Langevin Dynamics, Part II

HPC Assignment 3 - March, 2017

Kamyar Nazeri
Student ID: 100633486

Introduction

In this assignment, we are simulating the Langevin dynamics, making the code that utilizes interactions thread-parallel and analysing the output data.
These specifications have been incorporated into our simulation model:

- Particle trajectories are implemented in 2D.
- Equation is equipped with periodic boundary conditions.
- OpenMP [1] library is used to execute the code in parallel.
- Lennard-Jones potential model [2] is used to approximate the interaction between a pair of particles.

C++ Data Structure

The C++ code base for this assignment contains data-structures that represent Particle, Sector and Neighbor entities. Program procedures like ordering particles, assigning particles to a sector, and iterating over sectors are performed using arrays of aforementioned data-structures. The class diagram of the data-structures used in our program is shown in *Figure 1*:

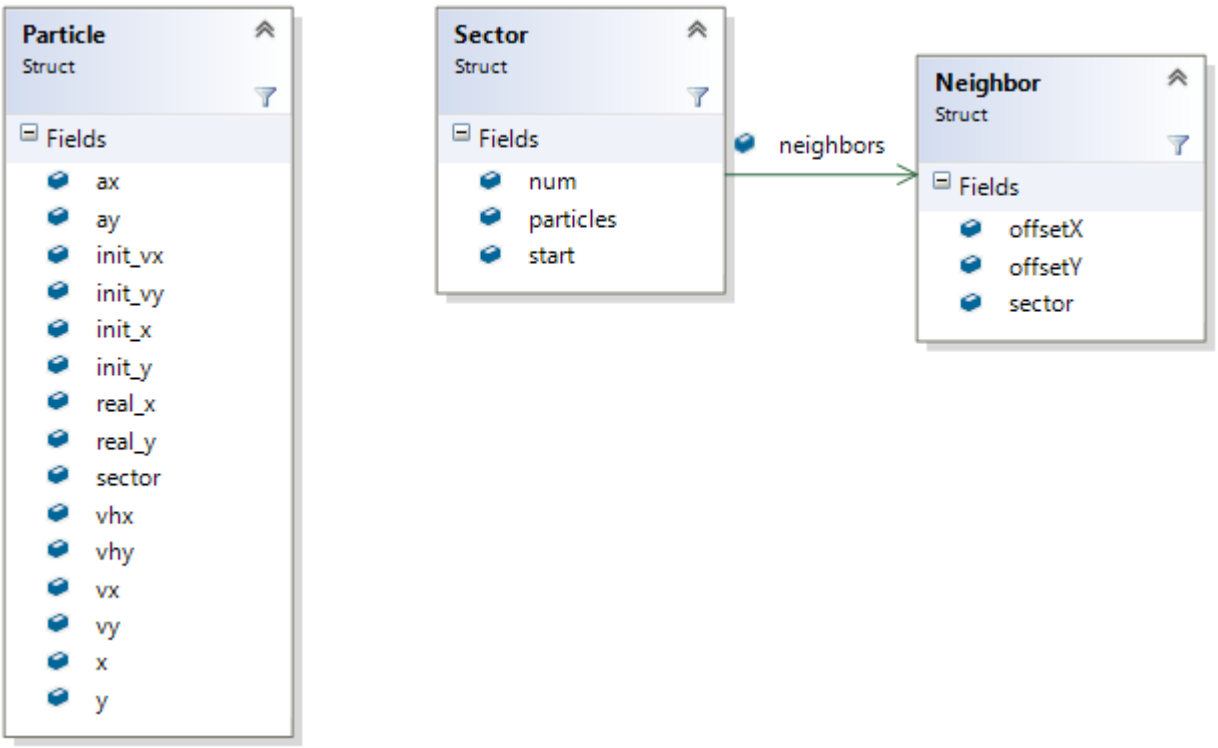

Figure 1: Class diagram: Particle, Sector, Neighbor

- *Particle* struct contains the values for initial position, real position, position, initial velocity, velocity, velocity verlet, acceleration and the corresponding sector.
- *Sector* struct contains the number of the sector in typewriter index, starting index of the particles, number of the particles and an array of its *Neighbors*
- *Neighbor* struct contains the index of a sector and its offset with the declaring sector when considering boundary condition.

Pseudo-Code

- **Pseudo-Code for the main loop (over time-steps):**

initialize time to zero: $time = 0$

WHILE time < total time
  FOR all particles → **p**
    calculate particle's velocity verlet: $vh = velocity + \frac{1}{2}(acceleration \times \Delta t)$
    calculate particle's position: $position = position + vh \times \Delta t$
  ENDFOR

  CALL *impose-boundary-condition* function passing array of particles
  CALL *order* function to order the particles positions by sector, passing particles and sectors
  CALL *random-number-generator* function to create a normally distributed random force: $\rightarrow \eta$
  remove number of particles in the dumpster from the actual particles count

  FOR all particles → **p**
    reset particle's acceleration: $acceleration = 0$
    calculate particle's acceleration using the random force: $acceleration = \frac{1}{m}(-\gamma v + \frac{\sqrt{2\gamma k_B T}}{\sqrt{\Delta t}}\eta)$
  ENDFOR

  FOR all sectors → **sec**
    FOR all particles in the sector *sec* → **p**
      FOR all neighbors of the sector *sec* → **nbr**
        FOR all particles in the neighbor *nbr* → **np**
          calculate the distance between *p* and *np* + offset of the neighboring sector → **dist**
          IF dist < range of interaction
            calculate the force using Lennard-Jones potential [2] → **F**
            calculate the acceleration using the interaction force: $acceleration = \frac{acceleration \times F}{m}$
          ENDIF
        ENDFOR
      ENDFOR
    ENDFOR
  ENDFOR

  FOR all particles → **p**
    calculate the velocity: $velocity = vh + \frac{1}{2}(acceleration \times \Delta t)$
  ENDFOR

  update time iteration: $time = time + \Delta t$
ENDWHILE

- **Pseudo-Code for neighbor-list subroutine:**

  LET **d** = linear number of sectors, **L** = domain size
  FOR all sectors → **sec**
    get sector's typewriter index → **index**
    compute the chess-board indices from the sector's typewriter index → **row, col**
    FOR $nrow = row - 1$ to $row + 1$
      FOR $ncol = col - 1$ to $col + 1$
        calculate the neighbor sector's offset when imposing boundary condition:
          left & top: -L, right & bottom: L
          $offsetX = (round(\frac{ncol+d}{d}) - 1) \times L$
          $offsetY = (round(\frac{nrow+d}{d}) - 1) \times L$
        calculate the neighboring sector index from the chessboard indices:
          $neighboring\ sector\ index = (nrow \times d) + ncol$
      ENDFOR
    ENDFOR
  ENDFOR

- **Pseudo-Code for order and assign subroutine:**

  LET **d** = linear number of sectors, **L** = domain size, **width** = $\frac{L}{d}$
  LET **particles** = array of all particles
  FOR all particles → **p**
    IF particle's position is in the domain: $p.x < \frac{L}{2}$ and $p.y < \frac{L}{2}$
      assign a sector index to the particle: $p.sector = \frac{p.y + \frac{L}{2}}{width} \times d + \frac{p.x + \frac{L}{2}}{width}$
      keep count of all particles in the sector: $sector.particles = sector.particles + 1$
    ELSE
      put the particle outside bounds in dumpster: $p.sector = d \times d$
      keep count of all particles in the dumpster: $dumpster.particles = dumpster.particles + 1$
    ENDIF
  ENDFOR

  LET $sum = 0$
  FOR all sectors → **sec**
    set sector's start index to sum of all particles in previous sectors: $sec.start = sum$
    calculate sum of all previous particles: $sum = sum + sec.particles$
    reset number of particles in the sector prior ordering: $sec.particles = 0$
  ENDFOR

  LET $copy = particles$
  FOR all particles in ***copy*** → **p**
    find p's sector → **sec**
    reorder the array of all particles by placing all particles with the same sector together:
      $particles[sec.start + sec.particles] = p$
    keep count of all particles in the sector: $sec.particles = sec.particles + 1$
  ENDFOR

Particles Initialization

When taking into account the interaction force between particles, the initial positions for the particles must not lead to excessive interaction forces. This requirement must hold even if the density of the particles is large.
In this assignment, we implement the initialization by drawing a mesh from all possible positions the particles can take on in the domain without interaction. We define an array of all those positions and use the *Fisher–Yates algorithm* [3] to shuffle this array. To initialize n particles, we draw the first n elements from the shuffled array. *Figure 2* represents this implementation in 3 stages. From left to right: draw a mesh, shuffle the array, take first n elements.

Figure 2: Random initialization of 7 particles in a domain than contains up to 25 particles

Finding best time step

In our simulation, values of variables are changed at distinct, separate "points in time", meaning time is viewed as a discrete variable. Thus, a non-time variable (position) jumps from one value to another as time moves from one time period to the next. This could lead to an increase in the number of particles escaping the domain due to the repelling force between the particles stepping onto each other. To fix this issue, one can set the time-step small enough so that no particle could jump more than the "interaction radius" in one time step. Using the velocity-verlet to calculate particle's change in position, we can find the minimum value for dt:

$$\frac{1}{2}\frac{F}{m}dt^2 < rc \quad \Rightarrow \quad dt < \sqrt{\frac{2m\ rc}{F}}$$

Where rc is the interaction radius, dt is the time-step, m is the particle's mass and F is a random force in the original Langevin equation. The following table shows the number the particles stepping onto each other (lost from the domain) for different values of time-step. As the time-step becomes smaller, fewer particles step onto each other. (from the equation above, for $m = 1$ and $sigma = 0.1$ we expect dt to be less than 0.03)

Particles(n): 40,000 Domain Size(L): 100 Range of Interaction(sigma): 0.1

Time Step	Iterations	Interactions	Lost Particles / Total Particles
1 s	100	43676	19200 / 40000
0.1 s	100	98164	6393 / 40000
0.01 s	100	91054	754 / 40000
0.001 s	100	84520	0 / 40000

- time-step = 1 seconds, 19,200 particles escape the domain out of 43,676 interactions.
- time-step = 0.001 seconds, no particles escape the domain out of 84,520 interactions.

Root-Mean-Square Displacement

To calculate the right root-mean-square displacement, we measure the deviation time between the position of a particle and its initial position:

$$MSD \equiv \frac{1}{N}\sum_{n=0}^{N}(x_n(t) - x_n(0))^2 \quad \Rightarrow \quad RMS \equiv \sqrt{MSD}$$

To measure the *MSD* we have to keep track of each particle's initial position. This means in the *Particle* data structure, we need to save the *initial* position of each particle. Also, to have a continuous displacement of a particle when it moves across the periodic boundaries, we need to save the *real* position of the particles (without imposing the boundary conditions) in the *Particle* data structure. The MSD is measured using the *real* and *initial* values. At each iteration, particles that leave the domain are put in the dumpster and removed from the calculation.
We are experiencing a spike in the mean-square displacement when the number of particles is relatively small. The spike happens almost when the particles start leaving the domain and they are put back in due to the boundary condition effect. The domain size in *figure 4* is twice the size of the domain in *figure 3*. Notice the spike in the MSD is happening almost twice as late. This is probably the result of the interaction force between the particles already inside the domain and those that are put back in due to imposing the boundary condition.

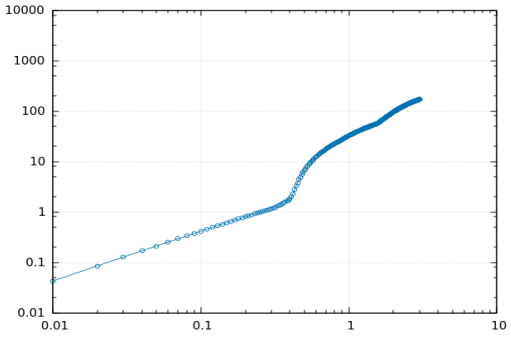

Figure 3: Particles: 1,000
domain size: 50

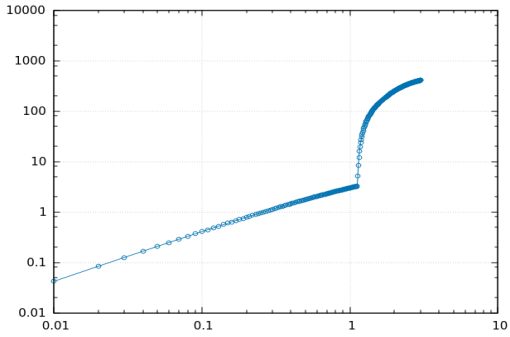

Figure 4: Particles: 1,000
domain size = 100

On the other hand, when the domain size is so large (*figure 5*) that no particle can leave the domain, or when there is a relatively large number of particles in the domain (*figure 6*) so that many particles experience the interaction force at each time-step, the ballistic-diffusive regimes are similar to the one without the interaction force and there's no spike in the MSD:

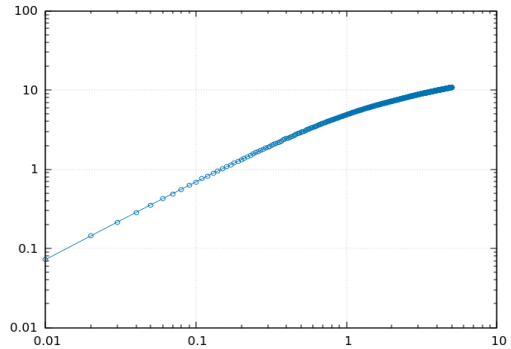

Figure 5: particles: 1,000
domain size: 1,000

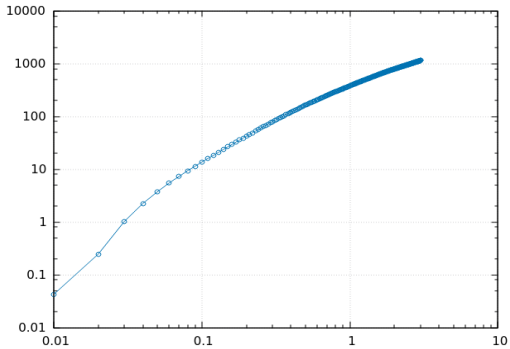

Figure 6: particles = 40,000
domain size: 100

Parallelization & Domain Composition

We use OpenMP [1] library to make the code thread-parallel. The pragma *omp parallel for* is used to fork additional threads to carry out the work for computation of the interactions. In addition, to increase the program performance, we use *domain composition* to assign the particles to sectors based on their positions. Each sector in the domain composition has 8 neighboring sectors. To compute the interaction force between particles, every particle in each sector will be compared with only the particles in the neighboring sectors. The following table shows the effect of parallelism and domain composition on the wall-time:

Particles(n): 40,000 Domain Size(L): 100 Range of Interaction(sigma): 0.1

Linear Number of Sectors	Wall-Time (seconds)	Parallel Wall-Time (seconds)
3	1008	526
5	205	132
10	51.2	34.4
20	13.2	12.5
100	4.2	6.8
200	4	6.6
400	4.3	6.8
800	5.7	8.1

Efficiency of the domain composition:

The minimum value for the linear number of sectors is 3 (i.e. 9 sectors), hence to simulate the serial, single domain implementation, one can choose the linear number of sectors to be 3. This means that each sector, together with its neighboring sectors, covers the whole domain, and, as a result, every particle is compared with all the particles in the domain. This leads to the worst program performance, and the biggest wall-time in our simulations: **1008 seconds**.
By increasing the number of sectors, the amount of wall-time decreases by a factor of $\frac{1}{d^2}$, with the best performance occurring around $\frac{n}{d^2} \approx 1$ which, for uniformly distributed particles, means there's almost one particle in each sector. In our simulations, this happens with $d = 200$, which results in a wall-time of **4 seconds**. Greater values for d mean that there are more sectors in the domain than the number of particles and would have a negative impact on the overall performance of the program.

Efficiency of the parallelization:

The efficiency of the parallelism depends on two variables: (1) the total number of the cpu cores available (two cores in our test machine), and (2) the number of flops for a single iteration of all sectors ($9 \times \frac{n^2}{d^4}$). As demonstrated in the table above, parallelism shows its benefits only when the number of flops gets to the millions (e.g. $\approx$ *23,000,000 flops* with $d = 5$) and it starts to lose efficiency, with a decline in the number of flops.
We cannot implement the optimum domain composition and parallelism at the same time since they are mutually exclusive; and with only 2 physical cores in our test machine, thread-parallelism would not be desirable and domain composition remains the best option.

Other time consuming tasks:

Besides "computation of the interactions", the only time consuming part of the program is writing data on the disk. Every other part of the program including ordering, imposing boundary conditions, calculating velocity and generating random numbers is of the order *O(n)* and relatively fast. Our tests indicate that the amount of time required to write data on disk for 40,000 particles is almost 3.8 seconds, while the times required for ordering, imposing boundary condition, generating random numbers and calculating particle velocities are 0.41 seconds, 0.03 seconds, 0.03 seconds and 0.08 seconds respectively. This means that writing is the performance bottleneck, and even by exploiting parallel algorithms for these tasks, one cannot expect much improvement in the overall value of the wall-time.
To make parts of the code run in parallel, we make use of the pragma *omp parallel sections* to divide the work among separate threads. The sections we define to be run in parallel are: calculating the velocities, writing particle positions on disk, and writing MSD on the disk. The performance gain using parallel section in our tests with 40,000 particles is merely 0.4 seconds:

```
#pragma omp sections
{
    #pragma omp section
    {
        // calculate the velocities
    }

    #pragma omp section
    {
        // write particles positions on disk: trajectories
    }

    #pragma omp section
    {
        // write mean-square displacement on disk: means
    }
}
```

Influence of the interactions on the MSD:

The influence of the interactions on the mean-square displacement is unclear with our simulations. As demonstrated in *Figure 6*, when there is relatively a large number of particles in the domain so that many particles experience the interaction force at each time-step, the ballistic-diffusive regimes are similar to the one without the interaction force.
However, the influence of the interactions on the velocities of the particles is distinct, as more interactions between particles means an increase in the velocity of the particles. The distribution of the velocities of particles is demonstrated in the following figures. Note that the distribution of the velocities becomes more spread out (*Figure 8*) when we incorporate the interaction force into the equation:

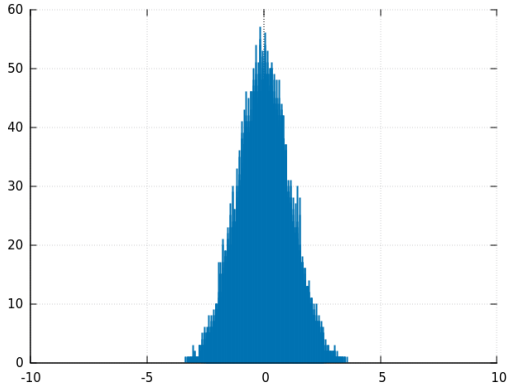

Figure 7: Velocity distribution no interaction

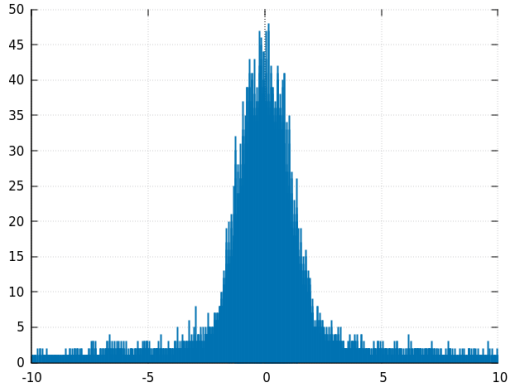

Figure 8: Velocity distribution with interaction

References

[1] The OpenMP API specification for parallel programming,
http://www.openmp.org/resources/openmp-compilers/

[2] Lennard-Jones potential,
https://en.wikipedia.org/wiki/Lennard-Jones_potential

[3] Fisher–Yates shuffle algorithm,
https://en.wikipedia.org/wiki/Fisher-Yates_shuffle