Communications collectives en MPI

1 Un mot d'histoire

Tout d'abord, il convient de rappeler quelques généralités sur ce qu'est MPI et sur ce que ce n'est pas. MPI n'est pas une bibliothèque de communication développée par une université ou une entreprise : c'est un standard qui est né au début des années 90 pour répondre à un besoin de clarification. En effet, à cette époque, la seule façon de faire du calcul parallèle efficace consistait à acheter une grosse machine parallèle propriétaire. Ces dernières étaient généralement livrées avec leur propre bibliothèque de communication qui n'était que rarement compatible avec celle de la machine précédente et jamais avec celle des concurrents. Il était donc très difficile de maintenir un programme à jour et un gros travail était nécessaire à chaque fois que l'on souhaitait changer de machine. Le standard MPI est donc né de la collaboration entre des universitaires et des industriels de tous domaines scientifiques. Cependant, si ce standard a permis de résoudre la majorité des problèmes que l'on pouvait avoir au moment de sa création, il souffre désormais d'un bon nombre de limitations et d'une inadéquation aux plates-formes de calcul actuelles. Par exemple, un des principaux inconvénients apparaît lorsque l'on souhaite utiliser un programme MPI sur une plate-forme de calcul distribuée à grande échelle : les bibliothèques MPI d'un centre de calcul à l'autre ne sont pas forcément identiques et rien dans le standard n'oblige une bibliothèque à perdre en performance pour être compatible avec une bibliothèque concurrente. Cependant, les implémentations de MPI sont encore largement utilisées sur des machines parallèles homogènes telles que les grappes de PCs.

Un nouveau standard MPI-2 a été mis en place il y a quelques années et résout certains des problèmes précédemment évoqués. Il existe cependant encore très peu de bibliothèques mettant en œuvre l'intégralité du standard MPI-2.

L'implémentation de MPI que nous utiliserons aujourd'hui est MPICH, une implémentation libre très répandue de MPI.

2 Introduction à l'utilisation de MPI

2.1 Initialisation et terminaison du programme

Un programme MPI commence en général par un appel de la fonction `MPI_Init` dont le prototypage est le suivant :

```
int MPI_Init(int *argc, char ***argv)
```

Cette fonction initialise les connections MPI en fonction des arguments passés à votre programme. C'est pourquoi avant de lire les arguments de votre programme, il convient de faire un appel à cette fonction.

Un programme MPI se termine généralement par l'appel de la fonction `MPI_Finalize`. Tous les processus doivent appeler ce programme avant leur terminaison. Cette opération est bloquante et ne termine que lorsque toutes les opérations de communications en cours ou en attente sont terminées.

Il peut être utile de savoir combien de processus participent au calcul et quel numéro on a. C'est ce que font les fonctions `MPI_Comm_size` et `MPI_Comm_rank` dont le prototype est le suivant. Un `MPI_Comm` est un groupe de processus MPI, ou communicateur. Dans un premier temps, on peut utiliser le groupe prédéfini `MPI_COMM_WORLD` qui regroupe l'intégralité des processus MPI participant au calcul.

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

Si ces explications ne vous suffisent pas, vous pouvez évidemment consulter la documentation en ligne :
- http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html pour le standard MPI 1.1
- http://www-unix.mcs.anl.gov/mpi/mpich1/docs.html pour l'implémentation MPICH
- http://www-unix.mcs.anl.gov/mpi/www pour l'API

2.2 Communications bloquantes

Les envois et les réceptions bloquantes se font grâce aux fonctions `MPI_Send` et `MPI_Recv` dont le prototype est le suivant :

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm )
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_tatus *status )
```

Quelques explications :
- `buf` représente l'adresse du buffer d'émission (pour `MPI_Send`) ou du buffer de réception (pour `MPI_Recv`).
- `count` est le nombre d'éléments à envoyer ou à recevoir.
- `datatype` est le type des éléments que l'on va envoyer (ou recevoir). On doit préciser le type des éléments car MPI peut effectuer des conversions de type si les architectures cibles ne représentent pas les objets de la même manière (*big endian/little endian*...). Il est possible de créer des types MPI de façon à faciliter la programmation mais nous n'utiliserons pour l'instant que le type `MPI_INT`.
- `dest` et `source` sont respectivement les indices des processeurs destinataires et sources
- `tag` est un nombre qui sert à identifier un transfert : la réception doit utiliser le même `tag`, mais un même `tag` peut être utilisé pour plusieur transferts.
- `comm` est le communicateur (groupe de processus) impliqué dans la communication ;
- `status` est un objet permettant de récupérer des informations sur la communication et ne nous servira pas pour l'instant.

Enfin une autre fonction qui s'avère souvent utile : la barrière de synchronisation.

```
int MPI_Barrier ( MPI_Comm comm )
```

Nous allons essayer de simuler la fonction `MPI_Bcast` dont le prototype est le suivant.

```
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root,
                MPI_Comm comm )
```

L'appel de cette fonction effectue une diffusion de données depuis la machine possédant le rang `root` vers toutes les machines du communicateur `comm`. Tous les processus du groupe effectuant la diffusion doivent appeler la fonction `MPI_Bcast` même si, selon leur rang dans le communicateur, cet appel n'a pas le même effet. Pour les processus qui ne sont pas l'émetteur, `buffer` sert à recevoir les données alors que pour l'émetteur, il contient les données à diffuser. Enfin, cette fonction est bloquante pour tous les processus impliqués. Lorsqu'un programme alterne des phases de calcul et des phases de communications, il y a donc intérêt à ce que l'équilibrage de charge soit parfait...

2.3 Communications non bloquantes

Les fonctions d'envoi et de réception non bloquantes `MPI_Isend` et `MPI_Irecv` ont le prototype suivant :

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request )
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request *request )
```

`request` est un *"numéro"* de communication. Cela permet de savoir si une communication est terminée ou pas, notamment grâce à la fonction `MPI_Test` ou à la fonction `MPI_Wait`.

```
int MPI_Test ( MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait ( MPI_Request *request, MPI_Status *status)
```

`flag` reçoit la valeur "vrai" si la communication est effectivement terminée. Il existe des variantes de ces fonctions, comme `MPI_Waitany`, `MPI_Waitsome` ou `MPI_Waitall`. N'hésitez pas à consulter la documentation en ligne pour en savoir plus.

2.4 Lancement du programme

Commencez par copier l'intégralité du répertoire `/home/vrehn/teaching/AlgoPar/tp1-src` dans votre répertoire de travail :

```
$ cp -r /home/vrehn/teaching/AlgoPar/tp1-src .
```

Vous y trouverez un programme MPI minimal (`simple.c`) ainsi qu'un makefile permettant de compiler (en tapant `make`) et de lancer des programmes MPI (en tapant `make run`). La bibliothèque MPI que nous allons utiliser s'appelle MPICH et est disponible sur une grande variété de plates-formes.

Dans le même répertoire, un script `generate-key.sh` est disponible qui vous permet de générer une clé et d'initialiser ssh pour que MPICH puisse se connecter via ssh à toutes les machines de la salle. Notez qu'à la première connection sur une machine, vous recevrez un message déclarant que l'authenticité de la machine n'est pas établie. Répondez simplement "yes." Les machines auxquelles vous avez accès sont par exemple `slsu0-xx` avec `xx` variant de 00 à 31.

3 Diffusion en MPI

▷ **Question 1** *Écrivez un programme où le processeur* 0 *diffuse aux autres processeurs un tableau d'entiers à l'aide de la fonction `MPI_Bcast`. Rajoutez un appel à la fonction `long_computation` juste après la diffusion et mesurez le temps nécessaire à ces opérations grâce à la fonction `ms_time` (ces deux fonctions sont disponibles dans le programme `simple.c` évoqué précédemment).*

▷ **Question 2** *Recommencez en écrivant vous même la diffusion à l'aide d'opérations bloquantes. Que pensez-vous des performances, notamment quand vous faites varier le nombre de processus ?*

4 Échange total de données

On suppose maintenant que les processeurs possèdent chacun un tableau d'entiers, et qu'il est nécessaire d'échanger ces données avant de pouvoir calculer.

▷ **Question 3** *Toujours en utilisant les primitives de communication bloquantes `MPI_Send` et `MPI_Recv`, écrivez un algorithme pour l'échange total de données (ou* All-Gather*). Comme précédemment, rajoutez un appel à `long_computation` après l'échange de données et mesurez le temps nécessaires à ces opérations avec `ms_time`.*

▷ **Question 4** *Pour éviter d'avoir à planifier précisément l'ordre des transferts, réécrivez un algorithme pour le* All-Gather *en utilisant des communications non bloquantes, mais en s'assurant que tous les transferts sont bien terminés avant le début du calcul. Comparez les performances obtenues avec l'algorithme précédent.*

▷ **Question 5** *Même question en utilisant `MPI_Allgather`.*

▷ **Question 6** *Même question avec un échange total de données personnalisé (ou* All-to-all*) en utilisant `MPI_Alltoall`.*

L'objectif de ce TD était de mettre en valeur un atout majeur de MPI, qui est l'existence de nombreuses primitives de communications collectives. Nous avons vu ici `MPI_Bcast`, `MPI_Allgather` et `MPI_Alltoall`, mais il en existe un grand nombre, permettant de manipuler aisément des données distribuées avec un grand niveau d'abstraction. L'avantage est également que ces primitives sont implémentées différemment selon le réseau de connexion sous-jacent, pour profiter au mieux des ressources de communication disponibles.