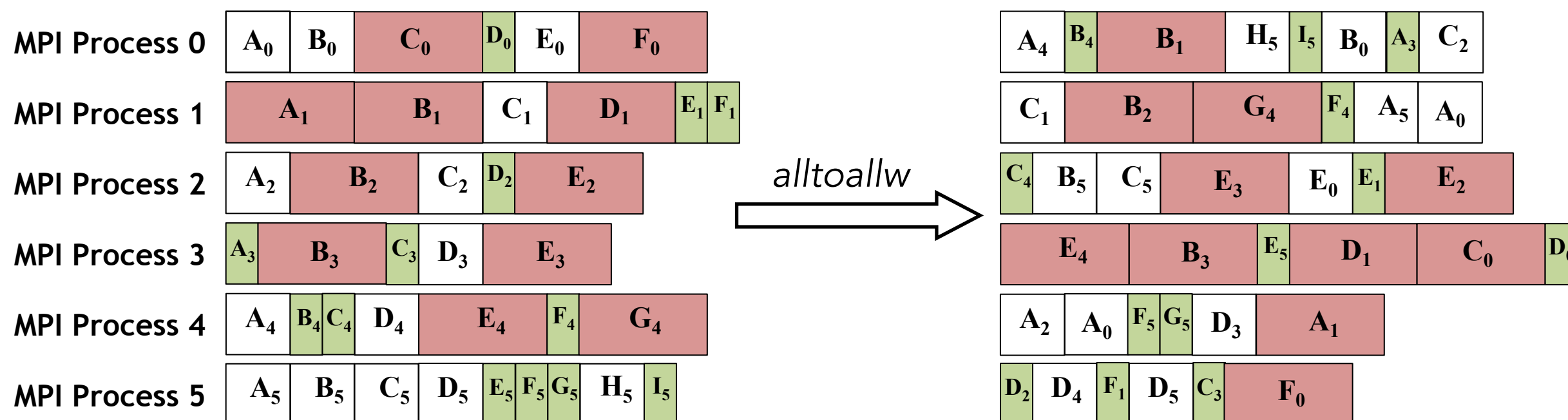
- Combine une réduction avec un scatterv
- Réduction sur un tableau
- Chaque processus a une partie de ce tableau
- Comme `MPI_Scatterv`, A_0, A_1, \dots peuvent ne pas avoir la même taille sur la partie du scatter



Les trois « erreurs »

Alltoallw

- Comme `MPI_Scatterv`, A_0, A_1, \dots peuvent être de tailles différentes
 - ➔ On a besoin d'un nouvel argument pour spécifier la taille de chaque donnée
- Comme `MPI_Scatterv`, les données A_i peuvent ne pas être contiguës en mémoire
 - ➔ On a besoin d'un nouvel argument pour spécifier l'adresse de chaque donnée
- De plus, les données A_i peuvent être de types différents
 - ➔ L'argument donnant le type est maintenant un tableau



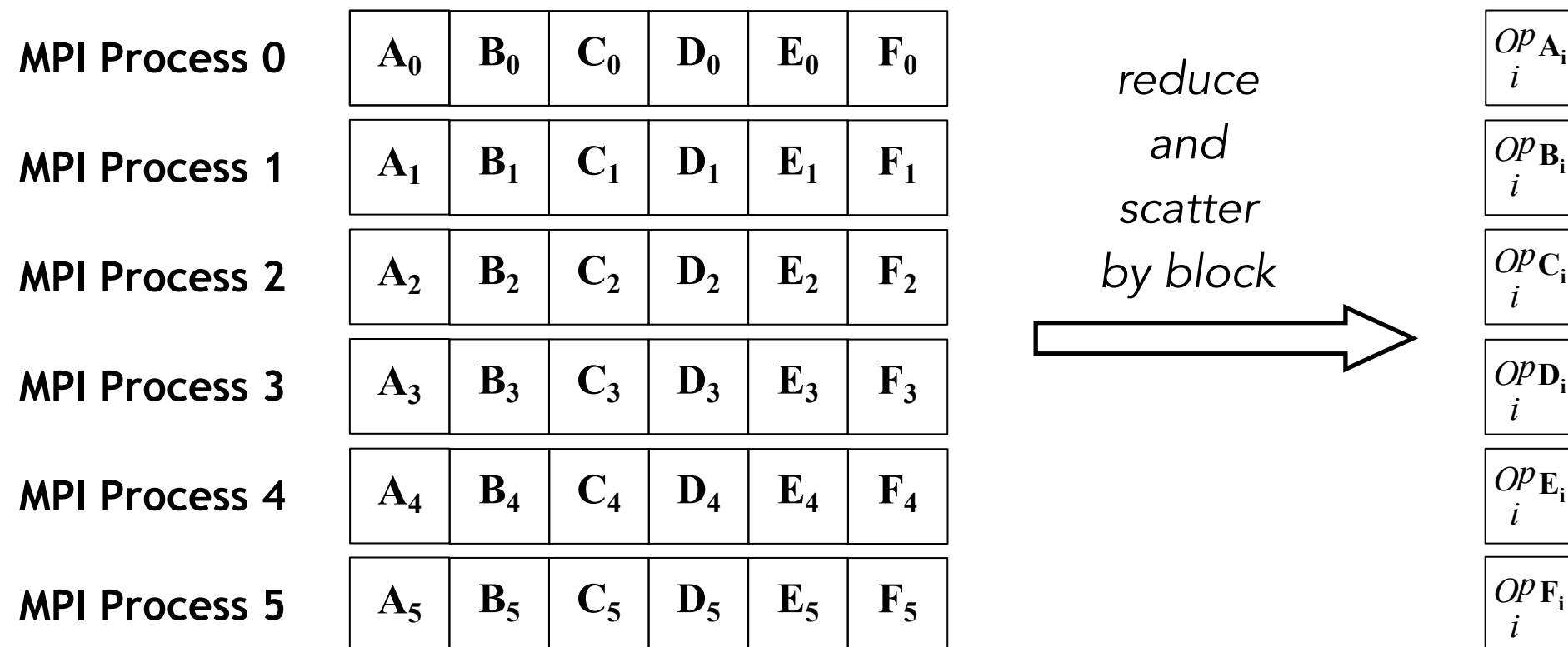
Les trois « erreurs » Alltoallw

```
int MPI_Alltoallw (  
    void *sendbuf(in),  
    int *sendcounts(in),  
    int *displs(in),  
    MPI_Datatype *sendtypes(in),  
    void *recvbuf(out),  
    int *recvcounts(in),  
    MPI_Datatype *recvtypes(in),  
    int root(in), MPI_Comm comm(in)  
);
```

Les trois « erreurs »

Reduce-Scatter block

- Combine une réduction et un scatter
- Réduction sur un tableau
- Chaque processus a une partie du tableau de même taille
- Les données sont contigus en mémoire



Les trois « erreurs »

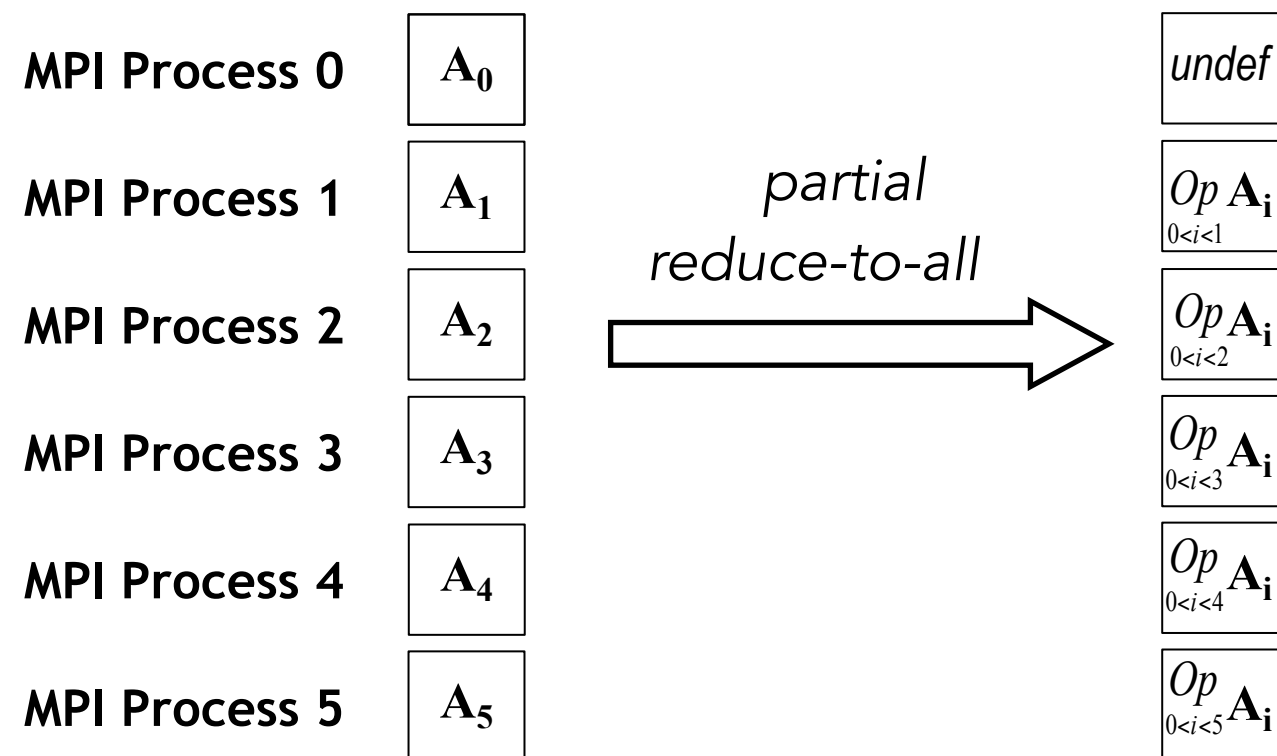
Reduce-Scatter block

```
int MPI_Reduce_Scatter_block (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int recvcnt(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    MPI_Comm comm(in),  
);
```

Les trois « erreurs »

Réduction partielle avec Exscan

- Tous les rangs collectent des données des processus avec un rang plus petit et appliquent une opération spécifique, sans sa propre contribution



Les trois « erreurs »

Réduction partielle avec Exscan

```
int MPI_Exscan (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int count(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    MPI_Comm comm(in),  
);
```

Des « erreurs » ??

- `Reduce_scatter_block` est la version simplifiée de `reduce_scatter`
 - Aurait du être la première fonction à apparaître
 - Par souci de cohérence au niveau des noms, `reduce_scatter_block` devrait être `reduce_scatter` et `reduce_scatter` devrait être `reduce_scatterv`
- `MPI_Scan` est juste `MPI_Exscan` + variables locales
 - Pas besoin de créer deux fonctions
 - `MPI_Exscan` est suffisante et aurait du être la première à apparaître
- `MPI_Alltoallw`
 - Permet d'échanger n'importe quoi
 - Plusieurs datatypes -> implementation et algorithme complexes
 - Pourquoi est-ce la seule collective qui autorise plusieurs datatypes?



Utilisation des collectives

- Si un processus termine une collective, cela ne veut pas dire que tous les processus ont terminé la collective.
- Si le processus courant n'a pas besoin du résultat mais seulement d'envoyer des données, la collective est équivalente à `MPI_Send`
 - Après l'appel à la collective, l'utilisateur peut réutiliser le buffer
 - Ca ne veut pas dire que les données ont été envoyées !
- Si tous les processus fournissent des données d'entrée et ont besoin d'un résultat, la collective est synchronisante
 - Cependant, la barrière est la seule collective officielle qui synchronise les processus

Utilisation des collectives

- Chaque collective est basée sur un communicateur qui définit l'ensemble des processus qui participent à la collective



- Que se passe t-il si deux rangs appellent différentes collectives?
- Que se passe t-il si deux rangs dans des communicateurs différents appellent la même collective?

Règles

- Chaque collective est basée sur un communicateur qui définit l'ensemble des processus qui participent à la collective



- Que se passe t-il si deux rangs appellent différentes collectives?
 - Que se passe t-il si deux rangs dans des communicateurs différents appellent la même collective?
-
- Tous les processus d'un même groupe doivent appeler une collective

Règles

- Chaque collective est basée sur un communicateur qui définit l'ensemble des processus qui participent à la collective



- Que se passe t-il si deux rangs appellent différentes collectives?
 - Que se passe t-il si deux rangs dans des communicateurs différents appellent la même collective?
-
- Tous les processus d'un même groupe doivent appeler une collective
 - Au sein d'un communicateur, tous les processus doivent avoir la même séquence de collectives.

Règles

- Chaque collective est basée sur un communicateur qui définit l'ensemble des processus qui participent à la collective



- Que se passe t-il si deux rangs appellent différentes collectives?
 - Que se passe t-il si deux rangs dans des communicateurs différents appellent la même collective?
-
- Tous les processus d'un même groupe doivent appeler une collective
 - Au sein d'un communicateur, tous les processus doivent avoir la même séquence de collectives.
 - Il n'y a pas de restriction entre deux rangs dans des communicateurs différents

Règles

- Chaque collective est basée sur un communicateur qui définit l'ensemble des processus qui participent à la collective



- Que se passe t-il si deux rangs appellent différentes collectives?
 - Que se passe t-il si deux rangs dans des communicateurs différents appellent la même collective?
-
- Tous les processus d'un même groupe doivent appeler une collective
 - Au sein d'un communicateur, tous les processus doivent avoir la même séquence de collectives.
 - Il n'y a pas de restriction entre deux rangs dans des communicateurs différents
-
- Attention aux arguments des collectives (arguments compatibles)
 - Attention aux communicateurs
 - Attention au flot de contrôle dans le programme

Utilisation des collectives

```
void f ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

```
void g ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    else  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

```
void h ( int r ) {  
    if( r == 0 ) {  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
        MPI_Barrier(MPI_COMM_WORLD);  
    } else {  
        MPI_Barrier(MPI_COMM_WORLD);  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
    }  
    return; }  
}
```

Utilisation des collectives

```
void f ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

Potentiellement incorrect
(dépend de la valeur de *r*)

```
void g ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    else  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

```
void h ( int r ) {  
    if( r == 0 ) {  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
        MPI_Barrier(MPI_COMM_WORLD);  
    } else {  
        MPI_Barrier(MPI_COMM_WORLD);  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
    }  
    return; }  

```

Utilisation des collectives

```
void f ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

Potentiellement incorrect
(dépend de la valeur de `r`)

```
void g ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    else  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

Correct

```
void h ( int r ) {  
    if( r == 0 ) {  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
        MPI_Barrier(MPI_COMM_WORLD);  
    } else {  
        MPI_Barrier(MPI_COMM_WORLD);  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
    }  
    return; }  
}
```

Utilisation des collectives

```
void f ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

Potentiellement incorrect
(dépend de la valeur de `r`)

```
void g ( int r ) {  
    if( r == 0 )  
        MPI_Barrier(MPI_COMM_WORLD);  
    else  
        MPI_Barrier(MPI_COMM_WORLD);  
    return;  
}
```

Correct

```
void h ( int r ) {  
    if( r == 0 ) {  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
        MPI_Barrier(MPI_COMM_WORLD);  
    } else {  
        MPI_Barrier(MPI_COMM_WORLD);  
        MPI_Reduce(MPI_COMM_WORLD, ...);  
    }  
    return; }  
}
```

Potentiellement incorrect
(dépend de la valeur de `r`)

Message Passing Interface

1. Rappels

2. Communications collectives

1. Communications bloquantes

2. Communications non bloquantes

Collectives non bloquantes

- Depuis MPI 3, **toutes les collectives ont une équivalence non bloquante**
 - ➔ Permet un recouvrement des communications avec du calcul

Plus de communications impliquées dans une collective que dans une P2P donc plus d'opportunités d'obtenir des performances
- Même sémantique que les P2P non bloquantes
 - ➔ Les appels non-bloquants initient la communication
 - ➔ L'opération n'est **pas finie au retour de l'appel**
 - ➔ Besoin d'une **opération de complétion** pour s'assurer que le buffer d'entrée peut être réutilisé (Wait* et Test*)
 - ➔ Le nom d'une communication non bloquante commence par un "I"

Règles d'utilisation des collectives non bloquantes



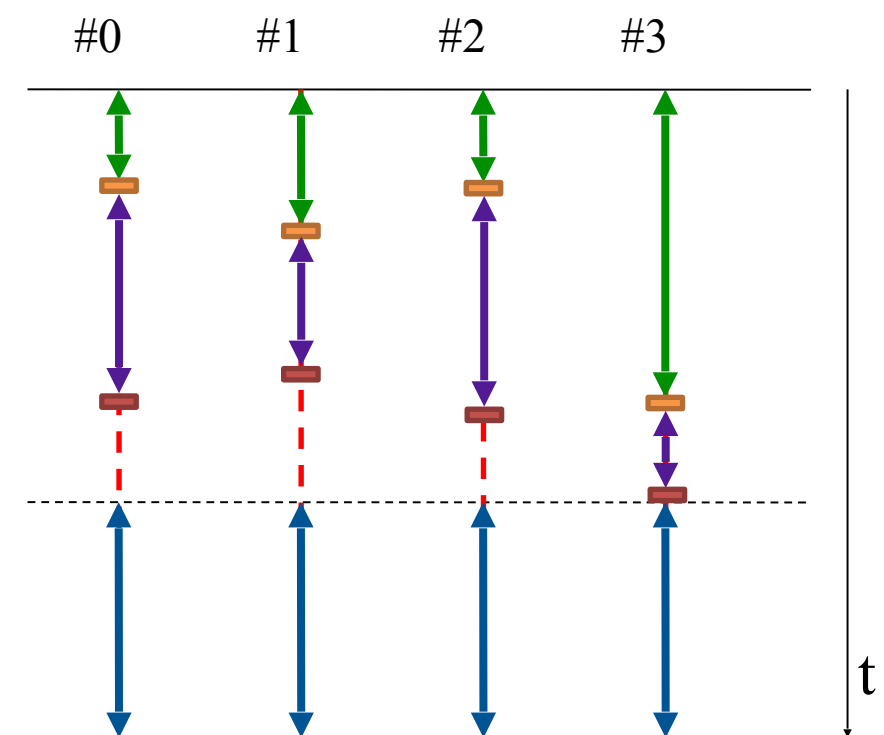
- **Tous les processus d'un communicateur** doivent appeler la collective non bloquante sur ce communicateur
- Au sein d'un communicateur, les processus doivent avoir la **même séquence de collectives** (bloquantes et non-bloquantes)
- Il n'y a aucune restriction entre des rangs de communicateurs différents
- Le **matching** collective **bloquante** \leftrightarrow collective **non bloquante** est **interdit**

Barrière non bloquante

Synchronise tous les processus appartenant à un communicateur

```
int MPI_Ibarrier( MPI_Comm comm, MPI_Request * req ) ;
```

```
MPI_Init(&argc, &argv);  
MPI_Request req;  
  
/* Work 1 */  
MPI_Ibarrier(MPI_COMM_WORLD, &req);  
  
/* Work 2 */  
MPI_Wait(req, status);  
  
/* Work 3 */  
MPI_Finalize();
```



Broadcast non-bloquant

```
int MPI_Ibcast (  
  
    void *buf(inout),  
  
    int count(in),  
  
    MPI_Datatype datatype(in),  
  
    int root(in),  
  
    MPI_Comm comm(in),  
  
    MPI_Request *req(inout)  
);
```

La requête **req** est mise à jour avec les informations données par l'appel de completion

Collectives non-bloquantes

```
int MPI_Iscatter (void *sendbuf(in), int sendcount(in), MPI_Datatype sndatyp(in),  
void *recvbuf(out), int recvcount(in), MPI_Datatype rcvtatyp(in), int root(in),  
MPI_Comm comm(in), MPI_Request *req(inout) );
```

```
int MPI_Igather (void *sendbuf(in), int sendcount(in), MPI_Datatype sndatyp(in),  
void *recvbuf(out), int recvcount(in), MPI_Datatype rcvtatyp(in), int root(in),  
MPI_Comm comm(in), MPI_Request *req(inout) );
```

...

A vous de jouer!