TD N°3 : **Programmation parallèle avec OpenMP**

Exercice 1

Soit le programme suivant :

```
...

for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0)
            # pragma omp parallel for num threads (thread count) private (i)
                for (i = 1; i < n; i++)
                    /*Work 1*/
        Else
            # pragma omp parallel for num threads (thread count) private (i)
                for (i = 1; i < n−1; i ++)
                    /*Work 2*/

...
```

1. Quel est l'inconvénient en termes de performance que présente ce programme ?
2. Modifier le programme afin de pallier cet inconvénient.

Exercice 2

Soit le programme suivant :

```
...
#pragma omp parallel for
for (int i = 0; i < n; i++)
      c[i] = a[i] + b[i];
...
```

1. En supposant *n* égal à 64 et le nombre de threads égal à 8, quelle est la tâche élémentaire de chaque thread et combien de tâches élémentaires sont exécutées par thread.
2. En gardant *n* égal à 64 et le nombre de threads à 8, modifiez le code pour augmenter la quantité de travail par tâche élémentaire de thread et réduire le nombre d'exécutions de celle-ci par thread à 2.
3. Proposez un programme utilisant "#pragma omp sections" qui donne la même quantité de travail à la tâche élémentaire et le même nombre d'exécutions par thread que dans la question 2.

Exercice 3

Soit le programme séquentiel suivant :

```
#include <stdio.h>
#define NUMBER 100000

int main () {
  int i, j, val, total = 0 , N = NUMBER;
  for (i = 2 ; i <= N ; i++)
  {
    val = 1;
    for (j = 2 ; j < i ; j++)
    {
      if (i % j == 0)
      {
        val = 0;
        break;
      }
    }
    total = total + val;
  }
  printf("Done. total = %d\n", total);
  return (0);}
```

1. Analyser le programme et déterminer la tâche réalisée par ce dernier.
2. Proposer une solution parallèle à ce programme afin d'améliorer sa performance.
3. Implémenter la solution proposée en donnant le code correspondant utilisant les directives et fonctions OpenMP pour un nombre de threads égal à 4.
4. Compiler (gcc -fopenmp) le code ci-après et l'exécuter en affichant les temps d'exécution de la version séquentielle et la version parallèle (reprendre le code et le compléter par la solution parallèle proposée, dans la partie "Traitement parallèle" et utiliser l'exemple de calcul du temps d'exécution employé dans la partie "Traitement séquentiel" du code).

```
#include <stdio.h>
#include <omp.h>
#define NUMBER 100000

int main () {
  int i, j, val, total = 0 , N = NUMBER;
  double start, end, time;

  //Traitement séquentiel
  start = omp_get_wtime();
  for (i = 2 ; i <= N ; i++)
  {
    val = 1;
    for (j = 2 ; j < i ; j++)
    {
      if (i % j == 0)
      {
        val = 0;
        break;
      }
    }
    total = total + val;
  }
  end = omp_get_wtime();
  time = end-start;
  printf("Done. total = %d\n", total);
  printf("Sequential time %f seconds\n", time);

  //Traitement parallèl {...}

  return (0);}
```

5. Comparer la solution parallèle à la solution séquentielle en déterminant l'accélération obtenue.
6. Changer le nombre de threads de 4 à 8. Recompiler et ré-exécuter le programme. Comparer cette exécution à la solution séquentielle.

Exercice 4

1. Soit le programme séquentiel ci-dessous réalisant le produit scalaire de deux vecteurs *a* et *b*. Proposer deux versions parallèles de ce programme en utilisant pour chaque version une des directive OpenMP suivantes :

- #pragma omp task ;
- #pragma omp for.

```
#include <stdio.h>
#define SIZE 256

int main ()
{
 double sum , a[SIZE], b[SIZE];
 sum = 0.;

 for (int i = 0; i < SIZE; i++)
 {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
    sum = sum + a[i]*b[i];
 }

 printf("sum = %g\n", sum);
 return 0; }
```

2. Le programme séquentiel ci-après permet la multiplication de deux matrices *a* et *b*. Proposer une solution parallèle pour la multiplication des matrices *a* et *b* permettant à tous les threads créés de collaborer pour produire chaque élément de la matrice *c*.

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main() {
const int DIM = 1000;
int i,j,k;
double **a, **b, **c;

a= (double**) malloc(DIM*sizeof(double*));
b= (double**) malloc(DIM*sizeof(double*));
c= (double**) malloc(DIM*sizeof(double*));

/* Initialisation */
for (i=0; i<DIM; i++) {
  a[i]=(double*) malloc(DIM*sizeof(double));
  b[i]=(double*) malloc(DIM*sizeof(double));
  c[i]=(double*) malloc(DIM*sizeof(double));

  for (j = 0; j < DIM; j++) {
    a[i][j] = (double)(i-j);
    b[i][j] = (double)(i+j);
    c[i][j] = 0.0;
  }
}

for (int i = 0; i < DIM; i++)
   for (int j = 0; j < DIM; j++)
     for (int k = 0; k < DIM; k++)
       c[i][j]+= a[i][k] * b[k][j];

return 0;
}
```