TP3_OpenMP

Night

February 2023

1 Introduction

2 Parcours en profondeur d'un arbre

2.1

On définit la fonction de manière récursive , en ajoutant 1 à la taille des arbres issus des feuilles gauches et droite si elles existent.

```
int inorderTraverse_Task(node_t * node, int depth)
{
    // TODO : remplir ici
    if (node == NULL) {
        return depth-1;
    }

    int leftDepth = inorderTraverse_Task(node->left, depth + 1);
    int rightDepth = inorderTraverse_Task(node->right, depth + 1);
    return (leftDepth > rightDepth) ? leftDepth : rightDepth;
}
```

On constate que la taille de l'arbre aléatoire est correcte.

```
[night@night-20b7s2ex01]   [~/S4/OpenMP/ETUDIANT/CODE/Tree-traversal]
      5 fichiers, 52Kb) $   ./tree-traverse.exe 12

          Total Nodes: 12
          Time Taken (Task Construct): 0.000761 sec
          Depth: 6 (expected 6)

  Inorder tree traversal using Task Construct ................Done
```

2.2

On utilise la directive pragma omp task pour déclarer les tâches leftDepth et rightDepth. Ces tâches peuvent être exécutées de manière indépendante, on définit donc la directive firsprivate(node).

```
int inorderTraverse_Task(node_t* node, int depth) {

    // TODO : remplir ici
    if (node == NULL) {
        return depth-1;
    }

    int leftDepth, rightDepth;
```

```
    #pragma omp parallel
    #pragma omp single firstprivate(node) nowait
    {
        #pragma omp task firstprivate(node)
        {
            leftDepth = inorderTraverse_Task(node->left, depth + 1);
        }

        #pragma omp task firstprivate(node)
        {
            rightDepth = inorderTraverse_Task(node->right, depth + 1);
        }

        #pragma omp taskwait
    }

    return (leftDepth > rightDepth) ? leftDepth : rightDepth;
}
```

2.3

Avant de créer chaque tâche, nous vérifions si la profondeur actuelle est inférieure ou égale à 5. Si c'est le cas, nous créons deux nouvelles tâches pour le traitement des fils gauche et droit et attendons leur achèvement avec taskwait().

```
int inorderTraverse_Task(node_t* node, int depth) {
    if (node == NULL) {
        return depth-1;
    }

    int leftDepth = 0, rightDepth = 0;

    #pragma omp parallel
    #pragma omp single nowait
    {
        if (depth <= 5) {
            #pragma omp task shared(leftDepth)
            {
                leftDepth = inorderTraverse_Task(node->left, depth + 1);
            }

            #pragma omp task shared(rightDepth)
            {
                rightDepth = inorderTraverse_Task(node->right, depth + 1);
            }
        }

        else {
            return 0;
        }
        #pragma omp taskwait
    }

    return (leftDepth > rightDepth) ? leftDepth : rightDepth;
}
```

2.4

On synchronise donc implicitement les tâches avec la directive taskgroup. (J'ai repris le code de la fonction2)

```
#pragma omp parallel
#pragma omp single firstprivate(node) nowait
#pragma omp taskgroup
{

}
```

3 Calcul de πpar Monte Carlo

On définit les macros avec un nombre de tâches 10 fois plus petit pour avoir un programme suffisamment rapide. Ensuite, on rend parallèle la boucle principale,avec une directive single pour éviter que tous les threads déclarent les mêmes tâches. On transforme aussi le count en créeant des variables intermédiaires const, et on ajoute une directive taskwait pour améliorer les performances de calcul de cette somme.

3.1 q5

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <inttypes.h>

#include <omp.h>

#define N_TASKS_PER_THREAD 1
#define N_TRIALS_PER_TASK 1E7

int main(int argc, char ** argv) {
  uint64_t count = 0;
  uint64_t nb_tests;

  srand(2023);

  const double start = omp_get_wtime();
  #pragma omp parallel shared(count,nb_tests)
  {
    #pragma omp single
    {
      nb_tests = N_TASKS_PER_THREAD * N_TRIALS_PER_TASK * omp_get_num_threads();
    }

    uint64_t local_count = 0;

    uint64_t local_task_counts[N_TASKS_PER_THREAD] = { 0 };
    for (int i = 0; i < N_TASKS_PER_THREAD; i++) {
      #pragma omp task firstprivate(i) shared(local_task_counts)
      {
        for (int k = 0; k < N_TRIALS_PER_TASK; k++ ) {
          const double x = rand() / (double)RAND_MAX;
          const double y = rand() / (double)RAND_MAX;
```

```
          local_task_counts[i] += (((x * x) + (y * y)) <= 1);
        }
      }
    }
    #pragma omp taskwait

    for (int i = 0; i < N_TASKS_PER_THREAD; i++) {
      local_count += local_task_counts[i];
    }

    #pragma omp atomic
    count += local_count;
  }
  const double end = omp_get_wtime();

  fprintf(stdout, "%llu of %llu throws are in the circle !\n", count, nb_tests);
  const double pi = ((double)count * 4) / (double)nb_tests;
  fprintf(stdout, "Pi ~= %lf\n", pi);
  fprintf(stdout, "Execution time: %fs\n", end - start);

  return 0;
}
```

```
      [night@night-20b7s2ex01]   [~/S4/OpenMP/td3/CODE/question5]
            5 fichiers, 44Kb)  $   ./pi_sequentiel.exe
7852024 of 10000000 throws are in the circle !
Pi ~= 3.140810
Execution time: 0.377660 seconds
      [night@night-20b7s2ex01]   [~/S4/OpenMP/td3/CODE/question5]
            5 fichiers, 44Kb)  $   ./pi_parallele.exe
31414580 of 40000000 throws are in the circle !
Pi ~= 3.141458
Execution time: 13.642840s
```

On constate que pour autant de cibles dans le cercle inscrit, la version parallèle est environ **50 fois plus lente** mais on gagne une **précision d'1 à 2 décimales sur pi**.

4 Produit matrice creuse / vecteur (SpMV)

4.1 q6

On implémente la fonction:

```
mult_CSR(CSR_Matrix_t*,
double const*, double)
```

, en créant les références définies dans la structure de A et en définissant les constantes intermédiaires afin d'optimiser les Mflops estimés.

```
void mult_CSR(CSRMatrix_t* A, double const* x, double* y)
{
  /* TODO : Kernel    compl ter */
  const int N = A->m_nrows;
  const double *m_values = A->m_values;
  const uint64_t *m_ja = A->m_ja;
  const uint64_t *m_ia = A->m_ia;

  for (int i = 0; i < N; i++)
```

```
  {
    const uint64_t M = m_ia[i];
    const uint64_t M_end = m_ia[i + 1];
    double value = 0;

    for (uint64_t k = M; k < M_end; k++)
      value += m_values[k] * x[m_ja[k]];

    y[i] = value;
  }
}
```

```
      [night@night-20b7s2ex01]   [~/S4/OpenMP/td3/CODE/question6]
             7 fichiers, 64Kb)  $   ./spmv.exe
NX: 100 NY: 100
NTest: 1
NROWS     : 10000
NNZ       : 49600

NORME Y= 65508.85
 Time : 0.000073
 MFlops : 1350.17
AvgTime : 0.000073
```

4.2 q7

Pour paralléliser l'algorithme SpMV en utilisant cette stratégie, on peut diviser la matrice en blocs de taille BxB, où B est une taille de bloc fixe. On peut alors utiliser des directives OpenMP pour paralléliser le calcul des blocs.

Chaque ligne du calcul étant indépendante, on affecte donc une tâche à chaque ligne, en faisant bien attention au partage des variables privées/publiques/ avec la tâche firsprivate.

On applique rend donc la boucle principale parallèle, avec une directive single pour éviter que tous les threads déclarent les mêmes tâches.

```
void mult_CSR_task(CSRMatrix_t* A, double const* x, double* y)
{
  #pragma omp parallel shared(A,x,y)
  #pragma omp single
  {
    const int N = A->m_nrows;
    const double *m_values = A->m_values;
    const uint64_t *m_ja = A->m_ja;
    const uint64_t *m_ia = A->m_ia;

    for (int i = 0; i < N; i++){
      const int M = m_ia[i];
      const int M_end = m_ia[i + 1];
      #pragma omp task firstprivate(i,M,M_end) shared(m_values,m_ja,x,y)
      {
        double value = 0;
        for (uint64_t k = M; k < M_end; k++){
          value += m_values[k] * x[m_ja[k]];
        }
        y[i] = value;
      }
    }
```

```
  }
}
```

```
      [night@night-20b7s2ex01]   [~/S4/OpenMP/td3/CODE/question7]
             7 fichiers, 68Kb) $   ls
CSRMatrix.c  CSRMatrix.h  CSRMatrix.o  Makefile  main.c  main.o  spmv.exe
      [night@night-20b7s2ex01]   [~/S4/OpenMP/td3/CODE/question7]
             7 fichiers, 68Kb) $   ./spmv.exe
NX: 100 NY: 100
NTest: 1
NROWS     : 10000
NNZ       : 49600

NORME Y= 65508.85
 Time : 0.009582
 MFlops : 10.35
AvgTime : 0.009582
```

(Il doit il y avoir une erreur car l'estimation des Mflops est trop faible)

4.3 q8

Les intérêts de paralléliser un algorithme utilisant SpMV pour cacluler le produit Matrice vecteur sont:

- Accélération des temps de traitement (redondances des tâches (surtout si l'on travaille dans un corps fini?), bibliothèque) optimisée)
- Utilisation efficace des ressources (pas de goulot d'étranglement, fiabilité)