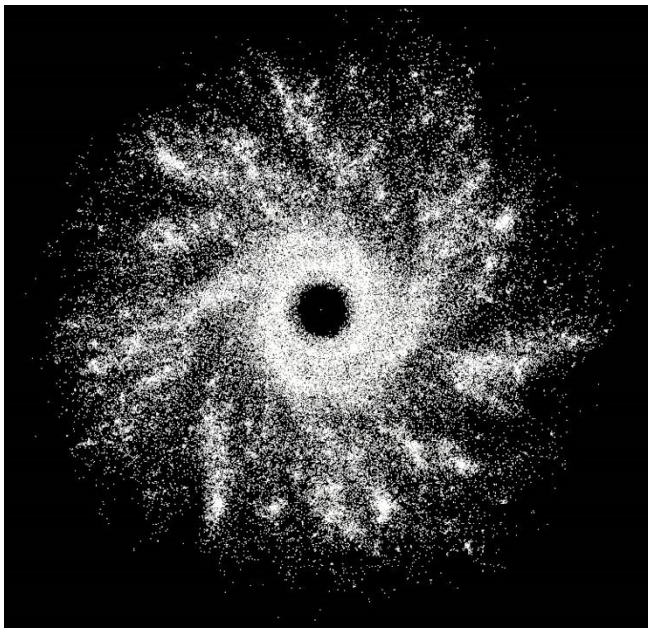
POLYTECH SORBONNE

Problème à N-corps : algorithmes et parallélisations



Auteur :
Rudio FIDA CYRILLE
Noah VEYTIZOUX
BEN-SOUSSEN
Elyas ASSILI
Camille HASCOET

Référent :
Roméo MOLINA

8 mai 2022

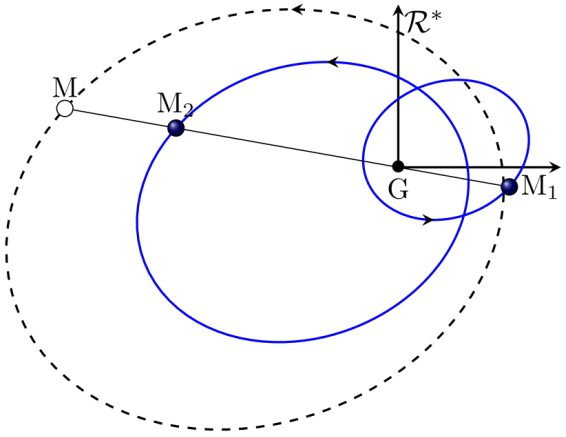
Table des matières

Chapitre 1

Présentation du projet

1.1 Sujet

Le problème a N-corps consiste à calculer le mouvement de N particules en connaissant leur masse et leur vitesse respective. Il s'agit donc de résoudre les équations du mouvement de Newton pour ces N particules. Le problème à N-corps est une simulation classique et importante pour l'astronomie, étant donné qu'il permet d'étudier la mécanique de corps céleste interagissant gravitationnellement. Le problème à deux corps et le problème à trois sont résolubles analytiquement. Pour N>3, il n'existe pas de résolution analytique, il faut donc utiliser des solutions approchées.



$$\begin{cases} a_1 = \dfrac{m_2}{m_1+m_2}a \\ a_2 = \dfrac{m_1}{m_1+m_2}a \end{cases}$$

FIGURE 1.1 – Solution du problème à 2 corps

1.2 Objectif : résolution la plus efficace possible du problème à N-corps

L'objectif du projet est donc de résoudre le plus efficacement possible le problème à n-corps afin de simuler par exemple des galaxies. Pour cela, nous utiliserons des solutions approchées calculées à partir de l'algorithme de Barnes-Hut. Nous étudierons également l'application de la parallélisation à notre programme. En pratique, le projet consiste à continuer et améliorer un projet déjà bien entamé afin d'acquérir de nouvelles compétences en C++, sur les algorithmes hiérarchiques , en parallélisation et plus généralement en optimisation de code.

1.3 Démarche

Un code C++ implémentant une interface graphique nous a été fourni pour le projet. Nous allons dans un premier temps calculer la force qui s'applique à chaque particules à partir de la formule de l'interaction gravitationnelle. La position se calcule alors en utilisant un intégrateur numérique (Méthode d'Euler, saute mouton...). Dans un deuxième temps, nous utiliserons l'algorithme de Barnes-Hut pour le calcul des forces. Enfin, nous utiliserons la parallélisation multi-thread pour accélérer nos calculs.

Chapitre 2

Aspect physique et mathématique du problème

Le problème consiste à calculer pour chaque particule la force exercée sur elle par toutes les autres à un instant t. Cela revient donc à résoudre les équations de Newton des N particules. La force prise en compte est ainsi l'interaction gravitationnelle d'un corps sur un autre.

Expressions des forces

Soit p_1 et p_2 deux particules, la force appliquée par p_2 sur p_1 est :

$$\vec{F}_{2\to1} = \frac{-Gm_1m_2}{\|\vec{p}_{2\to1}\|^2}\vec{p}_{2\to1} \tag{2.1}$$

où
— G est la constante de la gravitation $G = 6.672.10^{-11}Nm^2/kg^2$.
— m_1 est la masse de p_1.
— m_2 est la masse de p_2.

En pratique, on ne pourra pas appliquer directement cette formule. Ainsi, on en utilise une variante.
Posons $\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$ la position de la particule p_1 et $\begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$ celle de la particule p_2.
La distance d entre ces deux particules est alors donnée par la formule suivante : $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
Et la force appliquée par p_2 sur p_1 $F_{2\to1}$ est alors la suivante :
$F_{2\to1} = G * m_1 * m_2 * \begin{pmatrix} \frac{x_2-x_1}{d^3} \\ \frac{y_2-y_1}{d^3} \end{pmatrix}$ Cette formule est alors plus simple à implémenter.

Expression de l'accélération

On peut alors calculer l'accélération d'une particule avec la 2ème loi de Newton :

$$ma(t) = \Sigma\vec{F} = m\frac{dv}{dt} \tag{2.2}$$

où v est la vitesse de la particule, m sa masse et $\vec{F}$ les forces qui s'y appliquent.

Résolution numérique et calcul des positions

On obtient donc l'équation différentielle sur la position $OP_1(t)$:

$$\frac{d^2\vec{OP_1}(t)}{dt^2} = a(\vec{t}) \tag{2.3}$$

L'intégrateur permettra alors de résoudre numériquement et simplement cette équation différentielle qui serait compliquée à résoudre de manière exacte avec N particules pour un $N >= 3$.

Dans notre cas, nous utilisons d'abord une méthode d'Euler explicite afin de calculer la vitesse de chaque particule et ensuite sa position à chaque instant t.
Les vitesses s'obtiennent alors de la manière suivante :

$$\begin{cases} vx_{n+1} = vx_n + \Delta t * ax_n \\ vy_{n+1} = vy_n + \Delta t * ay_n \end{cases}$$

et les positions se retrouvent de la même manière :

$$\begin{cases} x_{n+1} = x_n + \Delta t * vy_n \\ y_{n+1} = y_n + \Delta t * vy_n \end{cases}$$

où Δt est le pas de discrétisation du temps.

Cependant, pour des problèmes de mécaniques, la méthode d'Euler n'est pas stable (divergence de trajectoire). Ainsi, il est préférable d'utiliser le schéma saute-mouton ("leap frog") qui est d'ordre 2 et conserve l'énergie mécanique des systèmes dynamiques.

Voici la forme de la version "Drift-kick-Drift" de la méthode :

$$\begin{cases} x_{n+\frac{1}{2}} = x_n + vx\frac{\Delta t}{2} \\ y_{n+\frac{1}{2}} = y_n + vy\frac{\Delta t}{2} \end{cases}$$

$$\begin{cases} vx_{n+1} = vx_n + ax_{n+\frac{1}{2}} \\ vy_{n+1} = vy_n + ay_{n+\frac{1}{2}} \end{cases}$$

$$\begin{cases} x_{n+1} = x_{n+\frac{1}{2}} + vx_{n+1}\frac{\Delta t}{2} \\ y_{n+1} = y_{n+\frac{1}{2}} + vy_{n+1}\frac{\Delta t}{2} \end{cases}$$

Le principe d'action-réaction

La 3ème loi de Newton appelée également principe d'action-réaction annonce que tout corps 1 exerçant une force sur un corps 2 subit une force de même intensité, de même direction mais de sens opposé, exercée par 2.

Cela se traduit ainsi : $\vec{F}_{2\to1} = -\vec{F}_{1\to2}$

Ce principe nous sera utile pour améliorer nos performances étant donné qu'il s'applique aux forces gravitationnelles.

Les paramètres physiques de la simulation

Les paramètres utilisés lors d'une simulation permettent de modifier le type de simulation et de résultat que l'on veut obtenir. Il est donc important de les maîtriser et de les connaître afin d'avoir des simulations cohérentes, notamment pour simuler des galaxies.

Chapitre 3

Analyse du code

Notre projet s'effectue dans la continuité d'un projet déjà bien entamé, ainsi, la première étape est de comprendre et d'assimiler le travail qui a déjà été effectué.

3.1 Vector.cpp Vector.h

Le fichier **Vector.cpp** permet la création de vecteurs. On y retrouve un constructeur pour les vecteur de deux dimensions et un deuxième pour les vecteurs de trois dimensions.

```
1  //Exemple d'un des constructeurs :vecteurs en 2 dimensions
2  Vec2D::Vec2D(double a_x, double a_y)
3          :x(a_x)
4          ,y(a_y)
5      {}
```

Le fichier Vector.h associé contient les différentes classes mettant en places les vecteurs.

3.2 Types.cpp - Types.h

Ce code ci possède des constructeurs nécessaires pour la mise en place des données liés aux particules. Nous avons :

- Un constructeur à vide des particules
- Un constructeur de particules avec une initialisation par 2 valeurs
- Un constructeur utilisant la structure d'une particule situé dans le Types.h
- Un constructeur servant d'overload, c'est-à-dire qu'elle spécifie plusieurs fonctions possédant le même nom
- Une fonction permettant la réinitialisation à 0 de nos données de particules
- Une fonction booléenne vérifiant si une donnée de particule est vide

Voici par exemple, le constructeur à vide de particules

```
1  \\Constructeur à vide de ParticleData */
2  ParticleData::ParticleData()
3     :m_pState(NULL)
4     ,m_pAuxState(NULL)
5  {}
```

Quand au fichier Types.h, on y retrouve différentes structures :

- **PODState** pour la position et la vitesse d'une particule
- **PODAuxState** pour la masse d'une particule
- **PODDeriv** pour la vitesse et l'accélération d'une particule
- **ParticleData** définissant une particule

3.3 ModelNBody.cpp - ModelNbody.h

Ce fichier contient les fonctions permettant la création de notre modèle à N-corps. Nous avons ainsi :

— Une classe permettant la création de notre modèle à N-corps. Il y figure un ensemble de donnée comme par exemple la masse du Soleil ou encore la constante gravitationnelle. Un destructeur de cette classe dans le but de libérer l'espace alloué par notre classe
— Des fonctions définissant la région d'étude,obtenant le temps des étapes, récupérant la région d'étude lors de la visualisation, ainsi que le centre de masse,calculant la vitesse orbitale d'une particule par rapport à une autre particule
— une fonction permettant d'obtenir la direction de la caméra et une autre sa position lors de la visualisation
— Des classes permettant la reintialisation du modèle à N-corps, initialisant la simulation du modèle, initialisant avec seulement 3 particules. Mais aussi des classes creant des black holes (=trou noir) accompagné de particules aléatoires, permettant la mise en place de l'arbre(cette notion d'arbre se clarifiera par la suite)
— Des fonctions obtenant le noeud racine de l'arbre, permettant de régler thêta et une autre classe le récupérant.Thêta est une variable qui va être utiliser lors de la mise en place de l'algorithme de Barnes-Hut
— Enfin une fonction booléenne qui permet de savoir lorsque la mise en place du modèle est terminé

Par exemple, voici le destructeur du modèle à N-corps

```
1  ModelNBody::~ModelNBody()
2  {
3     delete m_pInitial;
4     delete m_pAux;
5  }
```

Quant au ModelNbody.h , elle contient la classe permettant le traitement du modèle à N-corps

3.4 BHTree.cpp - BHTree.h

Ce fichier permet la création de fonction et de classe agissant sur un arbre. Un arbre est une structure contenant des informations(ici les particules) situés dans des éléments qu'on appelle noeuds.
On retrouve ainsi dans ce fichier la création et la reintilisation de l'arbre, des fonctions booléenne nous indiquant la racine (= noeud initial) de l'arbre, si un noeud est externe (=noeud sans descendant) ou encore si deux noeuds sont trop proche de l'un de l'autre. Des fonctions permettant d'obtenir les coordonnées du bord supérieur et inférieur d'un noeud, son centre de masse et sa variable thêta y sont définit.
S'y trouve aussi une fonction permettant de récupérer le nombre de particule situé dans les noeud et une autre nous indiquant le nombre des noeuds non assignés.
Une fonction calcule l'accélération causé par la gravitation d'une particule sur une autre et une autre calcule la force exercé par un noeud sur une particule.

Quand au BHTree.h, on y retrouve une classe implémentant l'arbre de Barnes-Hut avec un seul noeud. Par exemple dans cete classe, on retrouve plusieurs données situé dans un noeud comme le montre l'extrait de code suivant :

```
1    double m_mass;              // Masse de toutes les particules situés dans un noeud
2    Vec2D m_cm;                 // Centre de masse
3    Vec2D m_center;             // Centre d'un noeud
```

3.5 Les intégrateurs

Le code contient également un fichier dans lequel seront écrits les différents intégrateurs permettant de calculer les vitesses et positions de chaque particules. Pour l'instant, seul un simple intégrateur d'Euler est implémenté.

3.6 La visualisation

Des classes et méthodes gérant l'affichage des particules est aussi implémenté à partir des bibliothèques SDL et OpenGL, ce qui nous permet de pouvoir nous concentrer sur les calculs qui doivent être effectués.

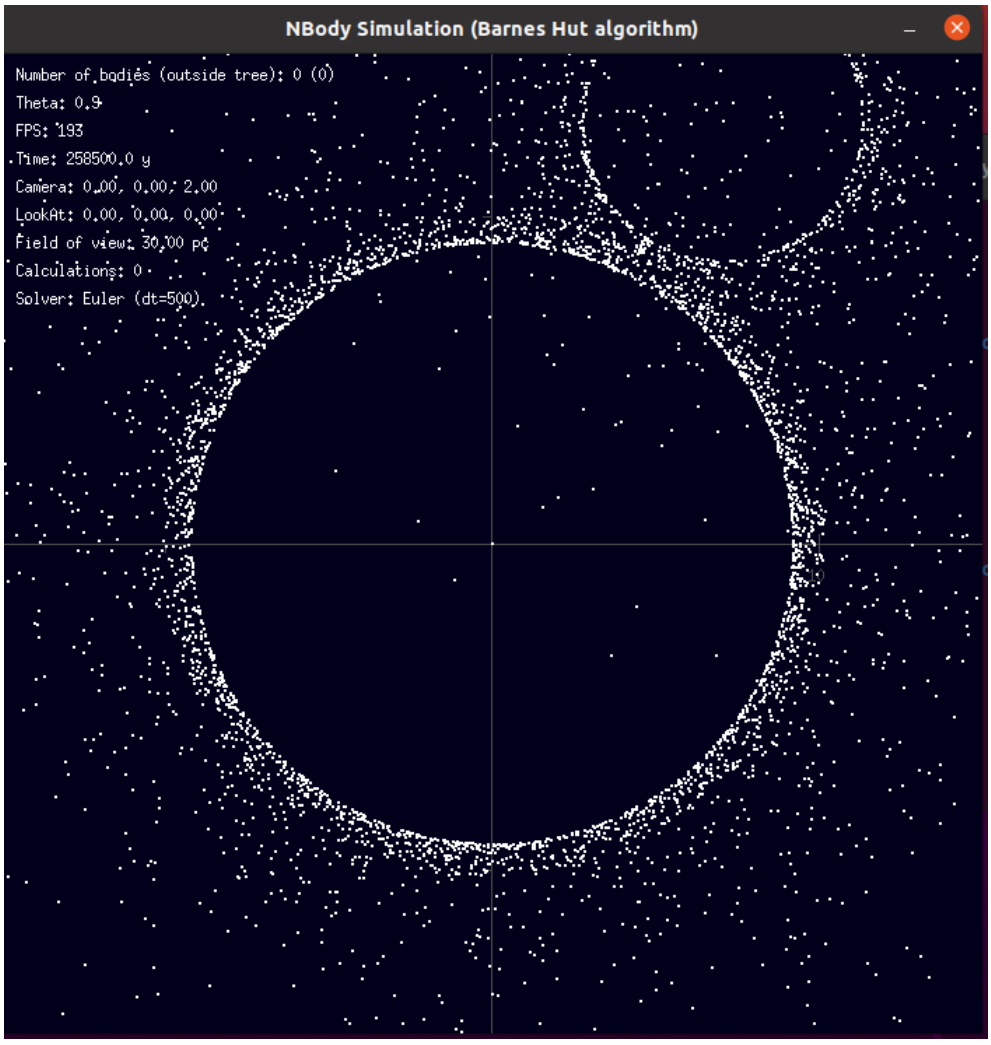


FIGURE 3.1 – Visualisation (aucun calcul effectué)

Chapitre 4

Implémentation naïve et optimisations

4.1 Méthode de calcul brute

La méthode de résolution dite naïve ou brute est la méthode de calcul la plus intuitive mais aussi la plus inefficace. Elle consiste simplement à calculer pour chaque particule les forces qui sont appliquées par toutes les autres. La complexité d'un tel calcul est en $O(N^2)$, il paraît alors évident que cette méthode de calcul est vraiment inefficace.

Voici par exemple, le résultat que nous obtenons avec cette méthode :

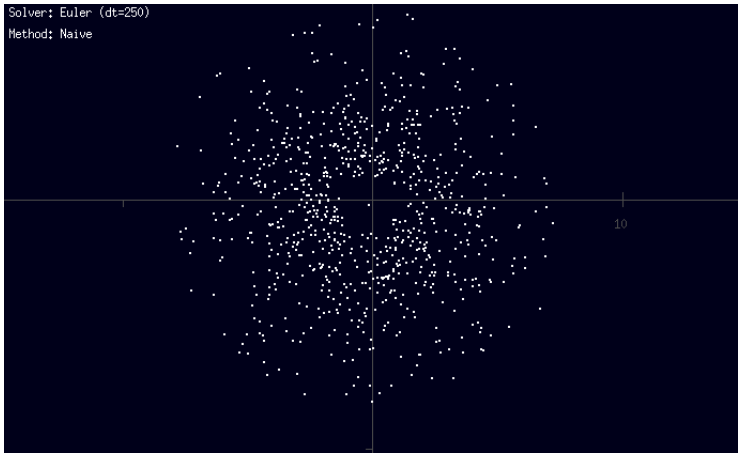


FIGURE 4.1 – Méthode naïve avec 1000 particules

On observe ainsi, que les interactions entre particules semblent cohérentes ce qui laisse apparaître une petite galaxie.

En terme de performance, comme prévu l'algorithme est inefficace et montre ses limites pour $N > 1000$ ce qui est bien trop petit pour pouvoir simuler des galaxies.

4.2 Optimisation de la méthode de calcul brute

Une première manière d'optimiser le calcul des forces est d'utiliser le principe d'action-réaction afin de ne pas effectuer plusieurs fois les calculs. Pour cela, nous pouvons suivre une démarche d'optimisation dynamique en utilisant plus de mémoire afin de réduire nos calculs. Ainsi, en stockant nos forces successives dans un tableau, nous pouvons éviter de faire des calculs déjà effectués.

Cela donne donc le nombre de calculs suivant :
$C = \sum_{n=1}^{N} n = \frac{N(N-1)}{2}$

On peut déjà observer que la complexité de ce calcul est toujours en $O(N^2)$ mais qu'utiliser cette méthode permet de diviser le nombre de calculs par 2 ce qui est déjà une optimisation intéressante. Cependant, étant donné que l'on garde toujours une complexité quadratique, cela n'est toujours pas suffisamment efficace pour simuler des galaxies. En pratique, nous pouvons simuler 2000 particules tout en ayant de bonnes performances mais au-delà de 2000, la méthode montre ses limites.

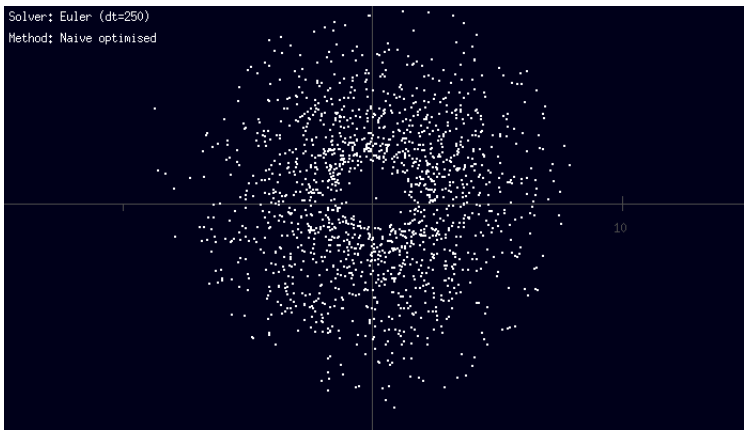


FIGURE 4.2 – Méthode naïve optimisée avec 2000 particules

Chapitre 5

Algorithme de Barnes-Hut

Dans cette partie, nous nous intéressons à l'algorithme de Barnes-Hut afin d'optimiser notre résolution du problème à N-corps.

5.1 Présentation de l'algorithme

L'algorithme de Barnes-Hut est algorithme hiérarchique inventé par Josh Barnes et Piet Hut en 1986. Il est basé sur l'utilisation d'un arbre appelé *quadtree* afin d'approximer le calcul des interactions gravitationnelles. Il permet de réduire les calculs de manière à obtenir une complexité en $O(Nlog(N))$, tout en restant physiquement correct. Sa fiabilité et son efficacité en fait alors l'algorithme le plus utilisé pour résoudre le problème à N corps.

Le principe général de l'algorithme est de regrouper les particules proches en une seule plus grosse particule selon leur distance à la particule dont on veut calculer les forces. Cela permet alors d'approximer correctement les forces touten réduisant les calculs nécessaires.

5.2 Implementation de notre propre algorithme

L'algorithme de Barnes Hut se décompose en trois majeurs parties. On construit tout d'abord notre arbre en insérant les différentes particules, puis on calcule la masse et le centre de masse des noeuds et finalement les forces appliquées entre nos différentes particules. Dans l'algorithme qui nous a été fourni, les fonctions qui codent ces différentes étapes étaient vide, nous avons ainsi effectuons notre propre implémentation de l'algorithme de Barnes-Hut. Rentrons plus en détail.

Avant de commencer ces étapes nous avons besoin de mettre en place la notion de quadrants. Un quadrant est une division de notre arbre soit le noeud. Comme nous travaillons en deux dimensions, nous avons besoin de quatre quadrant, un quadrant en haut à gauche, un autre en bas à gauche , un troisième à droite en haute et un dernier à droite en bas. On les appellera respectivement NW,SW,NE et SE en référence aux points cardinaux. Pour les créer nous avons fait appel à l'outil d'énumération qui assigne une valeur à chacune des variables (par exemple 0 pour NE).

Passons maintenant à la première étape de notre algorithme.

Soit la particule P1. Tout d'abord nous vérifions si P1 rentre dans l'arbre en regardant sa position. SI c'est le cas nous distinguons plusieurs cas. S'il existe plus d'une particule dans le noeud on crée le fils de ce noeud (donc un quadrant à l'intérieur de ce quadrant) et on insère P1 dans ce fils. S'il existe qu'une seule particule on remplace l'ancienne particule par P1 puis on crée le noeud fils qui va nous permettre d'insérer l'ancienne particule. Enfin s'il n'y a aucune particule dans le noeud (donc le noeud est une feuille), on insère directement P1 sans créer de fils. Important : chaque noeud ne doit contenir qu'une seule particule

La deuxième étape, le calcul de la masse et le centre de masse de chaque noeud.

Soit le noeud N1. Si N1 contient qu'une seule particule alors la masse et le centre de masse de cette particule sera aussi celle de N1. Si ce n'est pas le cas, la masse de N1 sera la somme des masses de chacun des fils noeuds et son centre du masse se calcule par la formule suivant

$$centre_de_masse_N1 = masse[1]*centredemasse[1]+...+masse[n]*centredemasse[n]/sommemassechacundesils \tag{5.1}$$

où n représente le nombre de fils.

Enfin pour la dernière étape le principe est simple nous calculons la distance d1 entre chaque particules puis nous comparons le rapport centre de masse du noeud sur d1 par rapport à thêta où thêta est un paramètre qu'on fixe aux alentours de 1. Si ce rapport est est petit alors la force exercé sur la particule sera la force exercé

par le noeud dans lequel il se trouve si le rapport est plus grand, la force exercé sur la particule sera alors la somme des forces de chacun des noeuds fils.

Nous avons ainsi le résultat suivant

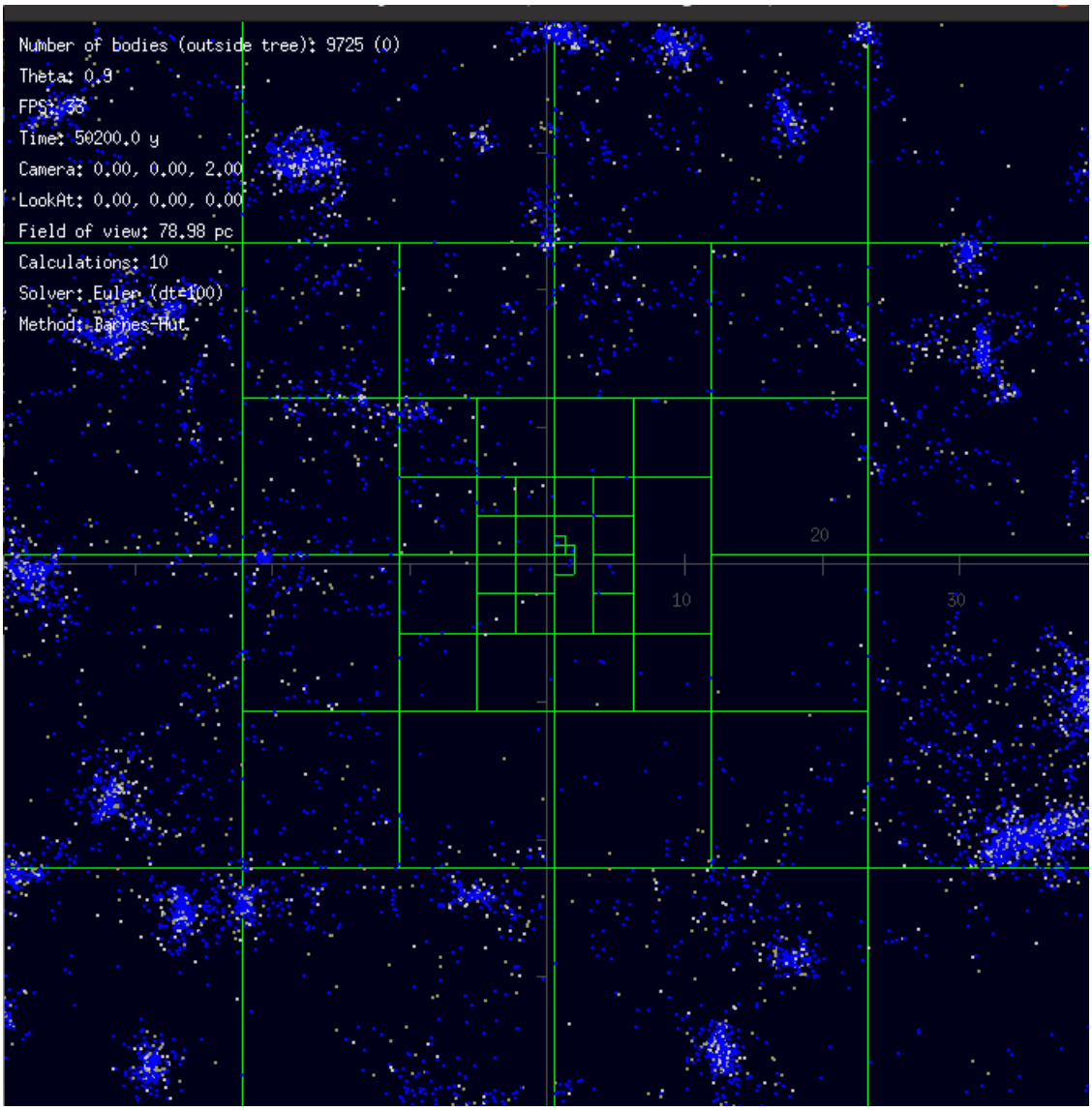


FIGURE 5.1 – Arbre construit avec l'algorithme de Barnes Hut

Chapitre 6

Parallélisation

Les programmes que nous écrivons sont tous séquentiels, c'est-à-dire que les instructions vont s'exécuter les unes à la suite des autres ce qui engendre des temps d'exécution conséquents pour les programmes lourds comme pour le calcul des forces gravitationnelles. La parallélisation ou programmation parallèle est un moyen d'optimiser notre programme et réduire son temps d'exécution. Elle consiste à effectuer des tâches de manière simultanées. Ainsi, un programme parallèle pourra exécuter en même temps des processus défini de manière séquentielle.

Ici, nous nous intéressons à la parallélisation multi-threads à mémoire partagée à travers l'interface de programmation (API) OpenMP.

Un thread ou processus léger est un fil d'exécution qui constitue un processus et permet donc d'exécuter du code machine dans le processeur. Ainsi, l'exécution d'un programme lance un processus qui va ensuite lancer plusieurs threads.

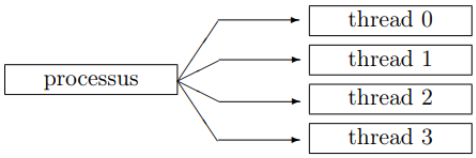


FIGURE 6.1 – Processus et threads

Dans le cas d'un programme séquentiel, seul le thread 0 effectuera une tâche tandis qu'en programmation parallèle, ils seront plusieurs.

La particularité des threads est qu'ils partagent la même zone mémoire ce qui permet donc la parallélisation à condition que le programme soit "compatible".

6.1 Fonctionnement d'OpenMP

6.1.1 Principe

OpenMp est une interface pour la parallélisation multi-threads à mémoire partagée. Elle permet simplement, à partir d'instruction similaire à celle du pré-processeur de paralléliser un programme.

Le principe est ici de paralléliser des blocs d'instructions comme des boucles. Ainsi, un programme utilisant OpenMp est constituée de région séquentielle et de région parallèle. En début de région parallèle, le thread 0 lance alors la création de nouveaux threads.

Il est important de préciser que pour avoir une parallélisation efficace, il est nécessaire d'éviter de refaire des calculs inutiles et de s'assurer que chaque tâches peut s'effectuer sans déranger les autres notamment au niveau de la mémoire partagée.

6.1.2 Directives et fonctions importantes

OpenMp est une API simple à utiliser, il est donc possible de paralléliser un code à partir d'instruction simples.

La plus importante et intéressante pour nous est celle permettant de paralléliser une boucle for :

```
#pragma omp parallel for
```

Voici également des fonctions qui peuvent s'avérer utiles :

```
omp_get_num_threads() //retourne le nombre total de threads utilisés
omp_set_num_threads(int) // spécifie un nombre de thread dans une région parallèle
omp_get_thread_num() // retourne le numéro du thread courant
```

6.2 Application à notre programme de résolution du problème à N-corps

Dans notre cas, les processus les plus lourds sont les calculs de forces gravitationnelle et la création de l'arbre, c'est donc ceux-ci que nous allons paralléliser.

- La construction de l'arbre : les particules peuvent être ajoutées parallèlement cependant, il est possible d'obtenir des problèmes de synchronisation, il est donc intéressant de vérifier si la parallélisation effective.
- Le calcul des forces : les calculs intermédiaires (distances...) sont stockés dans des variables temporaires ce qui facilite la parallélisation.

Chapitre 7

Analyse comparative des performances

Chapitre 8

Conclusion

Jusque-là, nous avons étudié le code qui nous a été fournit afin de commencer une implémentation naïve du problème à N-corps. Avant de résoudre naïvement le problème, nous allons recrée les fonctions nécessaires à l'initialisation à la simulation (nombre de trous noirs galactiques, taille des galaxies ou encore autre forme de départ plus singulière).

Nous reimplémenterons par la suite un intégrateur saute-mouton afin de remplacer celui qui était présent dans le code et qui ne fonctionne pas puis nous coderons tout ce qui permettra à la simulation naive de tourner.

Dans un deuxième temps, nous allons nous servir de l'algorithme de Barnes-Hut afin d'améliorer l'efficacité. Enfin nous allons nous servir de la bibliothèque OpenMP pour faire du multi-thread et ainsi rendre la simulation beaucoup plus efficace.

Annexes

Annexe 1

1 Lien vers le repository github

`https://github.com/Rudiio/Projet-N-corps.git`

Annexe 2