TME 1 - Introduction à MPI en utilisant **openmpi**

1 Introduction générale

Nous allons aborder la programmation d'applications réparties avec la librairie de communication par messages **MPI**. Une application répartie est composée d'un ensemble de processus communiquant soit au moyen d'une mémoire partagée, soit par échange de messages. C'est à ce deuxième type d'applications que nous nous intéresserons dans le cadre de ce module.

2 Introduction à MPI

MPI signifie **M**essage **P**assing **I**nterface et est la spécification par un ensemble de personnes de l'interface d'une librairie de communication par messages. Nous utiliserons la distribution **LAM** qui implémente **MPI** sur le réseau de stations des salles de TP.

2.1 Historique

Contrairement à **PVM** (**P**arallel **V**irtual **M**achine) qui est issu d'un projet de recherche universitaire, **MPI** est issu d'un consortium de constructeurs, universitaires et utilisateurs. Les participants à ce consortium sont principalement des américains. Quelques européens en font tout de même partie.

Dates clef :

Avril 92	Création d'un groupe de travail
Supercomputing'92 Novembre 92	Formalisation du groupe de travail et décision d'adopter les méthodes du groupe **HPF**
Supercomputing'93 Novembre 93	Brouillon de MPI-1
Mars 94	Finalisation de MPI-1
Novembre 96	Brouillon de MPI-2

Objectifs visés : Portabilité et bonnes performances.
Info : http ://www.mpi-forum.org

2.2 Caractéristiques d'un programme écrit en MPI

C'est un modèle de programmation par échange de messages.
- Le programme est écrit en Fortran, C ou C++.
- Nous nous limiterons au cas où chaque processus exécute le même programme, chacun pouvant exécuter des parties différentes grâce aux conditions de branchement portant sur le rang du processus.
- Toutes les variables du programme sont privées et locales à la mémoire d'un processus.

– Une donnée est échangée entre deux processus grâce à un appel à des sous-programmes de la bibiliothèque **MPI**.

Pour chaque application programmée, nous devrons commencer par définir un *support*, c'est-à-dire l'ensemble des machines exécutant (au moins) un processus de l'application.

2.3 Débuter avec MPI

La forme générale d'un programme **MPI** est la suivante :

```
#include <mpi.h>

int main(int argc, char* argv[]) {
    int rang, p;

    /* Initialisation de MPI.
       Aucune fonction MPI ne peut etre appelee avant l'appel a cette fonction */
    MPI_Init(&argc,&argv);

    /* Affecte a p le nombre de processus qui executent ce programme */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    /* Affecte a rang mon numero de processus */
    MPI_Comm_rank(MPI_COMM_WORLD, &rang);

        /* Instructions du programme */

    /* Apres l'appel a cette fonction, plus aucun appel
       a la bilioteheque MPI n'est possible */
    MPI_Finalize();

    return 0;
}
```

`#include <mpi.h>` est indispensable pour pouvoir utiliser les fonctions MPI.

L'appel à la fonction `MPI_Init` permet d'initialiser l'environnement. Aucune fonction MPI ne peut être appelée avant celle-ci. De la même façon, la fonction `MPI_Finalize` permet de désactiver et de nettoyer l'environnement. Aucun appel aux fonctions MPI n'est possible ensuite.

Une des principales caractéristiques de MPI est que toutes ses fonctions portent sur des communicateurs. Le communicateur par défaut, créé lors de l'appel à la fonction `MPI_Init`, est `MPI_COMM_WORLD`. Il comprend alors tous les processus exécutant l'application.

Les processus sont représentés par un numéro de rang (un entier), unique pour chaque communicateur. Les fonctions `MPI_Comm_size()` et `MPI_Comm_rank()` permettent de connaître respectivement le nombre de processus appartenant à un communicateur et le rang de chaque processus dans ce même communicateur. Ainsi le nombre p de processus dans `MPI_COMM_WORLD` est le nombre de processus de l'application MPI. Les processus se voient alors attribuer un numéro de rang unique entre 0 et $p-1$, correspondant à leur rang dans le communicateur `MPI_COMM_WORLD`.

Remarque spécifique à C : Tous les identificateurs MPI commencent par la chaîne de caractères `MPI_` . Pour les constantes, le reste de la chaîne est en général en majuscules, tandis que seule la lettre suivante le demeure pour les fonctions. Exemple : `MPI_Init()`.

2.4 Communications point à point

Une communication dite point à point a lieu entre deux processus, l'un appelé processus émetteur, l'autre processus récepteur.

MPI propose une grande panoplie de communications point à point. Nous n'en verrons qu'une partie. Pour chacune, il est cependant nécessaire de connaître l'enveloppe du message, qui est constituée des données suivantes :

1. le rang de l'émetteur dans le communicateur utilisé
2. le rang du destinataire dans le communicateur utilisé
3. l'étiquette (*tag*) du message
4. le nom du communicateur

Tout ceci s'apparente à la téléphonie, au courrier postal ou encore à la messagerie électronique.

Outre l'enveloppe, il est nécessaire de connaître le type de la donnée échangée et la taille du message. Les principaux types de données de base en C sont les suivants :

Type MPI	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Il est possible de créer ses propres types (types dérivés).

2.5 Exemple de communications point à point

Voici un petit programme où le processus de rang 1 envoie un entier au processus de rang 0, que ce dernier affiche :

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

int main(int argc, char* argv[]) {
    int rang, p, valeur, tag = 10;
    MPI_Status status;

    /*  Initialisation */

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

    if (rang == 1) {
        valeur = 18;
        MPI_Send(&valeur, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }
    else {
        if (rang == 0) {
            MPI_Recv(&valeur,1,MPI_INT,1,tag, MPI_COMM_WORLD,&status);
            printf("J'ai recu la valeur %d du processus de rang 1",valeur);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Un message est considéré comme un tableau d'éléments dont le type est soit un type de base soit un type dérivé. Pour l'envoi d'un message, la fonction `MPI_Send` a le prototype suivant :

```
MPI_Send(void *buf, int nb, MPI_Datatype dtype, int dest,
                                                int tag, MPI_Comm comm);
```

Le processus appelant cette fonction envoie un message dont l'adresse de début est `buf`, contenant `nb` éléments de type `dtype` au processus de rang `dest` dans le communicateur `comm`. L'étiquette `tag` sert à différencier les messages.

Pour la réception d'un message, la fonction `MPI_Recv` a le prototype suivant :

```
MPI_Recv(void *buf, int nb, MPI_Datatype dtype, int source,
                            int tag, MPI_Comm comm, MPI_Status *status);
```

Le processus appelant cette fonction reçoit un message du processus `source` dans le communicateur `comm`, composé d'au plus `nb` éléments de type `dtype`. Il range le contenu de ce message à l'adresse `buf`. L'étiquette `tag` sert à différencier les messages.

L'argument supplémentaire `status` permet d'obtenir après la réception des informations supplémentaires sur le message : `status.MPI_TAG` permet de connaître l'étiquette du message reçu. De la même façon, `status.MPI_SOURCE` permet de connaître l'expéditeur du message. Cela est utile lorsque `source` est remplacé par le joker `MPI_ANY_SOURCE` et l'étiquette `tag` par `MPI_ANY_TAG`. Nous verrons leur emploi dans nos premiers exemples.

2.6 Communications collectives

Les communications collectives permettent de faire en une seule opération une série d'opérations point à point.

Comme les communications point à point, elles concernent un communicateur donné mais doivent être appelées par l'ensemble des processus appartenant à ce communicateur.

La gestion des étiquettes est transparente et est à la charge du système. De ce fait, les communications collectives n'interfèrent jamais avec les communications point à point.

Nous donnons comme exemple la fonction `MPI_Bcast` qui permet de diffuser une donnée sur l'ensemble des processus.

```
MPI_Bcast(void *buf, int nb, MPI_Datatype dtype, int root, MPI_Comm comm);
```

Le processus `root` envoie les données contenues à l'adresse `buf` à l'ensemble des processus du communicateur `comm`. Chacun des processus range ce message à l'adresse de sa donnée `buf`.

2.7 Tests sur un message

Un processus qui exécute la fonction `MPI_Recv` est bloqué tant qu'aucun message n'est reçu. Il est parfois intéressant de simplement tester si un message est arrivé. Ceci peut être réalisé par un appel à la fonction `MPI_Iprobe` dont le prototype est :

```
MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
```

Si un message de type `tag` est arrivé en provenance de source, `*flag` prend la valeur 1. Sinon il prend la valeur 0. Il existe aussi une version bloquante de cette primitive : `MPI_Probe`. Celle-ci est utilisée principalement lorsque plusieurs types de messages circulent dans l'application. En fonction de `status.MPI_TAG`, on peut alors choisir une zone de stockage adaptée pour l'information reçue. Par exemple, si `status.MPI_TAG = 1`, et si les messages de type 1 transportent des chaînes de caractères, on appellera ensuite la fonction `MPI_Recv` avec une zone de réception de type tableau alors que si le message est de type 2 et contient un entier, on choisira une variable comme zone de réception.

3 openmpi

openmpi est un ensemble de programmes permettant l'utilisation de **MPI** sur un réseau de machines hétérogènes. Pour qu'un utilisateur puisse utiliser **openmpi** sur un support d'application répartie, constitué par exemple d'un PC sous linux (nous lui donnerons le nom pc_linux) et d'une station SUN (que nous appellerons station_sun), il faut que :

- cet utilisateur ait un compte sur les 2 machines
- **openmpi** soit installé sur les 2 machines. Chacun des path de l'utilisateur doit contenir la localisation du répertoire contenant les exécutables de **openmpi** même si ceux-ci ne sont pas localisés au même endroit sur les 2 machines.

Une fois que ces conditions sont remplies, nous pouvons passer à la compilation. Pour compiler le programme C `toto.c`, il suffit de taper la commande suivante :

```
mpicc -Wall -o toto toto.c
```

L'exécutable obtenu s'appelle donc `toto`.

Si la compilation s'est déroulée sans incident, on peut lancer l'exécutable.

3.1 Lancer un exécutable

Celui-ci est lancé à l'aide de la commande `mpirun`. L'option `-np #` précise le nombre de processus à créer dans votre application répartie :

```
mpirun -np 2 toto
```

lance deux processus exécutant le programme `toto`.

Sans autre information, l'ensemble des processus est exécuté sur la machine locale. Si l'on veut une application réellement répartie, il faut en définir le support, c'est-à-dire les machines sur lesquelles les processus vont s'exécuter. Ce support peut être défini dans un fichier du nom de votre choix, par exemple `bhost`.

Ce fichier contient le nom de toutes les machines sur lesquelles s'exécutera votre application répartie, à raison d'une machine par ligne. Soit, dans notre cas :

```
# Tout ce qui suit diese sur une ligne est considere
#  comme un commentaire

# Notre machine est composee de 2 noeuds :

pc_linux      # la machine locale
station_sun
```

Le nom du fichier décrivant le support est fourni au lancement de l'application au moyen de l'option `--hostfile`. Les processus sont ensuite distribués de manière circulaire sur l'ensemble des machines constituant le support. La commande

```
mpirun --hostfile bhost -np 3 toto
```

aurait créé trois processus, le premier sur le PC, le second sur la station SUN et le troisième sur le PC. Si le support comporte plus de machines que de processus, certaines machines ne seront pas utilisées.

4 Configuration à l'ARI

Commencez par vérifier la distribution de MPI qui est installée sur votre machine :

```
bash-3.2$ which mpicc
/usr/local/bin/mpicc
```

La construction de l'exécutable nécessite une édition dynamique de liens avec les bibliothèques de `openmpi`. Il faut donc préciser l'emplacement de ces bibliothèques. Ceci peut être fait en ajoutant dans votre fichier ∼ /`.bashrc` les lignes :

```
LD_LIBRARY_PATH=/usr/local/lib
export LD_LIBRARY_PATH
```

4.1 authentification

`openmpi` utilise `ssh` pour que les différentes machines de l'application puissent converser.

Par défaut, `ssh` demande le mot de passe, mais il est possible de changer ce mode de connexion. Plusieurs modes existent :

1. par reconnaissance des machines et les fichiers `.rhosts` ou `.shosts`,
2. par clés publique et privée,
3. par mot de passe.

Nous allons utiliser la deuxième méthode :

– Vous devez créeer des clés `ssh` grâce à la commande
`ssh-keygen -t dsa`
ces clés doivent avoir un mot de passe.
– Vous devez ensuite permettre l'accès à votre compte grâce à la clé :
`cd ~/.ssh/`
`cat id_dsa.pub >> authorized_keys`

Ensuite il est possible de se loguer sur une machine sans utiliser le mot de passe de votre compte mais celui de la clé. Il y a une différence, c'est qu'il existe un programme qui peut donner ce mot de passe automatiquement : l'agent `ssh`. Lorsque vous autorisez la connexion par clés, l'agent s'occupe de vous authentifier automatiquement. Pour cela, il suffit de le lancer :

```
ssh-agent bash
```

puis de lui donner le mot de passe :

```
ssh-add
```

Cette authentification unique permet de se loguer directement sur toutes les machines de l'ARI. Lorsque vous avez fini, vous détruisez l'agent par

```
ssh-agent -k
```

Il n'est pas nécessaire de regénérer la clé à chaque fois, mémorisez-la pour les prochains TPs.

5 TP

Pour le premier TP, nous allons manipuler les fonctions `Send` et `Recv`. La première étape consiste à faire marcher le programme suivant sur le support de votre choix.

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

int main(int argc, char* argv[]) {
   int         my_rank;       /* rang du processus      */
   int         p;             /* nombre de  processus   */
   int         source;        /* rang de l'emetteur     */
   int         dest;          /* rang du recepteur      */
```

```
   int         tag = 0;       /* etiquette du  message */
   char        message[100];
   MPI_Status  status;

   /* Initialisation  */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   if (my_rank != 0)  {
      /* Creation du message */
      sprintf(message, "Coucou du processus  %d!",  my_rank);
      dest = 0;

      MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   } else {
      for (source = 1; source < p; source++) {
         MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
         printf("%s\n", message);
      }
   }

   /* Desactivation */
   MPI_Finalize();
   return 0;
}
```

Question 1
Faites tourner l'application plusieurs fois avec un nombre égal de processus, puis en faisant varier ce nombre.

Question 2
Mettez des `printf` un peu partout dans le programme. Que se passe-t-il ?

Question 3
Remplacez la variable `source` dans le `MPI_Recv` par l'identificateur `MPI_ANY_SOURCE`. Faites les tests plusieurs fois de suite. Que se passe-t-il ? Expliquez.

Question 4
Ecrivez un programme tel que chaque processus envoie une chaîne de caractères à son successeur (le processus $rang + 1$ si $rang < p - 1$, le processus 0 sinon), puis reçoit un message du processus précédent. Une fois que votre programme fonctionne, remplacez `MPI_Send` par `MPI_Ssend`. Que se passe-t-il ?

Question 5
Tout en gardant `MPI_Ssend`, changez l'algorithme pour que le processus 0 envoie en premier son message au processus 1, qui n'enverra son message au processus 2 qu'après avoir reçu son message de 0. De la même façon, le processus 2 n'enverra son message au processus 3 qu'après avoir reçu de 1, et ainsi de suite...

6 Références

Documentations diverses :

http ://www.open-mpi.org/ : la page de référence pour la distribution Open MPI ;

http ://www-unix.mcs.anl.gov/mpi/ : accès à la distribution `mpich`, tutoriel, etc.

http ://www-unix.mcs.anl.gov/mpi/www : le `man` des commandes et des routines ;

http ://www.lam-mpi.org/ : la page de référence pour la distribution LAM ;