

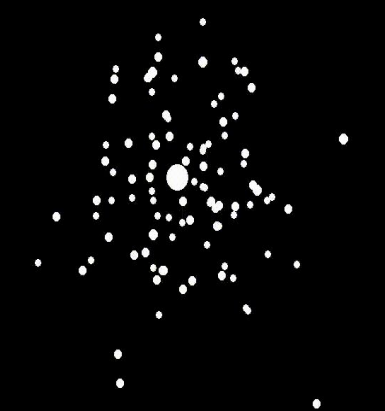
The background is a dark blue space-themed illustration. It features a ringed planet on the left, a spiral galaxy on the right, and various stars of different sizes and shapes scattered throughout. A large, rounded rectangular frame with a dotted line border encloses the central text.

Dark Matter Simulator

CS 205

Aditi Raju, Dennis Du, Jared Ni, Rafay
Azhar, Sezim Yertanator

Problem to Solve



Being able to model gravity in the universe has crucial scientific applications:

- Helps understand the complex dynamics of interactions b/w celestial bodies
- Useful for research in theoretical physics and supporting astrophysical discoveries
- Crucial for predicting celestial events and space mission planning
- Important educational tool for students and enthusiasts to play around with!

Project Motivation: we want an efficient way to model and simulate gravity accurately, and to update and visualise our system in real time.

Moreover, *N*-body simulations are very important in the sciences, not just physics!

Mathematical Model

Newton's law of gravitation: calculate force of attraction between two objects (as 3D vector)

$$\mathbf{F} = G \frac{m_1 m_2}{|\mathbf{r}|^2} \frac{\mathbf{r}}{|\mathbf{r}|}$$

Newton's second law of motion: calculate acceleration (vector) due to gravity

$$\frac{d^2 \mathbf{x}}{dt^2} = \frac{\mathbf{F}}{m}$$

Numerical integration to update velocity and position

$$\Delta \frac{d\mathbf{x}}{dt} \approx \frac{d^2 \mathbf{x}}{dt^2} \Delta t \quad \Delta \mathbf{x} \approx \frac{d\mathbf{x}}{dt} \Delta t$$

Why Parallelize?

- Want to be able to do large-scale simulations
 - E.g., the Milky Way contains hundreds of billions of stars
 - Although we cannot simulate such a large problem size (due to limited cluster resources), there is value in being able to handle millions or billions of stars to simulate galaxy dynamics such as “violent relaxation”, where resources don’t fit in memory
- Need for real-time visualization and simulation that can scale well. Parallelization is thus a solution to the computational bottleneck of a sequential implementation.



It's a lot of stars!

Sequential Baseline Algorithm

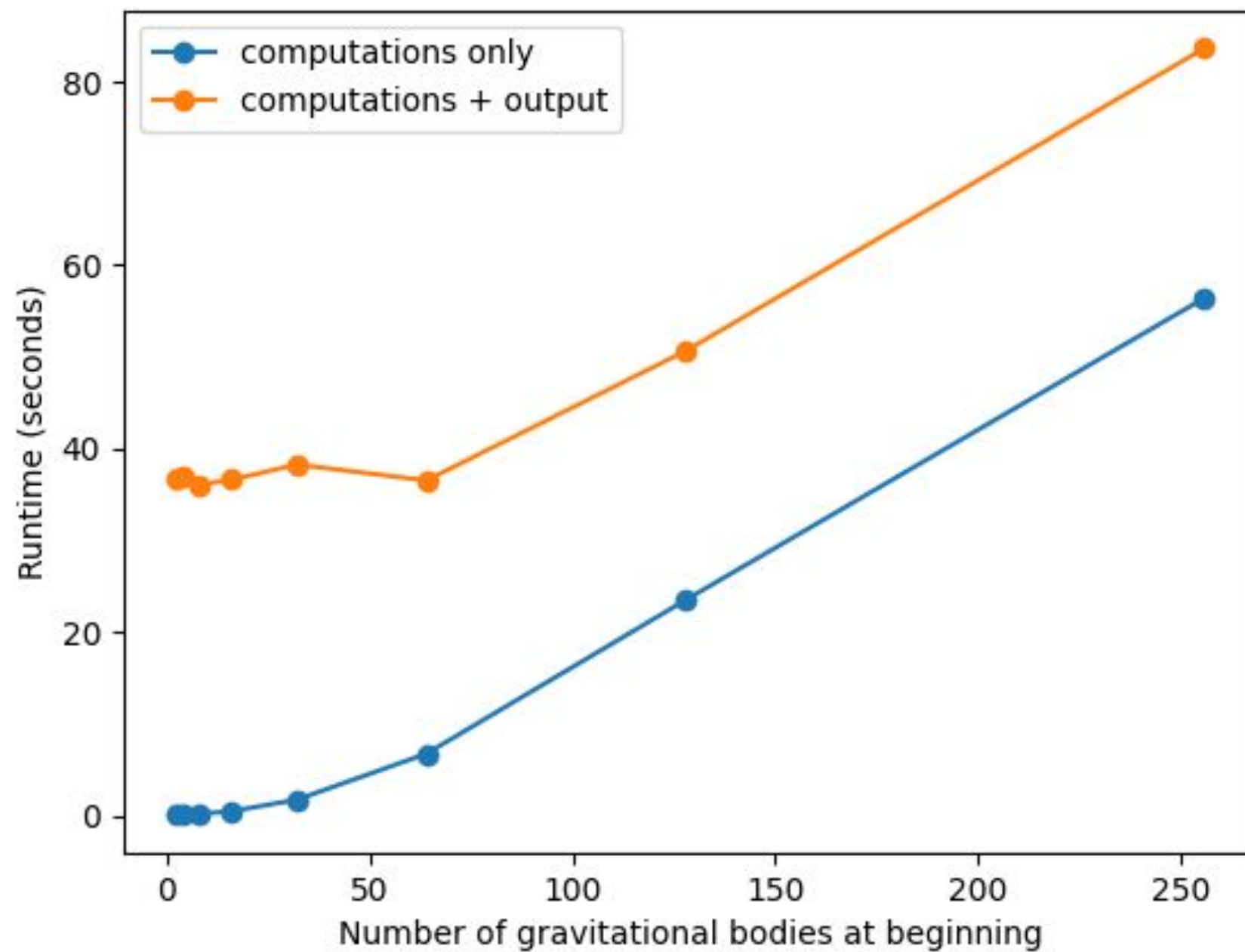
For each body **X** in the system:

For each body **Y** in the system except for **X**:

 Find the contribution of **Y** to the acceleration of **X**

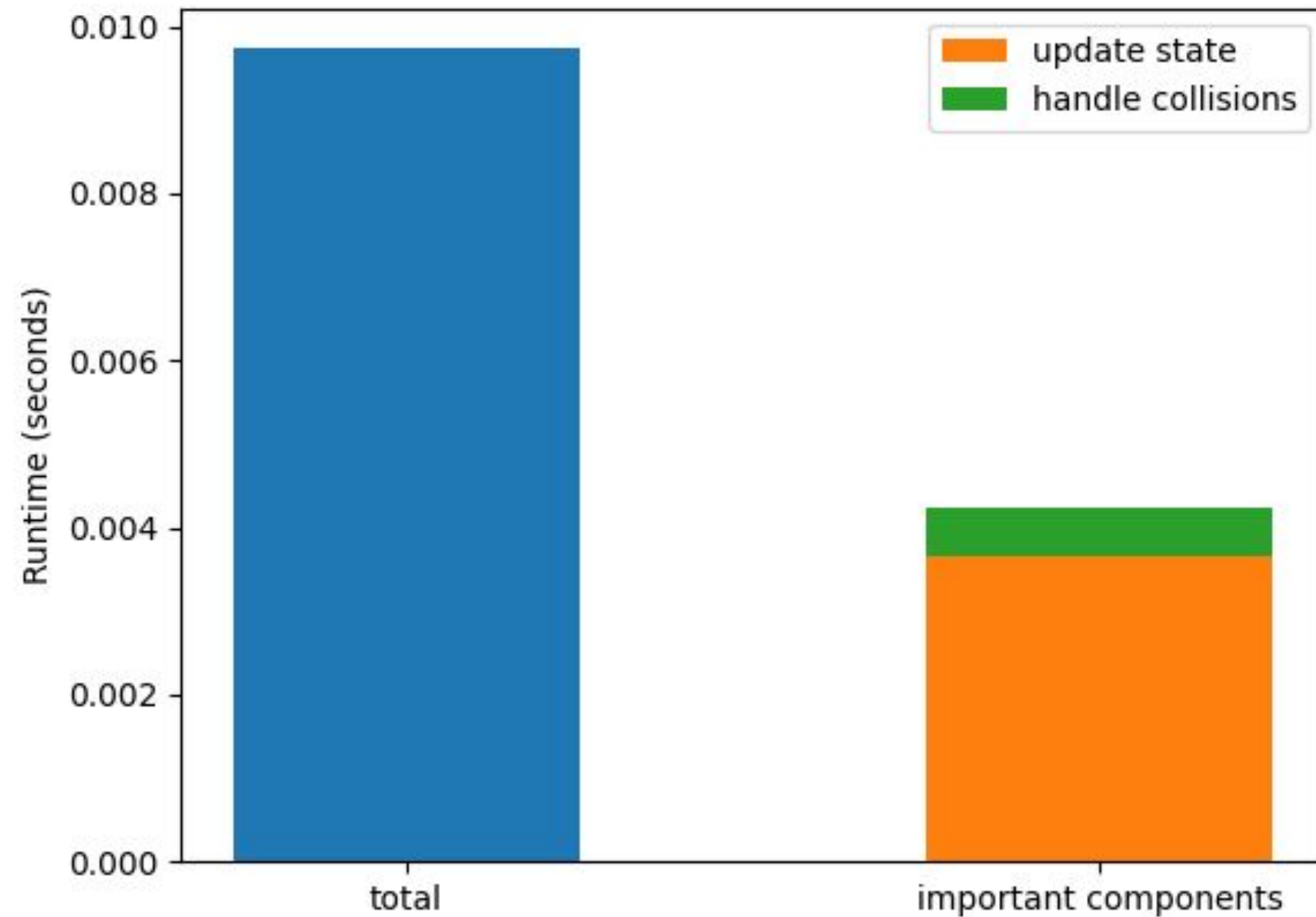
- Each of n stars must compute the sum of its gravitational interactions with all other $(n-1)$ stars
 - quadratic time adds up quickly if done sequentially for each timestamp
 - Thus, need more efficient to assign different star's computations to different threads

Sequential Baseline



- Begin with n randomly initialized objects, for powers of two from 2 to 256
- 30,000 position and velocity updates along with collision handling
- Runtime measured with and without writing output

Sequential Baseline



Important computations:

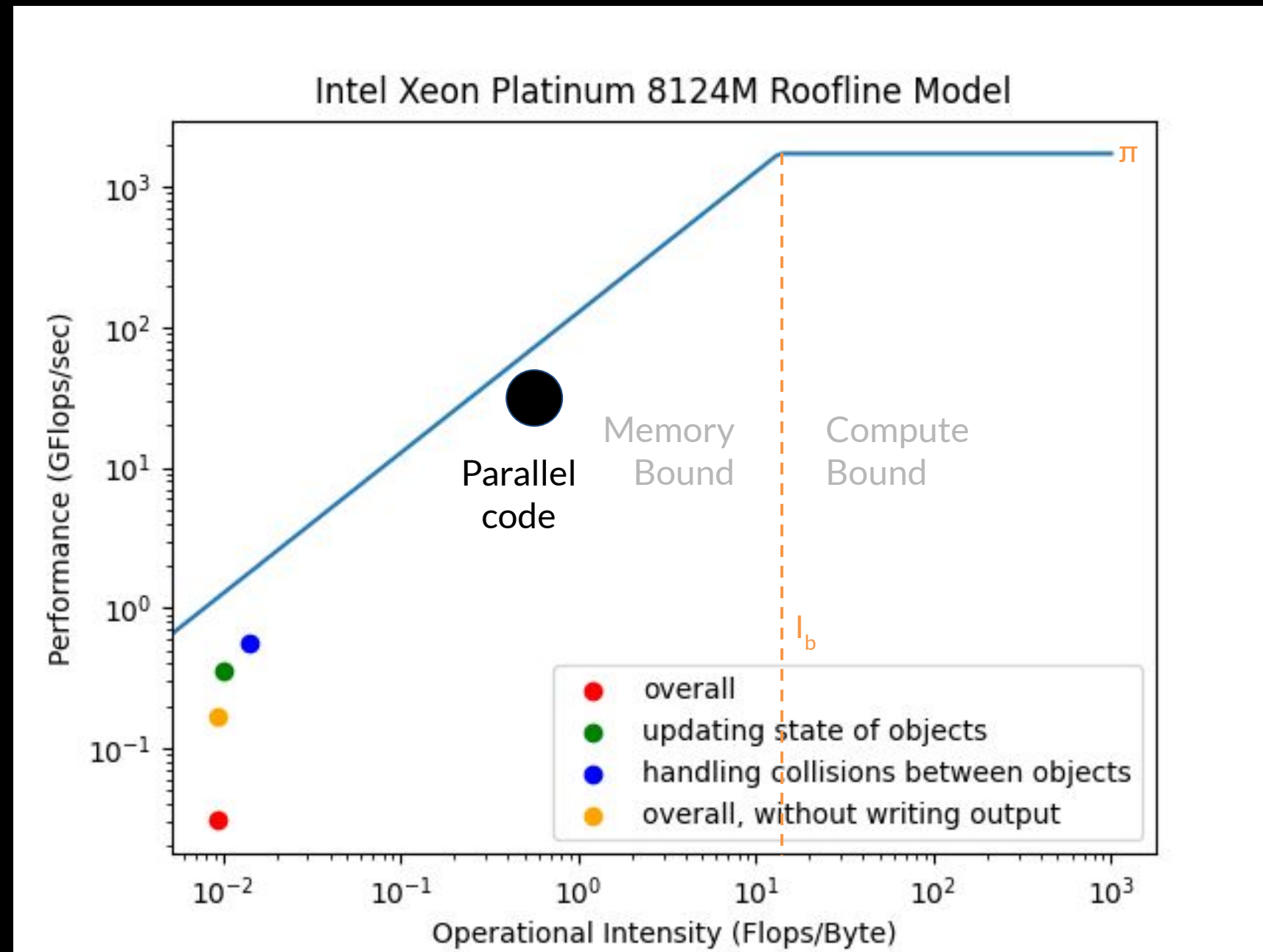
- Updating objects' position and velocity vectors
- Handling collisions between objects

Tested on one iteration with 256 objects

ROOFLINE CALCULATION

| | |
|---------------------------------------|---|
| Compute Hardware | Intel Xeon Platinum 8124M |
| Precision | Double precision |
| Frequency | 3.0×10^9 cycle/s |
| Number of Cores | 18 |
| SIMD Lanes | 512 bit / 64 bit = 8 |
| Total Flop Per Cycle | 4 Flop/cycle |
| Peak Floating Point Performance π | = frequency * number of cores * SIMD lanes * total flop per cycle = $(3.0 \times 10^9 \text{ cycle/s}) * (18 \text{ cores}) * (8) * (4 \text{ Flop/cycle}) = 1728 \text{ Gflop/s}$ |
| Peak Memory Bandwidth β | 128 GB/s |
| Ridge Point I_b | = $\pi/\beta = 1728 \text{ Gflop/s} / 128 \text{ GB/s} = 13.5 \text{ Flop/byte}$ |

ROOFLINE MODEL



Peak Floating Point Performance $\pi = 1728$ Gflops/s

Peak Memory Bandwidth $\beta = 128$ GB/s

Ridge Point $I_b = 13.5$ Flops/Byte

Parallel code (18 ranks and SIMD):


- flops: 115762920000
- memory transfer (bytes): 1193056266984
- runtime (seconds): 4.23157

So:

- operational intensity: 0.776 flops/byte
- performance: 27.356 Gflops/second

Forms of Parallelism

We exploit parallelism in our application by:



Shared memory with OpenMP

- Each thread computes updates for a small subset of the objects

SIMD

- Positions, velocities, and accelerations are 3D vectors
 - speed up calculations using SIMD registers

Distributed memory with MPI

- Each process handles a sub-region of the overall gravitational system
 - Each process shares the state of the region assigned to it with neighboring processes
-

Parallel Implementation

- Each spherical object represented by mass, radius, and xyz-coordinates of position and velocity at time 0 (initial conditions)
- MPI rank 0 reads initial state of universe from file and sends objects to all ranks
- The program repeats the following at each time step:
- Each process calculates accelerations due to gravity for its assigned objects and updates the positions and velocities
 - Each rank works on approximately n/p objects (where n is total number of objects, p is number of processes)
- Synchronization: updated object states are distributed among MPI ranks

Parallel Implementation

- Non-blocking MPI communication
 - Sends: root process can start working on its assigned objects while sending objects that are delegated to other ranks
 - Receives: Order of received updates from other ranks does not matter (since each one touches different objects) – can initiate all receives at same time
- OpenMP: Each rank further divides its assigned objects among multiple threads
- SIMD: 3D simulation involves vector calculations on (x,y,z) coordinates – wrote SIMD implementations of vector addition, subtraction, scalar multiplication, dot product

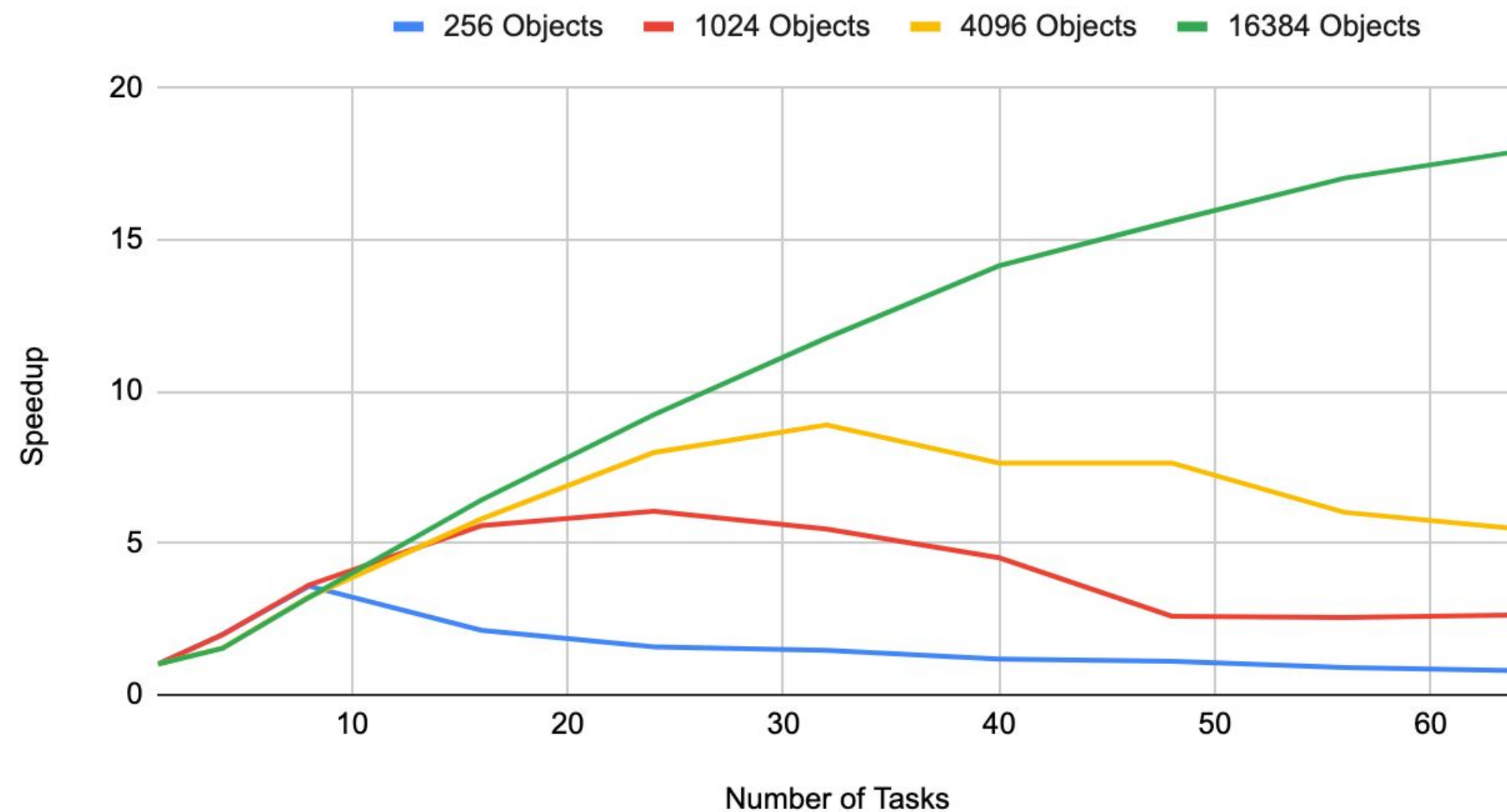


Use external libraries for rendering visualizations of universe:

- CImg: Used to output pixels of images
- ffmpeg: Used to take pixels produced by CImg and convert them into a video

Results

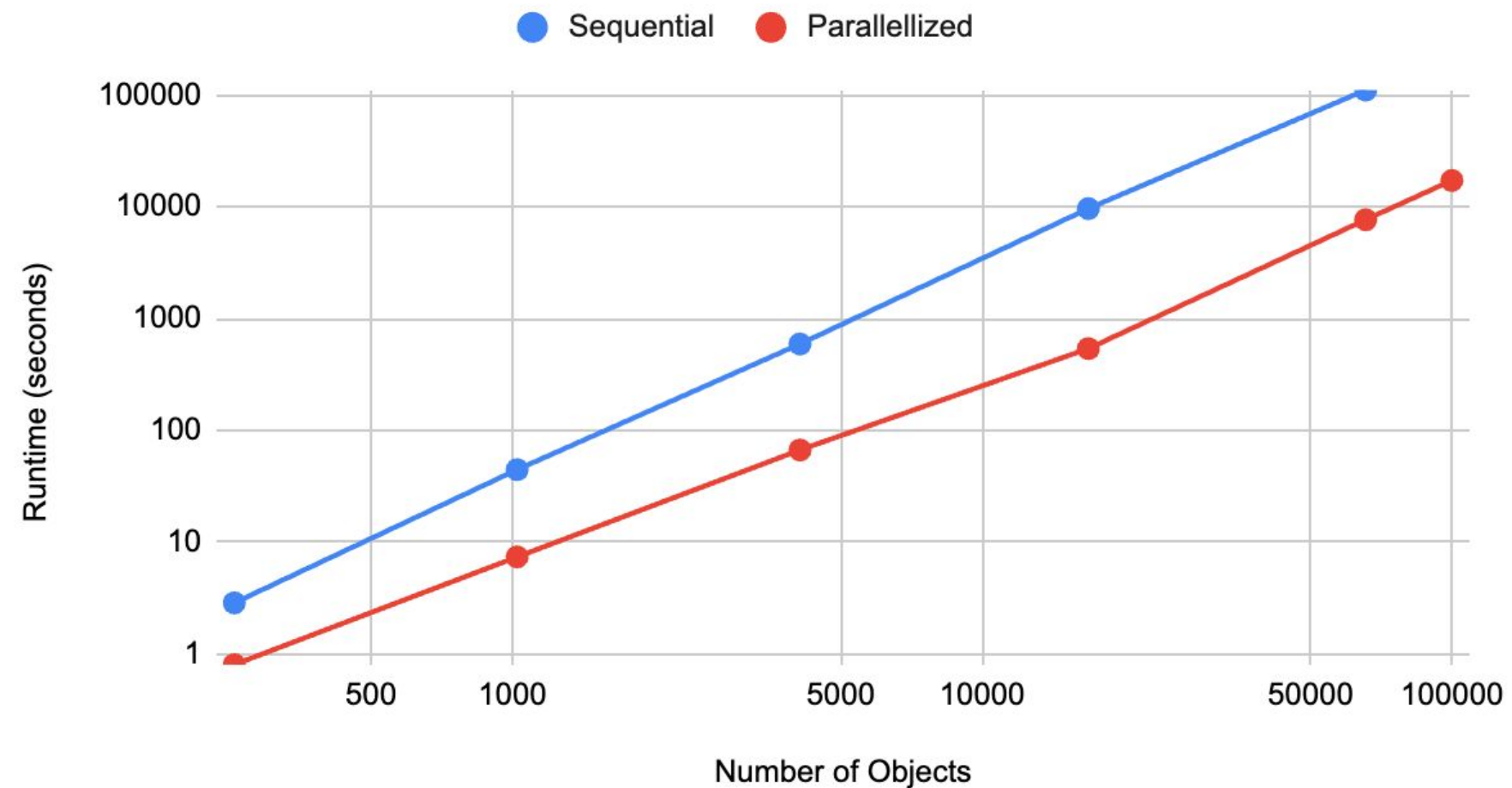
Strong Scaling



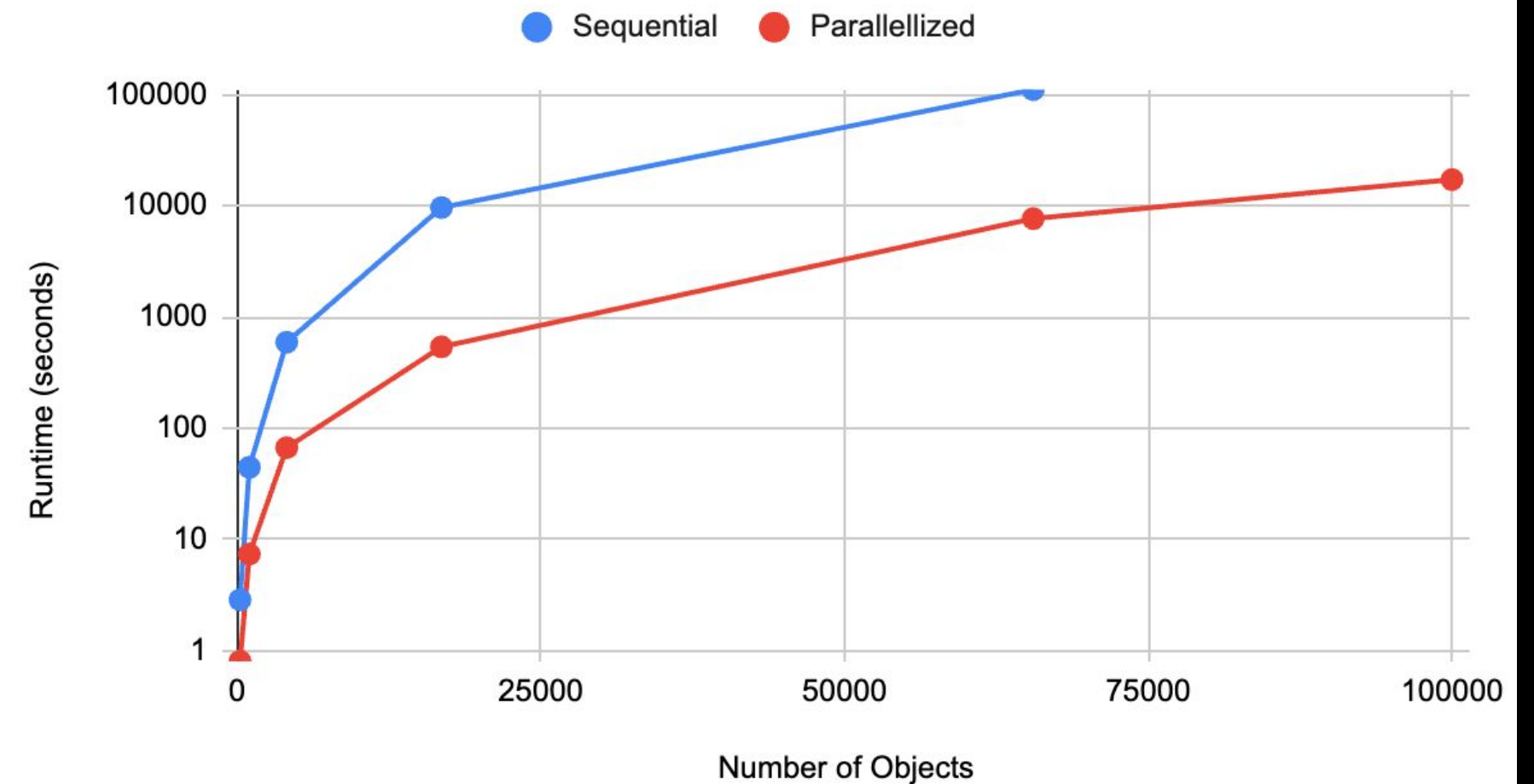
- A strong scaling analysis reveals that parallelization offers significant speedup gains
- The speedup is most noticeable for larger simulations
- For smaller simulations, increasing the number of tasks deteriorates speedup gains

Results

Sequential vs. Parallelized Models



Sequential vs. Parallelized Models



- Our parallelised code runs in orders of less magnitude time (log-log scale)

Correctness Test: Momentum

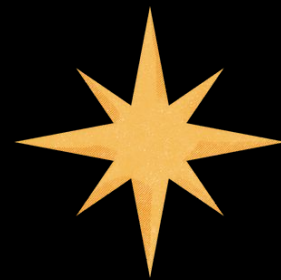
- A correct simulation should abide by physical laws
- Conservation of momentum – total momentum (sum of mass times velocity across all objects) should remain constant throughout the simulation
- Test: randomly initialized initial state, run with 18 ranks for 30,000 time steps
- Momentum remained constant at a value of $(-392839, 553118, -732516)$ throughout simulation

Key Insights and Takeaways



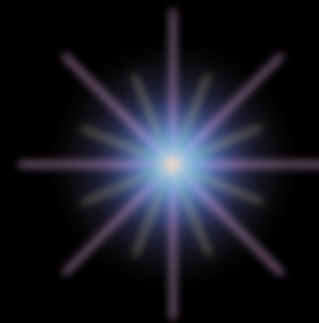
Lower-Level
Optimizations
are great

- Using SIMD instructions and operations leads to significant performance improvements



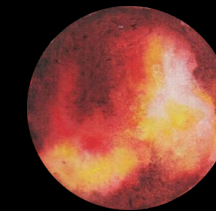
But, there is
always a
tradeoff

- A general takeaway from throughout the project is that **everything** comes at a cost - integrating low-level optimizations has an implementation overhead, harder to scale for a large project



More is not
always better

- A larger number of nodes doesn't always lead to less runtime; having too small a problem does not hide the overhead costs of parallelization



There is always
a better way of
doing things

- For almost all the code we wrote, we were able to optimise and refine it several times
- Cutting-edge, SoTA algorithms for doing many things we were trying to implement



Synchronization
matters!

- Synchronization b/w team members is as important as it is for openmp threads!
 - The project was a great way to learn about effective product management and communication
-

Further Extensions Explored

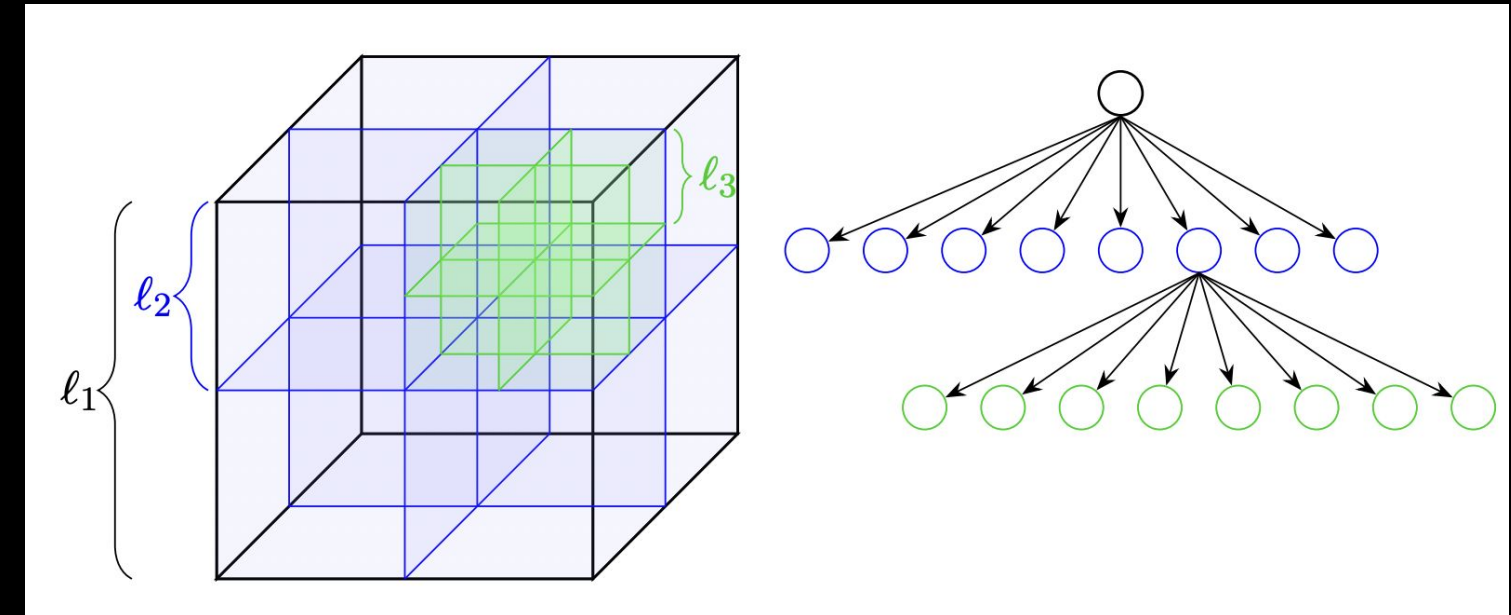
Key idea is to divide the simulation space into octrees via the Barnes-Hut method.

1. Each node carries data:

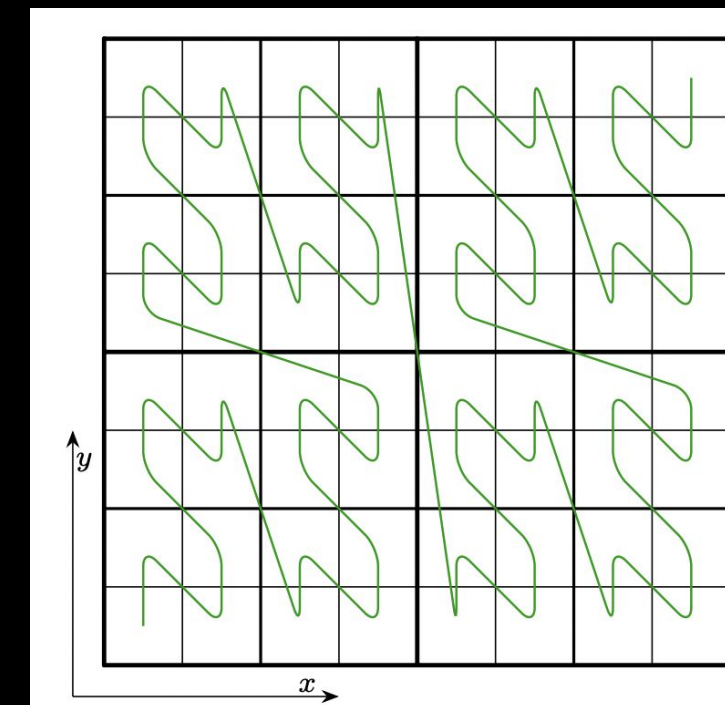
1. The Mass and position of the body (leaf node)
2. Center of mass and other coefficients required to calculate the force (internal node)

2. Dividing objects between different processes results in **load imbalance**. We deal with it using this approach: Assign (N/p) bodies to each process. Each process builds its local octree from the bodies it was given.

We will use a **space-filling curve (Morton's ordering)** to achieve a **balanced partition of bodies between processes**.



A. Brandt. On Distributed Gravitational N-Body Simulations. University of Western Ontario, 2022.



A. Brandt. On Distributed Gravitational N-Body Simulations. University of Western Ontario, 2022.

Extensions *(cont.)*

03

Processes merge their local octrees to create a global octree.*
We will perform a reduction by merging pairs of processes until the root process holds the final tree.

*Results in communication overhead: Deal with this using pairwise (log-wise) reduction to merge octrees

04

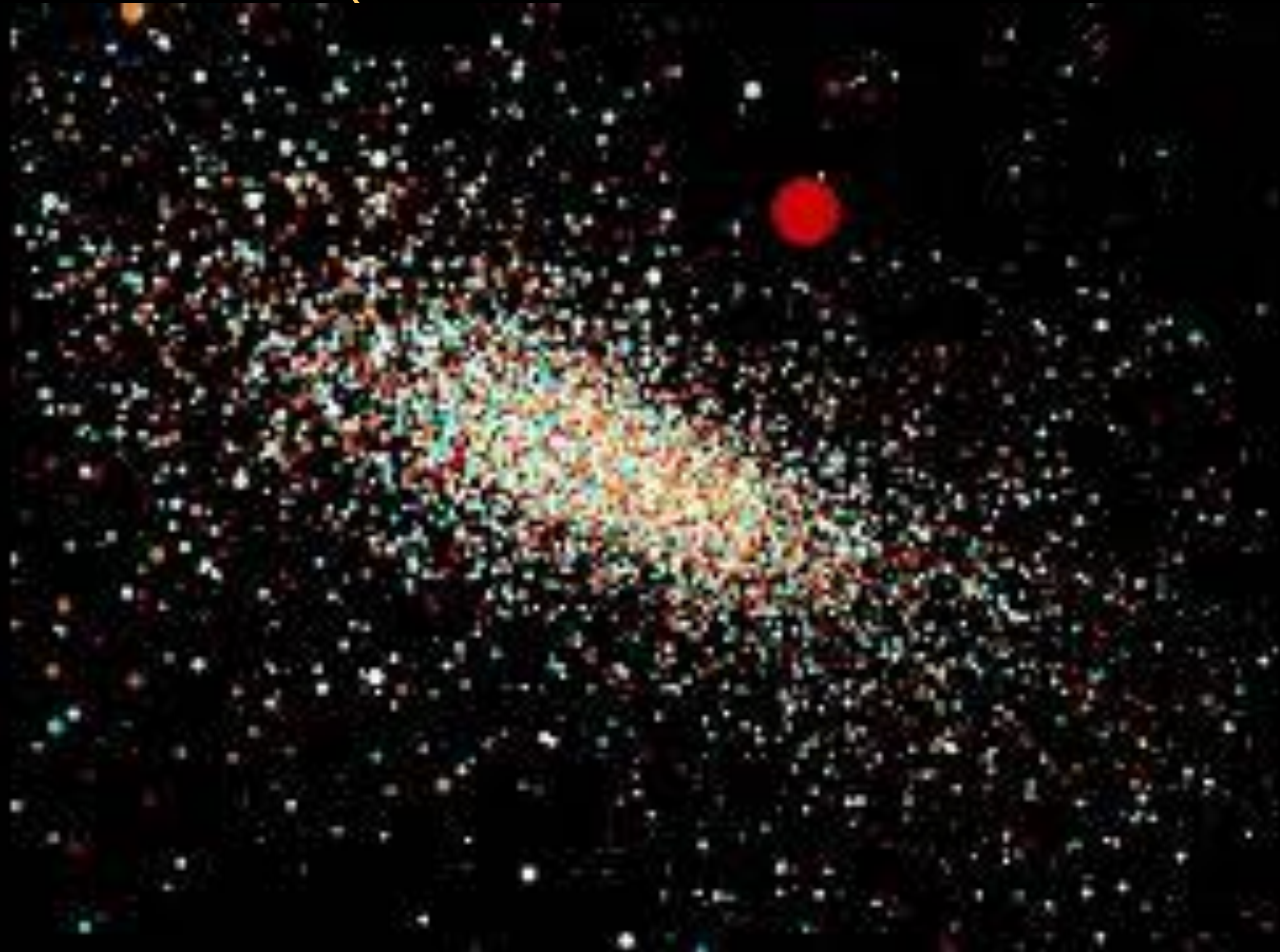
The root process broadcasts the global tree to other processes. Then, processes traverse the global tree from top to bottom in parallel to calculate the forces on each of their assigned bodies.

05

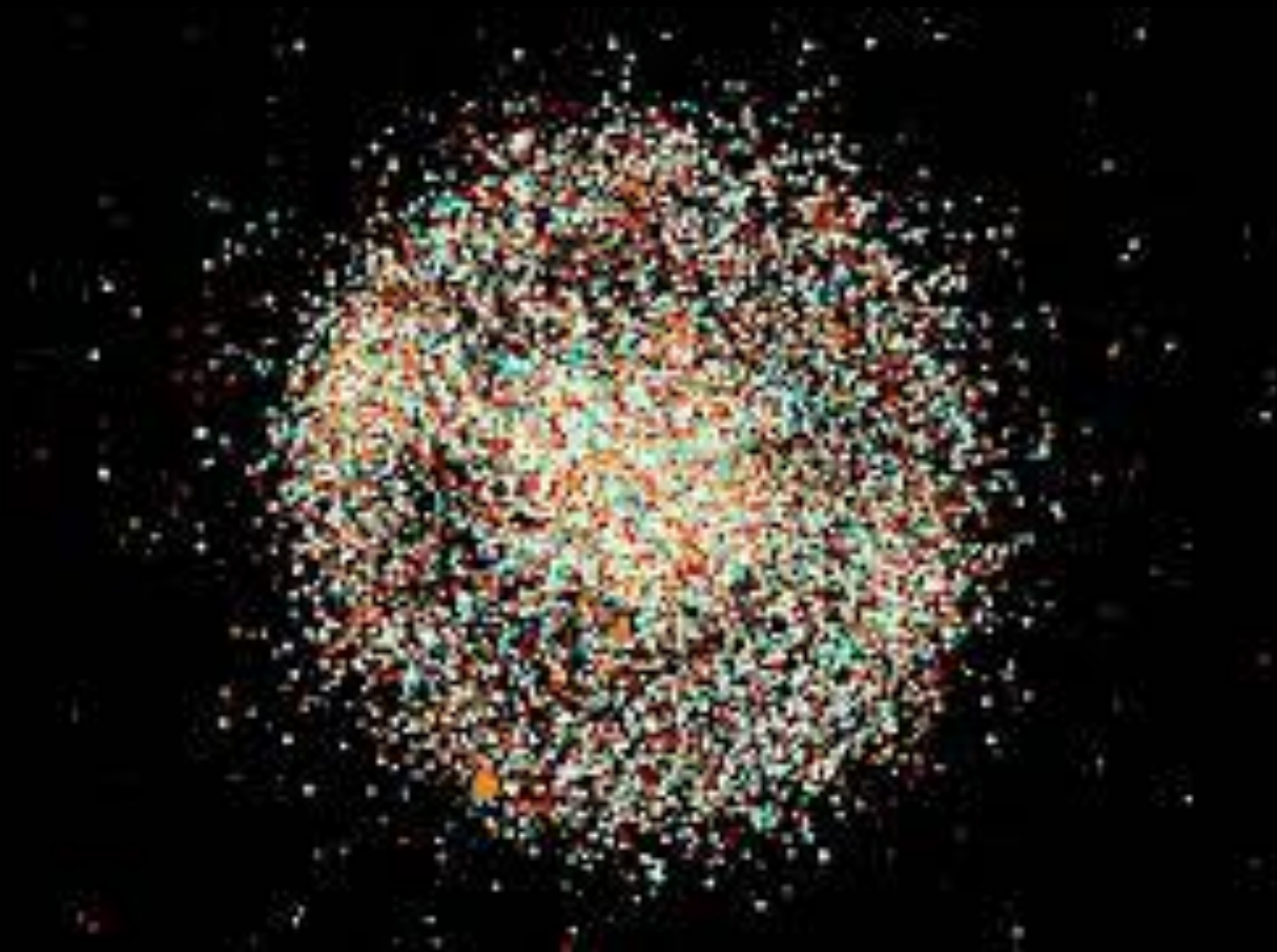
Update the velocity and position for each body using the calculated acceleration.



Demo (n=16384 bodies)



Demo (n=65536 bodies)



Thank You