TP2 MPI

Michel Donnet

October 25, 2023

Contents

1 Explication des algorithmes

1.1 Diffusion séquentielle

Le principe de cet algorithme est qu'il n'y a qu'un seul processeur qui envoie un message à tous les autres processeurs.

1.2 Anneau séquentiel

Le principe de cet algorithme est que chaque processeur envoie un message au processeur de rang supérieur à lui modulo nombre de processeurs (Si N est le nombre de processeurs, alors le processeur N - 1 va envoyer un message au processeur N modulo N = 0 ... Ainsi on fait une boucle). Et il recevra un message du processeur de rang inférieur à lui modulo nombre de processeurs (si on est le processeur 0, on va recevoir un message du processeur -1 modulo N = N - 1...)

1.3 Diffusion HyperCube

Le principe de cet algorithme est que chaque processeur échange avec le processeur dont le rang diffère de son rang de seulement un seul bit, et ainsi de suite...

On introduit donc un masque et on fait une opération de xor entre le rang du processeur et le masque, ce qui donne le rang du processeur différant de seulement un bit du rang du processeur courant. Si le rang de ce processeur est plus petit que le rang du processeur courant, on envoie un message à ce processeur, sinon on reçoit un message de ce processeur.

2 Explication du code

Dans tout notre code, j'ai défini $world\_size = N$ qui est le nombre de processeurs utilisés par le programme. Ensuite, chaque processeur est numéroté de 0 à $(N-1)$. Le numéro du processeur est dans notre cas unique et se trouve dans la variable $world\_rank$ de mon code.

2.1 Diffusion séquentielle

```
if (world_rank == 0) {
        for (int i = 1; i < world_size; i++) {
                MPI_Send(text, size, MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
}
else {
        char buffer[size];
        MPI_Recv(buffer, size, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
        printf("Your message");
}
```

À la première ligne de notre code, on regarde si on est le processeur de rang 0. Si c'est le cas, on envoie à tous les autres processeurs un message. Sinon, cela signifie qu'on n'est pas le processeur de rang 0, donc on lit le message envoyé par le processeur 0 et on affiche quelque chose.

2.2 Anneau séquentiel

```
MPI_Send(text, size, MPI_CHAR, (world_rank + 1) % world_size, 0, MPI_COMM_WORLD);
MPI_Recv(
         buffer,
         size,
         MPI_CHAR,
         (world_rank + world_size - 1) % world_size,
         0,
         MPI_COMM_WORLD,
         &status
);
printf("Your message");
```

On envoie un message au processeur de rang supérieur à notre rang, et on reçoit un message du processeur de rang inférieur à notre rang. Évidement, si N est le nombre de processeurs et qu'on est le processeur de rang N - 1, on va envoyer un message au processeur de rang 0 et si on est le processeur de rang 0, on va recevoir un message du processeur de rang N - 1, ce qui permet de faire justement un anneau. C'est pour cela que je fais un modulo nombre de processeurs dans le send et le receive de mon code.

2.3 Diffusion HyperCube

```
int mask = 1;
while (world_size > mask) {
       int process = world_rank ^ mask;
       if (process > world_rank) {
             MPI_Recv(
                      buffer,
                      size,
                      MPI_CHAR,
                      process,
                      0,
                      MPI_COMM_WORLD,
                      &status
             );
             printf("Your message");
       }
       else {
             MPI_Send(text, size, MPI_CHAR, process, 0, MPI_COMM_WORLD);
       }
       mask = mask << 1;
}
```

Tout d'abord, on commence par définir un masque à 1 (cela donne en binaire $000\ldots1$). Ensuite, on boucle tant que le masque est plus petit que le nombre de processeurs... En effet, on va faire à la fin du while un décalage de 1 bits vers la gauche (ainsi mask va passer de $000\ldots01$ à $000\ldots10$). Donc si par exemple on a 6 processeurs, on aura notre $world\_size$ qui sera égal à 6, soit 110 en binaire. Et notre masque va être égal tour à tour à 001, puis 010, puis 100, et on sortira de la boucle while quand le masque sera égal à 1000, soit quand le masque sera plus grand que notre $world\_size$.

Ensuite, supposons qu'on a un masque $001 = 1$ et le processeur $010 = 2$.

Le xor donnera donc $011 = 3$, ce qui est plus grand que le numéro ou rang de notre processeur, donc on réceptionne le message du processeur dont le rang est égal à cette valeur. Et si on a par exemple le processeur $011 = 3$, le xor donnera $010 = 2$, donc on envoie un message au processeur de rang $010 = 2$.

Et à la fin de notre boucle while, on fait un décalage de 1 bits vers la gauche du masque, ce qui nous donne un masque $010 = 2$.

2.4 Main

Voilà à quoi ressemble mon programme main.

```
 1  #include "exercises.h"
 2  #include <string.h>
 3  #include <iostream>
 4
 5  using namespace std;
 6
 7  int main(int argc, char * argv[]) {
 8          if (argc != 2) {
 9                  cerr << "Usage of the program" << endl;
10                  return -1;
11          }
12          int exercise_number = atoi(argv[1]);
13          switch (exercise_number) {
14                  case 1:
15                          exercise_1();
16                          break;
17                  case 2:
18                          exercise_2();
19                          break;
20                  case 3:
21                          exercise_3();
22                          break;
23                  default:
24                          cerr << "Usage of the program" << endl;
25                          return -1;
26          }
27          return 0;
28  }
```

Dans le main, on commence par vérifier que l'utilisateur a bien entré un paramètre lorsqu'il exécute le programme (lignes 8-11). Si ce n'est pas le cas, on le fait sortir du programme et on affiche un mode d'utilisation.

Si l'utilisateur a entré un paramètre, on le convertit en nombre (au moyen de la fonction atoi. Si elle échoue, elle retournera 0). Si on a les nombres 1, 2 ou 3, on exécutera l'exercice 1, 2 ou 3 suivant le nombre reçu, et si l'utilisateur a donné quelque chose différent de 1, 2 ou 3, on affiche un mode d'emploi du programme et on quitte.

3 Tests et discussion sur l'output

Pour tester mon code, j'ai simplement fait envoyer par les différents processeurs le message "Hello world !", et lorsque un processeur recevait un message, il affichait le message reçu, son rang et le rang du processeur ayant envoyé le message. Ainsi, je vérifiais "à la main" les résultats que j'avais obtenus en regardant si c'était cohérent avec ce qui m'étais demandé...

Mon code est composé de 3 fonctions, et il faut donner en paramètres lors de l'exécution du programme le numéro de l'exercice qu'on veut exécuter, à savoir 1 pour la diffusion séquentielle, 2 pour l'anneau séquentiel, et 3 pour la diffusion HyperCube

J'ai fait un makefile pour compiler et exécuter mon code. On peut directement exécuter un exercice en entrant "make 1" par exemple pour l'exercice 1.

```
CC = mpic++
OBJS = exercises.o
CFLAGS = -c -g -Wall
.PHONY = clean
tp2: main.cpp exercises.o
        $(CC) $(OBJS) main.cpp -o tp2
exercises.o: exercises.cpp
        $(CC) $(CFLAGS) exercises.cpp -o exercises.o
clean:
        rm $(OBJS) tp2
cleanoutput:
        rm ./err/* ./out/*
```

3.1 Diffusion séquentielle

Voilà l'output du programme de la diffusion séquentielle pour une exécution locale:

```
make
mpirun tp2 1

mpic++ -c -g -Wall exercises.cpp -o exercises.o
mpic++ exercises.o main.cpp -o tp2
Message received by processor 1 from processor 0: Hello world !
Message received by processor 2 from processor 0: Hello world !
Message received by processor 3 from processor 0: Hello world !
```

Ici, on veut que notre processeur 0 envoie un message aux autres processeurs... On peut remarquer avec l'output que notre objectif a bien été atteint, car on a bien le processeur 0 qui envoie un message à tous les autres processeurs.

3.2 Anneau séquentiel

Voilà l'output de l'anneau séquentiel pour une exécution locale:

```
make
mpirun tp2 2

Processor 3 receive from processor 2 the message: Hello World !
Processor 2 receive from processor 1 the message: Hello World !
Processor 1 receive from processor 0 the message: Hello World !
Processor 0 receive from processor 3 the message: Hello World !
```

Ici, on peut voir que le processeur 1 a envoyé un message au processeur 2, qui a envoyé un message au processeur 3 etc... Cela fait bien un cycle, donc on est content du résultat obtenu.

3.3 Diffusion HyperCube

Voilà l'output de la diffusion HyperCube, exécutée de nouveau localement.

```
make
mpirun tp2 3
```

```
Processor 0 received Hello world ! from processor 1
Processor 2 received Hello world ! from processor 3
Processor 0 received Hello world ! from processor 2
Processor 1 received Hello world ! from processor 3
```

Ici, notre masque est d'abord égal à 001. Donc le processeur $001 = 1$ envoie un message au processeur $000 = 0$ et le processeur $011 = 3$ envoie un message au processeur $010 = 2$. Puis notre masque est égal à 010. Donc le processeur $010 = 2$ envoie un message au processeur $000 = 0$ et le processeur $011 = 3$ envoie un message au processeur $001 = 1$. Notre output est bien ce qu'on attend, donc on est content.

Pour la diffusion HyperCube, il faut avoir un nombre pair de processeurs, sinon on va avoir une erreur, car tous les processeurs doivent travailler deux à deux, tandis qu'avec la diffusion séquentielle, on n'a pas besoin d'avoir un nombre pair de processeurs, ni dans l'anneau séquentiel.

4 Difficultés

Au début, j'ai eu des difficultés à faire tourner mon programme sur baobab, mais le compiler directement sur baobab en important le module foss a résoud mes problèmes de librairies. Voilà le contenu du script utilisé pour lancer le programme sur baobab:

```
#!/bin/sh
#SBATCH --job-name Michel_TP2          # Permit us to find easily our job
#SBATCH --output ./out/Michel_TP2-out.o%j    # Outputs will be written here
#SBATCH --error ./err/Michel_TP2-err.e%j     # Errors will be written here
#SBATCH --ntasks 16                    # Number of tasks in our job
#SBATCH --cpus-per-task 1              # Number of cpus per tasks
#SBATCH --partition debug-cpu          # Partition to use
#SBATCH --time 15:00                   # Maximum time execution

# Load modules for compiling and run program
module load foss
module load CUDA

echo $SLURM_NODELIST

# Compile program
make
```

```
# Run program. If the parameter is not given to execute an exercise,
# we print an error which will be in ./err/Michel_TP2-err.e%j
# Else, we execute program
if [ -n $1 ]; then
        srun --mpi=pmi2 ./tp2 $1
else
        echo "Error ! Usage of the script: ./run.sh [number of the exercise]" 1>&2
fi
```

Il faut donner le numéro de l'exercice qu'on veut exécuter en paramètre au script pour que celui-ci fonctionne. Cela donnera par exemple:

```
sbatch run.sh 1
```

On aura comme output 2 fichiers: un fichier nommé $\text{Michel}_{\text{TP2}}$-out.o%j qui se trouvera dans ./out/, et un fichier contenant les erreurs nommé $\text{Michel}_{\text{TP2}}$-err.o%j qui se trouvera dans ./err/. Si on ouvre les fichiers de ./out, on peut avoir les informations de l'output de notre programme exécuté sur les n processeurs. Si on ouvre les fichiers de ./err/, on aura tous les messages d'erreur.

J'ai eu comme autre problème le message d'erreur suivant:

```
srun: error: Couldn't find the specified plugin name for mpi/pmix_v3 looking at all fi
srun: error: cannot find mpi plugin for mpi/pmix_v3
srun: error: MPI: Cannot create context for mpi/pmix_v3
srun: error: MPI: Unable to load any plugin
srun: error: Invalid MPI type 'pmix_v3', --mpi=list for acceptable types
```

C'est pourquoi j'exécute ma fonction srun avec –mpi=pmi2 comme paramètre, car cela a résolu mon problème. En fait j'ai exécuté

```
srun --mpi=list
```

Et j'ai eu comme résultat

```
MPI plugin types are...
        none
        pmi2
        cray_shasta
```

Donc j'ai ajouté le paramètre –mpi=pmi2 à srun.