TÉCNICO LISBOA

# Parallel and Distributed Computing

Project Assignment

## PARTICLES SIMULATION

**Version 1.0 (23/01/2025)**

2024/2025

3rd Period

# Contents



# Revisions

| | |
|---|---|
| Version 1.0 (January 23, 2025) | Initial Version |

# 1 Introduction

The purpose of this class project is to give students hands-on experience in parallel programming on both shared- and distributed-memory systems, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of a simulator of particles moving in free space.

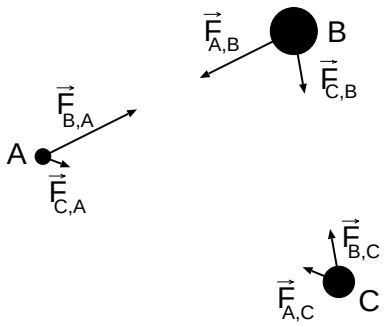
# 2 Problem Description

We consider a scenario where particles move freely in space and the only forces they are subjected to are due to each others gravity. To simplify, assume a 2D space, a square with side 1000 (the actual measurement units are not relevant).

The magnitude of the force between particles $A$ and $B$ is determined by the classical formula

$$F_{A,B} = F_{B,A} = G\frac{m_A \times m_B}{d_{A,B}^2}$$

where $m_A, m_B$ are the respective masses, $d_{A,B}$ is the distance between them and $G$ is the gravitational constant ($6,67408 \times 10^{-11}$). This is illustrated in the figure below.




Naturally, the force applied to each particle is the sum of the gravitational pull from all other particles. The resulting force determines the acceleration of the particle at each instant, which is used to determine, at each time-step, the new values of the particle's velocity and position:

$$\vec{F} = m.\vec{a} \qquad \vec{v}_{t+\Delta} = \vec{v}_t + \vec{a}_t.\Delta \qquad (x,y)_{t+\Delta} = (x,y)_t + \vec{v}_t.\Delta + \frac{1}{2}\vec{a}_t.\Delta^2$$

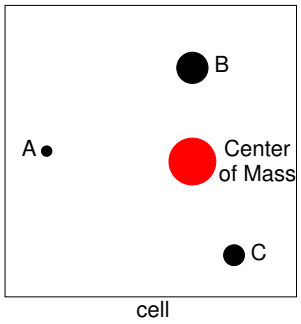
We will be using $\Delta = 0{,}1$.

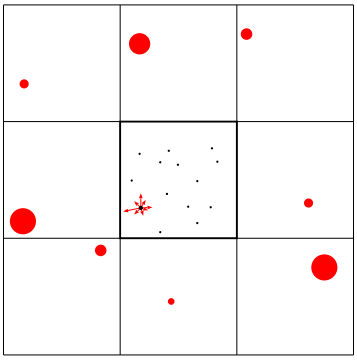
This problem, known as the $n$-body problem, with complexity $\Theta(n^2)$, is computationally very expensive for a large number of particles $n$. To avoid this complexity, for this assignment we use an approximation called particle-in-cell (PIC). It works by discretizing the space in a grid (in our case, 2D square cells). The particles in each cell define the center of mass of the cell $M$ (the sum of the masses of the $n$ particles in the cell) and coordinates computed as

$$M = \sum_{i=1}^{n} m_i \qquad (X,Y) = \frac{1}{M}\sum_{i=1}^{n} m_i \times (x_i, y_i).$$

Now the force on a particle is calculated only considering the particles in the same cell and the centers of mass of the eight adjacent cells. The figure on the left below illustrates the position of the

center of mass in a cell. The figure on the right considers the space divided in just $3 \times 3$ cells, indicates the center of masses in the adjacent cells and exemplifies the forces they cause on a single particle.



In our simulation we will consider that if two particles are within a distance $\epsilon = 0{,}005$ they collide and evaporate, hence both cease to exist and should not be considered for the rest of the simulation (to simplify the project, we ignore collisions between particles across cells).

The overall simulation is a sequence of time-steps, and each time-step consists of the following sequence of operations:

1. determine the center of mass of each cell;
2. compute the gravitational force applied to each particle;
3. calculate the new velocity and then the new position of each particle;
4. check for collisions.

We make the assumption that the sides wrap around, that is, if a particle exits through the side of the space it enters on the corresponding position on the opposite side of the space. Note that this also applies to the gravitational force, *e.g.*, a particle on a cell of the top is pulled upwards by the center of mass of the corresponding cell at the bottom.

The initial conditions, *i.e.*, mass, initial position and velocity of each particle, are defined by a routine `init_particles()` provided in the appendix of this document.

# 3 Implementation Details

## 3.1 Input Data

Your program should allow exactly five command-line parameters in this order:

1. seed for the random number generator
2. size of the side of the squared space of simulation
3. size of the grid (number of cells on each side)
4. number of particles
5. number of time-steps

The seed is a 31-bit integer required to initialize the random number generator used in the initialization routine, `init_particles()`. Note that we will be using positive and negative values here, where the sign indicates a uniform or a normal probability distribution, respectively. The size of the simulation space is a double. The remaining three values are all positive integers (will use the size of the grid $\geq 3$).

## 3.2 Output Data

The output of the program consists of two lines. The first line are the $x$ and $y$ coordinates of the final position of particle 0, two real values using three decimal digits. The second line is an integer indicating the number of particles that collided during the simulation. The submitted programs should send these output lines (and **nothing else!**) to the standard output, so that it can be validated against the correct solution. The project **cannot be graded** unless you strictly follow these rules!

## 3.3 Implementation Notes

Please consider the following set of recommendations:

- As stated before, the mass, initial position and velocity of each particle are defined by the routine `init_particles()` as provided (**use as is!** - may be with adjustments only to the names in the fields of the structure that describes the particles). The first three parameters are the first three parameters given in the command-line;
- To minimize numerical errors due to rounding, use the type `double` for real numbers;
- we may be running very large simulations, be sure to use **integers with 64 bits** for counters related to the particles and time-steps;
- For the formula that computes the gravitational force, use $G = 6{,}67408 \times 10^{-11}$;
- Consider a time-step $\Delta = 0{,}1$.

## 3.4 Sample Problem

To compute a single time-step in a instance problem with a 3×3 grid of size 2 and 10 particles, the command and respective output should be:

```
$ ./parsim 1 2 3 10 1
1.570 0.056
0
```

Other instances to help you validate your solution:

```
$ ./parsim 1 1 5 100 1
0.786 0.027
0
$ ./parsim -10 3 3 100 10
1.733 1.643
2
$ ./parsim -50 10000 200 500000 10
5025.384 5303.928
4
```

### 3.5 Measuring Execution Time

To make sure everyone uses the same measure for the execution time, the routine `omp_get_wtime()` from OpenMP will be used, which measures real time (also known as "wall-clock time"). This same routine should be used by all three versions of your project.

Hence, your programs should have a structure similar to this:

```
#include <omp.h>
<...>

int main(int argc, char *argv[])
{
    double exec_time;

    init_particles(...);
    exec_time = -omp_get_wtime();

    simulation();

    exec_time += omp_get_wtime();
    fprintf(stderr, "%.1fs\n", exec_time);

    print_result();        // to the stdout!
 }
```

In this way the execution time only accounts for the algorithm running time, and is sent to the standard error, `stderr`. The use of these two output streams allows the validation of the results and analysis of execution time to be performed separately.

Because this time routine is part of OpenMP, you need the include `omp.h` and compile all your programs with the flag `-fopenmp`.

## 4 Part 1 - Serial implementation

Write a serial implementation of the algorithm in C (or C++). Name the source file of this implementation `parsim.c`. As stated, your program should expect exactly five command-line input parameters.

This will be your base for comparisons and it is expected that it is as efficient as possible.

## 5 Part 2 - OpenMP implementation

Write an OpenMP implementation of the algorithm, with the same rules and input/output descriptions. Name this source code `parsim-omp.c`. You can start by simply adding OpenMP directives, but you are free, and encouraged, to modify the code in order to make the parallelization more effective and more scalable. Be careful about synchronization and load balancing!

Do not have hardcoded in your program the number of threads to use (unless for some reason your parallel strategy requires so) because we will be using the environment variable `OMP_NUM_THREADS` to evaluate the scalability of your program.

# 6 Part 3 - MPI implementation

Write an MPI implementation of the algorithm as for OpenMP, and address the same issues. Name this source code `parsim-mpi.c`.

For MPI, you will need to modify your code substantially. Besides synchronization and load balancing, you will need to create independent tasks, taking into account the minimization of the impact of communication costs. You are encouraged to explore different approaches for the problem decomposition.

You may also consider a combined MPI+OpenMP implementation.

# 7 What to Turn in, and When

You must eventually submit the sequential and both parallel versions of your program (**remember to use the filenames indicated above**), and a table with the times to run the parallel versions on input data that will be made available (for 1, 2, 4 and 8 parallel tasks for both OpenMP and MPI, and additionally 16, 32 and 64 for MPI).

For both the OpenMP and MPI versions (not for serial), you must also submit a short report about the results (2-4 pages) that discusses:

- the approach used for parallelization
- what decomposition was used
- what were the synchronization concerns and why
- how was load balancing addressed
- what are the performance results, and are they what you expected

The code, makefile and report will be uploaded to the Fenix system in a zip file. **Name these files** as `g<n>serial.zip`, `g<n>omp.zip` and `g<n>mpi.zip`, where `<n>` is your group number.

Two final notes:

- the command "make" should generate your executable, no flags/parameters should be needed. Also, the makefile should use the gcc compiler with a single optimization flag `-O2`;
- the parallel implementations should work with the OpenMP and MPI versions installed in the lab computers.

## 7.1 Deadlines

1st due date, serial version: **February 28th**, until 23:59.

2nd due date (OpenMP): **March 14th**, until 23:59.

3rd due date (MPI): **April 4th**, until 23:59.

# A Routine init_particles()

```
#define _USE_MATH_DEFINES
#include <math.h>
#define G 6.67408e-11
#define EPSILON2 (0.005*0.005)
#define DELTAT 0.1

unsigned int seed;
void init_r4uni(int input_seed)
{
    seed = input_seed + 987654321;
}
double rnd_uniform01()
{
    int seed_in = seed;
    seed ^= (seed << 13);
    seed ^= (seed >> 17);
    seed ^= (seed << 5);
    return 0.5 + 0.2328306e-09 * (seed_in + (int) seed);
}
double rnd_normal01()
{
    double u1, u2, z, result;
    do {
        u1 = rnd_uniform01();
        u2 = rnd_uniform01();
        z = sqrt(-2 * log(u1)) * cos(2 * M_PI * u2);
        result = 0.5 + 0.15 * z;      // Shift mean to 0.5 and scale
    } while (result < 0 || result >= 1);
    return result;
}

void init_particles(long seed, double side, long ncside, long long n_part, part
{
    double (*rnd01)() = rnd_uniform01;
    long long i;

    if(seed < 0){
        rnd01 = rnd_normal01;
        seed = -seed;
    }

    init_r4uni(seed);

    for(i = 0; i < n_part; i++) {
        par[i].x = rnd01() * side;
```

```
        par[i].y = rnd01() * side;
        par[i].vx = (rnd01() - 0.5) * side / ncside / 5.0;
        par[i].vy = (rnd01() - 0.5) * side / ncside / 5.0;

        par[i].m = rnd01() * 0.01 * (ncside * ncside) / n_part / G * EPSILON2;
    }
}
```