Molecular Dynamics Simulation of a Van der Waals Gas

Authors: Oriol Cabanes, Sílvia Álvarez, Elena Ricart, Alexandre Sureda and Laia Navarro

Facultat de Química i Física, Universitat de Barcelona. Carrer de Martí i Franquès, 1-11, 08028 Barcelona

Abstract: In this study, a simple molecular dynamics program has been built to deal with a Van der Waals gas (specifically helium) of N particles interacting through a Lennard-Jones potential in contact with a heat bath. The main goal of this project has been the parallelization of a sequential code to obtain, in a more effective way, several magnitudes such as energy, pressure and instantaneous temperature.

All codes have been performed using Fortran 90 language.

INTRODUCTION

Computer simulations try to reproduce the behaviour of a physical system by solving a mathematical model (usually not solvable analytically) with a given accuracy (realistic interactions and many particles). Molecular dynamics (MD) is a computer simulation in which atoms and molecules are allowed to interact for a fixed period of time, giving a view of the classical "evolution" of the system. The trajectories, energy and forces of the system are obtained by solving the Newton's equations of motion for a system of interacting particles.[1][2] To solve these ODE's numerically, it is necessary a good integrator. This allows us to obtain a temporal evolution of the system in specific conditions, which can be, NVE, NVT, NPT/NσT, etc.

Computationally, the simulations are "large" in two domains, the number of atoms and the number of timesteps. Many thousands or millions of atoms may usually be simulated to approach the micro-scale state. These conditions limit the simulations to the femtosecond scale, and thousands of timesteps are needed to simulate even picoseconds of "real" time.

Due to these computational demands, considerable efforts have been made over the last few decades to reduce the computational effort required for MD simulations.

Theoretically, adding more processors to carry out a simulation leads to a smaller execution time compared with its execution time using a single machine. But in practice, this hypothesis is not always satisfied.

The main goal of parallelizing a serial program is to obtain a faster program, then the criterion to be considered is the speedup gained in comparison of the serial program.

In this project we present a parallel Molecular Dynamics simulation for short range interaction with three parallel force algorithms.

CASE STUDY

Considering that interaction between helium atoms can be described using a Lennard-Jones potential with $\epsilon = 0.045439\ kJ\ mol^{-1}$ and $\sigma = 2.64$ Å (and the atomic mass of helium is $m = 4.0026\ g\ mol^{-1}$), we have constructed a sequential code to describe the dynamics of a system of N particles of this gas in a canonical ensemble (NVT ensemble).

Once this sequential code has worked properly, we have built the parallel program from it.

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

Lennard-Jones potential.

SEQUENTIAL PROGRAM

main.f90 contains the main program of the sequential part of the project, *SEQUENTIAL_MD*.

To start the study, we create an initial configuration by placing particles in a face-centered cubic (FCC) lattice inside of a cubic simulation box of length

$$L = \left(\frac{N}{\rho}\right)^{\frac{1}{3}} \quad (2)$$

taking care that there are no overlaps (it is located in the subroutine *FCC_Initialize*.

Then, we initialize the velocities of the system of N particles in a uniform distribution (it is also in Full_modul_Inicialitzar.f90: module *Inicialitzar*, subroutine *Uniform_velocity*) and we rescale them so that the kinetic energy is consistent with a given temperature (it is also in Full_modul_Inicialitzar.f90: module *Inicialitzar*, subroutine *Velo_Rescaling_mod*).

After that, and taking the rescaled velocities, we carry out a melting by performing a short simulation of the system in contact with a heat bath at high temperature: $k_B T = 100\epsilon$. This ensures that our system is in a fluid state with atoms not in fixed positions.

Then, we can perform the simulation of interest (the system in contact with a heat bath). Both melting and the main simulation are obtained by applying the velocity Verlet and the Andersen's thermostat. The heat bath of the main simulation is set at $k_B T = 54.65\ \epsilon$, that is 300 K.

As said above, to solve the equations of motion we have used the velocity Verlet algorithm which is convenient for thermalizing. To solve Newton's equations numerically we have discretized time into little time steps *dt*. Knowing the position of all particles at time *t* we can calculate the force on each of them at time *t*. To know the new positions of particles at time *t* + *dt* we solve Newton's equations. Once the positions of all particles are updated at this new time, we can calculate the forces again in all particles at time *t* + *dt*. Then, the trajectory of the particles is obtained repeating this sequence.

(The velocity Verlet is in verlet.f90 file: *Verlet_Algorithm* module, *VELO_VERLET* subroutine. The forces and the potential energy are calculated by using subroutines of two different modules: energy.f90: *Interaction_Cutoff_modul* module, *INTERACTION_CUTOFF* subroutine and L_J.f90: *Lennard_Jones* module, *L_J* subroutine. The Andersen's thermostat is in Full_Modul_Andersen.f90: *Andersen_modul* module, subroutine *Andersen* subroutine).

Velocity Verlet algorithm:

$$r_i(t + \Delta t) = r_i(t) + v_i(t)\Delta t + \frac{f_i(t)}{2m_i}\Delta t^2 \quad (3)$$

$$v_i(t + \Delta t) = v_i(t) + \frac{f_i(t) + f_i(t+\Delta t)}{2m_i}\Delta t \quad (4)$$

With probability $\Gamma\Delta t$, particle i interacts with bath (stochastic impulsive force) and its velocity is changed to:

$$v_i(t + \Delta t) = Normal\left(\mu = 0, \sigma = \sqrt{\frac{k_B T}{m_i}}\right) \quad (5)$$

where normal indicates random number normally distributed. To generate normally distributed numbers, we use the Box-Muller transformation:

$$Z_0 = \sigma\sqrt{-2log(U_1)}cos(2\pi U_2) \quad (6)$$

$$Z_1 = \sigma\sqrt{-2log(U_1)}sin(2\pi U_2) \quad (7)$$

where Z_0 and Z_1 are independent random variables with a normal distribution with average $\mu=0$ and standard deviation σ.

The magnitudes calculated in this project are potential energy, kinetic energy, total energy, pressure and instantaneous temperature. The final values of these magnitudes and their statistical treatment are in sample.f90: module *SAMPLE*, subroutine *SAMPLES*.

We have also calculated the radial distribution function, which is in sample.f90:

SAMPLE module, *gdr* subroutine, calling the *RAD_DIST_INTER* and *RAD_DIST_FINAL* subroutines that are in *Distribucio_Radial* module in Full_Modul_Distrib_Radial.f90.

It is important to notice that we have used periodic boundary conditions (PBC) so as not to lose particles during the simulation and to reproduce an "infinite" system (PBC.f90: *PBC* module, *PBC1* and *PBC2* subroutines).

Results and discussion

We have performed two types of calculations: without Andersen's thermostat and with Andersen's thermostat.

Parameters:

- Number of particles: 500
- density: 0.8 a.u
- Final time of the simulation: 50 a.u.
- Time step: 0.0005 a.u.
- σ: 2.64 Å
- ϵ: 0.045439 kJ/mol
- atomic mass: 4.0026 g/mol

We can see in *Figure 1*, that when our system is not in contact with a heat bath, the total energy is perfectly conserved as expected.

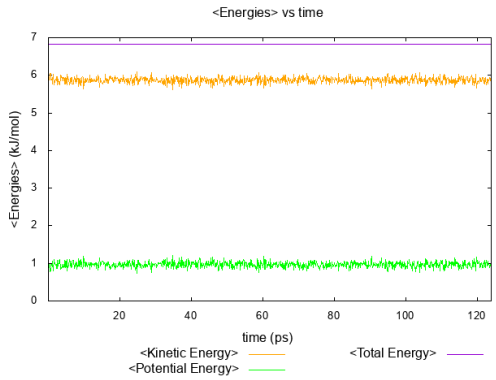

Figure 1. Energy results when the system is not in contact with a heat bath.

The graphic below contains the energy results when the system is in contact with a heat bath of $T = 300\,K$. The thermostat works properly since the instantaneous temperature obtained is the heat bath's temperature (see *Figure 2*).

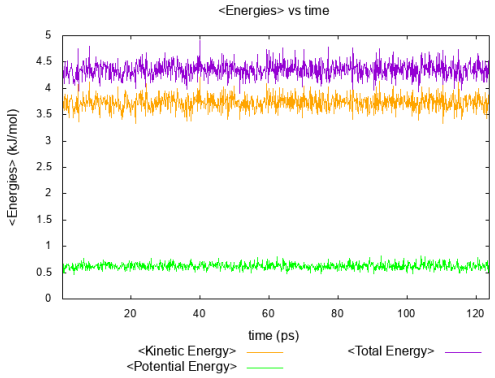

Figure 2. Energies when the system is in contact with a heat bath.

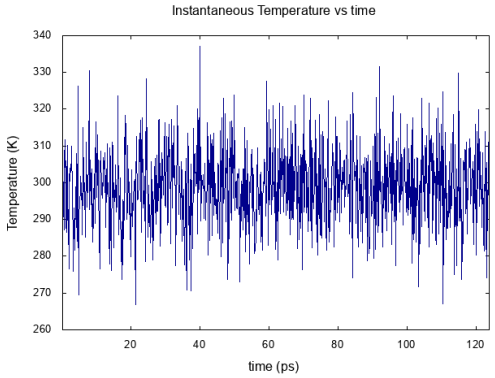

Figure 3. Instantaneous temperature when the system is in contact with a heat bath.

The results for the pressure are in *Figure 4*, and the results for the radial distribution function are shown in *Figure 5*.

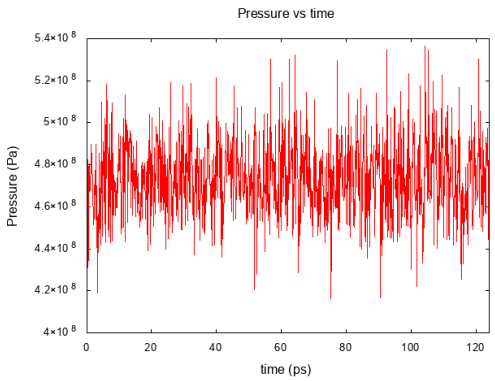

Figure 4. Pressure when the system is in contact with a heat bath.

The pressure has the order of magnitude expected.

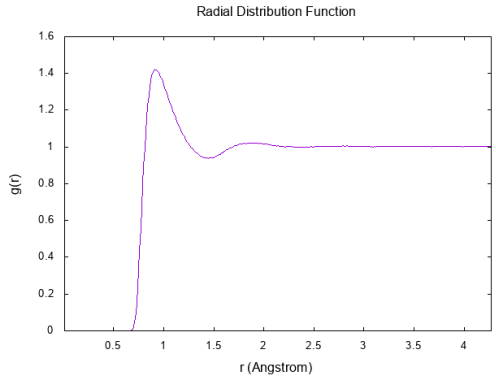

Figure 5. Radial distribution function when the system is in contact with a heat bath.

PARALLEL PROGRAM

To parallelize the sequential program, we have used the MPI (Message Passing Interface).

In order to use any MPI routine, every program, subroutine, and function that calls an MPI routine must include the header file 'mpif.h', immediately after the implicit statement.

See *Annex 1* for a detailed explanation of the MPI routines used.

Modules and subroutines

parallelization.f90

The algorithm used in this project to distribute the work for all the CPU's is called atom decomposition, each of the P processors is assigned a group of N/P atoms.

For implement the algorithm efficiently we must work with a balanced partition of particles for each processor. At first, we declare two important variables: the first one (*res)*, helps us to know the remainder of N/P particles. The second important declared integer variable (*a)* is the result of the previous division.

Atom decomposition algorithm

Input: n-particles and n-processors
Output: Quotient and remainder of N divided by P
For each CPU
  if P < res **then**
    first particle = (a+1) *CPU + 1
    last particle = first particle + a
  else
    first particle = CPU*a + res +1
    last particle = first particle + a -1
  end
end

In this way the distribution is balanced, since with the conditional that we implement, the vector is filled gradually for each processor.

Otherwise we will fill with the same number of particles for each processor but depending on the N / P residue, the last processor would be assigned many more particles than the other CPUs and we would have a totally unbalanced distribution.

To make the correct MPI communication between CPU's we need to define two more variables, one that will make us a pointer of each start of the N/P array stored in each processor, and the amount of particles stored in each processor.

Full_Modul_Inicialitzar.f90

In this work the initialization of the positions in an FCC lattice and the initial uniform velocities are computed only in one processor. Then with the MPI_BCAST operator all initial states of the particles are stored in each CPU.

This subroutine is not parallelized because does not require much computational effort.

energy.f90

The matrix force of the system is an N x N where N is the number of atoms. The i, j-th element holds the potential energy due to interaction of molecule i with molecule j. The diagonal elements of this matrix are 0, because the atom does not interact with itself.

These interactions can be computed for i = 1 to N-1 and j = i+1 to N and taking the advantage of the symmetry of the matrix all the interactions can be calculated.

This symmetry works perfectly for one processor, but in the multi process procedure, more considerations must be present for an efficient implementation.

Symmetric Matrix algorithm

In this algorithm the only parallelization that is performed is on the i atoms and taking advantage of the symmetry matrix we compute j = 1 + i to N.

Input: position r of all the particles
For each CPU
  For i = first particle **to** last particle
    For j = i+1 **to** N
      Compute
       Distance between i-j
       $F_{x,y,z}(i)$
       $F_{x,y,z}(j)$
       Compute Potential, Pressure
    end
  end
end
All Reduce(F**)** all-to-all sum
Reduce (Potential) Sum of all the CPU
Reduce (Pressure) Sum of all the CPU
Output: Force, Potential and Pressure

When the loop finalizes an All Reduce is computed to get the total interaction force of each particle. The total potential and pressure of the system is computed with a Reduce operation.

Dividing the atoms among processors and computing the interactions for i and j = i +1 we obtain an unbalanced code. See Figure 6, looking the schematic of the force loop, can be seen clearly that the processor 0 has more to compute than any other CPU. This is not a balanced parallelization and we expect that in large systems of atoms will give poor parallel performance.

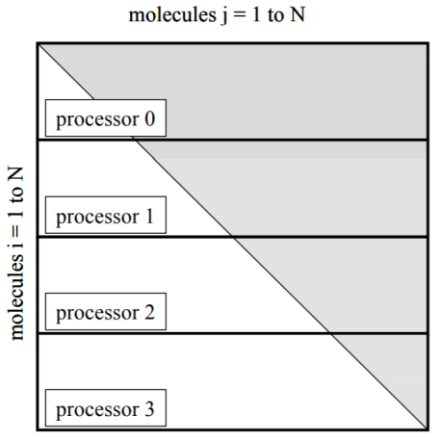

Figure 6. Scheme of Symmetric Matrix.

Extracted from [4]

Double Work algorithm

This method is very simple since ignores the symmetry of the force matrix and computes the interactions independently.

Input: position r of all the particles
For each CPU
  For i = first particle **to** last particle
    For j = 1 **to** N
      Compute
      Distance between i-j
      $F_{x,y,z}(i)$
      Compute Potential, Pressure
    end
  end
end
Reduce (Potential) Sum of all the CPU
Reduce (Pressure) Sum of all the CPU
Output: Force, Potential and Pressure

In the double work matrix, each CPU stores all the interactions of the atom i, for this reason the All Reduce operation is not used because each processor has the forces of all the range particles stored.

This code is totally balanced, each processor computes the same number of interactions, is characterized for minimizing the processor communication but with very computational effort for each CPU.

From this method, we expect a good parallel performance for large systems of particles and large number of CPUs.

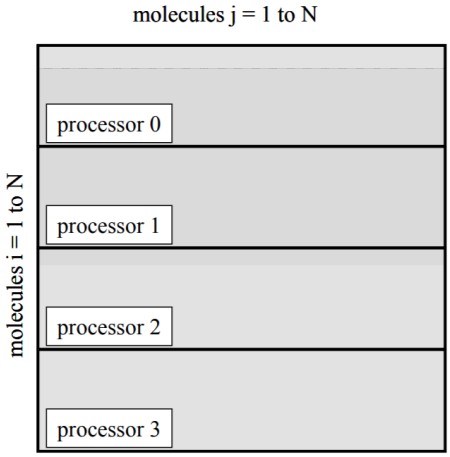

Figure 7. Scheme of Double Work Matrix.

Extracted from [4]

Symmetric and Balanced algorithm

This method takes the advantage of the symmetric force matrix used in the symmetric matrix algorithm and at the same time each processor has an equal amount of work.

The goal of this method is to implement a pair list for each atom, the list contains all the pair i-j atoms, then with the parallelization algorithm all the pairs are distributed uniformly to all the workers.

Input: n-particles
Output: Number of pair particles
For 1 **to** n-pairs
  Compute pair i-j
end
Output: Pair list(i-j)

Input: position r of all the particles
For each CPU
  For i = first pair **to** last pair
    Use Pair list(i-j)
      Compute
       Distance between i-j
       $F_{x,y,z}(i)$
       $F_{x,y,z}(j)$
       Compute Potential, Pressure
    end
  end
end
All Reduce(F**)** all-to-all sum
Reduce (Potential) Sum of all the CPU
Reduce (Pressure) Sum of all the CPU
Output: Force, Potential and Pressure

When the loop finalizes, an All Reduce is computed to get the total interaction force of each particle because each processor has stored the interaction force evenly distributed.

This method provides the minimum computational cost but an expensive communication between CPUs.

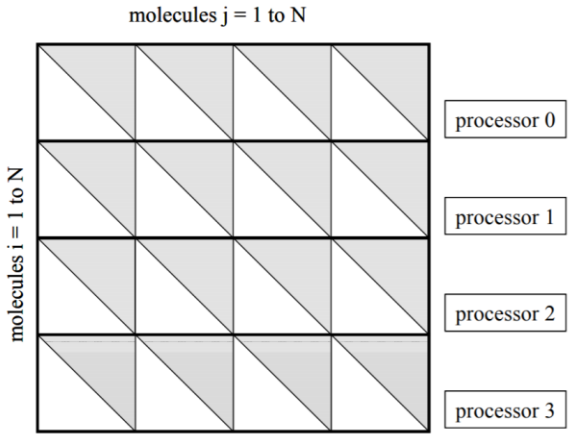

Figure 8. Scheme of Pair Matrix.

Extracted from [4]

verlet.f90

The velocity Verlet algorithm is computed for each partition of particles of each processor, for initialize the integration we need all the initial positions and velocities of all the particles stored on each CPU. This step is very important because the force is computed for each particle and all the pairs. When the force is computed, the first integration step is done for all the N/P particles. The next step is to do an all-to-all communication with all the processors, the function Allgatherv stores all the new positions of all the particles in each CPU for compute the force on the next step.

With all the forces calculated for each particle we can proceed to the second integration step where the new velocities are calculated.

Verlet algorithm

Input: initial position r and velocity v
For each CPU
  Compute Force
  For first particle **to** last particle
    Compute
      $r(t + \Delta t) = r(t)+v(t)*dt+0,5*F(t)*dt^2$
      $v(t + \frac{\Delta t}{2}) = v(t)+0,5*F(t)*dt$
    Apply PB conditions
  end
Allgatherv($r(t + \Delta t)$**)** all-to-all communication
  Compute Force
  For first particle **to** last particle
    Compute
      $v(t + \Delta t)=v(t)+0,5*F(t)*dt$
  end
end

Full_Modul_Andersen.f90

In this module, the Andersen's thermostat is created, using random numbers. It uses the Box-Muller transformation to obtain a uniform distribution of our velocities, adjusting them to a specific temperature. Now, these velocities are going to have the main role in the kinetic energy calculation.

Andersen algorithm

Input: position r and velocity v
For each CPU
  For first particle **to** last particle
    Compute
      $v_x = \sqrt{-2\,T\,\log(n_1)}\,\cos(2\,\pi\,n_2)$
      $v_y = \sqrt{-2\,T\,\log(n_1)}\,\sin(2\,\pi\,n_2)$
      $v_z = \sqrt{-2\,T\,\log(n_3)}\,\cos(2\,\pi\,n_4)$
  end
end

main.f90

This is the main program which contains its structure by using different modules and calling their subroutines in a sorted way.

First, we initialize the MPI with *MPI_INIT* routine and assign values to the taskid variable with *MPI_COMM_RANK* routine. Then, we call *MPI_COMM_SIZE* routine which to determine and store the number of processors into *numproc* variable, and finally we initialize the time with *MPI_WTIME*.

Once the MPI is initialized, we can start with the study of interest.

The first step is the initialization of the system by including all the variables and parameters and distribute all the atoms for each CPU with the parallelization algorithm.

We generate an FCC lattice and initialize the velocities so that the kinetic energy is consistent with the temperature . Then using the *MPI_BCAST* routine, all the initial state of the particles is stored on each CPU. The next step is thermalizing the system applying the velocity Verlet and the Andersen's thermostat for a few simulation steps. Once our system is in fluid state, we carry out the simulation of interest.

So, we use the Force algorithm and the velocity Verlet algorithm with the Andersen's thermostat to obtain new positions and velocities.

When the integration loop finalizes, the *MPI_GATHERV* is used to store velocities from each CPU to the Master processor. This step is required because the final values and their statistical treatment in Sample.f90 are carried out by the master processor. In this module, we have also calculated the radial distribution function.

To finish, we use *MPI_FINALIZE* to terminate the processor communication.

Results and discussion

We have performed two types of calculations with the same parameters as the sequential calculations seen previously, without Andersen's thermostat and with Andersen's thermostat.

Parameters:

- Number of particles: 500
- density: 0.8 a.u
- Final time of the simulation: 50 a.u.
- Time step: 0.0005 a.u.
- σ: 2.64 Å
- ϵ: 0.045439 kJ/mol
- atomic mass: 4.0026 g/mol

The results are close to those obtained in the sequential program. The energy is conserved as expected since the system is isolated.

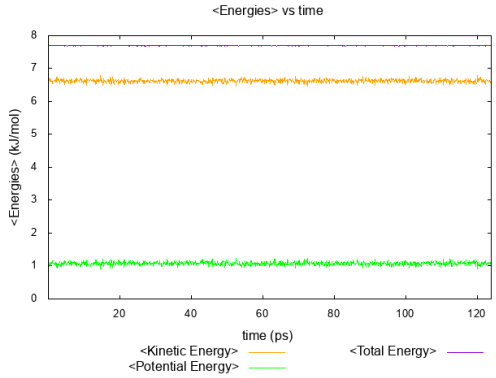

Figure 9. Energy results when the system is not in contact with a heat bath.

The graphic below shows the energy results when the system is in contact with a heat bath of $T = 300\,K$. The thermostat works properly since the instantaneous temperature obtained is the heat bath's temperature (see *Figure 10*).

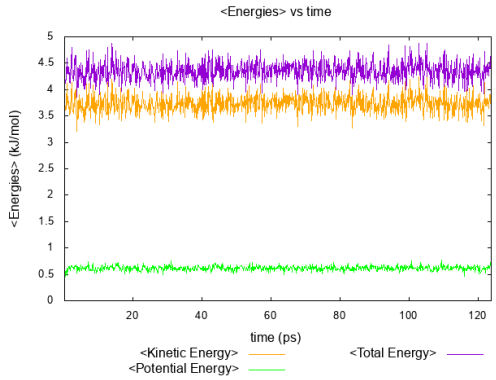

Figure 10. Energy results when the system is in contact with a heat bath.

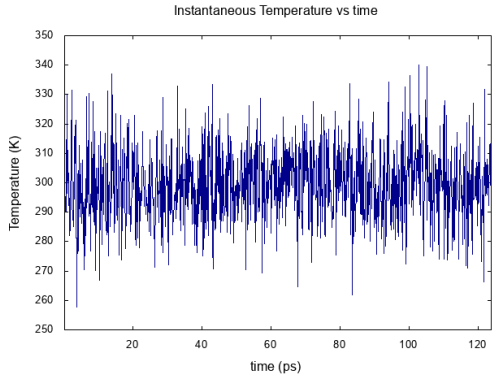

Figure 11. Instantaneous temperature when the system is in contact with a heat bath.

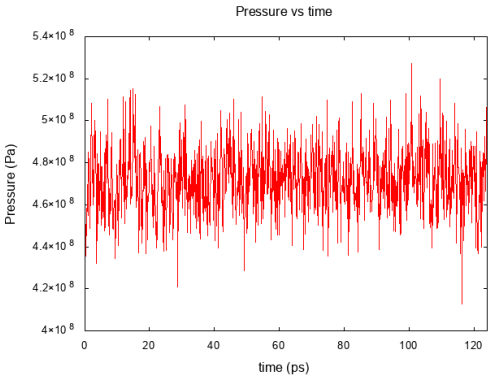

Figure 12. Pressure when the system is in contact with a heat bath.

The pressure has the order of magnitude expected.

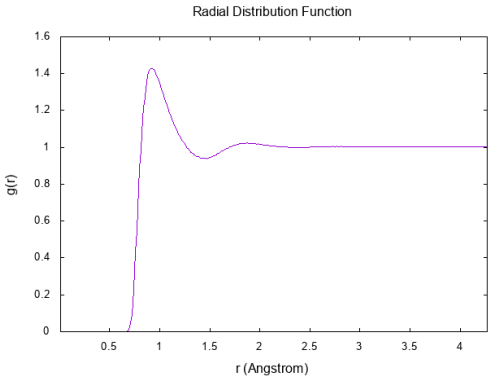

Figure 13. Radial distribution function when the system is in contact with a heat bath.

SPEEDUP

Since we have performed three parallel force algorithms a speedup and an efficiency comparison have been done.

In this way, we can study how each algorithm scales by changing the number of particles and the number of CPUs

- **Symmetric Matrix Algorithm (SM)**

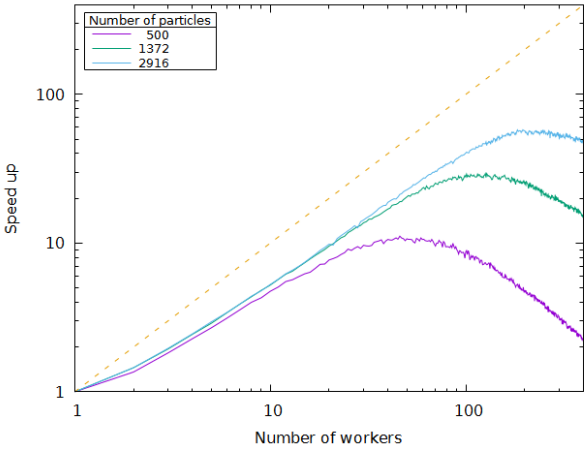

Figure 14. Speedup analysis of the SM algorithm for different number of particles.

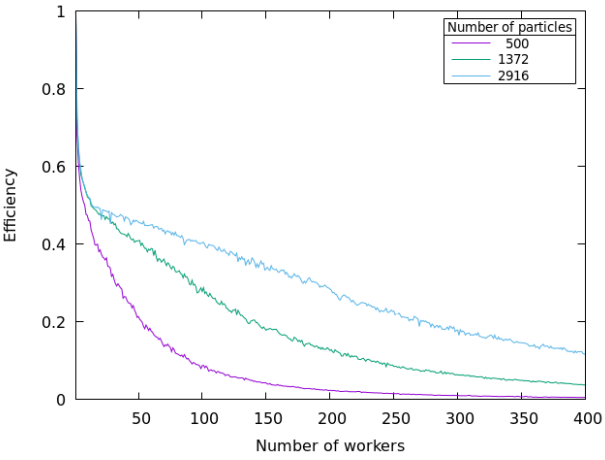

Figure 15. Efficiency analysis of the SM algorithm for different number of particles

- **Double Work Matrix (DW)**

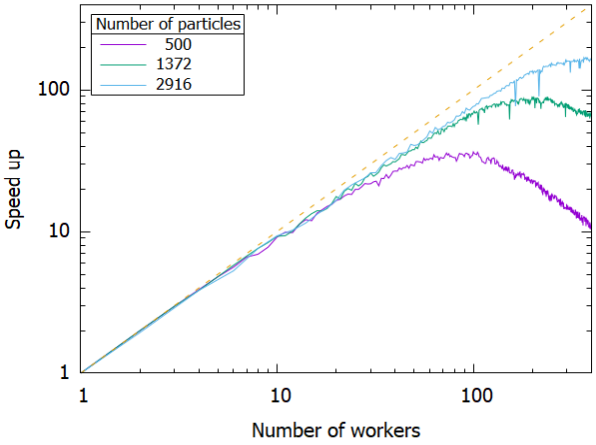

Figure 16. Speedup analysis of the DW algorithm for different number of particles

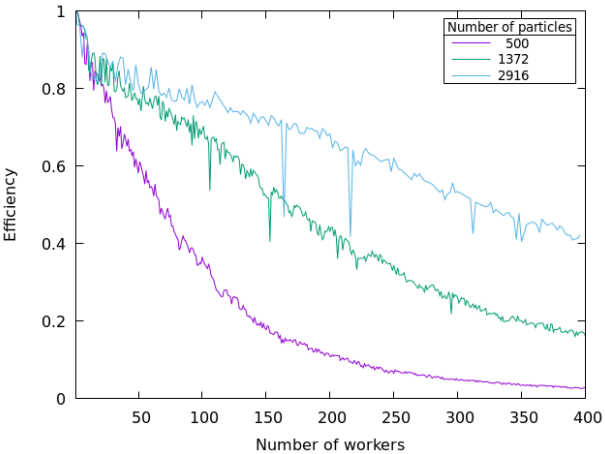

Figure 17. Efficiency analysis of the DW algorithm for different number of particles

- **Symmetric and Balanced algorithm (SBM)**

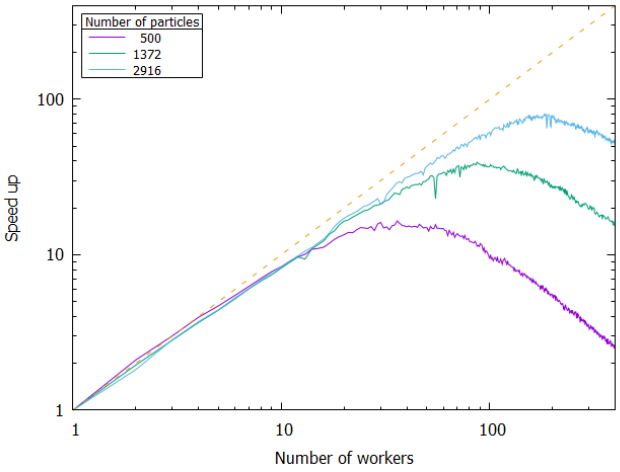

Figure 18. Speedup analysis of the SBM algorithm for different number of particles

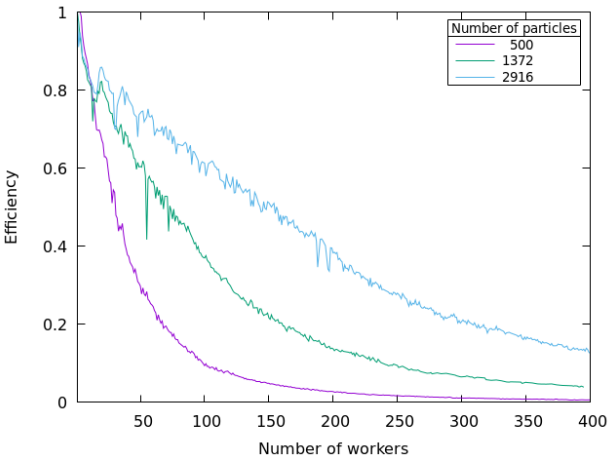

Figure 19. Efficiency analysis of the SBM algorithm for different number of particles

As it can be seen, the Symmetric Matrix algorithm does not scale correctly for small number of particles. Although as the system size increases the algorithm gets faster execution times in comparison with the serial program. With small number of particles, as the number of processors increases, it causes the system to become completely unbalanced, making the inter-process communication very inefficient.

This it can be seen in the efficiency analysis, the first simulations with a few numbers of CPUs the efficiency is greatly reduced with respect to the serial scheme and as the number of processors increases the curve stabilizes. With high number of particles, the allocation of work for each CPU is still poorly distributed, although you can see an increase in the execution speed as the number of processors increases. This is because the serial program with a very large number of particles requires a lot of computational effort, however, in the parallel program in this case the computational effort is better distributed, making the poor communication between processors not so important.

In the case of the double work algorithm, we see that it has a very different behaviour to the previous algorithm. This is because each processor method calculates the same number of interactions, so communication between processors is always balanced. The only limitation that this algorithm has is the computational cost of each processor since the communication is very good, therefore, as the system increases in number of particles the speedup increases considerably with respect to the serial program. This tendency is reaffirmed with the efficiency analysis

The last algorithm has a behaviour between the two previous algorithms, because in this case we use symmetry of the force matrix and at the same time we distribute the same interactions for each processor. Therefore, for few particle numbers we obtain better results than in the case of the symmetric matrix algorithm, since the communication between processors is better due to the uniform distribution of work unlike the first algorithm. However, with respect to the second algorithm, it is appreciated as the speedup continues being smaller, since in this case by this method we have a small computational cost, but much inter-process communication causing that the difference with the serial program is not very high.

As the number of particles increases, scales better because communication between processors becomes more efficient, although not better than the Double Work algorithm as communication is not limited. Surely for much higher particle systems will obtain that

this algorithm we could obtain higher efficiencies than to the other two algorithms, since it has no limitation on the cost of calculation and communication will be homogenized.

To reaffirm our explanations, we present a run time comparison of the algorithms for the same simulation times and different numbers of particles.

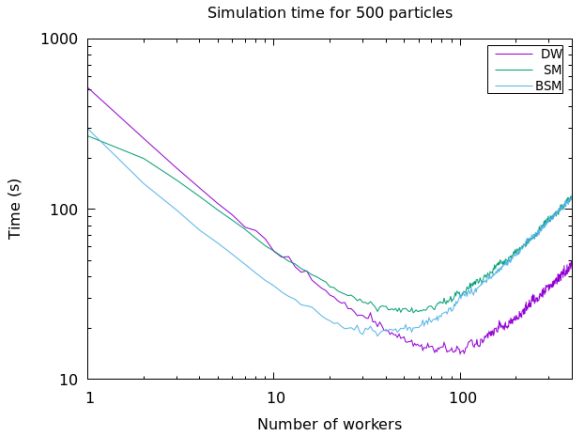

Figure 20. Run time comparison of the algorithms .

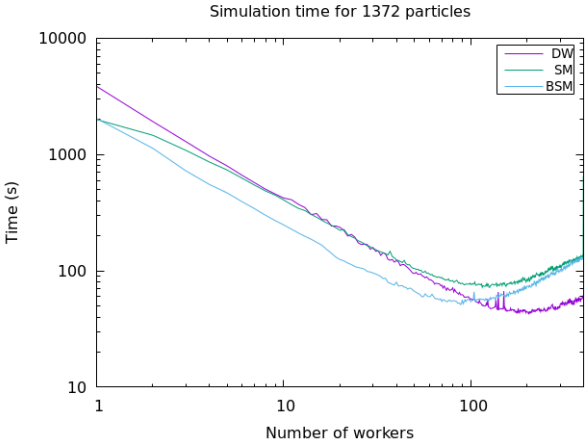

Figure 21. Run time comparison of the algorithms .

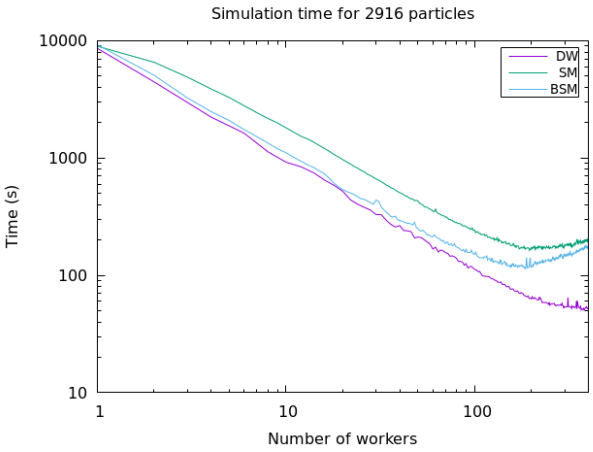

Figure 22. Run time comparison of the algorithms.

CONCLUSIONS

Firstly, from the speedup analysis we can see that more CPUs doesn't mean the best computational performance. Each program has an optimal ratio of number of particles and processors.

In this work we have constructed and implemented three kinds of parallel algorithms for MD simulations, each of them has advantages and disadvantages. The atom decomposition algorithm for distribute the work to each CPU is simple to implement and stores the particles in a balanced way. But it needs all-to-all communications, this communication costs makes runtime increase with many processors.

The comparison between the algorithms we can extract one of the most important factor that affect the speedup and the efficiency of the parallelization, if the problem can be divided into independent subparts and no communication is required, except for initialize the simulation and combine the final results, the speedup becomes linear. The Double Work method is the algorithm that exhibits more this tendency because less communications are required. If there are more communications involved like in the two other algorithms like the all-to-all communication used for the force, the speedup of the resultant parallel simulation becomes proportional to the computation communication ratio.

Surely if we increase the number of particles much more, the speed up and the efficiency would remain constant for the dual-work algorithm, for the computational cost limit, but it would increase for the BSM algorithm, since it has a reduced computational cost and communication between processors would be more homogeneous.

With all the results analysed, the more versatile algorithm for this range of number of particles is the Double Work algorithm.

REFERENCES

[1] C. J. Cramer, *Essentials of Computational Chemistry Theories and Models*. 2004.

[2] *F. Jensen, Introduction to Computational Chemistry. 2014.*

[3] https://mpitutorial.com

[4] D. Keffer, A primer for parallel implementation of molecular dynamics simulations, Dept. of Chemical Engineering, University of Tennessee, February, 2003

ANNEX 1: MPI ROUTINES USED

a) Routines to initialize MPI

1. **MPI_INIT(ierror)**

Initializes mpi. It is found in the main program. This routine is called once, before any other MPI routines are started.

2. **MPI_COMM_RANK(MPI_COMM_WORLD, taskid, ierror)**

Assigns a unique number from 0 to N_{proc}-1 (in the variable named taskid) to each processor. This routine is called once, in the main program, immediately after MPI_INIT.

MPI_COMM_WORLD: is an MPI intrinsic variable, defined and used by MPI. $taskid = rank\ of\ the\ processor$ and $ierr = 0$ if the subroutine exists without error.

One processor will be the root processor. In our case, we have named the root processor master. We have set $master = 0$.

3. **MPI_COMM_SIZE(MPI_COMM_WORLD, numproc, ierr)**

Determines the number of processors and stores that number in the variable named numproc. This routine is called once, in the main program, immediately after MPI_COMM_RANK.

4. **MPI_WTIME()**

Returns the current time and it is called twice in the main program. First, it is called immediately after MPI_COMM_SIZE, and the second time is called before MPI_FINALIZE which terminates the program.

It is available when MPI is used, so it is an advantage in relation to other time counters.

b) Routines to establish communication between processors

5. **MPI_BCAST(global_vector, length of the global_vector, MPI_DOUBLE_PRECISION, master=0, MPI_COMM_WORLD, ierr)**

Broadcasts a vector of values from the processor with $taskid = master$ to all other processors.

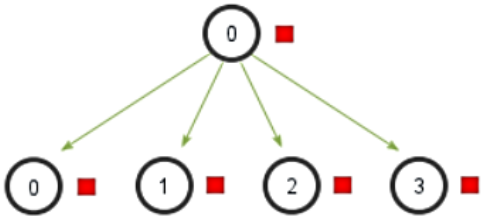

Fig A1. MPI_BCAST scheme.[3]

MPI_DOUBLE_PRECISION is a default MPI variable that allows us to work better with vectors with double precision.

This MPI extension is found in the main program, before the melting process. It is also in the initialization module to send the initial state information to all the others, before the velocity calculation started.

6. **MPI_GATHERV(send_data, send_count, MPI send_data_type, receive_data, receive_count, receive_data_type, master=0,MPI_COMM_WORLD)**

Takes elements from many proceses and gathers them to one single process; gathers them to the root process. The elements are ordered by the rank of the process from which they were received. That is, all the information of each processor goes to the root master.

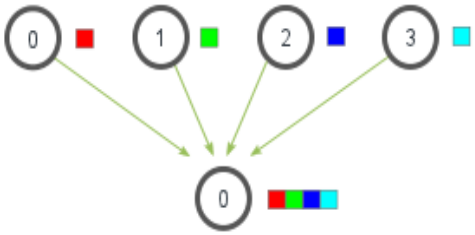

Fig A2. MPI_Gather scheme.[3]

7. **MPI_ALLGATHERV(local_vector, length of the local_vector, MPI_DOUBLE_PRECISION, global_vector, num_send, desplac, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)**

This routine gathers a scattered vector of values from all processors, and gathers them back into the global vector, storing a copy on each processor. That is, each processor gives the information to the others (including itself). When it finalizes, all the processors have the whole information.

The vector num_send of length equal to the number of particles of each CPU

The vector desplac is the pointer of the beginning of the particle arrays of each CPU

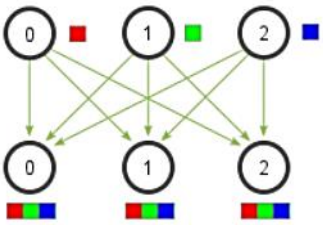

Fig 3. MPI_ALLGATER scheme.[3]

8. **MPI_REDUCE(send_data, recieve_data, MPI_DOUBLE_PRECISION, size of the data=1, MPI_OPERATION, master=0, MPI_COMM_WORLD)**

This routine can perform different types of calculation and return, for example, the maximum and the minimum value, the sum of the elements, the product of the elements, the rank of the process that owns the maximum or the minimum value, etc.

In our case, we have used MPI_SUM.

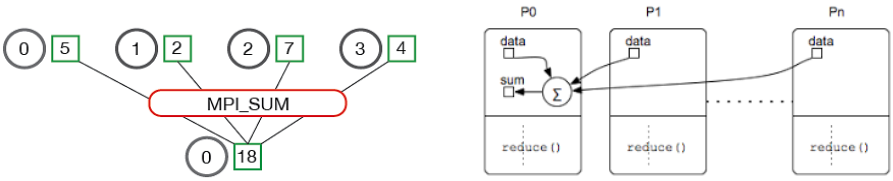

Fig 4. MPI_SUM example schemes.[4]

c) Routine to terminate MPI

9. MPI_FINALIZE(ierr)

Terminates the processor communication.