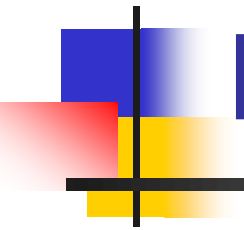
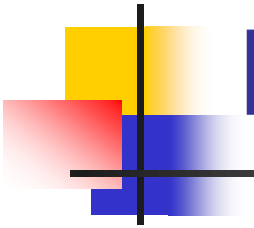


Programmation Parallèle et Distribuée

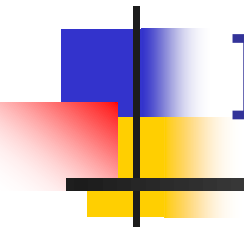
Cours 2 : Introduction à MPI & communications point à point

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

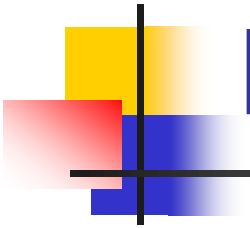


Plan du cours 2

- Introduction à MPI
  - Premier exemple
  - Compilation & exécution
  - Fonctionnalités basiques
- Communications point-à-point
  - Principe de l'échange de messages
  - Fonctionnalités MPI
  - Notion de communication bloquante



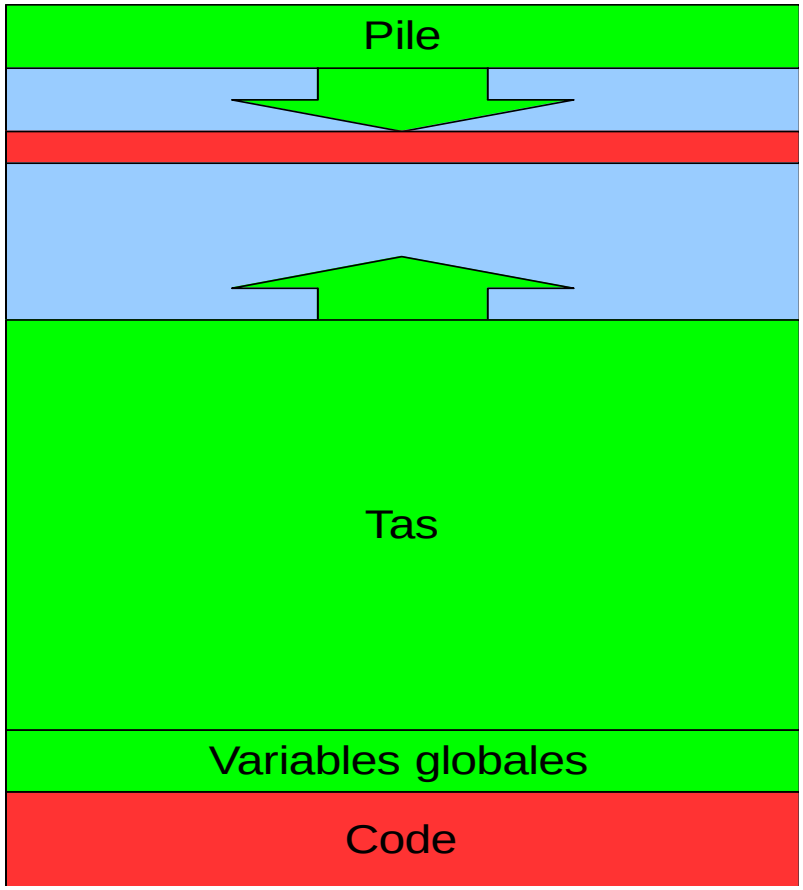
Introduction à MPI



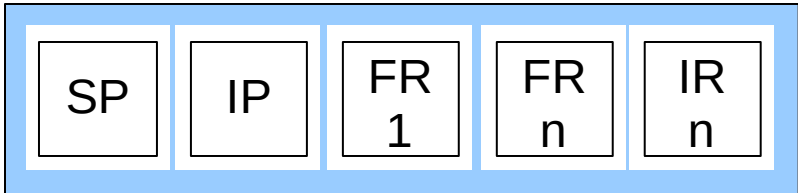
Définitions

- MPI : *Message-Passing Interface*
- API de haut niveau
  - Programmation parallèle
  - Paradigme de l'échange de messages
- Implémentation dans une bibliothèque
  - Utilisation par appels de fonctions
- Interfaces disponibles
  - C
  - C++
  - Fortran

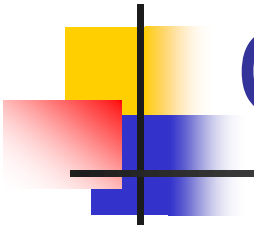
Processus



Mémoire

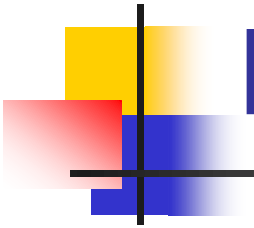


Structures



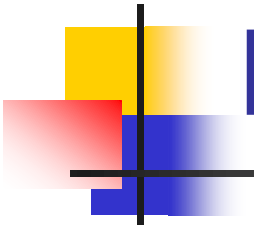
Contexte

- MPI est adapté à la programmation parallèle distribuée
- MPI naît de la collaboration d'universitaires et d'entreprises
  - On étudiera uniquement MPI 1.3
  - Il existe également une norme MPI 2
  - Il existe également une norme MPI3
- Très répandu dans le monde du calcul intensif
  - Permet d'utiliser plusieurs centaines (voire milliers) de processeurs pour exécuter des tâches parallèles
  - Chaque constructeur implémente MPI de façon optimisée sur leurs machines



Fonctionnalités MPI

- La bibliothèque MPI permet de gérer
  - l'environnement d'exécution
  - les communications point à point
  - les communications collectives
  - les groupes de processus
  - les topologies de processus
- MPI 2.0 permet en outre de gérer
  - les communications unidirectionnelles
  - la création dynamique de processus
  - le multithreading
  - les entrées/sorties parallèles (MPI/IO)
- Beaucoup de fonctionnalités (environ 120 fonctions pour MPI 1, plus de 200 pour MPI 2)
  - Néanmoins, la maîtrise des communications point à point et collectives est suffisante pour paralléliser la plupart des applications de façon efficace



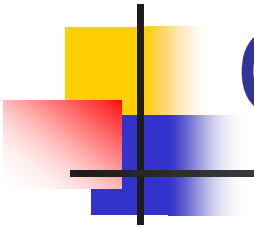
Pourquoi utiliser MPI ?

- MPI est avant tout une interface
- MPI est présent sur tout type d'architecture parallèle
- MPI supporte les parallélismes modérés et massifs
- Les constructeurs de machines et/ou de réseaux rapides fournissent des bibliothèques MPI optimisées pour leurs plateformes
- MPI est également disponible pour la plupart des machines du marché en *open source*
  - MPICH2 : http://www.mcs.anl.gov/research/projects/mpich2/
  - OpenMPI : http://www.open-mpi.org

Premier programme MPI

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

int main(int argc, char **argv) {

  /* initialisation de la bibliothèque MPI
     */
  MPI_Init(&argc, &argv);

  printf("Hello !\n");

  /* aucune fonction MPI après cet
     appel */
  MPI_Finalize();
  return 0;
}
```

- Inclusion d'un fichier
  - Notion de bibliothèque
- Tous les appels à la bibliothèque MPI commencent par le préfixe `MPI_`
- Aucun appel à MPI ne peut avoir lieu avant l'appel à `MPI_Init()`
- aucun appel à MPI ne peut avoir lieu après l'appel à `MPI_Finalize()`



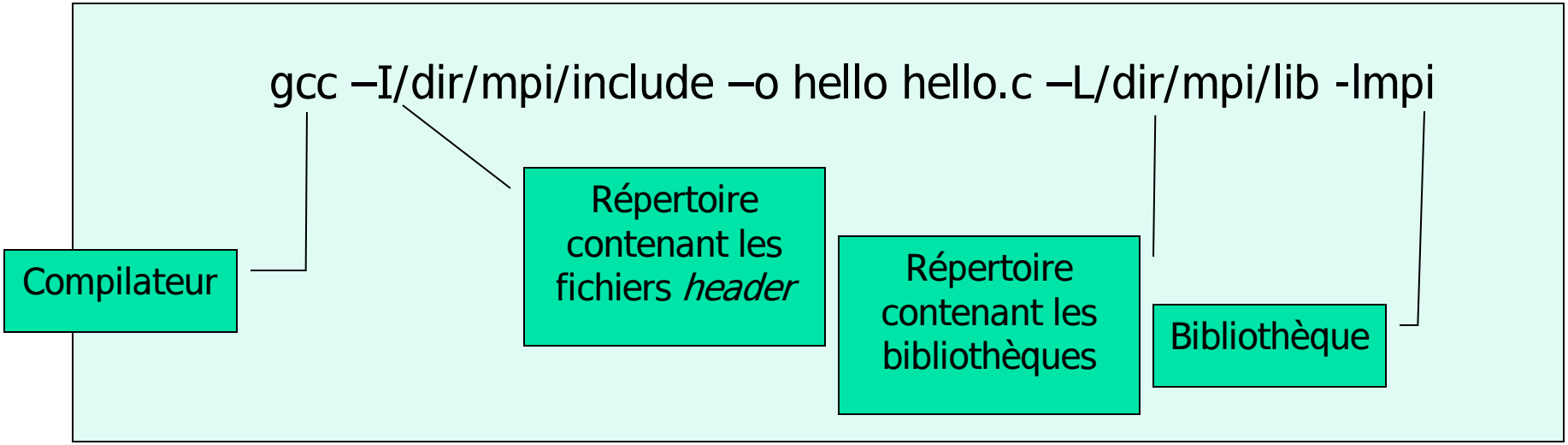
Compilation

- Processus de compilation comme n'importe quel bibliothèque !
- Mais : façon de compiler dépend de l'implémentation
  - Cas simple : utilisation d'un script `mpicc`
  - Cas complexe : compilation en mettant les chemins vers la bibliothèque *à la main*
- Cas simple
  - Un script/programme encapsule les options nécessaires

    ```
    mpicc -o hello hello.c
    ```
  - Les options sont passés au compilateur par défaut
    - Possibilité de changer ce compilateur

Compilation

- Cas complexe :
  - En l'absence de script → ajout des options à la main
- Options nécessaires pour utiliser une bibliothèque externe
  - Répertoires pour trouver les *headers*
  - Répertoires où trouver *physiquement* les bibliothèque
  - Nom de la bibliothèque à utiliser (*linker)*
- Exemple : la bibliothèque `libc`



Exécution

- Utilisation du script `mpirun`
- Syntaxe (options, format, ...) dépend de l'implémentation
  - Spécification obligatoire du nombre de processus

mpirun –np 4 ./hello

Hello !

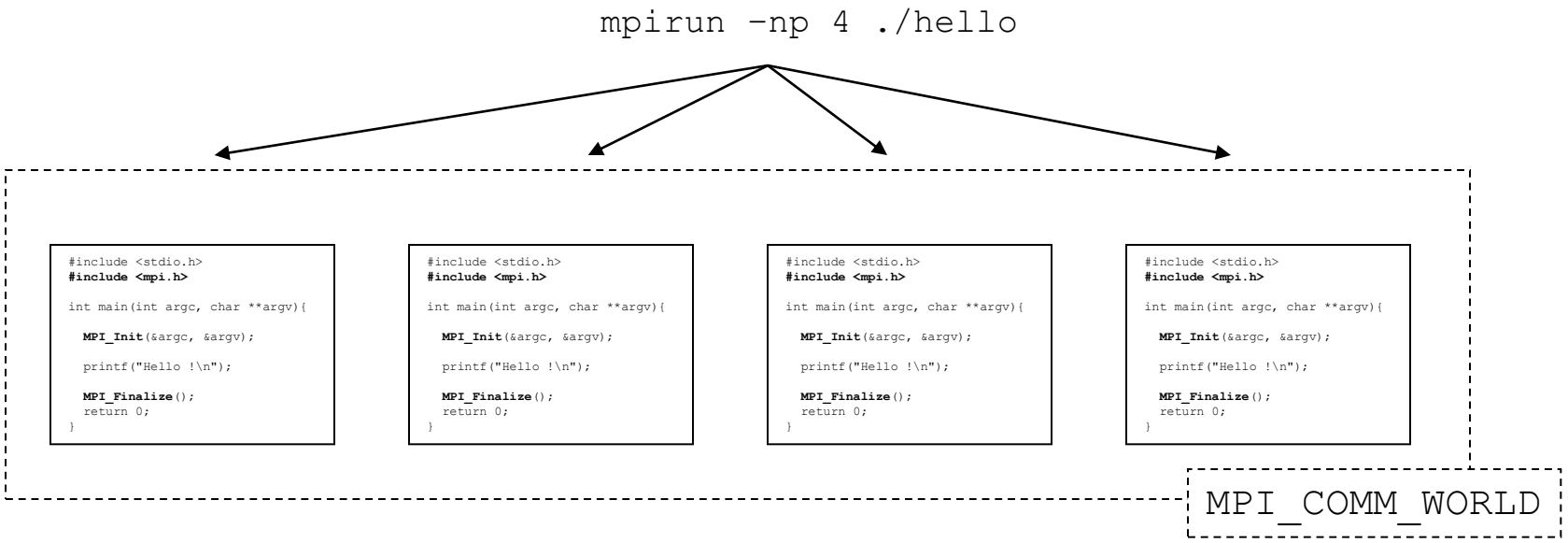
Hello !

Hello !

Hello !

- Remarques
  - Création de 4 processus
  - Chacun se lance indépendamment des autres

Communicateurs



- Regroupement des processus créés sous le communicateur prédéfinit MPI_COMM_WORLD
- Communicateur = ensemble de processus + contexte de communications
- Communicateur est de type MPI_Comm
- Remarque : dans ce cours, nous n'utiliserons que MPI_COMM_WORLD

Nombre total de processus

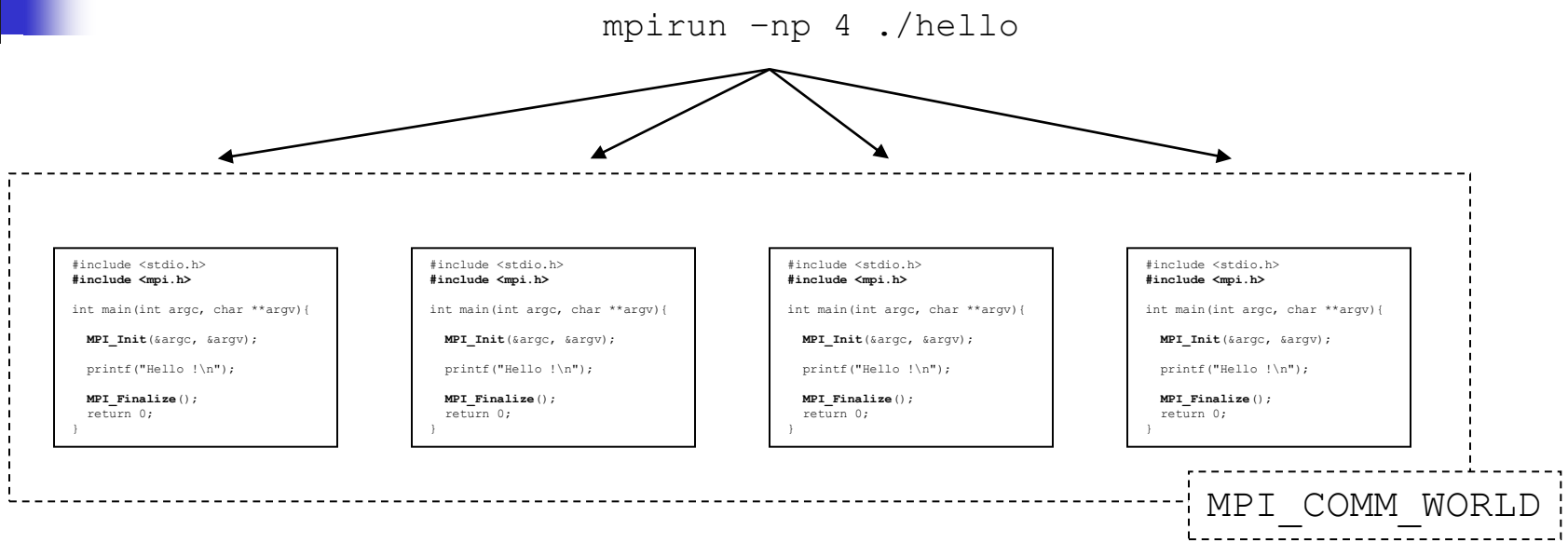
```
#include <stdio.h>
#include <mpi.h>

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);`

- MPI_Comm_size retourne dans `*size` la taille du communicateur `comm`
- Dans le cas où `comm == MPI_COMM_WORLD`, `MPI_Comm_size` retourne le nombre total de processus pouvant communiquer

Rang d'un processus



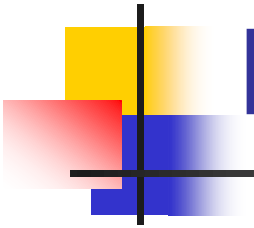
- Pour un communicateur donné, MPI associe à chaque processus un numéro compris entre 0 et N-1 (N étant la taille du communicateur)
- Le numéro unique associé au processus s'appelle le rang du processus ;
- La fonction `MPI_Comm_rank` retourne le rang du processus `*rank` dans le communicateur `comm` :

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

Rang d'un processus

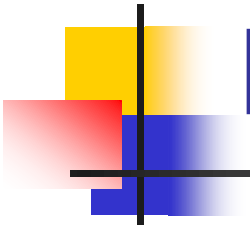
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {

  int N, moi;
  MPI_Init(&argc, &argv);

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

  printf("Mon rang est %d sur %d\n", moi, 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
%
```



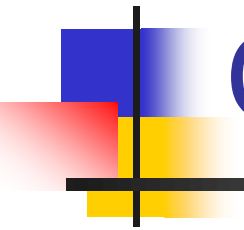
Rang d’un processus

- Le nombre de processus n'est pas forcement égal aux nombre de processeurs disponibles
- L'ordre d'exécution des processus n'est pas déterminé par leurs rangs : la notion d'ordre n'a plus de sens car l'exécution est parallèle, simultanée
- La notion de rang est fondamentale pour le modèle SPMD : c'est grâce à son propre rang que chaque processus va déterminer les données sur lesquelles il va travailler.

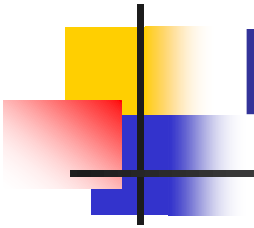


Résumé

- `MPI_Init()` et `MPI_Finalize()` doivent être respectivement la première et la dernière fonction MPI
- `MPI_COMM_WORLD` désigne l'ensemble des processus pouvant communiquer
- La taille d'un communicateur est retournée par `MPI_Comm_size()`
- Le rang d'un processus est retourné par `MPI_Comm_rank()`



Communications point à point



Echange de messages

- Caractéristiques d'un message
  - Tâche expéditrice
  - Tâche destinataire
  - Données à échanger
- Protocole
  - Expéditeur doit envoyer le message
    - *send*
  - Destinataire doit recevoir le message
    - *recieve*

Principe

- Soient 2 tâches parallèles T0 et T1
  - Chacune a son propre espace d'adressage
  - Chacune exécute des instructions indépendantes de l'autre tâche

Tâche T0

instruction1;
instruction2;

Tâche T1

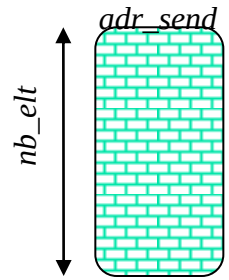
instruction1;
instruction2;

Principe

- T1 a besoin d'informations de T0 (point de synchronisation) :
  - T0 doit envoyer des données à T1 (*send*)
  - les données sont pointées par *adr_send* et de taille *nb_elt*

Tâche T0

instruction1;
instruction2;
send(*adr_send*, *nb_elt*, T1);



Tâche T1

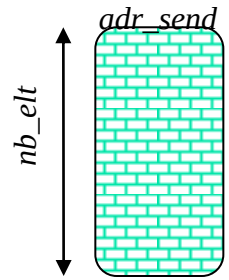
instruction1;
instruction2;

Principe

- T1 doit recevoir les données de T0 (*receive*) :
  - la taille du message *nb_elt* doit être déjà connue par le destinataire ;
  - le destinataire doit préallouer une zone mémoire pour recevoir les données (pointée par *adr_recv*)

Tâche T0

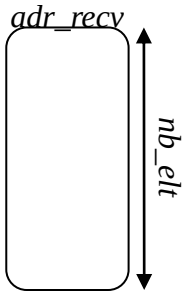
instruction1;
instruction2;
send(*adr_send*, *nb_elt*, T1);



Tâche T1

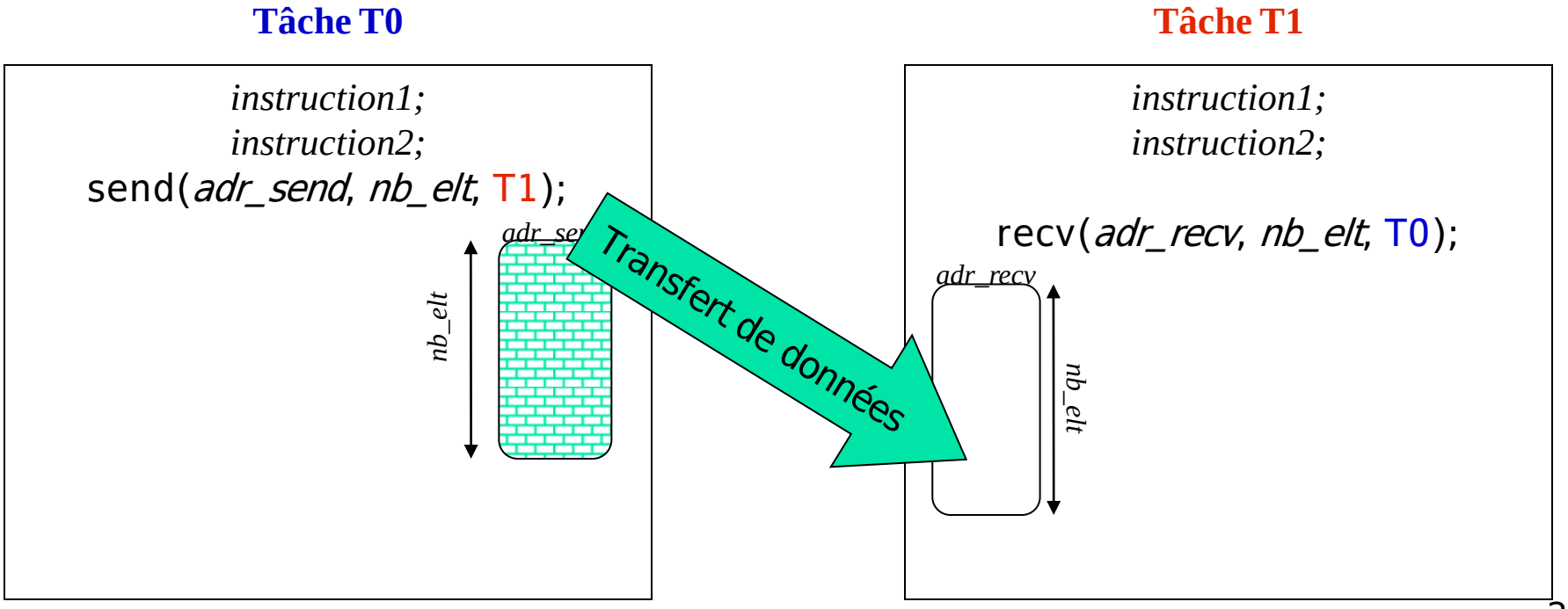
instruction1;
instruction2;

recv(*adr_recv*, *nb_elt*, T0);



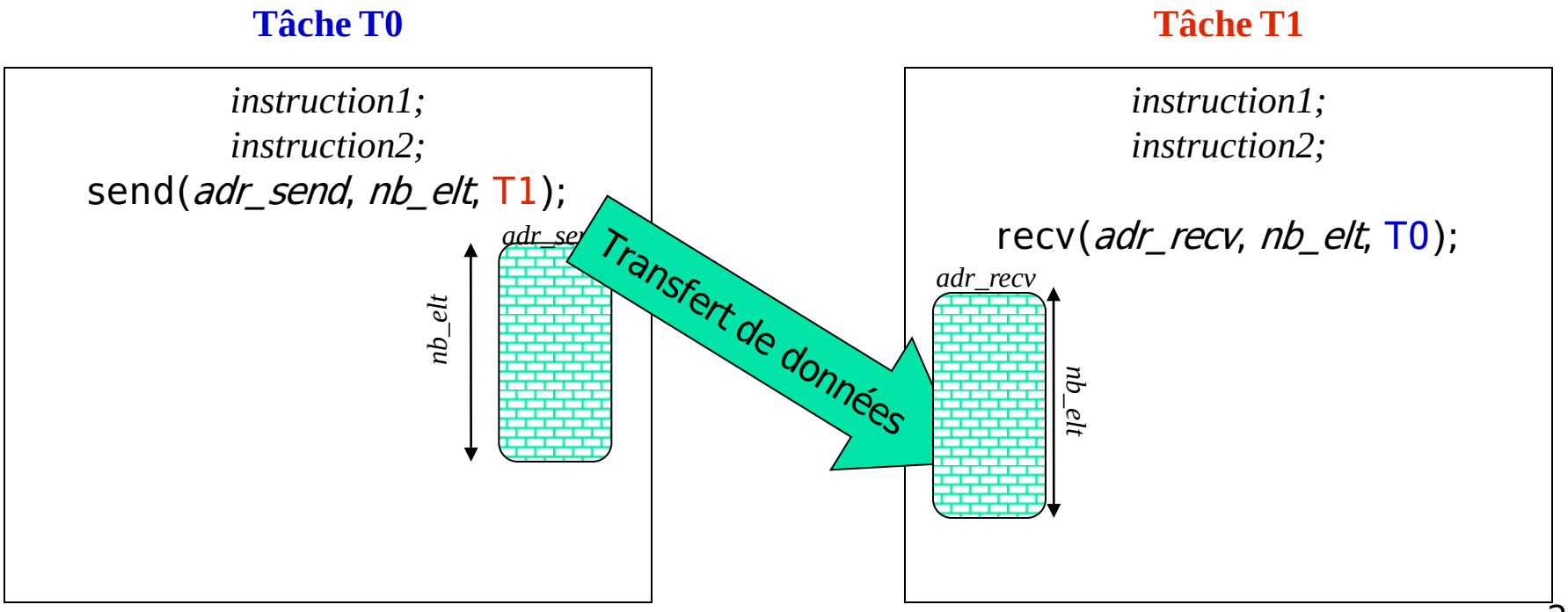
Principe

- L'échange d'information a lieu (communication) :
  - *send* bloque T0 tant que les données ne sont pas envoyées;
  - *receive* bloque T1 tant qu'il n'a pas reçu toutes les données;



Principe

- L'échange d'information a lieu (communication) :
  - *send* bloque T0 tant que les données ne sont pas envoyées;
  - *receive* bloque T1 tant qu'il n'a pas reçu toutes les données;

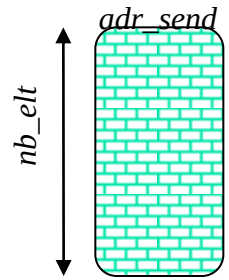


Principe

- T1 a une copie complète des données envoyées par T0

Tâche T0

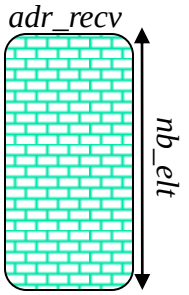
instruction1;
instruction2;
send(*adr_send*, *nb_elt*, T1);



Tâche T1

instruction1;
instruction2;

recv(*adr_recv*, *nb_elt*, T0);

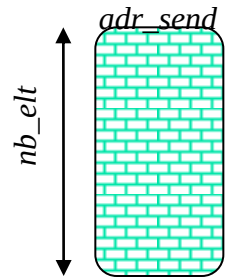


Principe

- Les tâches T0 et T1 peuvent reprendre leurs instructions en parallèle
- Les instructions suivantes de T1 peuvent utiliser les informations pointée par *adr_recv*

Tâche T0

instruction1;
instruction2;
send(*adr_send*, *nb_elt*, T1);

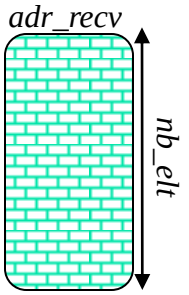


instruction3;

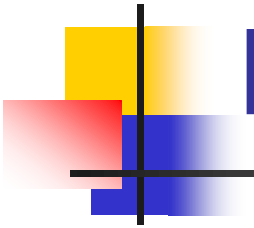
Tâche T1

instruction1;
instruction2;

recv(*adr_recv*, *nb_elt*, T0);



instruction3;



Exemple

- Somme des éléments d'un tableau distribué entre 2 tâches
- Hypothèse
  - Tableau tab de N rééls (N pair) distribué entre 2 tâches T0 et T1
- But :
  - T1 doit afficher la somme des N éléments du tableau tab
- Code ?
  - Indice : calculer sa somme partielle puis communiquer avec son voisin

Exemple

T0

```
double somme_partielle = 0.;
int i;


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


send(&somme_partielle, 1, T1);
```

T0 envoie la somme de sa motié de tableau à T1

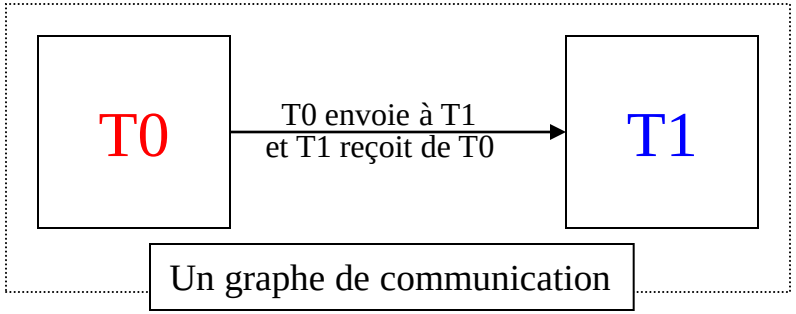
T1

```
double somme_partielle = 0.;
double s0;
int i;

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


recv(&s0, 1, T0);


printf("%d",s0+somme_partielle);
```

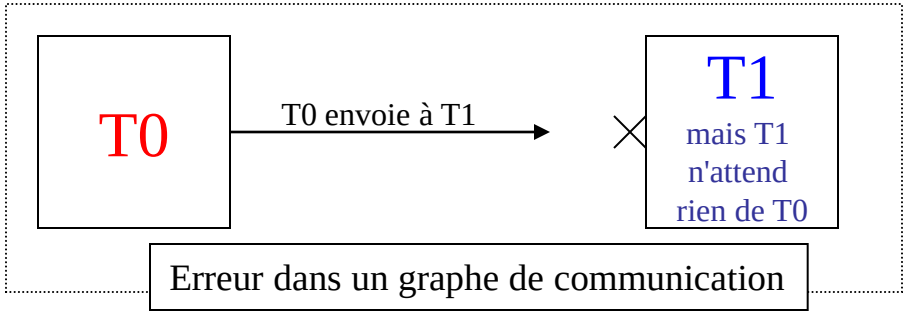
T1 a besoin de la somme partielle de T0

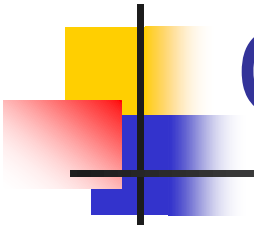
Dualité envoi/réception

- A chaque *send* correspond un *recv* (et réciproquement)
- Modélisation par un graphe orienté :
  - Nœuds représentent les tâches
  - Arcs représentent les communications entre tâches



- L'absence d'un envoi ou d'une réception entraîne le blocage entier de l'ensemble des tâches parallèles





Communication MPI

- Comment MPI interface l'envoi et la réception de messages ?

- Envoi de messages
  - Fonction MPI_Send

Fonction MPI_Send

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

Ces 3 arguments caractérisent le contenu du message à envoyer

Fonction MPI_Send

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

Adresse des données à envoyer

Les données à envoyer sont représentées par un tableau (buf) dont les éléments sont de type datatype.

MPI prédéfinit des types scalaires de base correspondant aux types scalaires C.

Fonction MPI_Send

Valeurs prédéfinies de type MPI_Datatype	*types C correspondants*
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	*un octet*
MPI_PACKED	*paquet de données non contiguës en mémoire*

Fonction MPI_Send

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

La taille du message est count.

Elle n'est pas exprimée en octets, mais en nombre d'éléments de type datatype (permet d'être portable).

Fonction MPI_Send

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

Communicateur dans lequel a lieu l'échange de message.

Communicateur = (sous-)ensemble de processus + contexte de communication (fera l'objet d'un autre cours)

MPI_COMM_WORLD représente l'ensemble des processus susceptible de communiquer entre eux.

Dans la suite du cours, et dans la plupart des cas, on utilisera MPI_COMM_WORLD.

Fonction MPI_Send

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

Rang du destinataire du message.

Le rang est valide dans le communicateur comm.

Dans le cas où comm vaut MPI_COMM_WORLD, dest est compris entre 0 inclus et le nombre total de processus exclu.

Fonction MPI_Send

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

L'étiquette tag identifie de façon unique le message.

Elle permet de distinguer plusieurs messages entre le même expéditeur et le même destinataire

Remarques

- `MPI_Send` est bloquant :
  - au retour de `MPI_Send`, le processus peut utiliser la mémoire pointée par `buf` de façon sûre (i.e. le message a bien été envoyé);
  - Remarque :
    - Un message a été envoyé par le processus source ne signifie pas que le message a été reçu par le processus destinataire

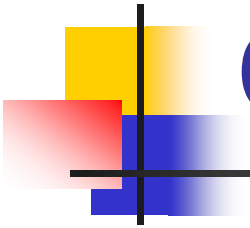
- Comment déterminer l'étiquette d'un message ?
  - Un exemple de détermination d'étiquette :

`tag = src * N + dest` , où

`N` est le nombre total de processus MPI,

`src` rang de l'expéditeur,

`dest` rang du destinataire;



Communication MPI

- Comment MPI interface l'envoi et la réception de messages ?

- Réception de messages
  - Fonction `MPI_Recv`

Réception de messages

```
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)
      );
```

Ces 3 arguments caractérisent le contenu du message à recevoir

Réception de messages

```
int MPI_Recv (
      void *buf(out),
      int count(in),
      MPI_Datatype data(in),
      int source(in),
      int tag(in),
      MPI_Comm comm(in),
      MPI_Status *status(out)
      );
```

Adresse de la zone mémoire des données à recevoir.

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

Taille (maximale) du message à recevoir. (attention aux messages perdus)

Exprimée en nombre d'éléments de type datatype.

La taille du message effectivement reçu est inférieure ou égale à count.

Réception de messages

```
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)
      );
```

Rang de l'expéditeur du message (i.e., rang du processus qui a effectué l'appel à `MPI_Send`).

Ce rang est valide dans le communicateur comm.

`MPI_Recv` admet une valeur supplémentaire pour source : `MPI_ANY_SOURCE` ➔ permet de recevoir un message de n'importe quel expéditeur

Étiquette du message.

Doit être la même que l'appel `MPI_Send` correspondant.

Réception de messages

```
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)
      );
```

"Statut" du message reçu (ensemble d'information retourné par `MPI_Recv`).

Réception et statut

- `MPI_Status` est une structure de données C :

```
struct MPI_Status {
    int MPI_SOURCE; /* expéditeur du message reçu : utile avec MPI_ANY_SOURCE */
    int MPI_TAG; /* étiquette du message reçu : utile avec MPI_ANY_TAG */
    int MPI_ERROR; /* code si erreur */
};
```

- Si la taille du message reçu n'est pas connue, il est possible d'extraire cette information avec la fonction `MPI_Get_count`

```
int MPI_Get_count(
    MPI_Status *status(in), /* statut retourné par MPI_Recv */
    MPI_Datatype datatype(in), /* type des éléments du message reçu */
    int *count(out) /* taille du message en nombre d'éléments de type datatype */
    );
```

Exemple MPI (2 tâches)

```
int main(int argc, char **argv) {
  double somme_partielle = 0., s0;
  int i, rang;
  MPI_Status status;
  MPI_Init(&argc, &argv); /* initialisation de la bibliothèque MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rang); /* récupération du rang du processus MPI */

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

  tag = 1000; /* un nombre pour identifier le message */
  if (rang == 0) {
    MPI_Send(&somme_partielle, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
  } else {
    MPI_Recv(&s0, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
    printf("Somme totale = %d\n", s0+somme_partielle);
  }
  MPI_Finalize(); /* aucune fonction MPI après cet appel */
  return 0;
}
```

Exemple MPI (p tâches)

```
som = 0.;                          /* Chaque processus a N/P éléments du tableau */
for( i = 0 ; i < N/P ; i++ )       /* distribué, et effectue sa somme            */
  som += tab[i];                   /* partielle                                  */

if (rang == 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 reçu de #%d\n", sta.MPI_SOURCE); /* rang de l'expéditeur */

    som += s; /* contribution du processus sta.MPI_SOURCE à la somme totale */
  }

} else {

  /* Tout processus différent de 0 envoie sa somme partielle au processus 0 */
  MPI_Send(&som, 1, MPI_DOUBLE, 0, rang, MPI_COMM_WORLD);
}
```

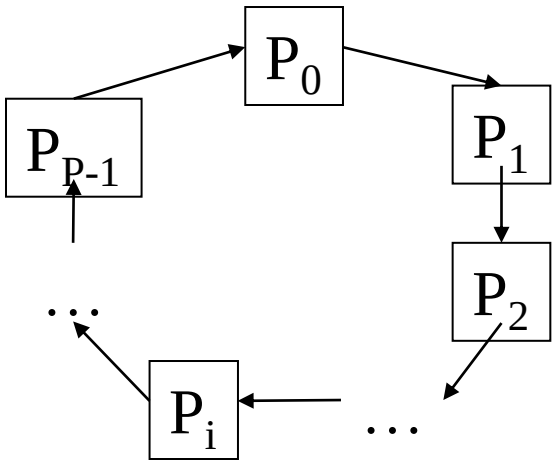
Communications bloquantes

- `MPI_Send` et `MPI_Recv` sont des appels bloquants
  - `MPI_Send` garde la main tant que le message n'a pas été envoyé
  - `MPI_Recv` garde la main tant qu'il n'a pas reçu le message
- Problème ?
  - Attention aux potentiels blocages
  - *Deadlocks*

Communication en anneau

```
gauche = (rang + P - 1) % P;
droite = (rang + 1) % P;

if (rang == 0)
   m = 0;

/* je reçois de la gauche */
MPI_Recv(&m, 1, MPI_INT, gauche, tag1, MPI_COMM_WORLD, &sta);

/* j'envoie à la droite */
MPI_Send(&m, 1, MPI_INT, droite, tag2, MPI_COMM_WORLD);
```

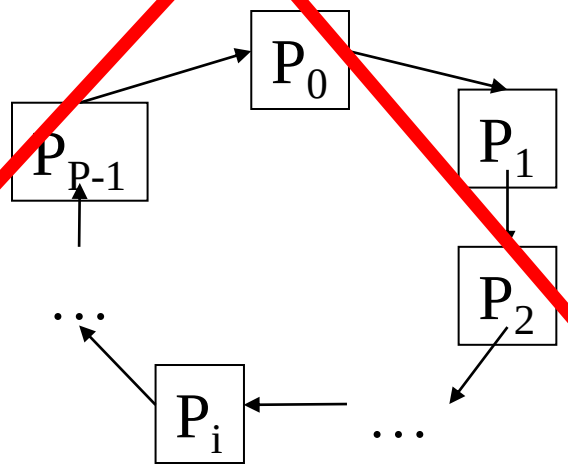
Exemple : un anneau :
les processus se passent un message dans l'ordre croissant de leurs rangs



Communication en anneau

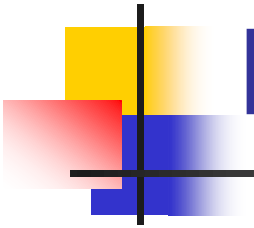
```
gauche = (rang + P - 1) % P;
droite = (rang + 1) % P;

if (rang == 0)
   m = 0;

/* je reçois de la gauche */
MPI_Recv(&m, 1, MPI_INT, gauche, tag1, MPI_COMM_WORLD, &sta);

/* j'envoie à la droite */
MPI_Send(&m, 1, MPI_INT, droite, tag2, MPI_COMM_WORLD);
```

Exemple : un anneau :
les processus se passent un message dans l'ordre croissant de leurs rangs



Chaque processus P_i attend un message de P_{i-1} avant d'envoyer le message à P_{i+1}.
Pour que P_i rende la main, P_{i-1} doit faire un envoi, mais P_{i-1} est bloqué car il attend lui-même un message de P_{i-2}, etc.…

⇒ **deadlock**



Résumé

- Un message est caractérisé par les rangs des tâches expéditrice et destinataire, par les données à échanger, et par l'étiquette du message
- Les envois et les réceptions doivent se faire de manière explicite
- A chaque envoi doit correspondre une réception (et réciproquement)
- MPI_Send et MPI_Recv sont des appels bloquants : attention aux *deadlocks*