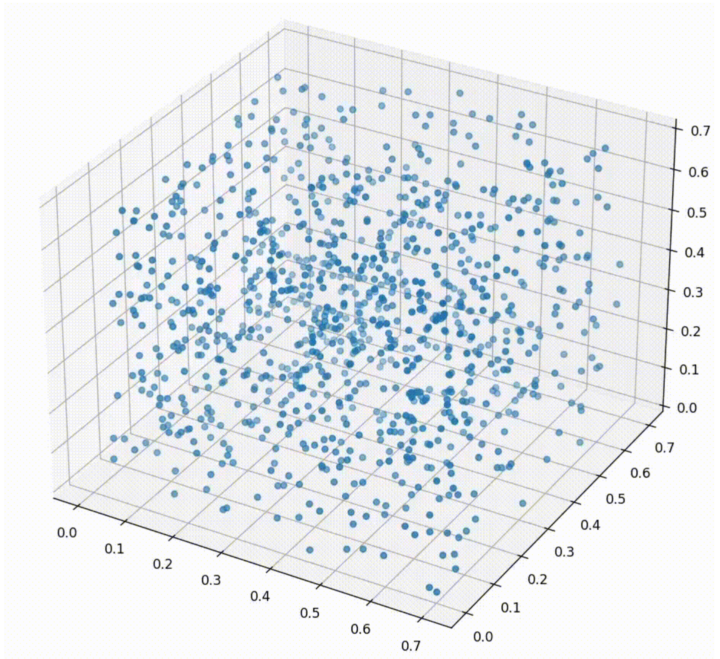
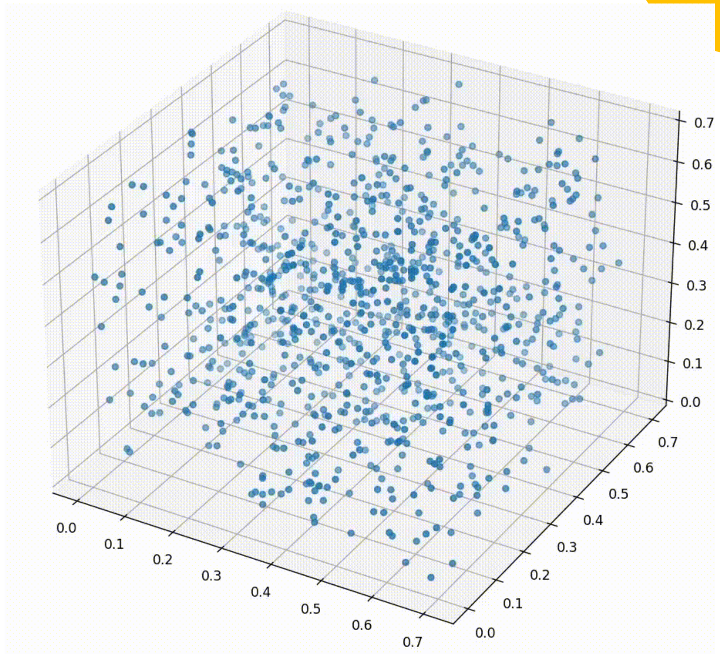
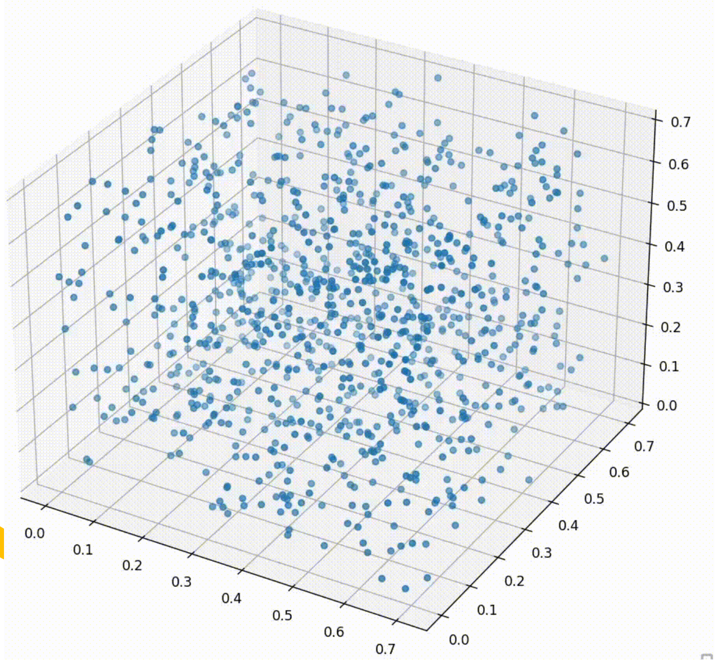
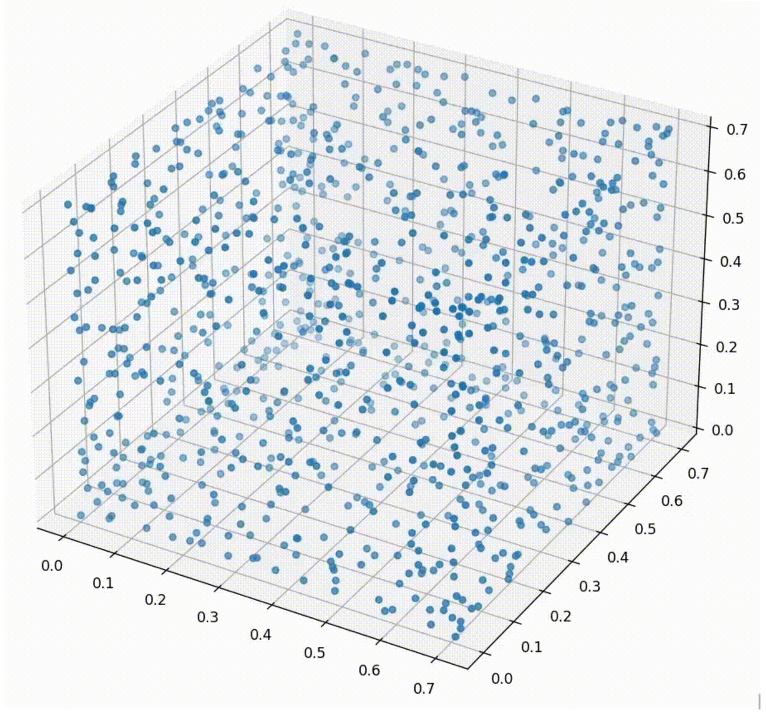
Valentina Moretti
Mattia Colbertaldo

NBODY

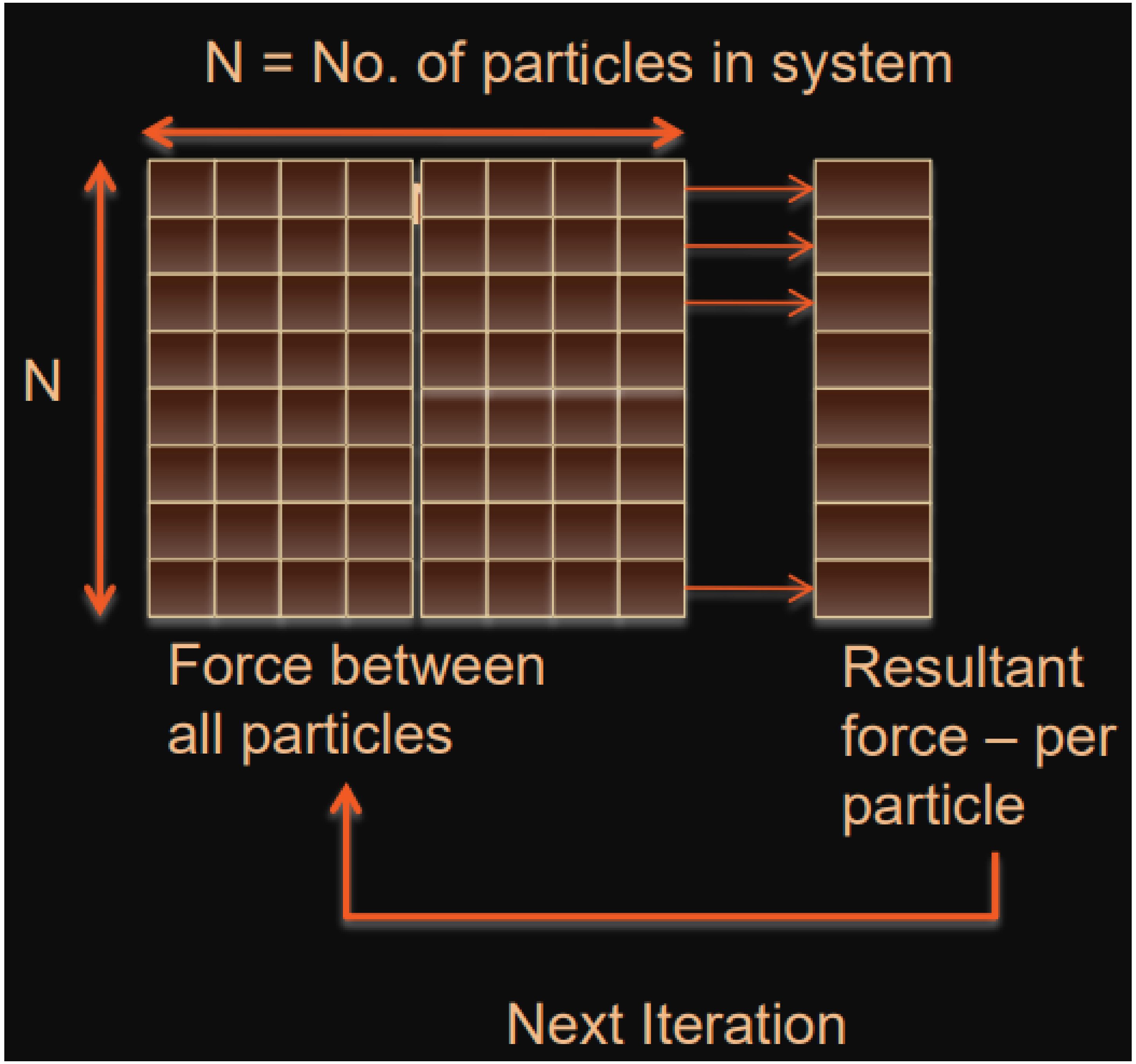

0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7

0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7

0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7

0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7

Work done

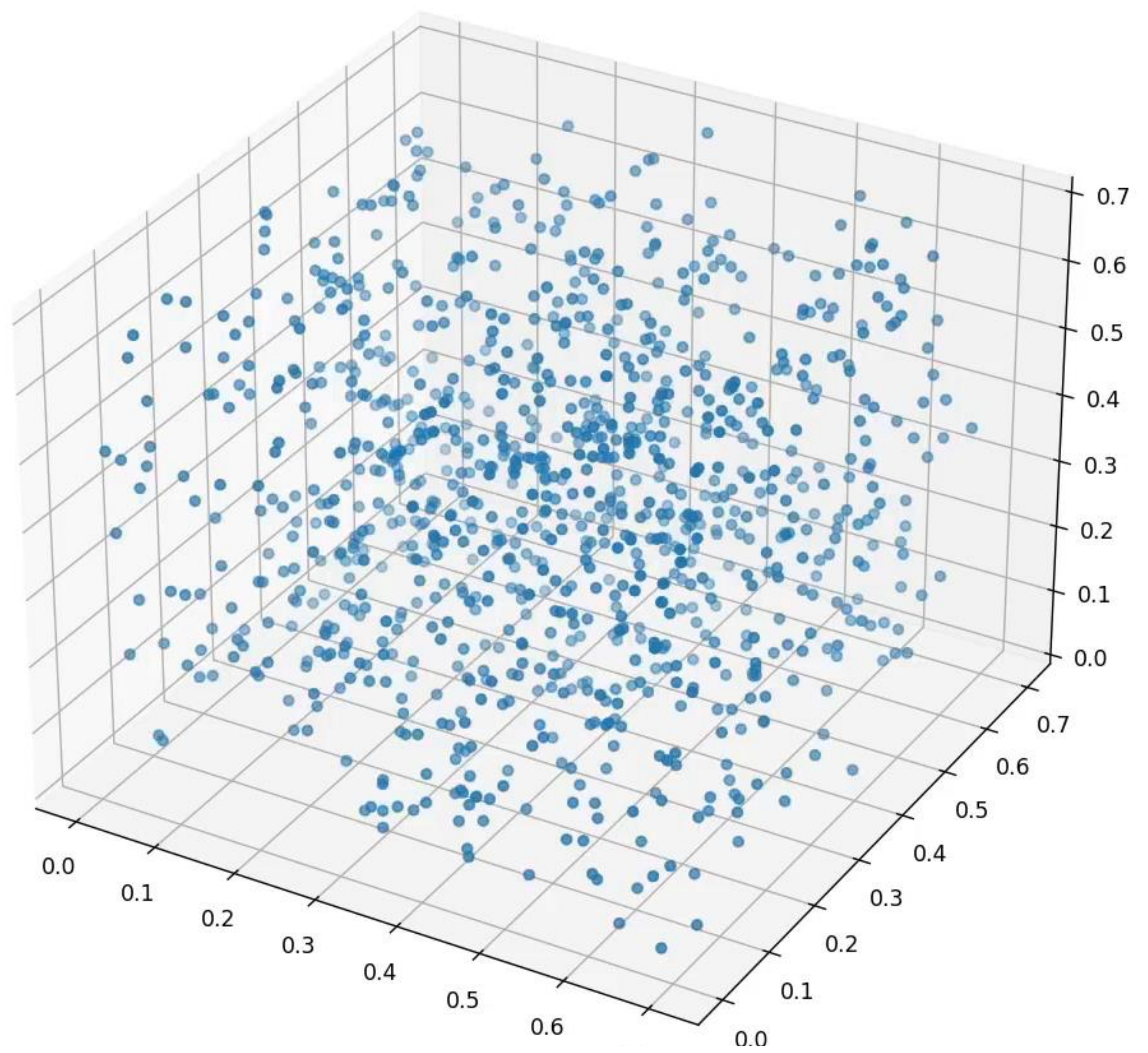
- From 2D to 3D
- Possibility of using different interaction forces: repulsive, gravitational, gravitational with assist, coulomb force (std::shared_ptr<AbstractForce> force;)
- Cuda
- Possibility of using different collision types in cuda and in Openmp

NBODY

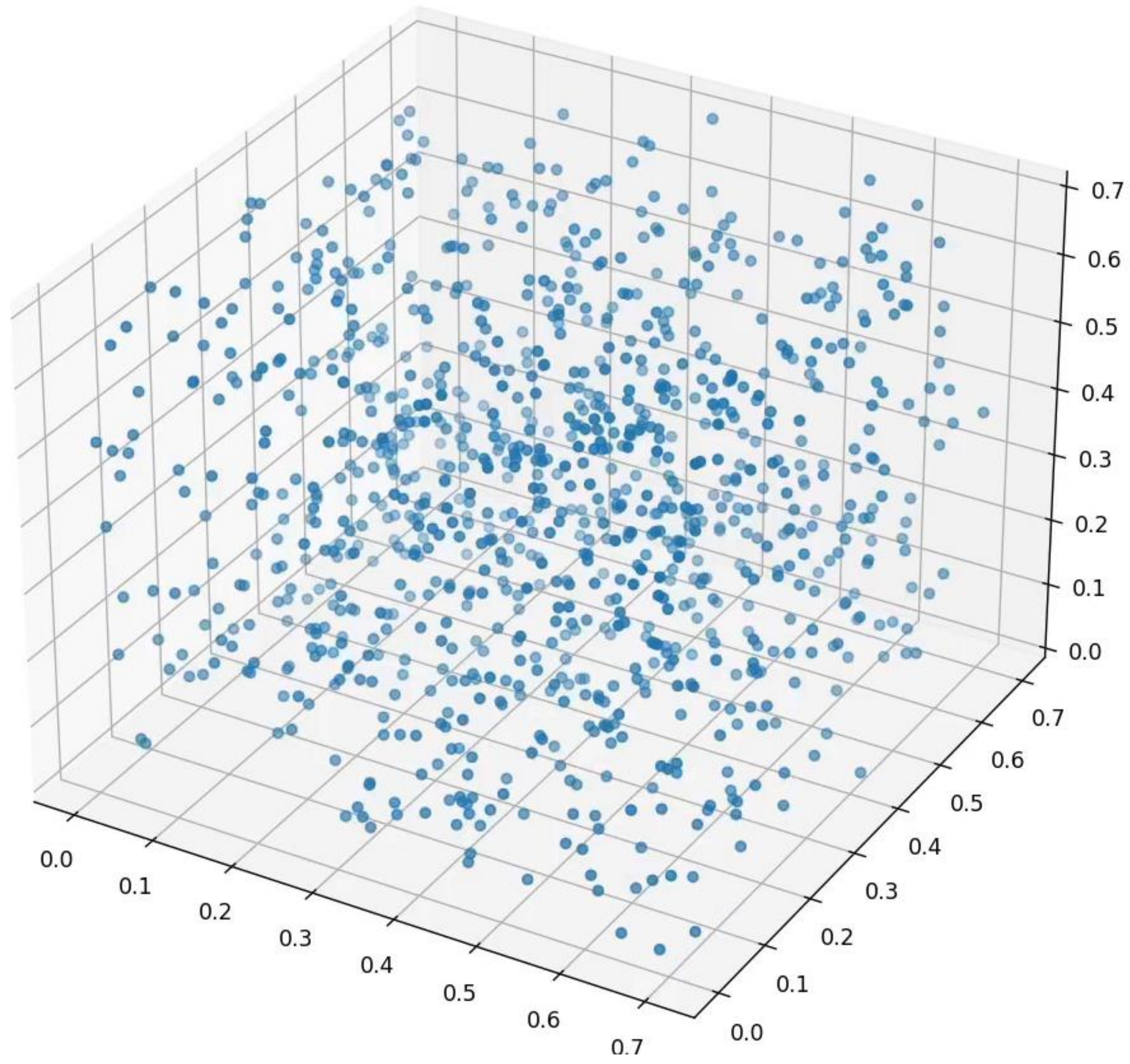
- An n-body simulation is a simulation of a system of particles under the influence of physical forces like gravity.
- Particle-particle interactions – simple, highly data parallel Algorithm
- Forces of each particle can be computed independently
- Accumulate results
- Add accumulated results to previous position of particles
- New position used as input to the next time step to calculate new forces acting between particles


N = No. of particles in system
N
Force between all particles
Resultant force – per particle
Next Iteration

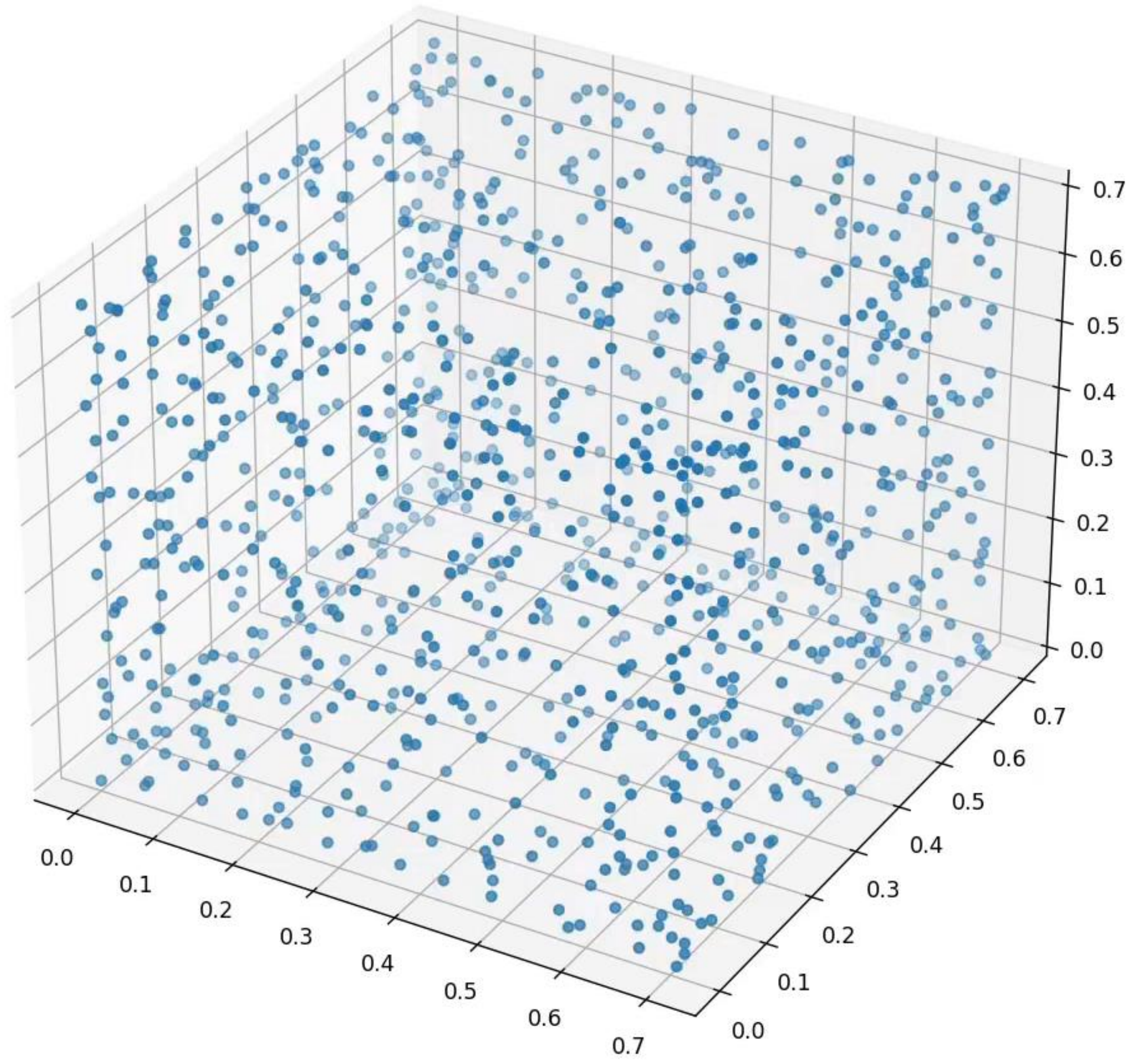
Gravitational force



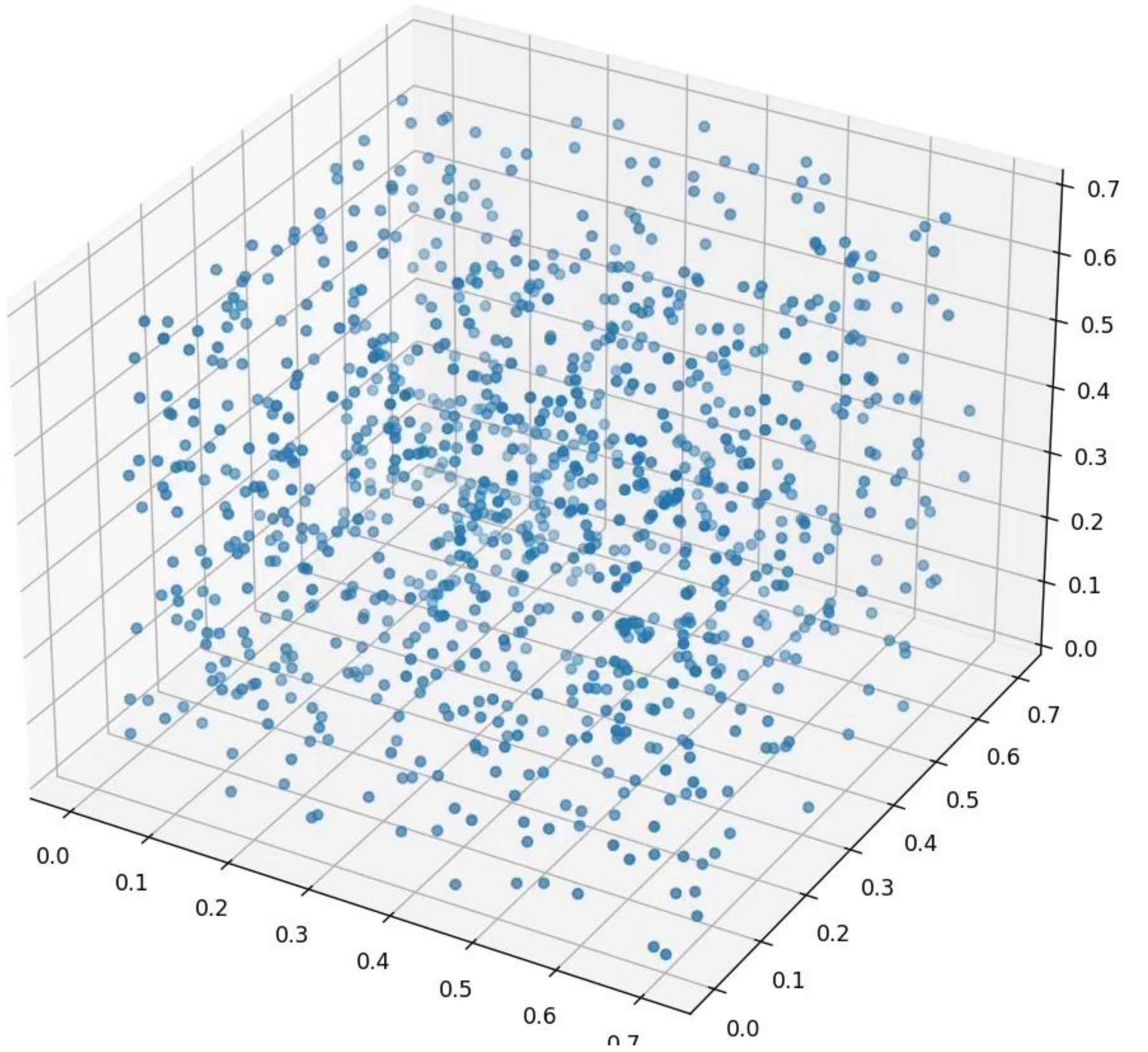
Coulomb force



Repulsive force + inelastic collision



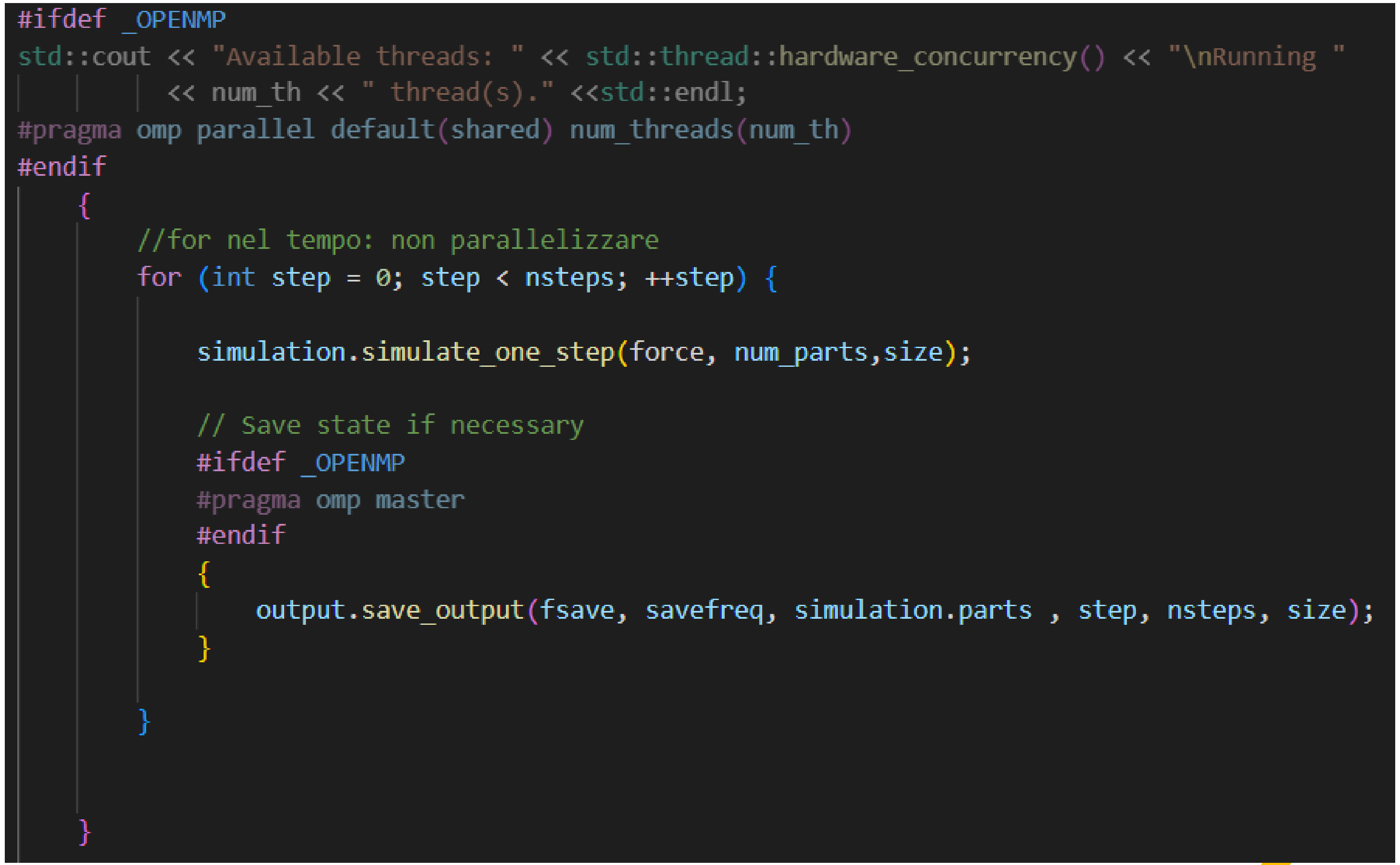
Proton force + elastic collision



OPENMP

Parallelization is achieved in this way:

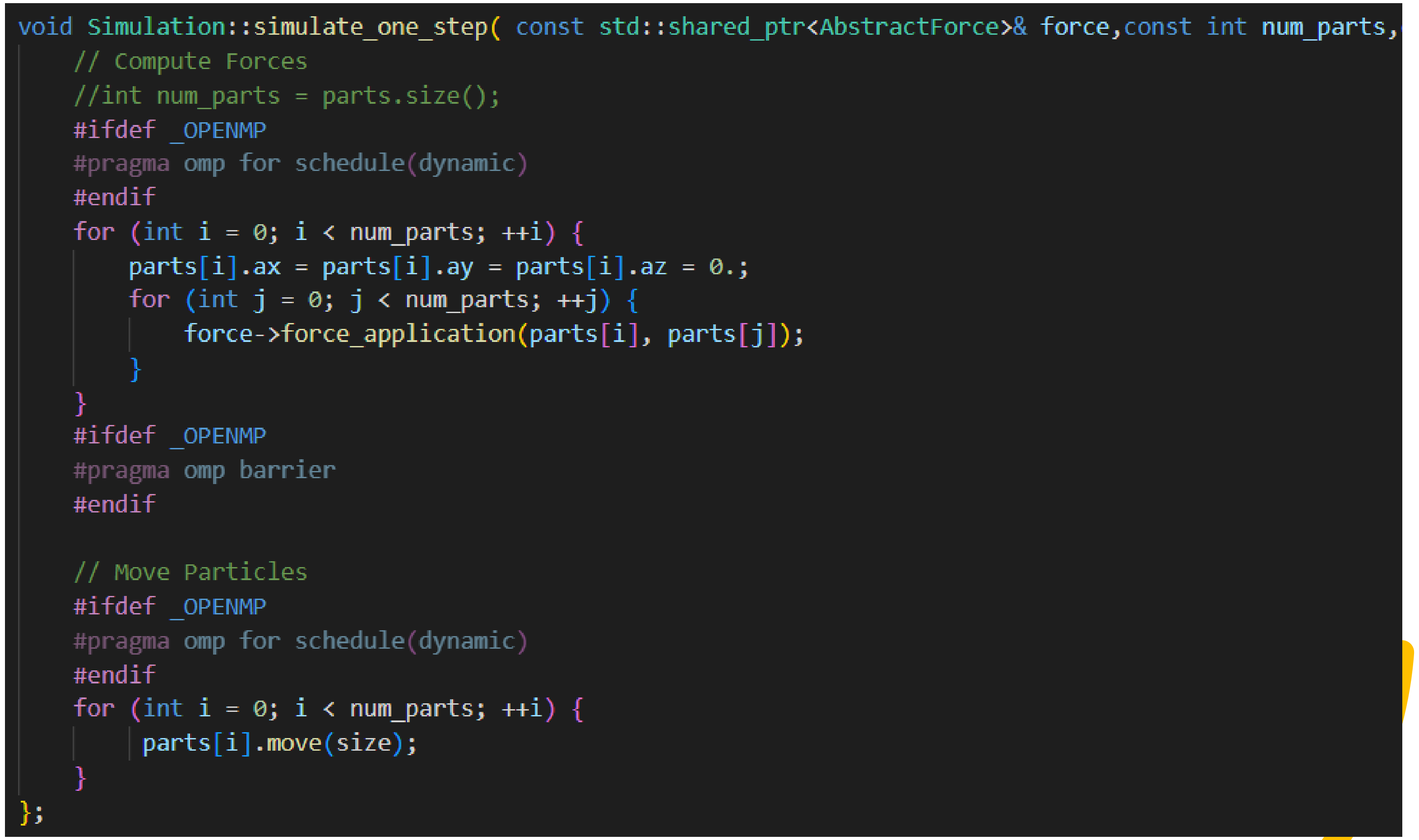
In main we have a temporal for loop which we don’t want to be parallel because it is referring about subsequent temporal window: we will save the positions of all particles once every “nstep” on an output file.



```
#ifdef _OPENMP
std::cout << "Available threads: " << std::thread::hardware_concurrency() << "\nRunning "
            << num_th << " thread(s)." <<std::endl;
#pragma omp parallel default(shared) num_threads(num_th)
#endif
    {
        //for nel tempo: non parallelizzare
        for (int step = 0; step < nsteps; ++step) {

            simulation.simulate_one_step(force, num_parts,size);

            // Save state if necessary
            #ifdef _OPENMP
            #pragma omp master
            #endif
            {
                output.save_output(fsave, savefreq, simulation.parts , step, nsteps, size);
            }

        }


    }
```

Parallelization is achieved in this way:

In simulate_one_step we perform two parallel loops: one for the calculation of the overall force applied on a particle p as the result of the interaction with all the other forces and one to change the positions of all particles.
Between the two loops we need a barrier to ensure that the final force to be applied on all particles is calculated before movin the particle itself.

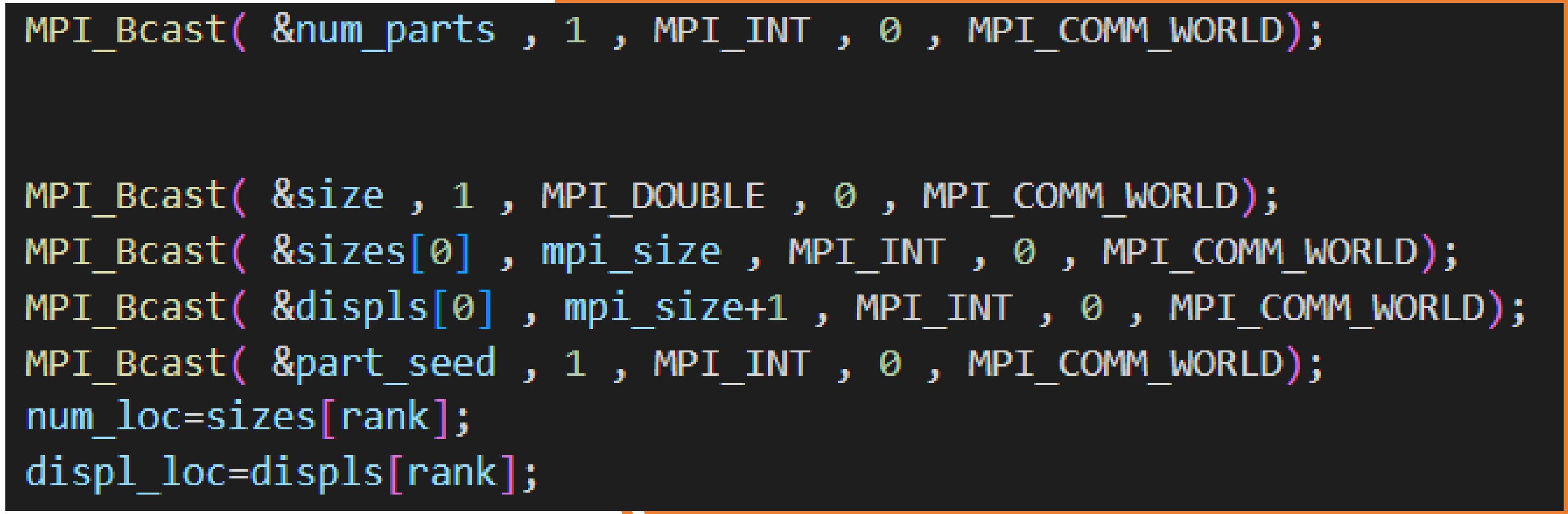


```
void Simulation::simulate_one_step( const std::shared_ptr<AbstractForce>& force,const int num_parts,
    // Compute Forces
    //int num_parts = parts.size();
    #ifdef _OPENMP
    #pragma omp for schedule(dynamic)
    #endif
    for (int i = 0; i < num_parts; ++i) {
        parts[i].ax = parts[i].ay = parts[i].az = 0.;
        for (int j = 0; j < num_parts; ++j) {
            force->force_application(parts[i], parts[j]);
        }
    }
    #ifdef _OPENMP
    #pragma omp barrier
    #endif

    // Move Particles
    #ifdef _OPENMP
    #pragma omp for schedule(dynamic)
    #endif
    for (int i = 0; i < num_parts; ++i) {
        parts[i].move(size);
    }
};
```

MPI

Communication:

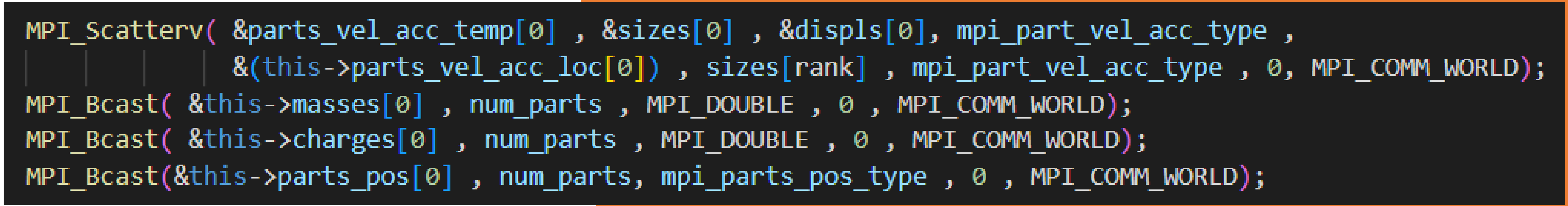
In main are broadcasted all the info needed by all processes



```
MPI_Bcast( &num_parts , 1 , MPI_INT , 0 , MPI_COMM_WORLD);


MPI_Bcast( &size , 1 , MPI_DOUBLE , 0 , MPI_COMM_WORLD);
MPI_Bcast( &sizes[0] , mpi_size , MPI_INT , 0 , MPI_COMM_WORLD);
MPI_Bcast( &displs[0] , mpi_size+1 , MPI_INT , 0 , MPI_COMM_WORLD);
MPI_Bcast( &part_seed , 1 , MPI_INT , 0 , MPI_COMM_WORLD);
num_loc=sizes[rank];
displ_loc=displs[rank];
```

Communication:

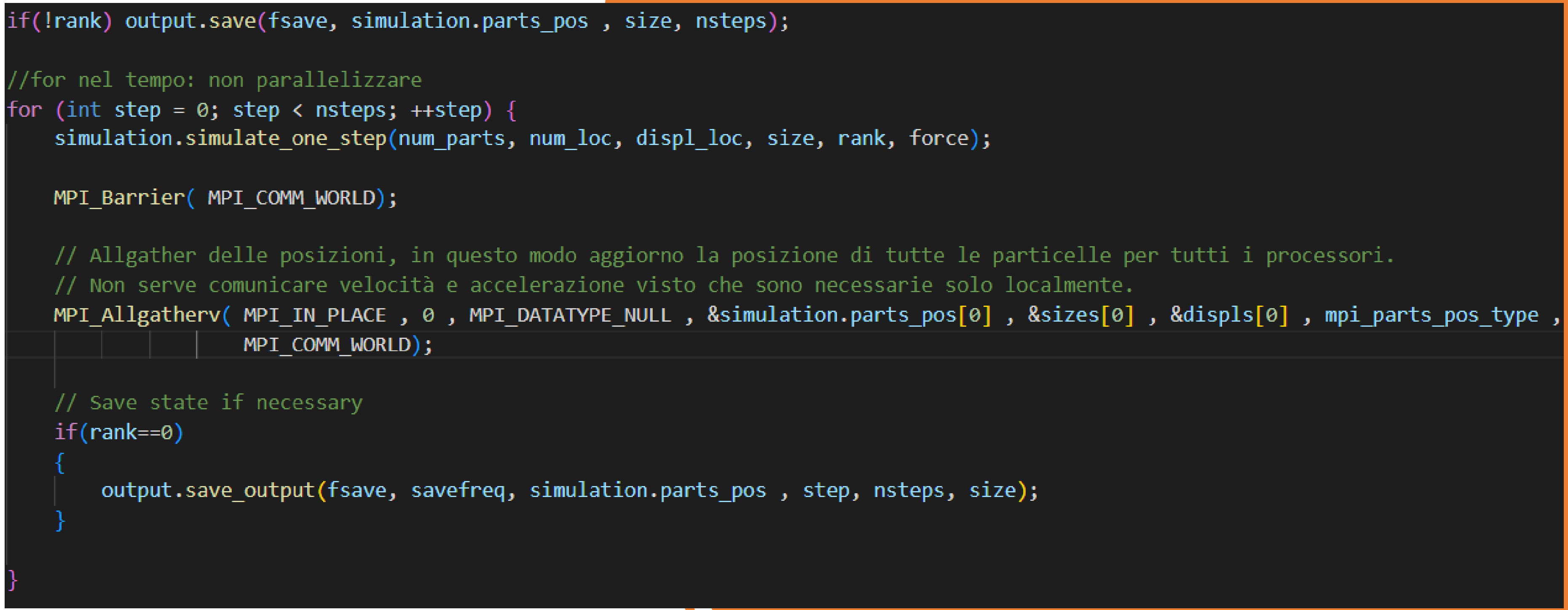
In init_particles (in Simulation.cpp):



```
MPI_Scatterv( &parts_vel_acc_temp[0] , &sizes[0] , &displs[0], mpi_part_vel_acc_type ,
              &(this->parts_vel_acc_loc[0]) , sizes[rank] , mpi_part_vel_acc_type , 0, MPI_COMM_WORLD);
MPI_Bcast( &this->masses[0] , num_parts , MPI_DOUBLE , 0 , MPI_COMM_WORLD);
MPI_Bcast( &this->charges[0] , num_parts , MPI_DOUBLE , 0 , MPI_COMM_WORLD);
MPI_Bcast(&this->parts_pos[0] , num_parts, mpi_parts_pos_type , 0 , MPI_COMM_WORLD);
```

Communication:

In main:



```
if(!rank) output.save(fsave, simulation.parts_pos , size, nsteps);

//for nel tempo: non parallelizzare
for (int step = 0; step < nsteps; ++step) {
    simulation.simulate_one_step(num_parts, num_loc, displ_loc, size, rank, force);

    MPI_Barrier( MPI_COMM_WORLD);

    // Allgather delle posizioni, in questo modo aggiorno la posizione di tutte le particelle per tutti i processori.
    // Non serve comunicare velocità e accelerazione visto che sono necessarie solo localmente.
    MPI_Allgatherv( MPI_IN_PLACE , 0 , MPI_DATATYPE_NULL , &simulation.parts_pos[0] , &sizes[0] , &displs[0] , mpi_parts_pos_type ,
                MPI_COMM_WORLD);

    // Save state if necessary
    if(rank==0)
    {
        output.save_output(fsave, savefreq, simulation.parts_pos , step, nsteps, size);
    }

}
```

Work scheduling:

Each process calculate the resulting force and movement of a subset of particles (given by sizes[rank] and displs[rank]). In order to do this, each process needs to know about the positions of all particles and it will fill a vector of accelerations and velocities of length sizes[rank]. This is done using a different data structure compared with Openmp. This decision was tanken in order to move less data as possible between processes: the only information that a process need to know about particles that doesn’t own to its assigned partition is the positions of the other particle (there is no point in sending also accelerations and velocities). In main after each call to simulate_one_step the info about the new position of particles calculate by the processors is gathered to all processors.

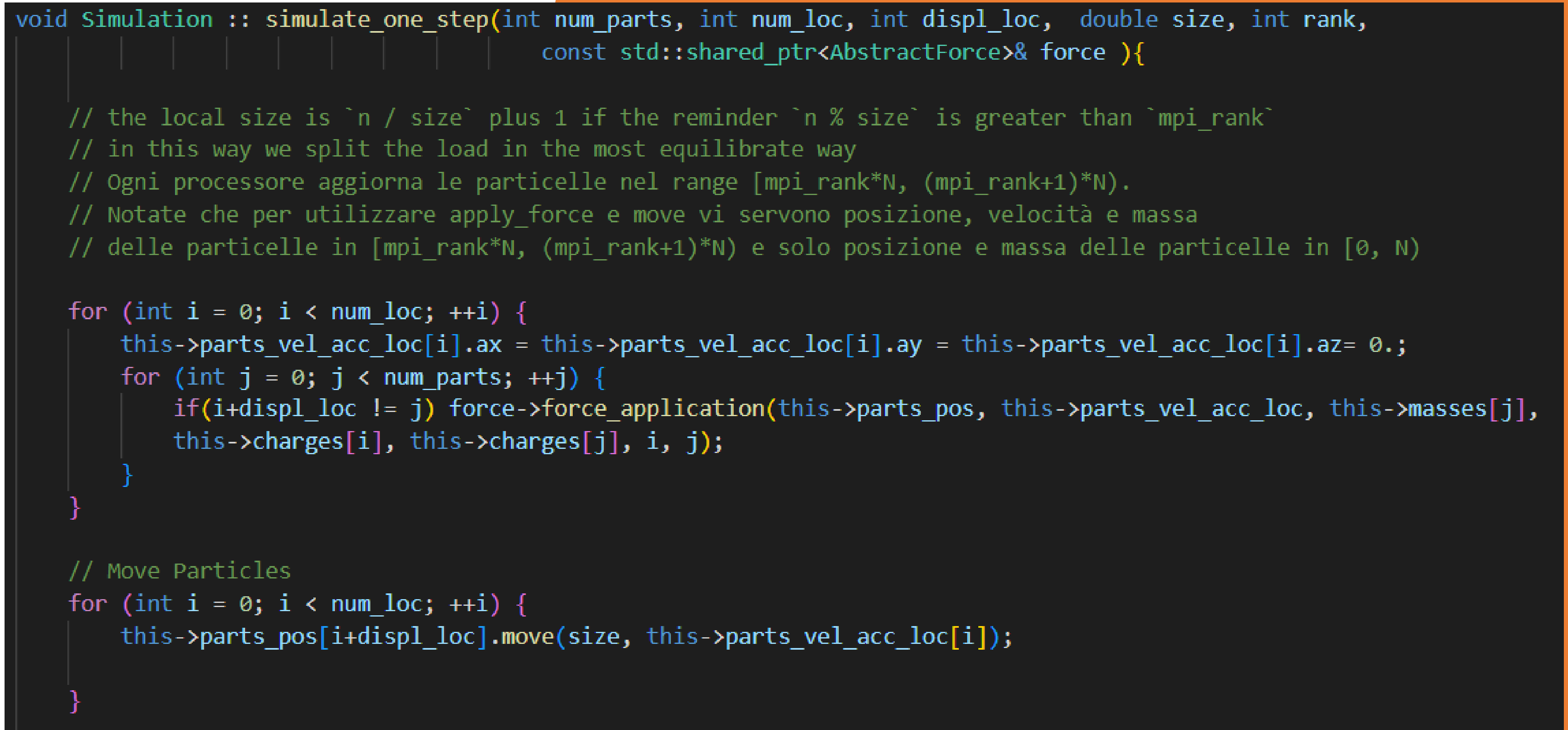
The only processor that is in charge of saving is the process with rank 0.

Computation and Parallelization are achieved in this way:

Each process calculate simulate_one_step on its partition of particle. In this case there is no need of the barrier between the for loop as they are executed sequentially on each processor and the same processor that has finished the calculation of a subset of particle is the one that will calculate the movement of the same subset of particles. So for sure the calculation of the resulting force on a given particle is finished before the function call to “move” on the same particle.

In both MPI e OpenMP the initialization is serialized: in the former case only rank 0 do it, in the latter we didn’t enclosed that part in a #pragma omp directive.

Computation and Parallelization are achieved in this way:



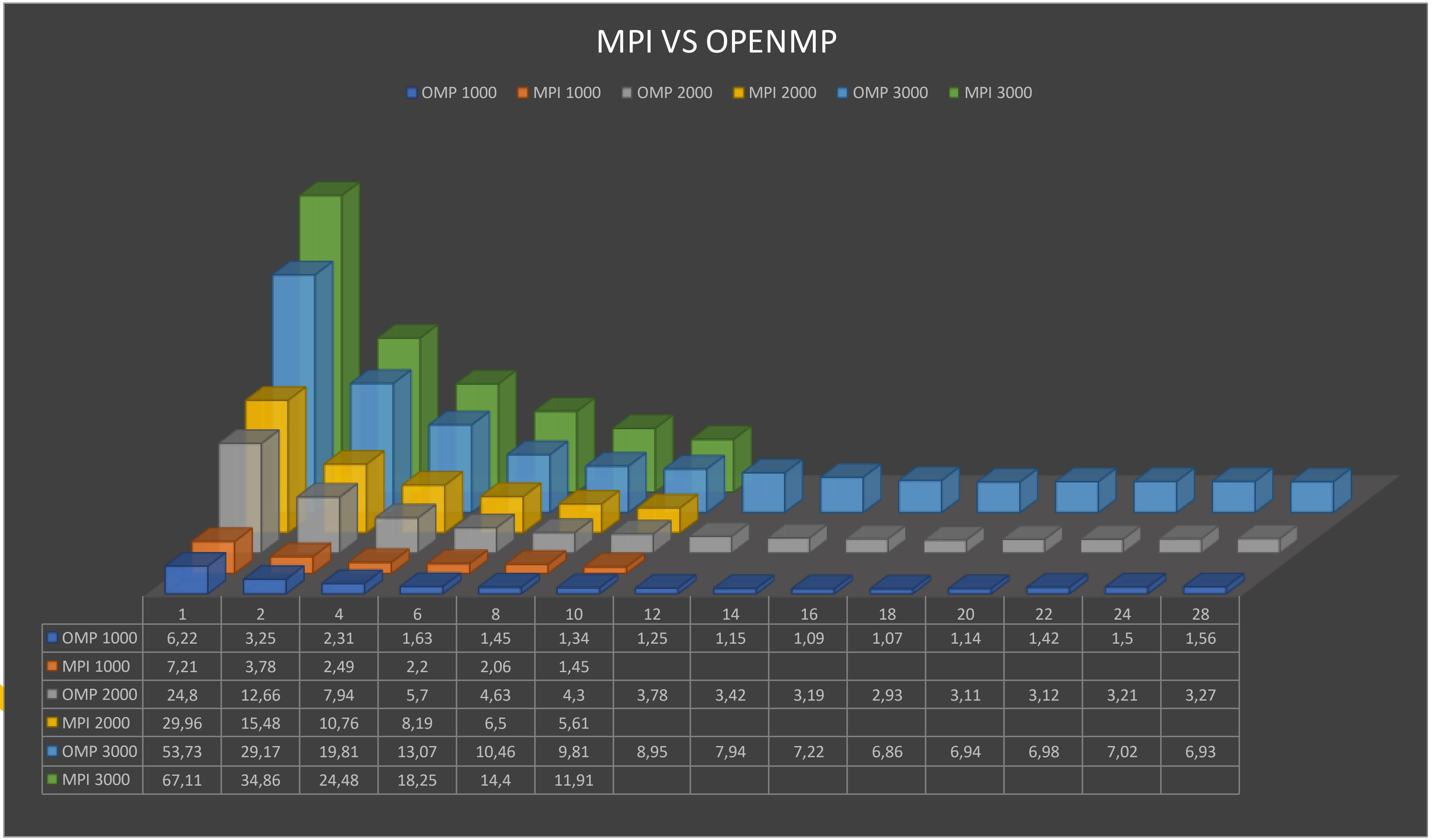
```cpp
void Simulation :: simulate_one_step(int num_parts, int num_loc, int displ_loc,  double size, int rank,
                                      const std::shared_ptr<AbstractForce>& force ){

    // the local size is `n / size` plus 1 if the reminder `n % size` is greater than `mpi_rank`
    // in this way we split the load in the most equilibrate way
    // Ogni processore aggiorna le particelle nel range [mpi_rank*N, (mpi_rank+1)*N).
    // Notate che per utilizzare apply_force e move vi servono posizione, velocità e massa
    // delle particelle in [mpi_rank*N, (mpi_rank+1)*N) e solo posizione e massa delle particelle in [0, N)

    for (int i = 0; i < num_loc; ++i) {
        this->parts_vel_acc_loc[i].ax = this->parts_vel_acc_loc[i].ay = this->parts_vel_acc_loc[i].az= 0.;
        for (int j = 0; j < num_parts; ++j) {
            if(i+displ_loc != j) force->force_application(this->parts_pos, this->parts_vel_acc_loc, this->masses[j],
            this->charges[i], this->charges[j], i, j);
        }
    }

    // Move Particles
    for (int i = 0; i < num_loc; ++i) {
        this->parts_pos[i+displ_loc].move(size, this->parts_vel_acc_loc[i]);

    }
```

Note:

We preferred not to implement collision in MPI to keep the good featuring of sending just small chunks of particles’ velocities. (Information of velocities of all particles is needed by a process to compute the effect of an elastic or anelastic collision).

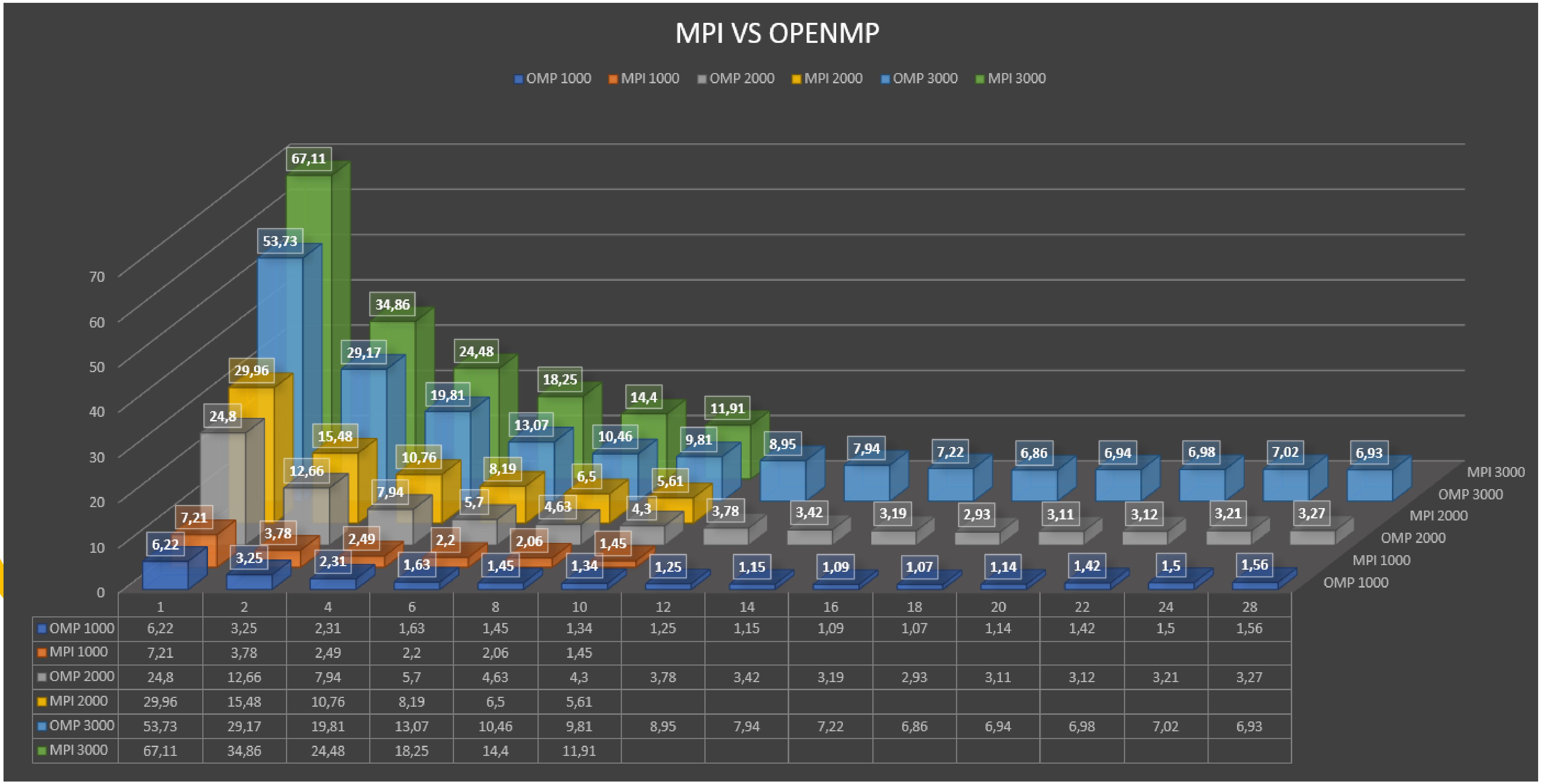
For the same reason in MPI we have chosen not to implement the optimization of the calculation of just the lower triangular part of the matrix representing interactions (while in Openmp we have implemented it).

Performance results on the 2D implementation:



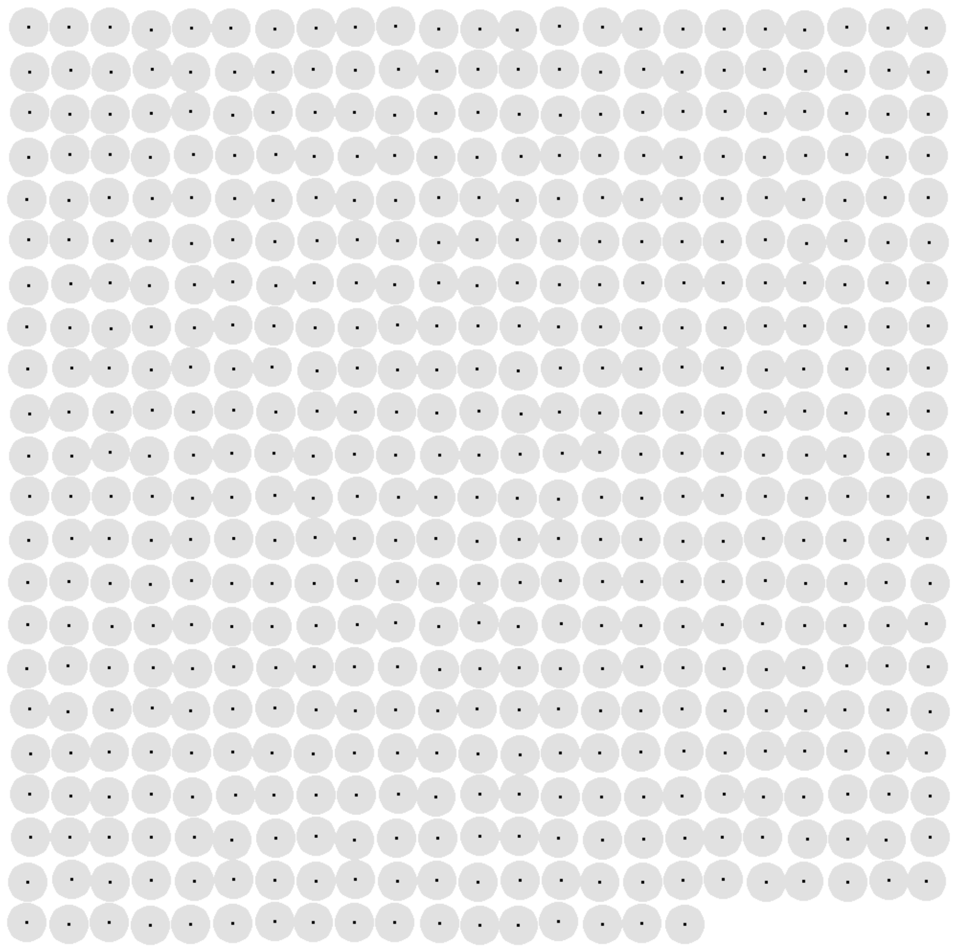
	1	2	4	6	8	10	12	14	16	18	20	22	24	28
OMP 1000	6,22	3,25	2,31	1,63	1,45	1,34	1,25	1,15	1,09	1,07	1,14	1,42	1,5	1,56
MPI 1000	7,21	3,78	2,49	2,2	2,06	1,45								
OMP 2000	24,8	12,66	7,94	5,7	4,63	4,3	3,78	3,42	3,19	2,93	3,11	3,12	3,21	3,27
MPI 2000	29,96	15,48	10,76	8,19	6,5	5,61								
OMP 3000	53,73	29,17	19,81	13,07	10,46	9,81	8,95	7,94	7,22	6,86	6,94	6,98	7,02	6,93
MPI 3000	67,11	34,86	24,48	18,25	14,4	11,91								

Performance results on the 2D implementation:



	1	2	4	6	8	10	12	14	16	18	20	22	24	28
OMP 1000	6,22	3,25	2,31	1,63	1,45	1,34	1,25	1,15	1,09	1,07	1,14	1,42	1,5	1,56
MPI 1000	7,21	3,78	2,49	2,2	2,06	1,45								
OMP 2000	24,8	12,66	7,94	5,7	4,63	4,3	3,78	3,42	3,19	2,93	3,11	3,12	3,21	3,27
MPI 2000	29,96	15,48	10,76	8,19	6,5	5,61								
OMP 3000	53,73	29,17	19,81	13,07	10,46	9,81	8,95	7,94	7,22	6,86	6,94	6,98	7,02	6,93
MPI 3000	67,11	34,86	24,48	18,25	14,4	11,91								

2D visualization (using pillow library):



CUDA

Data structure:
Particles are composed of thrust vector for both the host and the device.

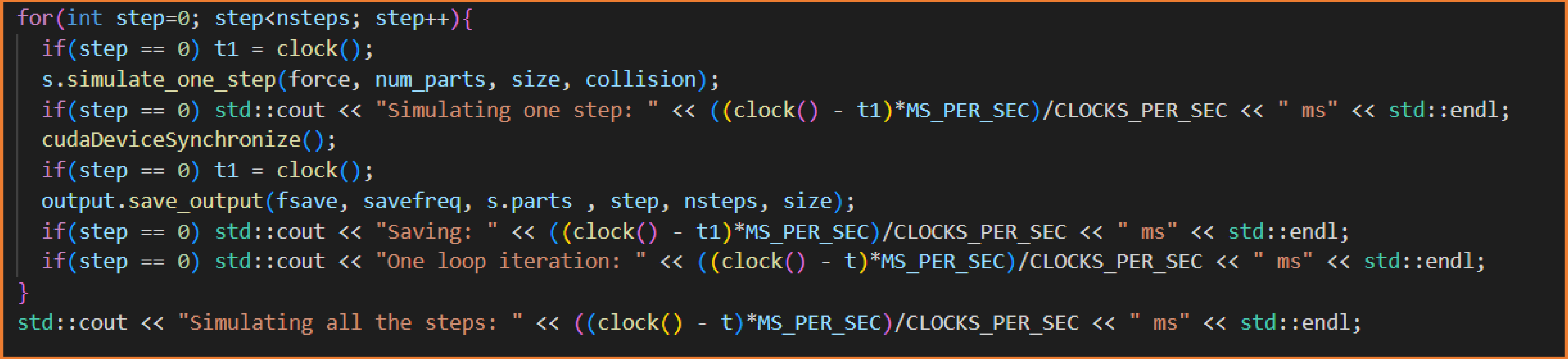
Why have we used thrust vectors?
Thrust provides two vector containers, host_vector and device_vector.
The = operator can be used to copy a host_vector to a device_vector (or vice-versa). The = operator can also be used to copy host_vector to host_vector or device_vector to device_vector. Individual elements of a device_vector can be accessed using the standard bracket notation. However, each of these accesses requires a call to cudaMemcpy.
All algorithms in Thrust have implementations for both host and device.
thrust::copy can copy data between host and device. Hides cudaMalloc and cudaMemcpy.

Why we need dx = thrust::raw_pointer_cast(vy.data()) ?
The reason is that we can’t pass a thrust device vector to the kernel because thrust device vector standard bracket notation is an host function, so we need to pass a pointer to that vector.

Synchronization :

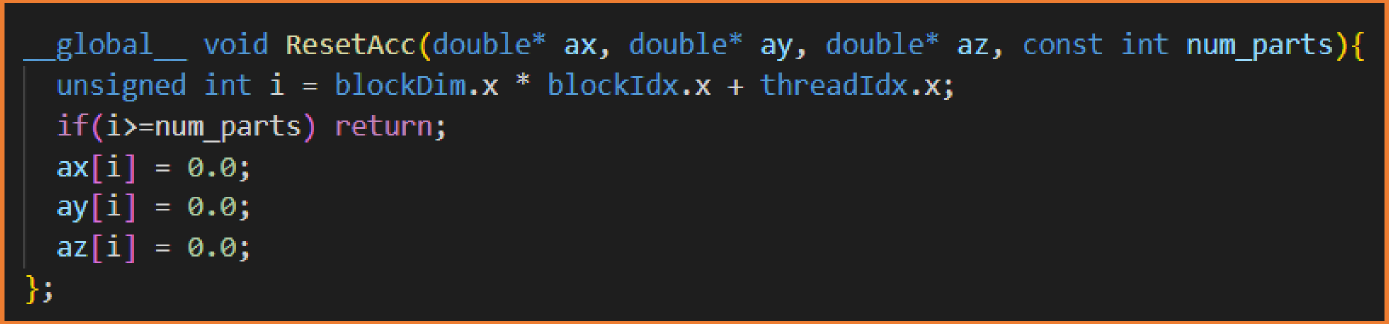
We synchronize at each step after the call to simulate_one_step with CudaDeviceSynchronize() (as with #pragma omp barrier, MPI_Barrier(MPI_COMM_WORLD)).

And inside simulate_one_step after each call to force_application (as it was done in openmp).

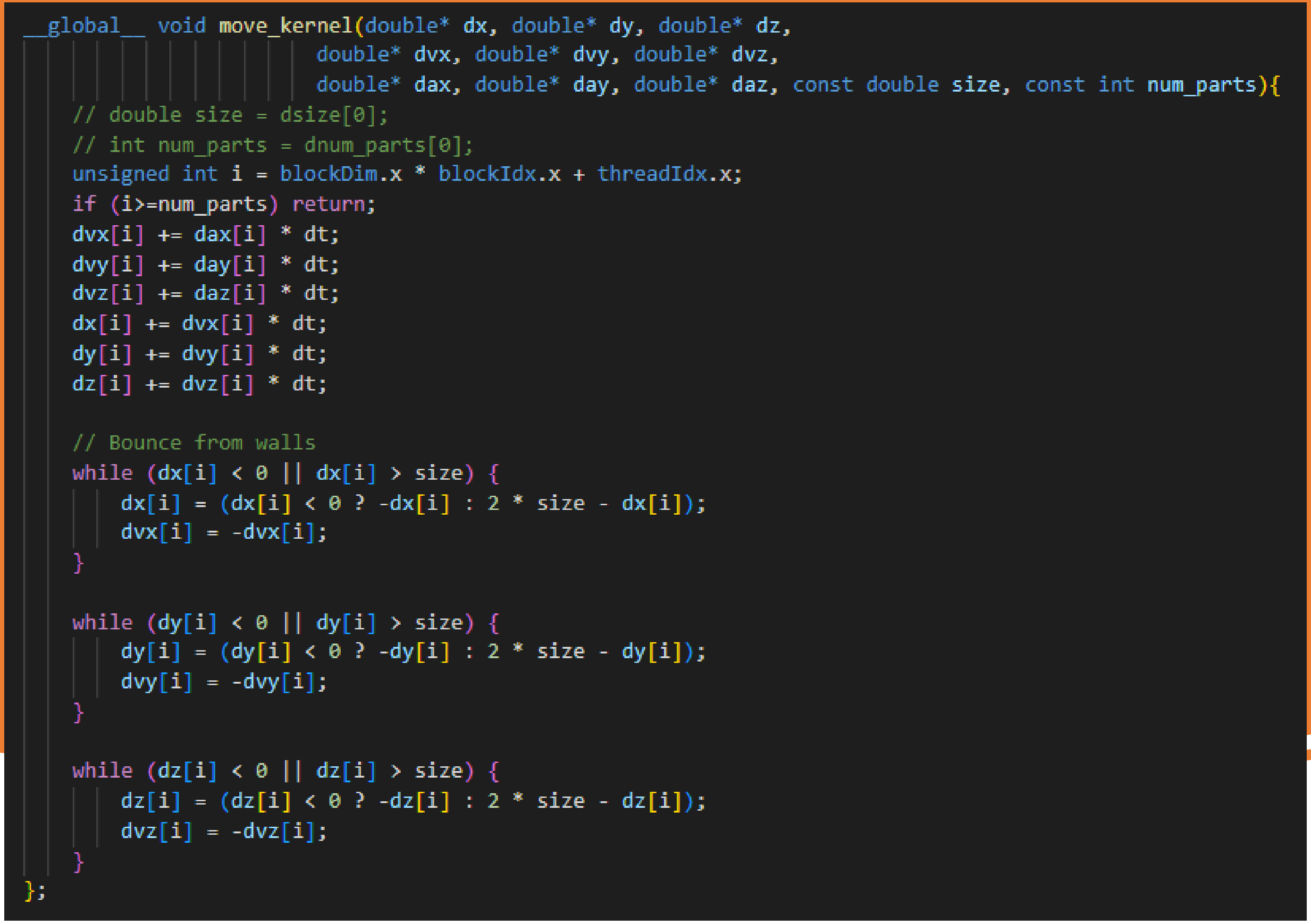


```
for(int step=0; step<nsteps; step++){
  if(step == 0) t1 = clock();
  s.simulate_one_step(force, num_parts, size, collision);
  if(step == 0) std::cout << "Simulating one step: " << ((clock() - t1)*MS_PER_SEC)/CLOCKS_PER_SEC << " ms" << std::endl;
  cudaDeviceSynchronize();
  if(step == 0) t1 = clock();
  output.save_output(fsave, savefreq, s.parts , step, nsteps, size);
  if(step == 0) std::cout << "Saving: " << ((clock() - t1)*MS_PER_SEC)/CLOCKS_PER_SEC << " ms" << std::endl;
  if(step == 0) std::cout << "One loop iteration: " << ((clock() - t)*MS_PER_SEC)/CLOCKS_PER_SEC << " ms" << std::endl;
}
std::cout << "Simulating all the steps: " << ((clock() - t)*MS_PER_SEC)/CLOCKS_PER_SEC << " ms" << std::endl;
```

Kernel functions in AllParticles



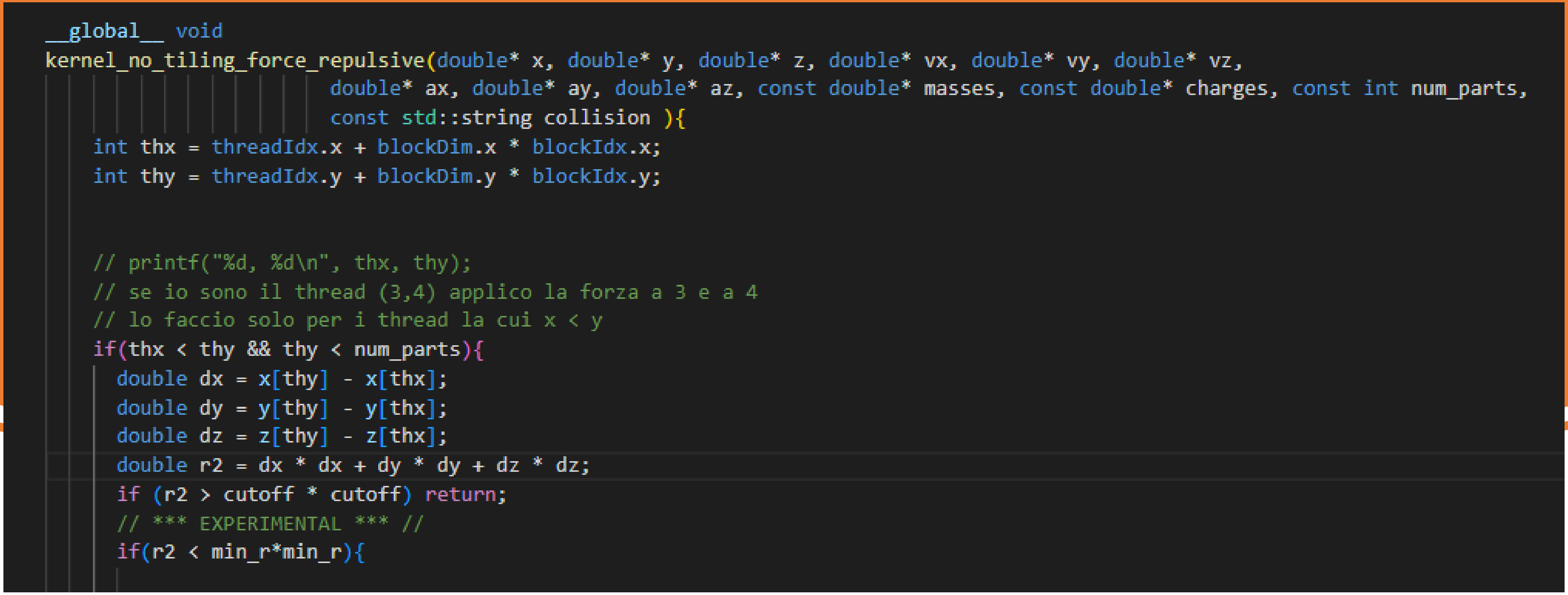
```
__global__ void ResetAcc(double* ax, double* ay, double* az, const int num_parts){
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
  if(i>=num_parts) return;
  ax[i] = 0.0;
  ay[i] = 0.0;
  az[i] = 0.0;
};
```



```
__global__ void move_kernel(double* dx, double* dy, double* dz,
                            double* dvx, double* dvy, double* dvz,
                            double* dax, double* day, double* daz, const double size, const int num_parts){
    // double size = dsize[0];
    // int num_parts = dnum_parts[0];
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i>=num_parts) return;
    dvx[i] += dax[i] * dt;
    dvy[i] += day[i] * dt;
    dvz[i] += daz[i] * dt;
    dx[i] += dvx[i] * dt;
    dy[i] += dvy[i] * dt;
    dz[i] += dvz[i] * dt;

    // Bounce from walls
    while (dx[i] < 0 || dx[i] > size) {
        dx[i] = (dx[i] < 0 ? -dx[i] : 2 * size - dx[i]);
        dvx[i] = -dvx[i];
    }

    while (dy[i] < 0 || dy[i] > size) {
        dy[i] = (dy[i] < 0 ? -dy[i] : 2 * size - dy[i]);
        dvy[i] = -dvy[i];
    }

    while (dz[i] < 0 || dz[i] > size) {
        dz[i] = (dz[i] < 0 ? -dz[i] : 2 * size - dz[i]);
        dvz[i] = -dvz[i];
    }
};
```

Note that class members are declared as normal host function but in their definition there is a call to a kernel function defined in the same cpp file.

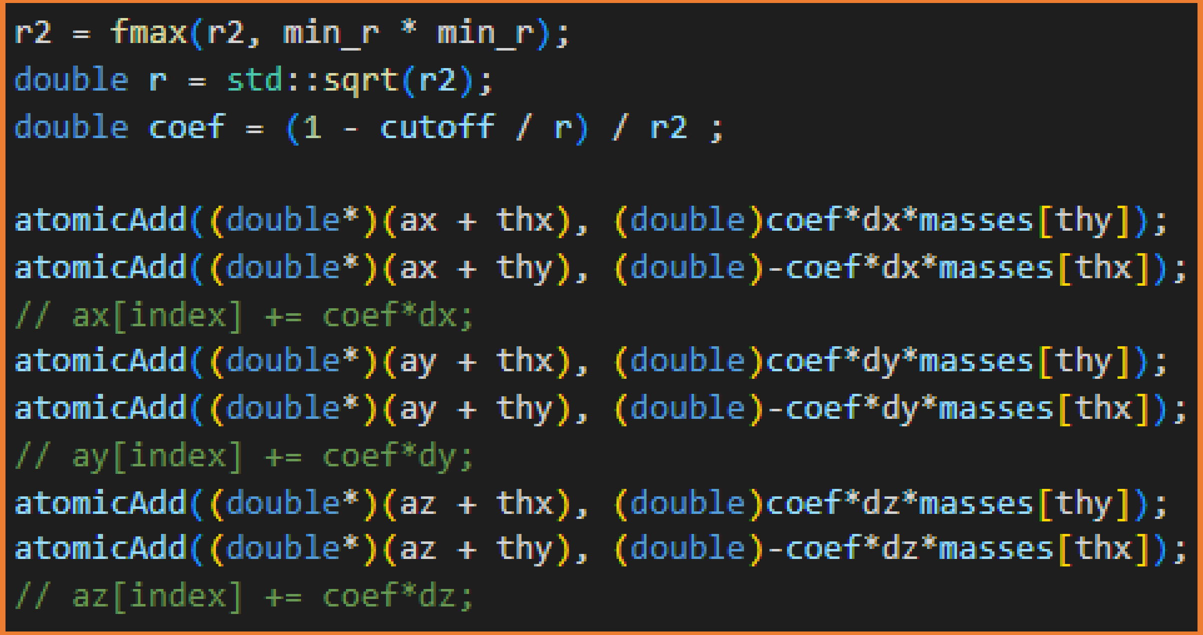
Kernel functions in Simulation :
Simulate_one_step calls force_application that is a method of PhysicalForces class that execute a different kernel for each type of force.
Possible optimization in force calculation:
The calculation of forces can be optimized exploiting the Newton's law of action and reaction. With this optimization the computation is performed just for the lower triangular part of the matrix representing force-interactions between each particle, the upper part is automatically deduced adding a minus sign.



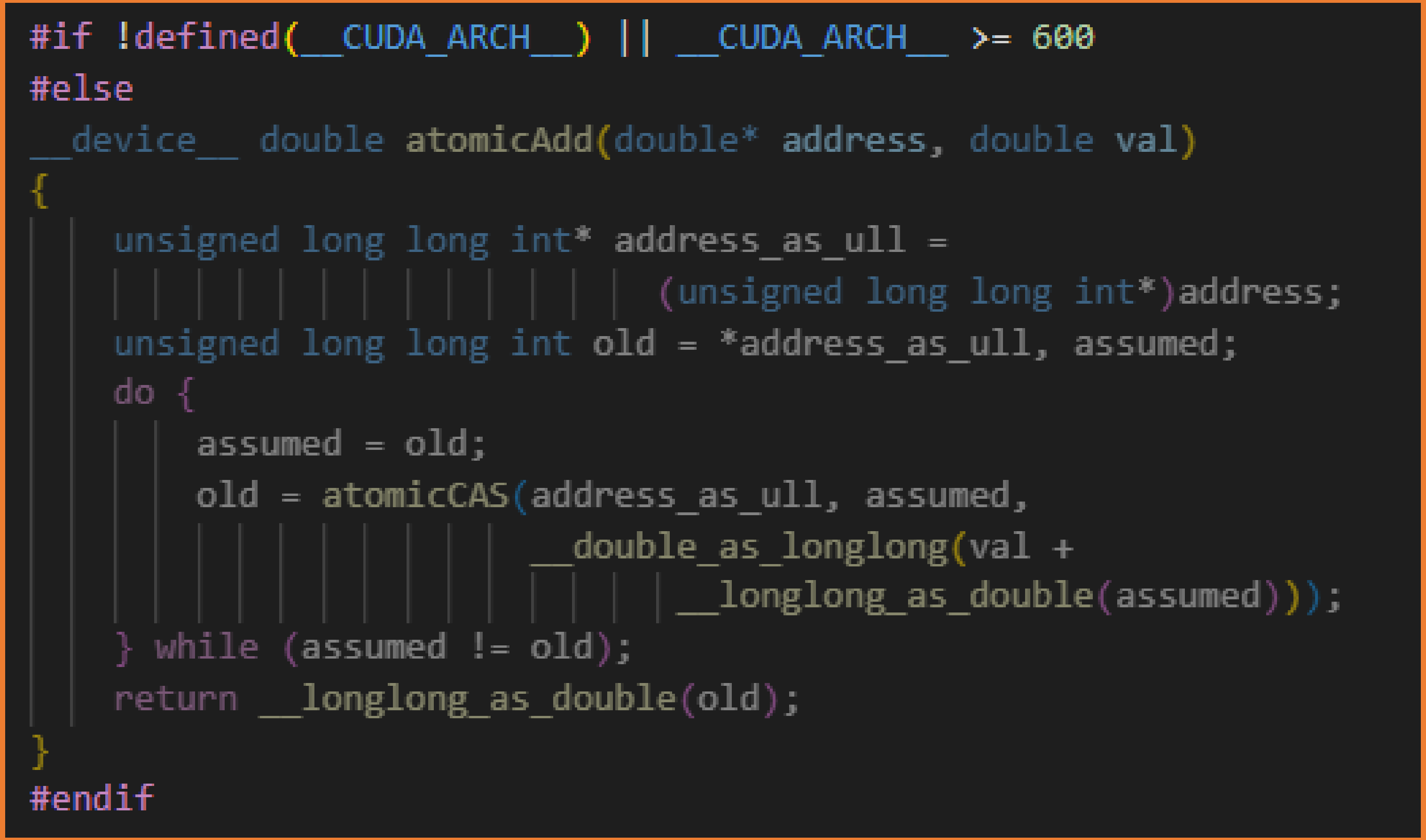
```
__global__ void
kernel_no_tiling_force_repulsive(double* x, double* y, double* z, double* vx, double* vy, double* vz,
                        double* ax, double* ay, double* az, const double* masses, const double* charges, const int num_parts,
                        const std::string collision ){
    int thx = threadIdx.x + blockDim.x * blockIdx.x;
    int thy = threadIdx.y + blockDim.y * blockIdx.y;


    // printf("%d, %d\n", thx, thy);
    // se io sono il thread (3,4) applico la forza a 3 e a 4
    // lo faccio solo per i thread la cui x < y
    if(thx < thy && thy < num_parts){
      double dx = x[thy] - x[thx];
      double dy = y[thy] - y[thx];
      double dz = z[thy] - z[thx];
      double r2 = dx * dx + dy * dy + dz * dz;
      if (r2 > cutoff * cutoff) return;
      // *** EXPERIMENTAL *** //
      if(r2 < min_r*min_r){
```

Concurrency:

in the kernels that compute resulting forces we concurrently access same memory locations to perform the write: an atomic add is needed to avoid data races.



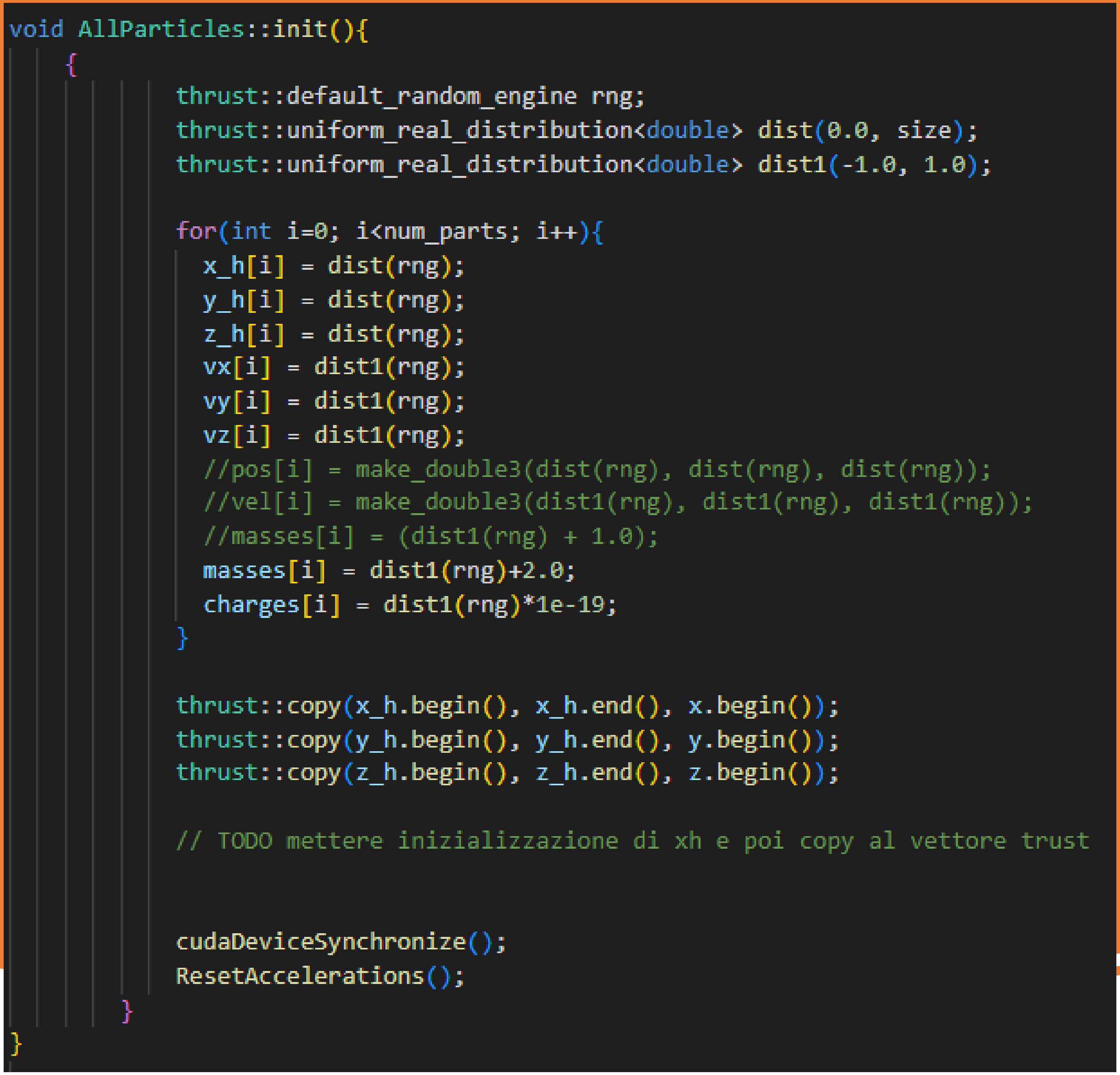
```
r2 = fmax(r2, min_r * min_r);
double r = std::sqrt(r2);
double coef = (1 - cutoff / r) / r2 ;

atomicAdd((double*)(ax + thx), (double)coef*dx*masses[thy]);
atomicAdd((double*)(ax + thy), (double)-coef*dx*masses[thx]);
// ax[index] += coef*dx;
atomicAdd((double*)(ay + thx), (double)coef*dy*masses[thy]);
atomicAdd((double*)(ay + thy), (double)-coef*dy*masses[thx]);
// ay[index] += coef*dy;
atomicAdd((double*)(az + thx), (double)coef*dz*masses[thy]);
atomicAdd((double*)(az + thy), (double)-coef*dz*masses[thx]);
// az[index] += coef*dz;
```



```
#if !defined(__CUDA_ARCH__) || __CUDA_ARCH__ >= 600
#else
__device__ double atomicAdd(double* address, double val)
{
    unsigned long long int* address_as_ull =
                            (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +
                            __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
#endif
```

Optimization in init() function:

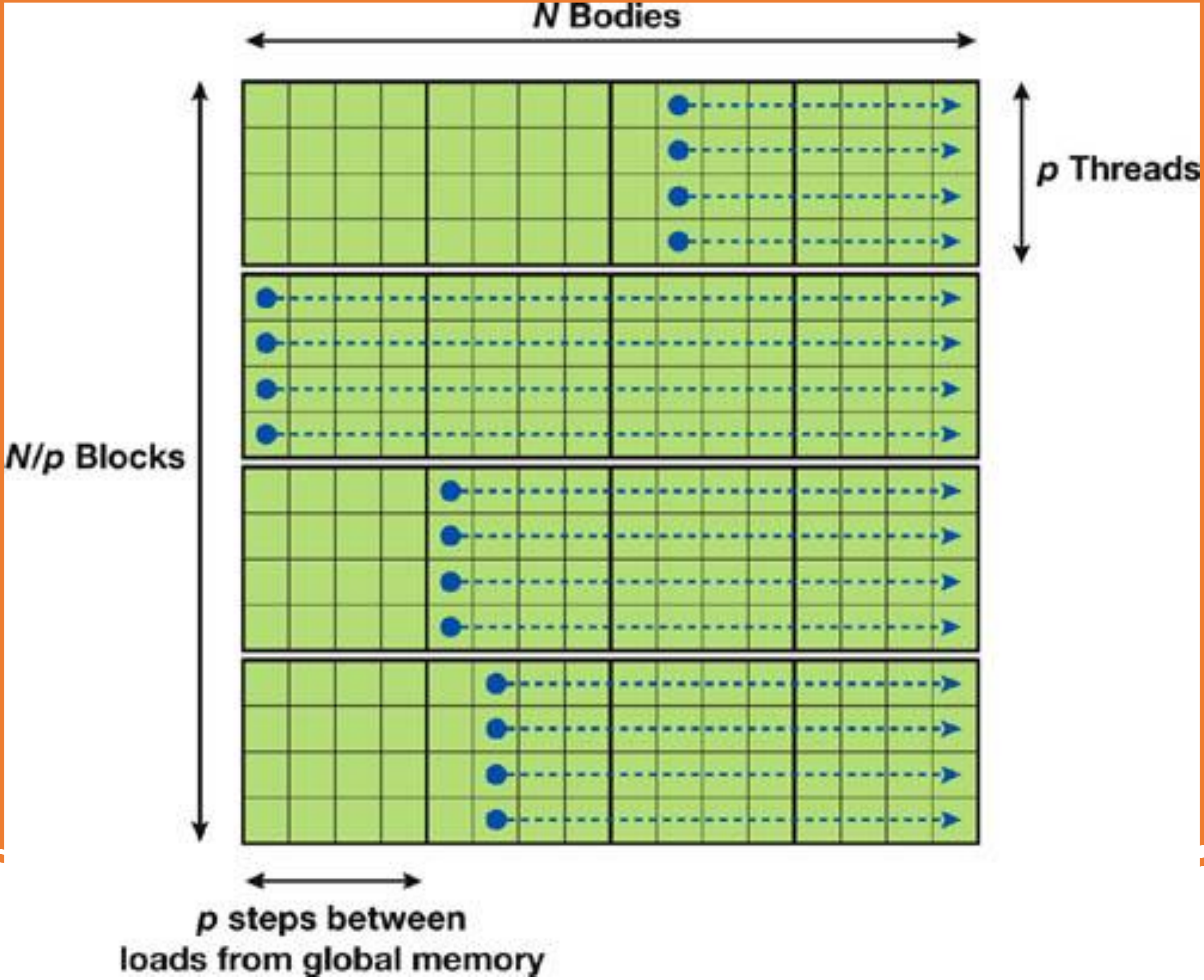
Thanks to thrust we could initialize all device vector’s element accessing them directly from the host function. However this would require the slow operation of cudaMemcpy for each single element.
So, instead of accessing device elements one at a time in order to initialize them, it was preferred to initialize first the host vector and then copying all the entire initialize host vector into the device one with thrust::copy. This alternative is much more efficient because do the transfer in batched fancy better exploiting bandwidth.



```
void AllParticles::init(){
    {
            thrust::default_random_engine rng;
            thrust::uniform_real_distribution<double> dist(0.0, size);
            thrust::uniform_real_distribution<double> dist1(-1.0, 1.0);

            for(int i=0; i<num_parts; i++){
              x_h[i] = dist(rng);
              y_h[i] = dist(rng);
              z_h[i] = dist(rng);
              vx[i] = dist1(rng);
              vy[i] = dist1(rng);
              vz[i] = dist1(rng);
              //pos[i] = make_double3(dist(rng), dist(rng), dist(rng));
              //vel[i] = make_double3(dist1(rng), dist1(rng), dist1(rng));
              //masses[i] = (dist1(rng) + 1.0);
              masses[i] = dist1(rng)+2.0;
              charges[i] = dist1(rng)*1e-19;
            }

            thrust::copy(x_h.begin(), x_h.end(), x.begin());
            thrust::copy(y_h.begin(), y_h.end(), y.begin());
            thrust::copy(z_h.begin(), z_h.end(), z.begin());

            // TODO mettere inizializzazione di xh e poi copy al vettore trust


            cudaDeviceSynchronize();
            ResetAccelerations();
        }
}
```

Optimization: tiles

- The vertical dimension shows the parallelism of the 1D grid of N/p independent thread blocks with p threads each. The horizontal dimension shows the sequential processing of N force calculations in each thread. A thread block reloads its shared memory every p steps to share p positions of data.
- We have taken the dimension of the tile equal to the dimension of the block.
- At each step of the computation it is loaded into the shared memory the information about two little sequences of particles (each of length equal to the x dimension of the block). The info loaded is about the position, velocity, acceleration, mass, charge of that little sequence of particles. At each step are calculated the interactions between each particle of the two sequence. The resulting acceleration is added to the partial one.


N Bodies
p Threads
N/p Blocks
p steps between
loads from global memory

Optimization: stringstream

Most of the time taken by the program is that used to save on file.
A first optimization was to save all the update on a buffer and just in the final step transfer it to the file (instead of saving on file at each step).

Moreover instead of writing to ofstream one by one directly

```
for (int i = 0; i < 10000; ++i) {file << x[i] << “ “ << y[i] << “ “ << z[i] << “\n” ;}
```

we write to a stringstream first, and then write to ofstream at once

```
ostringstream strstream;
for (int i = 0; i < 10000; ++i) {strstream << << x[i] << “ “ << y[i] << “ “ << z[i] << “\n” ;}
ofstream file("c:\\test.txt");
file << strstream.str();
```

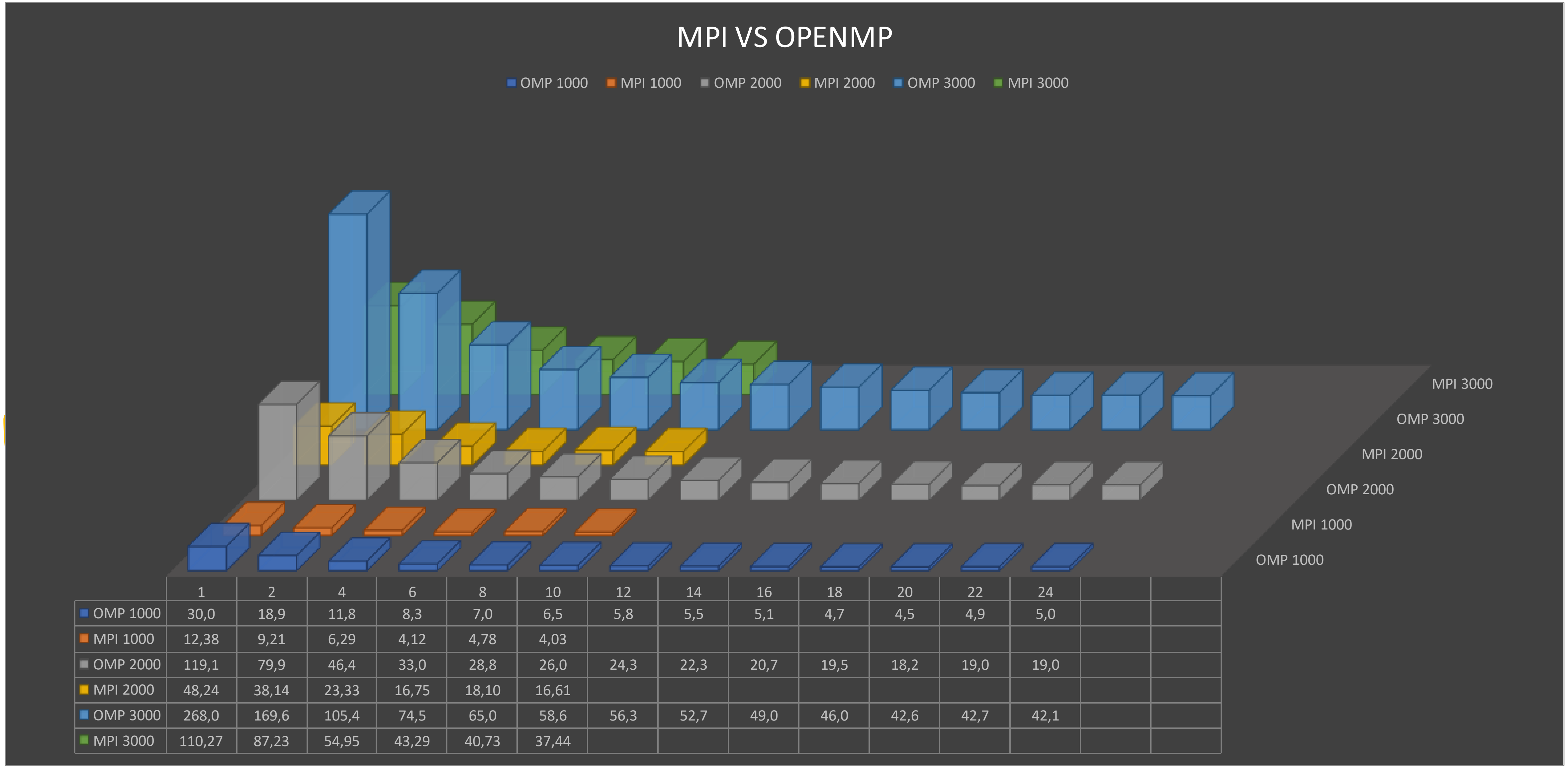
The second approach is faster because flushing to a string buffer will be much faster than flushing to the disk and each time there is a std::endl instead of '\n', std::endl flushes the buffer to disk.

Optimization: Newton’s law

It was not implemented the optimization of the calculation of just the lower triangular part of the matrix representing interactions (while in Openmp we have implemented it). This decision was taken accordingly to a study of the performances.
In this case Cuda run faster even if there are more calculations because of control divergence of the threads in a warp. The maximum performances in Cuda are obtained when all threads in a warp execute the same instructions, so they can be run by the SM all in parallel.
With Newton’s law the overall time would be increased by a factor of 5.

Performance results on the 3D implementation:

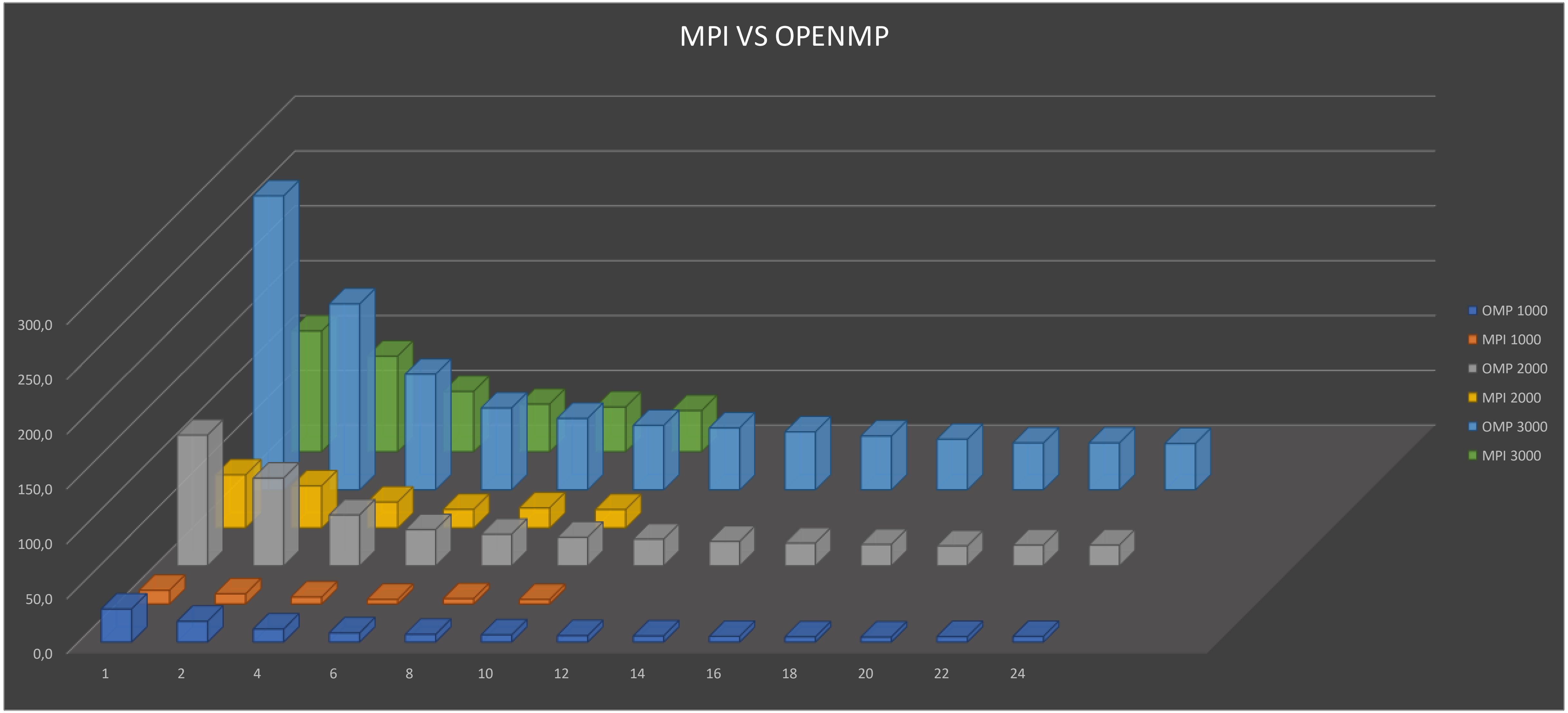
increasing number of particles and number of threads



	1	2	4	6	8	10	12	14	16	18	20	22	24		
🟦 OMP 1000	30,0	18,9	11,8	8,3	7,0	6,5	5,8	5,5	5,1	4,7	4,5	4,9	5,0		
🟧 MPI 1000	12,38	9,21	6,29	4,12	4,78	4,03									
⬜ OMP 2000	119,1	79,9	46,4	33,0	28,8	26,0	24,3	22,3	20,7	19,5	18,2	19,0	19,0		
🟨 MPI 2000	48,24	38,14	23,33	16,75	18,10	16,61									
🟦 OMP 3000	268,0	169,6	105,4	74,5	65,0	58,6	56,3	52,7	49,0	46,0	42,6	42,7	42,1		
🟩 MPI 3000	110,27	87,23	54,95	43,29	40,73	37,44									

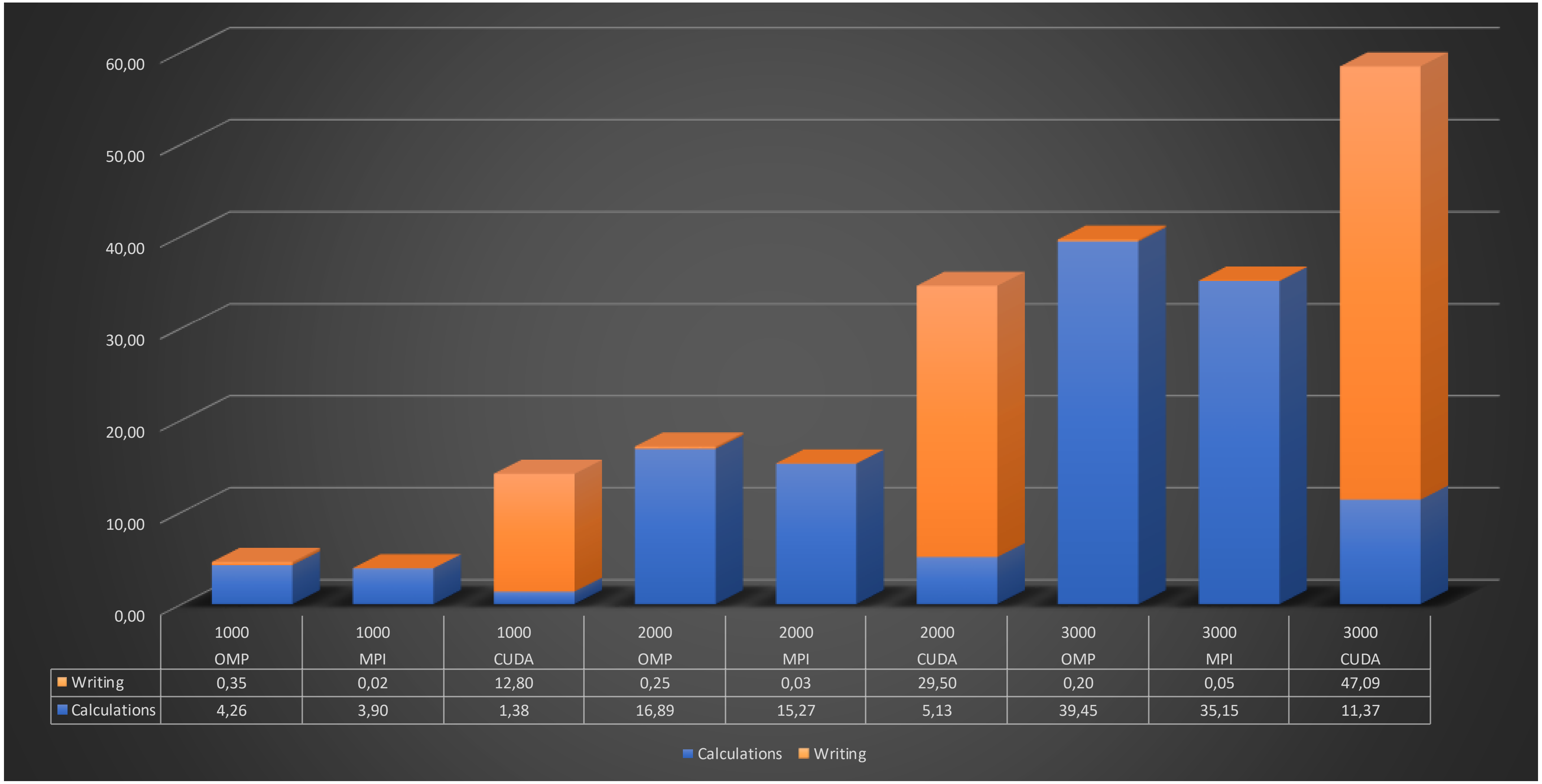
Performance results on the 3D implementation:

increasing number of particles and number of threads



Performance results on the 3D implementation:

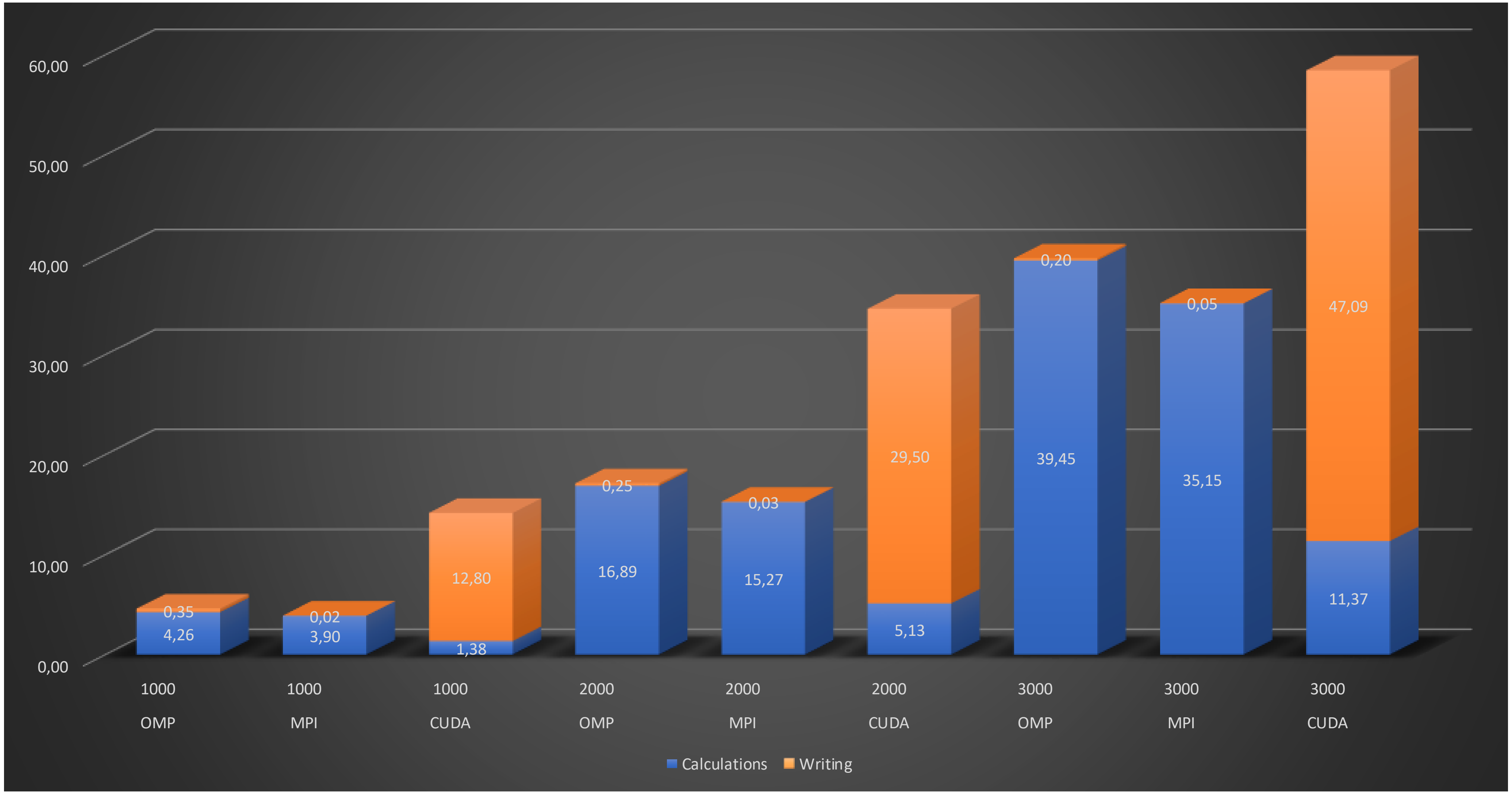
Cuda and MPI



	1000 OMP	1000 MPI	1000 CUDA	2000 OMP	2000 MPI	2000 CUDA	3000 OMP	3000 MPI	3000 CUDA
🟧 Writing	0,35	0,02	12,80	0,25	0,03	29,50	0,20	0,05	47,09
🟦 Calculations	4,26	3,90	1,38	16,89	15,27	5,13	39,45	35,15	11,37

Performance results on the 3D implementation:

OpenMP, MPI and Cuda with maximum number of parallel processing elements on increasing number of particles



Cuda’s performance results on the 3D implementation:

- On the overall, in our experiments the time needed to transfer the three buffers from GPU to CPU is negligible.
- Most of the time is taken by the saving operation.
- Computation with Cuda (in the graph on the previous slide is named “Writing”) requires much less time than with OpenMP or MPI
- Max block dimensions: [1024 , 1024 , 6]. Max thread in a block: 1024
  Max grid dimension: [2147483647 , 65535 , 65535]. We use square grids.

Note 1:
As we have the same type for position, velocity and acceleration and they are all in 3D, it is possible to use double3 instead of three doubles. Double3 is a CUDA vector type derived from the double type. Basically it is a struct of three doubles.

Note 2:
Unique pointer were preferred to shared pointers because they don't introduce communication overhead. Indeed in the serial, openMP and MPI implementations only unique pointers appear (std::unique_ptr<AbstractForce>). However in CUDA implementation we have (std::unique_ptr<AllParticles> and) std::shared_ptr<AbstractForce>. The shared pointer in this case is necessary because not only the class AllParticles own the pointer to AbstractForce. Indeeed Simulation own the pointer to AllParticles which has as a member the pointer to AbstractForce.

Note 3: possible improvement using openGL

The performance of the renderer can be improved by eliminating the need to save on file. This could be done using OpenGL vertex buffer. A more recent approach to this performance issue would be to use a general purpose GPU programming language (GPGPU) like nVidia's CUDA or the open standard OpenCL to update the particle buffer and vertex buffer directly on the GPU so that the memory never has to be moved off the GPU.

Sources:
https://www.3dgep.com/simulating-particle-effects-using-opengl/
https://www.nvidia.com/content/GTC/documents/1055_GTC09.pdf
https://ec.europa.eu/programmes/erasmus-plus/project-result-content/17c9af58-78ac-4276-abd6-25cace51f780/lec09-nbody-optimization.pdf
https://docs.nvidia.com/cuda/thrust/