Introduction à la Programmation MPI
TD 1 –
Prise en main et Communications Collectives Bloquantes

Exercice I : Prise en main

Se connecter sur le cluster :

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

Préparation environnement :

```
hpc01> module load mpi/mpich-3.2-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      2  drain hpc[02-03]
calcul*      up   infinite      1  drain hpc13
calcul*      up   infinite      9   idle hpc[04-12]
```

Vérifier l'allocation des ressources (nœuds en cours d'utilisation) :

```
hpc01> squeue
     JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
     12882    calcul     bash  dureaud  R       0:04      2 hpc[04-05]
```

Exécuter le programme MPI en contrôlant l'allocation de la ressource (ici 4 cœurs (-n 4) répartis sur 2 nœuds (-N 2)) :

```
hpc01> srun -n 4 -N 2  ./monprog.exe
hpc01> srun -n 2      ./monprog.exe
```

Travail à faire:

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

- son rang,
- le nombre total de processus MPI,
- 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 : Traitement image avec *collectives*

Contexte – prise en main

Allez dans le répertoire `traitement_image_coll/` .

Quelques mots sur FreeImage

Le traitement d'image s'appuie sur la bibliothèque FreeImage qui est déjà installée sur `hpc.pedago.ensiie.fr` :

```
/home/dureaud/softs/FreeImage
```

Par défaut, les makefiles utilisent cet emplacement mais si vous souhaitez faire votre propre installation vous pourrez utiliser la variable d'environnement :

```
export FREEIMAGE_ROOT=/votre/home/softs/FreeImage
```

(un fork des sources de FreeImage se trouve dans https://gitlab.com/ensiie-mpi/freeimage)

Programme séquentiel

Dans le répertoire `traitement_image_coll/sequentiel/`, le fichier `Image_seq.c` permet à partir d'une image d'entrée au format jpg `zelda.jpg` d'appliquer un traitement pour créer une nouvelle image `new_zelda.jpg` :

```
cd sequentiel/
make
./img_seq.exe ../zelda.jpg new_zelda.jpg
```

Le traitement appliqué est un décalage de 50 sur toutes les couleurs de tous les pixels (fonction `slide_effect`).

A vérifier : exécutez le programme séquentiel et visualisez les deux images et constatez que la nouvelle image est plus foncée que l'originale.

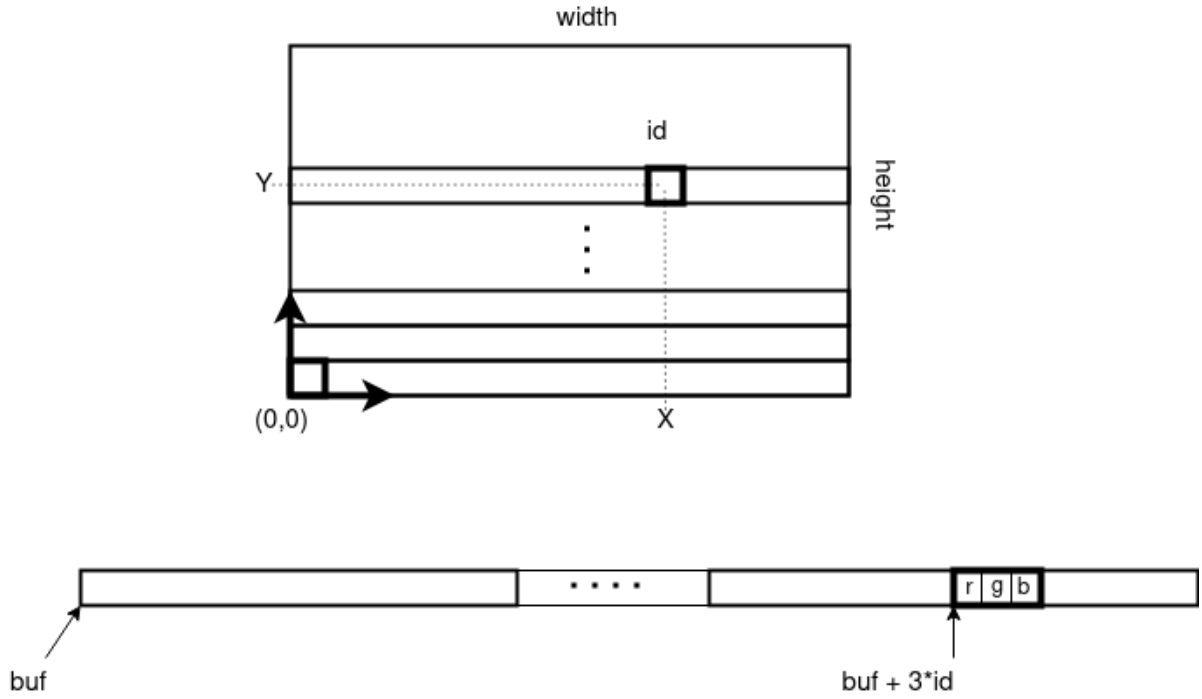

Fig 1 - Paramètres d'une image et son stockage

Exercice

A - Différencier le traitement par rang MPI

Allez dans le répertoire `slide_fx/`.

Modifiez le fichier `Image_slide_exo.c` pour que :

1. Seul le processus 0 lise l'image d'origine et la diffuse à tous les autres processus.
2. Chaque processus applique sur son image la fonction `slide_effect` avec un décalage égal à 10 fois le rang MPI.
3. Chaque processus sauve son image dans un fichier suffixé par le rang MPI.

Indication : diffusez d'abord les paramètres de l'image (voir `struct image_t` dans `utils/image.h`) avant de diffusez l'image elle-même.

B - Répartition du travail

Allez dans le répertoire `split/`.

On considère un nombre de processus MPI (`mpisize`) qui est multiple du nombre de lignes de l'image (`height`).
Dans le fichier `Image_split_exo.c`, ajoutez les fonctions MPI pour que :

1. seul le processus de rang 0 lise l'image source
2. distribue 1/`mpisize`-ième de l'image à chacun des processus, le processus 0 s'attribuant la première part
3. chaque processus sauve sa partie d'image dans un fichier dont le nom est suffixé par son rang.

Indication : dans le buffer `src.buf`, les lignes sont stockées les unes après les autres de manière contigüe (cf Fig 1)

C - Histogramme

Allez dans le répertoire `histo/` et travaillez sur le fichier `Image_histo_exo.c`.

Une fois l'image répartie entre les processus MPI (on parle d'image distribuée), calculez en parallèle un histogramme.
En découpant l'intervalle [0, 256[(qui représente les valeurs possibles de gris) en 8 intervalles réguliers, décomptez le nombre de pixels par intervalle de gris.
Pour rappel, la fonction `gray_formula(pixel_rgb)` permet de récupérer la couleur grise à partir des valeurs rgb du pixel.
Seul le processus de rang le plus élevé écrit dans un fichier `HistoNormlized.txt` les valeurs normalisées au nombre total de pixels de l'image d'origine.

D - Collecte d'un traitement distribué

Allez dans le répertoire `gray/` et travaillez sur le fichier `Image_gray_exo.c`.

Une fois l'image répartie entre les processus MPI (on parle d'image distribuée), transformez en parallèle l'image colorée en gris (ie appelez la fonction `gray_formula` pour chaque pixel).

Seul le processus 0 récupère l'image finale et l'écrit dans un fichier (le processus 0 peut réutiliser l'espace mémoire de l'image d'origine pour stocker en mémoire la nouvelle image grise reconstituée).

E - Recherche d'une couleur

Allez dans le répertoire `findcol/`.

Une image d'origine et une couleur (sous la forme de 3 entiers compris dans [0,255]) sont données en argument du programme.
Seul le processus 0 lit ces informations.

1) Diffusez la couleur avec un seul appel MPI à tous les processus (correction `Image_diffcol.c`)

2) Distribuez l'image en `mpisize` sous-parties dans le cadre général où `src.height` n'est pas forcément un multiple de `mpisize`. Pour ce faire, déterminez par processus le nombre de lignes à traiter ainsi que la position de la première ligne (correction `Image_splitv.c`).

3) Recherchez sur l'image distribuée le nombre total de pixels égaux à cette couleur, seul le processus 0 affiche cette information (correction `Image_findcol.c`, essayez sur `zelda.jpg` avec 3 procs et couleur 24 42 30)

4) Récupérez avec le processus 0 la liste des pixels identifiés par leurs identifiants globaux (ie. les id calculés à partir des dimensions de l'image d'origine) ainsi que les coordonnées globales (correction `Image_findcol2.c`)