MICHIGAN STATE UNIVERSITY

Project Report for Parallel Computing Course

"Molecular Dynamic simulations of the atoms of a Noble gas using the Lennard-Jones potential"

Fatemeh Fathi Niazi
Department of Computational Mathematics, Science and Engineering

fathinia@msu.edu

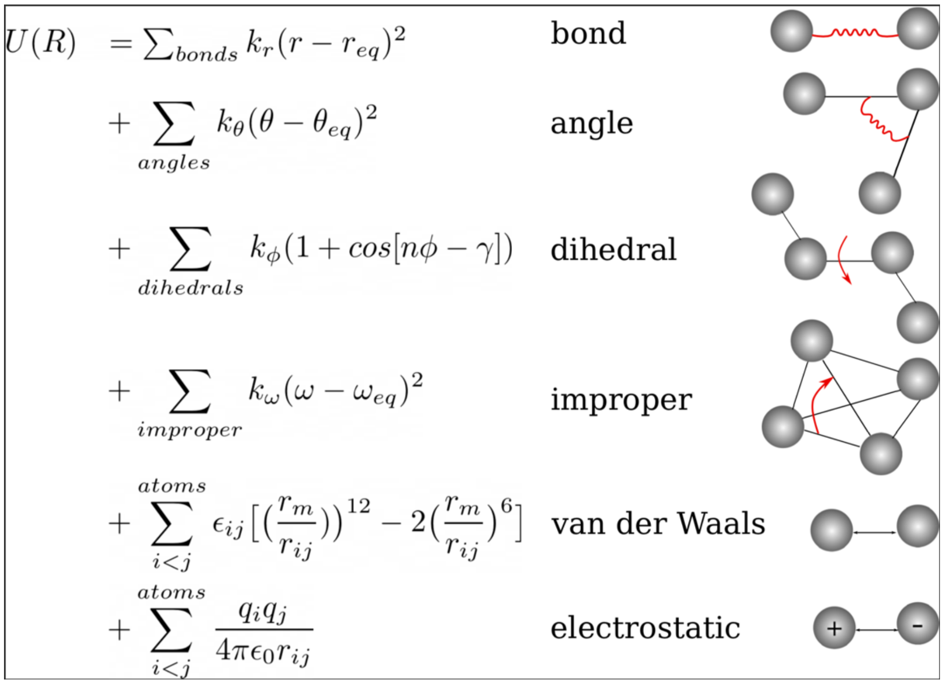
December 11

Abstract

We use molecular simulations to reduce the cost of experiment and search space. Through it, we can select the best result from simulations and do experiments with the selected parameters. In Molecular Dynamic (MD), you start with a given number of atoms and you want to be able to simulate to predict how these atoms will be moving according to the time. The result is a trajectory of all particles in the system as a function of time. Other properties like temperature, pressure, kinetic energy etc. can also be calculated. Parallelizing MD code is important because MD simulation is very time-consuming and expensive. The main reason for that is you should calculate all forces between all atoms of the system. For large number of atoms, it might be impractical. The goal of this project is to run MD simulations with large number of atoms using parallel version of force field function. We compiled and ran our codes using hpcc, intel 18 developed nodes (Two 2.4Ghz 20-core Intel Xeon Gold 6148 CPU (40 cores total)). In the parallel version of this code, "OpenMP" tools was used to make the serial version faster. We first made sure that the parallel code generates the same results, and then we compared its performance to the serial code and saw that the parallel code is much faster than serial code. Additionally, we found out that by using intel18 development node on hpcc, the optimum number of threads is 30. At the end, we visualize the trajectory of the simulation by VMD software.

1 Introduction

Classical molecular dynamics simulation has been in use since the late 1970s to overcome the computational cost of calculating quantum-mechanical motions for a molecular system. MD simulations use Newton's laws to approximate atomistic motions which makes it several orders of magnitude faster than quantum methods. MD can be used to study the movements, interactions and structure of a molecular system at the atomic level. Advances in computational equipment, especially the use of graphics processing units (GPUs) and high-performance computers, improved the performance of MD. Therefore, simulating biological processes such as ligand binding and unbinding pathways ligand unbinding kinetics protein folding with system sizes on the order of 500,000 atoms is now routine. Now let's briefly describe how an MD simulation is performed. First, an initial structure of a biomolecular system is obtained from experimental methods. Then using potential energy, U, that includes bonded and non-bonded interactions (Fig. 1), the forces are calculated on each atom using F = -∇U. By having the forces acting on every atom, Newton's second law of motion, F = ma, is solved to determine accelerations and velocities and to update the atom positions.



$$
\begin{aligned}
U(R) \;&= \textstyle\sum_{bonds} k_r (r - r_{eq})^2 && \text{bond}\\
&+ \sum_{angles} k_\theta (\theta - \theta_{eq})^2 && \text{angle}\\
&+ \sum_{dihedrals} k_\phi (1 + cos[n\phi - \gamma]) && \text{dihedral}\\
&+ \sum_{improper} k_\omega (\omega - \omega_{eq})^2 && \text{improper}\\
&+ \sum_{i<j}^{atoms} \epsilon_{ij} \left[\left(\frac{r_m}{r_{ij}}\right)\right)^{12} - 2\left(\frac{r_m}{r_{ij}}\right)^6 \right] && \text{van der Waals}\\
&+ \sum_{i<j}^{atoms} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} && \text{electrostatic}
\end{aligned}
$$

Figure 1: Energy function U used for driving forces. The molecular models of different atomic interactions are shown on the right side while their corresponding equations are included on the left side.

The first four terms of Fig. (1) are assigned to covalent bonds and the rest are related to noncovalent bonds. In this simulation, because we suppose the atoms are for a Noble gas, there is no bond between atoms. So, all the covalent bonds terms will be zero. Another assumption of our system is that the atoms are electronically neutral. Therefore, the electrostatic energy will be zero too. Our simplified energy equation is shown in Eq. (1), which is also called Lennard-Jones potential. where r_{ij} is the distance between two interacting particles, ε_{ij} is the depth of the potential well, and σ is the distance at which the particle-particle potential energy is zero.

$$E_{(R)} = \sum_{i<j}^{atoms} 4 \times \varepsilon_{ij} \left[\left(\frac{r_m}{r_{ij}}\right)^{12} - 2\left(\frac{r_m}{r_{ij}}\right)^{6}\right] \qquad \textbf{Eq. (1)}$$

The question is that if we know the energies between atoms, how can we determine the motions, or trajectories of the atoms. The forces on each atom are given by the gradient of the energy function with respect to the atomic positions. For instance, the force on particle 0 in x direction can be calculated using Eq. (2):

$$F_{x0} = -\frac{\partial E}{\partial x_0} = -\frac{\partial}{\partial x_0}\left[\sum_{i=i}^{n-1}\sum_{j=i+1}^{n}\left[\left(\frac{r_m}{r_{ij}}\right)^{12} - 2\left(\frac{r_m}{r_{ij}}\right)^{6}\right]\right] = -\sum_{j=0}^{atoms}\frac{\partial}{\partial r_{0j}}\left[\left[\left(\frac{r_m}{r_{ij}}\right)^{12} - 2\left(\frac{r_m}{r_{ij}}\right)^{6}\right]\right]\frac{\partial r_{0j}}{\partial x_0} \qquad \textbf{Eq. (2)}$$

2 **Methodology**:

2.1. Initialization

For the first step, we need to set initial positions and velocities to all particles in the system. The particles positions and number of particles should be compatible with the structure and the length of the box that we want to simulate. We attribute to each velocity component of every particle a value that is drawn from a uniform distribution in the [0,1], this initial distribution is Maxwellian neither in shape nor even in width. The velocities are shifted, such that the total momentum is zero. The resulting velocities are scaled to adjust the mean kinetic energy to the desired value. In thermal equilibrium, Eq. (3) should hold, where ν_α, k_B,T *and* m are respectively assigned to α Component of the velocity of a given particle, Boltzmann constant, temperature and mass of each particle.

$$\nu_\alpha{}^2 = \frac{k_B \times T}{m} \qquad \textbf{Eq. (3)}$$

2.2. Energy and Force Calculation

This is the most time-consuming part of molecular dynamic simulations. If we consider a model system with pairwise additive interactions, we have to consider the contribution to the force on particle *i* due to all its neighbors. All individual particles have interactions with each other, However, once the particles get very close to each other the repulsive force increases rapidly. As mentioned, the potential energy that corresponds to these attractive and repulsive forces is the Lennard-Jones potential (Eq. (1)). The Lennerd-Jones potential is shown in Fig. (2).

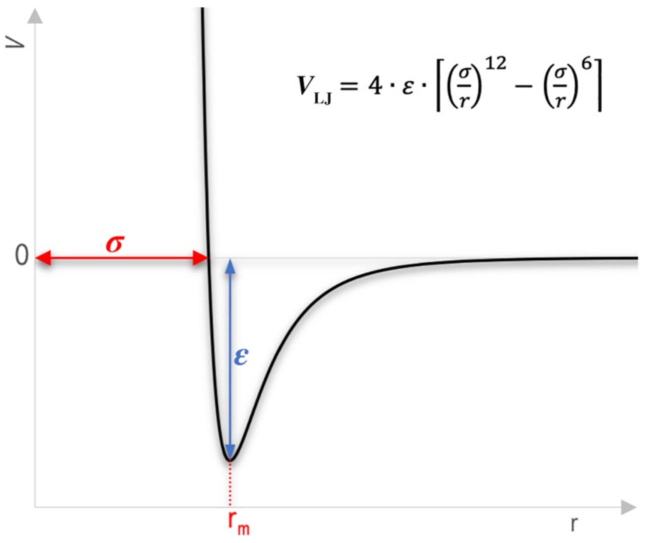


Fig. (2) Lennerd-Jones potential.

This potential has an attractive tail at large *r*, it reaches a minimum around 1.122, and it is strongly repulsive at shorter distance, passing through 0 at $r{=}r_m$ and increasing steeply as *r* is decreased further. For this reason, during initializing the position, we checked each two particles' distances and if the distance was not larger than r_m, we change the position of that particle.

2.3. Periodic Boundary Conditions

Often, the goal of atomic simulations is to measure bulk properties of a system. "Bulk" means that one wishes to describe the many-body interactions of an infinite system (the so-called thermodynamic limit). A particle in a bulk system should interact with an infinite volume of particles around it; it cannot be at a surface or boundary, where it will experience different interactions. Of course, we can simulate only a finite system on a computer, so we must make an approximation to the thermodynamic limit. Typically, the solution is to implement Periodic Boundary Conditions (PBC). This is similar to old video games where, when you come out of one side of the screen, you re-enter on the other side. So, after updating the velocities and forces, the positions were updated. Then, for each particle we apply PBC function. A brief procedure of updating atom positions and velocities is shown in Fig. (3).

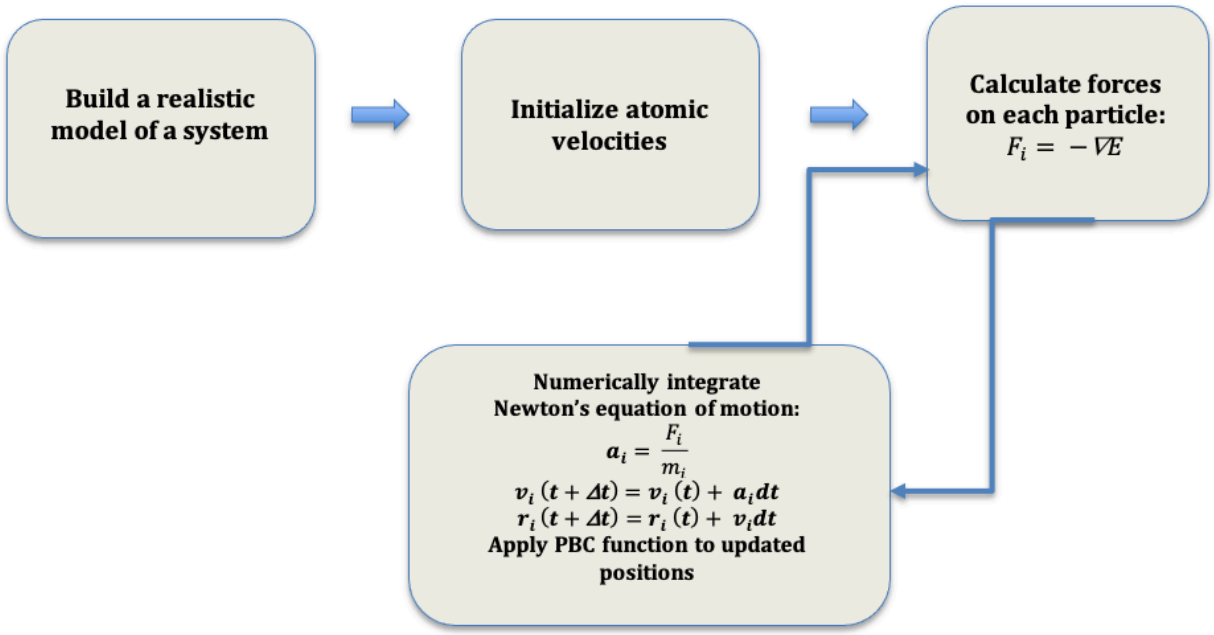

Build a realistic model of a system
Initialize atomic velocities
Calculate forces on each particle: $F_i = -\nabla E$
Numerically integrate Newton's equation of motion:
$a_i = \frac{F_i}{m_i}$
$v_i(t+\Delta t) = v_i(t) + a_i dt$
$r_i(t+\Delta t) = r_i(t) + v_i dt$
Apply PBC function to updated positions

Fig. (3) Procedures of a Molecular Dynamic simulation

3. Parallel Implementations:

The critical part of the code that parallelization can help the speed of running the simulation, is calculating forces on each particle. We used c++ programming language and "OpenMP" tools ("#pragma omp parallel for") in order to make force calculations and some other parts of the code (generate random positions, update velocities) faster.

OpenMP is specialized into parallelization of for loops. But a parallelization of some part of our code, like "calculation energy" and "kinetic energy" is tricky. There is a shared variable which is modified in every iteration. If we are not careful when parallelizing such for loops, we might introduce data races. Because multiple threads could try to update the shared variable (energy and ke) at the same time. OpenMP has the special reduction clause which can express the reduction of a for loop. So, for these two for loops in our code "#pragma omp parallel for reduce" was used to prevent any race condition.

4. Results:

In this section we compared the run time of serial code with parallel one, and investigate the effect of number of particles on the run time. We also obtained the optimum number of threads and calculate the speed up factor. Finally, we visualize the trajectory of particles during time.

Before comparing the speed of parallel code with serial code, we confirmed that the results of those codes are the same. We did it by setting constant values for initial positions and velocity instead of random values. We ran our codes in hpcc and used dev-intel18 development nodes (Two 2.4Ghz 20-core Intel Xeon Gold 6148 CPU (40 cores total) and 377GB of memory).

In order to see the effect of using "OpenMP" tools in parallel code, we run the serial version and parallel version with the same parameters. For these runs the number of particles was 100, length of box was 200 and time steps was 10000. We repeated each run with the same parameters 10 times and calculate the average run time to increase the accuracy of our data. Fig. 4. demonstrates the effect of increasing number of threads in run time.

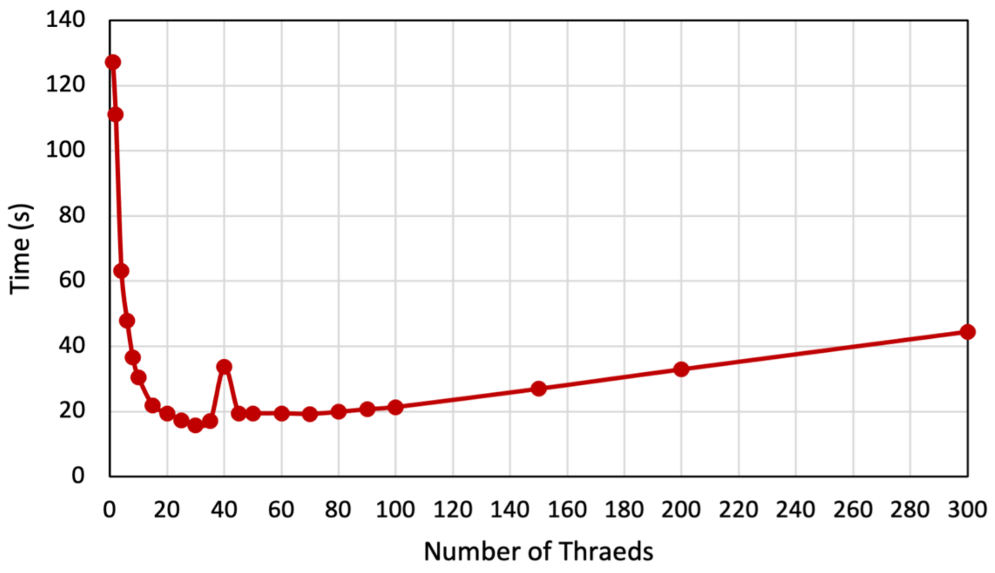


Fig. 4. The effect of using "OpenMP" tools on the run time for a constant set of parameters (Number of Atoms: 100, Length of Box:200, Time Steps: 10000)

Obviously, the parallel version is faster than serial one. From the results, it can be concluded that the optimum numbers of threads is 30. Although for the number of threads bigger that 30 the parallel version is still faster than serial one, the parallel code does not work efficiently after that. It is probably because of rising in the number of communications between threads. It is notable that theoretically, the optimal number of threads will be the number of cores you have on your machine. If your cores are "hyper threaded" it can run 2 threads on each core. Then, in that case, the optimal number of threads is double the number of cores on your machine. As it can be seen in Fig. 4., from 40 to 80 number of threads, the plot reaches equilibrium conditions and run time is approximately constant during these conditions. After the 80 number of threads (double of the number of the cores), we can see an increase in the run time. Additionally, when the number of thread is equal to the number of cores, we saw a sudden increase in run time.

Fig. 5. shows the effect of number of particles in run time of serial and parallel codes. For running the parallel code, we use the optimum number of threads (30).

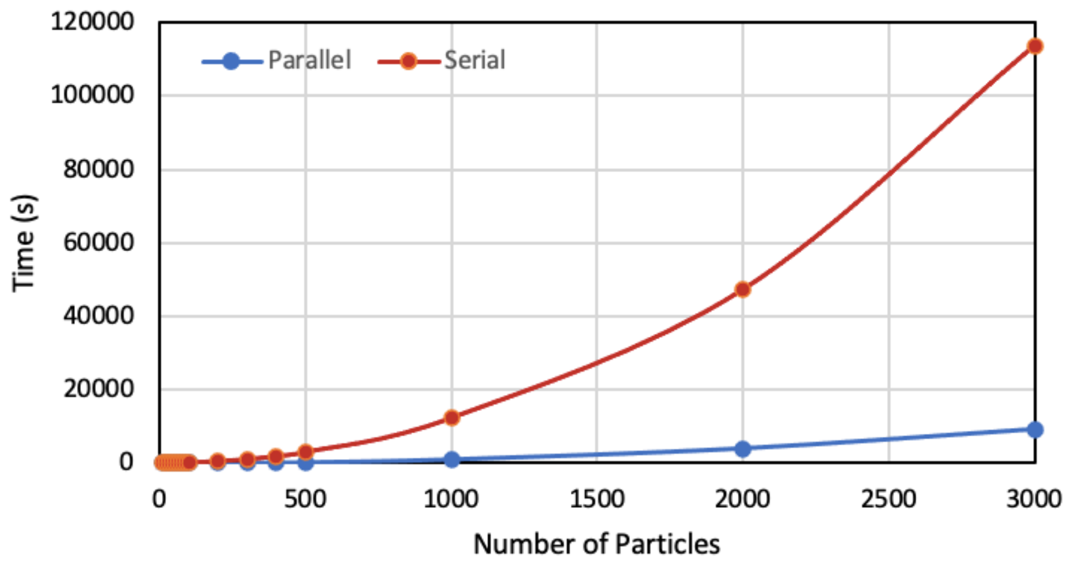


Fig. 5. The effect of increasing the number of particles in run time of serial and parallel code in time steps of 10000.

Finally, we visualized the motion of atoms during time via VMD software. Because we wanted to make the motion of particles more visible, we use 10 particles for this run. Fig. 6. shows four different positions of atoms during the time.

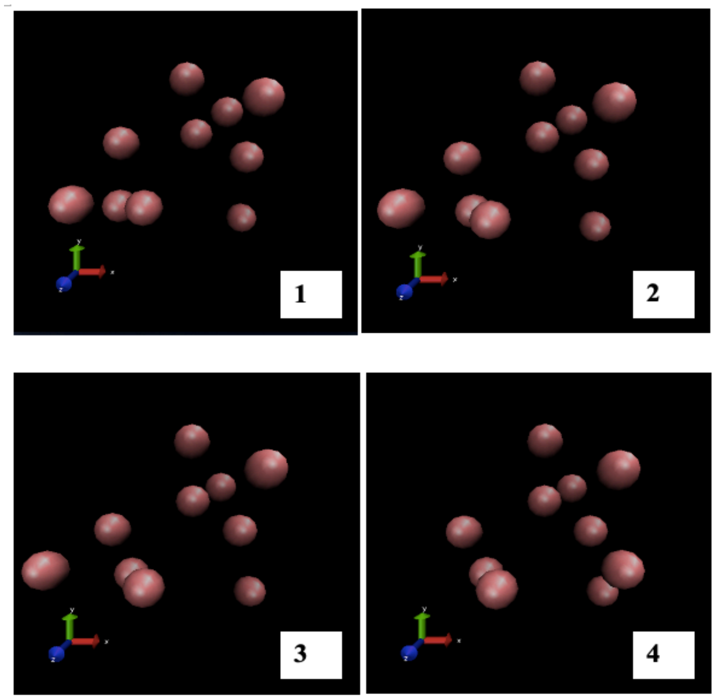


Fig. 6. The motion of particles during the time.

5 Discussion

The parallel code works well in terms of time, because the result showed that using "OpenMP" tools made it much faster than seral code, and the effect of this tool is more obvious when we ran the codes for larger number of particles. However, if I had more time, I would probably use "MPI" tools and also "Hybrid (MPI+OpenMP)" parallelization and compare these three different tools together.

6 References

- Eijkhout, Victor. "Introduction to high performance scientific computing", (2010).
-Nazanin Donyapour, Nicole M Roussey, and Alex Dickson. Revo: Resampling of ensembles by variation optimization. The Journal of Chemical Physics, 150(24):244112, 2019.
-Samuel D Lotz and Alex Dickson. Unbiased molecular dynamics of 11 min timescale drug unbinding reveals transition state stabilizing interactions. Journal of the American Chemical Society, 140(2):618–628, 2018.
-Andrew, R. Leach. "Molecular modeling principles and applications." Prentice Hall, London (2001).
- https://www.openmp.org/resources/refguides/

- https://en.wikipedia.org/wiki/Lennard-Jones_potential
- J Andrew McCammon, Bruce R Gelin, and Martin Karplus. Dynamics of folded proteins. Nature, 267(5612):585, 1977.
-https://coderedirect.com/questions/628182/openmp-set-number-of-threads-for-parallel-loop-depending-on-variable

7 Appendix

7.1. Serial Code

```
#include <iostream>
#include <vector>
#include <random>
#include <iomanip>
#include <cmath>
#include <math.h>
#include <string>
#include <sstream>
#include<fstream>
#include<chrono>

using namespace std;
using std::cout; using std::endl;
using std::vector;
using duration=std::chrono::seconds;

//units:
//time in ps
//distance in Angstroms
//energy in KJ/mol
//forces in (KJ/mol)/A
//mass in Daltons
//Temprature in Kelvin


// Function: compute shortest distant between two particles

vector<double> get_shortest_r (vector<double> pA, vector<double> pB, double l) {

    vector<double> d(1,3);

    for(int i=0;i<3;i++){

        d[i]=pA[i]-pB[i];

        if (d[i]<-l/2)
            d[i]+=l;

        else if(d[i]>l/2)
            d[i]-=l;
    }

    return d;
}

// Function: Apply Periodic Boundry Condition (PBC)

std::vector<std::vector<double>> apply_pbc (int N,std::vector<std::vector<double>> &position, double l,double
half_size_box){
    std::vector<std::vector<double>> pbc_position(N,std::vector<double>(3, 0));

    for (int i=0; i<N; i++){
        for(int j = 0; j < 3; ++j){
            pbc_position[i][j]=position[i][j];
        }
    }

    for (int i=0; i<N; i++){
        for(int j = 0; j < 3; ++j){
            if(pbc_position[i][j]>half_size_box)
                pbc_position[i][j]=pbc_position[i][j]-l;

            if(pbc_position[i][j]<= -half_size_box)
                pbc_position[i][j]=pbc_position[i][j]+l;

        }
    }

return pbc_position;
}



//Function: compute Energy

double Calculation_Energy(int N,double epsilon,std::vector<std::vector<double>> &position,double sigma,double
l){
    double energy=0;

    for (int i=0; i<N; i++){

        std::size_t wanted_row_A = i ; //extracting position of atom A
        std::vector<double> positionA = position[wanted_row_A]; // copy the position of atom A

        for(int j=0; j<i;j++) {
            std::size_t wanted_row_B = j ; //extracting positions of atom B
            std::vector<double> positionB = position[wanted_row_B] ; // // copy the position of atom B

            vector <double> R= get_shortest_r (positionA,positionB,l);
            double rlen = sqrt(R[0]*R[0] + R[1]*R[1] + R[2]*R[2]);

            energy +=4*epsilon*((pow((sigma/rlen),12))-(pow((sigma/rlen),6)));
        }
    }
    return energy;
}

//Function: compute forcess(take positions and output forces)


std::vector<std::vector<double>> Calculation_Forces (int N,double epsilon,double sigma,
std::vector<std::vector<double>> &position,double l){

    std::vector<std::vector<double>> new_force(N,std::vector<double>(3, 0));

    for (int i=0; i<N; i++){
        for(int j = 0; j < 3; ++j){
            new_force[i][j]=0;
        }
    }
    for (int i=0; i<N; i++){
        std::size_t wanted_row_A = i ; //extracting position of atom A
        std::vector<double> positionA = position[wanted_row_A] ; // copy the position of atom A

        for(int j=0; j<i;j++) {
            std::size_t wanted_row_B = j ; //extracting positions of atom B
            std::vector<double> positionB = position[wanted_row_B] ; // // copy the position of atom B
            vector <double> R= get_shortest_r (positionA,positionB,l);
            double rlen = sqrt(R[0]*R[0] + R[1]*R[1] + R[2]*R[2]);
            double sigma_over_r6= pow((sigma/rlen),6);
            double sigma_over_r12= pow((sigma/rlen),12);

            for(int k=0;k<3;k++){
                new_force [i][k]+=4*epsilon*((6*sigma_over_r6-12*sigma_over_r12)*R[k]/pow(rlen,2));
                new_force [j][k]-=4*epsilon*((6*sigma_over_r6-12*sigma_over_r12)*R[k]/pow(rlen,2));
            }
        }
    }
    return new_force;
}

int main(int argc, char *argv[]) {
    if (argc<4){
        cout<<"a.out Number of Particles, Box_Length,Time_Steps"<<endl;
        exit;
    }
```

```
int N=atoi(argv[1]);  //Number of particles
double l=atof(argv[2]); // Length of Box
int n_steps=atoi(argv[3]); //NUmber of Iterations

std::ofstream out;
out.open("result.txt");

double half_size_box=l/2;
double temp=50; //Kelvin
double dt=0.002;  //ps
double sigma=3.4; //Angstrom
double epsilon=0.238; //KJ/mol argon

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0,l);
std::vector<std::vector<double>> position(N,std::vector<double>(3, 0));
std::vector<std::vector<double>>mass(N,std::vector<double>(3,0));
std::vector<std::vector<double>>velocity(N,std::vector<double>(3,0));
std::vector<std::vector<double>>force(N,std::vector<double>(3,0));
std::vector<std::vector<double>>NEW_force(N,std::vector<double>(3,0));
//std::vector<std::vector<double>> topology(N*n_steps, std::vector<double>(3, 0));
//std::vector<std::vector<double>>v_t_half(N,std::vector<double>(3,0));
std::vector<std::vector<double>>vel_m_per_s(N,std::vector<double>(3,0));


//TIME START HERE
auto t1=std::chrono::steady_clock::now();


// generate random positions
for(int i = 0; i < N; ++i){ // loop over particles
   bool clashes = true;
   int n_attempts = 0;
   while (clashes) {
      n_attempts++;
      if (n_attempts > 1000) {
         cout<<"There is no hope"<<endl;
         exit;
      }
      for(int j = 0; j < 3; ++j) {
         position[i][j]=dis(gen);
      }
      clashes = false;
      if (i != 0) {
         for (int k = 0; k < i; ++k){ // loop over previously-defined particles
            // get distance between i and k
            vector<double> d = get_shortest_r(position[i],position[k], l);
```

```
                double d_len = sqrt( d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
                if (d_len < pow(2,(1/6))*sigma) {
                    clashes = true;
                    break;
                }
            }
        }
    }
}


//writing the initial positions into file
out<<N<<std::endl;
out<<"Initial Positions"<<endl;
for(auto row:position)
{
    out<<"Ar ";
    for(auto val:row)
    {
        out<<val<<" ";
    }
    out<<std::endl;
}

//generate masses
for(int i = 0; i < N; ++i){
    for(int j = 0; j < 3; ++j){
        mass[i][j]=39.9;
    }
}

//Avogadro constant 1/mol
double Na=6.02214076 * pow(10,23);

//Boltzman Constant Da. A^2 . ps-2. K-1
double K= 0.83139;

//Boltzman Constant in J. K-1
double Kb= 1.380649 * pow(10,-23);

//generate initial velocities via normally distributed random generator
std::mt19937 generator;
double mean = 0.0;
double stddev  = 1.0;
std::normal_distribution<double> normal(mean, stddev);

//velocity in Angstrom per pico_second
for(int i = 0; i < N; ++i){
    for(int j = 0; j < 3; ++j){
            double random=normal(generator);
            velocity [i][j]= sqrt (K*temp/mass[i][j]) * random; //calculating initial velocity using Boltzman Eq

    }
}

//iteration for number of time steps
for (int g=1;g<=n_steps;g++){


    //compute energy
    double energy=Calculation_Energy(N,epsilon,position,sigma,l);


    //compute forces
    force= Calculation_Forces (N,epsilon,sigma,position,l);

    //update the velocities and positions

    //(kJ/mol)/A = (k kg m2)/(A mol s2) = 1e2 (Da A/ps2)
    double factor = 100.0;

    for(int i = 0; i < N; ++i){
        for(int j = 0; j < 3; ++j){
            position[i][j]=position[i][j]+ velocity[i][j]*dt+0.5*force[i][j]*factor/mass[i][j]*pow(dt,2);
        }
    }

    //Apply pbc
    position=apply_pbc (N,position,l,half_size_box);

    NEW_force=Calculation_Forces (N,epsilon,sigma,position,l);

    for(int i = 0; i < N; ++i){
        for(int j = 0; j < 3; ++j){
            velocity[i][j]=velocity[i][j]+0.5*((force[i][j]+NEW_force[i][j])*factor/mass[i][j])*dt;
            force[i][j]=NEW_force[i][j];
        }
    }
    if (g % 100==0){
        //compute Kinetic Energy
        //velocity is in angstrom/ps
        double ke=0;
        double dalton_to_kg = 0.001;
        //angstrom_to_m = 1e-10
        //ps_to_s = 1e-12
        //angstrom_to_m/ps_to_s
        double conv_factor=100;
        for(int i = 0; i < N; ++i){
            for(int j = 0; j < 3; ++j){
                vel_m_per_s [i][j]= velocity[i][j]*conv_factor;
            }
        }

        //compute Kinetic Energy
        for (int i=0;i<N;i++){
            std::size_t wanted_row_V = i; //extracting velocity of atom i
            std::vector<double> vel_i = vel_m_per_s[wanted_row_V]; // copy the position of atom A
            double dot_product_V= inner_product(begin(vel_i),end(vel_i),begin(vel_i),0);
            ke += mass[i][0]* dalton_to_kg* dot_product_V;
        }

        // Kinetic Energy in KJ/mol
        double ke_kj_per_mol=0.5*ke/1000;


        //Boltzman Constant
        double K_KJ_per_mol_per_K=Kb*Na/1000;

        //Compute Temprature (Kelvin)
        double T= (2*ke_kj_per_mol)/(3*N*K_KJ_per_mol_per_K);

        //writing the positions on the text file

        //Printing the result
        cout<<"Iteration Number (" <<g<< "),Kinetic Energy: "<<ke_kj_per_mol<<" ,Temprature: "<<T<<endl;
        out<<N<<endl;
        out<<"Iteration Number = "<<g<<endl;
        for(auto row:position){
            out<<"Ar ";
            for(auto val:row){
                out<<val<<" ";
            }
            out<<std::endl;
        }
      }
    }

    //TIME FINISH HERE
    auto t2=std::chrono::steady_clock::now();

    //calculate the total time
    auto elapsed_total=std::chrono::duration_cast<duration>(t2-t1).count();

    cout<<"Time: "<<elapsed_total<<endl;

}
```

7.2. Parallel code:

```
#include <iostream>
#include <vector>
#include <random>
#include <iomanip>
#include <cmath>
#include <math.h>
#include <string>
#include <sstream>
#include<fstream>
#include<chrono>
#include <omp.h>

using namespace std;
using std::cout; using std::endl;
using std::vector;
using duration=std::chrono::microseconds;


//units:
//time in ps
//distance in Angstroms
//energy in KJ/mol
//forces in (KJ/mol)/A
//mass in Daltons
//Temprature in Kelvin


// Function: compute shortest distant between two particles

vector<double> get_shortest_r (vector<double> pA, vector<double> pB, double l) {

   vector<double> d(1,3);

   for(int i=0;i<3;i++){

      d[i]=pA[i]-pB[i];

      if (d[i]<-l/2)
         d[i]+=l;

      else if(d[i]>l/2)
         d[i]-=l;
   }

   return d;
```

```
}

// Function: Apply Periodic Boundry Condition (PBC)

std::vector<std::vector<double>> apply_pbc (int N,std::vector<std::vector<double>> &position, double l,double
half_size_box,int thread_cnt){
   std::vector<std::vector<double>> pbc_position(N,std::vector<double>(3, 0));

   //#pragma omp parallel for num_threads(thread_cnt)
   for (int i=0; i<N; i++){
      for(int j = 0; j < 3; ++j){
         pbc_position[i][j]=position[i][j];
      }
   }

   //#pragma omp parallel for num_threads(thread_cnt)
   for (int i=0; i<N; i++){
      for(int j = 0; j < 3; ++j){
         if(pbc_position[i][j]>half_size_box)
            pbc_position[i][j]=pbc_position[i][j]-l;

         if(pbc_position[i][j]<= -half_size_box)
            pbc_position[i][j]=pbc_position[i][j]+l;

      }
   }

return pbc_position;
}



//Function: compute Energy

double Calculation_Energy(int N,double epsilon,std::vector<std::vector<double>> &position,double sigma,double
l,int thread_cnt){
   double energy=0;
   #pragma omp parallel for num_threads(thread_cnt) reduction(+: energy)
   for (int i=0; i<N; i++){

      std::size_t wanted_row_A = i ; //extracting position of atom A
      std::vector<double> positionA = position[wanted_row_A]; // copy the position of atom A

      for(int j=0; j<i;j++) {
         std::size_t wanted_row_B = j ; //extracting positions of atom B
         std::vector<double> positionB = position[wanted_row_B] ; // // copy the position of atom B

         vector <double> R= get_shortest_r (positionA,positionB,l);
```

```
        double rlen = sqrt(R[0]*R[0] + R[1]*R[1] + R[2]*R[2]);

        energy +=4*epsilon*((pow((sigma/rlen),12))-(pow((sigma/rlen),6)));
      }
   }
   return energy;
}

//Function: compute forcess(take positions and output forces)


std::vector<std::vector<double>> Calculation_Forces (int N,double epsilon,double sigma,
std::vector<std::vector<double>> &position,double l,int thread_cnt){

   std::vector<std::vector<double>> new_force(N,std::vector<double>(3, 0));

  // #pragma omp parallel for num_threads(thread_cnt)
   for (int i=0; i<N; i++){
      for(int j = 0; j < 3; ++j){
         new_force[i][j]=0;
      }
   }

   #pragma omp parallel for num_threads(thread_cnt)
   for (int i=0; i<N; i++){
      std::size_t wanted_row_A = i ; //extracting position of atom A
      std::vector<double> positionA = position[wanted_row_A] ; // copy the position of atom A

      for(int j=0; j<i;j++) {
         std::size_t wanted_row_B = j ; //extracting positions of atom B
         std::vector<double> positionB = position[wanted_row_B] ; // // copy the position of atom B
         vector <double> R= get_shortest_r (positionA,positionB,l);
         double rlen = sqrt(R[0]*R[0] + R[1]*R[1] + R[2]*R[2]);
         double sigma_over_r6= pow((sigma/rlen),6);
         double sigma_over_r12= pow((sigma/rlen),12);

         for(int k=0;k<3;k++){
            new_force [i][k]+=4*epsilon*((6*sigma_over_r6-12*sigma_over_r12)*R[k]/pow(rlen,2));
            new_force [j][k]-=4*epsilon*((6*sigma_over_r6-12*sigma_over_r12)*R[k]/pow(rlen,2));
         }
      }
   }
   return new_force;
}



int main(int argc, char *argv[]) {
   if (argc<4){
```

```
   cout<<"a.out Number of Particles,Box_Length, Time_Steps, Number of Threads "<<endl;
   exit;
}

int N=atoi(argv[1]);  //Number of particles
double l=atof(argv[2]); // Length of Box
int n_steps=atoi(argv[3]); //NUmber of Iterations
int thread_cnt=atoi(argv[4]); //Number of Threads
double elapsed_time=0;

std::ofstream out;
out.open("result.txt");


double half_size_box=l/2;
double temp=50; //Kelvin
double dt=0.002;  //ps
double sigma=3.4; //Angstrom
double epsilon=0.238; //KJ/mol argon



std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0,l);
std::vector<std::vector<double>> position(N,std::vector<double>(3, 0));
std::vector<std::vector<double>>mass(N,std::vector<double>(3,0));
std::vector<std::vector<double>>velocity(N,std::vector<double>(3,0));
std::vector<std::vector<double>>force(N,std::vector<double>(3,0));
std::vector<std::vector<double>>NEW_force(N,std::vector<double>(3,0));
//std::vector<std::vector<double>> topology(N*n_steps, std::vector<double>(3, 0));
//std::vector<std::vector<double>>v_t_half(N,std::vector<double>(3,0));
std::vector<std::vector<double>>vel_m_per_s(N,std::vector<double>(3,0));


//TIME START HERE
double start_time=omp_get_wtime();


// generate random positions
#pragma omp parallel for num_threads(thread_cnt)
for(int i = 0; i < N; ++i){ // loop over particles
   bool clashes = true;
   int n_attempts = 0;
   while (clashes) {
      n_attempts++;
      if (n_attempts > 1000) {
         cout<<"There is no hope"<<endl;
         exit;
```

```
            }
          // #pragma omp parallel for num_threads(thread_cnt)
            for(int j = 0; j < 3; ++j) {
                position[i][j]=dis(gen);
            }
            clashes = false;
            if (i != 0) {
                for (int k = 0; k < i; ++k){ // loop over previously-defined particles
                    // get distance between i and k
                    vector<double> d = get_shortest_r(position[i],position[k], l);

                    double d_len = sqrt( d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
                    if (d_len < pow(2,(1/6))*sigma) {
                        clashes = true;
                        break;
                    }
                }
            }
        }
    }


    //writing the initial positions into file
    out<<N<<std::endl;
    out<<"Initial Positions"<<endl;
    for(auto row:position)
    {
        out<<"Ar ";
        for(auto val:row)
        {
            out<<val<<" ";
        }
        out<<std::endl;
    }


    //generate masses
    for(int i = 0; i < N; ++i){
        //#pragma omp parallel for num_threads(thread_cnt)
        for(int j = 0; j < 3; ++j){
            mass[i][j]=39.9;
        }
    }



    //Avogadro constant 1/mol
    double Na=6.02214076 * pow(10,23);
```

```
//Boltzman Constant Da. A^2 . ps-2. K-1
double K= 0.83139;

//Boltzman Constant in J. K-1
double Kb= 1.380649 * pow(10,-23);

//generate initial velocities via normally distributed random generator
std::mt19937 generator;
double mean = 0.0;
double stddev  = 1.0;
std::normal_distribution<double> normal(mean, stddev);

//velocity in Angstrom per pico_second
for(int i = 0; i < N; ++i){
   for(int j = 0; j < 3; ++j){
         double random=normal(generator);
         velocity [i][j]= sqrt (K*temp/mass[i][j]) * random; //calculating initial velocity using Boltzman Eq

   }
}
auto t1=std::chrono::steady_clock::now();
//iteration for number of time steps
for (int g=1;g<=n_steps;g++){


   //compute energy
   double energy=Calculation_Energy(N,epsilon,position,sigma,l,thread_cnt);


   //compute forces
   force= Calculation_Forces (N,epsilon,sigma,position,l,thread_cnt);

   //update the velocities and positions

   //(kJ/mol)/A = (k kg m2)/(A mol s2) = 1e2 (Da A/ps2)
   double factor = 100.0;

   for(int i = 0; i < N; ++i){
      for(int j = 0; j < 3; ++j){
           position[i][j]=position[i][j]+ velocity[i][j]*dt+0.5*force[i][j]*factor/mass[i][j]*pow(dt,2);
      }
   }

   //Apply pbc
   position=apply_pbc (N,position,l,half_size_box,thread_cnt);

   NEW_force=Calculation_Forces (N,epsilon,sigma,position,l,thread_cnt);

   #pragma omp parallel for num_threads(thread_cnt)
```

```
for(int i = 0; i < N; ++i){
   for(int j = 0; j < 3; ++j){
         velocity[i][j]=velocity[i][j]+0.5*((force[i][j]+NEW_force[i][j])*factor/mass[i][j])*dt;
         force[i][j]=NEW_force[i][j];
   }
}

if (g % 100==0){

   //compute Kinetic Energy
   //velocity is in angstrom/ps
   double ke=0;
   double dalton_to_kg = 0.001;
   //angstrom_to_m = 1e-10
   //ps_to_s = 1e-12
   //angstrom_to_m/ps_to_s
   double conv_factor=100;
   #pragma omp parallel for num_threads(thread_cnt)
   for(int i = 0; i < N; ++i){
      for(int j = 0; j < 3; ++j){
            vel_m_per_s [i][j]= velocity[i][j]*conv_factor;
      }
   }

   //compute Kinetic Energy
   #pragma omp parallel for num_threads(thread_cnt) reduction(+: ke)
   for (int i=0;i<N;i++){
      std::size_t wanted_row_V = i; //extracting velocity of atom i
      std::vector<double> vel_i = vel_m_per_s[wanted_row_V]; // copy the position of atom A
      double dot_product_V= inner_product(begin(vel_i),end(vel_i),begin(vel_i),0);
      ke += mass[i][0]* dalton_to_kg* dot_product_V;
   }

   // Kinetic Energy in KJ/mol
   double ke_kj_per_mol=0.5*ke/1000;


   //Boltzman Constant
   double K_KJ_per_mol_per_K=Kb*Na/1000;

   //Compute Temprature (Kelvin)
   double T= (2*ke_kj_per_mol)/(3*N*K_KJ_per_mol_per_K);

   //writing the positions on the text file

   //Printing the result
   cout<<"Iteration Number (" <<g<< "),Kinetic Energy: "<<ke_kj_per_mol<<" ,Temprature: "<<T<<endl;
   out<<N<<endl;
   out<<"Iteration Number = "<<g<<endl;
```

```
            for(auto row:position){
                out<<"Ar ";
                for(auto val:row){
                    out<<val<<" ";
                }
                out<<std::endl;
            }
        }


    }

    //TIME FINISH HERE
    double end_time = omp_get_wtime();
    elapsed_time = end_time-start_time;
    std::cout<<"Elapsed time: "<<elapsed_time<<std::endl;
}
```