

Message Passing Interface (MPI) – III

Les communications collectives

Amal KHABOU

`amal.khabou@lri.fr`

Ce qu'on a appris jusqu'ici

- Six fonctions MPI sont suffisantes pour programmer des machines à mémoire distribuée

```
MPI_Init(int *argc, char ***argv);  
MPI_Finalize ();
```

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

```
MPI_Send (void *buf, int count, MPI_Datatype dat,  
int dest, int tag, MPI_Comm comm);  
MPI_Recv (void *buf, int count, MPI_Datatype dat,  
int source, int tag, MPI_Comm comm, MPI_Status  
*status);
```

Pourquoi ne pas s'arrêter là?

- Performance
 - besoin de fonctions permettant d'exploiter au mieux les capacités des architectures
 - Besoin de fonctions pour les modèles de communication typiques
- Facilité d'utilisation
 - besoin de fonctions pour simplifier les tâches récurrentes
 - besoin de fonctions pour simplifier la gestion des applications parallèles

Pourquoi ne pas s'arrêter là?

- Performance
 - communication point à point
 - communications collectives
 - les types de données dérivés ...
- Facilité d'utilisation
 - fonctions de groupement de processus
 - gestion de processus
 - la gestion des erreurs

Opérations collectives

- Tous les processus d'un même groupe doivent participer à la même opération
 - le groupe de processus est défini par le communicateur
 - tous les processus doivent fournir les mêmes arguments
 - pour tout communicateur, on ne peut avoir qu'une seule opération collective en cours à la fois
- Les opérations collectives sont des abstractions de modèles de communication récurrents
 - facilite la programmation
 - permet des optimisations de bas niveau adaptées à l'architecture

Les opérations collectives MPI

MPI_Barrier

MPI_Bcast

MPI_Scatter

MPI_Scatterv

MPI_Gather

MPI_Gatherv

MPI_Allgather

MPI_Allgatherv

MPI_Alltoall

MPI_Alltoallv

MPI_Reduce

MPI_Allreduce

MPI_Reduce_scatter

MPI_Scan

MPI_Exscan

MPI_Alltoallw

D'autres opérations collectives MPI

- Créer ou libérer un communicateur est considéré comme une operation collective
 - e.g. `MPI_Comm_create`
 - e.g. `MPI_Comm_spawn`
- Opérations collectives I/O
 - e.g. `MPI_File_write_all`

.....

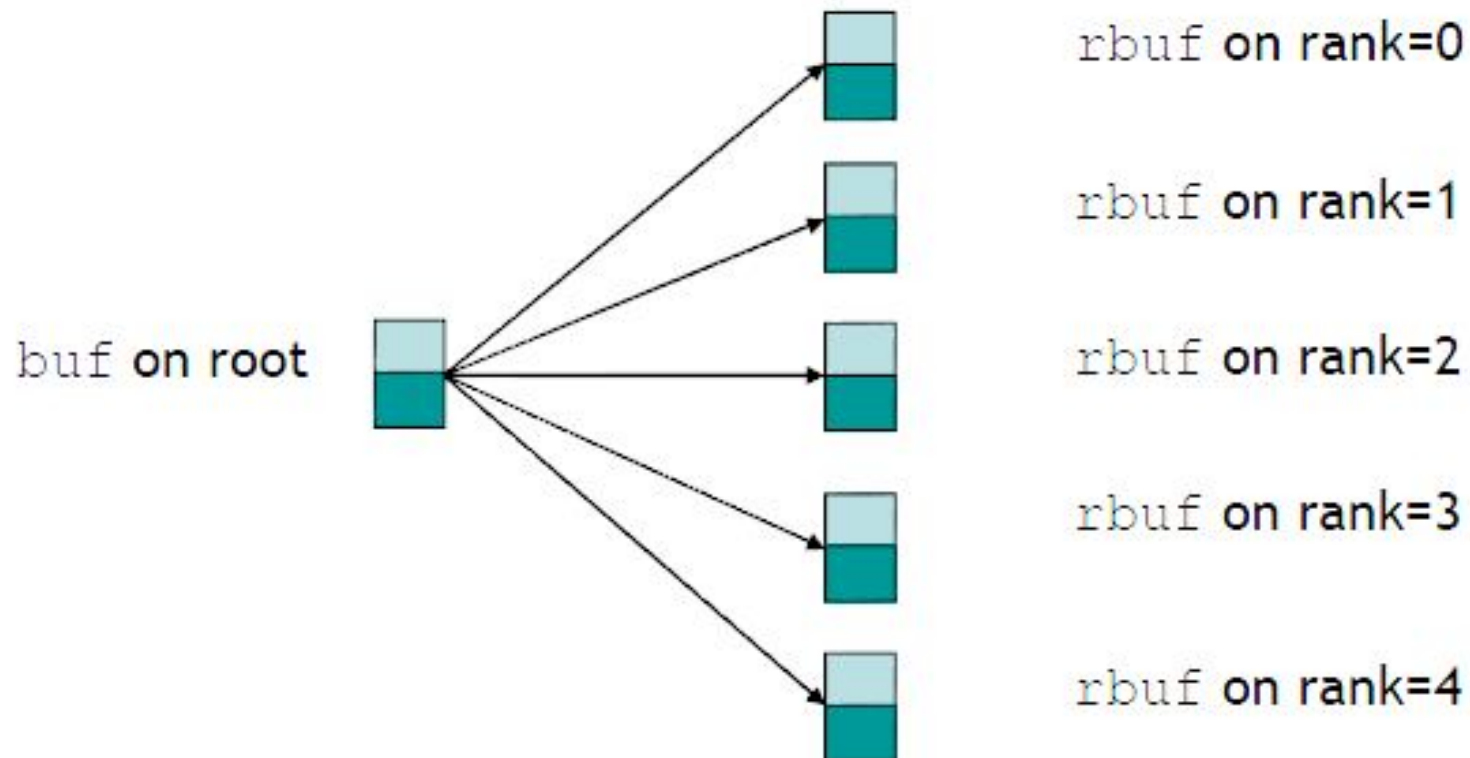
MPI_Bcast

```
MPI_Bcast (void *buf, int cnt, MPI_Datatype  
dat, int root, MPI_Comm comm);
```

- Le processus de rang `root` distribue les données stockées dans `buf` sur tous les processus du communicateur `comm`
- Les données dans `buf` sont identiques pour tous les processus après la fin du `bcast`
- Si on compare à une opération point-à-point : pas de tag car une seule opération collective est permise à la fois

MPI_Bcast (II)

```
MPI_Bcast (buf, 2, MPI_INT, 0, comm);
```



Exemple: distribuer des paramètres globaux

```
int rank, problemsize;
float precision;
MPI_Comm comm=MPI_COMM_WORLD;

MPI_Comm_rank ( comm, &rank );
if (rank == 0 ) {
FILE *myfile;
myfile = fopen("testfile.txt", "r");
fscanf (myfile, "%d", &problemsize);
fscanf (myfile, "%f", &precision);
fclose (myfile);
}

MPI_Bcast (&problemsize, 1, MPI_INT, 0, comm);
MPI_Bcast (&precision, 1, MPI_FLOAT, 0, comm);
```

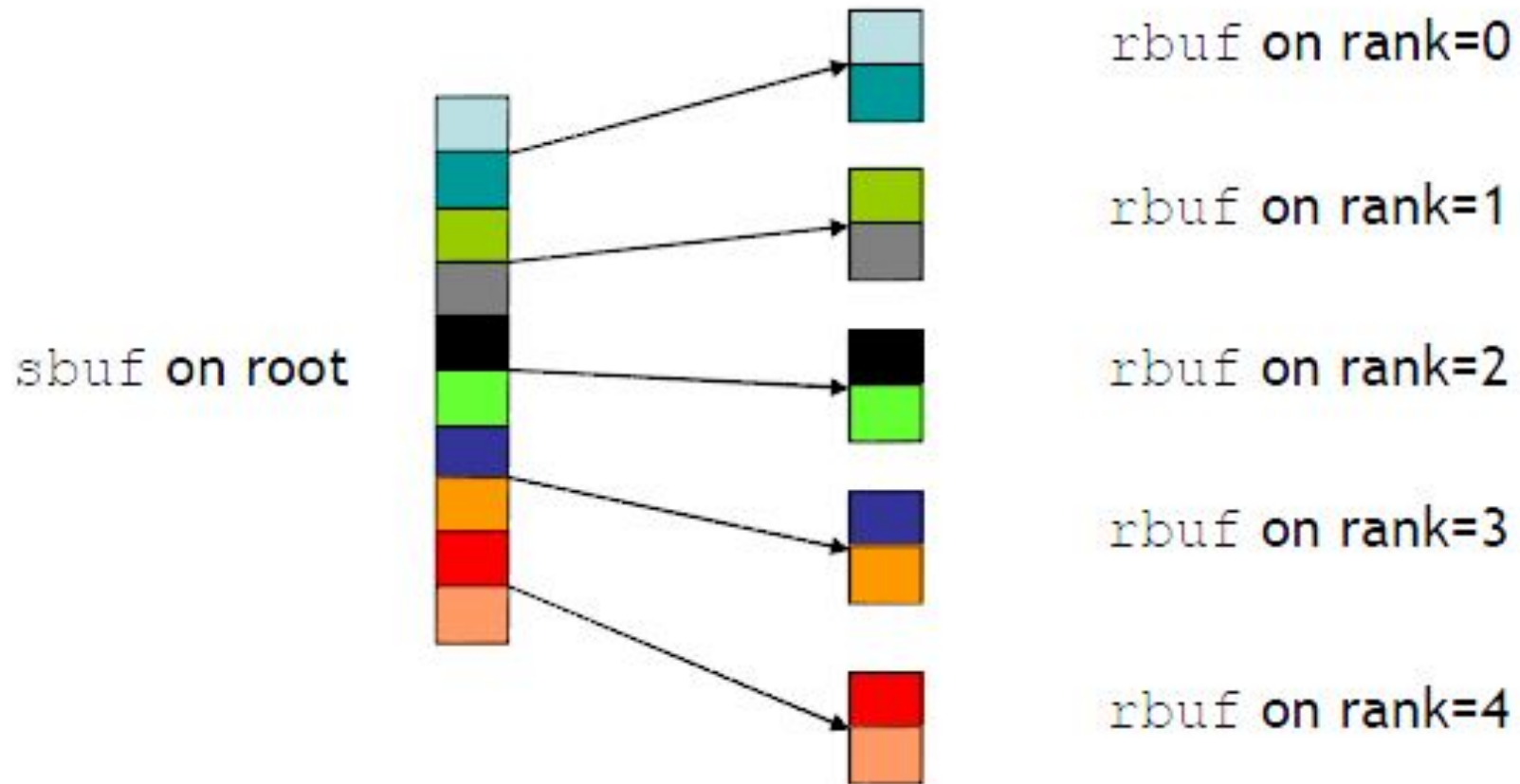
MPI_Scatter

```
MPI_Scatter (void *sbuf, int scnt, MPI_Datatype  
sdat, void *rbuf, int rcnt, MPI_Datatype rdat,  
int root, MPI_Comm comm);
```

- Le processus de rang `root` distribue les données stockées dans `sbuf` sur tous les processus du communicateur `comm`
- Différence par rapport à la diffusion: chaque processus reçoit un segment différent des données du processus `root`

MPI_Scatter (II)

```
MPI_Scatter (sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 0, comm);
```



Exemple: répartir un vecteur sur les processus

```
int rank, size;
float *sbuf, rbuf[3] ;
MPI_Comm comm=MPI_COMM_WORLD;

MPI_Comm_rank ( comm, &rank );
MPI_Comm_size ( comm, &size );

if (rank == root ) {
    sbuf = malloc (3*size*sizeof(float);
    /* set sbuf to required values etc. */
}
/* distribute the vector, 3 Elements for each
process
*/
MPI_Scatter (sbuf, 3, MPI_FLOAT, rbuf, 3, MPI_FLOAT,
root, comm);
if ( rank == root ) {
    free (sbuf);
}
```

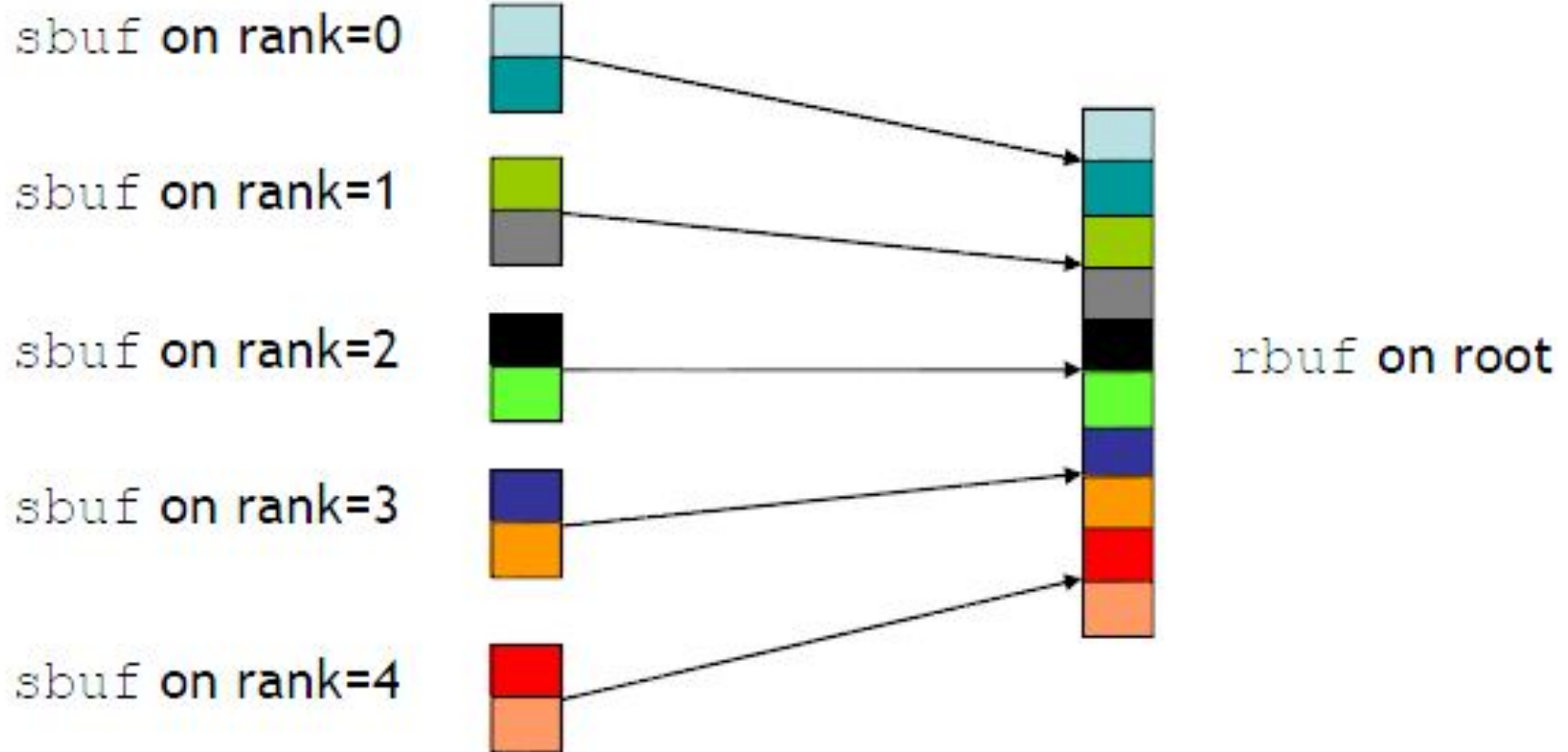
MPI_Gather

```
MPI_Gather (void *sbuf, int scnt, MPI_Datatype
sdat, void *rbuf, int rcnt, MPI_Datatype rdat,
int root, MPI_Comm comm);
```

- L'opération inverse de `MPI_Scatter`
- Le processus de rang `root` reçoit les données stockées dans `sbuf` de tous les autres processus du communicateur `comm` dans le `rbuf`
- Les arguments `rbuf`, `rcnt`, `rdat` sont seulement appropriés au processus `root`

MPI_Gather (II)

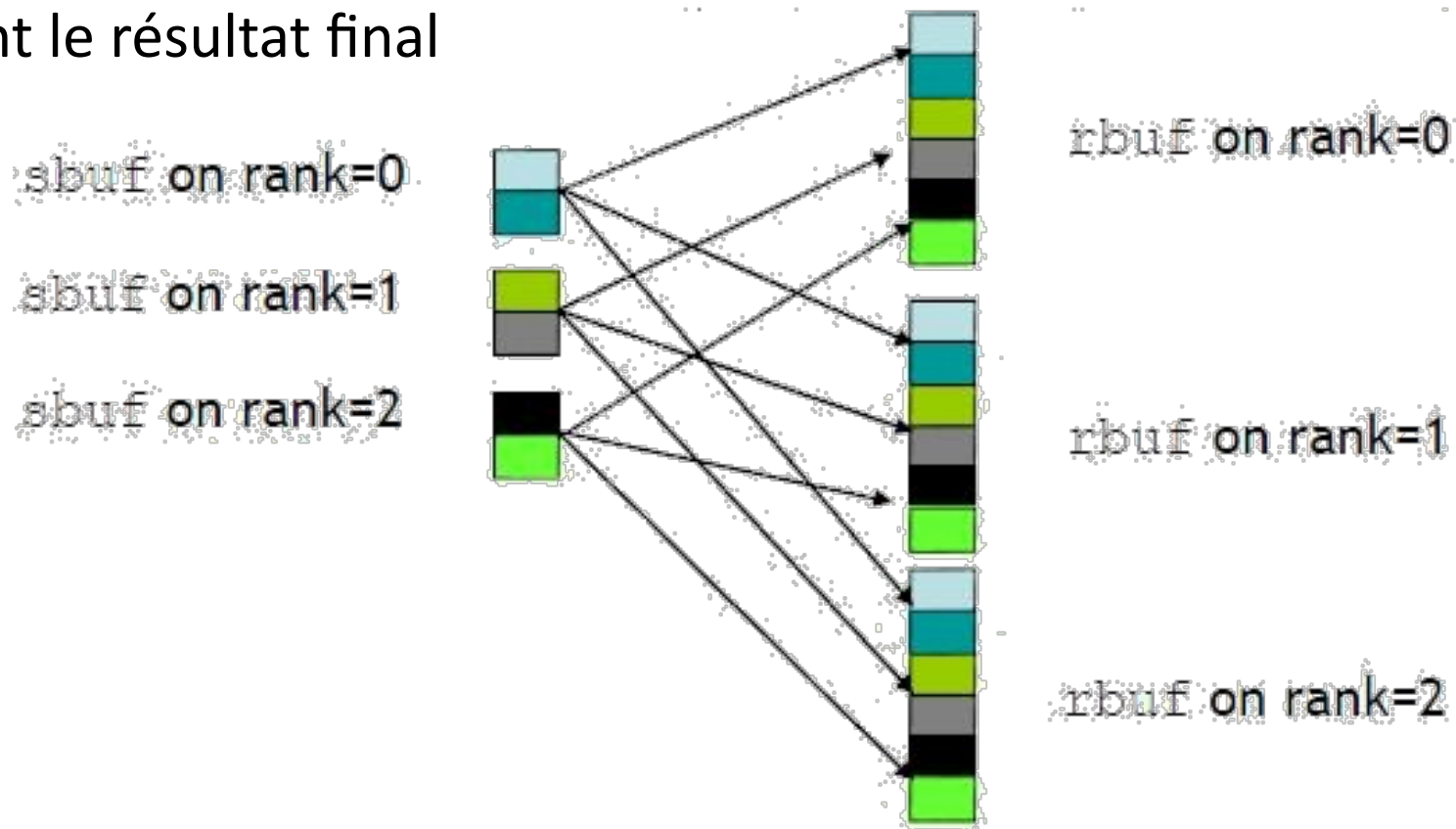
```
MPI_Gather (sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 0, comm);
```



MPI_Allgather

```
MPI_Allgather (void *sbuf, int scnt,  
MPI_Datatype sdat, void *rbuf, int rcnt,  
MPI_Datatype rdat, MPI_Comm comm);
```

- Identique au MPI_Gather, cependant tous les processus ont le résultat final



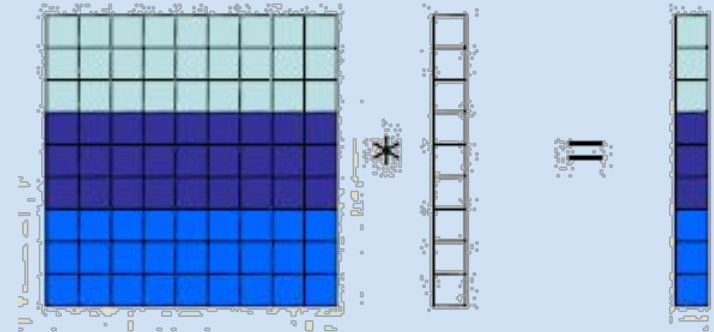
Exemple: multiplication matrice-vecteur avec distribution en blocs de lignes

```
int main( int argc, char **argv)
{
    double A[nlocal][n], b[n];
    double c[nlocal], cglobal[n];
    int i,j;
```

...

```
for (i=0; i<nlocal; i++) {
    for ( j=0;j<n; j++ ) {
        c[i] = c[i] + A(i,j)*b(j);
    }
}
```

```
MPI_Allgather( c, nlocal, MPI_DOUBLE, cglobal,
nlocal, MPI_DOUBLE, MPI_COMM_WORLD );
```



Each process holds the
final result for its part of c

Les opérations de réduction

```
MPI_Reduce (void *inbuf, void *outbuf, int cnt,  
MPI_Datatype dat, MPI_Op op, int root, MPI_Comm  
comm) ;
```

```
MPI_Allreduce (void *inbuf, void *outbuf, int  
cnt, MPI_Datatype dat, MPI_Op op, MPI_Comm  
comm) ;
```

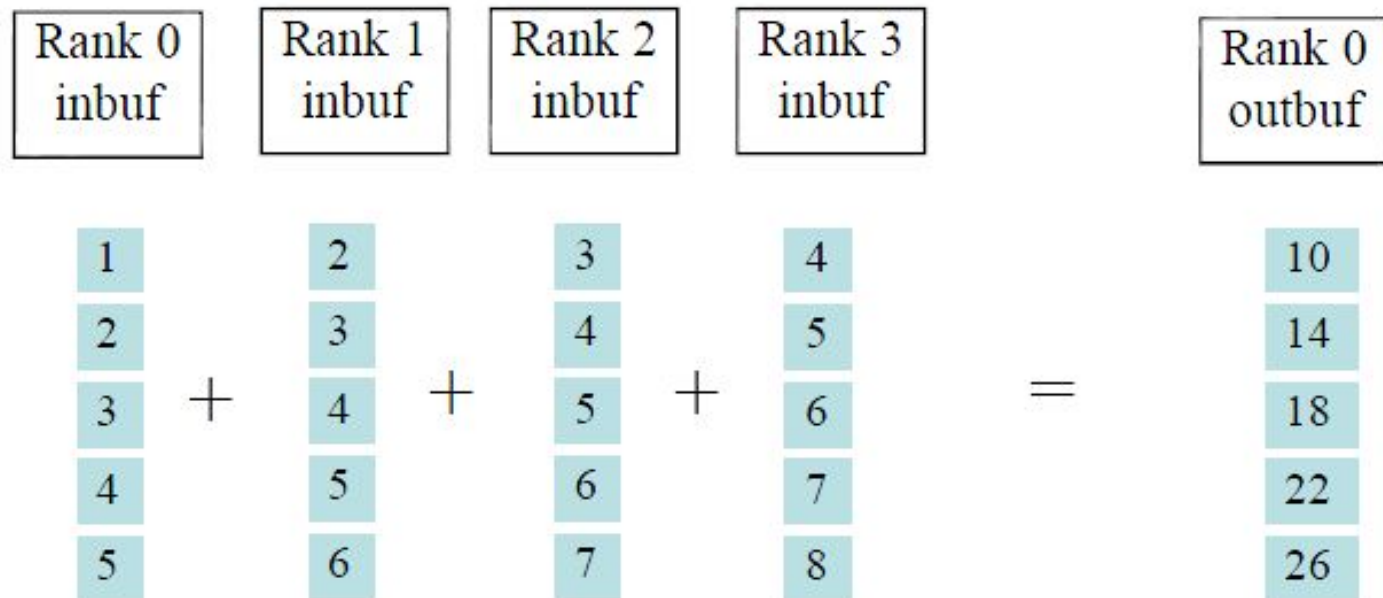
- Effectue des calculs simples (e.g. la somme, le produit) sur les données de tous les processus d'un communicateur
- MPI_Reduce
 - outbuf doit être fourni par tous les processus
 - Le résultat final est disponible au niveau du processus root
- MPI_Allreduce
 - Le résultat final est disponible sur tous les processus

Les opérations de réduction prédéfinies

- MPI_SUM sum
- MPI_PROD product
- MPI_MIN minimum
- MPI_MAX maximum
- MPI_LAND logical and
- MPI_LOR logical or
- MPI_LXOR logical exclusive or
- MPI_BAND binary and
- MPI_BOR binary or
- MPI_BXOR binary exclusive or
- MPI_MAXLOC maximum value and location
- MPI_MINLOC minimum value and location

Une opération de réduction sur des vecteurs

- L'opération de réduction est exécutée sur chacun des éléments



- Réduction de 5 élément avec root = 0

```
MPI_Reduce (inbuf, outbuf, 5, MPI_INT, MPI_SUM,  
0, MPI_COMM_WORLD);
```

Exemple: produit scalaire de 2 vecteurs


```
int main( int argc, char **argv)
{
    int i, rank, size;
    double a_local[N/2];
    double b_local[N/2];
    double s_local, s;
    ...
    s_local = 0;
    for ( i=0; i<N/2; i++ ) {
        s_local = s_local + a_local[i] * b_local[i];
    }

    MPI_Allreduce ( &s_local, &s, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD );
    ...
}
```

Exemple: multiplication matrice-vecteur avec distribution en blocs de colonnes

```
int main( int argc, char **argv)
{
    double A[n][nlocal], b[nlocal];
    double c[n], ct[n];
    int i,j;
    ...
    for (i=0; i<n; i++) {
        for ( j=0;j<nlocal;j++ ) {
            ct[i] = ct[i] + A(i,j)*b(j);
        }
    }
    MPI_Allreduce ( ct, c, n, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD );
}
```

Result of local computation in temporary buffer



MPI_Barrier

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

- Synchronise tous les processus d'un communicateur
 - aucun processus ne peut poursuivre l'exécution de l'application avant que chacun des processus atteigne cette fonction
 - Souvent utiliser avant des fonctions pour mesurer le temps passé sur différentes sections du code
- L'utilisation de la fonction `MPI_Barrier` est fortement déconseillée.