Rapport I03 : TP 2

Gabriel Hadjerci

Dans ce TP on considère le code de différences finies déjà utilisé dans les deux précédents TP. On cherche à fusionner la version MPI et la version Cuda afin d'obtenir un code parallélisé hybride pouvant être utilisé sur une machine dont chaque noeud contient une carte graphique.

1 Initialisation et structures de données

Avant de commencer les calculs il faut initialiser MPI et les structures de données. Pour cela on part du *main.cxx* de la version MPI. On ne modifie rien car tout est déjà pris en compte pour l'utilisation de MPI, l'initialisation comme la synchronisation des processeurs.

Cependant, **les calculs vont être réalisés sur les cartes graphiques**, c'est pourquoi il faut faire attention aux structures de données. Dans ce code, il y a deux classes utilisées :

— Les paramètres décrits dans $parameters.*$;
— Les valeurs décrites dans $values.*$.

Les paramètres contiennent, par exemple, les frontières du domaines où les indices utiles. C'est pourquoi on va conserver les fichiers $parameters.*$ du code MPI qui permettent de subdiviser le domaine en sous-domaines attribués à chaque processeur. De plus ils contiennent des données en plus comme les indices locaux et globaux, le rang du processus et des informations sur les voisins.

Les valeurs, quand à elles, contiennent les données de calcul. Comme on doit utiliser des données sur le CPU et sur le GPU, on conservera les fichiers $values.*$ du code Cuda. Finalement il y aura un set de valeurs par processus MPI défini sur le sous-domaine attribué au processus. Il faudra juste faire attention a adapté certaines fonction des valeurs comme $Values :: boundaries()$ ou $Values :: plot(int\ order)$ qui prennent en compte le rang et les voisins du processus concerné.

2 Calculs

On s'intéresse ensuite aux calculs qui auront lieux sur les cartes graphiques. Déjà, il n'y a pas besoin de modifier les codes Cuda *iteration.cu* et *variation.cu*. En effet, ce sont les mêmes calculs mais cette fois-ci sur les sous-domaines de chaque processus. L'appel de ces fonctions dans le code *scheme.cxx* reste donc le même que dans la version Cuda. La fonction $Scheme :: iteration\_domaine$ est donc celle de Cuda.

Pour conclure le calcul, il ne reste plus qu'a faire la réduction MPI dans la fonction $Scheme :: iteration()$. Pour cela on ajoute

$$MPI\_Allreduce(\&du\_sum\_local, \&du\_sum, 1, MPI\_DOUBLE, MPI\_SUM, m\_P.comm());$$

où du_sum_local est la sortie de $Scheme :: iteration\_domaine$ et du_sum la variation finale.

3 Synchronisation MPI

Enfin, il ne reste plus qu'à gérer les transferts entre voisins. Dans la version MPI cette action est réalisé dans la fonction $Scheme :: synchronize()$. Cependant dans la version MPI, les données sont sur les processeurs. Ici ça n'est pas le cas, les données sont sur les GPU. Il faut donc rapatrier les données GPU sur les CPU, faire l'échange en utilisant la fonction $Scheme :: synchronize()$, puis renvoyer les données sur les GPU.

Pour ce faire, on rajoute au début de la fonction $Scheme :: synchronize()$ un appel à la fonction $copyDeviceToHost$ de $Cuda.cu$ de tel sorte que la mémoire GPU soit transmise sur le CPU. Il faut faire attention et mettre une barrière MPI juste après pour s'assurer que tous les CPU ont bien les bonnes données au moment de la synchronisation MPI.

Enfin, on renvoit les données sur les GPU à l'aide de la fonction $copyHostToDevice$ de $Cuda.cu$ une fois la synchronisation MPI terminée.