Parallélisme à base de thread

(PBT 2022-2023)

TD3

Programmation par tâches

OpenMP

romain.pereira@cea.fr
marc.perache@cea.fr
hugo.taboada@cea.fr

Les objectifs de ce TD sont :
— Mise en pratique du modèle de programmation par tâches,
— Manipulation avec OpenMP.

I Rendu

Ce TD est noté. Le code source ainsi qu'un rapport au format `.pdf` répondant aux questions, et **détaillant l'implémentation**, est à envoyer sur le lien suivant, avant le 10/03/2023 à 23h59.

https://e.pcloud.com/#page=puplink&code=NewZSPwkQQ3zIuQBbqIQ0fegTV9Dd4yX

II Parcours en profondeur d'un arbre

Le fichier `Tree-traversal/tree-traverse.c` initialise un arbre binaire aléatoire. Le but de cet exercice est de calculer la profondeur de l'arbre à l'aide d'un parcours en profondeur.

Q.1: Remplissez la fonction `int inorderTraverse_Task(node_t *, int)` qui permet de calculer la profondeur d'un arbre en séquentiel.

Q.2: Paralléliser la fonction précédente en utilisant les tâches OpenMP

Q.3: A l'aide d'une clause sur le constructeur de tâche explicite, limitez la génération de tâches pour s'arrêter à une profondeur maximale de 5 (par exemple).

Q.4: Si vous en avez mis, retirez tous les points de synchronisation type `taskwait` et utilisez la directive `taskgroup` pour synchroniser implicitement vos tâches et leurs descendantes.

III Calcul de π par Monte Carlo

1 Définition de la méthode

Le calcul de π est un problème fréquemment étudié pour l'apprentissage du parallélisme. L'une des méthodes les plus courantes est la méthode approchée de

type *Monte Carlo*. On construit un carré et son cercle inscrit, de rayon r et de centre O. Le rapport de l'aire des 2 figures est $\alpha = \frac{\pi r^2}{(2r)^2} = \frac{1}{4}\pi$.

La méthode de Monte Carlo consiste à lancer des fléchettes manière aléatoire en direction de la cible (cf Figure 1). Lorsque le nombre de lancer tend vers l'infini, le rapport $\frac{\text{nombre de fléchettes dans le cercle}}{\text{nombre de fléchettes dans le carré}}$ tend vers α.

En pratique, on peut effectuer un nombre élévé de lancers, compter les fléchettes, estimer le rapport α, et enfin calculer $\pi \simeq 4\alpha$.

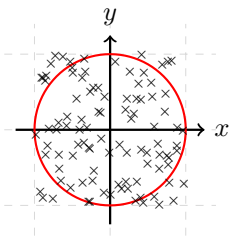

FIGURE 1 – Distribution aléatoire des fléchettes.

Pour simplifier les calculs, on peut poser $r = 1$ et définir O l'origine. Soit un point P de coordonnées $(x, y) \in [-1, 1]^2$ alors P est dans le cercle si et seulement si $x^2 + y^2 < 1$.

Q.5: À l'aide de la description du problème dans la section précédente, et du programme séquentiel présent dans le fichier `Pi_MT/pi_seq.c`, écrire un programme parallèle à base de tâches OpenMP 3.0 où chaque thread crée N tâches exécutant M lancers.

Vous pourrez également rendre M aléatoire par tâche afin d'observer l'impact du déséquilibre de charge sur le temps d'exécution.

IV Produit matrice creuse / vecteur (SpMV)

Une matrice creuse est une matrice contenant un nombre important de valeurs nulles. Pour limiter l'empreinte mémoire liée au stockage de la matrice, seules les valeurs non nulles sont stockées. Différents formats existent pour arriver à ce résultat. Dans le cadre de cet question, nous nous focalisons sur le format *Compressed Sparse Row* (CSR). Considérons une matrice carré, creuse, de taille $n \in \mathbb{N}$, elle est alors stockée sous forme de trois vecteurs :

- values - stocke l'ensemble des valeurs non nulles ligne par ligne de la matrice, de taille `NNZ`.
- JA - un tableau d'entiers qui enregistre l'indice de colonne de chaque valeur, il est de taille `NNZ`.
- IA - un tableau d'entiers qui contient les pointeurs vers le début de chaque ligne dans les tableaux `values` et `JA`. La taille de ce vecteur est de $N + 1$ éléments.

La structure d'une matrice *CSR* est la suivante

```
typedef struct
{
    int m_nrows, m_nnz ;
    double* m_values ;
    int* m_ja ;
    int* m_ia ;
} CSRMatrix_t;
```

Le code présent dans le répertoire `SpMV` contient les fonctions d'allocation et d'initialisation d'une matrice creuse au format `CSR`. Complétez les fichiers correspondants pour répondre aux questions.

Q.6: Le fichier `CSRMatrix.c` contient le squelette de la fonction `mult_CSR(CSR_Matrix_t*, double const*, double)` qui effectue le produit entre une matrice creuse et un vecteur. Remplissez cette fonction avec l'opération séquentielle correspondante.

Q.7: Proposez une parallélisation par tâches de l'algorithme précédent, en utilisant les tâches OpenMP, et en faisant en sorte que chaque tâche effectue un SpMV sur un sous-ensemble de lignes de la matrice.

Q.8: Quels avantages apportent la parallélisation par tâches d'un tel algorithme ?