Introduction à la programmation MPI
TD 2 –
Communications point-à-point non-bloquantes

Exercice I : Prise en main sur les communications non-bloquantes

Question 1 : le programme `ini_nonblock/exercice/deadlock.c` présente un blocage. Résoudre ce blocage en utilisant des communications **point-à-point non-bloquantes**.

Question 2 : compléter le programme `ini_nonblock/exercice/tantque.c` (`/* TRAVAIL A FAIRE */`) en utilisant des communications **point-à-point non-bloquantes**.

Exercice II : MPI_Waitall

Ouvrir le fichier `exo_waitall/exo_waitall.c` .
Lire le travail à effectuer `/* TRAVAIL A EFFECTUER */`.

Principe général du programme à écrire :

- le processus de rang 0 doit remplir et envoyer des tableaux à tous les processus de rangs impairs (les tableaux sont construits à partir de la fonction `fill_val_array`, pour plus de détails lire le fichier `exo_waitall.c`)
- chaque processus impair doit recevoir le tableau que lui envoie le processus 0 et appelle la fonction `check_val_array` pour vérifier que le contenu est bien correct.

Question 1 : Terminer le programme en effectuant uniquement des communications **non-bloquantes**. Pour ce faire, le processus 0 doit utiliser la fonction `MPI_Waitall`.

Question 2 : Quelle est la signification du `MPI_Waitall` pour le processus 0 ? Au choix :

a) barrière sur tous les processus ?
b) ou bien attente de tous les processus impairs ?
c) ou bien attente de la fin de tous les envois vers les processus impairs ?

Question 3 : L'appel à `MPI_Waitall` par le processus 0 est-il obligatoire ?

Question 4 : Pour les processus impairs,

1. est-il possible d'utiliser uniquement des réceptions bloquantes ?
2. Dans notre cas de figure, utiliser des réceptions *bloquantes* est-il aussi performant que des réceptions *non-bloquantes* ? Justifier

Exercice III : Convolution

Les programmes de création et modification d'image tel que gimp ou photoshop permettent de faire des effets sur les images. Par exemple, l'effet de splitting consiste à remplacer la valeur d'un pixel par la moyenne des pixels voisins. Cela atténue les contrastes et donne un effet plus flou à l'image. Dans cette partie, on se limitera à une image en 1 dimension (1D).
Cet algorithme peut être modélisé de cette manière :

$$g(x) = \sum_{i=-1}^{i=1} f(x+1) * (\frac{1}{3})$$

Question 1 : le programme **`convol.c`**, fourni dans le répertoire `convolution/`, effectue une convolution d'un tableau 1D de nombres flottants croissants avec 16MB répartis sur N processus. Relever le temps que prend ce programme pour 2, 4 et 32 processus MPI (affichage « `Pt2Pt Telaps` »).

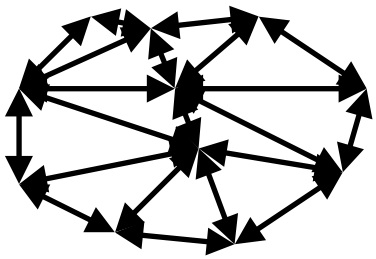
Question 2 : Transformer les communications de ce programme en des **communications non bloquantes**. Sur 100 passes du filtre, observez vous un gain ?

Question 3 : A présent on considère le programme **`convol2.c`** qui utilise des communications **bloquantes**. Dans ce programme, par passe du filtre, on applique la convolution **sur deux tableaux indépendants**.
Relever le temps que prend ce nouveau programme.
Transformer ce programme en utilisant des communications **non-bloquantes** en essayant **de recouvrir les communications par du calcul** (= effectuer du calcul pendant que les communications MPI progressent).
Comparer les performances entre les versions « bloquantes » et « non-bloquantes ».

Exercice IV : Graphe de communication

On représente par un graphe (non orienté) les communications entre *P* processus.

Chaque nœud du graphe représente un processus MPI.
Un arc entre deux nœuds définit l'existence d'envois/réceptions entre les deux processus correspondants.



Un graphe de communication est représenté par la structure suivante :

```
struct graphe_t
{
        int nb_noeuds ;

        /* tableau dimensionné à nb_noeuds
             nb_voisins[p] : retourne le nombre de nœuds directement connectés au nœud p
        */
        int *nb_voisins ;

        /* tableau à 2 dimensions
                voisins[p] : tableau dimensionné à nb_voisins[p]
                contient les numéros des nœuds directement connectés au nœud p
        */
        int **voisins ;
} ;
```

L'ensemble des voisins d'un processus p est $\{q = \text{voisins}[p][\text{iv}] \text{ où } 0 \leq \text{iv} < \text{nb\_voisins}[p]\}$.
Tout processus p ($0 \leq p < P$) doit envoyer nb_voisins[p] messages et recevoir nb_voisins[p] messages.

Pour un processus p donné, les buffers des messages à envoyer se trouvent dans le tableau char **msg_snd ; les tailles des buffers sont dans le tableau int *taille_msg_snd .
Autrement dit, le processus p doit envoyer le message msg_snd[iv] de taille taille_msg_snd[iv] au voisin q = voisins[p][iv] pour tout $0 \leq \text{iv} < \text{nb\_voisins}[p]$.

Pour un processus p donné, les buffers des messages à recevoir se trouvent dans le tableau char **msg_rcv ; les tailles des buffers sont dans le tableau int *taille_msg_rcv .
Autrement dit, le processus p doit recevoir le message msg_rcv[iv] de taille taille_msg_rcv[iv] du voisin q = voisins[p][iv] pour tout $0 \leq \text{iv} < \text{nb\_voisins}[p]$.

Soit la fonction

```
void echange( struct graphe_t *graphe,
              char **msg_snd, int *taille_msg_snd,
              char **msg_rcv, int *taille_msg_rcv ) ;
```

appelée par chaque processus p, et qui effectue les envois/réceptions définis par le graphe de communication graphe.

Travail à effectuer : Écrire la fonction echange en utilisant des communications **non bloquantes**.

Pour effectuer ce travail, répondre à ces questions :

1. pour un processus donné, déterminer le nombre de requêtes par voisin ;
2. en déduire le nombre total de requêtes gérées par un processus donné ;
3. pour un processus donné et son iv-ième voisin, déterminer le rang de ce voisin ;
4. terminer le travail.