Rapport de TIPE informatique : Simulation des systèmes gravitationnels et application à la cosmologie

Mattéo Clémot

Juin 2018

1 Mise en place de la simulation

1.1 Cadre

On considère un système gravitationnel, dont l'échelle peut aller de celle d'un amas d'étoile (10^{18} m) à celle de l'univers observable (10^{26}m) en passant par celle d'une galaxie (10^{21} m). On suppose que ce système obéit à la mécanique newtonienne. En dépit de la simplicité des lois, le problème n'admet la plupart du temps pas de solution analytique exacte. Afin de comprendre les phénomènes, une phase de simulation est souvent nécessaire.

Souvent, le nombre de constituants du système n'est pas atteignable avec les ordinateurs actuels : il y a de l'ordre de 10^{11} étoiles dans la Voie Lactée par exemple. La centaine de milliards de corps a d'ailleurs été atteinte en 2012 par l'Institut Max-Planck de physique [1]. Par conséquent, on adopte la notion de *corps virtuel*, représentant une certaine quantité de matière, et n'ayant pas de réalité physique. En pratique on a souvent :

$$1 \ll N_{\text{simulé}} \ll N_{\text{réel}}$$

Dans le modèle ponctuel, le potentiel et la force d'interaction gravitationnelle s'écrivent respectivement :

$$\Phi(r) = -\mathcal{G}\frac{m}{r}$$

$$\overrightarrow{F_{i\to j}} = -\mathcal{G}\frac{m_i m_j}{\|\overrightarrow{P_iP_j}\|^3}\overrightarrow{P_iP_j}$$

Or dans la simulation, les corps virtuels peuvent être arbitrairement proches, la force d'interaction étant alors arbitrairement grande. Ceci entraine des accélérations irréalistes pour des corps virtuels représentant une distribution diffuse de masse. On adopte donc le modèle de *Plummer* qui modifie le potentiel newtonien aux faibles distances. Dans ce modèle, le potentiel et la force s'écrivent :

$$\Phi_P(r) = -\mathcal{G}\frac{m}{\sqrt{r^2+\epsilon^2}}$$

$$\overrightarrow{F_{i\to j}} = -\mathcal{G}\frac{m_i m_j}{(\|\overrightarrow{P_iP_j}\|^2+\epsilon^2)^{3/2}}\overrightarrow{P_iP_j}$$

où ϵ est la longueur caractéristique de l'adoucissement du potentiel. Elle doit être choisie en fonction du système pour minimiser l'erreur. Il est montré dans [2] que ϵ doit être environ deux fois plus petite que la distance moyenne entre deux corps dans les régions les plus denses.

1.2 Discrétisation

L'écriture, pour chacun des N corps de la simulation, de la définition de la vitesse d'une part, et de la seconde loi de Newton d'autre part, conduit au système de $2N$ équations différentielles vectorielles suivant :

$$\begin{cases} \forall j \in [\![1, N]\!], \frac{\mathrm{d}\overrightarrow{r_j}}{\mathrm{d}t} = \overrightarrow{v_j} \\ \forall j \in [\![1, N]\!], m_j \frac{\mathrm{d}\overrightarrow{v_j}}{\mathrm{d}t} = m_j \overrightarrow{a_j} = \sum_{\substack{i=1 \\ i \neq j}}^{N} \overrightarrow{F_{i \to j}} \end{cases}$$

De nombreuses méthodes existent pour discrétiser ces équations différentielles dans le but de les résoudre numériquement, la plus simple étant la méthode d'Euler. Cependant, les systèmes étant ici conservatifs, on s'attache à la conservation de l'énergie, entre autres. La méthode de Verlet (également connu sous le nom d'intégrateur *leapfrog*) paraît ici adaptée au problème [3, 4] :

— elle conserve mieux l'énergie que la méthode d'Euler, et même que celles de Runge-Kutta, grâce à son caractère réversible qui en fait un intégrateur dit *symplectique* ;
— elle nécessite une seule évaluation de force par étape et par corps, comme la méthode d'Euler, ce qui est intéressant pour garantir de bonnes performances.

Étant donné le temps de discrétisation τ que l'on garde constant au cours de la simulation, le schéma d'intégration s'écrit avec cette méthode :

$$\begin{cases} \overrightarrow{r_{n+1}} &= \overrightarrow{r_n} + \tau \overrightarrow{v_n} + \frac{\tau^2}{2} \overrightarrow{a_n} \\ \overrightarrow{v_{n+1}} &= \overrightarrow{v_n} + \frac{\tau}{2} (\overrightarrow{a_n} + \overrightarrow{a_{n+1}}) \end{cases}$$

Il est donc nécessaire de garder en mémoire la dernière accélération calculée pour tous les corps.

1.3 Performance

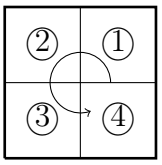
L'algorithme naïf de simulation du problème à N-corps, dit algorithme *PP* pour *Particle-Particle* [3, 5], consiste à calculer l'interaction entre chaque paire de particule. On a alors $\binom{N}{2} = \frac{N(N-1)}{2}$ évaluations de force par étape, d'où une complexité quadratique. En pratique, il est inutilisable en tant que tel sur un ordinateur personnel actuel dès 10^5 corps. À l'exception du cas des petits systèmes (amas ouverts...), l'algorithme naïf n'est donc pas pertinent. Ainsi, quitte à faire des approximations raisonnables, on souhaite mettre en œuvre un algorithme asymptotiquement meilleur.

2 Principe de Barnes-Hut

2.1 Généralités

En 1986, Piet Hut et Josh Barnes donnent leur nom à une méthode de résolution approchée du problème à N corps [6]. Il consiste à effectuer une division récursive de l'espace en cubes, en construisant un arbre de type *octree*, c'est-à-dire dont tous les nœuds sont d'arité 0 ou 8, de telle sorte que les feuilles de cet arbre ne contiennent au plus qu'un corps.

Pour des raisons de simplicité de représentation, j'implémente la simulation dans le plan ; les algorithmes et structures de données se transposent sans difficulté d'un cas à l'autre. En particulier, on utilisera donc des *quadtrees* plutôt que des *octrees* pour diviser récursivement le plan en quatre quadrants. En outre, on utilisera la numérotation arbitraire des quadrants ci-contre.



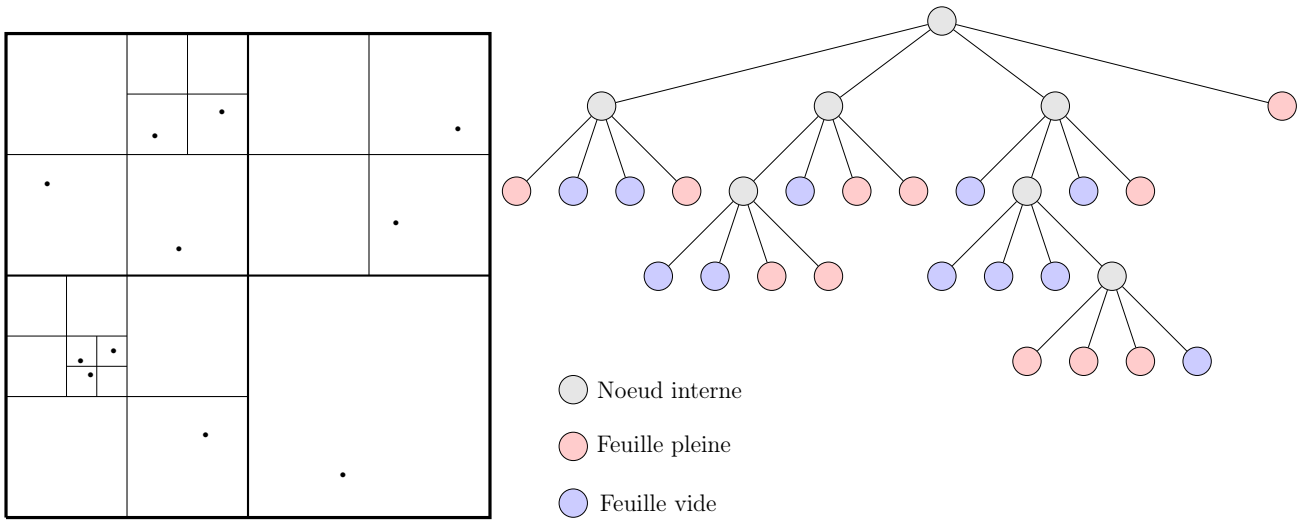


FIGURE 1 – Exemple de partition du plan pour un système à 11 corps

Par souci de performance, j'implémente les algorithmes en C++, dont j'exploite le paradigme orienté objet par la création d'une classe `Quadtree` (C.2) permettant la représentation de tels arbres.

2.2 Construction de l'arbre

La construction de l'arbre consiste à, à partir d'une racine vide dont le quadrant contient tous les corps du système, insérer un par un les corps, en les *descendant* récursivement dans le bon quadrant, jusqu'à ce qu'ils soient seuls dans leur nœud, quitte à étendre l'arbre en fin de descente. Il faut également mettre à jour le barycentre de chaque nœud par lequel on est passé, grâce à la formule :

$$\overline{x}^{(k+1)} = \frac{k\overline{x}^{(k)} + x_{k+1}}{k+1}$$

où k est le nombre de corps dans le nœud avant insertion du nouveau, et $\overline{x}^{(k)}$ et $\overline{x}^{(k+1)}$ le barycentre du nœud respectivement avant et après insertion.

Chaque descente se fait en $\mathcal{O}(h)$ où h est la hauteur de l'arbre. Avec une distribution pas trop hétérogène, on peut raisonnablement supposer $h = \mathcal{O}(\ln N)$, d'où un coût de construction de l'arbre en $\mathcal{O}(N \ln N)$. Une implémentation de cet ajout d'un nouveau corps est donnée en (C.4).

2.3 Évaluation de force

L'évaluation de la force subie par un corps est fondée sur un parcours en profondeur *avec sauts* de l'arbre. Elle s'appuie sur un critère d'acception contrôlé par un paramètre θ appelé *angle d'ouverture* qui traduit le degré d'approximation. En pratique, on choisit souvent θ entre 0.5 et 0.9 [7].

Soit un corps $i \in [\![1, N]\!]$. Lors du parcours de l'arbre, dès que l'on parvient à un nœud vérifiant la condition

$$\frac{d}{r} \leqslant \theta$$

où d est la taille du côté du quadrant associé au nœud et r la distance de son barycentre au corps i, on approxime la force exercée sur i par les corps dans le nœud, par celle exercé sur i par son barycentre. On passe dans ce cas directement au nœud voisin, d'où le terme "saut". Une implémentation de l'évaluation récursive d'une force est donnée en (C.5).

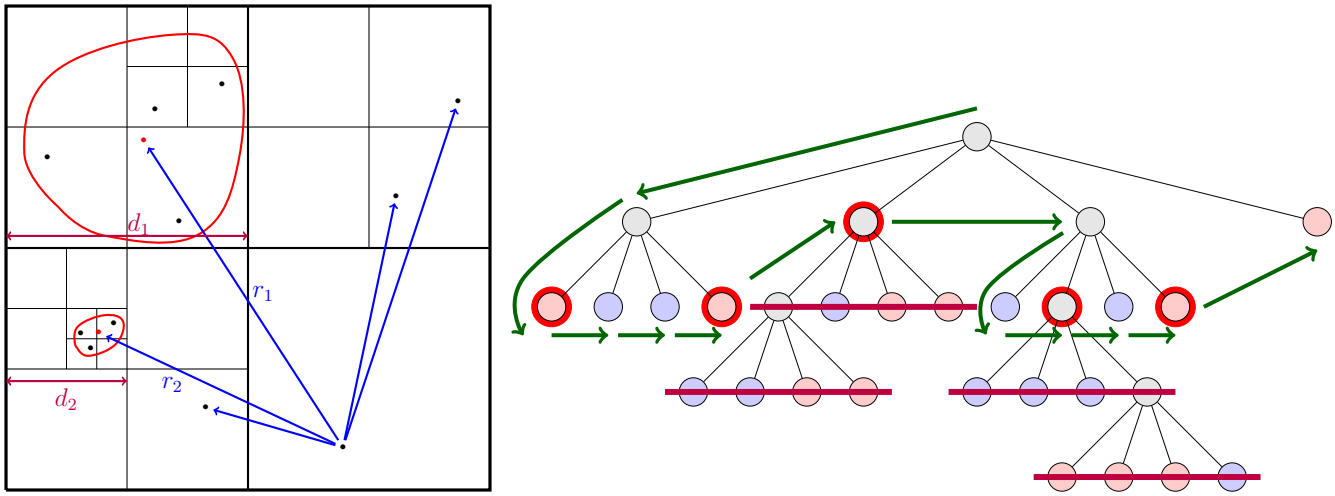


FIGURE 2 – Exemple où 5 forces sont évaluées au lieu de 10

3 Adaptation au parallélisme

3.1 Principe et contraintes du parallélisme

Dans la simulation du problème à N-corps, à une étape donnée, le calcul de l'accélération d'une particule i ne dépend pas du calcul de celle d'une autre particule j. La technique du parallélisme est donc particulièrement adaptée ici. Afin d'exploiter l'architecture parallélisée des ordinateurs actuels (microprocesseurs multicœurs, processeurs graphiques), on répartit le calcul des N accélérations sur plusieurs cœurs d'exécution.

À cette fin, j'utilise la classe fournie par C++11 `std::thread`, qui est limitée au CPU, puis la bibliothèque OpenCL.

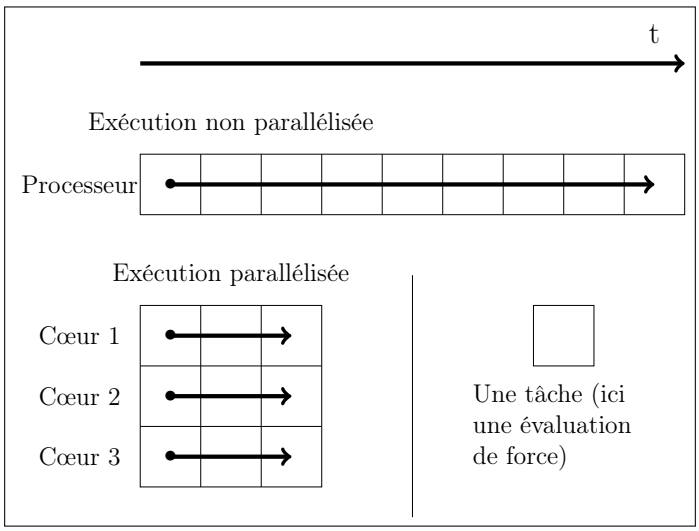


FIGURE 3 – Schéma montrant le principe du parallélisme en informatique

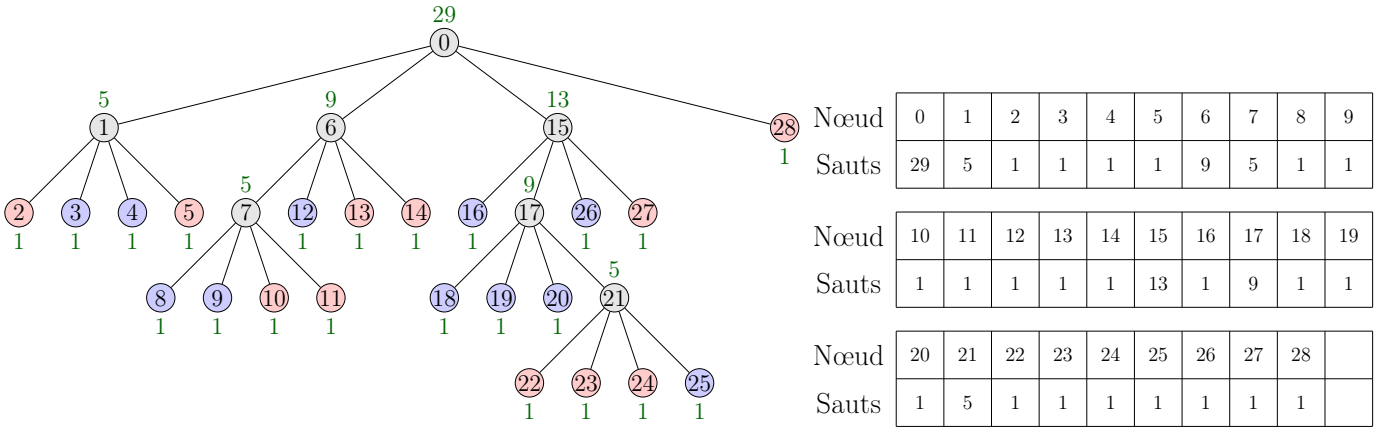
3.2 Linéarisation de l'arbre

On se propose de changer la manière de représenter en machine le *quadtree* de la simulation, en effet :

— le parcours d'un arbre représenté par une classe récursive est plus lent que le même arbre représenté par un tableau ;

— OpenCL n'est pas orienté objet.

Plus précisément, on essaye de *linéariser* l'arbre en le représentant par un tableau dynamique (dans l'implémentation, `std::vector`), sans compromettre la possibilité de faire des sauts évoqués en (2.3). Ce tableau doit contenir les nœuds de l'arbre ordonnés par un parcours en profondeur préfixe. Chaque élément (de type `Quadnode` défini en (C.1)) contient les propriétés intrinsèques du nœud (longueur du côté, coordonnées du barycentre, nombre de corps) mais également, afin de permettre les sauts, le nombre de nœuds dans le sous-arbre dont il est racine. Ce nombre étant obtenu après l'appel récursif sur les quatre enfants, il faut garder en mémoire la taille du tableau en cours en construction avant le rajout du nœud, qui n'est autre que l'indice de ce nœud dans le tableau final, pour la réutiliser après l'appel récursif en complétant l'attribut `jumps`. Une implémentation est donnée en (C.6).



Nœud	0	1	2	3	4	5	6	7	8	9
Sauts	29	5	1	1	1	1	9	5	1	1

Nœud	10	11	12	13	14	15	16	17	18	19
Sauts	1	1	1	1	1	13	1	9	1	1

Nœud	20	21	22	23	24	25	26	27	28	
Sauts	1	5	1	1	1	1	1	1	1	

FIGURE 4 – Tableau construit par le processus de linéarisation

L'évaluation d'une force se fait alors en parcourant le tableau obtenu, en sautant autant de cases que stocké dans `jumps` dès que la condition $\frac{d}{r} \leqslant \theta$ pour un nœud est vérifiée.

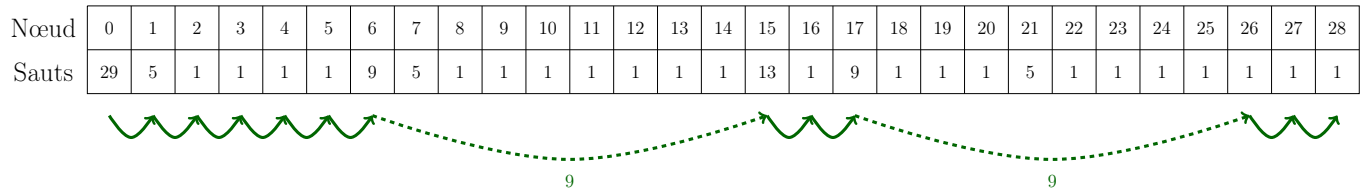


FIGURE 5 – Évaluation avec l'arbre linéarisé de la même force que précédemment

Un kernel OpenCL de l'algorithme de Barnes-Hut avec arbre linéarisé est donné en (C.7).

4 Étude de l'algorithme

L'angle d'ouverture θ permet de contrôler le niveau d'approximation : plus il est petit, plus l'algorithme de Barnes-Hut est précis, mais en contrepartie, le temps d'exécution est plus long car plus de forces sont calculées.

On remarque en outre que le cas limite $\theta = 0$ correspond en fait à l'algorithme naïf, puisqu'aucune approximation n'est effectuée, l'arbre étant alors à chaque évaluation totalement exploré.

4.1 Complexité temporelle

Déterminons la complexité temporelle d'une évaluation de force dans le cas $\theta \geqslant 1$: si un nœud n'est pas adjacent à celui du même niveau où se trouve le corps en cours de traitement, alors on a $d \leqslant r$ avec les notations précédentes, d'où $\frac{d}{r} \leqslant 1 \leqslant \theta$: il y a donc approximation et le sous-arbre partant de ce nœud n'est pas exploré. À chaque niveau de l'arbre, on a donc au plus 8 nœuds qui ne

vont pas faire l'objet d'une approximation et dont le sous-arbre va être exploré. Avec une hauteur en $\mathcal{O}(\ln N)$, la complexité d'une évaluation de force est alors en $\mathcal{O}(8 \ln N) = \mathcal{O}(\ln N)$. La complexité de l'évaluation de toutes les forces est donc en $\mathcal{O}(N \ln N)$. En ajoutant la construction du *quadtree* de même complexité temporelle, on obtient comme coût total de l'algorithme de Barnes-Hut $\mathcal{O}(N \ln N)$.

Le cas $\theta < 1$ est plus délicat. Il a été montré numériquement dans [8] que la complexité reste en $\mathcal{O}(N \ln N)$ pour $\theta \gtrsim 0,3$.

4.2 Résultats expérimentaux

J'ai mesuré le temps d'exécution de l'algorithme naïf et de celui de Barnes-Hut pour différentes valeurs de θ, pour une implémentation avec `std::thread` s'exécutant sur un processeur Intel® Core™ i5-4200H (2 cœurs à 3.4 Ghz), et pour une implémentation en OpenCL s'exécutant sur un processeur graphique Nvidia GeForce GTX 850M (640 cœurs à 0.9 Ghz). Les résultats sont donnés en Annexe A et semblent en accord avec les résultats théoriques.

4.3 Précision

Un développement quadrupolaire du potentiel des corps d'un nœud par rapport à leur barycentre est nécessaire pour trouver l'erreur effectuée par les approximations. En pratique, cette erreur est déjà acceptable lorsque $\theta \leqslant 1$. Barnes et Hut avancent dans [6] une précision de l'ordre de 1% pour $\theta = 1$.

5 Optimisation par une procédure de mise à jour

Les essais montrent que le calcul de l'arbre constitue une part non négligeable du temps d'exécution de chaque étape. Or il s'agit d'une étape dont la parallélisation massive paraît difficile.

Par ailleurs, on remarque que lorsque le temps de discrétisation de la simulation est choisi pertinemment, le déplacement de chaque corps à chaque étape est petit comparé à la taille du système. On essaye alors d'exploiter cette propriété pour mettre à jour efficacement l'arbre plutôt que le recalculer à chaque étape.

5.1 Description de l'algorithme

L'algorithme est fondé sur un parcours en profondeur de l'arbre à mettre à jour. Il consiste à faire *remonter* les corps qui ne sont plus dans le bon quadrant jusqu'à ce qu'il le soient, puis les insérer à partir de ce quadrant comme lors de la construction de l'arbre.

On utilise la structure de liste chaînée (`std::list` dans l'implémentation proposée) pour représenter les corps qu'il faut faire remonter, en effet :

— on n'a pas besoin d'accéder arbitrairement aux éléments de la liste, mais simplement d'en effectuer un parcours ;

— on aura besoin de supprimer certains éléments lors du parcours, ce que permet la structure de liste chaînée en temps constant, à l'inverse des tableaux par exemple.

Lorsqu'on arrive sur une feuille contenant un corps, on lit les nouvelles coordonnées de ce dernier grâce à l'attribut `_index` qui était mémorisé. S'il est toujours dans le quadrant, on met simplement à jour les coordonnées du barycentre de la feuille et on renvoie une liste vide. Dans le cas contraire, on met `_n` à 0 et on renvoie la liste ayant pour seul élément le corps dans la feuille.

Pour un nœud interne, on applique récursivement la procédure de mise à jour à ses quatre enfants et on concatène les quatre listes obtenues contenant les corps qui ont été remontés. On parcourt alors cette liste : si un corps est dans le nœud, on l'insère dans le bon quadrant et on le supprime de la liste. On finit par mettre à jour le barycentre du nœud grâce à celui de ses enfants.

5.2 Résultats

Avec cette méthode, on ne peut adapter l'arbre à l'extension spatiale du système, puisque les coordonnées des quadrants restent fixes lors de la mise à jour. Si un corps sort du quadrant associé à la racine de l'arbre, l'algorithme ne fonctionne pas. Il faut donc surveiller les bornes spatiales du système et recalculer si nécessaire l'arbre, en changeant la taille du quadrant de la racine afin de contenir tout le système.

Expérimentalement, pour une distribution uniforme, la mise à jour de l'arbre est de l'ordre de 5 fois plus rapide que son calcul total.

A Résultats expérimentaux

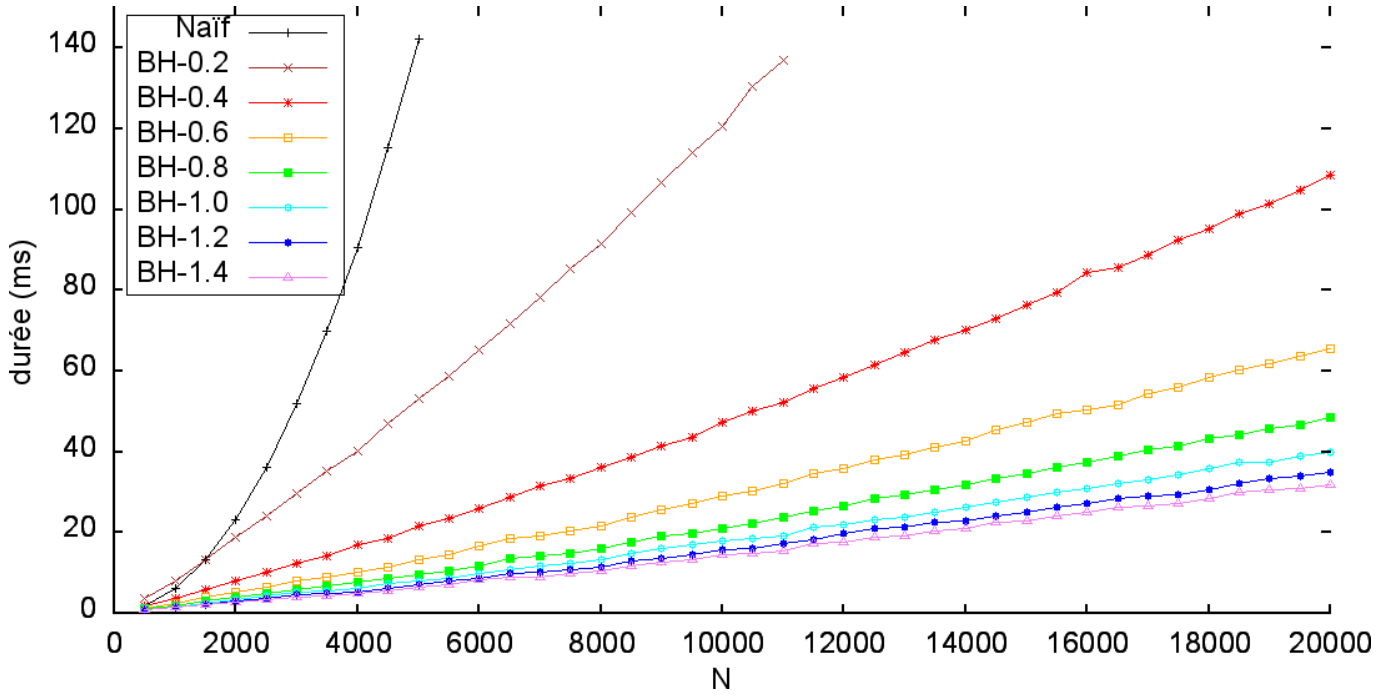


FIGURE 6 – Mesures pour N variant de 500 à 20000 effectuées avec l'implémentation utilisant `std::thread` sur un processeur Intel® Core™ i5-4200H (2 cœurs à 3.4 Ghz)

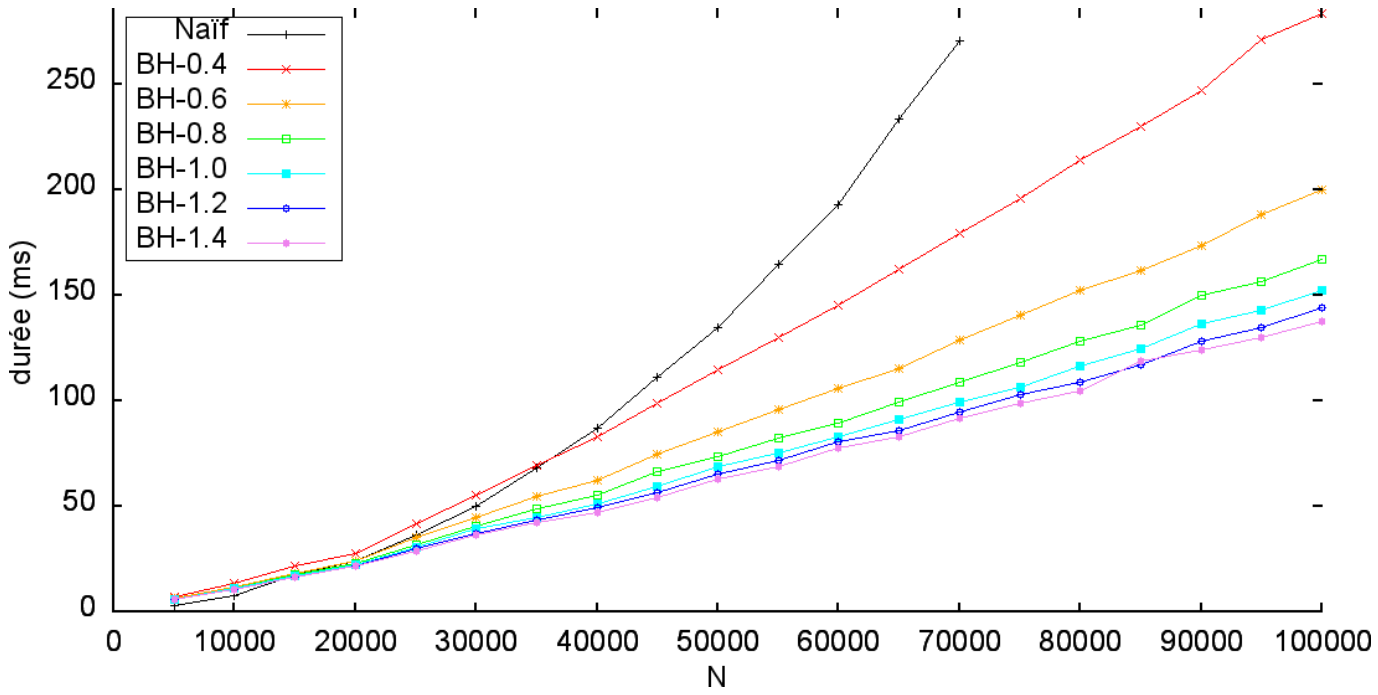


FIGURE 7 – Mesures pour N variant de 5000 à 100000 effectuées avec l'implémentation en OpenCL sur un processeur graphique Nvidia GeForce GTX 850M (640 cœurs à 0.9 Ghz)

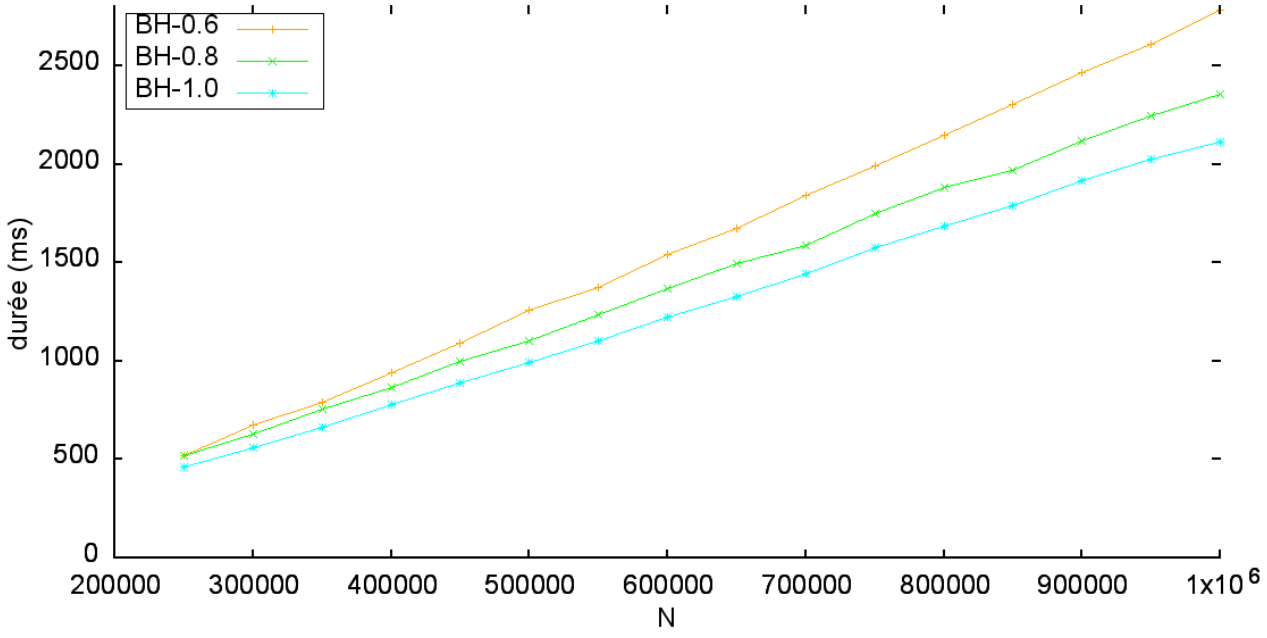


FIGURE 8 – Mesures pour N variant de 250000 à 1000000 effectuées avec l'implémentation en OpenCL sur un processeur graphique Nvidia GeForce GTX 850M (640 cœurs à 0.9 Ghz)

B Exemple de simulation du problème à N-corps

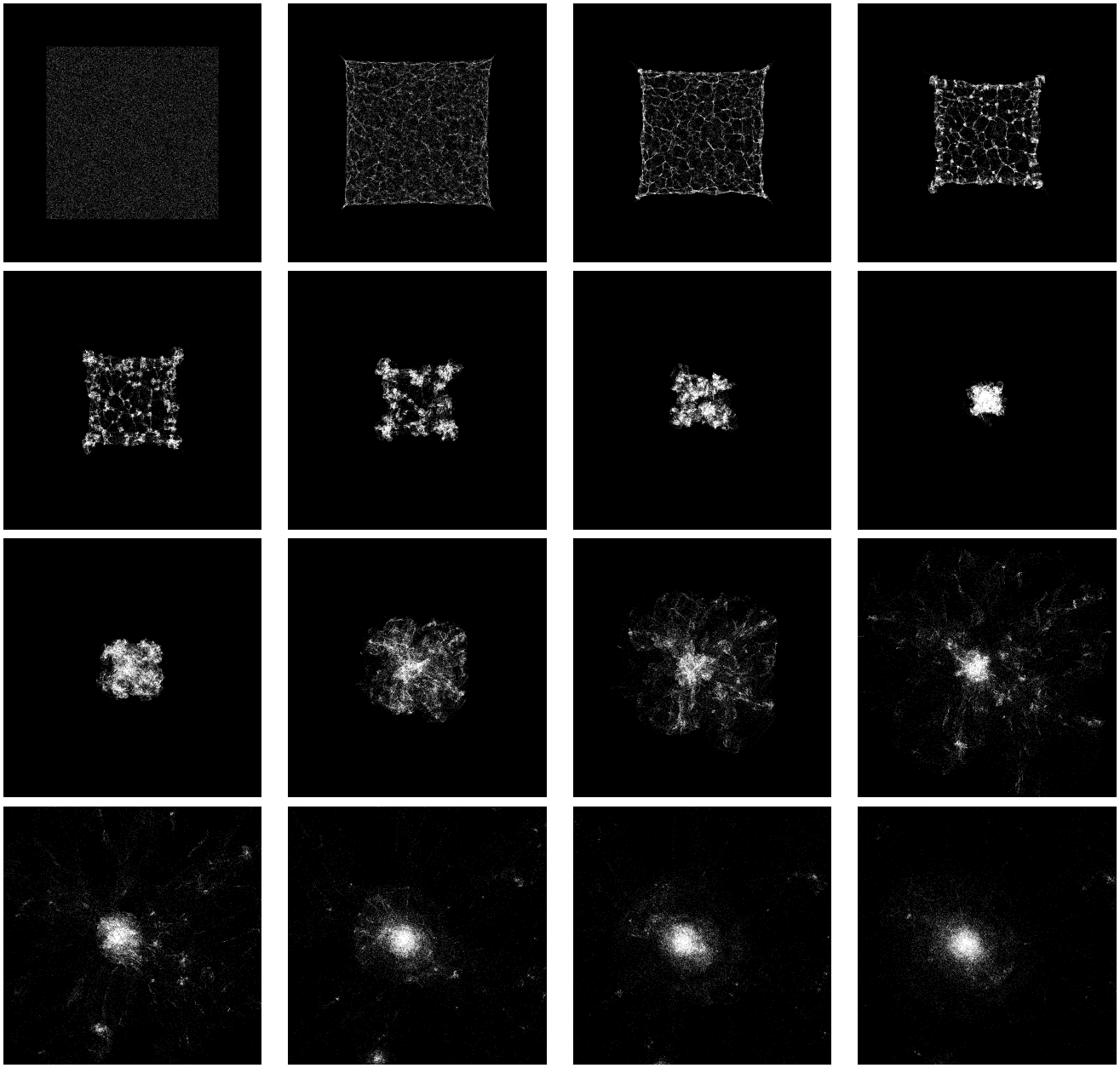


FIGURE 9 – Simulation effectuée à partir d'une distribution initiale statique uniforme sur un carré.

Références

[1] Max Planck Institute for Astrophysics : About the millennium-xxl simulation. http ://galformod.mpa-garching.mpg.de/mxxlbrowser/.

[2] S.A. Rodionov et N.Ya. Sotnikova : Optimal choice of the softening length and time step in n-body simulations. *Astronomy Reports*, 49(6):470–476, 2005.

[3] Tancred Lindholm : N-body algorithms. *In Seminar Presentati*, pages 1651–1656, 1999.

[4] J. Roussel : Méthode de verlet. FEMTO, la physique enseignée, 2014.

[5] Michele Trenti et Piet Hut : Gravitational n-body simulations. *arXiv preprint arXiv :0806.3950*, 2008.

[6] Josh Barnes et Piet Hut : A hierarchical O(N log N) force-calculation algorithm. *nature*, 324:446–449, 1986.

[7] Pierre Fortin : *Algorithmique hiérarchique parallèle haute performance pour les problèmes à N-corps*. Thèse de doctorat, Université Sciences et Technologies-Bordeaux I, 2006.

[8] Lars Hernquist : Performance characteristics of tree codes. *The Astrophysical Journal Supplement Series*, 64:715–734, 1987.