Advanced Informatics Tools

Memory of the project

Alejandro Cruz
Érika Ferreira
Sergio García
Zazo Meijs
Cristian Privat

April 5, 2018

Contents

Introduction

The goal of the project presented in this memory is to create a code that performs a molecular dynamics simulation of a Lennard-Jones fluid with constant temperature. The simulation has four main parts:

1. Generation of the initial state. Consisting of the geometry, initial positions, normally distributed random velocities and the first matrix of Lennard-Jones forces.

2. Equilibration of the system. To be able to work with constant temperature, it is necessary to run a certain number of steps the simulation until it stabilizes to the chosen temperature.

3. Production. It is the run of the main simulation, for every time step the temperature, total energy, kinetic energy, potential energy, pressure and velocities are saved.

4. Analysis. All the results given by the simulation are analyzed using binning analysis.

The code has been written by five people, and the coordination of the work was done through a GitHub repository.

Chapter 1

Algorithms explanation

Two programming languages have been used: Fortran and Python. The main program is coded in Fortran, and a part of the data visualization is in Python. The Fortran code is divided in different modules and there are two versions of the code, one in serial and the other coded in parallel using MPI.

All the parameters needed to run the simulation are read from a file name *param.dat*. In this section, the algorithms used for each part of the code will be explained.

1.1 Generating Geometry

This module generates a simple cubic crystal structure. To do it, the module reads from the parameter file (*param.dat*) the density of the box (ρ), in reduced units, and the number of atoms in a row (M parameter). After that, all parameters necessary to characterize the box are calculated. The calculated parameters are: number of atoms (N_atoms = M^3), cube edge (L_box = $\sqrt[3]{\frac{N\_atoms}{\rho}}$) and interparticle distance (aa = $\frac{L\_box}{M}$).

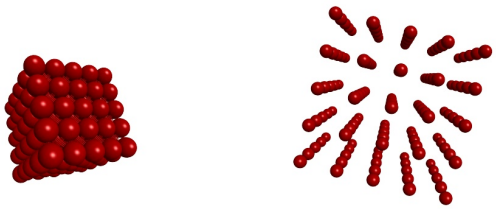


Figure 1.1: *Initial structure generated by the GenGeom module with a reduced density of 1 (left) and of 0.05 (right) and with M=5.*

This module returns the initial positions of the atoms in cartesian coordinates and reduced units (stored in a matrix) and the cutoff for the Lennard-Jones potential which is calculated from the cutoff ratio and all the parameters necessary to characterize the box commented above.

It has been constructed this way to assure that the atoms do not overlap in the generated geometry. On the other hand, there is no inconvenience on starting with a crystalline structure since having an equilibration after the generation of the crystalline structure causes it to melt if the system has enough energy or the temperature is high enough. The only disadvantage with this generation method is that the maximum ρ (in reduced units) value is 1.0. This is not a problem with the simulated system because Helium gas at 300K and approximately 1 atm has a density value below the threshold of the simple cubic structure.

In addition, the module allocates part of the memory for the velocities and forces matrices. The position of the box is placed on the (0,0,0) coordinate, which assures that all the atoms have positive coordinates.

1.2 Velocity initialization

The *initial_velocitymod* generates the initial velocities of the particles distributing them uniformly between $\frac{-\sigma}{2}$ and $\frac{\sigma}{2}$ where σ corresponds to the Sigma parameter in the *param.dat* file.

It is known that in the NVT collectivity the velocities distribution of gas particles in equilibrium is the Maxwell-Boltzmann distribution, meaning that they follow a Gaussian distribution. Therefore, if the initial velocities followed a Gaussian, this would favor the system to reach the thermodynamic equilibrium faster, but, due to the appliance of an equilibration after the system initialization where a thermostat is applied (see Section 1.5), what has been commented before has no relevance because the equilibration is enough to cause the particle velocities to follow the Maxwell-Boltzmann distribution.

It is also important to notice that the *momentummod* module contains two functions, *Kinetic_E* and *P_total* which calculate the kinetic energy and the total momentum of the system respectively. Even if the two functions can be parallelized, it has not been done because the writing process of the results has to be done in series or the data can be corrupted, and both functions are used during that process.

1.3 Integrators

The simulation can be runned using two different integrators which can be selected in the parameters data file. *Ver* is for the Velocity-Verlet algorithm and *Eul* is for the Euler algorithm.

Velocity-Verlet

The Velocity-Verlet algorithm is described by the following equations:

$$r_i(t + \Delta t) = r_i(t) + v_i(t)\Delta t + \frac{f_i(t)}{2m_i}\Delta t^2 \tag{1.1}$$

$$v_i(t + \Delta t) = v_i(t) + \frac{f_i(t) + f_i(t + \Delta t)}{2m_i}\Delta t \tag{1.2}$$

This integrators takes the matrices with the positions, velocities and forces information, and using a given Δt, it modifies the values and returns the new ones that correspond to the following time step. The steps followed by the program to perform this calculations are:

1. Calculate the matrix forces at time t.
2. Calculates the new positions with the formula 1.1.
3. Call the pbc module to apply the boundary conditions.
4. Calculate the new forces applying the LJ algorithm to the new positions.
5. Calculate the new velocities using equation 1.2.

1.3.1 Euler

The Euler algorithm is described by the following equations:

$$r_i(t + \Delta t) = r_i(t) + v_i(t)\Delta t + \frac{f_i(t)}{2m_i}\Delta t^2 \tag{1.3}$$

$$v_i(t + \Delta t) = v_i(t) + \frac{f(t)}{m_i}\Delta t \tag{1.4}$$

There are two distinctions from the previous integrator algorithm:

- While Velocity-Verlet requires two steps, Euler needs just one.
- There is no need to calculate the forces from the following time step to obtain the new velocities with equation 1.3.

The steps followed by the program are:

1. Calculate the matrix forces.
2. Update the positions matrix using the equation 1.3.
3. Call the pbc module to apply the boundary conditions.
4. Update the velocity matrix using equation 1.4.

1.4 Periodic Boundary Conditions

The main goal of this module is to give the system a toroidal geometry which gives periodic boundary conditions (PBC). Two different algorithms have been written in the module:

- A function that counts how many "boxes" the particle has moved.
- A subroutine that returns the particles, that have move outside of the box, to inside the box.

The module analyzes the coordinates of the positions (x,y,z) of each particle and compares them with the box length. If the coordinate value is higher than the box length or less than 0 (because there should not be any negative coordinates), it calls the function to know how many boxes the atom has moved and then aggregates or subtracts the box length to the coordinates.

1.5 Temperature control

To control the temperature, an Andersen thermostat is used. This thermostat works following these steps:

1. It reads the values of the probability (P) and the temperature (T).
2. It iterates though the list of velocities of each particle.
    a) For each particle it generates a random number ζ
        1) If $P > \zeta$ then it generates three random numbers in the standard distribution with a $\sigma = T^{\frac{1}{2}}$ and $\mu = 0$. The velocity of the particle is updated with the generated values.
        2) If $P < \zeta$, no changes are made.

The velocity of the particle changes its size and direction randomly, always following a standard distribution.

1.6 Units

All units used in the program are in reduced units, which values correspond to the Lennard-Jones parameters σ and ϵ. To output the results with the correct units, a correction is applied in the program. The corrections are:

$$Distance \rightarrow r' = \frac{r}{\sigma} \tag{1.5}$$

$$Energy \rightarrow u' = \frac{u}{\varepsilon} \tag{1.6}$$

$$Time \rightarrow \left(\frac{\varepsilon}{m\sigma^2}\right)^{\frac{1}{2}} t \tag{1.7}$$

$$Mass \rightarrow m' = 1 \tag{1.8}$$

$$Temperature \rightarrow T' = k_b T \tag{1.9}$$

1.7 Lennard-Jones Cutoff

Due to the amount of calculations needed to obtain the forces and the potential energy of every particle, a cutoff is used. This cutoff is written in the *param.dat* file and it has to be between 0.0 and 0.5, and it is multiplied by the length of the box to calculate the value of r_{CO}. The energy of the system will be then calculated as:

$$u^{LJ} = 4\left[\left(\frac{1}{r}\right)^{12} - \left(\frac{1}{r}\right)^{6}\right] \tag{1.10}$$

Where,

$$\begin{cases} r > r_{CO} & 0 \\ r < r_{CO} & u^{LJ} \end{cases} \tag{1.11}$$

To calculate the forces, the same criterion as in 1.11 is used.

1.8 Cutoff correction

To improve the calculation of the potential, a correction has been applied. The correction modifies the equations 1.10 and 1.11 as:

$$u^{LJ}_{CO} = 4\left[\left(\frac{1}{r_{CO}}\right)^{12} - \left(\frac{1}{r_{CO}}\right)^{6}\right] \tag{1.12}$$

Where,

$$\begin{cases} r > r_{CO} & 0 \\ r < r_{CO} & u^{LJ} - u^{LJ}_{CO} \end{cases} \tag{1.13}$$

1.9 Binning Analysis

We plotted the energies after 5000 initiation steps, as can be seen in figure 3.2. We see that after a few thousand steps the energies quickly approach a constant value, but as we see the energies still show fluctuation we need to apply binning analysis to approximate the average value and determine its standard deviation.

From visual observation we have decided that the last 40% of the data is sufficiently equalized. We have split this data in 10 equal sized bins of which we use the first 75% of data points to calculate a local average the following 25% of data points is used to decorrelate the local averages. The standard deviation is given for a 2 sigma (σ), which corresponds to a 95 % certainty interval, following the central limit theorem (citation?). So the resulting averages are given $\pm 2 * \frac{\sigma}{\sqrt{10}}$.

1.10 Reaper

Reaper is the algorithm that calls all the parts of the program and writes the output files. It is a complex algorithm and is easier to understand by reading the code.

1.11 Makefile

Makefile is an easy way of organizing the compilation of a code and can be used to automatize different actions. The makefile created for this computational scientific project performs the following actions:

- Compilation: Compiles the program. It will be recompiled every time one of the files *.f90* or *md.o*, is modified.
- Data: Generates the data files of a molecular dynamics simulation using the specified parameters in the*param.dat* file. These data are generated every time that file or one of the *.f90* or *md.o* are modified.
- Plot: Generates figures about several magnitudes of interest in a molecular dynamics simulation. The figures are independent and a single figure is replotted if the data used to generate it change or if the script to plot the figure is modified.
- Variables: Print the variables of the makefile file.
- Hardclean: Deletes all the generated files including the outputs of the molecular dynamics program, the plots and the program binary.
- Clean: Deletes the generated files except the outputs of the molecular dynamics program, the plots and the program binary.
- Help: Give instructions on how to use the makefile.

The explained makefile corresponds to the serial version of the program. For the parallel version, there is an adjustment of the makefile that executes the binary file in four processes by default. Four processes provide more benefits in terms of performance in the execution of the program.

Chapter 2

Parallel code

To speed up the execution of the dynamics, some parts of the code have been parallelized using fortran MPI; Message Passing Interface. The parallelized parts will be explained in this chapter.

It is important to mention that there was a problem with the generator of random numbers in the parallelized program. To fix it, the seed that enters in the generator is modified using the rank of the processor.

2.1 Generating geometry

The parallelization of the *geommod* module consists in dividing the system particles in different sets and each process depending on its rank, generates the initial positions of the particles for the given set, this is performed by the subroutine *GenGeom*. Each process has then the positions for the particles in a set and with the goal that each process has available the positions of all the particles of the system, every process shares its generated positions (array *Position_mat*) with the others by *MPI_BCAST* and once all processes have the information of the positions generated by the others, all of them write the positions in a determined region of the *Position_mat_Total* array. If this is repeated to all generated processes, then *Position_mat_Total* array will contain the positions of all system particles and, in addition, all the processes have access to the same *Position_mat_Total* array. The procedure to share the generated positions and write them in the correct array is performed by the *Regroup* subroutine.

This way of parallelizing has been chosen because all the processes have the same work load and the communication between them is minimized. It is important to minimize the communication because it slows down the performance.

2.2 Velocity initialization

The parallelization of the initial velocities generator is made similarly to the method exposed in the Section 1.1. Each process generates the initial velocities for a given number of particles using the *Initial_Velocity* subroutine, and later the different processes share the generated velocities by the *Regroup* subroutine. All the velocities are stored in the *Velocity_mat_Total* array.

The generation of initial velocities in parallel has an advantage in relation to the generation of positions which is that is not necessary to split the particles in sets because two particles cannot have the same initial position but can have the same initial velocities.

In the parallel version, to each particle a velocity between 0 and σ is assigned to all components. This is a problem because the center of mass of the system will have a net speed and so the system will move. To solve the problem, every process calculates the net speed, in each component, for the generated velocities, the sum is performed using *MPI_Reduce* and the results are stored in the *total_sum* vector (vector of three components) and to end the *total_sum* is divided between N_atoms and the resultant vector is subtracted from the vector of all particles velocities.

The choice of this parallelization is due to the same reasons outlined in the previous Sections.

2.3 Andersen Thermostat

The difference between the code programmed in serial is that the operations inside an interval given by the *rank* variable (rank and process number) and the number of total processes, *num proc* variable.

The modules called to generate the random numbers *mtmod* and the normal distribution *normaldist* do not enter in conflict with the parallelization because it generates a vector with random velocities for the three coordinates. Once the process ends, the matrix velocity is updated in the main program performing a reshape of the before mentioned vector.

2.4 Velocity Verlet algorithm

As to parallelize the code, the integration of the positions and the velocities has required to split the algorithm in two modules because is necessary to perform the calculation of the forces using the Lennard-Jones algorithm as to update the velocities. Therefore:

- *MPIvelocity_verletmod*: this module uses the 1.1 equation to update the positions and calls PBC to correct the values of the positions.

- *MPIvelocity_verlet2mod*: calculates the forces using the Lennard Jones mode and updates the velocities.

Finally, the main program stores the new positions and velocities in its original matrices.

2.5 Euler algorithm

Simpler case respect the parallelization of the Velocity Verlet algorithm. In this situation, it is not required to divide the module into two parts given that the calculation of forces is previously performed respect the integration of positions and velocities. The same procedure as the previous modules is followed (using the total number of processes and the current process to perform only the relevant operations within the process range) and finally positions and velocities are updated in the original matrix in the main program.

2.6 Periodic Boundary Conditions

To parallelize this module, the position matrix, that enters after the first step of the integrator module, breaks into smaller matrices with the size of the *partition* variable. Every set runs in a different processor and performs the works explained in Section 1.4 and after that the data is reunited again in the original matrix. In this way, the communication between the processors is minimized.

2.7 Lennard-Jones

To parallelize the Lennard-Jones potential, a subroutine that generates a matrix of size $Nx2$ where N is the number of interactions and 2 are the number of particles that interact with each other, has been created. The generated matrix is called *Lj_Int*. Then, a variable named *lll* which corresponds to the number of interactions, divides the interactions between the number of processors ($N/numproc$); therefore, in every processor the loop that gives the final LJ interactions is performed. Once all the matrices are finished, they are reunited using the command *MPI_ALLREDUCE*.

It is important to remark that, for time saving, the matrix of interaction is calculated in every processor individually, it is not sent to all the processors.

2.8 Trivial Parallelization of the Binning Analysis

The original program takes 0.903 seconds, including the creation of the plots, so is not worth optimising. Yet this was still done for educational purpose the binning

analysis was done in parallel. As python has a Global Interpeter Lock (GIL) which blocks access to python objects from multiple threads. Using the multiprocessing library instead of using parallel threads the instances are generated as independent processes which have there own instances of the python objects as such bypassing the GIL restrictions. This does generate a larger overhead as each of the processes have to be launched. As the processes are run by the operating system they are automatically spread over all available cores. The example implementation calculates the local averages of the binning analysis using ten spawned processes. As this is a near instantaneous calculation for our problem size the overhead of the process creation is many times greater than the calculation itself. For the test data the multiprocessing version took 1.42 seconds so a factor 1.55 longer than the serial version. Furthermore if you look at the code you see that it could also be done as an array based numpy calculation. As the numpy instances are also done outside of GIL these will likely be faster for any problem size. So no this parallelization was not useful, but it is parallel.

Chapter 3

Analysis of results

To test the program and have some data to analyze, the simulation was performed for the Helium (He) atom, with the following parameters:

Parameter	Value
M	5
Density	0.05
Cutoff ratio	0.5
Time delta	0.01
Seed	350
Sigma	1.0
Temp	2.0
Nsteps	10000
Nstabilize	5000
σ	1.0
ϵ	1.0
mass	1.0
Andersen probability	0.01
dr size	0.1

Table 3.1: Parameters used to run the simulation. All values are in reduced units.

It can be seen that simulation starts with 5000 steps to stabilize the temperature of the system and then the production is runned in 10000 steps. Therefore, this simulation has 50 ps of stabilization and 100 ps of production.

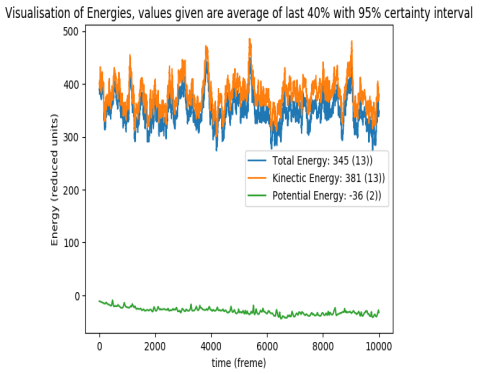
The probability of the Andersen thermostat used to perform the calculations is of the 10% of probability to change the particle velocity.

The simulation is tested using both integration algorithms explained and at different values of time delta and leaving the other parameters as shown in Table

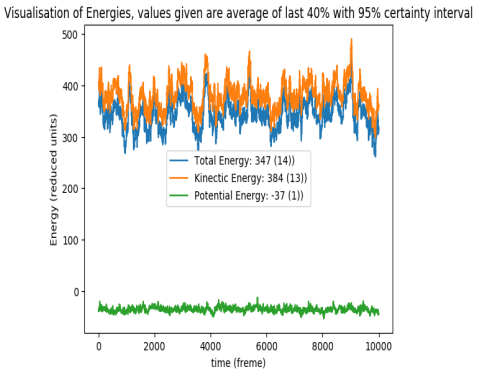
4.1.

3.1 Verlet

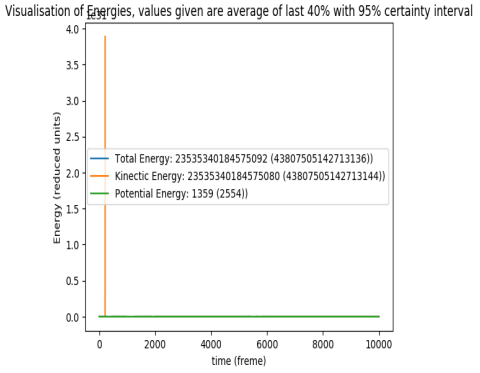
As it can be seen in Figure 3.1, the total energy has some peaks which means that it is not fully conserved. At low Δt the fluctuations are similar; but in Figure 3.1c no advances of the energy in front of the time is observed, there is only a high energy peak in the first steps. This divergence in the results can mean that the integration algorithm is not capable of computing the steps where the particles have a high displacement. This could happen because of the great repulsion generated by the Lennard-Jones potential at short distances makes the particles to move further away in a higher increase of time and enters inside the repulsion radium of the particle, and in the next step the particles fly away.



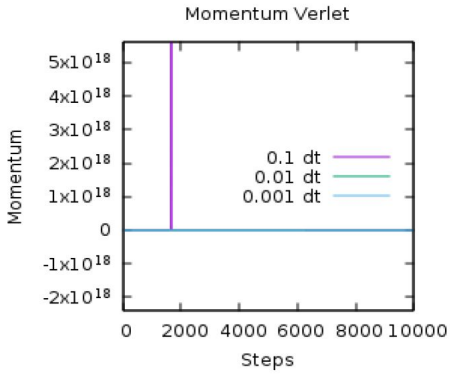
(a) Energies with $\Delta t = 0.001$



(b) Energies with $\Delta t = 0.01$



(c) Energies with $\Delta t = 0.1$



(d)

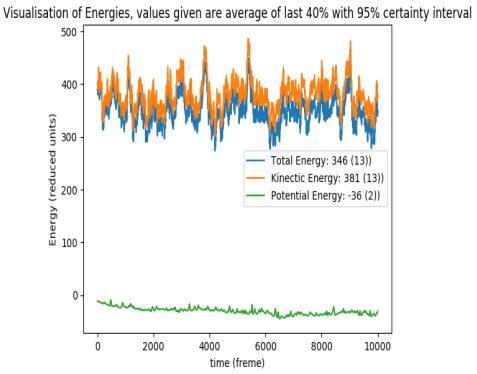
Figure 3.1: Captions showing the potential, kinetic and total energy at different values of Δt (a,b,c) and a graphic showing the momentums at different Δt.

In the Figure 3.1d, it can be seen how the moment is conserved for all the values of Δt used except for $\Delta t = 0.1$. Due to the fact that the system is isolated, the only

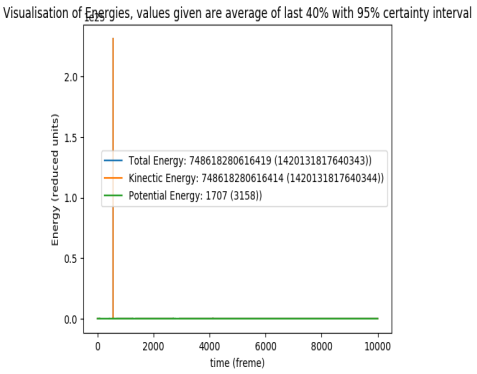
force acting in the system is a pair interaction function, with equal intensity and opposite direction; the system should not show a variation in the initial momentum. This momentum is close to 0 because before starting the calculations, once the initial velocities have been generated, the total momentum is changed to 0 to avoid the translation phenomenon in the system.

3.2 Euler

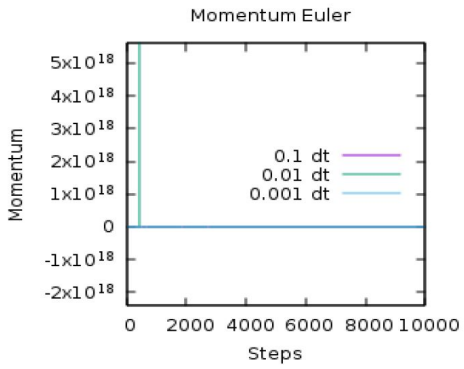
If Figure 3.1d and Figure 3.2c are compared, it is possible to observe that the Euler algorithm is capable to conserve the initial momentum for $\Delta t = 0.001$, but it diverges for the other values.



(a) Energies with $\Delta t = 0.001$



(b) Energies with $\Delta t = 0.1$



(c)

Figure 3.2: Captions showing the potential, kinetic and total energy at different values of Δt (a,b) and a graphic showing the momentums at different Δt.

This result could be because with the Euler algorithm, just an initial point is used to perform the calculations and with the Verlet algorithm, two points in the forces matrix are used.

3.3 Velocities distribution

When the Andersen thermostat is applied, the velocities of the particle have to follow an standard distribution. To verify that this happen, an histogram with all the velocities registered during the simulation is represented (Figure 3.3).

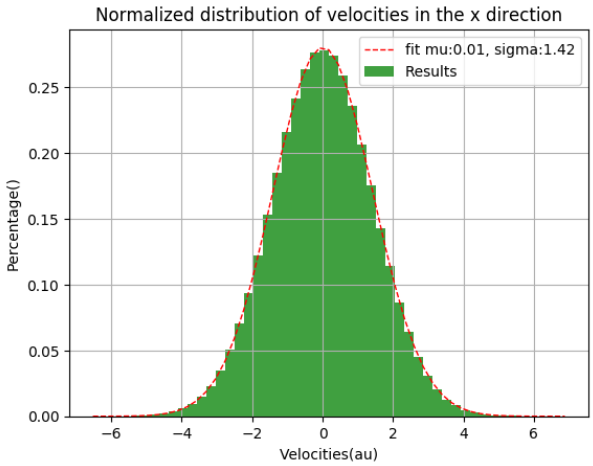


Figure 3.3: Histogram with the distribution of the velocities comparing with a Gaussian curve (red line).

To draw the Gaussian curve, a subroutine that writes a file with all the data of an analytical Gaussian centered in 0 ($\mu = 0$) and with $\sigma = T^{\frac{1}{2}}$. It is represented in Figure 3.3 with a green line.

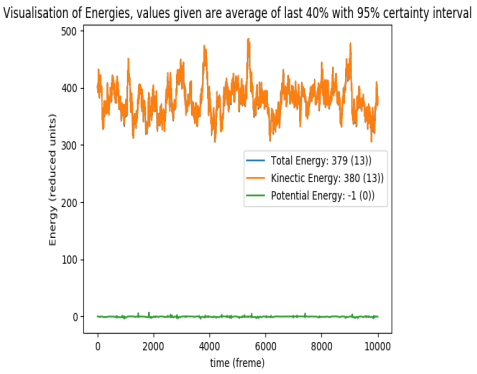
Due to the similarity between the obtained histogram and the analytical curve, the Andersen thermostat distributes correctly the velocities.

3.4 Pressure and density

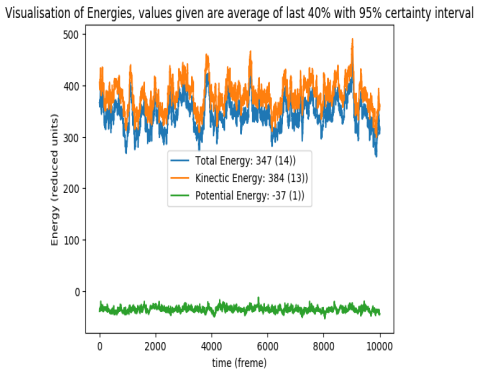
It is important to see how the changes in the density of the system affect the energies and the pressure of the system. As it can be seen in Figure 3.4a that in small ρ values, the potential energy tends to 0 due to the almost inexistent Lennard Jones interaction. The system depends only on the kinetic energy of the moving particles.

As the value of the density increases, the kinetic energy remains constant due to the interaction with the thermal bath, but the potential energy become more negative every time. The absolute value of the potential energy increases because of the short distance between the atoms.

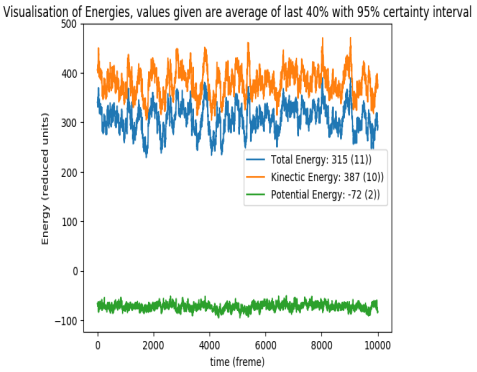
As a result of the packing that is being used and the great repulsion of the Lennard Jones forces, at densities higher than $\rho = 1.0$, the program fails and is not capable of running the dynamic.



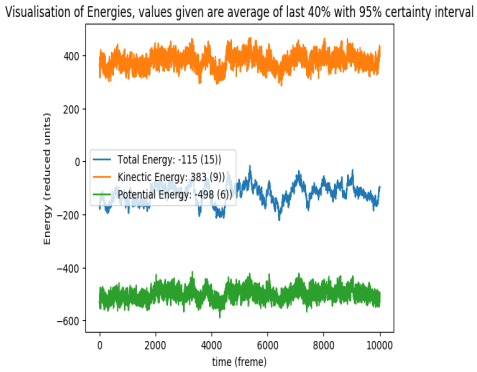
(a) Energies with $\rho = 0.001$



(b) Energies with $\rho = 0.05$



(c) Energies with $\rho = 0.1$



(d) Energies with $\rho = 1.0$

Figure 3.4: Captions showing the potential, kinetic and total energy at different values of ρ.

If the pressures are studied, it can be noticed that as the density increases, the pressure also increases.

$$\rho = 0.001 \rightarrow p = 2.01 \cdot 10^{-3} \pm 2 \cdot 10^{-5}$$

$$\rho = 0.05 \rightarrow p = 0.122 \pm 8 \cdot 10^{-3}$$

$$\rho = 0.1 \rightarrow p = 0.29 \pm 2 \cdot 10^{-2}$$

$$\rho = 1.0 \rightarrow p = 25 \pm 1$$

This is because the particles are closer to each other, so the LJ repulsion forces are higher.

3.5 Radial distribution function

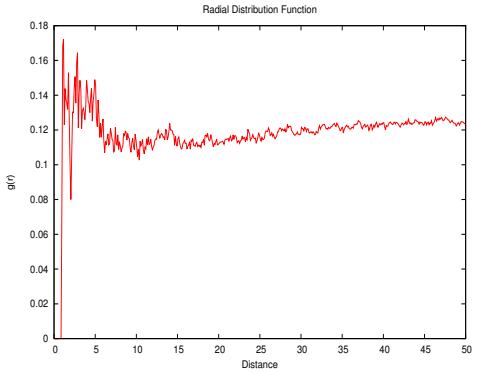
The radial distribution function is calculated at the same densities used in Section 3.4. In Figure 3.5, the four radial distribution functions are plotted.

It can be appreciated that for low ρ values, the g(r) presents a peak at a σ distance. This peak is produced for the attractive LJ interaction of the few particles that meet; and from that distance, the g(r) decreases and remains constant due to the increasing difficulty of finding another particle in the way.

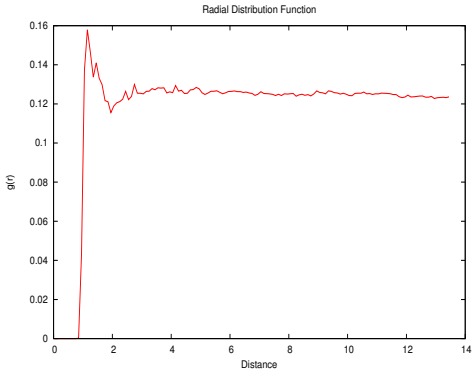
It is important to say that at $distance < \sigma$, the probability of two particles to meet is almost 0. This is represented in all captions of the Figure 3.5.

Taking a look at Figure 3.5d, it can be seen that the function represents local maxima and minima. This happens because the particles are structured in the space (simulation of high density solids and liquids), and so the particles moves are limited and the atoms can be found with high probabilities at more distances, and also nodes are found where the probability of finding particles is low. It is important to mention that the first peak (at distance σ) is always represented, which means that this distance is where it is most probable to find particles.

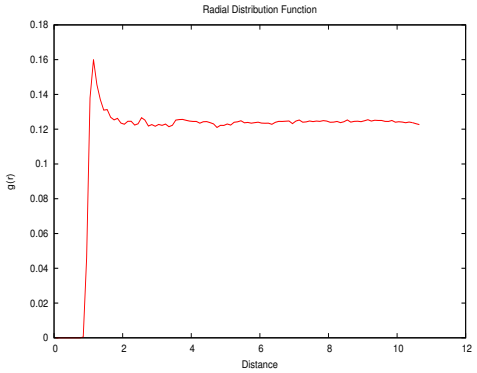
It can be concluded that as the system density increases, the particles have the tendency to organize in a certain structure to minimize the energy, and because of that maxima and minima are observed and become more defined as the density increases.



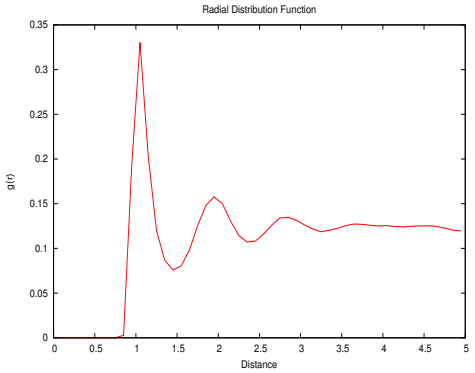
(a) RDF with $\rho = 0.001$



(b) RDF with $\rho = 0.05$



(c) RDF with $\rho = 0.1$



(d) RDF with $\rho = 1.0$

Figure 3.5: Captions showing the radial distribution function (RDF) at different values of ρ.

Chapter 4

Scalability study of parallel code

To evaluate the improvement of the program performance due to the parallelization, an scaling study is performed. Time data is collected running the program with 1, 2, 4, 8, 16 and 32 cores. The parameters used are:

Parameter	Value
M	32
Density	0.05
Cutoff ratio	0.5
Time delta	0.01
Seed	350
Sigma	1.0
Temp	0.3
Nsteps	200
Nstabilize	100
σ	1.0
ϵ	1.0
mass	1.0
Andersen probability	0.01
dr size	0.1
Integrator	Verlet

Table 4.1: Parameters used to run the simulation. All values are in reduced units.

This study was performed in the Nord3 cluster (BSC Barcelona Supercomputing Center). In the cluster, every node has 2 processors with 8 cores each.

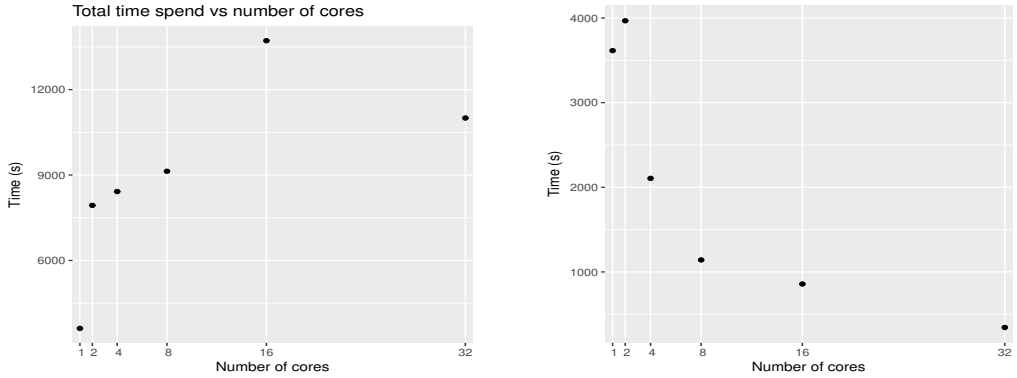
For this study, a series of time collectors (called *RIPTIME*) were added to the main program. Thanks to them, it is possible to see how much time the program takes in the different sectors of the program and also the total amount of time spent

in performing the whole simulation. In addition, it can be calculated a mean value for the time spent per core to perform the different tasks.

All the obtained information is presented in Table 4.2, and the important data is represented in the Figure 4.1.

	1	2	4	8	16	32
RIPTIME(1)	29.1338196	16.8250504	10.844677	8.06450272	9.03256416	5.95237112
RIPTIME(2)	2588.76978	1310.16992	692.34729	374.115387	278.653412	105.80661
RIPTIME(3)	2588.76978	1310.16992	692.34729	374.115387	278.653412	105.80661
RIPTIME(4)	7798.45947	3961.63965	2104.8916	1148.05579	868.058228	343.30545
RIPTIME(5)	7798.46338	3961.64355	2104.89551	1148.05569	868.062256	343.309448
TOTAL	7798	7935	8421	9133	13717	11002
TOTAL/Core	7798	3967.5	2105.25	1141.625	857.3125	343.8125

Table 4.2: Table showing the time results from the scalability study.



(a) **(b)**

Figure 4.1: Graphic representation of the data from Table 4.2. (a) Caption that show the total time spent to perform the simulation with a given number of cores. (b) Caption that show the time spend in a single core in the different groups.

From the Figure 4.1a it can be concluded that running the program in more than one cores is more time consuming. This happens because the communication between the cores to share the data is time consuming. As the number of cores increases, it also increases the time. The exceptions is found with 32 cores and this fact can be because two nodes are reserved for the work, which means that there are no other works consuming the processor resources.

In Figure 4.1b, it is represented the time spent per core in the different runs, which means that the values were calculated as $\frac{Total time spent}{number of cores}$. As it can be seen, from 1 to 2 cores the time increases. This happens because it is added the communication

between the cores to the calculation, and as it has been mentioned before, it is time consuming.

But, as the number of cores increases, the mean time spent decreases. When the program has control of an entire node instead of sharing resources with other works, the program runs faster because it has full access to the node supply.