ASSIGNMENT 3: THE GRAVITATIONAL N-BODY PROBLEM

Stefanos Tsampanakis
stefanos.tsabanakis@gmail.com

Sajad Sharhani
sharhai.sajad@gmail.com

Atefeh Aramian
.atefeh.aramian.0970@gstudent.uu.se

February 17, 2023

1 Introduction

The gravitational N-body problem is a classical problem that involves the mutual gravitational interactions between a set of objects. This problem can be modeled and solved computationally using numerical methods such as the N-body simulation. In this work, we present a C program that solves the N-body problem by utilizing optimization methods The optimization methods used in this program have been carefully chosen to balance computational efficiency and accuracy in the solution. The program has been tested with various N-body problems and the results have shown that the optimization methods used in the program have significantly improved the accuracy and computational performance compared to traditional N-body simulation techniques.

The main C program performs N-body simulations of star systems. It uses a basic brute-force algorithm to calculate the gravitational forces between the stars and was used as our base source code in which all various optimizations where applied.

The program has the following data structures:

1. A struct *star* to store the properties of each star. The structure has the following members:
   - x: the x-coordinate of the star
   - y: the y-coordinate of the star
   - m: the mass of the star
   - vx: the x-component of the velocity of the star
   - vy: the y-component of the velocity of the star
   - brightness: which is not used but kept to keep the file format
2. An array of struct *star* to store the properties of all stars in the simulation.

The structure of the code is as follows:

1. The program reads the initial star data from the binary file specified by the fileName input.
2. The program runs the simulation for `nsteps` steps by using a brute-force algorithm to calculate the gravitational forces between the stars.
3. At each time step, the program first updates the velocity of each star based on the net gravitational force acting on it.
4. Then, the program updates the position of each star based on its velocity.
5. The simulation results are saved in a binary file named `"result.gal"`.

```
    for(int n=0;n<nsteps;n++)// loop through all the time stepts
    {

        for(int i=0;i<N;i++)
        {

            ax=0;
            ay=0;
            for(int j=0;j<N;j++)
            {
                if(j!=i){
                    rx=stars[i].x-stars[j].x;
                    ry=stars[i].y-stars[j].y;
                    r=sqrt(rx*rx+ry*ry)+epsilon;
                    ax+=(-G*stars[j].m*rx)/(r*r*r);
                    ay+=(-G*stars[j].m*ry)/(r*r*r);
                }
            }

            stars[i].vx+=delta_t*ax;
            stars[i].vy+=delta_t*ay;
        }
        for(int i=0;i<N;i++)
        {
            stars[i].x+=delta_t*stars[i].vx;
            stars[i].y+=delta_t*stars[i].vy;
        }

    }
```

Listing 1: Implementation of the computations for the unoptimized version

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

The simulation is implemented using a simple brute-force algorithm and can be seen in Listing 1. For each time step, the program calculates the net gravitational force acting on each star by summing up the gravitational forces due to all other stars. The net gravitational force is then used to update the velocity of the star, and the velocity is then used to update the position of the star. This process is repeated for all `nsteps` time steps.

The codes are tested on 3 different processors, two of which are personal laptops and the third one of the Uppsala University's linux servers.The main tool used to assess the codes was Valgrind for checking memory leaks and cache and other memory related issues. Finally our codes where compared to each other by compiling them in 2 different ways. The first way being the -O3 optimisation flag that can be passed to the GCC compiler and the second way –Ofast -march=native -ffast-math -fopt-info-vec we will get in detail of those flags in the next section.

For the timings we used used the function seen in Listing [**greenwade93**]. which was taken from the lab sessions of this cource and the time measured was only for the time the code needed to compute and update the variables of motion of every particle, for every timestep.

We got the following results

For 3000 stars at 200 timesteps	
Proccessor	time in seconds
Intel(R)i7-1165G7 @ 2.80GHz	19.88098
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	24.86773
Intel(R)Xeon(R) E5520 @ 2.27GHz	73.59665

After testing with several .gal from N = 10 up to N = 10000 we plotted the corresponding results to verify that the complexity is of $\mathcal{O}(N^2)$ complexity. The figure can be viewed at Figure1 and we can see that it satisfies the expected behaviour.

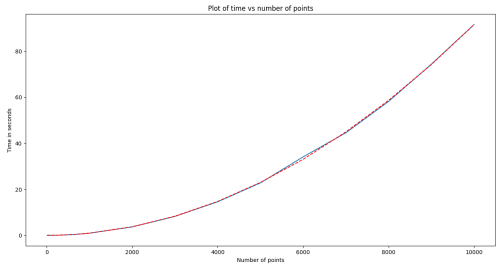

Figure 1: Exaction time Vs different N .

2 Optimazations

2.1 Compiler optimizations

GCC, the Gnu Compiler Collection, supports a wide range of optimizations when compiling our code. These optimizations can be specified using compiler flags.

Some of the most common optimization flags for GCC are:

- -O1: This flag enables basic optimization. It performs optimizations that are relatively simple and have a small impact on the compiled code size.
- -O2: This flag enables additional optimization. It performs more aggressive optimizations, including those that can have a moderate impact on the compiled code size.
- -O3: This flag enables aggressive optimization. It performs the most aggressive optimizations available, including those that can have a large impact on the compiled code size.
- -Ofast: This flag enables all optimization except those that may slow down the code or change its behavior in a way that is not compatible with the standard.
- -march=native: This flag optimizes the code for the host architecture, taking into account the specific instruction set and other features of the host.
- -ffast-math: This flag allows the compiler to generate code that makes use of faster but less accurate mathematical operations. The flag enables optimizations that are not compliant with the IEEE 754 standard for floating-point arithmetic, which is a widely used standard for representing and manipulating floating-point numbers.
- -fopt-info-vec: This flag provides information about vectorization in the optimized code. Vectorization is a technique used in compiler optimization where multiple data elements are processed simultaneously using vector instructions, rather than sequentially. This can greatly improve the performance of the code, especially when dealing with large arrays of data.

2.2 Algorithm optimizations

The code and its subsequent optimaztions were developed and debugged in the laptop device with the core i7 11 gen CPU.

After testing our unoptimized code we got the following results:

For 3000 stars at 200 timesteps		
Proccessor	-O3 flag(sec)	-Ofast and vect flags(sec)
Intel(R)i7-1165G7 @ 2.80GHz	8.32521	8.24813
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	6.31115	6.13101
Intel(R)Xeon(R) E5520 @ 2.27GHz	40.54420	40.54247

For 5000 stars at 200 timesteps		
Proccessor	-O3 flag(sec)	-Ofast and vect flags(sec)
Intel(R)i7-1165G7 @ 2.80GHz	22.68756	22.93767
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	17.51295	17.23855
Intel(R)Xeon(R) E5520 @ 2.27GHz	112.62272	112.62111

For 10000 stars at 200 timesteps		
Proccessor	-O3 flag(sec)	-Ofast and vect flags(sec)
Intel(R)i7-1165G7 @ 2.80GHz	90.78443	92.80147
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	68.85076	68.86672
Intel(R)Xeon(R) E5520 @ 2.27GHz	450.55941	450.59134

Note: For cases where N was less than 2000, our data was small enough to fit into the cache and thus, we decided to exclude them from our optimization comparison as it would not provide an accurate representation of the results.

The first step of optimizing our code involved a change in the way the computations were performed and finding a way to remove the if statement inside the inner loop. To achieve this, we broke down the loop into two separate parts, thereby skipping the if statement. In addition, we made some modifications to the calculations by introducing a denominator term and using the register command to initialize the ax and ay double variables 2. This was done since these variables were being used constantly and needed to be optimized for better performance. the results can be seen in the following tables

```
    for(int n=0;n<nsteps;n++)// loop through all the time stepts
    {

        for(int i=0;i<N;i++)
        {

            register double ax=0;
            register double ay=0;
            double denominator;
            for(int j=0;j<i;j++)
            {



                    rx=stars[i].x-stars[j].x;
                    ry=stars[i].y-stars[j].y;
                    r=sqrt((rx*rx)+(ry*ry))+1e-3;
                    denominator=1.0/(r*r*r);
                    ax+=(-G*stars[j].m*rx)*denominator;
                    ay+=(-G*stars[j].m*ry)*denominator;


            }
            for(int j=i+1;j<N;j++){

                    rx=stars[i].x-stars[j].x;
                    ry=stars[i].y-stars[j].y;
                    r=sqrt((rx*rx)+(ry*ry))+1e-3;
                    denominator=1.0/(r*r*r);
                    ax+=(-G*stars[j].m*rx)*denominator;
                    ay+=(-G*stars[j].m*ry)*denominator;

            }

            stars[i].vx+=delta_t*ax;
            stars[i].vy+=delta_t*ay;
        }
        for(int i=0;i<N;i++)
        {
            stars[i].x+=delta_t*stars[i].vx;
            stars[i].y+=delta_t*stars[i].vy;
        }

    }
```

Listing 2: Computational optimazations

For 3000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.29	x1.39
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.96	x1.11
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.99	x0.98

For 5000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.20	x1.38
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.98	x1.03
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.999	x0.98

For 10000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.30	x1.43
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.93	x1.08
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.99	x0.98

Another optimazation we considered was to take into account Newtons third law which states that for every action there is an opposite but equal reaction which in our case means F_{ij}=-F_{ji}. Using this method as seen in Listing3 the force between every pair of bodies needs to be calculated only once. The efficacy of this implementation had the opposite effects that we hoped. Our time increased and its likely caused in the increased overhead for accessing the memory and the extra loops that were needed to update the values.

For 3000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x0.63	x0.63
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.31	x0.36
Intel(R)Xeon(R) E5520 @ 2.27GHz	x1.2	x1.2

For 5000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x0.5	x0.54
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.30	x0.30
Intel(R)Xeon(R) E5520 @ 2.27GHz	x1.09	x1.12

For 10000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x0.40	x0.40
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.22	x0.21
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.87	x0.87

The Uppsala's Linux machines seemed to perform better on this code and we don't have an accurate explanation for this result.

In the next step, we focused on optimizing the cache and reducing branch misprediction. We used Valgrind with the flags –tool=cachegrind –branch-sim=yes to gain a deeper understanding of the performance of our algorithm. The removal of the if statement our branch misprediction rate was close to 0 %, but we noticed a high number of cache misses, with a total miss ratio of 15% in our data accesses.

We identified that the issue was with the arrangement of our data, leading to suboptimal spatial locality. The current data structure contained information about the brightness of the stars, which was never utilized in our computations and occupied valuable space in the cache. Our inner loop only required information about the position (x and y) and mass (m) of the stars. To resolve this issue, we created a new array of structures that contained only the necessary data Listing 4 and the code can be viewed in Listing 5.

For 3000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.30	x1.41
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.86	x1.09
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.96	x0.98

For 5000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.27	x1.40
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.86	x1.08
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.96	x0.97

For 10000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.25	x1.41
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.85	x1.09
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.96	x0.97

The changes proved to help a little and we reduced the cache miss ratio to 12.5% but the performance of our algorithm remained the same.

For the final step was to improve the code through vectorization. To achieve this, we made modifications to the new data structure we created in the previous step, by replacing individual values with arrays that corresponded to those values. This change was mainly aimed at enhancing the performance of the -fopt-info-vec flag. The implementation can be viewed in Listing 6.

For 3000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.26	x2.80
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.87	x2.17
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.96	x1.93

For 5000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.26	x2.8
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.87	x2.17
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.96	x1.93

For 10000 stars at 200 timesteps		
Proccessor	-O3 flag speedup	-Ofast and vect flags speedup
Intel(R)i7-1165G7 @ 2.80GHz	x1.25	x2.87
Intel(R)Core(TM) i7-7500U CPU@2.70GHz 2.90 GHz	x0.85	x2.18
Intel(R)Xeon(R) E5520 @ 2.27GHz	x0.96	x1.93

3 Conclusion

We observed that the intermediate steps did not produce the anticipated outcomes on older CPUs; nevertheless, the final version of the code, which leveraged vectorization, resulted in a substantial performance increase across all machines used. From this, we can conclude that the most critical aspect of the optimization process is to utilize the appropriate compiler optimization flag and vectorize as much as possible. All team members contributed equally to the project.

4 References

GNU Compiler Collection (GCC): https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

```
    double (* restrict Fx)[N] = malloc(sizeof(double[N][N]));
    double (* restrict Fy)[N] = malloc(sizeof(double[N][N]));

    double time1 = get_wall_seconds();
    for(int n=0;n<nsteps;n++)// loop through all the time stepts
    {

        for(int i=0;i<N;i++)
        {


            double denominator;
            for(int j=0;j<i;j++)
            {
                rx=stars[i].x-stars[j].x;
                ry=stars[i].y-stars[j].y;
                r=sqrt((rx*rx)+(ry*ry))+1e-3;
                denominator=1.0/(r*r*r);
                Fx[i][j]=(-G*stars[j].m*rx)*denominator;
                Fx[j][i]=-Fx[i][j];
                Fy[i][j]=(-G*stars[j].m*ry)*denominator;
                Fy[j][i]=-Fy[i][j];

            }

        }
        for( int i = 0; i < N; ++i )
        {
            register double ax=0;
            register double ay=0;
            for( int j = 0; j < i; ++j )
            {
                ax += Fx[i][j];
                ay += Fy[i][j];
            }

            for( int j = i+1; j < N; ++j )
            {
                ax += Fx[i][j];
                ay += Fy[i][j];
            }

            // Speed.
            stars[i].vx+=delta_t*ax;
            stars[i].vy+=delta_t*ay;
        }
        for(int i=0;i<N;i++)
        {
            stars[i].x+=delta_t*stars[i].vx;
            stars[i].y+=delta_t*stars[i].vy;
        }

    }
```

Listing 3: Implementing the Newtons 3rd law

```
typedef struct star_acc
{
    double x;
    double y;
    double m;
}star_acc;
```

Listing 4: New structure

```
    star_acc *comp= (struct star_acc*)malloc(N *3* sizeof(double));
    double time1 = get_wall_seconds();
    for(int i=0;i<N;i++){
        comp[i].x=stars[i].x;
        comp[i].y=stars[i].y;
        comp[i].m=stars[i].m;

    }
    for(int n=0;n<nsteps;n++)// loop through all the time stepts
    {

        for(int i=0;i<N;i++)
        {

            register double ax=0;
            register double ay=0;
            double denominator;
            for(int j=0;j<i;j++)
            {
                    rx=comp[i].x-comp[j].x;
                    ry=comp[i].y-comp[j].y;
                    r=sqrt((rx*rx)+(ry*ry))+1e-3;
                    denominator=1.0/(r*r*r);
                    ax+=(-G*comp[j].m*rx)*denominator;
                    ay+=(-G*comp[j].m*ry)*denominator;
            }
            for(int j=i+1;j<N;j++){

                    rx=comp[i].x-comp[j].x;
                    ry=comp[i].y-comp[j].y;
                    r=sqrt((rx*rx)+(ry*ry))+1e-3;
                    denominator=1.0/(r*r*r);
                    ax+=(-G*comp[j].m*rx)*denominator;
                    ay+=(-G*comp[j].m*ry)*denominator;
            }

            stars[i].vx+=delta_t*ax;
            stars[i].vy+=delta_t*ay;
        }
        for(int i=0;i<N;i++)
        {
            comp[i].x+=delta_t*stars[i].vx;
            comp[i].y+=delta_t*stars[i].vy;
        }

    }
    for(int i=0;i<N;i++){
        stars[i].x=comp[i].x;
        stars[i].y=comp[i].y;
        stars[i].m=comp[i].m;
    }
```

Listing 5: Cache optimazation

```
    star_acc comp;
    comp.x  = malloc( N * sizeof(double) );
    comp.y  = malloc( N * sizeof(double) );
    comp.m  =malloc( N * sizeof(double));
    double time1 = get_wall_seconds();
    for(int i=0;i<N;i++){
        comp.x[i]=stars[i].x;
        comp.y[i]=stars[i].y;
        comp.m[i]=stars[i].m;
    }
    for(int n=0;n<nsteps;n++)// loop through all the time stepts
    {

        for(int i=0;i<N;i++)
        {

            vector a={0,0};
            double denominator;
            for(int j=0;j<i;j++)
            {
                    rx=comp.x[i]-comp.x[j];
                    ry=comp.y[i]-comp.y[j];
                    r=sqrt((rx*rx)+(ry*ry))+1e-3;
                    denominator=1.0/(r*r*r);
                    a.x+=(-G*comp.m[j]*rx)*denominator;
                    a.y+=(-G*comp.m[j]*ry)*denominator;
            }
            for(int j=i+1;j<N;j++){
                    rx=comp.x[i]-comp.x[j];
                    ry=comp.y[i]-comp.y[j];
                    r=sqrt((rx*rx)+(ry*ry))+1e-3;
                    denominator=1.0/(r*r*r);
                    a.x+=(-G*comp.m[j]*rx)*denominator;
                    a.y+=(-G*comp.m[j]*ry)*denominator;
            }
            stars[i].vx+=delta_t*a.x;
            stars[i].vy+=delta_t*a.y;
        }
        for(int i=0;i<N;i++)
        {
            comp.x[i]+=delta_t*stars[i].vx;
            comp.y[i]+=delta_t*stars[i].vy;
        }
    }
    for(int i=0;i<N;i++){
        stars[i].x=comp.x[i];
        stars[i].y=comp.y[i];
        stars[i].m=comp.m[i];
    }
```

Listing 6: Vectorization