TP n° 1 : Parallélisation de programmes avec OpenMP – partie 1

Chaque code est accessible via un lien sur le sujet PDF ainsi qu'une archive disponible sous Moodle. **Ne copiez jamais les codes depuis le PDF** : des caractères spéciaux pourraient être copiés et empêcher la compilation.

1 Échauffement

1.1 Régions parallèles OpenMP

Soit le code suivant [Lien] :

```c
#include <stdio.h>
int main() {
  #pragma omp parallel
  printf("Hello\n");
  printf("World\n");
  return 0;
}
```

Compilez ce programme sans l'option `-fopenmp` et lancez-le. Qu'observez-vous ?
Un seul Hello, et un seul World.

Compilez ce programme avec l'option `-fopenmp`, et lancez-le. Quel est le nombre de threads par défaut ?

Modifiez le nombre de threads dynamiquement en changeant la valeur de la variable d'environnement `OMP_NUM_THREADS` (les variables d'environnement OpenMP n'ont pas été vues en cours, mais celle-là est bien pratique). Manipulation ?

Modifiez le nombre de threads statiquement à l'aide de la clause **`num_threads`**`(n)` de la directive **`parallel`**. Manipulation ?

Est-ce qu'il peut arriver que le mot « `World` » soit affiché avant le mot « `Hello` » ? Si oui, dans quel cas ? Si non, pourquoi ?

Modifiez le programme pour que chaque thread exécute les deux `printf`.

1.2 Partage de travail OpenMP

Soit le code suivant [Lien] :

```
#include <stdio.h>
#include <omp.h>
#define SIZE  100
#define CHUNK  10

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

  for (i = 0; i < SIZE; i++)
    a[i] = b[i] = i;

  #pragma omp parallel private(tid) reduction(+: sum)
  {
    tid = omp_get_thread_num();
    if (tid == 0)
      printf("Nb threads = %d\n", omp_get_num_threads());
    printf("Thread %d: starting...\n", tid);

    #pragma omp for
    for (i = 0; i < SIZE; i++) {
      c[i] = a[i] + b[i];
      sum += c[i];
      printf("Thread %d: c[%2d] = %g\n", tid, i, c[i]);
    }
  }
  printf("sum = %g\n", sum);
  return 0;
}
```

Que fait ce programme ?

Le premier thread affiche le nombre de threads.
Il additionne les éléments de deux listes et les mets dans une troisième. Il fait aussi la somme de la troisième liste.

Quelles sont les instructions exécutées par un seul thread ? Par tous les threads ?

Quel est le rôle de la directive `for` d'OpenMP ?

Quel est le statut des variables `i` et `tid` ? Quel est le statut des variables `a`, `b` et `c` ?

Quel est le rôle de la clause **`reduction`** ?

Spécifie qu'une variable privée à chaque thread fera l'objet d'une opération à la fin de leur exécution.

2 Parallélisation de codes avec OpenMP

Parallélisez les codes suivants en utilisant OpenMP. Des modifications des codes 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. Les codes disposent d'un mécanisme de vérification du résultat. Si vous obtenez le message « *Bad results* », votre parallélisation est incorrecte : étudiez mieux le problème et corrigez votre parallélisation. Discutez dans un second temps les problèmes de performance.

2.1 Directives OpenMP

Dans cette partie, seul l'ajout de directives OpenMP est autorisé, sans modifier le code C.

2.1.1 Addition de vecteurs

Étudiez et parallélisez le code suivant [Lien] :

```
void addvec_kernel(double c[N], double a[N], double b[N]) {

  for (size_t i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
  }
}
```

2.2 Somme des éléments d'un vecteur

Étudiez et parallélisez le code suivant [Lien] :

```
void sum_kernel(double* psum, double a[N]) {
  double sum = 0.;

  for (size_t i = 0; i < N; i++) {
    sum += a[i];
  }
  *psum = sum;
}
```

2.3 Produit matrice-vecteur

Étudiez et parallélisez le code suivant [Lien] :

```
void matvec_kernel(double c[N], double A[N][N], double b[N]) {
  size_t i, j;

  for (i = 0; i < N; i++) {
    c[i] = 0.;
    for (j = 0; j < N; j++) {
      c[i] += A[i][j] * b[j];
    }
  }
}
```

3 Restructuration de programmes

Dans cette partie, parallélisez les codes suivants en ajoutant des directives OpenMP et en modifiant les codes si nécessaire. Les modifications apportées au code d'origine devront être aussi minimes que possible.

3.1 Stencil 1D

Étudiez et parallélisez le code suivant [Lien] :

```
void stencil1D_kernel(double a[N], double b[N]) {
  for (size_t i = 1; i < N; i++) {
    a[i] = (a[i] + a[i - 1]) / 2;
    b[i] = (b[i] + b[i - 1]) / 2;
  }
}
```

3.2 Réduction et réinitialisation

Étudiez et parallélisez le code suivant, dans le cas où il n'est pas acceptable d'avoir des erreurs de précision et dans celui où on peut les tolérer [Lien] :

```
void reduction_reinit_kernel(double A[N][N], double* sum) {
  *sum = 0.;

  for (size_t i = 0; i < N; i++) {
    for (size_t j = 0; j < N; j++) {
      *sum += A[i][j];
      A[i][j] = 0.;
    }
  }
}
```

3.3 Multiplication de polynômes

Étudiez et parallélisez le code suivant en tentant de paralléliser la boucle externe pour plus de performance. Notez que le problème est difficile et qu'il faut d'abord dessiner le graphe de dépendances sur quelques itérations pour trouver puis implémenter le parallélisme [Lien] :

```
void polynomial_multiply_kernel(double c[2*N-1], double a[N], double b[N]) {
  for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
      c[i+j] += a[i] * b[j];
}
```