Introduction à MPI

Plan

MPI : Message Passing Interface

Premiers programmes/premières fonctions

Les communications point à point

MPI = Message Passing Interface ;

- Définition
  - bibliothèque, plus exactement API de haut niveau, pour la programmation parallèle obéissant au paradigme de l'échange de message ;
  - Interfaces C/C++ et Fortran disponibles

- Contexte
  - MPI est adaptéà la programmation parallèle distribuée ;
  - MPI est née de la collaboration d'universitaires et d'entreprises.
  - Très répandu dans le monde du calcul intensif.

Fonctionnalités MPI

- La bibliothèque MPI permet de gérer :
  1. l'environnement d'exécution ;
  2. les communications point à point ;
  3. les communications collectives ;
  4. les groupes de processus ;
  5. les topologies de processus ;
- MPI permet en outre de gérer :
  1. les communications unidirectionnelles ;
  2. la création dynamique de processus ;
  3. le multithreading ;
  4. les entrées/sorties parallèles (MPI/IO).

beaucoup de fonctionnalités (environ 120 fonctions pour MPI 1, plus de 200 pour MPI 2).

Néanmoins, la maîtrise des communications point à point et collectives est suffisante pour paralléliser un code de façon efficace ;

Pourquoi utiliser MPI

- MPI est avant tout une interface ;
- MPI est présent sur tout type d'architecture parallèle ;
- MPI supporte les parallélismes modérés et massifs ;
- Les constructeurs de machines et/ou de réseaux rapides fournissent des bibliothèques MPI optimisées pour leurs plate-formes ;
- MPI est également disponible pour la plupart des machines du marché en open source :
  - LAM-MPI : http ://www.lam-mpi.org ;
  - MPICH : http ://www-unix.mcs.anl.gov/mpi/mpich ;
  - OPEN-MPI

Plus précisément

MPI : Message Passing Interface

Premiers programmes/premières fonctions

Premier programme MPI

```
#include <stdio.h>
#include <mpi.h>
int main (argc, argv)
int argc ; char *argv[] ;
int rank, size ;

MPI_Init (&argc, &argv) ; /* starts MPI */
printf( "Hello world \n" ) ;
MPI_Finalize() ;
return 0 ;
```

Tous les appels de la bibliothèque MPI commencent par le préfixe `MPI_` ;
Aucun appel à MPI ne peut avoir lieu avant l'appel à `MPI_Init()` ;
Aucun appel à MPI ne peut avoir lieu après l'appel à `MPI_Finalize()` ;

Compilation et exécution

```
% mpicc essai-1.c -o essai
% mpirun -np 4 essai
Hello world
Hello world
Hello world
Hello world
```

[h]

- la procédure `mpirun` crée 4 processus, chaque processus exécutant le programme `essai` ;
- les façns d'exécuter MPI ne sont pas portables ! Elles dépendent des machines, mais elles sécifient au moins le nombre de processus et l'ensemble des processeurs sur lesquels MPI va s'exécuter. Par Exemple :

  MPICH : mpirun -np 4 -machinefile hosts essai
- les façons de compiler dépendent elles aussi des implémentations allant jusqu'à utiliser un script de compilation :

  MPICH :mpicc essai.c

Ensemble des processus : `MPI_COMM_WORLD`

- Lors de l'initiatialisation de l'environnement, MPI regroupe tous les processus créés sous le communicateur prédéfini `MPI_COMM_WORLD`
- communicateur = ensemble de processus + contexte de communications
- un communicateur est de type `MPI_Comm` ; (ici : il est conseillé d'utiliser `MPI_COMM_WORLD`

Nombre total de processus : `MPI_Comm_size`

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

- `MPI_Comm_size` retourne dans $*size$ la taille du communicateur *comm* ;

```
#include <stdio.h>
#include <mpi.h>
int main (argc, argv)
int argc ; char *argv[] ;
{ int rank, size ;
MPI_Init (&argc, &argv) ;
MPI_Comm_size (MPI_COMM_WORLD,
&size) ;
printf( "Hello world :s = %d\n", size ) ;
MPI_Finalize() ;
return 0 ;}
```

```
% mpirun -np 4
essai
Hello world :s = 4
Hello world :s = 4
Hello world :s = 4
Hello world :s = 4
```

Rang d'un processus : `MPI_Comm_rank`

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

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

Exemple

- Programme : #include <stdio.h>
  #include <mpi.h>
  int main (argc, argv)
  int argc ; char *argv[] ;
  { int rank, size ;
  MPI_Init (&argc, &argv) ;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank) ;
  MPI_Comm_size (MPI_COMM_WORLD, &size) ;
  printf("Hello world from process %d of %d\n", rank, size) ;
  MPI_Finalize() ;
  return 0 ;}
- Execution : % mpirun -np 4 essai
  Hello world from process 1 of 4
  Hello world from process 0 of 4
  Hello world from process 2 of 4
  Hello world from process 3 of 4

En Résumé

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

Plan

MPI : Message Passing Interface

Premiers programmes/premières fonctions

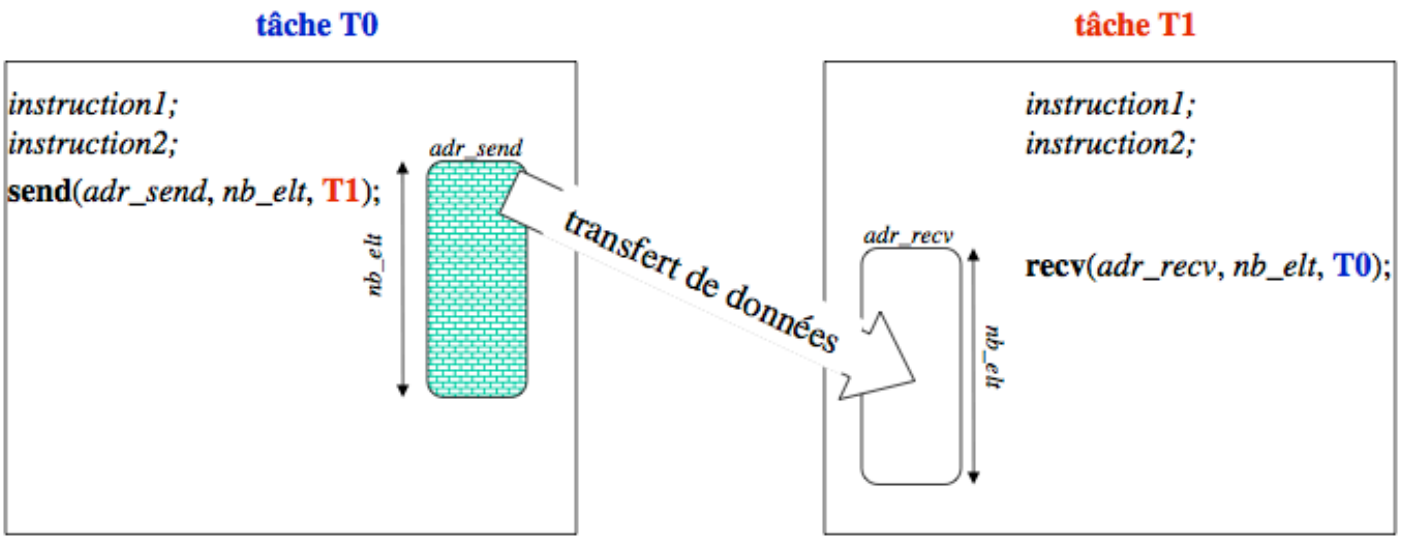
Les communications point à point

Echange de message : envoi (send)/réception (receive)

- Un message est caractérisé par :
  - une tâche expéditrice ;
  - une tâche destinataire
  - les données à échanger ;
- Pour échanger un message entre tâches
  - l'expéditeur doit envoyer le message (send) ;
  - le destinataire doit recevoir le message (receive) ;

Echange de message :principe

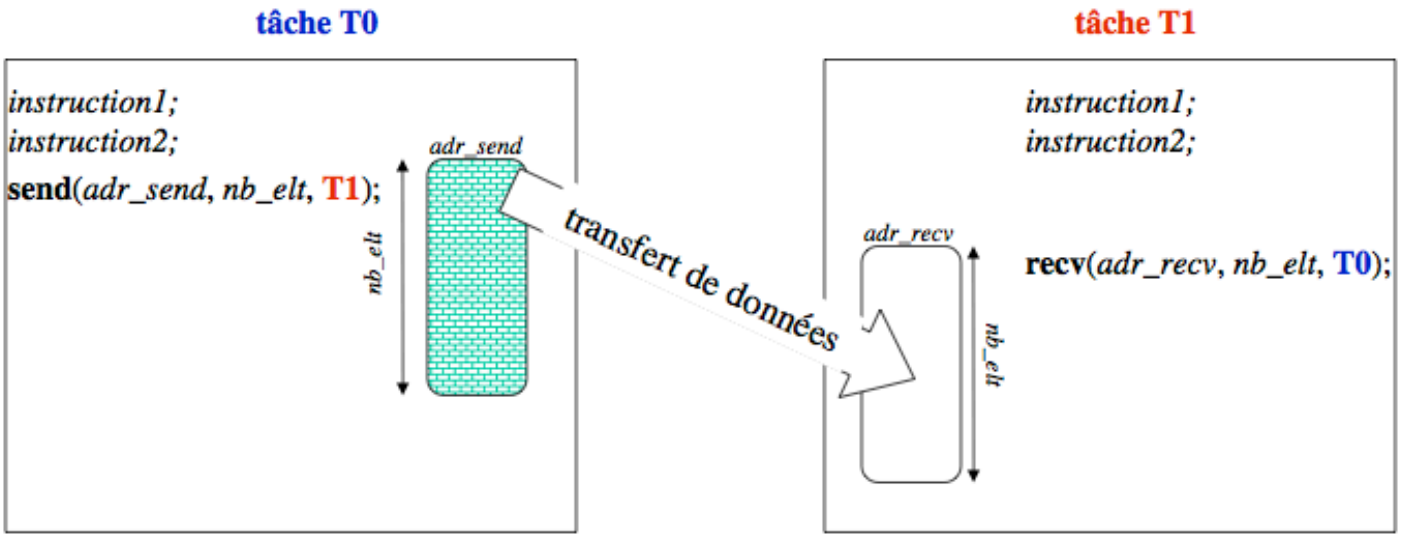
- Soit 2 tâches $T0$ et $T1$
  - Elles ont chacune leur propre espace d'adressage ;
  - Elle s'exécutent des instructions indépendantes en parallèle ;
- Pour mener à bien le calcul parallèle, $T1$ a besoin d'informations de $T0$ (point de synchronisation) :
  - $T0$ doit envoyer des données à $T1$: les données sont pointées par `adr_send` et de taille `nb_elt;`
  - $T1$ doit recevoir des données à $T0$:
    - la taille du message + le destinataire doit être connue par le destinataire (préallocation de la zone mémoire)



Echange de message :principe

L'échange d'information a lieu (par le réseau le plus souvent, on parle alors de communication)

- `send` bloque $T0$ tant que les données ne sont pas envoyées ;
- `reveive` bloque $T1$ tant qu'il n'a pas reçu toutes les données ;



Envoi : instruction MPI_SEND

```
int MPI_Send (




   );
```

Envoi : instruction MPI_SEND

```
int MPI_Send (
    void *buf,
```
(adresse des données à envoyer)

```
    );
```

Envoi : instruction MPI_SEND

```
int MPI_Send (
    void *buf,
    MPI_Datatype datatype ,


    );
```

(adresse des données à envoyer)

(type des données à envoyer)

Envoi : instruction MPI_SEND

```
int MPI_Send (
    void *buf,
    int count,
    MPI_Datatype datatype ,

    );
```

(adresse des données à envoyer)

(taille du message)

(type des données à envoyer)

Envoi : instruction MPI_SEND

```
int MPI_Send (
    void *buf, (adresse des données à envoyer)
    int count, (taille du message)
    MPI_Datatype datatype , (type des données à envoyer)
    int dest, (la destination)

    MPI_Comm comm, (dans quelle communicateur)
    );
```

Envoi : instruction MPI_SEND

```
int MPI_Send (
    void *buf, (adresse des données à envoyer)
    int count, (taille du message)
    MPI_Datatype datatype , (type des données à envoyer)
    int dest, (la destination)
    int tag,(drapeau du message)
    MPI_Comm comm, (dans quelle communicateur)
    );
```

Envoi : instruction MPI_SEND

Valeurs prédéfinies de type MPI_Datatype	*types C correspondants*
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	*un octet*
MPI_PACKED	*paquet de données non contiguës en mémoire*

Réception : instruction `MPI_recv`

```
int MPI_Rev (




   );
```

Réception : instruction `MPI_recv`

```
int MPI_Rev (
    void *buf, (adresse des données à recevoir)




    );
```

Réception : instruction MPI_recv

```
int MPI_Rev (
    void *buf,
    MPI_Datatype datatype ,


    );
```

(adresse des données à recevoir)

(type des données à recevoir)

Réception : instruction `MPI_recv`

```
int MPI_Rev (
    void *buf,
    int count,
    MPI_Datatype datatype ,

    );
```

(adresse des données à recevoir)

(la taille du message doit être inférieur ou égale à cette valeur)

(type des données à recevoir)

Réception : instruction `MPI_recv`

```
int MPI_Rev (
    void *buf, (adresse des données à recevoir)
    int count, (la taille du message doit être inférieur ou égale à
    cette valeur)
    MPI_Datatype datatype , (type des données à recevoir)
    int source, (la source)

    MPI_Comm comm, (dans quelle communicateur)

    );
```

Réception : instruction `MPI_recv`

```
int MPI_Rev (
    void *buf, (adresse des données à recevoir)
    int count, (la taille du message doit être inférieur ou égale à
    cette valeur)
    MPI_Datatype datatype , (type des données à recevoir)
    int source, (la source)
    int tag,(drapeau du message)
    MPI_Comm comm, (dans quelle communicateur)

    );
```

Réception : instruction MPI_recv

```
int MPI_Rev (
    void *buf, (adresse des données à recevoir)
    int count, (la taille du message doit être inférieur ou égale à
    cette valeur)
    MPI_Datatype datatype , (type des données à recevoir)
    int source, (la source)
    int tag,(drapeau du message)
    MPI_Comm comm, (dans quelle communicateur)
     MPI_Status *status (le statut du message reçu)
    );
```

Réception `MPI_Recv` et `MPI_Status`

- `MPI_Status` est une structure de données *C* :

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

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

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

Un exemple

- Programme : #include <stdio.h>
#include <mpi.h>
int main (argc, argv)
int argc ; char *argv[] ;
{ int rank, size ;
MPI_Init (&argc, &argv) ;
MPI_Comm_rank (MPI_COMM_WORLD, &rank) ;
if (rank == 0){ err = MPI_Send("hi", 3, MPI_CHAR, 1, 0,
MPI_COMM_WORLD) ;}
if (rank == 1){
err = MPI_Recv(str, 4, MPI_CHAR, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;
printf("%d recoit : %s",rank, str) ;}
MPI_Finalize() ;
return 0 ;}
- Execution : % mpirun -np 4 essai
1 recoit : hi

Pour aller encore plus loin

Communication bloquantes

- Un envoi send est dit bloquant si et seulement si au retour du send il est possible d'écrire dans le buffer d'envoi sans altérer le contenu du message.
- Une réception recv est bloquante si et seulement si au retour du recv le buffer de réception contient bien le contenu du message.

T0

```
a = 100;
send(&a, 1, T1);
a = 0;
```

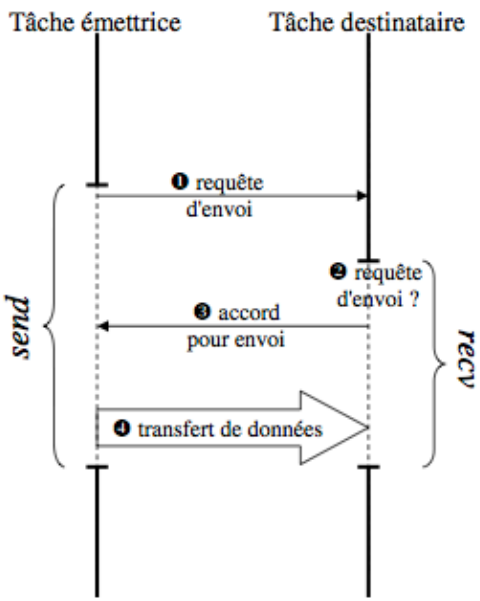
T1

```
recv(&a, 1, T0);
printf("%d\n", a);
```

Communication synchrone

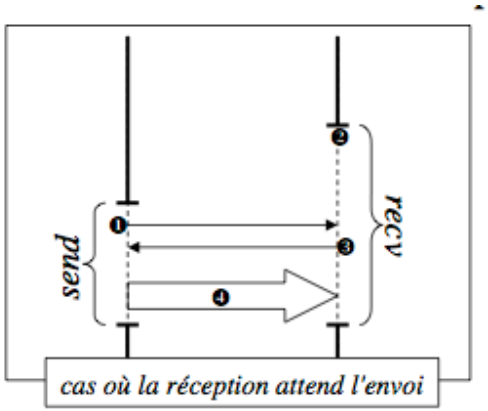
Définition

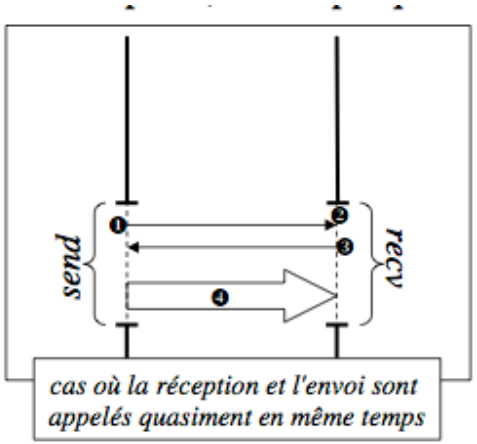
Un envoi synchrone bloquant rendra la main quand le message aura été reçu par le destinataire ;



1. Lors d'un envoi synchrone, l'expéditeur envoi au destinataire une requête d'envoi et attend que le destinataire lui réponde ;
2. Quand le destinataire débute sa réception, il attend une requête d'envoi de l'expéditeur ;
3. Quand le destinataire a reçu la requête d'envoi, il répond à l'expéditeur en lui accordant l'envoi ;
4. L'expéditeur et le destinataire sont alors synchronisé, le transfert de données (i.e. le message proprement dit) a lieu ; l'envoi et la réception sont alors terminés.

Inconvénients

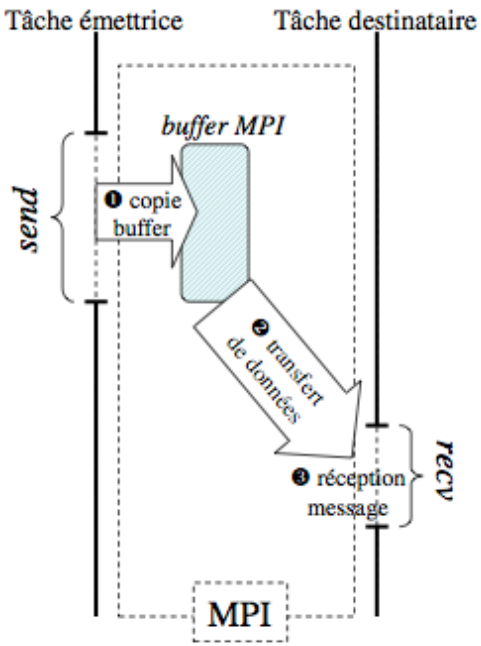




Communication bufferisée

Définition

Un envoi bufferisé bloquant rendra la main quand le message aura été copié dans un buffer géré par la bibliothèque de communication.



1. Lors d'un envoi bufferisé, l'expéditeur copie le message dans un buffer géré par la bibliothèque de communication ; l'envoi peut alors rendre la main.
2. La bibliothèque de communication s'approprie alors le message et peut l'envoyer au destinataire.
3. Le destinataire reçoit le message dès que possible.

Communication Standard

- fonction `MPI_Send` ;
- Protocole possible pour un envoi standard :
  - MPI considère une taille de message T :
    - si le message à envoyer est de taille inférieure à T, l'envoi est *bufferisé* (découplage entre l'envoi et la réception)
    - si le message à envoyer est de taille supérieure à T, alors l'envoi est synchronisé (pas de copie intermédiaire) ;

Communications non-bloquantes

Définition

Une communication est dite non bloquante si et seulement si au retour de la fonction, la bibliothèque de communication ne garantit pas que l'échange de message ait eu lieu.

- L'accès sûr aux données n'est donc pas garanti après une communication non bloquante !

- Pour pouvoir réutiliser les données du message, il faudra appeler une fonction supplémentaire qui complètera le message (i.e. qui assure un accès sûr aux données).

Envoi non bloquant : MPI_Isend

```
int MPI_Isend (
      void *buf(in),
      int count(in),
      MPI_Datatype datatype(in),
      int dest(in),
      int tag(in),
      MPI_Comm comm(in),
      MPI_Request *req(out)
      );
```

La signature est la même que `MPI_Send` hormis l'argument supplémentaire `MPI_Request *req`.

`MPI_Isend` demande une requête d'envoi.
L'identifiant de la requête est retourné dans `*req` (`MPI_Request` = type opaque encapsulant une requête MPI).

Un appel à `MPI_Isend` ne garantit pas que la bibliothèque de communication s'est appropriée le contenu du message.

Terminer une communication non bloquante :

```
int MPI_Wait (
        MPI_Request *req(inout),
        MPI_Status *sta(out)
        );
```

`MPI_Wait` bloquera jusqu'à ce que la requête de communication identifiée par `*req` soit terminée.

Des informations relatives à la communication sont retournées dans `*sta`.

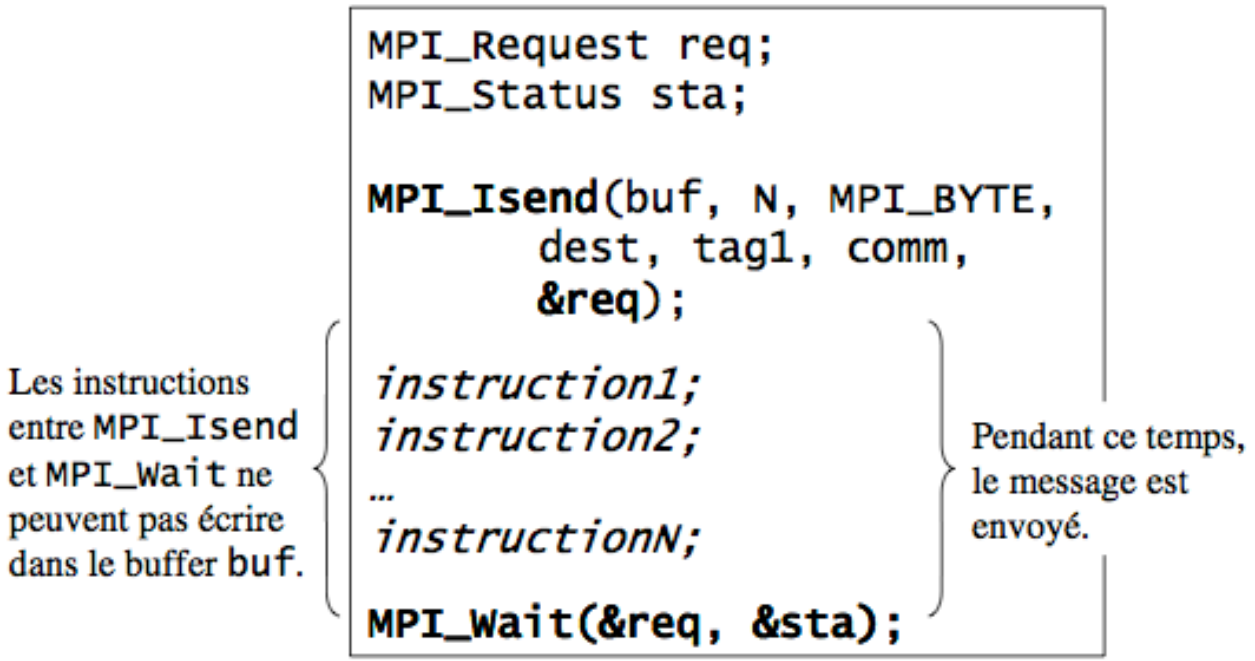
Au retour de `MPI_Wait`,

- `*req` est affectée à `MPI_REQUEST_NULL` (invalide la requête) ;
- il est possible d'écrire dans le buffer d'envoi utilisé par `MPI_Isend`.

Remarque :

`MPI_Send` $\Leftrightarrow$ `MPI_Isend` + `MPI_Wait`

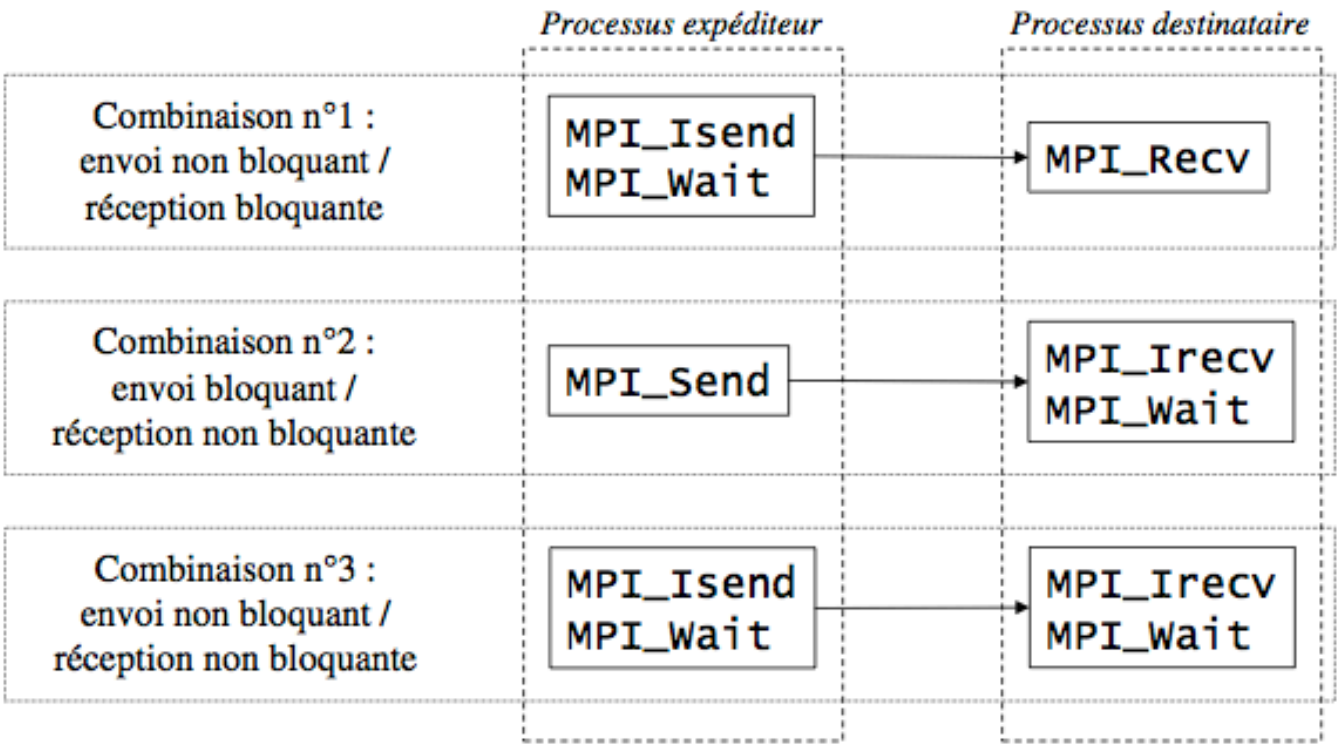
Exemple d'utilisation



Objectif

Couvrir les communications par le calcul

Récapitulatif (1/2)



Récapitulatif (2/2)

mode de communication / type d'appel	standard	bufferisé	synchrone
bloquant	`MPI_Send`	`MPI_Bsend`	`MPI_Ssend`
non bloquant	`MPI_Isend`	`MPI_Ibsend`	`MPI_Issend`