Parallel N-Body Simulation in C++ (with Parallel-For Construct)

Goal: Implement a parallel N-body simulation in C++ to compute the interactions between particles based on gravitational forces and simulate their motion using parallel for loops.

Before starting, note that the problem looks more complicated than it actually is.

1 Overview of the N-Body Problem

The N-body problem involves predicting the individual motions of a group of particles that interact with each other through gravitational forces. This problem is foundational in astrophysics, computational physics, and numerical simulations.

Key aspects of the N-body problem:

- **Gravitational Force**: Each particle in the system exerts a gravitational force on every other particle. The force between two particles is computed using Newton's law of gravitation:

$$F = G\frac{m_1 m_2}{r^2}, \tag{1}$$

  where $G = 6.674 \times 10^{-11}$ is the gravitational constant, m_1 and m_2 are the masses of the two particles, and r is the distance between them.

  The direction of the force is from the particle that applies the force and towards the particle that it is applied on. That is to say, for particles 1 and 2 at locations l_1 and l_2, the force applied by particle 2 on particle 1 is

$$\vec{F} = \frac{l_1 - l_2}{||l_1 - l_2||} G\frac{m_1 m_2}{||l_1 - l_2||^2}.$$

- **Softening Factor**: A small value is added to r^2 (or $||l_1 - l_2||^2$) to prevent numerical instability when two particles are very close (to avoid division by zero).

- **Equations of Motion**: The motion of each particle is updated based on the net gravitational force acting on it:

$$\vec{a} = \frac{\vec{F}}{m}, \quad \vec{v}_{new} = \vec{v}_{old} + \vec{a}\Delta t, \quad \vec{x}_{new} = \vec{x}_{old} + \vec{v}_{new}\Delta t.$$

- **Applications**: The N-body problem is used to model planetary systems, galaxy formation, and interactions between celestial bodies.

2 Programming Requirements

TODO. Implement the parallel N-body simulation in C++ by completing the following tasks:

1. **Define the Simulation Structure**: Create a structure or class to represent the state of the simulation. This should include:

- Particle properties: masses, positions, velocities, and forces (using vectors to store these values for all particles).
- Initialization functions:
  - Random initialization of particle properties (masses, positions, velocities) using appropriate random distributions.
  - Predefined configurations such as the solar system model with planets.
  - Load from file (in the recommended single-line TSV format).

2. **Force Calculation with Parallelization**: Write functions to:
   - Reset forces to zero at the start of each time step.
   - Compute the gravitational force between every pair of particles using the provided **parallel-for construct** (`OmpLoop::parfor()`).
   - The outer loop (over particle i) should be parallelized, where each thread computes the total force acting on its assigned particle. This ensures no race conditions.
   - Use a small softening factor to prevent division by zero.

   Important: You *must use the provided parallel-for abstraction* (`OmpLoop` class in `omp_loop.hpp`) for all parallelism. Do **not** use raw `#pragma omp`, `omp_get_thread_num()`, or manual partitioning. This wrapper already handles scheduling and granularity.

3. **Parallel Integration of Motion**: Implement functions to:
   - Update particle velocities based on computed forces.
   - Update particle positions based on their velocities and the time step.
   - Parallelize these updates using the same `OpenMP::parfor()` construct.

4. **Output State**: Output the state of the simulation (positions, velocities, and forces) to standard output in the recommended format so that it can be visualized.

5. **Simulation Loop**: Write a main simulation loop that:
   - Iterates over a fixed number of time steps.
   - Resets forces, computes interactions, updates motion, and periodically prints the state.

Output Requirements. Your program should output the state of the simulation (positions, velocities, and forces of all particles) at regular intervals to standard output in the recommended format.

Command-Line Interface: Your program should accept the following arguments:

```
<input> <dt> <nbstep> <printevery> <nbthreads>
```

Where:

- `input`: a number for random initialization, "planet" for solar system, or a filename.
- `dt`: time step size.
- `nbstep`: number of iterations (steps).
- `printevery`: print frequency.
- `nbthreads`: number of threads to use in parallel execution.

Output format: Each simulation state should be written in a single line with:

- The number of particles.
- For each particle: mass, position (x, y, z), velocity (vx, vy, vz), and force (fx, fy, fz).
- Values separated by tabs (`.tsv` format).

Visualization: You can visualize your results with:

```
python3 plot.py solar.tsv solar.pdf 10000
```

This creates one page per state when given multiple lines of output.

3 Guidelines and Common Mistakes

You must:

- Use the provided `OmpLoop` class (`omp_loop.hpp`) for all parallelism.
- Compile with `-fopenmp`
- Set the number of threads and granularity using:

```
omp.setNbThread(nbthreads);
omp.setGranularity(<chunk size>);
```

Do not:

- Use raw OpenMP pragmas (`#pragma omp`), manual partitioning, or `omp_get_thread_num()`.

4 Benchmark

TODO: On Centaurus, test and report execution times for:

- Solar system model: `dt = 200`, `nbstep = 5000000`.
- Random 100 particles: `dt = 1`, `nbstep = 10000`.
- Random 1000 particles: `dt = 1`, `nbstep = 10000`.

Compare performance using different numbers of threads and discuss your parallel speedup.

5 Submission Instructions

Submit a single archive containing:

- Source code files
- A `Makefile`
- A `README` file with compile/run instructions.
- Output logs for at least three configurations.
- A short comparison of sequential vs. parallel performance.