OS202 - Systèmes parallèles et distribués
Modélisation de tourbillons sur un tore en deux dimensions

Arthur COELHO RUBACK
Joaquim MINARELLI GASPAR
Matheus SANTOS SANO

Palaiseau, France

Mars 2023

Table des matières

1 Séparation interface-graphique et calcul **2**

2 Parallélisation en mémoire partagée **3**

3 Parallélisation en mémoire distribuée et partagée des calculs **4**

4 Réflexions sur l'approche mixte Eulérienne–Lagrangienne **5**

Introduction

Toutes les codes on été exécutes sur un ordinateur de configuration affiché en suite :
— Architecture : x86_64
— On-line CPU(s) list 0-7
— 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
— Thread(s) per core : 2
— Core(s) per socket : 4
— CPU max MHz : 4700,0000
— CPU min MHz : 400,0000
Les codes sont dans le GitHub dans le lien suivante :
https ://github.com/arthur-ruback/OS202

1 Séparation interface-graphique et calcul

Pour paralléliser le code, on commence par séparer l'interface graphique du calcul de la vitesse et le déplacement des particules et des tourbillons. Le processus 0 est responsable de l'affichage à l'écran, tandis que le processus 1 est responsable de la gestion des événements (claviers et fermeture de la fenêtre). À chaque fois qu'une commande est saisie (P, S, Up, Down), le processus 0 envoie un message au processus 1 indiquant quelle commande doit être exécutée avec la fonction "**MPI_Send**".

Le système a été conçu suivant une logique maître-esclave et donc le processus 0 envoye une demande "next frame" au processus 1 pour qu'il calcule. Par conséquence le processus 1 sera synchronisé au processus 0, ayant une phase de réception de commande, une phase de action (calcul) et une phase d'envoi de réponse.

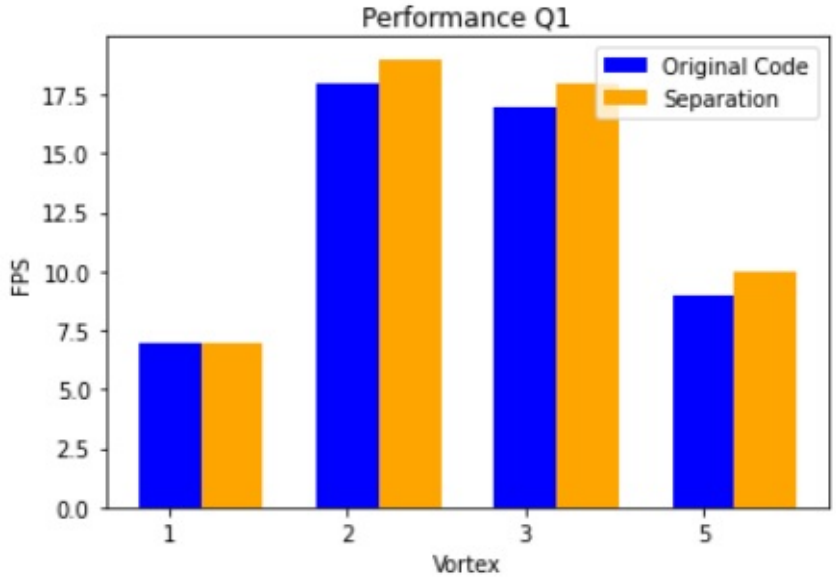
1. close the window and finish the other process ;
2. play animation ;
3. stop animation ;
4. advance step by step in time ;
5. half the time step ;
6. double the time step ;
7. next frame ;

Si la commande est de lire l'animation ou d'avancer l'animation étape par étape, le processus 1 effectue les calculs et envoie les données de cloud, du grid et des vortices au processus 0 à l'aide de la fonction "MPI_Send". Il convient de noter que la taille des données du cloud et du grid doit être deux fois supérieure au nombre de points, car les vecteurs des deux sont en 2D. Si le processus 1 est terminé, il envoie un message au processus 0 pour informer qu'il doit finir.

Après une analyse judicieuse, il a été constaté que la fonction *solve_RK4_fixed_vortices()* et *solve_RK4_movable_vortices()* étaient égales au début, en effet la fonction pour les tourbillons fixes appartient à celle mobile. Nous avons donc séparé la fonction mobile en : *solve_RK4_fixed_vortices()* et *updateVortices()*, la partie responsable pour la mise a jour du champs de vitesse et de la position des tourbillons.

En ce qui concerne le calcul du FPS, comme le code fourni par le professeur réalise l'affichage à l'écran et la gestion des événements de manière séquentielle, nous considérons un nouveau frame à chaque fois que le processus 0 reçoit les données du processus 1. Ainsi, on considère un frame lorsque l'écran est mis à jour.

Le graphe suivant montre l'accélération avec la séparation en deux taches décrit avant.



2 Parallélisation en mémoire partagée

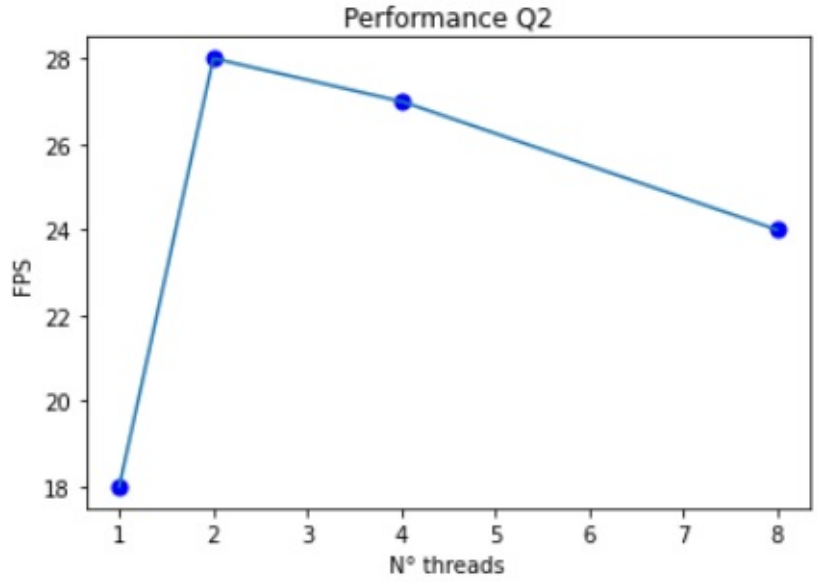
Il faut maintenant paralléliser les calculs à l'aide de la bibliothèque OpenMP, en utilisant le code suivant avant le début des boucles for plus coûteuses en calcul :

```
1 #pragma omp parallel for
```

Pour cela, on doit vérifier quels boucles prennent le plus de temps. Ainsi, une minuterie a été placée pour mesurer le temps d'exécution des boucles, en concluant que la plus coûteuse est la boucle de calcul de la position des particules.

De cette manière, cette boucle a été parallélisée avec OpenMP, et les performances en fonction du nombre de threads sont visibles sur la figure suivante :

FIGURE 1 – Performances pour différents nombres de threads



Nous pouvons voir que les meilleures performances ont été obtenues avec 2 threads. Dans ce cas, il y a eu une amélioration des performances de 60%..

3 Parallélisation en mémoire distribuée et partagée des calculs

Pour paralléliser la partie calcul dans plusieurs threads, le vecteur position a été divisé entre size-1, puisque le thread 0 n'effectue pas de calculs. De cette façon, comme nous devrons envoyer les données partitionnées de tous les threads au processus 0, qui est responsable de la mise à jour de l'écran, la fonction gatherv a été utilisée. Cette fonction présente en entrée le nombre de données qui seront envoyées par chaque thread (appelées recvcounts) et un décalage, qui est un vecteur qui indique le début et la fin de chaque donnée de processus. La déclaration de ces variables s'est faite comme suit :

```
// taille nuage de points partiel pour chaque processus
    int partialCloudSize = cloud.numberOfPoints() / (size-1);
    int offsets[size], recvcounts[size];
    offsets[0] = 0;
    recvcounts[0] = 0;
    for (int i = 1; i < size; i++)
    {
        recvcounts[i] = 2* partialCloudSize;
        offsets[i] = 2 * partialCloudSize * (i-1);
    }
```

Notez que les deux vecteurs ont été initialisés avec 0 pour le processus 0, car il ne doit effectuer aucun calcul. Le décalage a également une valeur de 0 pour le processus 1, car le processus 1 doit avoir les premières données de vecteur de nuage, il doit donc avoir un décalage nul.

De plus, un constructeur de "CloudOfPoints" a été déclaré pour chaque thread différent de 0, avec le nombre de points que chaque thread doit travailler. Cela entraîne la limitation du domaine de chaque thread par "partialCloudSize". Cette déclaration a été faite comme suit :

```
Geometry::CloudOfPoints processusCloud(partialCloudSize);
```

Ainsi, chaque fois que les threads terminent les calculs pour leurs ensembles de points spécifiques, ils envoient les données au processus 0 via le gatherv :

```
MPI_Gatherv(processusCloud.data(), 2* partialCloudSize, MPI_DOUBLE,
   cloud.data(), recvcounts, offsets,MPI_DOUBLE,0 , MPI_COMM_WORLD);
```

Le calcul de la gride et des tourbillons n'étant pas directement parallélisés, leurs valeurs sont envoyées au processus 0 par le processus 1, via le MPI_Send suivante (il est important de noter qu'un tel envoi n'est effectué que lorsque le programme exécute une situation de vortex mobile , puisque dans le cas d'un vortex fixe le champ de vitesse et la position du vortex ne sont pas mis à jour) :

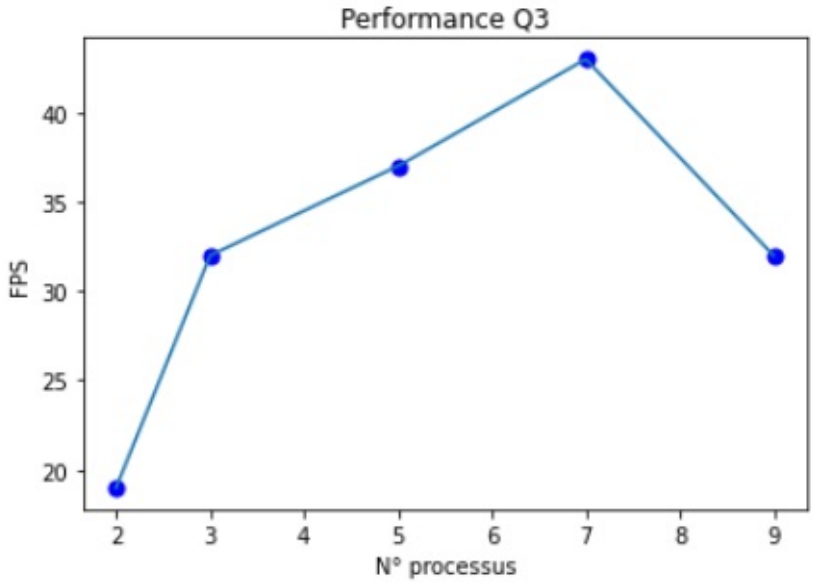
```
MPI_Send(grid.data(), 2 * grid.numberOfPoints(), MPI_DOUBLE, 0,
   FLAG_GRID, MPI_COMM_WORLD);
MPI_Send(vortices.data(), 3 * vortices.numberOfVortices(),
   MPI_DOUBLE, 0, FLAG_VORTICE, MPI_COMM_WORLD);
```

Le choix de ne pas paralléliser le calcul de gride a été fait en raison de la complexité de la façon dont le calcul de position est effectué et aussi du fait qu'il ne prend pas beaucoup de temps a calculer. Pour calculer la position suivante de chaque particule de nuage, le vecteur vitesse est utilisé. Mais à terme, avec le mouvement des particules, le vecteur vitesse se positionnerait en dehors du domaine des points de processus, ce qui rendrait nécessaire la transmission de cette information entre les processus respectifs. Comme le calcul de la gride pour la boucle n'est pas

coûteux en calcul, faire en sorte que tous les processus aient cette information n'affecte pas de manière significative les performances de calcul.

En utilisant les données avec 2 vortex en entrée, les résultats du programme parallèle pour différents nombres de processus peuvent être vus dans la figure suivante.

FIGURE 2 – Performances pour différents nombres de processus



Notez que pour plus de 7 processus, les performances du programme chutent. Cela se produit en raison des configurations de l'ordinateur utilisé, car dans ce cas, plus d'un thread effectue des calculs sur un processeur, ce qui fait que le programme perd plus de temps à changer de thread qu'à effectuer des calculs.

Enfin, si l'on compare la valeur de performance initiale du programme (17,5 FPS) avec la valeur finale pour le meilleur nombre de processus (47 FPS), on observe qu'il y a une accélération de 168%.

4 Réflexions sur l'approche mixte Eulérienne–Lagrangienne

La parallélisation MPI du calcul du champ de vitesse par une approche par sous-domaine nécessite de diviser le domaine de calcul en plusieurs sous-domaines, qui seront traités par différents processeurs. Cependant, la parallélisation des sous-domaines (qui est basée sur une approche eulérienne) peut générer des problèmes de communication entre différents processus. Cela se produit parce qu'éventuellement le calcul de la position d'une particule d'un fil "a" aura son champ de vitesse situé dans un autre fil, ce qui impliquera le besoin d'une communication entre les fils au moment du calcul.

Si cette stratégie est mise en place, 2 principales situations potentiellement problématiques peuvent survenir :

Très grande dimension : si la dimension est trop grande, selon le nombre de vortex, certaines zones du tore deviendraient statiques. Cependant, le programme continuerait à effectuer la même quantité de calculs pour ces régions, ce qui entraînerait l'exécution de code pas nécessaire par ces processus, car il n'y a pas de mouvement de particules.

De cette manière, nous pourrions utiliser une stratégie d'allocation de processus intelligente afin qu'un grand nombre de processus soient alloués autour du vortex, et les régions extrêmement éloignées des points d'initialisation du vortex aient un plus grand nombre de points par

processus, car peu de calculs seraient nécessaires. Un exemple d'algorithme qui pourrait être mis en œuvre pour gérer cette situation est le maître-esclave.

Grand nombre de particules : lorsqu'il y a un grand nombre de particules, la probabilité que les particules comprises dans le voisinage d'un "x" spécifique suivent le même mouvement de x est très élevée. De cette façon, il ne serait pas nécessaire de calculer le mouvement de toutes les particules, mais de n'en calculer que quelques-unes et de considérer que leurs voisines suivent le même schéma de mouvement.

Dans ce cas ou il y a une grande taille et grand nombre de particules, les deux stratégies expliquées précédemment s'appliquent.