

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

## Travail pratique 2

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

02/12/2020

## Travail pratique 2

On part d'un code parallélisé avec MPI 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 utilise des différences finies pour approcher les dérivées partielles et découpe  $\Omega$  en  $n_0 \times n_1 \times n_2$  subdivisions.

# Structure du code

Récupérer et décompresser un des fichiers `TP2_MPI.tar.gz` ou `TP2_MPI.zip`.

Se placer dans le répertoire `TP2/PoissonMPI` créé.

Le code 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 Scheme en lui donnant une variable de type Parameters et une fonction $f$ (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>synchronize()</code>	copie les valeurs sur le bord d'un domaine vers les domaines voisins
<code>getOutput()</code>	renvoie une variable de type Values qui contient les dernières valeurs calculées
<code>setInput(u)</code>	rentre dans Scheme les valeurs initiales

Sur chacun des  $p$  sous-domaines de  $\Omega$  (chaque sous-domaine est géré par un processus MPI)

Fonctions du type Parameters :

`imin(i)`            indice des premiers points intérieurs dans la direction  $i$  pour le sous-domaine courant

`imax(i)`            indice des derniers points intérieurs dans la direction  $i$  pour le sous-domaine courant

`imin_global(i)`    indice des premiers points intérieurs dans la direction  $i$

`imax_global(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  $i$

`xmax(i)`            coordonnée maximale de  $\Omega$  dans la direction  $i$

<code>itmax()</code>	nombre d'itérations en temps
<code>dt()</code>	intervalle de temps entre 2 itérations
<code>neighbour(k)</code>	indice des sous-domaines voisins (1-2 à gauche ou à droite suivant X) (3-4 en arrière ou en avant suivant Y) (5-6 en bas ou en haut suivant Z) -1 si pas de voisin sur un côté du sous-domaine
<code>rank()</code>	indice du processus (= nombre de processus)
<code>size()</code>	nombre du processus (= nombre de sous-domaines)
<code>freq()</code>	fréquence de sortie des résultats intermédiaires (nombre d'itérations entre 2 sorties)

**Pour un processus MPI  $P$**  : les points de calcul à l'intérieur du sous-domaine  $\Omega_p$  ont des indices  $(i, j, k)$  tels que:

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

**Pour un processus MPI  $P$**  : les points sur la frontière du sous-domaine  $\partial\Omega$  ont des indices  $(i, j, k)$  tels que:

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

Point frontière du sous domaine = point au bord du sous-domaine voisin ou point frontière du domaine global.

## Fonctions du type Values pour le sous-domaine courant :

<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:

```
mpirun -n X ./install/Release/PoissonMPI.exe
```

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

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

# Version hybride MPI-OpenMP (grain fin)

Récupérer et décompresser un des fichiers

[TP2\\_MPI\\_OpenMP\\_FineGrain\\_incomplet.tar.gz](#) ou  
[TP2\\_MPI\\_OpenMP\\_FineGrain\\_incomplet.zip](#)

Un répertoire `TP2/Poisson_MPI_OpenMP_FineGrain` est créé et contient le code source incomplet de la version MPI 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/PoissonMPI\_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):

```
mpirun -n 2 ./install/Release/MPIPoissonOpenMP_FineGrain  
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 TP2/PoissonMPI\_OpenMP\_CoarseGrain, paralléliser avec OpenMP grain grossier:

1. Ajouter une région parallèle OpenMP dans le programme principal autour de la boucle en temps
2. Identifier les instructions dans la région parallèle OpenMP 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](mailto: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.**