Programmation Parallèle et Distribuée
MPI - TD 1

Exercice I : Prise en main

Préparation environnement (sur les serveurs aquitaine, etc...) [1] :

export *MPI_PATH*=/users/ens/dureau/mpich-3.0.1/bin
export PATH=$*MPI_PATH*:$PATH

Compilation (se comporte comme un compilateur classique) :

prompt> ***mpicc*** monprog.c -o monprog.exe

On execute autant de fois que nécessaire le programme MPI :

prompt> ***mpiexec*** -n 4 ./monprog.exe
prompt> ***mpiexec*** -n 2 ./monprog.exe
prompt> etc …

Travail à faire:

Ecrire un programme MPI où chaque processus affiche :

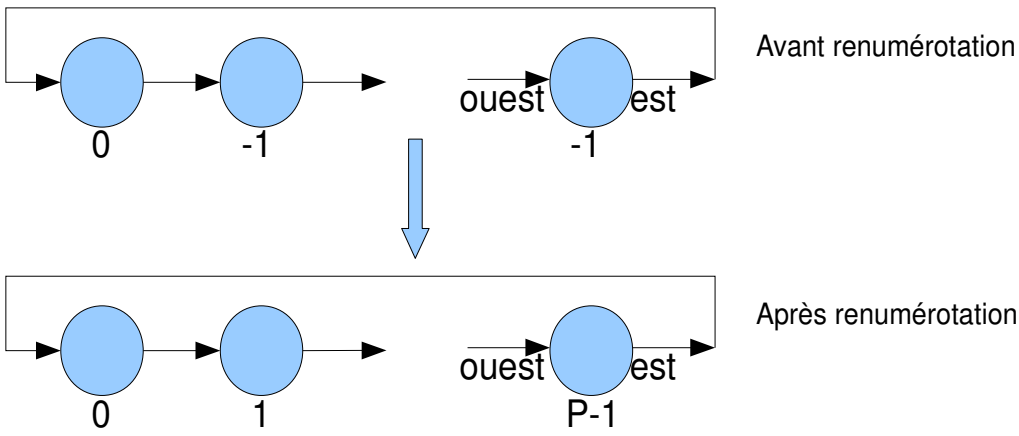
- son rang,
- le nombre total de processus MPI,
- et la machine hôte sur laquelle il s'execute (fonction *gethostname*, include *unistd.h*).

Le tester.

Exercice II : Renumérotation d'un anneau[2]

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

1 Pour installer MPI chez soi, lire en fin de TD « Note à ceux qui veulent installer MPI sur leurs ordinateurs »
2 Exercice largement inspiré de celui écrit par *Stéphane Vialle, Supelec*



On supposera les envois de messages bloquants non synchrones (ex : *send(est,&data_int)*), et avec sauvegarde des données à envoyer par recopie immédiate dans des tampons implicites. On supposera les réceptions de messages bloquantes (ex : *recv(ouest,&data_int)*). Dans cet exercice ces opérations permettent d'échanger facilement des *int*.

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

Une barrière est un point de synchronisation globale (*i.e.* synchronisation entre toutes les tâches parallèles).
Une tâche pourra quitter la barrière une fois que toutes les autres tâches seront entrées dans la barrière.
Soit void barrier() une fonction implémentant une barrière.

1. Proposez une implémentation de la fonction barrier() en utilisant un pseudo-langage parallèle (utilisez les primitives **send/recv*) :
   a. Pour 2 tâches parallèles :
      i - avec *ssend* : envoi bloquant synchrone ;
      ii - avec *bsend* : envoi bloquant bufferisé ;
   b. Pour *P* tâches parallèles :
      i - avec *ssend* : envoi bloquant synchrone ;
      ii - avec *bsend* : envoi bloquant bufferisé ;

2. Écrire la fonction barrier() pour *P* processus MPI (utilisez les fonctions MPI_Send et MPI_Recv) ;

3. Écrire un programme exécutable MPI utilisant une barrière sur tous les processus.

Note à ceux qui veulent installer MPI sur leurs ordinateurs (*facultatif*, ceci n'est pas un exercice et *n'est pas prioritaire* sur le travail en TD)

Récupérer /users/ens/dureau/PUBLIC/mpich-3.0.1.tar.gz ou bien télécharger-le sur le web
> tar cvzf mpich-3.0.1.tar.gz. Le dossier créé est le répertoire de compilation que l'on pourra détruire après installation définitive.

> cd mpich-3.0.1

Configuration :
> ./configure --prefix=*/usr/local/mpich-3.0.1* --disable-cxx --disable-f77 –disable-fc --disable-romio --enable-threads=runtime
Le répertoire désigné par --prefix= sera le répertoire d'installation définitive, le chemin donné ici n'est qu'un exemple.

Compilation (plusieurs minutes) :
> make

Installation (en fonction du répertoire final d'installation peut nécessiter les droits root) :
> sudo make install