TP 0 – Systèmes et Traitements Répartis GIS 5
Configurer MPI

Les programmes d'aujourd'hui nécessitent de plus en plus de calculs et de mémoires. Afin d'améliorer les performances d'exécution de ces programmes, on peut paralléliser le code afin de répartir les calculs sur différent(e)s machines/processeurs.

L'objectif des séances de TP de Système et Traitements Répartis est de se familiariser avec les fonctions de la bibliothèque MPI afin de rédiger du code parallèle. Nous verrons dans un premier temps comment communiquer avec d'autres processus, puis quelques fonctions classiques utilisées en MPI, et enfin nous travaillerons sur l'élaboration d'un code permettant d'effectuer des produits ''matrice – vecteur''.

Configurer MPI

0) Nous utilisons dans ce TP le package *openmpi*, qui repose sur *ssh*. Pour les installer chez vous, entrez en ligne de commande :
sudo apt-get install libopenmpi-dev openmpi-bin openmpi-doc
sudo apt-get install ssh

1) Générez une clé ssh sur votre compte en tapant '*ssh-keygen*' dans le terminal puis en validant sans mot de passe. Copiez *.ssh/id_rsa.pub* dans *.ssh/authorized_keys* pour autoriser cette clé à se connecter à toutes les machines.

2) Ajoutez la ligne suivante dans le fichier *.ssh/config* : *StrictHostKeyChecking no*. Cela évitera de devoir répondre '*yes*' à *ssh* à chaque fois que l'on se connecte à une machine sur laquelle on ne s'était pas connecté avant (vérifiez que c'est bien le cas en se connectant à une autre machine).

3) Forcer *openmpi* à n'utiliser que eth0 comme interface réseau en créant le fichier *.openmpi/mca-params.conf* contenant la ligne : *btl_tcp_if_include = eth0* .

4) Créer un fichier *host_file* contenant le nom des machines à utiliser :

host_file
hosts machineA slots=2 machineB slots=2 machineC slots=2

Les machines de cette salle étant des dual core, on se limitera à l'utilisation de 2 cœurs par machine avec *slots=2*.
Ainsi lorsque l'on voudra utiliser des processus sur ces différentes machines, les deux premiers seront sur machineA, les deux suivants sur machineB, les deux suivants sur machineC...

Vos machines s'appellent :
- gedeon01, …, gedeon14 (en C101)
- clodion01, …, clodion14 (en C102)
- phinaert01, …, phinaert14 (en C103)

Mais le réseau ne s'arrête pas au pas-de-porte !

TP 1 – Systèmes et Traitements Répartis GIS 5
Utilisation basique de MPI

Exercice 1 : un premier code

1) Déterminer ce que fait le code suivant, le rôle de chaque processus.
2) Que peut-on dire à propos du temps d'exécution ?

Compilation : mpicc -o code code.c

Exécution : mpirun -n 5 -hostfile my_hostfile ./code

Quelques options intéressantes : *-H machineA,machineB* (au lieu de -hostfile my_hostfile)
--display-map (affiche les processus utilisés)

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

int main(int argc, char* argv[]){
  int rang, Nprocs, val, val_recue, cible, etiquette;
  double debut, fin;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  debut = MPI_Wtime();
  MPI_Comm_rank(MPI_COMM_WORLD, &rang);
  MPI_Comm_size(MPI_COMM_WORLD, &Nprocs);

  if (rang == 0){
      val = 512; etiquette = 1; cible = 3;
      MPI_Send(&val, 1, MPI_INT, cible, etiquette, MPI_COMM_WORLD);
      printf("Processus[%d] : envoie la valeur %d.\n", rang, val);
  }
  else if (rang == 3){
    MPI_Recv(&val_recue, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("Processus[%d] : reçoit la valeur %d.\n", rang, val_recue);
  }

  fin = MPI_Wtime();
  printf("Processus[%d] : temps d'exécution = %fs \n", rang, fin-debut);

  MPI_Finalize();
  return 0;
}
```

Exercice 2 : Gestion de l'environnement MPI

Écrire un code qui fait afficher un message par chacun des processus, mais différent selon qu'il soit de rang pair ou impair.

Exercice 3 : Communication Ping-pong

Envoyer un tableau contenant 1000 nombres flottants du processus 0 vers le processus 1 puis faire renvoyer ce tableau reçu par le processus 1 vers le processus 0. Mesurer le temps de communication à l'aide de la fonction `MPI_Wtime()`.

Exercice 4 : Communication par anneaux

Écrire un code où le processus de rang 0 contient un jeton (nombre entier) qui va être mis en circulation entre `N` processus qui se le transmettent successivement dans l'ordre de leur rangs :
`0 → 1 → 2 → … → N - 1 → 0`.

TP 2 – Systèmes et Traitements Répartis GIS 5
Fonctions classiques de MPI

Exercice 1 : Envoyer le même message à plusieurs processus

`MPI_Bcast` *est une fonction One-to-all où un processus partage une même donnée à tous les processus (lui y compris). Par exemple,* `MPI_Bcast(message, length, MPI_CHAR, 0, MPI_COMM_WORLD);` *signifie que le processus de rang 0 envoie la variable* `message` *de type* `MPI_CHAR` *et de taille* `length` *à chaque processus inclus dans le groupe de processus* `MPI_COMM_WORLD`.

Attention : pour un envoi de chaîne de caractère, il faut compter le caractère `'\0'`, *ignoré par la fonction* `strlen()`.

Écrire un code utilisant `MPI_Bcast` où un processus envoie aux autres la date d'aujourd'hui, allouée dynamiquement. Les processus la recevant doivent afficher un message indiquant leur rang et la date reçue. On remarque que la taille de la date n'est pas connue (à moins que ?...).

Exercice 2 : Distribuer des messages différents à plusieurs processus

`MPI_Scatter` *est une fonction One-to-all où un processus diffuse une donnée sur tous les processus (lui y compris).* `MPI_Scatter(T, 10, MPI_INT, local_T, 10, MPI_INT, proc, MPI_COMM_WORLD);` *signifie que le processus* `proc` *répartit des morceaux du tableau* `T` *par paquets de 10 éléments sur les tableaux* `local_T` *de chaque processus inclus dans la groupe de processus.*

Écrire un code où un processus crée un tableau aléatoire de taille 1000 et envoie aux autres processus une partie différente de ce tableau avec `MPI_Scatter`. Chacun des processus affichera le morceau de tableau reçu.

Exercice 3 : Rassembler les messages de plusieurs processus

`MPI_Gather` *est une fonction All-to-one où un processus rassemble les données des autres (lui y compris).* `MPI_Gather(partial_T, N, MPI_INT, T, N, MPI_INT, proc, MPI_COMM_WORLD);` *signifie que* `proc` *va joindre chaque tableau* `partial_T` *de* `N` *éléments qu'il reçoit pour en former plus grand :* `T`.

On reprend le code de l'exercice 2. On veut ensuite que chaque processus calcule la somme des éléments de `local_T` et envoie ensuite son résultat au processus de rang 0 dans un nouveau tableau `U`. On calculera ensuite la somme des éléments de `U`.

Exercice 4 : Réduction et opérations directes

`MPI_Reduce` *est une fonction All-to-one où un processus fait une opération sur les données qu'il reçoit des autres. Quelques exemples d'opérations possibles avec* `MPI_Reduce` *:* `MPI_SUM` *(somme les éléments),* `MPI_PROD` *(multiplie les éléments),* `MPI_MAX` *(cherche le maximum des éléments),* `MPI_MIN` *(cherche le minimum des éléments).*

`MPI_Reduce(&terme, &resultat, 1, MPI_INT, MPI_PROD, proc, MPI_COMM_WORLD);` *signifie que* `proc` *va effectuer le produit de tous les* `terme` *reçus et les stockera dans* `resultat`.

Faire l'exercice 3 en utilisant `MPI_Reduce`.

TP 3 – Systèmes et Traitements Répartis GIS 5
Opérations entre matrices et vecteurs

Il existe d'autres fonctions, plus élaborées, qui peuvent s'avérer utiles : `MPI_Allgather`, `MPI_Allreduce`, `MPI_ReduceScatter`, `MPI_Alltoall`. *N'hésitez pas à les découvrir dans la documentation MPI. Vous pouvez utiliser certaines d'entre elles dans ce TP.*

Vous pouvez récupérer un code source sur la page web de Charles Bouillaguet, vous permettant de vous concentrer plus sur l'essentiel de l'exercice que sur de la programmation rudimentaire en C.

Exercice 1 : Produit matrice – vecteur *(A*x)*
Le but de cet exercice est d'écrire un code le plus optimal possible effectuant un produit matrice – vecteur (*A*x*) de trois façons différentes. Les coefficients de la matrice *A* et du vecteur *x* seront des doubles construits aléatoirement sur les processus qui en auront besoin (créer des fonctions à cet effet).

1ère méthode : Distribuer *A* par blocs de lignes sur les différents processus.
2ème méthode : Distribuer *A* par blocs de colonnes sur les différents processus.

Comparer ces méthodes, en terme de communication et de temps d'exécution.

Exercice 2 : *A*(A*x+x) + x*
Écrire un code effectuant l'opération *A*(A*x+x) + x* pour une matrice *A* et un vecteur *x* étant des doubles construits aléatoirement à l'aide des deux méthodes de l'exercice 1.

Exercice 3 : Calcul du vecteur propre dominant d'une matrice stochastique *(A*...*A*A*x)*
Dans cette exercice, *A* est une matrice stochastique, c'est-à-dire une matrice telle que la somme des coefficients de chaque ligne vaut 1 et chaque coefficient est positif ou nul. Les coefficients de la matrice stochastique *A* et du vecteur *x* unitaire (de norme 1) seront des doubles construits aléatoirement par chaque processus (reprendre les fonctions de l'exercice 1).
Le but de cet exercice est de calculer le vecteur propre dominant de *A* qui s'obtient à l'aide de l'algorithme suivant :

```
tant que ||y - x|| > epsilon
       x = y
       y = A*x
       y = y / ||y||
return y
```

1) Créer une fonction retournant la norme d'un vecteur.
2) Écrire deux versions du code, inspirées chacune d'une méthode de l'exercice 1, les plus optimales possible, calculant le vecteur propre dominant de *A*.

Exercice 4 : 3ème méthode – distribuer A par blocs sur les différents processus (facultatif)
Refaire les exercices 1 à 3 avec cette méthode. Cet exercice est difficile et facultatif.

Voici des pistes de réflexions :
1) Pour bien gérer les communications, on peut créer de nouveaux communicateurs qui communiqueront entre eux et pas avec les autres (des familles de processus). On pourra s'intéresser aux fonctions suivantes : `MPI_Cart_Create`, `MPI_Cart_coords`, `MPI_Comm_split`.
2) On peut aussi considérer une matrice à 3 dimensions : les 2 premières dimensions sont les coordonnées des processus, la 3ème la sous-matrice de A.