Introduction à OpenMP

1 Prise en main des constructions d'OpenMP

Dans cette partie, on se propose de regarder l'effet des directives et clauses principales d'OpenMP au travers de petits exemples. Ces exemples sont repris d'une présentation de Cédric Bastoul.

1.1 Premier exemple

Soit le programme suivant :

```
#include <stdio.h>

int main() {
  #pragma omp parallel
  printf("Hello\n");
  printf("World\n");
  return 0;
}
```

- Compilez ce programme avec et sans l'option `-fopenmp`, et exéctuez-le dans les deux cas
- Quel est le nombre de threads par défaut ?
- Changez le nombre de threads utilisés pour l'exécution du programme

1.2 Variables private et schedule

Soit le code suivant :

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

#define SIZE 100
#define CHUNK 10

int main () {

  int tid = -1;
  double a[SIZE], b[SIZE], c[SIZE];

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

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

```
}
```

- Analysez le programme : quelles sont les instructions exécutées par tous les threads ? Par un seul thread ? Que devrait afficher le dernier `printf` ?
- Exécutez le programme plusieurs fois. Que penser de l'ordre d'exécution des instructions ?
- Redirigez la sortie de l'exécution sur la commande `sort`. Observez la répartition des itérations.
- Recommencez plusieurs fois ; la répartition est-elle stable ?
- Changez la politique d'ordonnancement par `static`. Exécutez plusieurs fois le programme ; la répartition est-elle stable ?
- Quel avantage y a-t-il à avoir une politique d'ordonnancement dynamique ?

1.3 Clause nowait

Soit le code suivant :

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

#define SIZE 100
#define CHUNK 10

int main () {

  double a[SIZE], b[SIZE], c[SIZE], d[SIZE];

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

  #pragma omp parallel
  {
    #pragma omp for schedule(static) nowait
    for (int i = 0; i < SIZE; i += 1) {
      c[i] = a[i] + b[i];
    }

    #pragma omp for schedule(static)
    for (int i = 0; i < SIZE; i++) {
      d[i] = a[i] + c[i];
    }
  }

  for (int i = 0; i < SIZE; i += 1) {
    printf("%g ", d[i]);
  }
  printf("\n");
  return 0;
}
```

- Exécutez le programme plusieurs fois. Les résultats sont-ils cohérents ?
- Analysez le programme : quelles itérations vont-elles être éxécutées par quels threads ?
- Après analyse, l'utilisation de la clause `nowait` vous semble-t-elle raisonnable ?
- Changez la politique d'ordonnancement de la seconde boucle à `guided`, puis exécutez le programme plusieurs fois. Les résultats vous semblent-ils cohérents ?

1.4 Clause private

Soit le code suivant :

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
  int val;
  srand(0);
```

```
  #pragma omp parallel private(val)
  {
    val = rand();
    sleep(1);
    printf("My val : %d\n", val);
  }

  return 0;
}
```

- Compilez et exécutez ce code avec et sans `private(val)`.
- Qu'observez-vous et pourquoi ?
- Est-il possible que deux threads affichent la même valeur même avec la clause `private` ?

1.5 Variables threadprivate

Soit le code suivant :

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

int tid, tprivate, rprivate;
#pragma omp threadprivate(tprivate)

int main() {
    // On interdit explicitement les threads dynamiques
    omp_set_dynamic(0);
    printf("Région parallèle 1\n");

    #pragma omp parallel private(tid, rprivate)
    {
        tid = omp_get_thread_num();
        tprivate = tid;
        rprivate = tid;
        printf("Thread %d : tprivate = %d - rprivate = %d\n", tid, tprivate, rprivate);
    }

    printf("Région parallèle 2\n");
    #pragma omp parallel private(tid, rprivate)
    {
        tid = omp_get_thread_num();
        printf("Thread %d : tprivate = %d - rprivate = %d\n", tid, tprivate, rprivate);
    }
    return 0;
}
```

- Analysez le code suivant, puis exécutez-le et compilez-le.
- Les résultats sont-ils ceux attendus ?

2 Application Histogram avec OpenMP

Dans cette section, on se propose d'implémenter différentes versions de la fonction principale de l'application Histogram. Toutes les versions doivent utiliser une parallélisation OpenMP, et être testées avec 1 et 2 threads au minimum (toutes doivent rendre un résultat correct sauf `calc_omp0`, et éventuellement `calc_omp_reduction` selon la version d'OpenMP installée sur vos machines).
Il faut d'abord commencer par récupérer l'archive suivante :
`https://www-soc.lip6.fr/~meunier/cours/hpc_tp_omp.tar.gz`
Cette archive contient un fichier `histogram.c` qui est une adaptation de la version du TP5, et un Makefile (reprenez l'image du TP5). Comme pour le dernier TP, la version séquentielle est donnée.

2.1 Version sans synchronisation

Écrivez la fonction `calc_omp0` qui effectue le calcul en parallèle avec OpenMP, mais sans synchronisation particulière. Cette fonction produira donc un résultat incorrect, mais permettra de mesurer le surcout lié à la création des threads

2.2 Version avec `__atomic_add_fetch`

Écrivez une fonction `calc_omp_atomic_inc` qui utilise la fonction builtin de gcc `__atomic_add_fetch()` à l'intérieur de ses itérations.

2.3 Version utilisant la directive `critical`

Écrivez une fonction `calc_omp_critical` qui utilise le pragma OpenMP `critical` pour effectuer les incrémentations.

2.4 Version utilisant la directive `atomic`

Écrivez une fonction `calc_omp_atomic` qui utilise le pragma OpenMP `atomic` pour effectuer les incrémentations.

2.5 Version utilisant la directive `reduction`

Écrivez une fonction `calc_omp_reduction` qui utilise le pragma OpenMP `reduction` sur les différents tableaux pour effectuer les incrémentations.
Note : la possibilité d'utiliser des variables de type tableau pour une réduction n'est possible qu'à partir de OpenMP 4.5 ; il est donc possible que vous ne puissiez pas tester cette version sur votre ordinateur, selon la version de gcc installée.

2.6 Version utilisant des tableaux intermédiaires privés et la directive `atomic`

Écrivez une fonction `calc_omp_private_atomic` qui déclare dans un bloc parallèle des variables de type tableau privées, effectue le calcul parallèle sur ces variables privées (réfléchir quant à la possibilité d'utiliser `nowait`), puis contient une boucle pour ajouter les valeurs des tableaux locaux aux tableaux globaux (en utilisant la directive `atomic`).

2.7 Synthèse des résultats

Que penser des résultats obtenus en terme de performance ? Comment mettre en relations ces résultats avec ceux des versions utilisant les pthreads ?