Sam Vaharan
Civilingengörsprogrammet i teknisk fysik Uppsala universitet

projekt utfört: Februari 2025
High Performance Programming 1TD062 62013

Assignment

3
Ångströmslaboratoriet
February 20, 2025

UPPSALA
UNIVERSITET

Innehåll

1 Introduction

The gravitational N-body problem involves calculating the motion of N particles influenced by mutual gravitational forces. In our two-dimensional simulation, the gravitational force between two bodies is expressed as

$$\mathbf{F}_{ij} = -G \cdot m_i \sum_{j=0, j!=i}^{N-1} \frac{m_j}{(r_{ij} + \varepsilon_0)^3} \mathbf{r}_{ij}$$

where $\mathbf{r}_{ij}$ is the vector difference between the positions of particles i and j, $r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ is the distance between them, $G = 100/N$ is a gravitational constant scaled with the number of bodies, and $\varepsilon_0 = 10^{-3}$ is a softening factor used to avoid singularities when r_{ij} is very small.
The simulation uses the symplectic Euler time integration to update velocities and positions.

2 Code Structure and Implementation

The program performs the following steps:

1. Parse command-line arguments to obtain simulation parameters.
2. Read the initial particle data (positions, masses, velocities, and brightness) from a binary file.
3. For a specified number of timesteps, compute the gravitational forces between all particles and update their velocities and positions.
4. Write the final particle data to an output binary file.

2.1 Key Data Structures

A `Vector2D` struct is used to represent two-dimensional coordinates:

```
typedef struct {
    double x;
    double y;
} Vector2D;
```

Arrays of `Vector2D` are used for storing positions and velocities. The masses and brightness values are stored in dynamically allocated arrays.

2.2 Main Function Workflow

The main function begins by checking the number of arguments and parsing them. It then allocates memory for the particle properties and reads the initial configuration from a binary file. For each timestep, the program calculates the forces on each particle using the function `get_force_on_body` and updates the positions and velocities using `update_velocity_and_position`. Finally, it writes the simulation result to a binary file named `result.gal` and prints the total time taken for the simulation.

2.3 Force Calculation and Update Functions

The `get_force_on_body` function computes the net force on each particle by looping over all other particles, taking care to avoid self-interaction. A softening factor is added to the inter-particle distance to prevent numerical instabilities:

```
void get_force_on_body(const int nstars, const double G, const double e0,
                       Vector2D* position, double* mass, double* Fx, doub
    for (int i = 0; i < nstars; i++) {
        double sumx = 0;
        double sumy = 0;
        for (int j = 0; j < nstars; j++) {
            if (i != j) {
                double dx = position[i].x - position[j].x;
                double dy = position[i].y - position[j].y;
                double rije0 = sqrt(dx * dx + dy * dy) + e0;
                double pow_rije0 = 1.0 / (rije0 * rije0 * rije0);
                double temp = mass[j] * pow_rije0;
                sumx += temp * dx;
                sumy += temp * dy;
            }
        }
        Fx[i] = sumx * -G * mass[i];
        Fy[i] = sumy * -G * mass[i];
    }
}
```

The `update_velocity_and_position` function implements the symplectic Euler method:

```
void update_velocity_and_position(const double stepsize, const int nstars
                                  Vector2D* velocity, Vector2D* position,
                                  double* mass, double* Fx, double* Fy) {
    for (int i = 0; i < nstars; i++) {
        velocity[i].x += stepsize * Fx[i] / mass[i];
        velocity[i].y += stepsize * Fy[i] / mass[i];
        position[i].x += stepsize * velocity[i].x;
        position[i].y += stepsize * velocity[i].y;
```

```
    }
}
```

A helper function `get_wall_seconds` is used for wall-clock time measurements. The final code in its entirety is found in the Appendix.

3 Performance and discussion

All optimization tests were made on the Vitsippa server and 3000 stars with 100 timesteps and stepsize 10^{-5} and the following CFlags were used; *-O3 -g -Wall -Werror -march=native -funroll-loops -ffast-math*

Initial optimizations were applied to the mathematical computations even before the full code was operational. Based on previous lab experiences, it was known that avoiding a function call to `pow()` by instead computing the reciprocal and then multiplying yielded improved performance.

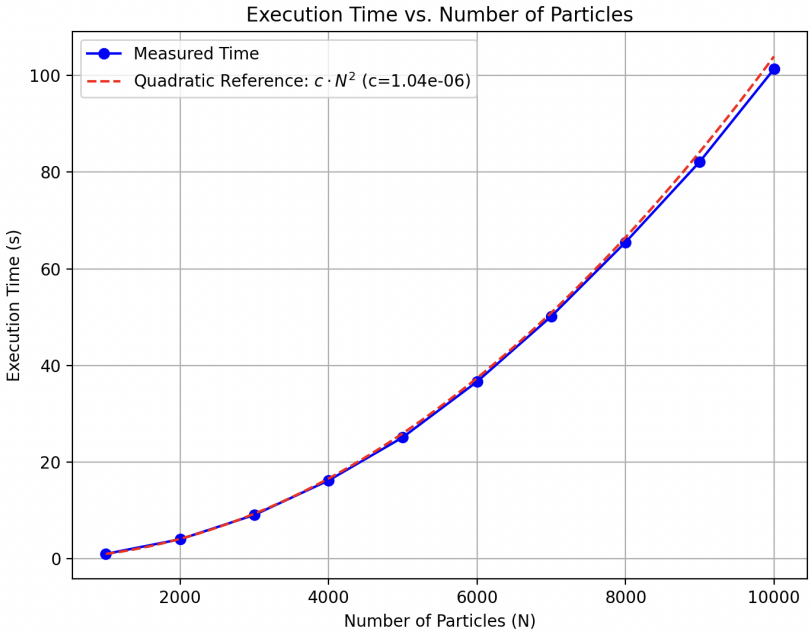
An attempt was made to merge the two functions responsible for the force and update calculations. However, this resulted in significant numerical errors, and the changes had to be reverted.

Another experiment involved eliminating the temporary variable in the force computation by directly accumulating the results into the force arrays. Contrary to expectations, this change increased the overall execution time by approximately 2 seconds. It is likely that this approach led to a higher number of memory accesses, which were more difficult for the compiler (using `-O3` optimizations) to optimize efficiently.

Finally, an effort was made to remove the `Vector2D` struct entirely. Although such a change is sometimes expected to improve performance, in this case it did not. Moreover, it made the code less intuitive for users.

Number of Particles (N)	**Time (s)**
1000	1.039002
2000	4.060855
3000	9.094192
4000	16.222872
5000	25.148232
6000	36.647541
7000	50.152841
8000	65.470331
9000	82.123578
10000	101.282502

Tabell 1: Measured execution time using 100 timesteps and stepsize 10^{-5}



Figur 1: Plot to show complexity N^2

4 Changes made

By incorporating the gravitational constant directly into the inner loop and updating both bodies' acceleration simultaneously, the modified code applies Newton's Third Law more efficiently. This reduces the need for an additional loop for scaling and potentially enhances performance by allowing for better vectorization. This made it so the final time for 3000 steps was recorded as **5.164 s** on Vitsippa, which is a great improvment. The changed code looks like this!

```
void get_force_on_body(const int nstars, const double G, const double
// Reset acceleration arrays to zero
for (int i = 0; i < nstars; i++) {
    ACCx[i] = 0;
    ACCy[i] = 0;
}
for (int i = 0; i < nstars; i++) {
    double posx = position[i].x;
    double posy = position[i].y;
    double mass_i = mass[i];
    for (int j = i+1; j < nstars; j++) {

        double dx = posx - position[j].x;
        double dy = posy - position[j].y;
        double rije0 = sqrt(dx * dx + dy * dy) + e0;
        double pow_rije0 = 1.0 / (rije0 * rije0 * rije0);

        double factor_i = mass[j] * pow_rije0;
        double factor_j = mass_i * pow_rije0;
```

```
                ACCx[i] -= factor_i * dx;
                ACCy[i] -= factor_i * dy;
                ACCx[j] += factor_j * dx;
                ACCy[j] += factor_j * dy;

            }
            ACCx[i] *= G;
            ACCy[i] *= G;
        }
}
```

A References

1. Copilot was used to help better comment the code and add standard error prints
2. get-wall-seconds function was taken from Lab6-Task02
3. The make file is one of the standard makefiles given in most labs
4. Some leading flags are used after Elsa Rosenblad said they would work better
5. The built in Writefull extension in overleaf was used for improved sentences and varied language
6. Finally this is the repository used for this assignment; `https://github.com/SamVarahram/Assignment3.git`

B Code

```
    // Copilot was used to help better comment the code and add standard
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include <sys/time.h>

// Struct to store the position of a body
typedef struct {
    double x;
    double y;
} Vector2D;

// Function prototypes
void get_force_on_body(const int nstars, const double G, const double e0,
```

```
void update_velocity_and_position(const double stepsize, const int nstars
static double get_wall_seconds();

int main(int argc, char* argv[]) {
    // Start the timer
    double start_time = get_wall_seconds();
    // Check if the correct number of arguments are given
    if(argc != 6) {
        fprintf(stderr, "Usage: %s <Number of stars> <input file> <Number
return 1;
    }

    // Read in command line arguments
    // Make everything const for optimization
    const int nstars = atoi(argv[1]);
    const char* input_file = argv[2];
    const int nsteps = atoi(argv[3]);
    const double stepsize = atof(argv[4]);
    const int graphics = atoi(argv[5]); // 0 or 1 for false or true
    if (graphics) {
        fprintf(stderr, "Graphics not supported\n");
        return 1;
    }

    const double G = 100./nstars;
    const double e0 = 1e-3; // Softening factor 10^-3

    // Create arrays to store the data and Vector2D to store the forces
    Vector2D* position = (Vector2D*) malloc(nstars * sizeof(Vector2D));
    double* mass = (double*) malloc(nstars * sizeof(double));
    Vector2D* velocity = (Vector2D*) malloc(nstars * sizeof(Vector2D));
    double* brightness = (double*) malloc(nstars * sizeof(double));

    if (position == NULL || mass == NULL || velocity == NULL || brightnes
        fprintf(stderr, "Error allocating memory\n");
        return 1;
    }
    // Initialize arrays to zero
    memset(position, 0, nstars * sizeof(Vector2D));
    memset(mass, 0, nstars * sizeof(double));
    memset(velocity, 0, nstars * sizeof(Vector2D));
    memset(brightness, 0, nstars * sizeof(double));

    // Read in the data
    FILE* file = fopen(input_file, "rb");
    if(file == NULL) {
        fprintf(stderr, "Error opening file\n");
        free(position);
```

```
    free(mass);
    free(velocity);
    free(brightness);
    return 1;
}


// Read in the input file
// Input file has structure:
/*particle 0 position x
particle 0 position y
particle 0 mass
particle 0 velocity x
particle 0 velocity y
particle 0 brightness
particle 1 position x*/
// The input file is binary
for (int i = 0; i < nstars; i++) {
    if (fread(&position[i], sizeof(Vector2D), 1, file) != 1 ||
        fread(&mass[i], sizeof(double), 1, file) != 1 ||
        fread(&velocity[i], sizeof(Vector2D), 1, file) != 1 ||
        fread(&brightness[i], sizeof(double), 1, file) != 1) {
        fprintf(stderr, "Error reading file\n");
        return 1;
    }
}
fclose(file);

double* Fx = (double*) malloc(nstars * sizeof(double));
double* Fy = (double*) malloc(nstars * sizeof(double));
memset(Fx, 0, nstars * sizeof(double));
memset(Fy, 0, nstars * sizeof(double));

// Loop over the timesteps
for (int time = 0; time < nsteps; time++) {
    get_force_on_body(nstars, G, e0, position, mass, Fx, Fy);
    update_velocity_and_position(stepsize, nstars, velocity, position
}




// Output the data in a binary file
FILE* output = fopen("result.gal", "wb");
if(output == NULL) {
    fprintf(stderr, "Error opening file\n");
    return 1;
}
```

```
    for (int i = 0; i < nstars; i++) {
        fwrite(&position[i], sizeof(Vector2D), 1, output);
        fwrite(&mass[i], sizeof(double), 1, output);
        fwrite(&velocity[i], sizeof(Vector2D), 1, output);
        fwrite(&brightness[i], sizeof(double), 1, output);
    }
    fclose(output);

    // Free the memory
    free(Fx);
    free(Fy);
    free(position);
    free(mass);
    free(velocity);
    free(brightness);

    // Print the time taken
    double end_time = get_wall_seconds();
    printf("Time taken: %f\n", end_time - start_time);
    return 0;

}

// Function to calculate the force on a body
void get_force_on_body(const int nstars, const double G, const double e0,
    for (int i = 0; i < nstars; i++) {
        double sumx = 0;
        double sumy = 0;
        for (int j = 0; j < nstars; j++) {
            if (i != j) {

                double dx = position[i].x - position[j].x;
                double dy = position[i].y - position[j].y;
                double rije0 = sqrt(dx * dx + dy * dy) + e0;
                double pow_rije0 = 1.0 / (rije0 * rije0 * rije0);

                double temp = mass[j] * pow_rije0; // pow(rij + e0, 3)
                sumx += temp * dx;
                sumy += temp * dy;
            }
        }
        Fx[i] = sumx * -G * mass[i];
        Fy[i] = sumy * -G * mass[i];
    }
}

// Function to calculate the acceleration of a body
```

```
void update_velocity_and_position(const double stepsize, const int nstars
    for (int i = 0; i < nstars; i++) {
        velocity[i].x += stepsize * Fx[i] / mass[i];
        velocity[i].y += stepsize * Fy[i] / mass[i];
        position[i].x += stepsize * velocity[i].x;
        position[i].y += stepsize * velocity[i].y;
    }
}

// Taken from Lab6 Task 02!
static double get_wall_seconds() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
    return seconds;
  }
```