TÉCNICO LISBOA

Parallel and Distributed Computing

Project Assignment

PARTICLES SIMULATION

Version 1.1 (25/03/2019)

2018/2019

2nd Semester

Contents

Revisions

Version 1.0 (February 25, 2019)	Initial Version
Version 1.1 (March 25, 2019)	Changed value of epsilon from 0,01 to 0,0005

1 Introduction

The purpose of this class project is to give students hands-on experience in parallel programming on both UMA and multicomputer 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.

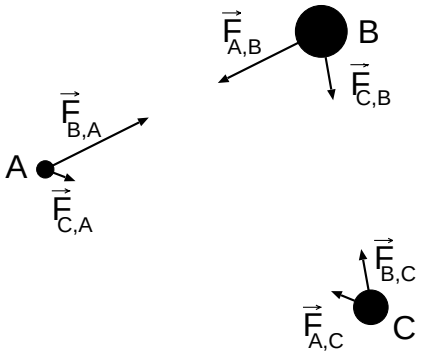
2 Problem Description

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 unit side.

The magnitude of the force between particles A and B is determined by the classic 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, Δ, the new values of the particle's velocity and position:

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

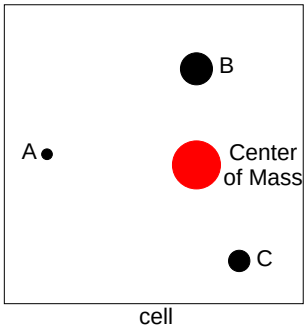
To simplify, assume $\Delta = 1$.

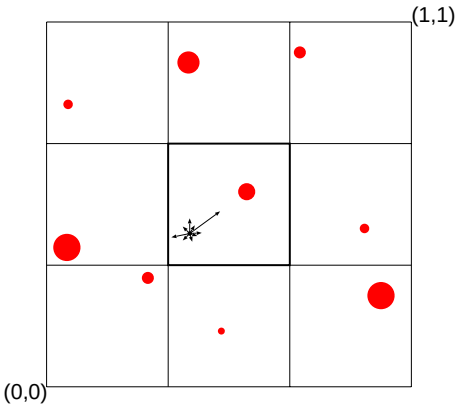
This problem, known as the n-body problem, with complexity $\Theta(n^2)$, is computationally very expensive for a large number of particles n (and hard to parallelize, since all computations depend on the previous result of all other computations!).

To avoid this complexity, for this assignment we use an approximation called particle-in-cell (PIC). It works by discretizing the space in cells (in our case, 2D squares). The particles in each cell define the center of mass of the cell, with a mass M which is the sum of the masses of the particles in the cell, and coordinates computed as

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

Now the force on a particle is calculated only from the centers of masses of its current and eight adjacent cells. The figure on the left below exemplifies the position of the center of mass in a cell. The figure on the right considers the space divided in just 9 cells, indicates the center of mass in each cell and depicts the forces applied to a single particle in the center cell.





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

- determine the center of mass of each cell;
- compute the gravitational force applied to each particle;
- calculate the new velocity and then the new position of each particle.

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. Although it may seem confusing, this scenario actually makes the computation simpler because it makes it uniform across all cells.

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.

The objective is to compute the position and velocities of the particles for a given number of time-steps.

3 Implementation Details

3.1 Input Data

Your program should allow exactly four command-line parameters, all positive integers, in this order:

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

3.2 Output Data

The output of the program consists of two lines, both with two real numbers using two decimal digits. The first line are the x and y coordinates of the final position of particle 0, and the second line are the x and y coordinates of the center of mass of the overall space at the end of 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 follow strictly these input and output 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!**). 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.
- For the formula that computes the gravitational force, use $G = 6,67408 \times 10^{-11}$.
- If the particle is too close to the center of mass, the force becomes too large. To avoid this, if the distance d is below a certain threshold ϵ, $d < \epsilon$, consider that the force is zero. We will use $\epsilon = 0,0005$.

The program must adhere to these rules in order for the results to be correctly validated!

3.4 Sample Problem

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

```
$ simpar 1 3 10 1
0.87 0.42
0.55 0.59
$
```

Other instances to allow you to validate your solution:

```
$ simpar 1 3 1000000 20
0.36 0.93
0.50 0.50
$ simpar 1 10 2000000 10
0.92 0.47
0.50 0.50
$ simpar 1 30 20000000 10
0.87 0.42
0.50 0.50
```

4 Part 1 - Serial implementation

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

This will be your base for comparisons and must be 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 `simpar-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!

6 Part 3 - MPI implementation

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

Extra credits will be given to groups that present 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 (**please 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). Note that we will **not** be using any level of compiler optimizations to evaluate the performance of your programs, so you shouldn't also.

You must also submit a short report about the results (1-2 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

You will turn in the serial version and OpenMP parallel version at the first due date, with the short report, and then the serial version again (hopefully the same) and the MPI parallel version at the second due date, with an updated report. Both the code and the report will be uploaded to the Fenix system in a zip file. **Name these files** as `g<n>omp.zip` and `g<n>mpi.zip`, where `<n>` is your group number.

1st due date (serial + OMP): **April 5th**, until 5pm.
Note: your project will be tested in the lab class just after the due date.

2nd due data (serial + MPI): **May 17th**, until 5pm.
Note: your project will be tested in the lab class just after the due date.

A Routine init_particles()

```
#define RND0_1 ((double) random() / ((long long)1<<31))
#define G 6.67408e-11
#define EPSLON 0.0005

void init_particles(long seed, long ncside, long long n_part, particle_t *par)
{
    long long i;

    srandom(seed);

    for(i = 0; i < n_part; i++)
    {
        par[i].x = RND0_1;
        par[i].y = RND0_1;
        par[i].vx = RND0_1 / ncside / 10.0;
        par[i].vy = RND0_1 / ncside / 10.0;

        par[i].m = RND0_1 * ncside / (G * 1e6 * n_part);
    }
}
```