Master 2 Calcul Scientifique et Applications 2017-2018
Programmation parallèle et sur GPU

Introduction à `openMP`

Exercices préliminaires

1. **Hello World** : écrivez un programme multi-thread dans lequel chaque thread affiche un message d'accueil avec son rang ainsi que le nombre total de threads participant à la région parallèle. Faites varier le nombre de threads à l'intérieur du programme pendant l'exécution et réaffichez le message d'accueil. Vérifiez également la valeur renvoyée par `omp_get_max_threads()` après chaque changement. Quelle est l'utilité de cette fonction ?

2. **Distribution du travail ou "Qui fait quoi ?"** : créez un tableau d'entiers de longueur N de l'ordre de 100. Initialisez par une boucle chaque élément du tableau avec le rank du thread qui est en train d'exécuter. Imprimez ensuite les éléments et le nombre d'itérations traitées par chaque thread. Faites varier le `schedule` et le `chunksize`. Que voyez-vous ? Notez en particulier la valeur par défault du `chunksize`. Que se passe-t-il si le nombre d'itérations n'est pas un multiple du nombre de threads ?

Exercices à rendre

1. **Multiplication matricielle** : écrivez un programme qui effectue la multiplication de deux matrices rectangulaires. Les matrices doivent être remplies avec des nombres réels aléatoires. Ajoutez les directives `openMP` dans le code pour faire exécuter la multiplication en parallèle. Mesurez le temps de calcul en utilisant un ou plusieurs threads, pour différentes tailles de matrice. Changez `schedule` de `static` à `dynamic`, et faites varier `chunksize`. Y a-t-il une différence dans les temps de calcul ? Quelle combinaison de `schedule` et de `chunksize` donne le meilleur temps de calcul ?

2. **Recherche de nombres premiers** : écrivez un programme qui compte le nombre de nombres premiers entre 1 et un entier fourni par l'utilisateur. Utilisez l'algorithme naïf : pour tester si un entier N est premier ou non, vérifiez s'il est divisible exactement par des entiers plus petits. Le programme doit afficher comme résultat le nombre de nombres premiers trouvés : il n'est pas nécessaire d'afficher les nombres premiers eux-mêmes. Ajoutez les directives `openMP` pour faire la recherche en parallèle, et mesurez le temps de calcul pour un et pour plusieurs threads. Changez `schedule` de `static` à `dynamic` : y a-t-il une différence dans les temps de calcul ? Variez également la taille des "chunks" et comparez avec la valeur par défaut.

PTO

3. **Histogramme** : écrivez un programme qui remplit un tableau carré de dimension N avec des valeurs réelles aléatoires entre 0 et 10. La valeur de N doit être lue par le programme. Construisez ensuite un histogramme des valeurs du tableau, en comptant le nombre de valeurs entre 0 et 1, 1 et 2, ... Le programme doit d'abord construire cet histogramme en exécution mono-thread (c'est-à-dire en mode série), et ensuite en parallèle en utilisant la directive atomique. Dans les deux cas, imprimez le nombre de valeurs dans chaque gamme afin de vérifier le bon fonctionnement du programme. Comparez les temps de calcul : que remarquez-vous ? Expliquez vos observations et modifiez ensuite le code pour améliorer sa performance.

4. **Tri par transposition impaire-paire** : cet algorithme réalise le tri de n éléments, où n est pair, en alternant successivement deux types de phase de tri. Soit (a_1, a_2, ..., a_n) la séquence à trier : dans la phase dite "impaire", chaque élément avec indice impair est comparé avec son voisin de droite, et ils sont échangés si le premier est plus grand que le deuxième. De la même façon, dans la phase "paire", chaque élément avec indice pair est comparé et éventuellement échangé avec son voisin de droite. En appliquant successivement une phase impaire suivie d'une phase paire, le tableau est trié après n phases.

   Ecrivez un programme qui effectue un tel tri sur un tableau de nombres réels, dont la longueur doit être fournie par l'utilisateur. Le programme doit afficher un message pour signaler si le tableau a bien été trié ou pas. Ajoutez les directives `openMP` pour réaliser le tri en parallèle. Comparez le temps nécessaire pour réaliser le tri en utilisant un ou plusieurs threads, pour quelques valeurs de n suffisamment grandes pour avoir un temps d'exécution non négligeable.

   Faut-il modifier le programme pour traiter le cas où n est impair ? Si oui, comment ?

5. **Dynamique moléculaire** : téléchargez le programme `md.cpp` du site du cours.

   Le programme `md.cpp` met en oeuvre un algorithme de simulation de dynamique moléculaire classique. Des milliers de particules sont confinées dans une boîte, dans des positions aléatoires et avec des vitesses et accélérations initiales nulles. Les particules interagissent entre elles via un potentiel et commencent à se déplacer. L'algorithme consiste à calculer la force totale agissant sur chaque particule à un instant t puis son déplacement et sa vitesse dus à cette force. Le programme affiche régulièrement les énergies potentielle et cinétique ainsi que leur somme afin de vérifier la conservation de l'énergie mécanique et la précision du calcul.

   Ajouter des directives `openMP` pour exécuter ce programme en paralllèle. Quel est le gain en performance par rapport à une execution en série ?

Rappel : Pour mesurer le temps de calcul d'une région parallèle, on utilise `omp_get_wtime` :

```
double t0 = omp_get_wtime();
#pragma omp parallel for
for (...) {...}
double t1 = omp_get_wtime();
```