Modèles et techniques en programmation parallèle hybride et multi-cœurs

Travail pratique n°1

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

02/12/2020

Travail pratique n°1

A partir d'un code séquentiel qui calcule une solution approchée du problème suivant :

Chercher u*:* $(x,t) \mapsto u(x,t)$*, où* $x \in \Omega = [0,1]^3$ *et* $t \geq 0$*, qui vérifie :*

$$\begin{aligned} \frac{\partial u}{\partial t} &= \Delta u + f(x,t) & \\ u(x,0) &= g(x) & x \in \Omega \\ u(x,t) &= g(x) & x \in \partial\Omega, t > 0 \end{aligned}$$

où f *et* g *sont des fonctions données.*

Le code séquentiel utilise des différences finies pour approcher les dérivées partielles et découpe Ω en $n_1 \times n_2 \times n_3$ subdivisions.
Récupérer le fichier `TP1_sequentiel.tar.gz`

Structure du code séquentiel

Récupérer et décompresser un des fichiers `TP1_sequentiel.tar.gz` ou `TP1_sequentiel.zip`.

Se placer dans le répertoire `TP1/PoissonSeq` créé.
Le code séquentiel est réparti en plusieurs fichiers principaux dans le sous-répertoire `src`:

main.cxx:	programme principal: initialise, appelle le calcul des itérations en temps, affiche les résultats
scheme(.hxx/.cxx):	définit le type `Scheme` qui calcule une itération en temps
values(.hxx/.cxx):	définit le type `Values` qui contient les valeurs approchées à un instant donné
parameters(.hxx/.cxx):	définit le type `Parameters` qui rassemble les informations sur la géométrie et le calcul

Fonctions du type `Scheme` :

`Scheme(P, f)`	construit une variable de type `Scheme` en lui donnant une variable de type `Parameters` et une fonction `f` (second membre de l'équation)
`iteration()`	calcule une itération (la valeur de la solution à l'instant suivant)
`variation()`	retourne la variation entre 2 instants de calcul successifs
`getOutput()`	renvoie une variable de type `Values` qui contient les dernières valeurs calculées
`setInput(u)`	rentre dans `Scheme` les valeurs initiales

Fonctions du type `Parameters` :

`n(i)`	nombre de points dans la direction i ($0 = X$,$1 = Y$, $2 = Z$), y compris sur la frontière
`imin(i)`	indice des premiers points intérieurs dans la direction i
`imax(i)`	indice des derniers points sur la frontière dans la direction i
`dx(i)`	dimension d'une subdivision dans la direction i
`xmin(i)`	coordonnée minimale de Ω dans la direction i
`itmax()`	nombre d'itérations en temps
`dt()`	intervalle de temps entre 2 itérations
`freq()`	fréquence de sortie des résultats intermédiaires (nombre d'itérations entre 2 sorties)

Les points de calcul à l'intérieur du domaine Ω ont des indices (i, j, k) tels que:

$$\begin{array}{lcccl} \texttt{imin(0)} & \leq & \texttt{i} & < & \texttt{imax(0)} \\ \texttt{imin(1)} & \leq & \texttt{j} & < & \texttt{imax(1)} \\ \texttt{imin(2)} & \leq & \texttt{k} & < & \texttt{imax(2)} \end{array}$$

Les points sur la frontière du domaine $\partial\Omega$ ont des indices (i, j, k) tels que:

```
i = imin(0)-1  ou  i = imax(0)
j = imin(j)-1  ou  j = imax(1)
k = imin(k)-1  ou  k = imax(2)
```

Fonctions du type `Values`:

`init()`	initialise les points du domaine à 0
`init(f)`	initialise les points du domaine avec la fonction $f : (x, y, z) \mapsto f(x, y, z)$
`boundaries(g)`	initialise les points de la frontière avec la fonction $g : (x, y, z) \mapsto g(x, y, z)$
`v(i,j,k)`	si `v` est de type Values, la valeur au point d'indice (i, j, k)
`v.swap(w)`	si `v` et `w` sont de type `Values`, échange les valeurs de `v` et `w`

- Pour compiler, se placer dans le répertoire `PoissonSeq` et taper:

  `./build.py`

  (si cela ne marche pas, taper `python ./build.py`).

- Pour exécuter, rester dans le même répertoire et taper:

  `./install/gnu/Release/PoissonSeq.exe`

  Pour voir les options d'exécution possibles, taper `./install/gnu/Release/PoissonSeq.exe --help`

Noter les valeurs obtenues et les temps de calcul affichés, ils serviront de référence pour évaluer les autres versions.

Version multithreads avec OpenMP (grain fin)

Récupérer et décompresser un des fichiers

`TP1_OpenMP_FineGrain_incomplet.tar.gz` ou
`TP1_OpenMP_FineGrain_incomplet.zip`

Un répertoire `TP1/Poisson_OpenMP_FineGrain` est créé et contient le code source incomplet de la version OpenMP grain fin.

Attention:

Si vous avez déjà récupéré ce fichier et modifié les sources qui y sont contenues, créez un autre répertoire ailleurs et travaillez dans ce nouveau répertoire, sinon vous écraserez les modifications déjà faites !

- Pour compiler, se placer dans le répertoire `TP1/Poisson_OpenMP_FineGrain` et taper:

  ```
  ./build.py
  ```

  (si cela ne marche pas, taper `python ./build.py`).

- Pour exécuter, rester dans le même répertoire et taper (sur une seule ligne):

  ```
  ./install/gnu/Release/PoissonOpenMP_FineGrain.exe \
   threads=<n>
  ```

  (à la place de <n> taper 3 pour exécuter sur 3 threads (par exemple)

On fournit aussi un script pour comparer les speedups pour différents nombres de threads:

`./run.py <nthreads>`

qui lance plusieurs exécutions:

- version séquentielle
- version OpenMP sur 1 thread
- version OpenMP sur 2 threads

  ...
- version OpenMP sur `<nthreads>` threads

et à la fin affiche un graphe: nombre de threads .vs. temps CPU/speedup.

Remarque:

> Pour pouvoir utiliser le script `run.py`, il faut que le paquet `matplotlib` (pour la version de python utilisée) soit installé

Première partie:

Dans le répertoire `TP1/Poisson_OpenMP_FineGrain`, paralléliser avec OpenMP grain fin:

1. Chercher les parties du code à paralléliser
2. Ajouter ou adapter les `pragmas`
3. Identifier les variables partagées et privées
4. Compiler, lancer le code avec différents nombres de threads
5. Comparer avec la version séquentielle
6. Si différent, revenir en (2)

Version multithreads avec OpenMP (grain grossier)

Récupérer et décompresser un des fichiers

TP1_OpenMP_CoarseGrain_incomplet.tar.gz ou
TP1_OpenMP_CoarseGrain_incomplet.zip

Un répertoire `TP1/Poisson_OpenMP_CoarseGrain` est créé et contient le code source incomplet de la version OpenMP grain fin.

Pour compiler et exécuter on utilisera la même procédure que dans le cas OpenMP grain fin.

Dans le parallélisme OpenMP gros grain, on découpe explicitement le domaine de calcul en plusieurs parties, chaque partie est calculée par un thread.

Le type `Parameters` possède 2 fonctions supplémentaires utiles pour le // gros grain :

`imin_local(i,iThread)`	indice des premiers points intérieurs dans la direction i, pour la partie calculée par le thread `iThread`
`imax_local(i,iThread)`	indice des derniers points sur la frontière dans la direction i, pour la partie calculée par le thread `iThread`

Seconde partie:

Dans le répertoire `TP1/Poisson_OpenMP_CoarseGrain`, paralléliser avec OpenMP grain grossier:

1. Ajouter une région parallèle dans le programme principal autour de la boucle en temps
2. Identifier les instructions dans la région parallèle qui doivent s'exécuter en séquentiel et celles qui peuvent s'exécuter en parallèle
3. Identifier les variables partagées et privées
4. Ajouter ou adapter les pragmas correspondantes
5. Dans chaque thread, faire une partie des boucles internes
6. Compiler, lancer le code avec différents nombres de threads
7. Comparer avec la version séquentielle
8. Si différent, revenir en (2)

Envoyez par mail à marc.tajchman@cea.fr :

- une description du travail réalisé (1-2 pages maximum)
- le code source, avec vos modifications, dans une archive (n'envoyez pas les répertoires `build` et `install` qui contiennent des binaires)
- les fichiers run***.log et speedups***.pdf que vous avez obtenus

avant le 20/12/2020, au plus tard le 23/12/2020.

Envoyez vos fichiers source même s'ils contiennent des erreurs.