



# Communication point a point et Communication collective Avec MPI

## Programmation Parallèle - MSIDSD -



**Mohamed walid hajoub**



[Mohamedwalidhajoub1@gmail.com](mailto:Mohamedwalidhajoub1@gmail.com)

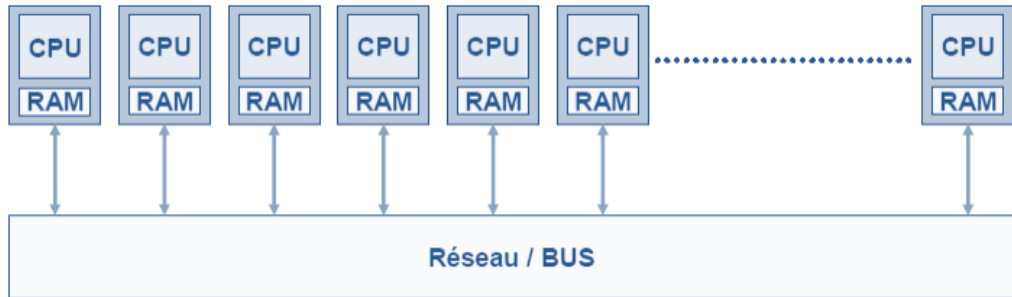
# Plan

- 1 Introduction
- 2 Généralité sur MPI :
- 3 Communication point a point et communication collective
- 4 Conclusion



# Introduction

- On dispose de  $n$  machines
- Ces machines sont connectées en réseau



- Comment utiliser la machine globale à  $n$  processeurs constituée par l'ensemble de ces  $n$  machines ?

# Introduction

Une réponse : le passage de messages (*message passing*)

- Faire exécuter un processus sur chaque processeur disponible
- Effectuer des transferts de données explicites entre les processeurs
- Synchroniser les processus explicitement

**Il nous faut une Interface pour Passer des  
Messages entre Processeurs MPI**

# Généralité sur MPI

**MPI** signifie Interface de transmission de messages.

- Il s'agit d'un ensemble de déclarations d'API sur le passage de messages, tels que :
  - l'envoi,
  - la réception,
  - la diffusion, etc.
- L'idée de "passer un message" est plutôt abstraite.
- Cela pourrait signifier passer un message entre :
  - des processus locaux sur des hôtes;
  - ou des processus distribués en réseau.



## Généralité sur MPI

Une application **MPI** est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI

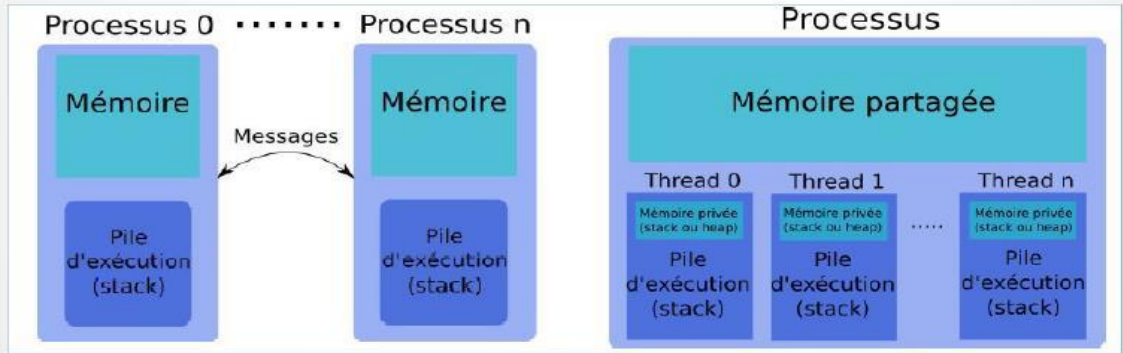
**MPI** est un moyen de programmer sur des périphériques de mémoire distribuée.

Le parallélisme avec MPI se produit là où chaque processus parallèle fonctionne dans son propre espace mémoire indépendamment des autres.

Offre des domaines de communication séparés

**MPI** utilise un schéma à **mémoire distribuée**

**OpenMP** utilise un schéma à **mémoire partagée**

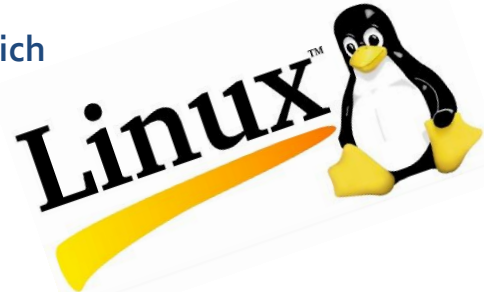


**Schéma MPI**

**Schéma OpenMP**

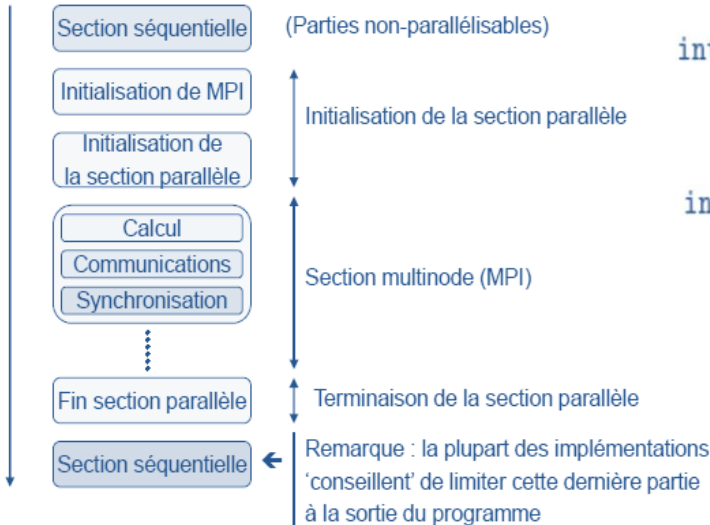
# Installation MPI Sous Linux

**Sudo apt-get update**  
**Sudo apt-get install mpich**





# Structure programme MPI



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

**Toute unité de programme appelant des sous-programmes MPI doit inclure un fichier d'en-têtes. Il faut inclure le fichier `<mpi.h>`.**

# Fonction de base MPI

Initialisation de l'environnement MPI

**MPI\_Init** (argc , char argv)

Obtention du rang du processus

C : MPI\_Comm\_rank (MPI\_COMM\_WORLD, rank)

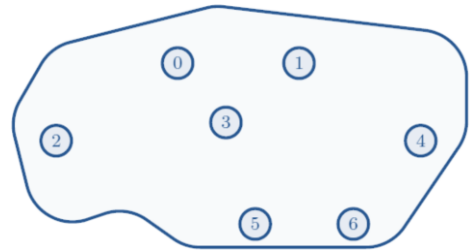
Obtention du nombre de processus

**MPI\_Comm\_size** (MPI\_COMM\_WORLD, &size)

Terminaison de l'environnement MPI (il est en général recommandé de terminer immédiatement après cette instruction)

**MPI\_Finalize**()

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



## Exemple MPI

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char ** argv) {

int rang, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rang);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

printf("Bonjour, je suis %d (parmi %d process)\n", rang, nprocs);

MPI_Finalize();
}
```

## Compilation et Exécution

Compilation : `mpicc -o test test.c`

Exécution : `mpirun -np 2 ./test`



```
(kali㉿kali)-[~/Desktop/MPI]
$ mpirun -np 2 ./test
Bonjour, je suis 0 (parmi 2 process)
Bonjour, je suis 1 (parmi 2 process)
```

# Structure d'un message MPI

---

## Structure d'un message MPI

Par l'intermédiaire d'un message, un process peut envoyer des données à un autre.

En plus des données, le message contient une "enveloppe" qui contient plusieurs champs:

- **Source**: rang du process qui envoie.
- **Destination**: rang du process qui reçoit
- **Etiquette**: entier qui identifie le message de manière unique.
- **Communicateur**: communicateur au sein duquel se fait l'échange.

Les données échangées sont typées (entiers, réels, etc. ou types dérivés personnels)

Il existe dans chaque cas plusieurs modes de transfert, faisant appel à des protocoles différents

# Communication point a point

---

## Communication point a point

Une communication dite point à point a lien entre deux processus, l'un appelé processus **émetteur** et l'autre processus **récepteur** (ou destinataire).

- Elles permettent d'envoyer et recevoir des données entre deux processus
- Les deux processus initient la communication, l'un qui envoie la donnée, le second qui la reçoit
- Les communications sont identifiées par des tags
- Il faut préciser d'avance la taille des éléments envoyés, ainsi que leur type



## Envoie de message

### Type

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

### Arguments

Les arguments sont tous des paramètres d'entrée à la fonction :

- buf            adresse du début du *buffer*
- count        nombre de valeurs à envoyer
- datatype    type de chaque valeur
- dest         rang du processus de destination
- tag          étiquette pour identifier le message
- comm        *communicator*: groupe de processus

## Envoie de message

Dans le cas le plus simple, on fixe *comm* à *MPI\_COMM\_WORLD*

- Comme un processus peut recevoir plusieurs messages (et éventuellement plusieurs d'un même émetteur), tag peut servir à les identifier. Si leur identité est connue (par le contexte du programme), on peut fixer tag à n'importe quelle valeur lors de l'émission et à *MPI\_ANY\_TAG* lors de la réception.
- L'émetteur place les valeurs à émettre dans des lieux successifs de la mémoire locale avant d'appeler *MPI\_Send*
- Si on fixe *count=0*, le message ne contiendra que ses informations de contrôle source, destination, tag et communicateur et pourra servir à une synchronisation ou à signifier un événement.

## Types prédéfinies MPI

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

## Réception de message

**Status** est une structure de type **MPI\_Status** contenant trois champs **status.MPI\_SOURCE**, **status.MPI\_TAG** et **status.MPI\_ERROR** contenant l'émetteur effectif, l'étiquette et le code d'erreur.

### Type

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

### Arguments

Les arguments `buf` et `status` sont des paramètres de sortie, les autres sont des paramètres d'entrée :

- le buffer de réception est constitué de `count` valeurs successives de type `datatype` situées à partir de l'adresse `buf`. Si moins de valeurs sont reçues, elles sont placées au début. S'il y en a trop, une erreur de débordement est déclenchée.

# Résume

## Envoi de données synchrone :

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

- Le *tag* permet d'identifier le message de façon unique

## Réception de données synchrone :

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

- Le *tag* doit être identique que le tag du Send

## Mise en œuvre

***MPI\_SUCCESS*** est une constante prédéfinie qui permet de tester le code de retour d'une fonction MPI.

```
if (status == MPI_SUCCESS )  
{  
  Tâches à faire .....  
}
```

- Les tags de communication permettent d'identifier une communication particulière dans un ensemble
- Il est possible dans le cas des opérations de réception, de recevoir depuis n'importe
- quel tag en utilisant le mot clef : **MPI\_ANY\_TAG** (à éviter)

# Application Communication point a point

---

# Communication Collective

---

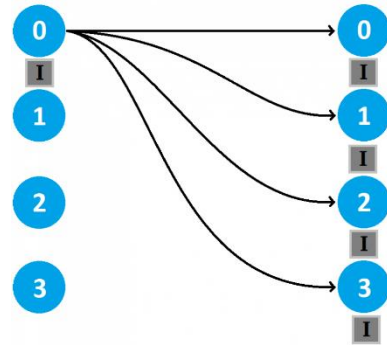


## Communication collective

Afin de simplifier certaines opérations récurrentes, on peut utiliser des opérations qui sont effectuées sur un ensemble de processus (sur leur domaine de communication)

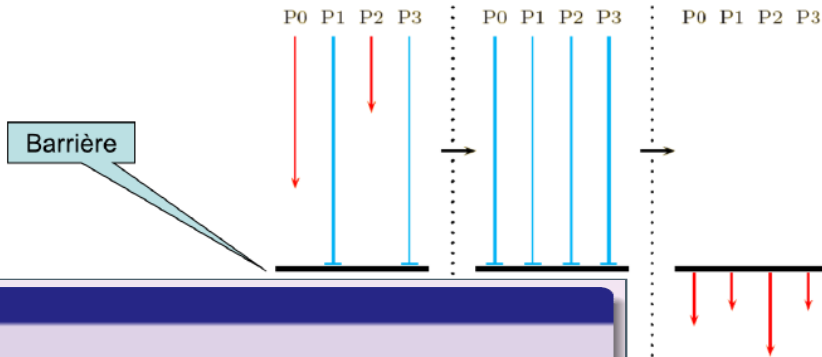
- Ces opérations sont typiquement :

- Des synchronisations explicites
- Des échanges de données entre processeurs :
  - Broadcast
  - Scatter
  - Gather
  - All-to-All
- Des réductions



## Communication collective : Synchronisation

Barrière de synchronisation : tous les processus d'un domaine de communication attendent que le dernier processus soit arrivé à la barrière de synchronisation avant de continuer l'exécution.



### Exemple en C

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

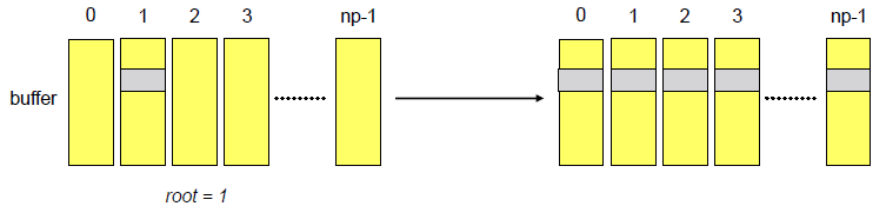
## Communication collective : Broadcast

Une opération de *broadcast* permet de distribuer à tous les processeurs une même donnée.

Communication de type un-vers-tous, depuis un processus 'root' spécifiée par tous les processus (identique pour tous) du domaine

Prototypes :

Int **MPI\_Bcast**(void\*buffer, int count, MPI\_Datatype datatype, int root, MPI\_Commcomm);

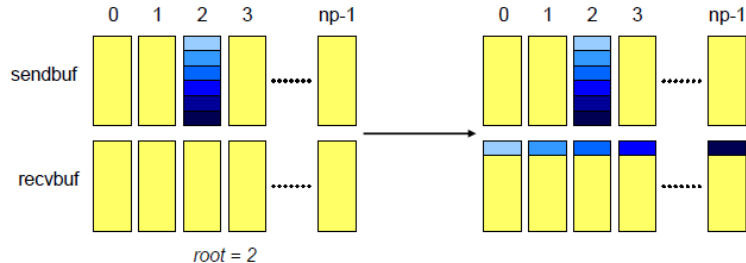


## Communication collective : Scatter

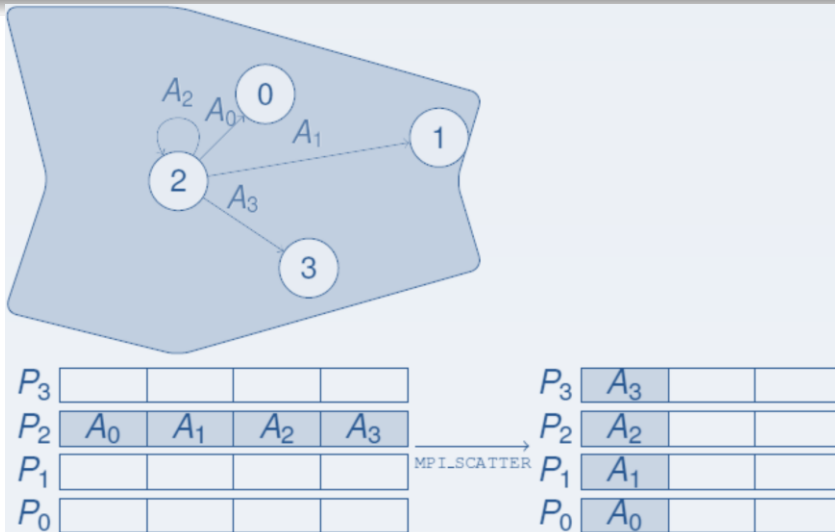
Opération de type un-vers-tous, où des données différentes sont envoyées sur chaque processus receveur, suivant leur rang

Prototypes :

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



## Communication collective : Scatter

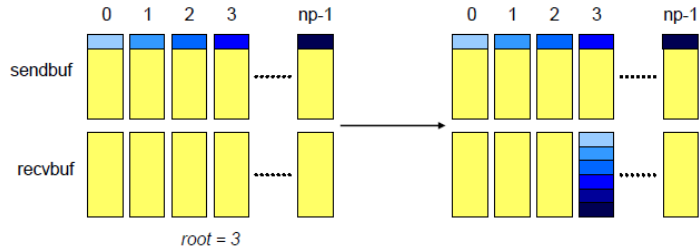


## Communication collective :Gather

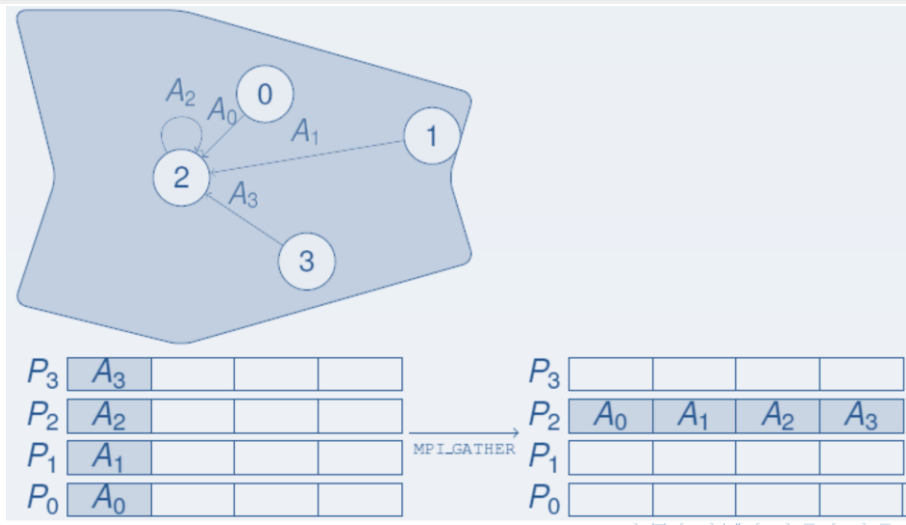
Opération de type tous-vers-un, où des données différentes sont reçues par le processeur receveur, suivant leur rang

Prototypes :

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



## Communication collective : Gather

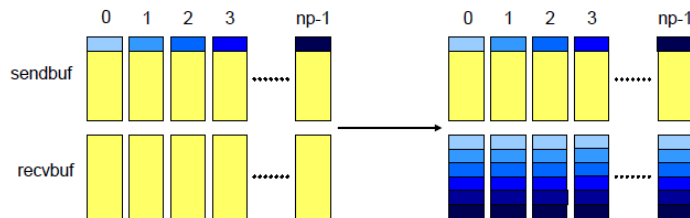


## Communication collective : ALLGather

Opération de type tous-vers-un, où des données différentes sont reçues par le processeur receveur, suivant leur rang

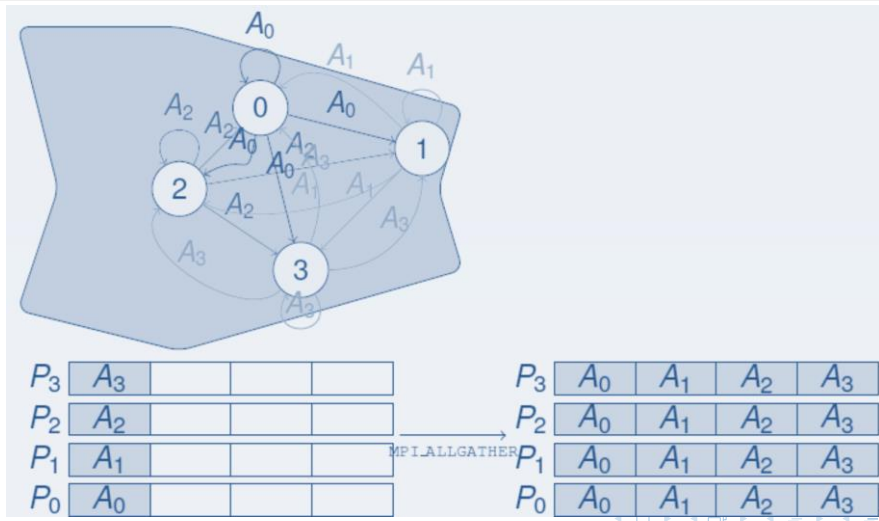
Prototypes :

```
Int MPI_AllGather(void* sendbuf, int sendcount, MPI_Datatype  
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,  
MPI_Comm communicator);
```





# Communication collective : ALLGather



## Communication collective : Réduction

Une réduction permet d'effectuer sur un des données distribuées dans un ensemble de processeurs une opération arithmétique de type addition, minima/maxima, ...

Prototype :

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

Dans la forme ***MPI\_Reduce()*** seul le processeur *root* reçoit le résultat  
Il existe la forme ***MPI\_AllReduce()***, ou tous les processus reçoivent le résultat


MPI_Op	Opération
MPI_MIN	Minimum
MPI_MAX	Maximum
MPI_SUM	Somme
MPI_PROD	Produit élément à élément
MPI_LAND	ET logique
MPI_BAND	ET bit par bit
MPI_LOR	OU logique
MPI_BOR	OU bit par bit
MPI_LXOR	OU exclusif logique
MPI_BXOR	OU exclusif bit par bit
MPI_MINLOC	Minimum et emplacement
MPI_MAXLOC	Maximum et emplacement


## Résume MPI


Écrire un programme en MPI qui permet de calculer la somme des éléments d'un Vecteur **V** de **N** entiers générés aléatoires ?


$$S = \sum_{k=0}^{N-1} V[k]$$


Code source vers lien suivant :

 broadcast.c

 gather.c

 reduce.c

 scatter.c

 sendRecv.c


 somme.c

 table.c

 test.c



<https://github.com/HajoubWalid2000/parallel-programming>

# Question ?

## Enrichir la discussion



Mohamed walid hajoub



[Mohamedwalidhajoub1@gmail.com](mailto:Mohamedwalidhajoub1@gmail.com)

**Merci pour vous attention**