UPPSALA UNIVERSITET

High Performance Programming

Assignment 3

Ekeroot, Lovisa
Lovisa.Ekeroot.0464@student.uu.se
Millberg, Johan
johan.millberg.0226@student.uu.se
Tennberg, Moa
Moa.Tennberg.2243@student.uu.se

Uppsala February 17, 2022

Contents

1 Introduction

1.1 The Problem

1.1.1 Description

The goal of Assignment 3 is to create the program "galsim", that should calculate the motion of an initial set of N particles that approximates the evolution of a galaxy, in two dimensionless spatial dimensions. Each particle in the simulation have the following properties: x- and y-position, x- and y-velocity, mass, and brightness. These properties should be taken both as input and output to the program, before and after calculations.

When starting the program, the user should tell the program which input file, number of time steps, number of particles, the size of each time step and if graphics should be on (1) or off (0), and this information should be told in this specific order. If the user do not input exactly five input arguments, then a message about the expected input arguments should be printed to the user and then the program should stop. When the simulation and calculations have been executed, the resulting data with the same particle properties as in the input data should be written to a result file. Note that the mass and brightness will be the same in both input and output data files.

Another part of the goal is to write the program as efficiently as possible. The section "Performance and Discussion" is thereby included for describing what have been done to make the program as efficient as possible.

1.1.2 Equations

To calculate the motions of the particles, Newton's law of gravitation in two dimensions is provided that gives the force exerted on particle i by particle j. For this Assignment, the so-called Plummer spheres which is a modified equation from Newton's law of gravitation is used, see Equation 1. This helps when the distance of two particles is very small (r«1) and then sets a maximum force between two particles. By using this equation the simulation gets smoother as well, and the force will not be undefined if $i = j$. This will be important when performing the calculations.

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

In Equation 1, G is the gravitational constant, m is the mass of particle i and j correspondingly, N is the total amount of particles, r_{ij} is the Euclidian distance between particle i and j, $\epsilon_o = 10^{-3}$ and $\mathbf{r}_{ij}$ is the distance vector between particle i and j. The following equations, Equation 2 - 4, presents how the distances are calculated, where $\mathbf{e}_v$ is the unit vector in direction v.

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

$$r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 \tag{3}$$

$$\hat{\mathbf{r}}_{ij} = \frac{\mathbf{r}_{ij}}{r_{ij}} \tag{4}$$

When updating the properties of the particles, one applies Equation 5 - 7. In these equations, $\mathbf{a}$ is the acceleration vector, $\mathbf{u}$ is the velocity vector, Δt is the time step and $\mathbf{x}_i$ is the vector of positions, all related to particle i. Note that all these equations and corresponding properties are including both spacial directions.

$$\mathbf{a}_i^n = \frac{\mathbf{F}_i^n}{m_i} \tag{5}$$

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n \tag{6}$$

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1} \tag{7}$$

This method is called the *Symplectic Euler* method, which preserves global properties of the gravitational system, and it is a straightforward method. It is also known that the method has computational complexity $\mathcal{O}(N^2)$, which makes it computationally expensive for large N.

In this Assignment, use $\Delta t = 10^{-5}$ and $G = 100/N$.

1.2 Procedure

To solve this assignment, the students first writes a program that works, without taking optimization into account. When this program is finished, the students tries to optimize the code such that it gives the best performance with respect to time measurement and providing a correct result. For the time measurements, the graphics used to visualize the solutions are not activated.

During this report, only the time measurements with respect to the final optimized code is going to be presented. The time measurements from the unoptimized code will therefore not be presented. The final code is provided in the Appendix.

2 Solution

The final optimized source code (galsim.c) and Makefile is provided in the Appendix.

In the Makefile, the -O3 and -march=native optimization option are being used. The -O3 optimization option may increase the executable size, and the -march=native option allows the compiler to automatically select the architecture to match your system. [1]

The code is structured by first implementing the necessary functions, followed by the main function. There are two functions created, one with the purpose to read and store initial data given by the input file, and one that enables the measuring of the execution time in wall seconds using the *text.h*. The function that reads and stores the initial data is called *set_ initial_ data*, and is inspired by the function *read_ doubles_from_file* in the given program *compare_ gal_files.c*. The function that measures the execution time in wall seconds is called *get_ timings*, and is inspired by the function *get_ wall_ seconds* from task 4 in Lab 5.

In the beginning of the main function, the program checks that the user has given the correct number of different input parameters. If the user gives invalid input arguments, the program will print an error message of an example input and then exit the program. This prevents the program from causing a segmentation fault. The user can then restart the program with correct input arguments to run the program.

In the beginning of main, when valid input arguments are given, the timer begins by using the function *get_ timings*. Then the program stores the given input arguments simultaneously as it declares multiple variables that are going to be used in the execution of the program. The program also allocates memory for the attributes of the particles and initializes an array containing doubles of size $2N$, where N is the amount of particles. The purpose of this array is to store the sum of forces in the x and y direction acting on each particle. Initially, this array was dynamically allocated in the same way as the array used for storing all attributes of the particles. This was later changed to a static array in order to decrease the amount of function calls to *malloc()* and *free()* by half. This increased the performance of the program.

The returned value of *set_ initial_ value()* is either 1 or 0, depending on if the data was successfully read from the file. The integer *successful* is set to the returned value of the function. If the value of *successful* is 0, the program terminates with an error message. This can happen if the program is not able to access the given file, or if the size of the file is unexpected. This prevents the program from causing a segmentation fault, which would provide the user with no information on why the program failed.

Once the initial data has been successfully read, the for-loop that loops once per time step begins. If the user has enabled graphics, all particles are drawn in the graphics window at the appropriate coordinates. The size of the circles that represent the particles are scaled based on the mass of each particle. After this, the summed forces in the *forces* array are set to 0 in another for-loop. This loop can utilize cache blocking as a performance enhancing technique. This is implemented by first checking if the amount of particles is divisible with the given block size. The block size in this case is set to 50, meaning that all input files with 100 or more particles will allow this technique to be used. If the input file contains less than 100 particles (with the exception of the file with 50 particles), the program will not utilize cache blocking and will instead iterate through all values in the array, setting each value to 0.

Once the sum of the forces acting on each particle has been reset, the program enters another for-loop that starts at 0 and ends at N. The iteration variable of this loop is i. The purpose of this loop is to iterate through all particles and calculate the sum of the forces acting on the particles in the x and y direction. The x and y coordinates as well as the mass of the particle are assigned to new variables, which decreases the amount of times the program has to find these values in the particles array. These values will be used in the innermost loop, and they are loop invariants to that loop. Assigning the values to variables outside the innermost loop increased performance.

After the loop invariants had been assigned to new variables, the program enters the innermost loop. This loop iterates from i to N, and the iterative variable is called j. In this loop, the bulk of the calculations are carried out. The calculations are done according to the Equations given in Section 1.1.2, but with a twist by applying Newtons Third Law that the force between two particles have the same magnitudes but opposite directions. This is applied in the code at line 141 to 144, and decreases the amount of calculations by a factor two. This decrease in iterations is due to the fact that summing both the force acting on particle i and particle j at the same time enables the innermost loop to go from i to N, instead of from 0 to N.

Once the program exits the innermost loop, the velocities in the x and y direction and the x any y coordinates of the particle i is updated directly in the particles array. This is done according to the Equations given in Section 1.1.2. After this, one iteration of the i loop is completed, meaning that the attributes of the particle i has been updated. This will be done for all particles, and then one iteration of the time loop is complete. This is then repeated for the amount of steps specified by the user.

When the program has finished the last iterations, it saves the final properties of all the particles in the file "result.gal", with the same format of the properties as in the given initial data file. The last thing that the program does is to free all the memory that has been allocated to the particles, and presents the time the execution took in wall seconds before exiting the run of the program.

3 Result

3.1 Time measurements

The final code is used for performing some time measurements in order to evaluate how the time depends on the amount of particles.

In Table 1, the amount of particles is presented together with the time it took for the calculations to be performed. The data is acquired by using the time step $\Delta t = 10^{-5}$ and 100 time steps in total, and the input file "ellipse_N_$\hat{N}$.gal", where $\hat{N}$ is the value of $\hat{N}$ provided in the Table, and the value of N provided in the Table as another input argument. The data in the Table is acquired using a 2.3 GHz Dual-Core Intel Core i5 CPU, with the compiler GCC version 11.2.0.

N	$\hat{N}$	wall seconds
10	00010	0.001
30	00030	0.002
50	00050	0.003
70	00070	0.003
90	00090	0.005
150	00150	0.009
300	00300	0.030
500	00500	0.073
700	00700	0.141
900	00900	0.215
1500	01500	0.625
2000	02000	1.107
3000	03000	2.497
5000	05000	6.441
7000	07000	12.728
9000	09000	20.951

Table 1: Investigation of the time dependency with respect to the number of particles, using a time step of 100

Plotting the data in Table 1 with the time as a function of N, results in Figure 1.

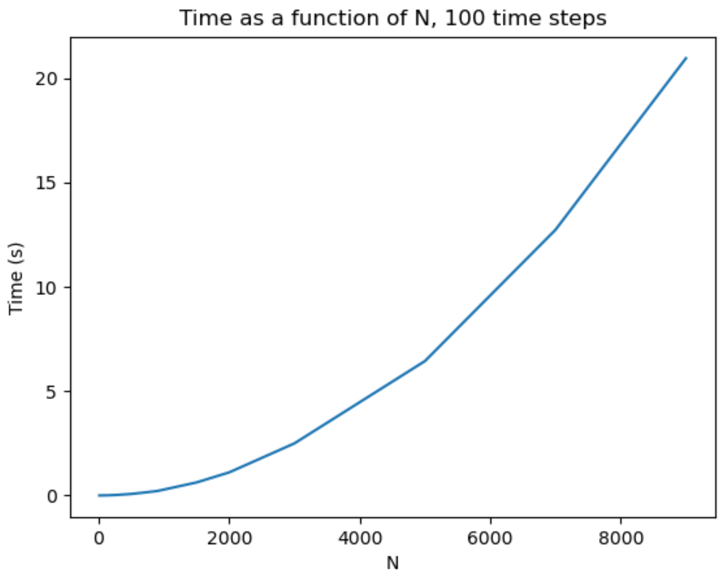

Figure 1: Resulting image with how time is dependent on the value of N with 100 time steps.

3.2 Graphics

When investigating the solution using graphics, several input files where investigated. The results from these are presented in Figure 2, Figure 3 and Figure 4. For all the executions of these programs, the time step $\Delta t = 10^{-5}$ are used.

Figure 2 presents the visual result from the input file "sun_and_planet_N_2.gal" with 100 time steps and N = 2. Figure 3 presents the visual result from the input file "ellipse_N_03000.gal" with 100 time steps and N = 3000. Figure 4 presents the visual result from the input file "ellipse_N_03000.gal" with 200 time steps and N = 3000.

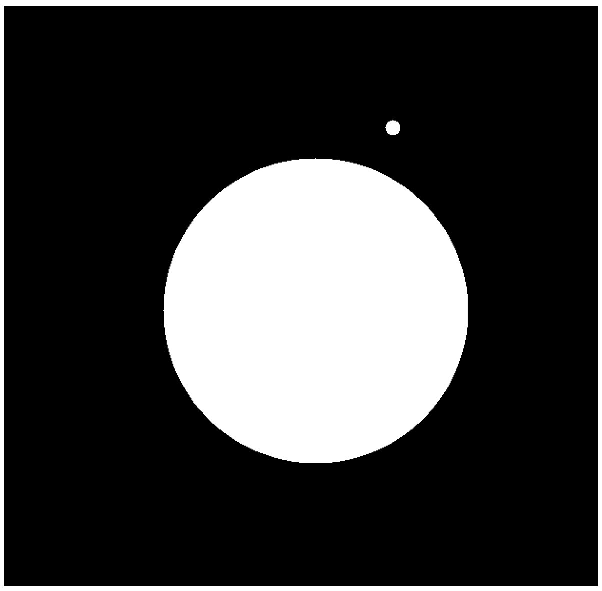

Figure 2: Resulting image from graphics for the input file "sun_and_planet_N_2.gal"

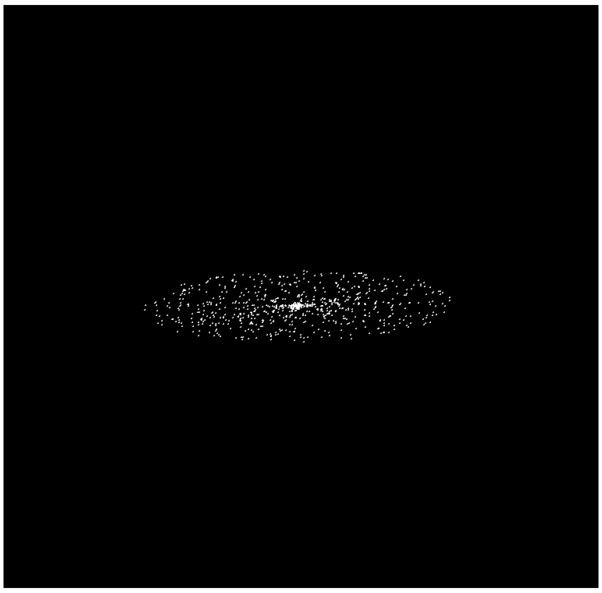
Figure 3: Resulting image from graphics for the input file "ellipse_N_03000.gal", with 100 time steps.

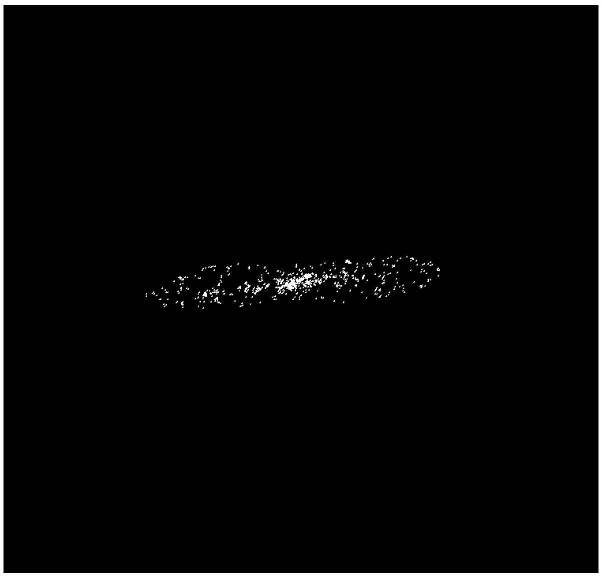
Figure 4: Resulting image from graphics for the input file "ellipse_N_03000.gal", with 200 time steps.

4 Performance and Discussion

4.1 General

To optimize the original code various methods were implemented. The following sections will give a description of the methods used and the performance improvements that were yielded.

4.2 Reducing Instructions

In the unoptimized code the calculations of the force and distance between particles was called 900000000 times, since both the loops inside the time-loop started at 0 and ended at N. Therefore, the distance between particles was calculated twice, it was not utilized that the distance between particles remain the same in each time step. Thus, by introducing a new array containing the forces between each particle in the x and y direction, the j-loop was optimized such that the number of iterations decreased to 450150000 times, which decreased the number of iterations by a factor of two and improved the performance of the program substantially.

Furthermore, in the unoptimized code, structs were used to store the values of each particle. Thus, when assigning the initial values to the structs a for-loop was utilized to transfer the data from the buffer to the structs. An additional for-loop was also necessary to store the final values in a binary file. To avoid the for-loops an array of size 6*N was used instead. However, this change did not affect the performance to a large extent.

Another optimization technique that was utilized was strength reduction, where cheaper arithmetic operations can be used to improve performance. To take the exponent of a value the function *pow()* can be used, however, since this function is computationally expensive it was not used in the unoptimized nor optimized code. Instead, the multiplication operator was used. Furthermore, there are two functions for determining the square root of a value *sqrt()* and *sqrtf()*, where *sqrtf()* is less precise and in theory faster. However, when implemented, *sqrtf()* did not improve the performance, and since the precision of the result was important *sqrt()* was used.

Another factor in for-loops that can decrease performance are loop-invariants, which are variables that do not change depending on the loop-variable. By placing the variables depending on the ith particle outside of the inner loop when calculating the distance and force between each particle loop invariants can be avoided. By removing the loop invariants the performance of the code increased slightly.

The *inline* keyword can be used to optimize a function call such that it is placed inside the loop, since function calls can cause larger computational costs if called a large number of times. The *inline* keyword was not used in the optimized version, however, inlining was used. In the unoptimized version the force calculations were performed in a separate function, however, to improve performance this function was directly placed in the inner most loop to avoid a large number of function calls.

4.3 Memory Usage

Initially the two arrays used for storing the data for all particles and their respective forces were initialized using *malloc()* and then the memory was freed at the the end of the code using the *free()* function to avoid memory leaks. Moreover, using *malloc()* can be computationally expensive since it strains the systems memory management system [2]. To improve performance the array used for the forces was initialized using a static array instead of *malloc()*, such that the number of calls to *malloc()* decreased by a factor of two. This optimization improved the performance marginally. However, allocating the array containing the data for the particles to a static array did not improve the performance.

For all constants the *const* keyword was used. This keyword was used in both the optimized and unoptimized code. Although when removing the keyword from all constant variables the performance decreases by approximately 0.02 seconds.

The use of the computers cache memory was further optimized by introducing blocking. Blocking was attempted on all loops in the main code, in an attempt to increase performance. However, the performance decreased marginally when blocking was implemented on the ith and jth loop. A performance increase was observed when blocking was implemented on the loop used for allocating zeroes to each element in the array used for calculating the forces. Blocking yielded a performance increases of on average 0.05 seconds.

4.4 Instruction-level Parallelism

To implement instruction-level Parallelism (ILP), one can simultaneously perform several instructions. This can be good for optimization in the case that the instructions are not dependent on each other. [3]

One ILP method is the so called *loop unrolling.* This method performs several loop iterations, but only in one iteration. The iterations are then done in groups with the size of the *unroll factor*[3]. This method was attempted, however, since a large number of calculations are performed in each iteration of the j-loop the code appeared unstructured, in addition when implemented the performance neither increased nor decreased. Thus, the method of *loop unrolling* was not used in the final version of the code.

Another ILP method is the so called *loop fusion*, where one tries to merge together several loops into one if the iteration statements are the same. [3] This is applied in the final code when iterating over the particles when for example storing the coordinates and the mass of a particle. The iteration statements are the same, but several instructions are given in the beginning of each iteration, instead of having multiple loops to do the exact same procedure. *Loop fusion* is also implemented when calculating the force between two particles, where multiple calculations are being executed.

By implementing Newton's Third Law (line 141 to 144) and figuring out a correct iteration statement, one reduces the amount of times that the forces needs to be calculated by a factor two. In these lines, *loop fusion* is also applied since if Newton's Third Law was not applied, multiple loops would be necessary to have to execute the same procedure. But in the final code, the storing of the calculated forces and executed with respect to both the index of the outer and inner loop, which improved the performance by approximately three to four seconds, which is previously stated as well.

A third thing mentioned about ILP, is to avoid branches (if-statements inside inner loops) in the code. [3] This has been applied to the code by implementing the program such that no branches are necessary to have for calculating the properties of the particles. The code only uses for-loops instead to go over each element, and by implementing the inner for loop to iterate from particle i to particle N, the program does not have to execute any if-statements for comparison of which particle it should take into account. The performance was not affected by having a if-statement or not in the inner most loop, but the code was easier to implement by not having it so therefore it is not implemented.

The if-statements that are included in the final code are with respect to using graphics, check so the input arguments are given correctly, the data can be read correctly and checking if there is a large amount of particles that should be included in the calculations. The if-statements related to the graphics does not affect the time measurements, since they are executed without using graphics. The if-statements related to the input arguments and valid reading data from the input file is necessary to have to not get "Segmentation Error" in the running of the code if these are invalid. And the if-statement related to using cache blocking still optimizes the code since for large values of N, cache blocking is faster than iterating over all particles.

Since the code lacks branches in the inner loops, *branch prediction* is not applied in the final program. There is no need of predicting the probability that the code should enter a branch, if there are no branches. But if there should have been branches implemented, it is a good idea to investigate if *branch prediction* optimizes the code by helping the compiler to choose the correct branch with respect to the probability that the specific branch should be entered.

Another ILP method that has been investigated is *auto-vectorization*, where computational loops are vectorized in arithmetic pipelines. [3] This is applied and used in the code by having the optimization flag "-O3" in the Makefile, which includes the the option "-ftree-vectorize". This improves the performance substantially compared to not using any optimization options, and therefore the "-O3" option is included.

4.5 Time dependency

By observing Figure 1 which is presented previous in the report, where the time is plotted with respect to the number of particles N, one can conclude that the graph follows the appearance of a squared function multiplied by some factor. Since the implemented program uses the Equations provided in section 1.1.2, which has the computational complexity $\mathcal{O}(N^2)$, it is not surprising that the execution times also follows N^2 behaviour.

From the observation of Figure 1 and knowledge about the implemented code, the students confirms the expected $\mathcal{O}(N^2)$ complexity.

5 References

As been presented in section 2, the two implemented functions in the beginning of the code that has been implemented from given functions in the course "High Performance Programming". These functions and their inspiration sources are provided in Table 2.

Function	Inspiration from
"set_initial_data()"	"compare_gal_files.c" in Assignment 3
"get_timings()"	"get_wall_seconds()" from Task4 in Lab05

Table 2: Table of implemented functions and their respective inspiration source.

[1] High Performace Programming. “Lab 5: Serial Optimization Part 1, Reducing instructions”. In: (Feb. 2022).

[2] High Performace Programming. “Lab 6: Serial Optimization Part 2, Memory Usage”. In: (Feb. 2022).

[3] High Performace Programming. “Lab 7: Serial Optimization Part 3, Instruction-Level Parallelism (ILP)”. In: (Feb. 2022).

Appendix

5.1 Makefile

```
CFLAGS=-Wall -O3 -march=native
INCLUDES=-I/opt/X11/include
LDFLAGS=-L/opt/X11/lib -lX11 -lm

galsim: galsim.o graphics.o
  gcc -o galsim galsim.o graphics.o $(LDFLAGS)

galsim.o: galsim.c graphics.h
  gcc $(CFLAGS) $(INCLUDES) -c galsim.c

graphics.o: graphics.c graphics.h
  gcc $(CFLAGS) $(INCLUDES) -c graphics.c

clean:
  rm -f ./galsim *.o
```

5.2 Optimized version of galsim.c

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


/*
Method of reading data from .gal files was inspired by the function
    read_doubles_from_file
in the given compare_gal_files.c
*/
int set_initial_data(int N, double** particles, const char* filename)
    {
    FILE* file = fopen(filename, "rb");
    if (!file) return 0;
    fseek(file, 0L, SEEK_END);
    size_t file_size = ftell(file);
    if (file_size != 6*N*sizeof(double)) return 0;
    fseek(file, 0L, SEEK_SET);

    fread(*particles, sizeof(char), file_size, file);

    fclose(file);
    return 1;
}

/*
get_timings() was inspired by the function get_wall_seconds() from
    Task 4 in Lab 5
*/
double get_timings() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    double sec = tv.tv_sec + (double)tv.tv_usec / 1000000;
    return sec;
}

int main(int argc, char *argv[]) {
    if (argc < 6) {
        printf("Use following syntax to run program: ./galsim N
   file_name amount_steps step_length graphics_on \n");
        return 0;
    }
    double time = get_timings();

    // Set input parameters
    const int N = atoi(argv[1]);
    const char* filename = argv[2];
    const int nsteps = atoi(argv[3]);
    const double delta_t = atof(argv[4]);
    const int graphics = atoi(argv[5]);
    const int windowWidth=800;
    int successful;

    // Declare variables
    double F_x, F_y, F_const;
    double r_x, r_y, r;
    double x, y, mass;
    double denom;

    // Declare constants
    const double G = (double) -100.0 / N;
    const double eps_0 = 0.001;

    // Store the properties of all particles in the array particles
    double *particles = (double*) malloc(N*sizeof(double)*6);

    // The sum of the forces in the x and y direction for each
   particle is stored in forces
    double forces[2*N];

    successful = set_initial_data(N, &particles, filename);

    if (!successful) {
        printf("Error reading initial data file. \n");
        return 0;
    }

    if (graphics != 0) {
        InitializeGraphics(argv[0],windowWidth,windowWidth);
        SetCAxes(0,1);
    }
    // Declaring iteration variables
    unsigned int t;
    unsigned int l;
    unsigned int i;
    unsigned int j;
    unsigned int bl;

    // Amount of blocks and blocksize
    const int blocksize = 50;
    const int nBlocks = (2*N)/blocksize;
    int l_start;
    for (t = 0; t < nsteps; t++) {
        /*
            Draws all particles if graphics are enabled
        */
        if (graphics != 0) {
            ClearScreen();
            for (l = 0; l < N; l++) {
                DrawCircle(particles[l*6+0], particles[l*6+1], 1, 1, particles[l*6+2]*0.002, 0);
            }
            Refresh();
            usleep(2000);
        }

        //Utilizes cache blocking if 2*N is divisible by the block size
        if ((2*N) % blocksize == 0) {
            for (bl = 0; bl < nBlocks; bl ++) {
                l_start = bl*blocksize;
                for (l = l_start; l < (l_start + blocksize); l++) {
                    forces[l] = 0;
                }
            }
        }
        else {
            for (l = 0; l < 2*N; l++) {
                forces[l] = 0;
            }
        }

        for (i = 0; i < N; i++) {
            x = particles[i*6];
            y = particles[i*6 + 1];
            mass = particles[i*6 + 2];

            /*
                Calculates all forces acting on particle i
            */
            for (j = i; j < N; j++) {
                r_x = x - particles[j*6 + 0];
                r_y = y - particles[j*6 + 1];
                r = sqrt(r_x*r_x + r_y*r_y);

                denom = (r + eps_0)*(r + eps_0)*(r + eps_0);
                F_const = G * mass * (particles[j*6 + 2]/denom);
                F_x = F_const * r_x;
                F_y = F_const * r_y;

                /*
                    Utilize the fact that the forces between the two particles are equal
                    but acting in the opposite direction, minimizing
```

```
the needed iterations
                */
                forces[i*2 + 0] += F_x;
                forces[i*2 + 1] += F_y;
                forces[j*2 + 0] += -F_x;
                forces[j*2 + 1] += -F_y;

            }
            // Update the properties of the particle i
            particles[i*6 + 3] = particles[i*6 + 3] + delta_t*(forces[
i*2+0]/particles[i*6 + 2]);
            particles[i*6 + 4] = particles[i*6 + 4] + delta_t*(forces[
i*2+1]/particles[i*6 + 2]);

            particles[i*6 + 0] = particles[i*6 + 0] + delta_t*
particles[i*6 + 3];
            particles[i*6 + 1] = particles[i*6 + 1] + delta_t*
particles[i*6 + 4];

        }
    }

    if (graphics != 0) {
        FlushDisplay();
        CloseDisplay();
    }

    FILE *ptr;

    ptr = fopen("results.gal", "wb");
    fwrite(particles, N*sizeof(double)*6, 1, ptr); // Write all data
to binary file
    fclose(ptr);

    free(particles);
    printf("Galsim program took %7.3f wall seconds.\n", get_timings()
- time);
    return 0;

}
```