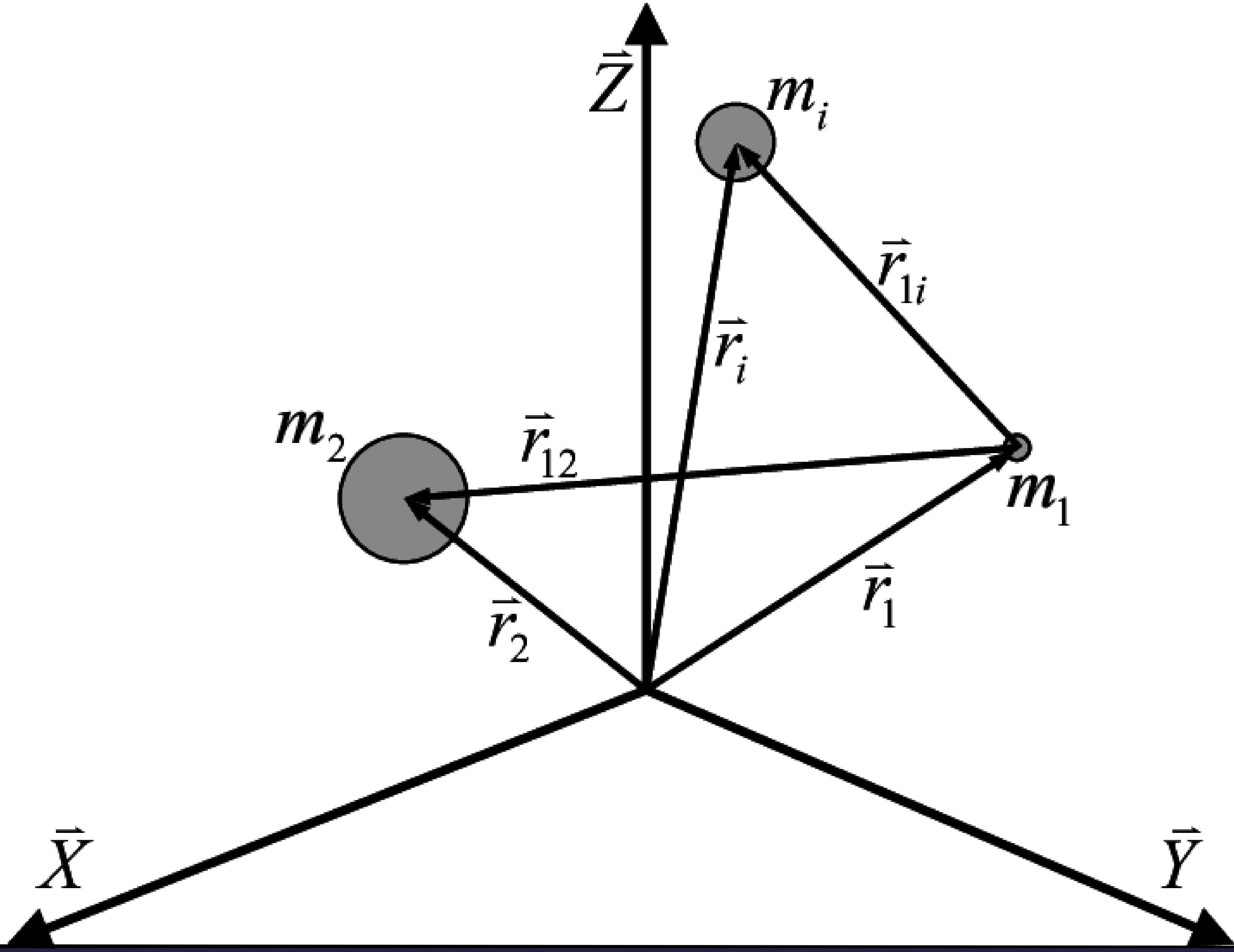
N-Body Problem

Boudarraja, Di Paola, Nuttini

Theorical Problem

Simulating a field of Force

- Electrical Field
- Magnetic Field
- Gravitational Forces



Mathematical Model

Newton’s Universal Gravitational Law

$$F_G = G \frac{m_1 m_2}{r^2} \hat{r}$$

Coefficient Matrix

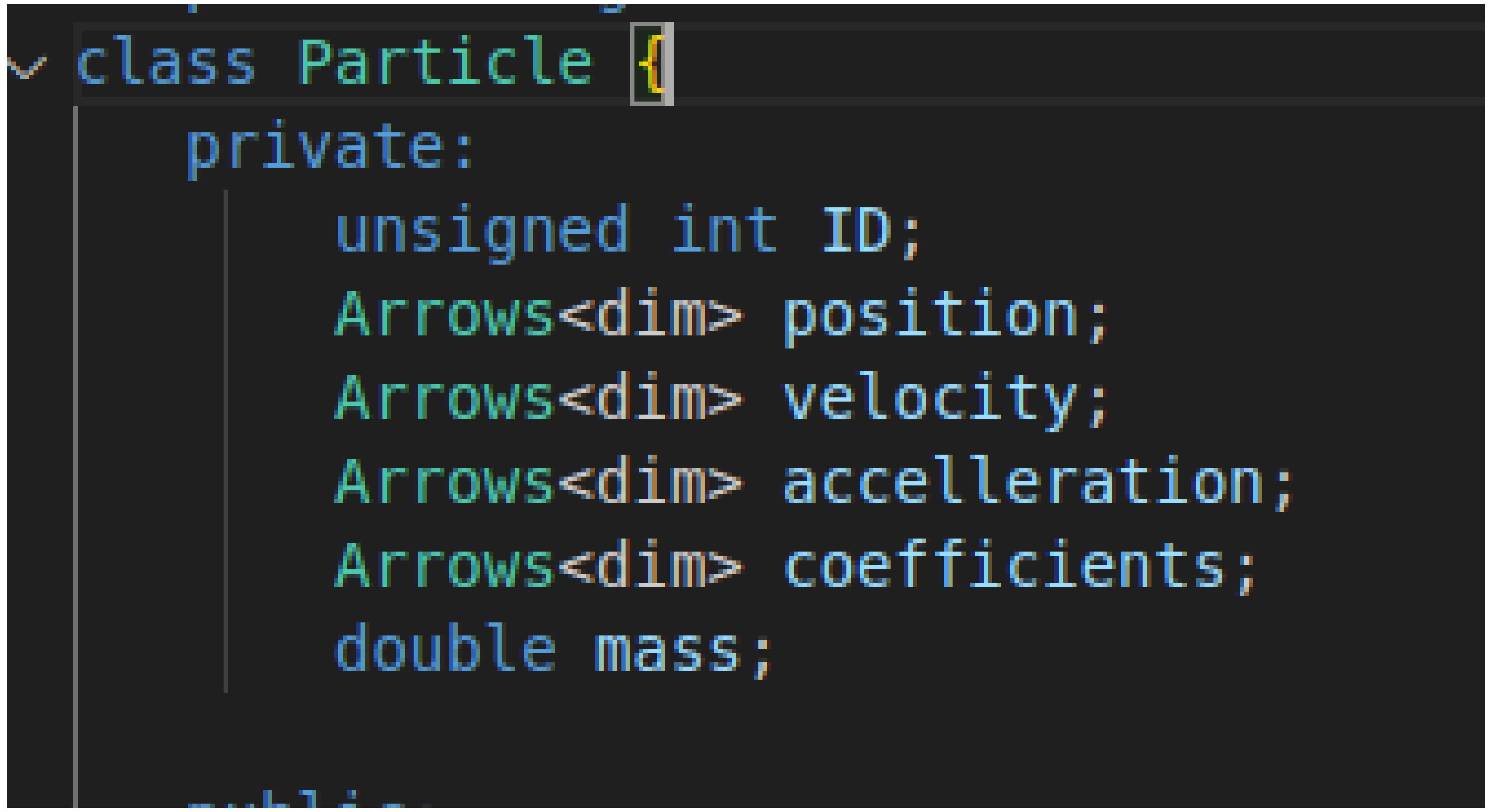
Coefficients calculated

$$a_{ij} = \frac{d_{ij} m_j}{\left\| d_{ij} \right\|}$$

$$\begin{bmatrix} 0 & a_{01} & \dots & a_{0j} \\ -a_{01} & 0 & \dots & a_{1j} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{i0} & -a_{i1} & \cdots & 0 \end{bmatrix}$$

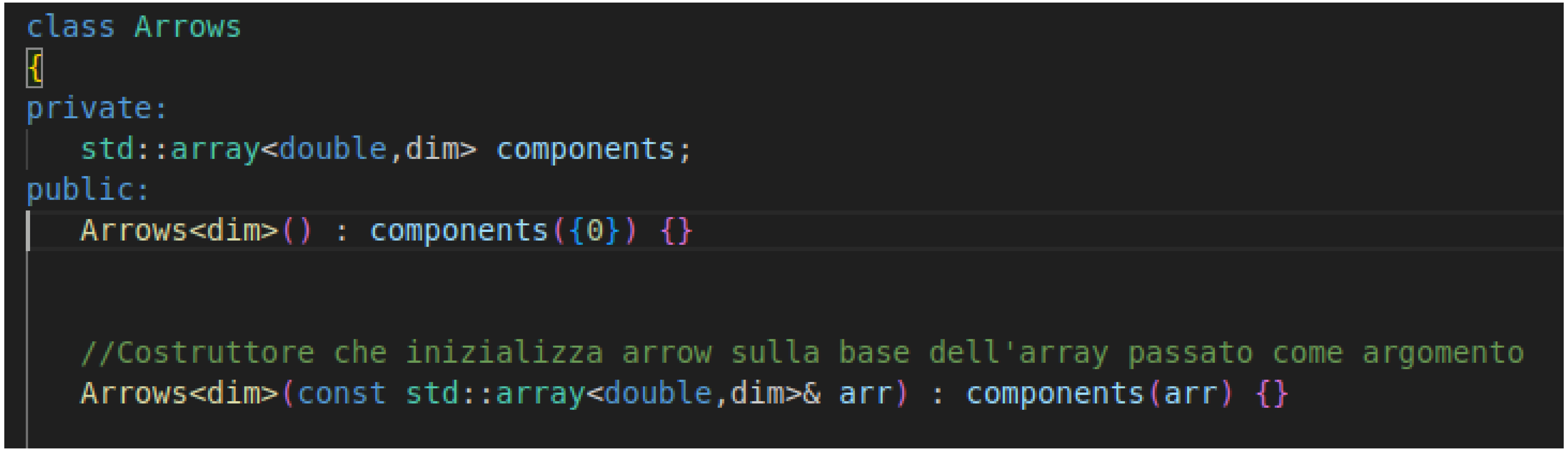
Attributes

Class: Particle



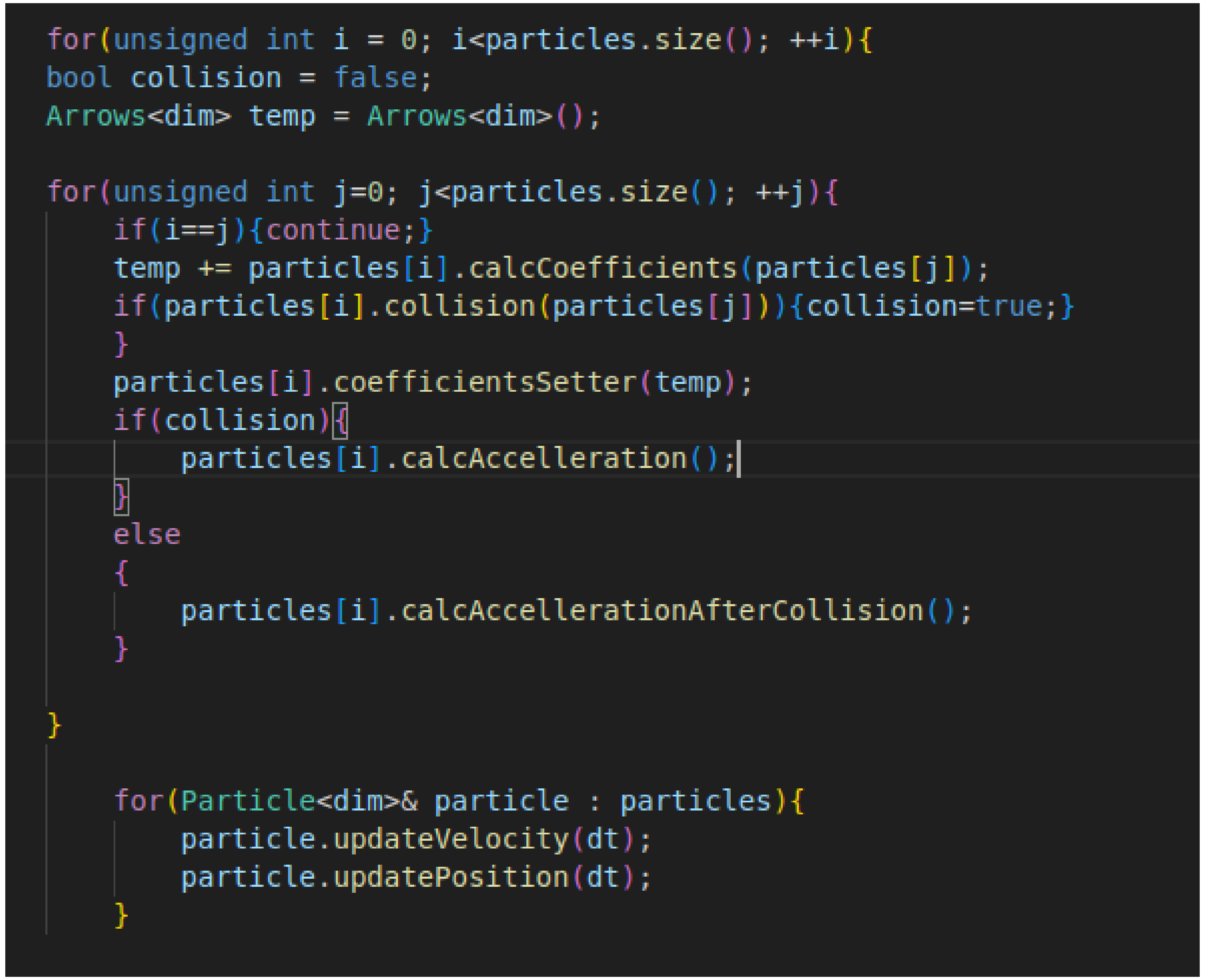
```
class Particle {
    private:
        unsigned int ID;
        Arrows<dim> position;
        Arrows<dim> velocity;
        Arrows<dim> accelleration;
        Arrows<dim> coefficients;
        double mass;

    public:
```

Arrows



```
class Arrows
{
private:
    std::array<double,dim> components;
public:
    Arrows<dim>() : components({0}) {}


    //Costruttore che inizializza arrow sulla base dell'array passato come argomento
    Arrows<dim>(const std::array<double,dim>& arr) : components(arr) {}

```

Serial implementation



```cpp
for(unsigned int i = 0; i<particles.size(); ++i){
bool collision = false;
Arrows<dim> temp = Arrows<dim>();

for(unsigned int j=0; j<particles.size(); ++j){
    if(i==j){continue;}
    temp += particles[i].calcCoefficients(particles[j]);
    if(particles[i].collision(particles[j])){collision=true;}
    }
    particles[i].coefficientsSetter(temp);
    if(collision){
        particles[i].calcAccelleration();
    }
    else
    {
        particles[i].calcAccellerationAfterCollision();
    }

}

    for(Particle<dim>& particle : particles){
        particle.updateVelocity(dt);
        particle.updatePosition(dt);
    }
```

Parallel Implementation

GENERATE PARTICLES

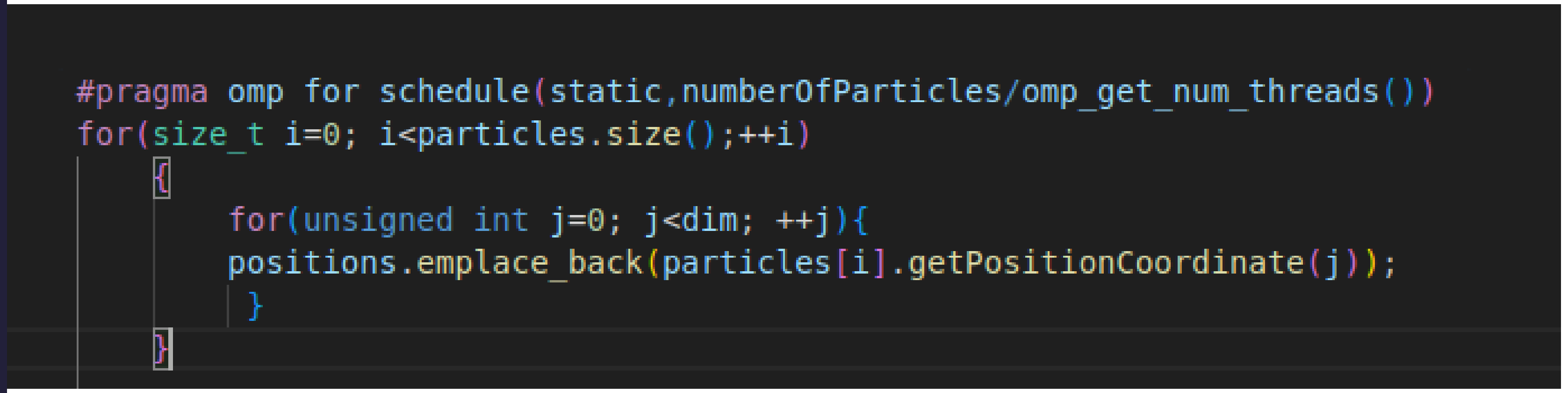
PROCESSING

PRINT POSITIONS

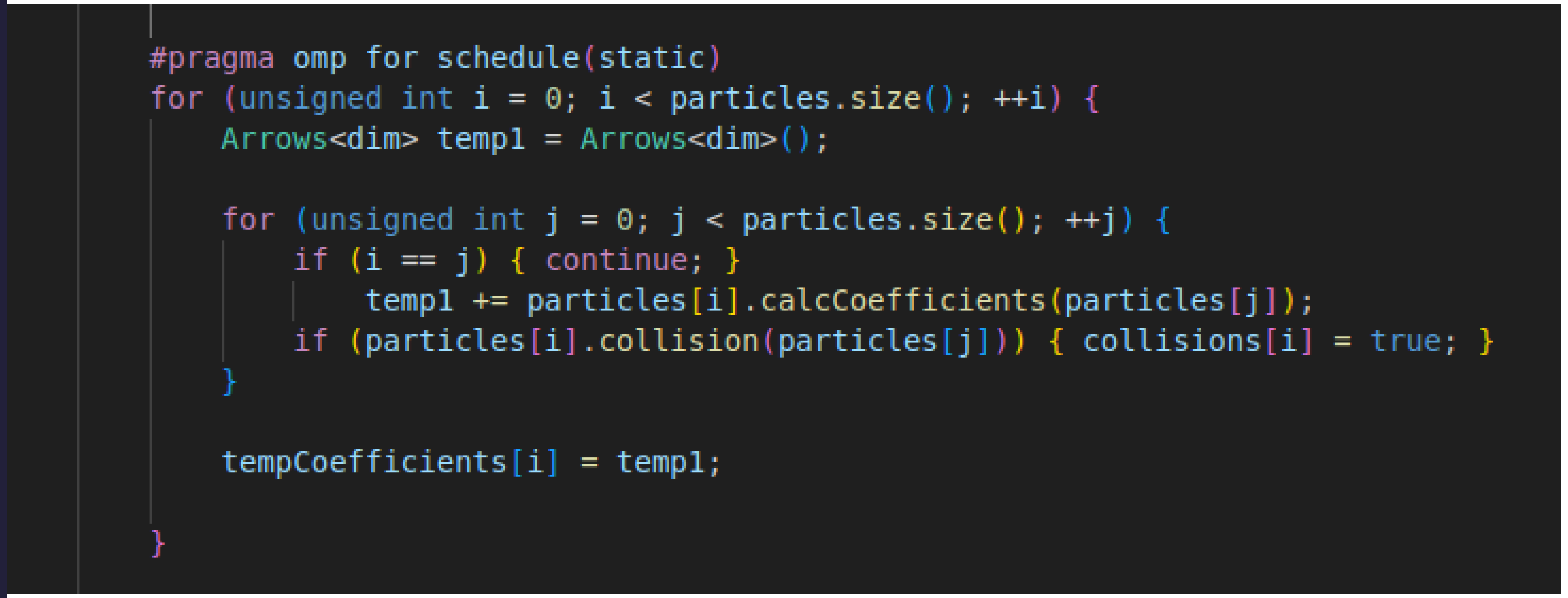
- Identifications of critical regions
- Identification of parallelizables processes
- choice of parallel Algorithm

Parallel Implementation

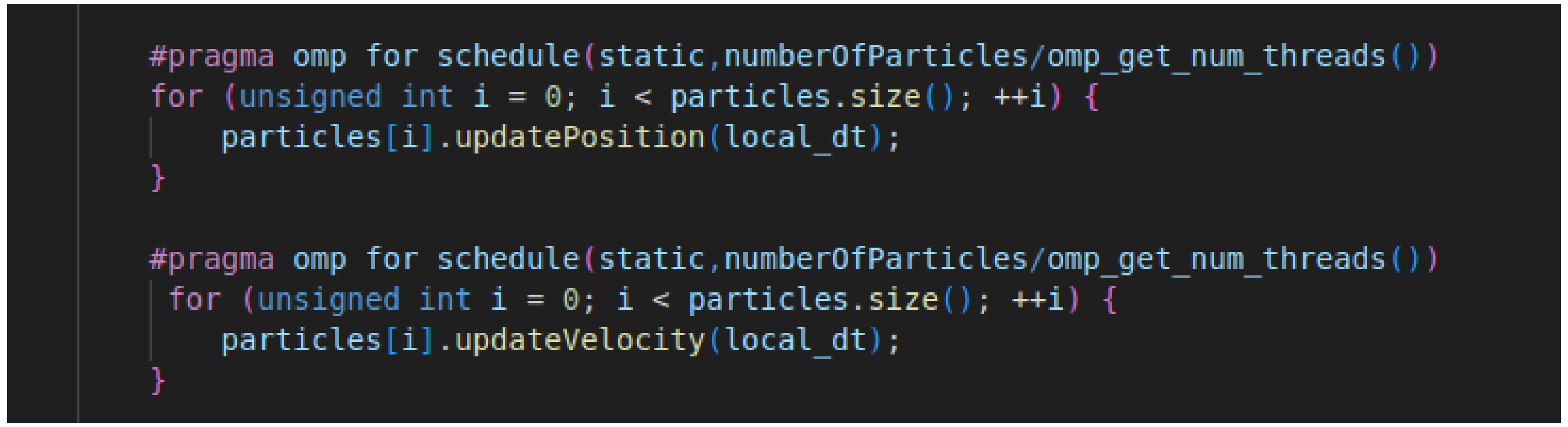
POLITECNICO
MILANO 1863



```
#pragma omp for schedule(static,numberOfParticles/omp_get_num_threads())
for(size_t i=0; i<particles.size();++i)
    {
        for(unsigned int j=0; j<dim; ++j){
        positions.emplace_back(particles[i].getPositionCoordinate(j));
         }
    }
```



```
#pragma omp for schedule(static)
for (unsigned int i = 0; i < particles.size(); ++i) {
    Arrows<dim> temp1 = Arrows<dim>();

    for (unsigned int j = 0; j < particles.size(); ++j) {
        if (i == j) { continue; }
            temp1 += particles[i].calcCoefficients(particles[j]);
        if (particles[i].collision(particles[j])) { collisions[i] = true; }
    }

    tempCoefficients[i] = temp1;

}
```



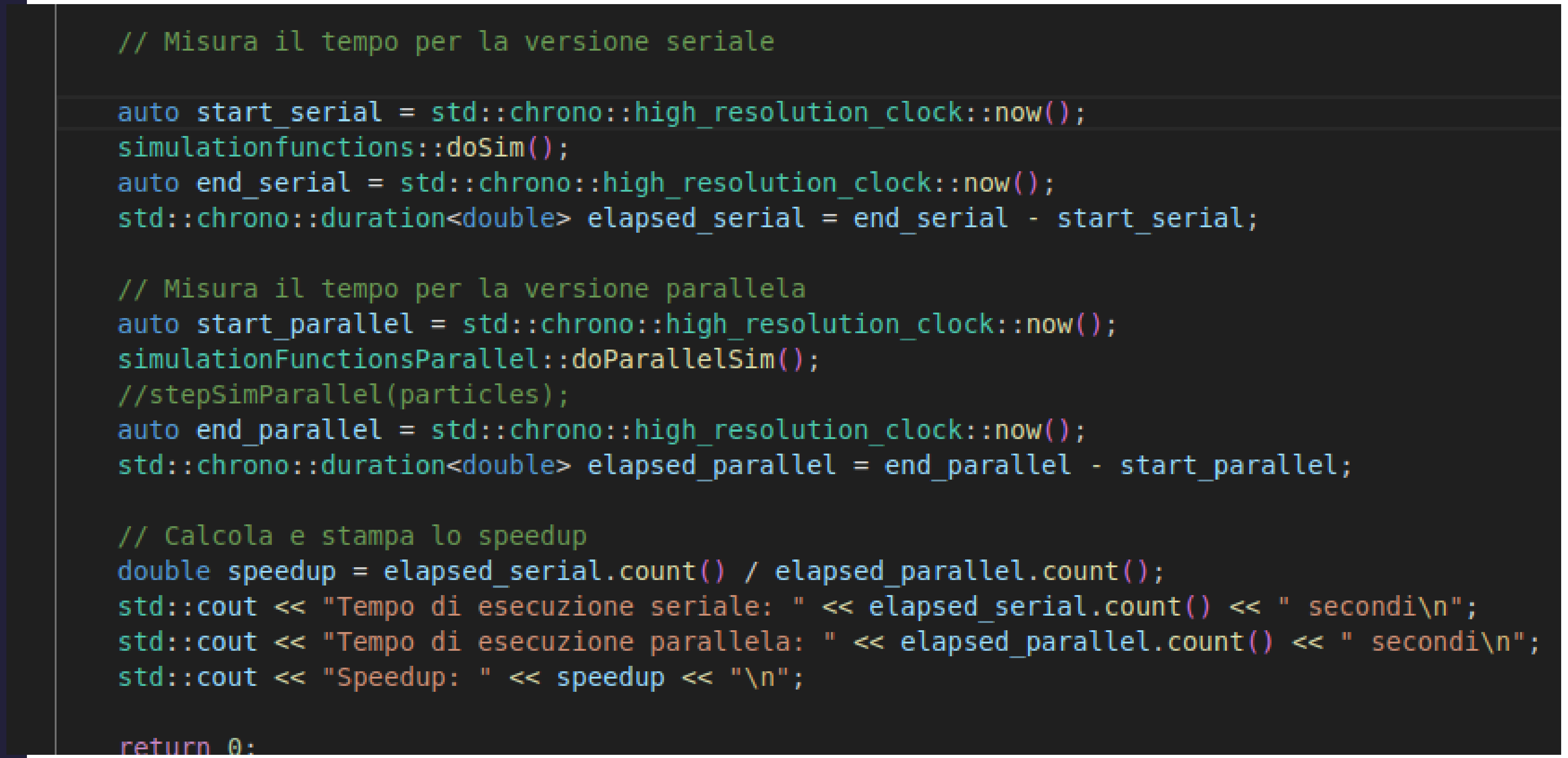
```
#pragma omp for schedule(static,numberOfParticles/omp_get_num_threads())
for (unsigned int i = 0; i < particles.size(); ++i) {
    particles[i].updatePosition(local_dt);
}

#pragma omp for schedule(static,numberOfParticles/omp_get_num_threads())
 for (unsigned int i = 0; i < particles.size(); ++i) {
    particles[i].updateVelocity(local_dt);
}
```

● **Coefficient initialization**

● **Coefficients update and setting**

● **Positions update**

Parallel Implementation

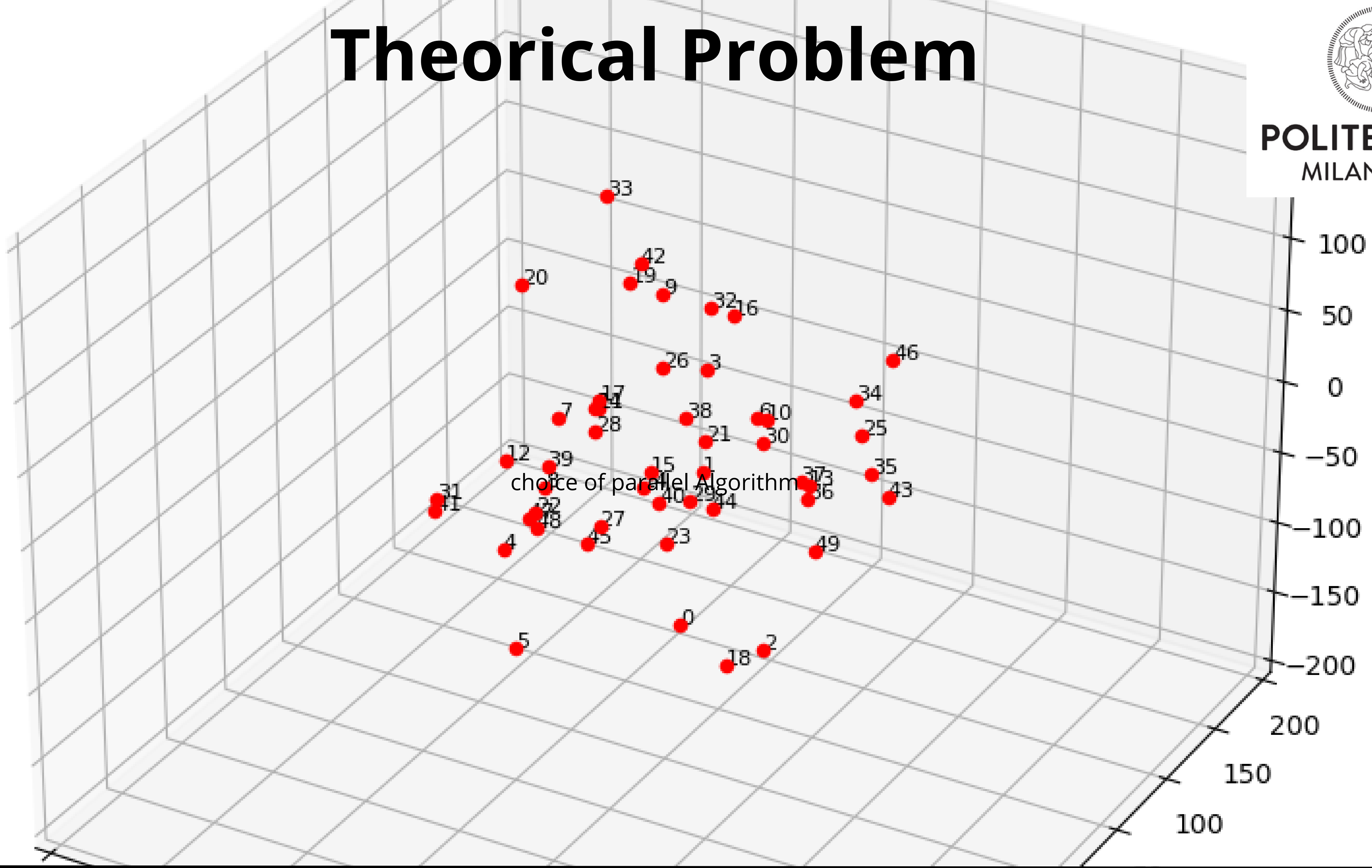


```
// Misura il tempo per la versione seriale

auto start_serial = std::chrono::high_resolution_clock::now();
simulationfunctions::doSim();
auto end_serial = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_serial = end_serial - start_serial;

// Misura il tempo per la versione parallela
auto start_parallel = std::chrono::high_resolution_clock::now();
simulationFunctionsParallel::doParallelSim();
//stepSimParallel(particles);
auto end_parallel = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_parallel = end_parallel - start_parallel;

// Calcola e stampa lo speedup
double speedup = elapsed_serial.count() / elapsed_parallel.count();
std::cout << "Tempo di esecuzione seriale: " << elapsed_serial.count() << " secondi\n";
std::cout << "Tempo di esecuzione parallela: " << elapsed_parallel.count() << " secondi\n";
std::cout << "Speedup: " << speedup << "\n";

return 0;
```

- **Performance evaluation**
- **Bottlenecks individuation**
- **Further implementation**

Theorical Problem

POLITECNICO
MILANO 1863



choice of parallel Algorithm