Problème à N corps - Attraction gravitationnelle

Formation MPI/OpenMP - Adrien Cassagne

1 Introduction

Le problème à N corps (*N-body* dans la littérature anglaise) est un problème assez classique de la mécanique de Newton : il consiste en la résolution des équations du mouvement de Newton. Ce problème peut cependant être généralisé et on le rencontre assez fréquemment dans la simulation numérique : cela fait de lui un bon cas d'école.

Au départ du problème (à l'instant t), pour chaque corps i, sa position $\vec{q_t}(i)$, sa masse m_i et sa vitesse $\vec{v_t}(i)$ sont connues. La force exercée entre deux corps i et j, à un instant t donné, est définie comme suit :

$$\vec{F_t}(i,j) = G.\frac{m_i.m_j}{(D_{i,j})^2}.\vec{u}, \tag{1}$$

avec G la constante gravitationnelle ($G = 6,67384 \times 10^{-11} m^3.kg^{-1}.s^{-2}$), $D_{i,j}$ la distance entre un corps i et un corps j et $\vec{u}$ le vecteur unitaire dirigé de i vers j. La force résultante pour un corps i donné, à un instant t donné, est la suivante :

$$\vec{F_t}(i) = \sum_{j \neq i}^{N} \vec{F_t}(i,j), \tag{2}$$

avec N le nombre de corps présents dans l'espace. L'accélération pour un corps i, à un instant t donné, est la suivante :

$$\vec{a_t}(i) = \frac{\vec{F_t}(i)}{m_i}. \tag{3}$$

La vitesse d'un corps i à un instant $t + dt$ dépend de la vistesse et de l'accélération à l'instant t :

$$\vec{v_{t+dt}}(i) = \vec{v_t}(i) + \vec{a_t}(i).dt. \tag{4}$$

Enfin, la position q d'un corps i à l'instant $t + dt$ dépend de la position, de la vitesse et de l'accélération à l'instant t :

$$\vec{q_{t+dt}}(i) = \vec{q_t}(i) + \vec{v_t}(i).dt + \frac{\vec{a_t}(i).dt^2}{2}. \tag{5}$$

Grâce aux équations 1, 2, 3, 4 et 5 il est possible de calculer la nouvelle position et la nouvelle vitesse pour tous corps i à l'instant $t + dt$.

1.1 Choix du pas de temps dt

Le plus simple est de choisir un pas de temps constant entre toutes les itérations cependant cela ne permet pas d'observer finement les intéractions surtout quand les corps se rapprochent de plus en plus. Nous avons donc fait le choix de calculer un nouveau pas de temps à chaque

itération pour qu'il soit adapté à la simulation. Par soucis de simplicité ce pas de temps dt sera commun à tout les corps du plan et devra respecter la condition suivante :

$$\|\vec{v_t}(i).dt + \frac{\vec{a_t}(i)}{2}.dt^2\| \leq 0.1 \times D_{i,j}, \tag{6}$$

avec j le corps le plus proche de i. Pour chaque corps i, un pas de temps sera calculé et le plus petit de ces pas de temps sera choisi. L'équation 6 traduit que la distance entre i_t et i_{t+dt} doit être inférieure à 10% de la distance entre i_t et j_t. Cela assure que deux masses quelconques ne se rapprochent pas plus de 20% entre les instants t et $t + dt$. Cependant, l'équation 6 n'est pas directement utilisable, elle donne lieu à la résolution d'un polynôme de degré 4 en dt. C'est pour cela que nous utilisons l'inégalité triangulaire qui permet de déterminer une nouvelle condition :

$$\|\vec{v_t}(i)\|.dt + \frac{\|\vec{a_t}(i)\|}{2}.dt^2 \leq 0.1 \times D_{i,j}. \tag{7}$$

L'équation 7 est de degré 2 ce qui est beaucoup plus raisonnable en temps de calcul.

1.2 Traduction algorithmique

D'un point de vue algorithmique, on calcule chaque corps en fonction de tous les autres. Cela se résume en l'imbrication de deux boucles `for` allant de 0 à N corps. Dans un langage `C-like` on peut l'exprimer de la façon suivante :

```
1 body b1[N]; // array which contains all bodies (at t)
2 body b2[N]; // empty array
3 for(unsigned int iBody = 0; iBody < N; ++iBody)
4   for(unsigned int jBody = 0; jBody < N; ++jBody)
5     if(iBody != jBody)
6       b2[iBody] = compute(b1[jBody]);
```

Listing 1 – Pseudo code illustrant un algorithme de type N corps

L'algorithme 1 montre une propriété très importante de cette classe de problème : pour N corps donnés, la complexité en terme de calcul est de $O(N^2)$. Il existe cependant une méthode permettant d'approximer et de résoudre le problème en $O(N \log N)$ mais nous ne l'utiliserons pas dans ce T.D. (voir simulation de Barnes-Hut).

2 Objectifs

Dans ce T.D. il est question d'utiliser les bases MPI et OpenMP acquises lors des précédents travaux et de les combiner sur un cas physique réel. Plus précisément nous allons :

1. simuler le déplacement de N corps dans le plan,
2. paralléliser les traitements d'un nœud avec OpenMP,
3. mettre en place un anneau de communication MPI,
4. paralléliser les traitements multi-nœuds avec MPI,
5. recouvrir les temps de communication par du calcul.

Nous allons effectuer une simulation dans le plan car il est relativement simple de la visionner sur un écran. Les corps seront répartis sur les processus MPI. Cela permet de traiter plus de corps que la limite mémoire d'un nœud le permet.

3 Préparation

Afin de ne pas perdre de temps, nous partirons d'un code séquentiel fonctionnel dans lequel toute la physique a déjà été implémentée. En premier lieu vous devez décompresser l'archive *nbody.tar.gz* dans votre */home/$USER* : `tar -xvvf nbody.tar.gz`. Vous possédez maintenant un dossier */home/$USER/nbody/* contenant les fichiers/dossiers suivants :

- *Makefile* : le *Makefile* permettant de compiler le projet,
- *bin/* : dossier contenant les exécutables,
- *data/* : dossier contenant les cas de test,
- *doc/* : dossier contenant des documents relatifs au projet (comme ce sujet),
- *obj/* : dossier contenant les fichiers compilés avant l'édition de liens,
- *objd/* : dossier contenant les fichiers compilés avant l'édition de liens (en mode debug),
- *reader/* : fichier source du code de visualisation des résulats d'une simulation,
- *src/* : dossier contenant le code source.

Pour compiler le code dans sa version séquentielle il suffit de taper la commande `make` et l'exécutable se construit dans *bin/nbody*. Le code source est écrit en `C 99` qui permet un peu plus de souplesse que le `C` traditionnel.

3.1 Les sources

3.1.1 Le point d'entrée : le `main`

Le point d'entrée du programme (fonction `main`) est contenu dans *src/main.c*. C'est dans ce fichier que sont définies les trois grandes étapes autour de la boucle des itérations (cf. list. 2) :

1. calculer l'accélération de tous les corps du plan `p`,
2. rechercher le plus petit pas de temps `dt` et le choisir pour tous les corps,
3. mettre à jour la position et la vitesse de tous les corps du plan.

```
1  void main()
2  {
3    double dt;
4    plan *p = createPlan(); // p contains the bodies
5    for(unsigned long iIte = 1; iIte <= NIterations; ++iIte)
6    {
7      computeAllLocalAcceleration(p);
8      dt = findLocalDt(p);
9      updateAllLocalPositionAndSpeed(p, dt);
10   }
11 }
```

Listing 2 – Code simplifié du calcul par itération

Les structures utilisées (ici `plan`) sont détaillées dans les sections suivantes.

3.1.2 Module `body`

Le module `body` (fichiers *src/body.c* et *src/body.h*) contient la structure `body` :

```
1 typedef struct
2 {
3   double mass; // body mass
4   double posX; // body position following x axis
5   double posY; // body position following y axis
6 } body;
```

Listing 3 – Structure `body`

Ainsi que la structure `localBody` :

```
typedef struct
{
  body   *b;                 // body mass and body position
  double speedX;             // body speed following x axis
  double speedY;             // body speed following y axis
  double accelerationX;      // body acceleration following x axis
  double accelerationY;      // body acceleration following y axis
  double closestNeighborLen; // contains the distance with the closest neighbor
} localBody;
```

Listing 4 – Structure `localBody`

Nous avons choisi d'utiliser deux structures (`body` et `localBody`) afin de décrire complètement les propriétés d'un corps. La structure `body` (cf. list. 3) contient la masse et la position du corps et la structure `localBody` (cf. list. 4) ajoute la vitesse et l'accélération. Cela est inutile pour le code séquentiel (on aurait pu mettre tous les champs dans une même structure). Cependant, quand vous implémenterez le code parallèle MPI, seules les propriétés de la structure `body` seront à communiquer aux autres nœuds. Les données supplémentaires de `localBody` sont uniquement nécessaires pour le calcul local à un processus.

3.1.3 Module `plan`

Le module `plan` (fichiers *src/plan.c* et *src/plan.h*) contient la structure `plan` :

```
typedef struct
{
  unsigned long nBody; // bodies number
  localBody     *lb;   // local body array values
} plan;
```

Listing 5 – Structure `plan`

Un plan (cf. list. 5) contient le nombre de corps `nBody` ainsi que les propriétés de chacun de ces corps dans le tableau `lb` (masse, position, vitesse et accélération).

3.1.4 Simuler un cas de test

Après avoir compilé le code (commande `make`), vous pouvez essayer de lancer un cas de test en exécutant le code comme suit :

```
> ./bin/nbody -f data/in/np1/in.testcase1 -i 100 -v -w
```

L'option `-f` permet de spécifier la racine d'un fichier d'entrée, l'option `-i` permet de choisir le nombre d'itérations à calculer, l'option `-v` active le mode verbose et enfin l'option `-w` permet d'écrire la solution à chaque itération dans *data/out/out.*.dat* (* correspond au numéro de l'itération).

Pour visualiser les solutions il faut compiler le `reader` avec la commande `make reader` et pour l'exécuter vous pouvez utiliser la commande suivante :

```
> ./bin/reader 800 600 data/out/out
```

Les deux premiers paramètres définissent la taille de la fenêtre de visualisation et le dernier paramètre permet de spécifier la racine des noms des fichiers à visualiser. Un fois la fenêtre de visualisation ouverte, appuyez sur la touche "flêche droite" pour lancer l'animation.

Si vous voyez deux corps s'éloigner puis se rapprocher alors le code fonctionne correctement et vous pouvez passer aux exercices de la section suivante.

4 Exercices

1. Paralléliser le code séquentiel initial avec OpenMP.

> **Aide :**
> Avant de foncer tête baissée, il est bon de refléchir un peu et de trouver où l'on passe le plus de temps : c'est cette zone qu'il faut essayer de paralléliser. Ici, il n'est pas question d'utiliser des primitives très complexes, un simple `#pragma omp parallel for schedule(runtime)` devrait suffir. Vous pourrez ensuite définir, à l'exécution du code, les variables d'environnement `OMP_NUM_THREADS` et `OMP_SCHEDULE` afin de trouver le meilleur compromis pour les performances.

2. Créer une structure MPI (`bodyMPI`) pour envoyer/recevoir des corps.

> **Aide :**
> La structure `body` qui est déclarée dans le fichier *src/body.h* ne fait pas partie des types de base que connaît MPI. Il faut donc déclarer une nouvelle structure MPI suivant les spécifications de la structure `body`. La création de structure MPI est assurée par la routine `MPI_Type_struct`. Une fois la structure MPI créée il faut nécessairement la déclarer avec la routine `MPI_Type_commit`. Avant de quitter le programme il faut penser à libérer la mémoire utilisée par la structure en appelant `MPI_Type_free`.

3. Mettre en place un anneau de communication MPI (voir fig. 1).

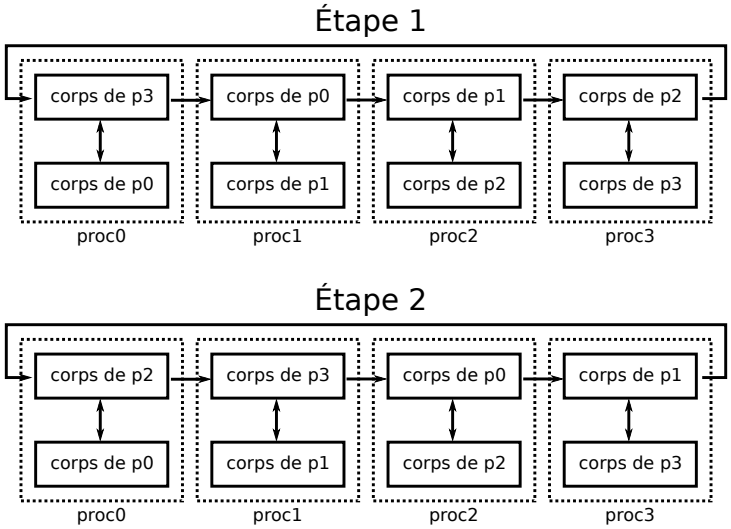

FIG. 1 – Anneau de communication pour 4 processus MPI

> **Aide :**
> Une itération est complète quand tous les processus MPI ont reçu les corps de tous les autres processus. Pour y parvenir il y a deux phases :
> — le calcul des accélérations des corps locaux entre eux (au sein d'un même processus),

— le calcul des accélérations des corps locaux en fonction des corps des autres processus.

Si il y a np processus MPI, alors il y a $np-1$ communications. Dans un premier temps, il faudra déterminer le rang des processus précédent et suivant dans l'anneau pour chaque nœud MPI. Ensuite, on utilisera un unique `buffer` pour envoyer et recevoir les corps. Avant chaque envoie de corps il faudra bien penser à copier les corps locaux dans le *buffer* MPI. Comme il est impossible d'envoyer et de recevoir en même temps (*buffer* MPI unique), il faudra déterminer qui envoie et qui reçoit en fonction du rang MPI. La réception et l'envoi peuvent être réalisés de façon bloquante avec les routines `MPI_Recv` et `MPI_Send`.

4. Trouver le plus petit pas de temps dt parmi tout les processus MPI et le choisir.

Aide :
À chaque itération, un nouveau pas de temps est calculé en fonction de la position des corps dans le plan. Dans la version parallèle à mémoire distrubuée il est impératif que tous les processus MPI aient le même dt. Pour y parvenir il faut utiliser une réduction de type minimum (voir `MPI_Allreduce`). Il doit maintenant être possible de lancer la simulation sur plusieurs nœuds.

5 Bonus

Si vous êtes arrivé ici c'est surement parce que vous êtes à l'aise et à partir de maintenant, le sujet sera volontairement moins précis. Si jamais vous rencontrez des difficultés n'hésitez pas à demander. L'objectif de cette partie est de traiter le recouvrement des communications par du calcul. Ce recouvrement est théoriquement possible dans le cas du prolème à N corps car il y a $O(N^2)$ calculs pour N données à échanger.

1. Implémenter le *double buffering* pour permettre un recouvrement calculs-communications (cf. fig. 2).

Aide :
Le problème de l'implémentation précédente est double, il n'est pas possible :
— d'envoyer et de recevoir des corps en même temps,
— d'envoyer et de recevoir des corps pendant que l'on calcul l'accélération avec d'autres corps.

Le *double buffering* permet d'éviter ce problème en utilisant deux *buffers* MPI au lieu d'un seul. Il est ainsi possible d'envoyer des corps à un processus même si ce dernier n'a pas terminé ses traitements. Attention, il faudra bien penser à inverser les *buffers* à chaque étape (au sein d'une même itération) pour que cela fonctionne : un coup on reçoit dans le `buffer0` et le coup d'après on reçoit dans le `buffer1`. Cette implémentation devrait déjà grandement améliorer les performances en supprimant des recopies de mémoire inutiles. Cependant le recouvrement calcul-communication n'est toujours pas possible à cause des communications bloquantes (`MPI_Recv` et `MPI_Send`). L'exercice suivant permet de lever cette limitation.

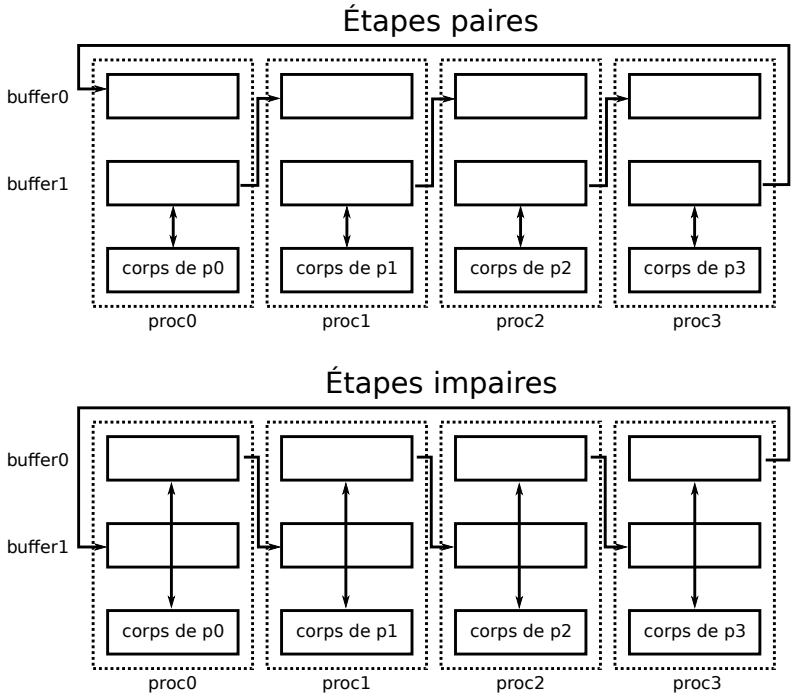

FIG. 2 – Anneau de communication pour 4 processus MPI avec *double buffering*

2. Utiliser les communications MPI non bloquantes.

> **Aide :**
> Les communications non bloquantes permettent d'envoyer et de recevoir des *buffers* pendant que l'on calcule. Voir les routines `MPI_Isend`, `MPI_Irecv` et `MPI_Wait`.

3. Utiliser les communications MPI persistantes.

> **Aide :**
> Les communications persitantes permettent de factoriser les temps d'initialisation des communications MPI. Soit t_1 le temps d'initialisation de l'envoi d'un *buffer*, t_2 le temps de l'envoi du *buffer* et t_3 le temps d'initialisation de la réception :
>
> $$t_{com} \simeq t_1 + t_2 + t_3$$
>
> Les communications persistantes permettent de diminuer fortement t_1 et t_3. Voir les routines `MPI_Send_init`, `MPI_Recv_init`, `MPI_Start` et `MPI_Wait`.

4. Utiliser les communications MPI persistantes collectives.

> **Aide :**
> Cela ne change pas grand chose en terme de performance mais permet de factoriser du code en fusionnant l'appel à l'envoi et à la réception des *buffers*. Voir les routines `MPI_Startall` et `MPI_Waitall`.