Environnent de développement

Le langage choisi est C, utilisé, conjointement aux primitives offertes par la bibliothèque de communication par échange de message MPI. L'acronyme MPI signifie Message Passing Interface, spécification d'environnements de programmation pour systèmes à mémoire distribuée. Dans le cadre des TPs, nous utiliserons l'implémentation OpenMPI (http://www.open-mpi.org).

Exécuter un programme openMPI nécessite l'inclusion des fichiers d'entête dans lesquels se trouvent les définitions des différentes primitives de MPI. Il existe 6 commandes de base :

- *MPI _Init* qui initialise un bloc de calcul utilisant MPI.
- *MPI_Finalize* qui termine un bloc de calcul MPI.
- *MPI_COMM* size qui détermine le nombre de processus qui seront engagé dans un calcul.
- *MPI_COMM rank* qui détermine l'identificateur d'un processus.
- *MPI_Send* primitive d'envoi de messages.
- *MPI_Recv* primitive de reception de messages.

***Fonctions de base MPI*[1]**

L'appel à des fonctions OpenMPI doit obligatoirement être précédé d'un appel à *MPI_Init* et se terminer par un appel à *MPI_Finalize*. Un programme MPI se décompose donc généralement de la manière suivante:

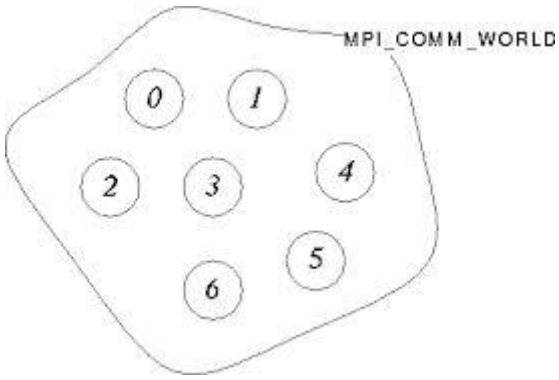
```
#include <mpi.h>
int main(int argc, char *argv[]){
   MPI_Init(&argc, &argv);
   /* CALCUL */
   MPI_Finalize();
```

[1] Pour plus d'information voire : http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html
Plusieurs exemples (source code) se trouvent :
http://www.cse.iitd.ernet.in/~dheerajb/MPI/Document/hos_cont.html
https://computing.llnl.gov/tutorials/mpi/exercise.html
http://static.msi.umn.edu/tutorial/scicomp/general/MPI/communicator.html
https://www.nics.tennessee.edu/mpi-tutorial

Pour mesurer les performances d'un programme, il est utile de chronométrer son temps d'exécution. On peut faire appel pour cela à la fonction *MPI_Wtime()* qui retourne le temps écoulé en secondes dans le processus courant.

Les communicateurs

Tous les appels à des fonctions MPI se basent sur la notion de communicateurs. Un communicateur est un paramètre utilisé pour spécifier à la fonction appelée l'ensemble des processus auxquels s'adresse l'appel. Un communicateur particulier appelé *MPI_COMM_WORLD* inclut tous les processus MPI en cours d'exécution liés au processus appelant.



Les primitives *MPI_COMM size()* et *MPI_COMM rank()* permettent à chaque processus de connaitre son environnement d'exécution, il obtient son propre *identifiant* et le *nombre de processus* qui exécutent le même programme que lui. Par exemple:

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

aura pour effet de stocker le nombre de processus liés au communicateur *MPI_COMM_WORLD* (c'est-à-dire tous les processus) et de stocker ce nombre dans la variable *size*.

À chaque processus est associé un identifiant appelé *rang*; ce rang, variant de *0 à (size-1)*, est propre à chaque communicateur. Un processus peut connaître son *rang* au sein d'un communicateur en faisant appel à la fonction *MPI_Comm_rank*:

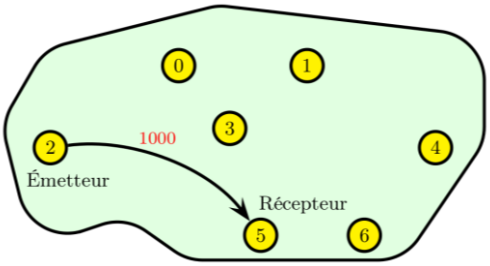
```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Le *rang* permet au programmeur d'identifier la source et la destination d'un message ou encore de séparer l'ensemble du calcul entre les différents processus.

Communications point à point entre processus

Les processus ont aussi besoin de communiquer entre eux. Par exemple, supposons qu'un processus *p1* veuille communiquer un résultat à un processus *p2*, il va utiliser un mode de communication de type point-à-point, c'est-à-dire un envoi de message simple. Pour qu'un processus émetteur puisse envoyer un message à un processus destinataire, différentes informations sont nécessaires:

- le *rang* du processus émetteur dans le communicateur utilisé;
- le *rang* du processus destinataire dans le communicateur utilisé;
- l'*étiquette* du message (valeur entière choisie par le programmeur);
- le *nom* du communicateur.



Les primitives utilisées pour ce type de communication sont :

*MPI_Send(void *,int, MPI Datatype, int, int, MPI Comm)* :

1. adresse du buffer d'émission,
2. nombre d'éléments de type datatype à envoyer/recevoir
3. type des éléments à envoyer/recevoir (parmi MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE);
4. destinataire (il s'agit de son rang),
5. tag (permet de différencier les messages entre eux), éventuellement MPI_ANY_TAG
6. communicateur (e.g. MPI_COMM_WORLD);

*MPI_Recv(void *, int, MPI Datatype, int, int, MPI Comm, MPI Status*)* :

1. adresse du buffer de reception,
2. nombre d'éléments,
3. type des éléments,
4. source, rang du processus dont provient le message,
5. tag, permet de différencier les messages à la reception,
6. status : état de la communication. Peut être ignoré avec MPI STATUS IGNORE

La fonction *MPI_Get_processor_name* permet d'identifier le nom de la machine exécutant le processus courant.

```
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Get_processor_name(processor_name,&namelen);
```

Génération de clés SSH pour MPI

Open MPI lance les processus sur les machines distantes en utilisant SSH (Secure SHell). Par défaut, SSH vous demande votre mot de passe durant la connexion, ce qui est gênant dans le cadre de l'utilisation de MPI. Une solution consiste à générer une clé SSH et à l'autoriser sur les machines distantes, la connexion se fera alors directement, sans demande de mot de passe.

Pour ce faire, utilisez simplement la commande suivante pour la génération des clés. Acceptez le chemin par défaut pour sauvegarder la clé ([$HOME/.ssh/id_dsa]) et entrez une phrase de passe pour votre paire de clés. Vous pourriez ne pas entrer de phrase de passe, auquel cas l'utilisation de *ssh-agent* n'est plus nécessaire. Cependant, ceci rendrait l'authentification plus faible puisque votre clé secrète ne serait pas encodée. Il est donc fortement décourager de le faire.

```
ssh-keygen -t dsa
```

Ensuite, il est nécessaire de ne pas mettre votre clé dans les clés acceptées sur toutes les machines sur lesquelles vous comptez faire tourner MPI. Pour ce faire, utilisez les commandes suivantes pour l'autorisation de la clé. La commande *scp* permet de transférer des fichiers d'une machine à une autre sur le réseau, de manière sécurisée. Dans la commande, remplacez *distant_machine* par l'IP ou le nom de la machine distante. Il faut exécuter la commande *scp* pour chaque machine distante où vous désirez faire tourner MPI.

```
cp ~/.ssh/id_dsa.pub ~/.ssh/authorized_keys
scp -r ~/.ssh distant_machine:
```

Vous avez désormais fait le nécessaire afin d'utiliser l'authentification DSA. Cependant, votre phrase de passe pour la clé vous sera demandée lors de la connexion (au lieu de votre mot de passe habituel). C'est pour cela que nous avons besoin d'utiliser *ssh-agent*. Pour lancer *ssh-agent*, utilisez la commande suivante :

```
eval ‘ssh-agent‘
```

Une fois *ssh-agent* lancé, il suffit de lui indiquer votre phrase de passe en utilisant la commande suivante.

```
ssh-add $HOME/.ssh/id_dsa
```

Vous devriez désormais pouvoir vous connecter sur les machines distantes (où vous avez copié votre dossier .ssh) dans phrase de passe. Testez qu'aucun mot de passe ne vous est demandé lors de l'utilisation de la commande suivante.

```
ssh machine_distante
```

Exercice 1 : Installation de OpenMPI

Un tutorial pour installer OpenMPI for Ubuntu:

http://edu.itp.phys.ethz.ch/hs12/programming_techniques/openmpi.pdf

Compilation/exécution:

Pour compiler les programmes utilisant la bibliothèque OpenMPI, il suffit d'utiliser le compilateur fourni par Open MPI : *mpicc*. Les paramètres sont similaires à ceux de *gcc*.

```
mpicc hello_c.c -o hello_c
```

L'exécution de programme MPI est réalisée à l'aide du programme *mpirun*.

```
mpirun -hostfile hosts -np 3 hello_c
```

Les principaux paramètres de *mpirun* sont :

- np <#> : indique le nombre de processus à utiliser
- hostfile <path> : indique le chemin vers le fichier contenant la liste des machines
- nolocal : indique a MPI de ne pas lancer de processus sur la machine locale

Exemple:

L'exemple suivant présente un simple programme de Hello World distribué utilisant MPI. Ce code permet de demander à chaque processus d'afficher le texte "Hello World" ainsi que son *rang* et le *nombre total de processus*.

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
    int rank, size;
  // Initialisation de la biblioth`que MPI
   MPI_Init(&argc, &argv);
  // Quel est mon rang ?
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 // Combien de processus dans l'application ?
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   printf("Hello, world, I am %d of %d\n", rank, size);
  // Finalisation du programme
  MPI_Finalize();
  return 0;
}
```

Exercice 2

Modifiez le programme précédent pour que les processus pairs n'affichent pas le même message que les processus impairs.

- Soit par exemple pour les processus de rang pair un message du genre :
  Je suis le processus pair de rang M
- Et pour les processus de rang impair un message du genre :
  Je suis le processus impair de rang N

Exercice 3

a. Ecrivez un programme dans lequel le processus 0 envoie un tableau de 10 réels au processus 1. Le processus 0 remplira le tableau, et le processus 1 affichera ses valeurs. Dans ce cas, on fera uniquement un ping (envoi d'un message du processus 0 au processus 1). Par exemple, envoyer un message contenant 1000 nombres réels du processus 0 vers le processus 1 (il s'agit alors seulement d'un ping).

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


int main(int argc, char *argv[]) {
  int rang,iter;
  int nb_valeurs=1000;
  int etiquette=99;
  double valeurs[nb_valeurs];
  MPI_Status statut;

  /* ? */

  if (rang == 0) {
    for (iter = 0; iter<nb_valeurs; iter++)
      valeurs[iter] = rand() / (RAND_MAX + 1.);
    /* ? */
  } else if(rang == 1) {
    /* ? */
    printf("Moi, processus 1, j'ai recu %d valeurs (derniere = %g)"
           "du processus 0.\n", nb_valeurs, valeurs[nb_valeurs-1]);
  }

  /* ? */
  return 0;
}
```

b. Modifiez le programme précédent pour que 1 renvoie les données modifiées à 0. Modifiez ensuite le programme pour prendre le temps de communication (grossissez la taille du message). Dans ce cas, on enchaînera le pong après le ping (le processus 1 renvoyant le message reçu du processus 0). Le processus 1 renvoie le message reçu au processus 0 et mesurer le temps de communication à l'aide de la fonction *MPI_WTIME()*.

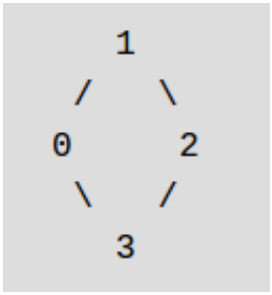
```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>


int main(int argc, char *argv[]) {
  int rang,iter;
  int nb_valeurs=1000;
  int etiquette=99;
  double valeurs[nb_valeurs];
  MPI_Status statut;
  double temps_debut,temps_fin;

  /* ? */

  if (rang == 0) {
    for (iter = 0; iter<nb_valeurs; iter++)
      valeurs[iter] = rand() / (RAND_MAX + 1.);
    temps_debut=MPI_Wtime();
    /* ? */
    temps_fin=MPI_Wtime();
    printf("Moi, processus 0, j'ai envoye et recu %d valeurs"
           "(derniere = %g) du processus 1 en %f secondes.\n",
          nb_valeurs, valeurs[nb_valeurs-1], temps_fin-temps_debut);
    /* ? */
  }

  /* ? */
  return 0;
}
```

c. Modifiez le programme précédent pour générer une courbe de temps de communications. Dans ce cas, on effectuera une répétition du ping-pong en faisant varier à chaque fois la taille du message à échanger, càd on fait varier la taille du message dans une boucle et mesurer les temps de communication respectifs.

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


int main(int argc, char *argv[]) {
  int rang,iter,iter2;
  int nb_tests=9;
  int nb_valeurs[nb_tests];
  int nb_valeurs_max=7000000;
  int etiquette=99;
  double *valeurs;
  MPI_Status statut;
  double temps_debut,temps_fin;

  /* ? */

  nb_valeurs[0]=0,nb_valeurs[1]=1,nb_valeurs[2]=10,nb_valeurs[3]=100;
  nb_valeurs[4]=1000,nb_valeurs[5]=10000,nb_valeurs[6]=100000;
  nb_valeurs[7]=1000000,nb_valeurs[8]=7000000;

  /* ? */

  valeurs = (double *) malloc(nb_valeurs_max*sizeof(double));

  for(iter=0; iter<nb_tests; iter++) {
    if (rang == 0) {
      for (iter2 = 0; iter2<nb_valeurs[iter]; iter2++)
        valeurs[iter2] = rand() / (RAND_MAX + 1.);
      temps_debut=MPI_Wtime();
      /* ? */
      temps_fin=MPI_Wtime();
      if (nb_valeurs[iter] != 0) {
        printf("Moi, processus 0, j'ai envoye et recu %8d valeurs"
               "(derniere = %4.2f) du processus 1 en %8.6f secondes, soit "
               "avec un debit de %7.2f Mo/s.\n",
               nb_valeurs[iter], valeurs[nb_valeurs[iter]-1],
               temps_fin-temps_debut,
               2.*nb_valeurs[iter]*8/1000000./(temps_fin-temps_debut));
      } else
        printf("Moi, processus 0, j'ai envoye et recu %8d valeurs en %8.6f "
               "secondes, soit avec un debit de %7.2f Mo/s.\n",
               nb_valeurs[iter], temps_fin-temps_debut,
               2.*nb_valeurs[iter]*8/1000000./(temps_fin-temps_debut));
    } else if(rang == 1) {
      /* ? */
    }
  }

  /* ? */
  return 0;
}
```

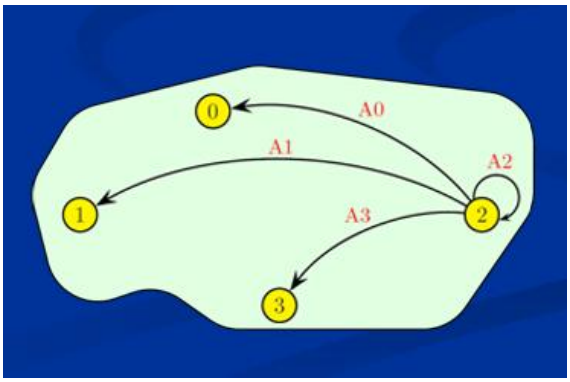
Exercice 4

Ecrire un programme permettant aux processus organisés en structure d'anneau (ring) de faire circuler entre eux un message. En fait, le processus principal (possédant le rang 0) initialise la variable message à 10 puis il fait passer le message au processus suivant. Ensuite, chaque processus fait passer le message au processus d'après de manière à ce que le message fasse le tour de l'anneau. À chaque passage du message, au niveau du processus principal, celui-ci décrémente la valeur du message. Lorsque le message atteint la valeur 0, les processus transmettent une dernière fois le message avant de s'arrêter. Modifier ce programme pour faire circuler le *rang* de chaque processus et afficher la *somme* des rangs de tous les processus. Dans la figure suivante, le message est envoyé de 0 à 1, puis de 1 à 2, puis de 2 à 3. La somme des rangs est 6.



Exercice 5[2]

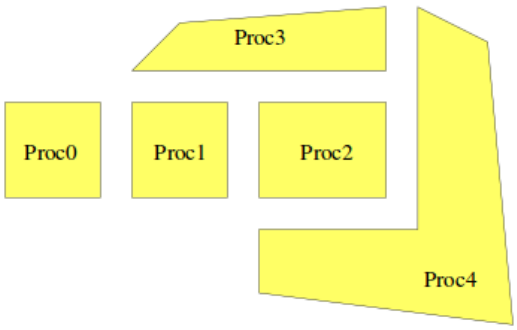
Ecrire un programme permettant à un nœud maître d'envoyer un entier lu sur l'entrée standard à chaque esclave.



Exercice 6

MPI permet de définir des topologies virtuelles de type cartésien ou graphe. Dans une topologie de type graphe chaque processus peut avoir un nombre quelconque de voisins.

[2] http://igm.univ-mlv.fr/~dr/XPOSE2006/BACHIMONT_BRUNET_PIASZCZYNKSI/prog_d2.htm



Numero Proc	Numero proc voisins
0	1
1	0, 2, 3
2	1, 3, 4
3	1, 2, 4
4	2, 3

nb_procs=5 index(1, 4, 7, 10, 12) liste_voisins=(1, 0,2,3, 1,3,4, 1,2,4, 2,3)

La fonction MPI_GRAPH_CREATE(comm_ancien, nb_procs, index, liste_voisins, reorganisation, comm_nouveau, code) permet de créer ce type de topologie[3].

Ecrivez un programme dans lequel on crée une topologie virtuelle (graphe). Faites afficher les informations suivantes : le rang de chaque nœuds et ses voisins.

Exercice 7

Implémenter 4 des 6 problèmes suivants (cf TDs et cours):

- Propagation contrôlée
- Calcul du nombre de processus à l'aide d'un parcours en largeur
- Identification des processus à l'aide d'un parcours en largeur
- Arbre de hauteur minimale
- Election de processus dans un anneau bidirectionnel
- Algorithme de Naimi trehel

[3] https://www.ljll.math.upmc.fr/SGI/pdf/pau-mpi.pdf