AMS-I03 Programmation hybride et multi-coeurs
Compte Rendu TP2

El Herichi Hafsa

22 Janvier 2020

Lors de ce TP, nous avons eu à implémenter la parallélisation hybride MPI-OpenMP d'un programme 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 MPI-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 cette 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 de la méthode du Grain Fin est d'ajouter quelques lignes de pragma permettant la parallélisation en OpenMP d'un code déjà parallélisé en MPI.
Il n'y a donc pas de grands changements à effectuer pour cela. Nous commençons donc par modifier l'initialisation de MPI pour faire comprendre au programme que cette parallélisation est hybride, afin qu'il prenne en compte la parallélisation en OpenMP également.
Nous allons donc dans le fichier main.cxx et nous modifions la ligne :

```
MPI_Init(&argc , &argv);
```

par le bout de code suivant :

```
int provided;

MPI_Init_thread(&argc,&argv, MPI_THREAD_FUNNELED, &provided);

if (provided < MPI_THREAD_FUNNELED)
    MPI_Abort(MPI_COMM_WORLD, -1);
```

La commande *MPI_THREAD_FUNNELED* permet de faire des appels MPI dans les régions séquentielles du programme parallélisé par OpenMP dans le cas du Grain Fin.

A présent, nous passons à la modification d'un autre fichier. Il s'agit du fichier *scheme.cxx* et cette fois-ci, nous allons uniquement rajouter un pragma qui permet de définir les différentes variables à protéger ainsi que celles à partager et celles pour lesquelles nous avons besoin de faire une réduction.

Ceci nous donne le bout de code suivant, pour lequel seule la première ligne de pragma a été ajoutée.

```
#pragma omp parallel for default(shared),private(i,j,k,x,y,z,du,du1,du2),
reduction(+:du_sum_local)
for (i = imin; i <= imax; i++)
  for (j = jmin; j <= jmax; j++)
    for (k = kmin; k <= kmax; k++) {
          .
          .
          .
    }

  MPI_Allreduce(&du_sum_local, &du_sum, 1, MPI_DOUBLE, MPI_SUM, m_P.comm());
  return du_sum;
}
```

Pour ce qui est du Grain Fin, ce sont les seules modifications que nous avons eu à effectuer afin que le programme soit fonctionnel et renvoie les résultats attendus en moins de temps.

Nous avons donc le tableau ci-dessous qui récapitule les résultats obtenus selon le nombre de Proc. et Threads avec lesquels nous avons lancé l'exécutable.

n	t	Temps d'exécution (s)
1	1	90.5
1	8	23.8
2	4	20.2
4	2	17.4
8	1	17.7

TABLE 1 – Evaluation du temps d'exécution du programme Grain Fin avec un nombre variant de Proc. et Threads.

Nous remarquons qu'au fur et à mesure que nous augmentons le nombre de Proc ou de Threads, le temps d'exécution diminue. Le gain de temps est donc considérable lorsque nous augmentons ne serait-ce que le nombre de Threads, comme nous pouvons le remarquer pour le cas $n = 1, t = 8$ pour lequel nous avons un temps d'exécution 3 fois plus petit que celui obtenu avec $n = 1, t = 1$.

Grain Grossier

La méthode du Grain Grossier requiert d'avantages de modifications du code que la méthode précédente. Nous aurons donc plusieurs fichiers à modifier et plusieurs pragma à ajouter afin que la parallélisation hybride puisse fonctionner.

On commence donc par le fichier *main.cxx*, qui sera grandement modifié pour permettre cette parallélisation.

Cette fois-ci, nous modifierons beaucoup plus la partie à paralléliser. Après l'ajout de plusieurs pragmas définissant chacun les régions à exécuter en séquentiel ou en parallèle, nous obtenons le bout de code suivant :

```
Values u_0(Prm);
u_0.boundaries(cond_lim);

#pragma omp parallel default(shared)
{
  u_0.init(cond_ini);
#pragma omp barrier
#pragma omp single
{
  C.setInput(u_0);
  T_init.stop();
  MPI_Barrier(Prm.comm());

  if (Prm.rank() == 0) {
    std::cout << "\n  temps init "
              << std::setw(10) << std::setprecision(6)
              << T_init.elapsed() << " s\n" << std::endl;
    }
  }//end pragma single


  for (int it=0; it < itMax; it++) {
  #pragma omp single
  {
    if (freq > 0 && it % freq == 0) {
      T_other.start();
      C.getOutput().plot(it);
      T_other.stop();
    }
  }//end pragma omp single
```

```
  #pragma omp master
      T_comm.start();
  #pragma omp barrier
      C.synchronize();
  #pragma omp barrier

  #pragma omp single
  {
      T_comm.stop();
  }

  #pragma omp master
        T_calcul.start();
  #pragma omp barrier
       C.iteration();
  #pragma omp barrier

  #pragma omp single
  {
      T_calcul.stop();

      if (Prm.rank() == 0) {

        std::cout << "iter. " << std::setw(3) << it
        << "  variation " << std::setw(10) << std::setprecision(4)
        << C.variation()
        << "  temps calcul " << std::setw(8) << std::setprecision(3)
        << T_calcul.elapsed() << " s"
        << "  comm. " << std::setw(8) << std::setprecision(3)
        << T_comm.elapsed() << " s"
        << std::endl;
      }
    }
  }
}
```

Ainsi, nous avons commencé par définir la partie que l'on souhaite exécuter en parallèle, grâce au pragma **#pragma omp parallel default (shared)**, puis nous avons introduit le pragma **#pragma omp single** à chaque fois que nous souhaitions que le code s'exécute en séquentiel. En plus de ces deux pragmas, il a fallu ajouter **#pragma omp barrier**, pour indiquer au programme qu'il fallait attendre que tous les threads aient fini leur travail avant d'aller plus loin, ainsi que le pragma **#pragma omp master**, utilisé pour indiquer qu'il fallait que ce soit le thread principal qui exécute la partie du code qui suivait.

Il reste à présent deux fichiers à modifier. Le fichier *scheme.cxx* et le fichier *values.cxx*. Commençons par le fichier *values.cxx*.

Dans ce fichier-là, il va falloir modifier les deux fonctions *init* en y ajoutant les lignes de code qui suivent, afin qu'elles puissent prendre en compte le changement d'indices en passant d'un thread à un autre.

```
#ifdef _OPENMP
  int iThread = omp_get_thread_num();
#else
  int iThread = 0.0;
#endif

  int imin_thread = m_p.imin_thread(0, iThread);
  int jmin_thread = m_p.imin_thread(1, iThread);
  int kmin_thread = m_p.imin_thread(2, iThread);

  int imax_thread = m_p.imax_thread(0, iThread);
  int jmax_thread = m_p.imax_thread(1, iThread);
  int kmax_thread = m_p.imax_thread(2, iThread);
```

Nous passons ensuite au fichier *schema.cxx*, pour lequel il faudra procéder à de nombreuses modifications afin qu'il puisse prendre en compte la parallélisation hybride souhaitée.

Dans ce fichier, il va falloir modifier les fonctions : iteration, iteration_domaine ainsi que synchronize.

L'idée est qu'il va falloir introduire des pragmas qui vont permettre l'exécution en parallèle de ces fonctions là en OpenMP en plus de MPI.

Pour la première fonction, ceci nous donne :

```
#ifdef _OPENMP
  int iThread = omp_get_thread_num();
#else
  int iThread = 0;
#endif

#pragma omp single
  m_duv = 0.0;

 double du_sum = iteration_domaine(
    m_P.imin_thread(0, iThread), m_P.imax_thread(0, iThread),
    m_P.imin_thread(1, iThread), m_P.imax_thread(1, iThread),
    m_P.imin_thread(2, iThread), m_P.imax_thread(2, iThread));

  #pragma omp barrier

  #pragma omp atomic
  m_duv += du_sum;

  #pragma omp single
  {
  m_t += m_dt;
  m_u.swap(m_v);
  }
```

Pour ce qui est des deux fonctions restantes, l'idée est d'indiquer au programmes des zones à exécuter en séquentiel, par l'ajout de **#pragma omp single**, qui devront englober les fonctions **MPI_AllReduce**. Tout cela afin de faire comprendre au programme que les communications entre processeurs doivent se faire par un seul et unique thread.

Malheureusement, cette partie du TP a été assez difficile à mettre en place puisque l'exécution annonçait des segfault ou alors le résultat était faussé dès qu'une modification avait eu lieu.

Nous avons néanmoins réussi à obtenir quelques résultats, en gardant telles quelles les fonctions citées plus haut. Certes, les résultats ne sont pas optimaux, puisque cette partie reste à améliorer, mais ils donnent une idée de ce que l'algorithme pourrait faire une fois bien implémenté.

n	t	Temps d'exécution (s)
1	1	153
1	8	31.9
2	4	28.7
4	2	25.7
8	1	24.5

TABLE 2 – Evaluation du temps d'exécution du programme Grain Grossier avec un nombre variant de Proc. et Threads.

Les résultats énoncés plus haut nous indiquent donc que la partie du code qui a été parallélisée a permis de réduire considérablement le temps d'exécution du programme.

Conclusion

Ce TP nous a permis de nous initier à la programmation hybride MPI-OpenMP afin de nous faire voir les résultats pouvant être obtenus par cette dernière et nous faire

découvrir comment cela marchait plus en détail. Ceci a donc permis la compréhension de la programmation hybride et son utilité.

Néanmoins, il reste toujours que la méthode du Grain Fin, plus simple à implémenter est plus performante en terme d'optimisation de temps d'exécution du programme et permet d'obtenir les mêmes résultats que la méthode par Grain Grossier. Il est donc préférable d'utiliser la première méthode pour une programmation hybride MPI-OpenMP.