

ECE 759
High Performance Computing for Engineering Applications
Assignment 4
Due 3/17/2025 at 23:59 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment4.pdf. Submit all plots (if any) on Canvas. Do not zip your Canvas submission.

All *source files* should be submitted in the **HW04** subdirectory on the **main** branch of your homework **git** repo with no subdirectories. Your **HW04** folder should contain **task2.cpp** and **task3.cpp**.

All commands or code must work on *Euler* without loading additional modules unless specified otherwise. A program may behave differently on your computer, so be sure to test on *Euler* before you submit. Note that this assignment is relevant to OpenMP, so the following line needs to be added to your **slurm** script:

- **#SBATCH --cpus-per-task=8** (or **-c 8** for short) should be added, which requests one node with 8 virtual cores (note the slight misnomer – Linux refers to virtual cores as cpus). The maximum number of threads required in this assignment is 8, so you should not ask for more than 8 cores.

Please submit clean code. Consider using a formatter like **clang-format**.

* Before you begin, copy the provided files from **Assignments/HW04** directory of the **ECE 759 Resource Repo**. Do not change any of the provided files since these files will be overwritten with clean, reference copies when grading.

Specify your GitHub link here:

The GitHub link to your code folder should be:
<https://github.com/FusionSandwich/repo759/tree/main/Assignments/HW04>

In this assignment, we will simulate a dynamical system of particles interacting gravitationally, commonly referred to as the **N-body** problem. Such simulations can model systems like the orbits of planets in the Solar System or the movement of stars in a galaxy.

First, review the N-body Python code in the file located at **Assignments/HW04/nbody.py**. Below is an introduction to the N-body problem:

Force Calculation:

We will assume a system of N point particles, indexed by $i = 1, \dots, N$. Every particle has the following properties:

- mass m_i
- position $r_i = [x_i, y_i, z_i]$
- velocity $v_i = [v_{x_i}, v_{y_i}, v_{z_i}]$

Each particle experiences the gravitational force exerted by all other particles according to Newton's law of universal gravitation, which follows the inverse-square law. The acceleration a_i of particle i is given by:

$$a_i = -G \sum_{j \neq i} \frac{m_j}{|r^j - r^i|^3} (r^j - r^i)$$

where G is the Gravitational constant.

To implement this, we have a Python function that computes the gravitational force on each particle using an input matrix of size $N \times 3$ that contains the positions of all particles:

```

1 def getAcc(pos, mass, G, softening):
2     """
3     Calculate the acceleration on each particle due to Newton's Law pos is an N x 3 matrix of positions
4     mass is an N x 1 vector of masses
5     G is Newton's Gravitational constant softening is the
6     softening length a is N x 3 matrix of accelerations
7     """
8
9
10    N = pos.shape[0] a =
11    np.zeros((N, 3))
12
13    for i in range(N):
14        for j in range(N):
15            if i != j:
16                dx = pos[j, 0] - pos[i, 0] dy = pos[j, 1] - pos[i, 1] dz = pos[j, 2] - pos[i, 2] inv_r3 = (dx**2 +
17                dy**2 + dz**2 + softening**2) ** (-1.5) a[i, 0] += G * (dx * inv_r3) * mass[j, 0] a[i, 1] +=
18                G * (dy * inv_r3) * mass[j, 0] a[i, 2] += G * (dz * inv_r3) * mass[j, 0]
19
20
21    return a
22
23
24
25

```

The *softening* parameter in the code is a small value added to avoid numerical issues when two particles are very close to each other. In such cases, the acceleration predicted by the inverse-square law would approach infinity. In reality, masses are not perfect point particles and have a finite size, so this parameter prevents unrealistic behavior in the simulation by smoothing out the gravitational force at short distances.

Time Integration:

The positions and velocities of the particles are updated using a 'kick-drift-kick' method. For each timestep Δt , the following steps are applied to every particle:

1. Half-step 'kick': Update the velocity based on the current acceleration:

$$v_{iT+1} = v_{iT} + \frac{\Delta t}{2} \times a_{Ti}$$

2. Full-step 'drift': Update the position using the new velocity:

$$r_{iT+1} = r_{iT} + \Delta t \times v_{iT}$$

3. Another half-step 'kick': Update the velocity again based on the new acceleration.

```

# Simulation Main Loop
for i in range(Nt): # (1/2) kick vel += acc * dt / 2.0

    # drift pos += vel * dt

    # ensure particles stay within the board limits pos[pos > board_size] =
    board_size pos[pos < -board_size] = -board_size

    # update accelerations
    acc = getAcc(pos, mass, G, softening)

    # (1/2) kick vel += acc * dt / 2.0

    # update time t += dt

```

This evolution is implemented in the code using a for-loop and the acceleration function defined earlier:

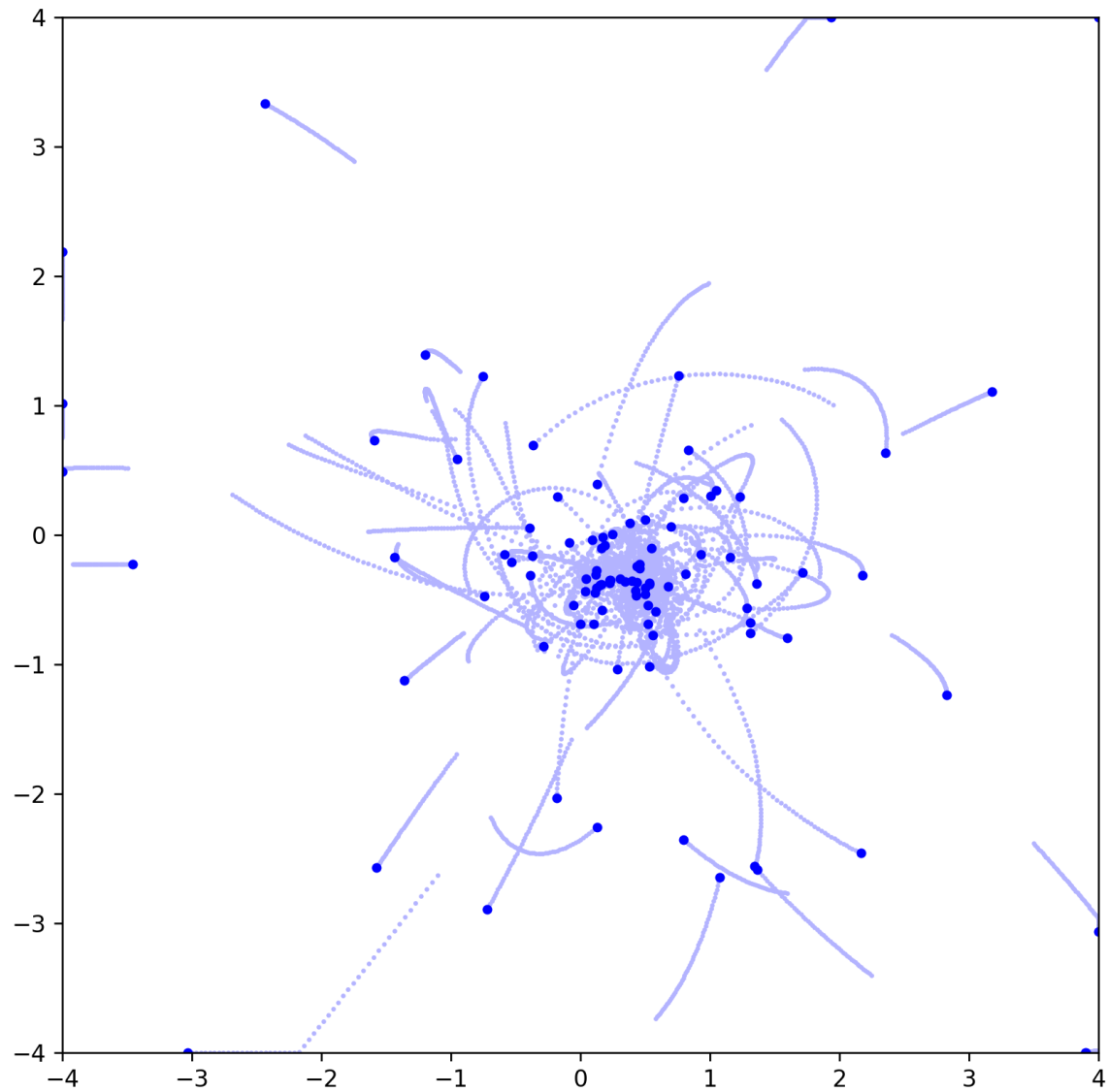
1
2
3
4
5
6
7
8
9 10
11
12
13
14
15
16
17
18
19
20

Initial Conditions:

To run the simulation, we need to specify the initial positions and velocities of the particles at time $t = 0$. You can experiment with different setups in the provided code to observe how the system evolves under various initial conditions.

Problem 1. Please review the introduction to the N-body problem above. Ensure that you understand both the algorithm and the Python code.

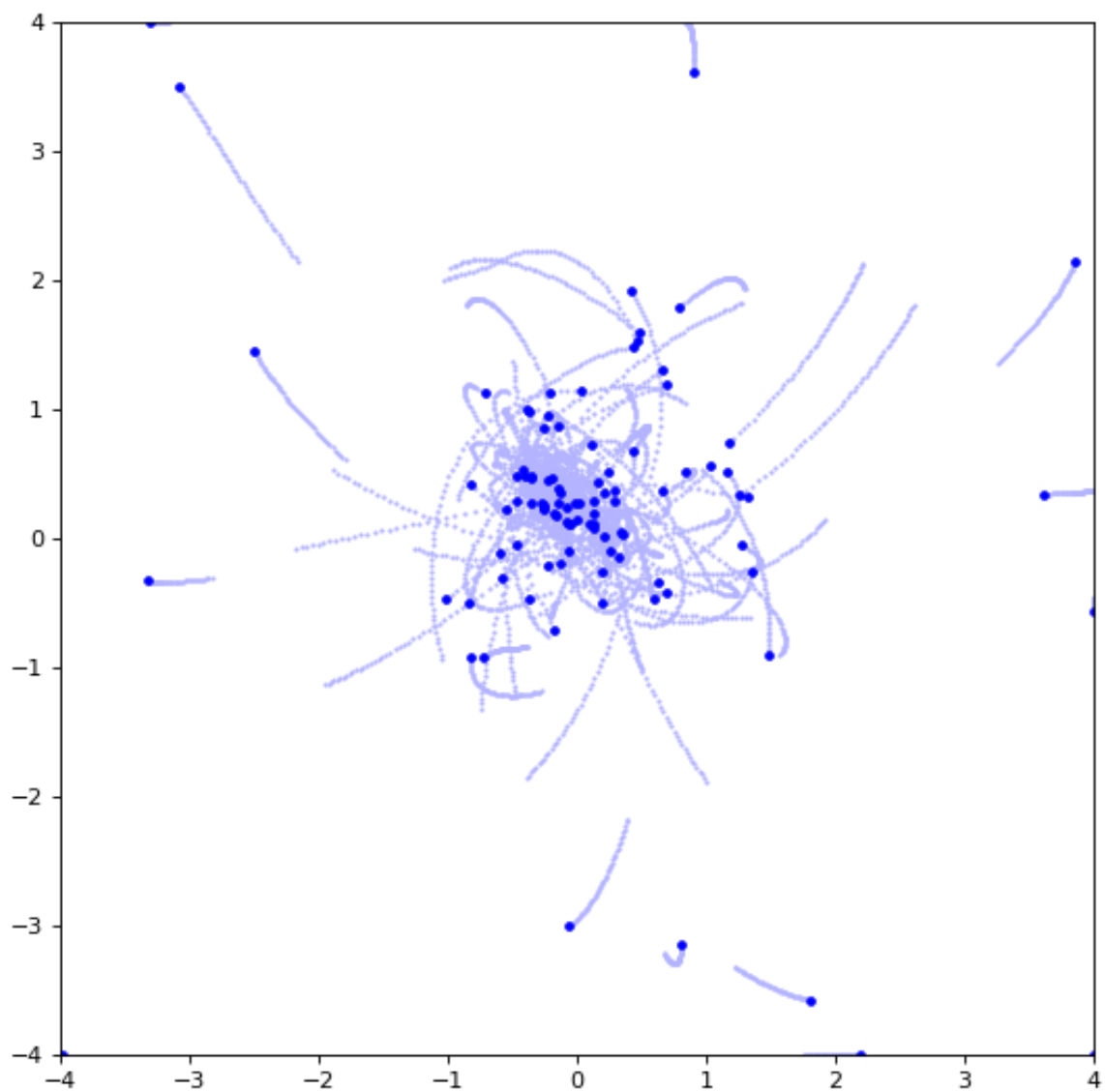
- a) Install Python library numpy and matplotlib: `python3 -m pip install numpy matplotlib`
- b) Run the `nbody.py` script either on your personal machine or on the Euler compute node. If you run it locally, you will see an animation of the N-body simulation. At the end of the simulation, the code will automatically save a plot. Rename this plot as `task1.png` and submit it to Canvas.



Problem 2. Port the N-body simulation code into C++. A skeleton code is provided in [Assignments/HW04/task2.cpp](#).

- a) Write a program [task2.cpp](#) that accomplish the following:
- Complete the provided skeleton code from [task2.cpp](#).

- Compile: `g++ task2.cpp -Wall -O3 -std=c++17 -o task2`
- Run the program; make sure you use `Slurm`:
`./task2 number of particles simulation end time`
- (Optional) For debugging and visualization, we provide a function `savePositionsToCSV`. This function saves the positions at each iteration to a CSV file. You can then use `plot_positions.py` to visualize the C++ simulation results. However, if you're measuring performance, disable this function as it introduces significant I/O overhead.



Problem 3. Parallelize the N-body C++ code using OpenMP.

- a) Write a program `task3.cpp` that accomplish the following:

- Add an argument `num_threads`, which specifies the number of threads to launch with OpenMP.
- Insert OpenMP directives (`#pragma omp parallel for`) in sections of the code that can be parallelized.
- Compile: `g++ task3.cpp -Wall -O3 -std=c++17 -o task3 -fopenmp`
- Run the program; make sure you use `Slurm` and request multiple CPU cores:
`./task3 number of particles simulation end time num threads`

Problem 4. Let's do experiments using different OpenMP scheduling policies.

- In `task3.cpp`, try the following scheduling policies: `static`, `dynamic`, and `guided`.
- On *Euler*, via `Slurm` do the following:

For each scheduling policy, run `task3` with the following parameters:

- `number of particles` = 100 or larger
- `simulation end time` = 100 or larger
- `num threads` = 1, 2, ..., 8

Generate plots called `task4.pdf` that show the time taken by the N-body simulation versus `num threads` on a linear-linear scale. The `task4.pdf` should contain three plots, one for each scheduling policy. Feel free to share the plots on Piazza.

Task4.pdf is in the repo