

# Langages du parallélisme

# Informations sur le cours

**6 séances x 2h**

**Examen écrit (OpenMP, MPI, MPI+X)**

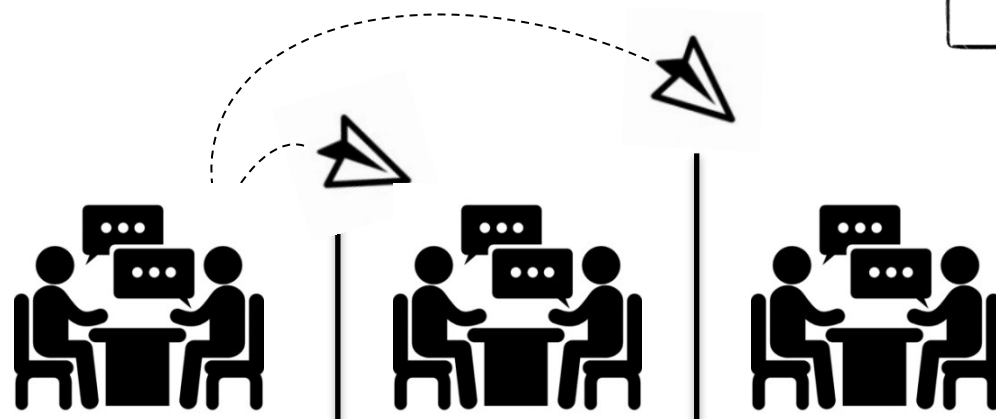
**Contact:** [emmanuelle.saillard@inria.fr](mailto:emmanuelle.saillard@inria.fr)

# Modèles de programmation parallèle



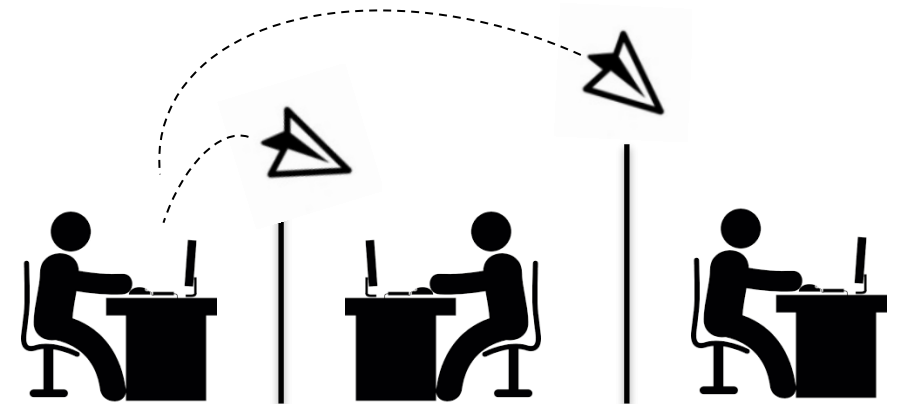
Modèle à mémoire partagée

1ÈRE PARTIE DU COURS: OPENMP



Programmation hybride

3ÈME PARTIE DU COURS: MPI+X



Modèle à mémoire distribuée

2ÈME PARTIE DU COURS: MPI

2

# MESSAGE PASSING INTERFACE (MPI)

# Message Passing Interface

1. Introduction à MPI
2. Communications point à point
  1. Communications dites bloquantes
  2. Communications non bloquantes
  3. Communications persistantes

# Message Passing Interface

## 1. Introduction à MPI

## 2. Communications point à point

1. Communications dites bloquantes

2. Communications non bloquantes

3. Communications persistantes

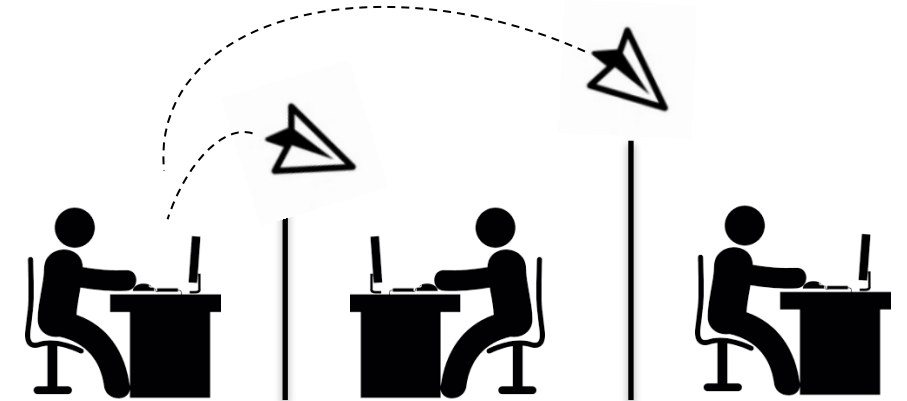
# Introduction

**MPI = Message Passing Interface**

**API de haut niveau**

Programmation parallèle

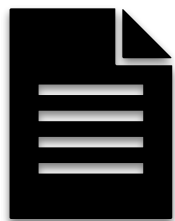
Modèle à mémoire distribuée



**Implementation comme une librairie**

→ fonctions

**Langages:** C, C++, Fortran



**Documentation:** <https://www.mpi-forum.org/docs/>

# Pourquoi utiliser MPI?

- **MPI est une interface**
- MPI est disponible sur tout type d'architecture
- MPI supporte le parallélisme intensif
- Les constructeurs de machines et/ou de réseau fournissent souvent leur propre version optimisée de librairie MPI (Intel, Cray,...)

**MPI est opensource et disponible sur les supercalculateurs actuels**

- **MPICH2:** <http://www.mcs.anl.gov/research/projects/mpich2/>
- **OpenMPI:** <http://www.open-mpi.org>



# Vue d'ensemble

## **MPI contient (MPI 1)**

- Un environnement d'exécution
- Communications point à point
- Communications collectives
- Groupes et topologie des processus MPI

## **MPI 2.0 ajoute**

- Communications dites one-sided
- Création dynamique de processus
- Multithreading
- I/O parallèle

## **MPI 3.0 ajoute**

- Communication collectives non bloquantes
- Nouvelles communications one-sided
- I/O collectives non bloquantes
- Collectives neighborhood

## **MPI 4.0 ajoute**

- Communications partitionnées
- Collectives persistantes
- Sessions MPI

# Vue d'ensemble

## MPI contient (MPI 1)

- Un environnement d'exécution
- Communications point à point
- Communications collectives
- Groupes et topologie des processus MPI

## MPI 2.0 ajoute

- Communications dites one-sided
- Création dynamique de processus
- Multithreading
- I/O parallèle

## MPI 3.0 ajoute

- Communication collectives non bloquantes
- Nouvelles communications one-sided
- I/O collectives non bloquantes
- Collectives neighborhood

## MPI 4.0 ajoute

- Communications partitionnées
- Collectives persistantes
- Sessions MPI

## Outils de vérification et de profiling

Ce qu'on verra en cours

# Programme Hello World!

```
#include <stdio.h>
/* MPI function signatures */
#include <mpi.h>

int main(int argc, char **argv)
{
    /* Initialization of MPI */
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    /* Finalization of MPI */
    MPI_Finalize();
    return 0;
}
```

# Programme Hello World!

```
#include <stdio.h>
/* MPI function signatures */
#include <mpi.h>

int main(int argc, char **argv)
{
    /* Initialization of MPI */
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    /* Finalization of MPI */
    MPI_Finalize();
    return 0;
}
```

→ Contient la signature des fonctions MPI

# Programme Hello World!

```
#include <stdio.h>
/* MPI function signatures */
#include <mpi.h>

int main(int argc, char **argv)
{
    /* Initialization of MPI */
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    /* Finalization of MPI */
    MPI_Finalize();
    return 0;
}
```

→ Contient la signature des fonctions MPI

## Syntaxe

Toutes les fonctions MPI commencent par **MPI\_**

# Programme Hello World!

```
#include <stdio.h>
/* MPI function signatures */
#include <mpi.h>

int main(int argc, char **argv)
{
    /* Initialization of MPI */
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    /* Finalization of MPI */
    MPI_Finalize();
    return 0;
}
```

→ Contient la signature des fonctions MPI

## Syntaxe

Toutes les fonctions MPI commencent par **MPI\_**

## Convention

Pas d'appel MPI avant MPI\_Init

Pas d'appel MPI après MPI\_Finalize

# Compilation

## Méthode simple

```
mpicc -o hello hello.c
```

## Qu'est ce qui se cache derrière la commande mpicc?

- Chemin où se trouvent les fichiers d'entête (ex., `mpi.h`)
- Chemin où se trouvent les librairies (ex., `libmpi.so`)
- Nom de la librairie à utiliser (linker)

`gcc -I/dir/mpi/include -o hello hello.c -L/dir/mpi/lib -lmpi`

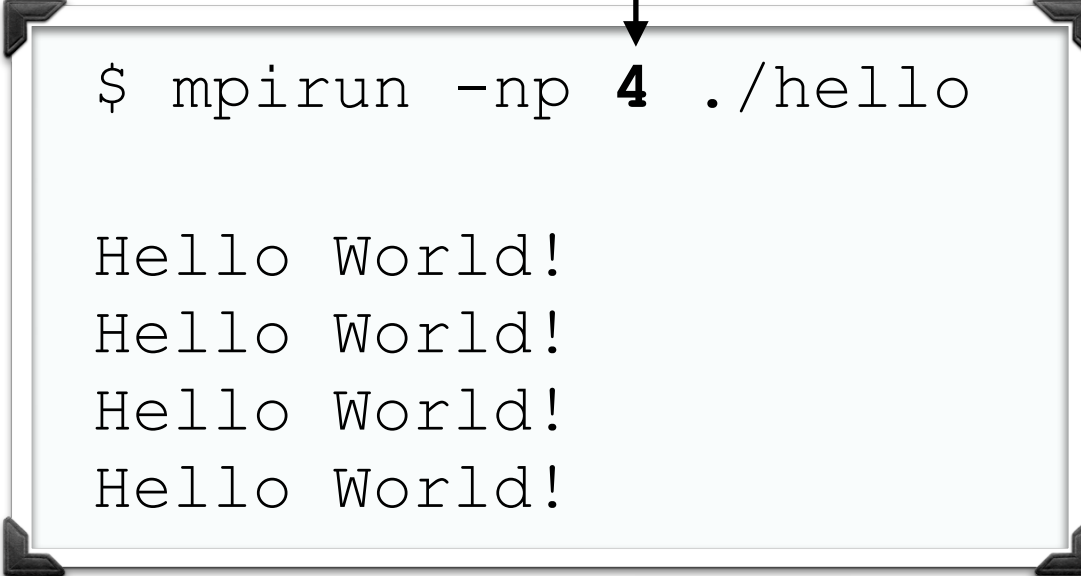
↑                      ↑                                      ↑                      ↑

Compilateur      Chemin du                      Chemin de                      Nom de

                    fichier d'entête                      la librairie                      la librairie

# Exécution

Nombre de processus



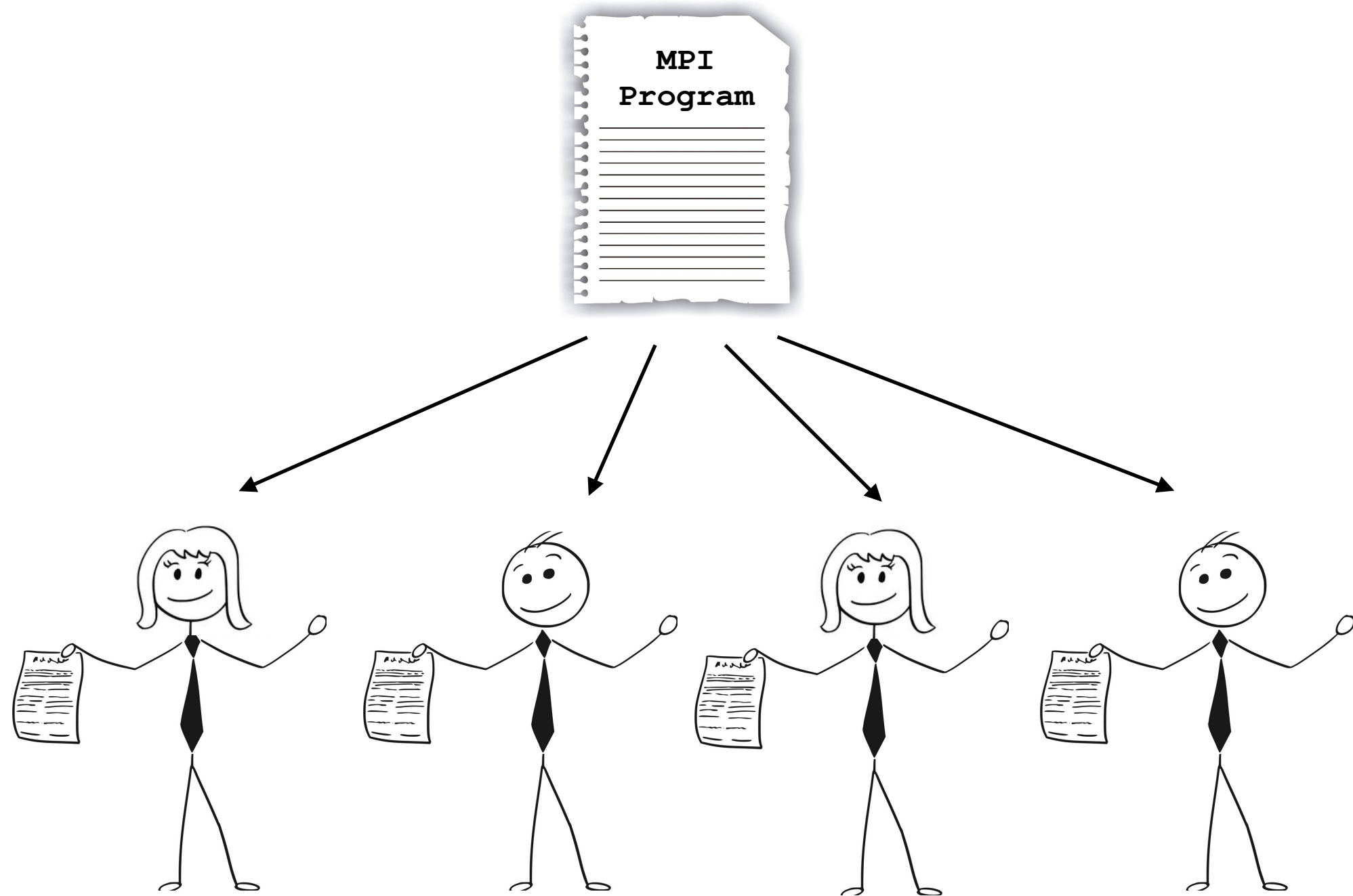
```
$ mpirun -np 4 ./hello  
  
Hello World!  
Hello World!  
Hello World!  
Hello World!
```

The diagram shows a terminal window with a light blue background and a black border. An arrow points from the text 'Nombre de processus' to the number '4' in the command 'mpirun -np 4 ./hello'. The terminal output shows four lines of 'Hello World!'.

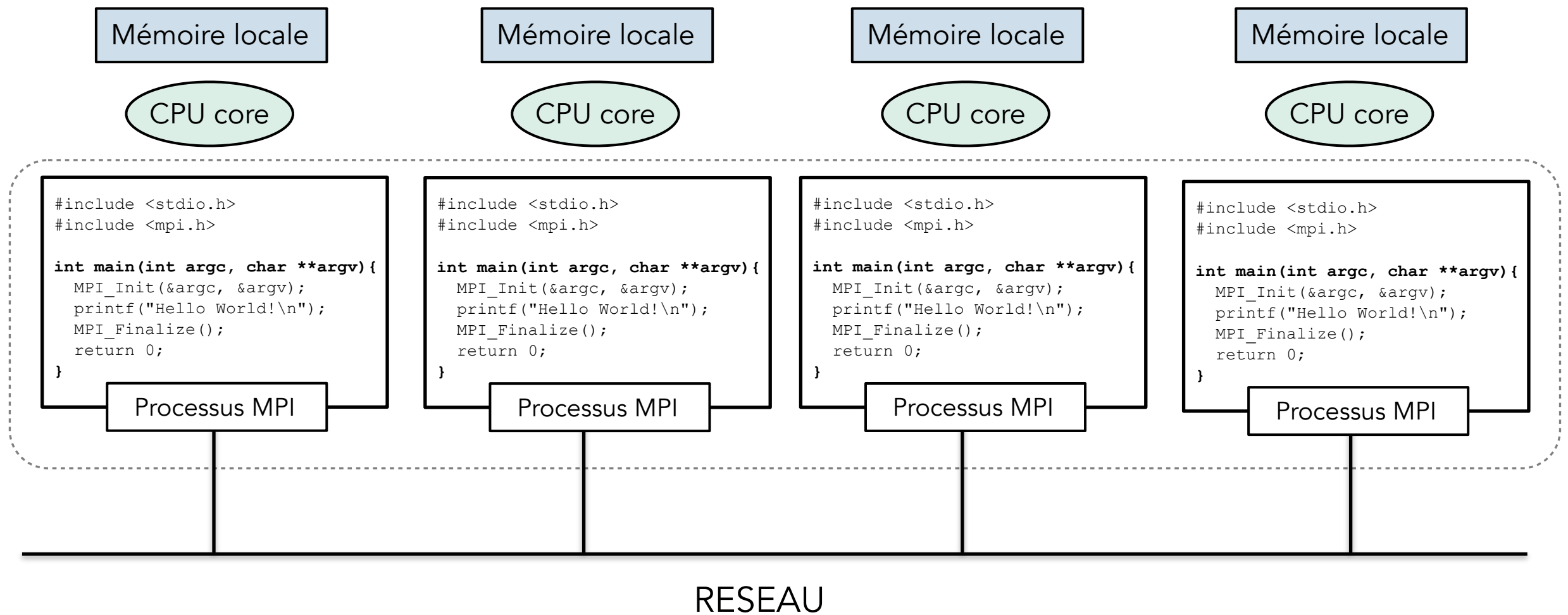
## Remarques

- ➔ Création de 4 processus
- ➔ Chaque processus a la même instruction
- ➔ Les processus sont indépendants pour l'exécution





# Communicateur

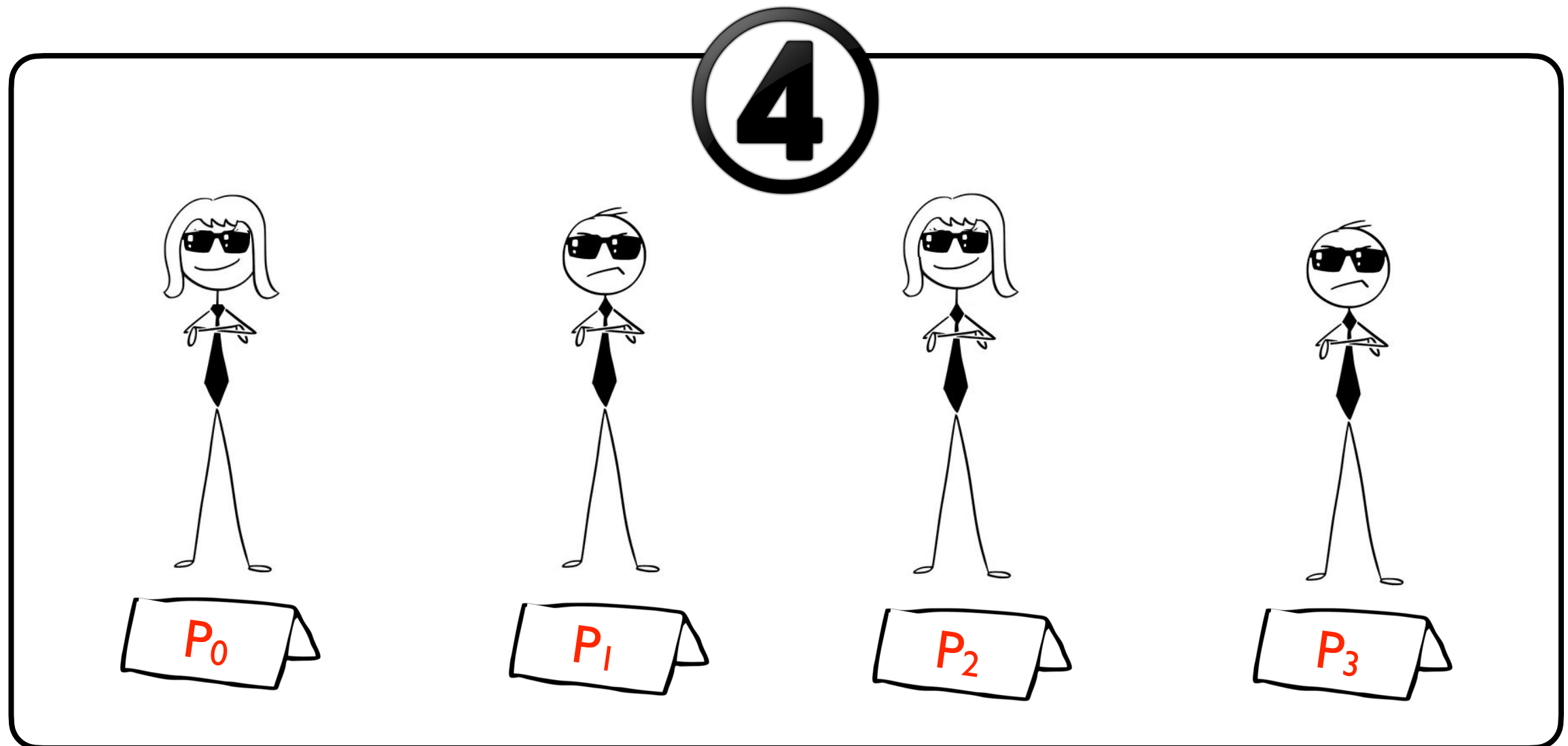


Programme MPI

```
Si je suis P0 alors
  Faire calcul A
Sinon
  Faire calcul B
```



# Un concept de base: Communicateur



**Communicateur = groupe de processus + contexte de communication**

- ➔ Prédéfini: `MPI_COMM_WORLD` avec tous les processus
- ➔ Type: `MPI_Comm`

# Nombre de processus

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int N;
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &N);
    printf("Nombre de processus = %d\n", N);

    MPI_Finalize();
    return 0;
}
```

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

```
Nombre de processus = 4
Nombre de processus = 4
Nombre de processus = 4
Nombre de processus = 4
```

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

- Retourne la taille du communicateur `comm` dans `size`
- Si `comm = MPI_COMM_WORLD`, `size` = nombre total de processus MPI dans l'application

# Rang d'un processus

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int N, me;
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &N);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    printf("Mon rang est %d sur %d\n", me, N);

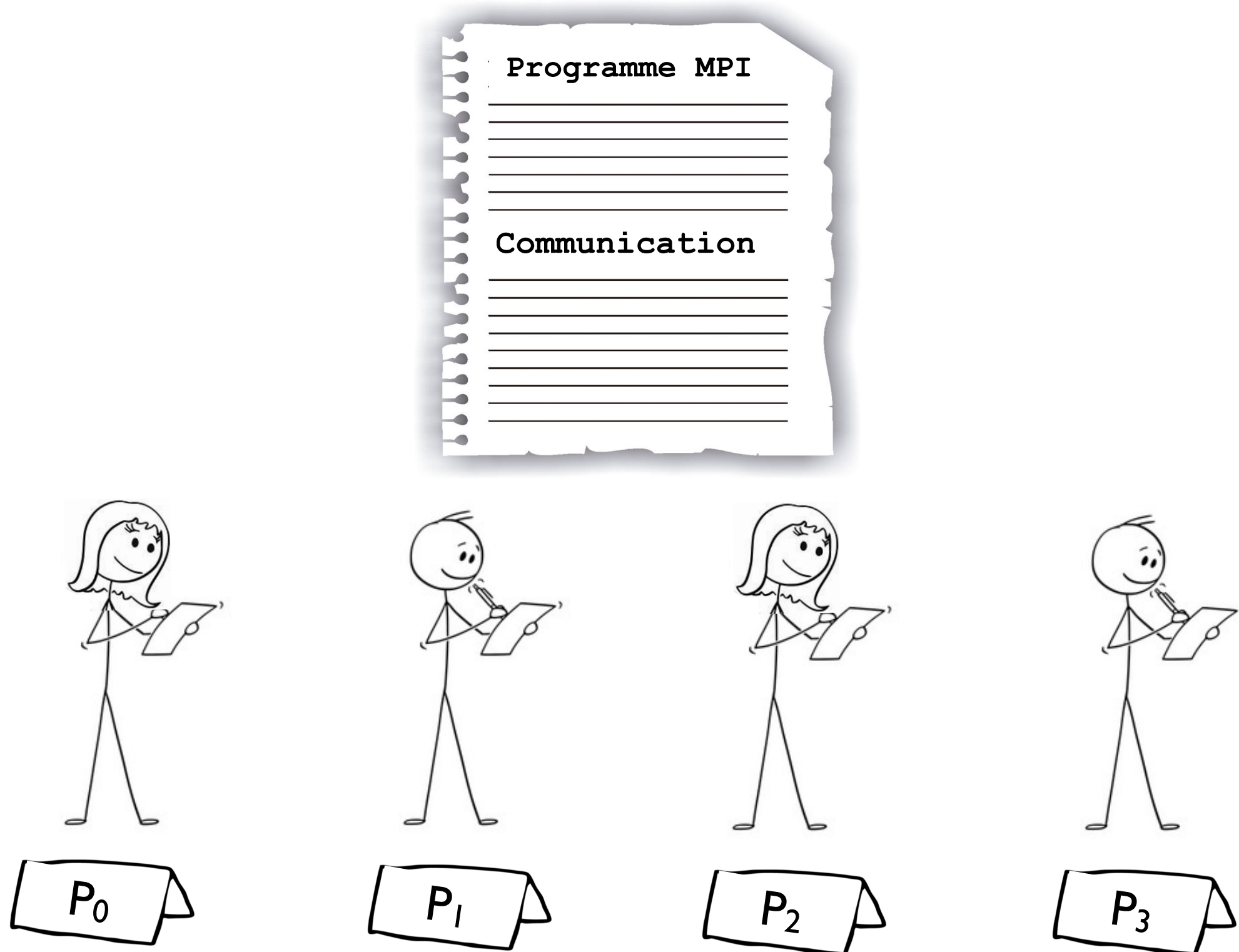
    MPI_Finalize();
    return 0;
}
```

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

```
Mon rang est 1 sur 4
Mon rang est 0 sur 4
Mon rang est 3 sur 4
Mon rang est 2 sur 4
```

**int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank);**

- Retourne le rang dans le communicateur comm dans rank
- Dans un communicateur, MPI attribue des rangs de 0 à size-1



# Message Passing Interface

## 1. Introduction à MPI

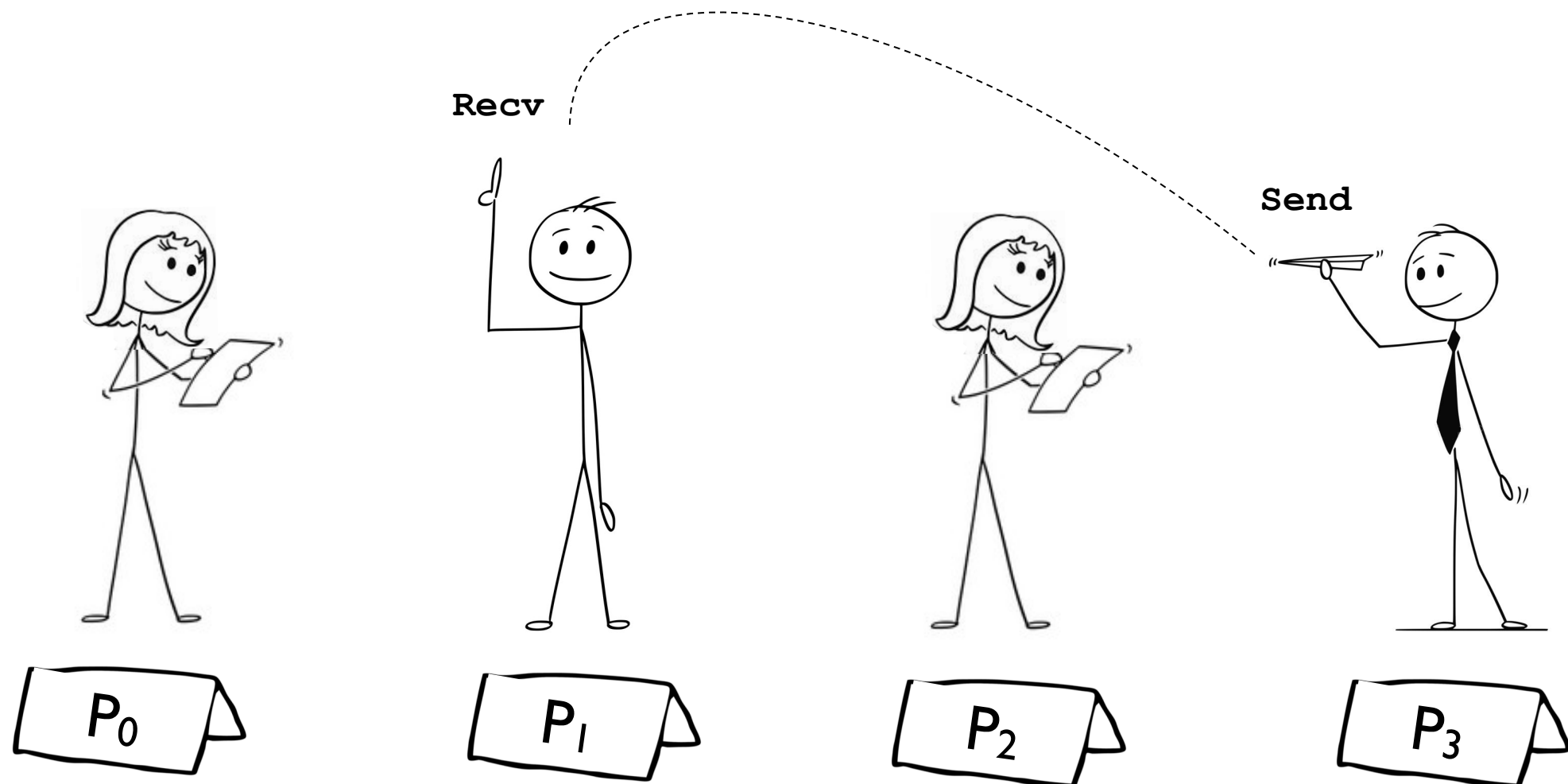
## 2. Communications point à point

1. Communications dites bloquantes
2. Communications non bloquantes
3. Communications persistantes



# Communications point à point

## Description d'un message

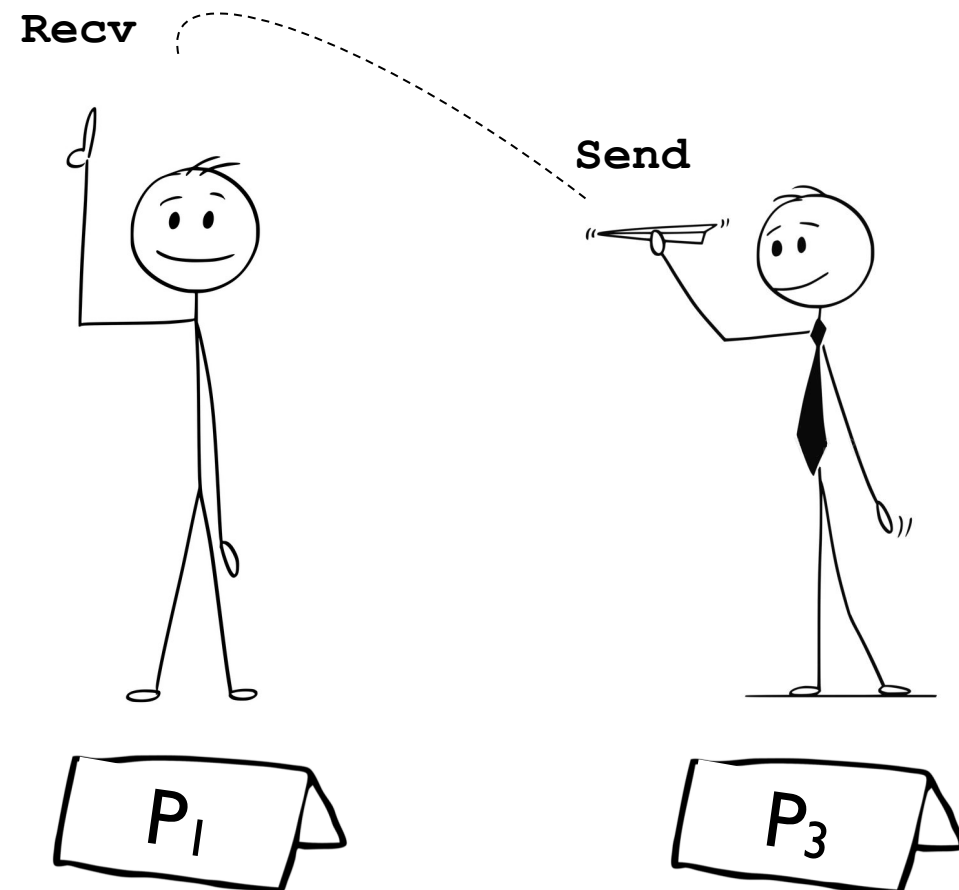


# Communications point à point

## Description d'un message



- Source = rang du processus émetteur
- Destinataire = rang du processus récepteur
- Etiquette du message (Tag)
- Communicateur



# Message Passing Interface

1. Introduction à MPI
2. Communications point à point
  1. Communications dites bloquantes
  2. Communications non bloquantes
  3. Communications persistantes

# Envoyer un message

```
int MPI_Send (
```

```
    void *buf(in),
```

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

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

```
    int dest(in),
```

```
    int tag(in),
```

```
    MPI_Comm comm(in)
```

```
);
```

} Caractéristiques du message à envoyer

# Principaux types de données

MPI_Datatype	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed 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	One byte
MPI_PACKED	Pack of non-contiguous data

# Recevoir un message

```
int MPI_Recv (
```

```
    void *buf(out),
```

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

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

```
    int source(in),
```

```
    int tag(in),
```

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

```
    MPI_Status *status(out)
```

```
);
```

Caractéristiques du message à recevoir

**NB:** Le type des données du message et le tag doivent être les mêmes côté émetteur et récepteur

# Communications 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)
```

Le matching send/recv est basé sur le tag

Le processus est bloqué dans la fonction MPI jusqu'à ce que

- **Côté réception:** les données distantes aient été copiées dans le buffer de réception
- **Côté émetteur:** le buffer d'émission puisse être modifié par l'utilisateur sans impacter le transfert du message

# Communications bloquantes

➔ Jokers pour les tags et sources: `MPI_ANY_SOURCE` et `MPI_ANY_TAG`

➔ `MPI_Status` est une structure C contenant 3 champs:

```
struct MPI_Status{  
    MPI_SOURCE  
    MPI_TAG  
    MPI_ERROR  
}
```

➔ `MPI_SUCCESS` permet de tester le code retour d'une fonction MPI

➔ Pour avoir la taille exacte du message, on interroge la variable status

`MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`

➔ Si on ne veut pas d'information sur le status, on utilise dans `MPI_Recv`  
`MPI_STATUS_IGNORE`



# Exemple 1

```
int main(int argc, char **argv)
{
    double p = 0., s0;
    int i, r;
    MPI_Status status;

    MPI_Init(&argc, &argv); /* Initialisation de la librairie MPI*/
    MPI_Comm_rank(MPI_COMM_WORLD, &r); /* rang */

    for( i = 0 ; i < N/2 ; i++ )
        p += tab[i];

    tag = 1000; /* Tag du message */

    if (r == 0) {
        MPI_Send(&p, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
    } else if (r == 1){
        MPI_Recv(&s0, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
        printf( "Sum = %d\n", s0+p );
    }
    MPI_Finalize();
    return 0;
}
```

## Exemple 2

```
sum = 0.;
for( i = 0 ; i < N/P ; i++ )
    sum += tab[i];

if (r == 0) {
    /* Le processus 0 reçoit P-1 messages dans n'importe quel ordre */
    for( t = 1 ; t < P ; t++ ) {

        MPI_Recv(&s, 1, MPI_DOUBLE,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, /* wildcards */
                 MPI_COMM_WORLD, &sta);

        printf("Message from rank %d\n", sta.MPI_SOURCE);

        sum += s; /* Contribution du processus sta.MPI_SOURCE à la somme globale */
    }
} else {
    /* Les autres processus envoient leur somme partielle au rang 0 */
    MPI_Send(&sum, 1, MPI_DOUBLE, 0, r, MPI_COMM_WORLD);
}
```

## Exemple 2

```
/* Code récepteur (rang=0) */
```

```
...
```

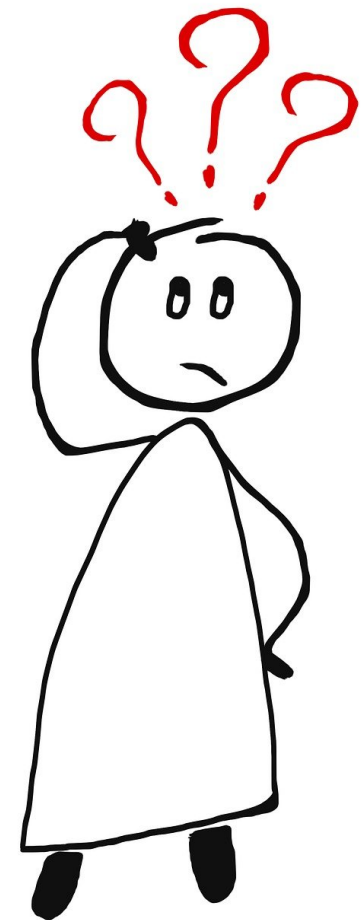
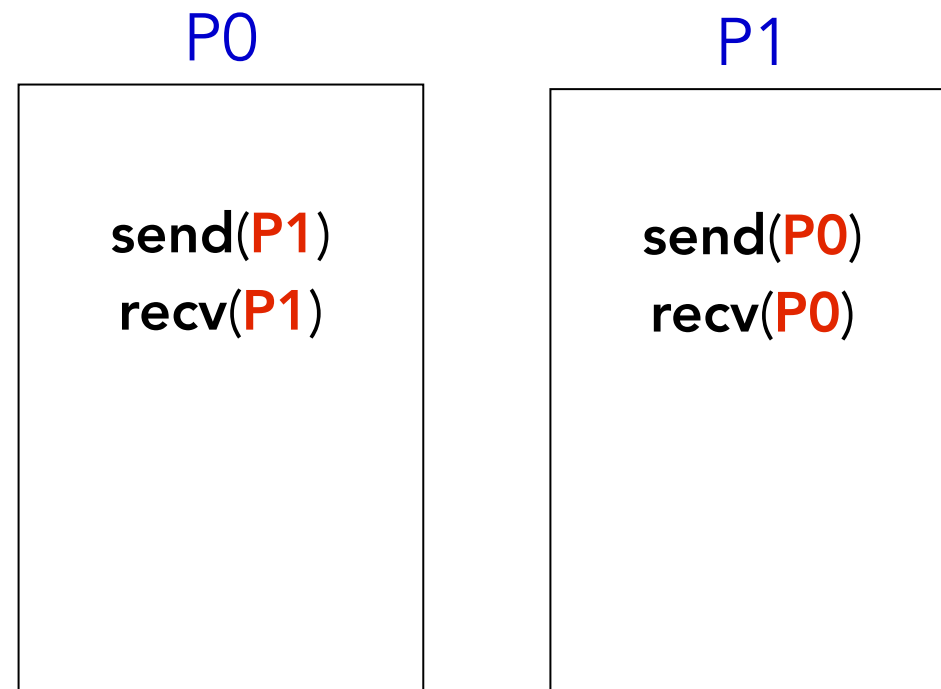
```
MPI_Recv(&s, 1, MPI_DOUBLE,  
        MPI_ANY_SOURCE, MPI_ANY_TAG,  
        MPI_COMM_WORLD, &sta);
```

```
printf("status:\n MPI_SOURCE:%d\n MPI_TAG:%d\n MPI_ERROR: %d\n",  
       sta.MPI_SOURCE, sta.MPI_TAG, sta.MPI_ERROR);
```

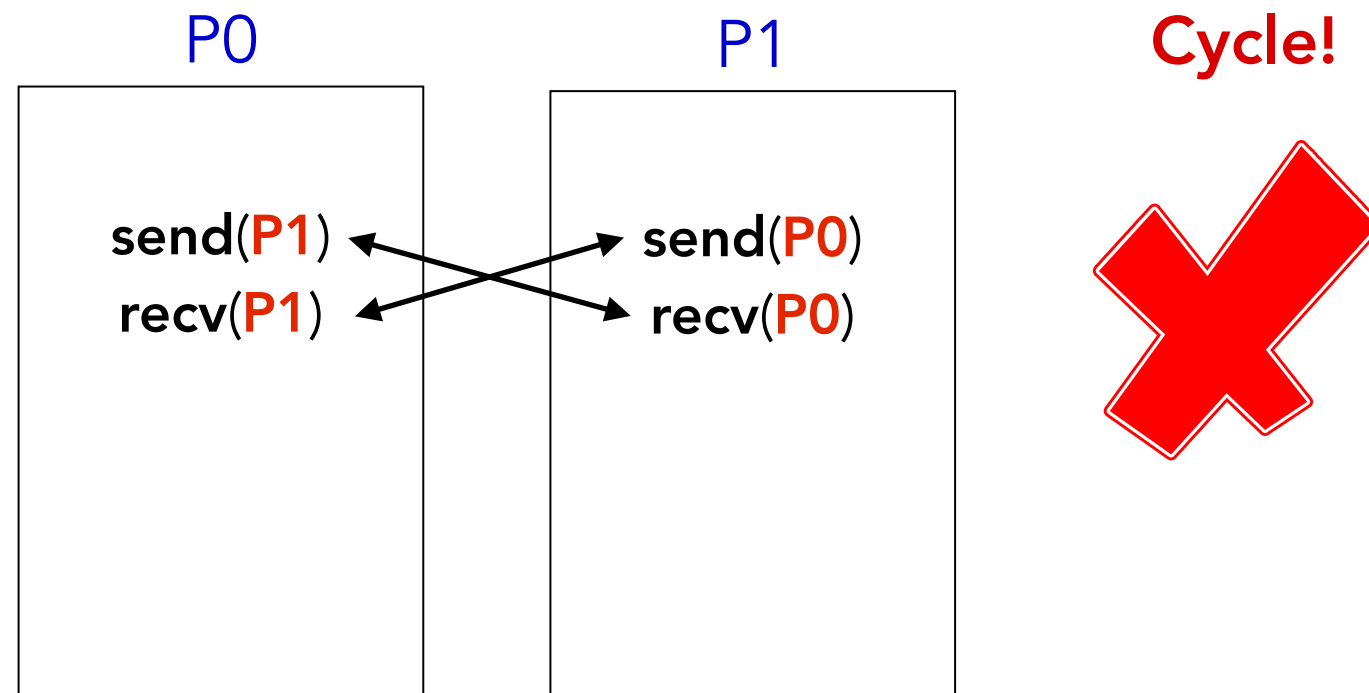
```
MPI_Get_count(&sta, MPI_DOUBLE, &count);
```

```
printf("Message size: %d\n", count);
```

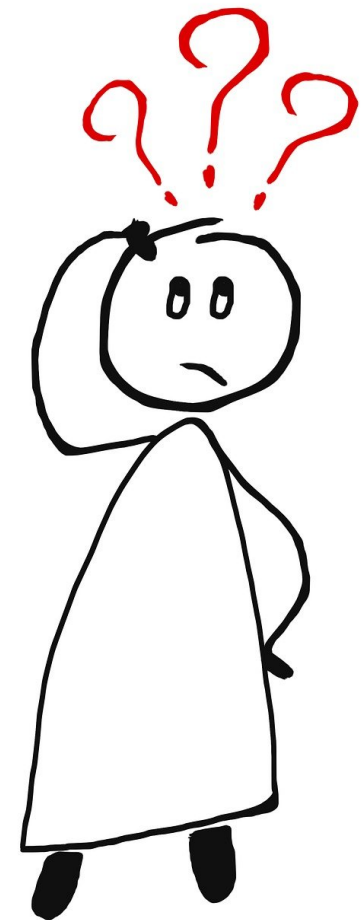
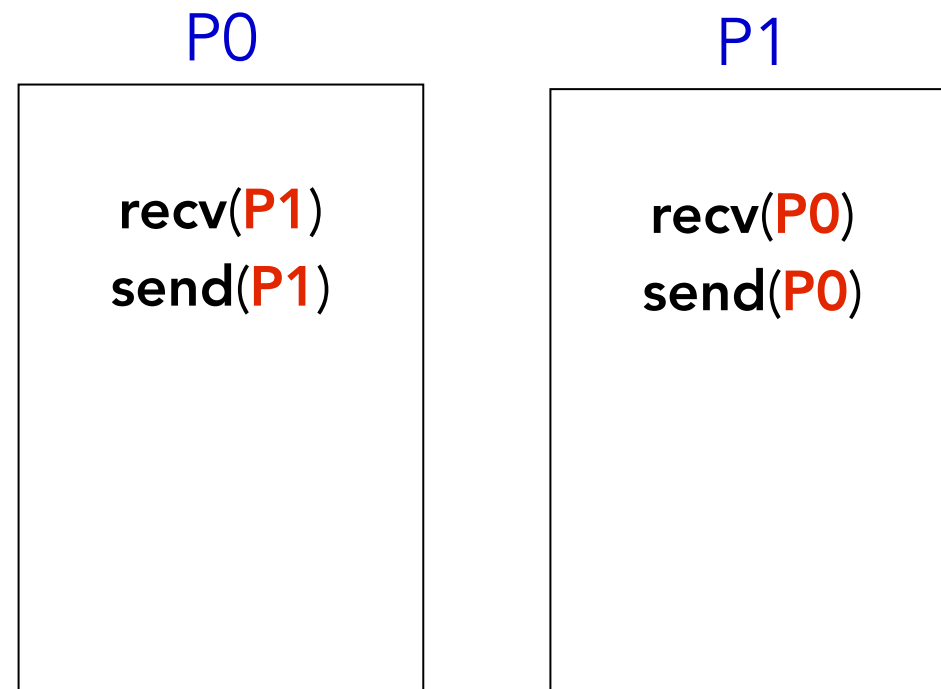
# Ordre des messages



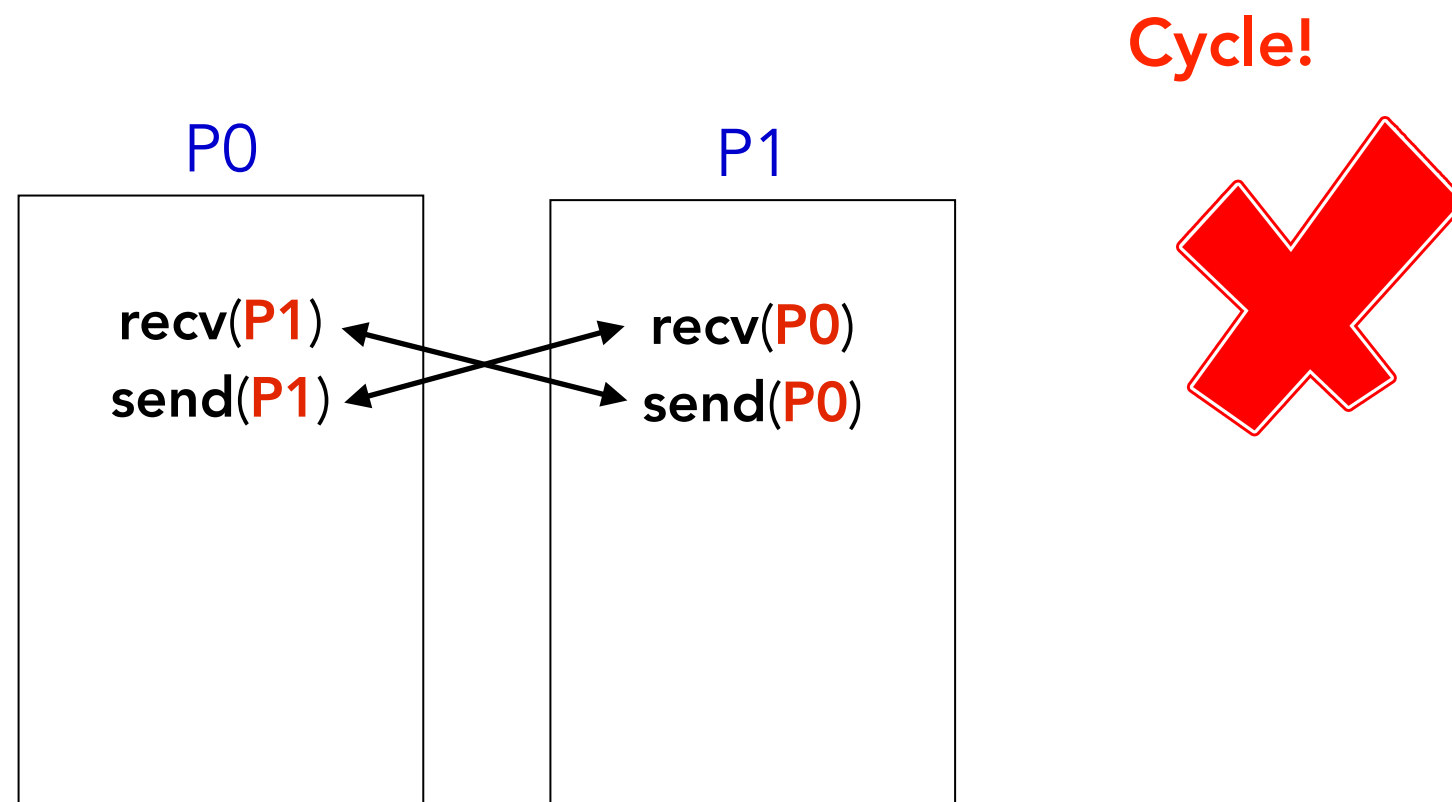
# Ordre des messages



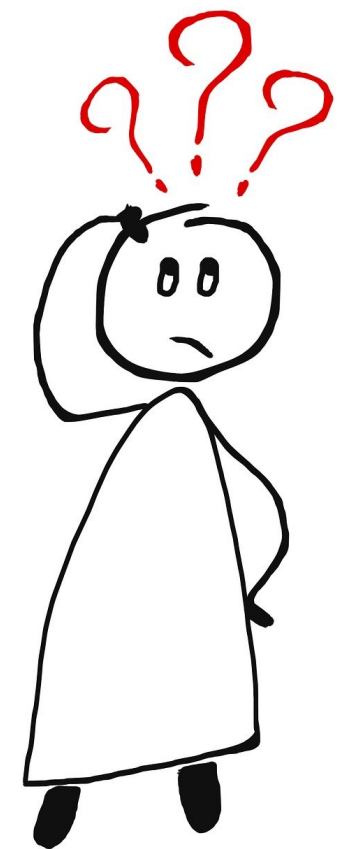
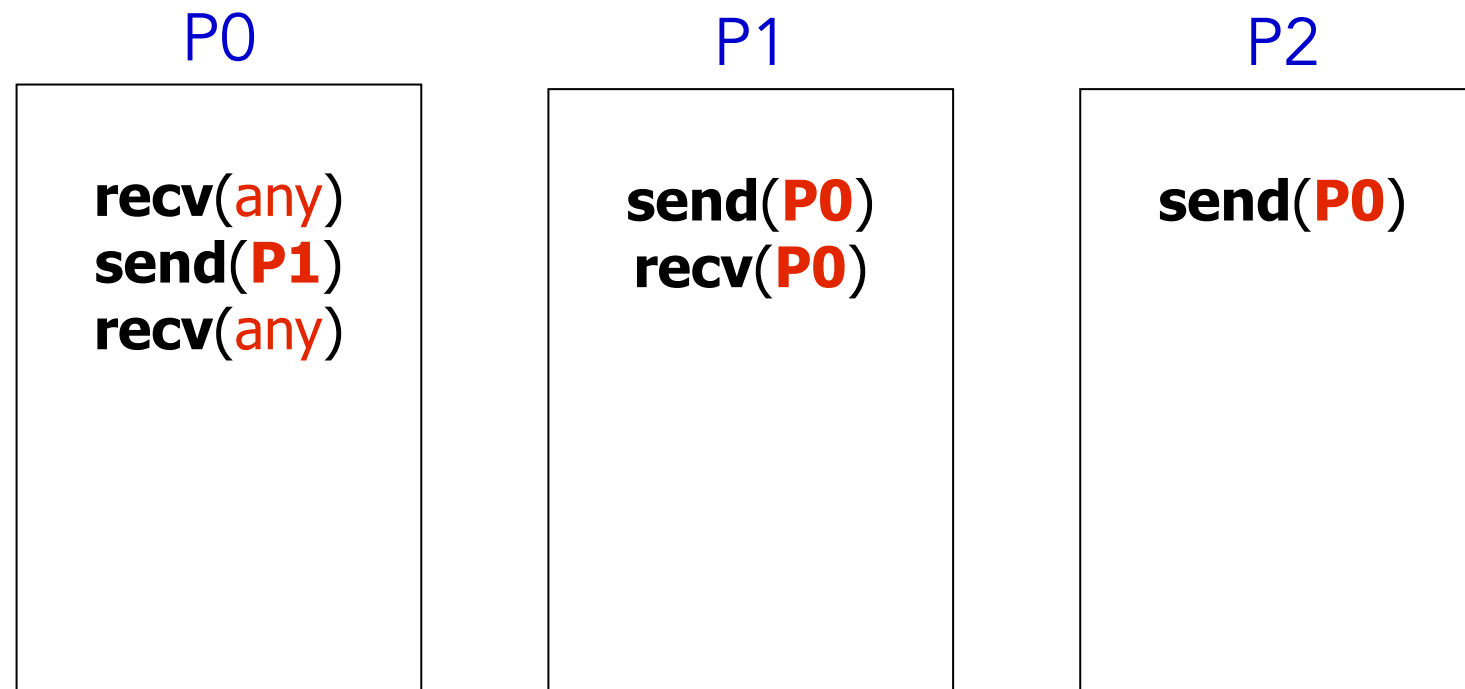
# Ordre des messages



# Ordre des messages

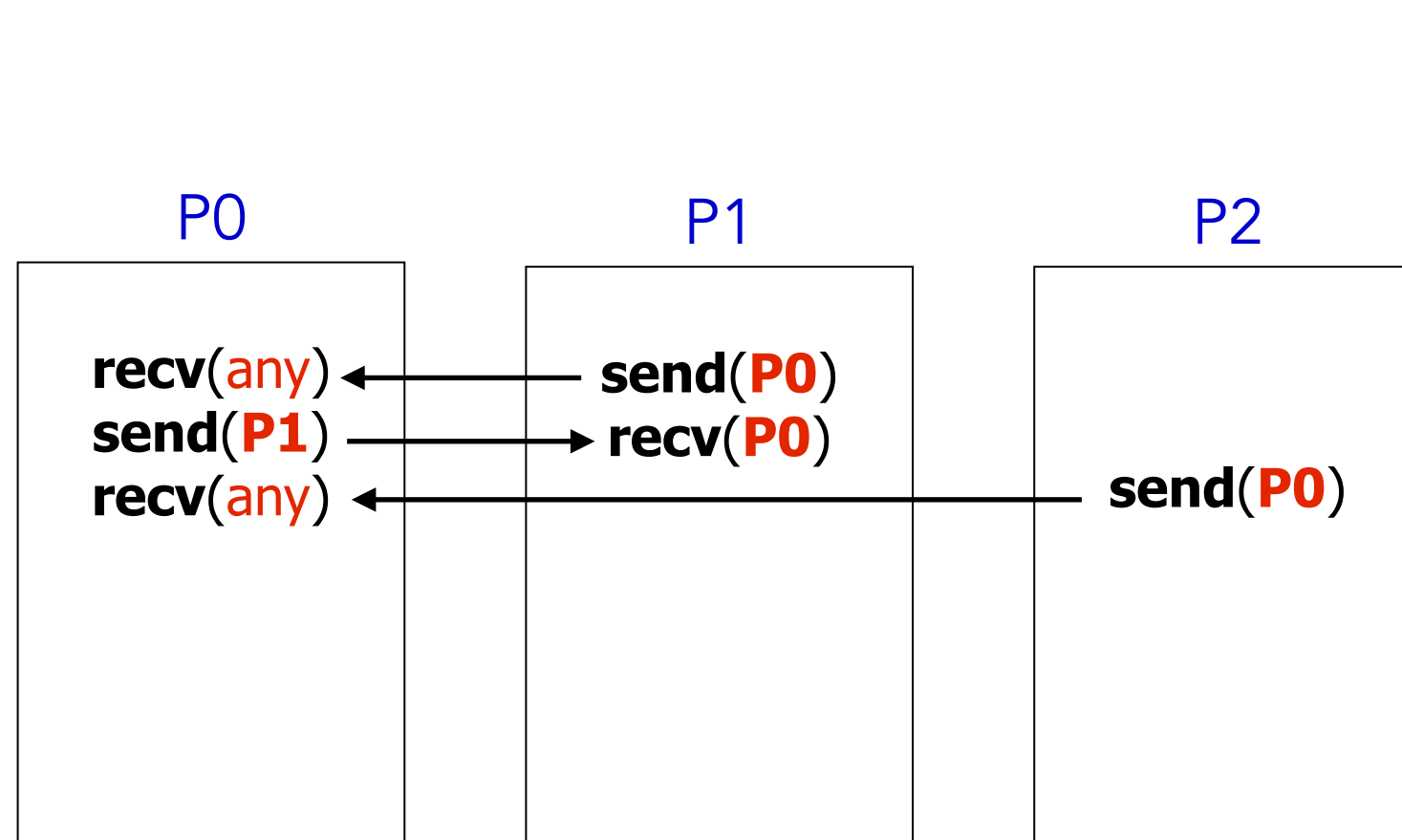


# Ordre des messages

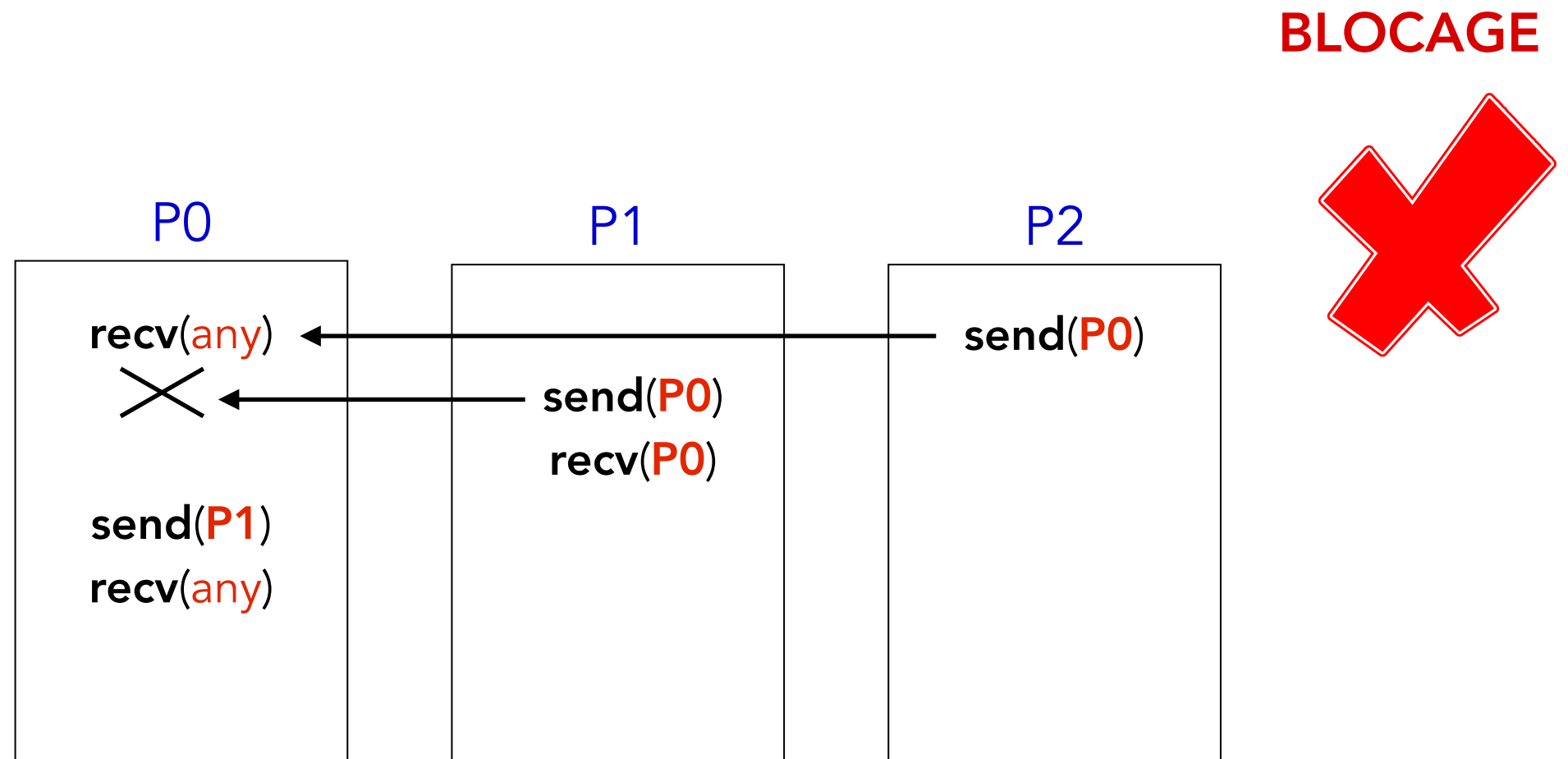




# Ordre des messages



# Ordre des messages



# Protocoles de communication

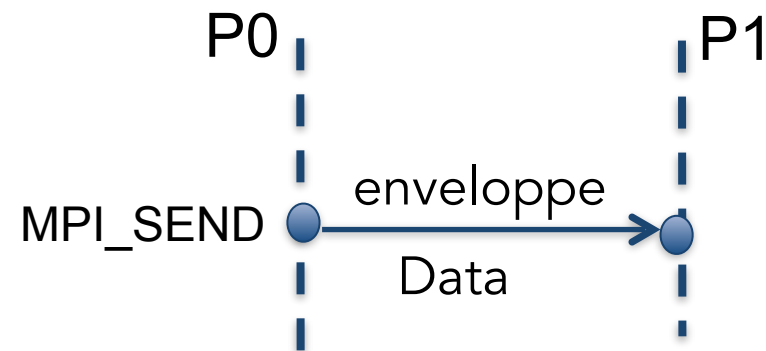
## Deux protocoles

- Eager pour des petits messages
- Rendez-vous

# Protocoles de communication

## Deux protocoles

- Eager pour des petits messages
- Rendez-vous

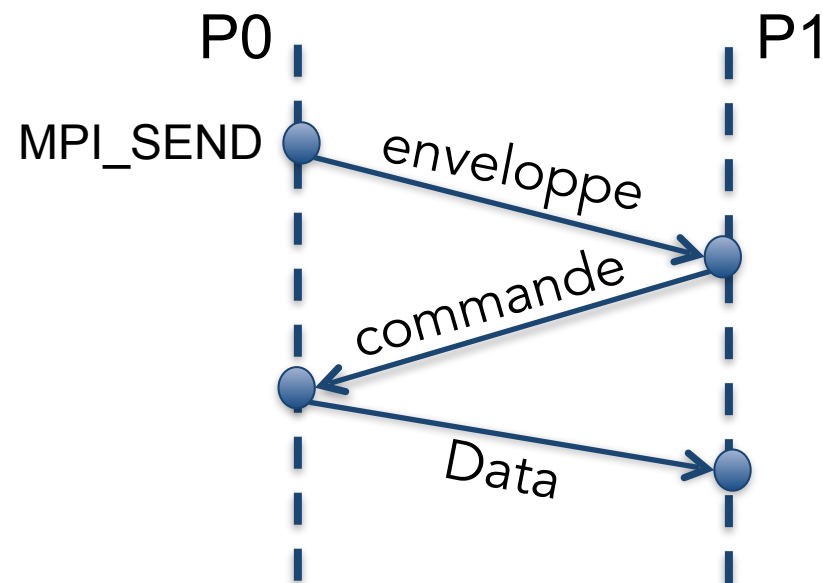


**Envoi direct**

# Protocoles de communication

## Deux protocoles

- Eager pour des petits messages
- **Rendez-vous**



1. Notification d'un message  
P1 prépare la réception
2. P1 prévient qu'il est prêt
3. P0 envoie les données

# Définitions

Une **opération** est un ensemble de procédures qui se découpent en 4 étapes: **initialisation** (*initialization*), **commencement** (*starting*), **completion** (*completion*) et **libération** (*freeing*).

- L'**initialisation** consiste à initier une opération: elle transmet les arguments de l'opération sans le contenu du message
- Le **commencement** transmet le contenu du message
- La **completion** rend la main sur le message et indique que le buffer de sortie a été mis à jour
- La **libération** rend la main sur la liste des arguments

# Définitions

Une procédure est **bloquante** si à la fin de l'appel, le programme peut réutiliser les ressources utilisées dans l'appel.

Pour une opération bloquante, les 4 étapes sont combinées en une seule procédure.

➔ Un envoi bloquant peut être :

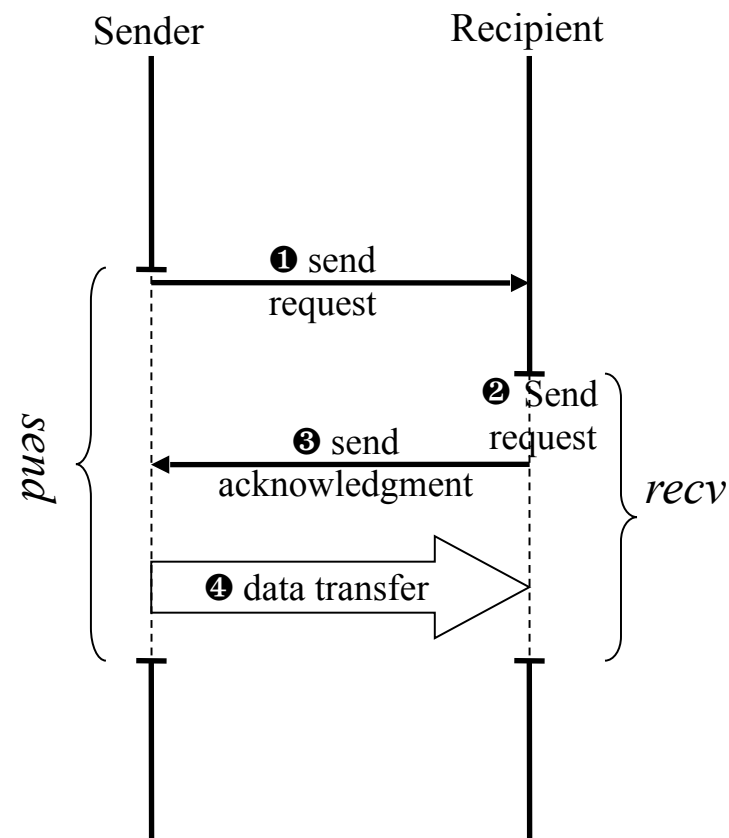
- **synchrone**: La communication est établie et le message est effectivement arrivé à destination (poignée de main)
- **asynchrone**: Si un buffer est utilisé pour stocker les données avant l'envoi

➔ Une réception bloquante retourne uniquement si les données sont reçues et prêtes à l'emploi.

# Modes de communication

## 3 modes de communication pour l'émission :

**Synchronous** : synchronise les processus d'envoi et de réception. L'envoi du message est terminé si la réception est postée et la lecture terminée.



❶ L'émetteur transfère une requête au récepteur et attend une réponse

❷ Lorsque le récepteur commence une fonction `recv`, il attend une requête de l'émetteur

❸ Quand le récepteur a la requête, il répond à l'émetteur

❹ L'émetteur et le récepteur sont maintenant synchronisés et le transfert de données peut avoir lieu

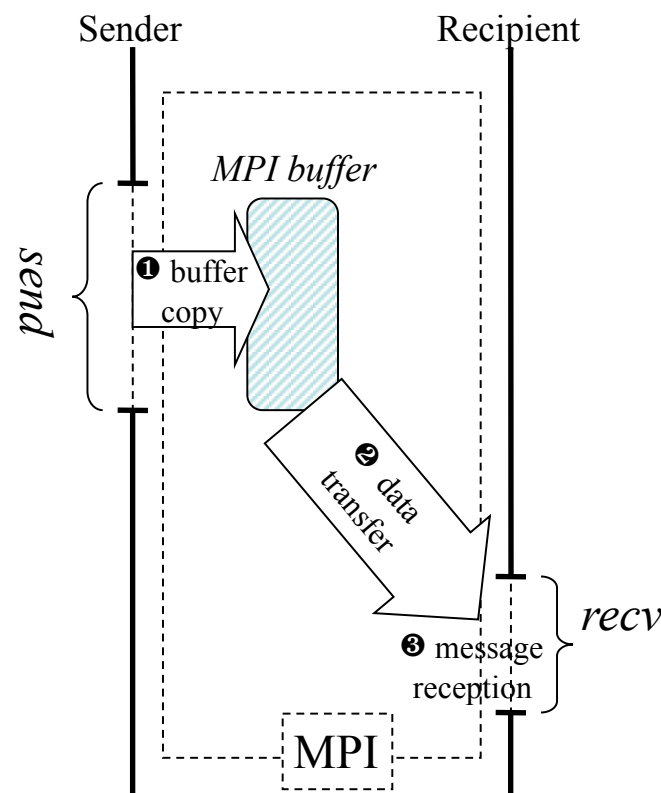


# Modes de communication

## 3 modes de communication pour l'émission :

**Synchronous** : synchronise les processus d'envoi et de réception. L'envoi du message est terminé si la réception est postée et la lecture terminée.

**Buffered** : Il est à la charge de l'utilisateur d'effectuer une recopie temporaire du message. L'envoi se termine lorsque la recopie est achevée. L'envoi est découplé de la réception.



❶ L'émetteur copie le message dans un buffer (géré par la librairie de communication). La fonction Send peut retourner

❷ La librairie de communication a une copie des données et l'envoie au récepteur

❸ L'émetteur obtient le message dès que possible

# Modes de communication

## 3 modes de communication pour l'émission :

**Synchronous** : synchronise les processus d'envoi et de réception. L'envoi du message est terminé si la réception est postée et la lecture terminée.

**Buffered** : Il est à la charge de l'utilisateur d'effectuer une copie temporaire du message. L'envoi se termine lorsque la copie est achevée. L'envoi est découplé de la réception.

**Standard** : communication bloquante ou non bloquante. Retour au programme après terminaison. Bufférisé ou non au choix de l'implémentation.

MPI définit un seuil  $T$

- ➔ Si la taille du message est plus petite que  $T$  -> mode bufferisé
- ➔ Si la taille du message est plus grande que  $T$  -> mode synchronous

# Modes de communication

## 3 modes de communication pour l'émission :

**Synchronous** : synchronise les processus d'envoi et de réception. L'envoi du message est terminé si la réception est postée et la lecture terminée.

**Buffered** : Il est à la charge de l'utilisateur d'effectuer une copie temporaire du message. L'envoi se termine lorsque la copie est achevée. L'envoi est découplé de la réception.

**Standard** : communication bloquante ou non bloquante. Retour au programme après terminaison. Bufférisé ou non au choix de l'implémentation.

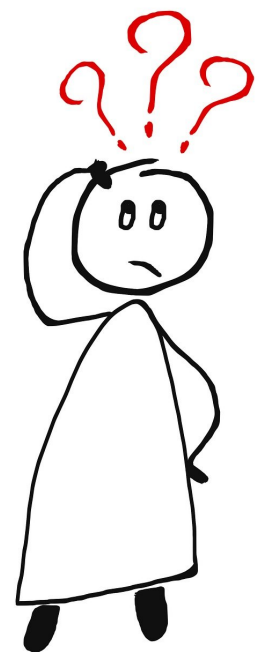
Standard	Buffered	Synchronous
MPI_Send	MPI_Bsend	MPI_Ssend

# Modes de communication

```
if ( rang == 0 )
    voisin = 1;
else if ( rang == 1 )
    voisin = 0;

MPI_Send(&msg1, N, MPI_BYTE, voisin, tag1, comm);
MPI_Recv(&msg2, N, MPI_BYTE, voisin, tag2, comm);
```

**Est-ce que le code est correct?**



# Modes de communication

```
if ( rang == 0 )
    voisin = 1;
else if ( rang == 1 )
    voisin = 0;

MPI_Send(&msg1, N, MPI_BYTE, voisin, tag1, comm);
MPI_Recv(&msg2, N, MPI_BYTE, voisin, tag2, comm);
```

**Est-ce que le code est correct?**

**NON**

- Si N est suffisamment petit → OK
- Si N est trop grand → Blocage

# Send/Recv bidirectionnel

```
int MPI_Sendrecv (  
    void *senbuf(in), int sendcount(in), MPI_Datatype sendtype(in),  
    int dest(in), int sendtag(in),  
    void *recvbuf(in), int recvcount(in), MPI_Datatype recvtype(in),  
    int source(in), int recvtag(in),  
    MPI_Comm comm(in)  
);
```

Opération qui combine un send et un receive bloquants

- ➔ Peut être associé à des appels send/recv bloquants
- ➔ Peut être associé à d'autres appels Sendrecv vers d'autres destinataires

**A vous de jouer!**