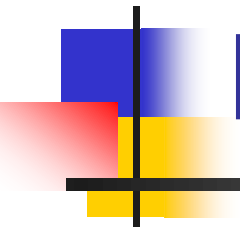
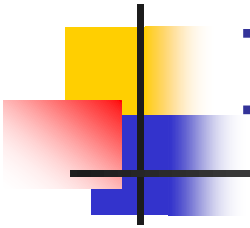


Programmation Parallèle et Distribuée

Cours 4 : Communications collectives

Patrick Carribault
David Dureau
Marc Pérache (marc.perache@cea.fr)

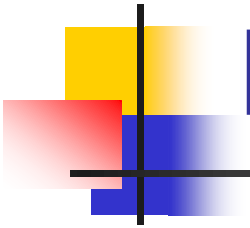


Introduction

- Communications collectives communications qui mettent en jeu tous les processus d'un communicateur
- Attention
  - Très coûteuses
  - Très pénalisantes pour les performances

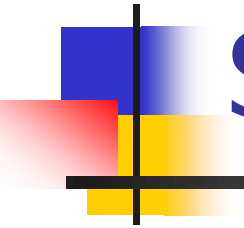
- Elles peuvent être écrites par des *send* et des *receive*, mais il est fortement conseillé d'utiliser les fonctions MPI correspondantes (car leurs implémentations sont certainement optimisées)

- Tous les processus doivent faire appel en même temps (au sens de l'algorithme) à la fonction MPI correspondante à l'opération collective.



Plan du cours 4

- Synchronisation
  - Barrière
- Echange de données
  - Diffusion
  - Distribution
  - Rassemblement
  - Réduction
- Gestion de données de taille différente



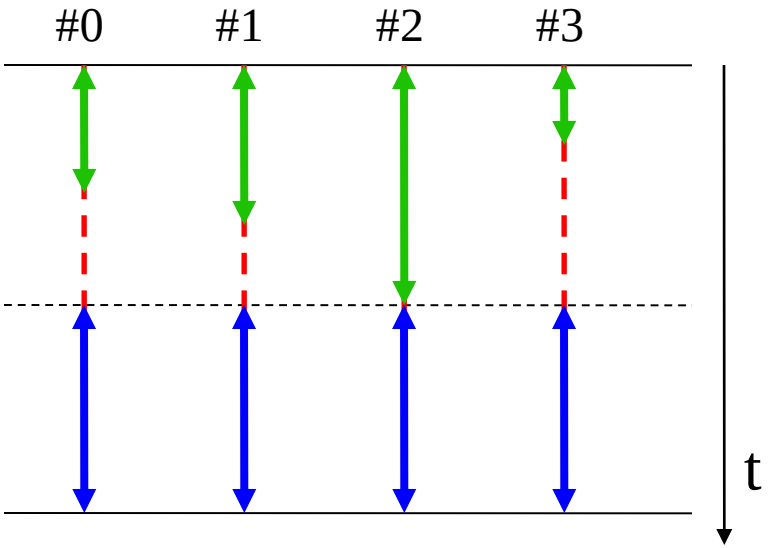
Synchronisation

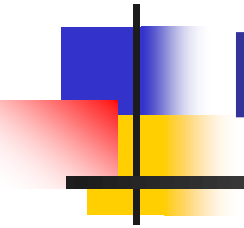
Barrière

- Synchronisation de tous les processus d'un communicateur

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

```
MPI_Init(&argc, &argv);

/* travail phase 1 */

MPI_Barrier(MPI_COMM_WORLD);

/* travail phase 2 */

MPI_Finalize();
```

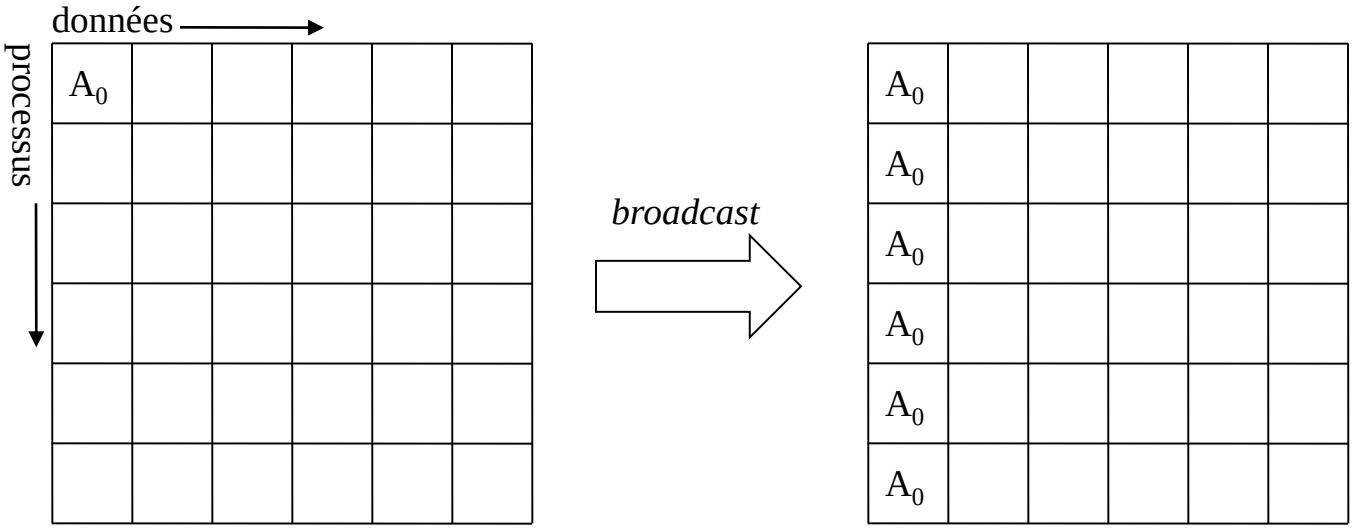




Echange de données

Diffusion (*broadcast*)

- Émet une information détenue par un processus vers tous les processus du communicateur
- Le processus qui détient initialement l'information est appelé racine *(root)*
- On parle d'émission un-vers-tous *(one-to-all)* : émission de la racine vers tous les autres processus



Diffusion (*broadcast*)

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

- Si le rang est égal à `root` : adresse de la zone mémoire des données à envoyer ;
- Si le rang est différent de `root` : adresse de la zone mémoire des données à recevoir.

La zone mémoire doit être allouée AVANT l'appel à `MPI_Bcast`.

Taille du message à émettre/à recevoir.

Exprimée en nombre d'éléments de type `datatype`.

Diffusion (*broadcast*)

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

Rang de la racine.

Ce rang est valide dans le communicateur `comm`.

Tous les processus du communicateur doivent utiliser la même racine.

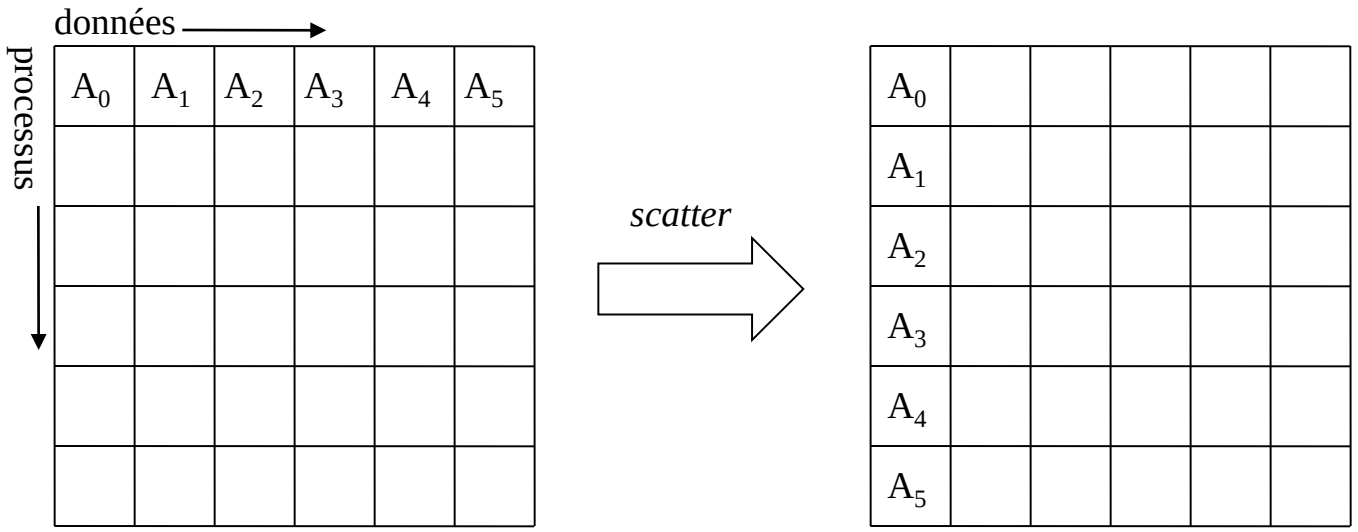
Diffusion (*broadcast*)

```
int moi, root;
float pi;

root = 0; /* le processus 0 est la racine */
…
MPI_Comm_rank(MPI_COMM_WORLD, &moi);
…
if (moi == root)
   pi = 3.14; /* seule la racine détient initialement l'information */

/* tous les processus doivent faire appel à MPI_Bcast
 */
MPI_Bcast(&pi, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

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

```
% mpirun -np 4 a.out
P0 : pi = 3.14
P3 : pi = 3.14
P1 : pi = 3.14
P2 : pi = 3.14
%
```

Distribution (*scatter*)

- La racine émet des données différentes (mais de même taille et de même type) à chaque processus
  - émission un-vers-tous (*one-to-all*) .



Distribution (*scatter*)

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

adresse de la zone mémoire des données à envoyer à chaque processus du communicateur `comm`.

`sendbuf` est valide pour la racine seulement.

Distribution (*scatter*)

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

Taille du message à envoyer à chaque processus.

Le tableau `sendbuf` est dimensionné à `sendcount` $\times P$ `sndtatyp` éléments où P est la taille du communicateur `comm`.

Les éléments à envoyer au processus de rang p ($0 \leq p < P$) débutent à l'adresse `sendbuf` + $p\times$ `sendcount`.

Distribution (*scatter*)

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

adresse de la zone mémoire des données à recevoir.

La zone mémoire doit être allouée AVANT l'appel à `MPI_Scatter`.

Distribution (*scatter*)

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

Taille du message à recevoir : égale à `sendcount`.

Exprimée en nombre d'éléments de type `rcvtatyp`.

Distribution (*scatter*)

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

Rang de la racine.

Ce rang est valide dans le communicateur comm.

Tous les processus du communicateur doivent utiliser la même racine.

Distribution (*scatter*)

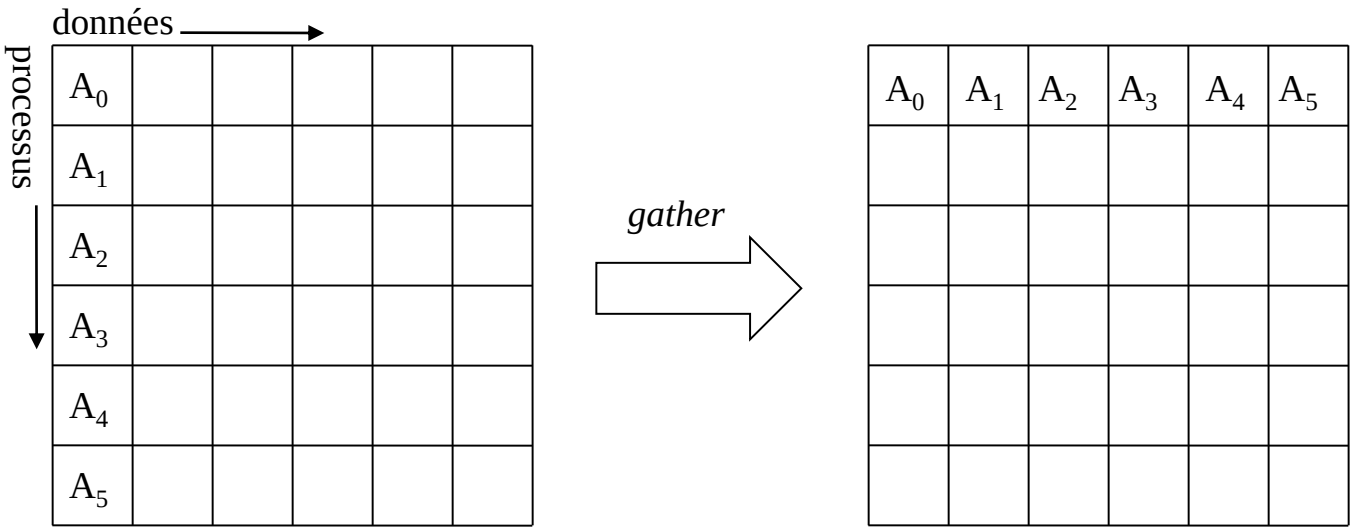
Le processus 0 lit le contenu d'un fichier et le distribue aux autres processus

```
int moi, val_recue, root;
int *sdbuf;

root = 0; /* le processus 0 est la racine */

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

if (moi == root) {
   sdbuf = (int*)malloc(P*sizeof(int)); /* tableau de P entiers */
   fd = fopen("donnees", "r");
   for( p = 0 ; p < P ; p++ )
      sdbuf[p] = lire_val_fichier(fd, p); /* récupération de la p-ième valeur du fichier donnees */
   fclose( fd );
} else { sdbuf = NULL; }

/* tous les processus doivent faire appel à MPI_Scatter */
MPI_Scatter(sdbuf, 1, MPI_INT,
            &val_recue, 1, MPI_INT, root, MPI_COMM_WORLD);

printf("P%d : val recue = %d\n", moi, val_recue);
```

```
% cat donnees
18
25
6
3
% mpirun -np 4 a.out
P0 : val recue = 18
P3 : val recue = 3
P1 : val recue = 25
P2 : val recue = 6
%
```

Rassemblement (*gather*)

- La racine collecte toutes les informations détenues (de même taille et de même type) par tous les processus du communicateur
  - c'est l'opération inverse de *scatter*
  - c'est une émission tous-vers-un (*all-to-one*)



Rassemblement (*gather*)

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

adresse de la zone mémoire des données à envoyer à la racine.

Rassemblement (*gather*)

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

Taille du message à envoyer à la racine.

Exprimée en nombre d'éléments de type `sndtatyp`.

Rassemblement (*gather*)

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

adresse de la zone mémoire des données à recevoir de chaque processus du communicateur `comm`.

Valide que pour la racine.

Rassemblement (*gather*)

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

Taille des données à recevoir par processus.

Doit être égal à `sendcount`.

Le tableau `recvbuf` est dimensionné à `recvcount`$\times P$ éléments de type `rcvtatyp` où P est la taille du communicateur.

Les données envoyées par le processus p ($0 \leq p < P$) seront stockées à l'adresse `recvbuf` + $p\times$ `recvcount`.

Rassemblement (*gather*)

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

Rang de la racine.

Ce rang est valide dans le communicateur `comm`.

Tous les processus du communicateur doivent utiliser la même racine.

Rassemblement (*gather*)

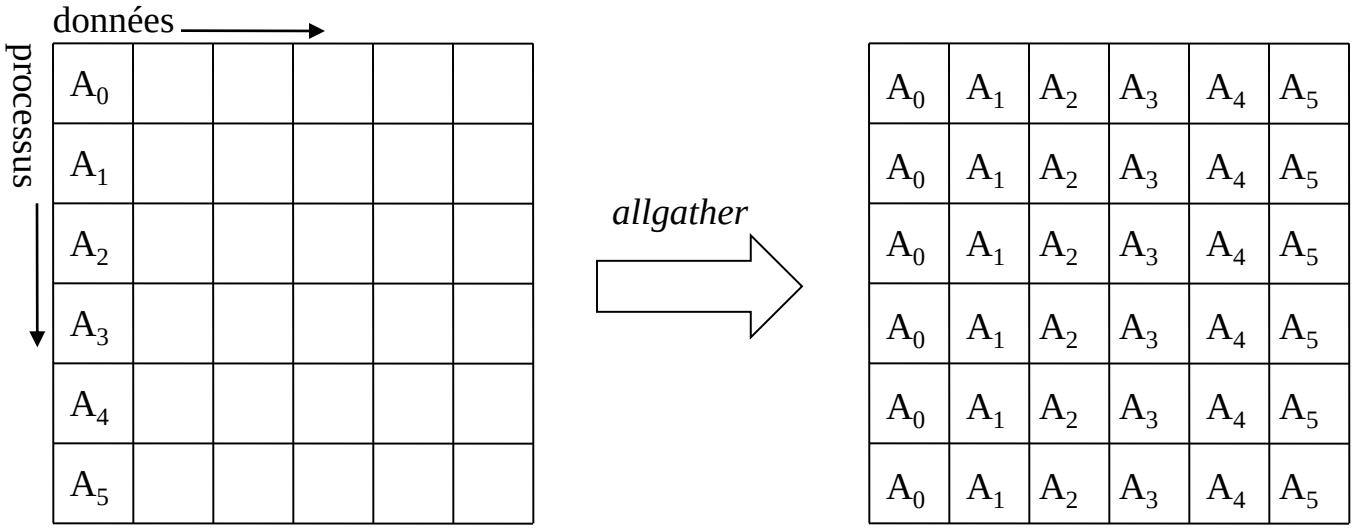
Le processus 0 récupère des données de tous les processus et les écrit dans un fichier

```
int moi, val_calcul, root;
int *rcvbuf;
root = 0; /* le processus 0 est la racine */

MPI_Comm_rank(MPI_COMM_WORLD, &moi);
MPI_Comm_size(MPI_COMM_WORLD, &P);
…
if (moi == root)
   rcvbuf = (int*)malloc(P*sizeof(int)); /* tableau de P entiers */
else rcvbuf = NULL;

val_calcul = … ; /* chaque processus effectue un calcul */
/* tous les processus doivent faire appel à MPI_Gather */
MPI_Gather(&val_calcul, 1, MPI_INT,
           rcvbuf, 1, MPI_INT, root, MPI_COMM_WORLD);

if (moi == root) { /* seule la racine écrit les résultats */
   fd = fopen("resultats", "w");
   for( p = 0 ; p < P ; p++ )
      ecrire_val_fichier(fd, p, rcvbuf[p]); /* écriture d'une valeur en position p dans resultats */
   fclose( fd );
}
```

```
% mpirun -np 4 a.out
% cat resultats
30
-100
23
19
```

Gather-to-all

- Equivalent à une opération *gather* mais tous les processus reçoivent le résultat, au lieu de seulement la racine



Gather-to-all

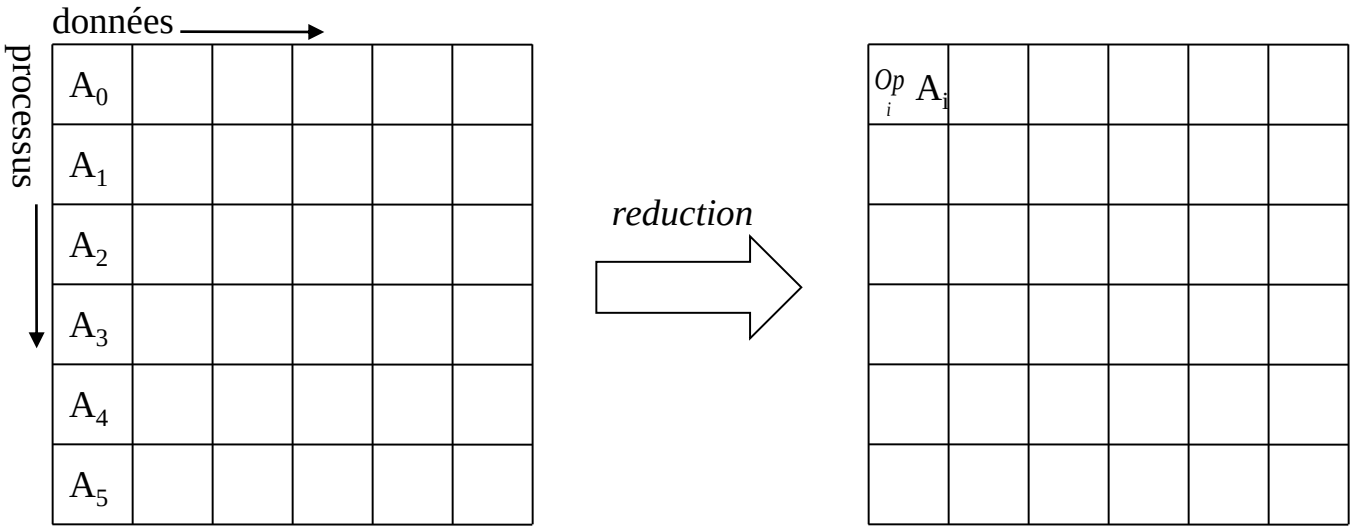
```
int MPI_Allgather (
void *sendbuf(in),
int sendcount(in),
MPI_Datatype sndtatyp(in),
void *recvbuf(out),
int recvcount(in),
MPI_Datatype rcvtatyp(in),
MPI_Comm comm(in),
);
```

Ces 3 arguments caractérisent le message à envoyer à tous les processus

Ces 3 arguments caractérisent le message à recevoir de tous les processus

Réduction (*reduction*)

- La racine rassemble les informations détenues par tous les processus et leurs applique un opérateur (comme une addition, une multiplication ...)



Réduction (*reduction*)

```
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),
);
```

Adresse des données à envoyer à la racine.

C'est un tableau de `count` éléments de type `datatype`.

Réduction (*reduction*)

```
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),
);
```

Adresse des données à recevoir.

Argument utilisé que par la racine.

C'est un tableau de `count` éléments de type `datatype`.

Réduction (*reduction*)

```
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 effectuer sur les éléments de tous les processus du communicateur.

Après l'appel, la racine aura :
recvbuf[i] = $op_{0 \le p < P}$ $sendbuf_p$ [i]
où $0 \le i <$ count
et P est la taille du communicateur comm.

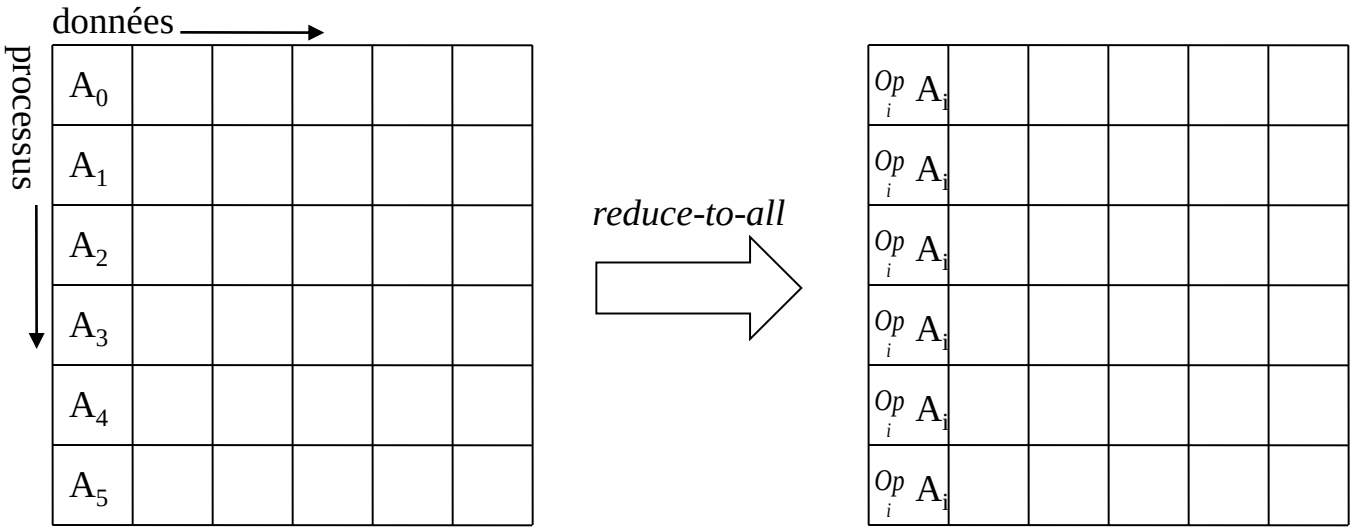
Réduction (*reduction*)

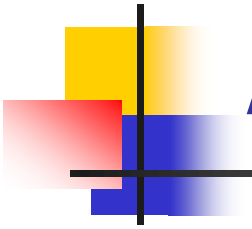
- Les opérations de réduction sont de type MPI_Op ;
- Créations d'autres opérateurs à l'aide de `MPI_Op_create()`

Opérations prédéfinies de type **`MPI_op`**	*signification*	*types correspondants*
`MPI_MAX`	Maximum	entiers et réels
`MPI_MIN`	Minimum	entiers et réels
`MPI_SUM`	Somme	entiers et réels
`MPI_PROD`	Produit	entiers et réels
`MPI_LAND`	Et "logique"	entiers
`MPI_BAND`	Et "bit à bit"	entiers et `MPI_BYTE`
`MPI_LOR`	Ou "logique"	entiers
`MPI_BOR`	Ou "bit à bit"	entiers et `MPI_BYTE`
`MPI_LXOR`	Ou exclusif "logique"	entiers
`MPI_BXOR`	Ou exclusif "bit à bit"	entiers et `MPI_BYTE`
`MPI_MAXLOC`	Maximum et position max	structures avec 2 entiers
`MPI_MINLOC`	Minimum et position min	structures avec 2 entiers

Reduce-to-all

- Equivalent à une réduction mais où tous les processus du communicateur récupère le résultat
  - Absence de racine





Reduce-to-all

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

Les données à réduire (locales au processus) se trouvent dans le buffer `sendbuf` (tableau de `count` éléments de type `datatype`).

Chaque processus reçoit les données réduites dans le buffer `recvbuf`.

Reduce-to-all

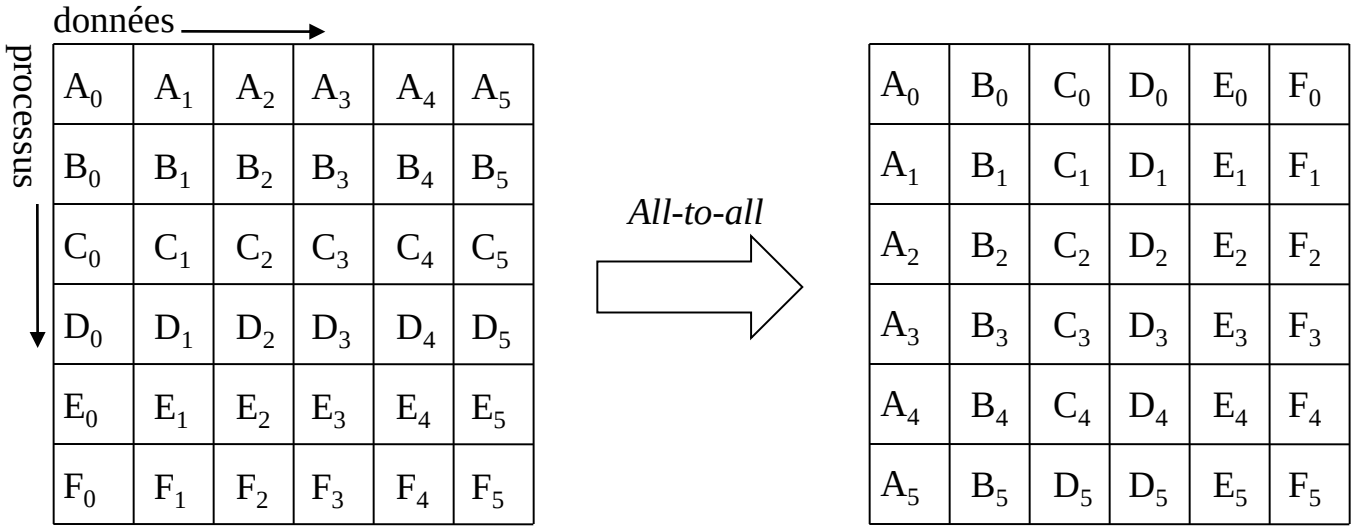
Somme des entiers de 1 à N

```
int P, N = 100;
int moi, i, som_glob, som_loc = 0;

…
MPI_Comm_rank(MPI_COMM_WORLD, &moi);
MPI_Comm_size(MPI_COMM_WORLD, &P);
…
/* hypothèse = N est divisible par P */
for( i = 1 + moi*N/P ; i <= (moi+1)*N/P ; i++ )
   som_loc += i;

MPI_Allreduce(&som_loc, &som_glob, 1, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);

/* Après la réduction, tous les processus ont dans som_glob la somme des entiers
de 0 à N-1 */

printf("1+…+%d = %d\n", N, som_glob);
```

```
% mpirun -np 4 a.out
1+…+100 = 5050
1+…+100 = 5050
1+…+100 = 5050
1+…+100 = 5050
%
```

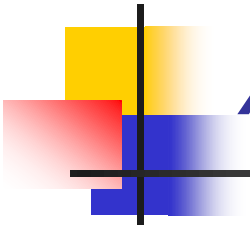
All-to-All

- Chaque processus du communicateur envoie des informations différentes à l'ensemble des processus du communicateur, et reçoit des informations différentes de tous les processus du communicateur
- C'est la communication collective la plus coûteuse :
  - P^2 échanges de données où P est la taille du communicateur
- Transposition de la matrice processus × données



A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	D_5	D_5	E_5	F_5



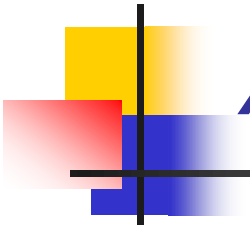
All-to-All

```
int MPI_Alltoall (
void *sendbuf(in),
int sendcount(in),
MPI_Datatype sndtatyp(in),
void *recvbuf(out),
int recvcount(in),
MPI_Datatype rcvtatyp(in),
MPI_Comm comm(in),
);
```

adresse de la zone mémoire des données à envoyer à chaque processus du communicateur comm.

Tableau de sendcount × *P* éléments de type sndtatyp, avec *P* taille du communicateur.

Les sendcount éléments commençant à l'adresse sendbuf $+p \times$ sendcount sont à envoyer au processus p $(0 \leq p < P)$.



All-to-All

```
int MPI_Alltoall (
void *sendbuf(in),
int sendcount(in),
MPI_Datatype sndtatyp(in),
void *recvbuf(out),
int recvcount(in),
MPI_Datatype rcvtatyp(in),
MPI_Comm comm(in),
);
```

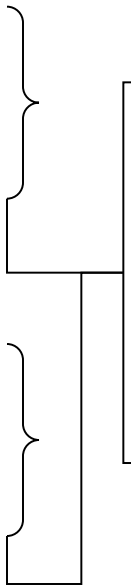
adresse de la zone mémoire des données à recevoir de chaque processus du communicateur comm.

Tableau de recvcount × *P* éléments de type sndtatyp, avec *P* taille du communicateur.

Les recvcount éléments commençant à l'adresse recvbuf +*p* × recvcount sont reçus du processus *p* ($0 \leq p < P$).

All-to-All

```
int MPI_Alltoall (
void *sendbuf(in),
int sendcount(in),
MPI_Datatype sndtatyp(in),
void *recvbuf(out),
int recvcount(in),
MPI_Datatype rcvtatyp(in),
MPI_Comm comm(in),
);
```



Les signatures doivent être égales :
sign(sendcount , sndtatyp) =
sign(recvcount , rcvtatyp)

En règle générale, on a :
sendcount = recvcount
sndtatyp = rcvtatyp

All-to-All

– Soit P le nombre de processus ;

– Soit A une matrice carrée de dimension $P \times P$ et dont les éléments sont des couples d'entiers;

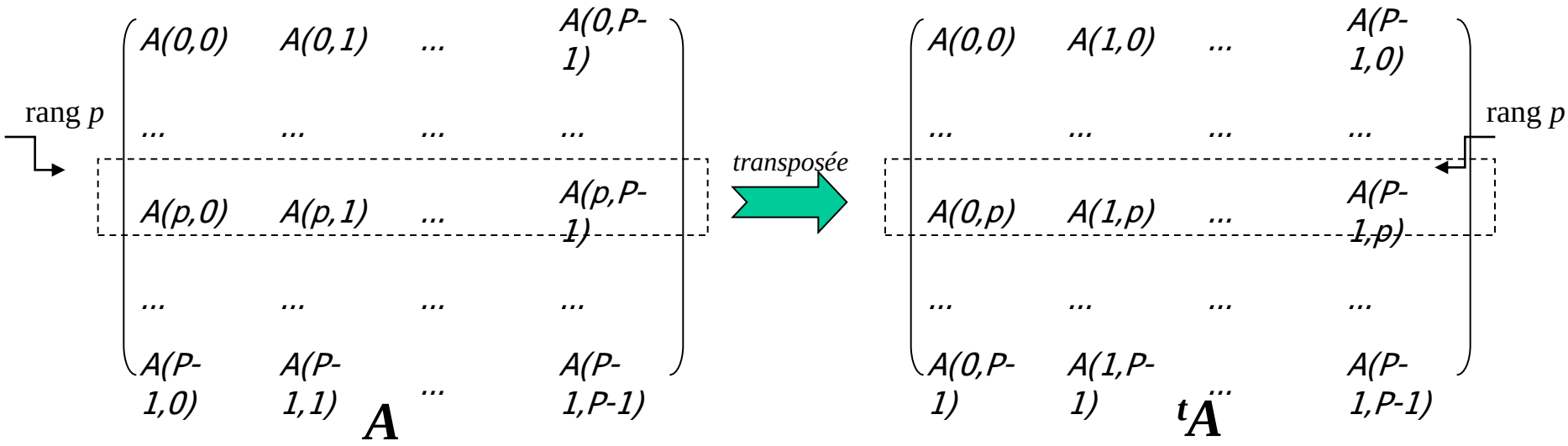
– Chaque processus p possède la ligne $A(p, *)$;

– On veut écrire l'algorithme qui détermine la matrice transposée ${}^tA$ définie par ${}^tA(i, j) = A(j, i)$;

– Exemple avec A définie par :

$$\forall\, i \in [0, P[,\ \forall\, j \in [0, P[,\quad A(i, j) = (i, j)$$

All-to-All


rang p
A(0,0) A(0,1) ... A(0,P-1)
... ...
A(p,0) A(p,1) ... A(p,P-1)
... ...
A(P-1,0) A(P-1,1) ... A(P-1,P-1)
A
transposée
A(0,0) A(1,0) ... A(P-1,0)
... ...
A(0,p) A(1,p) ... A(P-1,p)
... ...
A(0,P-1) A(1,P-1) ... A(P-1,P-1)
tA
rang p

All-to-All

```
int moi, P;

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

int *ligne = (int*)malloc(2*P*sizeof(int));
int *li_tr = (int*)malloc(2*P*sizeof(int));

/* chaque processus a la moi-ième ligne de la
   matrice A  (tableau ligne) */
printf("A[%d] = [ ", moi);
for(int p = 0 ; p < P ; p++) {
    int *elt = ligne + 2*p;
    elt[0] = moi; /* cas particulier … */
    elt[1] = p;   /* A(i,j) = (i,j)    */
    printf("(%d,%d) ", elt[0], elt[1]);
}
printf("]\n");  fflush(stdout);

MPI_Alltoall(ligne, 2, MPI_INT,
             li_tr, 2, MPI_INT, MPI_COMM_WORLD);

/* chaque processus a la moi-ième ligne de la matrice tA (tableau li_tr) */
printf("t(A)[%d] = [ ", moi);
for(int p = 0 ; p < P ; p++) {
    int *elt = li_tr + 2*p;
    printf("(%d,%d) ", elt[0], elt[1]);
} printf("]\n");
```

```
% mpirun -np 4 a.out
A[3] = [ (3,0) (3,1) (3,2) (3,3) ]
A[2] = [ (2,0) (2,1) (2,2) (2,3) ]
A[1] = [ (1,0) (1,1) (1,2) (1,3) ]
A[0] = [ (0,0) (0,1) (0,2) (0,3) ]
t(A)[0] = [ (0,0) (1,0) (2,0) (3,0) ]
t(A)[1] = [ (0,1) (1,1) (2,1) (3,1) ]
t(A)[2] = [ (0,2) (1,2) (2,2) (3,2) ]
t(A)[3] = [ (0,3) (1,3) (2,3) (3,3) ]
%
```

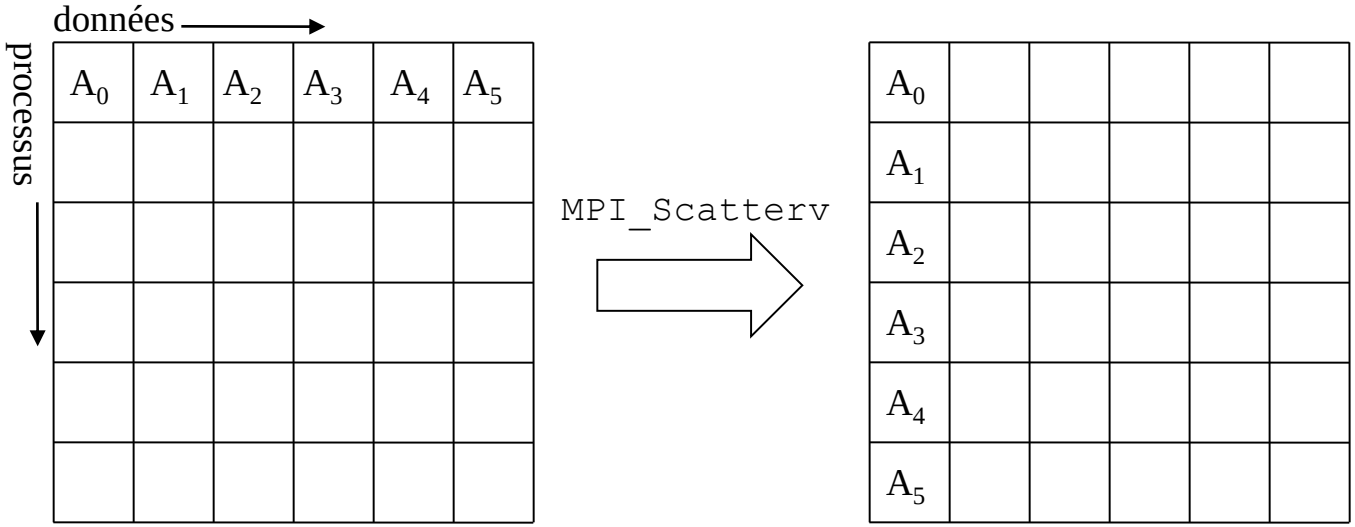
Gestion de données de taille différente

Données de tailles différentes par rang

- Certaines opérations collectives (distribution, rassemblement, alltoall) possèdent des versions étendues pour opérer sur des données dont la taille dépend du rang
- Les noms des fonctions se construisent à partir des noms "classiques" en y ajoutant le suffixe `v` (`v` comme variant)
- Liste :
  - MPI_Gather ➔ MPI_Gatherv
  - MPI_Allgather ➔ MPI_Allgatherv
  - MPI_Scatter ➔ MPI_Scatterv
  - MPI_Alltoall ➔ MPI_Alltoallv

Exemple : MPI_Scatterv

- Les données Ai sont de même type
- Contrairement à `MPI_Scatter`, A0, A1, etc. n'ont pas forcement la même taille (il faudra donc un tableau qui donnera la taille des données par rang)
- Contrairement à MPI_Scatter, les données Ai ne sont pas obligatoirement contiguës en mémoire (il faudra donc un tableau qui donnera les positions des débuts des données pour chaque rang)



Exemple : MPI_Scatterv

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

sendbuf = tableau des données à distribuer à chaque processus du communicateur comm.

Les sendcounts[p] éléments à envoyer au processus p ($0 \leq p < P$) débutent à l'adresse sendbuf + displs[p] .

Les éléments sont tous de type sndtatyp.

Ces arguments ne sont valides que pour la racine root.

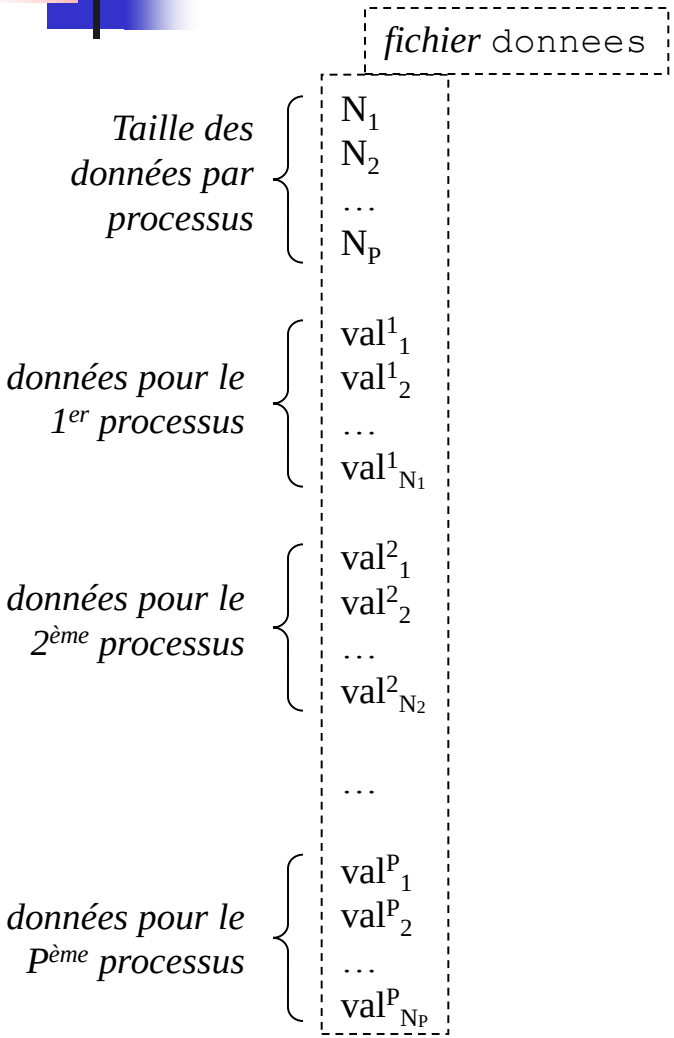
Exemple : MPI_Scatterv

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

recvbuf = tableau des données à recevoir.

Tableau de recvcount éléments de type rcvtatyp.

Exemple : `MPI_Scatterv`



Un seul processus lit un fichier (appelé `donnees`).

Le fichier permet d'attribuer du travail à chaque processus.

Le format (décrit ci à gauche) spécifie d'abord la taille des données par processus.

On trouve ensuite les données elles-mêmes (des nombres réels) regroupées par bloc.

À la fin de la phase de lecture, tous les processus ont en mémoire les données sur lesquelles ils doivent travailler.

Exemple : MPI_Scatterv

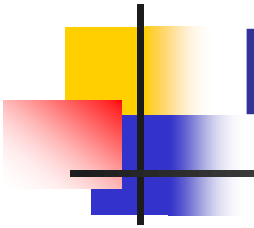
```
int sz;
int *szbuf, *displs;
double *sdbuf, *rcvbuf;

if (moi == root) {
   szbuf = (int*)malloc(P*sizeof(int));
   displs = (int*)malloc((P+1)*sizeof(int));
   fd = fopen("donnees", "r");
   displs[0] = 0;
   for( p = 0 ; p < P ; p++ ) {
      szbuf[p] = lire_val_fichier(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] = lire_val_reel_fichier(fd, p);
   fclose( fd );
}
```

/ Lecture des tailles des données pour chaque processus (tableau szbuf) Calcul des déplacements relatifs (tableau displs) */*

/ Lecture des données par bloc (tableau sdbuf). Ces blocs vont être distribués parmi tous les processeurs */*

Exemple : MPI_Scatterv

```
else {
   szbuf = displs = NULL;
   sdbuf = NULL;
}
/* tous les processus doivent faire appel à MPI_Scatter
   Chaque processus reçoit la taille des données dans sz
 */
MPI_Scatter(szbuf, 1, MPI_INT,
            &sz, 1, MPI_INT,
            root, MPI_COMM_WORLD);

rcvbuf = (double*)malloc(sz*sizeof(double));
/* tous les processus doivent faire appel à MPI_Scatterv
   Chaque processus reçoit ses données dans rcvbuf
 */
MPI_Scatterv(sdbuf, szbuf, displs, MPI_DOUBLE,
             rcvbuf, sz, MPI_DOUBLE,
             root, MPI_COMM_WORLD);
```



Résumé

- Les opérations collectives MPI sont optimisées et ne doivent pas être remplacées par des send/receive
- Les communications collectives sont coûteuses et doivent être utilisées au plus juste dans un logiciel parallèle (i.e., le strict nécessaire)
- Les communications collectives pour lesquelles les résultats sont reçus par tous les processus sont des fonctions de la forme `MPI_All*`
- Les communications collectives pour lesquelles les tailles des données dépendent des rangs sont des fonctions de la forme `MPI_*v`