Programmation Parallèle
TP - Introduction à OpenMP

Oguz Kaya
oguz.kaya@lri.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

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 3

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.

Part 4

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. Un code squelette `goldbach.cpp` est fourni pour travailler dessus.

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 êtes fournis un code séquentiel qui teste 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 $i = 4, \ldots, 8192$. Le coût de calcul est proportionnel à i^2 dans l'itération i, 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.

On vous demande par la suite de:

Exercice 4

a) Paralléliser le code séquentiel en utilisant `omp for` sans préciser d'ordonnancemment. Quelle est la stratégie d'ordonnancement de l'OpenMP par défaut (et la taille de bloc impliqée par ceci)?

b) Réaliser la parallélisation avec l'ordonnancement `static` et la taille de bloc 256.

c) Réaliser la parallélisation avec l'ordonnancement `dynamic`.

d) Réaliser la parallélisation avec l'ordonnancement `dynamic` et la taille de bloc 256.

e) Réaliser la parallélisation avec l'ordonnancement `guided`.

f) Réaliser la parallélisation avec l'ordonnancement `guided` et la taille de bloc 256.

g) Dans quelles stratégies avez-vous obtenu la meilleure performance? A-t-il du sens compte tenu de la compléxite du calcul?

Part 5

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:

$$b = Ax.$$

Utiliser le code squelette fourni dans le fichier `matvec.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.

Exercice 5

a) Coder la version séquentielle dans le champs indiqué du 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 une tâche.

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).

Part 6

Calcul de Fibonacci avec OpenMP Tasks

Dans cet exercice, nous allons paralleliser le calcul du nombre Fibonnacci[N] à l'aide de l'OpenMP Tasks.

Exercice 6

a) Réaliser une version récursive qui parallélise ce calcul avec `#pragma omp task` et `#pragma omp taskwait`

b) Réaliser une version itérative qui calcule tous les Fibonacci[i] dans un tableau de taille N et parallélise ce calcul avec `#pragma omp task` et la clause `depend(in:...)` et `depend(out:...)`

Part 7

Mergesort en parallèle avec OpenMP Tasks

Paralléliser le code séquentiel donné pour le mergesort en utilisant OpenMP Tasks. Limiter la taille minimum de tâches créées à une constant K (e.g., 100000) tel que la récursion ne génère plus de tâches à partir de cette taille-là. Experimenter avec ce parametre pour trouver la meilleur accéleration.

Part 8

Prefixe-somme en 2D avec OpenMP Tasks

Nous voulons calculer la prefixe-somme d'un tableau 2D $A[N][N]$ tel que $B[x][y] = \sum_{(i,j)=(0,0)}^{(x,y)} A[i][j]$. Nous allons utiliser OpenMP Tasks pour effectuer ce calcul avec une taille de tuile $B \times B$ (B divise N). Calcul de prefixe-somme de chaque tuile $B \times B$ sera effectué par une tâche. Il faudra liée les tâches avec des dépendences appropriées pour que le calcul soit correct. Une fois la parallélisation correcte, experiementer avec de différentes valeurs de B pour trouver la meilleur granularité.

Part 9

Produit matrice-matrice Strassen avec OpenMP Tasks

Nous voulons implanter l'algorithme de Strassen pour des matrices $C = AB$ de taille $N \times N$ chacune. L'avantage de cet algorithme est sa complexité supèrieure de $O(N^{2.8074})$ (au lieu de $O(N^3)$).

Exercice 7

a) Réaliser un code qui alloue initalise les matrices A, B, C de taille $N \times N$.

b) Implanter une version classique de la multiplication des matrices, puis paralléliser avec OpenMP For.

c) Implanter des fonctions pour multiplication et addition des matrices par blocs qui multipli les sous-matrices des deux matrices (A et B, par exemple).

d) Utiliser ses fonctions en suite pour calculer les sous matrices $M_1, \ldots, M_7$, puis $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$.

e) Paralléliser le code avec une utilisation correcte de l'OpenMP Tasks et tester avec de différentes taille minimum de bloc (pour terminer la récursion).

Part 10

Produit matrice-matrice efficace avec tuiles et vectorisation

Nous allons réaliser la multiplication de deux matrices A et B de taille $N \times N$

$$C = AB$$

, ce qui donne $C(i,j) = \sum_{k=1}^{N} A(i,k)B(k,j) = \sum_{k=1}^{N} A(i,k)B^T(j,k)$. Pour faciliter la programmation, nous allons stocker et utiliser B^T au lieu de B.

Exercice 8

a) Implanter une version naïve de la multiplication $C = AB$. Mesurer le temps d'exécution pour de différentes taille de dimension N, notamment pour les puissances de 2.

b) Fixer une taille de tuile B, puis implanter un algorithme qui multiplie les matrices par tuiles. Pour une taille de dimension assez grande (e.g. 4096), expérimenter avec de différentes valeurs de B afin de trouver la valeur optimale.

c) Pour une taille de tuile fixe (e.g. $B = 32$), paralléliser la multiplication des deux tuiles à l'aide de l'AVX. Penser notamment à profiter des instructions de type FMA. Penser également à cacher la latence avec ILP (instruction-level parallelism). Que donne-t-elle cette optimisation au niveau du temps d'exécution par rapport à la version précédente? Afficher également la performance en gigaflops/s (pour la multiplicatoin des deux matrices de taille $N \times N$, nous effectuons $2N(N-1)$ opérations.) et le rapport de performance par rapport à la performance crête d'un cœur du CPU.

d) Paralléliser le code avec OpenMP for et/ou tasks.