Modèle de programmation parallèle par passage de messages (OpenMPI)

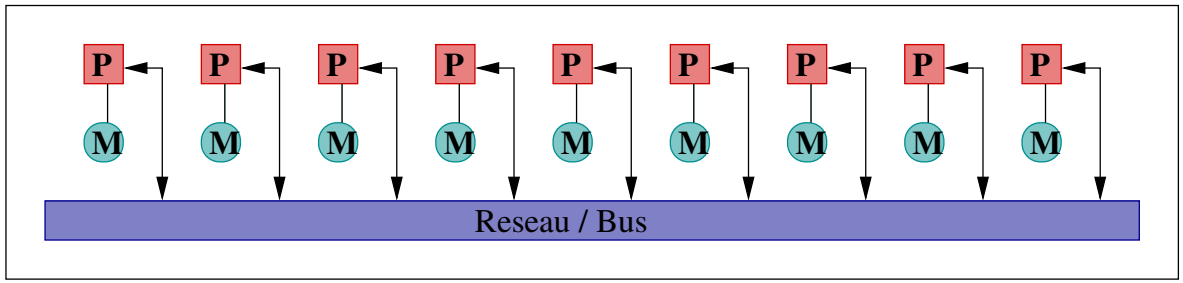
Imane YOUKANA

Université Mohamed Khider - Biskra

le modèle pour les machines parallèles à mémoire distribuée

Il nécessite de la parallélisation explicite

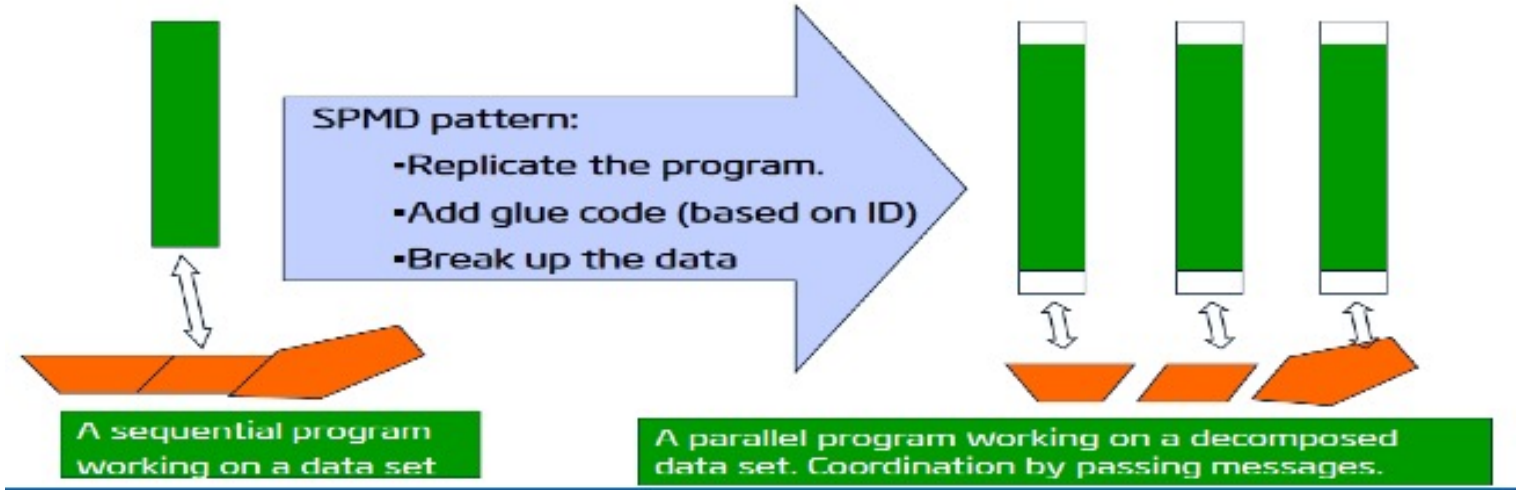
- L'utilisateur doit coder le parallélisme dans son programme.
- Pour exprimer le rôle de chaque processeur dans les phases de lecture ou de lecture/écriture de données distribuées.



Modèle de programmation OpenMPI

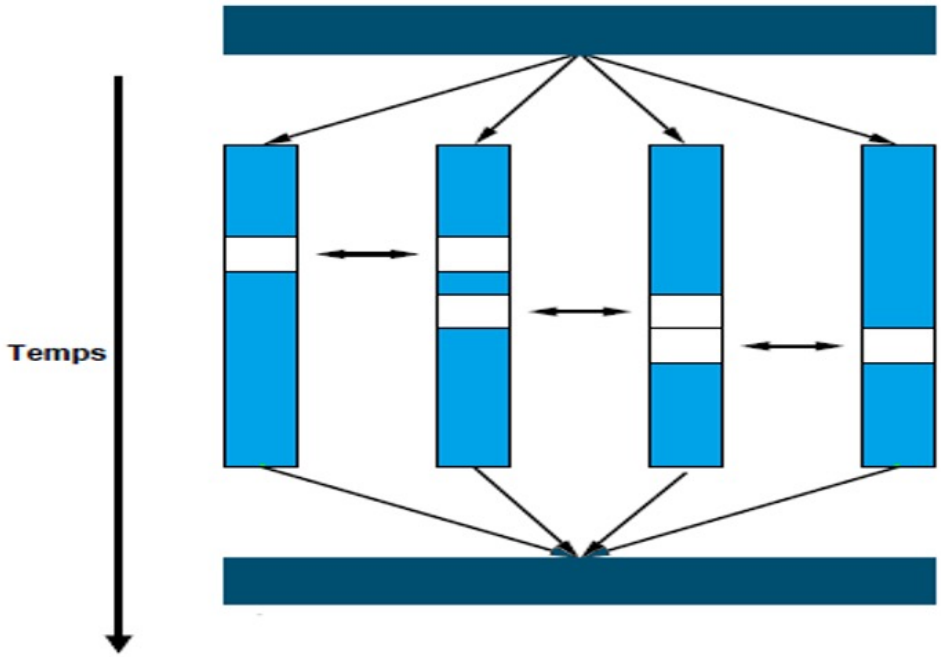
OpenMPI

Message Passing Interface est fondamentalement «Multiple Instruction Multiple Data (distributed memory): Chaque processus dispose de ses propres données et n'a pas d'accès direct aux données des autres processus.



Les échanges de messages

Les communications entre les processus sont effectuées en envoyant et en recevant les messages via un appel à des routines particulières et spécialisées (Send /Receive).



Modèle de programmation OpenMPI

MAITRE | **ESCLAVE**

création des esclaves

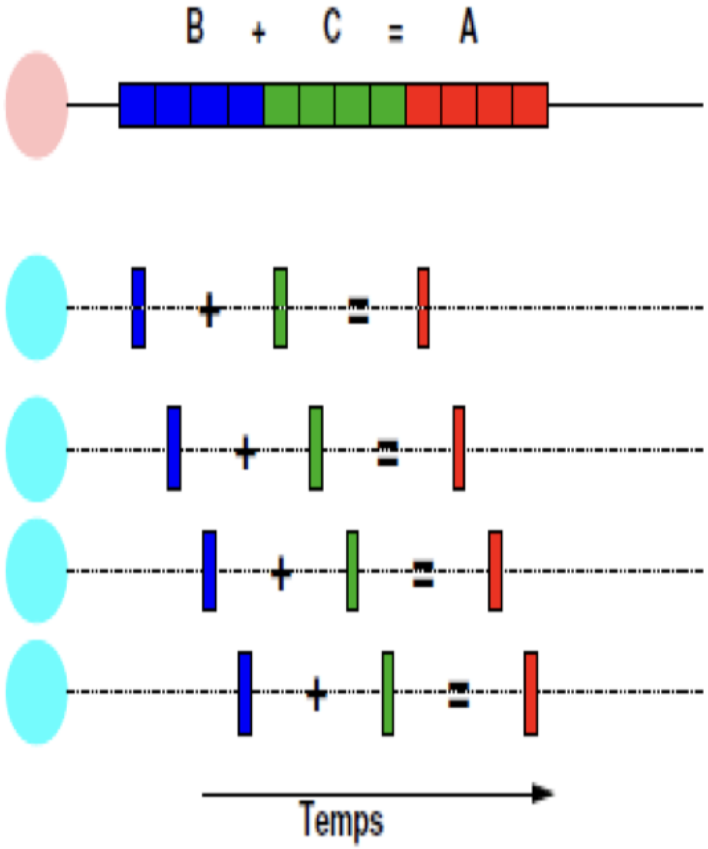
(partage B et C)

pour chaque esclave
  envoi (B,C) → réception (B,C)
fin boucle

A = B + C

pour chaque esclave
  réception (A) ← envoi (A)
fin boucle

(assemblage A)



Installation de OpenMPI

Installation Ubunbtu - Package

```
$ sudo apt-get install openmpi
                       openmpi-common
                       libopenmpi-dev
```

Compilation habituelle mais avec mpicxx

```
mpicxx -c exemple.c
mpicxx -o exemple exemple.o
```

Les opérations Send/Receive

- Les opérations de lecture/écriture nécessitent un moyen de partager des données distribuées.
- La base de ce modèle de programmation s'appuie sur les opérations élémentaires
  - d'envoi de messages : send(void *sendbuf, int size, int dest)
  - de réception de messages : receive(void *recvbuf, int size, int source)

La sémantique des opérations send/recv

La garantie des données

P0	P1
a=100 send(&a,sizeof(int),1) a=0;	receive(&a,sizeof(int),0) printf("%d",a);

Le processeur P1 reçoit 0 ou 100 ?

Les opérations send/receive bloquant

Caractéristiques

- Négociation entre l'émetteur et le récepteur avant la transmission
- Reprise des tâches à la fin de la transmission (fin envoi / fin réception)

Les inconvénients

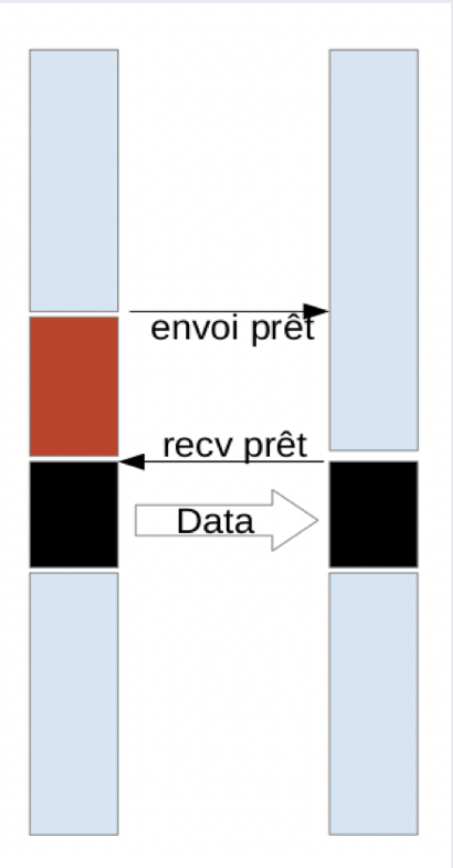
- Un taux d'attente important pour l'émetteur ou le récepteur.
- Des situations d'interblocage entre processeurs

P0	P1
send(&a,sizeof(int),1) receive(&b,sizeof(int),1)	send(&a,sizeof(int),0) receive(&b, sizeof(int), 0)

Les opérations send/receive bloquant

Déroulement

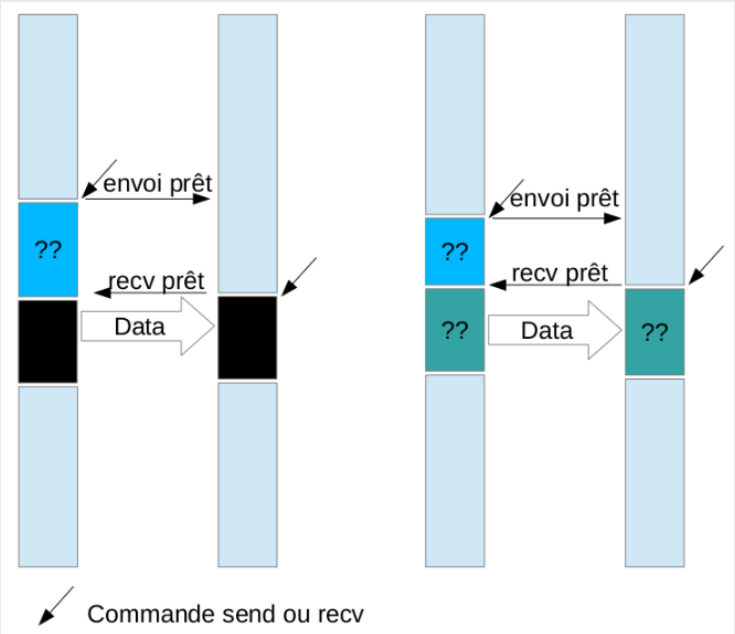
- Cas symétrique récepteur en avance
- \- Temps d'attente -> perte de performances
- \+ Les processeurs sont équilibrés en charge de travail
- \+ Peut constituer une forme de synchronisation



Les opérations Isend/Irecv non bloquant

Caractéristiques

- l'envoi et la réception ne sont pas couplées
- l'appel rend la main :(envoyer et retourner)
- Si le destinataire n'est pas dans une réception, le message est bufférisé
- Quand le destinataire entre dans une réception, il commence par regarder dans ses buffers si il n'a pas déjà reçu



MPI - **M**essage **P**assing **I**nterface

- MPI est une spécification et un standard
  - Les implémentations de MPI sont fournies sous forme de bibliothèques libres ou commerciales
  - Les deux principales: MPICH2, OpenMPI
- Langages supportés: C/C++, Fortran (python, java, perl ...)

MPI permet de gérer

- l'environnement d'exécution
- les communications point à point (Send, Recv)
- les communications collectives (Bcast, Reduce, Scatter,...)
- les groupes de processus et les communicateurs
- la topologie d'inter-connexion des processus(grilles, arbres,...)

Fichier **mpi.h**

Il doit être inclus en entête de tous les programmes MPI.

- Déclaration des prototypes de toutes les routines MPI
- Déclaration de l'ensemble des constantes MPI
- Déclaration de toutes les structures de données

Routines MPI

- Les routines MPI (en C) sont sous deux formes
  1. **MPI_Xxxx()**,
  2. **MPI_Xxxx_xxx()**.

4 routines MPI de base

- pour initialiser et terminer un programme SPMD parallèle
  1. MPI_Init,
  2. MPI_Finalize,
- pour identifier l'équipe de processus
  1. MPI_Comm_size,
  2. MPI_Comm_rank,

MPI sous langage C (2)

La structure d'un programme MPI

```
Inclure le fichier mpi.h

Initialiser l'environnement MPI
    ..........................
    ..........................
    Faire des calculs
    Appeler des routines MPI :
        - communiquer
        - synchroniser
    ..........................
    ..........................
Terminer l'environnement MPI
```

Les domaines de communication

Les communications MPI travaillent dans un communicateur soit un ensemble de processus pouvant communiquer entre eux.

MPI_Init initialise **le communicateur par défaut** MPI_COMM_WORLD qui comprend tous les processus impliqués dans l'exécution parallèle

- Il est possible de construire d'autres communicateurs comme un sous groupe de MPI_COMM_WORLD
- Le type d'un communicateur est `MPI_Comm`

Les domaines de communication

Taille du communicateur et identifiant du processus

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

- Cette routine retourne dans `nprocs` le nombre total de processus du communicateur `comm`

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

- Cette routine initialise `pid` l'identifiant relatif au communicateur `comm` du processus appelant
- `pid` est un nombre entier unique dans le communicateur `comm`
- Initialement, chaque processus a un identifiant unique $pid \in [0, nprocs - 1]$ dans MPI_COMM_WORLD.

Installation de OpenMPI

Installation Ubunbtu - Package

```
$ sudo apt-get install openmpi
                       openmpi-common
                       libopenmpi-dev
```

Compilation habituelle mais avec mpicxx

```
mpicxx -c exemple.c
mpicxx -o exemple exemple.o
```

MPI : Librairie de fonctions de communications

Les communications

- point-à-point : communications entre 2 processus dans un communicateur
- collectives : communications impliquant tous les processus d'un communicateur.

Conseil

Toujours utiliser les fonctions fournies par MPI. Elles sont souvent plus performantes que des versions "maison".

Routine MPI_Send

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

Paramètres

1. void ***buff**: Adresse du tampon de données à envoyer
2. int **count**: Nombre d'éléments dans le tampon de données
3. MPI_Datatype **dtype**: Type de chaque élément envoyé
4. int **dest**: Identifiant du processus de destination
5. int **tag**: Étiquette de message
6. MPI_Comm **comm**: Communicateur

Routine de réception bloquante MPI_Recv

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

Paramètres

1. void ***buff**: Adresse du tampon de reception
2. int **count**: Nombre d'éléments dans le tampon de données
3. MPI_Datatype **dtype**: Type de chaque élément envoyé
4. int **src**: Identifiant du processus émetteur,
5. int **tag**: Étiquette de message
6. MPI_Comm **comm**: Communicateur,
7. MPI_Status ***status**

Types de données en MPI

Type MPI	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_2INT	pair of int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	octet par octet

Exemple Send/Recv

$P_0 \rightarrow P_1$

```
int pid, nprocs, i;
int buff[10];
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
.....
.....
if (pid == 0){
  for(i=0;i<10;i++)
    buff[i]=i;
  MPI_Send(buff, 10, MPI_INT, 1, 9, MPI_COMM_WORLD);
}
if (pid ==1)
  MPI_Recv(buff, 10, MPI_INT, 0, 9, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
.....
.....
```

Type de communications collectives MPI

Les communications collectives peuvent être séparées en 3 catégories

- Une synchronisation globale (**MPI_Barrier**)
- Transferts/échanges de données
  * Diffusion globale des données **MPI_Bcast**
  * Diffusion sélective des données **MPI_Scatter**
  * Collecte des données réparties **MPI_Gather**
  * Collecte par tous les processus des données réparties **MPI_Allgather**
  * Echanges globaux **MPI_Alltoall**
- Opérations de réduction (**MPI_Reduce** et **MPI_Allreduce**)

Communications collectives

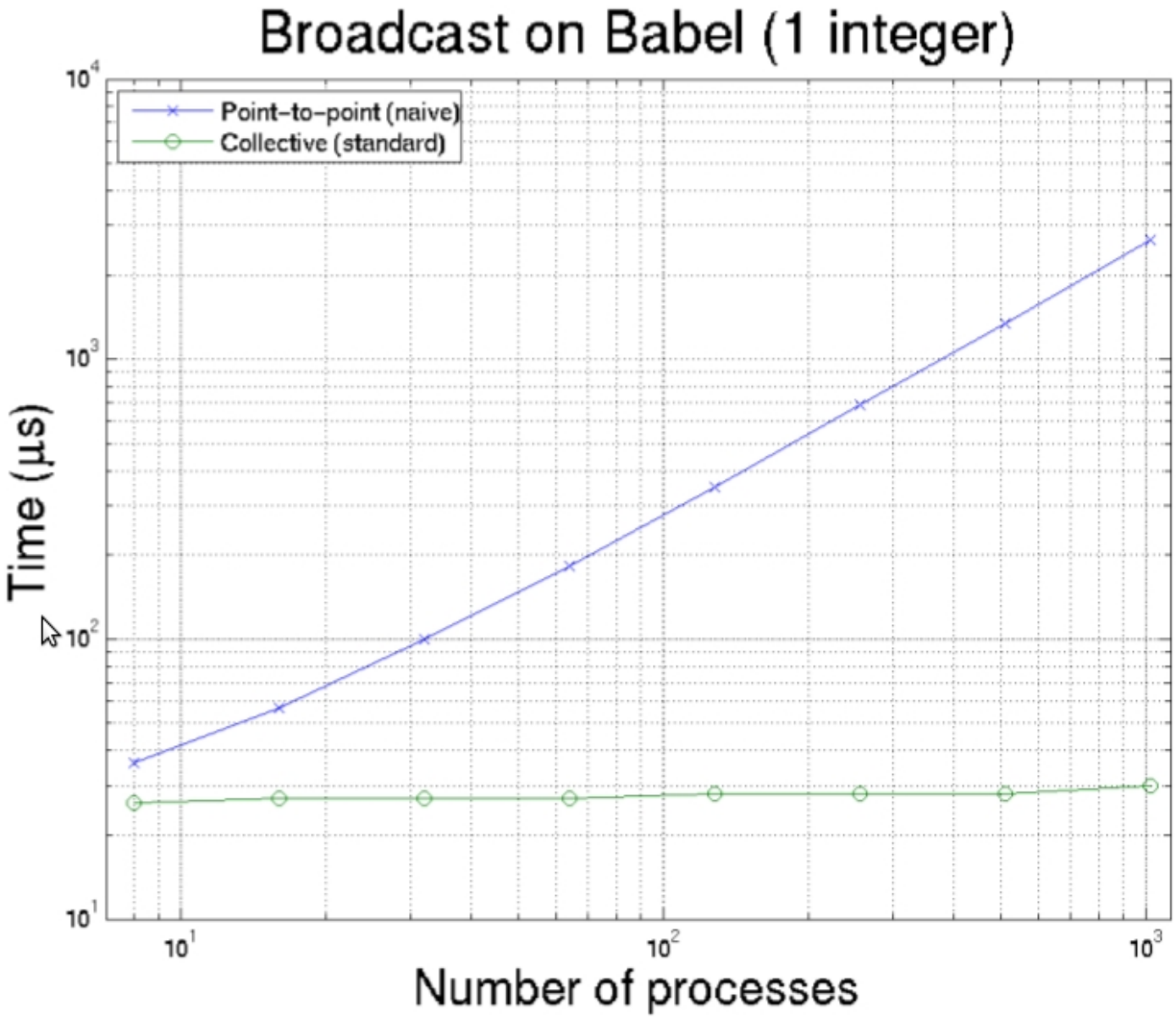
Avantages

- Les communications collectives sont **fortement optimisées**
- C'est l'équivalent d'une série de communications point-à-point en une seule opération

Autres caractéristiques

- Elles peuvent cacher au programmeur un volume de transfert très important
  - MPI_Alltoall avec 1024 processus implique 1 million de messages point-à-point
- Elles impliquent **tous les processus du communicateur**.

Performances

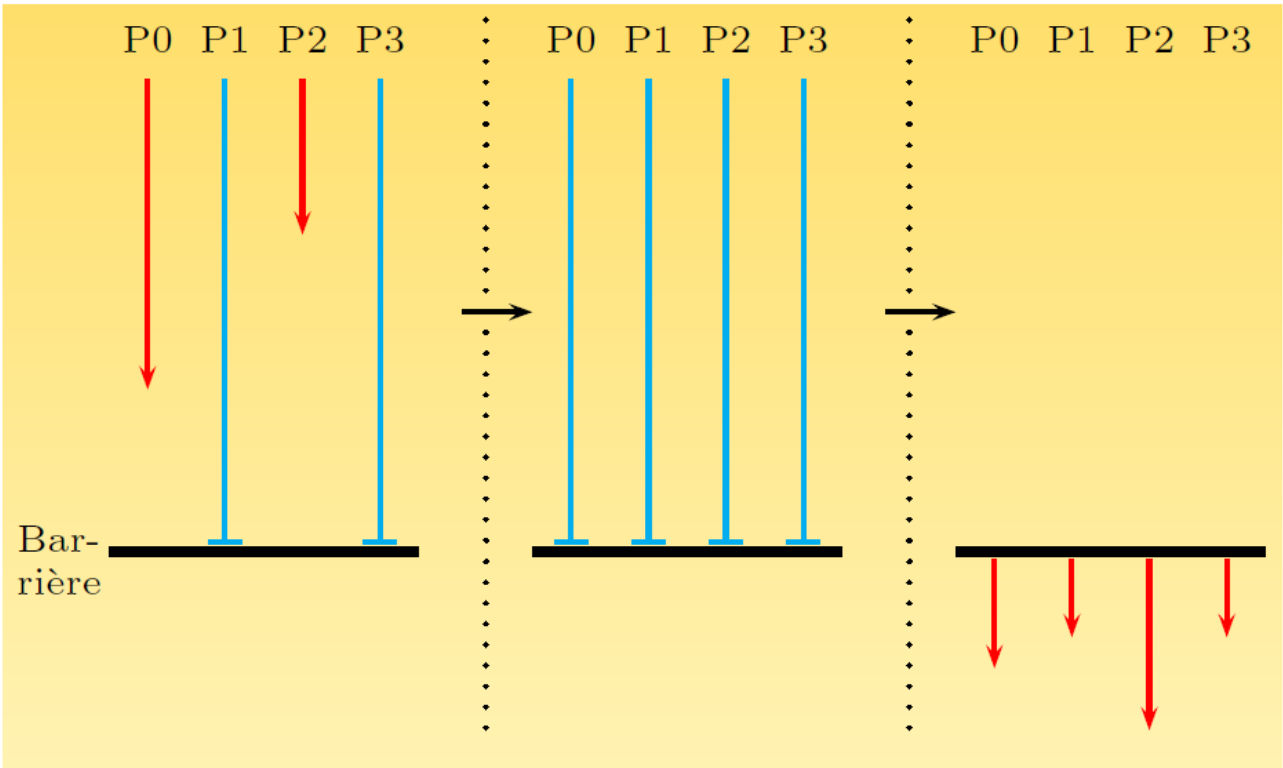


Synchronisation globale: MPI_Barrier

int MPI_Barrier(MPI_Comm comm)

- C'est une routine collective
- Elle permet de bloquer les processus du communicateur comm jusqu'à ce que le dernier soit arrivé à la barrière

Synchronisation globale



Le processus `root`

Un processus de référence

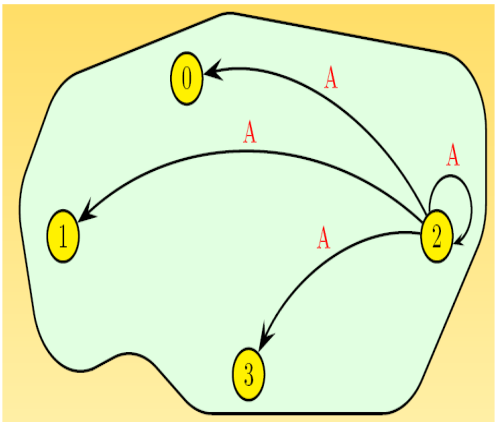
- Pour les communications un-vers-tous ou tous-vers-un un processus joue un rôle particulier
- Ce processus appelé `root` est donné en argument en ligne de commande
- Il sera systématiquement un paramètre de la routine de communication collective
- le programme doit marcher quelque soit l'identifiant de ce processus `root`

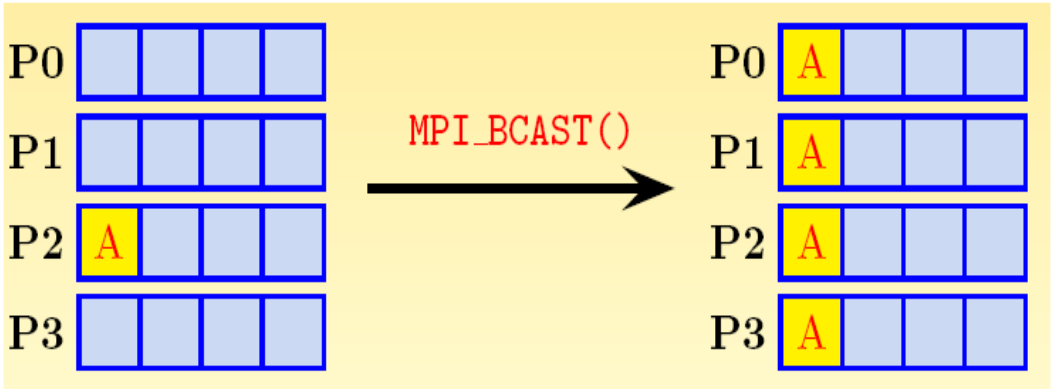
Diffusion générale: MPI_Bcast

Routine MPI_Bcast

- C'est une communication de type un-vers-tous
- Cette routine permet de diffuser à tous les processus une même donnée
- Elle doit être appelée par tous les processus dans un communicateur

Le processus 2 diffuse un message aux autres





Diffusion générale: MPI_Bcast

Prototype

```
int MPI_Bcast( void *buffer, int count,
               MPI_Datatype datatype,
               int root, MPI_Comm comm )
```

Paramètre

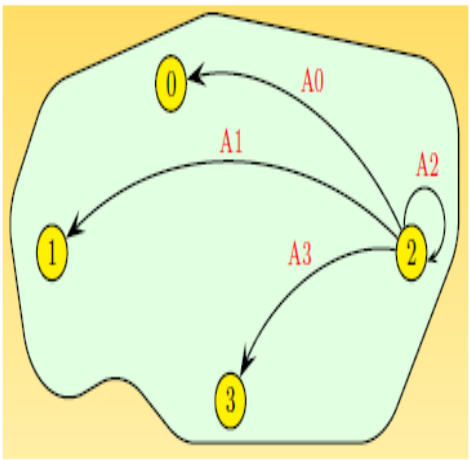
1. void* **buff**: Adresse du buffer
2. int **count**: Nombre d'éléments dans le tampon de données
3. MPI_Datatype **datatype**: Type des éléments envoyés
4. int **root**: Identifiant de la racine de la communication
5. MPI_Comm **comm**: Communicateur

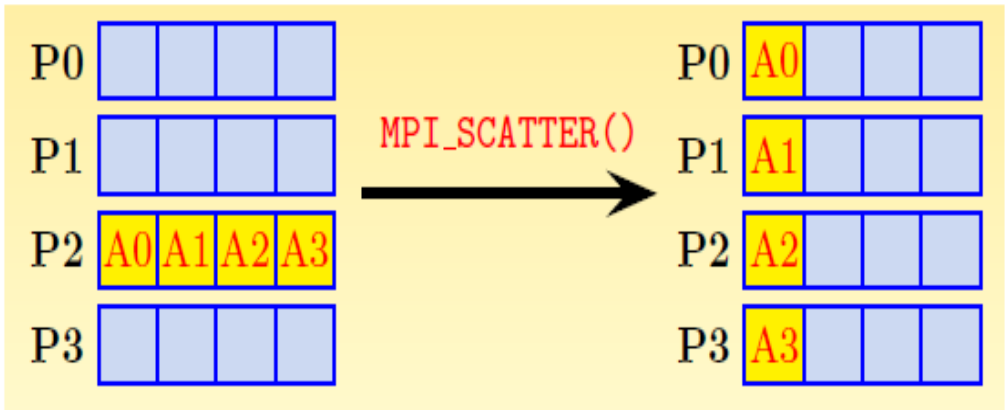
Diffusion sélective: MPI_Scatter

Routine MPI_Scatter

- Cette routine permet au processus `root` de répartir un message sur les processus du communicateur
- C'est une opération de type un-vers-tous, où des données différentes sont envoyées sur chaque processus, suivant leur rang

Le processus 2 répartit des données aux autres processus





Diffusion sélective: MPI_Scatter

Prototype

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

Paramètres

1. void ***sendbuf**: Adresse du tampon d'envoi
2. int **sendcount**: Nb d'éléments envoyés à chaque processus
3. MPI_Datatype **sendtype**: Type d'élément envoyé

Diffusion sélective: MPI_Scatter

Prototype

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

Paramètres

4. void ***recvbuf**: Adresse du tampon de réception
5. int **recvcount**: Nombre d'éléments reçus
6. MPI_Datatype **recvtype**: Type de chaque élément reçu
7. int **root** : Identifiant de la racine de la communication
8. MPI_Comm **comm**: Communicateur

Diffusion sélective: MPI_Scatter

Prototype

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

Interprétation

- Le processus root envoie au processus i sendcount éléments de type sendtype à partir de l'adresse sendbuf + i * sendcount
- Les données sont stockées par chaque récepteur à l'adresse recvbuf

MPI_Scatter -> MPI_Scatterv

Une répartition régulière

MPI_Scatter uniquement sur des données dont la taille est divisible par le nombre de processus

MPI_Scatter -> MPI_Scatterv

Prototype

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

Les nouveaux paramètres

1. int* **sendcounts**: Nombre d'éléments à envoyer par processus
2. int* **displs**: Déplacement dans le buffer d'envoi par processus

Gestion de la taille non divisible

Construction du pointeur et de la taille

- Soit un tableau de `20` entiers alloué et initialisé sur le processus `root`
- Soit une exécution parallèle sur `nprocs=6`

pid	0	1	2	3	4	5
sendcounts						
displs						

MPI_Scatter -> MPI_Scatterv

Gestion avec la routine Scatterv

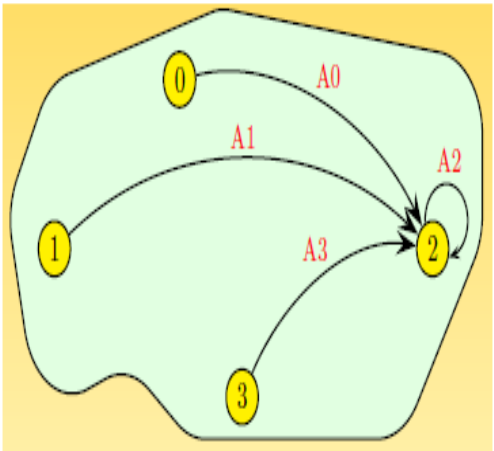
```
int* sendcounts;
int* displ;
int n_local = n/nprocs;
int reste = n%nprocs;
if (pid==root) {
    sendcounts = new int[nprocs];
    displ = new int[nprocs];
    int ptr = 0;
    for (int i=0; i<reste; i++) {
        sendcounts[i] = n_local+1;
        displ[i] = ptr;
        ptr+=(n_local+1);
    }
    for (int i=reste; i<nprocs; i++) {
        sendcounts[i] = n_local;
        displ[i] = ptr;
        ptr+=n_local;
    }
}
if (pid<reste)
    n_local++;
int* tab_r = new int[n_local];
MPI_Scatterv(tab,sendcounts,displ,MPI_INT,tab_r,n_local,MPI_INT,root,
    MPI_COMM_WORLD);
```

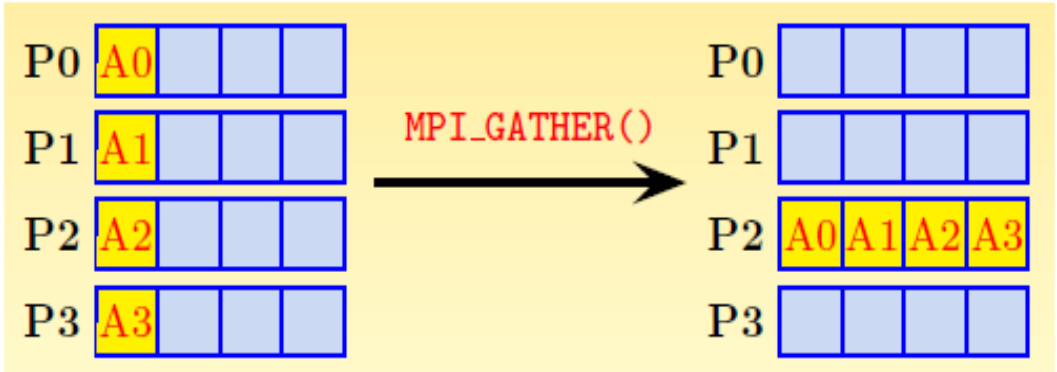
Rassemblement MPI_Gather

Routine MPI_Gather

- Cette fonction permet au processus racine de collecter les données provenant de tous les processus (lui y compris)
- Le résultat n'est connu que par le processus root
- C'est une communication de type tous-vers-un

Le processus 2 collecte des données depuis les autres processus





Collecte MPI_Gather

Prototype

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

Paramètres

1. void ***sendbuf**: Adresse du tampon d'envoi
2. int **sendcount**: Nombre d'éléments envoyés
3. MPI_Datatype **sendtype**: Type d'élément envoyé

Collecte MPI_Gather

Prototype

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

Paramètres

4. void ***recvbuf**: Adresse du tampon de réception
5. int **recvcount**: Nombre d'éléments reçus
6. MPI_Datatype **recvtype**: Type d'élément reçu
7. int **root**: Identifiant du processus racine
8. MPI_Comm **comm**: communicateur

MPI_Gather -> MPI_Gatherv

Une répartition régulière

On rassemble toujours *nprocs* $\times$ *taille_locale* données sur un processus

Sinon :

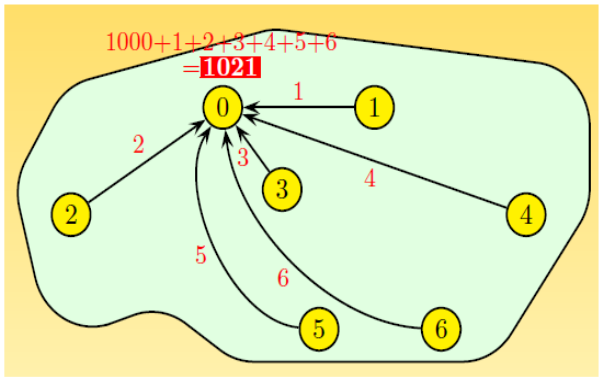
```
int MPI_Gatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    int* recvcounts, int* displs,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

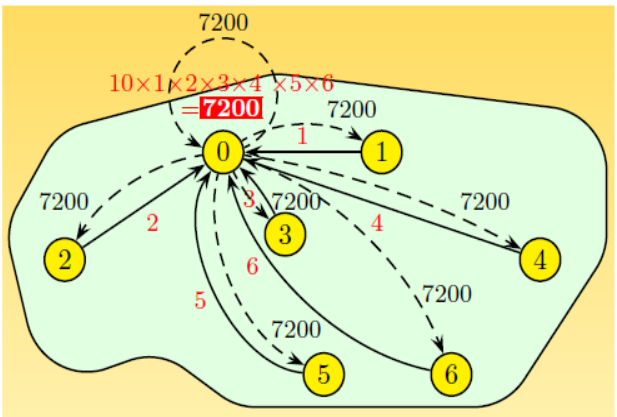
Les nouveaux paramètres

1. int ***recvcounts**: Nombre d'éléments reçus par processus
2. int ***displs**: Déplacement dans le buffer de réception

Opération classique

- La réduction est une opération appliquée aux données réparties sur un ensemble de processus pour n'obtenir qu'une seule valeur sur
  * un seul processus : **MPI_Reduce**
  * tous les processus : **MPI_Allreduce** (MPI_Reduce suivi d'un MPI_Bcast)





MPI_Reduce et MPI_Allreduce

Tableau des opérations principales

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_LXOR	XOR logique

Routine MPI_Reduce

Prototype

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count,MPI_Datatype datatype,
               MPI_Op op,int root,MPI_Comm comm)
```

Paramètres

1. void ***sendbuf**: Adresse du tampon d'envoi
2. void ***recvbuf**: Adresse du tampon de réception
3. int **count**: Nombre d'éléments envoyés
4. MPI_Datatype **datatype**: Type des éléments
5. MPI_Op **op** Opération réalisée sur des données envoyées
6. int `root`: Identifiant de la racine de la communication
7. MPI_Comm **comm**: Communicateur