High Performance Computing for Aerospace Engineering

Authors:

Angel Pan Du

Yi Qiang Ji Zhang

Alba Molina Cuadrado

Ivan Sermanoukian Molina

Professor/s:

Manel Soria

Arnau Miró

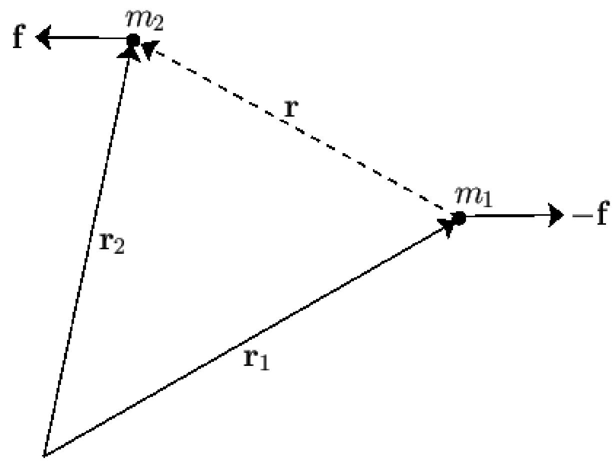
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

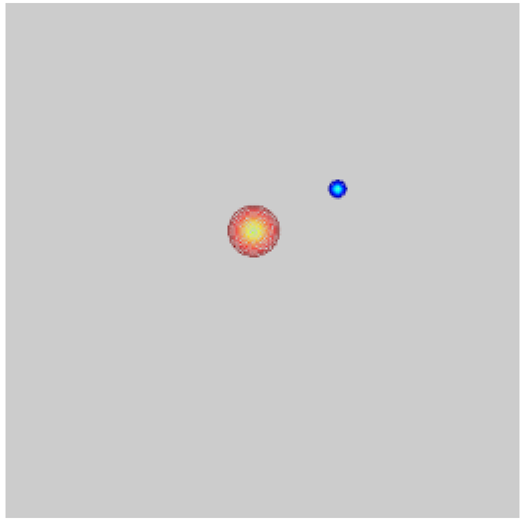
Table of contents

Introduction

- The **N-body problem** is one of the most famous problems in mathematical physics, with its first complete mathematical formulation dating back to Newton's Principia [1].
- The **N-Body Problem** is the problem of predicting the motion of a system of N particles over time when the only forces with which they interact are the forces described in Newton's Law of Universal Gravitation.
- Currently, the *two body problem* $(N = 2)$ has already been solved analytically and the solution is exact, in fact, they are conic sections.
- Nevertheless, until today, the analytical solution remains unsolved for $N > 2$. The equations cannot be solved analytically and we must look at numerical solutions.
- For instance, the three body problem has been described as chaotic.



Forces in a two body problem [2].



The evolution of two bodies interaction [3].

[1] Aarseth, S. (2003). Gravitational N-Body Simulations: Tools and Algorithms (Cambridge Monographs on Mathematical Physics). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511535246
[2] Fitzpatrick, R. (2016).Two-body problem. https://farside.ph.utexas.edu/teaching/celestial/Celestial/node11.html (accessed: 11-04-2021)
[3] Yanovsky, I. (2008).Point Vortices. Two-body problem. https://www.math.ucla.edu/~yanovsky/Research/VortexSheets/2body.htm (accessed: 11-04-2021)

Introduction (cont.)

- A **chaotic system** can be described as a system that are deterministic for an initial state, meaning that no randomness is involved in the development of future states of the system but are highly sensitive on initial conditions [1].

- It is actually not possible to write down all the terms of a general formula that describes the motion of three or more gravitational objects. The issue lies in how many unknown variables a N body system contains.

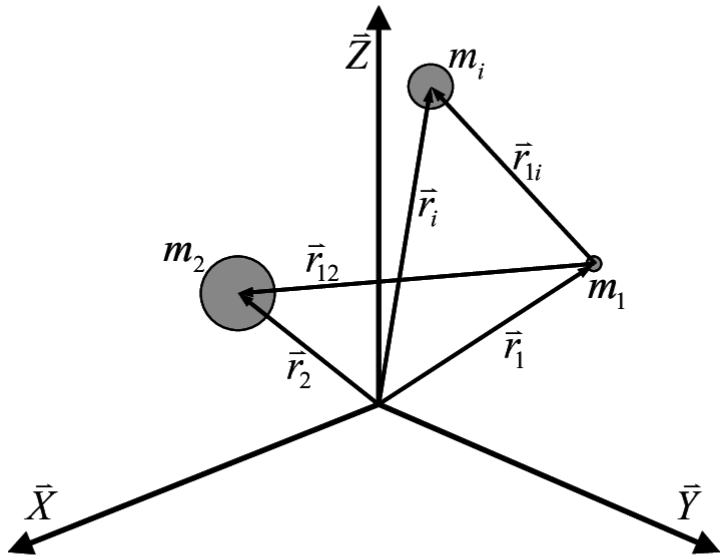
- For N>2, there are more unknowns than equations describing them.

[1] Lauterborn W.(2003) Chaotic Systems. https://www.sciencedirect.com/topics/computer-science/chaotic-systems

Introduction (cont.)

- The **N-Body problem** can be stated as:

- *“Given n bodies with masses and initial position and velocities, how will they evolve over time under gravitational interaction?”*



Forces on a 3 Body system [1].

- For N = 3, solutions exist in special cases (i.e. restricted three-body problem).

- In general, numerical methods must be used to simulate such systems.

[1] A. Stahl. Benjamin.(2008) Forces on a 3 Body system. https://www.researchgate.net/figure/The-N-body-problem_fig1_264840302

Introduction (cont.)

- Given N bodies with an initial position $\overrightarrow{x_i}$and velocity v_i for $1 \leq i \leq N$, the force vector $\overrightarrow{f_{ij}}$on body i caused by its gravitational attraction to body j is given by the following [1]:

$$\overrightarrow{f_{ij}} = G \frac{m_i m_j}{|\vec{r}_{ij}|^2} \cdot \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$$

- Where m_i and m_j are the masses of the bodies i and j respectively.
- Where $\vec{r}_{ij} = \overrightarrow{x_j} - \vec{x}_i$ is the vector from the body and G is the gravitational constant.

$$\vec{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \overrightarrow{f_{ij}} = G m_i \cdot \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{m_j \vec{r}_{ij}}{|\vec{r}_{ij}|^3}$$

For N = 5, the relationships between particles follows the subsequent pattern:

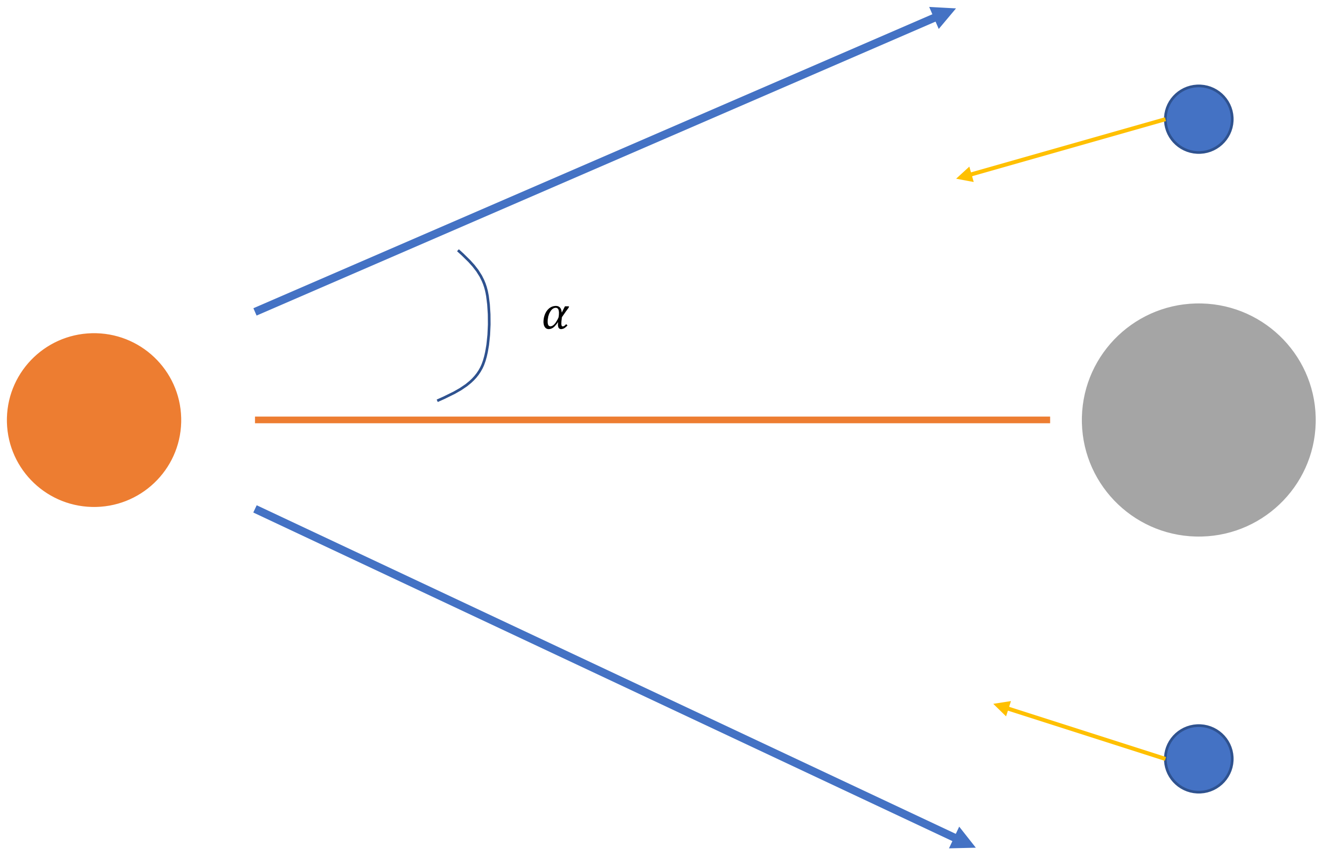
12 23 34 45
13 24 35
14 25
15

Relations: $\frac{N(N-1)}{2}$

[1] Nyland, L., Harris, M., & Prins, J. (2009). Fast N-body simulation with CUDA NVIDIA

Centre of Mass

$$\overrightarrow{f_{ij}} = G\frac{m_i m_j}{|\vec{r}_{ij}|^2}$$



- Using the centre of mass of a group of planets was firstly considered to reduce the amount of communications between cores. The idea was demonstrated to induce little error only for $\alpha \approx 0$ because of the distance values for each $\overrightarrow{f_{ij}}$ and it was therefore discarded.

What have we programmed ?

1. Adapted the Runge Kutta’s method from C++ to C.
2. Programmed the Twobody() problem in series in C.
3. Programmed the Twobody() problem in parallel in C.
4. Programmed the Nbody() problem in series using All Pairs algorithm.
5. Programmed the Nbody() problem in parallel using All Pairs algorithm.
6. The resultant plot animations in MATLAB.

All Pairs Algorithm

- The **All Pairs algorithm** is one of the simplest N-body computation. It is known as the direct sum or brute force method and it involves calculating the force between each pair of elements and then adding up the result forces on each element [1].

- The computation has two steps:
  1. Compute the forces on each element
  2. Move each element a bit based on this force, and then repeat.

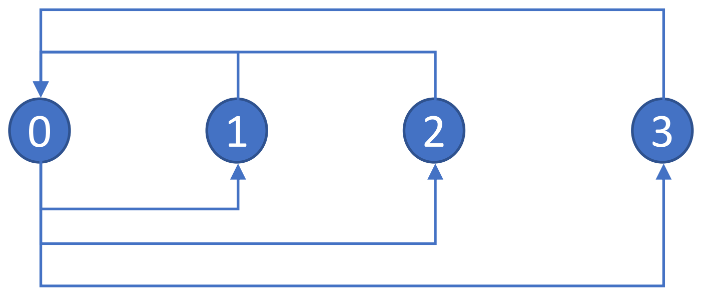
- Since each object computes the forces on it from each other object (N objects). Thus, the work complexity of this problem is a N^2 algorithm. Each object must compute its interaction with each other object so each of N-objects has to compute $N - 1$ forces.

$$N_{particles} \cdot (N-1)_{\frac{forces}{particle}} \approx O(N^2)$$

[1] Narlikar, G. (2009). Parallel N-Body Simulations. https://www.cs.cmu.edu/~scandal/alg/nbody.html (accessed: 11-04-2021)

All Pairs Algorithm

- The pseudocode is shown here:

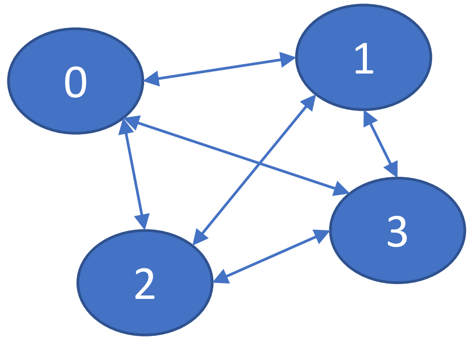


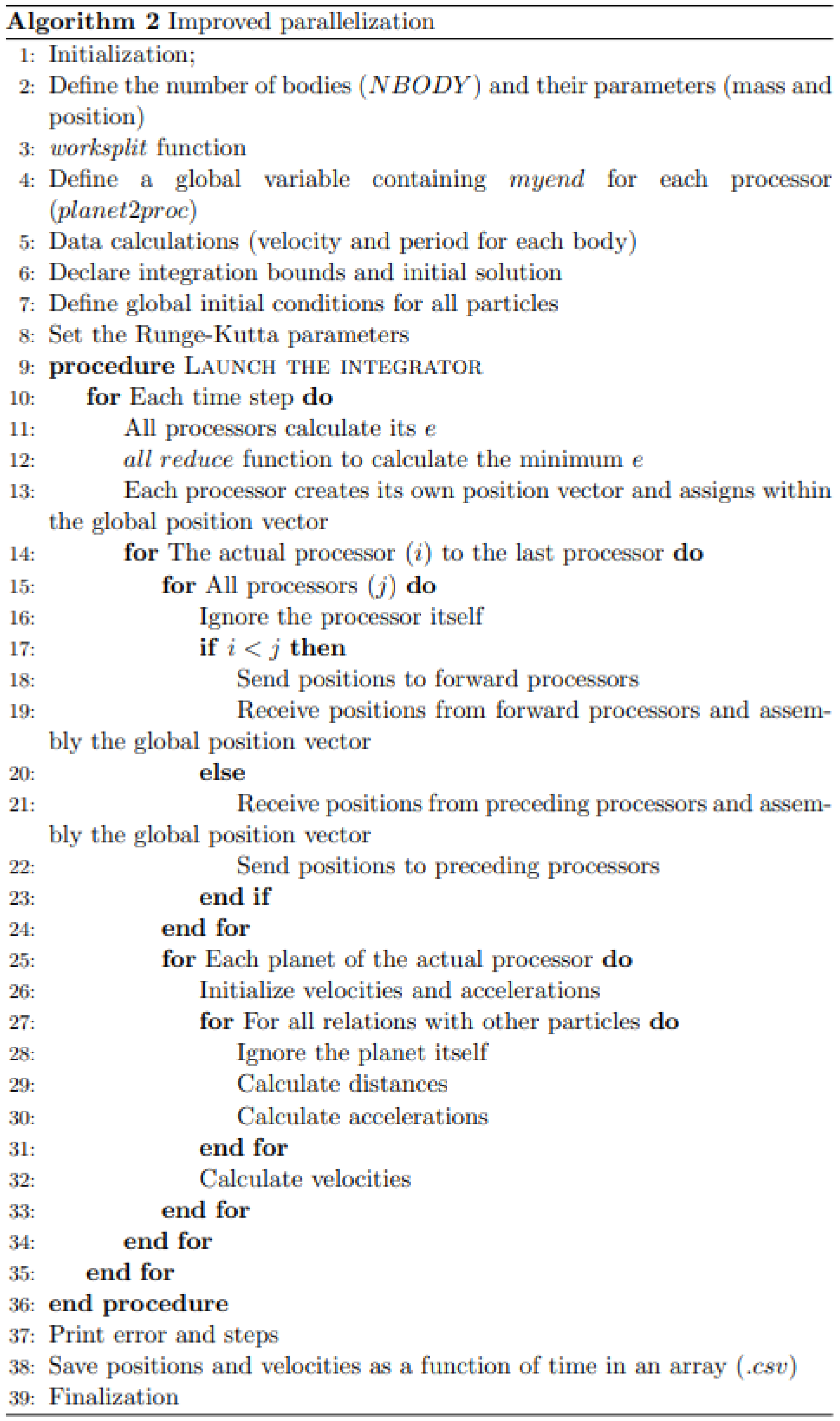
Algorithm 1 Parallelization

```
Initialization;
Define the number of bodies ($NBODY$) and their parameters (mass and position)
$worksplit$ function
Define a global variable containing $myend$ for each processor ($planet2proc$)
Data calculations (velocity and period for each body)
Declare integration bounds and initial solution
Define global initial conditions for all particles
Set the Runge-Kutta parameters
procedure LAUNCH THE INTEGRATOR
    for Each time step do
        All processors calculate its $e$
        $all\ reduce$ function to calculate the minimum $e$
        if $proc! = 0$ then
            Send positions to master processor ($proc == 0$)
        else Receive local position from other processors
            Assembly the global position vector
            Send the global position vector to other processors
        end if
        for Each planet of the actual processor do
            Initialize velocities and accelerations
            for For all relations with other particles do
                Ignore the planet itself
                Calculate distances
                Calculate accelerations
            end for
            Calculate velocities
        end for
    end for
end procedure
Print error and steps
Save positions and velocities as a function of time in an array ($.csv$)
Finalization
```

All Pairs Algorithm

- Another parallelization method:





Algorithm 2 Improved parallelization

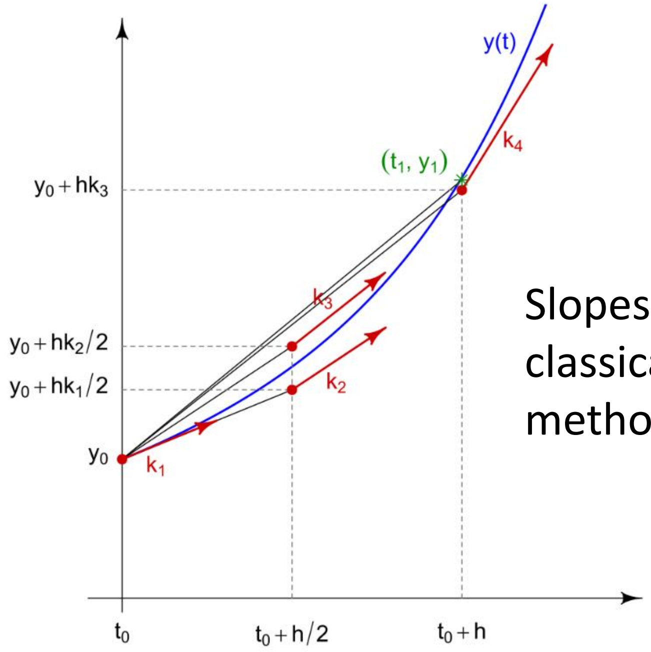
```
Initialization;
Define the number of bodies ($NBODY$) and their parameters (mass and position)
*worksplit* function
Define a global variable containing *myend* for each processor (*planet2proc*)
Data calculations (velocity and period for each body)
Declare integration bounds and initial solution
Define global initial conditions for all particles
Set the Runge-Kutta parameters
procedure LAUNCH THE INTEGRATOR
    for Each time step do
        All processors calculate its $e$
        *all reduce* function to calculate the minimum $e$
        Each processor creates its own position vector and assigns within the global position vector
        for The actual processor ($i$) to the last processor do
            for All processors ($j$) do
                Ignore the processor itself
                if $i < j$ then
                    Send positions to forward processors
                    Receive positions from forward processors and assembly the global position vector
                else
                    Receive positions from preceding processors and assembly the global position vector
                    Send positions to preceding processors
                end if
            end for
            for Each planet of the actual processor do
                Initialize velocities and accelerations
                for For all relations with other particles do
                    Ignore the planet itself
                    Calculate distances
                    Calculate accelerations
                end for
                Calculate velocities
            end for
        end for
    end for
end procedure
Print error and steps
Save positions and velocities as a function of time in an array (*.csv*)
Finalization
```

4th order Runge Kutta numerical method

- Runge Kutta is a iterative method which includes the Euler method used in temporal discretization for the approximate solutions of ordinary differential equations.
- Considering a differential equation with its initial condition:

$$\begin{cases} \dfrac{\mathrm{d}y}{\mathrm{d}x} = f\big(x, y(x)\big) \\ y(a) = y_0 \end{cases}$$

$$\begin{cases} k_1^n = f(x_n, y_n) \\ k_2^n = f\left(x_n + \dfrac{h}{2}, y_n + \dfrac{h}{2} k_1^n\right) \\ k_3^n = f\left(x_n + \dfrac{h}{2}, y_n + \dfrac{h}{2} k_2^n\right) \\ k_4^n = f(x_n + h, y_n + h k_3^n) \\ y_{i+1}^n = y_i^n + \dfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{cases}$$



Slopes used by the classical Runge-Kutta method

Zeltkevic, M. (1998).Runge-Kutta Methods. https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html (accessed: 11-04-2021)

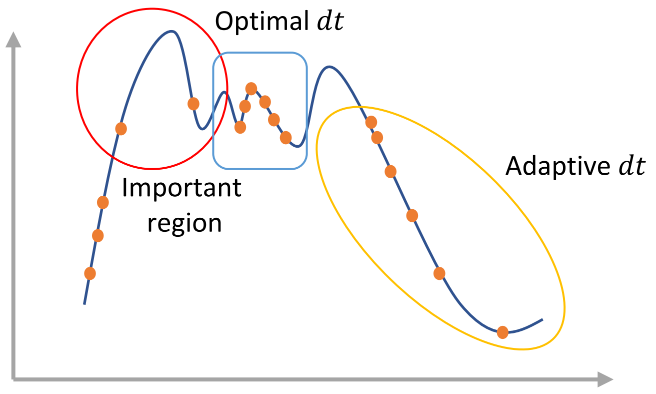
12th order Runge Kutta numerical method by Hiroshi Ono

- The Hiroshi Ono Runge Kutta method adapts the classical Runge Kutta but with a 12th order approximation. [1] [2]
- Besides, the advantage of this method is that it uses an adaptive step size depending on function and increasing its speed.
- An optimal way to implement the code is to represent the method by using the buthcler’s table which encompasses all the RK coefficients [2]:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array}$$

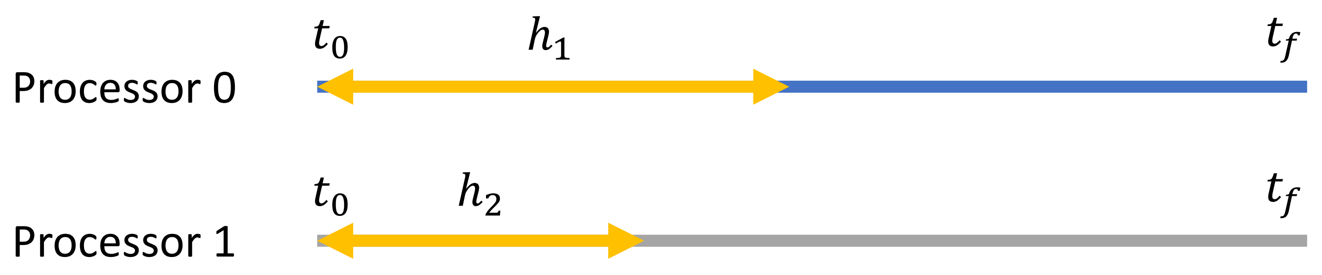
If the method is explicit, by the simplified tableau



[1] Press, W. H., & Teukolsky, S. A. (1998). Adaptive Stepsize Runge-Kutta Integration. Computers in Physics, 6 (188)
[2] Ono, Hiroshi. On the 25 stage 12th order explicit Runge-Kutta method. Transactions of the Japan Society for Industrial and applied Mathematics, Vol. 6, No. 3, (2006) pages 177 to 186

Parallelization

- What if we divide the calculation into parts? Here’s an overview:
- Each processor is responsible for N/cores part of the velocity and position update using a *worksplit()*.
- Master core reads initial data and broadcasts the local initial conditions to the other cores taking into account the particles within the core.
- Each processor is in charge of running the RK for all the particles of its core.
- Each processor has a different dt. Consider two processors:



$$e_1 = |\hat{f} - f|$$

$$e_2 = |\hat{f} - f|$$

$$\left\{ \begin{array}{l} 1)\ \min(e_1, e_2) \to h_{\min} \\ \\ 2)\ \min(h_1, h_2) \to h_{\min} \end{array} \right.$$

$$h_2 = h\_min$$

Parallelization (cont.)

- We have 2 routines:
    - A integration temporal routine (Runge Kutta)
    - The computation of the N Body
- Using MPI_ALLGATHER, the valid dt is selected by setting the minimum of dt to all processors.
- Afterwards, the processors calculates the position and velocities of all its particles.
- Each processor receives the updated positions and updates its conditions.
- Finally, each processor creates an output.csv with the positions and velocities of its particles.
- In order to view the results, the output file is ported to MATLAB so as to plot the results

Parallelization (cont.)

- We have 3 files:
- *main_NBP_MPI.c* which is the main file.
- *Nbodylib.h* which contains all the *#define* and *MACROS*.
- *Nbodylib.c* which contains the functions *worksplit()* and *Nbody().*

Parallelization (cont.)

```c
// Main
int main(int argc, char* argv[]) {

    int r; // For error checking

    // Start MPI
    r = MPI_Init(&argc, &argv);
    checkr(r, "Initiate");
    MPI_Status st;

    // Check rank and size of the processors
    int rank, size;
    rank = proc();
    size = nproc();

    array_size = size;
    planet2proc = (int*)malloc(sizeof(int)*array_size);

    // If the processor is higher than the number of processors, exit
    if (rank >= size) {exit(-1);}

    // Declare worksplit variables
    int mystart,mylocstart;
    int myend, mylocend;
    int bodies;
    for (int xproc=0; xproc<size; xproc++) {
      worksplit(&mystart, &myend, xproc, size, 0, NBODY - 1);
      if (xproc == rank) {
        bodies = myend - mystart + 1;
        mylocstart = mystart;
        mylocend = myend;
      }
      planet2proc[xproc] = myend;
    }
```

```c
// Data calculations

   double V1 = Vx(M0,R1);  // Mercury velocity
   double T1 = Tx(M0,R1);  // Mercury period
   double V2 = Vx(M0,R2);  // Venus velocity
   double T2 = Tx(M0,R2);  // Venus period
   double V3 = Vx(M0,R3);  // Earth velocity
   double T3 = Tx(M0,R3);  // Earth period
   double V4 = Vx(M0,R4);  // Mars velocity
   double T4 = Tx(M0,R4);  // Mars period
   double V5 = Vx(M0,R5);  // Jupiter velocity
   double T5 = Tx(M0,R5);  // Jupiter period
   double V6 = Vx(M0,R6);  // Saturn velocity
   double T6 = Tx(M0,R6);  // Saturn period
   double V7 = Vx(M0,R7);  // Uranus velocity
   double T7 = Tx(M0,R7);  // Uranus period
   double V8 = Vx(M0,R8);  // Neptune velocity
   double T8 = Tx(M0,R8);  // Neptune period
   double V9 = Vx(M0,R9);  // Pluto velocity
   double T9 = Tx(M0,R9);  // Pluto period


// Integration bounds and initial solution
double tspan[2] = {0.,T9};

// All initial conditions
double *y0_loc;
y0_loc = (double *)malloc(bodies*6*sizeof(double));

// Global initial conditions for all particles
for(int planet_p = 6*mylocstart; planet_p < (6*(mylocend + 1)); planet_p++) {

    double y0_temp[N] = {0., 0., 0., 0., 0., 0., /* Sun */
                         R1, 0., 0., 0., V1, 0., /* Mercury */
                         0., R2, 0., -V2, 0., 0., /* Venus */
                         0., R3, 0., -V3, 0., 0., /* Earth */
                         0., R4, 0., -V4, 0., 0., /* Mars */
                         0., R5, 0., -V5, 0., 0., /* Jupiter */
                         0., R6, 0., -V6, 0., 0., /* Saturn */
                         0., R7, 0., -V7, 0., 0., /* Uranus */
                         0., R8, 0., -V8, 0., 0., /* Neptune */
                         0., R9, 0., -V9, 0., 0. /* Pluto */
                     };
    y0_loc[planet_p-6*mylocstart] = y0_temp[planet_p];
}
```

Parallelization (cont.)

```
    // Runge-Kutta parameters
    RK_PARAM rkp = rkparams(tspan);
    rkp.h0 = 24.*3600.;
    rkp.eps = 1e-11;

    // Launch the integrator
    RK_OUT rko = odeRK("hiroshi912", NBody, tspan, y0_loc, 6*bodies, &rkp);

    // Finish
    printf("error = %.2e with %d steps\n", rko.err, rko.n);

    char str[15+nproc()];

    for(int xproc = 0; xproc < nproc(); xproc++) {
      if(xproc == proc()) {
        sprintf(str, "Output%d.csv", xproc);
        writerkout(str, &rko, 6*bodies);
      }
    }

    freerkout(&rko);
    free(y0_loc);
    free(planet2proc);

    // Finalise MPI
    MPI_Finalize();

    // End of the program
    exit(0);
}
```

Parallelization (cont.)

```c
// Worksplit

void worksplit(int *mystart, int *myend, int proc, int nproc, int start, int end)
{
    // Number of tasks
    int ntask = end - start + 1;
    // Number of tasks per processor
    int interval = ntask / nproc;
    // Tasks left
    int remainder = ntask % nproc;

    if (ntask < nproc)
    {
        printf("Less tasks than processors\n");
        exit(-1);
    }
    if (remainder != 0)
    {
        if (proc < remainder)
        {
            *mystart = start + proc * (interval + 1);
            *myend = *mystart + interval;
        }
        else
        {
            *mystart = start + remainder * (interval + 1) + (proc - remainder) * interval;
            *myend = *mystart + interval - 1;
        }
    }
    else
    {
        *mystart = start + proc * interval;
        *myend = *mystart + (interval - 1);
    }
}
```

```c
void NBody(double t, double* var, int n, double* varp) {

    /*
      Body 1: Perturbated
          var[0] = rx   var[3] = vx
          var[1] = ry   var[4] = vy
          var[2] = rz   var[5] = vz
      Body 2: Perturber
          var[6] = rx   var[9]  = vx
          var[7] = ry   var[10] = vy
          var[8] = rz   var[11] = vz
      Body 3: Perturber
          var[12] = rx   var[15] = vx
          var[13] = ry   var[16] = vy
          var[14] = rz   var[17] = vz
    */
    // varp = prime (derivative of var | var prime)

    int r;      // for error checking
    int start = 0;
    MPI_Status st;

    if(proc() != 0) {
        start = planet2proc[proc()-1] + 1;
      }

    double* y;
    y = (double*)malloc(3*NBODY*sizeof(double));

    if (proc() != 0) {

        double* pos; // vectors of positions
        pos = (double*)malloc(3*(BODIES(proc()))*sizeof(double));

        for (int body = 0; body < (BODIES(proc())); body++)
        {
            pos[3*body]       = VAR(body,0);
            pos[3*body + 1] = VAR(body, 1);
            pos[3*body + 2] = VAR(body, 2);
        }

       r = MPI_Ssend(pos, 3*(BODIES(proc())), MPI_DOUBLE, 0, proc() + nproc(), MPI_COMM_WORLD);
       checkr(r, "send");

       free(pos);

       r = MPI_Recv(y, 3*NBODY, MPI_DOUBLE, 0, proc(), MPI_COMM_WORLD, &st);
       checkr(r, "receive");
    }
```

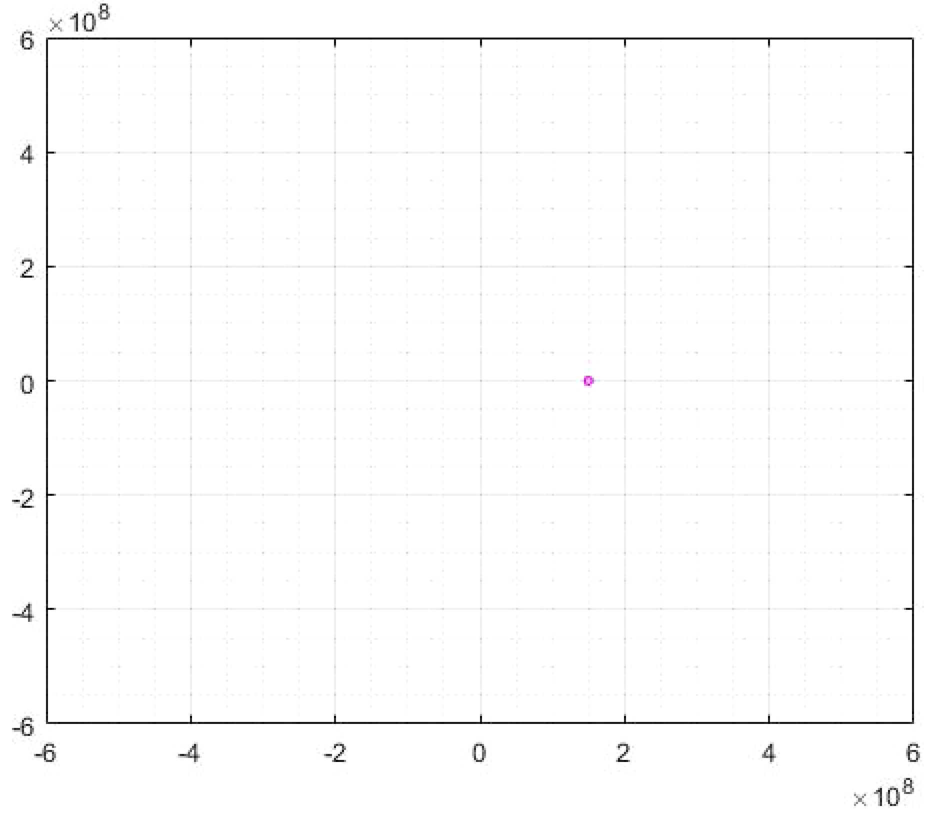
```c
       else { // If rank == 0

        double* var_loc;
        // var_loc = (double*)malloc(3*(BODIES(1))*sizeof(double));

        for (int y_counter = 0; y_counter <= planet2proc[0]; y_counter++) {
            Y(y_counter,0) = VAR(y_counter,0);
            Y(y_counter,1) = VAR(y_counter,1);
            Y(y_counter,2) = VAR(y_counter,2);
        }


        for (int xproc = 1; xproc < nproc(); xproc++) {

            var_loc = (double*)malloc(3*(BODIES(xproc))*sizeof(double));
            // var_loc = realloc(var_loc,3*(BODIES(xproc))*sizeof(double));

            r = MPI_Recv(var_loc, 3*(BODIES(xproc)), MPI_DOUBLE, xproc, xproc + n
proc(), MPI_COMM_WORLD, &st);
            checkr(r, "receive");

            for (int p = 0; p < (3*(BODIES(xproc))); p++) {
                Y(planet2proc[xproc - 1] + 1,p) = var_loc[p];
            }

            free(var_loc);

            r = MPI_Ssend(y, 3*NBODY,MPI_DOUBLE, xproc, xproc, MPI_COMM_WORLD);
            checkr(r, "send");
        }
    }
```

Parallelization (cont.)

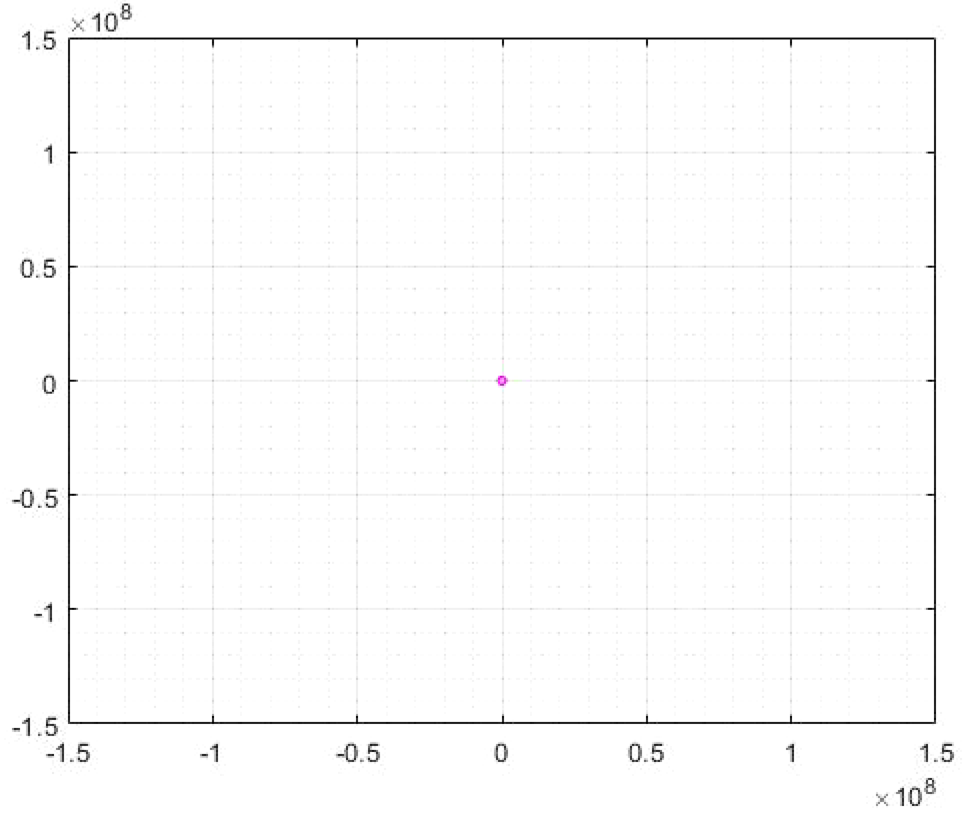
Parallelization (cont.)

```
        else { // If rank == 0

         double* var_loc;
         // var_loc = (double*)malloc(3*(BODIES(1))*sizeof(double));

         for (int y_counter = 0; y_counter <= planet2proc[0]; y_counter++) {
             Y(y_counter,0) = VAR(y_counter,0);
             Y(y_counter,1) = VAR(y_counter,1);
             Y(y_counter,2) = VAR(y_counter,2);
         }


         for (int xproc = 1; xproc < nproc(); xproc++) {

             var_loc = (double*)malloc(3*(BODIES(xproc))*sizeof(double));
             // var_loc = realloc(var_loc,3*(BODIES(xproc))*sizeof(double));

             r = MPI_Recv(var_loc, 3*(BODIES(xproc)), MPI_DOUBLE, xproc, xproc + n
    proc(), MPI_COMM_WORLD, &st);
             checkr(r, "receive");

             for (int p = 0; p < (3*(BODIES(xproc))); p++) {
                 Y(planet2proc[xproc - 1] + 1,p) = var_loc[p];
             }

             free(var_loc);

             r = MPI_Ssend(y, 3*NBODY,MPI_DOUBLE, xproc, xproc, MPI_COMM_WORLD);
             checkr(r, "send");
         }
     }
```

2 Body Sun-Earth (in Serie)



3 Body Sun-Venus-Earth (in Serie)



2 Body Sun-Mercury (in Parallel with 2 cores)

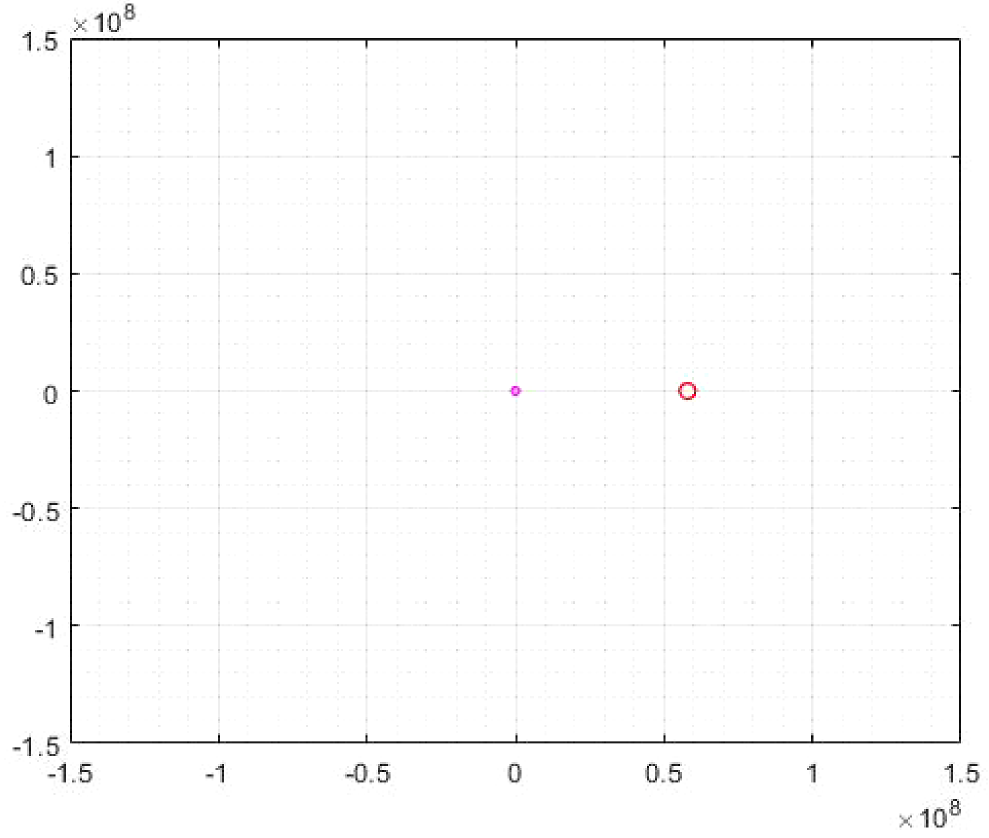
Sun

0	0	0	0	0	0	0
86400	0.0245110000000000	0.000584000000000000	0	1.00000000000000e-06	0	0
250682.142843000	0.205685000000000	0.0142310000000000	0	2.00000000000000e-06	0	0
458056.891552000	0.681010000000000	0.0863840000000000	0	3.00000000000000e-06	1.00000000000000e-06	0
677541.526709000	1.46891300000000	0.277192000000000	0	4.00000000000000e-06	1.00000000000000e-06	0
900692.956362000	2.54390600000000	0.643384000000000	0	5.00000000000000e-06	2.00000000000000e-06	0
1125253.78642900	3.86774100000000	1.23517000000000	0	6.00000000000000e-06	3.00000000000000e-06	0
1350550.43727600	5.39422400000000	2.09511500000000	0	7.00000000000000e-06	4.00000000000000e-06	0
1576340.24726600	7.07035800000000	3.25673900000000	0	8.00000000000000e-06	6.00000000000000e-06	0
1802517.17034800	8.83782300000000	4.74327000000000	0	8.00000000000000e-06	7.00000000000000e-06	0
2029046.25641200	10.6350090000000	6.56691100000000	0	8.00000000000000e-06	9.00000000000000e-06	0

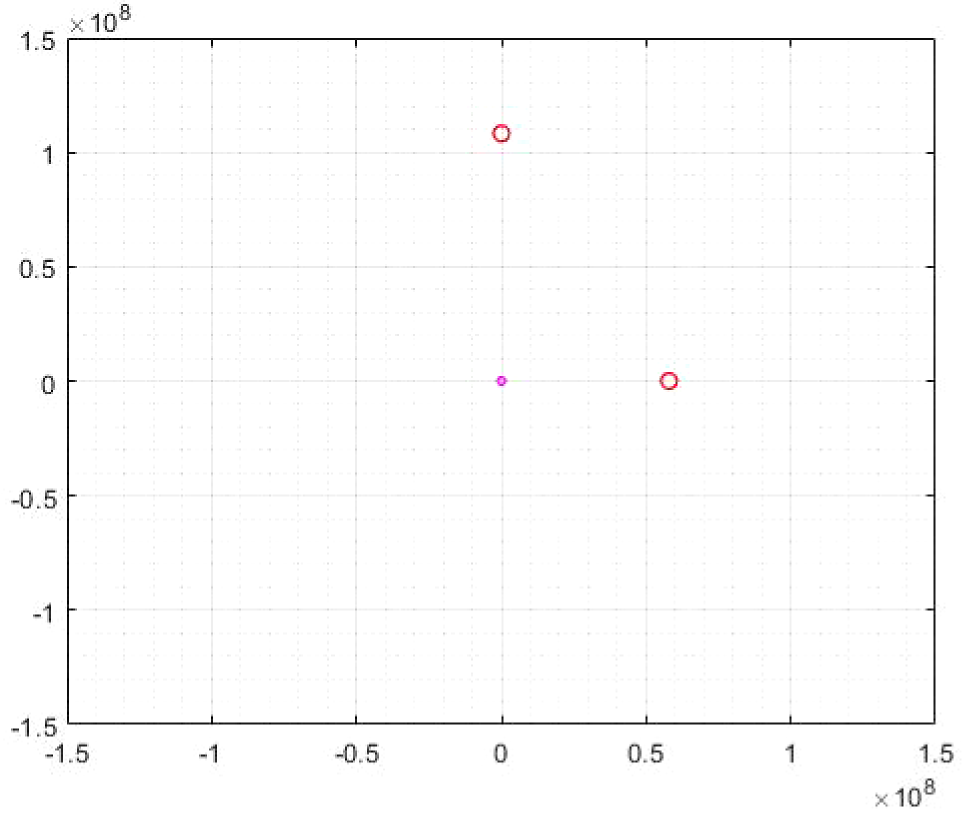
Mercury

t	x	y	z	vx	vy	vz
0	57909227	0	0	0	47.8792710000000	0
86400	57761533.9424860	4133251.54644200	0	-3.41736700000000	47.7571580000000	0
250682.142843000	56669834.6280460	11916728.3596100	0	-9.85273500000000	46.8545430000000	0
458056.891552000	53805683.6141540	21410907.9831080	0	-17.7025100000000	44.4864660000000	0
677541.526709000	49058030.2004960	30769924.2059480	0	-25.4405320000000	40.5611130000000	0
900692.956362000	42580473.3400290	39247698.4597760	0	-32.4499450000000	35.2054770000000	0
1125253.78642900	34603467.9627180	46433592.4855720	0	-38.3912330000000	28.6100990000000	0
1350550.43727600	25405377.1240080	52038882.1201350	0	-43.0256800000000	21.0051290000000	0
1576340.24726600	15305536.0609840	55849967.7614430	0	-46.1766810000000	12.6545920000000	0
1802517.17034800	4655363.32998600	57721796.3189560	0	-47.7243070000000	3.84904100000000	0
2029046.25641200	-6173898.99049100	57579171.0271870	0	-47.6063840000000	-5.10458100000000	0

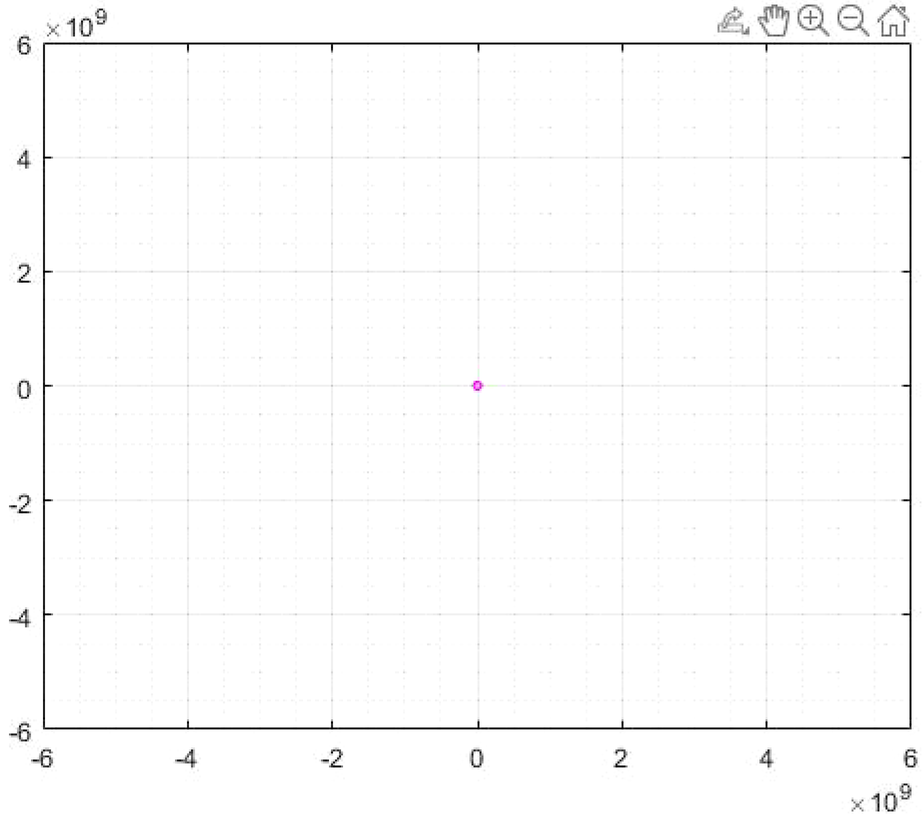
2 Body Sun-Mercury (in Parallel with 2 cores)



3 Body Sun-Mercury-Venus (in Parallel with 3 cores)



Solar System (in Parallel*)



*Sometimes the program is unstable to overflow

Validation of the results

- An orbiting particle has two important forms of energy: *gravitational potential energy*, U, and *kinetic energy*, K.
- The sum of the kinetic and gravitational potential energy is known as the total mechanical energy of the system, E.
- The sum of these two energies give us what is known as the total mechanical energy of the system, E. The equation for the total mechanical energy of an orbiting body is derived by combining the centripetal force of a satellite being equal to the force of gravity, with the magnitude of the force from Newton's 2nd law and remembering the equation for the centripetal acceleration [1]:

$$F = \frac{GMm}{r^2} = \frac{mv^2}{r}$$

$$mv^2 = \frac{GMm}{r}$$

$$\frac{1}{2}mv^2 = \frac{GMm}{2r}$$

$$E = K + U = \frac{GMm}{2r} - \frac{GMm}{r} = -\frac{GMm}{2r}$$

$$\boxed{E = -\frac{GMm}{2r}}$$

$$\text{Kinetic energy: } K = \frac{1}{2}\sum_{i=1}^{N} m_i v_i^2$$

$$\text{Potential energy: } W = -\frac{1}{2}G\sum_{i=1}^{N}\sum_{j=1,\neq i}^{N}\frac{m_i m_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|}$$

$$\text{Total energy: } E = K + W$$

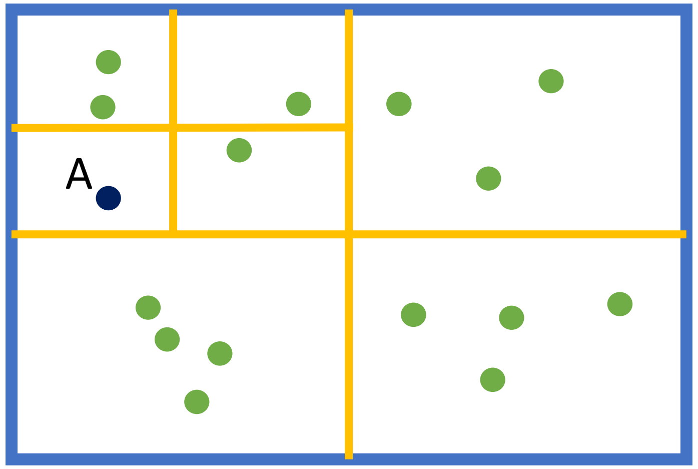
$$\text{Energy conservation: } E = \text{constant}$$

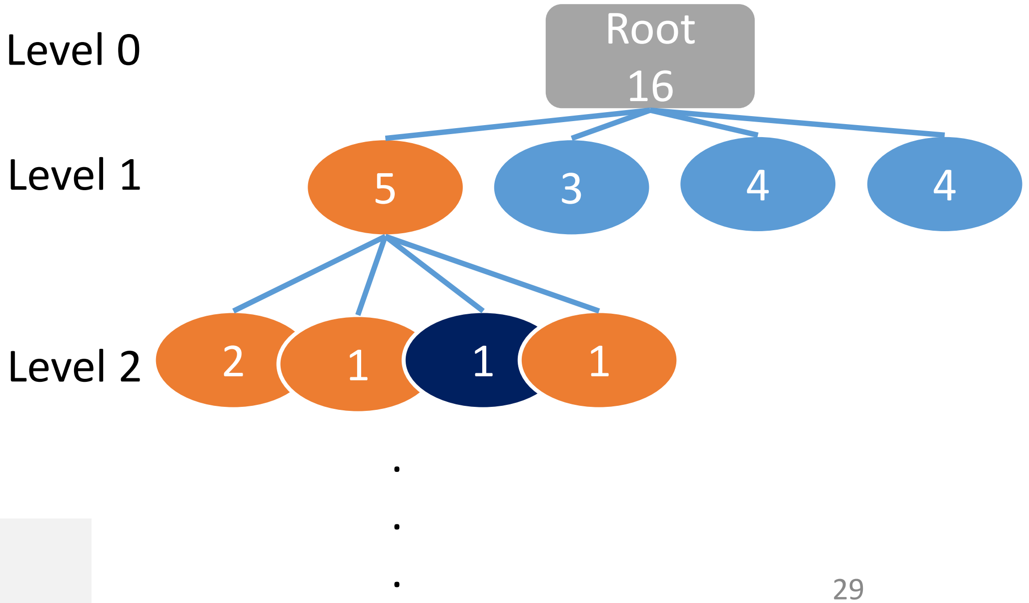
[1] Gieles, M. (University of Cambridge). (2011). Numerical Simulations of the Gravitational N-body problem. https://www.ast.cam.ac.uk/sites/default/files/topics_lecture_mgieles_19112011_0.pdf (accessed: 11-04-2021)

More optimized Algorithms

- **Barnes-Hut Oct-tree algorithm**: $O(NlogN)$ [1] [2] [3]
  - In 1996, Josh Barnes and Piet Hut developed an approximation algorithm for performing an N Body simulation that reduces the amount of computational time from $O(N^2)$ to $O(NlogN)$.
  - The main idea is that force calculations don't need to be exact for particles that are far away from each other. They proposed to group particles together and approximate them by their centre of mass. A tree is used to represent the hierarchical subdivision of space known as octants (quadrants in 2D).
  - The division continues until each particle has a cell itself.
  - Consider the highlighted particle A: At the first level of the tree, three cells (in blue) are far away and can be approximated by its centre of mass but within level it cannot be approximated expect the first quadrant of this level.

16 particles in a 2D Domain





[1] Barnes, J., & Hut, P. (1986). *A hierarchical O(N log N) force-calculation algorithm. Nature, 324(6096), 446–449.* doi:10.1038/324446a0
[2] Greengard, L. (1990). The Numerical Solution of the N-Body Problem. Computers in Physics, 4 (142).
[3] Liu, Pangfeng & Bhatt, Sandeep. (2001). Experiences with parallel N-body simulation. Parallel and Distributed Systems, IEEE.

Problems encountered

- Runge Kutta is strongly **non parallelizable.**
- Adaptative vs. constant time step.
- Difficult to verify the results.
- Enormous amount of send and receives which results in a slower computational time unless N is considerably high and a decent amount of computer cores.
- Several alternative methods where analysed for optimizing sends and receives.
- Overflow of data when sending data.

Further improvements

- Set sphere of influence depending on the mass and distance.
- Initial conditions must be written in situ, consider import from txt.
- Implement the Barnes-Hut approach.
- Implement a 3D plot of the motion.
- As bodies approach each other, the force between them grows without bound, which is an undesirable situation for numerical integration. In astrophysical simulations, collisions between bodies are generally precluded; this is reasonable if the bodies represent galaxies that may pass right through each other. Therefore, a softening factor $\varepsilon^2 > 0$ is added, and the denominator is rewritten as follows [1] [2]:

$$\overrightarrow{F_i} \approx Gm_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \vec{r}_{ij}}{\left(|\vec{r}_{ij}|^2 + \varepsilon^2\right)^{3/2}}$$

- To integrate over time, we need the acceleration $a_i = F_i/m_i$ to update the position and velocity of body i, and so we simplify the computation to this:

$$\overrightarrow{a_i} \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \vec{r}_{ij}}{\left(|\vec{r}_{ij}|^2 + \varepsilon^2\right)^{3/2}}$$

[1] Nyland, L., Harris, M., & Prins, J. (2009). Fast N-body simulation with CUDA NVIDIA
[2] Dyer, Charles, and Peter Ip. 1993. "Softening in N-Body Simulations of Collisionless Systems." The Astrophysical Journal 409, pp. 60–67.

Conclusions

- **All pairs N-body** is a brute-force, simple, high performance method. Nevertheless, there are other more efficient algorithms.
- There is a trade-off between:
  - Maximizing parallelism
  - More work per processor
- The most efficient way is to increase the amount of work on each processor and reducing the amount of parallelism as the speed of the communications between a single processor is faster than communicating between processors.
- The problem is implemented in 3D but the resultant plots are 2D.
- We learned how to version control our codes and work in a repository as a group using git.
- We have struggled a lot but we have been able to view programming from another point of view. Besides, we been able to think differently as we learned how memory is structured in a computer.
- We also learned how to debug effectively, for instance, we used *valgrind* tool to view where memory had conflicts.
- After this, we aim to develop code using MPI in our future problems.