Modèles et techniques en programmation parallèle hybride et multi-cœurs

Travail pratique 2

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

mise à jour le 06/01/2025

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 :*
>
> $$\begin{aligned} \frac{\partial u}{\partial t} &= \Delta u + f(x,t) \\ u(x,0) &= g(x) && x \in \Omega \\ u(x,t) &= g(x) && x \in \partial\Omega, t > 0 \end{aligned}$$
>
> *où* f *et* g *sont des fonctions données.*

Le code utilise des différences finies pour approcher les dérivées partielles et découpe Ω en $n_0 \times n_1 \times n_2$ subdivisions.

Structure du code

Récupérer et décompresser un des fichiers TP2_incomplet.tar.gz ou TP2_incomplet.zip.

Se placer dans le répertoire `TP2_incomplet/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` :

`Scheme(P)`	construit une variable de type `Scheme` en lui donnant 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
`synchronize()`	copie les valeurs sur le bord d'un domaine vers les domaines voisins
`getOutput()`	renvoie une variable de type `Values` qui contient les dernières valeurs calculées
`setInput(u)`	rentre dans `Scheme` les valeurs initiales

Sur chacun des p sous-domaines de Ω (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 Ω dans la direction i
`xmax(i)`	coordonnée maximale de Ω dans la direction i

`itmax()`	nombre d'itérations en temps
`dt()`	intervalle de temps entre 2 itérations
`neighbour(k)`	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
`rank()`	indice du processus (= nombre de processus)
`size()`	nombre du processus (= nombre de sous-domaines)
`freq()`	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 Ω_p ont des indices (i, j, k) tels que:

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

Pour un processus MPI P : les points sur la frontière du sous-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
```

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 :

`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`

Compilation et test sur une machine locale

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

  `./build.py`

  (si cela ne marche pas, taper `python3 ./build.py`).

- Pour lancer plusieurs exécutions sur 1, 2, 3, ..., 8 processus, tapez:

  `./run.py`

- Pour générer un graphe de performance des exécutions lancées par la commande run.py :

  `./plot.py`

Compilation et test sur le cluster cholesky

Sur cholesky, taper :

`./submit.py`

Cette commande compile le code, lance les exécutions et génère le graphe des performances.

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

Version hybride MPI-OpenMP

Le répertoire PoissonMPI_OpenMP contient le code parallélisé par MPI et OpenMP, mais où la parallélisation OpenMP est incomplète.

Compléter le code avec une parallélisation OpenMP grain fin.

Expliquer pourquoi une parallélisation grain grossier n'est pas intéressante ici (comparer avec le TP1).

Sur une machine locale,
pour compiler, tapez:

```
./build.py
```

pour exécuter le code, tapez :

```
./run.py --npMax 8
```

qui lance les exécutions :

- 1 processus - 1 thread,
- 8 processus - 1 thread,
- 4 processus - 2 threads, 2 processus - 4 threads et
- 1 processus - 8 threads

pour générer un graphe de performances, tapez:

```
./plot.py
```

Sur le cluster cholesky,

pour compiler et exécuter, tapez:

```
./submit.py
```

qui lance les mêmes exécutions que run.py utilisé sur une machine locale :

- 1 processus - 1 thread,
- 8 processus - 1 thread,
- 4 processus - 2 threads, 2 processus - 4 threads et
- 1 processus - 8 threads

pour générer un graphe de performances, tapez:

```
./plot.sh
```

Remarque

On rappelle que le nombre de processus MPI et de threads OpenMP est trop petit ici pour que la programmation hybride apporte un avantage/désavantage significatif par rapport au "tout MPI"

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

- une description du travail réalisé (1-2 pages maximum) qui inclus, si vous l'avez, le graphe des performances,
- le code source, avec vos modifications, dans une archive (n'envoyez pas les répertoires `build` et `install` qui contiennent des binaires),
- autant que possible, les fichiers log*.txt qui contiennent les sorties écran

avant le 31/01/2025.

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