Programmation Parallèle
TP - Introduction à OpenMP

Oguz Kaya
oguz.kaya@universite-paris-saclay.fr

Pour compiler un code `programme.cpp` avec OpenMP et générer l'executable `programme`, saisir

```
g++ -O2 -std=c++11 -fopenmp programme.cpp -o programme
```

Part 1

Hello World

Exercice 1

a) Écrire un programme qui ouvre une région parallèle dans laquelle chaque thread imprime son identifiant.

b) En suite, dans la même region parallèle, afficher "Hello World" par un seul thread.

c) Essayer de faire varier le nombre de threads par les trois manières que l'on a rencontré dans le cours (la variable d'environnement `OMP_NUM_THREADS`, la fonction `omp_set_num_threads()` et la clause `num_threads()`). Quel est l'ordre de préséance entr'eux?

Part 2

La somme d'un tableau d'entiers

Exercice 2

a) Écrire un programme C/C++ qui initialise un tableau `A[N]` de flottants tel que `A[i] = i` pour $0 \leq i < N$.

b) Faire une deuxième boucle qui calcul la somme des éléments de `A[N]`.

c) Paralléliser les deux boucles avec `#pragma omp for`.

d) Maintenant, paralléliser la deuxième boucle avec `#pragma omp sections` ayant 4 sections tel que chaque section parcours $N/4$ itérations de la boucle. Veillez à ce qu'il n y ait pas de concurrence parmi les threads en effectuant la réduction des sommes partielles.

e) Comparer le temps d'exécution séquentielle et le temps d'exécution parallèle du programme.

Part 3

Mergesort en parallèle avec OpenMP sections

Le but de cet exercice est de trier un tableau d'entiers en parallèle à l'aide de l'OpenMP sections. Le fichier `mergesort.cpp` fournit déjà un code squelette qui alloue et initialise le tableau `A[N]` à trier ainsi que `temp[N]` que l'on utilise comme "buffer" dans l'algorithme de mergesort.

Exercice 3

a) Créer une région parallèle avec 4 sections (ou 8, s'il y a au moins 8 cœurs dans la machine, à vérifier avec la commande `lscpu`) qui trie chacune $N/4$ éléments consécutifs du tableau $A[N]$ avec `std::sort`.

b) Dans la même région parallèle, après avoir terminé ses 4 sections, créer 2 sections chacune fusionnant deux tableaux triés de taille $N/4$ en un seul tableau trié de taille $N/2$, sur le buffer `temp[N]`. Pour ce faire, profiter de la fonction `merge` fournit dans le squelette.

c) Finalement, en dehors de la région parallèle, fusionner les deux tableaux triés de taille $N/2$ en un seul tableau de taille N, soit le tableau $A[N]$ trié.

d) Comparer le temps d'exécution séquentielle (avec 1 seul thread) et le temps d'exécution parallèle du programme. Quelle est l'accéleration obtenue? Quelle est l'éfficacité?

Part 4

Le calcul de π

Le nombre π peut être défini comme l'intégrale de 0 à 1 de $f(x) = \frac{4}{1+x^2}$. Une manière simple d'approximer une intégrale est de discrétiser l'ensemble d'étude de la fonction en utilisant N points.

On considére l'approximation suivante avec $s = \frac{1}{N}$:

$$\pi \approx \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x \approx \sum_{i=0}^{N-1} s \times \frac{f(i \times s) + f((i+1) \times s)}{2}$$

On écrira un programme qui parallélise une approximation de la valeur de π en utilisant OpenMP. On effectuera deux façons différentes de parallèlisation. Un code squelette `calcul-pi.cpp` est fourni pour travailler dessus.

Exercice 4

a) D'abord, écrire le code sequentiel qui calcule correctement la valeur π.

b) On peut alors répartir le calcul de π parmi P threads. Au premier, on parallelisera tout simplement la boucle principale à l'aide de `#pragma omp for`. Tester la performance en utilisant de differents nombres de threads.

c) Pour la deuxième stratégie "faite à la main", chaque thread parcourira N/P indices consécutifs dans la région parallèle pour calculer son `piLocal`. C'est à dire, on ne va pas utiliser `#pragma omp for`; on va plutôt répartir les itérations parmi les threads nous-même. Une fois que c'est calculé, les threads additioneront les uns après les autres leur `piLocal` dans `pi` qui soit une variable partagée par tous les threads (il faudrait utiliser une région `critical` ou `atomic` afin d'éviter le problème d'écriture concurrente en cet étape). Tester la performance en utilisant de differents nombres de threads.