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

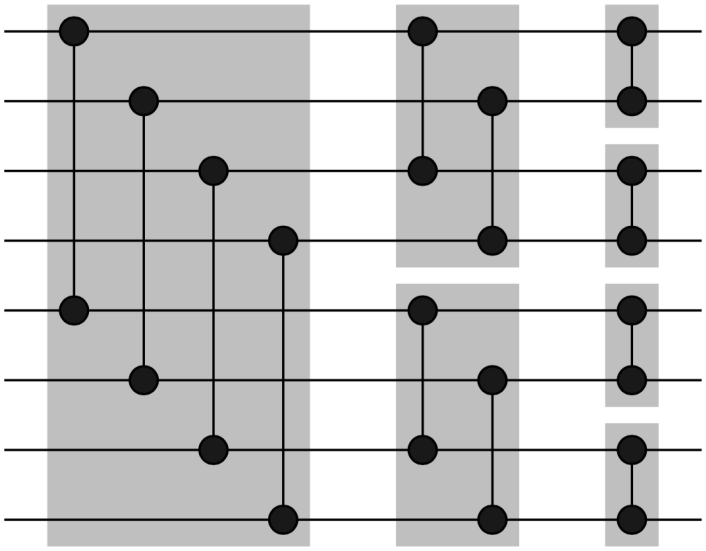
```
mpirun -np 8 ./bitonic-sort bitonic-array.txt parallel
```

, 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) Maintemant 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) Is the bitonic array sorted now? Are you sure? Well, we will see about that in a moment... Now we will try to perform the "mirror image" of the communication that we did in the first part. We will "gather" these scattered (and hopefully sorted!) elements in processes in the `arr` array of the master process (with rank 0). Refer to the MPI cheatsheet and documentation for the usage of `MPI_Gather`. Once you do this, the skeleton code will automatically validate if the array is sorted, and print an error otherwise for you to debug your code accordingly. No bread and water to you until the code sorts correctly! Now that you validated your code working for $N = 8$, try to test it for powers of two, from $N = 2$ up to $N = 64$.

d) Instead of using `MPI_Send` and `MPI_Recv`, we can make use of `MPI_Isend` and `MPI_Irecv`, which should liberate us from having to validate that we send and receive calls are in the right order. This time, do the communication using these non-blocking variants. Do not forget to use `MPI_Wait` at the end to make sure that the communication is completed!

e) Instead of doing one `MPI_Send` and `MPI_Recv`, once can also perform `MPI_SendRecv` to accomplish both communications at the same time (which could potentially be done faster)! Try to replace sends and receives in your code with `MPI_SendRecv`, then make sure it works correctly.

Question 3

a) Try to implement a basic version of the MPI routine `MPI_Scatter` in which the data type is set to be `int` and the block size is always 1. You should be only using two MPI routines `MPI_Send` and `MPI_Recv`. The function signature is provided in the code `scatter-gather.c`. Try to implement the function there, then replace it with the `MPI_Scatter` you use in the previous exercise. This time, when you compile the code, do not forget adding `scatter-gather.c` to the list of source files in `mpicc`. Make sure that everything works as expected!

b) Do the same, this time for `MPI_Gather`.

Part 3

Do not forget to keep a copy of the precious code you developed for later!