Introduction à la programmation MPI
TD 1 –
Prise en main et Communications point-à-point bloquantes

Exercice I : Prise en main

Se connecter sur le cluster :

```
prompt> ssh -Y hpc.pedago.ensiie.fr -l prenom.nom
```

mdp : /4-premieres-lettres-du-prénom/4-dernieres-lettres-du-nom/4/LEARN

Préparation environnement :

```
hpc01> module load mpi/openmpi-x86_64
```

Compilation (se comporte comme un compilateur classique) :

```
hpc01> mpicc monprog.c -o monprog.exe
```

Connaître l'ensemble des nœuds disponibles :

```
hpc01> sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
inter        up   infinite      1  down* hpc03
inter        up   infinite      1  drain hpc02
calcul*      up   infinite      5  down* hpc[09-13]
calcul*      up   infinite      5   idle hpc[04-08]
```

Allouer de la ressource (ici 4 cœurs (-n 4) répartis sur 2 nœuds (-N 2)) :

```
hpc01> salloc -n 4 -N 2
salloc: Granted job allocation 3058
```

Vérifier l'allocation des ressources :

```
hpc01> squeue
      JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
       3058    calcul     bash  dureaud  R       1:24      2 hpc[04-05]
```

On exécute autant de fois que nécessaire le programme MPI :

```
hpc01> mpirun -n 4 ./monprog.exe
hpc01> mpirun -n 2 ./monprog.exe
hpc01> etc …
```

NE PAS OUBLIER DE RELACHER LA RESSOURCE ALLOUEE :

```
hpc01> exit
exit
salloc: Relinquishing job allocation 3058

hpc01> squeue
      JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

Travail à faire:

Question 1 : Ecrire un programme MPI où chaque processus affiche :

- son rang,
- le nombre total de processus MPI,
- et la machine hôte sur laquelle il s'exécute (fonction `MPI_Get_processor_name`)
- le processus id (pid) (fonction `getpid`).

Le tester.
Les valeurs des pids sont elles identiques ? Explication ?

Question 2 : Rajouter la déclaration d'une variable `ma_var` et afficher l'adresse de cette variable par processus.

- Les adresses affichées sont elles identiques ? Explication ?

Question 3: Rajouter une instruction printf(« Avant MPI_Init\n ») juste avant l'appel à MPI_Init.

- Combien de message « Avant MPI_Init » apparaît à l'écran en fonction du nombre de processus MPI ? Explication ?

Exercice II : 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 Vérifier qu'il s'agit bien du même contenu que celui de P1
3 Utiliser la fonction `sleep` (`#include <unistd.h>`)

Exercice III : Questions pièges

Les programmes dans les fichiers `questions_pieges/pieges/piege*.c` comportent des erreurs.
Expliquez les erreurs, apportez les corrections.

Exercice IV : Deadlock

Programmez la section de code suivante (uniquement pour deux processus MPI) :

```
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) ;
```

Dans ce programme, les processus 0 et 1 veulent s'envoyer mutuellement les `n` octets contenus dans leurs buffers d'envois respectifs `buf_send`.

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

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

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`) ;

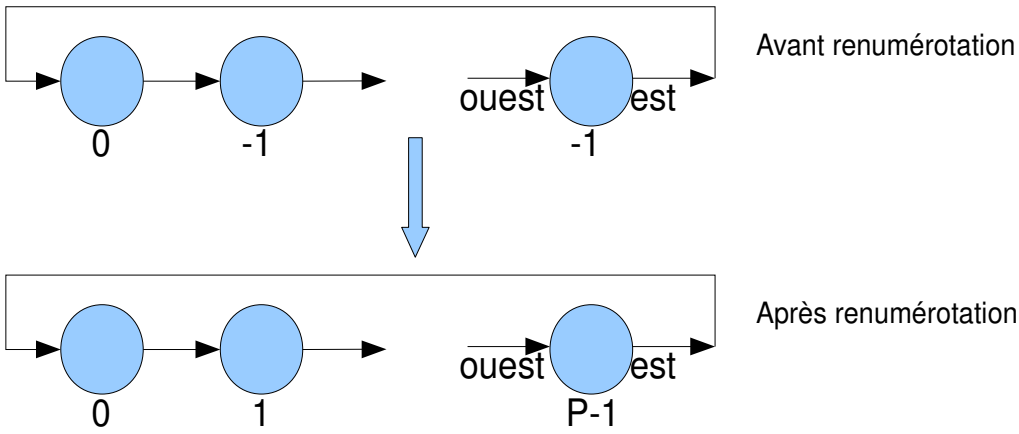
   Comment se comporte le programme dans chacun des cas a et b ? Débloquez le programme pour chacun des cas.

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

Exercice V : Renumérotation d'un anneau[4]

4 Exercice largement inspiré de celui écrit par *Stéphane Vialle, Supelec*

Soit un anneau unidirectionnel de *P* processus ayant leurs propres mémoires et ne communiquant que par envois de messages. Chaque processus possède un lien entrant (le lien *ouest*) et un lien sortant (le lien *est*), et une variable *me* contenant initialement 0 pour le processus 0, et -1 pour les autres.
Cet exercice vise à écrire une routine de numérotation des processus (initialisation correcte des variables *me*) sans connaissance du nombre de processus de l'anneau.



Dans cet exercice les fonctions *send* et *recv* permettent d'échanger facilement des *int*. On supposera les envois de messages bloquants (ex : *send(data_int, est)*). On supposera les réceptions de messages bloquantes (ex : *recv(data_int, ouest)*).

Question 1 : Écrire une routine de renumérotation (et une seule) s'exécutant dans chaque processus et qui affecte le bon numéro à chaque processus (qui écrit la bonne valeur dans chaque variable *me*), sans connaître le nombre total de processus (*P*).
Vous pouvez utiliser autant de variables, d'instructions *send* et *recv*, et d'accès à la variable *me* de chaque processus que vous le désirez. Mais vous ne devez pas supposer connu le nombre de processus avant la renumérotation, ni partager des variables entre les processus (la mémoire est purement distribuée).

MPI permet très simplement d'obtenir le rang d'un processus et le nombre total de processus. Néanmoins, nous allons écrire un programme MPI pour tester notre routine.

Question 2 : Ecrire un programme MPI qui initialise sur chaque processus les variables *me* (valeur avant renumérotation), *ouest* (rang MPI) et *est* (rang MPI).

Question 3 : Ecrire une fonction

int numerotation(int me_old, int ouest, int est)

qui retourne *me* après renumérotation. Vous devez utiliser uniquement les fonctions *MPI_Send* et *MPI_Recv*. Vérifier dans le programme principal que la valeur retournée de *me* est bien égale au rang du processus.

Question 4 : Ecrire une fonction

*void numerotation_nproc(int me_old, int ouest, int est, int *me, int *P)*

qui retourne **me* après renumérotation et le nombre total de processus dans **P*. Vérifier dans le programme principal que la valeur retournée dans **me* est bien égale au rang du processus et que celle dans **P* est bien égale au nombre total de processus MPI.

Exercice V : 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.