AMSI03 Projet 1
Programmation parallèle par Grain Fin et Grain Grossier

Qingqing HU

I. Description du problème

Étant donné un domaine $\Omega =]0,1[^3$, on cherche la solution $u(x,t), t \geq 0$ qui vérifie :

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

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

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

On le résout par intération pour un instant donné :

$$u(x,t^{n+1}) = (t^{n+1} - t^n)(\Delta u(x,t^n) + f(x,t^n)) + u(x,t^n)$$

II. Grain Fin

Par cette méthode, on crée une région parallèle chaque fois que l'on lance une itération. Donc, on parallélise le code dans la fonction *Scheme : :iteration_ domaine()* dans la fichier *scheme.cxx* en mettant *#pragma omp parallel for reduction(+ :du_ sum_ local) private(x,y,z,j,k,du,du1,du2)*.

Pour réduire un petit peu le temps d'initialisation, je met aussi *#pragma omp parallel for default(shared) private(...)* dans la fonction *Value : :init()* et *Values : :init(callback_ t f)* dans la fichier *value.cxx*.

Du fait qu'on ne utilise pas les fonctions de MPI dans la région parallèle, on choisit donc MPI_THREAD_SINGLE comme le niveau de compatibilité.

Le temps de calcul est le suivant :

nombre de processus	nombre de thread	temps initialisation(s)	temps calcul(s)	temps total(s)
1	8	2.40171	20.7	23.2
2	4	1.60991	17.4	19.2
4	2	0.869307	13.8	14.8
8	1	0.633034	13.2	14

TABLE 1 – Temps de Grain Fin

III. Grain Grossier

Par cette méthode, on crée une région parallèle dans le programme principal avant boucle en temps en mettant toutes les variables déjà définies *shared*. Par apport à la modification dans le fichier *main.cxx*, on met en parallèle l'instruction *C.iteration()* et *u_ 0.init(cond_ ini)*. On fait les autres instruction dans ce boucle s'exécuter en séquentiel par les commande *#pragma omp master* ou *#pragma omp single*. Du fait qu'on choisit MPI_THREAD_FUNNELED comme le niveau de

compatibilité, cela signifie que pour pouvoir faire des appels MPI dans le cas Grain Grossier, on doit faire des appels MPI dans les régions contrôlées par les pragma OpenMP MASTER. Donc, on utilise les pragma OpenMP MASTER pour *C.synchronize()* dans laquelle il y a des aplles MPI.

Concernant les modifications dans le fichier *values.cxx* et *scheme.cxx*, comme Grain Grossier s'appuie sur une décomposition de domaine, on utilise les indices locals (*m_ imin_ thread[rank][ithread]*, *m_ imax_ thread[rank][ithread]*) au cours de l'appelle de la fonction *C.iteration()* et *u_ 0.init(cond_ ini)*. Les indices locals sont *private* parce qu'ils sont définis dans la région parallèle.

Le temps de calcul est le suivant :

nombre de processus	nombre de threads	temps initialisation(s)	temps calcul(s)	temps total(s)
1	8	1.79546	23.3	25.2
2	4	0.739704	21.4	22.8
4	2	0.716078	15	15.9
8	1	0.89614	14	15.2

TABLE 2 – Temps de Grain Grossier

D'après les tableaux (1) et (2), on observe que les performances des Grain Groisser et Grain Fin sont similaire dans le cas 10 intérations. Et Grain Fin est un peu meilleur. De plus, en gardant le produit du nombre de processus et du nombre de threads constant, plus de processus on utilise, plus on gagne dans notre cas.

On expliquera un peu comment calculer les indices locals associés à chaque thread. En fait, le code suivant nous permet de découper le domaine assoicés à chaqua processus en quelques sous-domaine dans la direction ayant le plus de points.

```
int idecoupe = -1, iT, maxn = -1;
for (int i = 0; i < 3; i++) {
  m_imin_thread[i].resize(m_nthreads);m_imax_thread[i].resize(m_nthreads);

  for (iT = 0; iT < m_nthreads; iT++)
  {
    m_imin_thread[i][iT] = m_imin[i];
    m_imax_thread[i][iT] = m_imax[i];
// Les indices locals associé à chaque thread sont égals aux indices locals associé à chaque processus.
  }
  if (m_imax[i] - m_imin[i] > maxn) {
    idecoupe = i;maxn = m_imax[i] - m_imin[i];
// On trouve idecoupe qui signifie la direction ayant le plus de points dans son propre processus.
  }
}
maxn++;int di = maxn / m_nthreads;
int i0 = m_imin[idecoupe], i1;
for (iT = 0; iT < m_nthreads; iT++) {
  i1 = i0 + di;m_imin_thread[idecoupe][iT] = i0;
  m_imax_thread[idecoupe][iT] = i1;i0 = i1 + 1;
// On recalcule les indices locals associé à chaque thread dans la direction idecoupe.
}
m_imax_thread[idecoupe][m_nthreads - 1] = m_imax[idecoupe];
// Notez que le m_imax_thread associé le dernier thread doit être égal au m_imax du processus.
```