Calcul Haute Performance
TP n°1 - Prise en main avec MPI

Oguz Kaya
oguz.kaya@lri.fr

9/11/2018

Part 1

Hello world!

Question 1

a) On va commencer par écrire un program MPI complet dans lequel chaque processus récupère son rang et le nombre de processus dans `MPI_COMM_WORLD` puis les imprime. Une fois que vous avez le code (disont `hello-world.c`), compilez-le à l'aide de la commande

```
mpicc hello-world.c -o hello-world
```

puis lancez le program sur 8 processus à travers la commande

```
mpirun -np 8 ./hello-world
```

Que constate-t-on au niveau de l'affichage quand on l'execute plusieurs fois? Pourquoi?

Part 2

Tri parallèle d'un tableau bitonique

Dans cet exercice, on va essayer de trier un tableau d'entiers *bitonique*, c'est à dire les valeurs dans le tableau augment jusqu'à un certain indice, et descendent à partir de cet indice jusqu'à la fin. Par exemple, $1, 2, 5, 6, 8, 4, 2, 1$ est une séquence bitonique alors que $1, 2, 5, 6, 3, 4, 2, 1$ ne l'est pas car elle remonte après une descente ($6 - > 3 - > 4$).

Vous avez deux fichiers sources déjà fournis. Dans `bitonic-sort-skeleton.c`, on gére le lecture du rang de chaque processus dans la variable `procRank`, du nombre des processus disponible dans la variable `numProcs`, et des entiers dans le tableau `arr` (ce qui n'est rempli que dans le processus 0). A la fin, ce code vérifie également si le tableau `arr` est trié. Vous n'avez pas à toucher à ce fichier là!

Vous allez implanter dans le fichier `bitonic-sort-solution.c`. Ce code est directement inclu dans la fonction `main` du programme principal `bitonic-sort-skeleton.c` Les variables définiés que vous pouvez directement utiliser sont fournies à la tête du `bitonic-sort-solution.c` (elle sont déjà définies, ne les décommentez pas dans `bitonic-sort-solution.c`!). Pour le moment, vous pouvez ignorer les définitions des fonctions `MPI_ScatterSingleInt` et `MPI_GatherSingleInt`, et commencer votre implementation à partir de la dernière ligne du fichier.

On va trier un tableau bitonique de taille N en ordre non-décroissant en utilisant N processus (dont chaque processus contiendra un seul entier). On suppose que N est une puissance de 2 pour simplifier les choses. Pour le moment, vous pouvez démarrer avec $N = 8$ pour la suite. Exécutez le script `gen-bitonic-array.py` avec le paramètres `8` et `bitonic-array.txt` afin de générer un tableau bitonique de taille 8:

```
./gen-bitonic-array.py 8 bitonic-array.txt
```

Maintenant, compilez le code squelette avec le compilateur `mpicc` comme la suite

```
mpicc bitonic-sort-skeleton.c -o bitonic-sort
```

Finalement, exécuter le programme pour trier le tableau d'une manière séquentielle en tournant la commande

```
mpirun -np 8 ./bitonic-sort bitonic-array.txt sequential
```

, ce qui devrait afficher le tableau original et le tableau trié. Alors, on va essayer de trier le tableau en parallèle avec la commande

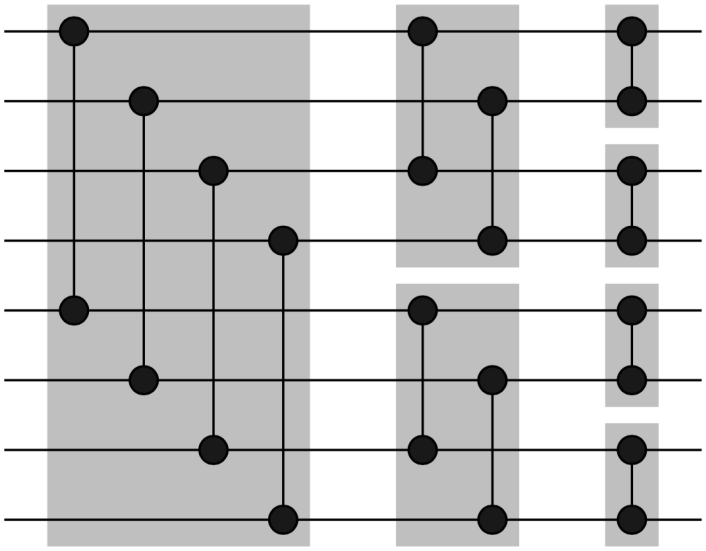
```
mpirun -np 8 ./bitonic-sort bitonic-array.txt parallel
```

, ce que ne trie pas le tableau en effet car on n'a rien implanté dans `bitonic-sort-solution.c`!

Question 2

a) On suppose que on n'a assez de mémoire que dans le processus 0, ce qui lit et stocke la séquence bitonique dans le tableau `arr` (ce qui est déjà rempli par le code squelette). Il est donc interdit d'alloer un tableau dans les autres processus, pourtant, on peut déclarer autant de variables que l'on souhaite. Alors, on a le tableau `arr` est alloué et rempli dans le processus 0. Premièrement, on va distribuer ce tableau aux processus tel que l'element `arr[i]` est possedé par le processus de rang i. En effet, on va utiliser la fonction `MPI_Scatter` afin de réaliser cette opération, et mettre cet element dans la variable locale `procElem` de chaque processus.

b) Maintenant que le tableau est distribué, on va itérer la-dessus en $log_2 N$ pas afin de le trier. A chaque itération, chaque processus devrait trouver le rang de son "pair", échanger son élément avec lui et garder le minimum (s'il a le rang inférieur) ou maximum (s'il a le rang supérieur) de ces deux éléments en fonction de sa position. On va effectuer la communication à l'aide de `MPI_Send` et `MPI_Recv`. N'hesitez pas à regarder le cheatsheet MPI pour l'utilisation de ces fonctions. On fournit le diagramme suivant qui résume les échanges à faire pour $N = 8$.



c) Est-il trié le tableau maintenant? Alors, on le verra tout à l'heure. Cette fois-ci, on va effectuer l'inverse de la communication que l'on a fait dans la première partie. On va "rassambler" les entiers des processus dans le processus de racine (ayant le rang 0) à l'aide de la fonction `MPI_Gather`. Une fois que vous l'avez fait, le code va automatiquement vérifier si le tableau est trié, et afficher une erreur s'il ne l'est pas. Une fois que le code fonctionne pour $N = 8$, on va le tester pour N égale aux puissances de 2 entre 2^1 et 2^6.

d) Au lieu d'utiliser `MPI_Send` et `MPI_Recv`, on pourrait profiter de `MPI_Isend` et `MPI_Irecv` afin de ne pas devoir mettre en place les envoies et les réceptions dans le bon ordre. Pour le coup, on effectue la communication avec ces variantes. Il ne faut pas oublier d'appeler `MPI_Wait` à la fin de veiller à ce que la communication est faite!

e) Au lieu d'utiliser `MPI_Send` et `MPI_Recv`, on pourrait aussi faire un appel à `MPI_SendRecv` afin d'effectuer l'échange des entiers entre les deux processus d'un seul coup. Allons-y!

Question 3

a) Dans cet exercice, on va implanter une version basique de la routine `MPI_Scatter` dont type de donné est toujours `int` et taille de message est toujours 1. On va le faire à l'aide de `MPI_Send` et `MPI_Recv`. La signature de la fonction est déjà fournie dans le code `scatter-gather.c`. Implementez-la, puis remplacer la adéquatement avec l'appel à `MPI_Scatter` dans votre implementation précèdente. Cette fois-ci, il faudrait compiler le code avec le nouveau fichier source `scatter-gather.c` comme la suit

```
mpicc bitonic-sort-skeleton.c scatter-gather.c -o bitonic-sort
```

b) Faisons pareil, mais pour remplacer `MPI_Gather` pour le coup.

Part 3

N'oublion pas de garder le code précieux que l'on a developpé pour la suite!