TP 2. Programmation multi-threads

Préparation

Récupérer l'archive compressée `TP2.tar.gz` et extraire les fichiers qui sont contenus dans cette archive :

```
cd <repertoire dans votre espace de travail>
cp /home/t/tajchman/AMSI03/2018-12-07/TP2.tar.gz .
tar xvfz TP2.tar.gz
```

Se placer dans le répertoire `TP2` :

```
cd TP2
```

et préparer les compilations dans les points suivants avec les commandes ci-dessous :

```
mkdir -p build
cd build
cmake ../src
cd ..
```

1 Code séquentiel

Le fichier `src/sinus_seq/sinus.cxx` calcule une approximation par série de Taylor de la fonction $x \mapsto \sin(x)$ pour un ensemble de valeur de $x \in [0, 2\pi]$. Les résultats sont sauvegardés dans un fichier.
Un script de commande unix `trace.sh` est fourni pour visualiser les résultats (graphes du sinus calculé par la machine et par la formule approchée, en utilisant `gnuplot`).

Question 1.

Se placer dans le répertoire TP2.

Compiler le code en tapant

```
make -C build sinus_seq
```

Exécuter le code en tapant

```
./build/sinus_seq/sinus_seq
```

Tracer le graphe des résultats et le visualiser en tapant

```
./trace.sh
```

Les courbes sur le graphe représentent les valeurs calculées par le sinus de la librairie standard et celles calculées par la formule de Taylor du programme.

2 Première version multi-threads avec OpenMP

On peut choisir a priori le nombre maximum de threads qui seront utilisés dans l'exécution d'un code, ce choix peut se faire de plusieurs façons.
Ici, l'utilisateur définira une variable d'environnement `OMP_NUM_THREADS` avec une valeur entière (entre 1 et le nombre de cœurs disponibles dans le processeur).
On peut aussi utiliser la fonction `omp_set_num_threads()` dans le code.

Question 2.

Examinez le fichier `src/sinus_openmp_1/sinus.cxx`, en particulier les lignes 47 à 52 et 86 à 91.

Compiler, exécuter le code en utilisant 3 threads et tracer les résultats en tapant

```
make -C build sinus_openmp_1
OMP_NUM_THREADS=3 ./build/sinus_openmp_1/sinus_openmp_1
./trace.sh
```

Relancer l'exécution et le tracé ci-dessus 3 ou 4 fois. Constater que les résultats sont (visuellement) similaires au cas séquentiel.

Question 3.

Comparer les temps de calcul entre

- la version séquentielle
  ```
  ./build/sinus_seq/sinus_seq 10000000
  ```
- la version multi-threads en utilisant 1 thread
  ```
  OMP_NUM_THREADS=1 \
    ./build/sinus_openmp_1/sinus_openmp_1 10000000
  ```
- la version multi-threads en utilisant 2 threads
  ```
  OMP_NUM_THREADS=2 \
    ./build/sinus_openmp_1/sinus_openmp_1 10000000
  ```
- la version multi-threads en utilisant 3 threads
  ```
  OMP_NUM_THREADS=3 \
    ./build/sinus_openmp_1/sinus_openmp_1 10000000
  ```
- la version multi-threads en utilisant 4 threads
  ```
  OMP_NUM_THREADS=4 \
    ./build/sinus_openmp_1/sinus_openmp_1 10000000
  ```

Interpréter les résultats.

3 Seconde version multi-threads avec OpenMP

La seconde version diffère de la première en essayant de diminuer le temps d'exécution d'une itération.

Question 4.

Comparer les fichiers source `src/sinus_openmp_1/sinus.cxx` et `src/sinus_openmp_2/sinus.cxx`.

Compiler l'exécutable

```
make -C build sinus_openmp_2
```

Exécutez (avec un argument égal à 1000) et tracez les résultats

```
OMP_NUM_THREADS=2 \
  ./build/sinus_openmp_2/sinus_openmp_2 1000 && ./trace.sh
```

Répéter la commande plusieurs fois et interpréter les graphes des résultats.

Suggérer et tester une possibilité d'amélioration.

4 Troisième version multi-threads avec OpenMP

Dans la troisième version, on parallélise une autre boucle (qui calcule la moyenne des valeurs)

Question 5.

Examinez le fichier source `src/sinus_openmp_3/sinus.cxx`.

Comparez-le avec les versions précédente.

Compilez l'exécutable `sinus_openmp_3` et refaire les tests parallèles.

5 Version multi-threads en utilisant les `std::threads`

Dans cette partie, il est utile d'avoir des notions de programmation objet C++.

> Dans le langage C++, une gestion des threads a été récemment introduite : les `std::threads`. Ceux-ci sont inspirés des `pthreads` (librairie "historique" de gestion des threads) mais adaptés au style de programmation objet C++.
>
> Par rapport à OpenMP, l'utilisateur doit définir "manuellement" plus de paramètres du calcul (par exemple, le découpage des boucles).
>
> Par contre, les std : :threads (et les autres fonctionnalités C++ liées au multi-threading) offrent la puissance de la programmation objet.

Le répertoire `src/std_threads` contient 2 versions.
Dans la première version (fichier src/std_threads/`sinus_v1.cxx`), les threads créés sont chargés d'exécuter des fonctions (au sens C).
Dans la seconde version (fichier src/std_threads/`sinus_v2.cxx`), les threads créés sont chargés d'exécuter des "objets fonctions".

Question 6.

> Compiler les codes et exécutez-les avec les commandes (le premier paramètre de l'exécution est le nombre de threads à utiliser, le second paramètre est la taille des vecteurs)
>
> ```
> make -C build sinus_std_threads sinus_std_threads_2
> ./build/sinus_std_threads/sinus_std_threads_v1 1 10000000
> ./build/sinus_std_threads/sinus_std_threads_v1 3 10000000
> ./build/sinus_std_threads/sinus_std_threads_v2 1 10000000
> ./build/sinus_std_threads/sinus_std_threads_v2 3 10000000
> ```
>
> Comparer les fichiers sources.

6 Parallélisation d'un (mini-)code avec OpenMP

On fournit un code C++ qui calcule une solution approchée de l'équation de la chateur $\partial u/\partial t = \nabla u + f$ sur un cube $[0,1]^3$ avec des conditions aux limites de Dirichlet.

Une version séquentielle du code se trouve dans `code/PoissonSeq` (comme version de référence) et une version de travail se trouve dans `code/PoissonOpenMP` à modifier suivant les indications ci-après.

Commandes

Se placer dans `code/PoissonSeq` et compiler le code séquentiel en mode Debug :

```
mkdir -p build
cd build
cmake -DCMAKE_BUILD_TYPE=Debug ../src
make
cd ..
```

Exécuter le code

```
./build/PoissonSeq n=300 m=300 p=300
```

Conserver une copie de l'affichage pour comparaison avec les versions OpenMP.

Une seconde version se trouve dans le répertoire `code/PoissonOpenMP` que vous modifierez.

Question 7.

1. Se placer dans `code/PoissonOpenMP` et compiler cette version comme précédemment.
2. Utiliser valgrind et kcachegrind pour repérer la partie du code qui prend le plus de temps calcul (comme dans le TP 1).
3. Ajouter ou modifier la(les) directive(s) OpenMP pour paralléliser cette partie si possible.
4. Recompiler le code. Il suffit de se placer dans `code/PoissonOpenMP` et de taper :
   ```
   make -C build
   ```
5. Exécuter plusieurs fois le code sur 1, 2, 3 et 4 threads.
6. Comparer les résultats et temps calcul affichés avec ceux de la version séquentielle. Si les résultats sont différents, c'est qu'il y a une erreur dans les intractions de parallélisation. Corriger et reprendre en 3.

Remarque

Quand on a parallélisé une partie du code ci-dessus, on refait une mesure avec `valgrind/kcachegrind` (ou un autre outil de profiling) pour voir si une autre partie du code pourrait être intéressante à paralléliser.