Modèles et techniques en programmation parallèle hybride et multi-cœurs - TP3

Pierrick Guichard

Février 2020

1 Introduction

On se propose dans ce projet de résoudre le problème de la chaleur en 3 dimensions :

$$\begin{aligned} &\textit{Trouver } u : (x,t) \mapsto u(x,t) \ \textit{ avec } x \in \mathbb{R}^3 \ \textit{ et } t \geq 0 \ \textit{ tel que} \\ &\qquad \frac{\partial u}{\partial t} = \Delta u + f(x,t) \qquad \textit{dans } \Omega \\ &\qquad u(x,0) = u_0(x) \qquad \textit{dans } \Omega \\ &\qquad u(x,t) = g(x) \qquad \textit{sur } \partial\Omega, \ t \geq 0 \end{aligned}$$

On utilise des différences finies pour approcher les dérivées partielles et on découpe Ω en $n_0 \times n_1 \times n_2$ subdivisions.

Plusieurs versions de ce code sont fournies : un code séquentiel, un code parallélisé avec openMP (grain fin), un code parallélisé avec MPI et un code parallélisé avec CUDA. Le but de ce TP est de proposer un code permettant de réaliser une version hybride MPI/Cuda : le domaine serait divisé en subdivisions, et chaque sous-domaine serait affecté à un thread MPI ; ensuite le calcul lié à ce sous-domaine serait effectué sur GPU grâce à CUDA.

2 Modifications du code

2.1 Fichiers à modifier

On se propose de partir de la version MPI du code et d'incorporer progressivement du code CUDA qui traite le calcul sur chaque noeud. On utilise tous les fichiers CUDA présents dans la version CUDA, et on modifie le fichier *scheme.cxx* de manière à à créer une interface entre le CPU et le GPU.

2.2 Algorithme

Ci-dessous on détaille l'algorithme hybride MPI-CUDA. Les calcul effectués sur le GPU sont écrits en couleur rouge.

1. Exécution du fichier *main.cxx* de la version MPI, conservée à l'identique. Initialisation des paramètres et d'une structure *scheme*, de type MPI.
2. Début de la boucle en temps. Appel de la fonction *scheme* :: *iteration*().
   (a) Calcul des indices du sous-domaine spécifique au thread MPI, réalisé de manière identique à ce qui est fait dans la version MPI.
   (b) Appel de la fonction *scheme* :: *iteration_domaine*(*idx_locaux*) de type CUDA, pour calculer sur GPU la solution du sous-domaine MPI.
      i. Appel de la fonction *iterationWrapper*() pour calculer la solution locale sur GPU, barrière CUDA, transfert de la solution au CPU.

ii. Calcul de la variation locale sur GPU avec $variationWrapper()$, barrière CUDA, transfert de la valeur au CPU.

(c) Barrière MPI. On dispose à ce stade de la valeur de la solution et de la variation pour chaque sous-domaine.

(d) Réduction MPI pour calculer la valeur de la variation globale.

(e) Échanges des valeurs de la solution aux bords de chaque sous-domaine (gérés chacun par un thread MPI) grâce à la fonction $scheme :: synchronize()$. Ceci est réalisé dans le main, de type MPI.

3. Fin de la boucle en temps et export des quantités d'intérêt (valeur de la solution, temps de calcul).

En résumé, on a seulement ajouté les fichiers CUDA à la version MPI, et on a adapté le fichier *scheme.cxx* de manière à réaliser une interface entre les deux version. On vérifie en particulier qu'aucune fonction MPI n'est appelée depuis le GPU. En particulier on fait attention de positionner les barrières aux bons endroits. Les barrières CUDA sont placées dans les fonctions $iterationWrapper()$ et $variationWrapper()$. On a besoin d'une unique barrière MPI placée dans la fonction $scheme :: iteration()$.

2.3 Structure de données

Du fait que le calcul est effectué sur GPU avec CUDA nous sommes obligés d'utiliser des structures de calcul adaptées à cet effet. Ainsi, nous utilisons les fichiers *values.cxx* et *values.hxx* de la version CUDA afin de fournir les structures de stockage de la solution. En particulier, on dispose dans ces structures de pointeurs pour stocker la solution sur le CPU (h_u), sur le GPU (d_u), ainsi qu'un booléen nous indiquant si ces deux valeurs sont égales ($h\_synchronized$). Ainsi, lorsqu'une nouvelle itération de la solution est effectuée il faut prendre garde de mettre la valeur de ce booléen à $False$ tant que la nouvelle valeur n'a pas été envoyée vers le CPU.

2.4 Détail des échanges de données

L'algorithme hybride MPI-CUDA nécessite un certain nombre de transferts de données qui sont détaillés dans le schéma ci-dessous.

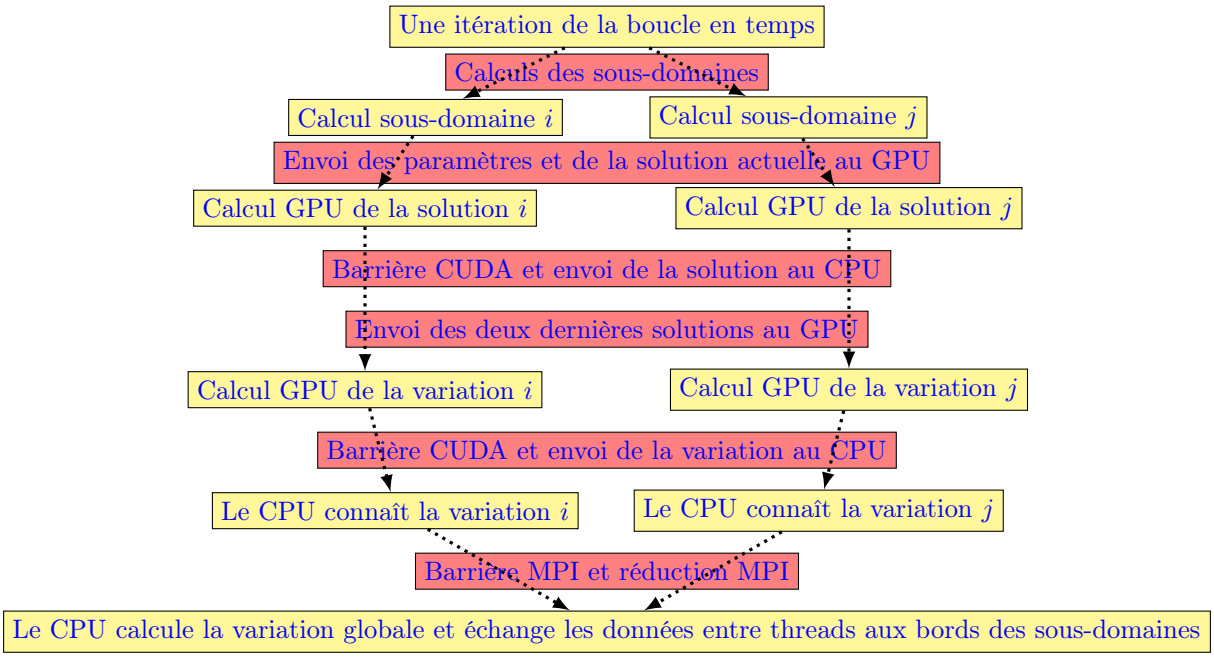

FIGURE 1 – Schéma récapitulatif de l'échange des données durant l'exécution de l'algorithme, pour deux threads i et j.

3 Conclusion

Ce projet nous a permis d'élaborer un code hybride MPI-CUDA qui permet de tirer un maximum de performance d'un machine disposant par exemple de cartes graphiques affectées chacunes à un noeud de calcul. Du fait que le code n'a pas été écrit ni exécuté il est possible que certains détails aient été oubliés ou qu'il reste d'autres parties du code à adapter mais celles-ci devraient pouvoir se déduire aisément de la structure globale du code présentée ci-dessus.