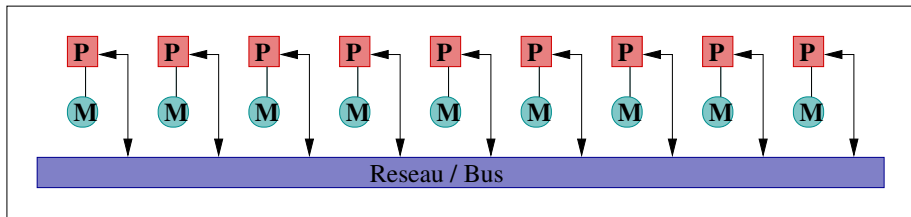


Programmation Parallèle pour les architectures à Mémoire Distribuée

Sophie Robert

Pôle info

Modèle de programmation par passage de messages



Il nécessite de la parallélisation explicite

- L'utilisateur doit coder le parallélisme dans son programme.
- L'accès aux données doit être exprimé dans le programme.
 - Pour exprimer le rôle de chaque processus dans les phases de lecture ou de lecture/écriture de données distribuées.

Modèle de programmation par passage de messages

asynchrone ou faiblement synchrone

- Il est adapté pour des programmes asynchrones
 - * Tous les processus effectuent des tâches concurrentes sans s'occuper des autres participants
- ou **faiblement synchrones**
 - * Les étapes de lecture/écriture des données synchronisent les processus mais sinon les tâches s'effectuent de manière asynchrone.

Modèle de programmation par passage de messages

Single Program Multiple Data

Les programmes respectent l'approche **SPMD** (Single Program Multiple Data) où le plus grand nombre de processus effectue les mêmes tâches sur des données différentes.

- Le programme unique utilise l'**identifiant des processus** pour différencier les instructions exécutées par chacun.

SPMD exemple

```
\\je connais mon identifiant pid
\\je connais le nombre total de processus nprocs
int a[5] = {0,1,2,3,4};
for (int i=0; i<5; i++)
    a[i]+=pid;
```

Sur 3 processus

P_0

0	1	2	3	4
?	?	?	?	?

P_1

0	1	2	3	4
?	?	?	?	?

P_2

0	1	2	3	4
?	?	?	?	?

SPMD exemple

```
\\je connais mon identifiant pid
\\je connais le nombre total de processus nprocs
int a[5] = {0,1,2,3,4};
if (pid==0)
    for (int i=0; i<5; i++)
        a[i]=0;
else
    for (int i=0; i<5; i++)
        a[i]+=pid;
```

Sur 3 processus

P_0

0	1	2	3	4
?	?	?	?	?

P_1

0	1	2	3	4
?	?	?	?	?

P_2

0	1	2	3	4
?	?	?	?	?

Les échanges de messages

Les opérations Send/Receive

- Les opérations de lecture/écriture nécessitent un moyen de partager des données distribuées.
- La base de ce modèle de programmation s'appuie sur les opérations élémentaires
 - * d'envoi de messages : `send(void *sendbuf, int size, int dest)`
 - * de réception de messages : `receive(void *recvbuf, int size, int source)`

La sémantique des opérations send/recv

La garantie des données



P0	P1
<pre>a=100 send(&a,sizeof(int),1) a=0;</pre>	<pre>receive(&a,sizeof(int),0) printf("%d",a);</pre>

Le processus P1 reçoit 0 ou 100 ?

Les opérations send/receive bloquant

Caractéristiques

- Négociation entre l'émetteur et le récepteur avant la transmission
- Reprise des tâches à la fin de la transmission (fin envoi / fin réception)

Inconvénients

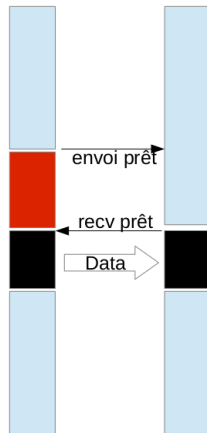
- Un taux d'attente important pour l'émetteur ou le récepteur.
- Des situations d'interblocage entre processus

P0	P1
<code>send(&a,sizeof(int),1)</code>	<code>send(&a,sizeof(int),0)</code>
<code>receive(&b,sizeof(int),1)</code>	<code>receive(&b, sizeof(int), 0)</code>

Les opérations send/receive bloquant

Déroulement

- Cas symétrique récepteur en avance
 - Temps d'attente -> perte de performances
 - + Les processus sont équilibrés en charge de travail
 - + Peut constituer une forme de synchronisation



Avec ou sans support hardware

L'interface réseau contient (ou pas)

- un processeur pour réaliser la communication indépendamment du CPU.
- un espace mémoire pour stocker les messages
- un composant DMA (Direct Memory Access) pour lire et écrire directement dans la mémoire centrale

-> **Meilleures performances avec du recouvrement calculs/communications.**

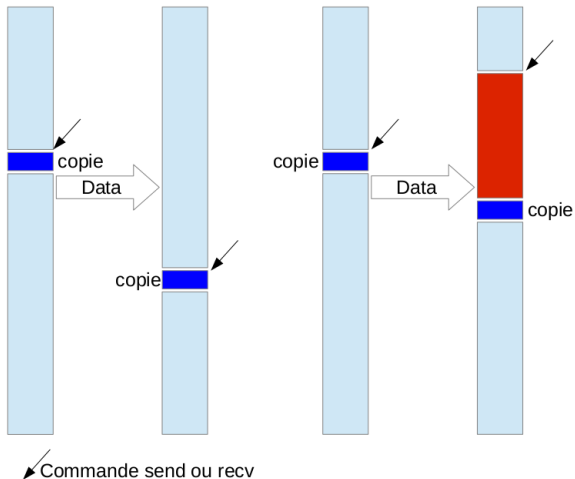
Les opérations send/receive avec bufferisation

Caractéristiques

- L'émetteur et le récepteur disposent d'un buffer pour stocker le message
- Une opération send est décomposée en
 - ① une bufferisation du message à transmettre (les données peuvent être modifiées sans altérer la transmission)
 - ② un envoi du buffer de manière asynchrone ou non selon le mode de communication sous jacent
- Une opération receive ne copie pas directement le message mais utilise également une bufferisation et
 - ① elle vérifie si le buffer contient le message à lire
 - ② ou **attend** jusqu'à réception du message dans ce buffer

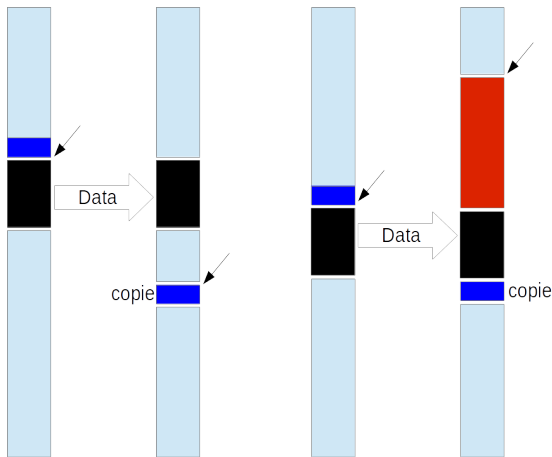
Les opérations send/receive avec bufferisation

Avec support hardware



Les opérations send/receive avec bufferisation

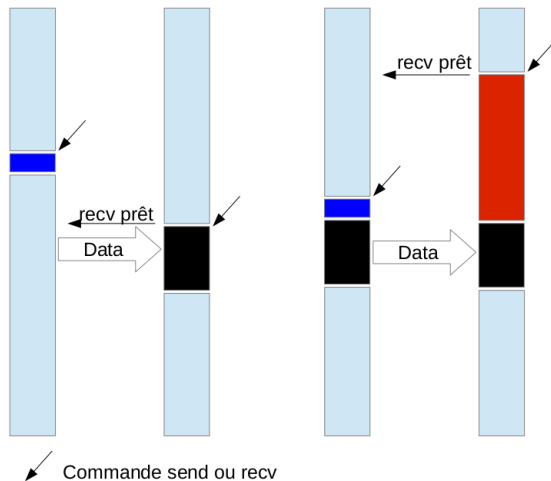
Sans support hardware



↙ Commande send ou recv

La bufferisation à un côté ?

Du côté émetteur



Bufferisation ou pas ?

Avantages/Inconvénients

- La bufferisation peut régler des temps d'attente et certains interblocages.
- Elle génère également un coût supplémentaire
 - * compensable par du recouvrement calcul/communication,
- Si la taille des buffers n'est pas suffisante, des coûts supplémentaires d'attente vont diminuer les performances
- Et il reste des cas d'interblocage !!

P0	P1
<code>receive(&a,sizeof(int),1)</code>	<code>receive(&a, sizeof(int), 0)</code>
<code>send(&b,sizeof(int),1)</code>	<code>send(&b,sizeof(int),0)</code>

Bufferisation ou pas ?

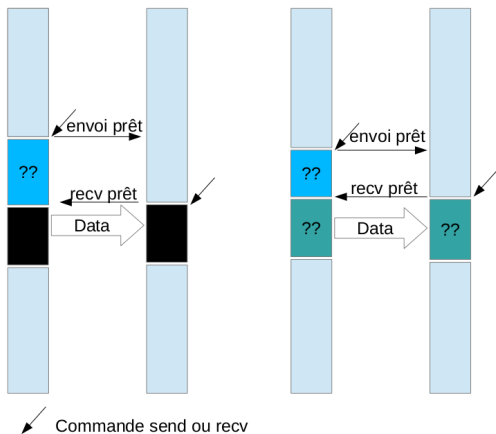
Si le programme est très synchrone

Si les opérations lecture/écriture ont lieu au même moment, on peut se passer du coût de la bufferisation

Les opérations send/receive non bloquant

Altération des données

- Comment gérer la partie où l'altération de la donnée peut modifier le comportement du programme.



MPI - Message Passing Interface

- MPI est une spécification et un standard
 - Les implémentations de MPI sont fournies sous forme de bibliothèques libres ou commerciales
 - Les deux principales : MPICH2, OpenMPI
- Langages supportés : C/C++, Fortran (python, java, perl ...)

MPI permet de gérer

- l'environnement d'exécution
- les communications point à point (Send, Recv)
- les communications collectives (Bcast, Reduce, Scatter,...)
- les groupes de processus et les communicateurs
- la topologie d'inter-connexion des processus (grilles, arbres,...)

Quelques liens

Distributions

- <https://www.open-mpi.org/>
- <https://www.mpich.org/>

Documentation

- <https://www.mpi-forum.org/>
- <https://www-unix.mcs.anl.gov/mpi/tutorial>

MPI en C

Fichier `mpi.h`

Il doit être inclus en entête de tous les programmes MPI.

- Déclaration des prototypes de toutes les routines MPI
- Déclaration de l'ensemble des constantes MPI
- Déclaration de toutes les structures de données

Routines MPI

- Les routines MPI (en C) sont sous deux formes
 - 1 `MPI_Xxxx()`,
 - 2 `MPI_Xxxx_xxx()`.

4 routines MPI de base

- pour initialiser et terminer un programme SPMD parallèle
 - 1 MPI_Init,
 - 2 MPI_Finalize,
- pour identifier l'équipe de processus
 - 1 MPI_Comm_size,
 - 2 MPI_Comm_rank,

MPI sous langage C (2)

La structure d'un programme MPI

Inclure le fichier mpi.h

Initialiser l'environnement MPI

.....
.....

Faire des calculs

Appeler des routines MPI :

- communiquer
- synchroniser

.....
.....

Terminer l'environnement MPI

Initialisation et terminaison de MPI

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

- C'est la première routine MPI exécutée par tous les processus
- Elle permet de débiter l'exécution parallèle
- Elle permet de diffuser les arguments donnés en ligne de commande
- Elle renvoie MPI_SUCCESS si l'appel n'a eu aucun problème

```
int MPI_Finalize(void);
```

- Elle termine proprement l'exécution parallèle et doit être appelée par tous les processus
- Elle renvoie MPI_SUCCESS si l'appel n'a eu aucun problème

Les domaines de communication

Les communications MPI travaillent dans un **communicateur** soit un ensemble de processus pouvant communiquer entre eux.

MPI_Init initialise le **communicateur par défaut**

MPI_COMM_WORLD qui comprend tous les processus impliqués dans l'exécution parallèle

- Il est possible de construire d'autres communicateurs comme un sous groupe de MPI_COMM_WORLD
- Le type d'un communicateur est MPI_Comm

Les domaines de communication

Taille du communicateur et identifiant du processus

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

- Cette routine retourne dans `nprocs` le nombre total de processus du communicateur `comm`

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

- Cette routine initialise `pid` l'identifiant relatif au communicateur `comm` du processus appelant
- `pid` est un nombre entier unique dans le communicateur `comm`
- Initialement, chaque processus a un identifiant unique $pid \in [0, nprocs - 1]$ dans `MPI_COMM_WORLD`.

Programme MPI en C

Premier programme

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int pid, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    printf("Process %d/%d\n", pid, nprocs);
    MPI_Finalize();

    return 0;
}
```

Installation et compilation d'un programme MPI

Installation Ubuntu - Package

```
$ sudo apt-get install openmpi  
openmpi-common  
libopenmpi-dev
```

Compilation habituelle mais avec mpicxx

```
mpicxx -c exemple.c  
mpicxx -o exemple exemple.o
```

Exécution en local (sans hostfile)

mpirun pour lancer l'exécution parallèle

```
mpirun -np 4 ./exemple
```

Résultat

```
Process 3/4
```

```
Process 0/4
```

```
Process 2/4
```

```
Process 1/4
```

Exécution sur une machine parallèle

Définir les différents hôtes des processus dans un fichier *hostfile*

```
machine_1 slots=1  
machine_2 slots=2  
.....  
machine_n slots=1
```

Exécution

```
mpirun --hostfile hostfile ./exemple1
```

Résultat sur chaque hôte



```
Process 3/4
```

MPI : Librairie de fonctions de communications

Les communications

- point-à-point : communications entre 2 processus dans un communicateur
- collectives : communications impliquant **tous les processus** d'un communicateur.

Conseil

Toujours utiliser les fonctions fournies par MPI. Elles sont souvent plus performantes que des versions "maison".

Communications point-à-point

Modes de communication

- bloquant
- non-bloquant

Types de communication

- standard
- synchrone
- bufferisé
- ready

Résumé

Types/Modes	Bloquant	Non bloquant
Envoi standard	MPI_Send	MPI_Isend
Envoi synchrone	MPI_Ssend	MPI_Issend
Envoi bufferisé	MPI_Bsend	MPI_Ibsend
Envoi ready	MPI_Rsend	MPI_Irsend
Réception	MPI_Recv	MPI_Irecv

Routine de réception bloquante MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status)
```

Paramètres

- 1 void ***buff** : Adresse du tampon de réception
- 2 int **count** : Nombre d'éléments dans le tampon de données
- 3 MPI_Datatype **dtype** : Type de chaque élément envoyé
- 4 int **src** : Identifiant du processus émetteur,
- 5 int **tag** : Étiquette de message
- 6 MPI_Comm **comm** : Communicateur,
- 7 MPI_Status ***status**

Types de données en MPI

Type MPI	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_2INT	pair of 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	octet par octet

Les étiquettes

- Elles permettent de distinguer les messages au niveau du récepteur pour recevoir les messages dans le bon ordre,
- S'il n'y a pas d'ambiguïté on peut utiliser `MPI_ANY_TAG` pour accepter tous les messages.

Les informations sur la réception

MPI_Status

- C'est une structure de données pré-définie par MPI,
- Elle nous permet d'obtenir des informations supplémentaires sur le message reçu.
- MPI_STATUS_IGNORE

Structure MPI_Status

```
typedef struct MPI_Status
{
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
};
```

Envoi bloquant **standard**

Envoi avec la routine MPI_Send

- Deux types de communications sont gérées
 - 1 Bufferisé
 - 2 Synchrone
- Le **choix automatique** dépend de
 - La taille du message à échanger,
 - L'implémentation de MPI.

Réception avec la routine MPI_Recv

La transmission se termine lorsque le message est arrivé

Routine MPI_Send

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

Paramètres

- 1 void ***buff** : Adresse du tampon de données à envoyer
- 2 int **count** : Nombre d'éléments dans le tampon de données
- 3 MPI_Datatype **dtype** : Type de chaque élément envoyé
- 4 int **dest** : Identifiant du processus de destination
- 5 int **tag** : Étiquette de message
- 6 MPI_Comm **comm** : Communicateur

Exemple Send/Recv

$P_0 \rightarrow P_1$

```
int pid , nprocs , i ;
int buff [10];
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
.....
.....
if (pid == 0){
    for(i=0;i<10;i++)
        buff[i]=i;
    MPI_Send(buff , 10, MPI_INT, 1, 9, MPI_COMM_WORLD);
}
if (pid ==1)
    MPI_Recv(buff , 10, MPI_INT, 0, 9, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
.....
.....
```

Envoi bloquant : Synchrones

Envoi avec la routine MPI_Ssend

- L'émetteur se signale d'abord au récepteur
- Il attend la réponse du récepteur
- Le transfert de données est réalisé
- L'appel de cette routine retourne au programme appelant lorsque le récepteur a bien reçu tout le message

Réception avec la routine MPI_Recv

- Le récepteur répond à l'émetteur
- Il provoque le transfert de données
- L'appel de cette routine retourne au programme appelant lorsque tout le message a bien été reçu

Routine MPI_Ssend

Prototype

```
int MPI_Ssend(void *buf, int count, MPI_Datatype dtype,
              int dest, int tag, MPI_Comm comm);
```

Paramètres

- 1 void *buf : Adresse initiale du buffer d'envoi
- 2 int count : Nombre d'éléments envoyés
- 3 MPI_Datatype dtype : Type de chaque élément envoyé
- 4 int dest : Identifiant du processus récepteur
- 5 int tag : Étiquette du message
- 6 MPI_Comm comm : Communicateur

Exemple Send/Recv

$P_0 \rightarrow P_1$

```
int pid , nprocs , i ;
int buff [10];
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
.....
.....
if (pid == 0){
    for(i=0;i<10;i++)
        buff[i]=i;
    MPI_Ssend(buff , 10, MPI_INT, 1, 9, MPI_COMM_WORLD);
}
if (pid ==1)
    MPI_Recv(buff , 10, MPI_INT, 0, 9, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
.....
.....
```

MPI_Send ou MPI_Ssend



Conseil

Pour contrôler son programme préférer **MPI_Ssend** à MPI_Send

Envoi bloquant : bufferisé

Envoi avec la routine MPI_Bsend

- L'émetteur copie d'abord les données dans un buffer local,
- Cette routine retourne immédiatement au programme appelant après la copie des données.

Réception avec la routine MPI_Recv (Pas de MPI_Brecv)

- Le récepteur se signale à l'émetteur
- Il provoque le transfert de données
- Cette routine retourne au programme appelant à la fin du transfert.

Envoi avec la routine MPI_Bsend

Au niveau de l'émetteur

```
MPI_Pack_size(XXX) //Calcul de la taille du tampon
.....
.....
malloc ou calloc(XXX) //Allocation la memoire du tampon
.....
.....
MPI_Buffer_attach(XXX) //Attachement du tampon
.....
.....
MPI_Bsend(XXX) //Envoi du message
.....
.....
MPI_Buffer_detach(XXX) //Detachement du tampon
free(XXX) //liberation du tampon
.....
```

Association d'un buffer local pour un processus

Calcul de la taille du buffer

- 1 Taille d'un message `MPI_Pack_size`
- 2 Plus le surcoût de taille pour chaque message : `MPI_BSEND_OVERHEAD`

`MPI_Pack_size`

```
int MPI_Pack_size(int incount, MPI_Datatype dtype, MPI_Comm comm, int *size);
```

- 1 `int incount` : Nombre d'éléments du paquet
- 2 `MPI_Datatype dtype` : Type en MPI d'un élément
- 3 `MPI_Comm comm` : Communicateur
- 4 `int *size` : La taille du message en octets

Attacher le buffer : Routine MPI_Buffer_attach

Prototype

```
int MPI_Buffer_attach(void *buffer, int size);
```

- 1 void *buffer : Adresse du tampon à attacher
 - 2 int size : Taille du tampon en octets
- Cette routine fournit au système un tampon afin de copier le message avant son envoi
 - Le tampon est utilisé uniquement par les messages envoyés par MPI_Bsend
 - Un seul tampon peut être attaché à un processus à la fois

Réception avec la routine MPI_Bsend

Prototype

```
int MPI_Bsend(void *buf, int count, MPI_Datatype dtype,
              int dest, int tag, MPI_Comm comm);
```

Paramètres

- 1 void *buf : Adresse du tampon d'envoi
- 2 int count : Nombre d'éléments à envoyer dans le tampon
- 3 MPI_Datatype dtype : Type de chaque élément envoyé
- 4 int dest : Identifiant du processus récepteur
- 5 int tag : Étiquette du message
- 6 MPI_Comm comm : Communicateur

Détacher un buffer : Routine MPI_Buffer_Detach

Prototype



```
int MPI_Buffer_detach(void *buffer, int *size);
```

Paramètres

- 1 void *buffer : Adresse du tampon à détacher
 - 2 int *size : Taille du tampon en octets
- Cette routine détache le tampon actuel associé à MPI_Bsend
 - L'appel à cette routine permet de bloquer le programme appelant jusqu'à ce que tous les messages dans ce tampon aient été transmis.
 - L'utilisateur peut réutiliser ou désallouer l'espace occupé par le tampon

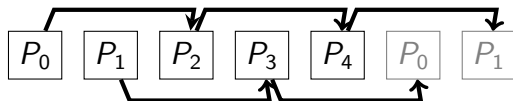
Envoi bloquant : bufferisé

Conclusion

- Ce mode est parfois plus efficace
- Mais il reste des situations de blocage 
- et attention aux temps de recopies 

Exemple 1

Schéma de communications



Exemple 1

```
int size_tab = atoi(argv[1]);
int* tab = new int[size_tab];
for (int i=0; i<size_tab; i++)
    tab[i] = i+pid;
int size = size_tab;
MPI_Pack_size(size_tab, MPI_INT, MPI_COMM_WORLD, &size);
size += MPI_BSEND_OVERHEAD;
int *buffer = new int[size];
MPI_Buffer_attach(buffer, size);
int to = (pid + 2) % nprocs;
int from = ((pid - 2) + nprocs) % nprocs;
MPI_Bsend(tab, size_tab, MPI_INT, to, 34,
    MPI_COMM_WORLD);
MPI_Recv(tab, size_tab, MPI_INT, from, 34,
    MPI_COMM_WORLD, &status);
MPI_Buffer_detach(buffer, &size);
```

Exemple 2

Distribution d'un tableau

Un processus accède à un ensemble de *nglobal* entiers (calcul, lecture fichier, etc) et souhaite les distribuer à l'ensemble des processus de `MPI_COMM_WORLD` ?

Par exemple

$$nglobal = 20$$

$$nlocal = ?$$

avec *nglobal* non divisible par le nombre de processus (*nprocs* = 6).

Combinaison de l'envoi et la réception

Routine MPI_Sendrecv

- L'échange combine :
 - l'envoi d'un message vers un processus
 - la réception d'un message venant du même processus ou d'un autre
- Les buffers et les types peuvent être différents
- C'est une routine très utile pour des opérations dans une chaîne de processus
- Elle est pratique et efficace

Routine MPI_Sendrecv_replace

- Le même buffer est utilisé pour l'envoi et la réception,
- L'implémentation gère le stockage intermédiaire additionnel

Routine MPI_Sendrecv

Prototype

```
int MPI_Sendrecv(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)
```

Paramètres

- 1 void *sendbuf : Adresse du tampon d'envoi
- 2 int sendcount : Nombre d'éléments envoyés
- 3 MPI_Datatype sendtype : Type de chaque élément envoyé
- 4 int dest : Identifiant du processus récepteur
- 5 int sendtag : Étiquette du message envoyé

Routine MPI_Sendrecv

Paramètres

- ⑥ void *recvbuf : Adresse initiale du tampon de réception
- ⑦ int recvcount : Nombre d'éléments de réception
- ⑧ MPI_Datatype recvtype, Type de chaque élément de réception
- ⑨ int source : Identifiant du processus émetteur
- ⑩ int recvtag : Étiquette du message de réception
- ⑪ MPI_Comm comm : Communicateur
- ⑫ MPI_Status *status : État du message de réception

Routine : MPI_Sendrecv

Exemple

- Chaque processus a deux buffers :
 - 1 buffer1 pour l'envoi
 - 2 buffer2 pour la réception
- Le processus d'identifiant i
 - 1 envoie les données du buffer1 au processus $i-1$
 - 2 reçoit les données du processus $i+1$, et les met dans buffer2.

Routine : MPI_Sendrecv

Extrait du code

```
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);

to = (pid + 2) % numprocs;
from = (pid - 2 + numprocs) % numprocs;

MPI_Sendrecv(buffer1, 10, MPI_INT, to, 123, buffer2, 10,
             MPI_INT, from, 123,
             MPI_COMM_WORLD, &status);
```

Routine MPI_Sendrecv_replace

Prototype

```
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)
```

Paramètres

- 1 void *buf : Adresse initiale du tampon d'envoi et de réception
- 2 int count : Nombre d'éléments envoyés et de réception
- 3 MPI_Datatype datatype : Type de chaque élément

Routine MPI_Sendrecv_replace

Prototype

```
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)
```

Paramètres

- ④ int dest : Identifiant de processus récepteur
- ⑤ int sendtag : Étiquette du message envoyé
- ⑥ int source : Identifiant du processus émetteur

Routine MPI_Sendrecv_replace

Prototype

```
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)
```

Paramètres

- ⑦ int recvtag : Étiquette du message de réception
- ⑧ MPI_Comm comm : Communicateur
- ⑨ MPI_Status *status : État du message de réception

Exemple 1 : nouvelle version

Extrait du code

```
// tab un tableau de n entiers sur chaque processus
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);

to = (pid + 2) % nprocs;
from = (pid - 2 + nprocs) % nprocs;

MPI_Sendrecv_replace(tab, 10, MPI_INT,
                    to, 123,
                    from, 123,
                    MPI_COMM_WORLD, &status);
```

Communication non-bloquante



Une communication non bloquante rend la main **avant que la communication ne soit terminée**. Il faut être très prudent dans son utilisation mais elle peut permettre d'optimiser son programme.

Principe

- Ce type de communications est moins coûteux (pas de messages au préalable, moins de synchronisation)
- Il faut utiliser du recouvrement calcul communication avant de modifier la donnée
 - 1 Initier la communication non-bloquante dès que la donnée à envoyer est disponible
 - 2 Lancer des calculs n'ayant pas besoin de cette donnée
 - 3 Vérifier l'état de la communication avant d'accéder à la donnée

Communications non-bloquantes

Type de communication

- standard : `MPI_Isend` - `MPI_Irecv`
- synchrone : `MPI_Ssend` - `MPI_Irecv`
- bufférisation : `MPI_Ibsend` - `MPI_Irecv`
- ready : `MPI_Irsend` - `MPI_Irecv`

L'utilisateur doit lui-même s'assurer que

Le message a bien été envoyé ou reçu avec

- **`MPI_Test`** pour tester si l'opération de communication est terminée
- **`MPI_Wait`** pour attendre que l'opération de communication se termine

Le mode non-bloquant Standard ou Synchrones

Envoi par MPI_Isend/MPI_Issend Réception par MPI_Irecv

- Le processus émetteur (resp. récepteur)
 - initialise l'opération d'envoi (resp. de réception)
 - mais **rend la main** avant sa réalisation
- L'appel sera terminé avant que le message ne soit parti (resp. reçu)
- Après l'initialisation et avant l'envoi (resp. avant la réception), d'autres calculs peuvent être faits

MPI_Irecv

Prototype

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

Paramètres

- 1 void ***buff** : Adresse initiale du tampon de données
- 2 int **count** : Nombre d'éléments dans le tampon de données
- 3 MPI_Datatype **datatype** : Type de chaque élément envoyé
- 4 int **src** : Identifiant du processus émetteur
- 5 int **tag** : Étiquette de message
- 6 MPI_Comm **comm** : Communicateur
- 7 MPI_Request ***request** : Requête de communication

Envoi non-bloquant standard MPI_Isend

Prototype

```
int MPI_Isend(void *buf, int count,  
             MPI_Datatype datatype, int dest, int tag,  
             MPI_Comm comm, MPI_Request *request)
```

Paramètres

- 1 void ***buff** : Adresse initiale du tampon de données,
- 2 int **count** : Nombre d'éléments dans le tampon de données,
- 3 MPI_Datatype **datatype** : Type de chaque élément envoyé,
- 4 int **dest** : Identifiant du processus de destination,
- 5 int **tag** : Étiquette de message,
- 6 MPI_Comm **comm** : Communicateur
- 7 MPI_Request ***request** : Requête de communication

Envoi non-bloquant synchrone MPI_Issend

Prototype

```
int MPI_Issend(void *buf, int count,  
              MPI_Datatype datatype,  
              int dest, int tag,  
              MPI_Comm comm, MPI_Request *request)
```

Principe

Aucune bufferisation et on ne doit modifier la donnée que lorsqu'elle a été **envoyée entièrement**.

Envoi non-bloquant bufferisé MPI_Ibsend

Prototype

```
int MPI_Ibsend(void *buf, int count,  
              MPI_Datatype datatype,  
              int dest, int tag,  
              MPI_Comm comm, MPI_Request *request)
```

Principe

Le buffer n'est pas géré par l'utilisateur. On peut modifier la donnée **dès la fin de la bufferisation.**

MPI_Isend : MPI_Issend ou MPI_Ibsend

Que signifie Standard ?

Bufferisation ou pas et c'est le choix de l'implémentation

Les routines de complétion

Test

- `MPI_Test`
- `MPI_Testall`
- `MPI_Testany`

Attente

- `MPI_Wait`
- `MPI_Waitall`
- `MPI_Waitany`

Routine MPI_Test

Prototype

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

Principe

- Elle vérifie l'achèvement d'une opération associée à **request**
- Elle donne **flag=true** si l'opération (send/recv) identifiée par **request** est terminée
- **request** est alors libérée et mis à MPI_REQUEST_NULL
- **status** contient des informations sur l'opération terminée

Une parmi plusieurs ou toutes

```
int MPI_Testany(int count, MPI_Request *array_of_req,  
               int *index, int *flag, MPI_Status *status)
```

```
int MPI_Testall(int count, MPI_Request *array_of_req,  
               int *flag, MPI_Status *array_of_status)
```

Routine MPI_Test

```
.....  
.....  
MPI_Irecv(buffer, 10, MPI_INT,  
          left, 123, MPI_COMM_WORLD,  
          &request);
```

```
MPI_Test(&request, &flag, &status);  
while (!flag)  
{  
    /* Do some work ... */  
    .....  
    .....  
    MPI_Test(&request, &flag, &status);  
}
```

Routine MPI_Wait

Prototype

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Principe

- Cette routine retourne au programme appelant lorsque l'opération identifiée par **request** est complète.
- **request** est alors libérée et mis à MPI_REQUEST_NULL par cet appel.
- **status** contient des informations sur la communication lorsque l'opération est terminée

Au moins une ou toutes

```
int MPI_Waitany(int count, MPI_Request *array_of_req,  
               int *index, MPI_Status *status)
```

```
int MPI_Waitall(int count, MPI_Request *array_of_req,  
               MPI_Status *array_of_status)
```

Routine MPI_Wait

Utilisation de MPI_Wait

```
.....  
.....  
MPI_Irecv(buffer, 10, MPI_INT,  
          left, 123, MPI_COMM_WORLD,  
          &request);  
/* Do some work ... */  
.....  
MPI_Wait(&request, &status);  
/* Do some work ... */  
.....
```

Routine MPI_Waitall

Utilisation de MPI_Wait

```
.....  
.....  
MPI_Irecv(buffer, 10, MPI_INT,  
          left, 123, MPI_COMM_WORLD,  
          request);  
MPI_Isend(buffer2, 10, MPI_INT,  
          right, 123, MPI_COMM_WORLD,  
          request+1);  
.....  
.....  
MPI_Waitall(2, request, status);  
.....  
.....
```