

# 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 :*

$$\frac{\partial u}{\partial t} = \Delta u + f(x, t)$$

$$u(x, 0) = g(x) \quad x \in \Omega$$

$$u(x, t) = g(x) \quad x \in \partial\Omega, t > 0$$

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

Le code séquentiel utilise des différences finies pour approcher les dérivées partielles et découpe  $\Omega$  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 :

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

## Fonctions du type Parameters :

<code>n(i)</code>	nombre de points dans la direction $i$ ( $0 = X, 1 = Y, 2 = Z$ ), y compris sur la frontière
<code>imin(i)</code>	indice des premiers points intérieurs dans la direction $i$
<code>imax(i)</code>	indice des derniers points sur la frontière dans la direction $i$
<code>dx(i)</code>	dimension d'une subdivision dans la direction $i$
<code>xmin(i)</code>	coordonnée minimale de $\Omega$ dans la direction $i$
<code>itmax()</code>	nombre d'itérations en temps
<code>dt()</code>	intervalle de temps entre 2 itérations
<code>freq()</code>	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:

$$\begin{aligned} \text{imin}(0) &\leq i < \text{imax}(0) \\ \text{imin}(1) &\leq j < \text{imax}(1) \\ \text{imin}(2) &\leq k < \text{imax}(2) \end{aligned}$$

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

$$\begin{aligned} i &= \text{imin}(0)-1 \quad \text{ou} \quad i = \text{imax}(0) \\ j &= \text{imin}(j)-1 \quad \text{ou} \quad j = \text{imax}(1) \\ k &= \text{imin}(k)-1 \quad \text{ou} \quad k = \text{imax}(2) \end{aligned}$$

## Fonctions du type Values:

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

- 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

## Premier travail:

Dans le répertoire TP1/Poisson\_OpenMP\_FineGrain, paralléliser avec OpenMP grain fin:

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

Envoyez votre code source modifié (compressé dans une archive), avec les résultats obtenus (copie des sorties écrans et fichiers pdf obtenus avec le script run.py) **le 20/12/2020 au plus tard.**

# 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.

**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\_CoarseGrain 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))

## Premier travail:

Dans le répertoire TP1/Poisson\_OpenMP\_fine\_grain, paralléliser avec OpenMP grain fin:

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

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