MPI
Une bibliothèque de communication par échange de messages

P. Fortin
Programmation parallèle (PPAR)
Master 2 Informatique - UPMC
(d'après le cours de J.-L. Lamotte)

Qu'est-ce que MPI ?

- *Message Passing Interface* : standard d'interface de bibliothèque de communication et d'environnement parallèle, permettant de faire communiquer par échange de messages des processus
  - distants
  - sur des machines qui peuvent être hétérogènes
- Pour des applications écrites en C, en C++ ou en Fortran
- Historique :
  - PVM : *Parallel Virtual Machine*
  - MPI-1 : 1994
  - MPI-1.2 : clarification du standard MPI-1 (→ version utilisée en TP)
  - MPI-2 : 1997 (et MPI-2.1 adopté en 2008)
  - MPI-3 : en cours de finalisation
- Exemples d'implémentations :
  - domaine public : LAM, MPICH, OpenMPI
  - implémentations constructeurs : IBM, SUN...

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

- Des communications point-à-point
  - plusieurs modes de communication
  - support pour les buffers structurés et les types dérivés
  - support pour l'hétérogénéité
- Routines de communications collectives
  - Communications dans un « groupe » ou un « sous-groupe » de processus
  - Opérations pré-définies ou définies par l'utilisateur

Comment programmer sous MPI?

- Chaque processus a son propre flot de contrôle et son propre espace d'adressage (→ MIMD)
  - mais tous les affichages sont renvoyés sur la machine locale
- Modèles de programmation possibles : SPMD ou MPMD
- Utilisation d'une représentation interne des données
  - masque l'hétérogénéité
- Gestion de la communication par l'intermédiaire des routines de la librairie
  - Les noms des routines MPI débutent par « **MPI_** »;

Primitives de Bases

- Pour **l'initialisation**, on utilise la primitive MPI_Init qui doit être la première fonction MPI appelée :

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

- Pour **sortir de MPI**, on utilise MPI_Finalize qui doit être la dernière fonction MPI appelée. Cette primitive doit être impérativement appelée par tous les processus.

```
int MPI_Finalize();
```


Notion de communicateur

- Type `MPI_Comm`
- Un ensemble statique de processus qui se connaissent.
  - Peut être créé ou détruit en cours d'application
  - Tous les processus d'un communicateur ont un **rang** différent, compris entre *0* et *P-1* (où *P* est le nombre de processus dans le communicateur)
- Chaque communication MPI a lieu par rapport à un communicateur.
  - Définit les processus concernés par la communication
  - Utile pour les communications collectives
- Un processus peut appartenir à plusieurs communicateurs
  - Peut avoir un rang différent dans chaque communicateur
- `MPI_COMM_WORLD` est un communicateur prédéfini qui contient tous les processus.

Primitives de Bases

- Combien de processus y a-t-il dans le communicateur ?

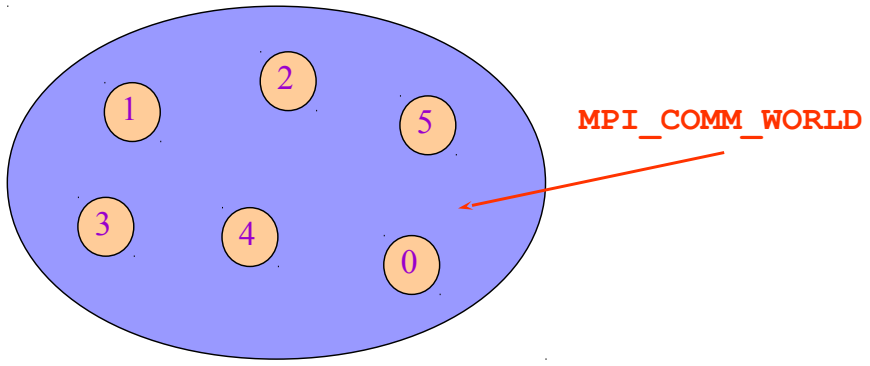
```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- Qui suis-je ?

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


MPI_COMM_WORLD

- Le communicateur `MPI_COMM_WORLD` contient tous les processus démarrés (statique en MPI-1).
- Chaque processus possède un rang unique dans `MPI_COMM_WORLD`.



Structure d'un message sous MPI

- Un message est divisé en une zone de données et une enveloppe :
  - ➤ Les données :
    - adresse du buffer ;
    - nombre d'éléments ;
    - type (pour masquer l'hétérogénéité).
  - ➤ L'enveloppe :
    - rang (identité) du processus
      - pour les envois : indique le destinataire ;
      - pour les réceptions : indique l'expéditeur ;
    - étiquette du message (tag) ;
    - communicateur.

Comment typer les messages ?

- L'étiquette du message : `int tag`
- Permet de séparer données et contrôle
- Valeur d'un tag : 0 .. UB (*Upper Bound*)
  - ➤ MPI garantit que UB ≥ 32 767
  - ➤ LAM sur Linux : UB = 134 973 172
- Un processus peut se mettre en attente d'un message de tag donné
  - ➤ le tag du message attendu doit être égal au tag d'un message reçu (qui n'est pas forcément le premier message reçu)
- Un processus peut se mettre en attente d'un message de tag quelconque : `MPI_ANY_TAG`

Contenu des messages MPI

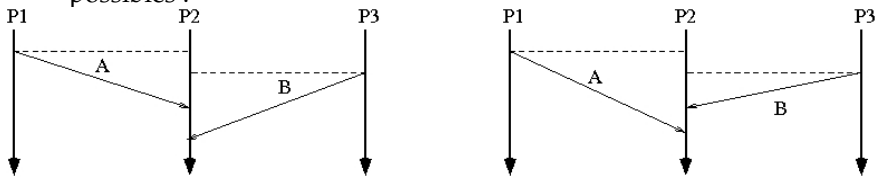
- Vous pouvez avoir
  - ➤ des types élémentaires,
  - ➤ des tableaux de types élémentaires,
  - ➤ des zones contiguës de données,
  - ➤ des blocs de types avec saut,
  - ➤ des structures,
  - ➤ ...
- Construction (éventuellement récursive) de ces types dérivés, puis enregistrement avec `MPI_Type_commit` (et destruction avec `MPI_Type_free`), par tous les processus.
- Création des types dérivés à l'exécution → peuvent dépendre des paramètres de l'application.

Types de données élémentaires

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

Les communications MPI

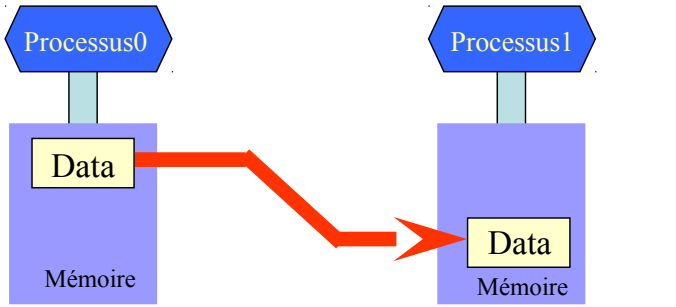
- Communications **fiables**
  - ➤ tout message émis est reçu exactement une fois (ni perte, ni duplication)
- Communications **FIFO** (*First In – First Out*)
  - ➤ pour tout couple de processus (Pi, Pj) :
    pour tout couple (m, m') de messages émis par Pi à destination de Pj :
    → si m est envoyé avant m', alors m est reçu avant m'
  - ➤ cette condition ne s'applique pas si les destinataires (ou les émetteurs) sont différents → système **non déterministe**, plusieurs exécutions possibles :



- Le buffer de réception n'est cependant pas une file à cause des étiquettes (voir plus loin).

La communication point-à-point

- Forme la plus simple de communication.




```
send(buffer, size, [tag], destination)
receive(buffer, buffer_size, [tag], [source])
```


Sous-ensemble MPI-1

- Il suffit de 6 routines pour écrire des programmes MPI simples :

```
MPI_Init(...)
MPI_Comm_size(...)
MPI_Comm_rank(...)
MPI_Send(...)
MPI_Recv(...)
MPI_Finalize()
```


MPI est simple!

Exemple

```
#include <mpi.h>
int main(int argc, char *argv[]){
  char msg[20];
  int my_rank;
  MPI_Status status;

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

  if (my_rank == 0) { /* Hi! I'm process 0! */
    strcpy(msg, "Hello C world !");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1 /* destinataire */,
             99 /* tag */, MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(msg, 20, MPI_CHAR, 0 /* emetteur */,
             99 /* tag*/, MPI_COMM_WORLD, &status);
    printf("I received %s!\n", msg);
  }
  MPI_Finalize();
}
```

MPI : modes de communication

- Opération à réaliser :
  - envoi par P0 du contenu du buffer A
  - réception par P1 des données et stockage dans un buffer B
- Options :
  - communication synchrone / asynchrone
  - communication bloquante / non bloquante

Communications bloquantes

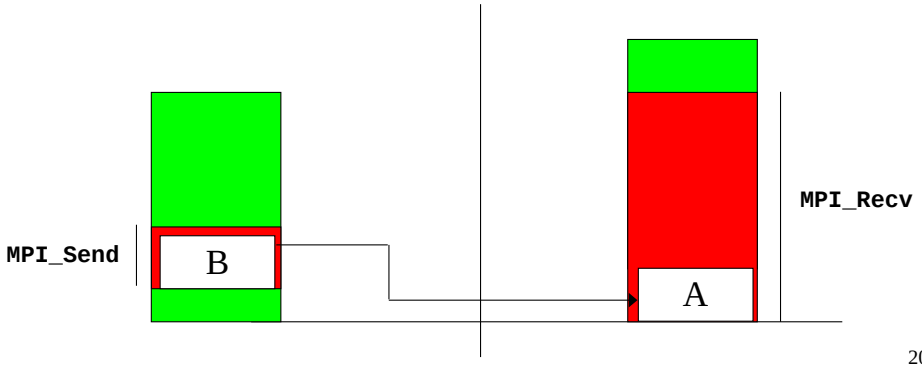
- Emission :
  - Lorsque l'émission se termine, le buffer qui contenait les données envoyées peut être réutilisé.
  - Dans le cas général, rien n'indique que les données aient été effectivement reçues par le destinataire.
- Réception :
  - Lorsque la réception se termine, les données sont disponibles dans le buffer du destinataire.

Communications non bloquantes

- La primitive se termine (retourne) « immédiatement »
  - Lorsqu'une émission non bloquante retourne, les données n'ont pas forcément été extraites du buffer d'émission.
  - Lorsqu'une réception non bloquante retourne, le buffer de réception n'a pas forcément été rempli.
- Nécessité de vérifier que la communication s'est terminée avant de (ré)utiliser le buffer
  - Primitives de test et d'attente : MPI_Test(), MPI_Wait()
- Primitive terminée ≠ communication terminée !

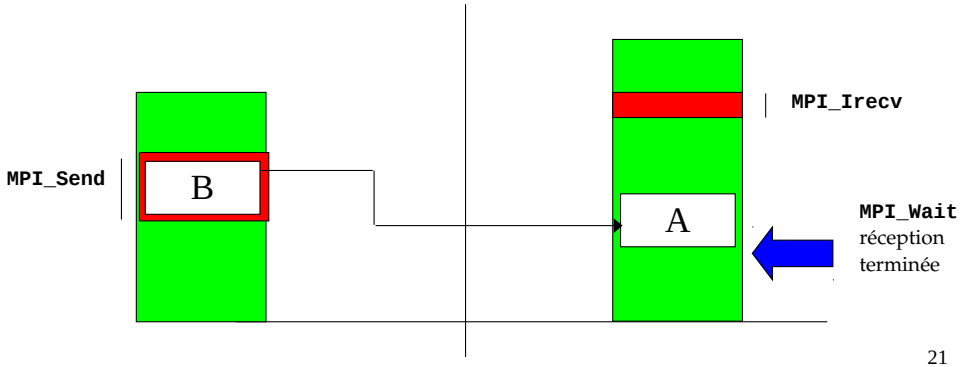
Réception bloquante MPI_Recv

- MPI_Recv retourne quand le transfert est terminé



Réception non bloquante MPI_Irecv

- MPI_Irecv() peut retourner avant même le début du transfert
- MPI_Wait() retourne quand le transfert est terminé

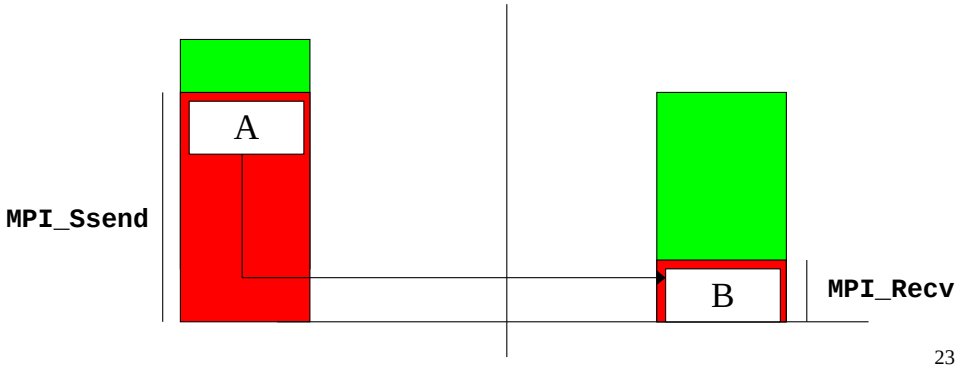


Emission synchrone ou standard

- Emission **synchrone** : terminée quand les données sont entièrement reçues par le destinataire
- Emission **standard** : terminée quand les données sont complètement :
  - soit reçues par le destinataire (mode synchrone ou « **rendez-vous** »)
  - soit transférées dans un tampon système intermédiaire (mode « **envoi immédiat** »)
- Emission standard - **attention** :
  - l'utilisation du tampon intermédiaire dépend de l'implémentation MPI, **et des conditions courantes** d'exécution de l'application
  - man MPI_Send LAM : This function **may** block until the message is receveid. Whether or not MPI_Send blocks depends on factors such as how large the message is, how many messages are pending to the specific destination, etc.
  - En général : messages courts → envoi immédiat
    messages longs → mode rendez-vous

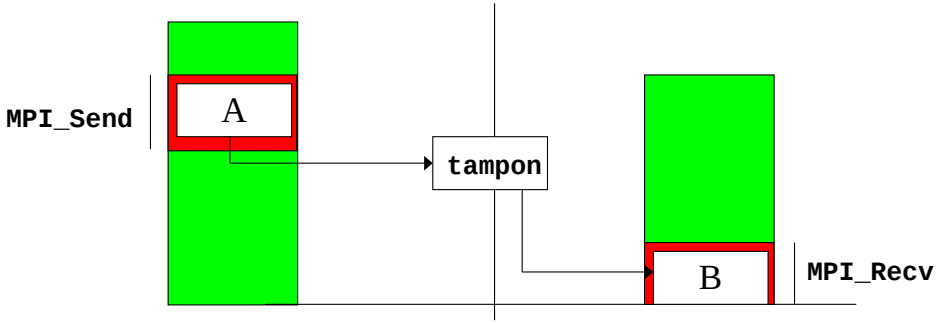
Emission bloquante synchrone MPI_Ssend

- MPI_Ssend() retourne quand les données sont entièrement reçues par le destinataire
- Niveau MPI : pas besoin de tampon intermédiaire, attendre que le récepteur soit prêt pour transférer



Emission bloquante standard MPI_Send

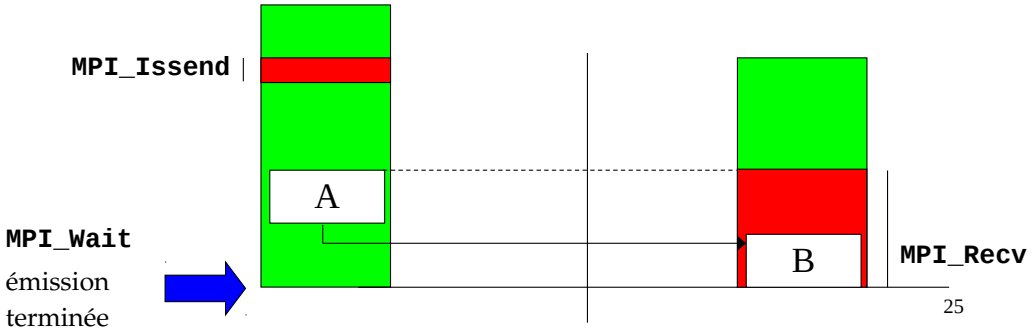
- MPI_Send() retourne quand les données sont reçues ou copiées dans un tampon intermédiaire
- Niveau MPI : utiliser un tampon intermédiaire, s'il peut contenir les données



Exemple avec tampon intermédiaire

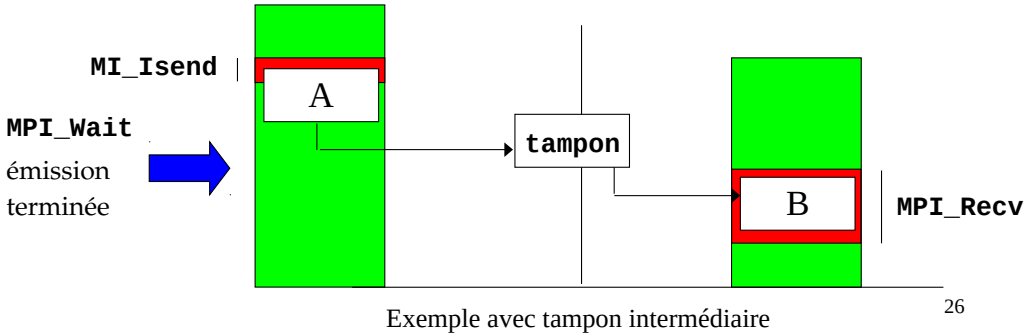
Emission non bloquante synchrone MPI_Issend

- MPI_Issend() peut retourner avant même le début du transfert
- MPI_Wait() retourne quand les données sont complè-tement reçues par le destinataire



Emission non bloquante standard MPI_Isend

- MPI_Isend() peut retourner avant même le début du transfert ou la recopie dans le tampon intermédiaire
- MPI_Wait() retourne quand les données sont complètement reçues par le destinataire ou recopiées dans le tampon intermédiaire



Exemple avec tampon intermédiaire

Synchrone ou standard ?

- Emission synchrone
  - ➤ Pas besoin de tampon système :
    - pas de recopie intermédiaire, pas d'utilisation mémoire supplémentaire
  - ➤ A éviter si l'on ne sait pas du tout quand sera exécuté un Recv :
    - l'émetteur risque d'être bloqué longtemps, inter-blocages (*deadlocks*) possibles...
- Emission standard
  - ➤ Ne se distingue de l'émission synchrone que si la bufferisation est effective
  - ➤ Le mode « envoi immédiat »
    - permet de ne pas attendre le déclenchement de la réception par le destinataire
      → poursuite des calculs applicatifs
    - mais consommation de ressources système :
      - taille des tampons intermédiaires en mémoire
      - temps CPU pour les copies vers/depuis les tampons intermédiaires
  - ➤ Se renseigner sur l'implémentation MPI, étudier la taille et le nombre des messages de l'application

Les primitives MPI

- Emission :
  - ➤ MPI_Ssend : synchrone bloquant
  - ➤ MPI_Send : standard bloquant
  - ➤ MPI_Issend : synchrone non-bloquant
  - ➤ MPI_Isend : standard non-bloquant
- Réception :
  - ➤ MPI_Recv : standard bloquant
  - ➤ MPI_Irecv : standard non-bloquant
- Les primitives non-bloquantes mobilisent beaucoup de ressources système
  - ➤ A n'utiliser que s'il y a de bonnes possibilités de recouvrement des communications par le calcul

Emission : les paramètres

- Emission bloquante : **Send** et **Ssend**
  - L'adresse de début de la zone d'émission : void* buf
  - Le nombre de données envoyées : int nb
  - Le type des données (homogènes) : MPI_Datatype dtype
  - L'identité du destinataire : int dest
  - L'étiquette du message : int tag
  - Le communicateur : MPI_Comm comm

- Emission non bloquante : **Isend** et **Issend**
  - En plus des paramètres ci-dessus, un identificateur de requête (paramètre de sortie) : MPI_Request *req
    → pour identifier par la suite l'émission dont on testera la terminaison

Réception : les paramètres

- Réception bloquante : **Recv** :
  - L'adresse de début de la zone de réception : void* buf
  - La taille de 'buf' en nombre de données de type 'dtype' : int nb
  - Le type des données (homogènes) : MPI_Datatype dtype
  - L'identité de l'émetteur : int source
  - L'étiquette du message : int tag
  - Le communicateur : MPI_Comm comm
  - Les informations complémentaires : MPI_Status *status

- Réception non bloquante : **IRecv**
  - En plus des paramètres ci-dessus, un identificateur de requête (paramètre de sortie) : MPI_Request *req
    → pour identifier par la suite la réception dont on testera la terminaison
  - Pas de 'status' en non bloquant : affecté seulement lorsque la réception est effective (voir `MPI_Wait`)

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 l'étiquette a priori
  - MPI_ANY_TAG

- Dans ce cas, possibilité de récupérer l'identité de l'émetteur ou l'étiquette du message à travers le « status »

L'objet Status

- Pour obtenir des informations sur le message après réception

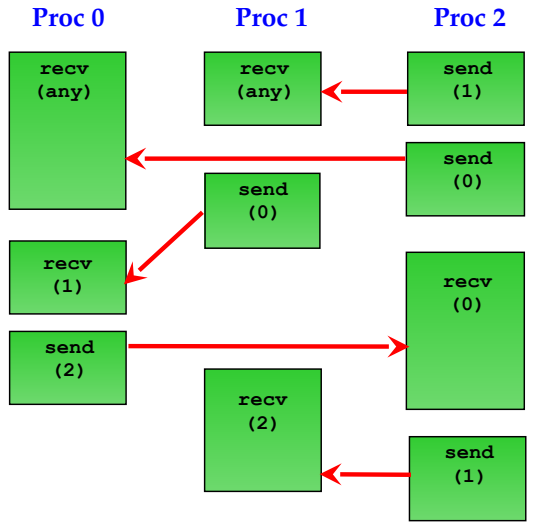
- Structure de type prédéfini **MPI_Status**
  - accès à la valeur de l'étiquette (tag) : **status.MPI_TAG**
  - accès à l'identité de l'émetteur : **status.MPI_SOURCE**

- 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

Exemple 2 (avec 3 processus)

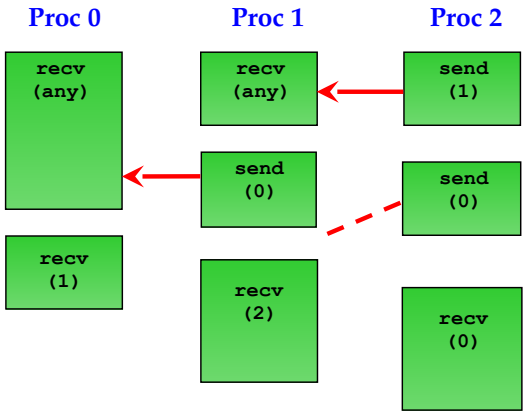
```
int main(int argc, char *argv[]){
   int msg = 2;
   int my_rank;
   MPI_Status status;

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

   if (my_rank == 0) { /* Hi! I'm process 0! */
      MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, 99,
                                          MPI_COMM_WORLD, &status);
      printf("Hello %d !\n", status.MPI_SOURCE);
      MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, 99,
                                          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();
}
```


Attention aux blocages…



Exécution OK

Ca peut arriver !



Blocage !

Fin de communication non-bloquante

- Tester l'arrivée du message
  - Le message que j'ai envoyé a-t-il été transmis ?
  - Le message que j'attends est-il arrivé ?

```
MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)
```

  - 'req' identifie la communication
  - 'flag' donne la réponse :
    - `*flag = 1` : la communication est terminée
    - `*flag = 0` : la communication est en cours
- Attendre l'arrivée du message :

```
MPI_Wait(MPI_Request *req, MPI_Status *status)
```

Exemple 3 (avec 2 processus)

```
#include <mpi.h>
int main(int argc, char **argv) {
  char msg[20];int my_rank;
  MPI_Status status; MPI_Request request;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank == 0) { /*-- process 0 --*/
    sleep(5);
    strcpy(msg, "Hello world !");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, 7, MPI_COMM_WORLD);
  } else {
    MPI_Irecv(msg, 20, MPI_CHAR, 0, 7, MPI_COMM_WORLD, &request);
    sleep(1);  /* je fais autre chose, du calcul par exemple */
    MPI_Wait(&request, &status);
    printf("Je recois : %s\n", msg);
    }
  MPI_Finalize();
}
```


Test du contenu d'un message

- Utile si le contenu du message dépend
  - De l'émetteur
  - Ou de l'étiquette
  - Ou des deux…

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

- On utilise le status pour identifier le message (si jokers) et/ou définir une zone de réception de la taille exactement nécessaire :
  MPI_Probe -> status -> MPI_Get_count -> malloc -> MPI_Recv

- Existe aussi en non-bloquant :

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


Le buffer n'est pas une file…

```
int main(int argc, char *argv[]){
   int msg = 3;
   int my_rank;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   if (my_rank == 0) { /* Hi! I'm process 0! */
      MPI_Send(&msg, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
      msg = 5;
      MPI_Send(&msg, 1, MPI_INT, 1, 99, MPI_COMM_WORLD);
   } else {
      MPI_Recv(&msg, 1, MPI_INT, 0, 99, MPI_COMM_WORLD, &status);
      printf("I received %d \n", msg);
      MPI_Recv(&msg, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, &status);
      printf("I received %d \n", msg);
   }
   MPI_Finalize();
}
```

Programme et exécution corrects !

(pas d'inter-blocage car messages courts : envoi immédiat)

Mais…

```
int main(int argc, char *argv[]){
   int msg = 3;
   int my_rank;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   if (my_rank == 0) { /* Hi! I'm process 0! */
      MPI_Send(&msg, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
      msg = 5;
      MPI_Send(&msg, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
   } else {
      MPI_Recv(&msg, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, &status);
      MPI_Recv(&msg, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, &status);
      printf("La valeur de msg est : %d \n", msg);
   }
   MPI_Finalize();
}
```

La valeur affichée est toujours 5

Communications collectives

Principes

- Routines de haut niveau permettant de gérer simultanément plusieurs communications.
- Doivent être appelées par tous les processus du communicateur

Exemples

- Barrière de synchronisation : tout le monde attend à un point de RDV
  - ➤ **`int MPI_Barrier(MPI_Comm comm)`**
    - Bloque les processus de **`comm`** jusqu'à ce qu'ils aient tous exécuté la primitive
- Broadcast : envoi d'un message à tout le monde;
- Répartition/collection de données
- Réduction (**`MPI_Reduce`**) : combinaison des données de plusieurs processus pour obtenir un résultat (somme, max, min...)
- Autres : **`MPI_Alltoall...`**

La diffusion d'une donnée

```
int MPI_Bcast(void* buf,
              int count,
              MPI_Datatype dtype,
              int root,
              MPI_Comm comm)
```

- root émet le contenu de sa variable **`buf`**
- Tous les processus de **`comm`** reçoivent le contenu de **`buf`**.

Exemple de broadcast

```
#include <mpi.h>

main(int argc, char **argv) {
   char msg[20];
   int my_rank;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   if (my_rank == 0) { /*-- process 0 --*/
      strcpy(msg, "Hello world !");
   }

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


Répartition de données

```
int MPI_Scatter(
      void* sbuf, int scount, MPI_Datatype sdtype,
      void* rbuf, int rcount, MPI_Datatype rdtype,
      int root,
      MPI_Comm comm)
```

- root envoie au processus i **`scount`** données à partir de l'adresse :

  `sbuf + i * scount * sizeof(sdtype)`

- Les données sont stockées par chaque récepteur à l'adresse **`rbuf`**.

Exemple de Scatter à 5 processus

```
#include <mpi.h>
main(int argc, char **argv) {
  char msg[10];
  char recu;
  int my_rank;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank == 2) { /*-- process 2 --*/
     strcpy(msg, "abcde");
  }

  MPI_Scatter(msg, 1, MPI_CHAR, &recu, 1, MPI_CHAR,
                    2, MPI_COMM_WORLD);
  printf("Je suis %d et je recois : %c\n", my_rank, recu);
  MPI_Finalize();
}
```


Collection de données

```
int MPI_Gather(
      void* sbuf, int scount, MPI_Datatype sdtype,
      void* rbuf, int rcount, MPI_Datatype rdtype,
      int root,
      MPI_Comm comm)
```

- Chaque processus (y compris root) envoie à root **scount** données à partir de l'adresse **sbuf**
- root stocke les données reçues par i à l'adresse :

```
rbuf + i * rcount * sizeof(rdtype)
```


Exemple de Gather à 5 processus

```
main(int argc, char **argv) {
  char msg[10];
  char envoi = 97;    /* = 61h : code ascii de a */
  int my_rank;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  envoi += (char) my_rank;
  MPI_Gather(&envoi, 1, MPI_CHAR, msg, 1, MPI_CHAR,
               3, MPI_COMM_WORLD);
  if (my_rank == 3) {   /* je suis root */
     msg[5] = '\0';     /* fin de chaine */
     printf("contenu de msg : %s\n", msg);
  }
  MPI_Finalize();
}
```

contenu de msg : abcde

MPI-2

- création et gestion dynamique des processus
- mécanisme de communications unilatérales *(one-sided communications)*
- entrées-sorties parallèles
- précision du fonctionnement des appels MPI en mode *multi-thread* : 4 niveaux possibles
  - *MPI_THREAD_SINGLE :* processus MPI mono-thread
  - *MPI_THREAD_FUNNELED :* processus MPI multi-thread mais seul le thread principal effectue les appels MPI
  - *MPI_THREAD_SERIALIZED :* processus MPI multi-thread mais un seul appel MPI à la fois
  - *MPI_THREAD_MULTIPLE :* pas de restriction
- ...