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*.

```
double Scheme::iteration_domaine(...)
{ ...Déclaration et définition des variables
#pragma omp parallel default(shared) private(j,k,x,y,z,du1,du2,du)
{
#pragma omp for reduction(+:du_sum)
for (i = imin; i < imax; i++)
  for (j = jmin; j < jmax; j++)
      for (k = kmin; k < kmax; k++) {...Calcul d'une itération
       du_sum += du > 0 ? du : -du;
      }
}
return du_sum;
}
```

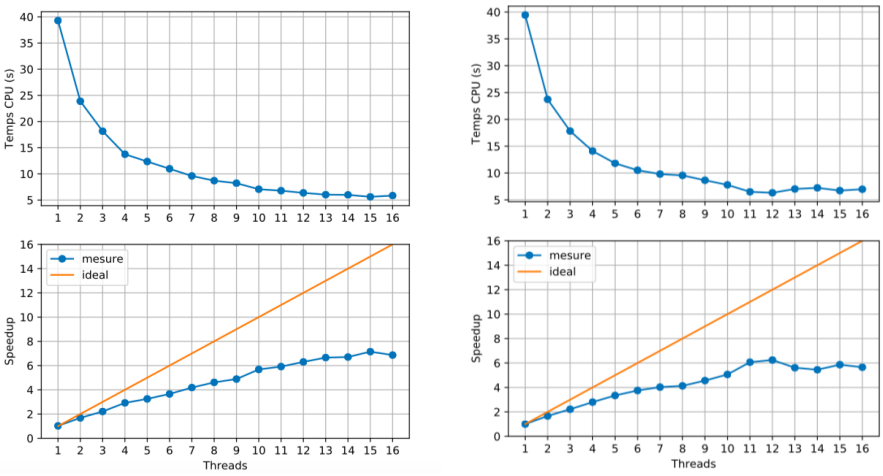
Pour réduire un petit peu le temps d'initialisation, je met aussi *#pragma omp parallel for default(shared) private(...)* dans la fonction *Value : :init()* et *Value : :boundaries()* dans la fichier *value.cxx*. Dans la suite, on garde cette modification de *value.cxx* pour la méthode Grain Grossier.

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*. On met en parallèle l'instruction *C.iteration()*. On fait les autres instruction dans ce boucle s'exécuter en séquentiel par les commande *#pragma omp master*. Comme Grain Grossier s'appuie sur une décomposition de domaine, on utilise les indices locals au cours de l'appelle de la fonction *C.iteration()*. Les indices locals sont *private* parce qu'ils sont définis dans la région parallèle.

```
bool Scheme::iteration(){
  #if defined(_OPENMP)
  int numthread = omp_get_thread_num();
  int nx1 = m_P.imin_local(0,numthread);int nx2 = m_P.imax_local(0,numthread);
  #else
  int nx1 = m_P.imin(0);int nx2 = m_P.imax(0);
  #endif
  double m_duv1;
  m_duv1 = iteration_domaine(nx1, nx2,m_P.imin(1), m_P.imax(1),m_P.imin(2), m_P.imax(2));
  #pragma omp master
  m_t += m_dt;
  #pragma omp single
  m_duv = 0;
  #pragma omp atomic
  m_duv+=m_duv1;
  #pragma omp single
  m_u.swap(m_v);
  return true;
}
```

IV. Grain Groisser versus Grain Fin



(a) Grain Fin (b) Grain Grossier

FIGURE 1 – Taille de problème 400x400x400, nombre d'itération = 10

Ici, on est dans un cas « scalabilité forte », c'est-à-dire qu'on analyse sa performance en variant nombre de thread pour un problème dont la taille est fixée. On voit que les résultats ne peuvent pas atteindre l'état idéal. Les raisons sont les suivantes : (1) Il y a un certain pourcentage de la composante séquentiel irréductible dans notre code, sourtout le temps de l'initialisation. Selon la loi d'Amdahl, si la composante séquentiel irréductible contribue pour une fraction α au temps d'exécution du programme séquentiel, alors l'accélération maximale est $1/\alpha$. (2) Lors de l'augmentation de nombre de threads, l'accélération est aussi limitée par nombre de coeur de la machine. Les performances des deux méthodes sont presque indentiques au début, j'ai séparément augmenté la taille de problème à 800x800x400 et le nombre d'itération à 30, ça ne change pas grandes choses.