THE BARNES-HUT METHOD

High Performance Programming

Yangmei Lin

VT 2024

1 Introduction

Gravitational N-body Simulations, which involve numerical solutions of the equations of motion for N particles interacting gravitationally, are widely used to solve astrophysical problems[1].

This project is to simulate the evolution of N particles in a gravitational simulation using Newton's law of gravitation within two dimensions. The straightforward solution involves computing the force between each pair of particles, resulting in a time complexity of $\mathcal{O}(N^2)$. An alternative approach is partitioning particles into different groups. For each particle i, the forces exerted on it can be approximated by considering equivalent objects representing these groups.

The Barnes-Hut algorithm is founded on this concept, using a quadtree to organize particles. The calculation process can be done by traversing the tree. The method can efficiently reduce the complexity to $\mathcal{O}(NlogN)$ on average.

2 Problem Description

The force exerted on particle i by j is given by

$$\mathbf{f}_{ij} = -\frac{Gm_i m_j}{r_{ij}^3}\mathbf{r}_{ij} = -\frac{Gm_i m_j}{r_{ij}^3}\hat{\mathbf{r}}_{ij} \tag{1}$$

where G (set as $100/N$ in this project) is the gravitational constant, m_i and m_j are the masses of the particle i and j. r_{ij} gives the distance between the particles, and $\mathbf{r_{ij}}$ is the vector from i to j. The normalized vector is represented by $\hat{\mathbf{r}}_{ij}$.

For stability, the expression used for calculating forces is a modified version called Plummer spheres, which introduces a small number ϵ_0 (set as 10^{-3} in this project) to avoid the forces becoming too large when $r_{ij} \ll 1$

$$\mathbf{F_i} = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij} \tag{2}$$

The Barnes-Hut algorithm is applied to compute forces, and the second-order Velocity Verlet method (see Eq.3) is used for time integration. Since the calculation the algorithm is doing is an approximation, errors are inevitable. However, by selecting appropriate parameters - the threshold θ_{max} for group division in the Barnes-Hut algorithm and the time step size Δt - these errors should be constrained below an expected value. The objective is to determine optimized values for these variables that meet the specified error tolerance.

$$\begin{aligned}
\mathbf{x}(t+\triangle t) &= \mathbf{x}(t) + \mathbf{v}(t)\triangle t + \frac{1}{2}\mathbf{a}(t)\triangle t^2 \\
\mathbf{a}(t+\triangle t) &= \frac{\mathbf{F}(t+\triangle t)}{m_i} \\
\mathbf{v}(t+\triangle t) &= \mathbf{t} + \frac{\mathbf{a}(t)+\mathbf{a}(t+\triangle t)}{2}\triangle t
\end{aligned} \tag{3}$$

After successfully implementing the algorithm, the code should be optimized using appropriate optimization tools and parallelized to minimize execution time as much as possible.

3 Solution Method

3.1 Description of the Algorithm and Implementation

The implementation of the Barnes-Hut algorithm mainly includes three steps:

- Building a quadtree for particles
- Computing the mass and centroid of each sub-square
- Computing forces exerted on each particle by traversing the quadtree

3.1.1 Data Struture

A particle is characterized by six properties, representing positions, mass, velocities, and brightness separately. Six separate arrays are declared to store the values retrieved from the given file such that only the necessary properties for computation are passed into corresponding functions. Additionally, some auxiliary arrays are used to store the current forces on particles for parallel computation and some for storing acceleration for second-order time iteration.

The node of the quadtree should have pointers pointing to its child nodes and variables describing particle properties, such as positions and masses. An integer variable named `PID` records the index of the particle in the original file and serves to differentiate between leaf nodes (representing particles) and non-leaf nodes (representing objects). Since all particle nodes have non-negative indices, non-leaf nodes are designated with an ID of "-1". For non-leaf nodes, the variables `pos_x`, `pos_y` denote the centroid coordinates, while `mass` represents the combined mass of particles within the object. Additionally, in the tree, each node represents a square region, so four variables are used to define the shape of squares.

3.1.2 Building the Quadtree

The `insert()` function is used to insert a particle into the quadtree. It accepts a pointer to a `(TNode*) tNode` and four parameters representing the particle's positions, mass, and index. This insertion process operates recursively, with `tNode` representing the tree or subtree where the particle is to be inserted.

The function starts by checking whether `tNode` represents a particle. If it does, it compares the position of the particle to be inserted with the position of the particle in `tNode`. If they have the same position, the function prints an error message and terminates the simulation.

For a node to represent a particle, the square it occupies requires to be split evenly into four smaller squares, as each square should contain only one particle. The particle in `tNode` is subsequently inserted into one of its subtrees. The index of the child node into which the particle should be inserted is calculated using a simple equation. A new `TNode` is then created at that index, and the properties of the particle in `tNode` are copied to the new node. Finally, `tNode` transitions to a non-leaf state by setting `PID` as "-1".

After the split is complete, the `tNode`, now a non-leaf node, updates the properties of the object it represents - the mass and centroid. Next, the index of the subtree where the particle will be inserted is calculated. If the subtree at that index is `NULL`, the particle can be inserted directly. Otherwise, the `insert()` function is recursively called on the appropriate child node.

3.1.3 Forces Computation by Pre-order Traversal

The computation of forces is the critical part of this algorithm. For each particle, the quadtree is traversed once. If `tNode` represents a particle, the calculation can be done directly. If not, a threshold called θ, calculated using Eq.4, is used to determine if the `tNode`'s region is sufficiently far from the particle (by comparing θ to θ_{max}), such that it can be treated as a single point mass. If this condition is met, the traversal is stopped, and the force is computed. Otherwise, the function recursively calls itself on each of `tNode`'s child nodes.

$$\theta = \frac{\text{width of current box containing particles}}{\text{distance from particle to center of box}} \tag{4}$$

3.2 Optimization and Parallelization

3.2.1 Optimization for Calculation Load

Division operations are generally less efficient than addition and should be minimized to improve efficiency. One approach is to transform division operations into addition. For example, we pre-calculate the reciprocal mass of particles and utilize it in force computations instead of performing division operations during runtime. This can enhance performance by reducing the computational overhead associated with division operations.

3.2.2 Reducing Branching by Optimizing If Statements

The presence of if statements can introduce branches in procedures. Many `if` statements can be avoided by using logical expressions. This approach is applied when calculating the insertion position of particles. If avoiding branches is challenging, the `switch-case` construct is often more efficient than `if-else-if` statements.

3.2.3 K-Means Clustering for Better Cache Performance

When attempting to parallelize the force computation process, the particles are distributed among threads. However, when particles with contiguous indices are not spatially close, threads responsible for processing these distant particles may traverse divergent branches of the tree structure. This can lead to the repeated loading of identical data due to the finite capacity of cache memory. Conversely, parallel processing for neighboring particles ensures traversal along consistent branches of the tree, thereby enhancing the likelihood of cache data reuse and fostering an improvement in overall computational efficiency [2].

For improved partitioning, we employ K-means, a widely used clustering algorithm, to group particles. The fundamental concept and pseudo-Python code for K-means clustering can be found in [3]. The C version code in this project is implemented based on that pseudo-code. Since particle positions typically do not change significantly over short time intervals, re-clustering after each time step is unnecessary. Therefore, an interval of 10^{-4} is chosen, taking into account the simulation times (T = 0.001s).

Clustering computations can be time-intensive, so it's crucial to strike a balance between cluster computation and cache performance. Figure 1 illustrates the variation in execution time across different values of k. Optimal performance is observed when k = 4. Additionally, Figure 2 displays the initial and final clustering results, providing a visual representation of the clustering.

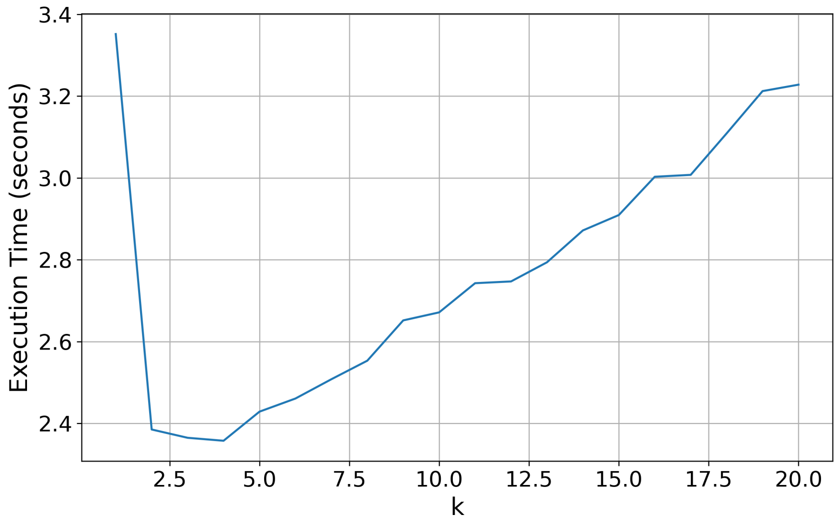
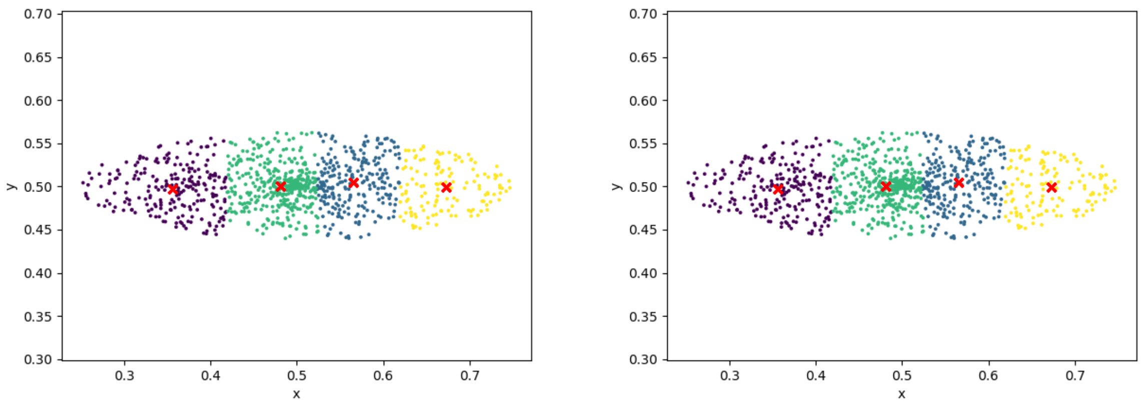


Figure 1: Execution time of different number of clusters



(a) Particle clusters, $T = 0$ (b) Particle clusters, $T = 0.001$

Figure 2: Particle clustering based on K-means, $k = 4$

3.2.4 Parallelization with OpenMP

The code parallelization is implemented using OpenMP, which simplifies the process with its straightforward directives.

The parallelism in the code focuses on two aspects. Firstly, it targets clustering. When labeling particles into different clusters, particles are distributed among threads to accelerate the distance calculation process. Secondly, the core of the code lies in time integration, with the most time-consuming parts being the insertion of particles and force computation.

The force computation part is parallelized using OpenMP's `#pragma omp parallel for` directive, which will distribute the iterations of the following `for` loop among threads.

Since the calculation of each iteration is independent, there are no synchronization issues. The `schedule(dynamic, CHUNK_SIZE)` clause means the iterations are divided into chunks of size `CHUNK_SIZE`, dynamically assigning these chunks to threads as they become available. Using `dynamic` instead of `static` is more suitable because the time taken for each iteration of the `barnesHut` function can vary significantly depending on the particle's position. With `dynamic`, better load balancing and faster execution times can be achieved.

We select the set for the number of threads as 1, 2, 4, 8, 16 to test the performance of the code both in serial and in parallel. Figure3(a) illustrates that the optimized number of threads is 8. The reason is that the machine used for testing has 4 cores with 2 threads on each. Using more than 8 threads won't provide any additional parallelism. Instead, it will lead to overhead from context switching and thread management.

The speedup for a parallel execution is defined as

$$S = \frac{T_s}{T_p}$$

where T_s is the time to execute the best serial algorithm, and T_p is the time to execute the parallel algorithm using p threads. Figure3(b) shows the speedup increases as N steadily increases when the number of threads is 8.

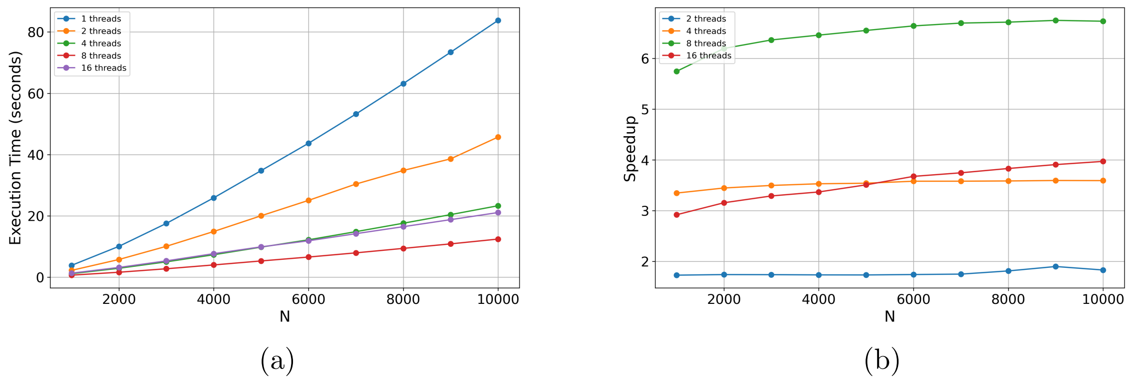


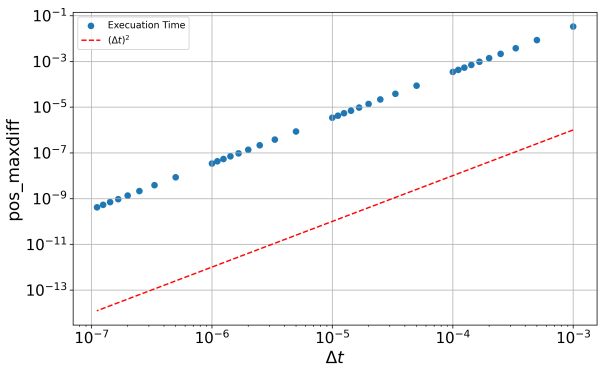
Figure 3: Execution time vs N with different number of threads (symplectic Euler method with Barnes-Hut algorithm, $\Delta t = 10^{-5}$, `nsteps` $= 200$, $\theta_{max} = 0.23$, k $= 4$)

3.3 Velocity Verlet Time Integration Method

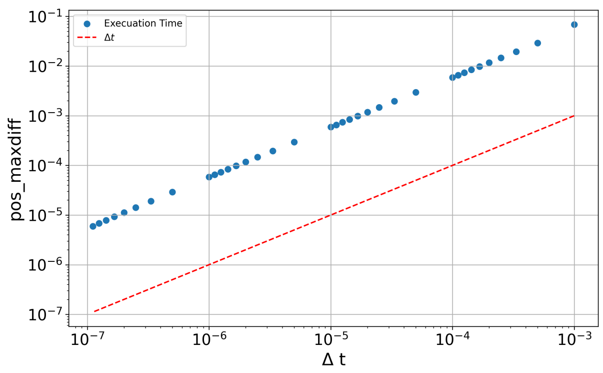
With the Velocity Verlet time integration method, it's possible to use a larger time step size, which can lead to better performance in terms of execution time while maintaining numerical accuracy.

For the implementation, an initial acceleration is required and calculated using the forces and masses. Therefore, we need to calculate the force once before entering the time integration loop.

We verified the order of the Velocity Verlet method by examining the relationship between the time step and the maximum position differences. As shown in Figure 4(a), by plotting the result on a logarithmic scale, the line exhibits a slope of 2, consistent with $(\Delta t)^2$, indicating that the method is second-order. Meanwhile, we can use the same procedure to verify that the symplectic Euler time integration method is first-order. The result is displayed in Figure 4(b).



(a) The second order Velocity Verlet method



(b) The first order symplectic Euler method

Figure 4: Verification of time integration methods: relationship between time step and maximum position differences (with `sun_and_planets_N _3.gal`, $\theta_{max} = 0$, $\triangle t = 10^{-8}$, time_steps $= 2 \times 10^6$ as the "true" solution, and go from $\triangle t = 10^{-3}$, nsteps $= 20$)

4 Experiments

The code performs well during testing. With `pos_maxdiff` below 10^{-3} for the case $N = 2000$ and $nsteps = 200$, we achieve an optimized $\theta_{max} = 0.23$ with `pos_maxdiff` $= 9.59 \times 10^{-4}$. This indicates the correctness of the implementation of the Barnes-Hut algorithm. Using the same θ_{max}, we verified that the time-complexity of the algorithm is $O(NlogN)$ on average (see Figure 5).

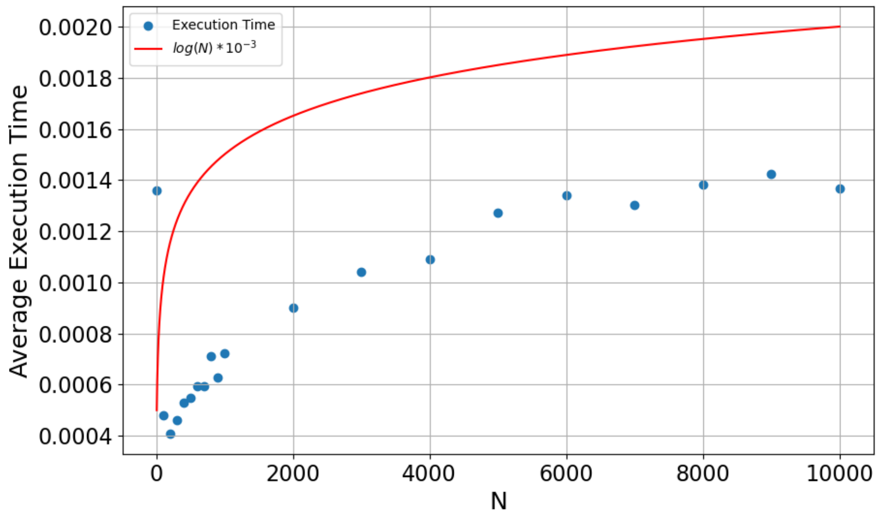


Figure 5: Verification of time complexity: average execution time vs N (using symplectic Euler method with Barnes-Hut algorithm, $\theta_{max} = 0.23$, `nsteps` $= 200$, $\Delta t = 10^{-5}$), k = 4

Then, we optimize the values of θ_{max} and $\triangle t$ to achieve minimal runtime while keeping the `pos_maxdiff` below 10^{-5}. A reference solution is constructed using the Velocity Verlet method with the Barnes-Hut algorithm, with parameters set as $N = 1000, \Delta t = 10^{-9}, nsteps = 10^6$.

Figure 6 illustrates the errors for the symplectic Euler method and Velocity Verlet method respectively, with Δt increasing from 10^{-7}, compared with the "true" solution above. From the figure, we observe that for `pos_maxdiff` below 10^{-5}, the time step size required for the Velocity Verlet method ($\Delta t = 2.5 \times 10^{-6}$, `pos_maxdiff` = 1.76×10^{-6}) is only one-tenth of the symplectic Euler method ($\Delta t = 2.5 \times 10^{-7}$, `pos_maxdiff` = 10^{-5}).

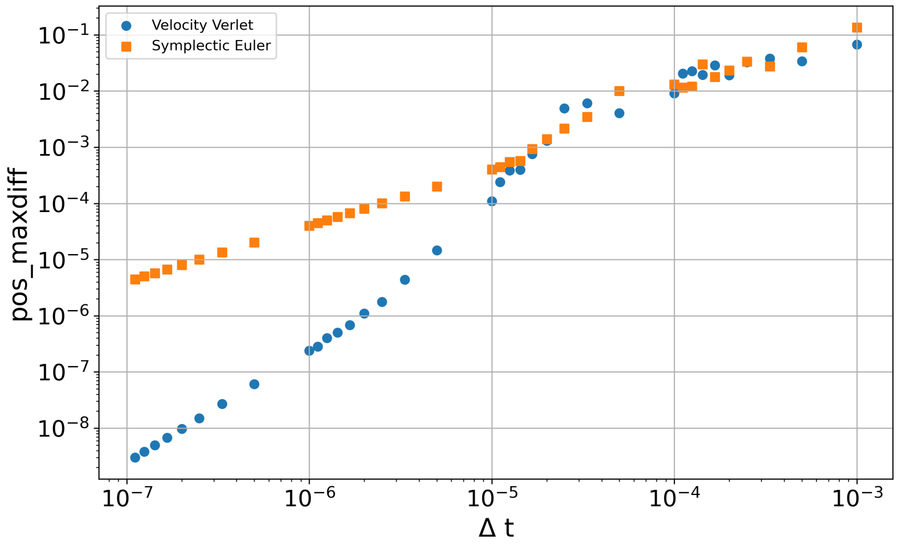
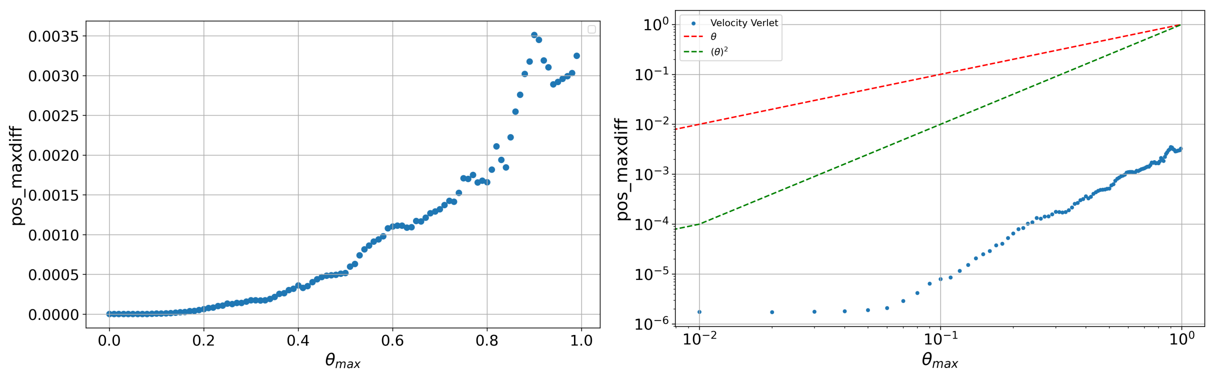


Figure 6: Error measurement for symplectic Euler method and Velocity Verlet method with increasing time step sizes (based on Barnes-Hut, $\theta_{max} = 0, T = 10^{-3}$)

To reduce the execution time, we introduce the Barnes-Hut algorithm, which also introduces errors, and the errors increase as θ_{max} increases. As shown in Figure 7, the errors increase quadratically. To control the error still below 10^{-5}, the optimal θ_{max} we can get is 0.11.



(a) (b)

Figure 7: Error measurement for symplectic Euler method and Velocity Verlet method with increasing θ_{max} (based on Barnes-Hut, $\Delta t = 10^{-5}$, nstep $= 100$)

With the K-means-based Barnes-Hut algorithm and Velocity Verlet method, we can obtain significant improvement in terms of execution time. Testing on the case $N = 1000$, $T = 10^{-3}$, and `pos_maxdiff` $< 10^{-5}$, the serial straightforward algorithm requires 24.79 wall seconds with $\Delta t = 2.5 \times 10^{-7}$ and `pos_maxdiff` $= 10^{-5}$, while the final optimized algorithm only cost 2.33 wall seconds with $\theta = 0.11$, $\Delta t = 2.5 \times 10^{-6}$, and `pos_maxdiff` $= 8.58 \times 10^{-6}$. We achieved a speedup of about 90.6%.

5 Conclusions

This project demonstrates the effectiveness of the implemented Barnes-Hut algorithm for gravitational simulations. Through several experiments, the correctness and performance of the code have been verified.

However, there are still opportunities for further optimization. An optimization possibility is constructing a tree locally for each cluster rather than globally. We can then parallelize the insertion process by assigning each cluster to a different thread. The size of the region will influence the balance of the tree. Therefore, it is better to recognize the boundary box for each cluster instead of using the entire region. Compared to dividing the region evenly, a simple approach called Orthogonal Recursive Bisection (ORB) divides the large square into several non-overlapping subrectangles. Each subrectangle contains approximately the same number of particles and is assigned to a thread [4].

Another idea is balancing the workload for force computation. Although each thread currently handles a similar number of particles, differences in the traversal depth for each particle can lead to workload imbalances. To address this, we can record the traversal depth for each particle and group them accordingly to ensure that each group has a similar workload [4].

References

[1] Michele Trenti and Piet Hut. "N-body simulations (gravitational)". In: *Scholarpedia* 3.5 (May 2008), p. 3930. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.3930.

[2] Youssef M. Marzouk and Ahmed F. Ghoniem. "K-Means Clustering for Optimal Partitioning and Dynamic Load Balancing of Parallel Hierarchical N-Body Simulations". In: *Journal of Computational Physics* 207.2 (2005), pp. 493–528. DOI: 10.1016/j.jcp.2005.01.021. https://doi.org/10.1016/j.jcp.2005.01.021.

[3] *Chris Piech. CS221: K Means.* https://stanford.edu/~cpiech/cs221/handouts/kmeans.html.

[4] *CS267: Fast Hierarchical Methods for the N-body Problem, Part 2.* https://people.eecs.berkeley.edu/~demmel/cs267/lecture27/lecture27.html.