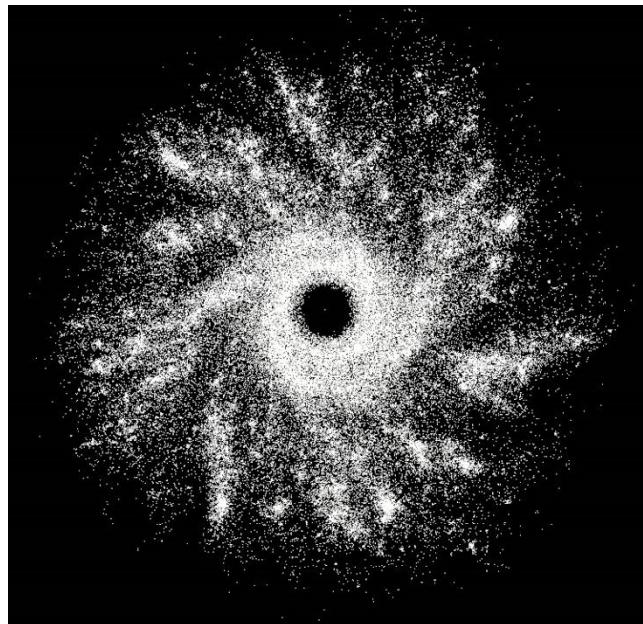


---

# Problème à N-corps : algorithmes et parallélisations

---



*Auteur :*

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

*Référent :*

Roméo MOLINA

# Table des matières

<b>1</b>	<b>Présentation du projet</b>	<b>1</b>
1.1	Sujet . . . . .	1
1.2	Objectif : résolution la plus efficace possible du problème à N-corps . . . . .	1
1.3	Démarche . . . . .	1
<b>2</b>	<b>Aspect physique et mathématique du problème</b>	<b>2</b>
<b>3</b>	<b>Analyse du code</b>	<b>4</b>
3.1	Visualisation . . . . .	4
3.2	ModelNBody.cpp // ModelNBody.h . . . . .	4
3.3	Vector.cpp // Vector.h . . . . .	4
3.4	Types.cpp // Types.h . . . . .	4
3.5	BHTree.cpp // BHTree.h . . . . .	4
3.6	Intégrateurs . . . . .	4
3.7	La visualisation . . . . .	5
<b>4</b>	<b>Implémentation naïve et optimisations</b>	<b>6</b>
4.1	Méthode de calcul brute . . . . .	6
4.2	Optimisation de la méthode de calcul brute . . . . .	6
<b>5</b>	<b>Algorithme de Barnes-Hut</b>	<b>8</b>
5.1	Présentation de l'algorithme . . . . .	8
5.2	Principe général de l'algorithme . . . . .	8
5.3	L'algorithme de Barnes-Hut . . . . .	8
<b>6</b>	<b>Parallélisation</b>	<b>10</b>
6.1	Fonctionnement d'OpenMP . . . . .	10
6.1.1	Principe . . . . .	10
6.1.2	Directives et fonctions importantes . . . . .	10
6.2	Application à notre programme de résolution du problème à N-corps . . . . .	11
<b>7</b>	<b>Analyse comparative des performances</b>	<b>12</b>
7.1	Construction de l'arbre . . . . .	12
7.2	Comparaisons des méthodes . . . . .	13
7.3	Séquentiel et parallèle . . . . .	13
7.3.1	Algorithme de Barnes-Hut . . . . .	13
7.3.2	Naïve optimisée . . . . .	14
7.3.3	Naïve . . . . .	14
<b>8</b>	<b>Conclusion</b>	<b>15</b>
	<b>Annexes</b>	<b>16</b>
	<b>Annexe 1</b>	<b>17</b>
1	Lien vers le repository github . . . . .	17
	<b>Annexe 2</b>	<b>18</b>

# Chapitre 1

## Présentation du projet

### 1.1 Sujet

Le problème à 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.

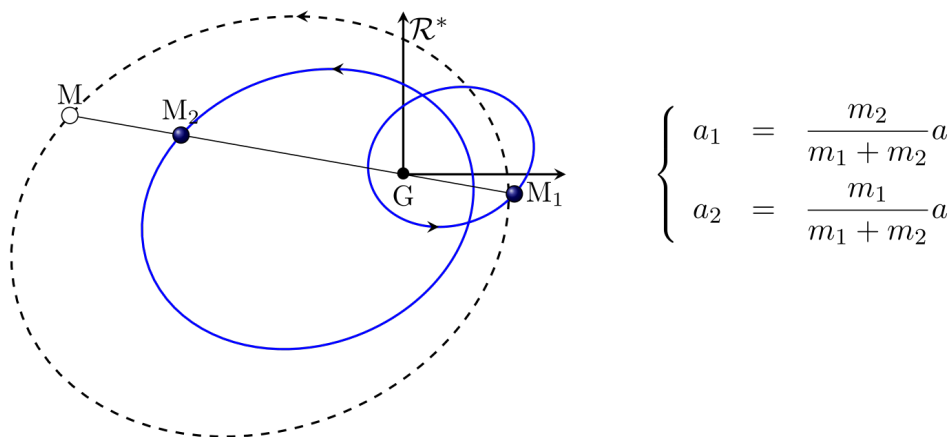


FIGURE 1.1 – Solution du problème à 2 corps

### 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 particule 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 :

$$\vec{F}_{2 \rightarrow 1} = \frac{-Gm_1m_2}{\|\vec{p}_{2 \rightarrow 1}\|^2} \vec{p}_{2 \rightarrow 1} \quad (2.1)$$

où

- $G$  est la constante de la gravitation  $G = 6.672 \cdot 10^{-11} Nm^2/kg^2$ .
- $m_1$  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  $p_1$   $F_{2 \rightarrow 1}$  est alors la suivante :

$$F_{2 \rightarrow 1} = G * m_1 * m_2 * \begin{pmatrix} \frac{x_2 - x_1}{d^3} \\ \frac{y_2 - y_1}{d^3} \end{pmatrix} \text{ Cette formule est alors plus simple à implémenter.}$$

### Expression de l'accélération

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

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

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  $OP_1(t)$  :

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

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  $N \geq 3$ .

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 :

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

où  $\Delta t$  est le pas de discrétisation du temps.

Cependant, pour des problèmes de mécanique, 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 :

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

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

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

### 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 :  $\vec{F}_{2 \rightarrow 1} = -\vec{F}_{1 \rightarrow 2}$

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. Les variables principales seront les masses des étoiles ainsi que la répartition initiale des étoiles. Les vitesses orbitales sont initialisées comme si la galaxie est déjà formée, même si cela peut paraître incohérent, c'est ce choix qui est fait dans la plupart des modélisations.

### Problème théorique : Newton ou Einstein ?

Malgré l'efficacité de la loi d'attraction gravitationnelle de Newton, il demeure un problème : son cadre d'application, celle-ci ne pouvant s'appliquer que lorsque les champs gravitationnels sont faibles ou modérés (peu de particules, particules peu massives, ou particules éloignées). Dans le cadre où ces conditions ne sont pas respectées, ce qui se passe en réalité souvent (lorsque deux particules ont des positions très proches), une des particules va se retrouver éjectée en adoptant une vitesse plus grande que celle de la lumière, ces éjections étant un réel problème. Pour résoudre ce problème théorique, nous aurions pu utiliser la relativité générale d'Einstein, mais nous avons plutôt opté pour l'ajout d'un paramètre *softening* qui va créer artificiellement une distance entre deux étoiles.

### Question de l'initialisation : Avec ou sans trou noir ?

La question des trous noirs est assez importante car ce sont des objets qui participent à une stabilité locale du centre des galaxies, stabilité qui de proche en proche impacte la stabilité de l'ensemble. Nous avons donc choisi d'en positionner un au centre de chaque galaxie.

## 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. Le fait que nous codons par dessus un code qui n'est pas le nôtre nous a compliqué la tâche lors de l'implémentation de certaines fonctions, comme l'intégrateur. L'analyse du code consistera en une description de chaque fichier et des structures utilisées, le code est très commenté.

### 3.1 Visualisation

L'affichage est géré par les fichiers *SDLWnd* et *NBodyWnd*, qui utilise les libraires SDL et OpenGL. Cette partie du code était déjà implémentée quand le code nous a été transmis, les principales modifications que nous avons fait consiste en des corrections de bug (touches pour zoomer, dézoomer, afficher l'arbre) ainsi qu'en l'ajout d'une coloration aux étoiles en fonction du mode d'initialisation choisi.

### 3.2 ModelNBody.cpp // ModelNBody.h

Ces fichiers contiennent la classe sur laquelle repose la simulation. C'est ici que l'on initialise la simulation et toutes les particules.

### 3.3 Vector.cpp // Vector.h

Le fichier *Vector.cpp* permet la création et la manipulation de vecteurs 2D et 3D. On y retrouve un constructeur pour les vecteur de deux dimensions et un deuxième pour les vecteurs de trois dimensions. Le fichier *Vector.h* associé contient les différentes classes des vecteurs.

### 3.4 Types.cpp // Types.h

Cette partie du code contient tous les structures et les méthodes nécessaires à la manipulation des particules (étoiles) et de leurs caractéristiques (position, vitesse, accélération).

### 3.5 BHTree.cpp // BHTree.h

Ces fichiers concerne la mise en place des différentes fonctions nécessaires à l'algorithme de Barnes-Hut. Il y a notamment les structures des Quadrant, ainsi que tous les informations sur les masses et les centres de masse qui y sont stockées. Les fonctions de construction, de suppression ou de calcul sur l'arbre nécessaires ont été complétées dans *BHTree.cpp*.

### 3.6 Intégrateurs

Le dossier *src* contient également les fichiers *IntegratorEuler*, *IntegratorLeapFrog* et *IIntegrator* qui sont des fichiers contenant des méthodes numériques qui nous permettent de calculer l'évolution des positions des particules en considérant un pas de temps  $\Delta t$  et en ayant au préalable effectué le calcul des forces exercées sur chaque particule grâce aux fichiers *BHTree*.

### 3.7 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.

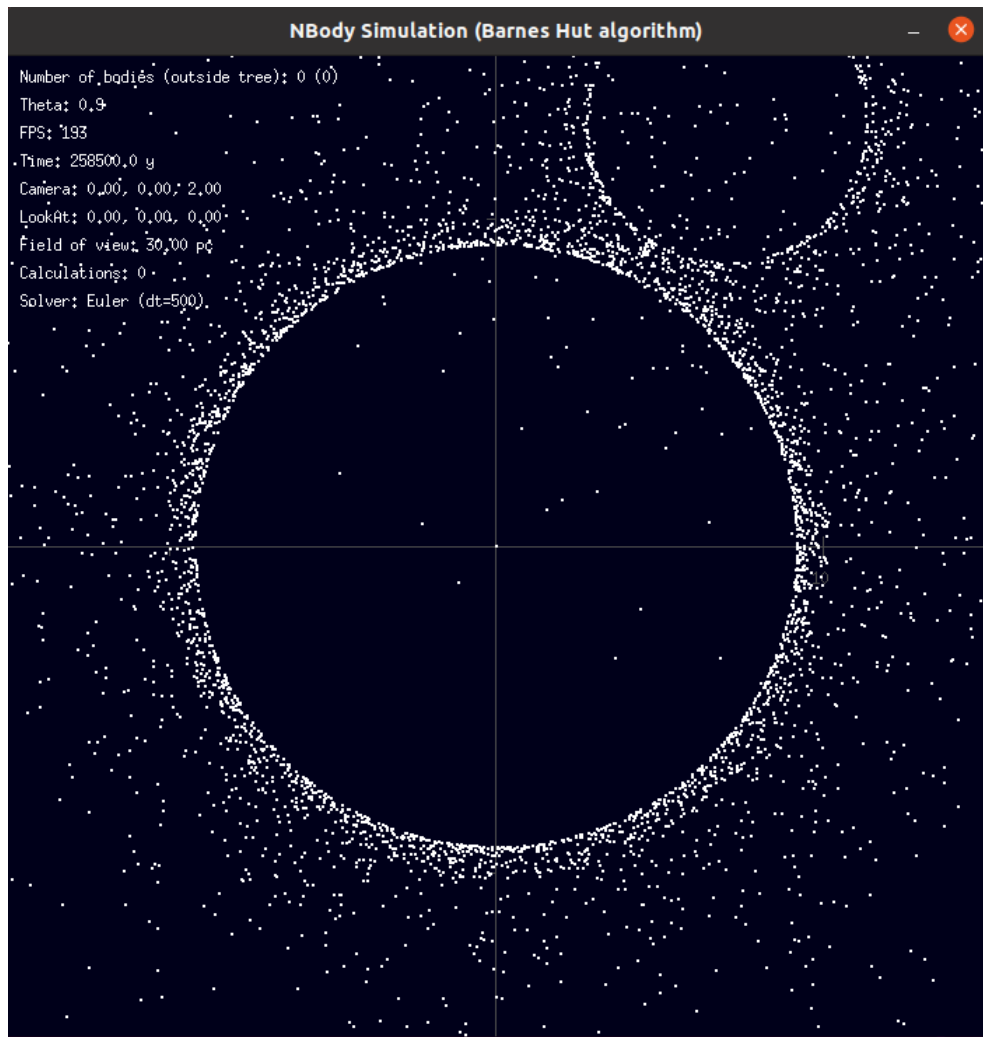


FIGURE 3.1 – Visualisation (aucun calcul effectué)

## 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 :

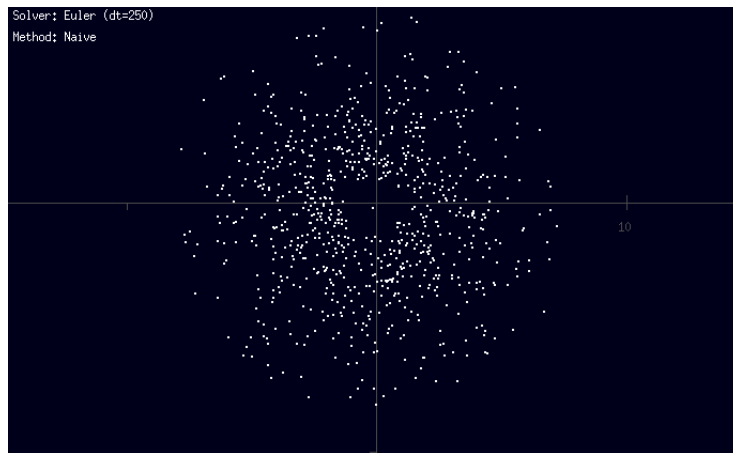


FIGURE 4.1 – Méthode naïve avec 1000 particules

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.



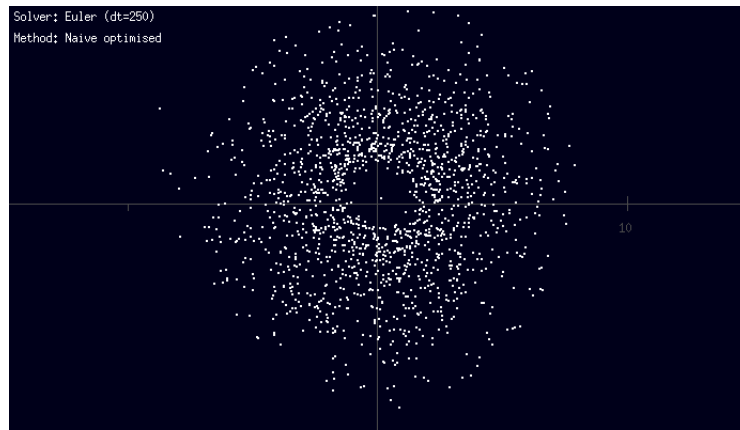


FIGURE 4.2 – Méthode naïve optimisée avec 2000 particules

## Chapitre 5

# Algorithme de Barnes-Hut

Pour accélérer les calculs et permettre de plus grandes simulations, nous allons dans cette partie s'intéresser à l'algorithme de Barnes-Hut.

### 5.1 Présentation de l'algorithme

L'algorithme de Barnes-Hut est un 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  $O(N \log(N))$ , 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.

### 5.2 Principe général de l'algorithme

L'idée est d'approcher les forces à longue portée en remplaçant un groupe de points éloignés par leur centre de masse. Il y a évidemment en contrepartie une légère part d'erreur et d'approximation mais ce schéma accélère considérablement le calcul. Notamment avec une complexité  $N \log(N)$  plutôt que  $N^2$ . Au centre de cette approximation se trouve un arbre : une « carte » de l'espace qui nous aide à modéliser des groupes de points comme un seul centre de masse. En deux dimensions, nous pouvons utiliser une structure de données quadtree, qui subdivise de manière récursive les régions carrées de l'espace en quatre quadrants de taille égale. (En trois dimensions, on peut utiliser un octree qui divise de la même manière un volume cubique en huit sous-cubes.)

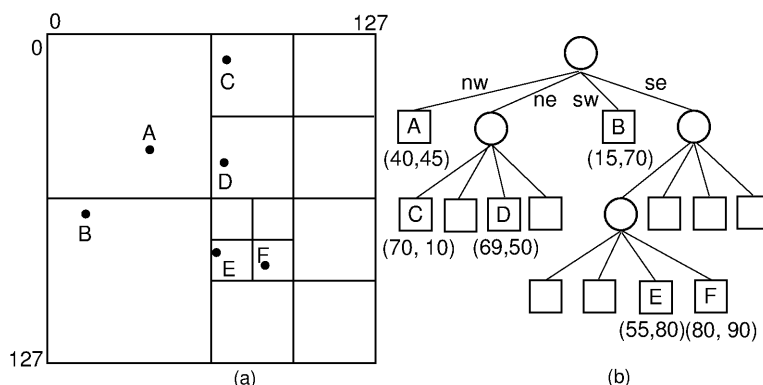


FIGURE 5.1 – Exemple de Quadtree

### 5.3 L'algorithme de Barnes-Hut

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 nœuds 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 vides, nous avons ainsi effectués notre propre implémentation de l'algorithme de Barnes-Hut. Rentrions 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 nœud. 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 haut 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

$$centre_{de\ masse\ N1} = masse[1]*centredemasse[1] + \dots + masse[n]*centredemasse[n] / sommemassechacun\ des\ fils \quad (5.1)$$

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  $\theta$  où  $\theta$  est un paramètre qu'on fixe aux alentours de 1. Si ce rapport 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

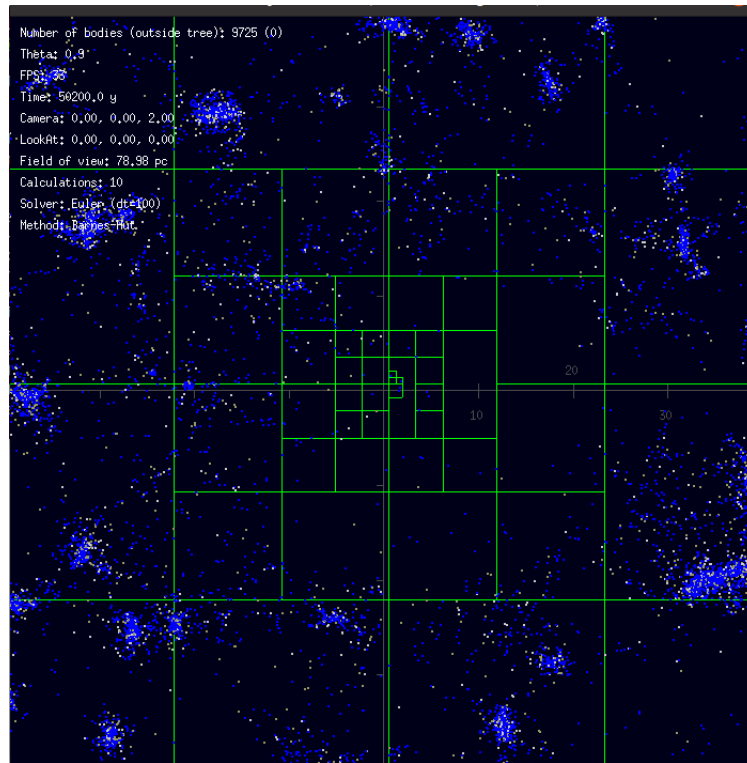


FIGURE 5.2 – Arbre construit avec l'algorithme de Barnes Hut

# 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.

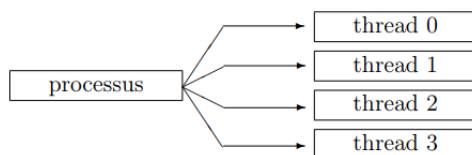


FIGURE 6.1 – Processus et threads

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 :

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

Voici également des fonctions qui peuvent s'avérer utiles :

```
1 omp_get_num_threads() //retourne le nombre total de threads utilisés
2 omp_set_num_threads(int) // spécifie un nombre de thread dans une région parallèle
3 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

Dans cette partie, nous analysons les résultats et performances des différentes méthodes et optimisations implémentées.

Pour cela, nous avons lancé des séries de tests avec 100 itérations et avec des nombres de particules différents. Nous avons donc mesuré et sommé le temps concerné pour chaque itération afin d'effectuer une moyenne pour chaque nombre de particules.

### 7.1 Construction de l'arbre

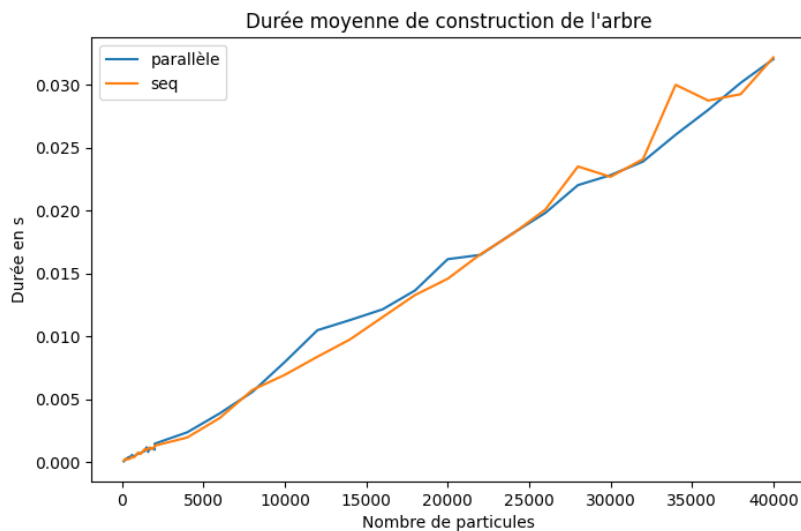
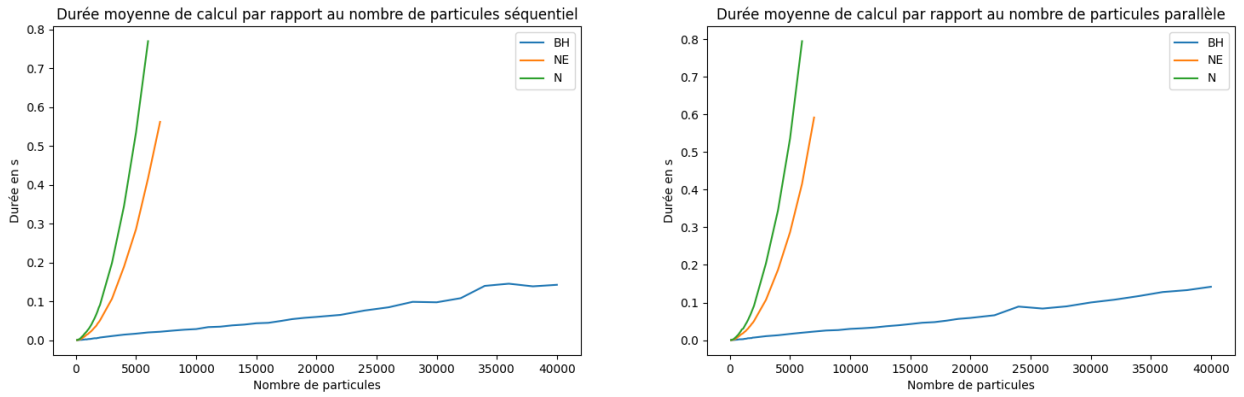


FIGURE 7.1 – Durée moyenne de construction de l'arbre

On peut voir sur la figure que pour des plus petits nombres de particules, la version séquentielle aura légèrement de meilleure performance alors que pour des valeurs plus grandes ( $>250000$ ), la version parallèle sera plus intéressante. Mais de manière globale, l'algorithme de création de l'arbre est aussi performant avec ou sans parallélisation. On peut alors penser que la parallélisation ne fonctionne pas correctement à cause de problème de compétition entre les threads : les insertions des particules ne sont pas totalement indépendantes.

## 7.2 Comparaisons des méthodes



Les résultats confirment la théorie. Il est clair que les méthodes brutes ont une complexité en  $O(N^2)$  alors que l'algorithme de Barnes-Hut a une complexité bien inférieure en  $O(N\log(N))$ . On peut également confirmer que l'algorithme naïf optimisé a une meilleure complexité que la version naïve.

## 7.3 Séquentiel et parallèle

### 7.3.1 Algorithme de Barnes-Hut

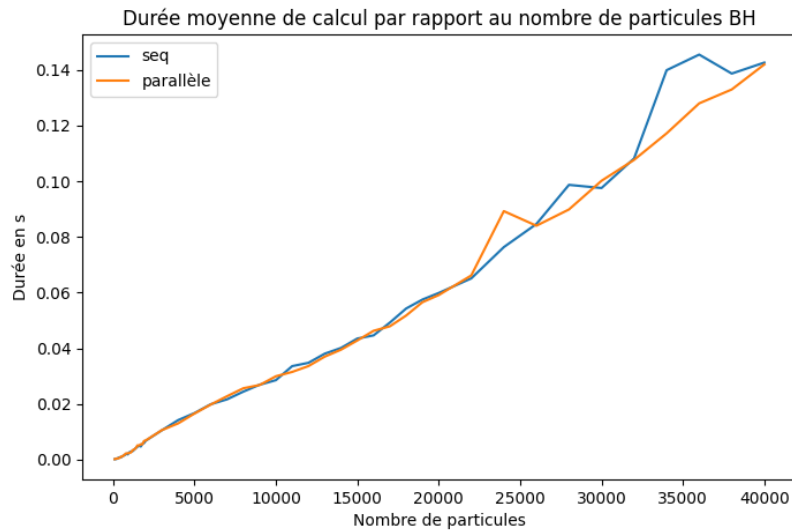


FIGURE 7.2 – Résultats pour l'algorithme de Barnes-Hut

On peut voir sur la figure que globalement la parallélisation n'améliore pas énormément l'efficacité de l'algorithme. Cependant, malgré tout, la version parallèle a une complexité plus "stable" et pour un grand nombre de particules ( $>2000$ ), il sera plus intéressant.

### 7.3.2 Naïve optimisée

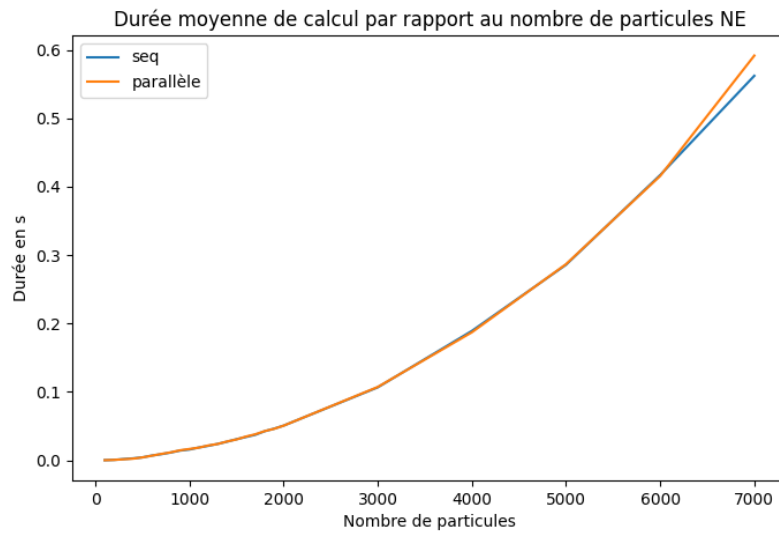


FIGURE 7.3 – Résultats pour l'algorithme naïf optimisé

Les résultats montrent clairement que la parallélisation n'améliore pas la complexité de l'algorithme et que pour des grandes valeurs elle ralentit les calculs. Cela s'explique simplement par les situations de compétition des threads lors de l'accès aux données. En effet, chaque calcul de force dépend de ceux des particules précédentes ce qui ralentit les calculs au final.

### 7.3.3 Naïve

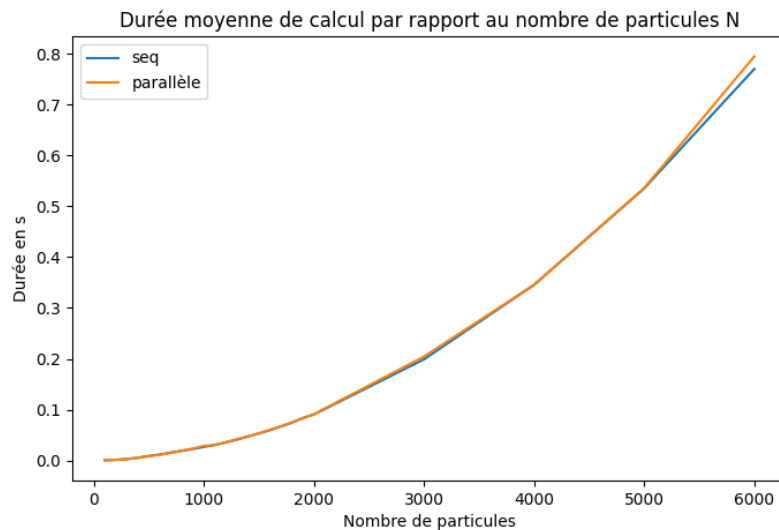


FIGURE 7.4 – Résultats pour l'algorithme naïf

De nouveau, la parallélisation ne change pas les performances de l'algorithme ce qui est plutôt surprenant étant donné que les calculs de forces sont tous indépendants les uns des autres. Cependant, on pourrait expliquer cela par la répétitions des calculs pour chaque particules étant donné que l'on n'utilise pas le principe d'action-réaction.

Les résultats obtenus dans cette partie ont permis de vérifier les résultats théoriques par rapport aux performances des algorithmes : l'algorithme de Barnes-Hut est bien plus performant que les autres. De plus, ils ont également montré que l'algorithme de Barnes-Hut est intéressant à paralléliser alors que les autres ne subissent aucune augmentation de performance.



## Chapitre 8

# Conclusion

Jusque-là, nous avons étudié le code qui nous a été fourni afin de commencer une implémentation naïve du problème à N-corps. Avant de résoudre naïvement le problème, nous allons recréer 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 réimplémenterons 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 naïve 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

salut @miscite0, author = Auteur Untel, title = Titre1, howpublished = <<http://www.url0.org/>>, year = 2014, note = "[Online; accessed 16-January-2014]"

@miscite1, author = Auteur Autre, title = Titre2, howpublished = <<http://www.url1.org/>>, year = 2014, note = "[Online; accessed 16-January-2014]"

@miscite2, author = Auteur Ailleurs, title = Titre3, howpublished = <<http://www.url2.org/>>, year = 2014, note = "[Online; accessed 16-January-2014]"

@miscite3, author = Auteur Lui, title = Titre4, howpublished = <<http://www.url3.org/>>, year = 2014, note = "[Online; accessed 16-January-2014]"

@miscite4, author = Auteur Elle, title = Titre5, howpublished = <<http://www.url4.org/>>, year = 2014, note = "[Online; accessed 16-January-2014]"

@bookcite6, author = Auteur Livre1, title = Titre Livre1, year = 2014, publisher = Editeur1