# Modèles et techniques en programmation parallèle hybride et multi-cœurs

## Travail pratique n°1

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

16/12/2023

# 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 $\Omega$ 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 $\Omega$ 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 $\Omega$ 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` |

Se placer dans le répertoire `PoissonSeq`,

- pour compiler:

  ```
  python ./build.py
  ```

- pour exécuter sur la machine locale, taper:

  ```
  python ./run.py
  ```

- pour exécuter sur le cluster Cholesky, taper:

  ```
  python ./submit.py
  ```

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

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)

Un répertoire `TP1_incomplet/PoissonOpenMP_FineGrain` a été créé quand vous avez décompressé l'archive. Il contient le code source incomplet de la version OpenMP grain fin.

Se placer dans le répertoire `PoissonOpenMP_FineGrain`,

- pour compiler:
  `python ./build.py`
- pour exécuter sur la machine locale, taper:
  `./install/PoissonOpenMP_FineGrain threads=<n>`

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

Ajouter l'option `--help` pour voir les autres options disponibles

- pour voir l'évolution de temps de calcul quand on fait varier le nombre de threads sur la machine locale, taper:
  `python ./run.py`

- pour voir l'évolution de temps de calcul quand on fait varier le nombre de threads sur le cluster Cholesky, taper:
  `python ./submit.py`

Ces deux commandes lancent plusieurs exécutions avec un nombre de threads différent, sauvegardent l'affichage dans un fichier texte (commençant par log_) et tracent l'évolution du temps de calcul dans un fichier de type pdf.

Quand vous travaillez avec le cluster Cholesky, utilisez `submit.py`, quand vous travaillez sur machine locale (par exemple un ordinateur portable ou une station de travail), utilisez `run.py`

On peut spécifier des options pour les scripts `run.py` et `submit.py`. Taper la commande avec l'option `--help` pour les afficher

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_incomplet/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 les version parallèles, évaluer les performances de la parallélisation et essayer d'expliquer ces performances.

# Version multithreads avec OpenMP (grain grossier)

Un répertoire `TP1_incomplet/PoissonOpenMP_CoarseGrain` a été créé quand vous avez décompressé l'archive. Il contient le code source incomplet de la version OpenMP gros grain. Se placer dans le répertoire `TP1_incomplet/PoissonOpenMP_CoarseGrain`.

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 des fonctions supplémentaires utiles pour le // gros grain :

| | |
|---|---|
| `startIndex(iThread)` | indice des premiers points intérieurs dans la direction $X$, pour la partie calculée par le thread `iThread` |
| `endIndex(iThread)` | indice des derniers points intérieurs dans la direction $X$, pour la partie calculée par le thread `iThread` |

Seconde partie:

Dans le répertoire `TP1_incomplet/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 6/1/2024.**

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