MPI
Une Bibliothèque de communication par messages

Master SAR 1
Module AR (4I403)

MPI
Une Bibliothèque de communication par messages

Master SAR 1
Module AR (4I403)

Qu'est ce que MPI ?

- Message Passing Interface
- Une API standard permettant de faire communiquer par messages des processus
  - Distants
  - Sur un ensemble de machines hétérogènes
  - Ne partageant pas de mémoire
- Pour des applications écrites en C, C++ ou Fortran

Qu'est ce qu'il y a dans MPI ?

- Des communications point à point
  - Plusieurs modes de communication.
    - Bloquant/non Bloquant
    - Asynchrone/Synchrone
  - Support pour les buffers structurés et les dérivés
  - Support pour l'hétérogénéité
- Routines de communication collectives
  - Broadcast dans un sous-groupe de processus
  - Opérations pré-définies et utilisateur

Qu'est ce qu'il y a dans MPI ?

- Des communications point à point
  - Plusieurs modes de communication.
    - Bloquant/non Bloquant
    - Asynchrone/Synchrone
  - Support pour les buffers structurés et les dérivés
  - Support pour l'hétérogénéité
- Routines de communication collectives
  - Broadcast dans un sous-groupe de processus
  - Opérations pré-définies et utilisateur

Comment programmer sous MPI ?

- **Chaque processus a son propre flot de contrôle et son propre espace d'adressage.**
- **Attention** : une variable déclarée globale dans le code source reste locale et privée par rapport au processus
  - Les affichages sont redirigés via le réseau sur le terminal qui a lancé le programme MPI
- **Utilisation d'une représentation interne des données**
  - Utile pour l'hétérogénéité (implique une définition de type générique)
- **Les processus ne peuvent communiquer que via les primitives MPI.**
  - Nous utiliserons le langage C : tous les noms de routines commencent par **« MPI_ »**

Primitives de Bases

- **Initialisation** : La première fonction à appeler

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

- **Finalisation** : doit être la dernière fonction MPI à être appelée

```
int MPI_Finalize();
```

- Ces deux procédures doivent être appelées exactement une fois par tous les processus
- Directive préprocesseur : `#include <mpi.h>`

Communicateur

- **Type :** `MPI_Comm`
- **Définit un ensemble statique de processus.**
  - Peut être créé ou détruit en cours d'application
  - Utile pour les communications collectives
- **Un processus a un ou plusieurs identifiants de la forme :**

  (communicateur, numéro)
  - Un processus peut appartenir à plusieurs communicateurs
  - Il peut avoir un numéro différent dans chaque communicateur
- **Chaque communication MPI doit préciser le communicateur concerné.**
  - Permet l'identification de l'émetteur / du destinataire
- `MPI_COMM_WOLRD` **est prédéfini et regroupe l'ensemble des processus du système**

Avoir des informations sur le communicateur

- **Le nombre de processus présents dans le communicateur comm:**

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

- **Mon numéro dans le communicateur comm:**

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

MPI_COMM_WORLD

- **Le communicateur `MPI_COMM_WORLD` contient tous les processus démarrés**

```
int rang ;
MPI_Comm_rank(MPI_COMM_WORLD, &rang) ;
```

- Le couple **`(MPI_COMM_WORLD, rang)`** identifie un processus de manière unique

Types de données

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

- Il est possible de construire ses propres types
- Puis construire de nouveaux types récursivement

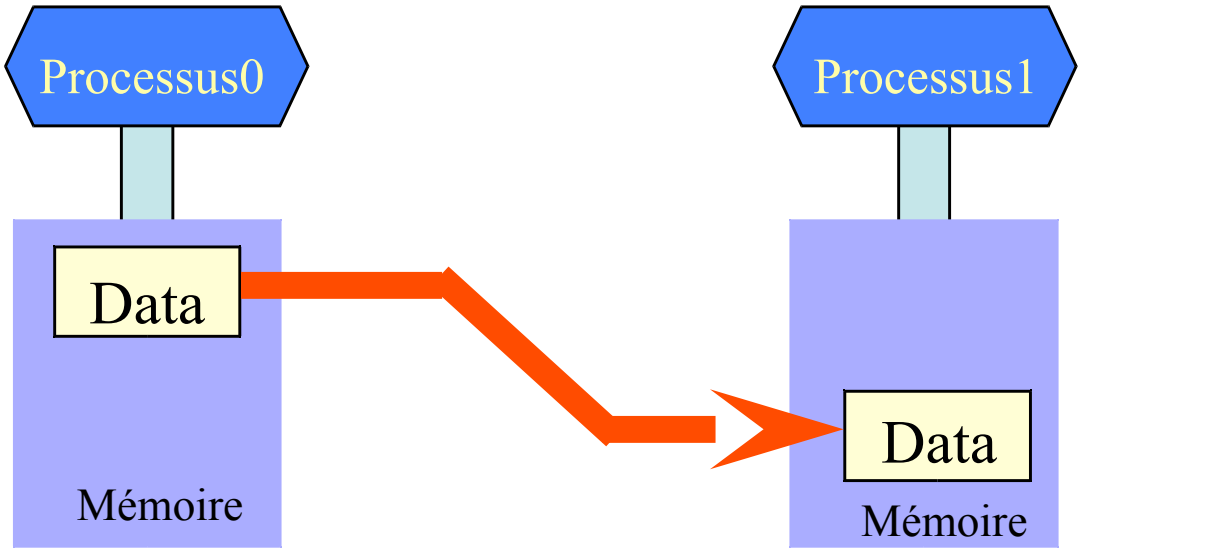
A cause de l'hétérogénéité, la construction de types est complexe !

Structure d'un message sous MPI

- **Un message est divisé en une zone de données et une enveloppe :**
- **Les données du message :**
  - Adresse du buffer contenant les données à envoyer
  - Nombre d'élément à envoyer
  - Type MPI des données (pour masquer l'hétérogénéité)
- **Les données de contrôle :**
  - le Communicateur
  - Le rang du processus (source/destination) dans ce communicateur
  - Le type du message (ou l'étiquette)

La communication point à point

- **Forme la plus simple de communication.**



Émission d'un message

- **Émission bloquante :**

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

Partie données

- **buf :** adresse des données à envoyer
- **count :** le nombre de données à envoyer
- **datatype :** le type MPI à envoyer

Partie contrôle

- **dest :** le processus destinataire du message (dans le communicateur considéré)
- **tag :** le type du message (ou l'étiquette)
- **comm :** le communicateur considéré

Réception d'un message

- **Consomme un message dans le buffer de réception**
- **Réception bloquante :**

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

Partie données

- **buf :** adresse du buffer où l'on va recevoir les données
- **count :** le nombre de données stockable dans le buffer
- **datatype :** le type de donnée MPI à recevoir

Partie contrôle

- **source :** le processus émetteur du message (dans le communicateur considéré)
- **tag :** le type du message (ou l'étiquette)
- **comm :** le communicateur considéré
- **status :** pointeur sur une variable MPI_Status pour avoir des informations complémentaires sur le message reçu

Les « jokers »

- **Pour recevoir un message dont on ne connaît pas l'émetteur *a priori***
  - **`MPI_ANY_SOURCE`**
- **Pour recevoir un message dont on ne connaît pas le type *a priori***
  - **`MPI_ANY_TAG`**
- **Possibilité de récupérer l'identité de l'émetteur ou le type du message à travers l'objet MPI_Status**

L'objet Status

- **Pour obtenir des informations sur un message dans le buffer de réception :**
  - L'émetteur
  - L'étiquette
  - Le nombre de données présent dans les données du message
- **Structure de type prédéfini `MPI_Status`**
  - accès à la valeur du tag : `status.MPI_TAG;`
  - accès à l'identité de l'émetteur : `status.MPI_SOURCE;`
- **Peut être interrogé par l'intermédiaire d'une routine**

```
MPI_Get_Count(&status, datatype, &count);
```

  - Renvoie dans **`count`** le nombre d'objets de type `datatype` reçus

Tester le buffer de réception

- Possibilité de savoir si un message est présent **sans le consommer**
- Version bloquante :

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- Version non-bloquante :

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,MPI_Status status);
```

  - **source :** l'émetteur attendu (dans le communicateur considéré)
  - **tag** : l'étiquette attendue
  - **flag** : paramètre résultat, vaut 0 s'il n'y a pas de message du processus source avec l'étiquette tag dans le buffer de réception
  - **comm** : le communicateur considéré
- Possibilité d'utiliser les jokers : le status identifie les messages et permet de définir une zone de réception appropriée pour les données

Primitives de communication MPI

- Bloquantes :
  - Asynchrone

    Émission (**`MPI_Send`**) : L'émission se termine lorsque le message a bien été envoyé sur le réseau, le buffer d'émission peut ainsi être réutilisé.

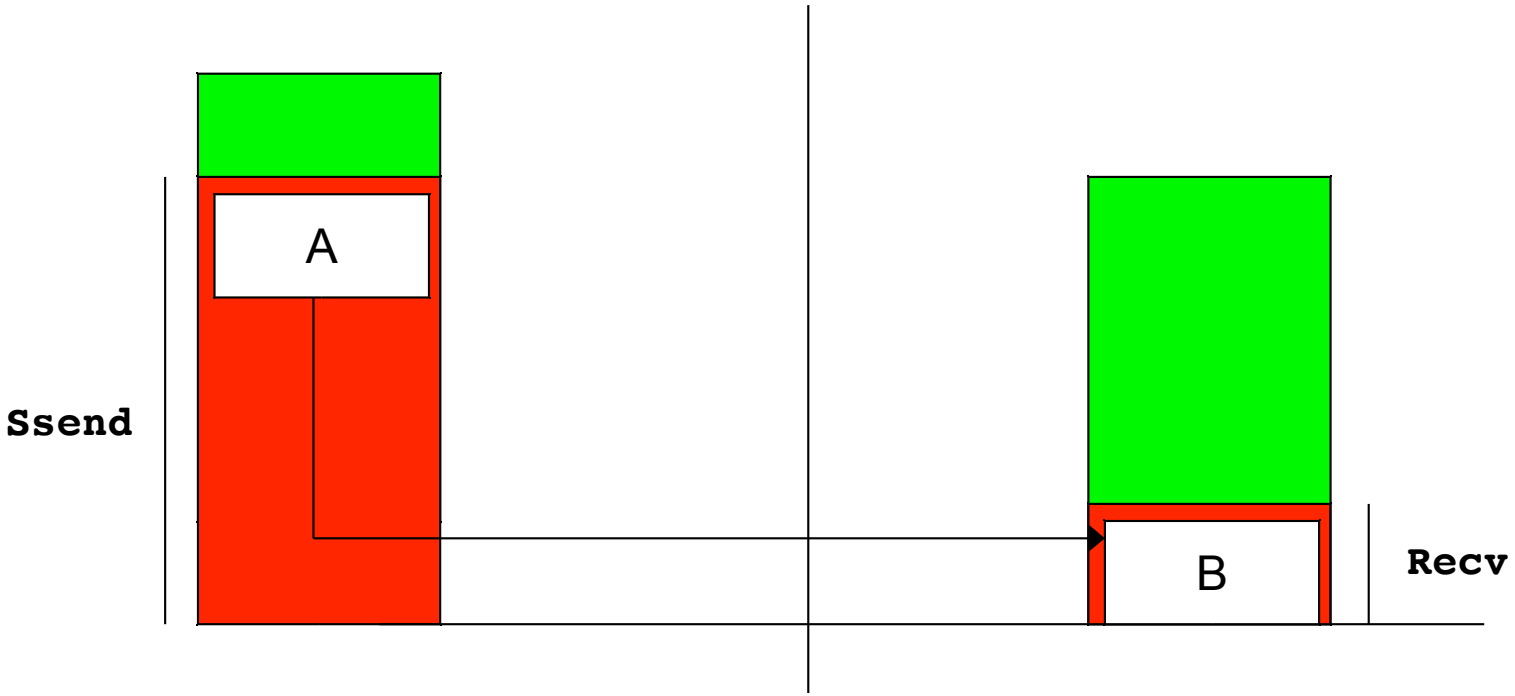
    Réception (**`MPI_Recv`**) : La réception se termine lorsque les données attendues sont disponibles dans le buffer de réception
  - Synchrone

    Émission (**`MPI_Ssend`**) : l'émission se termine lorsque le destinataire a fait sa réception.

- Non bloquantes : Permet de recouvrir les communications par le calcul. (**`MPI_Isend`**, **`MPI_Irecv`** et **`MPI_Issend`**)

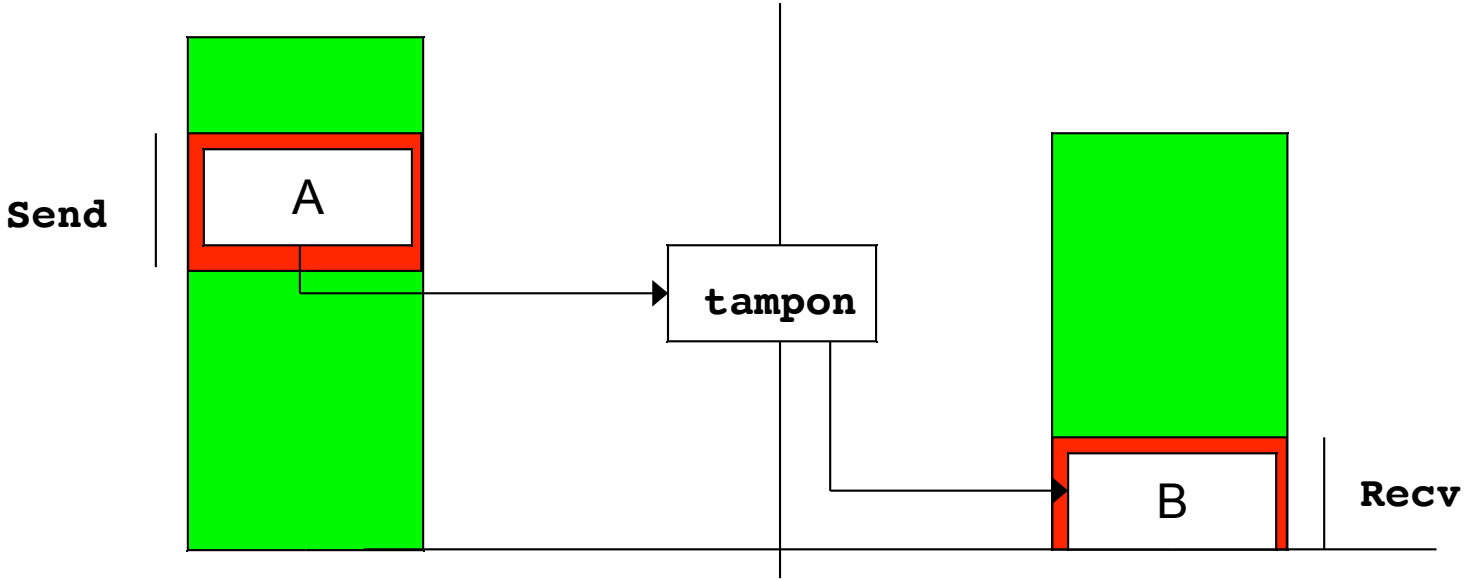
Émission synchrone bloquante

- L'émission ne peut s'exécuter que lorsque le destinataire appelle une primitive de réception (rdv)
  - Pas besoin de zone de stockage intermédiaire : les données sont copiées directement dans la zone de réception du destinataire.



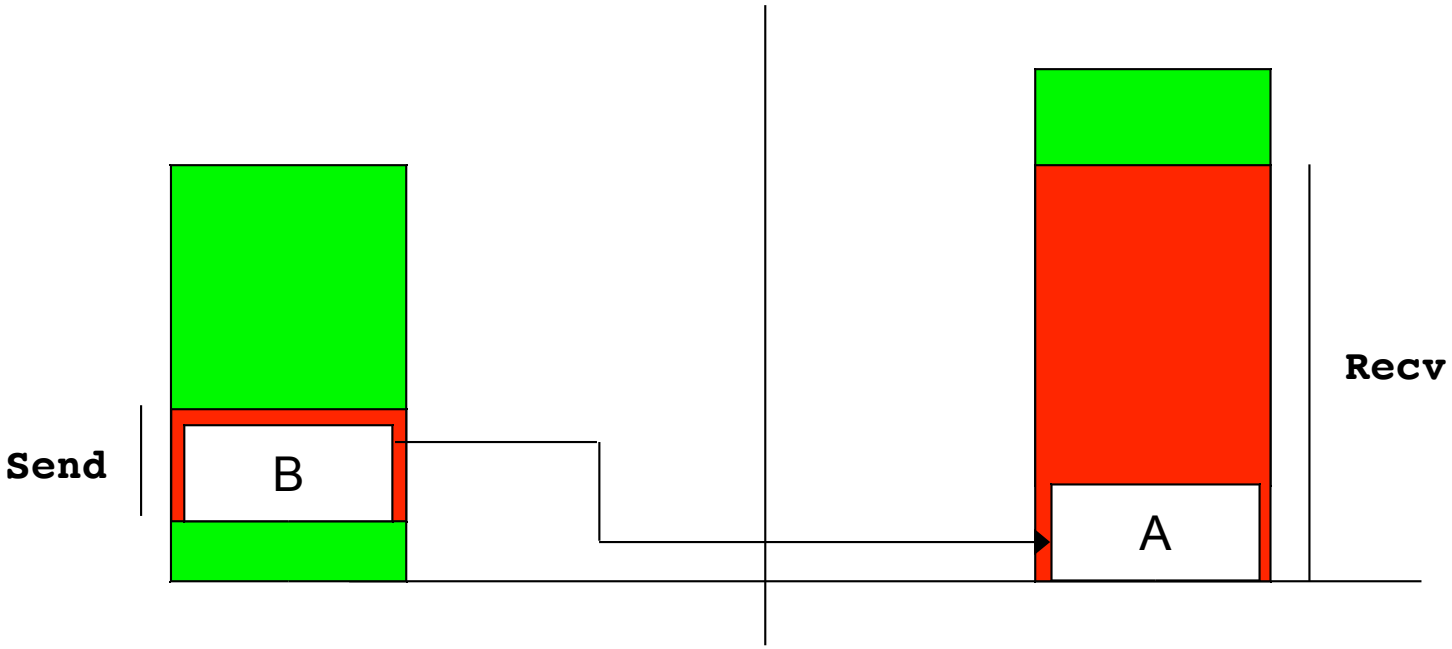
Émission asynchrone bloquante

- Lorsque l'émission se termine :
  - Soit les données ont été réçues
  - Soit elles sont en transit (copiées dans un tampon système)
- Possibilité d'une zone de stockage intermédiaire
  - Dépend de l'implémentation



Réception bloquante

- La réception se termine lorsque les données sont disponibles dans le buffer de réception



Exemple à 3 processus

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

#define TAG 99

int main(int argc, char *argv[]){
   int msg = 2;
   int rang;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rang);

   if (rang == 0) { /* 0 attend 2 messages */
      MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, &status);
      printf("Hello %d !\n", status.MPI_SOURCE);
      MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, &status);
      printf("Hello %d !\n", status.MPI_SOURCE);
   } else {
      MPI_Send(&msg, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
   }
   MPI_Finalize();
   return 0;
}
```

Les communications MPI

- **Communications fiables**
  - Tout message émis est reçu exactement une fois
- **Communications FIFO**
  - Pour tout couple de processus (Pi, Pj) :
  - Pour tout couple (m, m') de messages émis par Pi à destination de Pj :
    - Si m est envoyé avant m', alors m est reçu avant m'
  - Cette condition ne s'applique pas si les destinataires sont différents

- **Le buffer de réception n'est pas une file**

Le buffer n'est pas une file…

```
#include <stdio.h>
#include <mpi.h>
#define TAG1 99
#define TAG2 98

int main(int argc, char *argv[]){
   int msg = 3;
   int rang;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
   if (rang == 0) { /* 0 envoie a 1 */
      MPI_Send(&msg, 1, MPI_INT, 1, TAG1, MPI_COMM_WORLD);
      msg = 5;
      MPI_Send(&msg, 1, MPI_INT, 1, TAG2, MPI_COMM_WORLD);
   } else {
      MPI_Recv(&msg, 1, MPI_INT, 0, TAG2, MPI_COMM_WORLD, &status);
      printf("I received %d \n", msg);
      MPI_Recv(&msg, 1, MPI_INT, 0, TAG1, MPI_COMM_WORLD, &status);
      printf("J'ai recu %d \n", msg);
   }
   MPI_Finalize();
   return 0;
}
```

Ce programme fonctionne !!

Mais ...

```
#include <stdio.h>
#include <mpi.h>
#define TAG1 99

int main(int argc, char *argv[]){
   int msg = 3;
   int rang;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
   if (rang == 0) { /* 0 envoie a 1 */
      MPI_Send(&msg, 1, MPI_INT, 1, TAG1, MPI_COMM_WORLD);
      msg = 5;
      MPI_Send(&msg, 1, MPI_INT, 1, TAG1, MPI_COMM_WORLD);
   } else {
      MPI_Recv(&msg, 1, MPI_INT, 0, TAG1, MPI_COMM_WORLD, &status);
      MPI_Recv(&msg, 1, MPI_INT, 0, TAG1, MPI_COMM_WORLD, &status);
      printf("J'ai recu %d \n", msg);
   }
   MPI_Finalize();
   return 0;
}
```

La valeur affichée est toujours 5

Schéma d'un buffer de réception

Nombre de processus = 3
Nombre de types de message = 2

Sur chaque processus, un buffer de réception. Exemple :

Expéditeur	Tag	tête de file <-------- Messages ---------- queue de file										
Processus 0	0	m1									m10	
	1			m3						m9		
Processus 1	0		m2			m5						m11
	1				m4			m7				
Processus 2	0						m6					
	1								m8			

Si on note (processus, tag) les réceptions, donnez l'ordre dans lesquels les messages sont consommés avec la série de réceptions :

(any,any) (any,1) (2,any) (1,1) (1,0) (any,any) (0,any) (2,1) (1,any) (any,any) (any,any)

Communications collectives

- **Principes**
  - Routines de haut niveau permettant de gérer simultanément plusieurs communications
  - Doivent être appelées par tous les processus du communicateur
  - Utilisent les routines de communication point à point
- **Exemples**
  - Broadcast : envoi d'un message à tout le monde : **`MPI_Bcast`**
  - Barrière de synchronisation : **`MPI_Barrier`**
  - Distribution/rassemblement de données : `MPI_Scatter` / `MPI_Gather`
  - Réduction : combinaison des données de plusieurs processus pour obtenir un résultat : `MPI_Reduce`

Diffusion de données

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

- Le contenu de la variable **`buf`** du processus de rang **`root`** est envoyé à tous les processus du communicateur **`comm`**.
- Ces données sont copiées dans la variable **`buf`** de tous les processus de **`comm`**, y compris **`root`**.
  - **buffer** : l'adresse sur les données à diffuser
  - **count** : nombre de données à diffuser
  - **datatype** : type MPI des données
  - **root** : l'identifiant (dans le communicateur comm) du processus source
  - **comm** : le communicateur considéré.

Exemple de diffusion

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

int main(int argc, char *argv[]) {
   char msg[20];
   int rang;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rang);
   if (rang == 0) {   /*-- on choisit 0 comme emetteur --*/

      strcpy(msg, "Hello world !");
   }

   MPI_Bcast(msg, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
   printf("Je suis %d et je recois : %s\n", rang, msg);
   MPI_Finalize();
   return 0;
}
```

Multithread

- Permettre des processus multi-thread

```
int MPI_Init_thread(int *argc, char ***argv, int required, int
                    *provided);
```

- Choix de plusieurs niveaux d'exécution avec le paramètre **`required`**.
  - MPI_THREAD_SINGLE : un seul thread par processus (équivalent à MPI_Init)
  - MPI_THREAD_FUNNELED : plusieurs threads, mais un seul autorisé à utiliser les primitives MPI (celui qui a appelé MPI_Init_thread).
  - MPI_THREAD_SERIALIZED : plusieurs threads, mais les appels aux primitives MPI se font de manière exclusive : au plus un seul appel au même moment
  - MPI_THREAD_MULTIPLE : plusieurs threads peuvent appeler les primtives MPI sans aucune restriction. (Actuellement peu testé et encore instable !)
- Le paramètre résultat **`provided`** indique le niveau disponible sur les machines (dépend de l'implémentation et de la version)

Conclusion

- Programmation intuitive (une fois qu'on dispose d'un algorithme)
- Si le processus est séquentiel, c'est au programmeur de gérer l'indéterminisme :
  - le processus doit-il exécuter une émission ou une réception ?
- Possibilité d'une vraie exécution répartie (processus sur machine distante)
- Il suffit de 6 primitives pour écrire des programmes MPI simples :

```
MPI_Init
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Finalize
```

MPI est simple !