TP MPI N°1 : Prise en main

1. **Matériels disponibles (**vous utiliserez l'**etud** au cas où les nodes ne fonctionnent pas correctement **)**
   - **node26, node28** : CentOS 7.4, Intel Xeon CPU E5-2670 (2 NUMA nodes de 8 cores physiques chacun), gcc 4.8.5, OpenMP 3.1 (201107)

   - **etud** : Fedora release 25, Intel Xeon CPU E5-2650 (2 NUMA nodes de 12 cores physiques chacun), gcc 6.4, OpenMP 4.5 (201511)

2. **Mise en place de l'environnement MPI – exécution sur un seul node**
   - Se connecter à `etud`, puis connecter vous sur `node26` ou `node28` via `ssh`.
   - OpenMPI est installé dans le répertoire : `/usr/lib64/openmpi`
     - Vérifier l'accès des commandes MPI. Vous pouvez utiliser la commande `which`. Par exemple : `which mpicc` donne `/usr/bin/mpicc`. Pour connaître les informations sur la version d'OpenMPI installée, utiliser la commande : `ompi_info`.
     - Si `which` ne trouve pas de commandes MPI. Il faudra modifier votre `.bashrc` en ajoutant les lignes suivantes :

```
if ! (which mpicc>/dev/null 2>&1) && [ -d /usr/lib64/openmpi ]
then
    export PATH=/usr/lib64/openmpi/bin:$PATH
    export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64/openmpi/lib
fi
```

     - Vous pouvez aussi modifier `CPATH` ou `C_INCLUDE_PATH` pour les fichiers d'entête.

3. **Programmation : Qui suis je ?**

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv)
{
      int  rang, nbProcs;

      MPI_Init( &argc, &argv );
      MPI_Comm_rank( MPI_COMM_WORLD, &rang );
      MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

      MPI_Get_processor_name( proc_name, &nameLength) ;
      printf( " Hello from proc. %d of %d\n ",
            rang, nbProcs);

      MPI_Finalize();
      return 0;
}
```

   - Reprendre le premier exemple de la présentation du MPI (`hello.c`).

   - Compiler ce programme avec la commande `mpicc -o hello hello.c`. Vous pouvez ajouter les options usuelles de compilation de C.
     - Exécuter le programme sur le nœud courant : `mpirun -np 4 hello`
     - Ajouter la fonction `MPI_Get_processor_name` qui vous permet de connaître le nom du nœud sur lequel s'exécute un processus

- Utiliser la fonction `sched_getcpu()` pour récupérer le numéro du CPU sur lequel s'exécute un processus. Le fichier d'entête correspondant est `<sched.h>`.

4. Exécution d'un programme sur plusieurs nodes

- MPI utilise `ssh` pour l'exécution des processus distants. Si le serveur vous demande d'entrer le mot de passe à chaque exécution pour chaque nœud concerné, vous pouvez utiliser le procédé suivant pour éviter cela.
  On va utiliser `ssh` **sans passphrase** :
  - A la racine de votre répertoire privé, créer une paire de clés privée/publique **sans passphrase** à l'aide de :
    ```
    $ ssh-keygen -t rsa
    Enter file in which to save the key : ⏎
    Enter passphrase (empty for no passphrase): □
    Enter same passphrase again: ⏎
    ……
    $
    ```
  - Copier/ajouter la clé publique dans le fichier `~/.ssh/authorized_keys` :
    ```
    $ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
    ```
  - Les clés est créées et prêtes à être utilisées.
  - Fermer l'accès de votre répertoire privé par les autres pour plus de sécurité
    ```
    $ chmod 700 /home/etud/$USER
    ```
  - A la première connexion à un nœud, répondre 'yes' au processus d'authentification afin qu'il ajoute le serveur courant dans la liste des known_hosts du nœud.
- Exécution d'un programme MPI sur plusieurs nodes :
  - Editer un fichier des nodes (ex. `myhostfile`)
    ```
    $ cat myhostfile
      node26
      node28
    $
    ```
  - Exécution du programme sur plusieurs nodes :
    ```
    $ mpirun –np 64 –hostfile myhostfile hello
      ……
    $
    ```
  - Modifier le fichier `myhostfile` afin de limiter le nombre de processus sur les nodes :
    ```
    $ cat myhostfile
      node26 max_slots=16
      node28 max_slots=16
    $
    ```
  - `slots` / `max_slots` indique le nombre (maximal) de processeurs physiques d'un node. Avec ce fichier, `16` sera le nombre maximal de processus que vous pouvez exécuter sur un node (car un processus / processeur), donc la valeur maximale que vous pouvez spécifier avec `–np` sera `16`.

5. Programmation : Communication point-à-point

- Reprendre le deuxième exemple du cours, qui porte le nom « `p2p.c` ».

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
      int        rang, nbProcs, dest=0, source, etiquette = 50;
      MPI_Status statut;
      char       message[100];

      MPI_Init( &argc, &argv );
      MPI_Comm_rank( MPI_COMM_WORLD, &rang );
      MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );
      if ( rang != 0 ) {
         sprintf(  message, "Bonjour de la part de P%d!\n" , rang
         );
         MPI_Send( message, strlen(message)+1, MPI_CHAR,
                   dest, etiquette, MPI_COMM_WORLD );
      }
      else
         for ( source=1; source<nbProcs; source++ ) {
             MPI_Recv( message, 100, MPI_CHAR, source,
                       etiquette, MPI_COMM_WORLD, &statut );
             printf( "%s", message );
         }
      MPI_Finalize();
      return 0 ;
}
```

- Compiler le programme, puis l'exécuter.
- Remplacer le paramètre `source` de la fonction `MP_Recv` par `MPI_ANY_SOURCE`, exécuter plusieurs fois le programme et analyser les résultats d'affichage.

6. Modification du deuxième exemple

Soit `N` le nombre de processus d'une exécution,

- On demande de les organiser en anneau comme dans la figure 1.

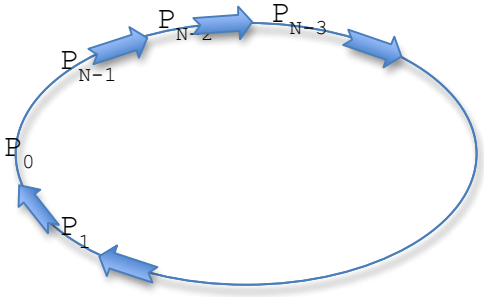

Figure 1. Organisation en anneau des processus

Le message de P_{N-1} est envoyé à P_{N-2}, concaténé au message de P_{N-2}, puis P_{N-2} envois le message à P_{N-3}, ainsi de suite jusqu'à P_0. P_0 reçoit le message de P_1 et l'afficher dans le terminal. Le message affiché prend la forme suivante :

`Bonjour de la part de` P_1, P_2, P_3, … P_{N-1}!

- Refaire ces communications avec un arbre binaire avec $N=2^n$, comme montre la figure 2. (optionnel)

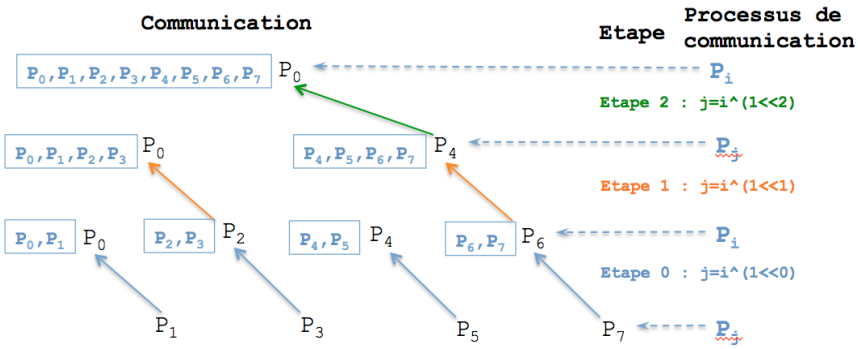

Figure 2. Organisation en arbre binaire des processus