HIGH PERFORMANCE PROGRAMMING
ASSIGNMENT 3

Yuanjing Yang, Linjia Zhong

February 2025

1 Introduction of Galaxy Simulation

1.1 The N-Body Problem

The N-Body Problem is a problem in physics that involves considering N point masses moving in a Newtonian reference system, with all the points influenced only by their mutual gravitational attraction [1]. According to Newton's law of gravitation in two dimension states, the force acting on particle i due to particle j is expressed by:

$$\mathbf{f}_{ij} = -\frac{Gm_i m_j}{r_{ij}^3}\mathbf{r}_{ij} = -\frac{Gm_i m_j}{r_{ij}^2}\hat{\mathbf{r}}_{ij}$$

where:

G is the gravitational constant;

m_i and m_j are the masses of the particles;

r_{ij} is the distance between the particles;

$\mathbf{r}_{ij}$ is the vector that represents the position of particle i relative to particle j;

$\hat{\mathbf{r}}_{ij}$ is the normalized distance vector.

If $\mathbf{e}_x$ and $\mathbf{e}_y$ are unit vectors in the x and y directions respectively, then we have:

$$\mathbf{r}_{ij} = (x_i - x_j)\mathbf{e}_x + (y_i - y_j)\mathbf{e}_y$$
$$r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$$
$$\mathbf{r}_{ij} = r_{ij}\hat{\mathbf{r}}_{ij}$$

Considering the distribution of N particles, We can straight-forwardly calculate the force exerted on particle i by the other $N-1$ particles(using C-style indexing), which is also the only force acting on particle i:

$$\mathbf{F}_i = -Gm_i \sum_{j=0, j\neq i}^{1} \frac{m_j}{r_{ij}^3}\mathbf{r}_{ij}$$

According to this formula, we can see that there is an inherent instability when $r_{ij} << 1$. Therefore, within the Plummer sphere model, we define a slightly modified force:

$$\mathbf{F}_i = -Gm_i \sum_{j=0, j\neq i}^{1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij}$$

where ϵ_0 is a small number and we will use 10^{-3} in this case.
To preserve global properties of the gravitational system (e.g. total energy),we should use the symplectic Euler time integration method.The velocity u_i and position x_i of particle i can be updated with:

$$\mathbf{a}_i = \frac{\mathbf{F}_i^n}{m_i}$$

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n + \Delta t \mathbf{a}_i$$
$$x_i^{n+1} == x_i^n + \Delta t \mathbf{u}_i^{n+1}$$

where:

Δt is the time step size;

$\mathbf{a}_i$ is the acceleration of particle$i$.

The time complexity is determined by the number of operations required for each time step. If we have N particles, we need to calculate the gravitational force exerted on each particle by the other N-1 particles. According to the formula, calculating the force on particle i takes $O(N-1)$ time. Then, we need to repeat this calculation for all N particles.Therefore, the total computation time is $O(N*(N-1))$, which simplifies to $O(N^2)$.For large values of N,the computational cost becomes significantly higher.

1.2 Problem Setting

In this simulation, we will implement a code that calculates the evolution of N particles in a gravitational simulation in two spatial dimensions,which approximates the evolution of a galaxy.
We have a file including an initial set of particles, with the particle masses and initial positions and velocities. And the positions of particles are in a $L \times W$ dimensionless domain (Use L = W = 1),which means that the x and y values will be between 0 and 1.
After simulating for a specified number of time steps, we will obtain a results file containing the final masses, positions, and velocities. These results can be visualized through graphics to observe the evolution.
Considering that our simulation sample is relatively small, and the total particles mass is also small, to make the gravitational effect more noticeable, we need gravity to scale inversely with the number of bodies.Thus, we set:

$$G = 100/N$$
$$\epsilon_0 = 10^{-3}$$
$$\Delta t = 10^{-5}$$

2 The Solution

2.1 Data structures

The binary input file format we used consist of a sequence of double numbers giving the mass,position x and y,velocity in the x and y direction, and "brightness" of each particle. The result file will has the same format.So the total file size will be $N * 6 *$ sizeof(double).
Among these properties, mass and "brightness" remain unchanged before and after execution.
Based on the file format, we construct the following structure to store the properties of the particles, and use an array to store all the particles:

```
typedef struct Particle {
    double x;       // position x (assumed to be in [0,1])
    double y;       // position y (assumed to be in [0,1])
    double mass;    // mass
    double v_x;     // velocity x
    double v_y;     // velocity y
    double bright; // brightness (can be used as a color
        value)
} Particle;
```

Listing 1: Particle structure

2.2 Code structure

```
  // Measuring time, learn from Lab5_Task4
  static double get_wall_seconds() {
      struct timeval tv;
      gettimeofday(&tv, NULL);
      return tv.tv_sec + (double)tv.tv_usec / 1000000;
  }
```

Listing 2: Measuring execution time using gettimeofday

The `get_wall_seconds()` function is used to measure the program's wall time, which refers to the cost time from system startup to the current moment, which is used for performance testing.

```
    // Read particles from a binary file, learn from
        Assignment2_part2
    void readfile(const char *filename, Particle *particle,
        const int N) {
        FILE *file = fopen(filename, "rb");
        if (file == NULL) {
            fprintf(stderr, "Cannot␣open␣input␣file:␣%s\n",
                filename);
```

```
            exit(1);
        }
        size_t n = fread(particle, sizeof(Particle), N, file
            );
        if(n != (size_t)N) {
            fprintf(stderr, "Error reading file. Expected %d
                particles, got %zu.\n", N, n);
            exit(1);
        }

        fclose(file);
    }
```

The function `readfile` is used to read particle data from a binary file. It opens the specified file and reads the corresponding number N of particles. It also performs error checking in case the file cannot be opened or the expected data is not read properly.

```
// Perform one simulation step (update accelerations,
    velocities, positions)
// Accept a_x and a_y as arguments, which avoids repeated
    memory allocation, improving performance. It comes from
    Deepseek.
// Here we use inline to reduce function calls, learned from
     Lab5_Task5
static inline void simulationStep(Particle *restrict
    particle, const int N, const double dt, double *restrict
    a_x, double *restrict a_y) {
    // F_i = -G * m_i * (m_j * r) / (r + eps)^3, precompute
        G to avoid redundant computations
    const double G = 100.0 / N;
    const double eps = 1e-3;

    // Reset accelerations to zero, learned from Lab5_Task4
    memset(a_x, 0, N * sizeof(double));
    memset(a_y, 0, N * sizeof(double));

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i == j)
                continue;  // Skip self-interaction
            // r_ij = (x_i - x_j) + (y_i - y_j)
            double r_x = particle[i].x - particle[j].x;
            double r_y = particle[i].y - particle[j].y;
            double r   = sqrt(r_x * r_x + r_y * r_y);
            double r_e = (r + eps) * (r + eps) * (r + eps);
            // Acceleration, a = -G * m_j * r / (r + eps)^3
            double temp = -G * particle[j].mass / r_e;
            a_x[i] += temp * r_x;
            a_y[i] += temp * r_y;
```

```
        }
    }

    for (int i = 0; i < N; i++) {
        // Velocity, v = dt * a
        particle[i].v_x += dt * a_x[i];
        particle[i].v_y += dt * a_y[i];
        // Position, x = dt * V
        particle[i].x   += dt * particle[i].v_x;
        particle[i].y   += dt * particle[i].v_y;
    }
}
```

The `simulationStep` function is designed to update the state of each particle in the galaxy simulation for one time step.It use the modified force equation that corresponds to Plummer spheres and the symplectic Euler time integration to calculate and update the accelerations,velocities, and positions.

```
// Write particles to a binary file, which we use chatgpt to
    modify our code
void binary_particle(const char *filename, const Particle *
   particle, const int N) {
    FILE *file = fopen(filename, "wb");
    if (file == NULL) {
        fprintf(stderr, "Cannot open output file: %s\n",
            filename);
        exit(1);
    }
    fwrite(particle, sizeof(Particle), N, file);
    fclose(file);
}
```

The `binary_particle` function is used to save the current state of the particles (e.g. position, velocity, mass, etc.) into a binary file, making it easier for computation or further analysis.

```
// Main simulation loop
while(step < nsteps) {
    simulationStep(particles, N, dt, a_x, a_y);

    step++;


    if (graphicsEnabled == 1) {
        ClearScreen();
        // when I use the input data's brightness, which is
            out of range, so we
        // need to normalize brightness to [0, 1]
        float min_bright = 0.0f;
        float max_bright = 1.0f;
```

```
14
15          for (int i = 0; i < N; i++) {
16              if (particles[i].bright < min_bright) min_bright
                    = particles[i].bright;
17              if (particles[i].bright > max_bright) max_bright
                    = particles[i].bright;
18          }
19
20          for (int i = 0; i < N; i++) {
21              particles[i].bright = (particles[i].bright -
                    min_bright) / (max_bright - min_bright);
22          }
23          // Draw each particle as a circle.
24          // Map particle coordinates (assumed in [0,1]) to
                screen coordinates, we learn it from chatgpt
25          for (int i = 0; i < N; i++) {
26              float screenX = (float)(particles[i].x *
                    windowWidth);
27              float screenY = (float)(particles[i].y *
                    windowHeight);
28              // making the radius proportional to the
                    particle's mass,  which comes from deepseek
29              float radius = 5.0f * (float)particles[i].mass ;
30              // use particle's brightness field for rendering
                    , which comes from deepseek
31              float color = (float)particles[i].bright;
32
33              DrawCircle(screenX, screenY, (float)windowWidth,
                     (float)windowHeight, radius, color);
34          }
35          Refresh();
36          // control the simulation speed, add a small delay,
                which comes from chatgpt
37          usleep(20000);
38
39          if (CheckForQuit()) break;  // Exit loop if quit
                signal received
40      }
41  }
```

This code snippet is the main simulation loop in the main function. It continues looping while step is less than the given N, simulating the particles' state over multiple steps, and displaying the results in a graphical window if needed.

3 Performance and Discussion

3.1 Implementation Time Measurement

```
int main(int argc, char *argv[]) {
    double startTime1 = get_wall_seconds();
    if (argc != 6) {
        printf("There should be galsim N filename nsteps
            delta_t graphics\n");
        return 1;
    }
```

The variable `startTime1` is initialized at the beginning of the main function, where the `get_wall_seconds` function is called to obtain the current wall time. And the program then continues to execute.

```
    double totalTimeTaken = get_wall_seconds() - startTime1
        ;
    printf("totalTimeTaken = %f\n", totalTimeTaken);
    return 0;
```

The variable `totalTimeTaken` calculates the difference between the current time at the end of the program execution and `startTime1`,which means the total running time of the program.

3.2 Experiments and Performance

Particles Number (N)	Runtime (s)
10	0.0073
20	0.0088
30	0.0083
40	0.0093
50	0.0110
60	0.0121
70	0.0148
80	0.0146
90	0.0179
100	0.0202
150	0.0370
200	0.0566
300	0.1106
400	0.2025
500	0.3123
600	0.4582
700	0.6090
800	0.7747
900	0.9843
1000	1.2390
1500	2.6919
2000	4.7969
3000	10.7788
4000	19.2148
5000	29.2542
6000	42.4580
7000	42.4580
8000	75.4925
9000	95.0584
10000	117.1811

Table 1: Simulation Runtimes for Different Particle Numbers

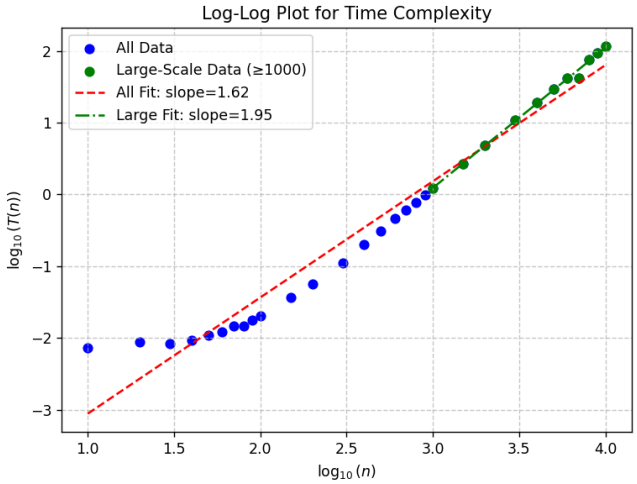

Figure 1: Log plot of Time complexity

From the log-log plot, we can see that the slope of all particle number is smaller than the expected slope, this is because At small particle counts, the entire dataset fits into the L1/L2 cache, resulting in extremely fast execution. As the particle count increases and the data moves into slower memory (L3 cache or RAM), the performance starts to reflect the expected $O(n^2)$ (the explanation of cache effects we learned from Chatgpt).

We calculate the running time for N=500,1000,2000 and 5000. From the figure above, we can see the relationship between the number of particles N and the simulation time. The curve follows a quadratic trend, indicating that the algorithm has a time complexity of $O(n^2)$.

3.3 Optimizations of Simulations

The optimization attempts as below:
inline: show a significant improvement, which reaches the improvement rate (74.59%), beacuse it can reduce functions calls.
loop unrolling: improve the performance by (25.07%), beacuse it can reduce loop overhead.
buffer: improve the performace slightly, it can store array contiguously in memory.
Techniques like **restrict** , **const**, **-ffast-math**, and **-march=native** have minimal impact.
We applied the optimization methods mentioned above. The following table shows a comparison of the running times for the optimized and original algorithms with different values of N.
The table above compares the performance improvement of different optimiza-

Table 2: Performance Improvement with Different Optimization Techniques (1500 Particles)

	Inline	Data Locality (Buffer)	Loop Unrolling	`restrict`	`const`	`-ffast-math`
Original	10.778086	2.750036	3.666598	2.782311	2.75322	2.751732
Optimization	2.738427	2.730311	2.747539	2.744423	2.74132	2.744206
Improvement Rate	74.59%	0.75%	25.07%	1.36%	0.77%	0.27%

tion methods applied to 1500 particles, all optimizations improve the performance.

3.4 Computer environment

CPU Model:Intel i7-14700HX
Compiler Version: gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0
Compiler Optimization option:
-O3 (for high-level optimizations)
-march=native
-ffast-math
-funroll-loops (loop unrolling)

References

[1] Meyer, K. R., Hall, G. R., & Offin, D. C. (2009). *Introduction to Hamiltonian dynamical systems and the N-body problem* (2nd ed.). Springer Science+Business Media. https://doi.org/10.1007/978-0-387-09724-4 p. 27

[2] High Performance Programming. (2022). *Assignment 3: The Gravitational N-Body Problem*. Uppsala University, Spring 2022.