TP1 : introduction à MPI

Version du 8 septembre 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)

Enseignants :
Pour le SMT (hyperthreading chez Intel) : n'utiliser qu'un seul processus ou thread par coeur physique (par ex. : si CPU quad-core Intel core i7 avec 2-way SMT (soit 8 coeurs logiques), n'utiliser que 4 processus MPI (ou 4 threads)) par défaut pour les mesures de performance (accélération, efficacité parallèle). Ils peuvent ensuite mettre autant de processus MPI ou de threads que de coeurs logiques pour voir le gain en performance apporté par le SMT.

Enseignants :
Exercices supplémentaires possibles : - implémenter réduction du TD1 dans les trois versions (anneau, tous vers O, arbre binaire) et étude des perfs qd nb procs augmente

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`.

Enseignants :
A Polytech Lille, c'est le répertoire :

```
/usr/bin/
```

donc rien à faire a priori ...

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`.

Enseignants :
A Polytech Lille, c'est le répertoire

```
/usr/lib/openmpi
```

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
```

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).

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
```

Enseignants :
Nécessaire au moins depuis OpenMPI-1.8.2 : il semble qu'OpenMPI se déploie en arbre depuis le noeud local sur tous les noeuds du fichier `hostfile`.

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

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

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

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.

Enseignants :
Pour leur tests, les étudiants ne doivent utiliser que les machines de leur salle, et non des machines situées dans d'autres salles (temps de communication MPI potentiellement beaucoup plus important).

Quelques noms de machines à Polytech Lille :

— Machines en C101 : gedeon01, ..., gedeon14
— Machines en C102 : clodion01, ..., clodion14
— Machines en C103 : phinaert01, ..., phinaert14

Autres infos salles Polytech Lille :

"Toutes les salles de TP sont sous linux debian (les linux sont natif sur les machines. Les Windows sont dans les machines virtuelles). Une partie est en wheezy et le passage en jessie est en court. Les salles B109, B104, C102 sont des machines recentes (core I5, 8Go de ram) Les autres salles sont des machines assez ancienne (AMD64, 4Go de ram) et sont en cours de renouvellement.

les machines sont dans un reseau prive. Elles accedent uniquement a l'exterieur via le proxy de polytech. Certain acces sur l'universite son autorises.

La version installee est : openmpi 1.4.5 ; la mise a jour n'est pas possible car c'est une vielle distribution. Sur les nouvelles salle on aura openmpi 1.6.5."

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.

Enseignants :
Connaissent-ils gdb en GIS5 ? A priori, non : à conseiller au plus motivés.

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.

> **Solution :**
> Juste pour montrer que l'ordre de réception des messages ne varie pas (car on précise l'émetteur (`source`) dans

l'appel à `MPI_Recv`.

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

Solution :
Même conclusion qu'à la question précédente : le fait de rajouter du "travail" à différents endroits dans le code ne modifie pas l'ordre de réception des messages. Attention : il n'y a aucune garantie sur l'ordre d'affichage entre les printf des processus de rang > 0 et les printf du processus de rang $0 \ldots$

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.

Solution :
Les messages sont reçus dans l'ordre de leur réception par le processus de rang 0. Le premier message reçu est affiché en premier, puis le second... Ainsi l'ordre peut varier d'une exécution à l'autre.

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`

Solution :

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

#define MSG_SIZE 100

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[MSG_SIZE];
    MPI_Status   status;

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


    /* Creation du message */
    sprintf(message, "Coucou du processus %d!", my_rank);
    dest = (my_rank == p-1 ? 0 : my_rank + 1);
```

```
    MPI_Send(message, strlen(message)+1, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);

    /* Reception du message */
    source = (my_rank == 0 ? p-1 : my_rank - 1);
    MPI_Recv(message, MSG_SIZE, MPI_CHAR, source, tag,
             MPI_COMM_WORLD, &status);
    printf("Je suis le processus %d et j'ai recu ce message du processus %d : %s\n",
           my_rank, source, message);

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

Si on remplace `MPI_Send` par `MPI_Ssend`, on obtient un interblocage : chaque processus est bloqué dans son appel à `MPI_Ssend`.

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...

Solution :

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

#define MSG_SIZE 100

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[MSG_SIZE];
    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){
      /* Processus 0
       */

      /* Creation du message */
      sprintf(message, "Coucou du processus %d!", my_rank);
```

```
        dest = (my_rank == p-1 ? 0 : my_rank + 1);

        MPI_Ssend(message, strlen(message)+1, MPI_CHAR,
                  dest, tag, MPI_COMM_WORLD);

        /* Reception du message */
        source = (my_rank == 0 ? p-1 : my_rank - 1);
        MPI_Recv(message, MSG_SIZE, MPI_CHAR, source, tag,
                 MPI_COMM_WORLD, &status);
        printf("Je suis le processus %d et j'ai recu ce message du processus %d : %s\n",
               my_rank, source, message);

    }
    else {
      /* Processus > 0
       */

      /* Reception du message */
      source = (my_rank == 0 ? p-1 : my_rank - 1);
      MPI_Recv(message, MSG_SIZE, MPI_CHAR, source, tag,
               MPI_COMM_WORLD, &status);
      printf("Je suis le processus %d et j'ai recu ce message du processus %d : %s\n",
             my_rank, source, message);

      /* Creation du message */
      sprintf(message, "Coucou du processus %d!", my_rank);
      dest = (my_rank == p-1 ? 0 : my_rank + 1);

      MPI_Ssend(message, strlen(message)+1, MPI_CHAR,
                dest, tag, MPI_COMM_WORLD);

    }

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

Il n'y a ainsi plus d'interblocage.

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 ?

Solution :

L'envoi standard (bloquant) `MPI_Send` offre en fait deux modes :

- un mode "envoi immédiat" (*eager mode*) pour les petits messages, dans lequel les messages sont envoyés immédiatement, même si le destinataire n'a pas commencé la réception correspondante (stockage dans un *buffer*) ;
- un mode "rendez-vous" pour les plus gros messages, dans lequel le corps du message n'est envoyé que lorque le destinataire est prêt à le recevoir (la réception correspondante a été lancée). On retrouve alors le comportement de l'envoi en mode synchrone (`MPI_Ssend`).

Il existe une limite de taille de messages entre ces deux modes, généralement appelée *MPI eager limit*, et souvent fixée par défaut à 64 ko (65536). Elle peut néansmoins varier suivant l'implémentation MPI et le système.

Lorsqu'on augmente la taille des messages jusqu'à 100 ko, on dépasse cette limite et on retrouve l'interblocage dû au mode "rendez-vous".

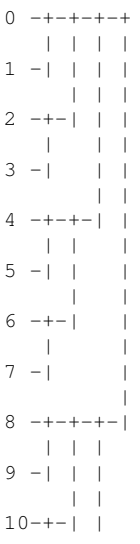
Exercice 3 – (facultatif)

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*.

Solution :

Schématiquement, la solution optimale, basée sur un arbre binaire, est la suivante (remarque : possible car l'addition est associative) :



L'algorithme correspondant est :

Pré-condition : r : numéro (rang) du processus ($0 \leq r \leq P-1$)

```
1:    /* Il y a N etapes, avec : 2^{N-1} < P <= 2^N */
2:  Pour i = 0 à ⌈log_2(P)⌉ − 1 faire
3:      /* Etape 'i' : difference entre source et destination 1 << i */
4:      /* (c'est-à-dire "1 decale de i", qui vaut 2^i ) */
5:    Si ((r >> i)&1) Alors     /* syntaxe C */
6:        /* Je dois emettre : */
7:      dest = r − (1 << i)     /* syntaxe C */
8:      Send(valeur, dest)
9:      break     /* Je quitte la boucle 'for' apres mon premier envoi. */
10:   Sinon
11:       /* Je dois recevoir, si mon emetteur à l'etape 'i' existe : */
12:     Si (r + (1 << i) < P) Alors     /* syntaxe C */
13:       orig = r + (1 << i)     /* syntaxe C */
14:       Recv(receive, orig)
15:       valeur+ = receive
16:     Fin si
17:   Fin si
18: Fin pour
19: Si r == 0 Alors
20:   afficher valeur
21: Fin si
```

Autres solutions possibles :

- tout le monde envoie en même temps au processus 0 : fort risque de contention réseau au niveau du processus 0 qui ne pourra de toute façon traiter qu'un seul message à la fois. Si de plus l'opération de réduction est plus coûteuse qu'une simple addition, tout le calcul sera fait par le processus 0.
- circulation d'un jeton sur l'anneau formé par les processus : P étapes alors que la solution avec l'arbre binaire nécessite $\lceil \log_2(P) \rceil$ étapes

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.

Solution :

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h> // ne pas oublier '-lm' a la compilation
#include <time.h>
#include "mpi.h"



int main (int argc, char * argv[]) {
  int my_rank; /* mon rang dans MPI_COMM_WORLD */
  int nproc, somme, orig, dest, i, receive;
  MPI_Status status;
  int nb_etapes = 0;

  /* Initialisation
   */
  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);


  if (my_rank == 0){
    printf("Nb_procs = %i\n", nproc);
  }

  /* Generation du nombre aleatoire :
   */
  srand48(clock() + my_rank);
  somme = ((double) 111) * drand48();
  /*** Ou avec rand()/srand():
   * srand() ; // ou : srand(time(NULL));
   * somme = rand()%100;
   */

  printf("my_rank=%d, initial value is %d\n", my_rank, somme);

  /* Calcul de la somme :
   */
  /* Il y a N etapes, avec : $ 2^{N-1} < nprocs <= 2^N $  */
  nb_etapes = (int) ceil(log(nproc)/log(2));
  if (my_rank == 0) {printf("Nb etapes : %i\n", nb_etapes);}
  for (i=0; i < nb_etapes; ++i) {
    /* Etape 'i' : difference entre source et destination 1<<i
       (i.e. "1 decale de i", qui vaut $2^i$ ) */
    /* Autre facon de faire : mask=1; puis mask<<=1; */

    //     if (my_rank == 0) {printf("i = %i\n", i);}
```

```
    if ((my_rank >> i) & 1) {
      /* Je dois emettre : */
      dest = my_rank - (1<<i);
      MPI_Send(&somme, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      break; /* Je quitte la boucle 'for' apres mon premier envoi. */
    }
    else {
      /* Je dois recevoir, si mon emetteur a l'etape 'i' existe : */
      if (my_rank + (1<<i) < nproc ) {
        orig = my_rank + (1<<i);
        MPI_Recv(&receive, 1, MPI_INT, orig, 0, MPI_COMM_WORLD,&status);
        somme += receive;
      }
    }

  } /* for i */

  if (my_rank==0)
    printf("somme = %d\n", somme);

  /* Terminaison :
   */
  MPI_Finalize();
}
```

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.

Solution :
Résultat dans le processus 0 :

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h> // ne pas oublier '-lm' a la compilation
#include <time.h>
#include "mpi.h"



int main (int argc, char * argv[]) {
  int my_rank; /* mon rang dans MPI_COMM_WORLD */
  int nproc, somme, orig, dest, i, receive;
  MPI_Status status;


  /* Initialisation
   */
  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);


  if (my_rank == 0){
```

```
    printf("Nb_procs = %i\n", nproc);
  }

  /* Generation du nombre aleatoire :
   */
  srand48(clock() + my_rank);
  somme = ((double) 111) * drand48();
  printf("my_rank=%d, initial value is %d\n", my_rank, somme);

  /* Calcul de la somme :
   */
  MPI_Reduce(&somme, &receive, 1, MPI_INT, MPI_SUM, 0 /* root */, MPI_COMM_WORLD);


  if (my_rank==0)
    printf("somme = %d\n", receive);

  /* Terminaison :
   */
  MPI_Finalize();
}
```

Résultat dans tous les processus :

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h> // ne pas oublier '-lm' a la compilation
#include <time.h>
#include "mpi.h"



int main (int argc, char * argv[]) {
  int my_rank; /* mon rang dans MPI_COMM_WORLD */
  int nproc, somme, orig, dest, i, receive;
  MPI_Status status;


  /* Initialisation
   */
  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);


  if (my_rank == 0){
    printf("Nb_procs = %i\n", nproc);
  }

  /* Generation du nombre aleatoire :
   */
  srand48(clock() + my_rank);
  somme = ((double) 111) * drand48();
  printf("my_rank=%d, initial value is %d\n", my_rank, somme);

  /* Calcul de la somme :
```

```
    */
   MPI_Allreduce(&somme, &receive, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

   printf("Processus %d : somme = %d\n", my_rank, receive);

   /* Terminaison :
    */
   MPI_Finalize();
}
```

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, ...