AMS-I03 Programmation hybride et multi-coeurs
Compte Rendu TP1

El Herichi Hafsa

22 Décembre 2020

Lors de ce TP, nous avons eu à paralléliser un programme en séquentiel qui permettait de résoudre l'équation de Poisson $\frac{\partial u}{\partial t} = \Delta u + f(x,t)$ sur un domaine $\Omega = [0,1]^3$. Le but était donc de le paralléliser en OpenMP par deux méthodes différente, la première est celle du Grain Fin et la seconde le Grain Grossier.

Nous allons énoncer les différences apportées au code pour permettre la parallélisation ainsi que les résultats obtenus pour chaque méthode ci-dessous. Ces résultats là ont été obtenus grâce aux tests des deux méthodes sur la machine *Salle* de l'ENSTA.

Grain Fin

Le principe du Grain Fin est de permettre une parallélisation du code par l'ajout de pragmas devant les boucles à paralléliser. Cela permet à OpenMP de faire le découpage de l'ensemble de valeurs des indices automatiquement.

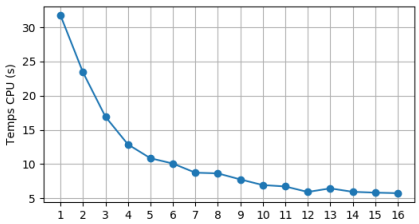
Pour cette méthode là, il a donc fallu ajouter des pragmas là où les boucles pouvaient être parallélisées.

Le seul fichier qu'il a fallu modifier a été le fichier scheme.cxx. On ajoute les pragma pour paralléliser, ensuite on identifie les variables privées qui sont $i, j, k, x, y, z, du, du1, du2$ et on remarque qu'il faut inclure une réduction pour *du_sum* pour avoir la somme globale que l'on cherche. On remarque ci-dessous les modifications apportées à ce fichier sur la première ligne.

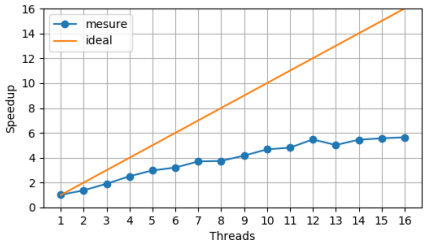
```
#pragma omp parallel for default(shared),
private(i,j,k,x,y,z,du,du1,du2), reduction(+:du_sum)

  for (i = imin; i < imax; i++)
    for (j = jmin; j < jmax; j++)
      for (k = kmin; k < kmax; k++) {
        du1 = (-2*m_u(i,j,k) + m_u(i+1,j,k) + m_u(i-1,j,k))*lam_x
            + (-2*m_u(i,j,k) + m_u(i,j+1,k) + m_u(i,j-1,k))*lam_y
            + (-2*m_u(i,j,k) + m_u(i,j,k+1) + m_u(i,j,k-1))*lam_z;

        x = xmin + i*m_dx[0];
        y = ymin + j*m_dx[1];
        z = zmin + k*m_dx[2];
        du2 = m_f(x,y,z);

        du = m_dt * (du1 + du2);
        m_v(i, j, k) = m_u(i, j, k) + du;
        du_sum += du > 0 ? du : -du;
      }
    return du_sum;
}
```

Une fois que cela est fait, on peut lancer notre programme et le tester avec différents nombres de threads. Ici, nous le testons pour des valeurs entre 1 et 16. On remarque alors l'évolution du temps de CPU ainsi que celle du speedup 1.

Nous remarquons une baisse considérable pour ce qui est du temps d'exécution entre le cas où nous utilisons 1 thread et le cas où nous utilisons 16 threads. Nous passons notamment de plus de 30s à environs 5s. Cependant, grâce à la courbe de l'accélération, nous remarquons que nous sommes assez loins du cas idéal où l'on devrait réduire notre temps d'exécution de 16. Nous n'avons donc pas une grande efficacité pour le cas où la parallélisation est faite par la méthode du Grain Fin.



(a) Evolution du temps CPU en fonction du nombre de Threads



(b) Accélération du Grain Fin

FIGURE 1 – Evolution du Temps CPU et accélération pour la parallélisation en Grain Fin

Grain Grossier

Contrairement à ce que nous avons fait pour la méthode du Grain Fin, nous n'allons pour le Grain Grossier pas modifier le fichier scheme.cxx mais uniquement le fichier main.cxx . Nous commençons par ajouter une région parallèle autour de la boucle en temps grâce à l'instruction

```
#pragma omp parallel for
```

que nous plaçons juste avant la boucle for en temps.

Cette fois-ci, lorsqu'une instruction peut être exécutée uniquement par un thread et n'a pas besoin d'être parallélisée, on iclut la ligne

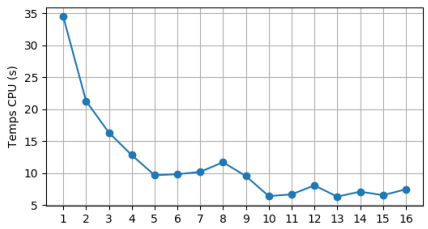
```
#pragma omp single
```

pour faire en sorte que ce qui suit soit exécuté en séquentiel et par un seul thread. Et pour finir, nous procédons à un découpage manuel des boucles internes pour attribuer à tous les threads un nombre égal d'instructions.

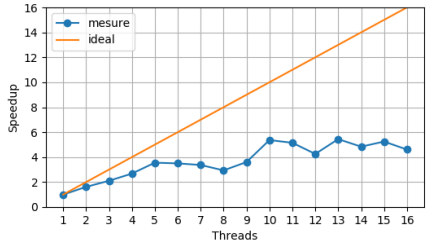
```
for (int iX = n1 ; iX < n2 ; iX ++)
          if (freq > 0 && iX % freq == 0) {
            T_other.start();
            C.getOutput().plot(iX);
            T_other.stop();
          }
```

Chaque Thread pourra ensuite exécuter ses instructions et le temps global sera considérablement réduit.

Nous regardons pour cette méthode les mêmes caractéristiques que pour l'autre. Et nous obtenons les graphiques suivants 2 :



(a) Evolution du temps CPU en fonction du nombre de Threads



(b) Accélération du Grain Fin

FIGURE 2 – Evolution du Temps CPU et accélération pour la parallélisation en Grain Grossier

Cette fois-ci encore, nous avons une diminution du temps qui est visible entre le cas où nous utilisons 1 thread et celui où nous en utilisons 16. Nous passons alors de 35s à 5s. Ceci n'est néanmoins pas assez suffisant puisque l'accélération est loin de se rapprocher du cas idéal.

Conclusion

En ce qui concerne l'implémentation des deux méthodes, il est certes plus facile d'implémenter celle du Grain Fin qui prend moins de détails en compte et qui fait le découpage automatiquement. Cependant, en comparant les courbes obtenues pour les deux méthodes, nous remarquons que la courbe de l'accélaration de la méthode du Grain Grossier se rapproche nettement plus de la courbe de l'accélération idéale. Nous avons donc une meilleure efficacité en utilisant la deuxième méthode que la première. Il est donc préférable d'utiliser cette méthode là même si son implémentation est moins facile que la première.