Modèles et techniques en programmation parallèle hybride et multi-cœurs

Travail pratique n°1

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

6/12/2024

Travail pratique n°1

On fournit un code séquentiel (non parallélisé) 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 utilise des différences finies pour approcher les dérivées partielles et découpe Ω en $n_1 \times n_2 \times n_3$ subdivisions.

But du TP

On demande de construire des versions parallélisées de ce code avec OpenMP, de comparer et interpréter leur comportement avec la version séquentielle.

Récupérer et décompresser un des fichiers `TP1_incomplet.tar.gz` ou `TP1_incomplet.zip`.

La décompression de l'un de ces fichiers crée 3 répertoires:

- `TP1_incomplet/PoissonSeq` contient la version séquentielle complète du code
- `TP1_incomplet/PoissonOpenMP_FineGrain` contient une version à compléter de la version "grain fin"
- `TP1_incomplet/PoissonOpenMP_CoarseGrain` contient une version à compléter de la version "grain grossier"

Structure du code séquentiel

Se placer dans le répertoire `TP1_incomplet/PoissonSeq`.

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

force.hxx: fonction qui calcule le second membre de l'équation en u point (x,y,z)

cond_ini.hxx: fonction qui calcule la valeur initiale de l'inconnue u en un point (x,y,z,t)

Fonctions du type `Scheme` :

`Scheme(P)`	construit une variable de type `Scheme` en lui donnant les paramètres géométriques et du schéma dans une variable de type `Parameters`
`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 intérieurs 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:

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

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

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 chaque version
(`PoissonSeq`, `PoissonOpenMP_FineGrain` et
`PoissonOpenMP_CoarseGrain`),
se placer dans le répertoire de la version et:

- pour compiler sur la machine locale:
  `python3 ./build.py`
- pour exécuter sur la machine locale, taper:
  `python3 ./run.py`
- pour exécuter sur le cluster Cholesky, taper:
  `python3 ./submit.py`
- pour générer un graphe des performances, taper:
  `python3 ./plot.py`

Pour voir les options de compilation et d'exécution possibles, ajouter l'option `--help` à l'une des commandes ci-dessus.

Quand vous travaillez avec le cluster Cholesky, utilisez

- `submit.py`, pour compiler et exécuter,
- `plot.py`, pour générer le graphe.

Quand vous travaillez sur machine locale (par exemple un ordinateur portable ou une station de travail), utilisez

- `build.py` pour compiler,
- `run.py` pour exécuter le code,
- `plot.py` pour générer le graphe.

Remarque:

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

Première partie: parallélisation OpenMP grain fin

Dans le répertoire `PoissonOpenMP_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. Si les résultats sont différents, revenir en (2)

Quand les résultats sont identiques entre la version séquentielle et la version parallèle, évaluer les performances de la parallélisation et essayer d'expliquer ces performances.

Seconde partie: parallélisation OpenMP grain grossier

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

Le code a une option supplémentaire `--balances n` : le nombre d'itérations où la répartition des calculs entre les threads est initialisée ou mise à jour (prendre n = 2 ou 3).

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

`balance()`	calcule pour une itération donnée la répartition des calculs entre les différents threads
`activateBalance(b)`	active ou non suivant que b est vrai ou faux (true/false), le recalcul de la répartition entre les threads
`nBalances()`	le résultat n de cette fonction indique que les n premières itérations doivent répartir la charge entre les threads
`startIndex(iThread)`	indice des premiers points intérieurs dans la direction X, pour la partie attribuée au thread `iThread`
`endIndex(iThread)`	indice des derniers points intérieurs dans la direction X, pour la partie attribuée au thread `iThread`

Seconde partie:

Dans le répertoire `PoissonOpenMP_CoarseGrain`, paralléliser avec OpenMP grain grossier:

1. Ajouter une grande 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 les pragmas et faire les tests
5. Une fonction de la classe Parameters : `balance`, recalcule le découpage du domaine, examiner cette fonction et l'utiliser pour essayer d'améliorer les speedups

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 5/1/2025 minuit.

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