# Travaux dirigés n°1

Xavier JUVIGNY

12 novembre 2020

## Table des matières



# 1 Une introduction à MPI

## 1.1 Une spécification d'interface

Le standard MPI (Message Passing Interface) est une **spécification** pour les développeurs et les utilisateurs d'une librairie d'échange de messages. En soi, ce n'est pas un bibliothèque, mais plutôt les spécifications de ce qu'une telle bibliothèque devrait être.

MPI propose fondamentalement un *modèle de programmation parallèle par échange de messsage* : les données sont déplacées d'un espace d'adressage d'un processus à l'espace d'adressage d'un autre processus à l'aide d'opérations coopératives sur chacun des processus.

Autrement dit, le but de cet interface est de proposer un standard largement utilisé pour écrire des programmes par échanges de messages. L'interface tente d'être :

- Pratique
- Portable
- Optimisée
- Flexible

Les spécifications de l'interface ont été définies pour les langages C et Fortran 90. La troisième version des spécifications de MPI propose également un support pour les langages Fortran 2003 et Fortran 2008.

En fait, les diverses biliothèques MPI existantes diffèrent dans les caractéristiques et la version de spécification qu'elles supportent. Les utilisateurs de ces bibliothèques doivent en avoir conscience.

## 1.2 Modèle de programmation

À l'origine, le standard MPI a été conçu pour des architectures à mémoires distribuées qui avaient commencé à émerger à cette époque (fin des années 1980 – début des années 1990).

Au fur et à mesure du changement des architectures des ordinateurs, les mises en œuvres de MPI ont adapté leurs bibliothèques aux architectures multi-cœurs avec mémoire partagée et aux différents protocoles réseaux.

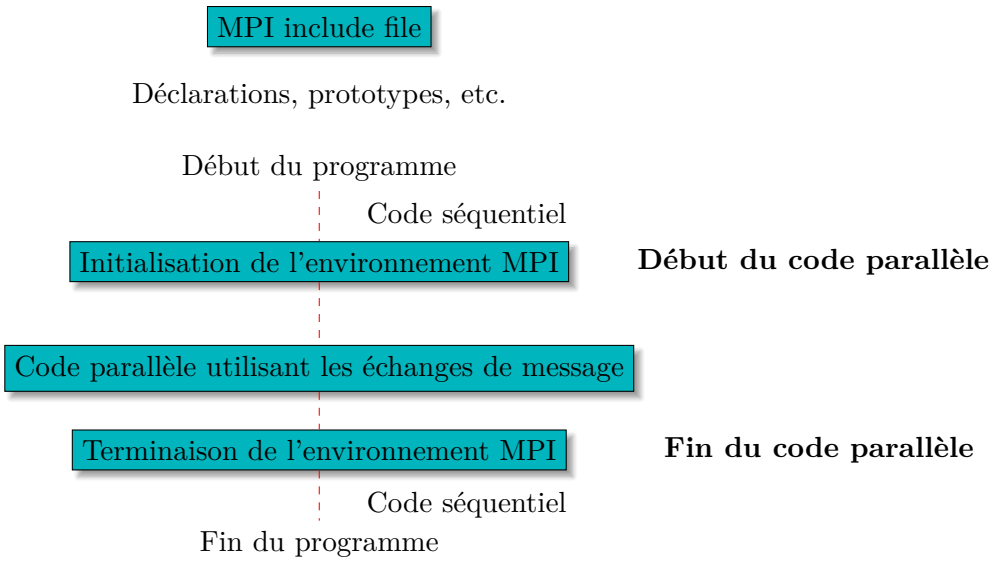
Aujourd'hui, MPI s'exécute sur virtuellement n'importe quelle plateforme :

- en mémoire distribuée ;
- en mémoire partagée ;
- en mémoire hybride (distribuée et partagée).

Néanmoins, le modèle de programmation reste clairement un modèle de **programmation sur mémoire distribuée** sans considération sur l'architecture sous-jacente de la machine.

**Tout parallélisme est explicite** : le programmeur a la responsabilité d'identifier correctement le parallélisme et doit mettre en œuvre ses algorithmes parallèles en utilisant l'interface MPI.

## 1.3 Structure générale d'un programme MPI




Le fichier d'entête est nécessaire pour toute utilisation de la bibliothèque MPI :

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

Les fonctions de l'interface MPI sont de la forme : `rc = MPI_Xxxxxxx(paramètre, paramètre, ....)`.

Par exemple,

```
int rc = MPI_Bsend(&buf, count, type, dest, tag, comm );
```

La valeur entière retournée est un code d'erreur. Si aucune erreur n'a été rencontrée, la valeur retournée vaut `MPI_SUCCESS`. Mais, selon le standard MPI, le comportement par défaut d'un appel MPI est de quitter si il y a une erreur. Cela signifie que vous ne serez pas capable de capturer un code d'erreur autre que `MPI_SUCCESS`. Les types d'erreurs affichés à destination de l'utilisateur dépendent de la mise en œuvre de MPI.

## 1.4 Communicateurs et groupe

MPI utilise des objets appelés communicateurs et des groupes qui définissent un ensemble de processus qui devront communiquer (échanger des messages) entre eux. La majorité des fonctions MPI demandent un communicateur en argument.

À l'initialisation de MPI, un communicateur est prédéfini : MPI_COMM_WORLD. Il inclut tous les processus créés par MPI.

Il est possible de définir un nouveau communicateur à partir d'un communicateur existant en prenant un sous–ensemble de processus parmi les processus du communicateur existant, mais cela dépasse le cadre de ce cours.

# 2 Fonctions de gestion de l'environnement

Ces fonctions sont utilisées pour interroger et définir l'environnement d'exécution MPI, et couvrent divers buts comme l'initialisation et la terminaison de l'environnement MPI, l'interrogation de l'identifiant du processus ou la version de MPI utilisée, etc.

## 2.1 Les fonctions de gestion de l'environnement

Les routines les plus communes sont :

`MPI_Init(&argc,&argv)` Initialise l'environnement d'exécution MPI. Dans un progamme MPI, cette fonction doit être appelée une fois seulement avant toutes autres fonctions MPI. La fonction `MPI_Init` est utilisée pour passer les arguments de la ligne de commande à tous les processus.

`MPI_Comm_size(comm, &size)` Retourne le nombre total de processus MPI dans le communicateur spécifié. Si le communicateur est `MPI_COMM_WORLD`, il représente alors le nombre des tâches MPI disponibles pour votre application.

`MPI_Comm_rank(comm,&rank)` Retourne le rang du processus MPI appelant dans le communicateur spécifié. Initialement, chaque processus se fera assigner un entier unique nommé *rang* entre 0 et le nombre de tâches - 1 dans le communicateur `MPI_COMM_WORLD`. Ce rang est souvent référé comme l'identité de la tâche. Si un processus devient associé avec d'autres communicateurs, il aura un rang unique (différent de celui attribué dans `MPI_COMM_WORLD`!) pour chacun de ces communicateurs.

`MPI_Abort(comm,errocode)` Termine tous les processus MPI associés avec ce communicateur. Dans la majorité des mises en œuvre de MPI, TOUS les processus sont arrêtés sans considération pour le communicateur.

`MPI_Wtime()` Retourne le temps réel passé en seconde ( double précision ) sur le processeur appelant.

`MPI_Wtick()` Retourne la résolution en seconde ( double précision ) de `MPI_Wtime`.

`MPI_Finalize()` Termine l'environnement d'exécution MPI. Cette fonction devra être la dernière fonction appelée dans tout programme MPI – aucune autre fontion MPI ne devra être appelée après celle-ci.

## 2.2 Exemple : Utilisation des fonctions de gestion de l'environnement

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

int main( int argc, char *argv[] ) {
  int numtasks, rank;

  // Initialisation de MPI
  MPI_Init( &argc, &argv );

  // Lit le nombre de tâches
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  // Lit mon rang
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Ici on peut commencer à faire de la programmation
  // parallèle par échange de messages.

  // On termine l'environnement MPI
  MPI_Finalize();

  return MPI_SUCCESS;
}
```

### 2.3 Compilation et exécution d'un programme MPI

Pour compiler un programme utilisant MPI, il est conseillé d'utiliser les outils fourni en standard avec la bibliothèque. Ces outils vont automatiquement trouver les bons chemins pour les fichiers d'entêtes et les bibliothèques utilisées par la mise en œuvre de MPI employée :

`mpicc` : Permet de compiler des sources C. Prend les mêmes arguments de compilation que le compilateur local (GNU C sous Linux par exemple).

`mpiCC` : Permet de compiler des sources C++. Prend les mêmes arguments de compilation que le compilateur local (GNU C++ sous Linux par exemple).

`mpif90` : Permet de compiler des sources Fortran 90. Prend les mêmes arguments de compilation que le compilateur local (GNU Fortran sous Linux par exemple).

Pour exécuter en parallèle sur $n$ tâches un programme MPI, on utilise l'exécutable `mpiexec` fourni en standard avec MPI. Son utilisation est la suivante :
`mpiexec [option] <`**program**`> [ <args> ]`
où

- <**program**> est le nom de l'exécutable à lancer. Il est identifié comme le premier argument non reconnu par MPI.
- `[ <args> ]` : Passe les arguments <args> à tous les processus créés par MPI. Ils devront toujours être les derniers arguments de mpiexec.
- Les options que l'on peut passer à mpiexec sont les suivantes :
  - `-np <n>` : Exécute <n> copies du programme dans autant de processus. Cette option ne peut être utilisée que dans un modèle de programmation SPMD ;
  - `-hostfile <hostfile>` : Utilise le fichier <hostfile> pour connaître le nom des machines ( ou des nœuds de calcul ) sur lesquels on lancera des tâches MPI

Le format du fichier <hostfile> est le suivant :

```
aa slots=a1
bb slots=b1
cc slots=c1
```

où `aa`, `bb` et `cc` sont des noms de machines, et `a1`, `a2` et `a3` sont des entiers indiquant à MPI le nombre de processus qu'on lui conseille de lancer pour chaque machine.

### 2.4 Exercices

**Un hello World parallèle** Écrire un programme qui, pour chaque processus, affichera à l'écran le message suivant :

`Bonjour, je suis la tâche n° xx sur yy tâches.`

où `xx` est le rang de la tâche et `yy` le nombre de tâches lancées par MPI.

## 3 Fonctions d'échanges de messages point à point

### 3.1 Les types d'opérations point à point

Les fonctions d'échanges de messages point à point permettent d'échanger des messages entre deux – et seulement deux – tâches. Une des tâches se charge de l'envoi de données tandis que l'autre tâche se charge de la réception correspondante.

Il y a différents types d'envoi ou de réception, utilisés pour différents buts. Par exemple :

- les envois synchrones ;
- les envois/réceptions bloquants ;

- les envois/réceptions non bloquants ;
- les envois mis en buffer ;
- les envois/réceptions combinés ;
- les envois “prêts”.

À noter que tout type d’envoi peut être reçu avec tout type de réception.
MPI fournit en outre plusieurs routines associées avec les envois et les réceptions telle que celles utilisées pour tester si un message a bien été reçu, ou si un message est prêt à la réception.
Les données sont transmises via un message, lequel est constitué :

- À l’envoi :
  - du numéro de la tâche destinataire
  - d’un numéro permettant d’identifier le message (à la réception)
  - de l’adresse des données à envoyer
  - du nombre de données en envoyer (en nombre d’éléments)
  - du type de données à envoyer.
- À la réception :
  - du numéro de la tâche expéditrice (peut être `MPI_ANY_SOURCE` pour recevoir le premier message arrivé provenant de n’importe quel tâche),
  - du numéro identifiant le message attendu (mais on peut très bien passer `MPI_ANY_TAG` pour recevoir le premier message arrivé provenant d’une tâche spécifiée ou non),
  - de l’adresse de la zone mémoire où on recevra les données
  - de la quantité de données (en nombre d’éléments) attendues et du type de données attendues.

Le type des données est passé à l’aide d’une structure MPI, `MPI_Datatype`, qui peut prendre les valeurs données dans le tableau suivant (avec le type correspondant) :

| Type MPI | Type C |
|---|---|
| MPI_INT | int |
| MPI_SHORT | short |
| MPI_LONG | long |
| MPI_LONG_LONG_INT | long long |
| MPI_UNSIGNED | unsigned |
| MPI_UNSIGNED_LONG | unsigned long |
| MPI_UNSIGNED_SHORT | unsigned short |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_CHAR | char |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_C_COMPLEX | float_Complex |
| MPI_C_DOUBLE_COMPLEX | double_Complex |
| MPI_BYTE | void* |
| MPI_PACKED | données hétérogènes |

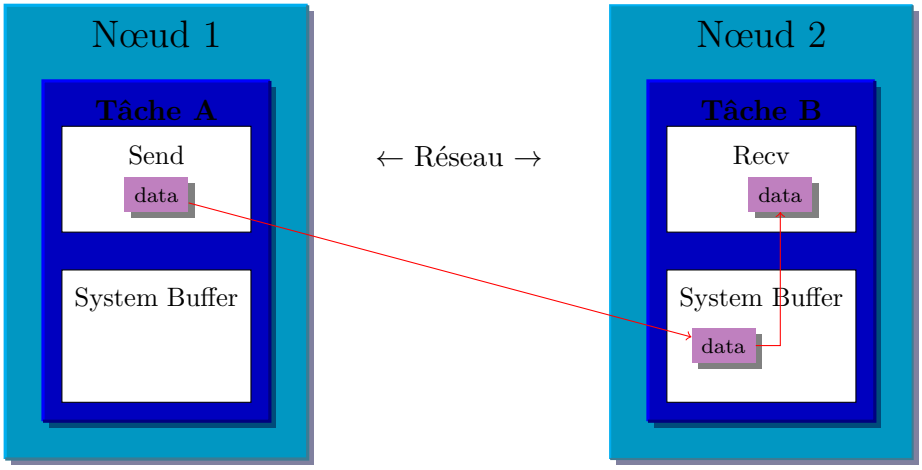
### 3.2 Buffering

Dans un monde parfait, les envois devraient être parfaitement synchronisés avec les réceptions correspondantes, mais en réalité c’est rarement le cas. D’une certaine manière, la bibliothèque MPI utilisée doit pouvoir gérer les données stockées lorsque les deux tâches sont non synchrones.
Par exemple :

- Une opération d’envoi a lieu cinq secondes avant que la réception soit prête – où se trouve le message avant que la réception soit faite ?

- Plusieurs envois arrivent simultanément sur la même tâche qui ne peut accepter qu'une réception à la fois – que deviennent les messages en attente de réception ?

La mise en œuvre de la bibliothèque MPI utilisée décide de la gestion de ces données dans tous les cas qui se présentent (cette gestion ne fait pas partie du standard !). Typiquement, une zone **buffer système** est réservée pour contenir les données en transit.




Cette zone privée est :

- opaque au programmeur et entièrement gérée par la bibliothèque MPI employée ;
- une ressource finie qui peut être facilement épuisée ;
- souvent mystérieuse et mal documentée ;
- capable d'exister du côté de l'envoi, de la réception ou des deux à la fois ;
- quelque chose qui devrait améliorer les performances du programme car permettant l'envoi et la réception en asynchrone.

L'espace d'adresse des variables du programme est appelée le **buffer d'application**. MPI permet à l'utilisateur de gérer son buffer d'envoi dans cet espace.

### 3.3 Bloquant contre non bloquant

La plupart des routines point à point de MPI peuvent être utilisées en communication bloquante ou non bloquante.

Bloquant
- Un envoi bloquant rendra la main seulement après s'être assuré que modifier le buffer d'application est "sûr". "Sûr" signifie ici que ces modifications n'altéreront pas les données qui devront être reçues par la tâche destinataire – elles peuvent par contre être stockées dans un buffer système.
- Un envoi bloquant peut être synchrone, ce qui signifie qu'il y a un échange d'information avec la tâche destinataire pour confirmer que l'envoi est sûr ;
- Un envoi bloquant peut être asynchrone si un buffer système est utilisé pour contenir les données à envoyer au destinataire ;
- Une réception bloquante rend la main seulement après que les données soient arrivées et prêtes à être utilisées par l'utilisateur.

Non bloquant
- Les envois et réceptions non bloquants se comportent de manière similaire – ils rendent la main immédiatement. Ils n'attendent pas qu'une étape de communication soit finie, telle que la copie du buffer d'application vers le buffer système ou l'arrivée du message attendu ;
- Les opérations non bloquantes interrogent simplement la bibliothèque MPI pour exécuter les opérations quand elles sont valides. L'utilisateur ne peut pas deviner le moment où cela va se passer ;

– Il est dangereux de modifier le buffer d'application tant que vous n'êtes pas certain que l'opération non bloquante demandée se soit exécutée. Il existe des routines de type "attente" pour s'en assurer.

– Les communications non bloquantes sont utilisées surtout pour recouvrir les communications avec des calculs et pouvoir ainsi gagner en rapidité d'exécution.

Envoi bloquant

```
myvar = 0;
for ( i = 1; i < numtasks, ++i ) {
  task = i;
  MPI_Send(&myvar,...,...,task,...);
  myvar = myvar + 2;
  // Do some works
  ...
  ...
}
```

Sûr. Pourquoi ?

Envoi non bloquant

```
myvar = 0;
for ( i = 1; i < numtasks, ++i ) {
  task = i;
  MPI_ISend(&myvar,...,...,task,...);
  myvar = myvar + 2;
  // Do some works
  ...
  MPI_Wait (...);
}
```

Dangereux. Pourquoi ?

## 3.4 Routines d'échange de message MPI

Les communications points à points MPI prennent en général une liste d'argument similaire.

| | |
|---|---|
| **Envoi bloquant** | `MPI_Send(buffer, count, type, dest, idtag, comm)` |
| **Envoi non bloquant** | `MPI_ISend(buffer, count, type, dest, idtag, comm, request)` |
| **Réception bloquante** | `MPI_Recv(buffer, count, type, source, idtag, comm, status)` |
| **Réception non bloquante** | `MPI_IRecv(buffer, count, type, source, idtag, comm, request)` |

où

**buffer** Adresse référençant les variables à envoyer ou recevoir. Dans la majorité des cas, c'est simplement le nom de la variable qui doit être envoyée ou dans laquelle on reçoit. Si c'est un simple scalaire, on passe la variable par adresse, ce qui se traduit en C par un symbole & avant la variable.

**count** Indique dans le cas d'un tableau le nombre d'éléments du tableau à envoyer.

**type** Pour des raisons de portabilité, MPI prédéfinit des types de données élémentaires. Voir la table donnée préalablement pour les types fournis par MPI.

**dest** Un argument pour les routines d'envoi spécifiant le numéro de la tâche destinée à recevoir le message ;

**source** Spécifie le numéro de la tâche dont on attend le message. Ce numéro peut être un "joker" et on peut attendre un message provenant de n'importe quelle tâche grâce à la variable `MPI_ANY_SOURCE`.

**idtag** Un entier arbitraire donné par le programmeur pour identifier son message. Ce nombre peut être compris entre 0 et 32767. En réception, la routine attend un message portant cet identifiant pour le recevoir. On peut là encore utiliser un "joker" à l'aide de la variable `MPI_ANY_TAG` ;

**comm** Le communicateur pour lequel les numéros des tâches utilisées sont valides. Sauf si le programmeur a créé des communicateurs, on utilise par défaut `MPI_COMM_WORLD`.

**status** Pour une opération de réception, indique la source et l'identifiant du message. En C, c'est un pointeur sur une structure de type `MPI_Status`.

**request** Pour les opérations non bloquantes, c'est une structure opaque servant à analyser l'état de l'envoi ou de la réception. Cette variable sera utilisée ensuite pour synchroniser la réception ou l'envoi (à l'aide d'une routine de type `MPI_Wait`).

### Fonctions d'échange de message bloquantes

- `MPI_Send` :

  C'est l'opération basique d'envoi bloquant. La fonction ne rend la main que lorsque le buffer d'application dans la tâche est de nouveau libre pour être reutilisé.

```
MPI_Send (&buf,count,datatype,dest,tag,comm)
```

Exemples :

```
int myval = 3;
double mybuf[100] = { ... };
MPI_Send(&myval, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
MPI_Send(mybuf, 100, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD );
```

- MPI_Recv

  Reçoit un message. Ne rend pas la main jusqu'à ce que le buffer d'application soit utilisable par l'utilisateur.

  ```
  MPI_Recv (&buf,count,datatype,source,tag,comm, &status)
  ```

  Exemples :

  ```
  int val;
  double buf[100];
  MPI_Status status;
  MPI_Recv(&val, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,&status);
  MPI_Recv(buf, 100, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &status );
  ```

- MPI_Ssend

  Envoi bloquant synchrone. Envoie un message et attend que le buffer d'application dans la tâche émettrice puisse de nouveau être reutilisé et que la tâche destinataire ait commencé à recevoir le message.

  ```
  MPI_Ssend( &buf, count, datatype, dest, tag, comm)
  ```

  Exemples :

  ```
  int val;
  double buffer[255];
  MPI_Ssend( &val, 1, MPI_INT, rank+1, 101, MPI_COMM_WORLD );
  MPI_Ssend( buffer, 255, MPI_DOUBLE, rank+1, 30, MPI_COMM_WORLD );
  ```

- MPI_Sendrecv

  Envoie un message puis attend une réception avant de bloquer. Bloque jusqu'à ce que le buffer d'application d'envoi soit de nouveau disponible pour utilisation et que le buffer d'application de réception contienne le message reçu.

  ```
  MPI_Sendrecv( &sendBuf, sendCount, sendType, dest, sendTag,
                &recvBuf, recvCount, recvType, source, recvTag,
                comm, &status );
  ```

  Exemples :

  ```
  int sendVal;
  double sendBuf[255];
  int recvVal;
  double recvBuf[255];
  MPI_Status status;
  MPI_Sendrecv( &sendVal, 1, MPI_INT, rank+1, 101,
                &recvVal, 1, MPI_INT, rank-1, 101,
                MPI_COMM_WORLD, &status );
  MPI_Sendrecv( sendBuf, 1, MPI_DOUBLE, rank+1, 101,
                recvBuf, 1, MPI_DOUBLE, rank-1, 101,
                MPI_COMM_WORLD, &status );
  ```

- MPI_Probe

  Effectue un test bloquant sur un message. Les "jokers" `MPI_ANY_SOURCE` et `MPI_ANY_TAG` peuvent être utilisés pour tester un message provenant de n'importe quelle tâche ou identifié par n'importe quel nombre. À la sortie de la fonction, le numéro de l'expéditeur et le numéro identifiant le message seront retournés dans le paramètre `status` (à l'aide de `status.MPI_SOURCE` et `status.MPI_TAG`).

```
MPI_Probe( source, tag, comm, &status );
```

- MPI_Get_count

  Retourne le numéro de l'expéditeur, l'identifiant du message et le nombre d'éléments d'un certain type reçu. Peut être utilisée avec des opérations de réception bloquantes ou non. Le numéro de l'expéditeur et l'identifiant seront retournés dans la structure de `status` comme `status.MPI_SOURCE` et `status.MPI_TAG`.

  ```
  MPI_Get_count( &status, datatype, &count )
  ```

Exemple d'utilisation des fonctions bloquantes d'échange de messages

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

int main(int argc, char *argv[])  {
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;   // required variable for receive routines

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // task 0 sends to task 1 and waits to receive a return message
  if (rank == 0) {
    dest = 1;
    source = 1;
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }
  // task 1 waits for task 0 message then returns a message
  else if (rank == 1) {
    dest = 0;
    source = 0;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

  // query recieve Stat variable and print message details
  MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
         rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

  MPI_Finalize();
}
```

**Fonctions non bloquantes d'échange de message** Les fonctions non bloquantes les plus utilisées pour échanger des messages sont données ci–dessous. On y donne également les fonctions bloquantes qui leur sont associées.

- MPI_Isend

  Utilise une aire en mémoire pour servir comme buffer d'envoi. La tâche continue immédiatement après l'appel sans attendre que le message soit copié du buffer d'application. Une requête est renvoyée permettant au programmeur de connaître l'état de l'envoi. Le programmeur doit s'assurer de ne pas modifier le buffer d'application jusqu'à ce qu'il ait testé si le buffer d'application est de nouveau prêt à l'emploi grâce aux fonctions `MPI_Wait` ou `MPI_Test`.

  ```
  MPI_Isend( &buf, count, datatype, dest, tag, comm, &request )
  ```

- MPI_Irecv

Utilise une aire en mémoire pour servir de buffer de réception. La tâche continue immédiatement après l'appel à la fonction sans attendre que le message soit copié dans le buffer d'application. Une requête est renvoyée permettant au programmeur de connaître l'état de la réception. Le programmeur doit utiliser une des fonctions `MPI_Wait` ou `MPI_Test` pour s'assurer que le message reçu est valable dans le buffer d'application.

```
MPI_Irecv( &buf, count, datatype, source, tag, comm, &request )
```

- `MPI_Issend`

  Envoi synchrone non bloquant. Cette fonction est similaire à `MPI_Isend` sauf que `MPI_Wait` ou `MPI_Test` indiquent si la tâche destinataire a reçu ou non le message.

```
MPI_Issend( &buf, count, datatype, dest, tag, comm, &request )
```

- `MPI_Test, MPI_Testany, MPI_Testall, MPI_Testsome`

  Ces fonctions testent le status d'un envoi ou d'une réception non bloquante spécifique à l'aide de la requête associée. Le paramètre "`flag`" est retourné comme logiquement vrai (=1) si l'opération est finie et logiquement fausse (=0) sinon. Pour plusieurs opérations non bloquantes, le programmeur peut demander à tester si une des opérations, toutes les opérations ou certaines opérations sont finies.

```
MPI_Test(&request, &flag, &status );
MPI_Testany( count, &array_of_requests, &index, &flag, &status );
MPI_Testall( count, &array_of_requests, &flag, &array_of_statuses );
MPI_Testsome( incount, &array_of_requests, &outcount, &array_of_offsets,
              &array_of_statuses );
```

- `MPI_Wait, MPI_Waitany, MPI_Waitall, MPI_Waitsome`

  Ces fonctions bloquent jusqu'à ce qu'un envoi ou une réception spécifié soit fini. Pour plusieurs envois/réceptions non bloquants, le programmeur peut spécifier si une des opérations, toutes ou certaines opérations sont finies.

```
MPI_Wait(&request, &status );
MPI_Waitany( count, &array_of_requests, &index, &status );
MPI_Waitall( count, &array_of_requests, &array_of_statuses );
MPI_Waitsome( incount, &array_of_requests, &outcount, &array_of_offsets,
              &array_of_statuses );
```

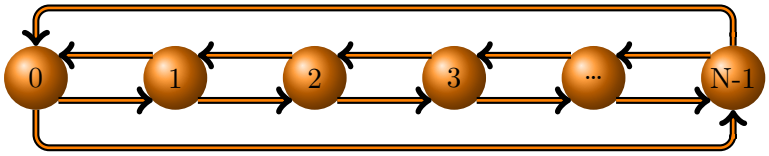
- `MPI_Iprobe`

  Effectue un test non bloquant pour un message. Les "jokers" `MPI_ANY_SOURCE` et `MPI_ANY_TAG` peuvent être utilisés pour tester un message provenant de n'importe quelle autre tâche ou ayant n'importe quel identifiant. Le paramètre entier "`flag`" est retourné avec une valeur logiquement vraie ( = 1 ) si le message est arrivé et logiquement fausse sinon ( = 0 ). Le numéro de l'expéditeur et l'identifiant du message sont retournés via `status` comme `status.MPI_SOURCE` et `status.MPI_TAG`.

```
MPI_Iprobe( source, tag, comm, &flag, &status )
```

Exemple de programme utilisant des messages non bloquants

Échange de données avec son plus proche voisin dans une topologie en anneau :




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

int main(int argc, char *argv[])  {
   int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
```

```
    MPI_Request reqs[4];    // required variable for non-blocking calls
    MPI_Status stats[4];    // required variable for Waitall routine

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // determine left and right neighbors
    prev = rank-1;
    next = rank+1;
    if (rank == 0)  prev = numtasks - 1;
    if (rank == (numtasks - 1))  next = 0;

    // post non-blocking receives and sends for neighbors
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

    MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    // do some work while sends/receives progress in background

    // wait for all non-blocking operations to complete
    MPI_Waitall(4, reqs, stats);

    // continue - do more work

    MPI_Finalize();
}
```

### 3.5 Exercices

Voir section suivante...

## 4 Exercices à rendre pour le TP

**Un hello World parallèle** Compilez et exécutez le programme HelloWorld qui affiche pour chaque processus :

`Bonjour, je suis la tâche n° xx sur yy tâches.`

où `xx` est le rang de la tâche et `yy` le nombre de tâches lancées par MPI.

Tester le programme avec 1 à 16 processus. Que constatez-vous ?

Faites une version où chaque processus écrit son message dans un fichier `Output000xx.txt` en utilisant le fichier `SkeletonMPIProgramWithFilesOutput.cpp` comme canvas de base.

Expliquez en quoi faire une sortie sur fichier est finalement plus clair qu'une sortie sur terminal.

**Envoi bloquant et non bloquant**

Envoi bloquant

Expliquez pourquoi la première version est sûre et la deuxième dangereuse à utiliser. Quelle modification faudrait-il faire pour que la deuxième version soit sûre (dans le sens qu'elle donnera toujours le même résultat) ?

```
myvar = 0;
for ( i = 1; i < numtasks, ++i ) {
  task = i;
  MPI_Send(&myvar,...,...,task,...);
  myvar = myvar + 2;
  // Do some works
  ...
  ...
}
```

Sûr. Pourquoi ?

Envoi non bloquant

```
myvar = 0;
for ( i = 1; i < numtasks, ++i ) {
  task = i;
  MPI_ISend(&myvar,...,...,task,...);
  myvar = myvar + 2;
  // Do some works
  ...
  MPI_Wait (...);
}
```

Dangereux. Pourquoi ?

**Circulation d'un jeton dans un anneau** On veut faire circuler un jeton dans un réseau topologiquement équivalent à un anneau (un exemple de jeton est un fichier de licence).

L'algorithme de circulation du jeton pour `nbp` tâches est le suivant :

1. La première tâche (tâche 0) génère un entier (dont la valeur est choisie arbitrairement par le programmeur, et considéré ici comme le jeton), et l'envoie au processus suivant (le processus 1) ; Il reçoit ensuite le jeton du processus `nbp`-1 et affiche sa valeur ;
2. Le processus 1 reçoit le jeton, incrémente sa valeur de 1, l'affiche et l'envoie au prochain processus ;
3. ...
4. Le processus $p$ reçoit le jeton, l'incrémente de 1, l'affiche, et l'envoie au processus suivant (processus $p+1$) ;
5. ...
6. Le dernier processus (processus `nbp`-1) reçoit le jeton, l'incrémente de 1, l'affiche et l'envoie au processus 0 ;

En vous servant du premier exercice où les processus affichent sur le terminal, mettez en œuvre l'algorithme décrit ci–dessus, puis compilez et testez sur un nombres de tâches divers. Pouvez vous expliquez pourquoi à l'exécution, les messages sont affichés dans l'ordre des numéros des processus (mis à part le processus 0 qui affiche en dernier) ?

Astuce :

1. Lorsque vous programmez en parallèle, essayez de penser au niveau local d'un processus et non au niveau global (sur tous les processus).
2. Pour déboguer votre programme, n'hésitez pas à écrire des messages avant et après envoi/réception de messages, suivit d'un appel à `fflush`. En effet, il arrive souvent lors de la mise au point d'un programme en parallèle de se retrouver dans une situation de *dead–lock*, c'est à dire dans une situation où un ou plusieurs processus attendent indéfiniment un message en réception alors que ce message ne sera jamais envoyé.

On veut modifier maintenant le programme de sorte que :

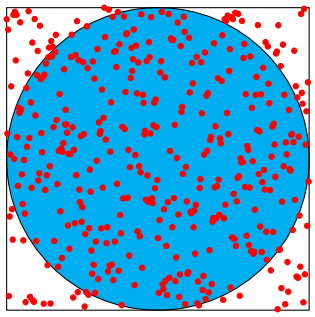
- Chaque tâche génère un entier différent (au choix du programmeur !)
- Envoie son entier au processus suivant (le processus `nbp-1` envoyant au processus 0)
- Incrémente de un l'entier reçu et l'affiche.

Mettez en œuvre, compilez et exécutez avec un nombre variable de tâches. Que constatez vous à l'affichage ? Pourquoi ?

**Calcul de pi par lancer de fléchettes** On veut calculer la valeur de pi à l'aide de l'algorithme stochastique suivant :

- On considère le carré unité $[-1;1] \times [-1;1]$ dans lequel on inscrit le cercle unité de centre $(0,0)$ et de rayon 1.
- On génère des points aléatoirement dans le carré unité.
- On compte le nombre de points générés dans le carré qui sont aussi dans le cercle.
- Soit $r$ ce nombre de points dans le cercle divisé par le nombre de points total dans le carré, on calcule alors pi comme $\pi = 4.r$.

Remarquez que l'erreur faite sur pi décroit quand le nombre de points générés augmente.



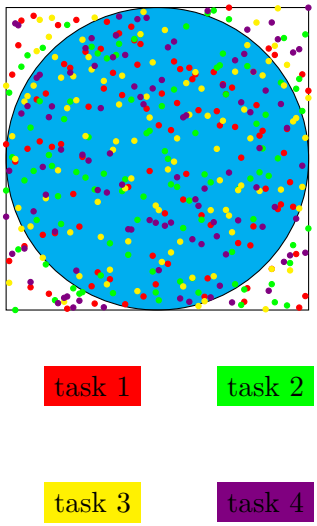
La fonction C++ permettant de générer les points dans le carré et calculer le ratio entre nombre de points dans le cercle et nombre de points générés est la suivante :

```
# include <chrono>
# include <random>
// Attention, ne marche qu'en C++ 11 ou supérieur :
double approximate_pi( unsigned long nbSamples ) {
  typedef std::chrono::high_resolution_clock myclock;
  myclock::time_point beginning = myclock::now();

  myclock::duration d = myclock::now() - beginning;
  unsigned seed = d.count();
  std::default_random_engine generator(seed);
  std::uniform_real_distribution<double> distribution(-1.0,1.0);
  unsigned long nbDarts = 0;
  // Throw nbSamples darts in the unit square [-1:1] x [-1:1]
  for ( unsigned sample = 0; sample < nbSamples; ++sample ) {
    double x = distribution(generator);
    double y = distribution(generator);
    // Test if the dart is in the unit disk
    if ( x*x+y*y<=1 ) nbDarts ++;
  }
  // Number of nbDarts throwed in the unit disk
  double ratio = double(nbDarts)/double(nbSamples);
  return ratio;
}
```

- Couper l'itération de boucle en plusieurs morceaux pouvant être exécutés par différentes tâches simultanément ;
- Chaque tâche exécute sa portion de boucle ;
- Chaque tâche peut exécuter son travail sans avoir besoin d'information des autres tâches ( indépendance des données ) ;
- La tâche maître ( que le programmeur aura choisi parmi ses tâches ) reçoit le résultat des autres tâches à l'aide d'échanges de message point à point.
- Dans un deuxième temps, essayer à l'aide de réceptions non bloquantes de recouvrir les réceptions par du calcul dans le processus maître.

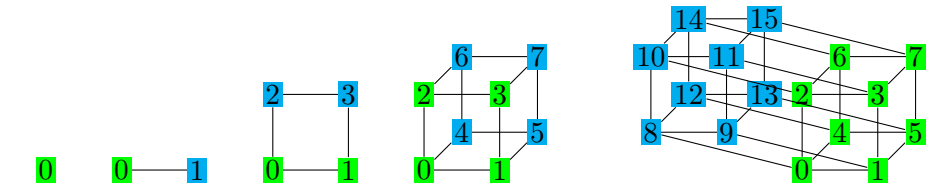
- Mesurez le temps mis par les deux versions (bloquant et non bloquant) et calculez pour chacune des deux versions l'accélération obtenue.




Le concept : diviser le travail parmi les tâches disponibles en communicant des données à l'aide d'appel à des fonctions d'envoi/réception point à point

**Diffusion d'un entier dans un réseau hypercube**$^\star$ On veut écrire un programme qui diffuse un entier dans un réseau de nœuds de calculs dont la topologie est équivalente à celle d'un hypercube de dimension $d$ ( et qui contient donc $2^d$ nœuds de calcul ).

1. Écrire un programme qui diffuse un entier dans un hyper cube de dimension 1 :
   - La tâche 0 initialise un jeton à une valeur entière choisie par le programmeur et envoie cette valeur à la tâche 1 ;
   - La tâche 1 reçoit la valeur du jeton de la tâche 0 ;
   - Les deux tâches affichent la valeur du jeton ;
2. Diffuser le jeton généré par la tâche 0 dans un hypercube de dimension 2 de manière que cette diffusion se fasse en un minimum d'étapes ( et donc un maximum de communications simultanées entre tâches ).
3. Faire de même pour un hypercube de dimension 3
4. Écrire le cas général quand le cube est de dimension $d$. Le nombre d'étapes pour diffuser le jeton devra être égal à la dimension de l'hypercube.



Hypercubes de dimension 0, 1, 2, 3 et 4