I03 TP3

Téo Boutin

Février 2021

1 Introduction

Par rapport au code MPI de base, 3 étapes seront modifiées. A l'initialisation, il faudra allouer de la mémoire sur le GPU et l'initialiser. L'itération devra être réalisé par le GPU au lieu du processeur du noeud MPI local. Enfin les communications entre noeuds MPI seront plus complexes car il faudra au préalable transférer els données calculées par les GPU vers les noeuds MPI.

2 Initialisation

Pour l'initialisation, je pense qu'il suffit d'utiliser le type Values du code CUDA fourni, et que l'on peut appeler les mêmes fonctions dans `main.cxx`.
Les valeurs seront ainsi initialisées sur les GPUs et CPUs, et les découpages entre processus MPI devraient être corrects car ils sont fait avant l'initialisation, lors de la création des paramètres. On pourra donc directement faire les calculs sur le GPU avec les données présentes dessus.

3 Itération par les GPU

Pour que l'itération soit faite par les GPUs, on peut appeler les fonctions du code CUDA iteration_wrapper et variation_wrapper dans la fonction Scheme::iteration_domaine du code MPI. Cela réalise l'itération par le GPU et permet de récupérer la variation pour le noeud MPI local. On peut ensuite faire un MPI_allreduce dans la fonction Scheme::iteration pour obtenir la variation totale.

Les fonctions seraient donc comme ceci :

```
1  void Scheme::iteration()
2  {
3    double m_duv_local = iteration_domaine(
4        m_P.imin(0), m_P.imax(0),
5        m_P.imin(1), m_P.imax(1),
6        m_P.imin(2), m_P.imax(2));
7
8    MPI_Allreduce(&m_duv_local, &m_duv, 1, MPI_DOUBLE, MPI_SUM, m_P
       .comm());
9    m_t += m_dt;
10   m_u.swap(m_v);
11   m_u.synchronized(false);
12 }
```

```
double Scheme::iteration_domaine(int imin, int imax,
                                 int jmin, int jmax,
                                 int kmin, int kmax)
{
  iterationWrapper(m_v, m_u, m_dt, m_n,
                   imin, imax, jmin, jmax, kmin, kmax);

  m_v.synchronized(false);
  return variationWrapper(m_u, m_v,
                          diff, partialDiff,
                          m_n[0]*m_n[1]*m_n[2]);
}
```

4 Communications/synchronisation des noeuds MPI

Dans la fonction `Scheme::synchronize`, il faut ajouter plusieurs choses pour assurer les communications.
Il faut que chaque noeud MPI récupère depuis le GPU les données à échanger avec les autres noeuds. Après les échanges MPI, il faut cette fois envoyer les nouvelles données aux GPUs.

Pour faire cela, je pense ajouter les 2 fonctions suivantes dans `values.cxx` :

```
void Values::syncHostWithDevice() const
{
if (!h_synchronized) {
    copyDeviceToHost(h_u, d_u, nn);
    h_synchronized = true;
  }
}

void Values::syncDeviceWithHost() const
{
if (!h_synchronized) {
    copyHostToDevice(h_u, d_u, nn);
    h_synchronized = true;
  }
}
```

On peut ensuite appeler `m_u.syncHostWithDevice()` au début et `m_u.syncDeviceWithHost()` à la fin dans la fonction `Scheme::synchronize` du code MPI fourni. Il faudrait également mettre un `m_u.synchronized(false)` juste après les échanges MPI.

On peut peut-être gagner en efficacité sur cette partie en ne transferant entre le GPU et le CPU que les données qui devront être échangées par les processus MPI, mais je ne vois pas comment faire avec les fonctions CUDA.