TP1 : introduction à MPI

Version du 27 août 2015

Nous allons utiliser **OpenMPI** qui implémente le standard **MPI** sur le réseau de stations de la salle de TP. Le mode de programmation utilisé sera le mode **SPMD** (le même exécutable pour tous les processus, des branchements en fonction du **numéro du processus** (son **rang**) permettant de leur faire exécuter des tâches différentes)

1 OpenMPI

OpenMPI est un ensemble de programmes permettant l'utilisation de **MPI** sur un réseau hétérogène de machines. Nous nous contenterons ici d'une utilisation sur un réseau homogène de machines. Pour qu'un utilisateur puisse utiliser **OpenMPI** sur une machine parallèle constituée par exemple de deux PC Linux accessibles par `ssh` (`secure sh`), il faut que :

— cet utilisateur ait un compte sur les 2 PC ;
— **OpenMPI** soit installé sur les 2 PC et que la variable d'environnement `PATH` de l'utilisateur contienne sur les 2 PC le répertoire contenant l'application **MPI** à exécuter.

De plus, pour avoir accès aux commandes **OpenMPI** le répertoire contenant ces commandes (communiqué par votre chargé de TP) doit être contenu dans la variable d'environnement `PATH`.

De même, pour que le programme s'exécute correctement sur chaque machine, le répertoire contenant les bibliothèques dynamiques d'**OpenMPI** (communiqué par votre chargé de TP) doit être contenu dans la variable d'environnement `LD_LIBRARY_PATH`.

Vous devez donc modifier votre fichier de configuration (`.bashrc` par exemple) en veillant à ce que les modifications soient bien prises en compte pour les shells non interactifs.

Concrètement, il faut rajouter les lignes suivantes dans votre `~/.bashrc` :

```
export PATH=foo:${PATH}
export LD_LIBRARY_PATH=bar:${LD_LIBRARY_PATH}
```

Attention, il faut les mettre au début du fichier `.bashrc`, avant les éventuelles lignes suivantes :

```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

Par ailleurs, il peut être nécessaire de vérifier que les lignes suivantes sont présentes dans le fichier `.bash_profile` :

```
# include .bashrc if it exists
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

Par défaut `ssh` demande le mot de passe à chaque connexion, mais il est possible d'utiliser un agent `ssh` avec un système de clefs privées et publiques pour éviter cela.

Pour cela :

— vous devez créer des clefs `ssh` grâce à la commande [1] :
```
$ ssh-keygen -t dsa
```
Ces clefs **doivent avoir un mot de passe**. Vous les générez **une fois pour toutes**, vous n'aurez **jamais** besoin de les recréer : vous les conservez pour les prochains TPs.
— Vous devez ensuite permettre l'accès à votre compte grâce à la clef :
```
$ cd ~/.ssh/
$ cat id_dsa.pub >> authorized_keys
```

1. Le signe `$` indique l'invite de l'interpréteur de commande

Ensuite il est possible de se connecter à une machine sans utiliser le mot de passe de votre compte mais celui de la clef. L'agent `ssh` peut alors donner ce mot de passe automatiquement.

Lorsque vous autorisez la connexion par clefs, 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 connecter directement sur toutes les machines du domaine, sans mot de passe. Vérifiez dès maintenant que vous pouvez effectivement vous connecter sur n'importe quelle machine distante de la salle sans mot de passe (et sans message d'erreur).

A garder pour Polytech Lille ? Enfin pour qu'OpenMPI puisse se déployer plus efficacement sur plusieurs noeuds, votre agent `ssh` doit pouvoir être transmis de noeud en noeud. Pour cela, ajouter les lignes suivantes au fichier `~/.ssh/config`

```
Host *
   ForwardAgent yes
```

Lorsque vous aurez fini à la fin du TP, vous détruirez l'agent `ssh` par

```
$ ssh-agent -k
```

1.1 Le schéma de boot

Le schéma de boot est défini dans un fichier du nom de votre choix (par exemple **hostfile**) que vous placez dans votre répertoire de travail. Celui-ci contient le nom de toutes les machines qui vont former votre machine parallèle, avec comme syntaxe un nom de machine par ligne. Vous devez inclure le nom de la machine sur laquelle vous êtes actuellement connectés. Soit, par exemple :

```
# Toute ligne commencant pas un dièse
# est considérée comme un commentaire
#
# Notre machine est composée de 2 noeuds
pc01
pc02
```

`pc01` est alors considéré comme le premier nœud de la machine, et `pc02` comme le second. Nous aurions pu ajouter toutes les machines disponibles sur notre réseau vérifiant les conditions ci-dessus. Cela ne veut pas dire qu'à chaque fois que nous exécuterons un programme, tous les nœuds seront utilisés.

1.2 Compilation et exécution de programme MPI

Pour compiler le programme C `toto.c`, il suffit de taper la commande suivante :

```
$ mpicc -o toto toto.c
```

L'exécutable obtenu s'appelle donc `toto`. Celui-ci est lancé à l'aide de la commande `mpirun`. L'option `-n #` stipule le nombre de processus à créer sur les nœuds de votre machine parallèle.

```
$ mpirun -n 2 -hostfile hostfile ./toto
```

Dans ce cas deux processus exécutant le programme `toto` sont lancés, le premier sur `pc01`, le second sur `pc02`. La commande

```
$ mpirun -n 3 -hostfile hostfile ./toto
```

aurait créé trois processus, le premier sur `pc01`, le second sur `pc02` et le troisième sur `pc01`. L'attribution des processus est cyclique par rapport au numéro des nœuds. Dans le cas de mesures de performance, vous ne devrez bien sûr pas lancer plusieurs processus sur un même nœud (sauf éventuellement sur un nœud multicœur).

1.3 Déboguer votre programme

Nous déboguerons nos programmes MPI avec des `printf` uniquement (attention à l'ordre des affichages entre les différents processus, voir section 2).

Il est aussi possible d'utiliser `gdb` : le plus simple est alors de lancer tous les processus MPI en local sur votre machine, mais ceci change l'exécution parallèle et ne permet donc pas de détecter tous les bugs qui apparaissent en parallèle.

Pour plus d'informations sur le déboguage avec **OpenMPI**, voir :
`http://www.open-mpi.org/faq/?category=debugging#serial-debuggers`

2 TP

Pour commencer, vous pouvez récupérer la documentation sur MPI (version 2.2) au format `pdf` à cette adresse : `http://www.mpi-forum.org/`.

Pour ce premier TP, nous allons manipuler les fonctions `Send` et `Recv`.

Exercice 1

Que doit afficher le programme MPI suivant ?

Vérifiez-le en compilant puis en exécutant ce programme avec **OpenMPI**.

```
#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.\n",valeur);
    }

    MPI_Finalize();
}
```

Exercice 2

Voici un programme un peu plus compliqué :

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

#define SIZE_H_N 50

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;
    char hostname[SIZE_H_N] ;

    gethostname(hostname,SIZE_H_N);

    /* 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 depuis %s!",
                my_rank, hostname);
        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("Sur %s, le processus #%d a recu le message : %s\n",
                   hostname, my_rank, message);
        }
    }

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

1. Faîtes tourner cette application plusieurs fois avec un nombre égal de processus, puis en les faisant varier.
2. Mettez des `printf` un peu partout dans le programme. Que se passe-t-il ?
3. Remplacez la variable `source` dans le `MPI_Recv` par l'identificateur `MPI_ANY_SOURCE`. Faîtes les tests plusieurs fois de suite. Que se passe-t-il ? Expliquez.

4. Écrivez 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), et qu'il reçoit un message du processus précédent. Une fois que votre programme fonctionne, remplacez `MPI_Send` par `MPI_Ssend`. Que se passe-t-il ? On appelera ce programme `ex_ssend.c`
5. Recopiez `ex_ssend.c` dans `ex_ssend_correcte.c` 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. Recopiez `ex_ssend.c` dans `ex_send.c` Remplacez le `MPI_Ssend` par `MPI_Send`. Faites varier la taille des données envoyées par le `MPI_Send` jusqu'à 100 ko. Que se passe-t-il ?

Exercice 3 – (facultatif)
On dispose de P processus et on souhaite implémenter un algorithme de réduction effectuant la sommation des valeurs entières possédées par chaque processus. La racine de l'algorithme de réduction sera le processus de numéro 0 (et c'est donc lui qui affichera le résultat).

1. Définissez un algorithme efficace approprié *sans routine de communication collective*.
2. Réalisez un programme parallèle MPI implémentant cet algorithme. Le programme calculera la somme globale des valeurs aléatoires entières générées par chaque processus, de façon à ce que le processus 0 reçoive et affiche la valeur somme.
3. Réécrivez votre programme à l'aide d'une routine de communication collective, puis modifiez le de façon à ce que la somme globale soit disponible simultanément sur tous les processus.

Exercice 4 – Nettoyage
A la fin du TP, ne pas oublier de tuer son agent `ssh` : `$ ssh-agent -k`
Vérifier ses processus sur la machine locale : `ps uxww`

3 Références

Quelques implémentations du domaine public :
— MPICH (Argonne NL Missisipi State U.)
`http://www.mcs.anl.gov/mpi/mpich`
— Open MPI :
`http://www.open-mpi.org/`
Livres :
— `http://www-unix.mcs.anl.gov/mpi/usingmpi/`
— `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`
— `http://fawlty.cs.usfca.edu/mpi/`
— *Parallel Programming in C with MPI and OpenMP*, M.J. Quinn, McGraw-Hill
Doccumentations diverses :
— documents officiels : `http://www.mpi-forum.org/`
— `http://www.idris.fr/data/cours/parallel/mpi/mpi\_cours.html`
— `http://www-unix.mcs.anl.gov/mpi/`
— Newsgroup : `comp.parallel.mpi`
— moteurs de recherche : mots clefs : mpi, openmpi, mpich, lam, message passing, ...