Barnes-Hut N-Body Simulation

Sachet Sunil Zode | sachetz

1. Introduction

This project aims to implement an efficient algorithm to simulate the N-Body problem using the Barnes-Hut algorithm. The project is designed and implemented in Go, with a major focus on parallelism and improving the performance of the simulation - mainly using 2 parallelisation techniques, BSP and work-stealing queues. The performance of the algorithm is tested across different sizes and input types, with the initial placement of the points generated to be random, in a circle and skewed to form a cluster with sparse outliers, the number of points tested in the range of 10 to 10000, and the number of iterations set between 10 and 1000. The project saves the location of each point after every time step, so as to visualize the movement of the points, the results of which are added to the project benchmarks.

2. N-Body Simulation and Barnes-Hut Algorithm

An N-body simulation is a computational technique used to simulate and analyze the dynamics of systems with multiple interacting particles or bodies, where "N" refers to the number of entities involved. These simulations are crucial in fields like astronomy for modeling celestial mechanics, including the movements of stars, planets, and galaxies, by calculating the gravitational forces between all pairs of objects and predicting their motions over time. Due to the computational complexity, especially as N grows, sophisticated algorithms are employed to efficiently approximate these interactions and evolve the system's state.

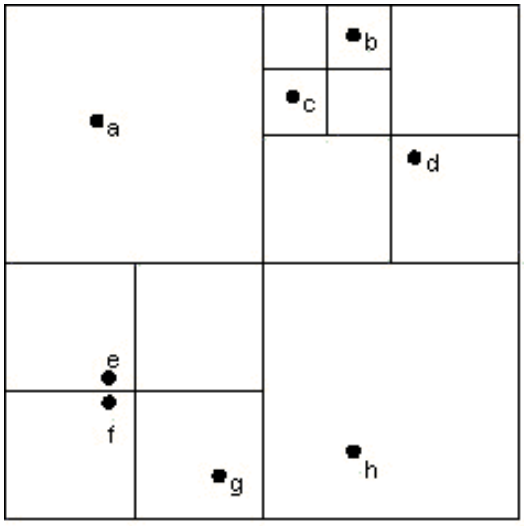
The gravitational force between any two bodies is calculated using Newton's law of universal gravitation. According to this law, the force $\vec{F}$ exerted on body 2 of mass m_2 by body 1 of mass m_1, separated by the displacement vector $\vec{r}$ from body 1 to body 2 is given by:

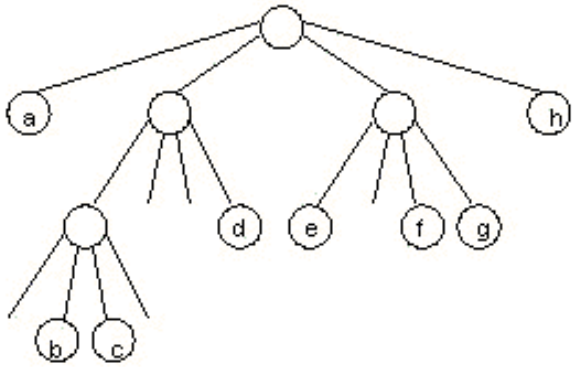
$$\vec{F} = \frac{Gm_1m_2}{|r|^3}\vec{r}$$

Where G is the gravitational constant.

The Barnes-Hut algorithm is an efficient computational method used to reduce the complexity of N-body simulations, where the calculation of forces (like gravity) between particles can be very computationally expensive. In a straightforward N-body simulation, calculating the gravitational force between every pair of bodies requires $O(N^2)$ calculations, where N is the number of bodies. This becomes impractical for large N. The Barnes-Hut algorithm addresses this issue by approximating the forces between distant particles to reduce the number of calculations needed. It does so by following these 4 steps:

a. Spatial Decomposition: The algorithm divides the simulation area into smaller regions using a quad-tree structure. Each node of the tree represents a spatial region, with the root node representing the entire simulation area and each leaf node representing a small subregion containing a single body.
b. Tree Construction: Bodies are inserted into the tree one at a time. If a node should contain multiple bodies (i.e., the bodies are in the same region), that node is subdivided into smaller nodes (children), and each body is placed in the appropriate child node based on its position.
c. Center of Mass: For each region represented by a node in the tree, the algorithm calculates the total mass and the center of mass (the weighted average position of all the masses in the region).
d. Force Calculation: When calculating the force on a particular body, the algorithm traverses the tree. For each node, it decides whether the node is far enough away to treat all the bodies contained in that node as a single body with their combined mass located at their center of mass. This decision is based on a criterion involving the ratio of the size of the region to the distance from the body to the region's center of mass. If the node is considered "far," the combined effect is used. If it's "near," the algorithm either uses the actual forces from individual bodies (if the node is a leaf) or recursively evaluates the node's children.





This approach significantly reduces the number of calculations, especially for large N to $O(N\ log\ N)$, making simulations of large-scale systems like galaxies feasible with available computational resources.

3. Parallelized Implementations

For the parallel implementation, each stage of the algorithm is executed concurrently across T threads, with each thread working on N/T particles.

Two parallel implementations are compared. First, a Bulk Synchronous Pattern (BSP) based approach is implemented, which is a great fit for this algorithm due to the nature of distinct supersteps. Barriers (built using condition variables) were used to separate logical supersteps to synchronize the execution of the different threads. No thread is allowed to proceed with further computation until all threads have reached that barrier, thus ensuring data consistency between the supersteps. This is an ideal solution for this system, as the Barnes-Hut algorithm is inherently divided into clear supersteps.

The second implementation involved the use of work-stealing queues. The skewness and clustering of the initial data points hinted that distributing the work in terms of number of points might not be an ideal approach, as in cases of clusters, there would be an imbalance in the work assigned to each thread. Work-stealing queues would effectively solve this problem by ensuring that threads have an opportunity to steal other threads' work, resulting in an effective increase in the overall parallelism. The work-stealing queue was implemented using a double-ended queue, where the bottom of the queue is accessible to the owner thread (and supports both the pop and push operations), whereas other steal can only pop any pending jobs from the top of the queue.

Note: Calculation of the center of mass is not parallelised. This is due to the inherently complex nature of this calculation, where the center of mass depends on multiple nodes within the subtree, and achieving this with a fixed number of threads involves the use of complex algorithms.

4. Instructions to Run

Replicating the experiment and graph generation

To replicate the experiment, the user must **cd** to the /<path-to-local>/proj3/benchmarks directory, update the **graph.sh** file with the correct parameters for slurm, and execute the command: **sbatch graph.sh**.

The process then calls a python script that handles the execution of the process to replicate the experiment. The python script performs the following operations:

a. Calls the process in sequential mode and records the time
b. Calls the process with combinations of various parameters - pattern of initializing points, number of iterations/time steps, number of points, number of threads and the parallel mode (BSP or work stealing)
c. Calculates the observed speedup from the execution times
d. Generates graphs between the number of threads and speedup comparing different parameter combinations.

The slurm logs are generated in the slurm/out folder, and the benchmarks are recorded in the benchmarks.txt file, both located in the /<path-to-local>/proj3/benchmarks folder.

Running specific versions of the program

To run specific versions of the program, the user must **cd** to the /<path-to-local>/proj3 directory and run the nBody.go file as **go run nBody.go <args>**. The process supports the following arguments, in order:

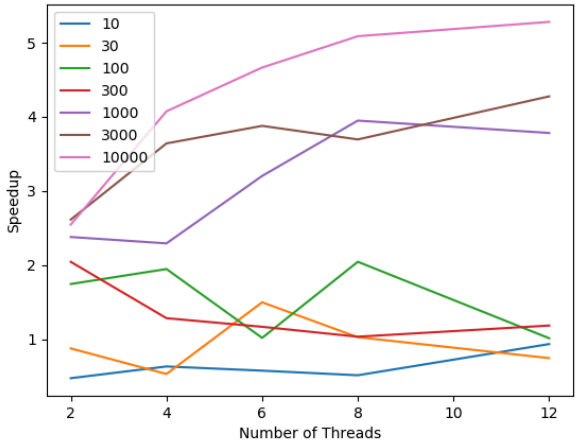
a. Mode - this takes values **s** (sequential), **bsp** (barrier based parallelisation), **ws** (work stealing based parallelisation. The default is sequential.
b. nPoints - This is the number of generated points. The default is 3000.
c. nIters - This is the number of iterations/time steps. The default is 200.
d. numThreads - Number of threads for the parallel implementations. The default is 8.
e. Logging - This indicates whether logging or file write is enabled. The default is true.
f. initPoints - This is the initial setup of points. This takes values **random** (which initializes points randomly in a subspace), **circle** (which initializes points in a circle), and **skewed** (which initializes points in a cluster and adds some sparse outliers).

If logging is set to true, the process writes the generated location of the points after every time step into a file **particles_<mode>.py**. These change of locations can be converted into a gif using the **plot_particles.py** file in the benchmarks folder, by providing **<mode>** as an argument. The result of two such sample runs with 16 points and 200 iterations initialized in a circle is included in the repository.

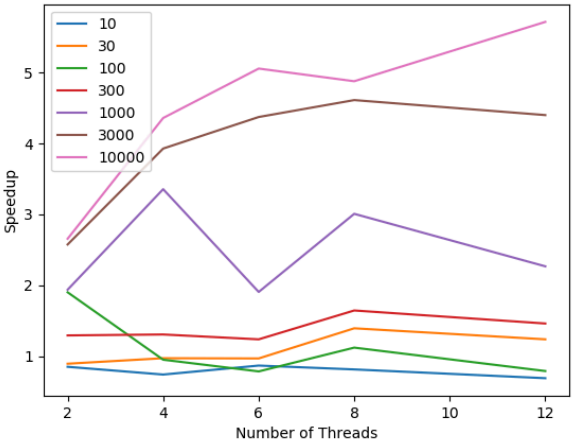
5. Results

The process is tested for a varying number of iterations, number of points, and number of threads for different initial points arrangement across both parallel implementation modes. Each graph plots the number of threads with the speedup measured as a best of 3 experiments, across different numbers of points.

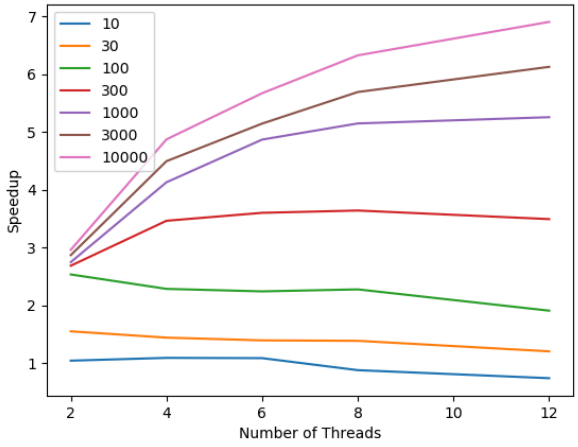
Points initialized in a Circle



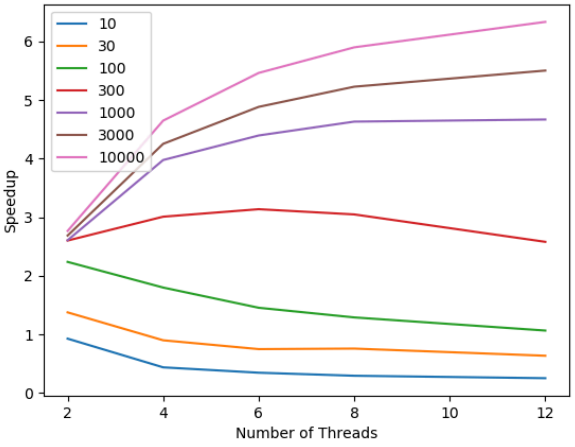
BSP - 10 iterations



Work Stealing - 10 iterations

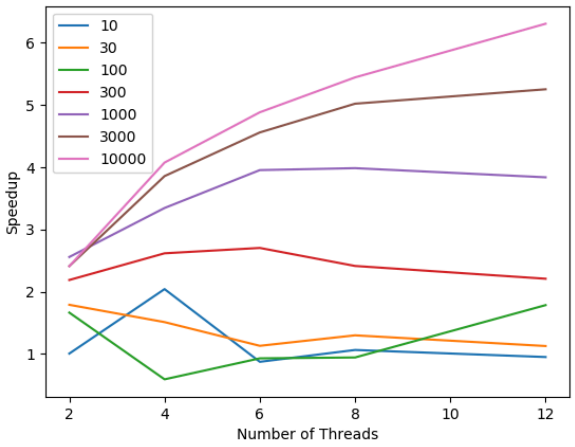


BSP - 1000 iterations

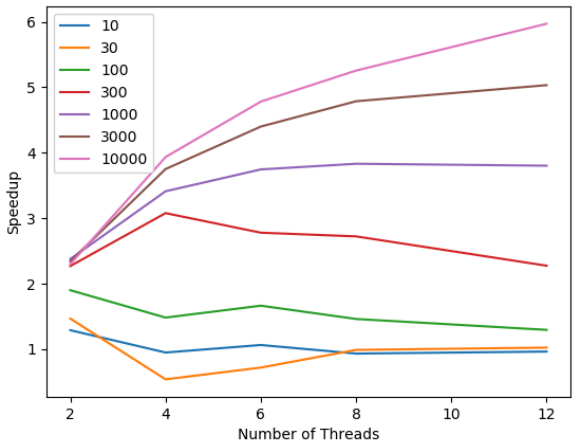


Work Stealing - 1000 iterations

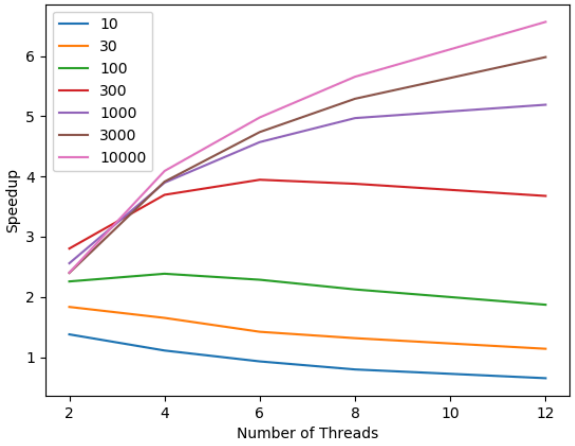
Points initialized randomly in a subspace



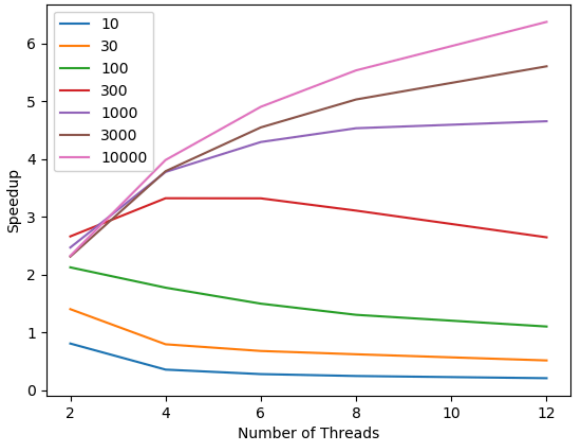
BSP - 10 iterations



Work Stealing - 10 iterations

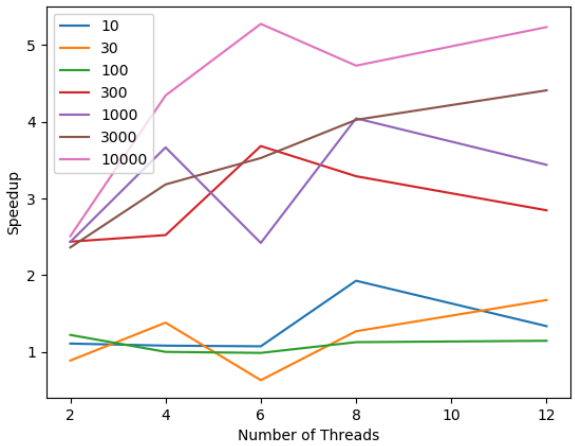


BSP - 1000 iterations

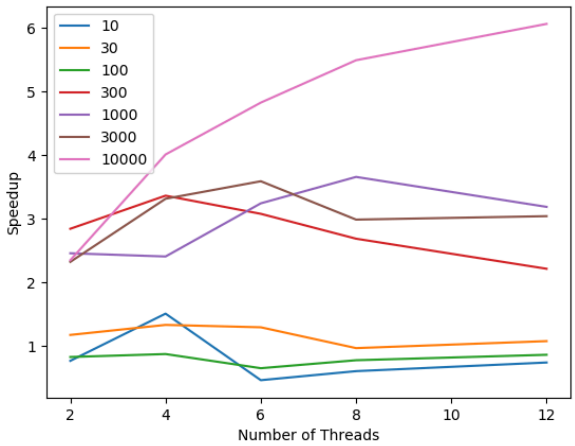


Work Stealing - 1000 iterations

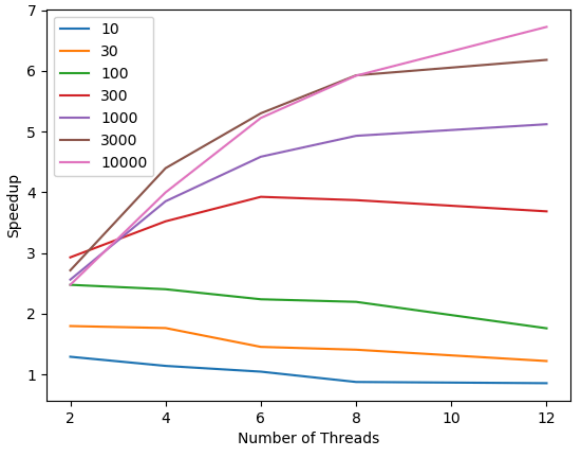
Points initialized with a skew



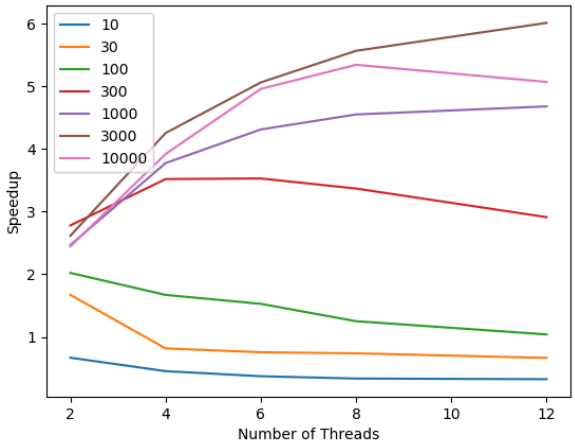
BSP - 10 iterations



Work Stealing - 10 iterations



BSP - 1000 iterations



Work Stealing - 1000 iterations

Observations:

a. As supported by the higher number of iterations, for the initialization of points in all three types, the speedup increases with an increase in the number of threads and number of points. An exception can be seen in the skewed points initialization, where for 10000 points, the speedup is less than that for 3000 points. This can be attributed to a large number of points being present in a cluster that increases the depth of the tree and therefore increases the complexity of the problem.
b. For a small number of iterations, we observe that the work stealing implementation has slightly better performance than the BSP implementation, and for a large number of iterations, the two implementations have similar performances, with BSP having slightly better speedup. This can be due to the rebuilding of the work stealing queues for each step of the iteration, which can act as an overhead for the process, as the amount of work per superstep is relatively small.
c. The performance of all three implementations of points is relatively similar, with a speedup of around 6 observed for all cases.

6. Challenges

There are several challenges associated with the parallelization of the N-Body algorithm.

a. The addition of the points into the quad tree could result in data races, resulting in unintentional removal of points from the quad tree. To circumvent this, a fine grained locking mechanism is implemented on the quad tree that locks the node for any sort of a modification to the quad tree on node insertions. As there are no delete operations, validations are not required.
b. To synchronize the execution of the superstep, a barrier based implementation is used. This barrier is implemented using locks and condition variables, where each thread increments the shared counter and checks if the counter matches the total number of threads. If so, it broadcasts for all threads and leaves the barrier, and if not, it suspends until a broadcast is received, following which it leaves the barrier.
c. The work stealing implementation introduces an additional layer of complexity. A double ended queue was implemented that only stores the indices for each thread's execution region, thus improving the memory consumption of the process. This queue supports parallel popTop and popBottom operations to facilitate work stealing, and is implemented as a lock free data structure, improving the performance of the system. This is done by using compareAndSwap operations and introducing stamps to handle the ABA problem.

7. Hotspots and Bottlenecks

There are 5 supersteps that are the hotspots of the process:

a. Building the Quad-Tree
b. Computing Center of Mass
c. Calculating the Force
d. Updating the Position of each particle

The force calculation step is the most time intensive step as it could potentially involve the computation of force for each pair of particles. This superstep, along with the building of the quad-tree and the position update for each particle was parallelized successfully for a large boost in the overall performance of the system.

The center of mass calculation was not parallelized due to its complex nature in distributing the work across a fixed number of threads. Each internal node representing the center of mass may depend on multiple computations before it, which implies that there is no clear separation between the various levels of processing in this step. The impact of this step on the overall performance is minimized by the fact that this step only involves a single traversal of the tree, and as such does not have a large contribution to the overall execution time.

8. Reasons for Limited Speedup

There are various attributes of this algorithm that could explain the limited speedup achieved in the two implementations:

a. The BSP implementation is limited by the use of barriers, which could result in lower parallelism if the work division is not balanced. The work-stealing based approach would solve this problem, however the small execution time of the superstep and the overhead of reinitialising the work stealing queues for each superstep result in a very similar performance from this approach when compared to the BSP implementation.
b. Calculation of the center of mass is a bottleneck in the current implementation of the algorithm, however, due to its complex nature, it is difficult to divide this superstep into smaller, balanced tasks across a fixed number of threads.
c. The execution time for each task is fairly small, and as the number of points increase, this granularity results in a lower performance gain with an increase in the number of threads due to the overhead in spawning threads and work stealing queues taking up a large share of the total execution time of the process.