UPPSALA
UNIVERSITET

Assignment 3
High Performance Programming

Performad by:
Grupp 9
Parwand Batti
Jakob Gölén
Simon Persson

February 14, 2022

1 The Problem

The goal of this assignment was to simulate the evolution of a galaxy with a number of particles using information about their position, velocity and the force between particles. Due to an instability when particles are very close, the force on one particle denoted i from all the other particles in the system can be calculated using equation (1), assuming there are N, particles in the system.

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

G is the gravitational constant and is set to $\frac{100}{N}$. r_{ij} is the distance between particle i and particle j, and $\hat{r}_{ij}$ is the normalised direction. Epsilon is a small number which limits the maximum force between two particles and was chosen as 10^{-3}.

In order to find out how the particles move in time, a numerical method must be used. In this assignment, the sympletic euler is used. With a given initial value for the velocity and position of the particles in the system, the forces can be calculated using equation (1). Then, by calculating the acceleration and velocity, we then update the position of each particle at the next time step. In this assignment, the time step was set as $\Delta t = 10^{-5}$ and the positions for each particle could be updated using equation (2).

$$\begin{cases} \overline{a}_i^n = \frac{\overline{F}_i^n}{m_i} \\ \overline{v}_i^{n+1} = \overline{v}_i^n + \Delta t \cdot \overline{a}_i^n \\ \overline{x}_i^{n+1} = \overline{x}_i^n + \Delta t \cdot \overline{v}_i^{n+1} \end{cases} \tag{2}$$

The values of the variables and the equations for the force calculation and the step calculation was given in the assignment description. [1]

2 The Solution

All of the tests were performed on a `Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz` running on `Ubuntu version 20.04` under Windows Subsystems for Linux. The program was compiled with `gcc version 9.3.0`. First, the inputs were configured. The program was setup such that it took arguments for number of particles, starting configuration in an input file, number of steps, step size and if graphics should be activated or not. The position (x and y), velocity (x and y), mass and brightness was saved in arrays and the initial values were read from the input file. Arrays on the heap i.e. initialized by `malloc`, was faster than arrays kept on the stack.

Next, a for-loop was initialized over the number of steps. Inside that loop, another loop for all the particles was initialized. A third loop was created where all the forces was calculated. An if-statement ensured that the force was not calculated for one particle on itself, i.e. $i \neq j$. By taking an inspiration of [2], the positions and velocities could be updated (after calculating the forces) using a for-loop. When the number of steps has been reached and the for-loops has ended, a file called "result.gal" is created. The final position (x and y), the mass, the final velocity (x and y) and the final brightness is written in that order for every particle to `result.gal`. Finally, the memory used for the arrays are freed to ensure no memory leaks, which was verified by `valgrind`.

Graphics was implemented by including the provided `graphics.c` and `graphics.h` file in the `galsim.c` file. The functions that controls graphics were put in if-statements, which would only run if graphics was set to 1 as an input, otherwise they're were ignored. These if statements did not affect the runtime if graphics were set to 0.

3 Performance and discussion

Using only the `-O3` optimization flag and no manual optimisation of the code, the run time was 3.8402 wall seconds for $N = 3000$, 100 steps and $\Delta t = 10^{-5}$. After doing several optimizations techniques as described below, the run time was decreased to just 1.8323 wall seconds using the same input parameters.

There are some optimizations applied to the code that improved performance and reduced execution time. Other optimizations have been tested but they did not affect the performance and therefore they haven't been used in the code. Firstly, the summation in the formula for the force has been computed using for-loop instead of using if-statement to check if i is not equal j. To achieve this, the content of the for-loop needed to be implemented separately in a void function. To reduce the cost caused by calling a function, has the keyword `static inline` been used. Because the function had several pointers as input, the keyword `restrict` to avoid pointer aliasing.

Secondly, two large dynamic memory allocations were used for all input and output vectors instead of allocating all vectors separately. Having many small memory allocations can sometimes lead to performance problems and longer execution time. In addition, it was checked that the program has no memory leak using the command `valgrind`. Memory leak could adversely affect the performance.

Furthermore, several optimization flags have been used, which improved the performance significant. The optimization flags that have been used here are `-O3`, `-ffast-math`, `-march=native`, `-faggressive-loop-optimizations`. [3] Other flags have been tested but they gave no particular effect. Worth noting that the last two flags together with counters of the type `unsigned int` gave faster execution time. Additionally, those flags were very important in this case due to a lot of for-loops and calculations such as multiplications and additions. Many calculations at the same line has been avoided and the strength reduction technique has been applied. Declaring variables as global made performance worse, therefore the variables tried to be declared locally.

However, the optimization method loop unrolling was expected to has some impact on the performance, but unfortunately the performance got worse with loop unrolling. This may be due to the fact that the work done in the for-loops was a lot and the calculations were dependent. The calculations under the for-loops have been implemented in functions and in the for-loops during the main, these functions have been called. Then, loop unrolling was applied and the performance didn't get better.

To ensure that the program has no errors, the `gdb` and `valgrind` debugger have been used. To verify the complexity of the algorithm, the execution time was measured and plotted as function of N for 100 steps and $\Delta t = 10^{-5}$, see Figure 1. Measuring time was done using `get_wall_seconds()`, which means that the time is given in wall seconds. The timer was started just before the algorithm and stopped just after. In this way we know that the execution time is only for the algorithm and does not include other parts of the program, such as read/write files, allocate memory, declare variables, etc. Figure 1 shows that the measured time increase much faster for larger N, which was expected for an algorithm with complexity $\mathcal{O}(N^2)$.

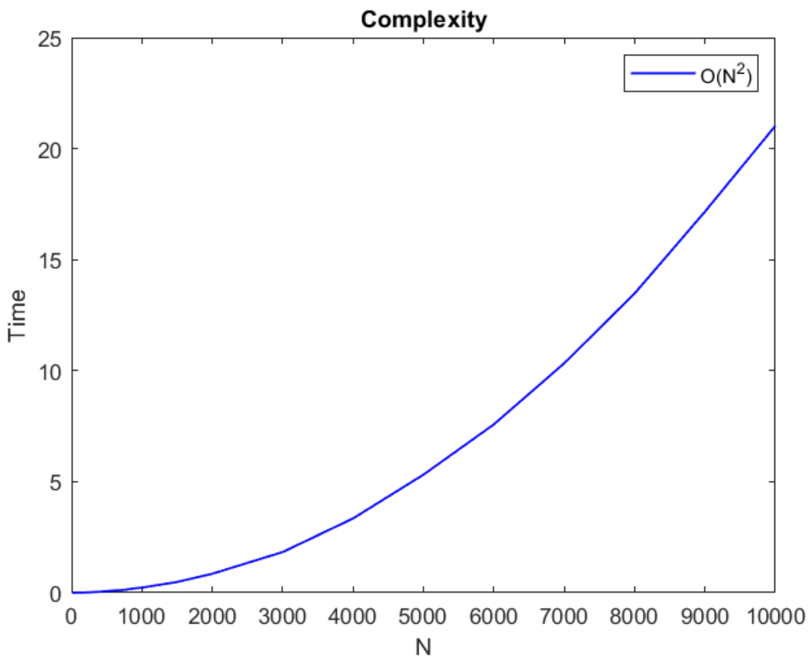

Figure 1: Verification of complexity $\mathcal{O}(N^2)$

References

[1] Assignment 3. The gravitational n-body problem, high performance programming, 2022. Uppsala University.

[2] Samuel S Cho. N-body problem, 2011. [Retrieved 2022-02-10] http://users.wfu.edu/choss/CUDA/docs/Lecture%2012.pdf.

[3] The GCC Team. Optimize options (using the gnu compiler collection (gcc)), 2022. [Retrieved 2022-02-11] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

A C code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
```

```c
#include <math.h>
#include "graphics.h"


// Defining the timer
static double get_wall_seconds() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
    return seconds;
}

// The function for the summation in the formula for the force
static inline void SumInForce(int i, int j, double EPS, double *
   restrict sum_x, double *restrict sum_y, double *restrict inputs){
    double r_x, r_y, r_vec, r_ij, num, temp1, temp2, temp3, temp4;
    r_x  = inputs[i*6 + 0] - inputs[j*6 + 0];
    r_y  = inputs[i*6 + 1] - inputs[j*6 + 1];

    // The distance between the particle
    r_vec = (r_x * r_x) + (r_y * r_y);
    r_ij  = sqrt(r_vec);

    // The summation
    temp2   =  r_ij + EPS;
    temp3   = temp2 *temp2;
    num     = temp2 * temp3;
    temp1   = 1.0/num;
    temp4   = temp1 * inputs[j*6 + 2];
    *sum_x += temp4 * r_x;
    *sum_y += temp4 * r_y;
}


int main(const int argc, char *argv[]){

    // Checking if the input arguments to the program are corret
    if (argc != 6) {
        printf("Give 5 input arguments: int N, filename.gal , int
   nsteps, double delta_t, graphics (0 or 1) .\n");
        exit(1);
    }

    // The input arguments
    char filename_in[70];
    strcpy(filename_in, argv[2]);
    const int N             =  atoi(argv[1]);
    const int nsteps        =  atoi(argv[3]);
    const double delta_t    =  atof(argv[4]);
    const int graphics      =  atoi(argv[5]);

    // Graphics settings
    const float circleRadius = 0.003, circleColor = 0;
    const int windowLength = 400;
    const int windowWidth = 400;

    // Creating a window and setting the axes if graphics routines are
   turned on
```

```c
    if (graphics == 1){
        InitializeGraphics(argv[0], windowWidth, windowLength);
        SetCAxes(0, 1);
    }

    // Alocatting the memory dynamically
    double *inputs, *outputs;
    inputs  = (double*)malloc(6*N*sizeof(double));
    outputs = (double*)malloc(6*N*sizeof(double));

    FILE *file_in, *file_out;
    // Opening the input file
    file_in = fopen(filename_in, "r");
    if (file_in == NULL) {
        printf("Error! Can't open the file.\n");
        exit(1);
    }
    // Reading the input file
    fread(inputs, sizeof(double), 6*N, file_in);

    // Closing the input file
    fclose(file_in);

    // Declaring variables
    const double EPS = 0.001;
    const double G = (double) (100.0/N);
    unsigned int i, j, n, k;
    double sum_x, sum_y, temp10, F_x, F_y, a_x, a_y, v1_x, v1_y, p2_x,
  p2_y;
    double start_time, time_taken;

    // Starting the timer
    start_time = get_wall_seconds();

    // Implementing the algorithm
    for(n = 0; n < nsteps; n++){
        for(i = 0; i < N; i++){
            // Initialize the sum variables
            sum_x = 0;
            sum_y = 0;

            // Summing as long as i is not equal j
            for(j = 0; j < i; j++){
                SumInForce(i, j, EPS, &sum_x, &sum_y, inputs);
            }
            for(j = i+1; j < N; j++){
                SumInForce(i, j, EPS, &sum_x, &sum_y, inputs);
            }
            // Adding the masses from input to output
            outputs[i*6 + 2] = inputs[i*6 + 2];

            // Adding the brightness from input to output
            outputs[i*6 + 5] = inputs[i*6 + 5];

            // Calculating the forces of particle i
            F_x = -G * inputs[i*6 + 2] * sum_x;
            F_y = -G * inputs[i*6 + 2] * sum_y;

            // The acceleration of particle i
            temp10 = 1.0/inputs[i*6 + 2];
            a_x = F_x * temp10;
            a_y = F_y * temp10;

            // The updated velocity of particle i at step n + 1
            v1_x = delta_t * a_x;
            v1_y = delta_t * a_y;
            outputs[i*6 + 3] = inputs[i*6 + 3] + v1_x;
            outputs[i*6 + 4] = inputs[i*6 + 4] + v1_y;

            // The updated position of particle i at step n + 1
            p2_x = delta_t * outputs[i*6 + 3];
            p2_y = delta_t * outputs[i*6 + 4];
            outputs[i*6 + 0] = inputs[i*6 + 0] + p2_x;
            outputs[i*6 + 1] = inputs[i*6 + 1] + p2_y;

            // Clearing the screen
            if (graphics == 1)
                ClearScreen();
        }
        // Updating the positions and velocities
        for(k = 0; k < N; k++){
            inputs[k*6 + 0] = outputs[k*6 + 0];
            inputs[k*6 + 1] = outputs[k*6 + 1];
            inputs[k*6 + 3] = outputs[k*6 + 3];
            inputs[k*6 + 4] = outputs[k*6 + 4];

            // Drawing the stars/particles
            if (graphics == 1)
                DrawCircle((float)(inputs[k*6 + 0]), (float)(inputs[k*6
    + 1]), 1, 1, circleRadius, circleColor);
        }

        // Refreshing and sleeping
        if (graphics == 1){
            Refresh();
            usleep(1500);
        }
    }
    // Flushing and closing the display
    if (graphics == 1){
        FlushDisplay();
        CloseDisplay();
    }

    // Stopping the timer and measuring the execution time
    time_taken = get_wall_seconds() - start_time;

    // Printing the execution time
    printf("Time taken = %.4f\n", time_taken);

    // Creating new file for the outputs
    file_out = fopen("result.gal", "w");

    // Writing the data to the output file
    fwrite(outputs, sizeof(double), 6*N, file_out);

    // Closing the output file
    fclose(file_out);

    // Freeing the allocated memory
    free(inputs);
    free(outputs);

    return 0;

}
```
Listing 1: C code