Project Report

Sayed Ahmad Almohri, John Sayut, Amir Nazemi, Bjorn Kierulf
NERS 570: Scientific Computing
Instructor: Dr. Brendan Kochunas

1. Project Title

C(H)EETAH-PMCMD:
Comparing Enormous Efficient Thermodynamic simulation of Argon tHrough Parallel Monte Carlo and Molecular Dynamics

2. Team Members

Sayed Ahmad Almohri, John Sayut, Amir Nazemi, Bjorn Kierulf

3. Introduction

This work studies the behavior of Argon as a function of its reduced density. Specifically, we will aim to show the how Argon's reduced pressure behaves as the density increases. Using the Lennard-Jones potential [1] we will run multiple molecular dynamics (MD) simulations which will act as our ground truth. We also developed our in house parallel Markov chain Monte Carlo code with the Metropolis–Hastings algorithm [2]. In both methods, we run the simulations under the canonical (NVT) ensemble [3]. Our MD simulations are run through LAMMPS [4], which is an open-source MPI parallel software developed in Sandia national lab capable of running large scale atomic level simulations. The main goal is to show that we can recover the same trend in our pressure vs density plot across the two different simulation methods that we use. Additionally, we want to show our capability to develop a parallelized Monte Carlo code with observable speedup.

3.1 Science Background

When we consider atomistic simulations, the 2 main available methods are molecular dynamics (MD) and Monte Carlo (MC). MC is a statistical method employing random sampling to effectively explore a system's configurational space. It is especially beneficial in cases where ensemble averages are desired, and the details of dynamic behaviors are less critical. Such scenarios might include studying equilibrium properties and phase diagrams. On the other hand, MD is a computational method that solves Newton's equations of motion for a system of atoms, enabling the exploration of dynamic behaviors and time-dependent properties of a system, such as diffusion and phase transitions. **Figure 1** shows the steps taken in a traditional molecular dynamics simulation.

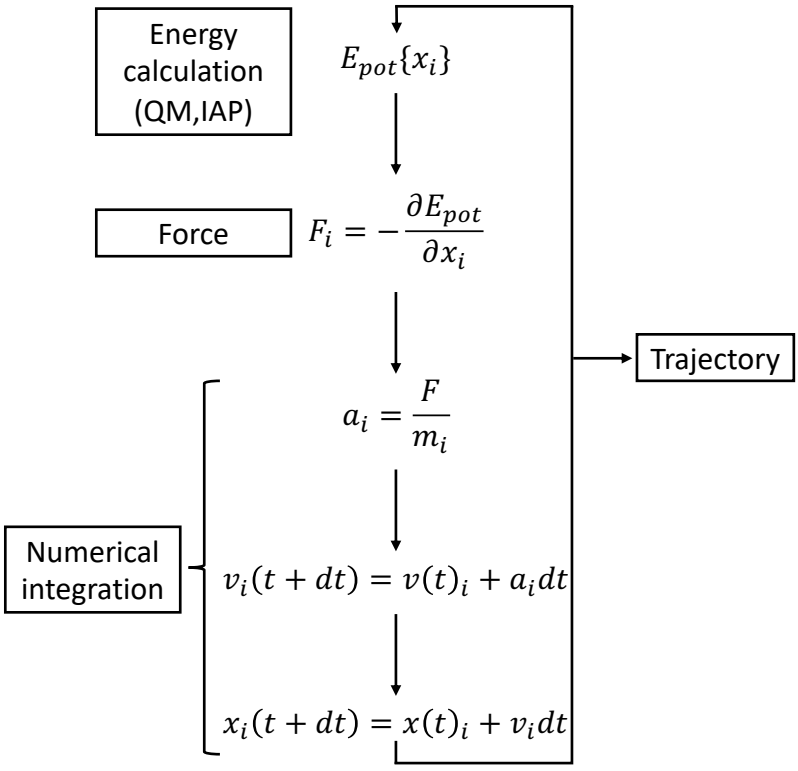

Figure 1: Diagram showing the steps needed in molecular dynamics. Starting from an energy calculation either via an interatomic potential or quantum methods, this step is the main step in molecular dynamics and is the most important part of this cycle which depends on the accuracy of the simulation. The more accurate your energy calculation is the more accurate your simulation will be.

Markov Chain Monte Carlo (MCMC) is a statistical method widely used in scientific computing to sample from complex probability distributions. It is particularly valuable for exploring the configurational space of physical systems, as it avoids the explicit calculation of the partition function, which is computationally infeasible for many-body systems.

The simulations are performed in the canonical ensemble (NVT), where the number of particles (N), volume (V), and temperature (T) are fixed. In this ensemble, the probability of a system being in a configuration C is proportional to the Boltzmann factor:

$$P(C) \propto e^{-\frac{E(C)}{k_B T}}, \tag{1}$$

where $E(C)$ is the total energy of configuration C, k_B is the Boltzmann constant, and T is the temperature.

The Metropolis-Hastings algorithm is a key component of MCMC. It generates a Markov chain of states by:

1. Proposing a new configuration C' from the current state C.

2. Accepting C' with probability:

$$P_{\text{accept}} = \min\left(1, e^{-\frac{\Delta E}{k_B T}}\right), \tag{2}$$

where $\Delta E = E(C') - E(C)$. If $P_{\text{accept}} < 1$, a random number $r \in [0, 1]$ is generated, and the move is accepted if $r < P_{\text{accept}}$.

Both our MD and MC simulations use the Lennard-Jones potential for our energy calculation step. The Lennard-Jones potential is a widely-used mathematical model that describes the interaction between a pair of neutral atoms or molecules. The potential is characterized by two main components: a repulsive term that prevents atoms from coming too close to each other and an attractive term that accounts for the van der Waals forces at longer ranges. The equation for the Lennard-Jones potential is given by:

$$V_{LJ}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{3}$$

where:

- $V_{LJ}(r)$ is the potential energy as a function of the distance between two atoms.
- ε is the depth of the potential well, which indicates the strength of the attractive interaction.
- σ is the finite distance at which the interparticle potential is zero (also known as the collision diameter).

The first term $(\frac{\sigma}{r})^{12}$ represents the repulsive part of the potential, arising due to Pauli repulsion at short distances. The second term $(\frac{\sigma}{r})^{6}$ represents the attractive part, accounting for the long-range van der Waals forces as shown in **Figure 2**.

The simplicity and computational efficiency of the Lennard-Jones potential make it an excellent choice for simulating non-bonded interactions in various materials and molecular systems, e.g. the Argon system which we are simulating. By accurately capturing both the repulsive and attractive forces between particles, the Lennard-Jones potential is instrumental in determining the structural, thermodynamic, and dynamic properties of our nonreactive Argon system.

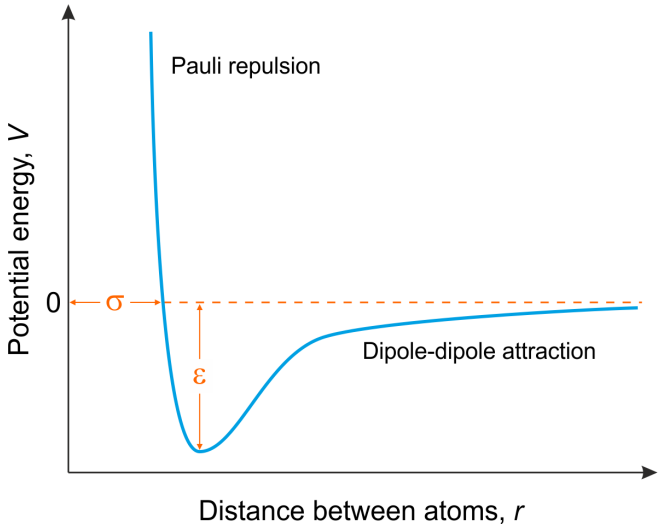

Figure 2: Plot showing the Lennard-Jones potential showing the Pauli repulsion (steep rise at short distances) and dipole-dipole attraction (smooth decline at longer distances). The x-axis represents the distance between two atoms, while the y-axis represents the potential energy. The equilibrium point, where the potential energy is minimum, indicates the balance between the repulsive and attractive forces.

3.2 Goals

These goals match learning objectives of the course and show proper understanding and usage of materials covered in the course

- **Modularity (Goal 1)**: By writing an object-oriented code for the MCMC simulation.
- **Parallelization of MCMC using OpenMP (Goal 2)**: By implementing parallel processing using OpenMP, which will allow the distribution of simulation tasks across multiple threads.
- **Parallelization of MCMC using MPI (Goal 3)**: By implementing parallel processing using MPI, which will allow the distribution of simulation tasks across multiple nodes and processors.
- **Parallel performance analysis of MD (Goal 4)**: By using existing MPI parallelization of CPU in an open-source MD Package(LAMMPS).

4. Methodology

4.1 Markov Chain Monte Carlo (MCMC) Simulation

The Monte Carlo simulation for Argon atoms was implemented within the canonical ensemble (NVT), where the number of atoms (N), volume (V), and temperature (T) remain fixed. The

simulation employs the Markov Chain Monte Carlo (MCMC) approach with the Metropolis-Hastings algorithm to sample configurations based on the Boltzmann distribution.

4.1.1 Simulation Steps

The simulation begins with an initialization phase, where system parameters, including the Lennard-Jones potential parameters ϵ (interaction strength) and σ (collision diameter), are defined. The initial atomic configuration is generated on a simple cubic lattice, ensuring that the initial density corresponds to the desired value. The system energy is calculated using the Lennard-Jones potential as shown in **Equation 3**.

The main simulation loop proceeds iteratively as shown in **Figure 3**, where in each step:

1. A random atom is selected for a trial move.
2. A trial displacement is generated, displacing the atom within a maximum allowable distance. Periodic boundary conditions are applied to ensure the atom remains within the simulation box.
3. The energy change due to the trial move, $\Delta E = E(C') - E(C)$, is computed using the Lennard-Jones potential.
4. The trial move is accepted with probability:

$$P_{\text{accept}} = \min\left(1, e^{-\Delta E/k_B T}\right). \tag{4}$$

   If the move is rejected, the atom remains in its original position.

After an equilibration phase, during which the system is allowed to reach thermal equilibrium, data collection begins. The simulation periodically outputs the total energy, pressure, and atom configurations, enabling analysis of the system's thermodynamic properties.

4.1.2 Equilibration and Sampling

The equilibration period ensures that the system reaches a steady state, minimizing the influence of the initial configuration. Following equilibration, the simulation collects ensemble averages for properties such as pressure, energy, and specific heat. These quantities are analyzed to assess the accuracy and efficiency of the MCMC simulation.

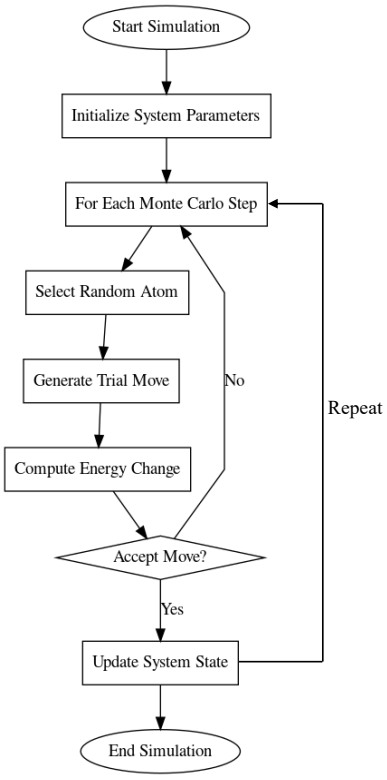

Figure 3: Flowchart of Metropolis Hasting algorithm of Markov Chain Monte Carlo approach.

4.2 Molecular Dynamics (MD) Simulation

Molecular Dynamics is a deterministic method that integrates Newton's equations of motion for each particle in the system. Given the initial positions and velocities of the particles, MD computes the evolution of the system by numerically solving:

$$m_i \frac{d^2 \mathbf{r}_i}{dt^2} = \mathbf{F}_i$$

where m_i is the mass of particle i, $\mathbf{r}_i$ is its position, and $\mathbf{F}_i$ is the net force acting on it, which is derived from the interatomic potential, such as the Lennard-Jones potential shown in **Equation 3**. **Figure 1** shows the process of running an MD simulation.

MD simulations are typically used to explore the time evolution of systems, allowing us to calculate time-dependent properties such as the Mean Squared Displacement (MSD) and the diffusion coefficient. Additionally, the conservation of total energy in MD simulations makes it a suitable method for studying thermodynamic properties at equilibrium.

5. Description of work performed

5.1 Goal 1: Object-Oriented Code for MCMC

The simulation code was designed using object-oriented principles to enhance modularity, maintainability, and reusability. The implementation centers around two main classes: `system_coordinates` and `LJ_model`.

The `system_coordinates` class manages the initialization, e.g., atomic positions, simulation box dimensions, and trajectory file output. Its primary methods include `generate_coords`, which initializes the atomic configuration on a lattice, and `write_frame`, which outputs the system configuration in the LAMMPS trajectory format for postprocessing.

The `LJ_model` class implements the Lennard-Jones potential, including energy and force calculations. The method `get_eij` calculates the pairwise potential energy, while `get_fij` computes the forces between two atoms. The method `get_single_particle_contributions` calculates the energy and force contributions of a single atom with respect to all other atoms in the system.

At last, the main function of the code includes the core Markov Chain Monte Carlo steps iteration and usage of the corresponding aforementioned classes. The UML Diagram of the code can be seen in **Figure 4**.

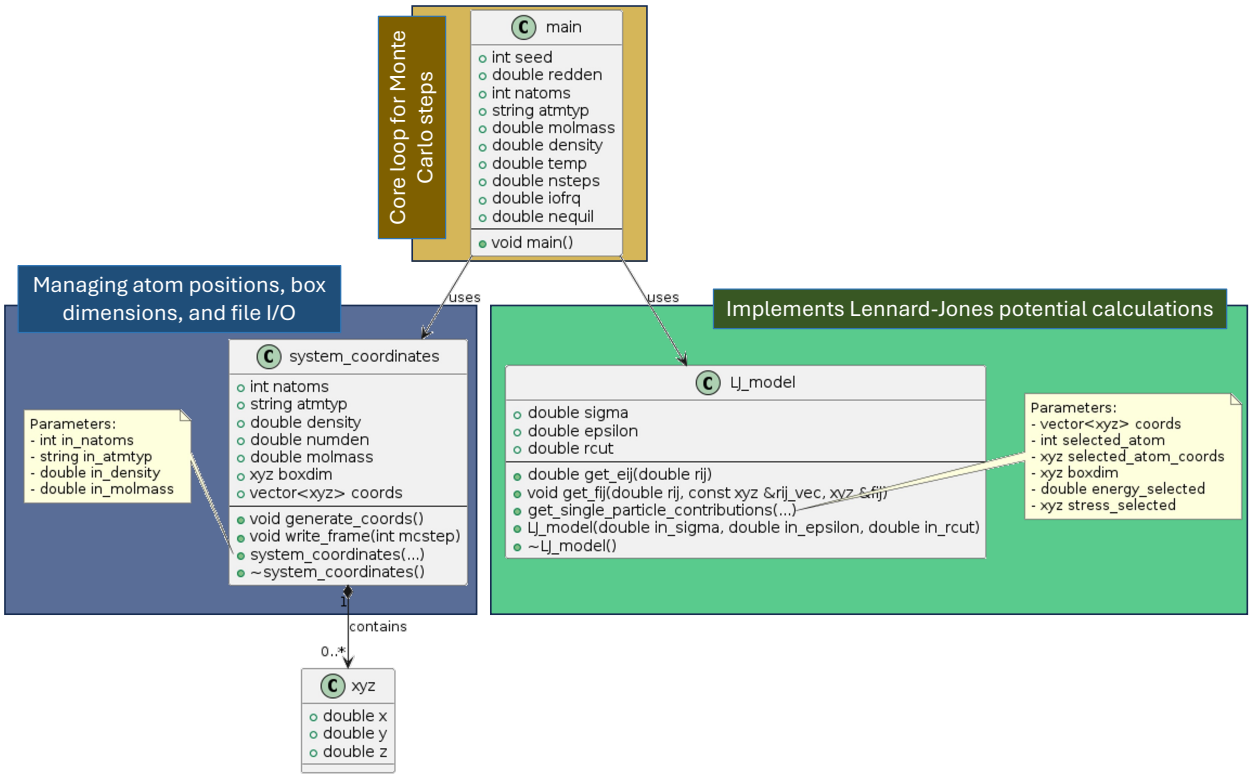

Figure 4: UML diagram of the object oriented code of Metropolis Hasting algorithm of Markov Chain Monte Carlo approach.

This modular design separates the responsibilities of managing system state and performing physical calculations, improving the clarity and flexibility of the code.

5.2 Goal 2: Parallelization of MCMC using OpenMP

The MCMC simulation is computationally intensive due to the pairwise interactions required to evaluate the Lennard-Jones potential. To accelerate these computations, OpenMP was used to parallelize two critical sections of the code.

The first parallelized section is the initialization phase, where pairwise energy and pressure are calculated for the initial configuration. This step involves a nested loop over all atom pairs. OpenMP directives (`#pragma omp parallel for`) were used to distribute the outer loop across threads, with a `reduction` clause ensuring thread-safe accumulation of the total energy and stress tensor. Dynamic scheduling was chosen to balance workloads effectively, as the cost of computations varies based on the interatomic distances.

The second parallelized section involves the method `get_single_particle_contributions`, where the energy and forces for a single atom are computed. Here, OpenMP parallelized the loop over all other atoms, with reductions applied to energy and force components. Static scheduling was chosen to provide a good balance between load distribution and overhead reduction.

5.3 Goal 3: Parallelization of MCMC using MPI

We seek to parallelize our MCMC code using both OpenMP and MPI to compare their efficiency. Prior to completing the code for this project, we hypothesized that splitting the atoms between many threads would become memory-intensive as we increased the number of atoms. For this reason, we determined that it would be valuable to parallelize via both methods. While OpenMP has shared memory with easy communication between threads, MPI requires explicit communication without this convenient shared memory. Additionally, messaging time and the nature of messaging can be altered for optimal computation.

To parallelize the code using MPI, we have two primary tasks to accomplish related to the memory and communication challenges of this system. First, the domain must be accurately distributed to the many processors, and the moved atom must correspondingly have its information distributed to each processor. Second, the atom that we move must be selected randomly, random not just within a processor but among the entire system for the sake of matching the underlying thermodynamic assumptions of Monte Carlo simulations.

For the first task, we divide our domain of atoms equally among all ranks. We define a system with equivalent size for each local system with appropriate dimensions and save the global number of atoms as an integer on each processor. When one atom is moved on the "owner" rank, we broadcast the old coordinates and new coordinates of the associated atom for energy calculations on each subsystem. This allows us to have each processor calculate only a subset of the atoms, a load-unbalanced partitioning that parallelizes the problem

evenly.

For the second task, we seek a truly random selection of an atom within our entire system, rather than any processor's subsystem. In this implementation, we apply the Mersenne Twister algorithm on the initializing rank to identify an owner system, and broadcast this value. Then, we randomly select an atom on that subsystem to displace and broadcast for energy calculation. This is likely not the fastest implementation, but the nested random selection makes for an appropriate selection satisfying the underlying thermodynamic assumptions of Monte Carlo simulations.

5.4 Goal 4: Implementation of LAMMPS

We implemented LAMMPS, an open-source package for solving MD for our system of Argon atoms. We used CPU parallelization to execute this code, solving for reduced pressure values as a function of reduced density as described in the Results section. The Great Lakes implementation is not configured for GPUs, but we were satisfied with the highly effective parallelization of this problem on CPUs. We also executed a strong scaling study and a weak scaling study for this code, to qualitatively compare with our MCMC code along with appropriate context for MCMC convergence.

6. Results and Discussion

6.1 Object-Oriented Code Design Justification

The overall goals of our steps to make the code more object-oriented were to increase modularity, maintainability, and reusability. The most repetitive section of the code involves iterating across our system class and calculating system energy, so our design choices center around two main classes: `system_coordinates` and `LJ_model`. By developing more modular classes with better encapsulated data, we are able to interpret and change more about how the code works through many fewer lines of code than before. Clarify and flexibility are greatly improved through this organization. There are some further improvements that could be made to simplify the outermost interface, but ultimately the parallelization schemes for OpenMP and MPI are made simpler and more interpretable through the organizational work of applying these object-oriented principles to our code.

6.2 Verification (pressure vs. density)

This section shows comparisons between the pressure vs. density relationship for Argon predicted by the three codes we used: LAMMPS Molecular Dynamics, and our Monte-Carlo code with OpenMP or MPI parallelism. This comparison is aimed at verifying our code implementations by comparing them to the ground truth of MD.

As shown in **Figure 5** reduced pressure has an increasing trend by increasing the reduced density. The results of Monte-Carlo with OpenMP and MPI are in good agreement with

the LAMMPS Molecular Dynamics results in the reduced density range of $\rho^* < 0.5$ and $\rho^* > 0.7$. However, there is a slight discrepancy in the range of $0.5 < \rho^* < 0.7$. The corresponding discrepancy is due to fluctuations during the phase transition. In this range, a phase transition occurs within the system, meaning that the system transitions from gas-like behavior to fluid-like behavior, and there will be higher fluctuations in properties like reduced pressure during the phase transition, which causes the discrepancy between the LAMMPS Molecular Dynamics and Monte-Carlo results. However, if the simulations were repeated several times, the results of the Monte-Carlo simulation would fall within the error bars of the Molecular Dynamics results in the corresponding range.

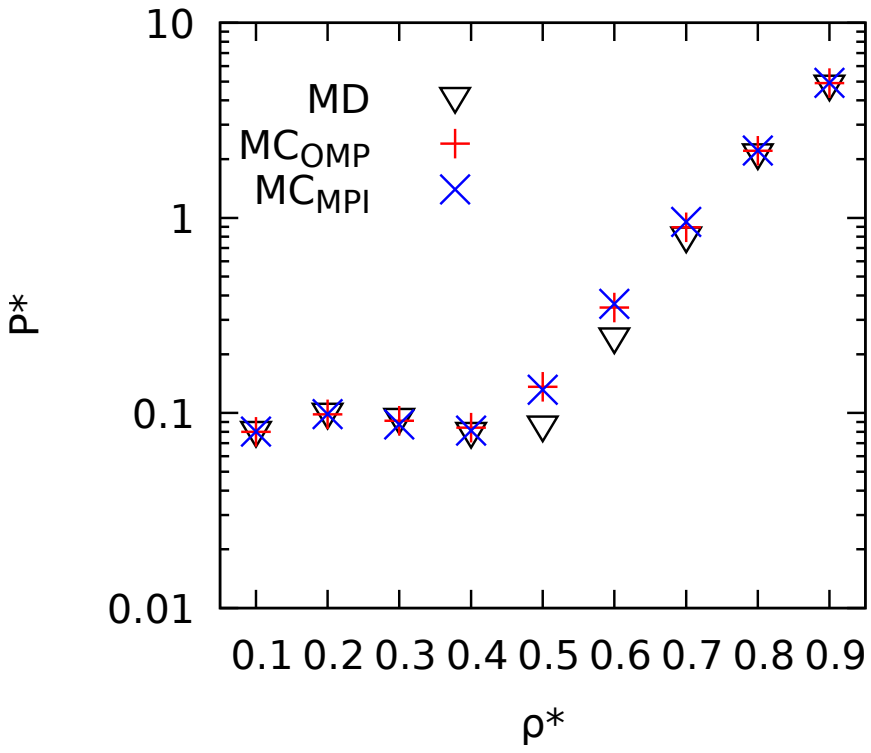

Figure 5: Reduced pressure as a function of reduced density. In this figure, MD results are treated as the ground truth and our MCMC results are matching our ground truth.

6.3 Parallel scaling studies

6.3.1 Strong scaling of OpenMP Monte-Carlo

OpenMP parallelism grants better performance than the serial code, and we see speedup for additional threads until we reach 16 threads, where our efficiency is at 27%. This indicates that our OpenMP implementation improves the speed of the code until we reach relatively few atoms per processor.

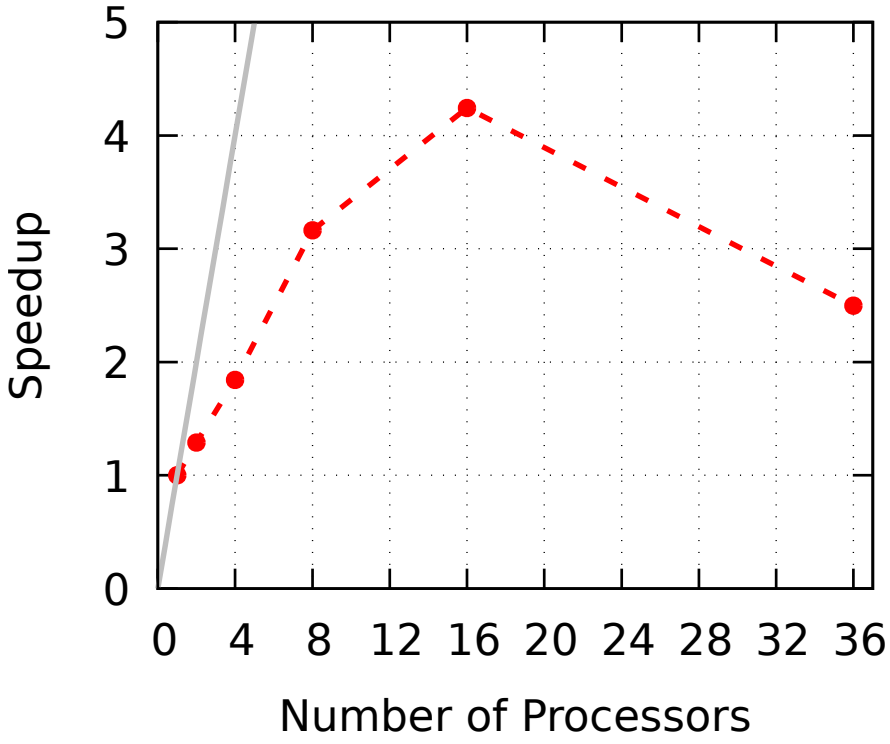

Figure 6: Strong scaling of OpenMP implementation of Monte-Carlo.

6.3.2 Strong scaling of MPI Monte-Carlo

MPI parallelism grants even improved performance over the OpenMP implementation. The communication schemes and lower-level programming grant up to six times speedup in the case of using multiple threads on two nodes until performance gains falter. For the problem of 4000 atoms, it is valuable to use MPI and performance gains can be developed relative to OpenMP, indicating that the problem tends to be memory-limited.

6.3.3 Weak scaling of MPI Monte-Carlo

While strong scaling tests how much speedup parallelism could provide for a fixed problem size, weak scaling tests how much the performance degrades as parallelism increases but the problem size per processor is constant. In theory, the computational work that each processor does should be constant, but the communication overhead with other processors increases, slowing down the computation. We chose to set the work to 256 atoms per processor, because in the strong scaling studies, that corresponds to around 30% efficiency. We chose the numbers of atoms and processor cores to be easily achievable using the initialization options provided in LAMMPS.

The iterations/second is plotted against the number of processors below. Contrary to expectations, the speed of iterations increases as the number of processors is increased. This

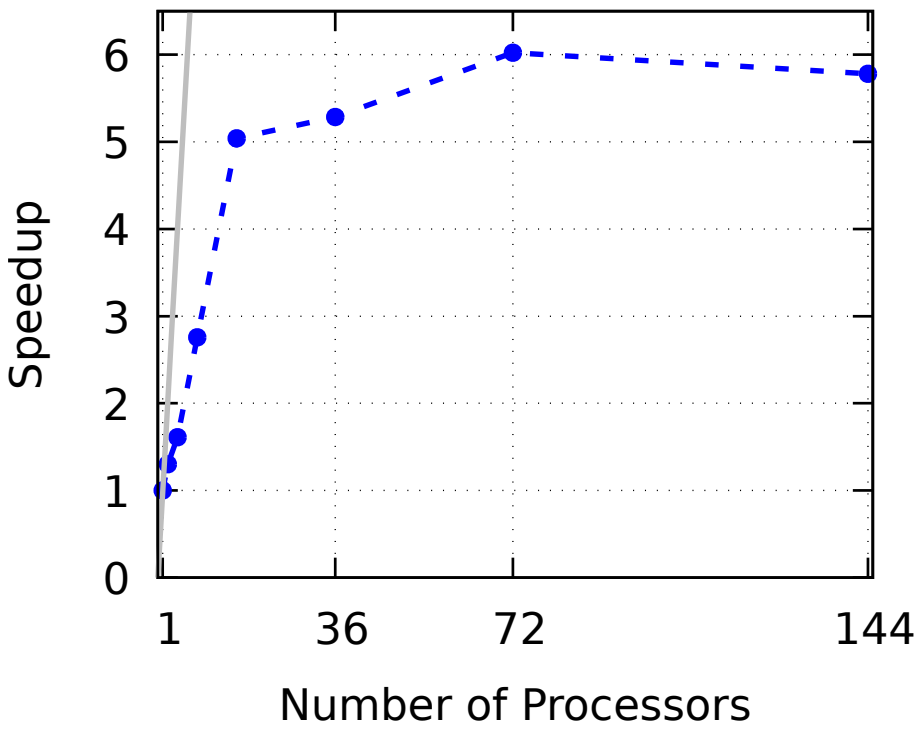

Figure 7: Strong scaling of MPI implementation of Monte-Carlo.

happens because the Monte-Carlo code only calculates the forces and energies between atoms that are close together. As the number of atoms increases and the domain grows, atoms are further apart on average. This means that a smaller proportion of the atoms are close enough to the selected atom that their energy needs to be computed. Since the atoms are randomly distributed among processors, as the number of atoms grows, the work done by each processor actually decreases, explaining the better-than-ideal weak scaling. If we were to test at even higher numbers of processors, we would see the iterations/second continue decreasing.

6.3.4 Strong scaling of LAMMPS

We conducted a strong scaling study of the Molecular Dynamics code LAMMPS to see how finely it could be parallelized. We kept the total atoms at 4000 and increased the number of processors. The speedup vs. the number of processors is shown in **Figure 9**, with the ideal behavior as the solid gray line. The efficiency is better than either of our implementations, with ∼30% efficiency even at 144 cores.

6.3.5 Weak scaling of LAMMPS

We conducted a weak scaling study of LAMMPS, to see how well it would scale to highly parallel simulations of large systems. As in the MPI weak scaling study, we used 256 atoms

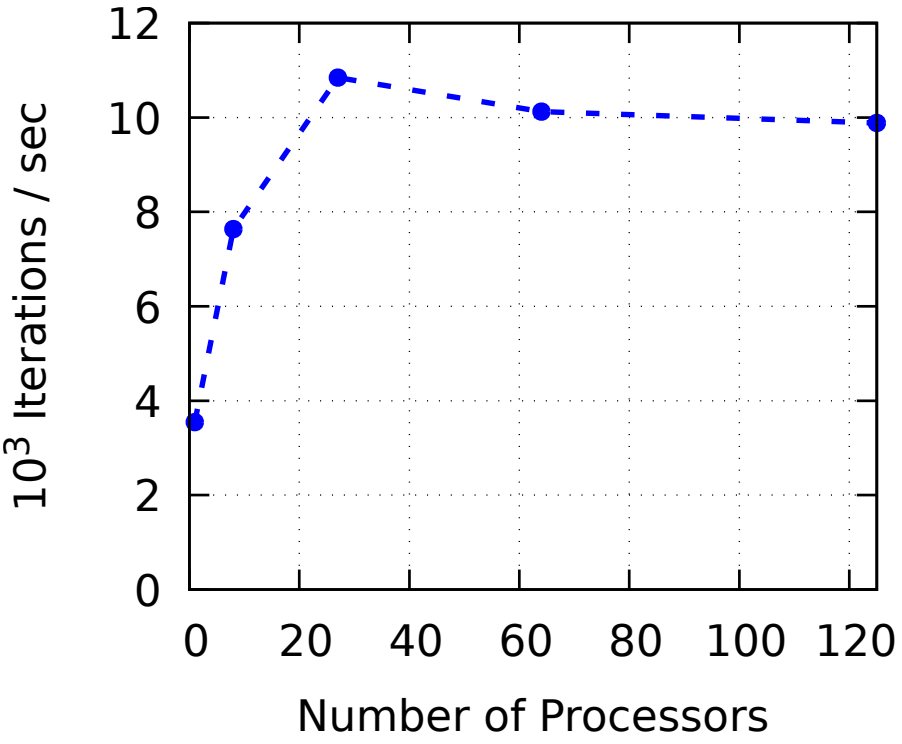

Figure 8: Weak scaling of MPI Implementation of Monte-Carlo.

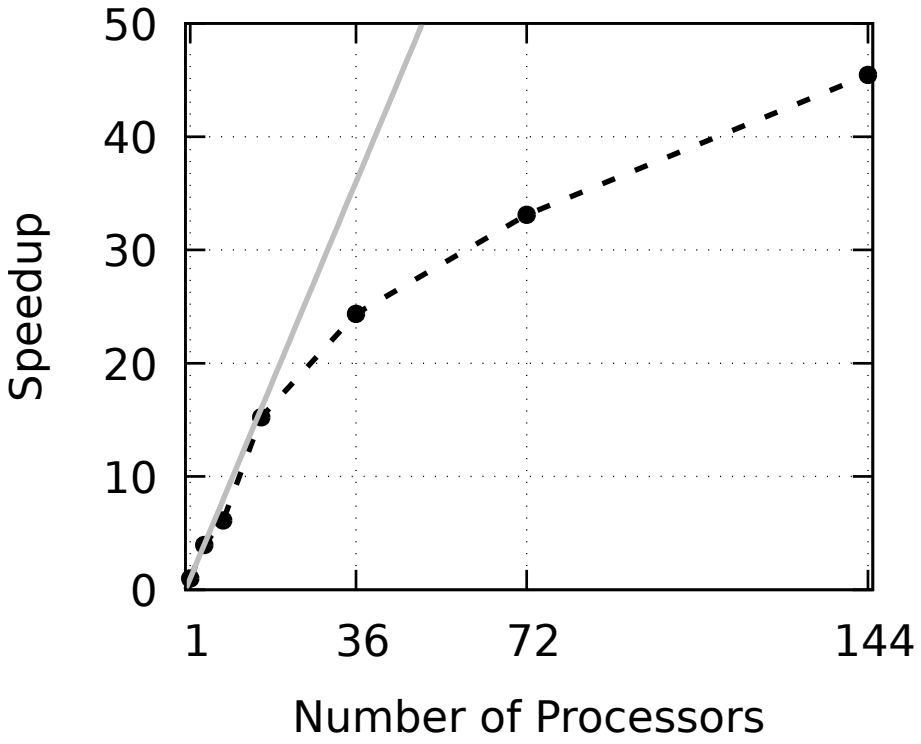

Figure 9: Strong scaling of LAMMPS Implementation of Molecular Dynamics.

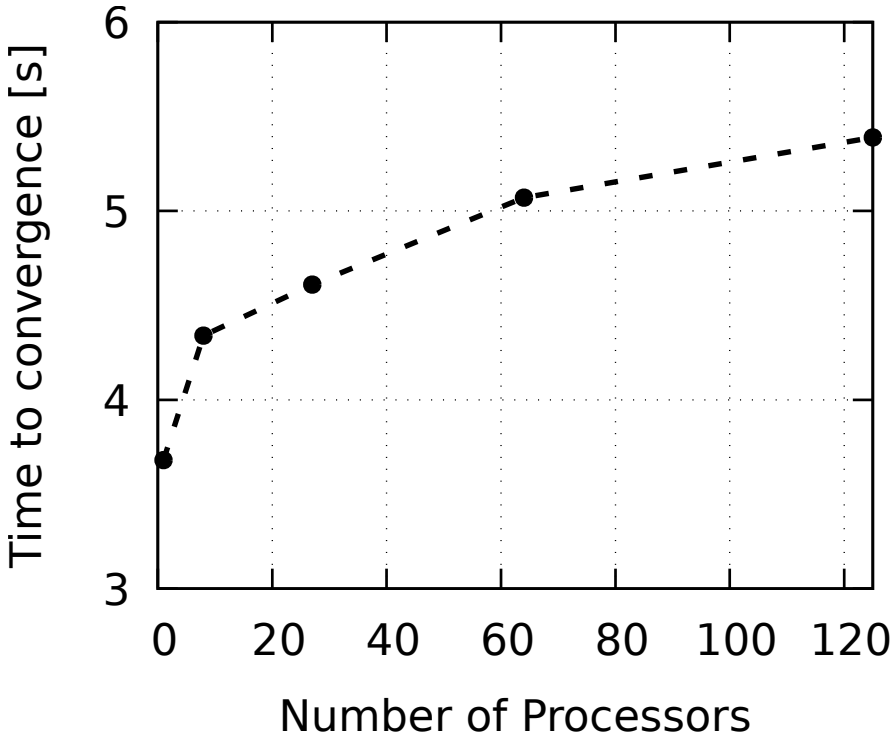

Figure 10: Weak scaling of LAMMPS Implementation of Molecular Dynamics.

per processor. LAMMPS sets the initial condition based on a replicated crystal structure, which means that the total number of atoms has discrete possible values. This became a challenge of how to choose the numbers of atoms and processors to keep the atoms per processor constant. We did the math in a spreadsheet and found the combinations of crystal replications and numbers of processors that kept the atoms per processor at 256, which is why the numbers of processors we tested are seemingly irregular.

The time to convergence vs. the number of processors is shown in **Figure 10** for atoms per processor at 256. The time to convergence increases more than 40% when scaling up to 120 processors, which is considerably worse than expected. A recent paper documenting LAMMPS tests the weak scaling and shows excellent weak scaling up to thousands of cores [4]. The key difference is, that paper uses $14,000$ atoms per processor, while we use 256. That means the simulation was much more arithmetically intense for the scaling tests in the documentation paper, so the relative amount of communication required was lower in the paper's tests. That difference explains why we see non-ideal weak scaling even at only a few hundred processors.

Table 1: Time to convergence for different numbers of atoms

Number of Atoms (atoms/processor=256)	**256**	**2048**	**6912**	**16384**	**32000**
MC Time to Convergence [s]	18.59	48.73	452.33	Did not converge	
MD Time to Convergence [s]	3.68	4.34	4.61	5.07	5.39

6.4 Molecular dynamics vs. Monte-Carlo performance comparison

Comparing the weak scaling results of the MPI Monte-Carlo code and LAMMPS would suggest that our Monte-Carlo code scales more efficiently. However, for the weak scaling studies, we timed a constant number of iterations, while the meaningful metric is the time to reach equilibrium. Though MD simulations can look at non-equilibrium scenarios, MC can only give equilibrium results. So to compare apples-to-apples, we compare the time each method takes to reach equilibrium and compare those in a weak-scaling context.

Because neither code has automatic equilibrium detection, we run the codes for a fixed number of iterations and time the whole execution, then visually detect equilibrium by looking at plots against iterations. This visual detection is inexact and somewhat arbitrary, but when looking at plots of quantities of interest against iterations, it is generally quite clear where to mark convergence. From there, we assume all iterations take the same amount of time and calculate the time to convergence.

We found that the Molecular Dynamics simulation converged at around $10,000$ iterations regardless of the number of atoms. This is expected because each MD iteration updates the positions of all of the atoms. On the other hand, the the Monte-Carlo simulation took more iterations to converge for larger numbers of atoms. This is also expected, because each MC iteration updates only one atom at a time.

The times to convergence of LAMMPS and our MPI Monte-Carlo code are compared in table 1. Note that we ran the Monte-Carlo code to a maximum of $5,000,000$ iterations, and if we ran it for a greater number of iterations the cases that did not converge should eventually converge. This table shows that the true weak scaling of the MPI Monte-Carlo code is very poor, due to the algorithm only updating one atom at a time.

7. Conclusion

This project was an investigation into the parallel performance of Monte-Carlo and Molecular-Dynamics simulation methods. We implemented our own Monte-Carlo code and parallelized it with OpenMP and MPI, and compared with an open-source Molecular Dynamics code. Our implementations of OpenMP and MPI had decent ($\sim$30% efficiency) strong scaling for our target problem (4000 atoms) up to about half a node on Great Lakes, while LAMMPS had decent strong scaling up to 4 nodes. The weak scaling of our MPI implementation was better-than-ideal up through 3 nodes on Great Lakes, which was seemingly better than LAMMPS's weak scaling. This was partially explained by the true work per processor de-

creasing in our weak scaling, due to only computing local pairwise interactions. Additionally, because of the difference in Monte-Carlo and Molecular Dynamics algorithms, the former took more iterations to converge for larger systems, while the latter does not. This resulted in LAMMPS outperforming our MPI Monte-Carlo implementation in weak scaling of compute-time-to-equilibrium. In fact, LAMMPS outperformed both of our implementations in absolute terms in addition to scaling better. This was contrary to expectations, because Molecular Dynamics can be used to study more complicated phenomena, while Monte-Carlo can only collect equilibrium statistics. However, our strict adherence to the "perturb one atom at a time" formalism of Monte-Carlo, lead to an unavoidable scaling in the convergence time with the system size. Finally, because LAMMPS has been well-optimized by experts, and our code was written by non-experts and has not been optimized, it should not be surprising that LAMMPS outperforms for collecting equilibrium statistics. We also found that the weak scaling of LAMMPS was worse than the published result, because we used a much smaller problem size per processor than the LAMMPS team did when measuring the weak scaling.

Through rewriting the code in an object-oriented manner, we were able to better compartmentalize different processes and identify important sections to parallelize. The new classes allow for much better control over the flow of information in the code, and results in a more understandable code where functions changed once can be effective in changing the overall structure. The function calls are organized such that programmers can understand inputs and outputs, which was very helpful in our parallelization steps, especially in the case of MPI. This illustrates why object-oriented coding is helpful, and also why it can be challenging.

By parallelizing the MCMC code using OpenMP, we learned that OpenMP is simpler to implement for our problem, and that we can still see performance gains relative to our serial code. In our case, we ran a low enough number of atoms such that reasonable speedup and efficiency can be seen. We found that OpenMP underperformed MPI both in terms of scaling and in terms of absolute time, and we don't have a good explanation for why this happened. Our takeaway here is that though OpenMP is easier to implement, it can be hard to see what's going on behind the scenes. This makes it difficult to diagnose where performance bottlenecks are, and consequently it can be hard to make it efficient.

By parallelizing the MCMC code using MPI, we learned that our problem could be classified as semi-memory limited. Our scaling study indicates that working with the explicit data sharing of MPI allows for efficient parallelization, even if not to the level of professionally designed codes. It also allowed us to push the weak scaling study to include multiple nodes. We learned that testing weak scaling can be hard because of confounding factors of the underlying algorithm. For example, while scaling the size of the system, the true work per processor for the Monte-Carlo algorithm decreased, and the iterations to convergence increased. These two confounding factors confused the results of the weak scaling study, showing the importance of considering the algorithm while performing scaling tests.

Through our LAMMPS implementation, we learned about the differences between the two methods and how convergence plays a significant role in our understanding of computational speed to our solution. Since the MD method moves all of the atoms in the system simultaneously, convergence speed was largely independent of the system size.

One future optimization would be finding a random number across multiple cores using seed-based random number generation rather than communicating the value to many cores. This would likely improve the code's speed since this calculation is performed for every iteration, meaning thousands of times per minute of runtime. Another optimization that we would be interested in doing is pursuing load balancing algorithms in our parallel code. This would ensure that cores do not have too few atoms within the distance calculation window, and it would preserve the computational efficiency of the windowed energy calculation rather than just computing energies for every atom. Lastly, we performed a general time study of our serial code, but did not perform a profiling of our parallel code. Such a profiling operation would ensure that we take advantage of all simple optimizations for our parallel code prior to shrinking communication, implementing non-blocking communication, and pursuing further, more complicated parallel optimization strategies.

References

[1] ScienceDirect, *Lennard-jones potential - an overview*, https://www.sciencedirect.com/topics/physics-and-astronomy/lennard-jones-potential, Accessed: 2024.

[2] D. Frenkel and B. Smit, *Understanding molecular simulation: From algorithms to applications*. Academic Press, 2002.

[3] M. S. Shell, *Thermodynamics and Statistical Mechanics: An Integrated Approach* (Cambridge Series in Chemical Engineering). Cambridge University Press, 2015.

[4] A. P. Thompson, H. M. Aktulga, R. Berger, *et al.*, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108 171, 2022. DOI: 10.1016/j.cpc.2021.108171.

Appendices

8. Code on GitHub

Code for this project can be accessed on GitHub at https://github.com/jsayut1/molecular_dynamics_scicomp/tree/main. Switch to branches labeled "OpenMP" or "MPI" to examine the code for those respective implementations.