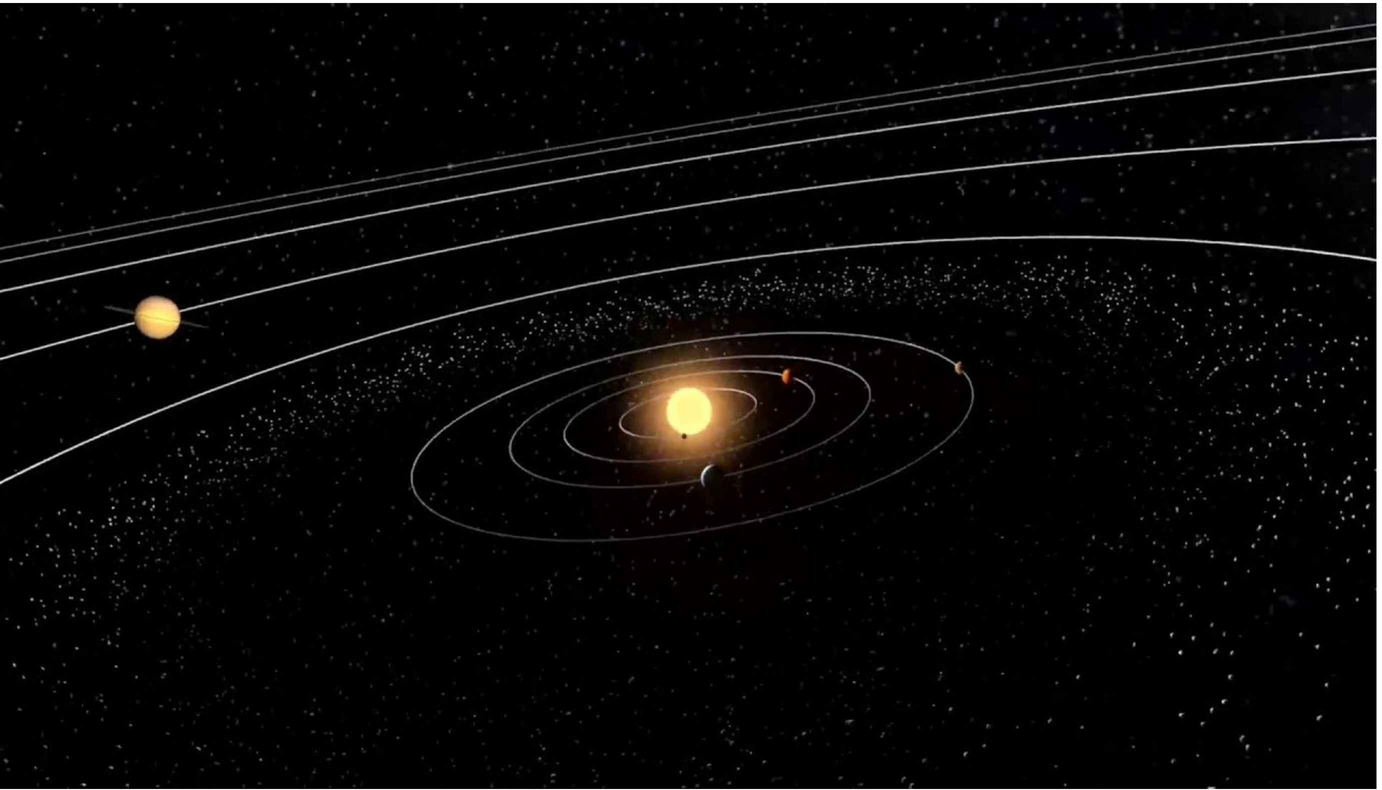
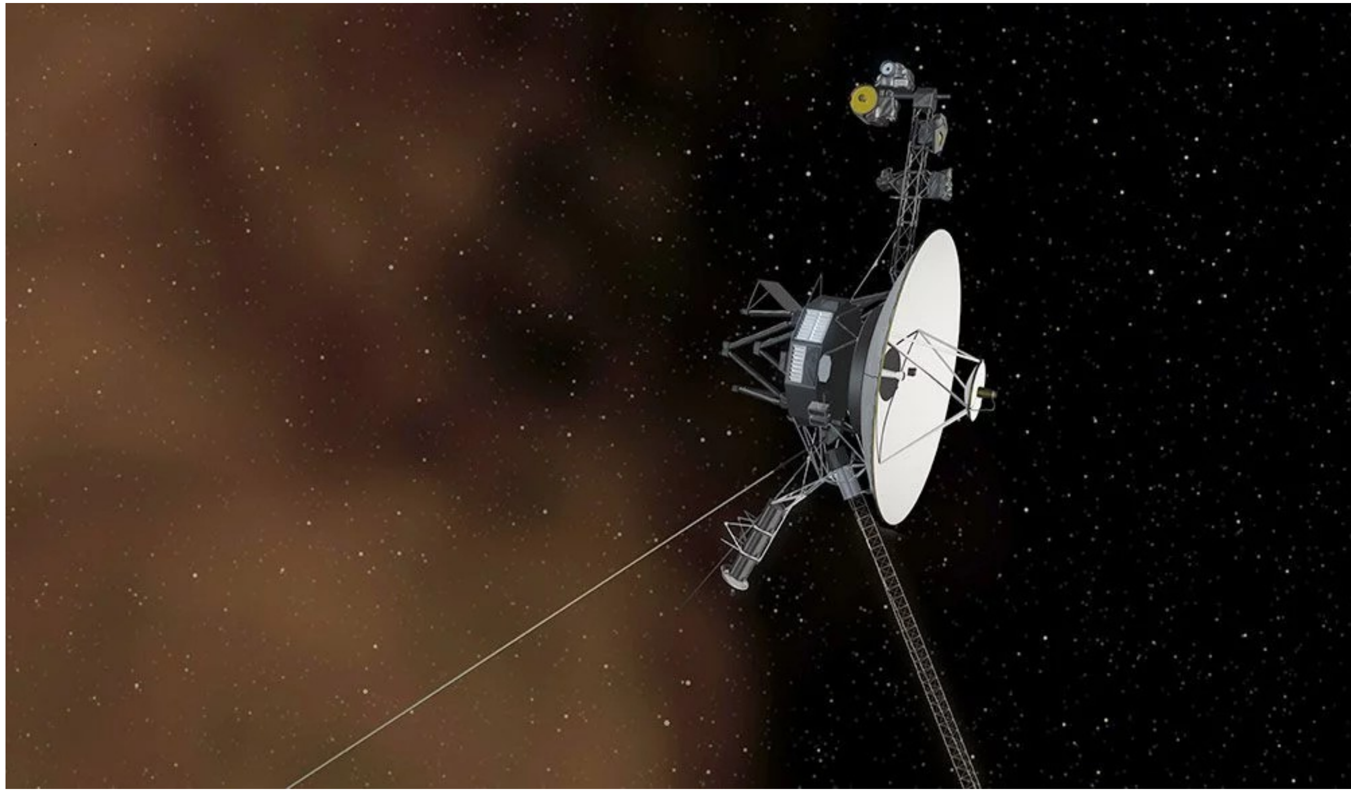
GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Simulation of an N-Body System and Swing-by Maneuver

Aaron Nagel – aaron.nagel@stud.uni-goettingen.de
Yannik Feldner – yannik.feldner@stud.uni-goettingen.de
Supervisor: Jack Ogaja

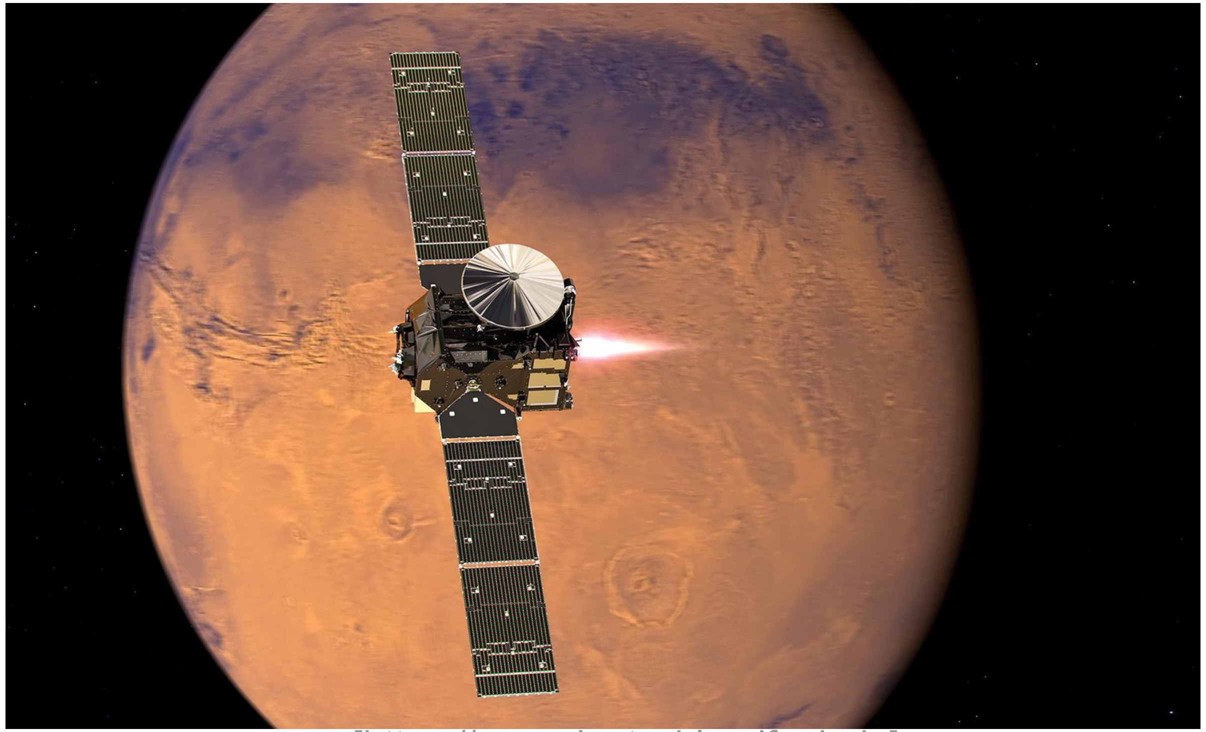
13.09.2022





Presentation Outline

- Motivation
- Problem description: N-Body Solarsystem and Physics
- Approach
  - Numerical Setup and Initialization
  - Sequentiel Implementation
  - Parallel Implementation
- Performance analysis
- Conclusion



[https://www.deutschlandfunk.de]

Motivation

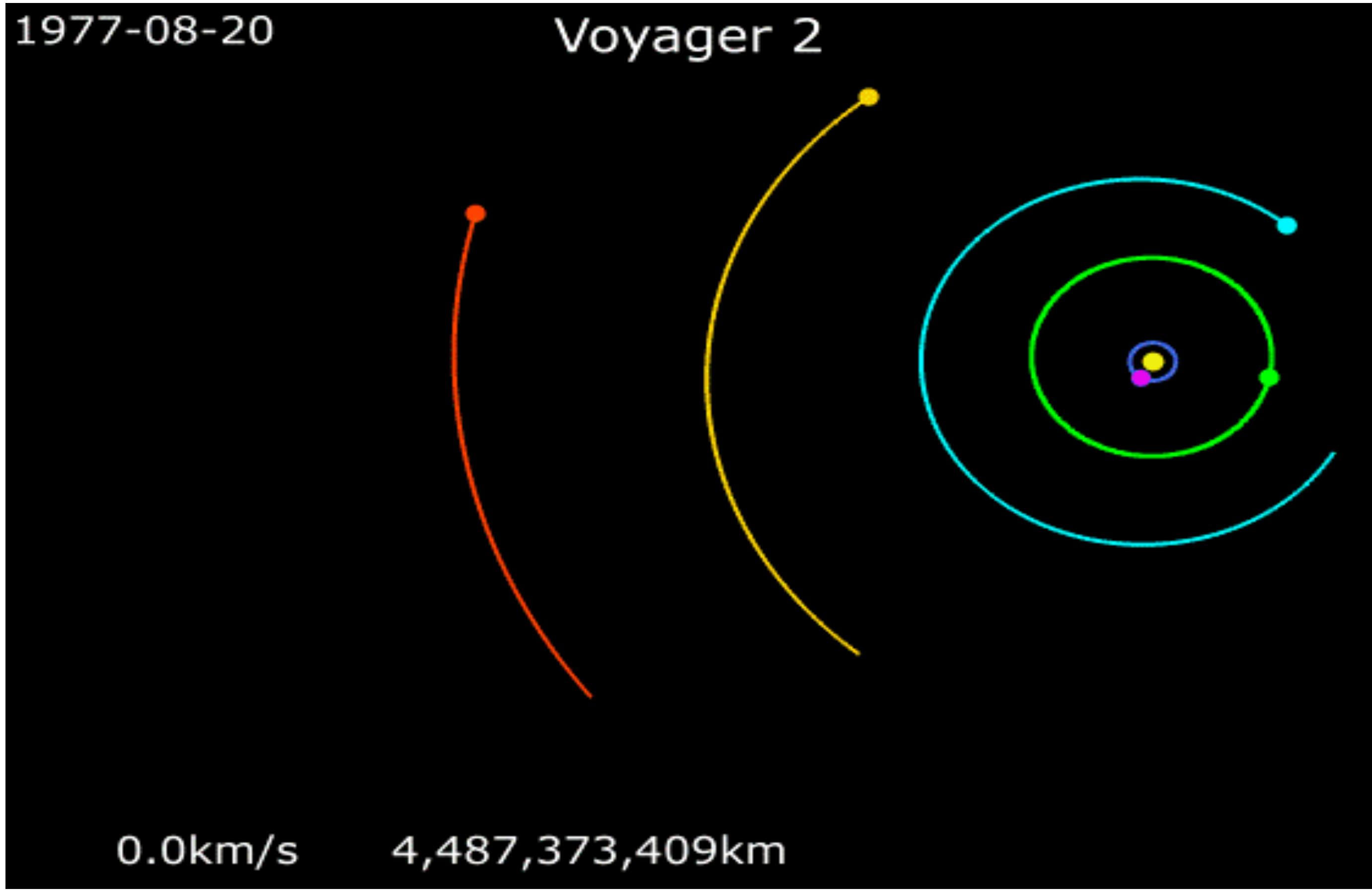
- project Idea is inspired by the Voyager program [VP]
- one of the most successful programs conducted by NASA
- goal of the VP:
    - Observations and studies of the outer planets of our solar system
    - Observations of the interstellar space

  ↳ needs to reach the Solar escape velocity in order to leave the heliosphere

- periodic alignment of the outer planets in 1970s as foundation

↳ Our goal: perform a Grand Tour maneuver with gravity assist similar to the VP

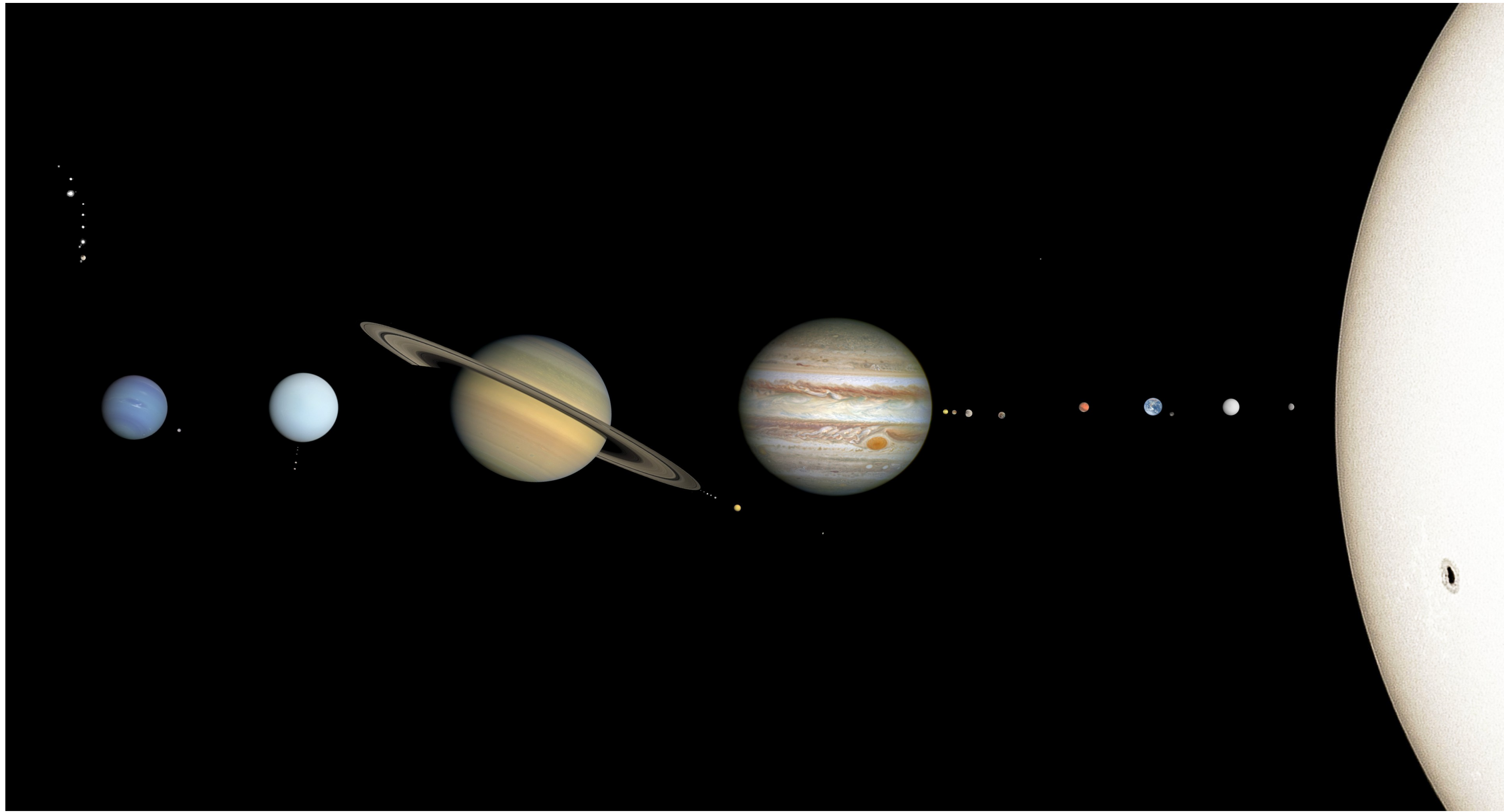
[https://en.wikipedia.org/wiki/Gravity_assist]



The Problem – N-Body Solor System

- Simulation of a standard model solar system
  - Sun: $M_{sun} >> m_N$
  - „Terrestrial“ or rocky bodies
  - Gas Giants
  - Ice Giants
- problem is given by the N-Body problem of physics

  ↳ gravitational interaction of the N-Bodies with each other



The Problem – Physics

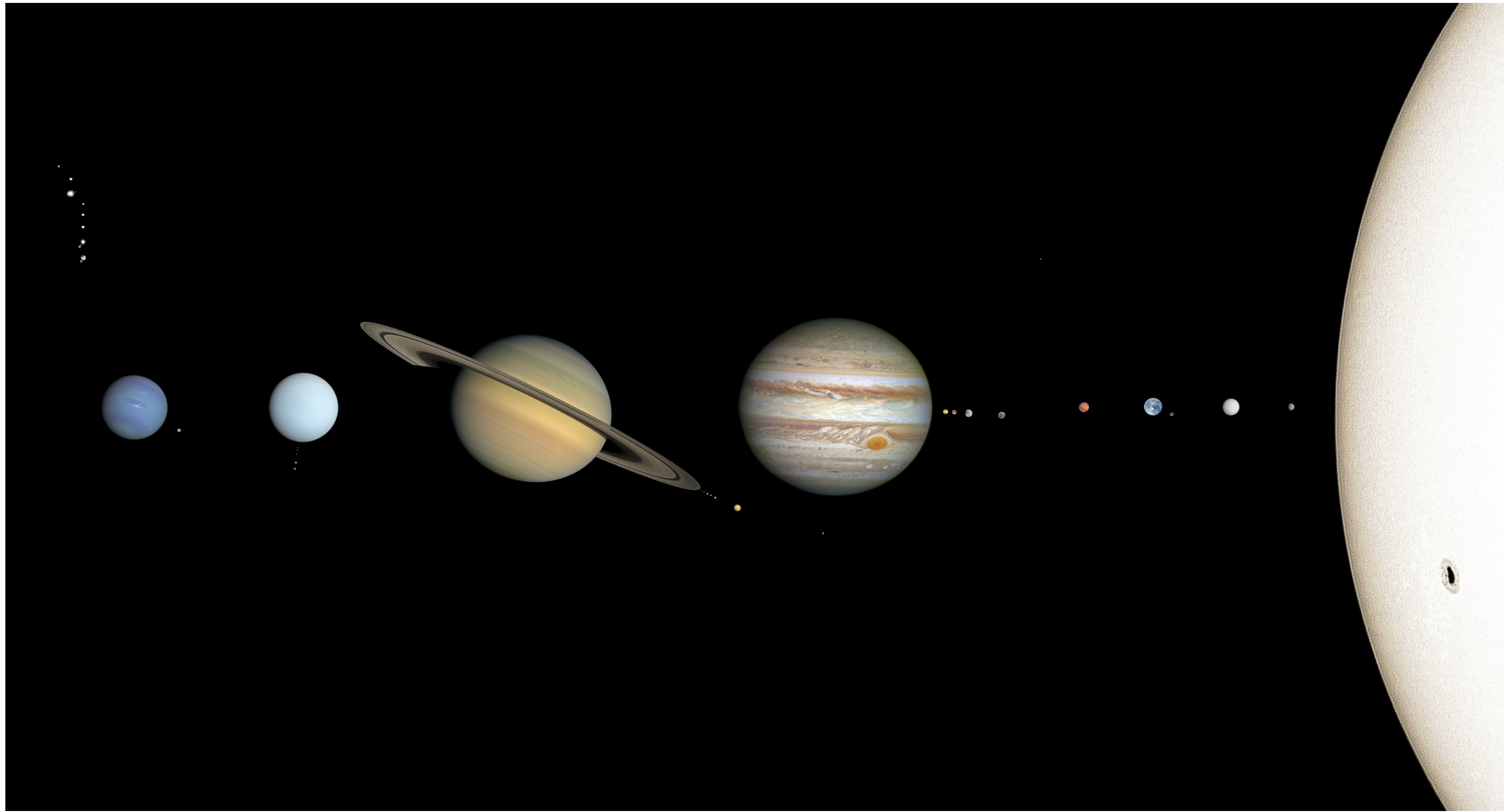
- gravitational force between two bodies:

$$F_{ij} = \frac{G m_i m_j}{d^2} \quad \text{with} \quad d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 \cancel{+ (z_i - z_j)^2}}$$

- using Newton's second law: $F = m \cdot a = m_i \cdot \frac{\mathrm{d}^2 r_i}{\mathrm{d}t^2}$

- interaction of the N-bodies as:

$$m_i \frac{\mathrm{d}^2 r_i}{\mathrm{d}t^2} = \sum_{i=1, i \neq j}^{N} \frac{G m_i m_j}{d^2}$$



- N^2-interactions

The Problem – Physics

- Interaction of the N-bodies as:

$$m_i \frac{\mathrm{d}^2 r_i}{\mathrm{d}t^2} = \sum_{i=1, i \neq j}^{N} \frac{G m_i m_j}{d^2}$$

- use Lagrangian of the system:

$$\mathbf{L} = \frac{1}{2} \sum_{i=1, i \neq j}^{N} m_i \cdot ||\dot{r}_i||^2 - \sum_{i=1, i \neq j}^{N} \frac{G m_i m_j}{d^2}$$

using the momentum: $p_i = m_i \cdot \frac{\mathrm{d}r_i}{\mathrm{d}t}$

The Problem – Physics

- Interaction of the N-bodies as:

$$m_i \frac{\mathrm{d}^2 r_i}{\mathrm{d}t^2} = \sum_{i=1, i \neq j}^{N} \frac{G m_i m_j}{d^2}$$

- use Lagrangian of the system:

$$\mathbf{L} = \underbrace{\frac{1}{2} \sum_{i=1, i \neq j}^{N} \frac{||p_i||^2}{m_i}}_{E_{kin} = T} - \underbrace{\sum_{i=1, i \neq j}^{N} \frac{G m_i m_j}{d^2}}_{E_{pot} = U}$$

$E_{pot} = U \longrightarrow H = T + U$

The Problem – Physics

- Interaction of the N-bodies as:

$$m_i \frac{\mathrm{d}^2 r_i}{\mathrm{d}t^2} = \sum_{i=1, i \neq j}^{N} \frac{G m_i m_j}{d^2}$$

- use Lagrangian of the system:

$$\mathbf{L} = \underbrace{\frac{1}{2} \sum_{i=1, i \neq j}^{N} \frac{||p_i||^2}{m_i}}_{E_{kin} = T} - \underbrace{\sum_{i=1, i \neq j}^{N} \frac{G m_i m_j}{d^2}}_{E_{pot} = U}$$

E_{kin} = T E_{pot} = U → H = T + U

- Hamiltons equations of motion:

$$\frac{\mathrm{d}r_i}{\mathrm{d}t} = \frac{\partial H}{\partial p_i} \quad \text{and} \quad \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial H}{\partial r_i}$$

→ 4N differential equations

Approach – Numerical Setup

- code is written in C++
- Initialization of System Parameters

  ↳ Nondimensionalization

```
// PHYSICAL VALUES
const int Nbody = 11;
const int NAstroids = 200;
const int Ntot = Nbody + NAstroids;
const double AU = 1.496e11;              // Astronomical unit
const double a = 3.169e7;
const double M_earth = 5.972e24;
const double M_sun = 332948.6*M_earth;

const double G = (6.674e-11)*M_earth*a*a/(AU*AU*AU);     // Gravitational constant
// const double G = 6.674e-11;     // Gravitational constant
// const double SCALE = 200.0/AU;              // 1AU = 100 Pixels
const double TIMESTEP = 3600.0*24.0;         // Timestep = 1 day
```

Astronomical Unit [AU]: distance earth-sun

Anomalistic year: Time in which earth is crossing the perihelion (nearest point so the sun)

Earth mass in [kg]

Gravitational constant in terms of Earth masses, anomalistic years and Astronomical Units

- Therefore earth velocity is given by 2π
- Object parameters are scaled with earth velocities, earth mass and AU

Approach – Initialization of Objects

- every object is defined with:
  - positions
  - velocities
  - accelerations
  - radius and animation radius
  - mass
  - pixel corresponding to the scaled position
  - orbit array
  - type

```
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;


    /* Creating Array for the Orbit of the Object */
    const int orbit_size = 4000;
    double orbit_x[4000]; // array with size 2000 to draw the trajectory [x]
    double orbit_y[4000]; // array with size 2000 to draw the trajectory [y]




    void create();
    void init(const char *name, double x0, double y0, double u0,
        double v0, double r, double m, const int *col, int anim_r, Object_class init_type);


    void destroy();

};
```

Approach – Initialization of Objects

- initialization by specifying desired parameters or initialization function:

```
for(int i = Np; i < Na; i++){


    //create random angle:
    theta = rand_val(0, 360);

    //create random dist:
    dist = mean_dist + rand_val(-delta_dist, delta_dist);

    //create x and y position for dist+angle:
    x = sin(theta)*dist;
    y = cos(theta)*dist;

    //create corresponding mean velocity:
    mean_vel = sqrt((G*M_sun/M_earth)/(dist));

    //add noise on mean velocity:
    vel = mean_vel + rand_val(-delta_vel, delta_vel);

    //create x and y velocities:
    vel_x = cos(theta)*vel;
    vel_y = sin(theta)*vel;

    //create random mass:
    mass = mean_m + rand_val(-delta_m, delta_m);

    //initialize object:
    Objects[i].init("AST", x, y, -vel_x, vel_y, 0.0, 1.0, WHITE, 2, ASTROID);
}
```

```
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;


    /* Creating Array for the Orbit of the Object */
    const int orbit_size = 4000;
    double orbit_x[4000]; // array with size 2000 to draw the trajectory [x]
    double orbit_y[4000]; // array with size 2000 to draw the trajectory [y]




    void create();
    void init(const char *name, double x0, double y0, double u0,
        double v0, double r, double m, const int *col, int anim_r, Object_class init_type);


    void destroy();

};
```

Approach – Initialization of Objects

- initialization by specifying desired parameters or initialization function:

Orbital velocity: $v_o \approx \sqrt{\frac{GM}{r}}$

```
    for(int i = Np; i < Na; i++){


        //create random angle:
        theta = rand_val(0, 360);

        //create random dist:
        dist = mean_dist + rand_val(-delta_dist, delta_dist);

        //create x and y position for dist+angle:
        x = sin(theta)*dist;
        y = cos(theta)*dist;

        //create corresponding mean velocity:
        mean_vel = sqrt((G*M_sun/M_earth)/(dist));

        //add noise on mean velocity:
        vel = mean_vel + rand_val(-delta_vel, delta_vel);

        //create x and y velocities:
        vel_x = cos(theta)*vel;
        vel_y = sin(theta)*vel;

        //create random mass:
        mass = mean_m + rand_val(-delta_m, delta_m);

        //initialize object:
        Objects[i].init("AST", x, y, -vel_x, vel_y, 0.0, 1.0, WHITE, 2, ASTROID);
    }
```

```
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;


    /* Creating Array for the Orbit of the Object */
    const int orbit_size = 4000;
    double orbit_x[4000]; // array with size 2000 to draw the trajectory [x]
    double orbit_y[4000]; // array with size 2000 to draw the trajectory [y]




    void create();
    void init(const char *name, double x0, double y0, double u0,
        double v0, double r, double m, const int *col, int anim_r, Object_class init_type);


    void destroy();

};
```

Approach – Initialization of Objects

- initialization by specifying desired parameters or initialization function:

for rocket escape velocity: $v_e = \sqrt{\frac{2GM}{r}}$

earth escape velocity: ~ 12 km/s

```
    for(int i = Np; i < Na; i++){


        //create random angle:
        theta = rand_val(0, 360);

        //create random dist:
        dist = mean_dist + rand_val(-delta_dist, delta_dist);

        //create x and y position for dist+angle:
        x = sin(theta)*dist;
        y = cos(theta)*dist;

        //create corresponding mean velocity:
        mean_vel = sqrt((G*M_sun/M_earth)/(dist));

        //add noise on mean velocity:
        vel = mean_vel + rand_val(-delta_vel, delta_vel);

        //create x and y velocities:
        vel_x = cos(theta)*vel;
        vel_y = sin(theta)*vel;

        //create random mass:
        mass = mean_m + rand_val(-delta_m, delta_m);

        //initialize object:
        Objects[i].init("AST", x, y, -vel_x, vel_y, 0.0, 1.0, WHITE, 2, ASTROID);
    }
```

```
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;


    /* Creating Array for the Orbit of the Object */
    const int orbit_size = 4000;
    double orbit_x[4000]; // array with size 2000 to draw the trajectory [x]
    double orbit_y[4000]; // array with size 2000 to draw the trajectory [y]




    void create();
    void init(const char *name, double x0, double y0, double u0,
        double v0, double r, double m, const int *col, int anim_r, Object_class init_type);


    void destroy();

};
```

Sequentiel Approach – Calculate Forces

- first calculate the distances between the Object i $\in$ N and every other object in N
- calculate gravitational force via:

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

- calculate the force in x/y direction
- calculate accelerations via:

$$F = m \cdot a$$

```
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;
```

Sequentiel Approach – Update State

- Euler Scheme for temporal discretization:
  - calculate acceleration of the N-Bodies:

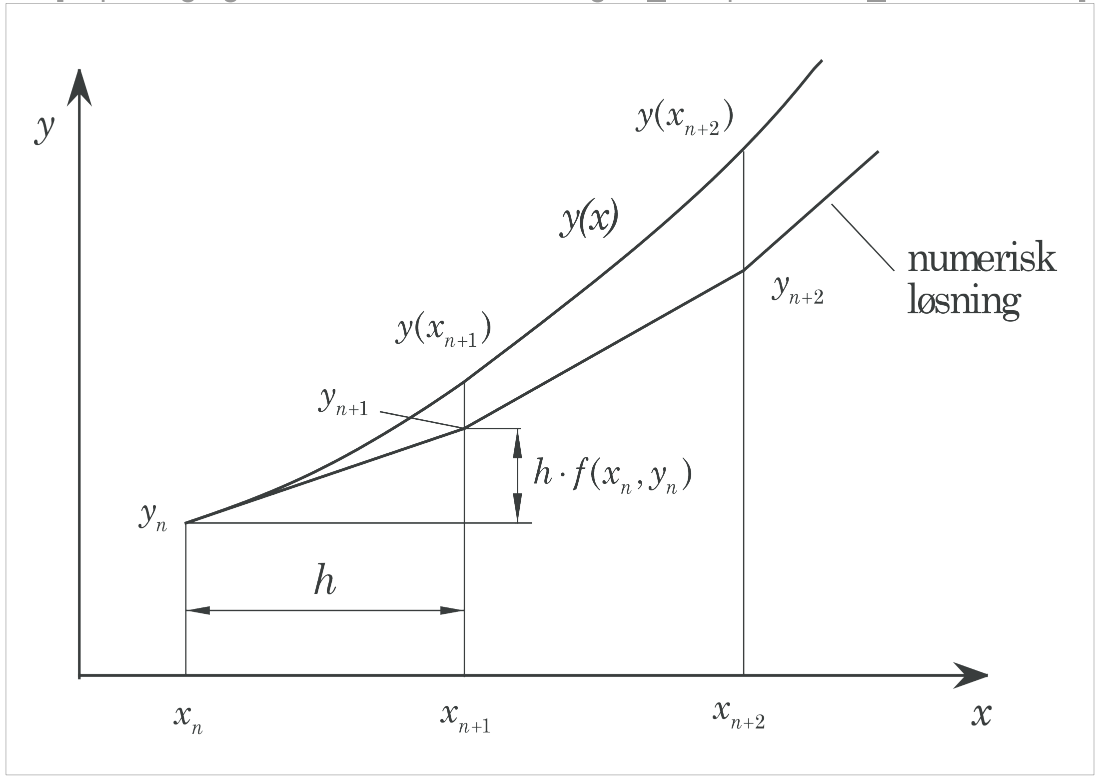
$$y'(x) = f(x, y(x))$$

  - calculate new velocities:

$$y_{n+1}(x) = y_n + h \cdot f(x_n, y_n)$$

  - do the same for positions.

[http://lrhgit.github.io/tkt4140/allfiles/digital_compendium/._main010.html]

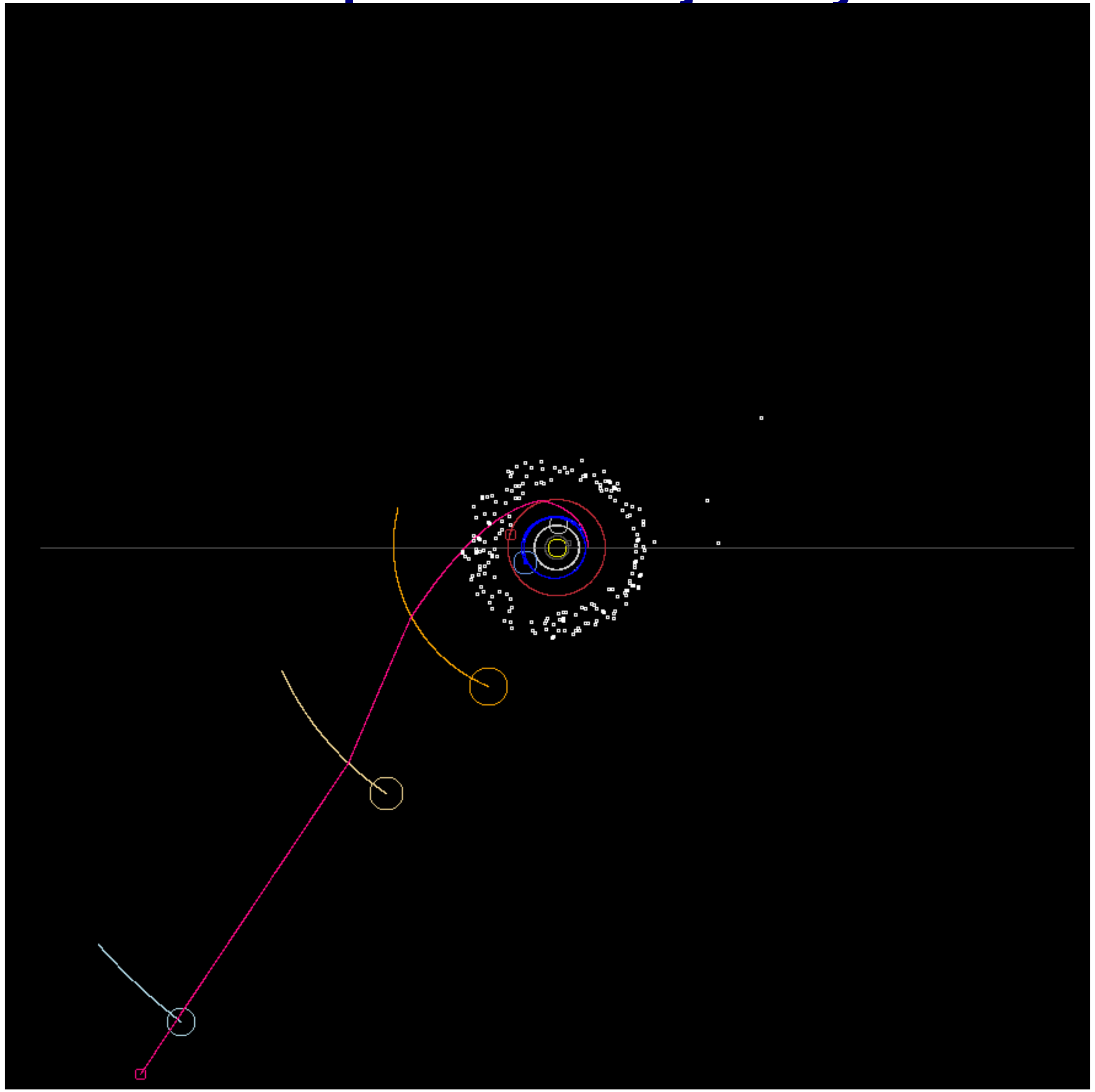



```
void System::update(Object *Plt, double delta_t, double t){

    // ------ 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;
```

Sequentiel Solution

Simulation Video

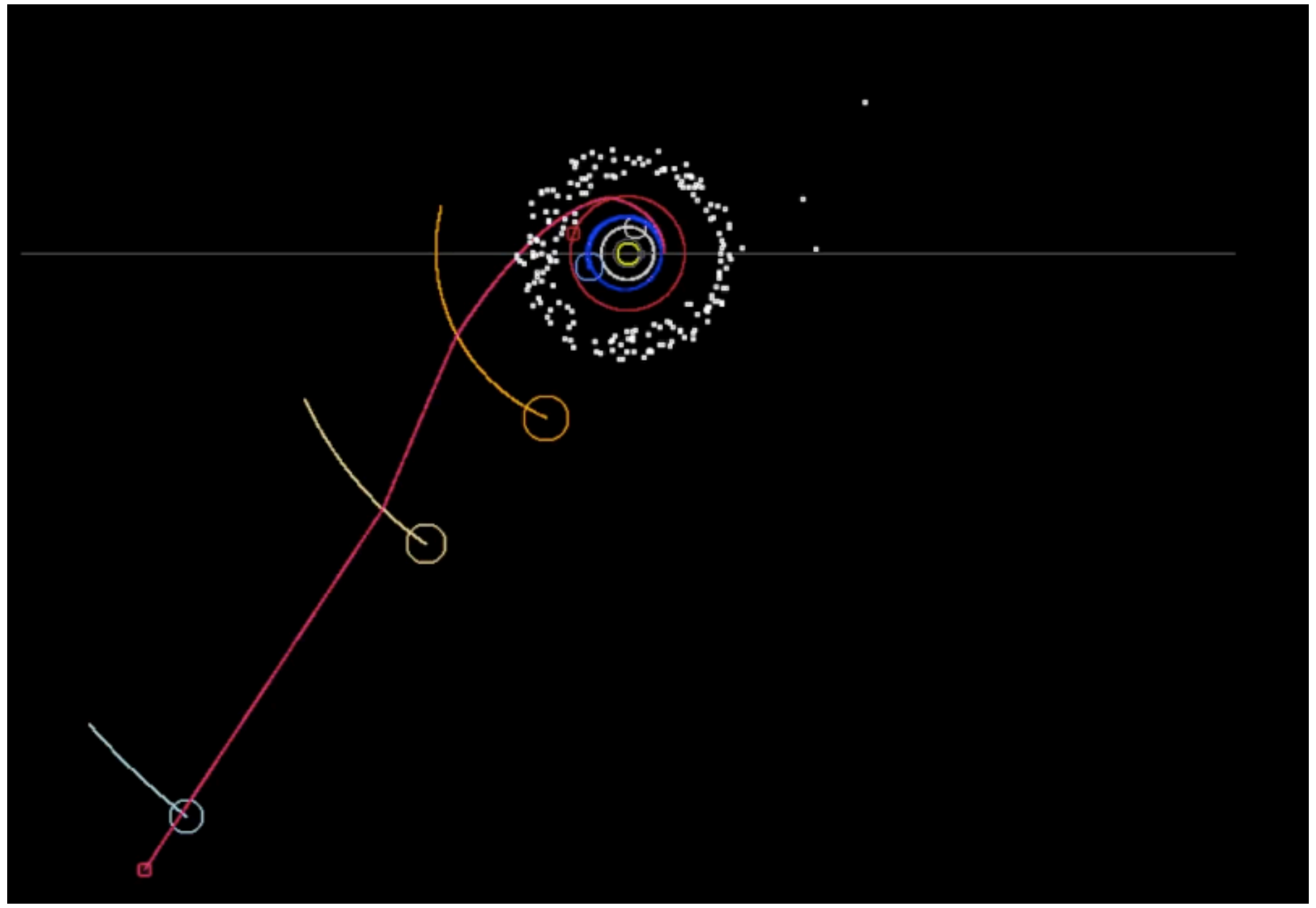
Spacecraft Trajectory



GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

N-Body System with Swing-by

MPI Parallelization and Performance



Aaron Nagel
Yannik Feldner

13.09.2022

Parallelization

Distribution of work: Where to parallelize

Simulation Loop:

Attraction — Calculate forces from all positions — Distribute to P-1 processors to calculate N/P-1 forces

Collect forces from P-1 procs.

update — Update positions — Distribute to P-1 processors to perform Euler step

Parallelization

Initialization

All Ranks:

- Initialize MPI
- Initialize own *System *sys* for allocating memory and usage of **methods**
- Allocating memory for **sending** and **receiving** data

Rank 0: coordination

Rank 1 to P-1: calculation

Simulation Loop:

Parallelization

Distribution of data

Rank 0: coordination

Rank 1 to P-1: calculation

Simulation Loop:

attraction

- Prepare data to send

send

- Receive **all** positions
- Sort positions in *Objects*

for N/(P-1) Objects:

- sys.attraction(*sys.Objects*)
- Prepare data to send

send

- Receive forces
- Sort forces into system of rank 0

Parallelization

Distribution of data

Rank 0: coordination

Rank 1 to P-1: calculation

Simulation Loop:

Update

- Prepare data to send

send

- Receive **all** $\vec{x}$, $\vec{v}$ and $\vec{a}$
- Sort data in *Objects*

for N/(P-1) Objects:

- sys.update(*sys.Objects*)
- Prepare data to send

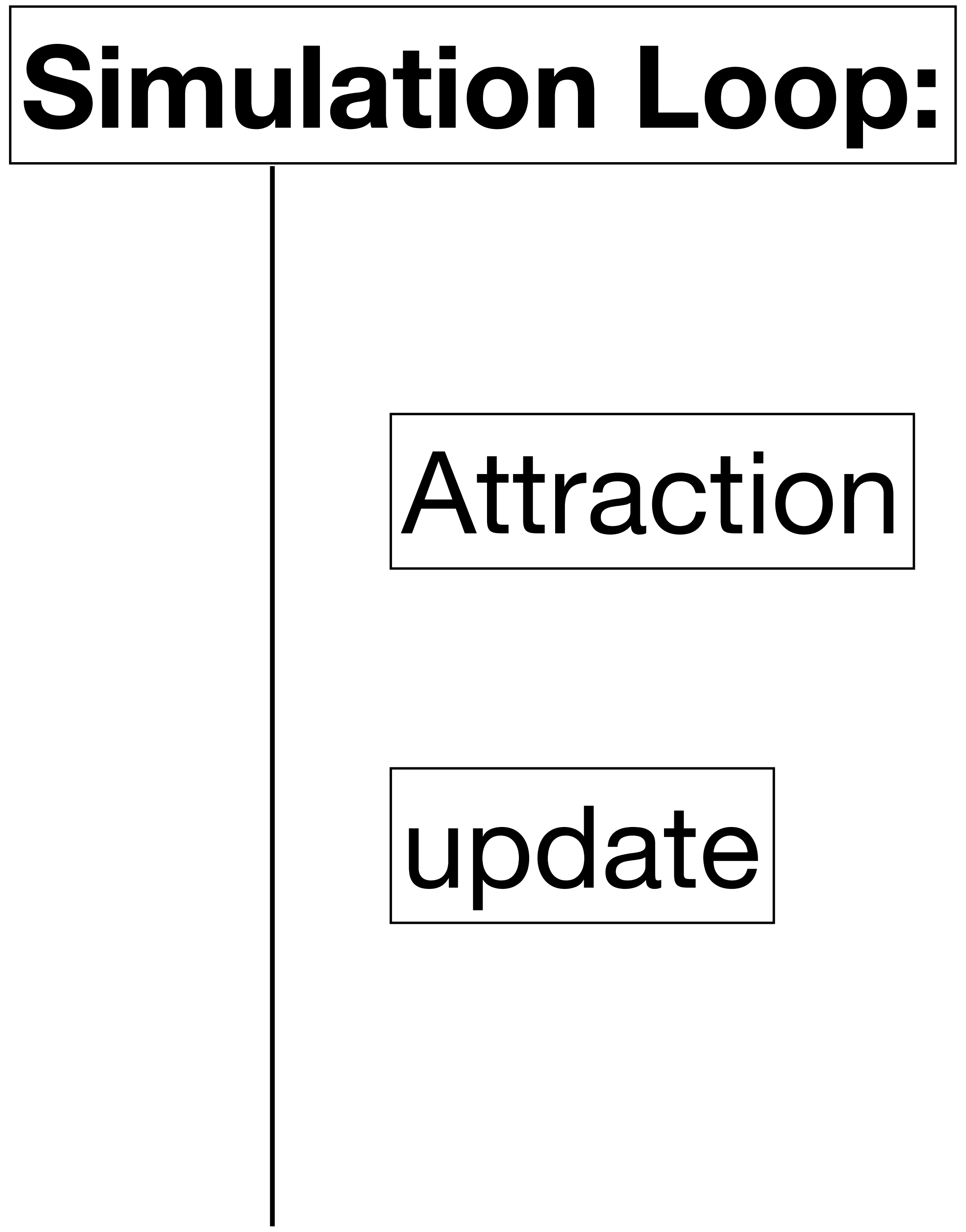
send

- Receive data
- Sort data into system of rank 0

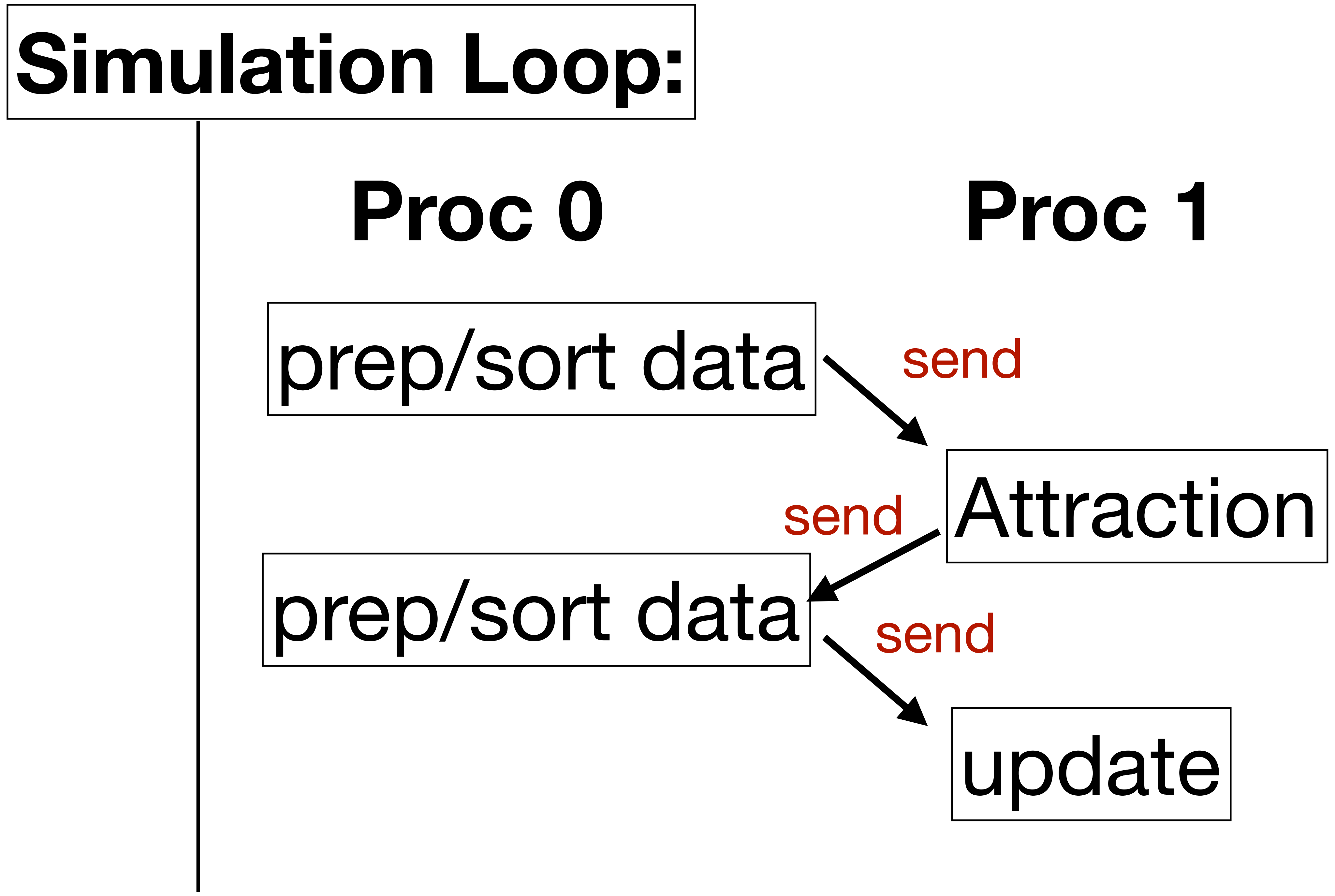
Parallelization

Expectation between sequential and parrallel

Sequential Solution (np = 1)



Parallel Solution on np = 2



Expectation: $t(n_p = 1) < t(n_p = 2)$

Parallelization

Work on processors 1 to P-1

```
int done = 0;
while(!done){

    // ------------ update accel: ----------
    MPI_Recv(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(mass, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // einsortieren von Zustandsvektor in system
    for(int i = 0; i < N; i++){
        sys.Objects[i].x = y[2*i];
        sys.Objects[i].y = y[2*i+1];
        sys.Objects[i].mass = mass[i];
    }

    // update accelerations using sys.attratcion Routine:
    // only update accelerations of the N/(num_proc-1) Objects that the
    // current processor proc. = rank is in charge:
    for(int i = (rank-1)*N/(num_proc-1); i < rank*N/(num_proc-1); i++){
        sys.attraction(&sys.Objects[i]);
        // write calculated accelerations of Object i in state vector:
        a[2*i] = sys.Objects[i].ax;
        a[2*i + 1] = sys.Objects[i].ay;
    }

    MPI_Send(a, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

```

Rank 1 to P-1: calculation

- Receive **all** positions
- Sort positions in *Objects*

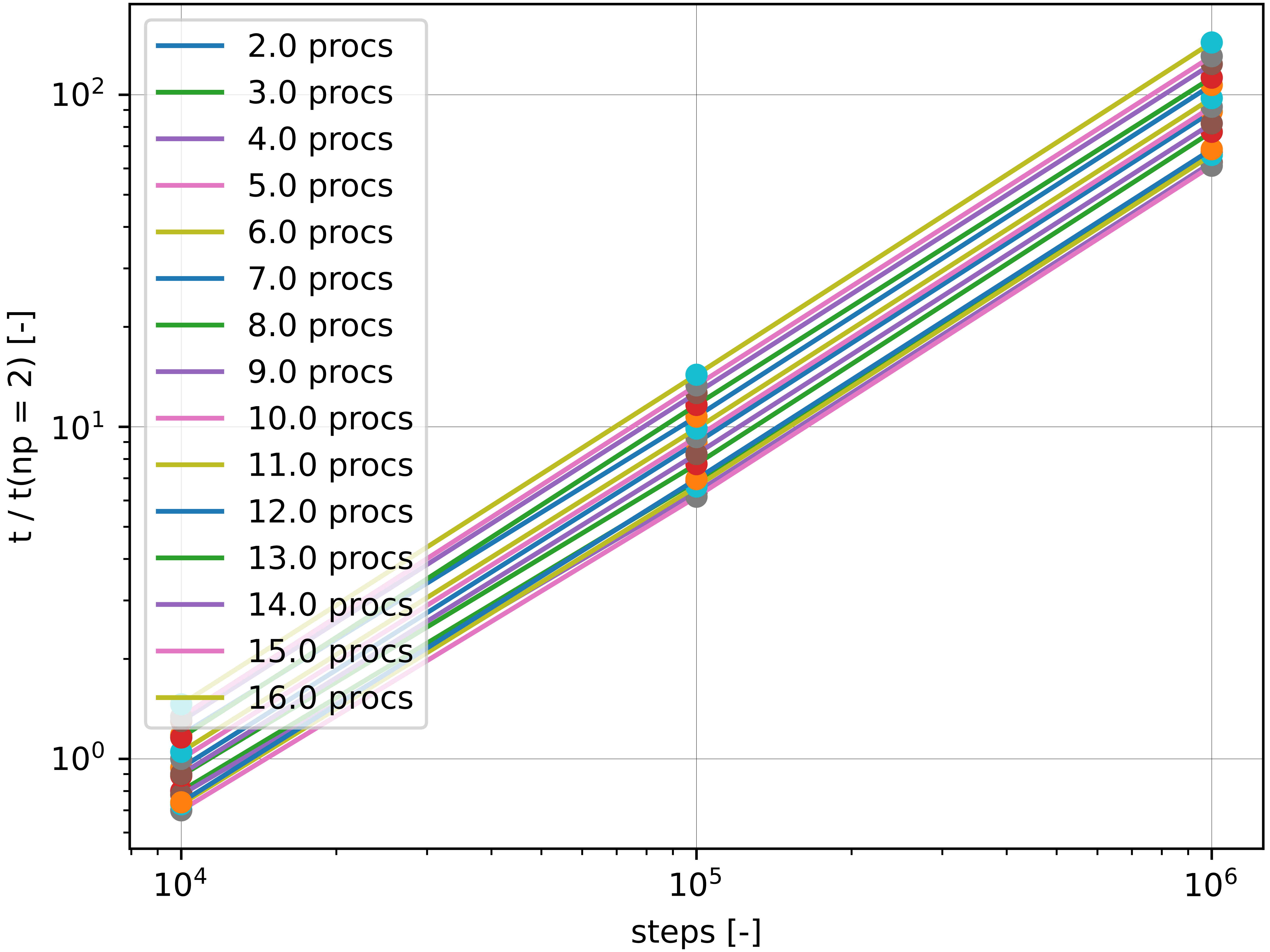
for N/(P-1) Objects:

- sys.attraction(*sys.Objects*)
- Prepare data to send

Parallelization

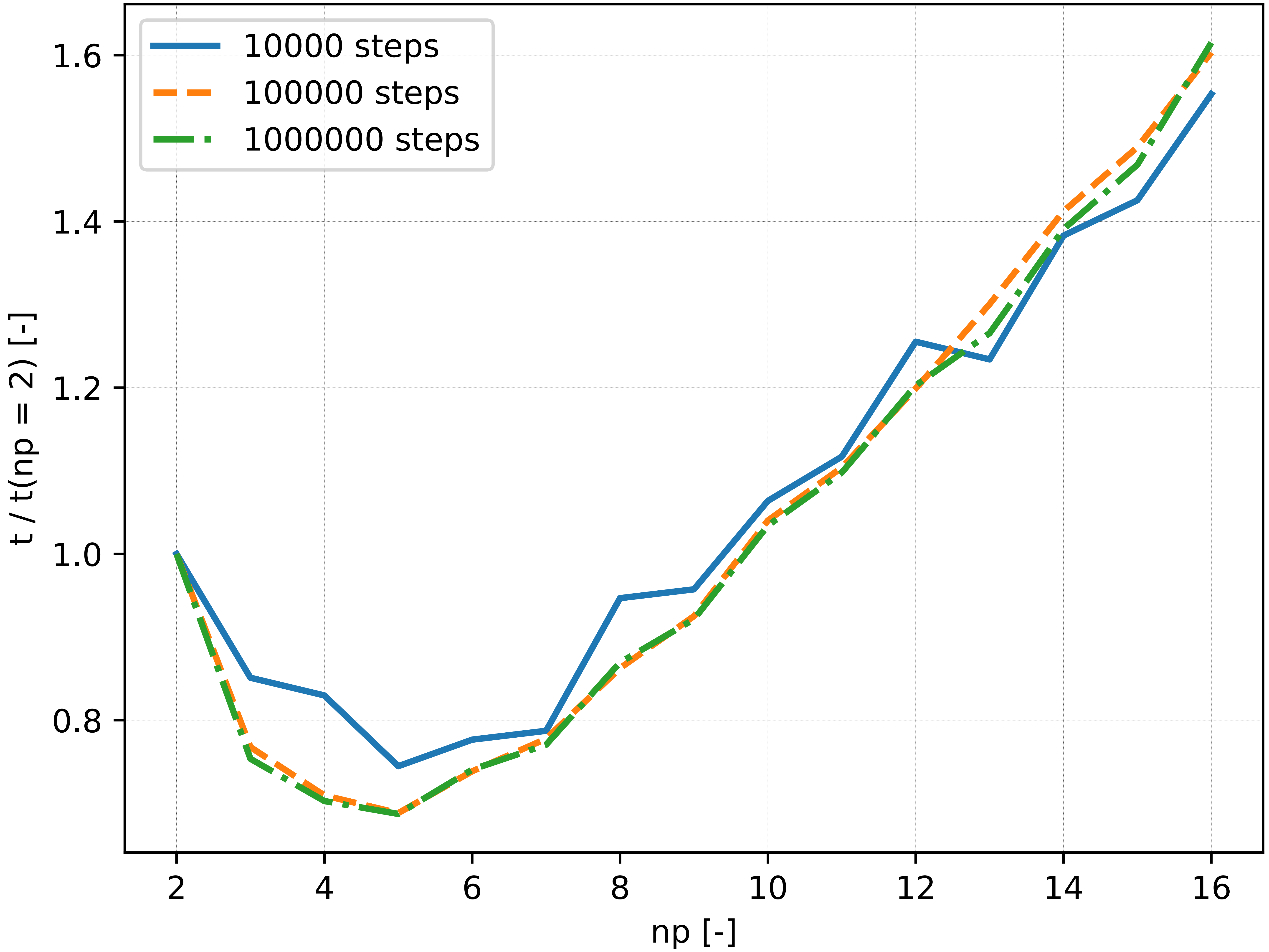
Performance

Simulation time dependence



Parallelization
Performance

Simulation time dependence



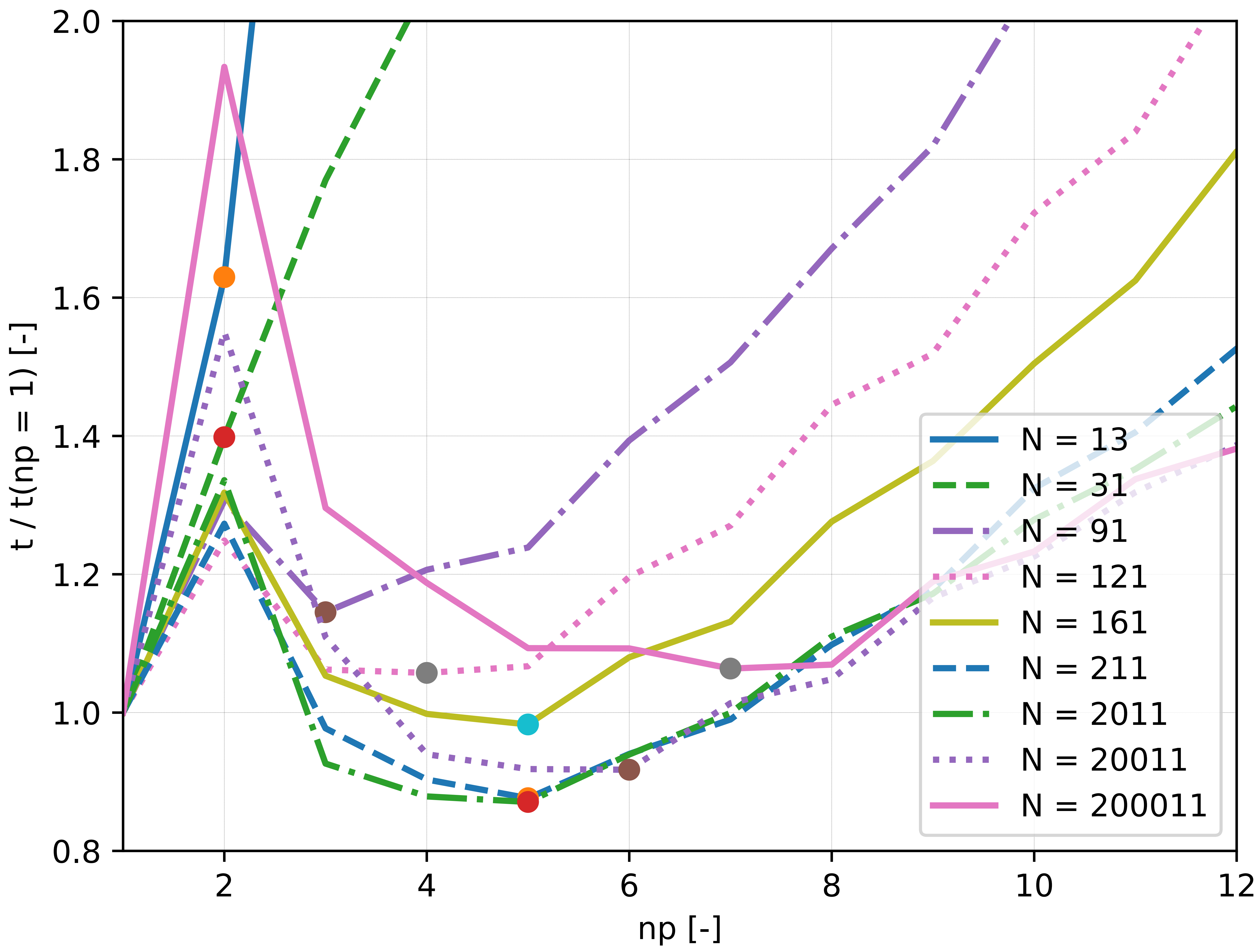
Parallelization

Performance

System size dependence

→ timing without MPI_Init()

→ Find balance between efficient **work load distribution** and MPI **communications**



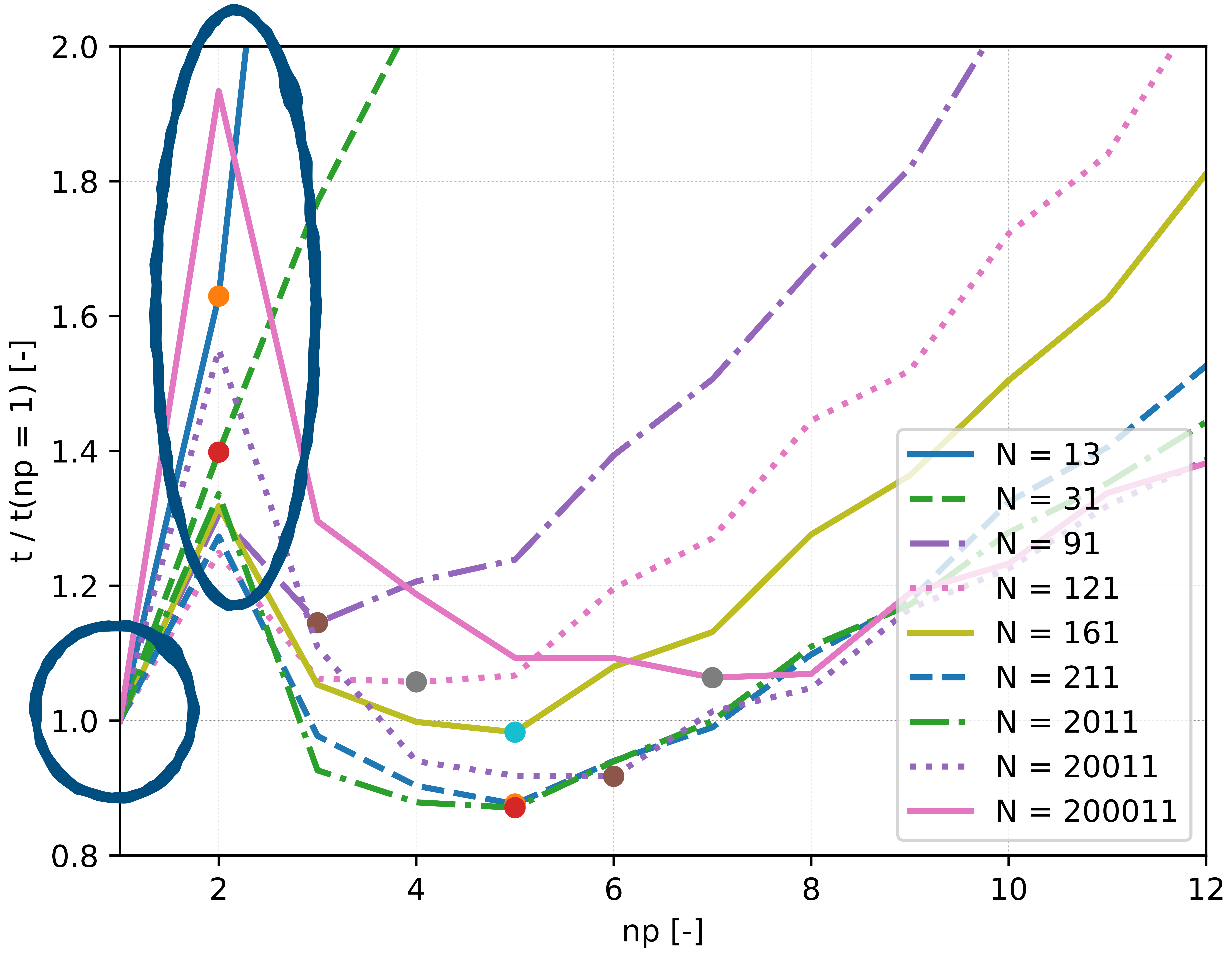
Parallelization
Performance

System size dependence

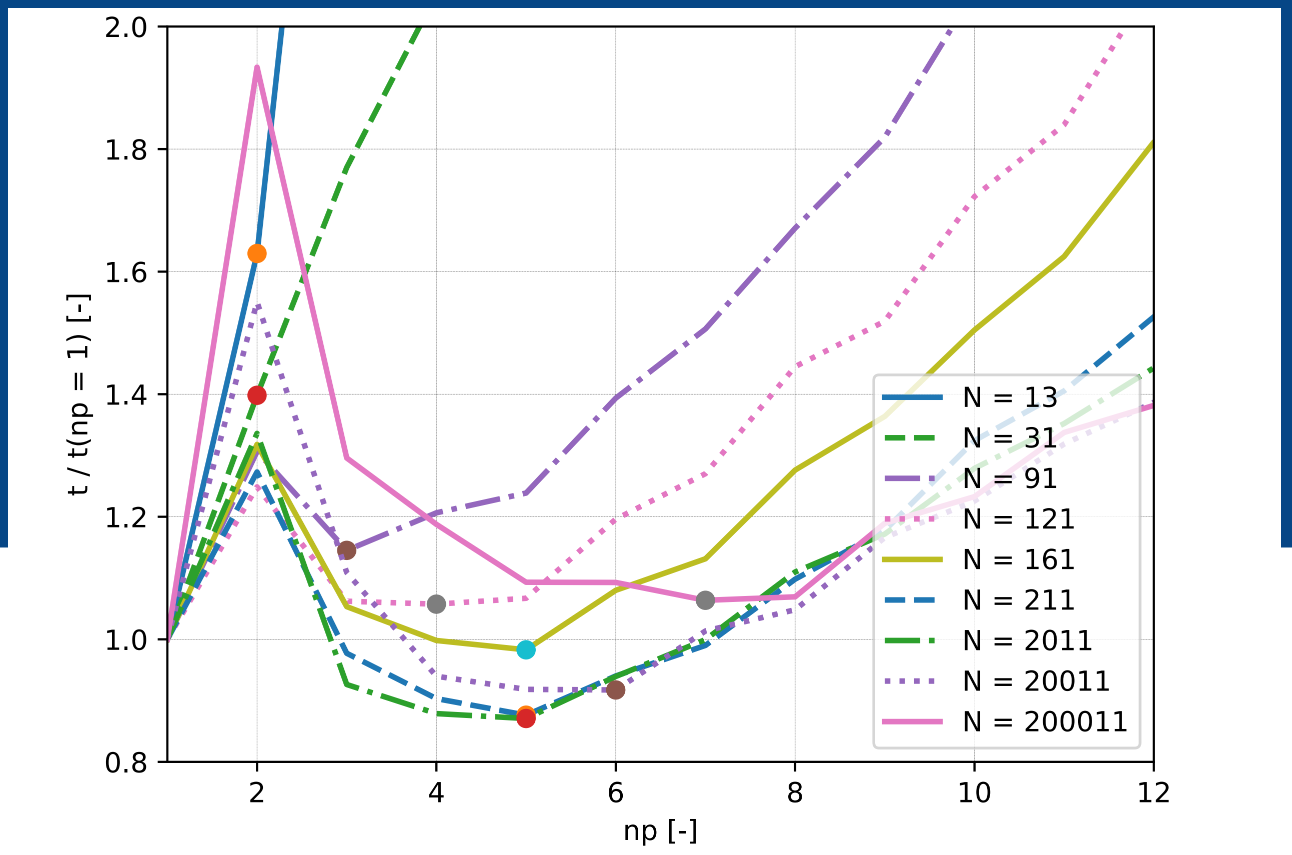
→ timing without MPI_Init()

→ Find balance between efficient **work load distribution** and MPI **communications**

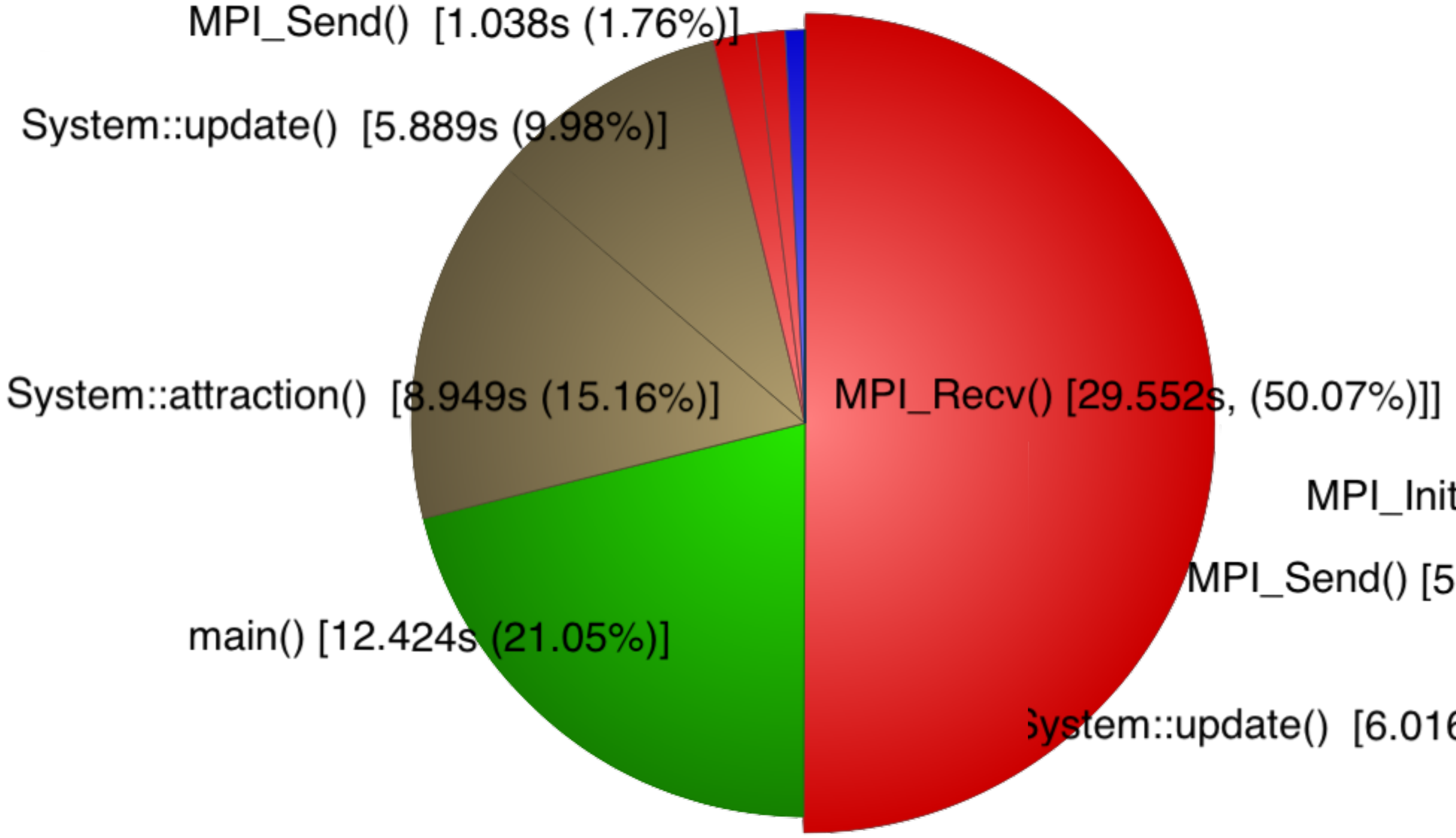
Sequential solution *(np = 1)* faster than parallel solution on two procs *(np = 2)*
→ Expectation ✓



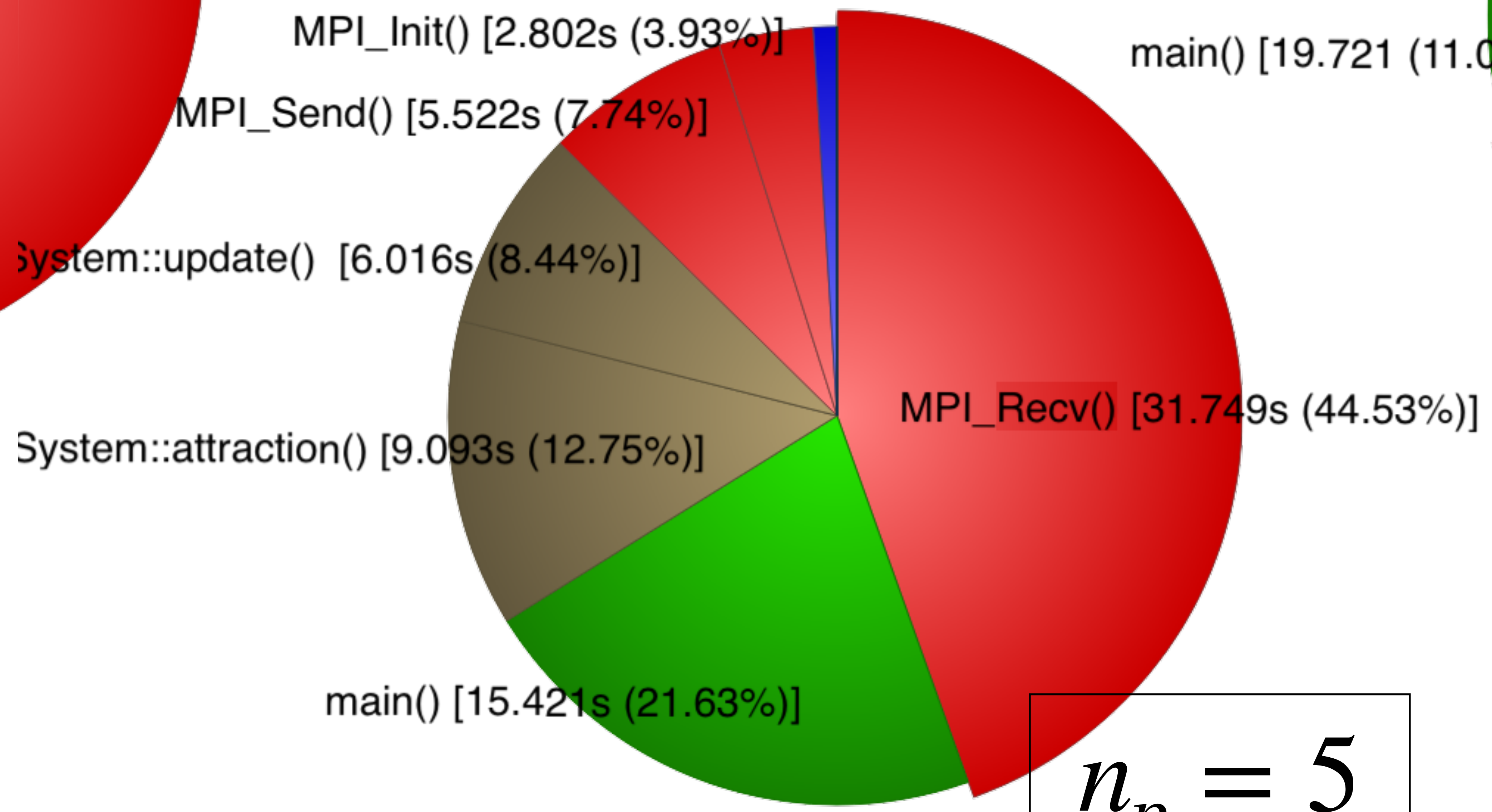
Parallelization
Performance



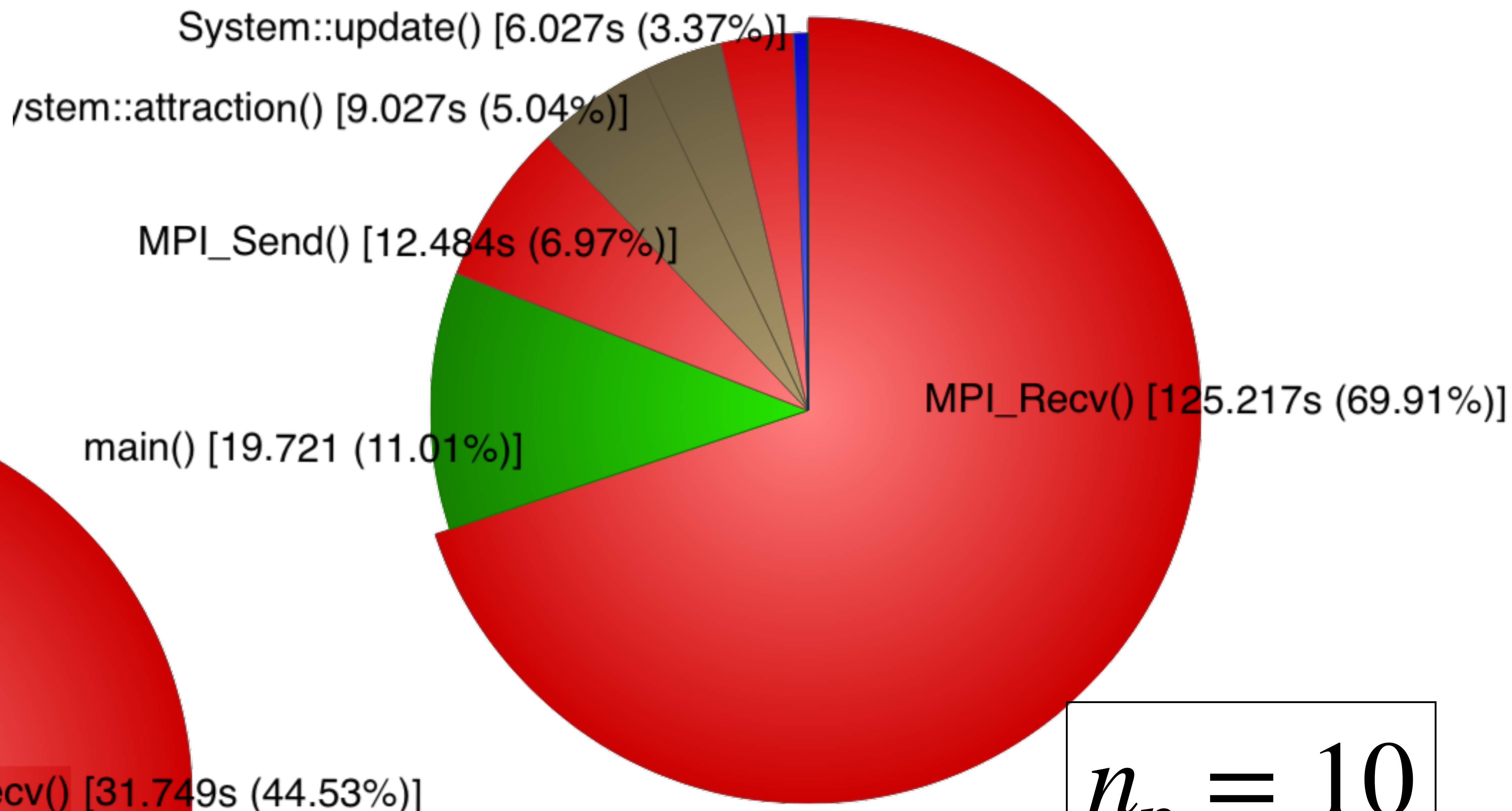
Number of Objects: $N = 211$



$n_p = 2$



$n_p = 5$



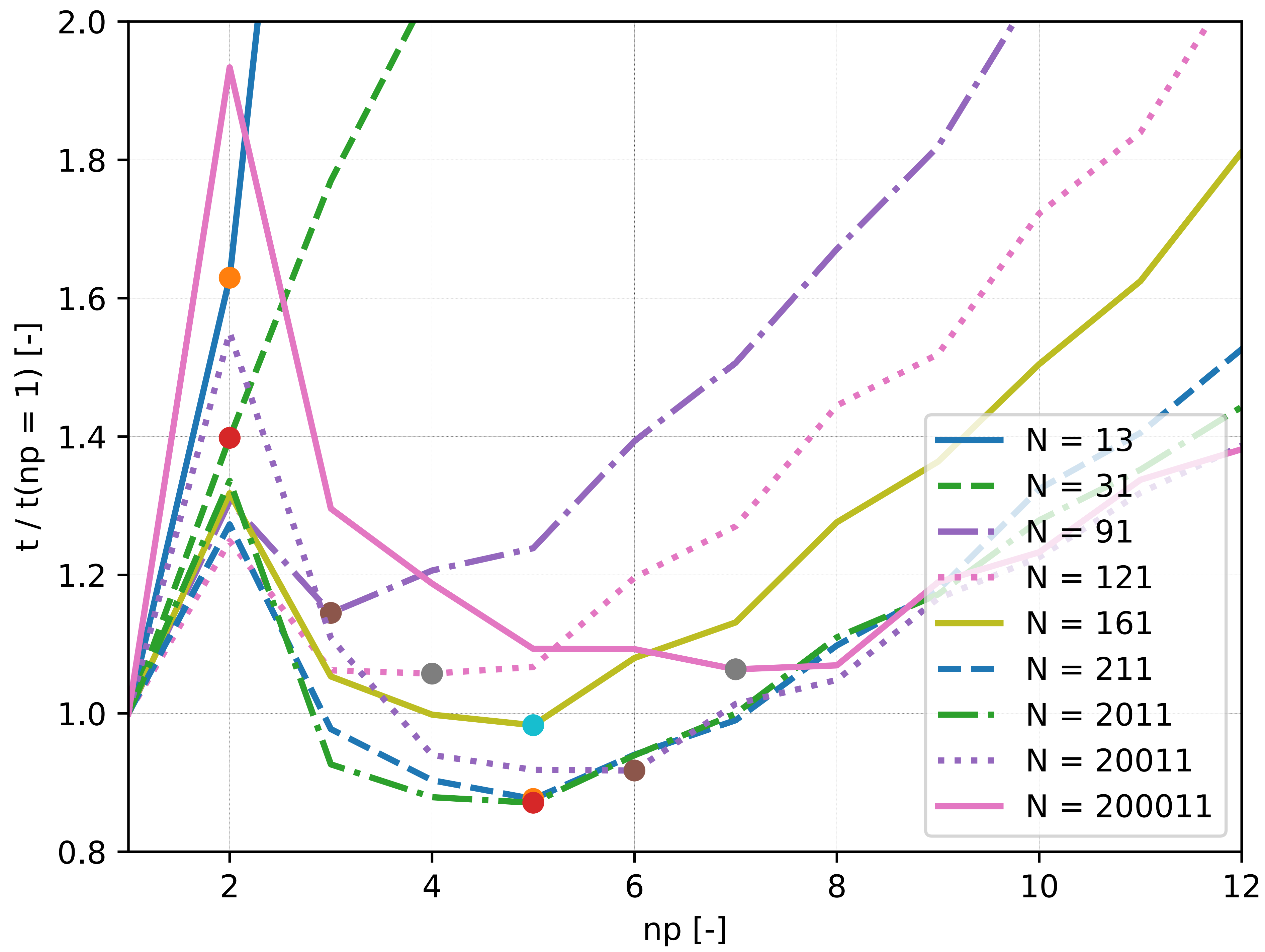
$n_p = 10$

Parallelization

Imporvements

- Masses m don't change and mass[N] vector should be accessible by every proc. after sys.init() on every proc.
  by sys.mass → ~~MPI_Send(mass, ...)~~

- Only send necessary data for performing Euler Step on procs.
  → array length N/(P-1) instead of 2N



Parallelization

Work on processors 1 to P-1

```
        // ---------- update pos. ---------- //
        // receive data:
        MPI_Recv(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(v, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(a, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // einsortieren von Zustandsvektor in system:
        for(int i = 0; i < N; i++){
            sys.Objects[i].x = y[2*i];
            sys.Objects[i].y = y[2*i+1];
            sys.Objects[i].u = v[2*i];
            sys.Objects[i].v = v[2*i+1];
            sys.Objects[i].ax = a[2*i];
            sys.Objects[i].ay = a[2*i+1];
        }
        // >>> Euler SChritt
        // perform Euler step only for the N/(num_proc-1) Objects that the
        // current processor proc. = rank is in charge:
        for(int i = (rank-1)*N/(num_proc-1); i < rank*N/(num_proc-1); i++){
            // printf("Calc. attr. for Obj. %d from proc. %d \n", i, rank);
            sys.update(&sys.Objects[i], delta_t);
            // write calculated accelerations of Object i in state vector:
            y[2*i] = sys.Objects[i].x;
            y[2*i + 1] = sys.Objects[i].y;
            v[2*i] = sys.Objects[i].u;
            v[2*i + 1] = sys.Objects[i].v;
            a[2*i] = sys.Objects[i].ax;
            a[2*i + 1] = sys.Objects[i].ay;
            pxl[2*i] = sys.Objects[i].pxl_x;
            pxl[2*i + 1] = sys.Objects[i].pxl_y;
        }
```

```
        // Send results to proc. 0 for visualization and further coordinatiion:
        MPI_Send(y, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        MPI_Send(v, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        MPI_Send(a, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        MPI_Send(pxl, 2*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

send

$i \in (p-1)N/(P-1)$

$i \in pN/(P-1)$ **is sufficient**

- Receive **all** $\vec{x}$, $\vec{v}$ and $\vec{a}$
- Sort data in *Objects*

for N/(P-1) Objects:

- sys.update(*sys.Objects*)
- Prepare data to send

Conclusion

- Results show exactly expected behavior
- Find balance between efficient work load distribution and MPI communications
  → minimum t in $t(n_p)$ plot
- Can **not reduce number** of Send() and Recv() calls
  but the **amount of data** by reducing array length
- MPI not the best choice for N-body problems
  → e.g. shared memory approach should perform better
- **Advantage**: gained good understanding on how MPI works by N-body