COSC3500
2D Orbital Simulation Report

Maxwell Bo (43926871)

August 17, 2018

Description

The task was to create a stock-standard, 2-dimensional gravitational n-body simulator. All bodies were to be assumed to be point masses. The simulation was to be accurate, maintaining a constant total energy, and exhibiting phenomena such as apsidal precession. The simulation was to accept arguments specifying the granularity of the simulation (number of time steps, number of instances of data export), and a file specifiying masses and their initial positions and velocities. The simulation was to produce its output as fast as possible, with minimal slowdown with an increasing n number of bodies.

The simulator did not need to handle collisions between bodies. The MS1 simulator was to be free of OpenMP multiprocessing, which would be implemented in preparation for the MS2 submission.

Implementation

At a high-level, the initial naive simulator:

1. Parsed input parameters and files
2. Constructed `Body` class instances, representing each point mass
3. Packed the `Body`s into a `std::vector<Body>`, to maximize cache locality
4. Calculated forces between all pairwise combinations of n-bodies ($O(n^2)$)
5. Performed Euler's method to derive new velocities and positions
6. Output all necessary data
7. `GOTO 4`

By using a Quadtree, ('a tree datastructure in which each internal node has exactly four children') and the *Barnes-Hut* algorithm[1], the total cost of force calculation could be reduced to $O(n \log n)$, by grouping close-together bodies and approximating forces between the singular grouped pseudo-body, and distant bodies. New Quadtrees were constructed on each seperate simulation step.

Dehen and Read note that the Euler method 'performs very poorly in practice', further noting that 'errors are proportional to Δt^2'. They contrast it with the second-order *Leapfrog* symplectic integrator, which is 'heavily used in collisionless N-body applications'.[2]

Leapfrog can be expressed many in forms[3] including a synchronised form:

$$
\begin{aligned}
x_i &= x_{i-1} + v_{i-1/2}\,\Delta t \\
a_i &= F(x_i) \\
v_{i+1/2} &= v_{i-1/2} + a_i\,\Delta t
\end{aligned}
$$

which only requires a single acceleration calculation per every two half timesteps (the timestep Δt must be constant to maintain stability), and a 'kick-drift-kick' form

$$
\begin{aligned}
v_{i+1/2} &= v_i + a_i \frac{\Delta t}{2} \\
x_{i+1} &= x_i + v_{i+1/2}\Delta t \\
v_{i+1} &= v_{i+1/2} + a_{i+1} \frac{\Delta t}{2}
\end{aligned}
$$

that is stable with variable timstepping, but incurs an additional acceleration calculation per every two half timesteps.

The synchronised form was implemented, but attempts to implement the kick-drift-kick form, and variable timestepping, were left unfinished.

Thus, the final implementation:

1. Parsed input parameters and files
2. Constructed `Body` class instances, representing each point mass
3. Packed the `Body`s into a `std::vector<Body>`, to maximize cache locality
4. Inserted all `Body`s into a fresh `QuadTree` on full timesteps, traversing the `QuadTree` with every `Body` to calculate forces ($O(n \log n)$)
5. Performed the appropriate *Leapfrog* step to derive new velocities *or* positions
6. Output all necessary data
7. `GOTO 4`

Correctness

I personally believe that the simulation is relatively accurate. By visualising the results with `matplotlib`, we see something that resembles an n-body simulator. The total energy is flat, observing coefficients of variation as low as 0.001%. Euler's method consistently demonstrated coefficients of variation three times higher than that of *Leapfrog*. Due to recommendations by literature, and observed data, the use of Euler's method was gradually phased out during my testing to speed up the process.

Barnes-Hut caused a significant increase in observed coefficient of variation, averaging 0.08% across multiple runs. Furthermore, total energy was observed to step up and down at varying intervals1. I suspect that this was because certain force calculations were causing groups of bodies to be approximated as a single pseudobody, after other had strayed too far from the pseudobody's quadtree's node's centre of mass.

When bodies are in close proxmity, anomalous total energies are observed1. Furthermore, simulations with higher numbers of bodies produce more random low energy outliers (likely due to a greater number of close encounters). There seems to be no signficant difference between Euler method and *Leapfrog*, with respect to observation of anomalies.

Performance & Scaling

Henceforth, the use of the ‘recognizable’ refers to eyeballing the output data, and making no significant effort to investigate the data more rigorously.

The *Barnes-Hut* algorithm appeared to be dominated by its constant factor.3 Generating input files with a high number of bodies that did not cause ’spontaneous combustion’ (where bodies would fly away from each other in every direction), and unconstrained Quadtree growth (where bodies achieve escape velocity, rapidly expanding the size of the tree and causing a collapse in simulation accuracy) proved difficult. Further testing on `goliath` is needed to discover the n that causes *Barnes-Hut* to be more performant than the brute-force approach.

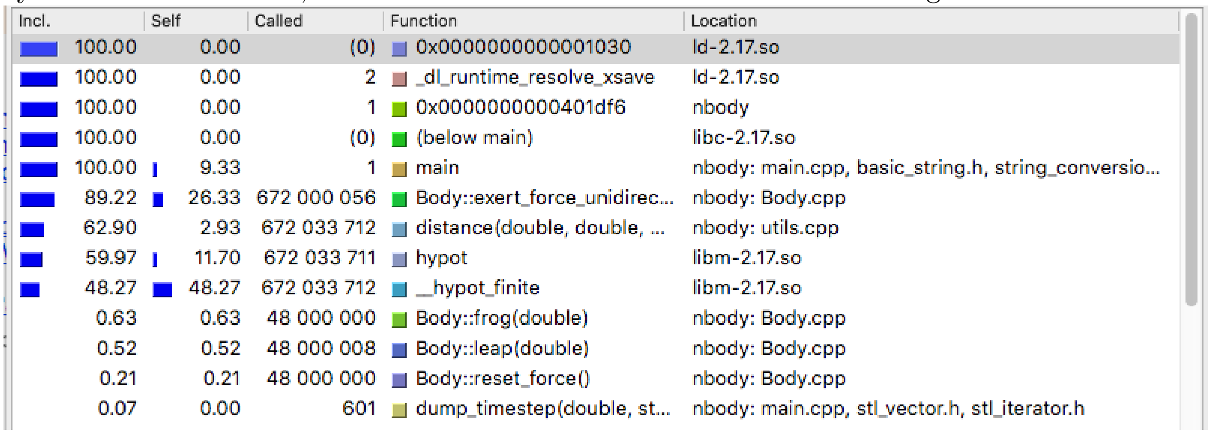
Strangely, $n = 16$ with *Barnes-Hut* enabled took longer than expected, when compared to $n \in \{4, 8, 12\}$. Further investigation is required.

The addition of the `-march=native` compiler flag, which enables the use of all CPU specific instructions, provided no recognizable improvement in running time, but was left enabled in the instance that it improved performance on `goliath`.

The use of both th GCC and Clang Profile-Guided Optimisation features provided no recognizable improvement in running time.

Distressingly, `-O0`, `-O1`, `-O2`, `-O3` showed no recognizable improvement in running time. `-Ofast` led to an -4%-ish performance regression.

By profiling with `callgrind`, we saw that execution was dominated by only one incredibly costly user-defined method, with an exclusive cost of 26.33% of total running time.



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000001030	ld-2.17.so
100.00	0.00	2	_dl_runtime_resolve_xsave	ld-2.17.so
100.00	0.00	1	0x0000000000401df6	nbody
100.00	0.00	(0)	(below main)	libc-2.17.so
100.00	9.33	1	main	nbody: main.cpp, basic_string.h, string_conversio...
89.22	26.33	672 000 056	Body::exert_force_unidirec...	nbody: Body.cpp
62.90	2.93	672 033 712	distance(double, double, ...	nbody: utils.cpp
59.97	11.70	672 033 711	hypot	libm-2.17.so
48.27	48.27	672 033 712	__hypot_finite	libm-2.17.so
0.63	0.63	48 000 000	Body::frog(double)	nbody: Body.cpp
0.52	0.52	48 000 008	Body::leap(double)	nbody: Body.cpp
0.21	0.21	48 000 000	Body::reset_force()	nbody: Body.cpp
0.07	0.00	601	dump_timestep(double, st...	nbody: main.cpp, stl_vector.h, stl_iterator.h

Performance fixes were divised.

```
 void Body::exert_force_unidirectionally(const Body& there) {
     const double m1 = m;
     const double m2 = there.m;

     const double delta_x = there.x - x;
     const double delta_y = there.y - y;

-    const double r = distance(x, y, there.x, there.y);
+    const double r = hypot(delta_x, delta_y);
-    const double r2 = r * r;
+    const double r2 = pow(r, 2);

     const double F = (G * m1 * m2) / r2;

     // turn the displacement vector between our two points into a force vector
     // of the desired magnitude
     const double scale_factor = F / r;
     Fx += delta_x * scale_factor;
     Fy += dumpsdelta_y * scale_factor;
 }
```

Instead of recalculating Δx and Δy twice (the second time in `distance`), we calculate them only once and make a direct call to `hypot`, rather than making a call to `distance`. We also used the specialized `pow` provided by `cmath`. This yielded a recognizable 15%ish performance improvement.

```
diff --git a/Assignment_1/src/Body.cpp b/Assignment_1/src/Body.cpp

+void Body::exert_force_bidirectionally(Body& there) {
+    const double m1 = m;
+    const double m2 = there.m;
+
+    const double delta_x = there.x - x;
+    const double delta_y = there.y - y;
+
+    const double r = hypot(delta_x, delta_y);
+    const double r2 = pow(r, 2);
+
+    const double F = (G * m1 * m2) / r2;
+
+    // turn the displacement vector between our two points into a force vector
+    // of the desired magnitude
+    const double scale_factor = F / r;
+
+    Fx += delta_x * scale_factor;
+    Fy += delta_y * scale_factor;
+
+    there.Fx -= delta_x * scale_factor;
+    there.Fy -= delta_y * scale_factor;
+}

diff --git a/Assignment_1/src/main.cpp b/Assignment_1/src/main.cpp
@@ -258,8 +259,7 @@ int main(int argc, char **argv) {
                 for (size_t j = i + 1; j < bodies.size(); j++) {
                     auto& y = bodies[j];
-                    x.exert_force_unidirectionally(y);
-                    y.exert_force_unidirectionally(x);
+                    x.exert_force_bidirectionally(y);
                 }
             }
         }
```

This fix halved execution time, for obvious reasons.[1].

```
diff --git a/Assignment_1/src/Body.cpp b/Assignment_1/src/Body.cpp
@@ -83,11 +82,11 @@ double Body::kinetic_energy() const {
 double Body::gravitational_potential_energy(const Body& there) const {
```

[1]I had to throw out all my old profile data

```
     const double R = distance(x, y, there.x, there.y); // final distance, aka, to edge

-    return (-G * m * there.m) / R;
+    return (-Gm * there.m) / R;
 }

@@ -101,7 +100,7 @@ void Body::exert_force_unidirectionally(const Body& there) {

-    const double F = (G * m1 * m2) / r2;
+    const double F = (Gm * m2) / r2;

@@ -113,7 +112,6 @@ void Body::exert_force_unidirectionally(const Body& there) {

-    const double F = (G * m1 * m2) / r2;
+    const double F = (Gm * m2) / r2;

@@ -189,6 +189,7 @@ std::vector<Body> parse_input_file(std::ifstream& input_fh) {

     for (size_t i = 0; i < bodies.size(); i++) {
         bodies[i].m = masses[i];
+        bodies[i].Gm = G * masses[i];
     }
```

Precomputing Gm yielded a 3%ish performance improvement.

Parallelism

Implementation

OpenMP

I slapped as many `#pragma omp parallel for shared(bodies)` on as many loops as I could find, and quickly discovered massive lock contention issues associated with `#pragma omp critical`. Pairwise force updates between bodies:

```
    for (size_t i = 0; i < bodies.size() - 1; i++) {
        auto& x = bodies[i];

        for (size_t j = i + 1; j < bodies.size(); j++) {
            auto& y = bodies[j];
                x.exert_force_bidirectionally(y);
        }
    }
```

where

```
void Body::exert_force_bidirectionally(Body& there) {
    ...
    #pragma omp critical
    {
    Fx += delta_x * scale_factor;
    Fy += delta_y * scale_factor;
    there.Fx -= delta_x * scale_factor;
    there.Fy -= delta_y * scale_factor;
    }
}
```

which had provided massive performance gains in the serial implementation over unidirectional updates was now causing massive parallel slowdowns, where increasing the `OMP_NUM_THREADS` caused a proportional increase in walltime. Instead, a `critical`-less comparison schema was devised,

```
#pragma omp parallel for shared(bodies)
for (size_t i = 0; i < bodies.size(); i++) {
    auto& x = bodies[i];

    for (size_t j = 0; j < bodies.size(); j++) {
        auto& y = bodies[j];

        if (&bodies[i] != &bodies[j]) {
            // XXX: Do NOT swap these around. You will cause
            // race conditions
            x.exert_force_unidirectionally(y);
        }
    }
}
```

that provided the sought after parallel speedup. This parallel speedup was found to be present at where n-bodies was high. Testing focussed primarily on $n > 5000$,[2] where force and integration calculation overhead dominated thread creation overhead. Parallel speedup vanished around the $n \approx 500$ mark.

MPI

Looking at our high-level implementation, the following two steps consume the vast majority of our computing power:

1. Inserted all `Body`s into a fresh `QuadTree` on full timesteps, traversing the `QuadTree` with every `Body` to calculate forces ($O(n \log n)$)
2. Performed the appropriate *Leapfrog* step to derive new velocities *or* positions

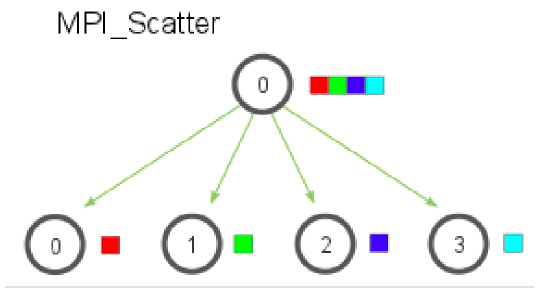
The *Leapfrog* step is the easiest step to distribute amongst many nodes, as it does not require the comparison of a body to all other bodies - each node can compute the step on a subset of all bodies, and later combine each subset with relatively simple MPI code.

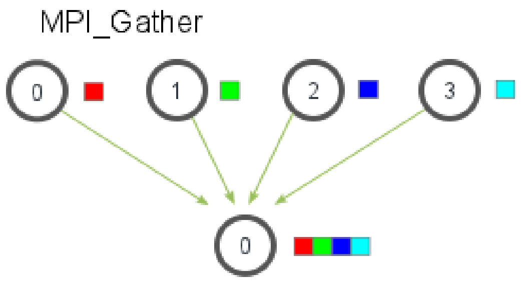
This can be achieved with `MPI_Scatterv` and `MPI_Gatherv` [3]. We decompose each dimension of our `Body`s into arrays, (e.g. the x position, the x veolocity, etc.), `MPI_Scatterv` these arrays,

[2]In MS1, I failed to test simulations with a large number of n, as I was focussed on testing the accuracy of the implementation, and generated input files would cause high numbers of collisions that made accuracy assessment difficult. Testing $n = 10000$, we find a 10x speedup with by using a serial Barnes-Hut implementation (Avg. 1.12s, as opposed to 9.87s without Barnes-Hut). Barnes-Hut afforded a much simpler parallelization scheme, as each body could calculate its force against the quadtree concurrently, without locks

[3]These `v` variants support custom chunking sizes e.g. if we have 15 bodies we can send $4, 4, 4, 3$ bodies to each node respectively, which `MPI_Scatter` and `MPI_Gather` do not support dynamic chunking

recompose them into `Body`s, run `Leapfrog` on this subset on each node, decompose the subset into arrays, `MPI_Gatherv` these arrays back to the root node, and recompose them into `Body`s.





```
std::vector<Body> sbodies = scatter_bodies(bodies, size, rank);

#pragma omp parallel for shared(bodies)
for (size_t i = 0; i < sbodies.size(); i++) {
    auto& body = sbodies[i];

    if (step % 2 == LEAP) {
        body.leap(timestep);
    }
    else {
        body.frog(timestep);
    }
}

bodies = gather_bodies(sbodies, size, rank, bodies.size());
```

where `sbodies` is our subset of bodies distributed to each node.

Because force calculation is only performed on the root node, this messaging architecture is *master-slave*, where the root node coordinates gathers and scatters. Slave nodes, are, unfortunately, underutilized. Further work could be sent distributing force calculation work, doing away with a root node entirely, except for logging.

```
Total CPU time on rank 0 was 89.820000
Total CPU time on rank 2 was 36.750000
Total CPU time on rank 3 was 36.960000
Total CPU time on rank 1 was 37.040000
```

Verification & Correctness

Because of my confidence in the correctness of the serial implementation, it served as a reference implementation for the verification of the parallel implementation.

Both implementations were run with identical input parameters, and their outputs were `diff`d. When run without MPI enabled, outputs were identical. When run with MPI, floating point abnormalities (initially affecting the least significant digits of each floating point value) gradually caused each diff to diverge. I hypothesize that the MPI serialization and deserialization procedure was truncating precision and causing this divergence. Perhaps some option exists to ensure that MPI guarantees floating point precision.

Scalability

We have mulitple dimensions that we can change to measure the scalability of the parallel implementation.

1. The number of bodies (n) (`512, 1024, 2048, 4096, 8192, 16384`)
2. `ENABLE_BARNES_HUT` (`true, false`)
3. `--nodes` (`Disabled, 1, 4, 16`)
4. `--ntasks` (`1, 4, 16`)
5. `--ntasks-per-node` (`1, 4`)
6. `--cpus-per-task` (`1, 4, 16`)
7. `#pragma omp parallel schedule(static, dynamic, guided)`

Running tests with all valid combinations of these variables, with a `numTimeSteps` as low as 8, would produce useful data, across multiple dimension, with minimal load on cluster. It would be unlikely that this plan would be curtailed.

Testing

Changes since MS2

In MS2, we decomposed each dimension of our `Body`s into arrays, (e.g. the x position, the x veolocity, etc.), `MPI_Scatterv`ed these arrays, recomposed them into `Body`s, ran `Leapfrog` on this subset on each node, decomposed the subset into arrays, `MPI_Gatherv`ed these arrays back to the root node, and recomposed them into `Body`s. We were also allocating a fresh new `Vector` after `Body` recomposition. This was, frankly, insane, and took a program with very low memory usage and pressure in the serial implementation to one that constantly allocated and deallocated memory on every single step.

By declaring `MPI_Datatype MPI_Body` with `MPI_Type_contiguous(8, MPI_DOUBLE, &MPI_Body)`, we're now able to send and receive subsets of `Body`s with just a single `MPI_Scatterv` and `MPI_Gatherv` call. We also `MPI_Scatterv` and `MPI_Gatherv` directly into `Vector`s allocated at the start of the program, without intermediate array allocations. This eliminated transient out-of-memory errors, and likely reduced MPI overhead.

Report content This content adds to the content from previous reports. The ECP describes the previous reports as chapters, you can use this opportunity to do a cleanup edit of your previous content but you aren't required to do a major rewrite. The content you will be adding should contain: The testing actually performed [Do not rewrite your previous section to match this]. Comment on any differences between your test plan and what you actually executed. Give your results. Analyse your results. Does your implementation scale? Summarise your findings for the semester (eg what worked and what didn't) including any limitations you expect scaling up your implementation to solving truely massive problem instances.

Marking criteria: /20 Code compiles out of repo /2 Testing description /8 Analysis /6 Summarise findings /4

References

[1] J. E. Barnes and P. Hut, "A hierarchical O(n-log-n) force calculation algorithm," *Nature*, vol. 324, p. 446, 1986.

[2] W. Dehnen and J. I. Read, "N-body simulations of gravitational dynamics," *European Physical Journal Plus*, vol. 126, p. 55, May 2011.

[3] R. D. Skeel, "Variable step size destabilizes the störmer/leapfrog/verlet method," *BIT Numerical Mathematics*, vol. 33, pp. 172–175, Mar 1993.

Figure 1: Observed energy anomaly while bodies in close proxmity - Leapfrog

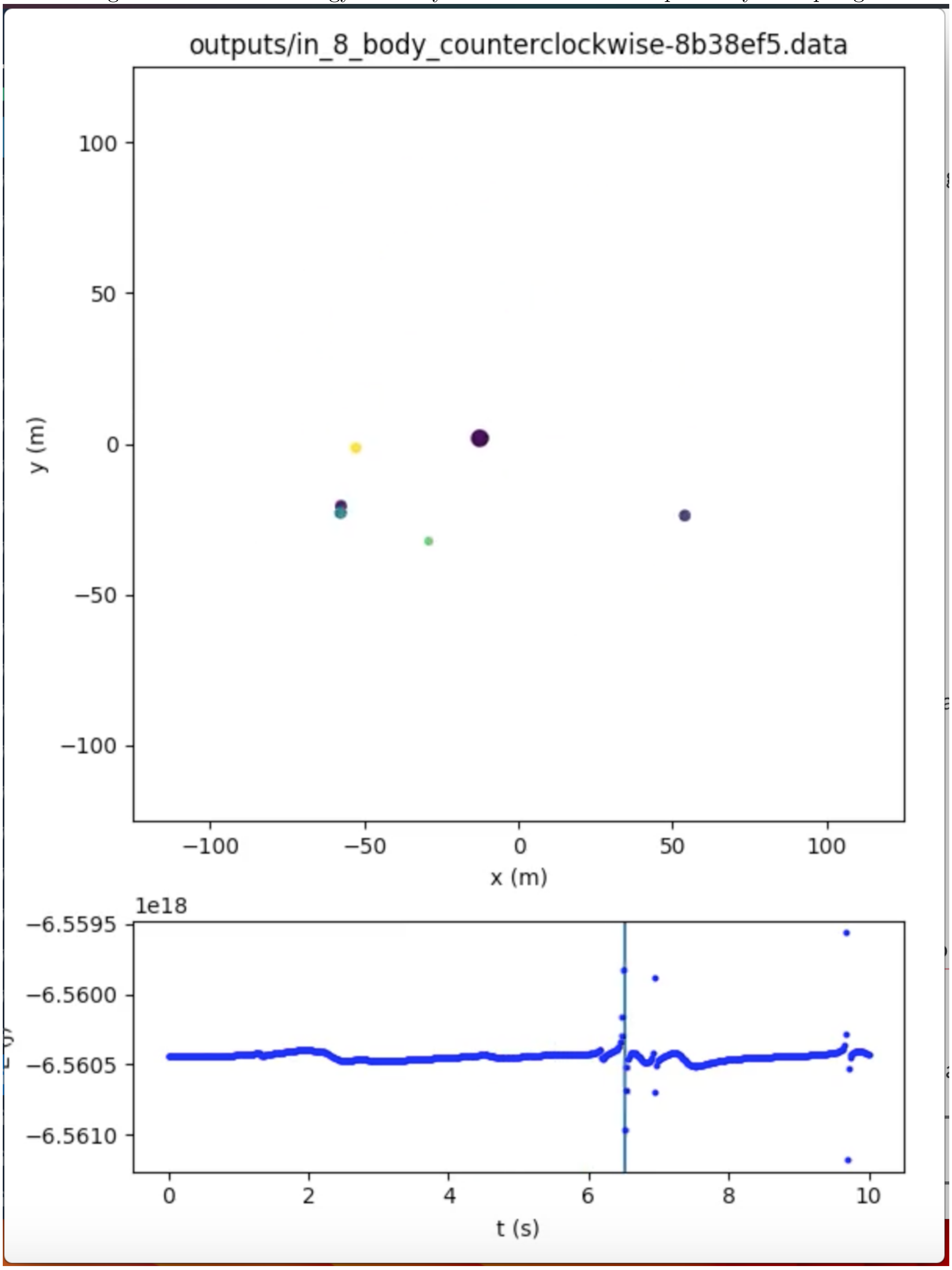


Figure 2: Barnes-Hut energy variation - Leapfrog

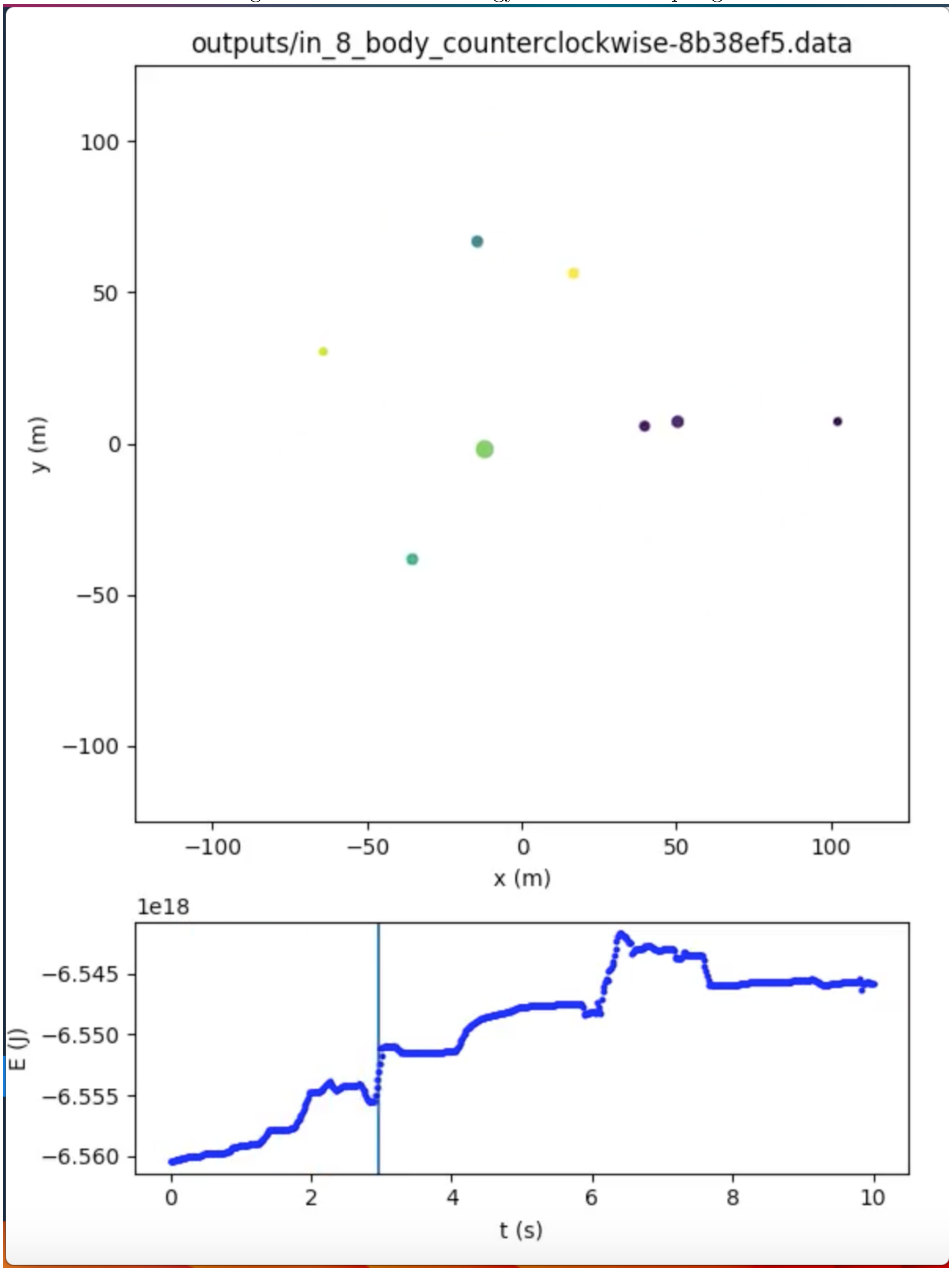
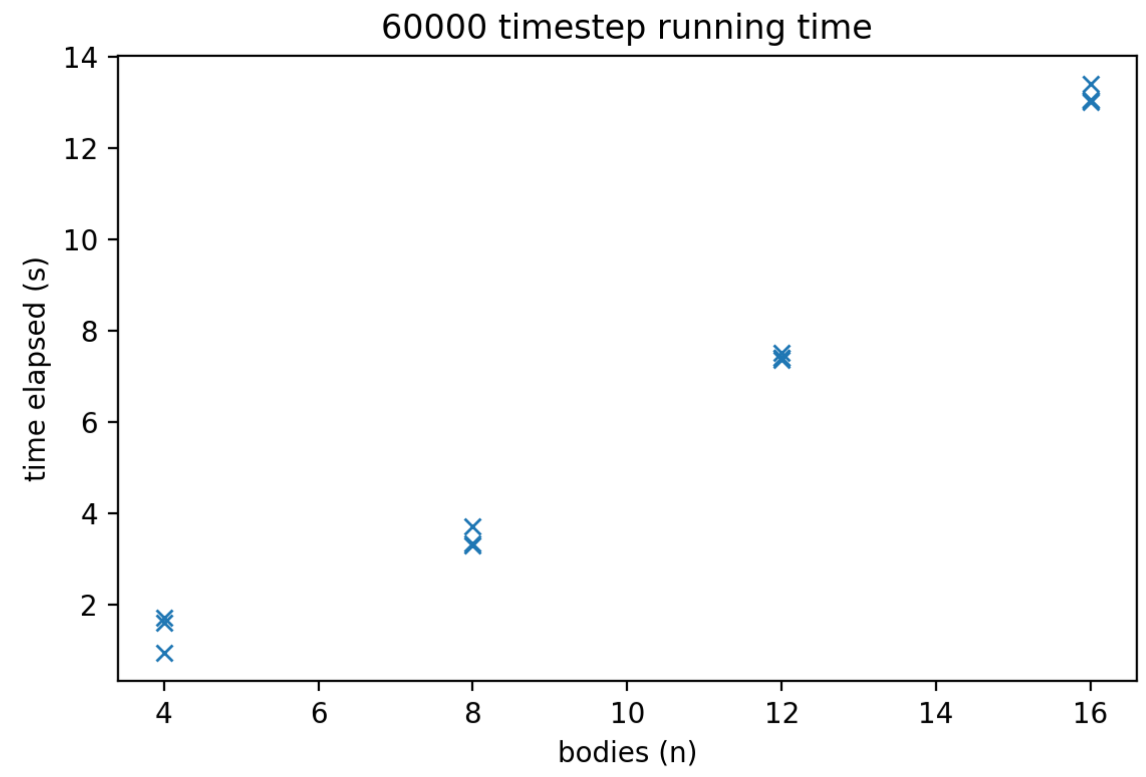
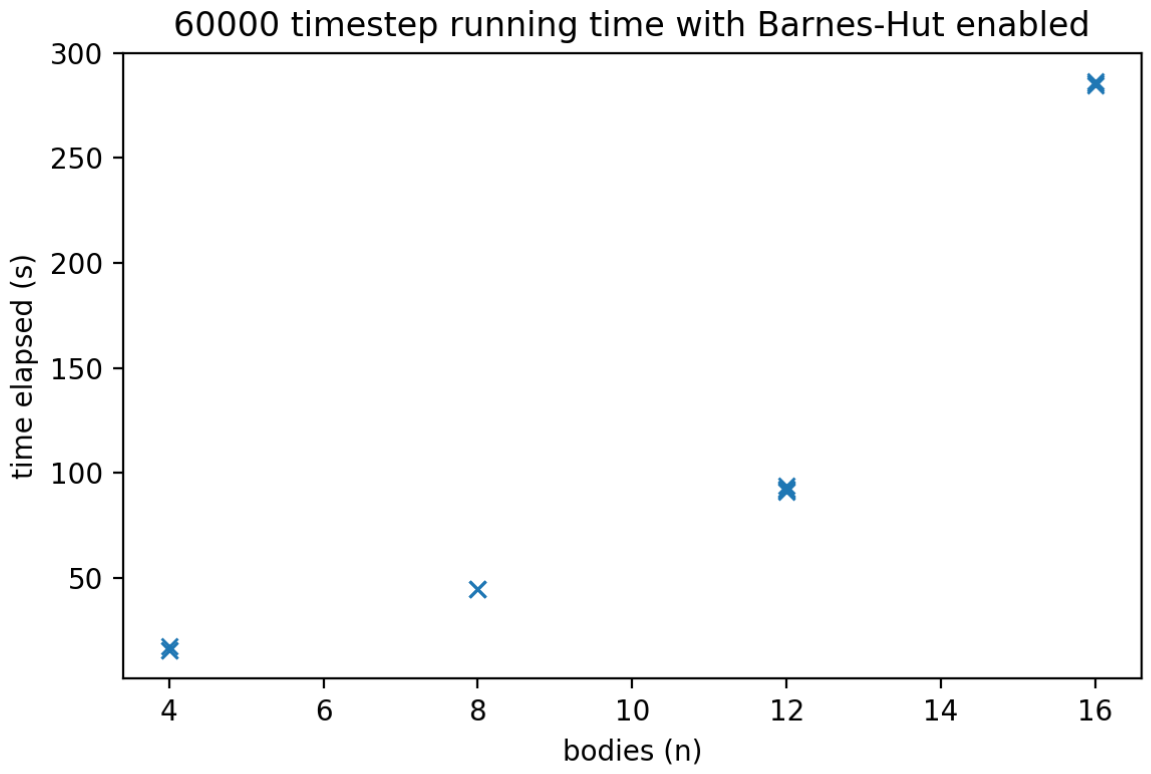


Figure 3: Scaling characteristics


60000 timestep running time
time elapsed (s)
bodies (n)


60000 timestep running time with Barnes-Hut enabled
time elapsed (s)
bodies (n)