CBEMD Parallelized Molecular Dynamics in Various Thermodynamic Ensembles

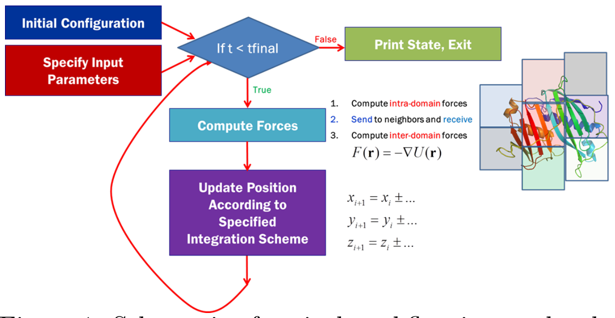
Carmeline J. Dsilva, George A. Khoury, Nathan A. Mahynski,
Junyoung Park, Arun Prabhu, and Francesco Ricci

1 Introduction

We propose to develop a software package to perform parallelized molecular dynamics (MD) calculations for several different thermodynamic ensembles. Molecular dynamics is a technique commonly used to simulate the motions of a system of interest on a microscopic level; one can then calculate macroscopic properties from the microscopic features of this system. Properties such as diffusion coefficients, specific heats, and chemical potentials, which are often difficult or impossible to measure in experiment, are commonly calculated in MD. It has been used to quantitatively measure the folding rate and stability of proteins using probability and enthalpic interactions and entropy through coordinate variance, to measure the melting point of a protein through constructing a sigmoidal curve tracking the root-mean square deviation from the folded structure as a function of temperature, to measure the diffusivities of atoms and larger molecules in solution and in different environments and to perform measurements at temperature extremes that are higher and lower than are experimentally convenient. This list is non-exhaustive. Because of the myriad of properties that can be calculated, MD has applications in such fields as material science, pharmaceutical design, cellular biology, thermodynamics, and fluid mechanics. Due to its relative simplicity and its ability to be massively parallelized, MD is a common but powerful tool for many scientists and engineers today.

2 Background on Molecular Dynamics

MD first initializes a system of particles, usually thought of as atoms, molecules, or another relevant (molecular-scale) group. These particles have initial positions and may have initial velocities. If no initial velocities are assigned, they can be computed. Given an interaction potential $V(\vec{r})$ between these particles, MD integrates Newton's equation of motion, $\vec{F} = m\vec{a}$, forward in time. Schematically it can be loosely illustrated as is shown in Figure 1.



Initial Configuration
Specify Input Parameters
If t < tfinal
False
Print State, Exit
True
Compute Forces
1. Compute intra-domain forces
2. Send to neighbors and receive
3. Compute inter-domain forces
$F(\mathbf{r}) = -\nabla U(\mathbf{r})$
Update Position According to Specified Integration Scheme
$x_{i+1} = x_i \pm \ldots$
$y_{i+1} = y_i \pm \ldots$
$z_{i+1} = z_i \pm \ldots$

Figure 1: Schematic of typical workflow in a molecular dynamics simulation.

Thus, the flow of the program follows a well defined set of steps:

1. Initialize positions and choose a timestep, dt
2. Calculate the forces $\vec{F} = -\nabla V(\vec{r})$
3. Numerically integrate to calculate the displacement over dt, using the relationships $\vec{F} = m\vec{a}$, $\vec{a} = \frac{d\vec{v}}{dt}$, and $\vec{v} = \frac{d\vec{r}}{dt}$
4. Update positions and time
5. Repeat from step 2 until final time

Users of any molecular dynamics program must provide the initial configuration for the particles. The user must also choose or supply the relevant potential function V. Different potential functions can be used to describe the system at different levels of accuracy. These functions typically try to capture van der Waals interactions due to coupling of electron clouds, but can also incorporate additional effects, such as long range charge-charge interactions and solvation interactions (Debye screening effects). Our project will provide an easily extensible interface so that the user can incorporate additional interactions.

For numerical stability, the timestep dt is typically on the order of a femtosecond. The longest simulations reported for macromolecules such as proteins are on the millisecond timescale, but such lengthy simulations can require specialized supercomputing hardware such as ANTON or GPUs. Although the time integration must be done sequentially, at each step, the evaluation of the potential energy and forces can be parallelized, which is the primary reason MD has been so successful on large computer clusters. Alternatively, N independent molecular dynamics simulations can be performed in parallel and often this can allow the sampling of broader space by not being trapped in local minimum.

There are many different numerical integration algorithms that have been developed, each with a slightly different purpose and numerical stability. In this project, we implemented the Verlet integrator and the

Andersen thermostat, which uses the Velocity Verlet algorithm. Both of these integration schemes are developed specifically for molecular dynamics.

Several existing MD packages, such as HOOMD, LAMMPS, GROMACS, and NAMD, incorporate such ensembles. Members of our group use some of these codes in our every day research, but we do not know much about their internal structure. Therefore, it was a prudent exercise to develop our own codebase with a well-written interface that we can each extend for our custom applications.

2.1 Verlet integrator

The Verlet integrator is designed to maintain constant energy (E), volume (V), and number of particles (N). This NVE ensemble is known in thermodynamics as the *microcanonical ensemble.* Such a system is a closed, isolated system, with no interactions with any external bath. This scheme's equilibrium is characterized by maximum entropy.

The Verlet integrator equations are

$$\vec{r}(t+\Delta t) = 2\vec{r}(t) - \vec{r}(t-\Delta t) + \frac{\vec{F}(t)\Delta t^2}{m}$$

$$\vec{v}(t+\Delta t) = \frac{\vec{r}(t+\Delta t) - \vec{r}(t)}{\Delta t}$$

These equations can be used to update the positions and velocities of the atoms in the system, given the current positions and forces on the atoms.

2.2 Andersen thermostat

The Andersen thermostat is designed to maintain constant temperature (T), volume (V), and number of particles (N). This NVT ensemble is known in thermodynamics as the *canonical ensemble.* This scheme is setup in a way such that the system is coupled with an external heat bath that is maintained at a setpoint. Its equilibrium is characterized by the minimum Helmholtz free energy, and energy is not conserved in the NVT ensemble. The kinetic temperature is given by the equipartition theorem.

$$\frac{3}{2}Nk_BT = \sum_{i}^{N_{particles}} \frac{1}{2}m_i\vec{vi}^2$$

In this description, the temperature setpoint specified by a user is denoted as T_{bath}. The system temperature is denoted as $T_{instant}$. In the Andersen thermostat, the velocity Verlet algorithm is used for integration of the equations of motion.

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{\vec{f}(t)}{2m}\Delta t^2$$

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{\vec{f}(t+\Delta t) + \vec{f}(t)}{2m}\Delta t$$

In the first iteration, the forces are calculated between all of the particles. Then, knowing the velocities and forces at time t, we update $\vec{r}(t)$ to determine

$$\vec{v'}(t) = \vec{v}(t) + \frac{\vec{f}(t)}{2m}\Delta t$$

Then, the force is calculated again after the positions and velocities are updated to updated the velocities at $t + \Delta t$.

$$\vec{v}(t+\Delta t) = \vec{v'}(t) + \frac{\vec{f}(t+\Delta t)}{2m}\Delta t$$

As the particle velocities are being updated, using the equipartition theorem the temperature is calculated as

$$T = \sum_{i=1}^{N_{particles}} mv^2$$

Using this, the instantaneous system temperature can be calculated as

$$T_{instant} = T/(3N_{particles})$$

The velocities of particles interacting with the heat bath being applied are next updated. A Gaussian distribution with mean zero and variance $\sigma^2 = T_{bath}$ must be constructed, denoted as $Gauss(0, \sigma^2)$. Thus, the standard deviation of such distribution is simply

$$\sigma = \sqrt{T_{bath}}$$

Then, for each particle, a uniformly distributed random number between zero and one is calculated. If this number is less than the interaction parameter ν multiplied by the timestep dt, then the velocities of that particle are updated as

$$v(i) = Gauss(0, \sigma^2)$$

The above algorithm proceeds until a pre-specified number of steps has completed.

3 Code Goals and Structure

Our code was written with the goals of generalizability and efficiency. Namely we aim to produce a molecular dynamics codebase that can perform simulations

1. Using different ensembles
2. Using an arbitrary number of particles
3. Using an arbitrary system size
4. Using arbitrary potentials
5. That is parallelized using MPI

Our code is written as a mixture of C and C++, and utilizes many object-oriented features. C++ was chosen because of its excellent memory management and ability to be object oriented. C was used to create convenient structs that needed to be constantly passed around between processors (such as the MPI Atom structure). Using C structs was less expensive than using a C++ class as a container to be constantly passed around. Design decisions were constantly assessed to conform to style, were made to be convenient for us to program and for others to adapt, and, for ease of understanding.

The code is optimized for small to medium-sized systems. That is, a pairwise matrix of interactions exists, and we do not make use of neighbor lists in the implementation. In terms of make vs. buy, it is 99.9% make. That is, the only components of the ≈ 4000 lines of code produced that were not written were BOOST, MPI, and Googletests.

The modular class structure allows for a general coding framework, such that users can easily define their own integrators, pair potentials, and bond types. We have parallelized our code using MPI.

We have defined various classes for the different parts of our MD simulation.

1. Atom: The atom class contains only an atom structure. This structure is passed between processors when the code is run in parallel.
2. Integrator: The integrator class is an abstract base class. It defines the implementation structure for integrators. We have implemented two integrators (Verlet and Andersen), but our code can easily be extended to other integration schemes.

3. Interaction: The interaction class is an abstract base class. It defines the interaction between two atoms, including a function pointer to different interaction potentials. We have implemented standard Lennard-Jones potential, the harmonic bond potential, and the finitely extensiable nonlinear elastic (fene) bond potential. It is optimized for small systems since it avoids bond lookup.
4. System: The system class stores the information for the total simulation, including the box size, a list of atoms, and the interactions between atoms (stored as a matrix of function pointers to different interactions). When the code is run in parallel, each processor contains a system object that contains the atoms owned by that processor.

We also have files "read_interaction.cpp" and "read_xml.cpp", which handle file I/O.
The flow of our code is as follows:

1. The code reads in the starting configuration from the user-supplied xml file, and the relevant energy parameters from the user-supplied energy file. Atoms are read from the xml file and assigned to the correct processor.
2. A system is initialized for each processor containing the relevant system parameters from the xml file.
3. An integrator is initialized on each processor.
4. The forces on each atom are calculated
5. When the code is run in parallel, the "ghost" atoms near the edges of each processor domain are passed to the relevant neighboring processor, and the forces between these ghost atoms and the atoms owned by the processor are calculated
6. The atomic positions are updated via the "step()" function for the instantiated integrator
7. Any atoms that have moved outside of the processor domain are passed to the relevant processor.

Steps 4-8 are repeated for each timestep. At each time step, the configuration of the system is output to a file. This output file can be read by VMD, a molecular visualization package so a movie of the simulation can be visualized.

3.1 Input and Output

Our code is initialized using user-supplied xml and energy files. The xml file contains information such as the positions, velocities, and masses of all the atoms in the simulation. The data can be read in any order its provided by invoking the rewind() function. A sample xml file is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<hoomd_xml version="1.4">
<configuration time_step="0" dimensions="3" natoms="2" >
<box lx="10" ly="10" lz="10"/>
<position num=2">
6.22 5 5
3.78 5 5
</position>
<velocity num="2">
0 0 0
0 0 0
</velocity>
<mass num="2">
1
1
</mass>
<diameter num="2">
```

```
1
1
</diameter>
<type num="2">
A
A
</type>
<bond num=1">
feneA 0 1
</bond>
</configuration>
</hoomd_xml>
```

The energy file contains the parameters for each interaction type in the xml file, as shown below. The parameters are the values of ϵ, σ, Δ, U_{shift}, and r_{cut}.

```
PPOT A A slj 1.0 1.0 0.0 0.0 2.4
PPOT A B slj 1.0 1.0 0.0 0.0 2.4
PPOT B B slj 1.0 1.0 0.0 0.0 2.4
BOND feneA fene 1.0 1.0 0.0 30.0 2.4
```

It can be read in as a cascade similar to the input xml file. Its advantages include a "factory-style" workflow for generating the interaction matrix, which avoids constantly needing to perform "bond lookup." Instead, its implemented as just a direct computation.

The shifted Lennard-Jones potential using the energy parameters, can be written as

$$U(r) = 4\epsilon((\frac{\sigma}{r-\Delta})^{12} - (\frac{\sigma}{r-\Delta})^{6}) + U_{shift}, r - \Delta < r_{cut}$$

and the Finitely Extensible Nonlinear Elastic Model bond potential (FENE bond) can be written as

$$U(r) = -\frac{1}{2}kr_0^2 ln(1 - (\frac{r-\Delta}{r_0})^2) + U_{WCA}$$

The force on a particle can be thus calculated as

$$F_i = -\frac{\partial U}{\partial r}\frac{\partial r}{\partial x_i} = -\frac{\partial U}{\partial r}\frac{x_i}{r}$$

At each timestep, our code writes the current configuration to an output file in "xyz" format. This file contains the positions of every atom in the simulation, and the "xyz" format is easily read by VMD and other molecular visualization packages.

To parse the xml file, we used the Boost C++ library to allow for discrepancies in the xml file (such as extra white space, input arguments in arbitrary orders, etc.).

3.2 Domain decomposition

In the final version of our code, we utilize a one-dimensional domain decomposition, where the simulation box is decomposed into vertical slabs. Each vertical slab and the atoms it contains are assigned to a processor, and atoms are passed between processors as needed using MPI. However, we began our project with attempting a three-dimensional domain decomposition, and it was utilized in the majority of the duration of the project. Although we could not get this algorithm completely debugged by the end of this project, we will include information on the structure and implementation since it comprised a large part of our efforts.

The 3-D domain decomposition attempts to spatially split the simulation box into a number of identical domains based on the number of processors available.

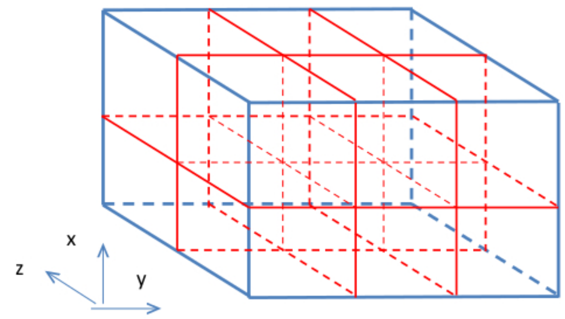


Figure 2: An example of the domain decomposition. The set in use is $\{2, 3, 2\}$ (see step 3 of the algorithm).

Each processor will then be responsible for one domain. Parallelization will be most efficient if each processor has approximately the same load.

If we assume that the particles are uniformly distributed in the simulation box, then the most efficient way to partition the system is one in which the individual domains are as cubic as possible. This is what the 3-D domain decomposition algorithm attempts to do.

Note: The 3-D domain decomposition assumes that the simulation box is a rectangular parallelepiped with one corner positioned at (0,0,0).

The algorithm is as below

1. Generate the prime factorization of the number of processors available (e.g. $36 = 2 \times 2 \times 3 \times 3$).
2. Generate a set of 3 numbers by combining the factors in different ways such that the product is still the number of processors available (e.g. for the case of 36 processors, possible sets are $\{2, 3, 6\}, \{3, 2, 6\}, \{1, 4, 9\}, \{2, 1, 18\}$, etc.).
3. Divide the simulation box into domains by dividing the x, y, z directions respectively into a number of parts based on the 3 numbers in one of the sets generated in step 2. (See Fig. 2 for an example)
4. Check how cubic the domains are by evaluating $diff = (d_1 - d_2)^2 + (d_2 - d_3)^2 + (d_3 - d_1)^2$; where d_1, d_2, d_3 are the x, y, z dimensions of a domain (since all the domains are the same size and shape).
5. Repeat steps 3-4 for all the sets generated in step 2 while keeping track of the set that resulted in the smallest value of diff.
6. Divide the simulation box into domains based on the set which generated the smallest value of diff.
7. Assign the domains to processors by stepping through each domain in the x-direction from lowest to highest and then in y and then in z. (See Fig. 2 for an example)

The code implements steps 2-4 through the recursive function gen_sets in domain_decomp.cpp. The function init_domain_decomp in domain_decomp.cpp starts the process from step 1 through 6 calling gen_sets along the way.

4 Validation

We have compared the accuracy of the results of our simulations generated by our code to LAMMPS, a commercial/academic molecular dynamics software package. Specifically, the radial distribution function $g(\overrightarrow{r})$ was calculated over the course of a 1000 timestep simulation for a 1000 particle system. The initial coordinates and velocities of this system was input to LAMMPS and into CBEMD independently and simulated. Next, the radial distribution function was calculated for the output generated by each method. The comparison is shown in Figure 3.

The radial distribution function is a very important metric that describes the probability of finding a particle at a distance r from a given particle, averaged across all particles. From $g(\overrightarrow{r})$, its possible to derive macroscopic thermodynamic values such as the potential energy and pressure of a system. *It is evident in 3 that the $g(\overrightarrow{r})$ produced by CBEMD is identical to that produced by LAMMPS, validating the accuracy of our program.*

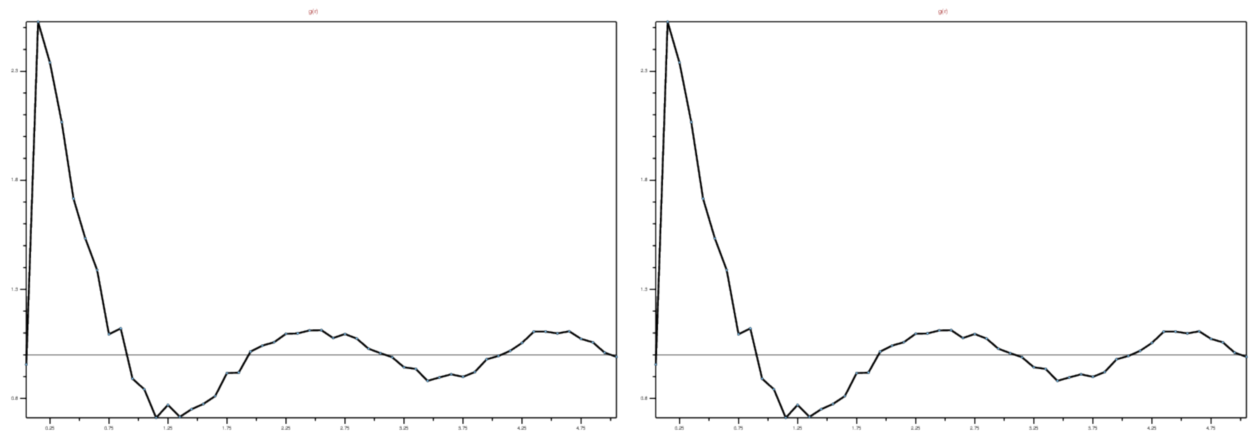


Figure 3: Comparison of the radial distribution function $g(r)$ for our code (left) and LAMMPS (right) for a 1000 atom system of Lennard Jones particles at $t = 1000$ timesteps. The y-axis is the radial distribution function g(r) and the x-axis is the radius r.

4.1 Testing

Testing was performed using GoogleTests to test individual key components of CBEMD. The ultimate test is the exact agreement between results generated by CBEMD and that produced by the commercial package LAMMPS as shown above. Nevertheless, 22 tests were written to test components of domain decomposition, calculation of the harmonic bond, fene bond, and shifted Lennard-Jones potentials, file i/o, and two-body and many body interactions. In addition, tests with domain decomposition included tests on one processor and in parallel, given an arbitrary number of processors.

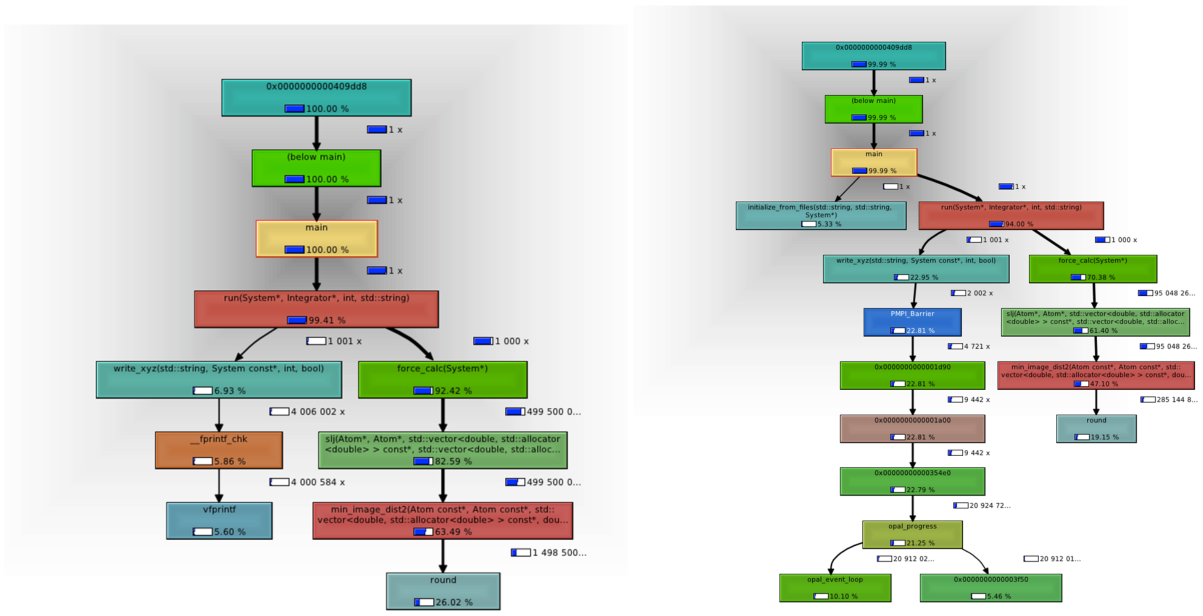


Figure 4: The results from profiling using KCacheGrind while running the code in serial (left) and in parallel on four processors (right).

5 Profiling Results

We used gprof to profile our code and highlight the computational bottlenecks (see Figure 4).

Regions in execution that are large portions of runtime are the force-calculation, with the calculation of the shifted lennard-jones potential and the computation of the minimum-image distance taking up the majority of the time. Similarly, the use of the pow command is unnecessary when the atomic computations can be used instead.

6 Optimization

From the results of our profiling, we opted to optimize the "min_image_dist2()" and the "slj()" functions.

The "min_image_dist2()" uses the "round" function from the math library. For dense systems, or for short simulations, this is inefficient, because most atoms are still near each other. Therefore, we replaced the "round" function with a while loop that adjusts the distance until it is the minimum image distance.

The "slj()" function uses the "pow" function from the math library to compute r^6. However, this function call is probably expensive and could be replaced by $r \times r \times r \times r \times r \times r$.

Therefore, we performed several optimizations to our code to address these expensive components. The results are shown in Figure 5. The removal of the round function in the minimum image distance calculation reduced the time from 1000s to $\approx$ 700s, which is a dramatic reduction. Conversely, comparing the runtime where the slj() function uses pow, vs. when it does the multiplications explicitly shows no improvement. This was initially unexpected. We uncovered the cause to be the use of the intel compiler for this run, which when using the optimization flag, will automatically convert uses of pow to explicit multiplications. As a result, we learned more about how compilers optimize code given to it to compile, which was very interesting.

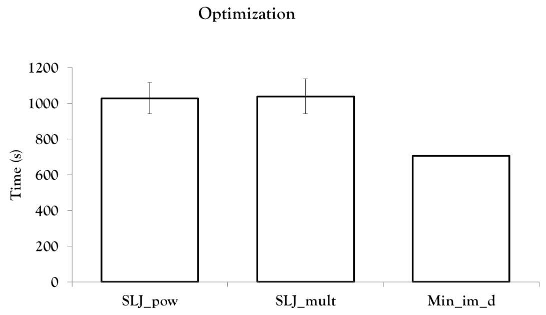


Figure 5: The results of optimization will be explained here.

7 Scaling

We looked at how the execution time of our code scaled with the number of processors.

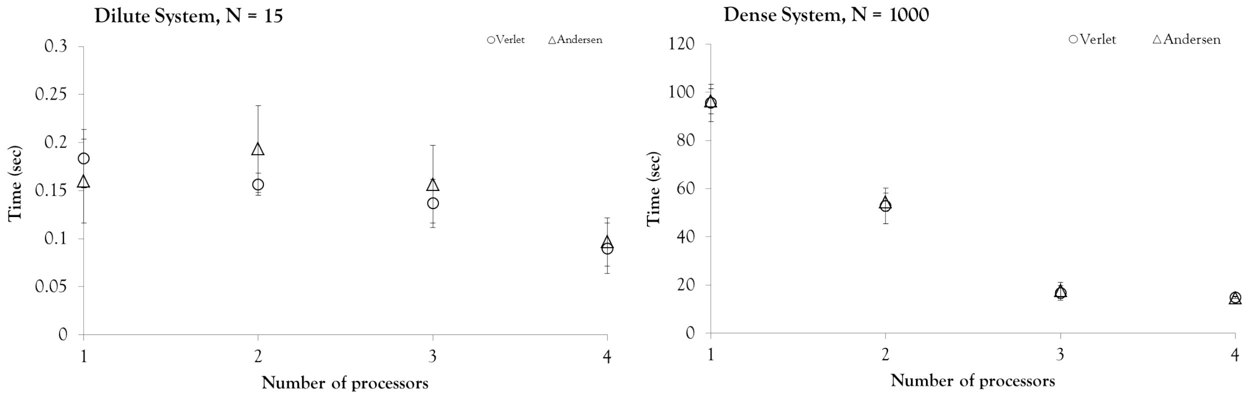
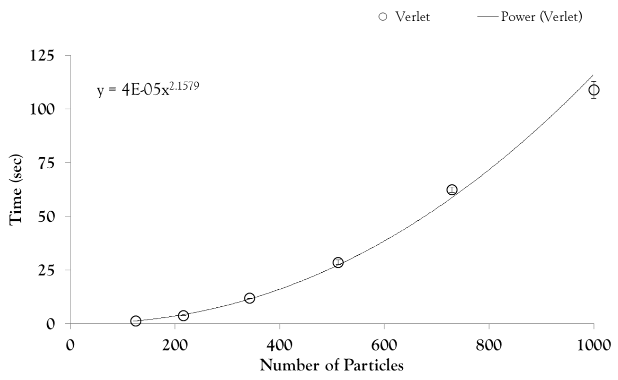


Figure 6: Scaling results as a function of number of processors for a dilute system (left) and a dense system (right).

For the dilute system, there is marginal gains achieved when increasing the number of processors used, since the system is dense and it is rare that information needs to be passed from processor to processor since the interactions are quite sparse. Conversely, for a dense system, there are many instances where information must be passed from processor to processor, so partitioning the calculation across a number of processors decreases the total time.

We next assessed scaling with system size. Theoretically, scaling should be quadratic since interactions are pairwise.



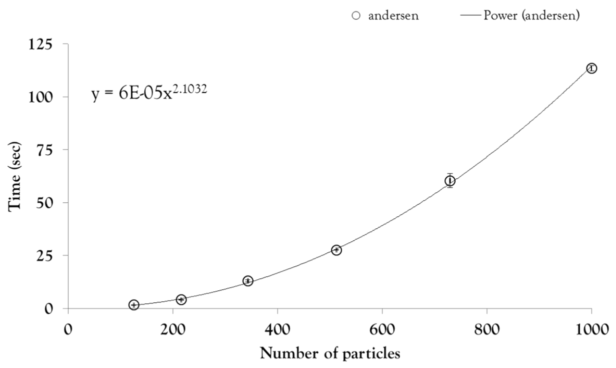


Figure 7: Scaling results for integration under the N,V,E ensemble (Verlet, left) and N,V,T ensemble (Andersen thermostat, right).

It is evident that the both the Verlet and Andersen integrators scale quadratically with system size, as expected. This result is promising as it indicates that the design of our program led to the expected scaling.

8 Sample Simulation Results

9 Documentation

All documentation of the code was done in Doxygen. A pdf file, refman.pdb is a reference manual containing full documentation of all of the functions implemented. In addition, the code comes with a README.md file that contains the instructions to compile and run the code, as well as how to run the code on the university clusters, with sample PBS scripts and sample run commands.

10 Conclusion

A CONCLUSION MUST BE WRITTEN.

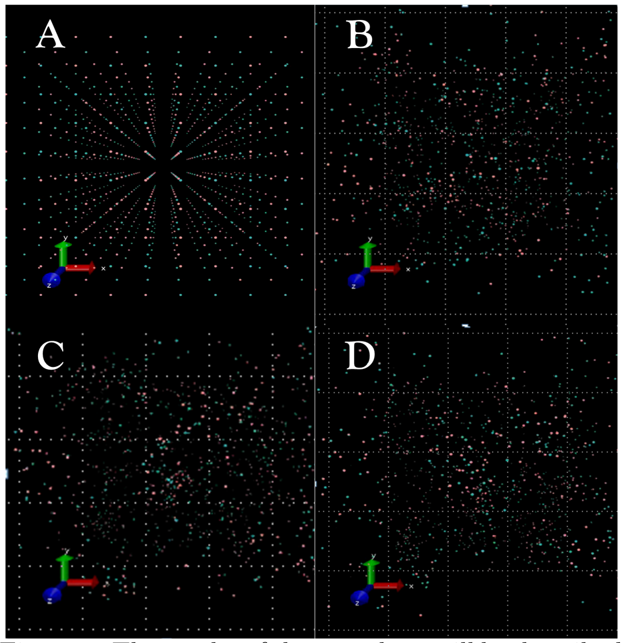


Figure 8: The results of the snapshots will be described here.