TP n° 2 : Parallélisation de programmes avec OpenMP – partie 2

Terminez tout d'abord les exercices du TP 1 (sauf le dernier que vous pourrez reprendre une fois ce TP terminé) si ce n'est pas encore le cas. Dans ce nouveau TP, parallélisez tous les codes en utilisant OpenMP. Des modifications seront parfois nécessaires en plus de l'ajout de directives OpenMP. Détaillez le cas échéant vos modifications de code, vos choix de partage de travail et de statut des données pour avoir des résultats corrects. Discutez dans un second temps les problèmes de performance.

1 Produit matrice-matrice

Étudiez et parallélisez le code suivant [Lien] :

```
void matmat_kernel(double C[N][N], double A[N][N], double B[N][N]) {
  size_t i, j, k;

  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      C[i][j] = 0.;
      for (k = 0; k < N; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Faites des mesures et une courbe du temps d'exécution en fonction du nombre de threads par stratégie : (1) code original, (2) code parallélisé avec ordonnancement statique et (3) code parallélisé avec ordonnancement dynamique. Discutez les résultats.

1.1 Calcul de Pi

Étudiez et parallélisez le code suivant [Lien] :

```
void pi_kernel(size_t nb_steps, double* pi) {
  double term;
  double sum = 0.;
  double step = 1./(double)nb_steps;

  for (size_t i = 0; i < nb_steps; i++) {
    term = (i + 0.5) * step;
    sum += 4. / (1. + term * term);
  }

  *pi = step * sum;
}
```

1.2 Tri par énumération

Étudiez et parallélisez le code suivant [Lien] :

```
void enumeration_sort_kernel(double tab[N]) {
  size_t i, j;
  size_t* position = malloc(N * sizeof(size_t));
  double* copy     = malloc(N * sizeof(size_t));

  for (i = 0; i < N; i++) {
    position[i] = 0;
    copy[i] = tab[i];
  }

  for (j = 0; j < N; j++) {
    for (i = 0; i < N; i++) {
      if ((tab[j] < tab[i]) || ((tab[i] == tab[j]) && (i < j))) {
        position[i]++;
      }
    }
  }

  for (i = 0; i < N; i++)
    tab[position[i]] = copy[i];

  free(position);
  free(copy);
}
```

1.3 Tri à bulle

Étudiez les dépendances dans le code du tri à bulle ci-après. Celles-ci devraient interdire toute parallélisation, mais un changement d'algorithme permet de réaliser une version parallèle relativement proche. Tentez de trouver vous-même des modifications permettant de paralléliser ce code, et si vous n'avez pas l'intuition, utilisez le mot clé « tri pair-impair » (*odd-even sort*) sur un moteur de recherche pour vous guider, puis implémentez-en une version parallèle [Lien] :

```
void bubble_sort_kernel(double tab[N]) {
  size_t i, j;
  double temp;

  for (i = 0; i < N ; i++) {
    for (j = 0; j < N - i - 1; j++) {
      if (tab[j] > tab[j + 1]) {
        temp = tab[j + 1];
        tab[j + 1] = tab[j];
        tab[j] = temp;
      }
    }
  }
}
```