Chapitre 2

Message Passing Interface

La Message Passing Interface est une *spécification* (une `interface` au sens de Java ou de OCaml). Il y a *plusieurs* bibliothèques qui implémentent cette interface : on parle des bibliothèques MPI. Cetraines sont open-source (OpenMPI, MPICH2, LAM-MPI, ...) et d'autres sont propriétaires. Mais comme l'interface est standardisée, elles sont en principe interchangeables.

Une bibliothèque MPI offre au programmeur un certain nombre de fonctions qui facilitent grandement la mise au point de programmes distribués qui doivent communiquer par le réseau. Au coeur d'une bibliothèque MPI se trouvent des fonctions qui servent à faire transiter des *message*, c'est-à-dire des paquets de données plus ou moins arbitraire, d'une machine à une autre.

Pourquoi utiliser des bibliothèques spéciales pour transférer des messages d'une machine à une autre, au lieu d'utiliser les fonctionnalités offertes par les systèmes d'exploitations (ouverture de connections TCP, etc.) ? Il y a plusieurs raisons à cela :

- Sur des machines de HPC, ou même sur des *clusters* haut-de-gamme, le réseau est parfois « exotique » (Infiniband, Omnipath, réseaux spéciaux sur machines IBM ou Fujitsu). L'utilisation des bibliothèques MPI permet de ne pas avoir à se préoccuper de la nature précise de l'infrastructure matérielle dans les applications.
- Les bibliothèques MPI simplifient (entre autre) le problème du *nommage* des machines : elles sont numérotées de 0 à $n-1$. Plus besoin de gérer leur URL ou leur adresse IP.
- Les bibliothèques MPI offrent des fonctionnalités que les OS n'offrent pas, en particulier des fonctions de communications *collectives*, auxquelles participent non pas deux, mais toutes les machines du groupe.
- Sur les machines conçues pour le calcul haute-performance, et équipées de réseaux spéciaux, des bibliothèques MPI adaptées peuvent exploiter le matériel au mieux de ses possibilités sans que l'application soit modifiée, ni même recompilée.

Le standard MPI est né du fait que ces bibliothèques étaient indispensable pour pouvoir programmer des applications parallèles, mais plusieurs bibliothèques incompatibles étaient généralement offertes par les fabriquants de machines de calcul scientifique. Pour augmenter la portabilité des applications, ces bibliothèques ont été unifiées au début des années 1990, pour aboutir à la première version de la spécification MPI.

Les objectifs de la spécifications MPI étaient d'obtenir un système *i*) simple d'utilisation (on verra que 6 fonctions permettent d'écrire des programmes parallèles), *ii*) portable d'une machine à l'autre, *iii*) capable d'aboutir à des applications parallèles de haute-performance.

Ce document ne décrit pas toutes les fonctionnalités de MPI. Sont notamment absentes : les IO parallèles et les « *one-sided communications* ».

2.1 Généralités

Ce document décrit l'interface C des bibliothèques MPI. Il existe d'autres interfaces, plus ou moins standardisées, pour d'autres langages. En plus de fonctions qui peuvent être appelées depuis les applications, les bibliothèques MPI fournissent un *environnement d'exécution*, et notamment un moyen standardisé de *lancer* une application parallèle : le programme `mpiexec`. Celui-ci peut utiliser des moyens sophistiqués pour lancer l'application parallèle rapidement sur un grand nombre de machines (déploiement en arbre, notamment).

Toutes les fonctions sont définies dans le fichier d'en-tête `mpi.h`. Toutes les fonctions et toutes les constantes ont des noms qui commencent par `MPI_`. Presque toutes les fonctions renvoient un code d'erreur. Si tout va bien, celui-ci est égal à la constante `MPI_SUCCESS`. En cas d'erreur, sa valeur est différente, mais n'est pas spécifiée. De toute façon, ce n'est pas la peine de vérifier ce code de retour : le comportement par défaut de MPI consiste à faire stopper net l'application parallèle en cas d'erreur.

Les fonctions MPI ne retournent (en principe) rien d'autre qu'un code d'erreur. Ceci signifie que lorsqu'elles doivent retourner des valeurs, il faut leur passer un pointeur vers une zone de la mémoire où elles peuvent écrire ce qu'elles ont besoin de retourner à l'application.

Démarrage et terminaison Un programme doit commencer par appeler la fonction `MPI_Init`, et finir par appeler `MPI_Finalize`. Voici un programme MPI minimal :

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

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

    /* main program */

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

On peut donner à `MPI_Init` les *adresses* de `argc` et `argv`, ce qui lui permet non seulement de lire les arguments spécifiques à MPI, mais en plus de les retirer pour que l'application elle-même n'ait pas à les gérer. On peut passer `NULL` à la place de ces deux arguments.

Il est interdit d'appeler une autre fonction MPI avant l'initialisation et après la finalisation.

Communicateur et rang Une fois que l'environnement d'exécution qui accompagne une bibliothèque MPI a lancé plusieurs processus (éventuellement sur plusieurs machines), chacun de ces processus se voit attribuer un numéro, qui est son *rang*.

Ces numéros servent « d'adresse » au sein des opérations MPI : quand on envoie un message à quelqu'un, on indique le rang de la destination.

Dans toutes les opérations MPI on doit préciser un *communicateur*, c'est-à-dire un « contexte de communication ». Un message envoyé dans un communicateur ne peut pas être reçu dans un autre. Les opérations collectives ont lieu à l'intérieur d'un communicateur, et mettent en relation tous les processus de ce communicateur. Lors du lancement d'une application MPI, il y a un seul communicateur qui contient tout le monde, et qui s'appelle `MPI_COMM_WORLD`.

Le rang d'un processus est en fait son rang *au sein d'un communicateur donné* (on verra plus tard comment partitionner un communicateur en plusieurs morceaux). La « numérotation » des processus à laquelle il a été fait allusion ci-dessus correspond en fait aux rangs dans `MPI_COMM_WORLD`.

Deux fonctions particulièrement utiles permettent d'accéder au rang du processus courant, et à la taille d'un communicateur :

```c
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Ces deux fonctions, qui prennent un communicateur en argument (le plus souvent `MPI_COMM_WORLD`) écrivent à l'adresse spécifiée par le deuxième argument la taille du communivateur et le rang du processus actuel dans le communicateur, respectivement.

2.2 Opérations point-à-point

Les opérations dites « point-à-point » sont celles qui consistent à transférer un message d'un processus A à un processus B. Si ces deux processus s'exécutent sur la même machine, la plupart des bibliothèques MPI effectuent alors une simple copie en RAM. Si les deux processus sont sur deux machines différentes, alors le message est acheminé par le réseau.

Les deux fonctions de base sont :

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

et

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

qui servent à envoyer et recevoir des messages, respectivement.

Description du message Un « message » est en fait un *tableau* de valeurs d'un type donné, dont il faut préciser le type et la taille. Dans les deux fonctions ci-dessus, les trois premiers arguments indiquent à la fonction les données qui constituent le message :

`buf` est un pointeur sur le début du tableau

`count` indique le nombre d'éléments du tableau

`datatype` indique le type des entrées du tableau

La fonction d'envoi lit le tableau à envoyer à l'endroit indiqué par `buf`, tandis que la fonction de réception écrit à l'endroit indiqué par `buf` le tableau reçu. La fonction de réception n'alloue pas de mémoire pour stocker le message reçu, c'est de la responsabilité du programmeur : il faut fournir un pointeur vers une zone pré-allouée et suffisamment grande. Lors de la réception, `count` doit indiquer la taille de la zone *allouée*, et pas forcément la taille précise du tableau qui doit être reçu. La fonction de réception peut recevoir un tableau plus petit ; et cela lui permet de déclencher une erreur si on tente de lui envoyer un tableau plus grand que le *buffer* d'arrivée.

Il faut donc potentiellement connaître à l'avance une *borne supérieure* sur la taille des messages qu'on veut recevoir. Comment faire quand on ne connaît pas cette taille à l'avance ? Il suffit d'envoyer d'abord *un* entier, qui indique la taille du tableau.

Voici une liste non-exhaustive des types possibles :

type MPI	type C
`MPI_CHAR`	`char`
`MPI_SHORT`	`short int`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_LONG_LONG_INT`	`long long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_INT8_T`	`int8_t`
`MPI_INT16_T`	`int16_t`
`MPI_INT32_T`	`int32_t`
`MPI_INT64_T`	`int64_t`
`MPI_BYTE`	

L'intérêt d'indiquer à la bibliothèque MPI le type des données transmises est que ces données n'ont pas forcément la même représentation en mémoire sur toutes les architectures, et cela permet à la bibliothèque de faire d'éventuelles conversions de manière transparente. Le cas le plus classique est le cas d'un entier de 32 bits : l'octet de poids faible est-il stocké à l'adresse x ou bien à l'adresse $x + 3$?

Le type `MPI_BYTE` décrit, lui, une valeur d'un octet que la bibliothèque MPI ne cherchera pas à interpréter ni à convertir.

Enveloppe du message En plus du message à proprement parler, des « meta-données » accompagnent le message. Elles permettent de trier les messages et de sélectionner ceux qu'on souhaite recevoir.

`dest` est le rang du processus destinataire du message.

`source` est le rang de l'expéditeur.

`tag` est un entier fixé par l'application et qui sert à distinguer différents types de messages.

`comm` est le communicateur sur lequel le message est envoyé.

Lors de l'envoi, la source est déterminée automatiquement (c'est le rang du processus qui envoie), mais la destination doit être spécifiée. *A contrario*, lors de la réception, il faut indiquer de quel expéditeur on veut recevoir un message (et la destination est bien sûr le rang du processus qui exécute la fonction de réception).

Lors de l'envoi, un « *tag* » (« étiquette », littéralement) doit être assigné au message. C'est un entier qui peut prendre les valeurs de 0 à 32767. Lors de la réception, on doit indiquer un *tag*, et on ne reçoit que les message qui portent le bon *tag*. Cela permet de distinguer différents types de messages.

Il faut noter que lors de la réception, on peut spécifier la valeur spéciale `MPI_ANY_SOURCE` pour dire qu'on veut bien recevoir des messages de la part de n'importe qui. On peut également indiquer le *tag* spécial `MPI_ANY_TAG` pour dire qu'on veut bien recevoir des messages portant n'importe quel *tag*.

Il est possible de s'envoyer un message à soi-même, mais attention au risque de *deadlock* !

Status lors de la réception Si on utilise `MPI_ANY_SOURCE` ou bien `MPI_ANY_TAG`, alors on peut recevoir un message... dont on ne connait pas le tag et/ou pas l'expéditeur. Et dans tous les cas, on ne connaît pas forcément à l'avance la taille précise du message reçu (on sait qu'il est plus petit que `count`).

Ces informations sont cependant retournées par la fonction de réception via l'argument `status`, qui doit indiquer l'adresse en mémoire d'une variable de type `MPI_Status`. Cette variable est un `struct`, et elle contient (entre autre) deux champs `MPI_SOURCE` et `MPI_TAG` qui contiennent ces informations pour le message reçu. On peut aussi lire la taille exacte du message dans le statut (anglais : « status », français : « statut ») avec la fonction :

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

Si on a pas besoin du status, on peut passer la valeur spéciale `MPI_STATUS_IGNORE`, et la bibliothèque MPI n'indiquera pas les informations en question.

2.2.1 Modes de communication

Les deux fonctions décrites ci-dessus sont *bloquantes*. De manière assez logique, un appel à la fonction de réception ne « rend pas la main » tant que le message n'a pas été reçu et qu'il n'est pas exploitable par la suite du programme.

Du côté de la fonction d'envoi les choses sont un peu plus subtiles. Dans tous les cas de figure, lorsque la fonction d'envoi rend la main, alors le contenu du tableau `buf` peut être réutilisé et modifié par le programme sans que cela compromette le bon déroulement de l'envoi. Par contre, il n'est pas garanti que le message ait été reçu (ni même « vraiment envoyé » !) lorsque la fonction d'envoi rend la main.

Il y a en fait non pas une, mais *quatre* fonctions d'envoi, qui diffèrent dans la manière dont elles se synchronisent avec le destinataire.

`MPI_Bsend` Cette fonction envoie un message en mode « *buffered* ». Le message est d'abord copié dans un « *buffer* » (littéralement : zone tampon) sous le contrôle de la bibliothèque MPI, puis la main est rendue au programme. La bibliothèque se charge d'envoyer le message toute seule, parallèlement à l'exécution du reste du programme. Cette fonction termine correctement même si le destinataire n'a pas encore lancé `MPI_Recv`. Par contre, elle peut renvoyer une erreur s'il n'y a pas assez d'espace alloué au stockage en RAM des messages « en transit ».

`MPI_Ssend` Cette fonction envoie un message en mode « *synchronous* ». Cette fonction peut être lancée même si le destinataire n'a pas encore lancé `MPI_Recv`. Cependant, la fonction ne rend la main que lorsque le destinataire a commencé à recevoir le message.

`MPI_Send` Cette fonction envoie un message en mode « par défaut ». Cette fonction peut être lancée même si le destinataire n'a pas encore lancé `MPI_Recv`. La bibliothèque MPI décide toute seule si elle rend la main avant l'envoi du message (comme dans le mode *buffered*) ou bien si elle attend la disponibilité du destinataire (comme dans le mode *synchronous*). Par exemple, la fonction peut choisir de ne pas dupliquer un énorme message pour économiser la mémoire, mais peut choisir de faire des copies des petits messages pour rendre la main plus vite.

`MPI_Rsend` Cette fonction envoie un message en mode « *ready* ». Contrairement aux trois autres, cette fonction ne peut être lancée que si le destinataire a déjà lancé `MPI_Recv`. Une erreur est renvoyée dans le cas contraire. À ceci près, la fonction est identique à `MPI_Send`.

On pourrait se demander à quoi sert à fonction `MPI_Rsend`, mis à part à poser des problèmes au programmeur. En fait, c'est une question de performances. Pour comprendre sa raison d'être il faut imaginer comment est implémenté le mode *synchronous* :

1. L'expéditeur envoie un message « requête d'envoi de message ».
2. Le destinataire stocke cette requête.
3. Lors de l'exécution de `MPI_Recv`, le destinataire renvoie un message « permission d'envoi de message ».
4. L'expéditeur envoie le message, et rend la main.

En appelant `MPI_Rsend`, le programmeur donne une information supplémentaire à la bibliothèque : il lui fait la promesse que le destinataire est déjà prêt. Du coup, les trois premières étapes du protocole précédent peuvent être supprimées, et l'envoi peut terminer plus vite. Mais il est en pratique difficile d'obtenir la garantie que le destinataire est « déjà prêt » et cette fonction est difficile à utiliser.

Les bibliothèques MPI garantissent qu'entre un expéditeur et une destination donnée, les messages sont livrés dans l'ordre où ils sont envoyés.

Même si c'est paradoxal, `MPI_Recv` peut terminer sur le destinataire *avant* que l'opération d'envoi correspondante ne termine sur l'expéditeur.

2.2.2 Risques de *deadlock*

L'usage de `MPI_Recv` d'un côté, tout comme de `MPI_Ssend` ou de `MPI_Send` de l'autre, peut créer des situations de *deadlock* (« interblocage mutuel »). Considérons quelques exemples.

Processus de rang 0 :
```
MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, MPI_STATUS_IGNORE);
```
Processus de rang 1 :
```
MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, MPI_STATUS_IGNORE);
MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);
```

Ce premier exemple fonctionne correctement. Il marcherait même si on utilisait des `MPI_Ssend`. Si c'était le cas, le processus 0 serait bloqué par l'envoi tant que 1 n'a pas commencé à recevoir. Ensuite, 0 serait bloqué en attendant la réception, mais une fois la réception de 1 terminée, l'envoi de 1 vers 0 commencerait, et débloquerait 0.

Processus de rang 0 :
```
MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, MPI_STATUS_IGNORE);
MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
```
Processus de rang 1 :
```
MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, MPI_STATUS_IGNORE);
MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);
```

Ce deuxième exemple ne fonctionne pas correctement car les deux processus vont attendre mutuellement de recevoir un message en provenance de l'autre : ils sont en *deadlock*.

Processus de rang 0 :
```
MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, MPI_STATUS_IGNORE);
```
Processus de rang 1 :
```
MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);
MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, MPI_STATUS_IGNORE);
```

Ce troisième exemple *peut* ne pas marcher. Avec des `MPI_Ssend` ça ne marche pas : les deux processus bloqueraient sur l'envoi en attendant mutuellement que l'autre soit prêt à recevoir. Par contre, avec des `MPI_Bsend`, ça marcherait, car les deux processus reprendraient la main alors que les messages seraient toujours en attente d'envoi. Ils pourraient alors tous les deux entamer la réception. Avec `MPI_Send`, ça va donc dépendre des choix de la bibliothèque MPI.

Ce dernier exemple est le plus subtil, car le *deadlock* ne se produit pas forcément tout le temps (ça dépend des choix de la bibliothèque).

On pourrait être tenté d'utiliser `MPI_Bsend` systématiquement pour éviter les *deadlocks* de ce type, mais on prend alors le risque d'avoir des erreurs par manque de mémoire. En sens inverse, un programme qui fonctionne correctement en utilisant uniquement `MPI_Ssend` est un programme « sûr ».

2.2.3 Échange de messages

Il est fréquent de voir deux processus MPI s'échanger des messages (A envoie à B et B envoie à A) — il s'agit des trois exemples de *deadlock* potentiel discutés ci-dessus.

Du coup, comme c'est a) fréquent et b) source d'erreurs, c'est prévu par la spécification MPI :

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status);
```

Cette fonction réalise à la fois (et simultanément) l'envoi et la réception. Le message lu dans `sendbuf` est envoyé au processus de rang `dest`, et le message reçu en provenance de `source` est placé dans `recvbuf`. Le `status` est renseigné pour le message reçu, sauf si c'est `MPI_STATUS_IGNORE`. Il est possible d'échanger des messages de tailles et de types différents.

Les deux *buffer* doivent être distincts et ne pas se chevaucher.

Cette contrainte est parfois un problème ; c'est notamment le cas lorsque deux processus veulent littéralement s'échanger le contenu de deux variables. Il existe donc une autre fonction *ad hoc* faite exprès pour ça :

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
                         int dest, int sendtag, int source, int recvtag,
                         MPI_Comm comm, MPI_Status *status);
```

Une fois que cette fonction est terminée, le contenu initial de `buf` a été envoyé au processus `dest` et le message reçu depuis `source` est écrit dans `buf` (en remplaçant sa valeur initiale).

Ces deux fonctions peuvent interagir avec des envois et des réceptions normaux. Elles peuvent également utiliser les « jokers » `MPI_ANY_SOURCE` et `MPI_ANY_TAG` pour `source` et `recvtag` respectivement.

2.2.4 Sondage de la file de messages

Le fait que les messages soient livrés dans l'ordre où ils sont émis implique que chaque processus MPI possède une *file* de messages en attente de réception. Il est possible d'examiner le contenu de cette file d'attente, notamment pour obtenir des informations sur un message qu'on *va* recevoir, mais *avant* la réception elle-même. En particulier, il est possible de déterminer *s'il y a* un message qui peut être reçu à l'instant t. Un autre intérêt principal consiste à connaître la taille du message qui va arriver, pour allouer un *buffer* de réception de la bonne taille.

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

La fonction `MPI_Probe` a presque les mêmes arguments que `MPI_Recv`, à l'exception du *buffer* d'arrivée. Elle ne rend la main que lorsqu'un message qui satisfait les critères est présent dans la file d'attente (elle peut donc bloquer jusqu'à ce qu'un envoi ait lieu). Alors, le `status` est renseigné. On peut passer la valeur spéciale `MPI_STATUS_IGNORE`, mais en l'occurrence c'est idiot car l'intérêt principal consiste à pouvoir observer le `status` avant la

La fonction `MPI_Iprobe`, elle, est non-bloquante (« Immediate ») et rend la main instantanément. Le le `flag` reçoit O ou 1 selon qu'un message qui satisfait les critères est disponible ou pas. Si oui, le `status` est également renseigné. Elle permet en particulier de tester si un message est arrivé ou pas, de manière non-bloquante.

2.3 Opérations non-bloquantes

Il y a des variantes *non-bloquantes* (*immédiates*) des fonctions d'envoi et de réception. Ces fonctions rendent la main dès que possible, avec l'idée que l'opération désirée (envoi ou réception) va avoir lieu en tâche de fond. Concrètement, ceci est mis en oeuvre par la bibliothèque MPI soit via des threads, soit en utilisant du *hardware* adapté.

L'intérêt principal, c'est que cela permet de mettre en oeuvre le recouvrement des communications par des calculs (c.a.d. qu'on continue d'occuper le processeur à faire des calculs, alors que les communications ont lieu en même temps). Sur des machines avec des interfaces réseau haut de gamme, le surcoût engendré par ces communications « en tâche de fond » peut être quasi-nul.

Les équivalents « non-bloquants » des fonctions déjà vues sont les suivants :

```
int MPI_Isend (const void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Issend(const void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Ibsend(const void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irsend(const void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

et

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request);
```

Ces fonctions peuvent (et doivent, en principe) rendre la main avant que l'opération en question soit terminée. En particulier, ce sont toutes des opérations *locales* : leur durée d'exécution ne dépend pas de l'état des autres processeurs.

Elles *initient* une opération de communication, mais ne garantissent pas sa terminaison. L'opération de communication se déroule de manière asynchrone, « en tâche de fond », parallèlement à l'exécution du programme qui continue. Il faut fournir à toutes ces fonctions un pointeur vers un `MPI_request`, qui est un objet opaque contenant des informations sur l'opération asynchrone en cours d'exécution.

Le caractère non-bloquant de la réception signifie la chose suivante : l'appel à `MPI_Irecv` indique qu'on autorise la bibliothèque MPI à écrire un éventuel message dans le *buffer* d'arrivée. La fonction ne bloque pas jusqu'à ce qu'un message arrive, et donc il ne faut pas lire le contenu de `buf` avant d'avoir obtenu la confirmation que la réception d'un message est bien achevée.

Le caractère non-bloquant de l'envoi signifie que l'appel à `MPI_Isend` (et autres) autorise la bibliothèque MPI à lire le contenu du buffer d'envoi. La fonction ne bloque pas jusqu'à ce que le contenu de `buf` ait été entièrement lu, donc il ne faut pas le modifier avant d'avoir obtenu la confirmation que l'opération est terminée.

Complétion des requêtes asynchrones Une fois qu'une opération de communication asynchrone est lancée, il faut vérifier qu'elle s'est terminée. Deux fonctions permettent de le faire :

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

Ces deux fonctions prennent un (pointeur vers un) `MPI_Request` en argument. Celui-ci doit avoir été initialisé par le lancement d'une opération asynchrone. La fonction `MPI_Wait` rend la main seulement une fois que l'opération de communication en question est terminée. L'objet `MPI_Status` est renseigné (sauf si c'est la valeur spéciale `MPI_STATUS_IGNORE`, auquel cas il est ignoré). Elle est donc bloquante et « non-locale ».

A contrario `MPI_Test` est non-bloquante et locale. Elle écrit une valeur non-nulle dans `flag` si et seulement si l'opération est terminée, sinon elle écrit 0 dans `flag`. Si l'opération est terminée, le status est renseigné (sauf si c'est `MPI_STATUS_IGNORE`).

Lorsqu'une réception est terminée, alors le message est disponible dans le *buffer* d'arrivée. Lorsqu'un envoi est terminé, alors dans tous les cas le *buffer* d'envoi peut être modifié. La terminaison d'un envoi a la même signification que lorsque la fonction bloquante correspondante termine : pour `MPI_Issend` (mode *synchronous*), cela signifie que la réception a commencé ; `MPI_Ibsend` (mode *buffered*) cela signifie que la copie du message vers un autre *buffer* (interne à MPI) est terminé, etc.

Lorsque `MPI_Wait` rend la main, les ressources associées à la requête asynchrone sont libérées, et l'argument `request` est fixé à la valeur spéciale `MPI_REQUEST_NULL`. C'est également le cas lorsque `MPI_Test` rend la main avec `flag` $\neq 0$.

Il **faut** exécuter soit `MPI_Wait` soit `MPI_Test` (en obtenant `flag` $\neq 0$) pour *chaque* opération asynchrone lancée. Sinon, les ressources associées aux requêtes ne sont jamais libérées d'une part, et `MPI_Finalize` déclenchera une erreur si des opérations sont encore « en cours » à la fin du programme d'autre part.

Des messages émis en mode bloquant peuvent être reçus en mode non-bloquant et vice-versa.

Annulation des requêtes asynchrones Il est possible de demander l'annulation d'une requête asynchrone en cours d'exécution.

```
int MPI_Cancel(MPI_Request *request);
```

Cette opération est locale et termine immédiatement. La requête est « marquée » comme étant « en demande d'annulation ». En aucun cas une requête peut n'être que partiellement exécutée : c'est tout-ou-rien. Alors, de deux choses l'une :

— Soit tout se passe comme si la requête n'avait jamais été lancée (si c'était un envoi, alors le message ne sera pas reçu ; si c'est une réception, le *buffer* d'arrivée ne sera pas altéré, etc.).
— Soit la requête se déroule normalement.

Dans les deux cas, il est toujours nécessaire de « nettoyer » la requête avec `MPI_Wait` ou `MPI_Test`. Par contre, `MPI_Wait` termine forcément sur une requête marquée pour annulation, quoi que fassent les autres processus (c.a.d. la fonction se comporte comme une opération locale).

On peut tester si une requête a été effectivement annulée (c.a.d. n'a pas eu lieu et n'aura pas lieu) avec :

```
int MPI_Test_cancelled(const MPI_Status *status, int *flag)
```

Attention, cette procédure n'est pas forcément locale (elle peut engendrer une communication pour déterminer son résultat).

2.4 Opérations collectives

L'un des intérêts principaux des bibliothèques MPI réside dans la disponibilité d'opérations *collectives*. Ces opérations sont des opérations de groupe, dans lesquelles tous les processus d'un même communicateur interagissent.

Barrière

```
int MPI_Barrier(MPI_Comm comm);
```

C'est un mécanisme de synchronisation. La fonction ne rend la main sur un processus que lorsque tous les autres processus du communicateur `comm` ont eux aussi appelé la fonction.

2.4.1 Opérations de transfert de données

Toutes les opérations collectives ont une particularité : *tous* les participants doivent appeler la fonction collective pour que son action ait lieu.

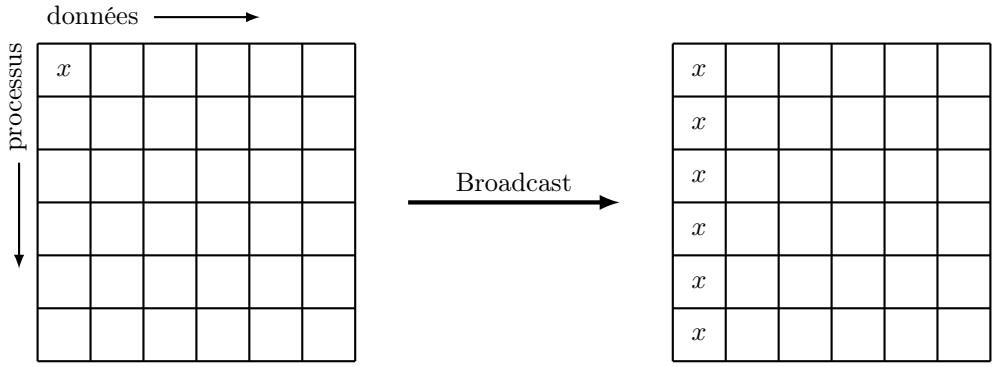
« *Broadcast* »

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

Cette fonction permet le « *broadcast* » (littéralement « diffusion »). Un tableau est copié à l'identique de la mémoire d'un processus particulier, celui dont le rang est `root` dans le communicateur `comm`, vers la mémoire de tous les autres processus du communicateur.

Les arguments `count`, `datatype`, `comm` et `root` doivent être identiques sur tous les processus.

Lorsque la fonction rend la main, alors le tableau dont l'adresse est donnée par `buffer` contient la même chose que celui sur `root`. La fonction n'alloue pas de mémoire pour stocker le tableau reçu (c'est la responsabilité du programmeur). Notez que la taille du tableau doit être connue à l'avance même chez ceux qui reçoivent.



Dispersion et rassemblement

```
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
           int root, MPI_Comm comm);
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm);
```

L'opération « *Gather* » (littéralement « rassembler ») rassemble sur le processus `root` des données éparpillées sur tous les autres processus. Chaque processus transfère vers `root` les `sendcount` éléments de son tableau `sendbuf`.

Sur le processus `root`, toutes les données de tous les processus atterrissent dans le tableau `recvbuf`, qui doit donc être de taille $\texttt{sendcount} \cdot P$, où P désigne le nombre de processus du communicateur `comm`. Si $\texttt{sendbuf}_i$ désigne le tableau du i-ème processus, alor le résultat de l'opération est :

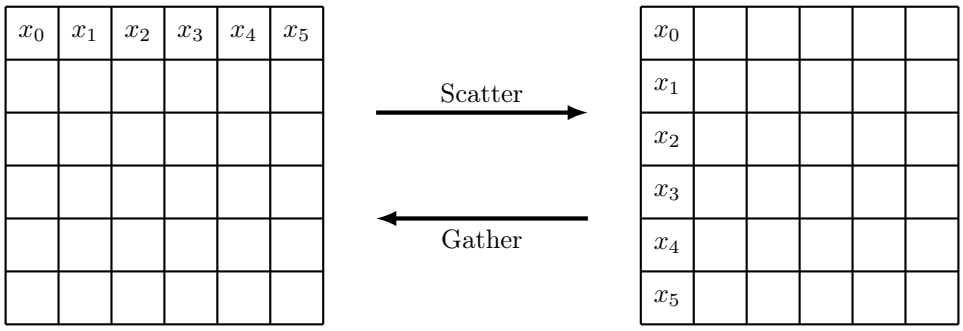
$$\texttt{recvbuf}[i \cdot \texttt{sendcount} : (i+1) \cdot \texttt{sendcount}] \leftarrow \texttt{sendbuf}_i, \qquad 0 \leq i < P$$

Les arguments `sendcount` et `sendtype` doivent partout être égaux aux valeurs de `recvcount` et `recvtype` sur `root`. Cela signifie que sur `root`, `recvcount` indique le nombre d'élément reçus depuis chaque autre processus, et pas le nombre total d'éléments reçus.

Il existe deux fonctions `MPI_Gatherv` et `MPI_Scatterv` qui permettent de transférer un nombre d'éléments qui n'est pas le même pour chaque processus du communicateur.

Les arguments décrivant le tableau de destination sont ignorés par la fonction, excepté sur le processus `root` (donc ailleurs on peut mettre n'importe quoi).

Un petit détail : même sur `root`, il faut donner un `sendbuf` valable, dont le contenu sera copié (en RAM) dans `recvbuf`. Comme ceci n'est pas forcément souhaitable, on peut passer à la place de `sendbuf` la valeur spéciale `MPI_IN_PLACE`. Dans ce cas, aucun transfert n'a lieu de `root` vers lui-même. Ceci est adapté au cas où les bonnes données sont déjà au bon endroit dans `recvbuf`.



L'opération « *Scatter* » (« dispersion ») est l'exact opposé. Les éléments d'un gros tableau sur `root` sont répartis sur les P processus du communicateur. Les arguments ont les mêmes significations. Cette fois cependant, la description du tableau d'envoi est ignorée sur tous les autres processus que `root`.

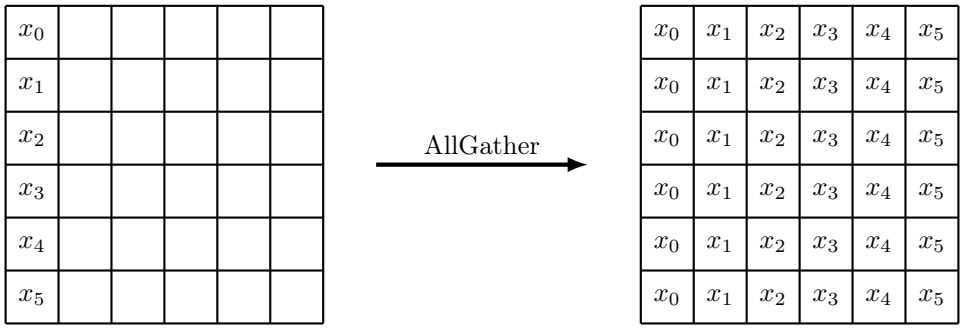
Sur `root`, il est possible de passer `MPI_IN_PLACE` à la place de `recvbuf`. Dans ce cas-là, aucune donnée n'est transférée de `root` vers lui-même.

All-Gather

```
int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```

La fonction « All-gather » est un équivalent de `MPI_Gather`, à ceci près que le résultat n'arrive pas uniquement sur un processus racine, mais sur tous les processus du communicateur. Par conséquent, les `sendcount` et `recvcount` doivent être égaux partout, et partout égaux entre eux (on se demande pourquoi il faut donner deux fois...).

Si *tous* les processus utilisent la valeur spéciale `MPI_IN_PLACE`, alors les tableaux d'envoi sont ignorés. Les données envoyées par le i-ème processus à tous les autres sont alors lues dans le tableau de destination, à l'endroit où auraient atterri celles qu'ils se serait envoyé lui-même.



Il existe une fonction `MPI_Allgatherv` qui permett de transférer un nombre d'éléments qui n'est pas le même pour chaque processus du communicateur.

Si on n'avait pas `MPI_Bcast`, on pourrait obtenir le même effet en utilisant une combinaison des autres fonctions : `MPI_Bcast` est équivalent à `MPI_Scatter` suivi de `MPI_Allgather`.

De même `MPI_Allgather` est équivalent à `MPI_Gather` suivi de `MPI_Bcast`.

All-to-All

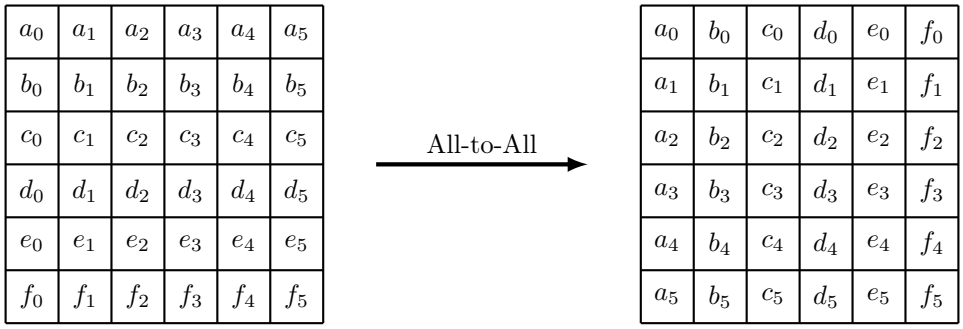
```
int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm);
```

Cette fonction est une variante généralisée de la précédente. Dans `MPI_Allgather`, le processus de rang i envoie ses données à tous les autres processus : il leur envoie la même chose à chacun. Dans `MPI_Alltoall`, le processus i envoie la j-ème « portion » de son tableau au processus de rang j. C'est un peu comme si tous les processus dispersaient leurs tableaux en même temps.

Les `sendcount` et `recvcount` doivent être égaux partout, et partout égaux entre eux (on se demande donc pourquoi il faut les donner deux fois...).

Les deux *buffers* doivent être disjoints et ne pas se chevaucher. Mais si *tous* les processus passent la valeur spéciale `MPI_IN_PLACE` à la place de `sendbuf`, alors les données à envoyer sont lues dans `recvbuf`, et elles sont écrasées par les données reçues.

Il existe aussi `MPI_Alltoallv` et `MPI_Alltoallw` qui offrent plus de flexibilité.



2.4.2 Opérations de « réduction »

Les opérations de réduction consistent à appliquer une opération (somme, produit, min, max, etc.) sur des données qui sont réparties sur les différents processus d'un communicateur. Il faut à chaque fois préciser l'opération. Il est possible de définir ses propres opérations, mais les bibliothèques MPI en fournissent déjà un certain nombre : `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, dont les significations sont transparentes.

Reduce

```
int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```

Cette fonction combine, élément par élément, les différents tableaux `sendbuf` de tous les processus. Le résultat se trouve dans le tableau `recvbuf` sur le processus `root`. Plus précisément, il se passe :

$$\texttt{recvbuf}[j] \leftarrow \texttt{sendbuf}_0[j] \oplus \texttt{sendbuf}_1[j] \oplus \cdots \oplus \texttt{sendbuf}_{P-1}[j], \qquad 0 \leq j < \texttt{count}$$

Comme d'habitude, P désigne la taille du communicateur, et $\oplus$ désigne l'opération sélectionnée. La valeur de `recvbuf` est ignorée ailleurs que sur `root` (donc on peut mettre n'importe quoi).

Sur `root`, il est possible de passer `MPI_IN_PLACE` à la place de `sendbuf`. Dans ce cas-là, la « contribution » de `root` est prise dans `recvbuf`.

All-Reduce Il y a deux variantes de `MPI_Reduce` qui diffèrent par le placement des résultats au sein du communicateur. Ces deux variantes sont simulables à partir des autres opérations, mais existent séparément pour des questions de performance. On rappelle que dans `MPI_Reduce` le résultat des calculs est rapatrié sur un processus spécifique.

```
int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Cette fonction se comporte comme `MPI_Reduce`, mais au lieu que les résultats soient concentrés sur une « racine », ils sont « broadcastés » sur l'ensemble des processus. C'est exactement comme si on faisait :

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);
MPI_Bcast(recvbuf, count, datatype, root, comm);
```

Il y a une variante « en place » : si *tous* les processus passent `MPI_IN_PLACE` à la place de `sendbuf`, alors les données sont lues dans `recvbuf`.

Reduce-Scatter

```
int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf, int recvcount,
                             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Celle-ci se comporte comme `MPI_Reduce`, mais les résultats sont dispersés sur l'ensemble des processus. C'est exactement comme si on faisait :

```
MPI_Reduce(sendbuf, tempbuf, count, datatype, op, root, comm);
MPI_Scatter(tempbuf, count, datatype, recvbuf, count, datatype, root, comm);
```

Il y a aussi une variante *in-place*, qui fonctionne comme la précédente.

2.5 Partitionnement des communicateurs

Il est parfois souhaitable de pouvoir effectuer des opérations collectives sur une partie seulement de l'ensemble des processus MPI. Lorsqu'une application MPI démarre, un seul communicateur est disponible, `MPI_COMM_WORLD` qui contient l'ensemble de tous les processus.

Pour cela, un mécanisme permet de partitionner un communicateurs en plusieurs sous-communicateurs.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```

Cette fonction est une opération collective (donc *tous* les processus du communicateur `comm` doivent l'appeler). Chaque processus peut spécifier une valeur arbitraire de `color` et de `key`. Chaque processus récupère un nouveau communicateur `newcomm`, dont il fait partie.

Tous les processus qui ont indiqué la même `color` appartiennent au même (sous-)communicateur `newcomm`. Au sein de `newcomm`, les processus ont des rangs $(0, 1, 2, \dots)$; ils sont numérotés par valeur de `key` croissante (leur rang dans l'ancien communicateur `comm` sert à départager les cas d'égalité).

Le même communicateur de départ peut être découpé plusieurs fois de plusieurs manière différentes. Un exemple typique est celui où les processus sont disposés sur une grille en deux dimensions. Il est alors commun de former des sous-communicateurs pour les lignes et les colonnes.