Projet OS202 - Programmation Parallèle

Modélisation de tourbillons sur un tore en deux dimensions

COSTA WERNECK André
GRECO PICOLI Leonardo

March 19, 2023

1 Introduction

L'objectif de ce projet est de paralléliser un programme qui simule une modélisation de tourbillons sur un tore en deux dimensions en utilisant les outils informatiques **MPI** (Mémoire distribuée) et **OpenMP** (Mémoire partagée).

Le projet est implémenté en **C++** et a pour but d'explorer et d'implémenter différentes manières de paralléliser la modélisation.

Les caractéristiques de l'ordinateur utilisé pour la mise en œuvre et l'exécution du projet sont présentées ci-dessous. Il convient de préciser qu'il s'agit de la sortie de la commande lscpu de linux.
Architecture: x86-64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian **Address sizes**: 39 bits physical, 48 bits virtual
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 140
Model name: 11th Gen Intel(R) Core(TM) i7-11390H @ 3.40GHz
Stepping: 2
CPU MHz: 2918.399
BogoMIPS: 5836.79
Virtualization: VT-x
Hypervisor vendor: Microsoft
Virtualization type: full
L1d cache: 192 KiB
L1i cache: 128 KiB
L2 cache: 5 MiB
L3 cache: 12 MiB

La parallélisation du code est divisée en 3 parties principales:
- **Partie 1**: Séparation interface-graphique et calcul
- **Partie 2**: parallélisation en mémoire partagée
- **Partie 3**. parallélisation en mémoire distribuée et partagée

Un dernier exercice est encore proposé pour la fin du projet, qui consiste en la réflexion sur l'approche mixte Eulérienne-Lagrangienne.

2 Séparation interface-graphique et calcul

2.1 Premières consignes

Dans cette partie on cherche à séparer l'interface graphique du calcul proprement dit.

On entend par **interface-graphique** :

- L'affichage à l'écran
- La gestion des événements (claviers et fermeture de la fenêtre).

Le **calcul** concerne quant à lui le calcul du champ de vitesse et le déplacement des particules (et des tourbillons si ceux-ci sont mobiles).

Remarquons que ces calculs ne sont nécessaires que si on avance en temps. Il faut donc que le processus s'occupant de l'interface graphique envoie des "ordres" aux processeurs s'occupant du calcul à chaque itération de la boucle d'événement pour que le processus effectuant les calculs puisse(nt) savoir si un calcul est nécessaire ou non.

2.2 L'implementation du code

Pour faire cette partie, on commence par initialiser le MPI:

Listing 1: C++ MPI Communication

```
1 MPI_Init(&nargs, &argv);
2 MPI_Comm_size(MPI_COMM_WORLD, &n_ranks);
3 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

On utilise la variable my_rank = 0 comme le processus responsable pour faire la partie interface-graphique. Pour gérer les entrées de l'utilisateur, nous enregistrons dans une variable appelée *action* toutes les entrées possibles de l'utilisateur, qui sont stockées sous la forme d'un **char**. Nous envoyons l'action au processus 1 via la commande **MPI_Send**, afin qu'il commence à effectuer les calculs. Un extrait du code est présenté ci-dessous :

Listing 2: Process 0 - MPI Broadcasting user action

```
while (myScreen.pollEvent(event) && action != ACTION_EXIT)
{
    // event resize screen
    action = NO_ACTION;
    // event play animation
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::P))
        action = ACTION_PLAY;
    // event stop animation
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))
        action = ACTION_STOP;

    /* ... */

    // if an action was taken by the user, broadcast it to all other
        processes.
    if (action != NO_ACTION)
        for (int i = 1; i < n_ranks; i++)
            MPI_Send(&action, 1, MPI_CHAR, i, TAG_USER_ACTION,
                MPI_COMM_WORLD);
}
```

D'autre part, le processus de rang 1 écoute via la commande **MPI_Iprobe** si le processus racine (=0) a envoyé une action utilisateur (en utilisant la balise TAG_USER_ACTION comme référence). Une fois que MPI_Iprobe détecte qu'il y a des messages d'action utilisateur à recevoir, la commande **MPI_Recv** est utilisée pour enregistrer l'action utilisateur et agir en conséquence. Le code est illustré ci-dessous :

Listing 3: Process 1 - MPI Receive user action

```
while (action != ACTION_EXIT)
{
    bool advance = false;
    // checking if process 0 has sent messages
    MPI_Iprobe(0, TAG_USER_ACTION, MPI_COMM_WORLD, &flag, &status);
    if (flag)
    {
        // reading keyboard pressed
        MPI_Recv(&action, 1, MPI_CHAR, 0, TAG_USER_ACTION,
            MPI_COMM_WORLD, &status);
        switch (action)
        {
        case ACTION_PLAY: // play
            animate = true;
            break;
        case ACTION_STOP: // stop
            animate = false;
            break;
    /* ... */
```

De même, le processus 0 écoute le processus 1 pour recevoir les calculs effectués et mettre à jour les variables **cloud**, **grid** et **vortices** qui sont utilisées pour être affichées à l'écran. Le processus 0 utilise également la commande **MPI_Iprobe** pour vérifier s'il y a des données à recevoir du processus 1. L'échange de données se fait en utilisant la communication par **MPI_Gather** comme le montre le code ci-dessous :

Listing 4: Process 1 - MPI Gather - Send calculated data

```
if (my_rank == 1)
    MPI_Send(vortices.data(), 3 * vortices.numberOfVortices(),
        MPI_DOUBLE, 0, TAG_DATA, MPI_COMM_WORLD);
MPI_Gatherv(&cloudData[cloudPos], 2 * cloudNumberOfPoints,
    MPI_DOUBLE, NULL, NULL, NULL, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Gatherv(&gridData[gridPos], 2 * gridNumberOfPoints, MPI_DOUBLE,
    NULL, NULL, NULL, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Listing 5: Process 1 - MPI Gather - Send calculated data

```
MPI_Iprobe(1, TAG_DATA, MPI_COMM_WORLD, &flag, &status);
if (flag)
{
    MPI_Recv(vortices.data(), vortices.numberOfVortices() * 3,
        MPI_DOUBLE, 1, TAG_DATA, MPI_COMM_WORLD, &status);

    MPI_Gatherv(NULL, 0, MPI_DOUBLE, cloud.data(), cloudRecvcounts,
        cloudDispls, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    MPI_Gatherv(NULL, 0, MPI_DOUBLE, grid.data(), gridRecvcounts,
        gridDispls, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* ... */
```

2.3 Résultats

Après avoir terminé la première partie de la parallélisation, nous avons mesuré le nouveau FPS et constaté une différence de 30 FPS à 40 FPS, ce qui se traduit par une *speedup* de **1,33**.

3 Parallélisation en Mémoire Partagée

3.1 L'implementation du code

Après avoir réalisé la séparation de l'interface graphique et de la partie calcul à l'aide de la bibliothèque MPI, il a été recherché des moyens d'accélérer encore plus l'exécution du programme à partir de la parallélisation des boucles pour la partie calcul de la position et de la vitesse des particules, réalisée via l'interpolation Runge Kutta. Pour cela, il a été utilisé l'instruction suivante :

Listing 6: C++ OpenMP

```
#pragma omp parallel for
```

de la bibliothèque OpenMP qui permet la création de plusieurs threads pour l'exécution concurrente de la boucle for dans chaque processus créé.

Dans le fichier "runge-kutta.cpp", dans lequel se trouvent les appels aux fonctions de calcul, les méthodes mises en œuvre ont été analysées afin de vérifier si elles pouvaient être parallélisées ou non. En termes d'interdépendance des variables, il a été conclu que la seule boucle for de la fonction "solve-RK4-fixed-vortices" et les 3 boucles de la fonction "solve-RK4-movable-vortices" étaient parallélisables. Cependant, en calculant le temps de calcul de chacune d'entre elles, les résultats suivants ont été obtenus :

solve-RK4-fixed-vortices	
Loop For	Time of Computing (s)
1	0.100357

Table 1: Computing time Loops For

solve-RK4-mouvable-vortices	
Loop For	Time of Computing (s)
1	0.043164
2	0.00000182
3	0.000000068

Table 2: Computing time Loops For

De cette façon, il est devenu évident, à la fois par la complexité de la parallélisation et, principalement, par le temps d'exécution de chaque boucle, que les boucles à paralléliser dans notre programme étaient les deux premières de chacune des méthodes mentionnées ci-dessus, puisqu'elles présentent un temps de calcul séquentiel considérablement plus significatif et, par conséquent, justifient une parallélisation.

Ainsi, nous avons obtenu les résultats suivants après l'approche du multithreading :

solve-RK4-fixed-vortices.cpp		
Loop For	Time of Computing (s)	Speed-Up
1	0.0678375	1.4793735

Table 3: Computing time Loops For (with OpenMP)

solve-RK4-mouvable-vortices		
Loop For	Time of Computing (s)	Speed-up
1	0.0237518	1.817294

Table 4: Computing time Loops For (with OpenMP)

Il convient de préciser que ces résultats sont locaux, c'est-à-dire qu'ils ne concernent qu'un seul ordinateur. Il n'a pas été possible de réaliser l'expérience de parallélisation des boucles d'affichage, car il n'était pas possible d'utiliser une autre machine.

3.2 Résultats

Après avoir terminé la deuxième partie de la parallélisation avec OpenMP, nous avons mesuré le nouveau FPS et constaté une différence de 40 FPS à 60 FPS, ce qui se traduit par une **speed-up de 1.5** par rapport à première partie et un **speed-up de 2** par rapport au programme original sans aucune parallélisation.

4 Parallélisation en Mémoire Distribuée et Partagée des calculs

Dans cette phase du projet, nous avons rassemblé les connaissances appliquées jusqu'à présent pour essayer de gagner encore plus en performance. Par conséquent, en plus de créer plusieurs threads pour exécuter des parties du programme, nous avons parallélisé encore plus les mêmes boucles pour, à travers une implémentation multiprocessus. Nous avons créé une distribution équilibrée de la charge de travail pour chaque processus de sorte que, tant dans le calcul de la grille que dans le calcul du nuage, c'est-à-dire les calculs de déplacement et de vitesse de chaque particule, chaque processus avait une charge de travail très similaire, comme proposé dans l'énoncé.

Tout d'abord, nous avons modifié l'en-tête des fonctions afin d'indiquer pour chaque méthode le nombre de processus et le rang de chacun d'entre eux. À partir de là, la charge de travail a été répartie, pour chaque processus, de manière pratiquement égale, comme suit :

Listing 7: Headline of solve-RK4-fixed-vortices

```
Geometry::CloudOfPoints
Numeric::solve_RK4_fixed_vortices(double dt, CartesianGridOfSpeed
    const &t_velocity, Geometry::CloudOfPoints const &t_points, int
    myrank, int nranks){
    constexpr double onesixth = 1. / 6.;
    using vector = Simulation::Vortices::vector;
    using point = Simulation::Vortices::point;

    Geometry::CloudOfPoints newCloud(t_points.numberOfPoints());
    // On ne bouge que les points :
    std::size_t begin = myrank * t_points.numberOfPoints() / nranks;
    std::size_t end = (myrank + 1) * t_points.numberOfPoints() /
        nranks;
    if (myrank == nranks - 1) end = t_points.numberOfPoints();
    // #pragma omp parallel for
    for (std::size_t iPoint = begin; iPoint < end; iPoint++)
    {
        point p = t_points[iPoint];
        vector v1 = t_velocity.computeVelocityFor(p);
        point p1 = p + 0.5 * dt * v1;
        p1 = t_velocity.updatePosition(p1);
        vector v2 = t_velocity.computeVelocityFor(p1);
        point p2 = p + 0.5 * dt * v2;
        p2 = t_velocity.updatePosition(p2);
```

```
22          vector v3 = t_velocity.computeVelocityFor(p2);
23          point p3 = p + dt * v3;
24          p3 = t_velocity.updatePosition(p3);
25          vector v4 = t_velocity.computeVelocityFor(p3);
26          newCloud[iPoint] = t_velocity.updatePosition(p + onesixth *
                dt * (v1 + 2. * v2 + 2. * v3 + v4));
27      }
28      return newCloud;
29  }
```

De cette manière, chaque processus reçoit la même quantité de données et n'effectue le calcul que pour une petite partie de la matrice, ce qui améliore considérablement le temps d'exécution comme on le verra a posteriori. La parallélisation à l'aide de MPI a été faite exactement de la même manière pour la fonction solve-RK4-movable-vortices.

Il est important de souligner que nous avons observé une meilleure performance en supprimant le "parallel for" (openmp) du code, une fois que, grâce au coût de communication entre les différents threads et aux éventuels conflits de cache, l'exécution de plusieurs processus avec plusieurs threads chacun laissait les coûts totaux très élevés, ne compensant pas son application. Ci-dessous un graphique représentant les performances obtenues en FPS en fonction du nombre de processus :

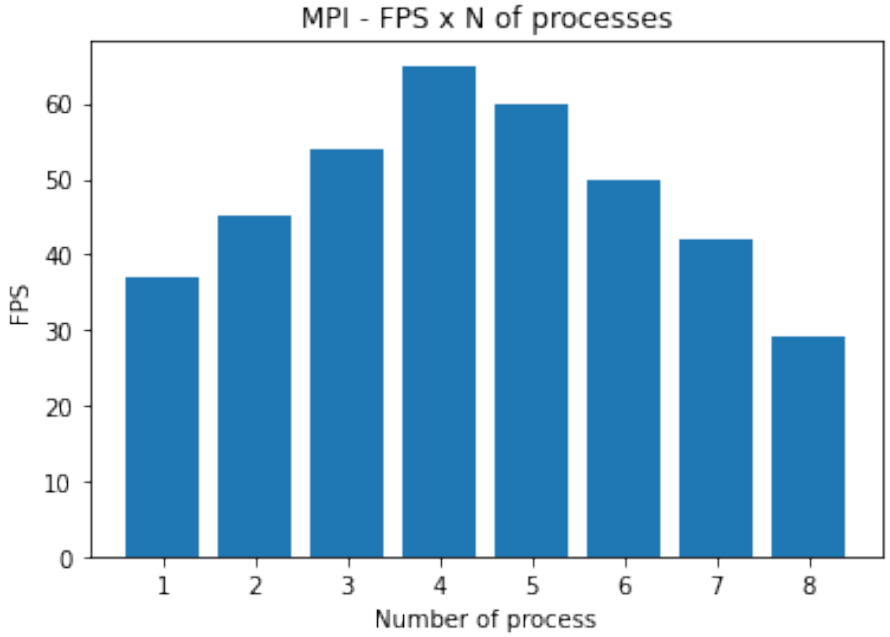

Figure 1: MPI - FPS x N

5 Réflexions sur l'approche mixte Eulérienne-Lagrangienne

La parallélisation MPI du calcul du champ de vitesse est une tâche complexe lorsqu'elle est réalisée par une approche mixte Eulérienne-Lagrangienne. Pour paralléliser cette tâche, il est nécessaire de diviser le domaine de calcul en plusieurs sous-domaines qui seront traités par différents processeurs. Cependant, la parallélisation des sous-domaines basée sur une approche Eulérienne peut entraîner des problèmes de communication entre les différents processus, car le calcul de la position d'une particule peut être traité dans un autre sous-domaine où se situe le champ de vitesse correspondant, ce qui entraîne la nécessité d'une communication entre les sous-domaines lors du calcul. Cette problématique peut être résolue en mettant en place une stratégie intelligente d'allocation de processus qui permettra à un grand nombre de processus d'être alloués autour du vortex, tandis que les régions éloignées des points d'initialisation du vortex auront un plus grand nombre de points par processus.

L'approche mixte Eulérienne-Lagrangienne peut également poser des problèmes dans le cas où le maillage est de très grande dimension et où le nombre de particules est important. En effet, si la dimension est trop grande, certaines zones du tore deviendraient statiques, ce qui entraînerait l'exécution de code inutile par certains processus, car il n'y a pas de mouvement de particules. Pour résoudre cette problématique, il est possible de mettre en place une stratégie d'allocation de processus intelligente en utilisant l'algorithme maître-esclave.

Dans le cas d'un grand nombre de particules, la probabilité que les particules comprises dans le voisinage d'un point spécifique suivent le même mouvement est très élevée. De cette façon, il ne serait pas nécessaire de calculer le mouvement de toutes les particules, mais seulement de quelques-unes, tout en considérant que leurs voisines suivent le même schéma de mouvement. Ainsi, en mettant en place cette stratégie, la tâche de calcul sera grandement simplifiée.

Enfin, dans le cas d'un maillage de très grande taille et d'un très grand nombre de particules, les deux stratégies expliquées précédemment s'appliquent. Cependant, il est important de noter que la mise en place de ces stratégies peut s'avérer complexe et nécessiter une grande expertise en informatique scientifique pour être correctement implémentée.

6 GitHub

Le lien pour pouvoir accéder au projet dans notre repository GitHub est :
/AndreWerneck/OS202/ProjetEnsta2023