Accelerate Molecular Dynamics Simulations

Hao Yu, Tianyi Xu, Hanyu Wang

Abstract

Molecular dynamics(MD) simulations numerically solve Newton's equation of motion for a specific interatomic potential of an interacting particle system, and predict the time evolution of the system. This technique has applications in studying the physical and chemical properties of systems with large numbers of particles. In this project, we focus on reducing the computational time of 2-dimensional MD simulations. First, we capture the short range interactions by building a neighborhood table for each particle; second, we use OpenACC and OpenMP to parallel the MD code to save computation time. We will discuss the performance difference based on runtime results. At last, we generate the animation to visualize the trajectories of particles in the 2-dimensional plane.

Molecular Dynamics Simulations

Newton's equations of motion for particles is the classical way to simulate molecular dynamics.

$$m\frac{d^2 r_i}{dt^2} = F_i(r_1, r_2, r_3, \ldots r_n),\ i = 1, 2, 3, \ldots, N. \qquad \text{eq.1}$$

In equation 1, r_i refers to position vectors, and F represents the forces acting on the N particles in the system. Force in equation 1 is often generated from potential functions which are formulated as a sum over interactions between the particles of the system. The simplest representation of potential is the "pair potential", in which the total potential energy can be calculated from the sum of energy contributions between pairs of atoms. Lennard-Jones potential is an example of pair potential:

$$V(r) = 4\varepsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \qquad \text{eq. 2}$$

In the Lennard-Jones potential equation, r is the distance between two interacting particles, ε is the strength of the interaction, and σ is the distance at which the potential energy V is zero. Figure 1 shows the relationship between V and r.

The two equations above are essential to understand and simulate molecular dynamics. Once we are done calculating the interactions between each particles at different time steps, we are able to simulate the particles' movements.

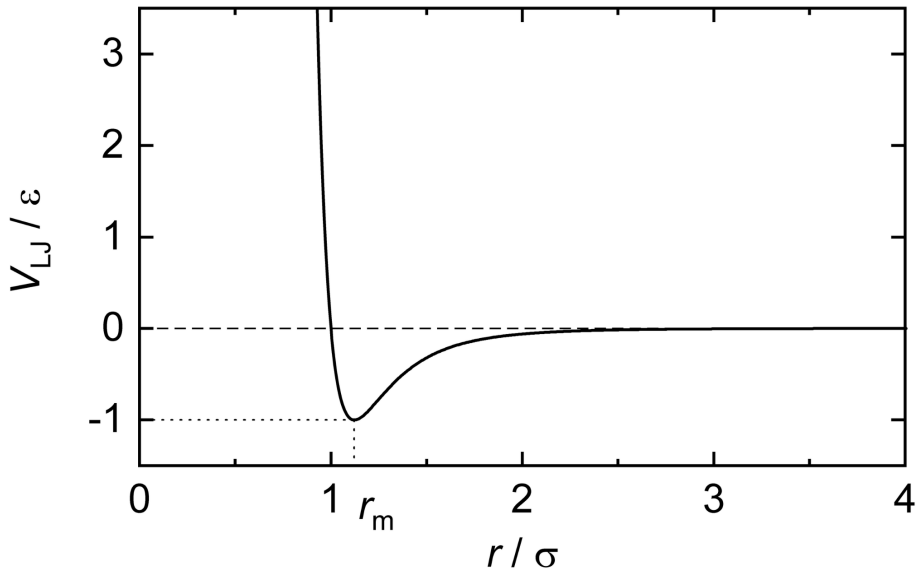

Figure 1. Graph of the Lennard-Jones potential function: Intermolecular potential energy V_{LJ} as a function of the distance of a pair of particles. The potential minimum is at $r = r_m = 2^{1/6}\sigma$.

Neighborhood table

Neighborhood table is a data structure technique which could facilitate the reduction of iterations in molecular dynamics simulation. In the conventional version of molecular dynamics simulation, the distance between each pair of two molecules in the whole space is re-calculated in every simulated time interval (or iterations in a program perspective). For example, let's focus on molecule A, the distance between A and all the other molecules are calculated, and compared with a certain distance R_{cut} , which is the threshold distance we regard as effective for the interatomic potential. For distance larger than R_{cut}, the interatomic effect is small enough to be neglected. Additionally, by comparing the radius of the space where all the molecules stand in and this threshold distance, we observe that for an evenly distributed start condition, the possibility of molecules moving into this threshold distance is very low. In other words, only one small portion of the total molecules are regarded as the effective neighbors of one molecule, in one simulation interval. In the program perspective, the majority of distance computation, which scales $O(N^2)$ where N is the total number of molecules in the space, can be saved if one structure could maintain the data of effective molecules. Neighborhood table is utilizing this property, which maintains all the molecules within a certain distance as one molecule's neighbor. All the molecules would have their own neighborhood, and all together form the neighborhood table. During the simulation, this neighbor relationship would be updated since the molecules are moving due to interatomic forces. Hence the neighborhood table will be updated periodically in a fixed interval. To prolong this update interval, we add another distance

called R_{skin}. By constructing the neighborhood with regard to this R_{skin}, the area between R_{skin} and R_{cut} would form a buffer, as implied in Figure 2.

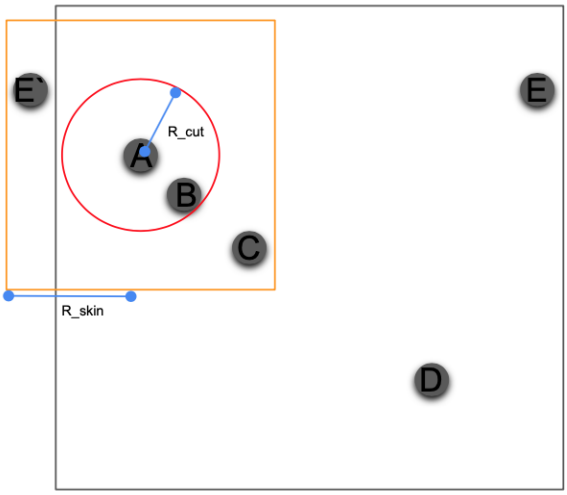

Figure 2. Illustration of neighborhood table referencing for 2D molecule simulation (all letter-indexed dots are particles, orange square represents the R_skin and red circle represents the R_cut)

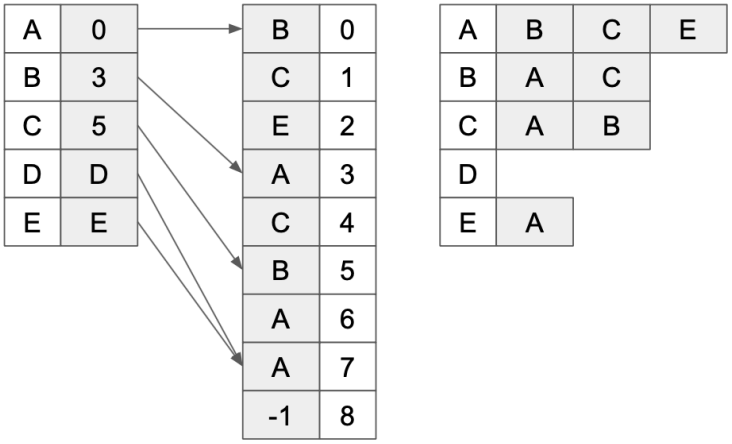

Figure 3. Illustration of two versions of neighborhood table

In our project, two different versions of the neighborhood table have been implemented. One is called the array version, and the other is called the vector version. In Figure 3, we constructed these two neighborhood tables based on the particle relationships in Figure 2.
In the array version, neighbors of all particles are maintained in the same array; the content of the left-hand side array represents the start index of its neighborhood in the right-hand side neighborhood table. In the vector version, each particle owns one vector which contains its neighborhood members.

Parallel Methods

OpenACC

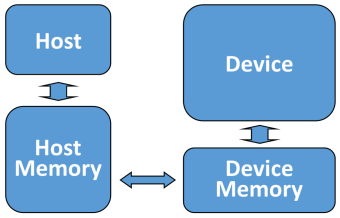

Figure 4. OpenACC's Abstract Accelerate Model

OpenACC is a programming model that uses high-level compiler directives to expose parallelism in the code and parallelizing compilers to build the code for accelerators like GPU or other many-core architectures. OpenACC supports offloading both computation and data from a host device to an accelerator device. In many cases, the host is CPU and the device is GPU, so it is necessary for the compiler to manage data movement efficiency between the two memories and to ensure the correct results. Huge performance improvements will be gained after optimizing data movement.

In our work, we optimized the basic version of molecular dynamics code. First, allocate memory on GPU and copy arrays that store acceleration, velocity and position of all particles from host to GPU when entering region and copy the data to the host when exit region. The inner loop is parallelizable and we avoid relocating arrays every iteration from the previous step.

```
t=0
while t<T_end
        #pragma acc data copy(a,v,r)
        #pragma acc parallel loop
        for i=1 to N do
                F_i←0
                #pragma acc loop reduction(+:V)
                for j = 1 to N do
                        r_i,j ← r_j−r_i
                        V(||r_i,j||)
                        F_i←−∂V(||r_i,j||)/∂r_i
                        a_i=F_i/m_i
                        v_i(t+dt)=v_i(t)+a_i dt
                        r_i(t+dt)=r_i(t)+v_i(t)dt+½*a_i(dt)^2
                end for
        end for
        t ← t + dt
```

Figure 5. MD Basic and OpenACC derivatives pseudocode

Results

Basic MD and Neighborhood Table

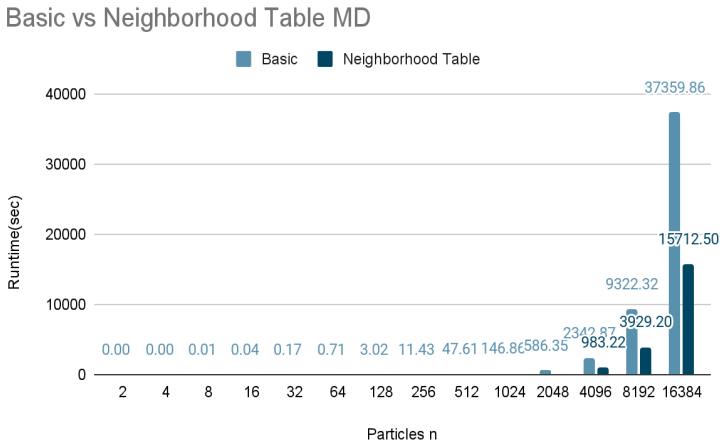

Figure 6. Baseline Runtime of MD basic and MD with neighborhood table in a L×L grid(L=4096), time step is 0.005, total iteration is 5000. All particles have mass 1.

We run the runtime baseline with 2 to 2^{14} or 16384 particles in the grid, and the results are shown in Figure 6. Considering the short range focus of Lennard-Jones potential normally used in MD, the neighborhood table reduces the number of particles needed to compute force in the inner loop and reduces a portion of the computational time.

OpenACC

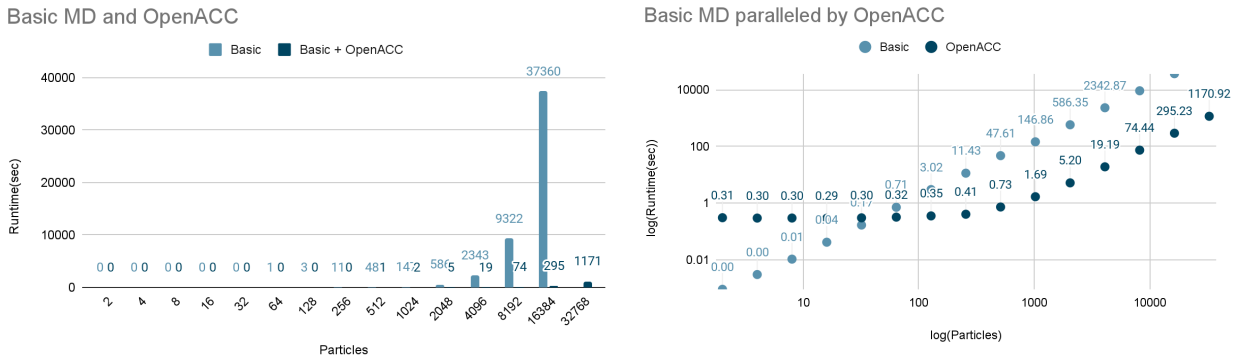

Figure 7. OpenACC accelerated MD basic results, original runtime (right), log scale runtime(left)
We use OpenACC to parallel compute the same tests as the baseline and achieve significant computational efficiency improvements. Before parallel computation, it takes about 37,000 seconds to simulate 16384 particles, which is about 10 hours. Since the complexity of MD is

$O(N^2)$, it is expected to take around 40 hours to finish the simulation for 32768 particles in the grid. In contrast, after paralleling OpenACC, the later computation only takes 1171 seconds. This is a significant improvement compared to the neighborhood table, since it utilizes the multi-processing architecture of GPU.

OpenMP

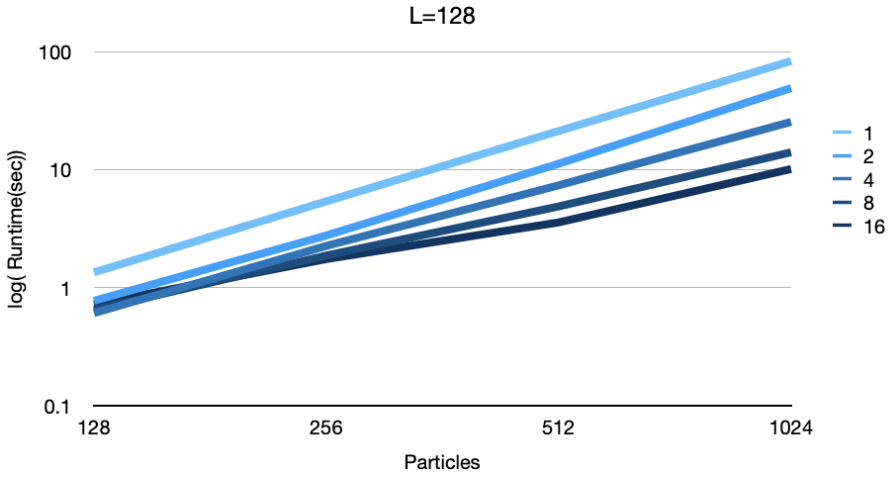

Figure 8 OpenMP result for small space (128), and small number of particles(128-1024)

We observe that for a small number of particles, the overhead of the OpenMP thread construction process is more profound, especially when the particle number is less than 512. The runtime of these multi-thread programs do not scale linearly to the number of threads in this section.

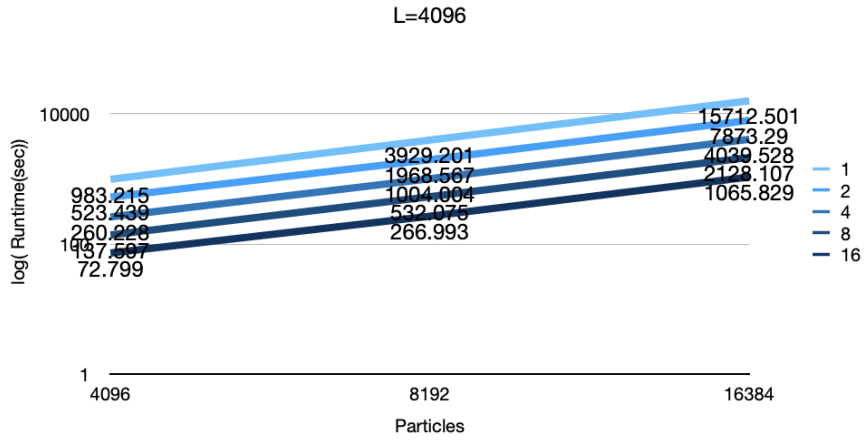

Figure 9 OpenMP result for large space (4096), and large number of particles(4096-16384)

As the number of particles increases, the efficiency of the multi-thread version of our MD simulation increases. We could observe a linear performance gain as the number of threads

increases. This implies that the underlying program execution structure is well balanced for multi-thread implementation.

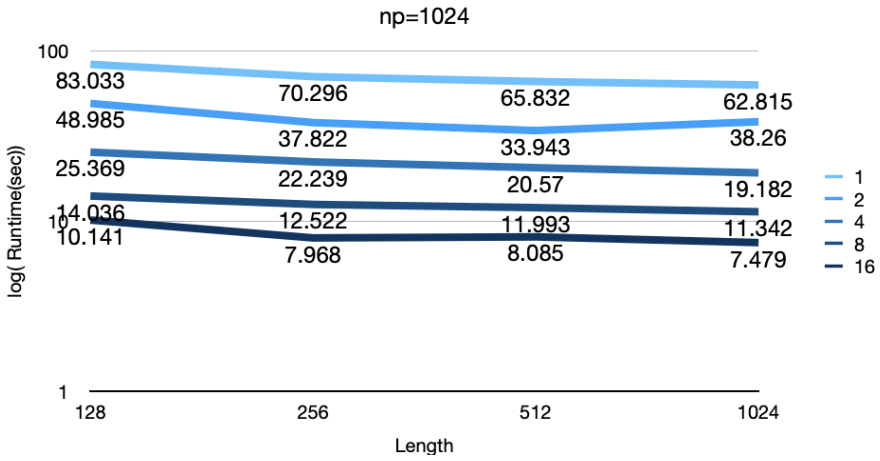

Figure 10 Comparison of computational time in seconds for different space length with same number of particles

In Figure 10, we conclude that the leading factor of our MD simulation is the number of particles, and the parameter of space length doesn't affect the computate time since the particles are sparse enough.

Animation

We animated the particle's movements using Matlab. We first imported the particle's coordinates in x-y plane at each time step. Then we implanted two for loops to sort the coordinates of each particle. Finally, we plotted the coordinates of each particle at different time steps and combined

the plots using built-in “VideoWriter “ function in MATLAB to generate the short video of particle movements.

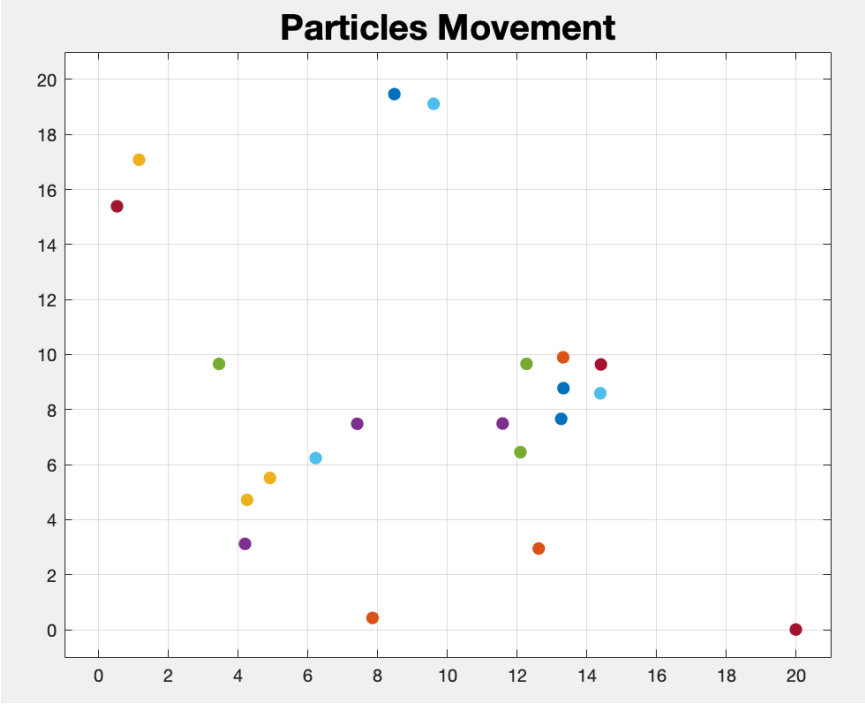

Figure 11. Individual graph plotted in MATLAB

Conclusion

Our project aimed to figure out the best and most efficient way to simulate molecular dynamics. We first investigated two versions of the neighborhood table, and we found that the neighborhood table saved 60% computational time compared to the brute force calculation. Then we parallelize the code using openACC. After implementing openACC, the run time of our code decreased significantly for simulating a large number of particles. However, when less than 100 particles were simulated, the basic code took less time to run. We also tried using openMP to help us accelerate the run time of our code. The results showed that the more threads there are, the less run time needed for calculating a large number of particles.