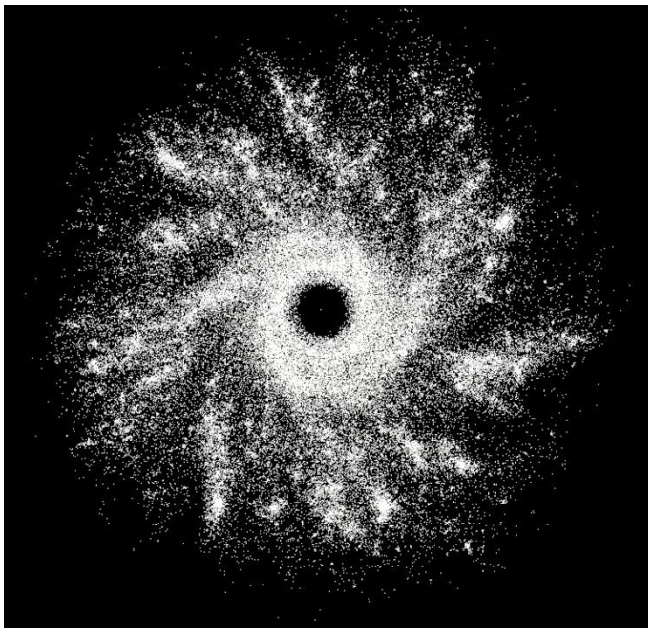
POLYTECH SORBONNE

Problème à N-corps : algorithmes et parallélisations



Auteurs :
Rudio FIDA CYRILLE
Noah VEYTIZOUX
Elyas ASSILI
Camille HASCOET

Référent :
Roméo MOLINA

20 mai 2022

Table des matières

Partie 1

Présentation du projet

1.1 Sujet

Le problème à N-corps consiste à calculer le mouvement de N particules en connaissant leur position et leur vitesse initiales respectives. 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 célestes interagissant gravitationnellement. Le problème à deux corps et le problème à trois corps sont résolubles de manière exacte mais ces solutions sont peu efficaces, quant au problème à 4 corps, il ne possède pas de solution exacte sans postulat supplémentaire. Pour $N > 4$, il n'existe pas de résolution exacte, il faut donc utiliser des solutions approchées.

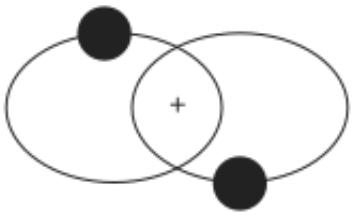


FIGURE 1.1 – Solution du problème à 2 corps
source : `https://www.chegg.com`

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 des galaxies. Il sera alors intéressant de comparer les performances de ces méthodes. Pour cela, nous utiliserons des solutions approchées calculées à partir de différents algorithmes dont l'algorithme de Barnes-Hut. Nous étudierons également l'application de la parallélisation à notre programme. En pratique, le projet consiste à compléter et améliorer un projet déjà commencé afin d'acquérir de nouvelles compétences en C++, sur les algorithmes hiérarchiques, en parallélisation et plus généralement en optimisation de code.

1.3 Démarche

Pour chaque particule, il est nécessaire de calculer la force gravitationnelle qui s'y applique puis d'utiliser un intégrateur numérique pour calculer sa position. Pour le calcul des forces, il est alors intéressant d'utiliser différents algorithmes notamment le calcul naïf et l'algorithme de Barnes-Hut. Pour l'intégration, il est possible d'utiliser la méthode d'Euler ou la méthode saute mouton. Par la suite, il sera intéressant d'explorer les différentes possibilités d'optimisation, telle que la parallélisation multi-thread.

Partie 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. L'unique force prise en compte est ainsi l'interaction gravitationnelle d'un corps sur un autre, les autres étant considérées comme négligeables à cette échelle.

2.1 Un problème mécanique

2.1.1 Force gravitationnelle

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.
— $\vec{p}_{2\to1}$ est le vecteur allant de p_2 vers p_1.

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 : $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
La force appliquée par p_2 sur p_1 $F_{2\to1}$ est alors la suivante :

$$F_{2\to1} = G * m_1 * m_2 * \begin{pmatrix} \frac{x_2 - x_1}{d^3} \\ \frac{y_2 - y_1}{d^3} \end{pmatrix}$$

2.1.2 Expression de l'accélération

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

$$\Sigma\vec{F} = m\vec{a}$$
$$\text{où } \vec{a} = \frac{d\vec{v}}{dt} \text{ et } \vec{v} = \frac{d\vec{p}}{dt}$$

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

2.1.3 Calcul de la position

On obtient donc l'équation différentielle sur la position $\vec{Op_1}(t)$ de la particule $p1$ par rapport à l'origine du référentiel O :

$$\frac{d^2\vec{Op_1}(t)}{dt^2} = \vec{a(t)} \tag{2.2}$$

L'intégrateur permettra alors de résoudre numériquement et simplement cette équation différentielle, ce qui se substituera à une solution exacte.

2.1.4 Conservation de l'énergie

Dans un système mécanique ne comprenant que des forces conservatives, l'énergie mécanique se conserve en fonction du temps. Il s'agit du théorème de l'énergie mécanique.

L'énergie mécanique E_m est la somme de l'énergie cinétique E_c du système et de son énergie potentielle E_p :

$$E_m = E_c + E_p$$

Dans le cas du problème à N-corps, la seule force prise en compte est l'interaction gravitationnelle qui est conservative, on peut donc appliquer le théorème de conservation de l'énergie potentielle. Pour N corps, on a alors :

$$E_c = \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2 \text{ et } E_m = -\sum_{i=1}^{N} \sum_{j=1, j\neq i}^{N} \frac{G m_i m_j}{|r_j - r_i|}$$

avec :

— v_i la vitesse de la $i$ème particule.
— m_i la masse de la $i$ème particule.
— G la constante gravitationnelle.
— r_i la position de la $i$ème particule.

Le théorème de l'énergie cinétique sera utile en pratique pour vérifier si nos algorithmes sont corrects et s'ils respectent l'aspect physique du problème.

2.1.5 Principe d'action-réaction

La 3ème loi de Newton, appelée également principe d'action-réaction, énonce 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 : $\vec{F}_{2\to1} = -\vec{F}_{1\to2}$

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

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

2.1.7 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. Ce comportement est dû à la modélisation physique du système ; l'illustration de la concurrence entre ces deux théories de la gravitation est l'orbite de Mercure, qui soumise à un champ gravitationnel fort car proche du Soleil, n'est pas parfaitement décris par les équations de Newton. 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.

2.1.8 Question de l'initialisation : avec ou sans trou noir ?

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

2.2 L'aspect mathématique du problème : résolution numérique

Pour calculer la position de chaque particule, il est nécessaire de résoudre l'équation différentielle donnée dans la section 2.1.3, grâce aux intégrateurs numériques suivant.

2.2.1 La méthode d'Euler explicite

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.

Il est important de noter pour plus tard que diminuer Δt permet d'augmenter la précision de la simulation mais réduit ses performances. C'est le paramètre principal concernant la précision de l'évolution de notre système.

2.2.2 La méthode saute-mouton (Leap frog)

Pour des problèmes de mécanique, et notamment dans notre cas, la méthode d'Euler n'est pas stable (divergence de trajectoire et/ou non conservation de l'énergie du système). Ainsi, il est préférable d'utiliser le schéma saute-mouton ("leap frog") qui est d'ordre 2, et a l'avantage de conserver l'énergie mécanique des systèmes étudiés.

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

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

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

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

où Δt est le pas de discrétisation du temps.

Partie 3

Analyse du code

Notre projet s'effectue dans la continuité d'un projet déjà bien entamé, ainsi, la première étape a été de comprendre et d'assimiler le travail déjà 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 utilisent 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 des 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 concernent la mise en place des différentes fonctions nécessaires à l'algorithme de Barnes-Hut. Il y a notamment les structures des Quadrants, ainsi que toutes 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* et *IntegratorLeapFrog* 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 Δt et en ayant au préalable effectué le calcul des forces exercées sur chaque particule grâce aux fichiers *BHTree*.

Partie 4

Implémentation naïve et optimisations

4.1 Méthode de calcul naïve

Cette méthode de résolution, dite naïve, 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 $O(N^2)$.

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

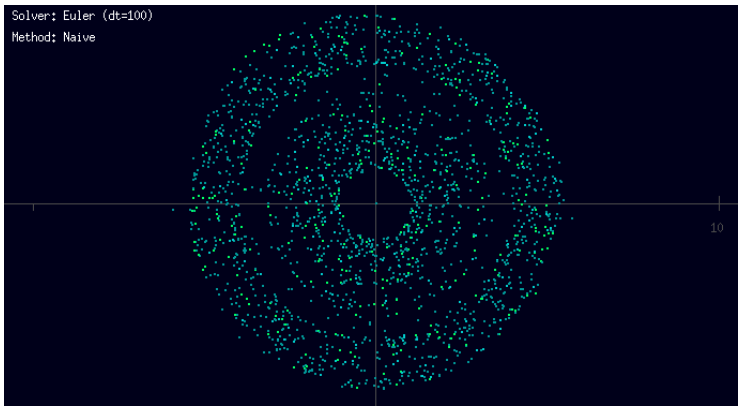


FIGURE 4.1 – Méthode naïve avec 1000 particules

En terme de performances, comme prévu l'algorithme est inefficace et montre ses limites pour $N > 2000$ ce qui est bien trop petit pour pouvoir simuler des galaxies. Pour mettre en perspectives, la Voie Lactée compte entre 100 et 200 milliards d'étoiles.

4.2 Optimisation de la méthode naïve

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 faire de la programmation dynamique en se basant sur la réciprocité de l'interaction gravitationnelle. Ainsi, si l'on considère les particules $p1$ et $p2$, lors du calcul de la force $\vec{F}_{2\to1}$ de $p2$ sur $p1$, nous pouvons stocker directement la force de $p1$ sur $p2$, $\vec{F}_{1\to2}$ qui vaut $-\vec{F}_{2\to1}$, ce qui nous permet de ne pas refaire le calcul lors du calcul des forces qui s'appliquent sur $p2$.

On effectue donc à la place $C = \sum_{n=1}^{N} n = \frac{N(N-1)}{2}$ calculs.

On peut déjà observer que la complexité de ce calcul est toujours en $O(N^2)$, mais l'amélioration de cette méthode est un premier pas. Cependant, étant donné la complexité quadratique, nous ne pouvons toujours pas simuler une galaxie avec beaucoup d'étoiles. En pratique, nous pouvons simuler 2500 particules tout en ayant de bonnes performances mais au-delà de 2500, la méthode montre ses limites.

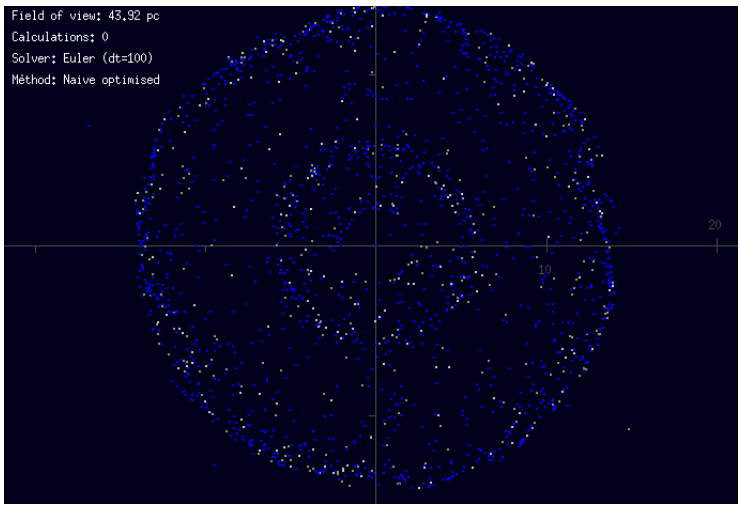


FIGURE 4.2 – Méthode naïve optimisée avec 2000 particules colorées

4.3 Vérification énergétique des modèles

Il est nécessaire de vérifier si nos modèles sont bien implémentés et cohérents avec la physique du problème. Pour cela, un moyen de procéder est de calculer l'énergie mécanique du système à partir de son énergie cinétique et de son énergie potentielle. En effet, l'énergie mécanique du système se conserve au cours du temps étant donné que la force gravitationnelle est conservative. Cela permet aussi de vérifier la validité de notre intégrateur car l'instabilité d'un intégrateur se traduirait par le non respect de ce principe de conservation de l'énergie mécanique.

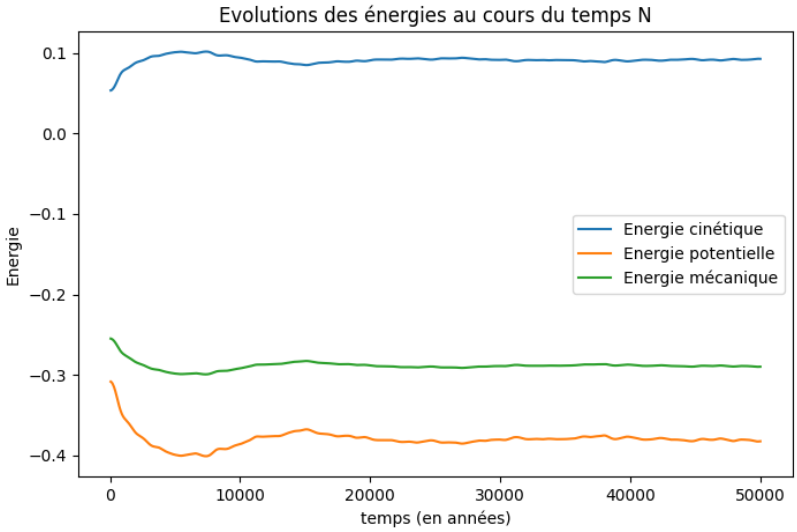


FIGURE 4.3 – Bilan énergétique méthode naïve
($\Delta t = 50$ ans, 1000 itérations et 2000 particules)

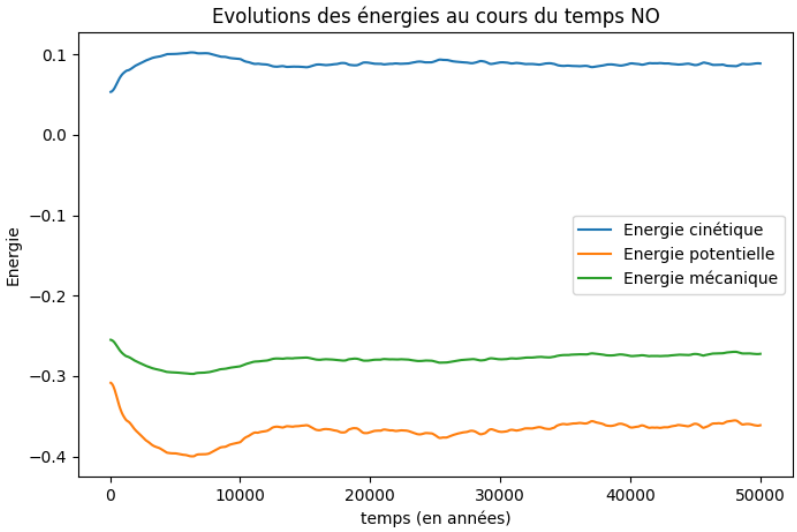


FIGURE 4.4 – Bilan énergétique méthode naïve optimisée
($\Delta t = 50$ ans, 1000 itérations et 2000 particules)

Pour les deux méthodes, au début de la simulation les énergies varient beaucoup. Cela est dû aux conditions initiales qui ne respectent pas l'aspect physique du problème. On peut voir le premier saut comme un indicateur d'incohérence de nos conditions initiales. On peut voir sur la figure 4.3 que l'énergie mécanique de la méthode naïve est bien constante à partir de $t = 20000$, ce qui valide bien le modèle. Cela nous informe aussi sur la vitesse de convergence de la méthode : le modèle converge en 400 itérations.

Sur la figure 4.4, on peut voir que de même l'énergie mécanique se conserve bien pour la méthode naïve optimisée à partir de la 240ème itération. Ainsi, la méthode converge plus vite que la méthode naïve.

Il est important de noter que pour les deux méthodes, on peut observer de légères variations qui sont dues aux approximations des intégrateurs numérique, de la précision du calcul, et du paramètre *softening*.

Partie 5

Algorithme de Barnes-Hut

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

5.1 Présentation de l'algorithme

L'algorithme de Barnes-Hut est un algorithme hiérarchique (cf. un algorithme qui classe les calculs à effectuer en fonction d'un paramètre) 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.

5.2 Principe général de l'algorithme

L'idée est d'approcher les forces à longue portée en considérant un groupe de points éloignés comme équivalent à leur centre de masse. Il y a évidemment en contrepartie une légère erreur due à l'approximation mais ce schéma accélère considérablement le calcul. Cette erreur peut être contrôlée à partir d'un paramètre appelé θ. Cet algorithme a une complexité en $O(Nlog(N))$ au lieu du $O(N^2)$ des méthodes naïves. 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 cube en huit sous-cubes.

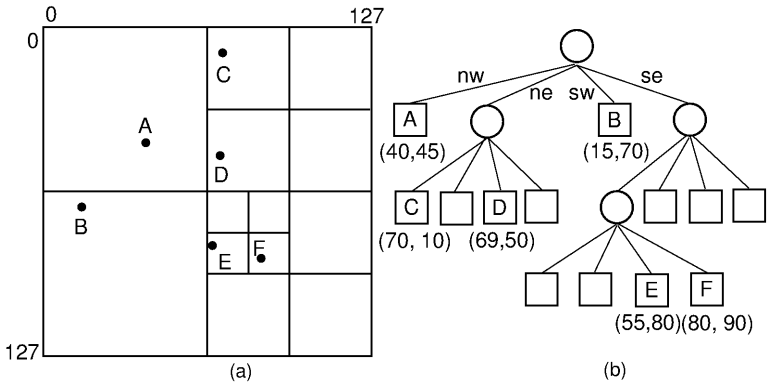


FIGURE 5.1 – Exemple de Quadtree
`https://www.lrde.epita.fr`

5.3 L'algorithme de Barnes-Hut

L'algorithme de Barnes Hut se décompose en trois étapes majeures :

1. la construction de l'arbre
2. le calcul des centres des masses et des masses correspondants à chaque nœud
3. le calcul des forces appliquées sur les particules

Selon la version de l'algorithme utilisée, les étapes 1 et 2 peuvent être effectuées simultanément.

Avant de détailler ces étapes nous avons besoin de mettre en place la notion de quadrant. Un quadrant est une division de notre arbre et donc un nœud de l'arbre. Comme nous travaillons en deux dimensions, nous avons besoin de quatre quadrants, 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 (North-West),SW (South-West),NE (North-East) et SE (South-East) en référence aux points cardinaux.

5.3.1 La construction de l'arbre

Elle consiste à insérer successivement les particules dans l'arbre.

Soit la particule $p1$. Tout d'abord nous devons vérifier que $p1$ doit bien être insérée dans l'arbre. Si c'est le cas nous distinguons plusieurs cas.

— S'il existe plus d'une particule dans le nœud, on doit insérer la particule dans le fils du nœud auquel correspond sa position donc dans un des quatre quadrants. S'il n'existe pas, on crée ce fils.

— S'il existe qu'une seule particule dans le nœud, on réinsère l'ancienne particule dans le fils auquel correspond sa position puis on effectue l'insertion de $p1$ dans le quadrant qui lui correspond. Si les fils n'existent pas, on les crée avant l'insertion.

— Enfin s'il n'y a aucune particule dans le nœud, il s'agit d'une feuille, on insère directement $p1$ sans créer de fils.

Les particules ne sont insérées que dans les feuilles de l'arbre, les nœuds peuvent contenir plusieurs particules mais les particules en elles-mêmes ne seront stockées dans l'arbre que dans des feuilles de l'arbre (nœuds sans fils). Ainsi, à chaque itération, on découpe nos nœuds en 4 carrés jusqu'à avoir inséré toutes les particules et que chaque carré ne contienne au plus qu'une seule particule comme on peut le voir dans la figure 5.1.

5.3.2 Le calcul des masses et centres de masse

Les calculs des masses et centres de masses se calculent naturellement en utilisant la récursivité en commençant par les feuilles de l'arbre.

Soit le nœud n. Si n ne contient qu'une seule particule alors il s'agit d'une feuille et sa masse correspond à la masse de la particule et son centre de masse sera la position de la particule. Si ce n'est pas le cas, la masse de n sera la somme des masses de chacun des fils et son centre du masse se calcule par la formule suivante

$$CM_n = \frac{m_1 * CM_1 + m_2 * CM_2 + m_3 * CM_3 + m_4 * CM_4}{\sum_{i=1}^{4} m_i} \tag{5.1}$$

où $\forall i \in \{1, 2, 3, 4\}$, CM_i est le centre de masse d'un fils et m_i est sa masse.

5.3.3 Le calcul des forces

L'avantage de l'utilisation d'un arbre est que le calcul des forces gravitationnelles pour chaque particule se fait simplement en se servant de la récursivité, à partir de la racine de l'arbre. De plus, comme dit précédemment, le niveau de précision de l'algorithme et donc le nombre de calculs dépendent du paramètre θ qu'on fixe autour de 1. Nous prendrons $\theta = 0.9$.

Pour chaque particule p, on parcourt l'arbre depuis sa racine.

— Si le nœud actuel est une feuille, on calcule simplement la force exercée par la particule qu'elle contient sur la particule actuelle.

— Sinon, on calcule le rapport $\frac{l}{D}$ où l est la longueur d'un coté du nœud actuel et D est la distance entre la particule p et le centre de masse du nœud actuel. Si $\frac{l}{D} < \theta$ alors on calcule la force, appliquée à p à partir du centre de masse et de la masse du nœud. Sinon, elle sera la somme des forces exercées par les fils de ce nœud.

Nous avons ainsi par exemple le résultat suivant :

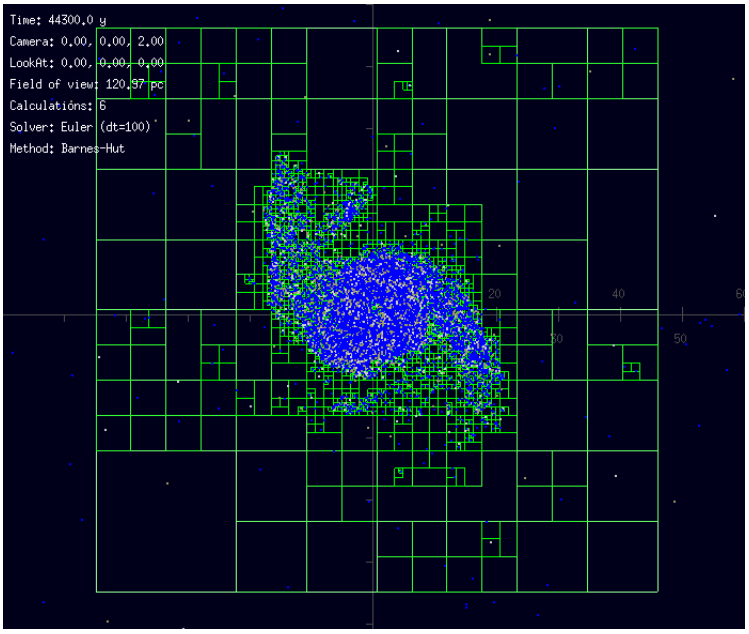


FIGURE 5.2 – Arbre construit avec l'algorithme de Barnes Hut avec 20000 particules

Il devient alors possible de pousser les simulations plus loin pour simuler notamment la formation de galaxies (spirales ou sphériques) en augmentant considérablement le nombre de particules. Cela montre alors l'intérêt et l'efficacité de l'algorithme de Barnes-Hut.

5.4 Vérification du modèle

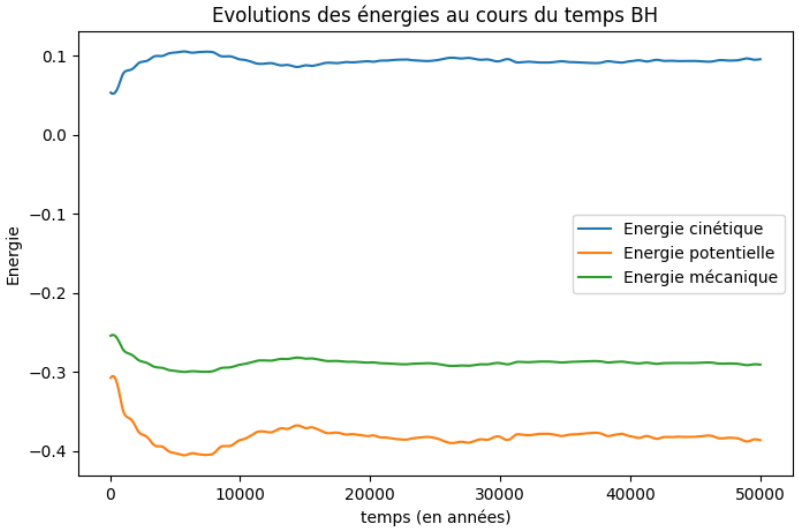


FIGURE 5.3 – Bilan énergétique méthode de Barnes-Hut
($\Delta t = 50$ ans, 1000 itérations et 2000 particules)

L'énergie mécanique se conserve à partir de la 260ème itération, avant ça, le modèle est perturbé par les conditions initiales. Ainsi, le modèle est bien valide physiquement malgré les approximations effectuées, ce qui montre la puissance de l'algorithme. Il a une vitesse de convergence située entre celles des deux méthodes naïves.

Partie 6

Parallélisation

Les programmes que nous écrivons sont 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 par exemple 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 simultanée. Ainsi, un programme parallèle pourra exécuter en même temps des processus définis de manière séquentielle et réduire son temps d'exécution.

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 qui vont exécuter en même temps des instructions similaires.

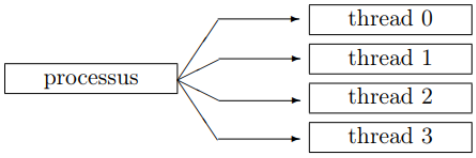


FIGURE 6.1 – Processus et threads
source : `https://ccub.u-bourgogne.fr`

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 constitué de régions séquentielles et de régions parallèles. 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 de s'assurer que chaque tâche peut s'effectuer sans requérir les autres, notamment au niveau de la mémoire partagée (par exemple que le calcul n°41 ne doit pas avoir besoin du calcul n°40, qui n'a peut-être pas encore été fait).

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'instructions 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() //renvoie le nombre total de threads utilisés
omp_set_num_threads(int) // définit le nombre de thread à utiliser dans une région parallèle
omp_get_thread_num() // renvoie le numéro du thread courant
```

OpenMp permet aussi de définir si une variable doit être privée ou partagée entre tous les threads en utilisant la clause *private*().

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 et la création de l'arbre, c'est donc ceux-ci que nous avons parallélisé. Nous avons fixé le nombre de threads à 4 et toutes les variables sont partagées entre les threads.

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

Partie 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 particules différents. Nous avons donc mesuré et sommé le temps concerné pour chaque itération afin d'effectuer une moyenne pour chaque nombre de particules. Pour chaque méthode, les paramètres physiques seront les mêmes. Pour les versions parallélisées, le nombre de threads utilisés est toujours fixé à 4.

7.1 Construction de l'arbre

Les tests de parallélisation de notre algorithme de construction du *quadtree* ont montré qu'il n'est pas parallélisable. En effet, effectuer les insertions en parallèle engendre des erreurs lors des calculs de force par l'utilisation de cet arbre, qui font diverger les particules. Cela pourrait s'expliquer par le fait que les insertions de particules sont dépendantes les unes des autres et ne peuvent donc être effectuées en parallèle au risque de fausser les calculs. Ainsi, pour pouvoir le paralléliser, il est nécessaire de modifier au préalable le fonctionnement de l'algorithme d'insertion.

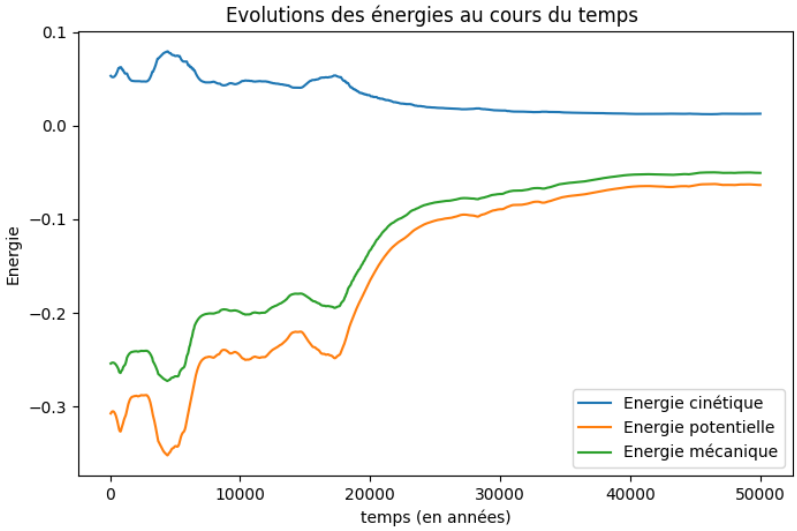


FIGURE 7.1 – Bilan énergétique pour la parallélisation de la construction de l'arbre ($\Delta t = 50$ ans, 1000 itérations et 2000 particules)

On peut voir sur la figure 7.1 que ces erreurs se traduisent simplement par l'absence de conservation de l'énergie mécanique au cours du temps, montrant ainsi que la parallélisation de la construction de l'arbre n'est pas utilisable.

7.2 Comparaisons des méthodes

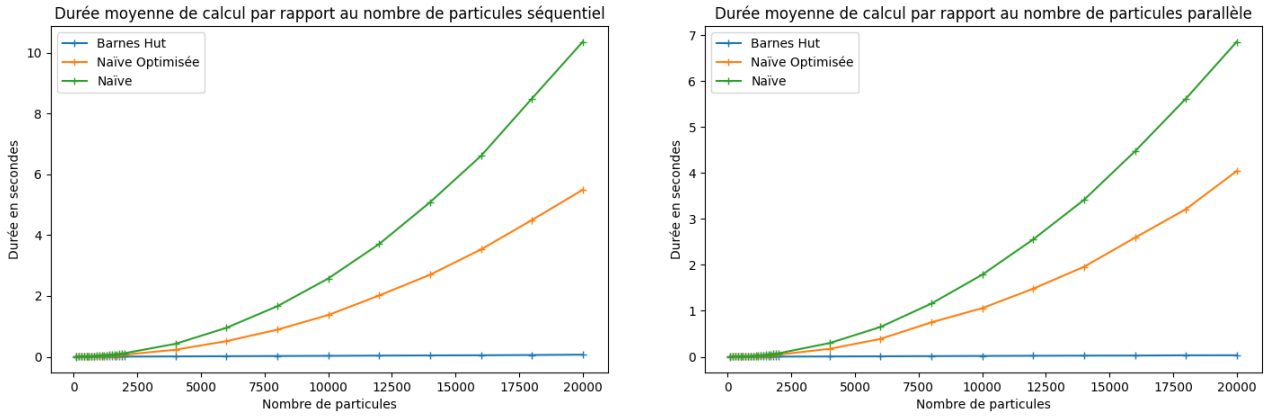


FIGURE 7.2 – Durée moyenne en fonction du nombre de particule

Les résultats confirment la théorie. Il est clair que les méthodes naïves ont une complexité en $O(N^2)$ alors que l'algorithme de Barnes-Hut a une complexité bien inférieure en $O(Nlog(N))$. On peut également confirmer que l'algorithme naïf optimisé a une meilleure complexité que la version naïve. De plus, on peut déjà remarquer que l'ordre de grandeur des durées est plus petit pour les versions parallélisées.

7.3 Séquentiel et parallèle

7.3.1 Naïve

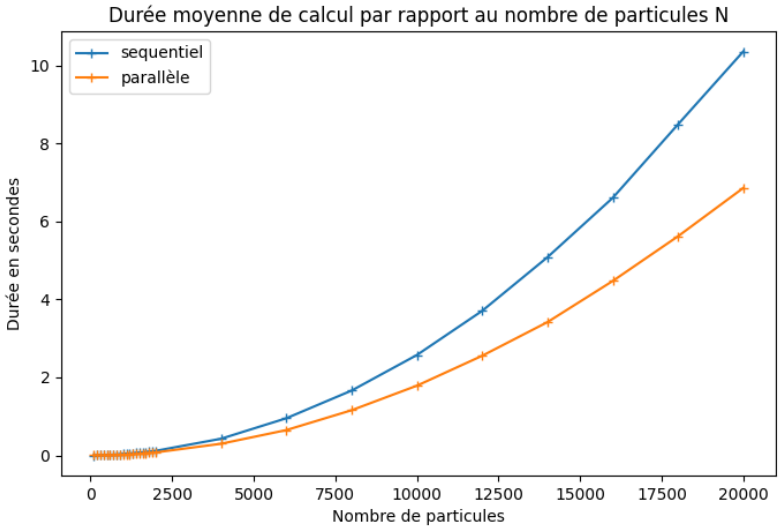


FIGURE 7.3 – Résultats pour l'algorithme naïf

La parallélisation fonctionne bien sur la méthode de calcul naïve, le calcul parallélisé étant 1.5 plus rapide que le calcul séquentiel. Cette amélioration n'a cependant aucun effet sur la complexité. Les résultats de la figure 4.3 ont été obtenus avec en utilisant cette version parallèle ce qui confirme le fait que la parallélisation soit valide.

7.3.2 Naïve optimisée

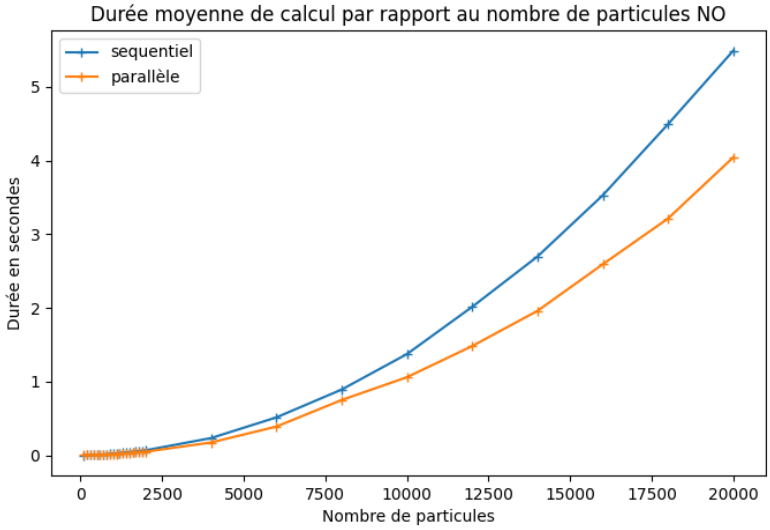


FIGURE 7.4 – Résultats pour l'algorithme naïf optimisé

La parallélisation fonctionne également bien sur la méthode de calcul naïve optimisée, le calcul parallélisé étant 1.4 plus rapide que le calcul séquentiel. La figure 4.4 a été obtenue en utilisant la version parallèle donc la méthode naïve optimisée reste cohérente physiquement malgré la parallélisation.

7.3.3 Algorithme de Barnes-Hut

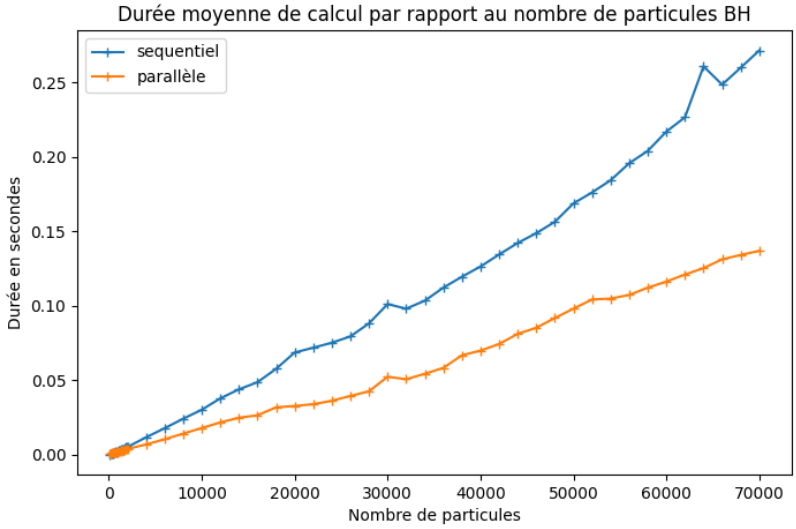


FIGURE 7.5 – Résultats pour l'algorithme de Barnes-Hut

On peut voir sur la figure que globalement la parallélisation améliore de manière conséquente l'efficacité des calculs de Barnes-Hut. En effet, la version parallélisée est 1.7 fois plus rapide que la version séquentielle. Dans les deux cas la complexité est en $O(log(N))$. De plus, les résultats de la figure 5.3 proviennent de la version parallélisée de l'algorithme, ainsi, la parallélisation respecte l'aspect physique du problème.

7.3.4 Résultats de la parallélisation

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 la parallélisation permet bel et bien d'optimiser les calculs à conditions de respecter les conditions de parallélisation. En moyenne, le gain de vitesse avec 4 threads est d'environ 1.53 ce qui est plutôt élevé. Afin d'améliorer notre parallélisation, nous aurions pu augmenter le nombre de threads ou équilibrer davantage la charge entre les différents threads.

Partie 8

Conclusion du projet

8.1 Conclusion

Le projet consistait à étudier différentes manières de résoudre numériquement le problème à N-corps afin d'obtenir la simulation la plus satisfaisante. Au final, nous avons implémenté 3 méthodes de calcul différentes pour les forces :

- la méthode naïve qui calcule pour chaque particule les forces qui s'y applique de la manière la plus basique possible
- la méthode naïve optimisée qui utilise le principe d'action-réaction afin de diminuer le nombre de calculs
- l'algorithme de Barnes-Hut qui utilise un arbre afin de réduire considérablement le nombre de calculs en utilisant des approximations.

Au final, l'algorithme de Barnes-hut est bien plus rapide et efficace que les autres, et malgré les approximations qu'il fait, il produit une évolution valide. Il constitue pour le moment la meilleure résolution approchée possible au problème à N-corps. De plus, la parallélisation permet d'avoir des performances bien plus intéressantes et montre notamment son intérêt lorsqu'on augmente le nombre de particules à plusieurs dizaines de milliers. Cependant, la parallélisation de l'insertion des particules dans l'arbre montre l'importance de la compatibilité des algorithmes, au risque d'avoir une parallélisation inefficace, voire même une inconsistance de nos calculs.

De nombreux axes de développement du projet sont possibles tel que l'optimisation de la parallélisation, l'utilisation d'une carte graphique pour simuler encore plus de particules, ou l'extension de la simulation en 3D, afin d'observer des structures qui ne sont pas nécessairement stables en 2D.

8.2 Impacts du projet

La résolution du problème à N-corps est extrêmement importante dans des domaines tels que l'astronomie. Elle permet aux scientifiques de ces domaines d'émettre des hypothèses ou de les vérifier de façon accessible et rapide, en modifiant les paramètres au besoin, du fait que les objets d'études des disciplines liées sont difficilement observables et manipulables.

De plus, l'optimisation du calcul a un fort impact sur la recherche dans ces domaines, le calcul haute-performance étant un nouvel enjeu de nos sociétés. En effet, la réduction des temps de calculs permet dans un premier temps d'augmenter les volumes de données traitées, ainsi que la précision des simulations. De plus, réduire les temps de calcul permet de réduire l'énergie dépensée, et donc de diminuer la pollution liée à la production énergétique. Ces améliorations sont indispensables à de plus grandes échelles.

8.3 Bilans personnels

8.3.1 Camille

Je pense que ce projet m'a beaucoup aidé en informatique, notamment par l'apprentissage d'un tout nouveau langage (C++). J'ai beaucoup aimé le principe de l'algorithme de Barnes-Hut et avoir travaillé et réfléchi dessus a été très formateur. Je pense avoir apporté des idées au groupe pour nos algorithmes, ainsi que des points de vue différents pour d'autres approches et une meilleure compréhension globale.

8.3.2 Noah

Ce projet, à l'intersection de la physique, des mathématiques et de l'informatique, nous a permis de développer nos compétences techniques en C++, parallélisation et optimisation. L'interdisciplinarité m'a vraiment plu car cela nous a permis de nous intéresser à beaucoup de chose différentes. Cependant, le fait que la majeure partie du code était déjà fournie nous a contraint à étudier le code pendant une longue période et nous a confronté à des difficultés d'implémentation de certaines fonctions. Je pense avoir su contribuer au développement du projet en suggérant des idées adaptées aux difficultés rencontrées.

8.3.3 Rudio

Au final, à travers ce projet, j'ai pu développer des connaissances dans différents domaines. Il m'a poussé à dépasser mes limites pour comprendre le fonctionnement du code qui était dans un langage que je ne maîtrisais pas et pour assimiler les aspects physiques et mathématiques du problème. Cela m'a donc permis de connaître les principaux aspects et concepts du C++ et de la programmation orientée objet mais aussi de la parallélisation multi-thread avec OpenMP. J'ai également pu améliorer mes connaissances en mécanique et en résolution numérique. Ainsi, malgré les difficultés rencontrées, j'ai vraiment apprécié le projet et son déroulement notamment grâce aux résultats que nous avons pu obtenir. De plus, le travail de groupe m'a plu étant donné que chaque membre du groupe s'est investi et a apprécié le projet. J'aurais cependant aimé pouvoir poussé le projet plus loin en explorant d'autres façons d'optimiser le programme.

8.3.4 Elyas

Le projet était exactement ce que j'espérais, le fait que nous avons procédé à la simulation d'objets célestes m'a permis d'apprendre plusieurs notions techniques (au niveau informatique) mais aussi scientifiques. L'utilisation du C++ a été aussi très intéressante cependant je regrette qu'on n'ait pas exploité plus ce langage. Ce projet m'a aussi permis de voir que l'utilisation des mathématiques est un outil très important notamment pour l'amélioration des calculs avec la mise en place des différents intégrateurs.

Annexes

1 Lien vers le repository Github

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

2 Différentes simulations

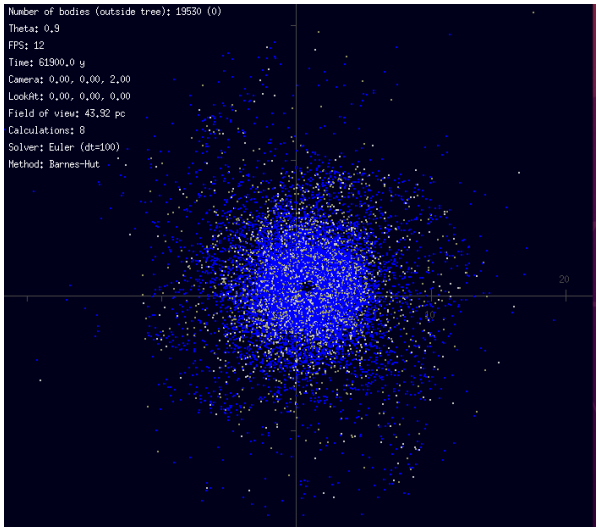


FIGURE 1 – Galaxie sphérique

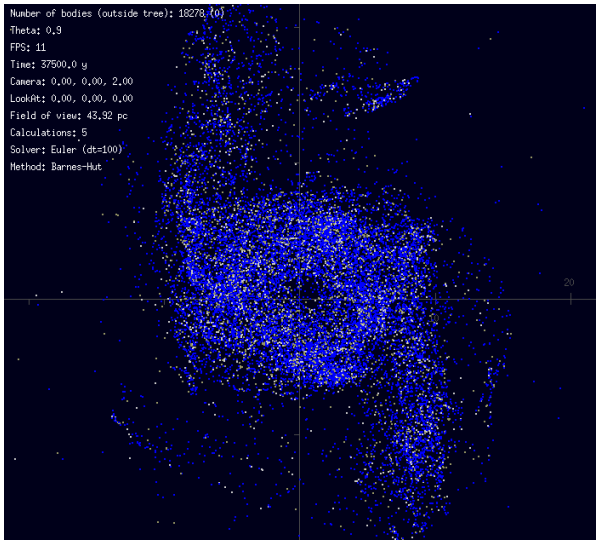


FIGURE 2 – Galaxie spirale

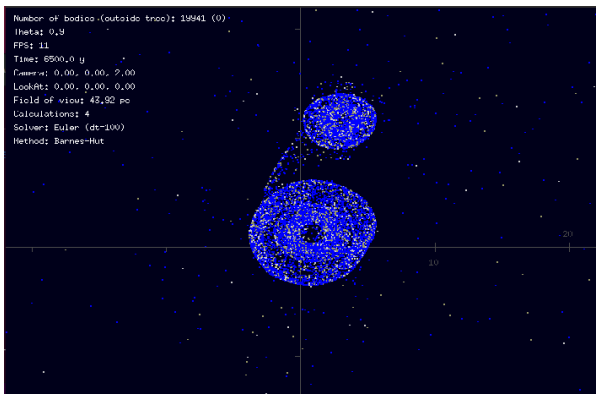


FIGURE 3 – 2 galaxies

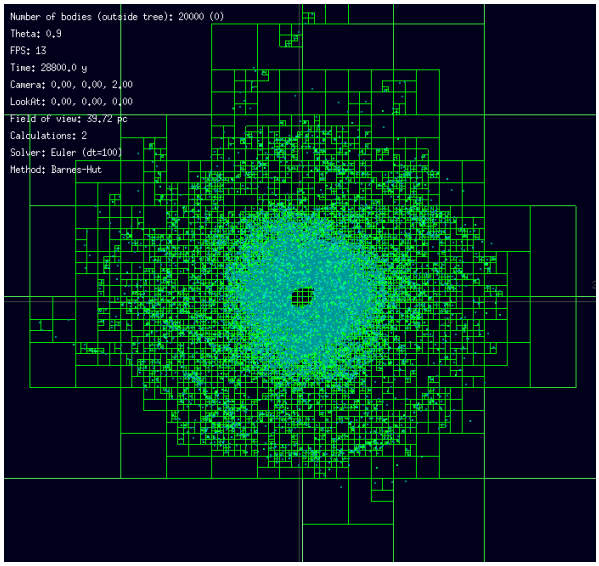


FIGURE 4 – Quadtree

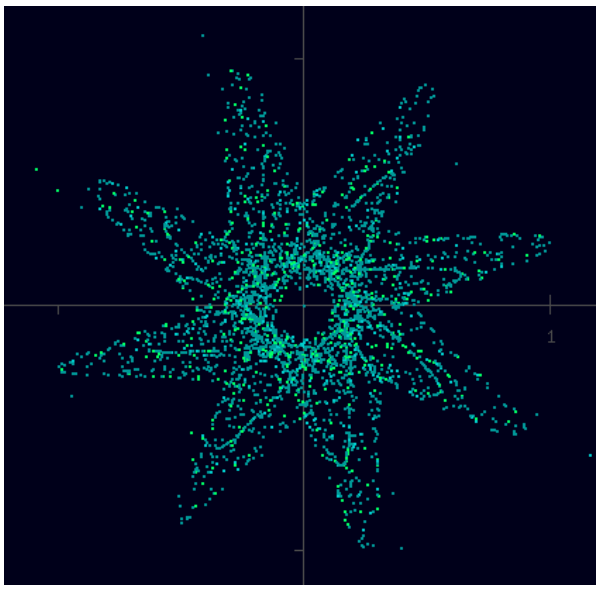


FIGURE 5 – Galaxie-atome

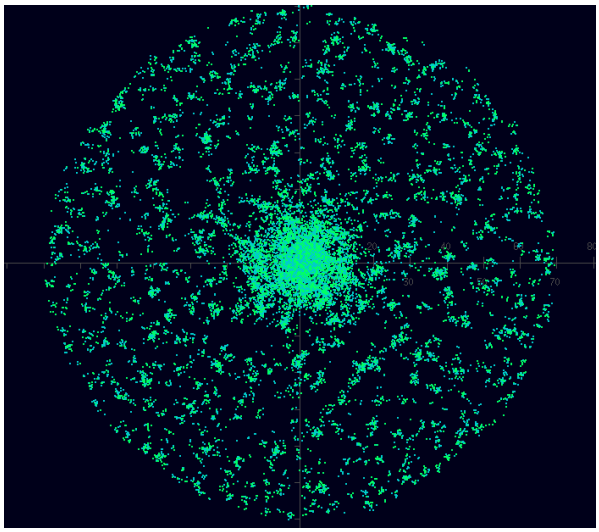


FIGURE 6 – Galaxie sphérique à gros diamètre

Bibliographie

[1] Mancheron Alban. Une introduction à la programmation parallèle avec open mpi et openmp, 2018.

[2] Josh Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 1986.

[3] Leah Birch, Elizabeth Finn, and Karen Yu. The n-body problem. 2012.

[4] Jalel Chergui and Pierre-François Lavallée. Openmp parallélisation multitâches pour machines à mémoire partagée, 2017.

[5] Centre de Calcul de l'université de Bourgogne. *Bases de la parallélisation multitâches : OpenMP (Open Multi-Processing)*. 2016.

[6] Marion Dierickx et Stephen Portillo. N-body building, December 2013.

[7] Gargne Fabien. Algorithme de barnes-hut pour le problème des n-body. 2005.

[8] Laurent Garcin. Méthode d'euler et gravitation, 2016.

[9] O. Marguin. *C++ : LES BASES*. 2003/2004.

[10] Philip Mocz. Create your own n-body simulation (with python), 2020.