OS202 Rapport de Projet

1. Séparation Interface-graphique et Calcul

Pour séparer l'affichage de calculs, j'ai utilisé 2 processus, P[0] pour l'affichage et P[1] pour les calculs.

```
#define   P_AFFICHAGE   0
#define   P_CALCUL   1
```

P[0] s'occupe de l'affichage et la gestion des évènements. C'est-à-dire que, en plus des anciennes actions, à chaque itération de boucle de myScreen, P[0] doit:

- s'il y a un évènement de Closed, envoyer un signal à P[1], et fermer myScreen.
- s'il y a un évènement de KeyBoard::Upou KeyBoard::down, envoyer un signal à P[1], et modifier chez lui dt.
- envoyer un signal contenant la valeur de isMobile à P[1].
- recevoir les données de cloud (et vortices) depuis P[1] lorsque isMobile=0 (=1), et renouveler les données chez lui.

Différentes valeurs ont été définies pour des signaux différents (Closed, Up, Down, isMobile). Et les fonctions MPI_Send(), MPI_Recv() sont utilisé pour envoyer/recevoir les signaux et les données.

Le processus de calcul P[1] doit:

- attendre les signaux envoyés par P[0] et faire des opérations selon la valeur du signal. (Closed: sortir du boucle; Up: dt*=2; Down: dt/=2; isMobile=0/1: utiliser différents function pour les calculs).
- À chaque itération, si P[1] a fait des calculs, P[1] envoie les données vers P[0].

2. Parallélisation en mémoire partagé

Pour les vortex immobiles, il faut juste modifier dans runge_kunta.cpp les fonctions de calculs solve_RK4_fixed_vortices et solve_RK4_movable_vortices pour des vortex fixes ou mobiles respectivement.

Dans solve_RK4_fixed_vortices, la boucle de calcul des points de cloud:

```
#pragma omp parallel for
for ( std::size_t iPoint=0; iPoint<t_points.numberOfPoints(); ++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);
    vector v3 = t_velocity.computeVelocityFor(p2);
    point p3 = p + dt*v3;
    p3 = t_velocity.updatePosition(p3);
    vector v4 = t_velocity.computeVelocityFor(p3);
    newCloud[iPoint] = t_velocity.updatePosition(p + onesixth*dt*(v1+2.*v2+2.*v3+v4));
}
return newCloud;
```

Dans solve_RK4_fixed_vortices, la boucle de calcul des points de cloud et celle de des

vitesses de points sont parallélisés. (Le nombre de vortex est petit donc la parallélisation de leur calculs n'est pas très intéressant.)

```
    // On ne bouge que les points :
#pragma omp parallel for
    for ( std::size_t iPoint=0; iPoint<t_points.numberOfPoints(); ++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);
        vector v3 = t_velocity.computeVelocityFor(p2);
        point p3 = p + dt*v3;
        p3 = t_velocity.updatePosition(p3);
        vector v4 = t_velocity.computeVelocityFor(p3);
        newCloud[iPoint] = t_velocity.updatePosition(p + onesixth*dt*(v1+2.*v2+2.*v3+v4));
    }
    std::vector<point> newVortexCenter;
    newVortexCenter.reserve(t_vortices.numberOfVortices());
#pragma omp parallel for
    for (std::size_t iVortex=0; iVortex<t_vortices.numberOfVortices(); ++iVortex)
    {
        point p = t_vortices.getCenter(iVortex);
        vector v1 = t_vortices.computeSpeed(p);
```

Pour mesurer la performance, on utilise le FPS. On mesure le moyen de FPS pour 100 cycle de calcul (qui commence dès qu'on presse sur 'P').

En faisant variant la valeur de OMP_NUM_THREADS de 1 à 4, on observe les résultats suivants:

n\fps	simSimu	tripV	manyV	Corn	One
ref	9.68303	9.09388	6.98237	23.0205	4.70848
1	10.0992	10.5248	6.60373	21.7617	5.08283
2	12.1348	10.0432	7.52642	17.3694	5.54322
3	12.7627	12.1247	8.07613	20.592	5.76173
4	11.3234	11.5826	8.1218	23.1587	6.1377

On trouve qu'avec openmp, il y a une petite amélioration.

3. Parallélisation en mémoire distribuée et partagée les calculs

Pour pouvoir paralléliser, on construit une nouvelle fonction `sub_solve_RK4_vortices_new_cloud()` qui faire uniquement le calcul des particule. De plus, il prend en entrée son rang, le nombre de particule chez lui et fait la partie de calcul attribué à lui. Pour le cas de vortex immobile, cette fonction suffit. Alors pour chaque processus de calcul il faut:

```
subCloud = sub_solve_RK4_vortices_new_cloud(dt, grid, cloud, rank, subnbOfPts0, recvcountsCloud[rank]);
MPI_Allgatherv(subCloud.data(), recvcountsCloud[rank], MPI_DOUBLE, newCloudData.data(), recvcountsCloud,
displsCloud,  MPI_DOUBLE, global);
```

On utilise Allgatherv() pour récupérer les données de cloud à chaque itération après la fin

de son calcul. Après l'appel à Allgatherv(), tous les processus incluant celui d'affichage obtient les données de cloud. P[0] peut commencer son affichage.

Allgatherv() permet de prend en considération que la répartition des données entre les processus de calcul peut être inéquitable et que le processus d'affichage n'envoye aucune donnée.

Pour le cas de vortex mobile, j'ai crée une autre fonction solve_RK4_movable_vortices_update_vortices() qui fait tout le reste sauf la partie de mise à jour de vitesse des particules. La mise à jour de vitesse des particule est complété en appelant directement la fonction updateVelocityField(t_vortices); Pour chaque processus il faut:

```
subCloud = sub_solve_RK4_vortices_new_cloud(dt, grid, cloud, rank, subnbOfPts0, recvcountsCloud[rank]);
MPI_Allgatherv(subCloud.data(), recvcountsCloud[rank], MPI_DOUBLE, newCloudData.data(), recvcountsCloud,
displsCloud,  MPI_DOUBLE, global);
solve_RK4_movable_vortices_update_vortices(dt, grid, vortices, cloud);
grid.updateVelocityField(vortices);
```

Après l'appel à Allgatherv(), tous les processus incluant celui d'affichage obtient les données de cloud. Les processus de calcul l'utilise pour des calcul de vortex et de vitesse.

Tous les processus de calcul le font en même temps, et après le processus de rang=1 (P[1]) se charge d'envoyer les données de vortices (vortex) et de grid (vitesse) à P[0].

n\fps	simSimu	tripV	manyV	Corn	One
1	8.13276	8.27294	4.3674	18.7728	3.71479
2	14.9498	8.78587	7.79518	34.0823	3.33834
3	16.0047	11.7766	8.56603	50.5372	3.02141

Ici, n représente le nombre de processus de calcul, ça veut dire qu'il y a (n+1) processus ensemble. Nous pouvons voir que dans tous les cas il y a une augmentation de FPS. Pour les problèmes où il y a moins de données (cornertest, n=4000), l'augmentation de performance est plus importante. Au contraire, pour onevortexsimulation (n=40000), l'augmentation est négligeable. C'est parce que je travaille sur un seule ordinateur.

4. Réflexions sur l'approche mixte Eulérienne–Lagrangienne

Pour l'approche eulégienne conjointe à l'approche lagrangienne, la grille de calcul est divisée en plusieurs sous-domaine (eulégien). Nous pouvons maintenir dans tous les processus un tableau de status des particules qui montre à quelle sous-domaine appartient-ils. À chaque itération, on collecte tous les paricules qui partent dans un autre domaine, nous pouvons utiliser MPI_Bcast() pour mettre à jour la nouvelle status de ces particule dans tous les sous-domaines.

Or, la méthode de l'approche lagrangienne nous exige de partitionner les particules et des vortex sur des noeuds différents, selon quelle sous-domain il appartient. Donc il faut aussi une méthode de trouver à partir d'une particule donné à quel sous-domain it appartient.

De plus, si les particules et les vortex peuvent interagir entre eux, la partition en

sous-domain peut introduire des problèmes de calcul. Cette problème peut être solvé en utilisant

• Dans le cas d'un maillage de très grande dimension:

Ça deviendra plus facile de calculer en sous-domaine. Nous pouvons utiliser la méthode de eulégien.

• Dans le cas d'un très grand nombre de particules:

La mise à jour des status des particules sera génante. Nous pouvons utiliser la méthode purement lagrangien.

• Et dans le cas d'un maillage de très grande taille ET un très grand nombre de particules ?

Nous pouvons conjoindre les deux.