

# Parallel Computing for Molecular Dynamics Simulation

Final Project of 6.338J/18.337J: Applied Parallel Computing

**Student Name: Yi Zhang**

## 1. Introduction

In materials modeling, we often need to follow the evolution of the materials by calculating the positions of all the atoms. For crystalline solid state materials, initially all the atoms are at well-defined lattice positions. Each atom is under the influence of other atoms. The forces of other atoms will cause the atom to move. This is a complex multi-body problem and does not have an analytic solution. As a result, computer is often used to simulate the evolution of the system.

This final project will focus on one particular material: Gold. Both serial code and parallel code will be produced to simulate the system. Code will be written in C. In the parallel case, MPI is used. The accuracy of the parallel code is tested by comparing its results to the results of the serial code; the accuracy of the serial code is assumed. In reality, the accuracy of the serial code should be tested by comparing the results with physical reality, but this is a course on parallel computing, so we'll focus on the parallel code instead.

The performance of the parallel code is compared with the serial code. We have also measured the performance of the parallel code as number of processes increases; as the system size increases; and as the neighbor list updating interval changes. The neighbor list updating interval will be explained later. The performance measurement results are shown in section 6.

## 2. The system to simulate

We will use Gold as our model system. It is actually a very simple system because there is only one type of atoms in the system. Here are some experimental properties of gold (from WebElements: <http://www.webelements.com/>)

Name: gold

Symbol: Au

Atomic weight: 196.96655

Structure: ccp (cubic close-packed), with cell size:  $a = b = c = 4.0782$  Angstrom. ccp is also called fcc: face-centered cubic.

Bulk modulus: 220 GPa

Melting point: 1337.33 K

We will use Lennard-Jones potential to model the force between atoms of the system. In Lennard-Jones potential, forces exist between every pair of atoms, and the force (direction and magnitude) depends only on the distance between the two atoms. This force is not affected by the presence or non-presence of other atoms. This is why this

potential is called pair potential. If the distance between atoms is  $r$ , the potential energy  $E$  is:

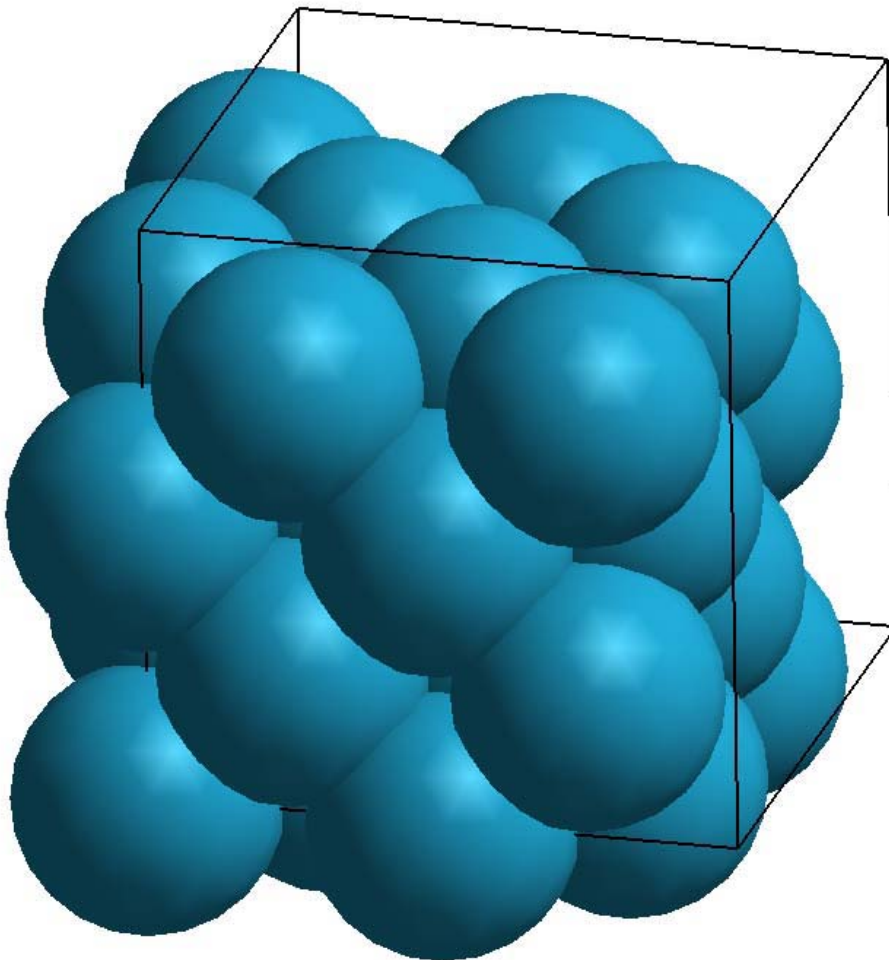
$$E = A/r^{12} - B/r^6$$

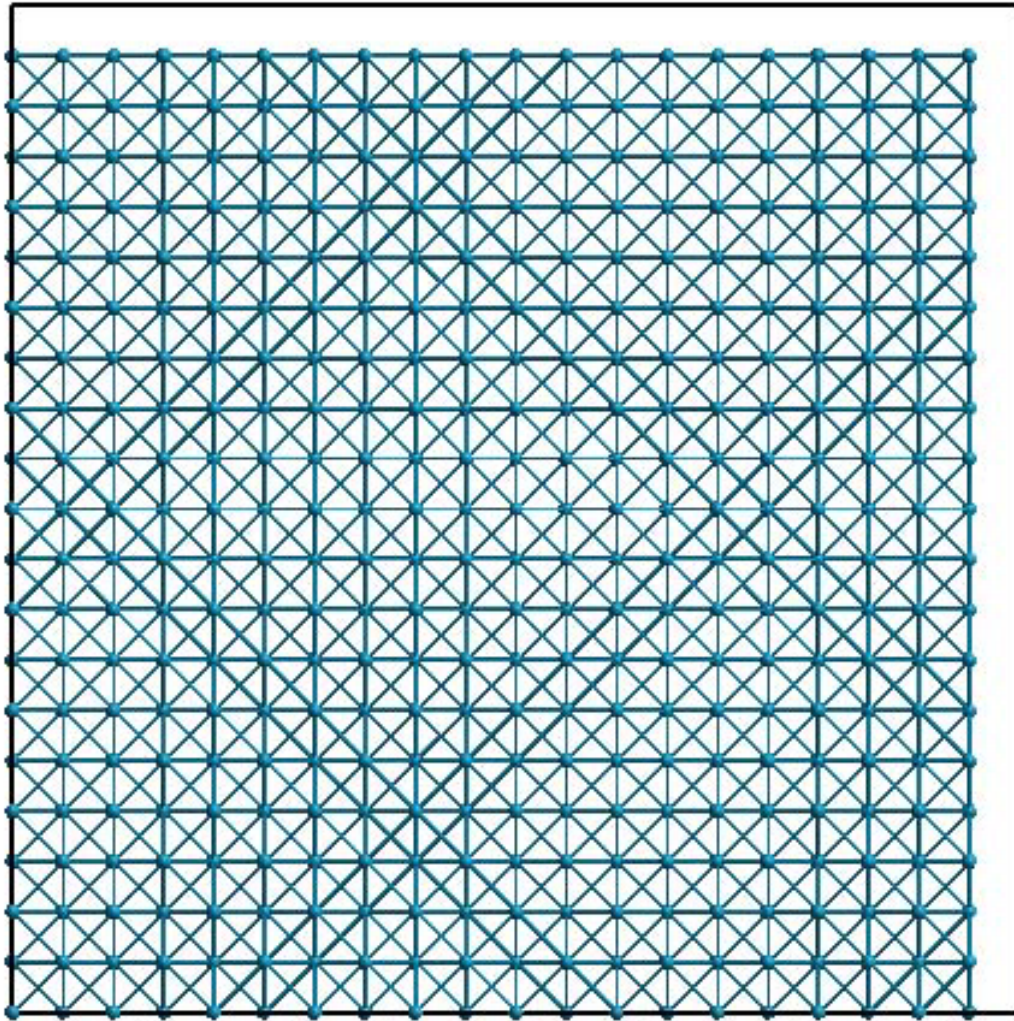
Here  $r$  is in unit of angstrom,  $E$  is in unit of eV, and  $A$  and  $B$  are two parameters to be chosen. Here  $A$  and  $B$  are chosen to be:

$$A = 158675.062040 \text{ (eV*angstrom}^{12}\text{)}; B = 463.262145 \text{ (eV*angstrom}^6\text{)}$$

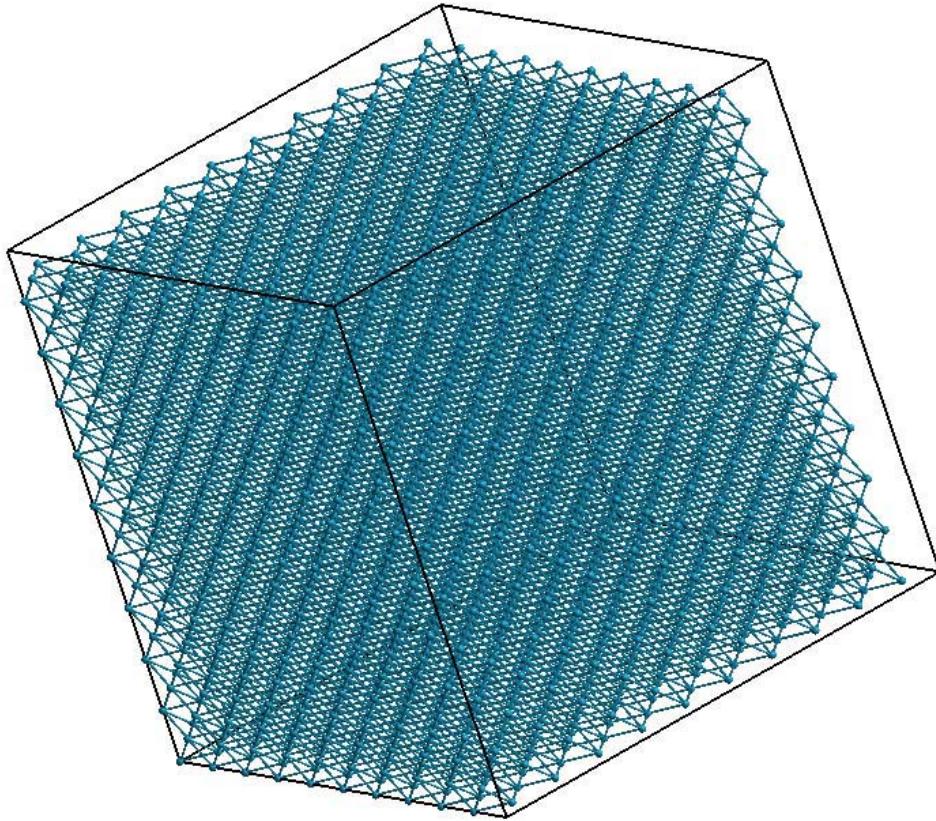
Put these potential parameters into GULP (a software which is irrelevant to what we are trying to do in this course), we can calculate that the lattice constant: 4.078200 (Angstrom), and the bulk modulus: 219.99974 (GPa). These are very close to the experimental values given above. This is of no surprise because we have fitted the potential parameters (the values of  $A$  and  $B$ ) to these two particular properties.

Here are some pictures of the Gold (Au) crystal structure:



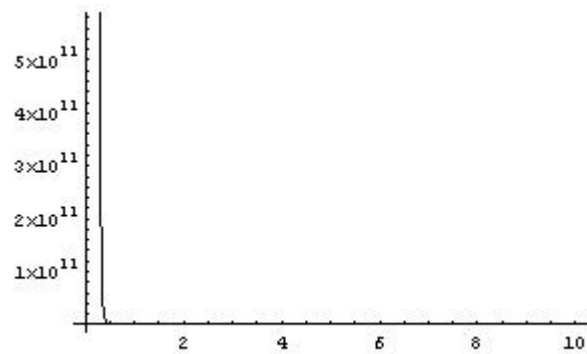






Atoms are in a regular crystal lattice.

Here is a graph of the potential chosen:



From the the potential graph, it can be seen that the potential value goes to zero very quickly. Out of a certain chosen cutoff distance, the potential energy and force between the two atoms are so small that they can be ignored all together. This is a great speed up for computer code: without this, we have to calculate the influence of every other atoms on a given atom; now we only need to consider the atoms within a certain sphere. Because the number of atoms in this sphere is much smaller than the total number of atoms in the system, we get a computational speed-up.

We will use a cutoff of 6 angstroms here. That is, atoms with distance greater than 6 angstrom are considered not exerting any force on each other. The Au system has a lattice constant of 4.0782 angstrom. This is the distance of the cubic unit cell, not between the closest neighbors. In face-centered cubic structure, the distance between closest neighbors is  $4.0782 / \sqrt{2}$ ; distance between second closest neighbors is 4.0782; and distance between third closest neighbors is  $4.0782 * \sqrt{2}$ , which is 5.767 angstroms. So a cutoff of 6 angstrom has already included the influence of third nearest neighbor, which is sufficient for this non-ionic system. Also, we can calculate the potential energy at  $r = 6$  directly use the formula  $E = A/r^{12} - B/r^6$ , the value is: 0.01 eV, which is indeed negligible.

### 3. The serial code

The serial code is in source file serial.c. It reads input from a given file and writes results to a given output file. The input and output files are specified as command line parameters. The first command line parameter is input file name; the second is output file name. This code needs only two command line parameters.

The format of the input file is as follows. The parameters specified in the input file will be explained after the description of the input file format.

1<sup>st</sup> line: super cell size, 3 integers.

2<sup>nd</sup> line:

1<sup>st</sup>: time interval per step, in units of ps (pico seconds,  $10^{-12}$  seconds), double type.

2<sup>nd</sup>: equilibration time step, integer type.

3<sup>rd</sup>: velocity scaling interval, integer type.

4<sup>th</sup>: total time step, integer type.

3<sup>rd</sup> line:

1<sup>st</sup>: temperature, in unit of Kelvin, double type.

2<sup>nd</sup>: force cutoff value, in unit of angstrom, double type.

4<sup>th</sup> line:

1<sup>st</sup>: start time step of dumping atom positions to output file, integer type.

2<sup>nd</sup>: time step interval of dumping atom positions to output file, integer type.

5<sup>th</sup> line:

time step interval of updating neighbor list.

Here is the explanation of what these input parameters mean:

Super cell size: A unit cell of Gold contains 4 atoms. As described before, it has a face-centered cubic structure. The 4 atoms of the unit cell are located at (0, 0, 0), (0.5a, 0.5b, 0), (0.5a, 0, 0.5c) and (0, 0.5b, 0.5c), here  $a = b = c = 4.0782$  angstroms. To build the super cell for our simulation, we need to replicate the unit cell in x, y and z directions. For example, specifying a super size of  $10*10*10$ , we have totally 4000 atoms in our system.

Time interval per step: we will advance the system in discrete time intervals, this is the value of the step size.

Equilibration time step: we are simulating the system at a given temperature. At the start of the simulation, we will give all the atoms in the system velocities according to the Maxwellian Velocity Distribution. The system temperature is determined by the kinetic energy of the atoms, which is determined by the velocities of the atoms. As the evolution of system, the system will move away from the designated temperature. To move the system back to the desired temperature, we will scale the velocities of all atoms according to the desired temperature. This artificial scaling occurs only at the beginning of the simulation. For example, if you specify equilibration time step to be 500, then the velocity scaling will only occur at the beginning 500 time steps.

Velocity scaling interval: as described above, we need to re-scale atom velocities at the beginning of the simulation. We don't need to scale velocity every time step. This parameter specifies the time step interval to scale velocities.

Total time step: the total time step to run.

Temperature: the system temperature.

Force cutoff value: atoms with distance longer than this cutoff value are considered not to have any force on each other.

The start time step of dumping atom positions to output file: start at this time step, atom positions are outputted for later analysis. This start time step generally should be larger than equilibration time step, because during equilibration we are scaling velocities and the system is considered not to be in stable state.

Time step interval of dumping atom positions to output file: the code will output atom positions every few time steps, according to the value of this setting.

Time step interval of updating neighbor list: we mentioned before that a list of neighbors within force cutoff is maintained to reduce the computation cost. This neighbor list needs to be updated periodically because atoms move around. We adapt a simple method here: the user specify the time step interval to update neighbor list. Alternatively, we could devise some criteria to automatically update neighbor list. For example, we can decide that the neighbor list is updated whenever atoms are moved far away enough from its position during the last updating of neighbor list. How far away will trigger the updating of neighbor list could again be a configurable parameter.

If the system is in high temperature, the atoms are moving fast, so the updating interval should be relatively short; on the other hand, if the system is in low temperature, the updating interval could be long for better performance.

Note that no error checking is performed on the input file, so if you specified a mal-formatted input file, the effect is undefined.

The format of the output file is as follows:

First part is an echoing of input parameters. This echoing can be used to make sure that the input parameters are read correctly by the program.

After that, the output file contains the dump of positions of the atoms. The first line of each dump will be “timestep n time”, where n is the time step and time is the time elapsed since start of the simulation. For example, if time interval is 0.001 ps, and one of the atom positions dump occurred at time step 1000, then this line will be:

```
timestep 1000 1
```

After this, the positions of the atoms are outputted one-by-one. Each atom is outputted on two lines. The first line will be the index of the atom; the second line will contain 3 numbers, which will be the x, y and z coordinates of the atom at the time.

After describing the input and output file format, now we will give a brief description of the serial source code itself.

In the unit cell, the lattice constant is  $a = 4.0782$ . We have 4 atoms at:

```
(0, 0, 0)
(0.5 * a, 0.5 * a, 0)
(0.5 * a, 0, 0.5 * a)
(0, 0.5 * a, 0.5 * a)
```

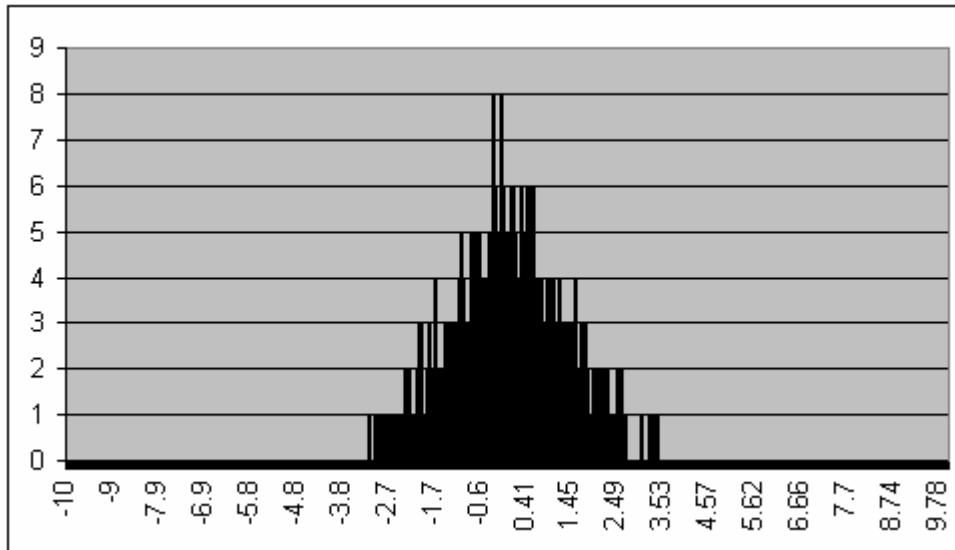
These 4 atoms are considered to be in cell (0, 0, 0). We need to generate the positions of atoms in other cells. This is very easy: just add multiple of the lattice constant to the original coordinates of the 4 atoms, based on which cell they are in. For example, to generate the all atoms in cell (1, 1, 2), we only need to add  $a$  to x coordinates,  $a$  to y coordinates, and  $2*a$  to z coordinates.

Repeat this process to all cells in the super cell, the initial coordinates of all atoms are generated.

Next step is to generate the initial velocities according to Maxwellian distribution and to the temperature specified in the input file. Maxwellian distribution is actually Gaussian distribution, so we will first generate velocities using Gaussian distribution with mean 0.0 and standard deviation 1.0. Later we will scale the velocities according to the given temperature value.

To generate Gaussian random numbers, the code uses the polar method of G. E. P. Box, M. E. Muller, and G. Marsaglia, as described by Donald E. Knuth in The Art of Computer Programming, Volume 2: Seminumerical Algorithms, section 3.4.1, subsection C, algorithm P. This random number generator is coded in function nextGaussianRand().

To test the Gaussian random number generator, I generated 4000 numbers using it, as shown in the following graph:



We can see the signature bell shape in the figure.

So the first step in generating initial velocities is simply call the random number generating method:

```
for (i = 0; i < nAtoms; ++i) vx[i] = nextGaussianRand();
for (i = 0; i < nAtoms; ++i) vy[i] = nextGaussianRand();
for (i = 0; i < nAtoms; ++i) vz[i] = nextGaussianRand();
```

After that, we do a velocity scale according to the given temperature.

```
velocityScale(vx, vy, vz, nAtoms, temperature);
```

This function is also called during the equilibration steps when the velocities are scaled according to the given temperature. The algorithm to scale the velocities is quite simply. From the equi-partition law, at temperature  $T$ , each atom should have kinetic energy  $3/2 kT$ , where  $k$  is the Boltzmann constant. If we have totally  $nAtoms$  atoms, the total kinetic energy should be:  $targetE = 3/2 kT * nAtoms$ . The real kinetic energy of an atom can be calculated as  $\frac{1}{2} mv^2$ . Sum over all atoms, we get the real kinetic energy  $realE$ . The scaling factor will be  $targetE / realE$ . Because we squared the velocity when calculating kinetic energy, the scaling factor for velocity will be  $\sqrt{targetE/realE}$ . Each velocity is then multiplied by this scaling factor.

We then enter the main loop of the simulation. In every time step, we check whether the velocity needs to be re-scaled, according to the parameter given in the input file. We will also check to see whether the neighbor list needs to be updated.

In both calculating neighbor list and calculating the force between two atoms, we need to calculate the distance between two atoms. Periodic boundary condition is used here. The real material contains in the order of  $10^{23}$  atoms, which is considered to be infinity comparing to what we can actually simulate. So the simulation super cell is not in isolation. We can imagine that the simulation is repeated infinitely in three dimensions. So every atom has infinite images in the space. When an atom leaves the simulation cell on one side, its image comes in from another side.



To calculate the distance between atom A and atom B, we consider the distances between all of A's images (including A) and all of B's images (including B), and the shortest distance of all these distances will be the distance we will use. This is based on the assumption that the effect of the second and higher order nearest images pairs can be neglected comparing the nearest pair of images. Because the repetition unit is the super cell, when we choose the super cell, we ought to make it large enough such that only nearest images pair need to be considered (yet small enough so that the simulation can be done fast).

A one-dimension example is used to illustrate how to calculate distance when periodic boundary condition is used. Say we have a simulation box of size length Len in this one dimension example. Atom i is at  $x[i]$ , and atom j is at  $x[j]$ . Normally, the distance between  $x[i]$  and  $x[j]$  will be the absolute value of  $xDis = x[j] - x[i]$ . To incorporate periodic boundary condition, we need to use:  $xDis - Len * \text{floor}(xDis / Len + 0.5)$ . Floor is the mathematical function that calculates the largest integer less than or equal to the given parameter.

For example, if Len is 10,  $x[i]$  is at 1,  $x[j]$  is at 9, then  $x[j] - x[i] = 8$ . However,  $x[i]$ 's image is at  $1 + 10 = 11$ , so the nearest distances between images pairs is  $11 - 9 = 2$ . Using the formula, we have  $xDis - Len * \text{floor}(xDis/Len + 0.5) = 8 - 10 * \text{floor}(8 / 10 + 0.5) = 8 - 10 * \text{floor}(1.3) = 8 - 10 = -2$ , which has the correct absolute value we want.

Do similar calculations for the y and z dimensions, we can get yDis and zDis. The distance can then be calculated as:  $\text{distance} = \sqrt{xDis^2 + yDis^2 + zDis^2}$ .

Also, when advancing atom position from  $r(t)$  to  $r(t+dt)$ , if the positions go beyond the boundary, we need to change to the positions of the image that comes in from the other side. For example, for x coordinates, we need to keep it between 0 and length of simulation box in the x direction: xLen.

$$0 \leq rx < xLen$$

In the imaging process, we move the x coordinates in units of xLen:

$$rx\_new = rx + xLen * n$$

where n is an integer. Thus:

$$0 \leq rx + xLen * n < xLen$$

From which:

$$(-rx) / xLen \leq n < (xLen - rx) / xLen = (-rx) / xLen + 1$$

which means:

$$n = \text{ceil}(-rx / xLen).$$

Thus:

$$rx[i] = rx[i] + xLen * \text{ceil}(-rx / xLen)$$

Similarly for y coordinates and z coordinates arrays.

The neighbor list is kept in variable neighbor, which is of type int\*\*. neighbor[i] will contain neighbors for atom i; neighbor[i][0] will be the size of the neighbor list, and then neighbor[i][1] -> neighbor[i][neighbor[i][0]] will be the neighbors of atom i. During updating of neighbor list, if the number of neighbors exceeds the original memory spot allocated, the memory is reallocated with double the original size; old neighbors are moved from the old place to the new place, and the memory for old neighbor array will be freed.

In the mainly loop of the simulation, we calculate the force experienced by each atom. The potential energy used is:  $E = A/r^{12} - B/r^6$ , from which we can calculate the force F:

$$F = -\frac{\partial E}{\partial r} = \frac{12A}{r^{13}} - \frac{6B}{r^7}$$

where  $A = 158675.062040$  (eV/angstrom<sup>12</sup>);  $B = 463.262145$  (eV/angstrom<sup>6</sup>).

Divide F by mass of atoms, we get accelerator a. From this, we have several algorithm to choose to advance system from t to t + dt. Here, the Velocity Verlet algorithm is used:

$$r(t + dt) = r(t) + v(t)dt + \frac{1}{2}a(t)dt^2$$

$$v(t + dt) = v(t) + \frac{1}{2}[a(t) + a(t + dt)]dt = v(t) + \frac{1}{2}a(t)dt + \frac{1}{2}a(t + dt)dt$$

During each time step, we calculate r(t+dt) and v(t+dt) from r(t) and v(t). These calculations are calculated repeatedly until the total number of time step is reached.

The final source code is in file serial.c.

## 4. The parallel code

Now we turn to the parallel code, which is in file parallel.c.

At first, I tried to partition the super cell into parts based on geometry. For example, if we have 8 processes, and denote the super cell lengths as xLen, yLen and zLen, the eight parts could be:

Part 0:  $0 \leq x < xLen/2$ ;  $0 \leq y < yLen/2$ ;  $0 \leq z < zLen/2$ .

Part 1:  $0 \leq x < xLen/2$ ;  $0 \leq y < yLen/2$ ;  $zLen/2 \leq z < zLen$ .

Part 2:  $0 \leq x < xLen/2$ ;  $yLen/2 \leq y < yLen$ ;  $0 \leq z < zLen/2$ .

Part 3:  $0 \leq x < xLen/2$ ;  $yLen/2 \leq y < yLen$ ;  $zLen/2 \leq z < zLen$ .

Part 4:  $xLen/2 \leq x < xLen$ ;  $0 \leq y < yLen/2$ ;  $0 \leq z < zLen/2$ .

Part 5:  $xLen/2 \leq x < xLen$ ;  $0 \leq y < yLen/2$ ;  $zLen/2 \leq z < zLen$ .

Part 6:  $xLen/2 \leq x < xLen$ ;  $yLen/2 \leq y < yLen$ ;  $0 \leq z < zLen/2$ .

Part 7:  $xLen/2 \leq x < xLen$ ;  $yLen/2 \leq y < yLen$ ;  $zLen/2 \leq z < zLen$ .

Part  $i$  ( $0 \leq i \leq 7$ ) will be given to process  $i$  for simulation.

However, this method of partition based on geometry is rejected because of the difficulty to calculate the neighbor list. Because atoms move around, to keep track of where they are is not an easy job. When atoms move from one geometric part belonging to one process to a geometric part belonging to another process, we need to actually “move” the atom from the memory of the old process it belongs to to the memory of the new process. It’s even more difficult to maintain the neighbor list. You can not keep every piece of information about the neighbors in a separate array, because there will be heavy overlap in the neighbor list, say atom A will be neighbor of both atom B and atom C, and keep all the information about the neighbors will definitely result in memory space waste. It is much better to just keep a reference or index of the neighbor. When you need the positions of the neighbors, you can then use this index to get them from an array. Only one array is kept so memory is not wasted. However, if we use the geometric partition method, we don’t have a general way to find to which process a given neighbor belongs to. We only have an index, but the partition of the atoms to processes is based on geometrical location.

Instead of the geometric partition, we will just partition the atoms based on their indices. Every process will get roughly  $nAtoms/numProcs$  atoms for simulation. Say  $m = nAtoms / numProcs$ , then process 0 will have atoms from 0 to  $m - 1$ ; process 1 will have atoms from  $m$  to  $2m - 1$ . Process 0 through process  $numProcs - 2$  will each have  $m$  atoms; process  $numProcs - 1$  will then have all the remaining atoms.

Every process will keep an array containing all the positions of every atom, but it will only change the part that belongs to it. For example, process 0 will still contain the positions of all the atoms, but it will only update the positions of atoms 0 to  $m - 1$ .

Now neighbor list will be maintained by an index into the whole array. Every process can just use its local array for positions of the neighbors.

Atom positions are constantly changing. However, we won’t synchronize the whole array of positions every time step. That will be too much a computational and communication burden. The whole point of introducing neighbor list is that we don’t have to update it in every time step. Instead, we will only synchronize when it is time to update neighbor list.

As in serial code, we maintain a neighbor list for each atom until it is time to update. However, in serial part, the atom positions of the neighbors are always up to date, because they are updated every time step. In parallel code, the positions are not up to date. Atoms are “moved” in other processes every time step, but their positions will be updated locally only during atom positions synchronization, which happens before updating neighbor list.

This isn’t the only place that will result in different simulation results between the parallel and the serial codes. Another place is regarding the calculation of the

acceleration. Because  $v(t+dt) = v(t) + \frac{1}{2}a(t)dt + \frac{1}{2}a(t+dt)dt$ , we first calculate the acceleration at time  $t$  based on  $r(t)$ , we can then advance the system positions from  $r(t)$  to  $r(t+dt)$ , and calculate the first part of  $v(t+dt) = v(t) + \frac{1}{2}a(t)dt + \frac{1}{2}a(t+dt)dt$ :  $v(t+dt) = v(t) + \frac{1}{2}a(t)dt$ . Then the acceleration  $a(t+dt)$  is calculated, and finally  $\frac{1}{2}a(t+dt)dt$  is added to the velocities to finish calculating velocities for time  $t+dt$ . In the next time step, we don't need to re-calculate  $a(t)$ : the values  $a(t+dt)$  from the previous time step is still valid.

However, in the parallel code, because the atom positions are not synchronized between processes every time step, positions of atoms that belong to other processes are out-of-date, which means  $a(t)$  will also be slightly different from the serial code. Even after synchronizing atom positions, the  $a(t)$  will still be different because  $a(t)$  is the  $a(t+dt)$  in the last time step, which uses the “old” atom positions.

These are the only two places where there will be difference between the result of the parallel code and the result of the serial code.

This will not be a very big problem because we have already chosen the time step interval to update the neighbor list such that atoms are not moved very far between neighbor list updating. Otherwise, the serial code will not be accurate itself.

In any case, this approximation won't be any worse than our previous many approximations made. For example, the empirical potential model and the potential parameters used are best seen as approximations to the real system.

This way, the programming is much easier than the method of partitioning the system into geometric regions. Now we can know where atoms are simply by looking at their indices. For example, we know atom 0 is located at process 0, so during synchronization, we broadcast atom information using process 0 as the root.

Note that the velocity array and acceleration array are not synchronized. This will reduce roughly 2/3 of the communication cost. We don't really need the velocities and accelerations of every atom in every process. We only need the positions of neighbors to calculate the force experienced by a given atom.

Every process uses different output file so that we can avoid the communication overhead to bring the positions of every atoms to a certain node. Process  $i$  has its output file name as `[out_file]_i`, where `[out_file]` is the output file you specified on the command line. For example, if you specified output file `parallel_out`, then the output file of process 5 will be `parallel_out_5`.

Every process only output the positions of atoms that belong to it. After the job exits, you can post-process all the output files to combine them into a single one.

Process 0 will be responsible to read the input file, generate the initial positions and velocities, just as in the serial code. After reading the input parameters, process 0 will broadcast these parameters to every other process.

Then, process 0 will generate the initial positions and initial velocities, and send the atoms to the process where they belong to according to their indices.

In the parallel code, velocity scale is done some what differently from the serial code, because the velocity array is not synchronized across all processes. Every process will then calculate its local kinetic energy sum. These partial sums are summed together at process 0. Process 0 will then calculate the velocity scale factor, and broadcast the factor value back to every other process. All processes then update the part of the velocity array that belongs to it. It is easier to understand by looking at the source code directly: please see function `parallelVelocityScale`, and compare with the serial counterpart: `velocityScale`.

Neighbor list is calculated as in serial case, but again the neighbor list is only calculated for the atoms that belong to this process. Also, before calculating neighbor list, a synchronization of atom positions array is performed.

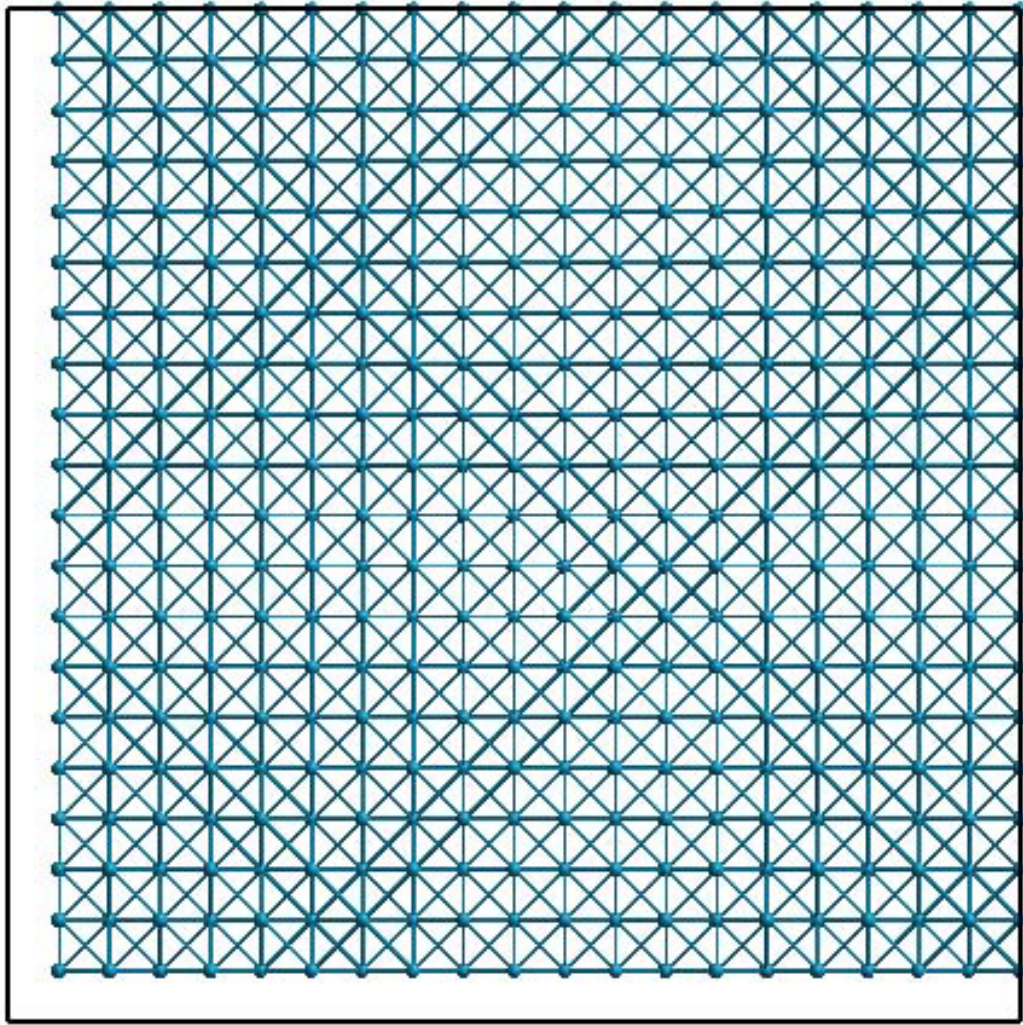
## **5. Accuracy of the parallel code**

The accuracy of the parallel code is determined by comparing the output from the parallel code with that from the serial code.

However, we can not compare the output directly. Remember before the simulation, the initial velocities are generated according to the Maxwellian distribution. A random number generator is used for this purpose, so even different runs of the same code will produce different results due to different generated random numbers.

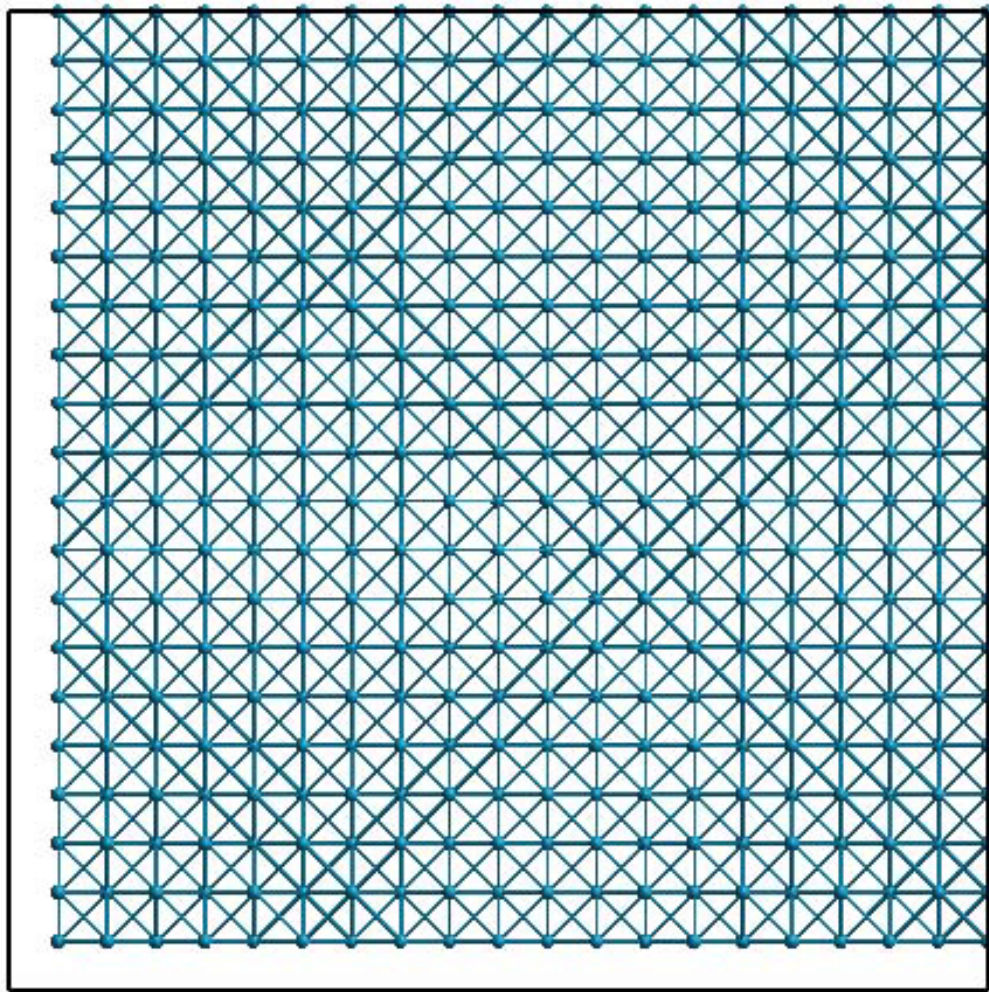
To get around this difficulty, during the accuracy test of parallel code, I temporarily disabled the random velocity generator. The randomness comes from the function `nextUniformRand`, which uses the c function `rand()` and `RAND_MAX` to generate a random number between 0 and 1. I changed this function to always return 0.2, thus all the randomness is removed from the program. This is only used for the purpose of comparing the results of parallel and serial codes. The modified codes and the output files are contained in the directory `accuracy`.

First, we run the simulation for only 1 time step. The output from the serial code is visualized in following figure:



The output from the parallel code is visualized in following figure:





This figure is pretty much the same as the figure for serial code, and they are also the almost the same to the initial perfect crystal structure. This is expected because after only one time step, the system hardly moved at all.

A more numbers-oriented comparison can be performed like this: from the output of each code, we have the positions of each atom positions. We can calculate the distance between the position of atom  $i$  from the serial code and the position of atom  $i$  from the parallel code. We can calculate the average of this distance. If this average distance is small, the result of the parallel code is close to the result of the serial code.

In this case, because we run the system only for 1 step, the position should be exactly the same. Remember that the only algorithm difference between the serial and parallel code is that in each process of the parallel code, we didn't update the positions of atoms that do not belong to this process. Now we only run for 1 step, so there is no update to be done anyway. Let's see whether the average distance is indeed zero.

Before calculating the distance, we should mention again the periodic boundary condition. We should calculate the distance between closest image pair. For example, here the length in the  $x$  direction of the super cell is 40.782. So an atom at  $x = 0$  and an atom at  $x = 40.782$  have distance 0, not 40.782.

The average distance is calculated by CalcAveDistance.java, which is an ad hoc java code developed to do the job. First, because the output from parallel code is scattered as output\_1, output\_2, ..., we need to combine them into a single output file the same format as the output file from the serial code. Then use this combined file and the output file from the serial code as the input files to CalcAveDistance.java, the calculated average distance is indeed 0.0, as expected.

It isn't much interesting to run the system for only 1 time step. Next we run the system for 100 time step, and the average distance is 0.1134226 angstrom, which is acceptable.

So we have shown that the result of the parallel code is acceptable comparing to the result of the serial code, but is the result of the serial code accurate? Normally we should verify that the simulation is in accordance with physical reality. However, because this project is about parallel programming instead of physics or materials science, we will assume that the serial code does the right thing. Instead, we will focus on the comparison of the serial and parallel code.

## 6. Performance of the parallel code

Now we turn to the performance of the parallel code. The parallel code has timing built-in. At the beginning of the code, we call `tStart = MPI_Wtime()`; near the end of the code, we call `tFinish = MPI_Wtime()`. `tFinish - tStart` will then be the time, in seconds, the parallel code has run.

There is no timing built into the serial code. It is not difficult to add this functionality though. Here I will use the Linux "date" command for timing. A small shell script is created, in which the serial code run is sandwiched between two "date" commands. The date command will print the date and time on the console. After that, I will manually calculate the time used by the serial code by reading the two printed date value.

We will investigate the effects of the size of the system, number of processes, and neighbor list updating interval.

### Size of the system

The system is run to 1000 time step; neighbor list is updated every 10 steps; 4 processes are used for parallel code.

System size	10*10*10	15*15*15	20*20*20
Serial code time (seconds)	243	1868	9256
Parallel code time (seconds)	96.8537	836.006	4243.07
Serial time/parallel time	2.51	2.23	2.18

Clearly there is benefit to use the parallel code. 2.2 time speed-up using 4 processors isn't too bad.

### number of processes

The system size is  $15 \times 15 \times 15$ ; system is run to 1000 time step; neighbor list is updated every 10 steps

# of processes	serial code	2	4	8
Running time (seconds)	1868	1665.79	836.006	405.113

It is seen that the parallel code scales well. That is, we can benefit more by using more processors.

### **neighbor list updating interval**

The system size is  $15 \times 15 \times 15$ ; # of processes: 8; system is run to 1000 time step.

Neighbor list update interval	10	100	never
Running time (seconds)	405.113	90.2313	54.192

As expected, the parallel code takes more time when we update the neighbor list more frequently. Neighbor list updating is the real performance bottleneck in the parallel code, because before each neighbor list updating, a synchronization of all atom positions is performed, which invokes large data movement. Choosing a good neighbor list updating interval is thus very important. We can not choose too high a value, for accuracy reason; and we do not want to choose too small a value, for performance reason.

## **7. Conclusion**

In this specific scenario, it is worth the effort to develop the parallel code, because the speed-up is significant.

Admittedly, in many aspects the code is over-simplified. For example, we have only one type of atoms in the system; we don't have long-range forces such as columbic forces; the neighbor list updating interval is specified as an input parameter, whereas in reality we may not be able to know the correct value beforehand. However, it is none-the-less a good start.