Table des matières

Chapitre 17

Exécution et débogage de programmes parallèles MPI avec OpenMPI

17.1 Introduction : Caractéristiques d'un *cluster* et du *cluster* utilisé dans le cours

Voici deux définitions de ce qu'est un *cluster* :

> *Co-located collection of mass-produced computers and switches dedicated to running parallel jobs. The computers typically do not have displays or keyboards [and] some of the computers may not allow users to login [except head node]. All computers run the same OS and have identical disk images. [They also] use a faster, [generally] switched networked, e.g., gigabit Ethernet. [Qui03]*
>
> *Any collection of distinct computers that are connected and used as a parallel computer, or to form a redundant system for higher availability. The computers in a cluster are not specialized to cluster computing. In other words, the computer making up the cluster [...] are not custom-built for use in the cluster. [MSM05].*

Un *cluster* — une grappe de processeurs, une grappe de calcul — est donc une machine parallèle MIMD de type *multi-ordinateurs* ayant les caractéristiques suivantes :

- Les diverses machines — les divers «noeuds» — sont intégrées en un système unique, (généralement) *avec un réseau dédié.*
- Les noeuds travaillent sur un seul et même problème.
- Les divers noeuds sont des machines de «faible coût», où chaque noeud «pourrait» en théorie être utilisé comme une machine ou station de travail normale. Toutefois, en pratique, généralement un seul des noeuds — le «head node» — est accessible directement aux usagers.

La machine utilisée dans le cours INF7235

Le *cluster* que nous allons utiliser pour la partie du cours sur MPI est la machine `turing.hpc.uqam.ca` du LAMISS.

Cette machine, acquise en 2014, possède les caractéristiques suivantes :

- 1 *head node* — `turing.hpc.uqam.ca`
- 30 noeuds — `{quark20,...,quark49}.hpc.uqam.ca`
  - Chaque noeud compte 4 processeurs Intel Xeon `x86_64`
- Système d'exploitation : *Scientific Linux release 6.8 (Carbon)*.

17.2 Étapes pour l'exécution d'un programme MPI

Pour compiler et exécuter un programme MPI/C avec OpenMPI (version de MPI installée sur le *cluster* `turing`), deux étapes sont nécessaires, décrites plus bas. Notez toutefois que des cibles `make` appropriées ont été définies pour les laboratoires, donc il suffira généralement d'exécuter «`make compile`» puis «`make run`».

1. On compile le programme sur le noeud maître (`turing`). Par exemple :

   ```
   $ mpicc -o hello -std=c99 hello.c
   ```

2. On lance l'exécution du programme en spécifiant le nombre de «processeurs» qu'on désire utiliser, et ce avec la commande `mpirun`. Par exemple :

   ```
   $ mpirun -np 6 hello
   quark20.hpc.uqam.ca
       numProc = 0, nbProcs = 6

   quark20.hpc.uqam.ca
       numProc = 1, nbProcs = 6

   quark20.hpc.uqam.ca
       numProc = 2, nbProcs = 6

   quark20.hpc.uqam.ca
       numProc = 3, nbProcs = 6

   quark21.hpc.uqam.ca
       numProc = 4, nbProcs = 6

   quark21.hpc.uqam.ca
       numProc = 5, nbProcs = 6

   ----------------------------------------
   ```

   Note : Il peut s'agir de «processeurs virtuels» si la valeur indiquée est supérieure au nombre de noeuds **et** que les noeuds à utiliser ont été spécifiés avec l'option `--hostfile` — voir plus loin.

Soulignons que n'importe quel commande ou programme (exécutable) peut être exécuté avec `mpirun` :

```
$ mpirun -np 3 date
ven dec 18 14:35:02 EST 2015
ven dec 18 14:35:02 EST 2015
ven dec 18 14:35:02 EST 2015


$ mpirun -np 2 ls -l hello
-rwxr-xr-x 1 tremblay users 9542 18 dec 14:33 hello
-rwxr-xr-x 1 tremblay users 9542 18 dec 14:33 hello


$ mpirun -np 6 uname -n
quark21.hpc.uqam.ca
quark21.hpc.uqam.ca
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca


$ mpirun -np 6 --map-by node uname -n
quark20.hpc.uqam.ca
quark21.hpc.uqam.ca
quark23.hpc.uqam.ca
quark25.hpc.uqam.ca
quark22.hpc.uqam.ca
quark24.hpc.uqam.ca
```

17.3 Spécification des processeurs

Plusieurs processeurs par noeud vs. un seul par noeud

Par défaut, **tous les processeurs** de chaque noeud[1] sont utilisés avant de passer aux processeurs du noeud suivant pour activer de nouveaux processus :

```
$ mpirun -np 10 uname -n
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca
quark22.hpc.uqam.ca
quark22.hpc.uqam.ca
quark21.hpc.uqam.ca
quark21.hpc.uqam.ca
quark21.hpc.uqam.ca
quark21.hpc.uqam.ca
```

Par contre, on peut n'utiliser qu'un seul processeur par noeud en spécifiant l'option «`--map-by node`» :

```
$ mpirun -np 6 --map-by node uname -n
quark20.hpc.uqam.ca
quark23.hpc.uqam.ca
quark21.hpc.uqam.ca
quark25.hpc.uqam.ca
quark24.hpc.uqam.ca
quark22.hpc.uqam.ca
```

[1] Quatre (4) processeurs pour les noeuds du *cluster* `turing`.

Processeurs virtuels = plusieurs processus par processeur

Si on spécifie l'option `--hostfile`, on peut aussi utiliser des «**processeurs virtuels**», i.e., on peut exécuter **plusieurs instances** du programme sur un même processeur physique — le terme technique est «*oversubscription*» = *run more processes than processors*:

```
$ cat un-host.txt
quark28.hpc.uqam.ca

$ mpirun -np 6 --hostfile un-host.txt uname -n
The authenticity of host 'quark28.hpc.uqam.ca (192.168.20.28)' can't be estab
RSA key fingerprint is
90:b9:58:0c:7c:b4:78:9f:10:e8:55:ac:a1:ca:c1:3a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'quark28.hpc.uqam.ca' (RSA) to the list of known h
quark28.hpc.uqam.ca
quark28.hpc.uqam.ca
quark28.hpc.uqam.ca
quark28.hpc.uqam.ca
quark28.hpc.uqam.ca
quark28.hpc.uqam.ca


$ cat deux-hosts.txt
quark20.hpc.uqam.ca
quark21.hpc.uqam.ca

$ mpirun -np 10 --hostfile deux-host.txt uname -n
quark20.hpc.uqam.ca
quark21.hpc.uqam.ca
quark21.hpc.uqam.ca
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca
quark21.hpc.uqam.ca
quark20.hpc.uqam.ca
quark21.hpc.uqam.ca
quark20.hpc.uqam.ca
quark21.hpc.uqam.ca
```

Note : Remarquez le message dans le premier appel à `mpirun` : on a un tel message lorsqu'on utilise (on se connecte à) un noeud *pour la première fois* — et seulement cette première fois.

Application MPMD

Finalement, il est aussi possible d'exécuter une application de style **MPMD** (*Multiple Program, Multiple Data*) plutôt que SPMD (*Single Program, Multiple Data*). Dans ce cas, l'application est alors composée de plusieurs programmes distincts :

```
$ mpirun -np 2 date : -np 3 hostname : -np 1 hello
quark21.hpc.uqam.ca
-rw-r--r-- 1 tremblay users 1477 14 mar 18:56 hello.c
-rw-r--r-- 1 tremblay users 6166 10 mar  2016 min.c
-rw-r--r-- 1 tremblay users 5429  9 mar 11:19 min-max.c
-rw-r--r-- 1 tremblay users 4248  9 mar 11:19 somme-vecteurs.c
mar mar 14 18:57:07 EDT 2017
mar mar 14 18:57:07 EDT 2017
quark20.hpc.uqam.ca
quark20.hpc.uqam.ca
```

Dans un tel cas, les communications doivent se faire en tenant compte que **tous les programmes indiqués font partie du groupe** (`MPI_COMM_WORLD`).C'est ce qui explique que l'exécution suivante ne fonctionne pas correctement, i.e., le programme bloque après avoir émis les dates et on doit le terminer avec `^C` :

```
$ mpirun -np 3 date : -np 2 hello
mar mar 14 19:04:09 EDT 2017
mar mar 14 19:04:09 EDT 2017
mar mar 14 19:04:09 EDT 2017
^CKilled by signal 2.
Killed by signal 2.
Killed by signal 2.
Killed by signal 2.
Killed by signal 2.
```

17.4 Stratégies pour *déboguer* un programme parallèle

> *This brings up one of the most important points to keep up in mind when you're debugging a parallel program: Many (if not most) parallel program bugs have nothing to do with the fact that the program is a parallel program. Many (if not most) parallel program bugs are caused by the same mistakes that cause serial program bugs. [Pac97]*

Pour déboguer un programme parallèle, il est généralement préférable de procéder, en gros, comme suit :

a. On vérifie tout d'abord le bon fonctionnement du programme pour un unique processus s'exécutant sur un unique processeur... quand c'est possible!?

b. On vérifie le bon fonctionnement pour deux ou plusieurs processus mais s'exécutant sur un seul et unique processeur — voir plus bas.

c. On vérifie le bon fonctionnement du programme pour deux ou plusieurs processus s'exécutant sur deux processeurs.

d. On vérifie le bon fonctionnement avec plusieurs processus s'exécutant sur plusieurs processeurs.

Autres trucs :

- Lorsqu'on utilise des `printf` pour générer une trace d'exécution, il est préférable de mettre une instruction `fflush` immédiatement après le `printf`, pour assurer que les impressions se fassent dans un ordre qui reflète le plus possible l'exécution réelle :

  ```
  fflush( stdout );
  ```

  Il faut aussi savoir que les programmes parallèles peuvent parfois contenir des *Heisenbug*. Plus précisément, il peut arriver qu'un programme parallèle ne fonctionne pas — par exemple, à cause d'erreurs de synchronisation entre processus — mais que le programme fonctionne sans problème quand on rajoute des instructions `printf` pour tenter de le déboguer![2]

  Lorsqu'on utilise des traces d'exécution, il faut aussi tenir compte du fait que la quantité totale d'information générée *sera multipliée par le nombre de processeurs* — ce qui peut parfois rendre difficile de comprendre les informations ainsi produites.

[2] On parle de Heisenbug car l'effet d'observer le programme modifie son comportement, dans l'esprit du principe d'incertitude d'Heisenberg en physique.

- Toujours pour la génération de traces d'exécution avec `printf`, il est préférable d'indiquer explicitement le numéro du processus générant un élément de la trace. Si ce numéro est mis (en préfixe) au début de la ligne, alors on peut ensuite utiliser l'outil Unix `sort` pour produire une liste exacte de ce qui est imprimé par chaque processus — voir aussi plus bas pour une façon rapide d'indiquer les numéros de processus avec l'option d'exécution «`-l`».

- Lorsqu'une erreur d'exécution est générée par MPI, le message contient généralement le numéro du processeur (virtuel) et autres informations. Par exemple, le message suivant a été produit par le processeur 0 :

```
mpirun has exited due to process rank 0 with PID 32099 on
node quark20 exiting improperly. There are two reasons this could occur:
[...]
```

17.5 Erreurs typiques dans des programmes MPI

Note : Les deux premières sous-sections sont une traduction (partielle) de l'appendice C du livre de M.J. Quinn [Qui03].

17.5.1 Erreurs conduisant à des interblocages (*deadlock*)

- Un seul processus exécute une opération de communication collective (e.g., `MPI_Reduce`, `MPI_Bcast`, `MPI_Scatter`).

  Une solution à ce problème est de ne jamais mettre des opérations de communication collective dans du code s'exécutant conditionnellement.

- Deux ou plusieurs processus essaient d'échanger de l'information, mais en utilisant `MPI_Recv`, et ce avant que des appels appropriés à `MPI_Send` aient été effectués.

  Différentes solutions sont possibles, par exemple : toujours s'assurer d'effectuer les `MPI_Send` avant les `MPI_Recv` ; utiliser plutôt des appels à l'opération `MPI_Sendrecv` ; utiliser des appels à `MPI_Irecv`.

- Un processus essaie de recevoir des données d'un processeur, qui n'effectue jamais d'envoi approprié.

  Ceci est souvent causé par l'utilisation d'un mauvais numéro de processus. Lorsque possible, il est préférable d'utiliser les opérations collectives de communication. Si ce n'est pas possible, il faut s'assurer que le «patron» des échanges entre les processus soit le plus simple possible.

- Un processus essaie de recevoir des données *de lui-même.*

17.5.2 Erreurs conduisant à des résultats erronnés

- Incohérence au niveau des types utilisés dans l'opération `Send` vs. `Recv`.

  Pour éviter ce problème, il faut s'assurer au niveau du code source que chaque opération d'envoi possède une unique opération de réception correspondante, et vérifier (et re-vérifier) le code source.

- Arguments d'un appel à une opération de communication transmis dans le mauvais ordre.

17.5.3 Comportements non spécifiés de MPI

- «*The MPI specification purposefully does not mandate whether or not collective communication operations have the side effect of synchronizing the processes over which they operate.*»

  Donc, des opérations telles que `MPI_Bcast`, `MPI_Reduce`, etc., peuvent, ou non, créer une barrière de synchronisation entre les processus impliqués.

- «*According to the MPI standard, message buffering may or may not occur when processes communicate with each other using* `MPI_Send`.»

  Cette question sera abordée ultérieurement.

17.6 Autres options d'exécution

Divers options peuvent être spécifiées à `mpirun` pour aider à déboguer un programme :

- On peut demander, avec l'option «`--tag-output`», que chaque impression sur `stdout` soit automatiquement préfixée du numéro du processus : voir figure 17.1.

  Idem pour «`--timestamp-output`», mais un *timestamp* est aussi affiché.

```
# Sans l'option --tag-output :
# ============================

$ mpirun -np 3 --map-by node hello
quark20.hpc.uqam.ca
    numProc =  0, nbProcs =  3

quark21.hpc.uqam.ca
    numProc =  1, nbProcs =  3

quark22.hpc.uqam.ca
    numProc =  2, nbProcs =  3

----------------------------------------

# Avec l'option --tag-output :
# ============================

$ mpirun -np 3  --map-by node --tag-output hello
[1,0]<stdout>:quark20.hpc.uqam.ca
[1,0]<stdout>:  numProc =  0, nbProcs =  3
[1,0]<stdout>:
[1,1]<stdout>:quark21.hpc.uqam.ca
[1,1]<stdout>:  numProc =  1, nbProcs =  3
[1,1]<stdout>:
[1,2]<stdout>:quark22.hpc.uqam.ca
[1,2]<stdout>:  numProc =  2, nbProcs =  3
[1,2]<stdout>:
[1,0]<stdout>:----------------------------------------
```

Figure 17.1: Utilisation de l'option «`--tag-output`» pour indiquer le numéro du processus sur les lignes de sortie. Le premier nombre indique le *process jobid* alors que le deuxième indique le rang du procesus dans le communicateur.

- L'option «`--mca mpi_param_check 1`» permet que des vérifications soient effectuées en cours d'exécution.

  Par exemple, soit une communication réalisée entre deux processus distincts avec les deux instructions suivantes exécutées :

```
MPI_Send( NULL, 0, MPI_DATATYPE_NULL, 1, 0, MPI_COMM_WORLD );
.................
int signal;
MPI_Recv( &signal, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE );
```

  La Figure 17.2 montre l'effet d'exécuter cette communication. La troisième exécution montre aussi... que cette option *est activée par défaut dans Open-MPI — ce qui n'est pas le cas dans toutes les mises en oeuvre de MPI.*

```
// Processus 0
 MPI_Send( NULL, 0, MPI_DATATYPE_NULL, 1, 0, MPI_COMM_WORLD );

// Processus 1
int signal;
MPI_Recv( &signal, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE );
```

```
==========================================================================
```

```
$ mpirun -np 3  --mca mpi_param_check 0 hello
quark20.hpc.uqam.ca
        numProc =  0, nbProcs =  3

quark22.hpc.uqam.ca
        numProc =  2, nbProcs =  3

quark21.hpc.uqam.ca
        numProc =  1, nbProcs =  3
```

```
==========================================================================
```

```
$ mpirun -np 3  --mca mpi_param_check 1 hello
quark20.hpc.uqam.ca
        numProc =  0, nbProcs =  3

[quark20.hpc.uqam.ca:32099] *** An error occurred in MPI_Send
[quark20.hpc.uqam.ca:32099] *** on communicator MPI_COMM_WORLD
[quark20.hpc.uqam.ca:32099] *** MPI_ERR_TYPE: invalid datatype
[quark20.hpc.uqam.ca:32099] *** MPI_ERRORS_ARE_FATAL: your MPI job will now abort
--------------------------------------------------------------------------
mpirun has exited due to process rank 0 with PID 32099 on
node quark20 exiting improperly. [...]
```

Figure 17.2: Effet de certaines vérifications effectuées par `mpi_param_check`.
Note : Par défaut, dans les versions récentes d'`OpenMPI`, `mpi_param_check` vaut 1, donc pas besoin de le spécifier explicitement. (MCA = *Modular Component Architecture*)

Références

[MSM05] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[Pac97] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman Publ., 1997.

[Qui03] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.