Introduction à la Programmation MPI
TD 2 –
Communications point-à-point bloquantes

Exercice I : Ping pong

On demande d'écrire 3 programmes MPI qui font intervenir 2 processus MPI.
On désigne par :

P0, le processus MPI de rang 0.	P1, le processus MPI de rang 1.

Programme 1 (ping) :

P0 envoie un entier de valeur 10 à P1.	P1 affiche la valeur reçue.

Programme 2 (pong) :

P0 affiche le contenu du tableau[2].	P1 remplit et envoie un tableau de 10 réels double précision à P0[1].

Programme 3 (ping-pong) :

P0 envoie la valeur 10 à P1. P0 doit afficher le contenu de ce tableau.	Après la réception de ce message, P1 doit attendre 5 secondes[3] avant de remplir et envoyer un tableau de 10 réels à P0.

1 Vous êtes libres du contenu du tableau envoyé par P1
2 Vérifiez qu'il s'agit bien du même contenu que celui de P1
3 Utilisez la fonction `sleep` (`#include <unistd.h>`)

Exercice II : Maître/Esclaves

Ouvrir le fichier **`master_slave/exercice/master_slave_exo.c`**

Principe du programme :

- le processus 0 joue le rôle du maître, les autres processus sont les esclaves ;
- tant qu'il y a des données à lire, le processus 0 lit des données (fonction `read_data`) : pour une lecture, le maître envoie ce tableau au premier esclave disponible ;
- chaque esclave attend du maître un message dont il ne connaît pas la nature par avance : *« données à traiter »* ou bien *« fin du travail de l'esclave »* ;
- si la nature du message est *« données à traiter »*, l'esclave appellera la fonction `process_data` puis se mettra en attente du prochain message venant du maître;
- si la nature du message est *« fin du travail de l'esclave »*, l'esclave terminera son travail.

Paralléliser ce programme avec MPI en complétant les rubriques `/* TRAVAIL A FAIRE */`.
Ce programme fonctionnera avec au moins deux processus MPI.

Quelques indications :

- utiliser les étiquettes des messages pour indiquer les natures des messages ;
- la fonction `MPI_Probe` permet d'attendre n'importe quel type de message ;
- la fonction `MPI_Get_count` permet de récupérer la taille d'un message à recevoir ;
- `MPI_ANY_SOURCE` et `MPI_ANY_TAG` permettent à `MPI_Recv` d'attendre un message de n'importe quelle source avec n'importe quelle étiquette.

Exercice III : Deadlock

Ouvrez le programme deadlock/exercice/deadlock.c.

Dans ce programme (valable uniquement pour deux processus MPI), les processus 0 et 1 veulent s'envoyer mutuellement les n octets contenus dans leurs buffers d'envois respectifs buf_send (voir section de code ci-dessous).

```
char *buf_send = calloc(n, sizeof(char)) ;
char *buf_recv = calloc(n, sizeof(char)) ;

if (rang == 0)
        vois = 1 ;
else
        vois = 0 ;

MPI_Send(buf_send, n, MPI_BYTE, vois, 0, MPI_COMM_WORLD) ;
MPI_Recv(buf_recv, n, MPI_BYTE, vois, 0, MPI_COMM_WORLD, &sta) ;
```

1. Déterminez (par exécution successive) la valeur seuil de n pour laquelle le programme bloque.

2. Expliquez en quoi cette section de code n'est pas sure.

3. Remplacez l'envoi standard par :
   a. un envoi synchrone ;
   b. un envoi bufferisé (en utilisant les fonctions MPI_Buffer_attach et MPI_Buffer_detach et la variable MPI_BSEND_OVERHEAD) ;

   Comment se comporte le programme dans chacun des cas a et b ? Débloquez le programme pour le cas a.

4. Réécrivez cette section de code pour qu'elle fonctionne quelle que soit la valeur n et ceci avec un envoi standard.

Exercice IV : Implémentation d'un *broadcast*

Le programme `algo_bcast/exercice/mpi_bcast.c` prend en argument un entier n qui représente la taille en octets d'un tableau.
Seul le processus de rang 0 remplit ce tableau et le diffuse 100 fois aux autres processus en utilisant la fonction `MPI_Bcast`.

Question 1 : mesurer le temps pris par ce programme initial avec 48 processus repartis sur 4 nœuds pour des tailles respectives de 10, 100, 1 000, 10 000, 100 000, 1 0000 000 et 10 000 000.

A présent, on désire implémenter nous-même notre propre communication collective mais en utilisant uniquement les communications point-a-point `MPI_Ssend` et `MPI_Recv`.

Question 2 : Implémenter l'**algorithme linéaire** vu en cours (voir figure 1). Remplacer l'appel à `mpi_bcast` par une fonction **`linear_bcast`**. Faire les mesures de temps et les comparer au programme initial.

Question 3 : Implémenter le **premier algorithme en arbre binaire** vu en cours (voir figure 2). Remplacer l'appel à `mpi_bcast` par une fonction **`btreev1_bcast`**. Faire les mesures de temps et les comparer au programme initial.

Question 4 : Implémenter le **deuxième algorithme en arbre binaire** vu en cours (voir figure 3). Remplacer l'appel à `mpi_bcast` par une fonction **`btreev2_bcast`**. Faire les mesures de temps et les comparer au programme initial.

Question 5 : Implémenter un algorithme qui prenne en compte la topologie du cluster. Faire les mesures de temps et les comparer aux programme précédents. Pour ce faire :

a) **Créer des communicateurs** pour les processus qui appartiennent aux mêmes nœuds (utiliser la fonction **`MPI_Comm_split_type`** et la variable **`MPI_COMM_TYPE_SHARED`**).

b) Designer un processus maître par nœud et créer le communicateur de tous les processus maîtres

c) En s'appuyant sur le deuxième algorithme en arbre binaire, implémenter le *broadcast* en diffusant d'abord le tableau entre les nœuds puis en le diffusant à l'intérieur des nœuds.

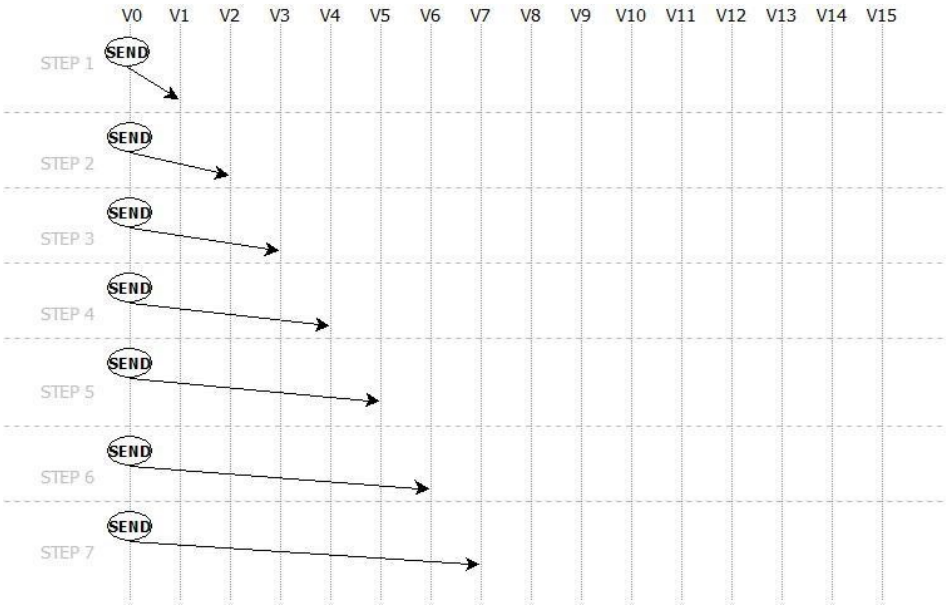

Fig 1 – Algorithme linéaire

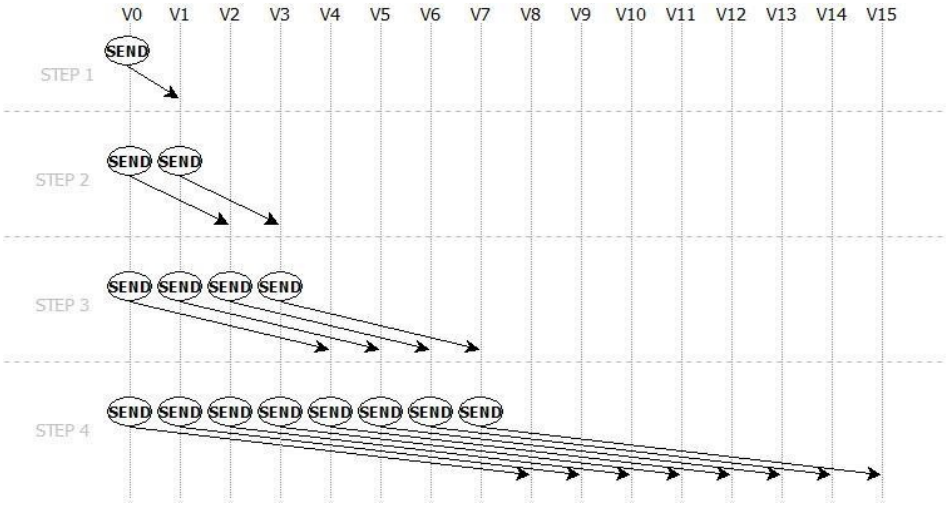

Fig 2 – Binary tree V1

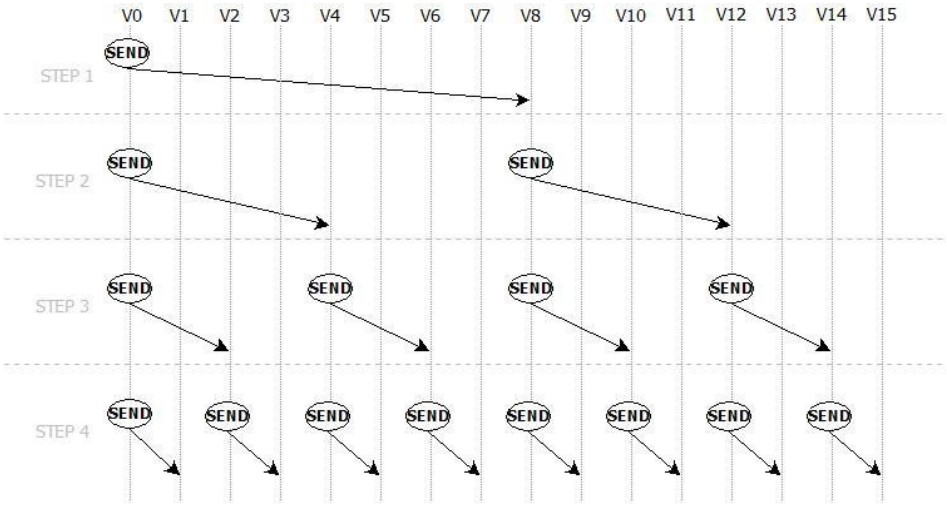

Fig 3 – Binary tree V2