Programmation Parallèle
TP - Introduction à OpenMP

Oguz Kaya
oguz.kaya@lri.fr

Part 1

Hello World

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

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

Exercise 1

a) Ecrire 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()`)

Part 2

Calcul de Pi

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 va proposer un programme qui parallélise une approximation de la valeur de PI en utilisant OpenMP. On effectuera deux façons différentes de parallèlisation.

Exercise 2

a) D'abord, ecrire le code sequentiel qui calcul 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 strategie "à la main", dans la région parallèle chaque thread parcourira `N / P` indices consécutifs pour calculer son `pi_local`. Une fois que c'est calculé, les threads additioneront les uns après les autres leur `pi_local` dans `pi` (il faudrait utiliser une région `critical` ou `atomic` afin d'éviter le problème de concurrence en cet étape). Tester la performance en utilisant de differents nombres de threads.

Part 3

La conjecture de Goldbach

Le but de cet exercice est de démontrer l'intérêt de l'utilisation de la clause `schedule` de l'OpenMP afin d'améliorer les performances d'un code parallélisé.

Dans le monde de la théorie des nombres, d'après la conjecture de Goldbach: chaque nombre pair supérieur à deux est la somme de deux nombres premiers.

Dans cet exercice, vous allez écrire un code pour tester cette conjecture. Le code permet de trouver le nombre de paires de Goldbach pour un nombre pair donné i (c'est à dire le nombre de paires de nombres premiers P_1, P_2 tels

que $P_1 + P_2 = i$) pour $i = 2 \dots 8000$. Le coût de calcul est proportionnel à $i^{2.5}$, donc pour obtenir une performance optimale avec plusieurs threads la charge de travail doit être distribuée en utilisant intelligemment la clause `schedule` de l'OpenMP.

Exercise 3

a) Ecrire la fontion `estPrime` qui teste la primalité d'un nombre donné.

b) Ecrire une fonction `goldbach` qui permet de trouver le nombre de paires de nombre premiers pour un nombre pair donné.

c) Pour chaque valeur de $i = 1 \dots 8000$, remplir un tableau `numPairs[i]` en utilisant la fonction `goldbach` Le tableau *numpairs* contiendra alors le nombre de paires de Goldbach pour chaque nombre pair jusqu'à 8000.

d) Une fois le tableau `numpairs` a été rempli avec le nombre de paires de Goldbach, tester que la conjecture de Goldbach est vraie, c'est à dire que pour chaque nombre pair supérieur à 2 il existe au moins une paire de Goldbach.

Le résultat obtenu doit être en adéquation avec l'exemple suivant:

Nombre pair	Nombre de paires de Goldbach
2	0
802	16
1602	53
2402	37
3202	40
4002	106
4802	64
5602	64
6402	156
7202	78

On vous demande par la suite de:

Exercise 4

a) Paralléliser le code séquentiel en utilisant OpenMP (ajouter les directives nécessaires autour de la boucle appelant la fonction `goldbach`) sans préciser d'ordonnancemment.

b) Vérifier que votre code trouve la bonne solution en utilisant plusieurs threads.

c) Afin d'accélérer la vitesse de calcul, utiliser un ordonnancement d'abord statique `schedule(static)` puis dynamique `schedule(dynamic)`, en précisant la taille du bloc. Récapitulez tous les résultats dans un tableau et analysez-les. Pour mieux comprendre l'effet des différentes méthodes d'ordonnancement vous pourriez afficher le thread traitant chaque itération.

d) Utiliser un ordonnancement guidé `schedule(guided)`, analyser l'effet de cet ordonnancement sur les performances de votre programme.

Part 4

Produit matrice-vecteur en OpenMP

Le but de cet exercice est d'écrire un programme qui calcule le produit d'une matrice A de taille $N \times N$ et d'un vecteur x de taille N:

$$Ax = b.$$

Utiliser le code squelette fourni dans le fichier `matvec-squelette.cpp` qui alloue les vecteurs x, b et la matrice A orientée par les lignes (c'est à dire, les premières N éléments correspondent à la première ligne $A(0,:)$, puis la deuxième ligne $A(1,:)$, etc.). La matrice A et le vecteur x sont initialisés par le code squelette.

Exercise 5

a) Coder la version séquentielle dans le créneau indiqué dans le code, puis mesurer le temps d'exécution.

b) Implanter la version parallelisée avec `omp for` et mesurer le temps d'exécution.

c) Réaliser une autre version basée sur OpenMP Tasks tel que chaque produit scalaire $b(i) = A(i,:)x(:)$ est effectué par un tache.

d) Afficher l'accéleration et l'efficacité.

e) Tester la performance pour toutes les tailles de dimension entre $2^0, \ldots, 2^{12}$. A partir de quelle taille constatez-vous un gain de performance? Modifier la version parallèle tel qu'elle performe mieux pour les petites tailles de dimension. Pour ce faire, utiliser la clause d'OpenMP correspondante ou faire un branchement explicit (qui execute la version séquentielle ou parallèle en fonction de la taille de dimension).