Universität Göttingen
Institute for Computer Science
Practical Course on High-Performance Computing

MPI Simulation of a N-Body Solar System with Swing-by Maneuver

September 30, 2022

Name:	Yannik Feldner (yannik.feldner@stud.uni-goettingen.de)
Name:	Aaron Nagel (aaron.nagel@stud.uni-goettingen.de)
Tutor:	Jack Ogaja
Supervisor:	Prof. Dr. Julian Kunkel

Contents

List of Figures ii

List of Tables ii

1 Introduction 1

2 N-Body Solar System 3
2.1 Physics of N gravitational interacting bodies 3
2.2 Swing-by Maneuver 6

3 Methods 8
3.1 Program structure 8
3.2 Sequential implementation design 9
3.2.1 Definition of [planetary] objects and the spacecraft 9
3.2.2 Initialization of the N-body system 12
3.2.3 Visualization and animation with SDL 17
3.3 Solution algorithm 19
3.4 Parallel implementation design 24

4 Implementation 27
4.1 MPI implementation 27
4.2 Expectation 29

5 Analysis and performance 30
5.1 Run time 30
5.2 *Vampir* analysis 35

6 Improvements 38

7 Conclusion 39

References 40

8 Appendix 41
8.1 MPI implementation: main function 41

List of Figures

1 Visualization Swing-by maneuver 7
2 SDL-visualization of the swing-by maneuvers 18
3 Visualization of the Euler scheme 20
4 Simulation loop and parallelization-scheme 25
5 Data- and workload distribution 26
6 Run time expectation and simulation loop sequential/parallel 29
7 Variation study for the simulation time steps 31
8 Run time study for different processors over system size 1 32
9 Run time study for different processors over system size 2 33
10 Run time study for different processors over system size 3 34
11 *Vampire* analysis $n_p = 2$ 36
12 *Vampire* analysis $n_p = 5$ 36
13 *Vampire* analysis $n_p = 10$ 37

List of Tables

1 Reference simulation parameters and reference results 30
2 List of minima marked in figure 9 34

Listings

1 Definition of the object-class and typedef enum in the `body.h`-file. 10
2 `Object::init()`-function given in the `body.cpp`-file 11
3 `System::create`-function to initialize the objects included in the solar syste. 13
4 `System::init`-function to initialize the planetary objects, the sun and the spacecraft with the physical and numerical values needed for the simulation and defined in the object class. 14
5 Randomized initialization of the bodies in the asteroid belt implemented in the function `System::belt_init()` 15
6 `System::attraction()`-function to calculate the acceleration for one of the N bodies and the corresponding force acting on it due to gravitational interactions. 21
7 `System::update()`-function, which performes the Euler-step for the velocities and positions in x and y direction. 22

8 MPI Implementaion as constructed in figure 5 27
9 MPI Implementaion 41

1 Introduction

The goal of this project is to simulate a deep space or interplanetary mission of a conventional space launch system leaving it's home planet. The project idea is inspired by the Voyager 1 and Voyager 2 mission, which was one of the most successful programs conducted by NASA. The main achievement of the Voyager 1 and Voyager 2 was the observation of the outer planets of our solar system and the interplanetary space. The execution of this project was possible because of the observation, that the outer planets of our solar system would be in periodical alignment in the late 1970s. With this constellation it was possible to perform a number of swing-by maneuvers due to the gravitational interaction of the space transport system and the planet that should be observed leading to a deflection of the vehicle trajectory, allowing it to reach the next planet and increasing its velocity beyond the solar escape velocity. Therefore it was even possible to enter the interplanetay space.

In this project a similar trajectory to the Voyager mission will be performed for a test vehicle by implementing the gravitational laws in a C++ program code, representing the well known N-body problem of classical physics.

First of all a sequential version of the code was developed with the backward Euler scheme used for the temporal discretization of the problem, where the second order ordinary differential equation for the gravitational acceleration of the N-bodies was decomposed in two first order differential equations for the velocity and the position. The next step is given by the revision of the code and a parallel implementation by using the Message Passing Interface (MPI). This was done due to personal preference. We saw this project as an opportunity to get a better understanding of MPI since we are using this approach of parallel computing in various work related problems in the field of fluid mechanics.

The report will first give a brief explanation of the basic physical principles needed to implement the N-body problem. Afterwards the sequential approach and the implementation of the solution algorithm will be discussed leading to the updated parallel implementation of our program. For the parallel implementation, the general mechanism of the parallelization will be explained and afterwards we will take a closer look at the MPI implementation in our `main.cpp`-function. With this parallel implementation we conducted a performance analysis using runtime outputs in our program code and the analysis program *Vampir* together with *scorep*. The reference simulation for comparision is given by a number of objects with $N_{\text{tot}} = 211$, a maximum number of simulation steps equal to max_steps $= 10^5$, a simulation time on one processor of $t(n_p = 1) = 7.0$s and a computation time on two processors of $t(n_p = 2) = 8.9$s. First of all a linear increase

in computational time for an increasing amount of bodies could be observed, which was important to decrease the number of simulation steps for larger simulations in order to save time while still showing the same behaviour. In general the runtime for different numbers of processors show a parabolic behaviour of the runtime for $n_p \geq 2$, exhibiting a minimum value for in the interval of [2;8] for increasing system sizes. The increase of the computational time after the global minimum can be explained by the significant increase in communication time between the processors, making larger number of processors cost inefficient.

With the help of *Vampire* the time split between the `main`-function, Euler step and the MPI-communication can be shown. It is clearly visible, that MPI needs the most computation time due to the communication of the comparable large system. In order to improve this, the OpenMP approach with shared memory could be used in order to avoid the sending process of large amounts of data.

2 N-Body Solar System

The N-Body Problem is a well known fundamental problem in the field of physics and is used to describe the motion of gravitationally interacting bodies. In scope of this work the interaction of N-bodies will be observed in the context of a solar system with 1 sun, 8 planets, N_a asteroids in an asteroid belt between the 4th and the 5th planet of the solar system and an interplanetary space transport system. The goal of the simulation will be to perform multiple swing by or gravity assist maneuvers of the spacecraft by using the gravitational interaction of the vehicle and the corresponding planet. In order to get a basic understanding of the underlying physics of this N-body problem the general equations and principles will be explained in the following section.

2.1 Physics of N gravitational interacting bodies

This section deals with the basic physics of the N-Body problem and the corresponding equations needed to solve for the motion of celestial objects in a fictional solar system, which is inspired by our own solar system. The distances between the major bodies or planets in our fictional solar system and the sun are equal to the distances in our solar system, while the masses of some planets are changed in order to create a stronger interaction between the spacecraft and the corresponding planet. The mass of the spacecraft is given by 3.24×10^5kg, which is in range of the magnitude for comparable, conventional space launch systems like the Falcon 9 rocket produced by SpaceX or Ariane 5 constructed by the Ariane Group. Unless specified differently, the equations in this section are based on or derived from equations stated in Demtröder [2].
In order to describe the state of motion for any given body, the specific momentum is defined as:

$$\vec{p} = m \cdot \vec{v} = m \cdot \frac{\mathrm{d}\vec{r}}{\mathrm{d}t}, \tag{1}$$

where m is the mass of the observed body, $\vec{v}$ the velocity of its motion, $\vec{r}$ the position of the body and t the time. While the motion of a free moving particle is constant, a variation of the momentum implies an interaction between the observed body and another body or force field. The cause of the momentum variation is defined as the force acting on the body:

$$F = \frac{\mathrm{d}\vec{p}}{\mathrm{d}t} = m \cdot \frac{\mathrm{d}\vec{v}}{\mathrm{d}t} + \frac{\mathrm{d}m}{\mathrm{d}t}\vec{v}. \tag{2}$$

In general the mass of the observed body can be assumed to be constant. The rocket system can be seen as a special case. In general spacecrafts generate their thrust with combustion, chemical or electrical engines with a given mass flow, therefore the mass of

the spacecraft changes over time during the propelled flight phases. Since the mass of the spacecraft is significantly smaller than the mass of the other planets and especially the sun we decided to neglect this variation of the mass and assumed the aforementioned dry mass. Additionally we are not observing collisions of the different bodies and only their gravitational interactions since collisions are extremely unlikely under normal conditions and the orbits of the planets are not intersecting each other. Therefore Newton's second law can be simplified and the force can be specified as:

$$\vec{F} = m \cdot \vec{a} = m \cdot \frac{\mathrm{d}^2 \vec{r}}{\mathrm{d}t^2}, \tag{3}$$

where $\vec{a}$ is the acceleration and therefore the second derivative of the position vector $\vec{r}$.
In order to describe the interaction of the two bodies N_i and N_j Newton's law of gravity is used:

$$F_{ij} = \frac{G m_i m_j}{d^2}, \tag{4}$$

where G represents the gravitational constant with a value of $G = 6.6743015 \times 10^{-11} \frac{\mathrm{N \cdot m^2}}{\mathrm{kg}}$, m_i and m_j are the masses of the corresponding bodies N_i and N_j respectively and d is the distance.
The distance between the two bodies in a three dimensional space $\mathbb{R}^3$ is given by the euclidean norm of the distance vector between the two objects, which is defined as:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}, \tag{5}$$

where x, y and z are the position of the body with the given index.
The calculation of the distance between the planetary objects can be reduced to two dimensions with the observation, that the solar system and celestial formations in general can be assumed to be flat. During the formation process of celestial formations an accretion disk consisting of diffuse material and particles in an orbital motion forms around the massive central body, which is typically a star or black whole depending on the observed formation (solar system / galaxy). The acting forces together with the conservation of angular momentum lead to a spiraling motion of the accretion disk around the central body and a flat formation of the system. The planetary objects in this accretion disk form via collisions of the small particles. More detailed infromation on this topic can be taken from e.g. Rafikov [8]. In this case we only use the given statement to justify the assumption of a flat solar system in order to reduce the calculation of the distance

between two objects to:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \tag{6}$$

By combining the gravitational force eq.(4) with Newton's second law eq.(3), the acceleration of the body N_i resulting from the gravitational influence of body N_j can be calculated via:

$$m_i \frac{\mathrm{d}^2\vec{r}_i}{\mathrm{d}t^2} = \frac{G m_i m_j}{d^2} \Longleftrightarrow \frac{\mathrm{d}^2\vec{r}_i}{\mathrm{d}t^2} = \frac{1}{m_i}\frac{G m_i m_j}{d^2}. \tag{7}$$

If we now assume the gravitational interaction of N bodies instead of just two bodies, the acceleration of the body N_i can be calculated by observing the pairwise interaction with every body leading to:

$$\frac{\mathrm{d}^2\vec{r}_i}{\mathrm{d}t^2} = \frac{1}{m_i}\sum_{i=1, i\neq j}^{N} \frac{G m_i m_j}{d^2}. \tag{8}$$

This needs to be done for every single body leading to N^2-interactions and calculations that need to be solved in order to describe the physics of the system.

The movement of the different bodies can be approximated as Kepler orbits, describing the motion of the observed body relative to the central body. This simplification of the system can be done because of the significantly larger mass of the central body given by the sun. For comparison, the mass of the sun equals $M = m_{\mathrm{sun}} = 1.988 \times 10^{30}$kg, while the mass of earth is given by $m_{\mathrm{earth}} = 5.972 \times 10^{24}$ and is taken as reference mass. Therefore the mass of the stellar object in the center of our solar system is 332948.6 times higher than the mass of our reference body. As a result the central body can be assumed as center of mass or barycenter for the entire system and generates the main gravitational force acting on the planets orbiting it.

In case of the planets and asteroids we can observe steady elliptical and circular orbits. For these special cases the gravitational, attracting force between the central body and the observed planet or asteroid:

$$F_G = \frac{G \cdot m \cdot M}{r^2} \tag{9}$$

where M is the mass of the sun and m the mass of the observed celestial body, has to be equal to the centripetal force given by:

$$F_z = \frac{m \cdot v^2}{r}. \tag{10}$$

With this force balance:

$$F_G = \frac{G \cdot m \cdot M}{r^2} = F_z = \frac{m \cdot v^2}{r} \tag{11}$$

we can derive a constraint for the orbital velocity of a body with a steady orbit around its central body:

$$v_{\text{orbit}} = \sqrt{\frac{G \cdot M}{r}}. \tag{12}$$

A similar equation can be derived for the escape velocity of a body. This is needed for the initial value of our spacecraft. In case of the escape velocity, the kinetic energy of the spacecraft has to be larger than the gravitational binding energy of the corresponding planet [9]:

$$\frac{1}{2} m_{\text{rocket}} v^2 > \frac{G M_{\text{planet}} \cdot m_{\text{rocket}}}{r} \tag{13}$$

leading to an escape velocity of [9]:

$$v_{\text{escape}} > \sqrt{\frac{2 \cdot G \cdot M_{\text{planet}}}{r}}. \tag{14}$$

2.2 Swing-by Maneuver

In this section the general idea behind a gravity assist or swing-by maneuver will be explained in order to generate a basic understanding of the mechanism behind it. A visualization of a swing-by maneuver for four different cases is given in fig.(1), which is taken from [12] and modified for our purpose. The explanation of the swing by maneuver is based on the information taken from ([1], [6], [13]).
During a gravity assist maneuver a relatively small body approaches a planetary object with significantly higher mass and is redirected due to the gravitational interaction between these two objects. Therefore no further implementations of physical principles besides the concepts delivered in sec.(2.1) needs to be done. With this beeing said, the following section is only discussing the swing-by maneuver in a conceptual way. In order to understand the movement of the spacecraft during the swing-by maneuver, we will observe the behaviour in two different inertial frames. First of all in the inertial frame of the planet and afterwards in the inertial frame of the central body of the observed solar system, which is given by the sun.
By only observing the interaction of the spacecraft with the planet assisting in the swing-by maneuver in the inertial frame of the planet, the relative velocity between the two objects does not change (fig.(1) pictures on the left side), while the direction of the spacecrafts movement is deflected due to the gravitational influence. This leads to the conclusion, that the overall energy of the spacecraft remains the same in this observation. If we now change perspective to the inertial system of the sun we observe a different behaviour. First of all in the changed inertial system, the observed planet has a velocity

itself, which is given by the orbital velocity around the sun and the spacecraft has a relative velocity to the sun. During the approach of the spacecraft to the planet, the gravitational influence of the planet increases leading to a change in relative velocity of the spacecraft related to the sun. The new velocity is the superposition of the initial velocity of the spacecraft and the additional velocity added due to the gravitational influence of the planet, which is carrying the spacecraft with it through the space. If the spacecraft now approaches from behind the planet on its orbit, it is forced in a temporal orbit around the planet and moving in the same direction, leading to an increase in velocity. On the other hand, if the spacecraft approaches the planet in front of it and therefore enters an orbit in the opposite direction of the planetary movement, the spacecraft is decelerated. Both referenced cases can be seen in our simulation results later on.

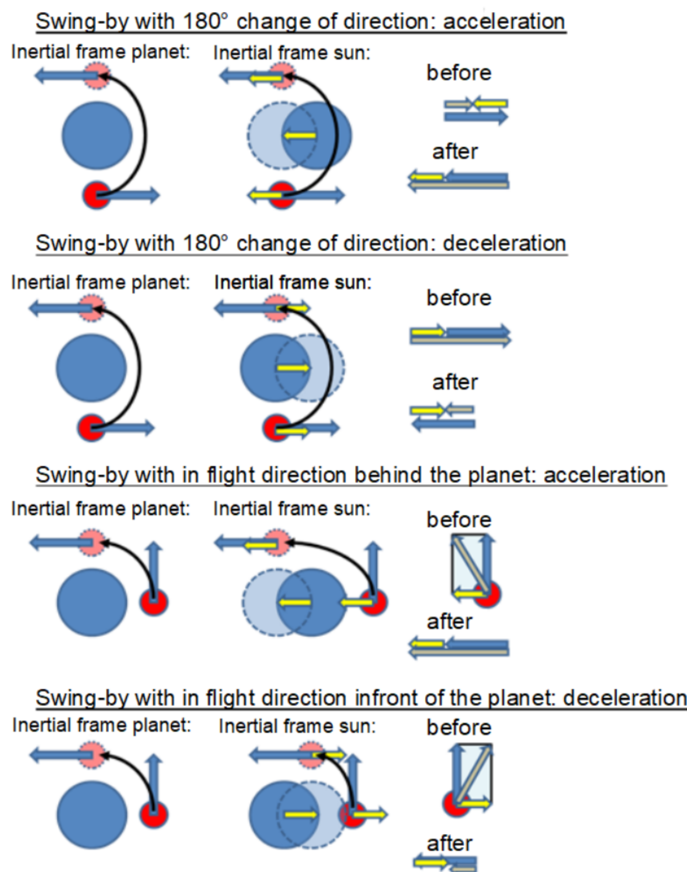


Figure 1: Visualization of a swing by maneuver with 90° and 180° deflection of the object trajectory for the acceleration and deceleration cases. The graphic is taken from [12], adjusted and is translated to English.

3 Methods

In the following section the different approaches of the implementation of our code will be explained and observed. First of all the general program structure and implementation approach will be discussed. Afterwards we will take a look at the initialization of the N-body system and the different celestial objects and the space craft based on the fundamental physical requirements discussed and derived in sec. 2. With this base understanding of the simulation problem the solution algorithm including the different simulation and update steps will be explained together with the implemented code and an explanation of the general formalism of an Euler step. In the end the sequential and parallel program design will be discussed.

3.1 Program structure

The final simulation program of our project is written in C++, while the first approach was written in Python and the visualization was done by using the pygame library. With python it was easier to implement a first simulation approach with good visualizations for the first course meeting. Afterwards we decided to change the programming language based on performance reasons and personal preferences of the group members. In C++ the visualization of the planetary motion and the swing-by maneuver of the space transport system is done by using the SDL2-library. With this beeing said we will only concentrate on the final C++ implementation. In this section we will give a rough overview over the different files published in the git-repository or project folder respectively as a starting point. More detailed information on the different simulation routines and implementations will be given in the following sections sec.(3.2.2-3.4)
The program is decomposed in different header and C++ files based on their functionality. The central piece of the program structure is of course the `main.cpp`-file including the `main()`-function. Besides the execution of the different initialization and cleaning functions for the system and the animation the main-function is used to handle the main, time resolved simulation loop of the system. Additionally the parallelization of the simulation routine is solely organized in the `main()`-function, which has two major advantages. First of all the sequential and parallel simulation can be executed by using the same project folder by only swapping the corresponding `main.cpp`-files and secondly allowed a distribution of workload between the group members by restricting the changes of the parallel implementation on one file for one person while the other person was working and implementing the sequential program structure and vice versa while still beeing compatible.

The `body.cpp`-file and `body.h`-file are used to define and implement the Object class for the planetary objects and the spacecraft itself and the creation, initialization and destruction-functions. Additionally the `system.cpp`-file and `system.h`-file are used to initialize the whole system by using the already esthablished body-class and in addition handels the solution algorithm. Finally a header and C++ file for the visualization with SDL2 are created.
The program can be compiled by using the `Makefile`-file with the standard `make`-command and cleaned again by using `make clean` additional information are provided in the deposited `README`.

3.2 Sequential implementation design

In this section the sequential implementation design and the different parts of the code will be explained. As already mentioned in the previous section, the parallel implementation is of course based on the sequential implementation and the only difference is the modified `main.cpp`-file. Therefore all statements and explanations that will be delivered are also correct and accurate for the parallel implementation.

3.2.1 Definition of [planetary] objects and the spacecraft

First of all the implementation of the `Object`-class in the corresponding `body.h`-file and `body.cpp`-file will be discussed. The program code is given in lst.(1).
The object class is used to initialize every single object used in the simulation. Therefore, in order to distinguish between the different object types, a data type alias is generated with the `typedef enum`-routine. The different types are `SUN`, `PLANET`, `ASTEROID` and `ROCKET`. These keywords will be used later on to apply different functions on the different object types. For example, the orbit of the objects will only be drawn for the planets and the spacecraft in order to generate a clear visualization of the asteroid belt, otherwise the individual objects in the belt could not be distinguished and would blur into each other.
Additionally we defined a function for the spacecraft to check if it crossed the orbit of a planet. This could be used to run the simulation with different velocities of the spacecraft and potential boosting phases to adjust the trajectory in order to reach the desired planet for varying initial conditions in subsequent projects.
The object class itself is set to public allowing every function to access the object and is defined with a given color value for the visualization and a name. Additionally the needed initial conditions of the observed object given by the position, the velocity and the acceleration are given in x and y direction together with the mass. The radius of the

planetary object could also be defined, which can be used to observe potential collisions of objects. Since those collisions are in general very unlikely they are neglected in this work and therefore the radius is not needed. In contrast to that, the animation radius must be defined to allow the visualization of the object in addition to the x and y position in the SDL simulation window in units of pixel. The orbit transit variable is an integer value for the spacecraft, that counts the transition of the planetary orbits starting with a value of 4, since the starting planet configuration is given by the earth Earth-moon system and they are considered as the 3rd and 4th body in our solar system. The final variables are the `orbit_size` and the `orbit`-coordinates in terms of the x and y coordinates. Every position of the observed object will be saved in the `orbit`-arrays in order to draw their orbits and potentially print them in additional `.txt`-files for further postprocessing.

```
typedef enum{
  SUN,
  PLANET,
  ASTROID,
  ROCKET

} Object_class;

class Object{
    public:

    const int *color;
    const char *name;

    double x;    // positions
    double y;
    double u;    // velocities
    double v;
    double ax;   // acceleration
    double ay;
    double radius;
    double mass;

    int pxl_x;
    int pxl_y;
    int anim_radius;
    Object_class type;
    int orbit_transit;

    int orbit_size = 4000;
```

```
31      double orbit_x[4000];
32      double orbit_y[4000];
33      // double *orbit_x;
34      // double *orbit_y;
35
36
37      void create();
38      void init(const char *name, double x0, double y0, double u0,
39          double v0, double r, double m, const int *col, int anim_r,
40          Object_class init_type);
41
42
43      void destroy();
44  };
```

Listing 1: Definition of the object-class and typedef enum in the `body.h`-file.

The three functions `Object::create()`, `Object::init()` and `Object::destroy()` are defined in the `body.h`-file and initialized in the `body.cpp`-file. The `Object::create()`- and `Object::destroy()`-functions can be used for larger orbit arrays to allocate and free the needed memory, while the `Object::init()`-function (see lst.(2)) is used to initialize the various objects in our solar system with their initial and starting conditions. Therefore the function takes the name, the x- and y-position and velocities, the radius, the mass, the visualization color, the animation radius and the type of the object as input and updates the corresponding parameters for the observed object. The initialization procedure for the N-body sysetm will be observed in the next section.
The initial position of the object in the SDL-visualization is calculated in [lst.(): line 17-18]. Since the coordinate-origin is shifted to the middle point of the visualization window in the `anim.h`-file later on, we add the x- and y-position scaled with a predefined scaling factor (which is defined in the `anim.h`-file).

```
1   void Object::init(const char *name_param, double x0, double y0,
2       double u0, double v0, double r, double m, const int *col,
3       int anim_r, Object_class init_type){
4
5       name = name_param;
6       color = col;
7
8       x = x0;
9       y = y0;
10      u = u0;
11      v = v0;
```

```
12
13     radius = r;
14     mass = m;
15
16     anim_radius = anim_r;                //Breite
17     pxl_x = ORIGx + x0*scale;          //x-Postition
18     pxl_y = ORIGy + y0*scale;          //y-Postition
19
20
21   /* Define the type of the initialized object:
22      Decide if the object is either the sun, a planet,
23      an astroid or the rocket                 */
24
25   type = init_type;
26
27     if(type == 3){
28       //printf("%s", name);
29       orbit_transit = 4;
30       //printf("%i", orbit_transit);
31     }
32 }
```

Listing 2: `Object::init()`-function given in the `body.cpp`-file

3.2.2 Initialization of the N-body system

The solar system is initialized in the `system.h` and `system.cpp`-file. As a first step we will take a look at the definitions made in the `system.h`-file, which are needed for the initialization of our boundary conditions.
First of all the RGB-values for the object-visualization are defined, these values will be used in the initialization-function. Additionally the number of bodies given by the different object types is specified and the total number is calculated. We decided to scale the physical values of our implementation in terms of astronomical units $\mathrm{AU} = 1.496 \times 10^{11}\mathrm{m}$, the mass of earth $m_{\mathrm{earth}} = 5.972 \times 10^{24}\mathrm{kg}$ and the anomalistic year $a = 3.169 \times 10^{7}\mathrm{s}$ in order to get rid of the high potency values. The astronomical unit is defined as the mean distance between earth and sun, while the anomalistic year is given by the time in which earth completes one orbit through its perihelion or the nearest point so the sun. Since the time is scaled with the anomalistic year, the velocity of earth is just given by $v_{\mathrm{earth}} = 2 \cdot \pi$. In addition to this, the gravitational constant, which is usually given by $\mathrm{G}_{\mathrm{phys.}} \approx 6.67 \times 10^{-11} \frac{\mathrm{m}^3}{\mathrm{kg} \cdot \mathrm{s}^2}$ has to be scaled in units of earth masses, anomalistic years and

AU leading to:

$$G_{\text{scaled}} = \frac{G_{\text{phys.}} \cdot M_{\text{earth}} \cdot a^2}{\text{AU}^3}. \tag{15}$$

The `system.h`-file includes a system class, in which various function for the initialization of the system, the initialization for the asteroid belt and a randomizer for the objects in the belt, the rocket status and the solution algorithm are defined. In the next step we will take a closer look at initialization functions implemented in the `system.cpp`-file, while the solution algorithm will be explained in sec.(3.3).
In the first step the solar system is created in the `System::create`-function (lst.(3)) by initializing the $N = N_p + N_a$-objects, where N_p is the number of planetary objects, the sun and the rocket stage and N_a is the number of asteroids. The previous values saved in the allocated memory locations are cleared in the for loop given in [lst.(3): line 11-13].

```
void System::create(){
    Objects->create();

    Np = Nbody;
    Na = NAstroids;
    N = Np + Na;

    // create empty Objects:
   Objects = new Object[N];
   for(int i = 0; i < N; i++){
       Objects[i].init("no_name", 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, BLUE, 10,
   PLANET);
   }
}
```

Listing 3: `System::create`-function to initialize the objects included in the solar syste.

With the completed memory allocation of the system objects and their parameters, the initialization of the different parameters is carried out in the `System::init`-function (lst.(4)). We decided to create a predefined planetary constellation of our system inspired by the voyager mission and a certain alignment of the major outer planets to guarantee multiple swing-by maneuvers of the spacecraft. Therefore the initialization of the sun, the planets and the spacecraft is done by calling the `Object::init()`-function for each individual body with predefined values, that are calculated beforehand (see. lst.(4): line 6-10). Because of the fact, that our system is rescaled in terms of astronomical units, earth masses and anomalistic year, the corresponding values for the different objects have to be sclaed in the same fashion. Since the velocity of earth is given by just $v_{\text{earth}} = 2 \cdot \pi$,

the velocity of the planets is calculated by using the following correlation:

$$v_{\text{n rescaled}} = \frac{v_{\text{n dim.}}}{v_{\text{earth dim.}}} \cdot 2\pi \tag{16}$$

where $v_{\text{n dim.}}$ and $v_{\text{earth dim.}}$ are the velocity of the observed planet and earth respectively in units of $\left[\frac{m}{s}\right]$ and $v_{\text{n rescaled}}$ is the rescaled velocity.

```
void System::init(){

    timestep = 0;
    double vel_earth = 2*M_PI; // velocity of the earth in nondim. units
    //init sun:
    Objects[0].init("sun", 0.0, 0.0, 0.0, 0.0, 16.0, 332948.6, YELLOW, 20,
   SUN);
    //init mercury:
    Objects[1].init("mercury", 0.387 , 0.0, 0.0, 1.592*vel_earth , 0.3829,
    0.055, DARK_GREY, 3, PLANET);
    //init venus:
  Objects[2].init("venus", 0.723, 0.0, 0.0, 1.1754*vel_earth , 0.9499,
    0.815, LIGHT_GREY, 8, PLANET);

  ...
  ...
}
```

Listing 4: `System::init`-function to initialize the planetary objects, the sun and the spacecraft with the physical and numerical values needed for the simulation and defined in the object class.

For the initialization of the objects in the asteroid belt another approach was chosen (see lst.(5)). In this case we wanted to implement randomized objects in a mean distance of $d_{\text{mean}} = 2.5\text{AU}$ to the central body of the simulation, which corresponds to the asteroid belt between Mars and Jupiter, the mean variation in distance is defined as $d_{\text{var}} = 0.5\text{AU}$. The mean mass of one asteroid equals $m_{\text{mean}} = 4 \times 10^{-14} m_{\text{earth}}$, while the deviation is given by $m_{\text{var}} = 2 \times 10^{-14} m_{\text{earth}}$.
The asteroids are initialized in a loop over every body considered inside the asteroid belt and by assigning a random angle between $[0°; 360°]$ and a distance in the interval of $[d_{\text{mean}} - d_{\text{var}}; d_{\text{mean}} + d_{\text{var}}]$. This is done by using the `System::rand_val(double min, double max)`-function which is generating a random value in the interval of `[double min; double max]` for the angle and the variation in distance. Afterwards the position of the asteroid is calculated with the generated distance and the sin or cos of the generated angle

for the x- and y-position respectively [lst.(5): line 34-41]. In order to create an object with circular orbit, the mean velocity has to be calculated via eq.(12) as explained in sec.(2.1) additionally a small random variation of the velocity can be added to the mean velocity resulting in a slightly changed elliptical orbit of the object. Finally the velocity of the asteroid is split into the x- and y-component and the mass is randomly generated based on the mean mass and the variation in mass. With these produced values for one of our observed asteroids, the `Object::init`-function is called to assign the values to their corresponding object parameters.
The same procedure can be done for various different planets or initial conditions in another project based on this code. Additionally the `System::status_Rocket(Object *Plt)`-function, which checks if the spacecraft has crossed the orbit of a planetary object, can be combined with a random constallation of planets in order to find an optimal launch date or additional boosting stages of the spacecraft for various gravity assit maneuvers. Since it was already difficult enough to find a constellation, that allowed multiple gravity assist maneuvers, the randomized initialization of planets was not implemented in this project but the option to come back to this problem is kept open.

```
1  void System::belt_init(){
2      //==================================================================//
3      // Initialization of Asteroids and Belt-Objects                     //
4      //==================================================================//
5
6    // mean distance and position and dist. variation
7    double mean_dist = 2.5;
8    double delta_dist = 0.5;
9    double dist = 0.0;
10   double x = 0.0;
11   double y = 0.0;
12
13   // velocity of an asteroid in the asteroid belt
14   double vel_earth = 2*M_PI;
15   double mean_vel = 0.6351*vel_earth;
16   //double mean_vel = 0.0*vel_earth;
17   double delta_vel = 0.0*vel_earth;
18   double vel = 0.0;
19   double vel_x = 0.0;
20   double vel_y = 0.0;
21
22   // velocity of an asteroid in the asteroid belt
23   double theta = 0.0;
24
```

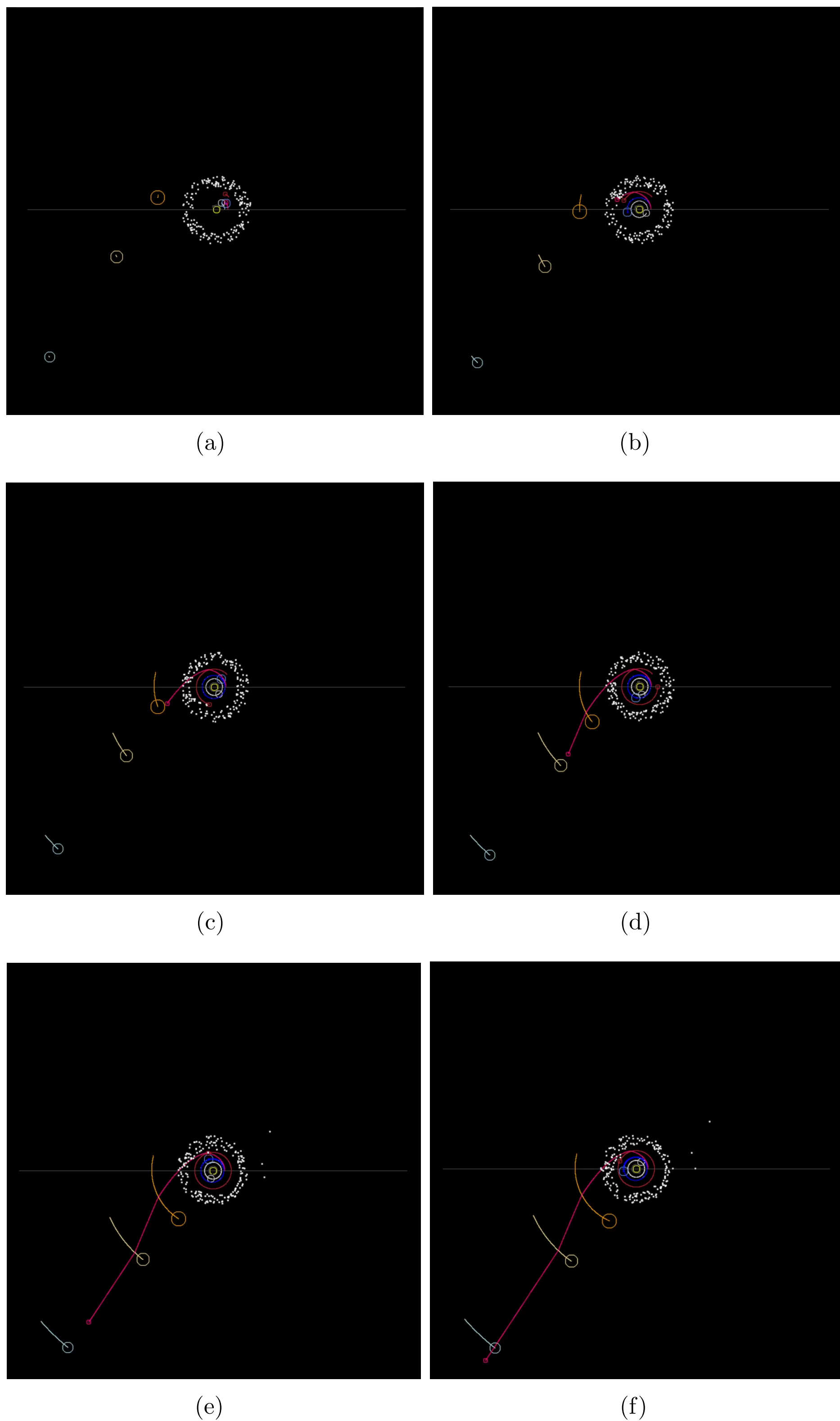
```
25   // mass of an asteroid in the asteroid belt:
26   double mean_m = 4*pow(10,-14);
27   double delta_m = 2*pow(10,-14);
28   double mass = 0.0;
29
30   srand(time(NULL));
31
32   for(int i = Np; i < Na; i++){
33
34     theta = rand_val(0, 360);
35     //printf("%f\n", theta);
36
37     dist = mean_dist + rand_val(-delta_dist, delta_dist);
38     //printf("%f\n", dist);
39
40     x = sin(theta)*dist;
41     y = cos(theta)*dist;
42
43     mean_vel = sqrt((G*M_sun/M_earth)/(dist));
44
45     vel = mean_vel + rand_val(-delta_vel, delta_vel);
46
47     vel_x = cos(theta)*vel;
48     vel_y = sin(theta)*vel;
49
50     mass = mean_m + rand_val(-delta_m, delta_m);
51
52
53     Objects[i].init("no_name", x, y, -vel_x, vel_y, 0.0, 1.0, WHITE, 2,
    ASTROID);
54   }
55 }
```

Listing 5: Randomized initialization of the bodies in the asteroid belt implemented in the function `System::belt_init()`

Besides the `System::attraction(Object *Plt)`- and `System::update(Object *Plt, double delta_t)`-function, that implement the solution algorithm of the N-body problem and will be discussed in sec.(3.3), the last two function in the `system.cpp`-file are given by the `System::write_orbit(Object *Plt)`- and `System::destroy()`-function. The first of the two functions is used to save all visited positions of every object in the corresponding orbit-vector in order to use it in the visualization scheme and the second function frees the objects and cleans the allocated memory.

3.2.3 Visualization and animation with SDL

The problem visualization with the SDL2-library is solely done for the sequential implementation of the program code in order to save computation time and avoiding the implementation of SDL2 on the cluster. In order to compensate for this, the orbit-arrays can be saved in a `.txt`-file and visualized on the home computer by using SDL2 in C++ or switch to a visualization with python with e.g. pygame. By comparing the results of the orbit arrays for the sequential and parallel implementation the expectations of an unchanged physical behaviour of the objects can be confirmed and therefore a validation of our parallel implementation procedure can be given. Since the SDL2 routines are simply used for the visualization and therefore not the main focus of the report, we won't go into a very detailed description of them. More detailed information can be taken from the SDL documentation [5].
The visualization of the simulation output is organized in the `anim.h`- and `anim.cpp`-file. First of all the main simulation parameters are defined in the `anim.h`-file containing the height and width of the visualization window in units of pixel and the origin of the coordinate system, which would normally be in the top left portion of the output window is set to the center point. The scale parameter, which is used to scale the simulation output in terms of pixels in the simulation window is defined and can be adjusted depending on the desired visualization frame of the solar system. Additionally the main functions used in the `anim.cpp`-file are defined in the class `Anim`.
In the `Anim::init()`-function, the `SDL_Init(SDL_INIT_VIDEO)`-function is called to initialize the SDL-video output, while `Anim::create()` defines the position and the size of the generated output window. The `Anim::destroy()`-function uses SDL2-routines to close the visualization window after it is finished.
The last two important functions are the `Anim::DrawCircle(SDL_Renderer * renderer, int32_t centreX, int32_t centreY, int32_t radius)`-function which is used to create circles of various sizes for the visualization of the planetary objects, while the main routine is implemented in `Anim::doRender(SDL_Renderer *renderer, Object*object, int timestep)`. In this function, the differnt colors of the output window for the background and coordinate axes are defined. Additionally the main loop of the function is running over the bodies of the system [0; Ntot], gets their color value and draws the corresponding circle for the object. Finally the orbit of the observed objects is drawn in case that they are not an asteroid.
The final visualization output is saved as a video in MP4-format and attached to this report. A picture for different timestamps of the output video is given in fig.(2).



(a) (b) (c) (d) (e) (f)

Figure 2: SDL-visualization of the swing-by maneuvers for different observation points increasing in time. The time timestamps are given in the format hours:minutes:seconds corresponding to the timestamps of the attached simulation video.
(a) 00:00:00, (b) 00:00:12, (c) 00:00:46, (d) 00:00:59, (e) 00:02:45 and (f) 00:03:15

3.3 Solution algorithm

For the temporal discretization of the N-body problem and therefore the calculation of the new positions, velocities and accelerations of the observed bodies the standard forward Euler scheme, which is an explicit method, will be used. While explicit numerical methods only need the current information about the state of the observed system and therefore are easier to implement, implicit methods also need information about the state in the next time step, which is supposed to be calculated. The following section is based on the fundamental theories and principles described in the textbook written by Ferzinger [3]. Additionally a visualization of the Euler scheme can be taken from fig.(3).
The basic equation, that is needed to be solved for the N-body problem is the second order ordinary differential equation for the acceleration of the bodies eq.(8):

$$\frac{\mathrm{d}^2\vec{r}_i}{\mathrm{d}t^2} = \frac{1}{m_i}\sum_{i=1,i\neq j}^{N}\frac{Gm_im_j}{d^2} = \underbrace{\frac{1}{m_i}\sum_{i=1,i\neq j}^{N}\frac{Gm_im_j}{|r_i - r_j|^2}}_{f}. \tag{17}$$

for the sake of simplicity we are going to rewrite this equation in terms of a generalized variable Φ, which corresponds to the position vector $\vec{r}$ and a function depending on this variable on the right hand side of the equation f that could also depend on the time. Additionally the second order differential equation can be reduced to a first order differential equation by taking an additional calculation step. The position of the object is given by $\vec{r_{new}} = \vec{r_{old}} + \vec{v} \cdot \Delta t$, where $\vec{r}$ are the old and new positions of the object, $\vec{v}$ is the current velocity and Δt is the time step. The same counts for the acceleration a: $\vec{v_{new}} = \vec{v_{old}} + \vec{a} \cdot \Delta t$. Therefore the second order differential equation can be reduced to a first order differential equation by first calculating the velocity and afterwards the position instead of calculating the position directly from the acceleration. With this in mind the general form of the differential equation can be written as [3]:

$$\frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} = f(t, \Phi(t)) \quad \text{and} \quad \Phi(t_0) = \Phi^0, \tag{18}$$

where Φ^0 are the initial conditions for the observed objects given and explained in sec. 3.2.2. The goal of the Euler scheme is now to calculate the next point in space-time for the moving objects after a given timestep Δt. In case of the Euler method eq.(18) will be integrated over time starting from the current time t_n to the next point in time given by

$t_{n+1} = t_n + \delta t$:

$$\int_{t_n}^{t_{n+1}} \frac{\mathrm{d}\Phi}{\mathrm{d}t} = \Phi^{n+1} - \Phi^n = \int_{t_n}^{t_{n+1}} f\left(t, (\Phi)\right) \mathrm{d}t, \tag{19}$$

with $\Phi^{n+1} = \Phi\left(t_{n+1}\right)$. This exact solution of the problem can not be solved without knowing the solution of the integral on the right hand side of the equation. Therefore the left Riemann sum is used as an approximation of the integral. The principle of the Riemann sum says, that we can divide an interval between $[\Phi^n, \Phi^{n+N}]$ in m subintervals with an individual length of [11]:

$$\Delta t = \frac{\Phi^{n+N} - \Phi^n}{m}. \tag{20}$$

Therefore the integral of the whole interval can be approximated via [11]:

$$\int_{t_n}^{t_{n+N}} f\left(t, (\Phi)\right) \mathrm{d}t = \Delta t \cdot \left[f(\Phi^n) + f(\Phi^n + \Delta t) + ... + f(\Phi^{n+N} - \Delta t)\right]. \tag{21}$$

Since we are only observing the next time step in the position update of the Euler method, the mechanism reduces to:

$$\int_{t_n}^{t_{n+1}} f\left(t, (\Phi)\right) \mathrm{d}t = \Delta t \cdot f(\Phi^n). \tag{22}$$

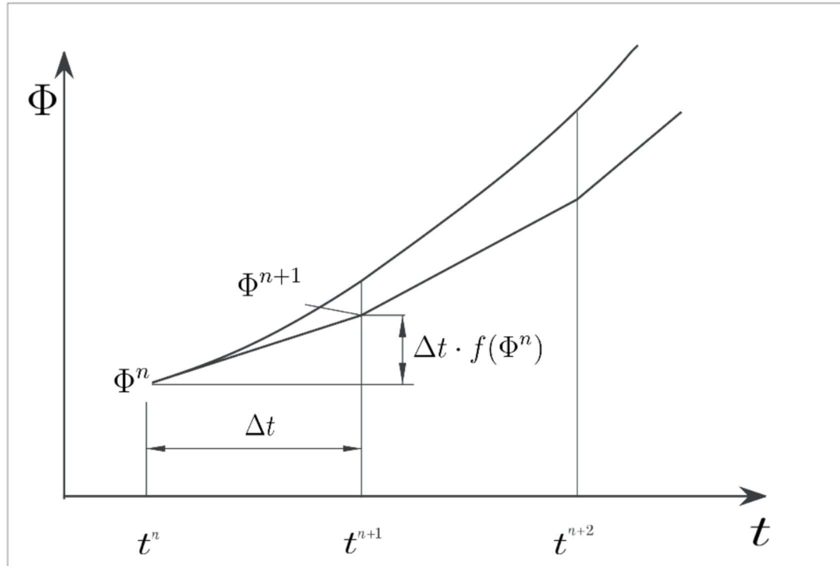


Figure 3: Visualization of the Euler scheme. Image was taken from [7] and modified.

If we plug this result into eq.(19) the update scheme for the Euler mechanism can be

written as [3]:

$$\Phi^{n+1} - \Phi^n = \Delta t \cdot f(\Phi^n) \Longrightarrow \Phi^{n+1} = \Phi^n + \Delta t \cdot f(\Phi^n) \tag{23}$$

After delivering the theoretical principles of the solution algorithm, the implementation of the Euler scheme will be observed.
The first step of the solution mechanism is given in the `System::attraction()`-function (see lst.(6)). This function takes an object as input and calculates the force acting on this body depending on the gravitational influence of the remaining $N-1$ bodies. After the variable initialization and the cleaning of the previous accelerations, the main function loop starts. In this loop, the distance between the observed planet and the remaining $N-1$ bodies is calculated [lst.(6): line 16-17] which is in return used to calculate the gravitational force acting on the observed body by using eq.(9). Afterwards the force needs to be split in the x and y component respectively in order to resolve the motion of the body in the vector space $\mathbb{R}^2$. Since the Euler scheme is reduced to a system of first order ordinary differential equations as explained previously, the algorithm needs the corresponding accelerations of the observed bodies to calculate the velocities and positions [lst.(6): line 27-28]. This is done by using Newton's second law (eq.(3)) and rewriting the equation in terms of the acceleration by dividing by the planetary mass. The calculated values are saved in the Object class and are used to update the velocities and positions in the `System::update()`-function. The `System::update()`-function takes the predefined timestep of the simulation, aswell as the observed object as input and performs the Euler-step by solving eq.(23) for the velocities in x and y direction [lst.(7): line 4-5] and the x- and y- positions [lst.(7): line 7-8] respectively. Additionally the rescaled x- and y-position for the SDL-visualization are calculated in in units of pixels [lst.(7): line 7-8].

```
1  void System::attraction(Object *Plt){
2
3      double force = 0.0;
4      double dist_x = 0.0;
5      double dist_y = 0.0;
6      double dist = 0.0;
7      double F = 0.0, F_x = 0.0, F_y = 0.0;
8
9      // set forces to zero and calculate them new:
10     for(int i = 0; i < Np; i++){
11         Plt->ax = 0.0;
12         Plt->ay = 0.0;
13     }
```

```
    for(int i = 0; i < Np; i++){
        dist_x = Plt->x - Objects[i].x;
        dist_y = Plt->y - Objects[i].y;
        dist = sqrt(dist_x*dist_x + dist_y*dist_y);

        // do not calculate force on body on itself
        // also, because mathematically, you would devide by dist=0
        if(strcmp(Objects[i].name, Plt->name) != 0){
            F = -(G * Plt->mass * Objects[i].mass)/(dist*dist);
            F_x = F * (dist_x/dist);
            F_y = F * (dist_y/dist);

            Plt->ax += F_x/Plt->mass;
            Plt->ay += F_y/Plt->mass;

            // to imporve perfomance, one could also use the calculated
            // F_x and F_y to update forces on the other objects
            // but then they also need to be reset to zero seperately
            // before accumulated
            // Planets[i].fx -= F_x/Planets[i].mass;
            // Planets[i].fy -= F_y/Planets[i].mass;

        }

    }

}
```

Listing 6: `System::attraction()`-function to calculate the acceleration for one of the N bodies and the corresponding force acting on it due to gravitational interactions.

```
void System::update(Object *Plt, double delta_t){

    // ------ EULER step: ------ //
    Plt->u += Plt->ax * delta_t;
    Plt->v += Plt->ay * delta_t;

    Plt->x += Plt->u * delta_t;
    Plt->y += Plt->v * delta_t;

    // calc position in rendering:
    Plt->pxl_x = ORIGx + scale*Plt->x;
    Plt->pxl_y = ORIGy - scale*Plt->y;
```

```
13 }
```

Listing 7: `System::update()`-function, which performes the Euler-step for the velocities and positions in x and y direction.

3.4 Parallel implementation design

To increase the performance of the simulation, the workload is distributed to multiple processors to calculate attraction and position for a group of objects simultaneously. The distribution of workload is done by *Message Passing Interface (MPI)* to parallelize the computations. MPI is used to send information to different processors that each will perform a subset of calculations or tasks that need to be done at a certain time in a program.
Before considering how the workload should be distributed, it is necessary to identify those parts of the program where calculations can be performed parallely. The main simulation loop performs discrete calculations of the N orbits in each iteration step at a fixed time step t_i. When calculating the attraction for each object at time step t_{i+1}, only the positions of all objects at the previous time step t_i are needed. These results are available when calling the attraction function, thus, the attraction of all objects can be calculated individually without the need of partial results. The workload of calculating the attraction of N objects can be distributed to P processors, each calculating the attraction of $\sim N/P$ objects. Now, this is not an exact value for two reasons: first, processor 0 is usually used to coordinate all the workload distribution which results in a total of $P-1$ working processors and secondly, because there will be the number of distributed objects of the integer division per processor, which may result in a slightly different number for all processors for non-zero remainder.
Before calculating the discrete position update for the time step t_{i+1}, the attraction for the objects in this time step t_{i+1} are fully determined. When this is done, all the results can be collected and the next part of the program for parallelization can be identified. Updating the positions of all objects can be done simultaneously again. The results will be collected before the next loop cycle for the next time step starts again. This data collecting will be a performance bottleneck but can not be avoided because all the positions need to be updated before the attraction, even for a single object, of a new time step can be calculated. An overview of the workload distribution is shown in figure 4.
To have multiple processors available to communicate with each other, the main program that each of them calls will initialize MPI with an MPI communicator and its own rank and total rank size at the very beginning.
As described in in section 3.2 for the sequential implementation, the functions *attraction* and *update* are defined as methods in the `System` class operating on the `Object` class. To have access to these methods, each processor is suppose to initialize these objects (note: here, object is referred to the expression used to denote a data structure in object-oriented

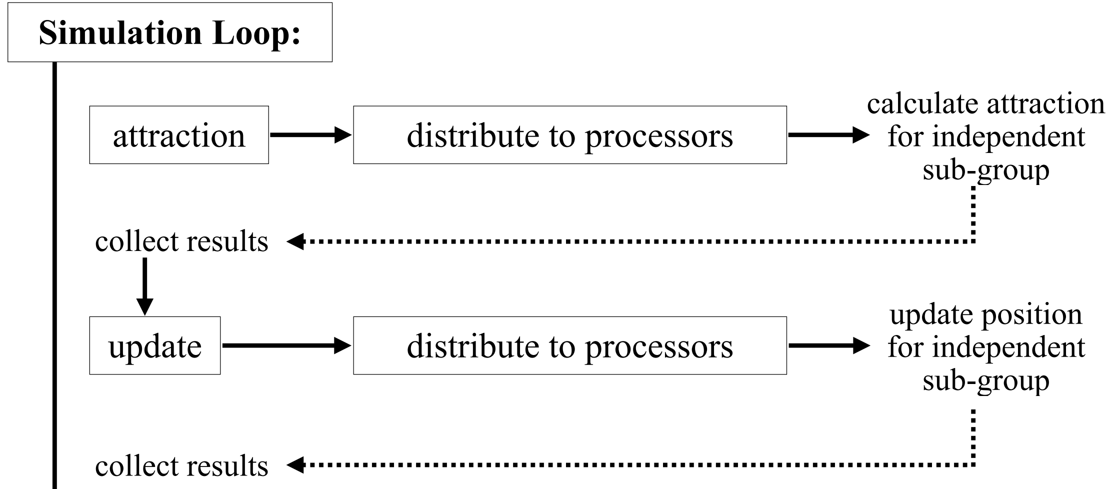


Figure 4: Where to parallelize: it is shown the simulation loop with the two functions *attraction* and *update* and their distribution of tasks to multiple processors.

programming. Unfortunately, the class used to describe sun, planets, rocket, asteroids is called `Object`).
To send and receive the data between the processors, each processor additionally allocates memory for buffer arrays for positions $\vec{x}$, velocities $\vec{v}$ and accelerations $\vec{a}$, each of length $2N$ and masses $\vec{m}$ of length N, for the N objects in the two dimensional simulation.

Before the data can be sent to the processors, the relevant data needs to be extracted from the `system` and `objects` classes. To calculate the attraction of an object on any processor, the position and masses of every single object in the system is needed. Thus, all the positions and masses need to be send to every processor. Once they have received these data, each processor will operate only on a subset of objects, determined by its own rank. The received data will be sorted from the buffer arrays into the `system` class to be able to perform the *attraction* method on certain objects. For a total of N objects in the system with $P-1$ working processors, processor with rank $p \in [1, P-1]$ will perform the calculation of the objects with indices:

$$i \in [(p-1) \cdot N/(P-1),\ p \cdot N/(P-1)]. \tag{24}$$

Once the calculation for all these objects are finished, the data will be sorted into the buffer arrays to be prepared to get sent to the coordinating processor with rank $p = 0$ again. Processor 0 collects all the calculated attraction from all the processors and sorts the data into its own `system` class allocated on on itself. So `system` on processor 0 is the only memory that does not serve as some sort of buffering storage but actually contains the fully determined system state of the time step. The implementation plan is shown in figure 5.

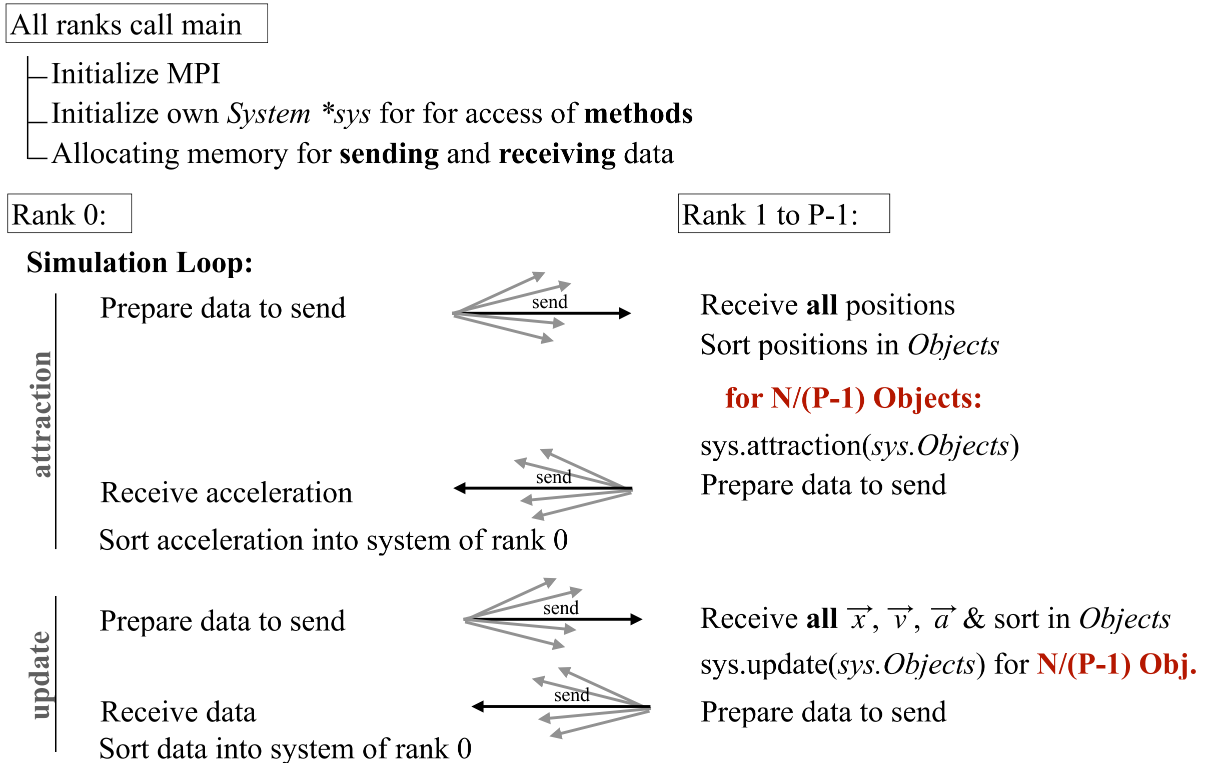


Figure 5: Details on how data and workload is distributed and coordinated using MPI.

4 Implementation

4.1 MPI implementation

Translating the design of MPI parallelization for the simulation as constructed in figure 5, the main code in the simulation loop for coordinating processor `rank==0` and the working processors `rank!=0` is shown in listing 8. The complete main function implementation can be seen in appendix section 8.1 in listing 9.

```
if(rank == 0){
while(!anim.done){
    // ------ calc. attraction ---------- //
    for(int p = 1; p < num_proc; p++){
      MPI_Send(y, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
      MPI_Send(mass, N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
    }
    for(int p = 1; p < num_proc; p++){
      MPI_Recv(a, 2*N, MPI_DOUBLE, p, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // SORT:
      for(int i = (p-1)*N/(num_proc-1); i < p*N/(num_proc-1); i++){
        sys.Objects[i].ax = a[2*i];
        sys.Objects[i].ay = a[2*i+1];
      }
    }

    // ------- update all positions: ------------ //
    for(int p = 1; p < num_proc; p++){
      MPI_Send(y, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
      MPI_Send(v, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
      MPI_Send(a, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
    }
    for(int p = 1; p < num_proc; p++){
      MPI_Recv(y, 2*N, MPI_DOUBLE, p, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv of v, a, pxl ...
      // SORT:
      for(int i = (p-1)*N/(num_proc-1); i < p*N/(num_proc-1); i++){
        sys.Objects[i].x = y[2*i];
        sys.Objects[i].y = y[2*i+1];
        and sort v, a and axl ...
      }
    }
```

```
35          ...
36          timestep++;
37    }     }
38
39    if(rank != 0){
40    while(!done){
41      // ------ calc. attraction --------- //
42      MPI_Recv(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
43      MPI_Recv(mass, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
44      // SORT:
45      for(int i = 0; i < N; i++){
46        sys.Objects[i].x = y[2*i];
47        sys.Objects[i].y = y[2*i+1];
48        sys.Objects[i].mass = mass[i];
49      }
50      for(int i = (rank-1)*N/(num_proc-1); i < rank*N/(num_proc-1); i++){
51        sys.attraction(&sys.Objects[i]);   // <<< CALCULATE
52        // SORT:
53        a[2*i] = sys.Objects[i].ax;
54        a[2*i + 1] = sys.Objects[i].ay;
55      }
56      MPI_Send(a, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
57
58      // ---------- update pos. ---------- //
59      MPI_Recv(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
60      ... and MPI_Recv v and a ...
61      // SORT:
62      for(int i = 0; i < N; i++){
63        sys.Objects[i].x = y[2*i];
64        ... sort y, v and a ...
65      }
66      for(int i = (rank-1)*N/(num_proc-1); i < rank*N/(num_proc-1); i++){
67        sys.update(&sys.Objects[i], delta_t);    // <<< EULER SPEP
68        // SORT:
69        ... and sort y, v, a and pxl ...
70      }
71      MPI_Send(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
72      ... and MPI_Send v, a and pxl ...
73    }  }
```

Listing 8: MPI Implementaion as constructed in figure 5

4.2 Expectation

They way the sequential solution is extended to a parallel solution, the amount of data exchange must be worth compared to the amount of calculations that are done. This means that for small system sizes, too many processors might not be efficient when there is to much communication compared to calculation. This should hold in general.
One specific expectation of the run time independent of the system size should be the case of comparing the run time of the sequential implementation with the run time of the parallel implementation on two processors. As described in section 3.2 of the parallelization design, MPI is used and data is distributed from processor 0 to all other processors, where the calculations are made. This means that for a parallel run on two processors, only processor 1 will perform calculations while processor 0 will send and receive all data to and from processor 1. If the calculations are performed on only one processor and additional sending of data is made, this parallel program run time generally should be larger than the run time performed by the sequential implementation, where calculations are also made on only one processor but no data need to be sent and received. For a plot of run time depending the number of processors $t(n_p)$, the curves generally should satisfy $t(n_p = 1) < t(n_p = 2)$. The additional tasks by a parallel implementation on two processors compared to the sequential implementation is shown in figure 4.2.

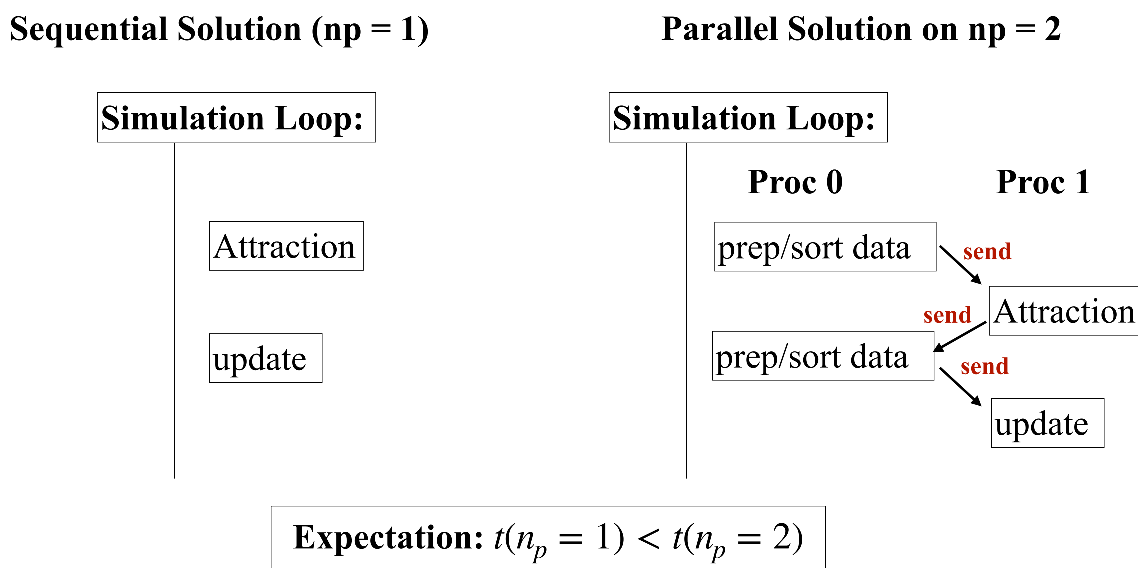


Figure 6: Expectation of run time comparing sequential implementation with parallel MPI implementation on two processors.

5 Analysis and performance

The analysis is split into two parts: the first part will show plots of the pure run time analysis and the second section goes into more detail of MPI initialization time, send and receive time, and actual performance fraction time using *Vampir* in section.

Reference case

The reference simulation from where parameters are varied and performance is studied is done by the parameters shown in tabel 1.

Table 1: Reference simulation parameters and reference results

number of objects	number of time steps	$t(n_p = 1) =: t_{\text{seq}}$	$t(n_p = 2)$
$N = 211$	$\texttt{max_steps} = 10^5$	7.0s	8.9s

5.1 Run time

To measure the pure run time of the program running on different number of processors, `time.h` is used and simple measurement points are inserted. For a fixed number of processors, the MPI initialization time should be constant whereas the overall run time depends on the number of performed time steps. For a dominating amount of simulation run time compared to the MPI initialization time, this fraction could be neglected but to not have to worry about that, the time measurement starts after the MPI initialization and after the memory allocation. Details on the proportion and fraction in run time including the MPI initialization and further details on send, receive and calculation run time will be done in section 5.2. Additionally, when denoted in the y-axis label, the measured run times $t(n_p)$ are normalized with the run time on one processor $t(n_p = 1)$ or two processors $t(n_p = 2)$, so the results are independent of the system size and its number of objects.
The first analysis has been done to make sure that the run time increases linearly with the amount of time step that are performed. This was important because it allows to compare the run time of large systems, so with many bodies N, with small systems even for different number of performed time steps. To make sure that this holds indeed, a plot for different number processors for varying the number of time steps is shown in figure 7 and a linear dependence can be verified.

Another visualization of the same data, now normalized, is shown in figure 8. Due to

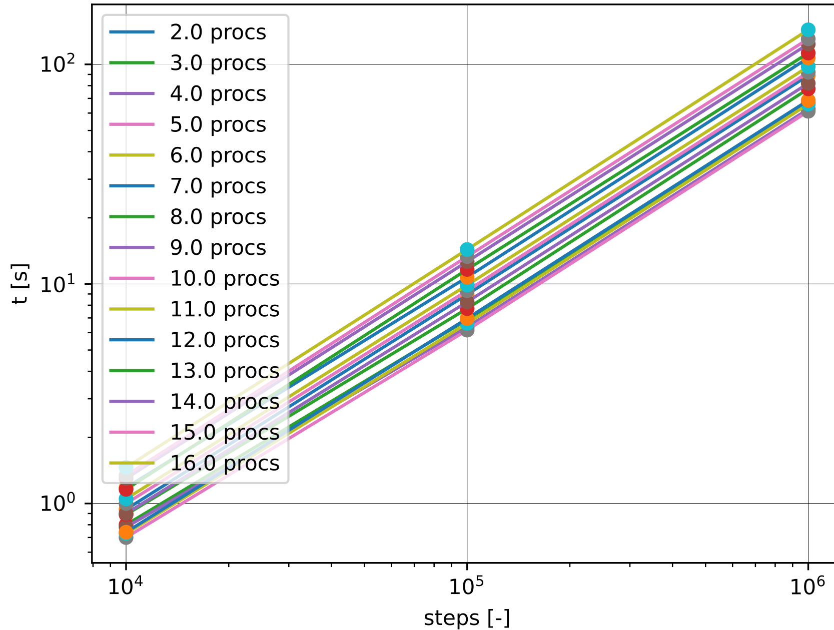


Figure 7: Variation for the number of simulation time steps to verify a linear increasing run time using the reference case.

the normalization factor $t(n_p = 2)$, the results can be compared well and show the independence of the number of total time steps. The plot shows the $t(n_p)$ graphs for a total of $10^4, 10^5$ and 10^6 time steps. Note that the graph with the 10^5 steps is the reference case. The graph with the smallest number of total time steps show deviations to the smoother looking curves, which will be the case due to a smaller set of time steps acting as a sort of samples.

An interesting observation is the minimum value at $n_p = 5$. For The reference case, the normalized run time decreases to a value of 0.69, which indicates a reduction of run time on $n_p = 5$ processors of about 30% compared to a run on $n_p = 2$ processors. With even more processors, the run time increases again as described in section 6 due to too much MPI communication. Using $n_p > 9$ processors, the run time is even larger than the run time on $n_p = 2$ processors. This is just as expected.

To this point, the observations and expectations indicate, that the efficiency of multiple processors should depend on the proportion of system size to the number of processors used. Thus, for increasing system sizes, the minimum of the run time for varying number of processors used, should shift towards a larger number of processors. The larger the system size, the more objects are in the system and thus more processor will be more

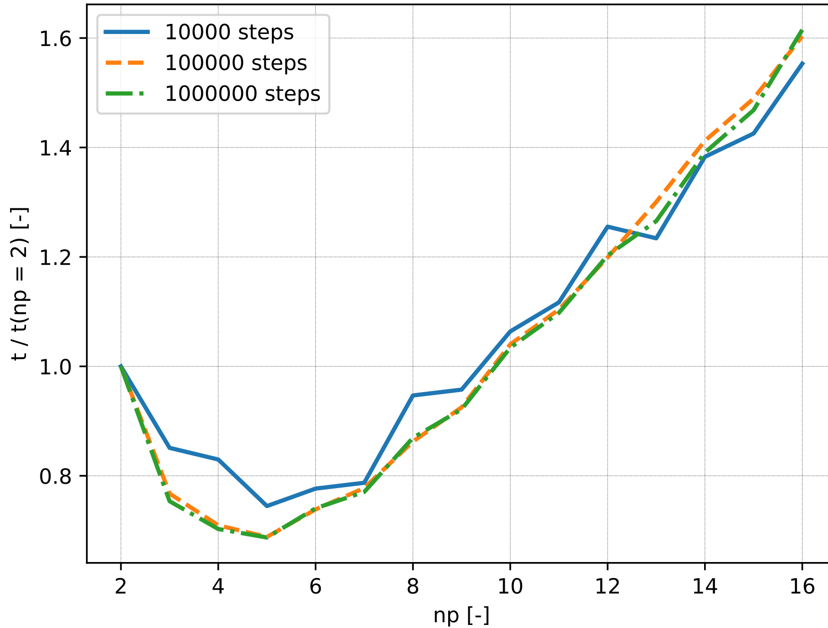


Figure 8: $t(n_p = 2)$-Normalized run time on processor number dependence n_p of the reference case for varying number of maximal time steps.

efficient up to the point where too many processors need to much time to communicate compared to the amount actual calculation time needed. Taking a look at figure 9 with marked minima t_{min} confirms these considerations. The exact values of the minima are listed in table 2. The data normalized with the run time on two processors instead of one is shown in figure 10.
Comparing figure 9 and 10 show two interesting things:

1. There is only a range of the system size where a parallel solution using MPI is faster than a sequential solution. In our simulations this can be identified by those minima that have a value of less than 1.0 in figure 9 and in table 2 in column with normalization factor $t(n_p = 1)$. From our data we can estimate this system size to be within a range of $161 \leq N \leq 20011$.
2. For a system with sufficiently large size, there is always a number of processors greater than two, where the run time is smaller than the run time on only two processors. This can be seen by the minima all being less or equal 1.0 in figure 10 and in table 2 in column with normalization factor $t(n_p = 2)$. *Sufficiently large* can be gained from our data to be more than $N_{\text{min}} = 91$ but might be smaller. As already mentioned, the location of the

minimum shifts towards more processors for larger system sizes.
Figure 9 and 10 also show really nicely the expectation that $t(n_p = 1) < t(n_p = 2)$ as explained in section 4.2 and shown in figure 6 generally hold for arbitrary system sizes.

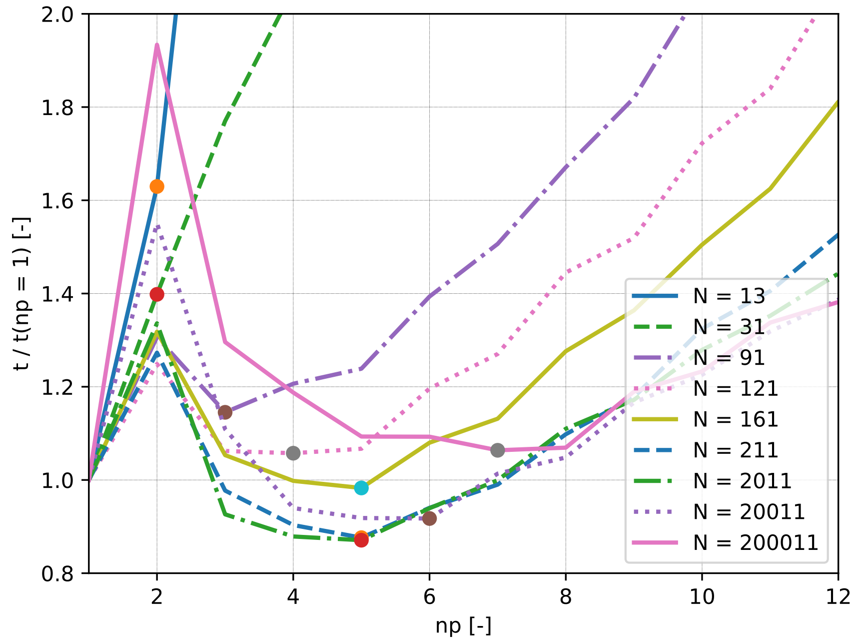


Figure 9: $t(n_p = 1)$-Normalized run time on processor number dependence for different system sizes.

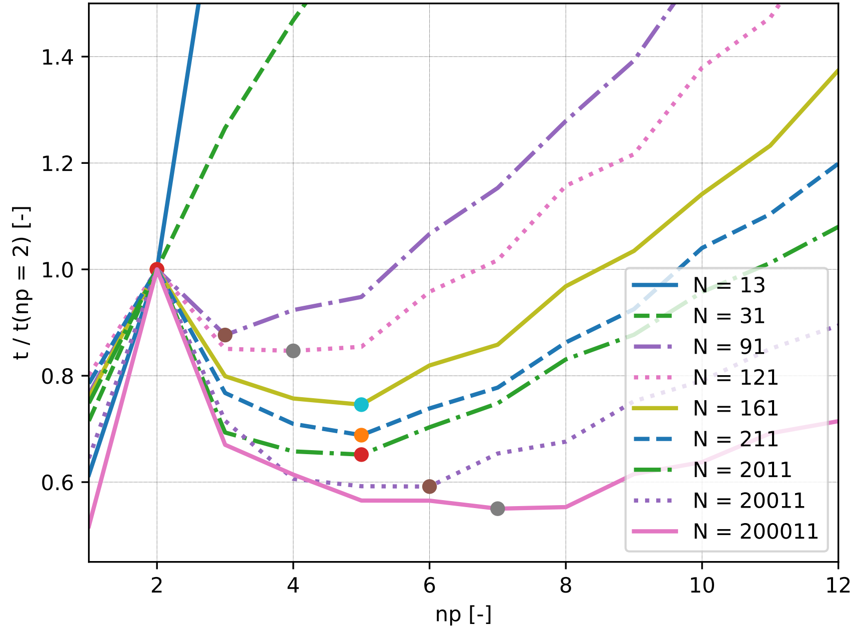


Figure 10: $t(n_p = 2)$-Normalized run time on processor number dependence for different system sizes.

Table 2: List of minima marked in figure 9

N	$n_{p,\min}$	t	$t/t(n_p = 1)$	$t/t(n_p = 2)$
13	2	0.88	1.63	1.00
31	2	1.58	1.40	1.00
91	3	3.55	1.15	0.88
121	4	4.42	1.06	0.85
161	5	5.16	0.98	0.75
211	5	6.16	0.88	0.69
2011	5	57.76	0.87	0.65
20011	6	65.94	0.92	0.59
200011	7	117.93	1.06	0.55

5.2 *Vampir* analysis

This section will take a detailed view on the fraction and proportion of the different computation times of the reference case with parameters as listed in table 1. In particular, the percentage portion of `MPI_Init()`, the MPI communication by `MPI_Send()` and `MPI_Recv()`, the simulation calculation by `System::attraction()` and `System::update()`, as well as the `main()`-function itself are displayed as pie charts. The results are investigated for a run of the reference case on $n_p = 2, 5, 10$ processors. To gain the measurement data, the program is instrumented by *Score-P* [10] and the results are visualized by *Vampir* [4].
Score-P is a tool that takes a program code as input and instruments it in such a way, that performance and event tracing measurements are taken when executing the program [10]. It allows to measure the run time of certain functions but also to track the time evolution of function calls, even distributed along multiple processors. It produces *open trace files* (.otf/.otf2) containing the results that can be visualized with e.g. *Vampir*. Before instrumenting with *Score-P*, special filter can be set to specify the tracing if for example not very single detail is supposed to be tracked.
Vampir is capable of open *open trace files* to visualize the performance of a tracked program. It resolves the time evolution of such a program on multiple processors but also display the total accumulated run time for each function call.
Score-P and *Vampir* are used as described in this section to analyze the performance of the simulation using the reference case with parameters as listed in table 1 on multiple processors. The way the simulation is constructed, a large amount of MPI communication is expected. As shown in figure 8, a minimum run time for $n_p = 5$ is observed for the reference case. It is suggested, that the amount of communication is too large relative to the computation time needed when using more processors.
It is expected that the overall calculation time of `System::attraction()` and `System::update()` should accumulate to roughly the same value, no matter how many processors are used. The total number of calculation are still fixed by the number of objects N. The difference is, that more processors will split up the work and do these calculation simultaneously. So while the overall run time T of the calculation is roughly constant, P processors will perform the calculations in roughly $T/(P-1)$ simultaneously (recap that processor 0 is coordinattng). In a pie chart displaying the total time spent in function `System::attraction()` and `System::update()`, these time should be roughly constant.
In contrast to that, the communication of `MPI_Send()` and `MPI_Recv()` should increase

for the use of more processors. So for using more processors, a constant total calculation time and an increasing communication time will lead to an increasing fraction of MPI communication time and a decreasing fraction of calculation time. These results are shown as pie charts in figure 11 to 13. The `MPI_Init()` is only shown in figure 12 because its fraction is too small to be displayed by *Vampir* for $n_p = 2, 10$ in figure 11 and 13.

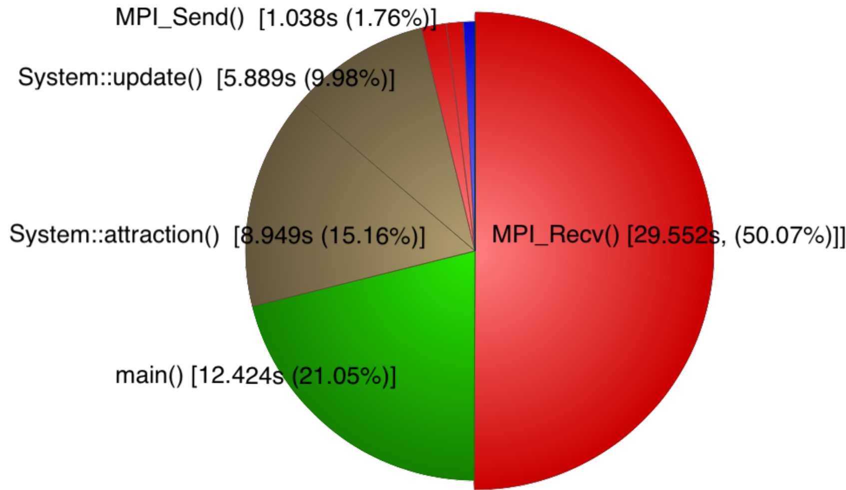


Figure 11: Fraction in [%] and accumulated time in [s] of run time of the functions for the reference case with parameters listed in table 1 on $n_p = 2$ processors.

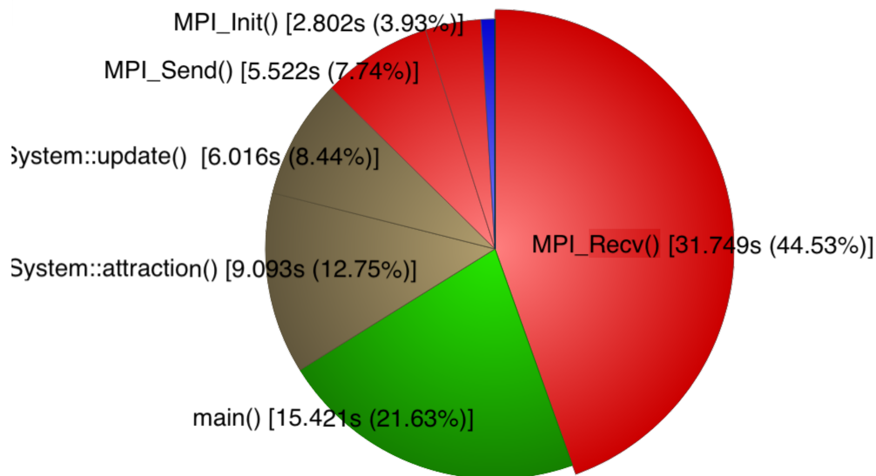


Figure 12: Fraction in [%] and accumulated time in [s] of run time of the functions for the reference case with parameters listed in table 1 on $n_p = 5$ processors.

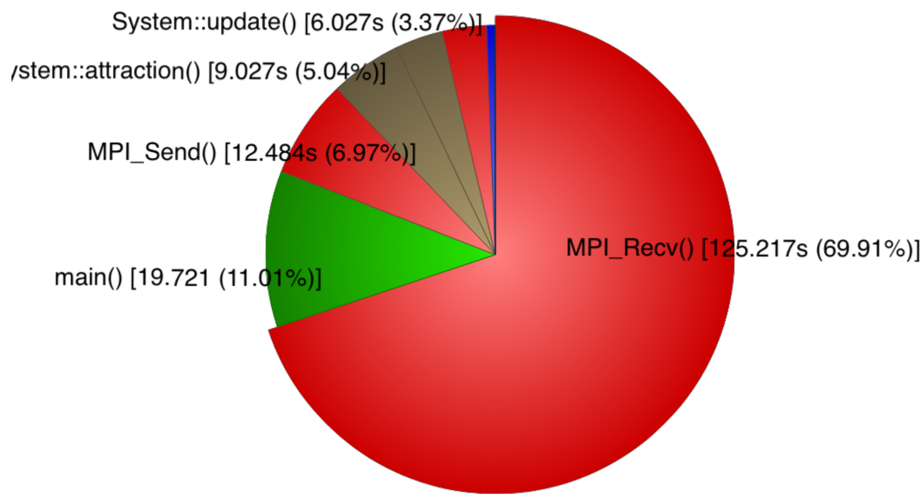


Figure 13: Fraction in [%] and accumulated time in [s] of run time of the functions for the reference case with parameters listed in table 1 on $n_p = 10$ processors.

6 Improvements

This section is suppose to discuss some improvements that has been discovered during implementing and especially during analyzing the performance.
At first, the implementation was done such that it does not restrict to constant masses for all the objects. Especially the rocket actually looses lots of mass on its flight but in the end, this implementation was not done. Thus, there is no need for sending the array with the masses $\vec{m}$ to the processors in each time step. Changing masses still can be added to the code but the focus of this work is set to analyzing and understanding MPI and not necessarily the physical details.
The other improvement that should significantly increase the performance is the amount of data that is sent: each processor p performs calculation of objects with index $i \in [(p-1) \cdot N/(P-1),\ p \cdot N/(P-1)]$, so for $N/(P-1)$ objects. In every time step, all the positions get sent to all the processors. This is necessary because for calculating the attraction of any body, the positions of all other objects are needed. But for processor p performing the Euler-step on its $N/(P-1)$ current position $x_{i(p)}$, velocity $v_{i(p)}$ and acceleration $a_{i(p)}$, only the time step Δt and the data of these arrays with their $N/(P-1)$ are needed. The program yet still sends all $2N$ entries of each array to all the processors, also if they only operate on the objects with the indices as derived in (24). Also instead of sending all the arrays with in a separate `MPI_Send()` call, their shorter array versions could be packet into one array and send via one send-call. This would reduce the number of MPI send- and receive calls and also the amount of data sent to each processor.

Implementing theses changes should improve performance but only to some limited extend. Generally, the method of using MPI for the N-body might not even be a good choice after all. The increasing message parsing for large data will always limit the performance. A shared memory approach should get rid of this problem and would have been the much better choice in terms of performance. As mentioned, this is of course problem specific for the N-body problem.

7 Conclusion

MPI parallelization has been used to parallelize computations in a N-body simulation to increase the performance. After deriving all the necessary physics and getting into initialization details, a sequential simulation performing Euler-steps is set up. Visualization is performed using the *SDL* library and working simulation of a N-Body system with a rocket performing Swing-by maneuver is presented.
The code has been inspected and parts where calculations can be calculated simultaneously are identified. A parallelization plan is set up and a detailed design is constructed as shown in figure 5. With implementing the parallelization using MPI, the performance of the program is analyzed and method specific expectation are drawn. The run time is investigated for varying the number of processors, the number of total time steps and the number of total objects in the simulation. Furhter, a detailed look at the fractions of run time for the specific calculation and communication functions is taken. These results are used in the last section 6 to set up improvements and a critical questioning of the method itself.
The results of the analysis show exactly the expected behavior. As discussed in section 5.2, an efficient balance between workload distribution and MPI communications need to be found, which is caused by the nature of how MPI is constructed as a message parsing parallelization scheme. For simulations with only few calculations gained from a overall large amount of data, MPI is not efficient. For too many processors, the run time generally increases due to the increasing messaging between processors.
The section 6 improvements lists some slight changes that can be made to increase the overall performance but should not be able change the general behavior of the run time for scaling the simulation and running on multiple processors.
As seen in section 5.2, MPI might not be the best choice due to the large fraction of communication. Thus, for the N-body simulation, a parallelization scheme using a shared memory approach should perform much better.
An advantage of using MPI in the N-body system is of course an academic one: having to worry about why the analysis looks the way it does, presents a much better understanding on how MPI works. Gaining a better performance for the use of more processors would have not encouraged to take such a detailed look and spend such detailed thoughts. Therefore a lesson not only how to gain performance but also which parallelization scheme is even capable of improving my simulation, is learned.

References

[1] Astrokramkiste. *Swing-by*. URL: https://astrokramkiste.de/swing-by (**urlseen** 27/09/2022).

[2] Wolfgang Demtröder. *Experimentalphysik 1 - Mechanik und Wärme*. **volume** 5. Springer, 2008.

[3] Joel H. Ferziger **and** Milovan Peric. *Computational Methods for Fluid Dynamics*. **volume** 2. Springer, 1997.

[4] GWT-TUD GmbH. *Vampir*. 2005. URL: https://vampir.eu (**urlseen** 28/09/2022).

[5] Sam Lantinga **and** SDL-community. *Simple DirectMedia Layer*. URL: https://www.libsdl.org (**urlseen** 27/09/2022).

[6] Bernd Leitenberger. *Swing-by*. URL: https://www.bernd-leitenberger.de/swingby.shtml (**urlseen** 27/09/2022).

[7] Github Numerical Methods for Engineers. *Euler's method*. URL: http://lrhgit.github.io/tkt4140/allfiles/digital_compendium/._main010.html (**urlseen** 29/09/2022).

[8] Roman R Rafikov. "Properties of gravitoturbulent accretion disks". **in** *The Astrophysical Journal*: 704.1 (2009), **page** 281.

[9] Sciencetopia. *Escape Velocity: Definition, Formula, Derivation*. URL: https://www.sciencetopia.net/physics/escape-velocity (**urlseen** 28/09/2022).

[10] German BMBF project SILC **and** US DOE project PRIMA. *Score-P*. URL: https://www.vi-hps.org/projects/score-p/ (**urlseen** 28/09/2022).

[11] Wikipedia. *Escape Velocity: Definition, Formula, Derivation*. URL: https://de.wikipedia.org/wiki/Explizites_Euler-Verfahren (**urlseen** 29/09/2022).

[12] Wikipedia. *Swing-by*. URL: https://de.wikipedia.org/wiki/Swing-by#/media/Datei:Swingbyvxy.png (**urlseen** 27/09/2022).

[13] Wikipedia. *Swing-by*. URL: https://de.wikipedia.org/wiki/Swing-by#:~:text=Der%5C%20englische%5C%20Begriff%5C%20Swing-by,(etwa%5C%20einem%5C%20Planeten)%5C%20vorbeifliegt. (**urlseen** 27/09/2022).

8 Appendix

Work distribution

The following list provides an overview over the main areas of responsibility for the different group members. With this beeing said, we encountered various problems and challenges we had to overcome and therefore were working together on most of the code and problems by debugging and correcting the code of each other, additionally to longer research phases and problem discussions.
Yannik Feldner:

- Physical research and funcamentals
- Swing by and asteroid belt: initialization of all bodies
- sequential implementation with attraction and Euler-step

Aaron Nagel:

- parallelization design
- MPI implementaion
- Analysis and performance

8.1 MPI implementation: main function

```
1
2
3 int main(int argc, char *argv[]) {
4
5     // ——— MPI Routine ——— //
6     // unique rank is asined to each process in a communicator
7     int rank;
8     // Total number of ranks
9     int num_proc;
10    // the machine we are on
11    char name[80];
12    // length of machine name
13    int length;
14    // Initialize the MPI execution environment
15    MPI_Init(&argc, &argv);
16    // get this process' rank (process within a communicator)
17    // MPI_COMM_WORLD is the default communicator
```

```
18     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
19     // get the total number of ranks in this communicator
20     MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
21     // get the name of the processor
22     // implementaion specific (may be get hostname, username or systeminfo)
23     MPI_Get_processor_name(name, &length);
24 
25     // declare system on every proc. for access to sys methods
26     System sys;
27     sys.create();
28     sys.init();
29     sys.belt_init();
30 
31     int N = sys.N;
32 
33     // allocate storage for state-vector needed for sending
34     // and receiving data via MPI_:
35     double *y = new double[2*N];
36     double *v = new double[2*N];
37     double *a = new double[2*N];
38     double *pxl = new double[2*N];
39     double *mass = new double[N];
40 
41     // rank zero coordinates
42     if(rank == 0){
43 
44     // only rank zero collect all data to perform visualization:
45     Anim anim;
46     anim.init();
47     anim.create();
48 
49     // start time. Movements start when t = 0.0
50     double t = 0.0;
51 
52     int step = 0;
53 
54     //Event loop
55     while(!anim.done){
56         //Render display
57         anim.doRender(anim.renderer, sys.Objects, sys.timestep);
58 
59         // time:
60         t += delta_t;
```

```
61
62          // start when time hits zero: t = 0:
63          if(t > 0){
64              // fill state-vectors:
65              for(int i = 0; i < N; i++){
66                  y[2*i] = sys.Objects[i].x;
67                  y[2*i + 1] = sys.Objects[i].y;
68                  v[2*i] = sys.Objects[i].u;
69                  v[2*i + 1] = sys.Objects[i].v;
70                  a[2*i] = sys.Objects[i].ax;
71                  a[2*i + 1] = sys.Objects[i].ay;
72                  mass[i] = sys.Objects[i].mass;
73              }
74              // ------- calc. attraction ---------- //
75
76              // every proc. needs all positions and masses:
77              for(int p = 1; p < num_proc; p++){
78                  MPI_Send(y, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
79                  MPI_Send(mass, N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
80              }
81              // receive all calculated accelerations for the timestep:
82              // start receiving in order starting from proc. 1:
83              for(int p = 1; p < num_proc; p++){
84                  MPI_Recv(a, 2*N, MPI_DOUBLE, p, 0,
85                      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
86                  // write this into forces of the Obj. N/(num_proc-1) Obj.:
87                  for(int i = (p-1)*N/(num_proc-1); i < p*N/(num_proc-1); i
   ++){
88                      sys.Objects[i].ax = a[2*i];
89                      sys.Objects[i].ay = a[2*i+1];
90                  }
91              }
92              // -------- update all positions: ------------ //
93              // distribute data
94              for(int p = 1; p < num_proc; p++){
95                  // send all entries to each proc.:
96                  MPI_Send(y, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
97                  MPI_Send(v, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
98                  MPI_Send(a, 2*N, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
99              }
100             // receive all updated pos and vel and pxl for the timestep:
101             for(int p = 1; p < num_proc; p++){
102                 MPI_Recv(y, 2*N, MPI_DOUBLE, p, 0,
```

```
103                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
104                 MPI_Recv(v, 2*N, MPI_DOUBLE, p, 0,
105                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
106                 MPI_Recv(a, 2*N, MPI_DOUBLE, p, 0,
107                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
108                 MPI_Recv(pxl, 2*N, MPI_DOUBLE, p, 0,
109                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
110                 // only sort from correct indices !
111                 for(int i = (p-1)*N/(num_proc-1); i < p*N/(num_proc-1); i
    ++){
112                     sys.Objects[i].x = y[2*i];
113                     sys.Objects[i].y = y[2*i+1];
114                     sys.Objects[i].u = v[2*i];
115                     sys.Objects[i].v = v[2*i+1];
116                     sys.Objects[i].ax = a[2*i];
117                     sys.Objects[i].ay = a[2*i+1];
118                     sys.Objects[i].pxl_x = pxl[2*i];
119                     sys.Objects[i].pxl_y = pxl[2*i+1];
120                 }
121             }
122             // ------- finished updating positions ------- //
123             // write orbit:
124             for(int i = 0; i < sys.N; i++){
125                 sys.write_orbit(&sys.Objects[i]);
126             }
127 
128         }
129         // for testing, stop after MAX_SETPS iterations:
130         if(t > MAX_SETPS*delta_t){
131             anim.done = 1;
132         }
133         // send status to all procs.
134         for(int p = 1; p < num_proc; p++){
135             MPI_Send(&anim.done, 1, MPI_INT, p, 0, MPI_COMM_WORLD);
136         }
137         step++;;
138         sys.timestep++;;
139     }
140     // finished sim.
141     anim.destroy();
142 
143     }
144 
```

```
145
146     // all procs. which are not zero (cause zeros coordinates sim)
    calculate:
147     if(rank != 0){
148
149     // send and receive as long as rank zero does not tell sim is done:
150     int done = 0;
151     while(!done){
152
153         // ---------- update accel: ---------- //
154         MPI_Recv(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
155         MPI_Recv(mass, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
156         // sort into system array
157         for(int i = 0; i < N; i++){
158             sys.Objects[i].x = y[2*i];
159             sys.Objects[i].y = y[2*i+1];
160             sys.Objects[i].mass = mass[i];
161         }
162
163         // update accelerations using sys.attratcion Routine:
164         // only update accelerations of the N/(num_proc-1) Objects that the
165         // current processor proc. = rank is in charge:
166         for(int i = (rank-1)*N/(num_proc-1); i < rank*N/(num_proc-1); i++){
167             sys.attraction(&sys.Objects[i]);
168             // write calculated accelerations of Object i in state vector:
169             a[2*i] = sys.Objects[i].ax;
170             a[2*i + 1] = sys.Objects[i].ay;
171         }
172
173         MPI_Send(a, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
174
175         // ---------- update pos. ---------- //
176         // receive data:
177         MPI_Recv(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
178         MPI_Recv(v, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
179         MPI_Recv(a, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
180
181         // sort into system array:
```

```
182         for(int i = 0; i < N; i++){
183             sys.Objects[i].x = y[2*i];
184             sys.Objects[i].y = y[2*i+1];
185             sys.Objects[i].u = v[2*i];
186             sys.Objects[i].v = v[2*i+1];
187             sys.Objects[i].ax = a[2*i];
188             sys.Objects[i].ay = a[2*i+1];
189         }
190         // >>> Euler Schritt
191         // perform Euler step only for the N/(num_proc-1) Objects that the
192         // current processor proc. = rank is in charge:
193         for(int i = (rank-1)*N/(num_proc-1); i < rank*N/(num_proc-1); i++){
194             sys.update(&sys.Objects[i], delta_t);
195             // write calculated accelerations of Object i in state vector:
196             y[2*i] = sys.Objects[i].x;
197             y[2*i + 1] = sys.Objects[i].y;
198             v[2*i] = sys.Objects[i].u;
199             v[2*i + 1] = sys.Objects[i].v;
200             a[2*i] = sys.Objects[i].ax;
201             a[2*i + 1] = sys.Objects[i].ay;
202             pxl[2*i] = sys.Objects[i].pxl_x;
203             pxl[2*i + 1] = sys.Objects[i].pxl_y;
204         }
205 
206         // Send results to proc. 0 for visualization and further
    coordinatiion:
207         MPI_Send(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
208         MPI_Send(v, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
209         MPI_Send(a, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
210         MPI_Send(pxl, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
211 
212         // recv status from 0:
213         MPI_Recv(&done, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
214     }
215     }
216 
217     // free storage of sys which is allocated on every processor,
218     // so every proc. need to free (no if rank ... necessary):
219     sys.destroy();
220 
221     // free strage of state vercors
222     free(y);
```

```
223     free(v);
224     free(a);
225     free(pxl);
226     free(mass);
227 
228     // Terminate MPI execution environment
229     MPI_Finalize();
230 
231     return 0;
232 }
```

Listing 9: MPI Implementaion