AMSI03-1

Nikita Matckevich

19.12.2020

1 Première partie:

Le code était parallélisé avec OpenMP grain fin. Les boucles parallélisées sont:

1. les boucles 3d pour initialiser les champs dans `values.cxx:init`
2. les boucles 2d pour définir les valeurs du bord `values.cxx:boundaries`
3. la boucle principale dans la fonction `scheme.cxx:iteration_domaine` (la réduction était ajoutée pour la variation)

Toutes les opérations de MPI sont executées dans les parties séquentielles du code.

2 Seconde partie:

Le code était parallélisé avec OpenMP grain grossier:

1. La seule région parallèle a été créée dans la fonction principale juste après la construction de l'objet `Values` jusqu'au point d'arrêt du temporisateur `T_total`. Sans prendre en compte les pragmas, la structure du code à l'intérieur de la région était inchangée. Les pragmas `single` ont été utilisés pour la journalisation.
2. La fonction `values.cxx:boundaries` contient plusieurs boucles 2D et elles ne peuvent pas être parallélisées avec les fonctions min-max fournies pour les indices. Pour simplifier le code, nous avons utilisé le `pragma omp for` dans cette fonction (donc les calculs explicites de partitionnement d'index ne sont pas nécessaires).
3. À l'intérieur de la fonction `scheme.cxx:iteration`, chaque thread exécute `scheme.cxx:iteration_domaine` sur sa propre partie du maillage et renvoie la somme locale de variation. À l'étape suivante, la variation globale est calculée comme la somme des sommes locales pour chaque thread, la sommation se situe dans la région atomique `omp atomic`. Ensuite, la barrière garantit que tout le champ est mis à jour pour chaque processus

paradigme	1t/8p	2t/4p	4t/2p	8t/1p
Sans OpenMP	15.1s (0.101s)	—	—	—
Grain fin	15.4s (0.010s)	23.1s (0.010s)	26.6s (0.081s)	26.3s (0.047s)
Grain grossier	35.9s (0.392s)	47.5s (0.286s)	48.8s (0.270s)	46.1s (0.001s)
Speedup g. fin	0.98	0.66	0.57	0.58
Speedup g. grossier	0.42	0.32	0.31	0.33

Table 1: Comparaison des accélérations pour différentes techniques de parallélisation OpenMP et pour different nombre de processus MPI et de threads OpenMP.

MPI. A l'intérieur d'une région suivante (`omp single`) la mise à jour de la variable de temps et l'initialisation de la variation globale à zéro sont effectuées, ainsi que l'échange de deux vecteurs de données. L'opération `MPI Allreduce` est effectuée dans cette région pour calculer la somme totale pour le domaine. Finalement, la barrière garantit que la somme peut être utilisée par "caller" lorsque la fonction retourne. Ainsi, les side-effets indésirables ne sont pas créés après l'exécution de la fonction. Les variables partagées ont été ajoutées dans le class `scheme` pour stocker les valeurs locales de chaque processus MPI.

La partie importante de l'algorithme de parallélisation "grain grossier" est une partition de domaine de chaque processus sur sous domaines et le processus de délégation de sous-zones à différents threads. Pour cela, les fonctions `i..._thread(*, *)` ont été utilisées. Initialement chaque domaine de calcul MPI est rectangulaire et peut etre representé comme l'ensemble des indices (i, j, k), $i \in (i_{min}, ..., i_{max})$, $j \in (j_{min}, ..., j_{max})$, $k \in (k_{min}, ..., k_{max})$. L'idée de partition est de trouver une dimension parmi i, j, k telle que le nombre des indices de domaine est maximal le long de cette dimension. Par exemple, dans notre cas la dimension k sera choisie, car le nombre des indices k est égal au nombre pour j et supérieur ou égal au nombre des indices de i.

3 Analyse des résultats

Les accélérations pour différents types de paradigmes OMP et pour différents nombres de threads et de processus MPI sont présentées dans le tableau 1.

Théoriquement, l'accélération de la parallélisation OMP grain grossier devrait être meilleure, car les créations et destructions dynamiques de threads sont minimisées. En pratique, j'obtiens que le temps d'execution est augmenté en 200% pour la parallélisation grossiére. La raison de ce changement n'est pas comprehensible pour moi. Ma version est que le niveau d'optimisation de compilateur de version `R` de `...CoarseGrain` n'est pas la même. Cependant, même avec une autre version du code binaire on ne peut pas avoir de changements si forts de temps d'execution. Il y a certainment de conflits interns dans mon code que je n'arrive pas à comprendre.