Parallization of Molecular Dynamics Simulations

Ivan Otero, Harrison Van Til

24-623: Molecular Simulation of Materials
Final Project Report
Carnegie Mellon University
December 15, 2017

1 Abstract

In the assignments for class, the simulations were run by one core. Most computers have more than one core at their disposal. Parallel computing is investigated in order to make molecular dynamics simulations run more quickly by using more than one core. Limitations to parallel computing are also investigated and discussed. Three different algorithms are explored to run the simulations in parallel: force decomposition, atom decomposition, and spatial decomposition. The force decomposition method was implemented in an NVT ensemble and run on a Intel Xeon processor. Simulations were run using a variety of system sizes in order to see the scalability of the force decomposition method. The largest system size of 2048 atoms was determined to have 92.77% parallelization efficiency, and a runtime speedup of 3.32 was seen running on four cores compared to one.

2 Introduction

Molecular dynamics (MD) is a widely used method of computer simulation for studying the physical movements of atoms and molecules. Various properties of complex liquids, solids, and molecules that would be impossible to determine analytically can be estimated using MD simulations. Originally developed in the mid-1950s, it has been refined and successfully applied to many areas of science, including physics, materials science, and biochemistry. In a typical MD simulation, the positions and velocities of the atoms or molecules in the system are evolved over time using Newtons equations of motion. More specifically, a potential energy function is used for the system, from which the individual forces on each atom are derived.

While molecular dynamics is a powerful method of simulation, it does have its limitations. In general, the force on each atom depends on the positions of every other atom, and thus the computational cost of MD simulations scales as $O(N^2)$. This leads to simulations quickly becoming intractable, as systems of interest can be quite large. Spatially, the size of system that can be feasibly simulated is limited by the length scale of atoms, which on the order of Angstroms. Simulating a system of hundreds of nanometers in all 3-dimensions would require many millions of atoms [2]. Temporally, the timesteps used in MD simulations are limited by the frequency of atomic vibrations, which are on the order of femtoseconds. Therefore, to simulate picoseconds of time requires tens or hundreds of thousands of time steps [3]. Without computational improvements, investigating phenomena of interesting would quickly become intractable due to the $O(N^2)$ scaling of MD simulations.

Many algorithmic improvements have been developed to improve the computational efficiency of MD simulations. Most notably, the use of a cutoff radius and neighbor lists has been shown to reduce the scaling to $O(N^{3/2})$ [1]. However, one of the most significant computational improvements for MD simulations is parallel computing. This is largely due to the fact that MD simulations are inherently parallel. That is, the position and velocity of and force on each atom can be calculated independently of all other atoms within each timestep, and the motions of all atoms are only coupled when advancing to the next timestep. With well-designed parallel algorithms, this can lead to massive gains in computational efficiency. State-of-the-art simulations on 100,000+ core supercomputers can simulate systems with

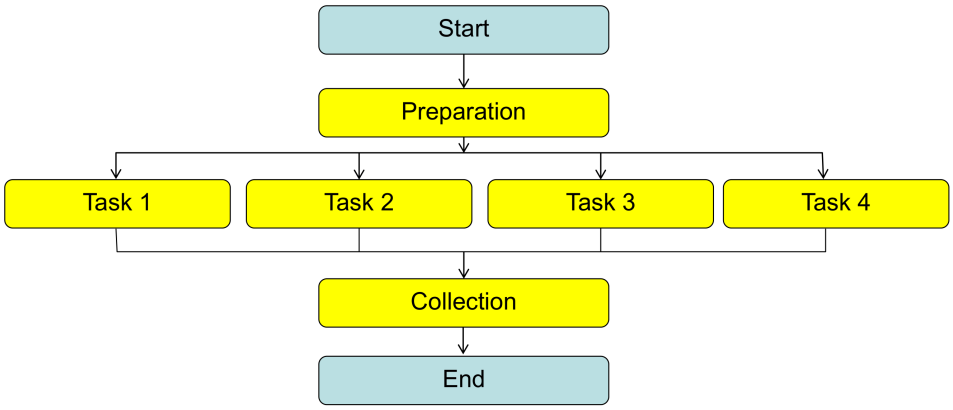

Figure 1: Parallel computing flowchart.

billions of atoms, enabling the study of micron-scale phenomena derived from atomic-level forces.

The idea behind parallel computing is to break up a main computation into several small tasks that can be run simultaneously. This process is called decomposition. In a process called preparation, each task is designated to a thread which is in charge of performing the operations for that task. Once each task has been completed, the information gathered from each task has to be collected in order to get the final result. A flowchart of parallel computing can be seen in Figure 1 to visualize the process.

Parallel software takes advantage of modern computer hardware with multiple cores. A core is an independent central processing unit (CPU) that is capable of carrying out a single instruction set at one time. Each instruction set sent to a CPU is called a thread. The main distinction between a thread and a core is that a thread is software and a core is hardware. Modern laptops, desktops, and smartphones all have multi-core cores capable of several parallel computations. Server clusters for running numerical simulations may have dozens or even hundreds of cores, and leading supercomputers have more than 100,000 cores.

By doing many calculations in parallel, a lot of time can be saved. There are some requirements that need to be meet for this to happen. The tasks should not be dependent on each other. If tasks depend on each other, a thread would wait for another thread to be finished before it can start. Tasks should also have a similar load. If they are not balanced, then there would be cores not being utilized while calculations are being performed. Even if these two conditions are met, there is still a limit to how much improvement one can gain from computing in parallel regardless of how many cores are being used. Amdahls law states that the improvement that will be obtained follows:

$$S_{latency}\left(c\right) = \frac{1}{(1-p) + \frac{p}{c}} \tag{1}$$

where $S_{latency}$ is the runtime improvement, c is the number of cores, and p is the percentage of the code that is being run in parallel. We can see the visualization of the equation in Figure 2. We see that all values of p eventually reach a plateau. Higher values of p plateau to a higher $S_{latency}$, and it takes more cores to reach that plateau. The $(1-p)$ term in the

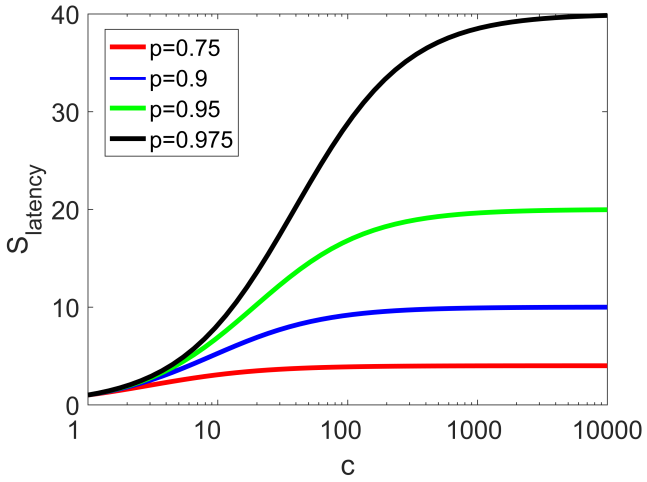

Figure 2: Theoretical speedup of parallel programs, according to Amdahl's Law.

denominator of the equation is what causes the plateau in the graphs. This is due to the preparation and collection phases in parallel codes. If it were not for those phases, then there would be infinite improvement when there is an infinite amount of cores.

3 Background

There are many approaches to parallelizing MD simulations that have been developed over the past two decades. While some interesting approaches simulate replica, uncoupled systems in parallel in order to access longer timescales [4], most parallel MD algorithms decompose a single, coupled system in order to access larger system sizes. It is important to note the distinction between algorithms which efficiently simulate larger system sizes versus algorithms which efficiently simulate longer timescales. For example, it is unlikely that a parallel algorithm designed to efficiently simulate a 1 million atom system for 10 thousand time steps will be able to simulate a 10 thousand atom system for 1 million timesteps with similar computational effort. This is due to the communication overhead and serial portions of the algorithm, as mentioned in the earlier discussion of Amdahls Law. In the present investigation, only algorithms which decompose a single, coupled system are considered, as these are more commonly used in practice. An overview of three parallel algorithms is presented: force decomposition, atom decomposition, and spatial decomposition. A high-level overview of each algorithm is discussed, along with their relative benefits and drawbacks. Finally, a more detailed discussion of an implementation of the force decomposition algorithm is presented, along with simulation results and comparison to theory and single-core benchmarks.

In classical MD simulations, the force on each atom is determined using a pairwise potential function such as the Lennard-Jones potential, seen in Equation 2.

$$u\left(r\right) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} + \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2}$$

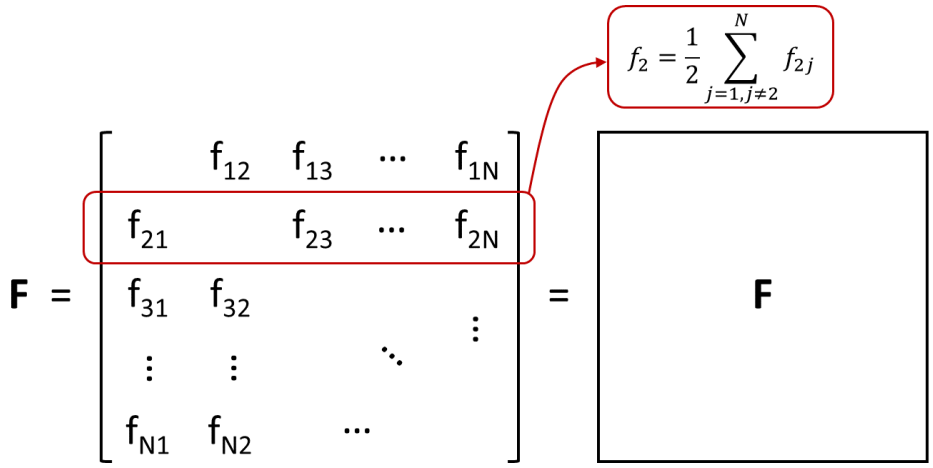

Figure 3: The force matrix F, which is helpful for visualizing the calculations required to compute the net force on each atom.

The force on each atom is then the sum over the contributions from every other atom. A useful construct for visualizing this calculation is to consider the $N \times N$ force matrix, seen in Figure 3. Note that this matrix is simply for visualization purposes, and in practice an $N \times N$ array is never explicitly stored in MD programs. Two of the three parallel algorithms discussed here, force decomposition and atom decomposition, essentially break this force matrix into smaller blocks to be computed by multiple cores simultaneously.

3.1 Force Decomposition Algorithm

The basic idea behind the force decomposition algorithm is to assign vertical blocks of the force matrix to each core, as seen in Figure 4. From a more physical standpoint, every core computes a partial force for every atom, and at the end of each time step this partial forces are combined to determine the total force on each atom. A more detailed explanation of the force decomposition algorithm follows.

First, the current positions of all atoms are communicated to all cores. Second, each core computes its elements of the force matrix, summing results into partial sums for each atom. Third, the partial sums from all cores are collected. Fourth, a single, central core computes the total force on all atoms by combining the partial sums from each core. Finally, the single, central core updates the positions and velocities of all atoms according to Newtons laws of motion, and the algorithm proceeds to the next time step which will start again from step one.

An important consideration with any parallel algorithm is load balancing. As discussed earlier, if the load on all cores is not well balanced, then many cores will sit idle waiting for a few cores to finish their tasks and communicate their results before the algorithm can proceed to the next step. With the force decomposition algorithm, the issue of load balancing is resolved by ensuring all vertical blocks of the force matrix have nearly equal numbers of non-zero elements.

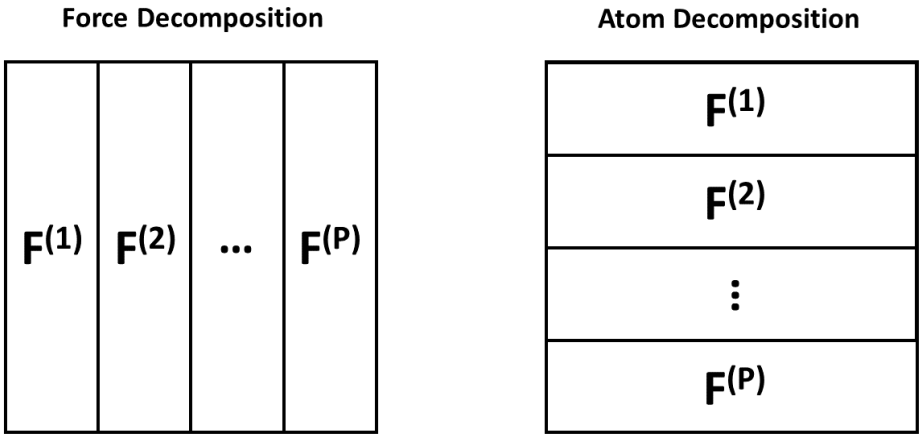

Figure 4: Schematic of how the force decomposition and atom decomposition algorithms break up the atomic force calculations. Left: the force decomposition algorithm assigns columns of the force matrix to each core. Right: the atom decomposition algorithm assigns rows of the force matrix to each core.

The major benefits of the force decomposition algorithm are that it is relatively easy to implement, and that immediate runtime improvements can be seen for small numbers of additional cores. For these reasons, the force decomposition algorithm was chosen to be implemented and studied in this investigation. For running simulations on personal computers, which typically have only two or four cores, a plateau of diminishing returns for runtime improvements will not be reached with this algorithm. However, the force decomposition algorithm requires global communication between all cores at each time step. As the number of cores is increased to dozens or hundreds, this communication overhead will limit the number of cores that can be used effectively. Other algorithms are generally used for massively parallel simulations beyond what personal computers are capable of.

3.2 Atom Decomposition Algorithm

The idea behind the atom decomposition method is to split up the problem by having a processor take a subset of the atoms, N/P, in the system. The processor will run the force calculations and velocity verlet calculations for its N/P atoms. At the end of each timestep, the processors will communicate the updated positions to the other processors. When calculating the net forces of the particles, the processors are taking rows of the force matrix as opposed to columns that was done in the force decomposition, as seen in Figure 4.

The atom decomposition follows a simple series of steps for each time step. Each processor receives a list of the atoms and acquires the positions of the other atoms that are not assigned to it. Then the processors calculate the net force on its N/P atoms. With the newly calculated forces, the processors update the position of their N/P atoms. At the end, the new positions are shared to all of the processors. These steps are repeated for as long as the simulation needs to run.

The main strength of this algorithm is its simplicity. It is a simple concept that does

not require a more detailed understanding of the system in order to implement. The main disadvantage is that global communication is required. Since every processor needs to know where all of the atoms are in the system, updated positions will have to be communicated at every time step. The communication scales with N and will not change regardless of how many processors are being used [2]. Because of this, there will be a limit to how many processors can be used effectively. Due to this limitation, the atom decomposition will most likely not be used in massive simulations where there are billions of atoms in the system.

3.3 Spatial Decomposition Algorithm

The idea behind the spatial decomposition method is to split up the system into different subdomains. Each processor would then be in charge of all calculations involving atoms in their subdomain. The processors will also need to know the positions of atoms that are within the cutoff distance of the subdomain. Figure 5 is an example of a system split up into nine subdomains which helps illustrate the decomposition. Each subdomain only needs to share and receive information with its surrounding subdomains. In a two-dimensional case, a subdomain will only interact with a maximum of eight subdomains. In a three-dimensional case, a subdomain will interact with a maximum of 26 other subdomains. This means that for large systems, communication between processors will be reduced. The system will scale as N/P instead of N which is necessary for massive systems.

Once the subdomains have been defined and assigned to the processors, the spatial decomposition algorithm follows a series of steps for each time step. The processor first acquires the positions of the atoms in neighboring subdomains. This requires the use of linked list since the neighboring atoms are not fixed. Once the processor has all of the necessary information, it will calculate the net forces of all of the atoms in its subdomain. With the new

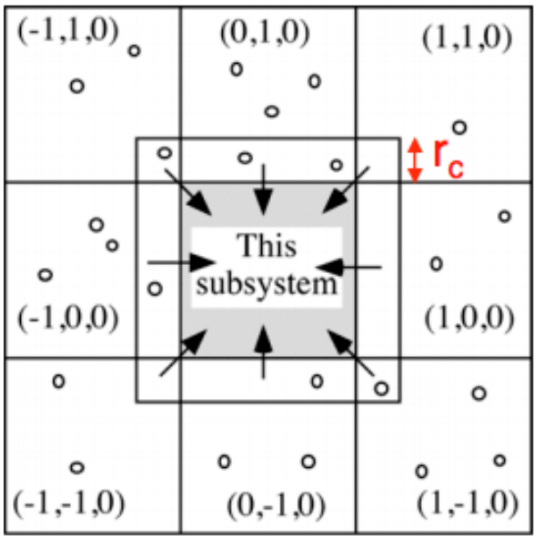

Figure 5: An example of a 2-D system broken up into 9 sub-domains, for the spatial decomposition algorithm.

net forces, the processor will then update the positions of the atoms. Lastly, the processor will communicate the new positions to the neighboring processors.

The main advantage of the spatial decomposition method is that it reduces communication between processors. This allows the runtime to scale better for massive systems. The scaling is dependent on a good load balance. This means that the subdomains need to be roughly the same size and need to have a similar number of atoms. These conditions will not be met if the system has a non-rectangular shape or if the system does not have a uniform density. It is important to have a good load balance in order to prevent processors having to stand idle while a single processor finishes its calculations. The main disadvantage of the spatial decomposition is the complexity in implementing it. There are a lot of details to keep track of and special programming techniques such as linked lists are necessary in order to handle the communication between processors. Due to the intricacy of the spatial decomposition, we decided to not attempt implementing it. If one wants to run an MD code using the spatial decomposition method, open source programs such as LAMMPS have built in functions that can do this for them.

4 Methodology

As expected, only a few changes to our original NVT code had to be made in order to implement the force decomposition method. The changes we made only corresponded to the force calculations. In our original code, we had one function called ComputeNetForces which would loop through all of the atoms in the system to calculate the net force acting on each atom. To take advantage of parallel computing, we wrote two functions: ComputeForces-pCores and ComputeForces-SubRoutine. We were unable to generalize the number of cores and make a variable for it in the code. We had to manually write ComputeForces-1Cores, ComputeForces-2Cores, ComputeForces-3Cores, and ComputeForces-4Cores in order to use up to four threads.

The ComputeForces-pCores functions all had the same structure. They prepare for parallel computing by assigning a different force array to each thread. The force arrays are

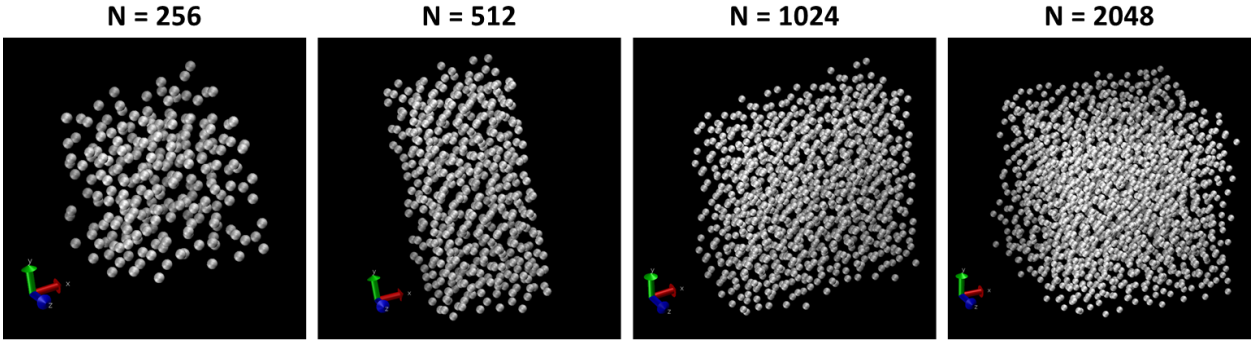

Figure 6: Visualizations of the systems simulated in this investigation.

used as a temporary placeholder for partial sums since different threads cannot access the same variables at the same time. Afterwards, it would call the ComputeForces-SubRoutine that would calculate partial forces. When the partial forces have been completed, all of the partial force arrays are summed together in the main force array. The main force array is then used in the velocity verlet function where the program will resume running in serial.

The ComputeForces-SubRoutine function runs the same way as the original ComputeNet-Forces function but it has two distinct differences. The first difference pertains to the calculations of the partial forces. If Newtons Third Law is taken into account, half of the elements in the force matrix are equal but opposite, and therefore do not need to be computed. Let F be the original force matrix where the third law is not taken into account, and let G be a load balanced force matrix that accounts for Newtons Third Law. The matrix G is defined by:

$$G_{i,j} = \begin{cases} 0 & (i > j) \ \& \ (i+j) \ is \ even \\ 0 & (i < j) \ \& \ (i+j) \ is \ odd \\ F_{i,j} & else \end{cases} \tag{3}$$

where i and j are the row and column numbers, respectively. The second difference pertains to the force calculation loops. In the original code, we used a nested set of for loops in order to check every atom with every other atom. The loops would run from $0 < i < N$ and $0 < j < N$. When running in parallel, we no longer want the loops to cover every single atom pair interaction. We only want to calculate the columns of the force matrix. The loops in the parallel code run from $0 < i < N$ and $Start < j < End$, where $Start$ and End are specific to the thread.

We used a variety of different systems when running our parallelized code. The first simulation that we ran was with the liquid256 system that we used in our previous assignments. In this case, we used the same parameters as in assignment number 4: $L_x = L_y = L_z = 6.8$, $R_c = 2.5$, thermostat time constant $\tau_T = 0.05$, and randomized initial velocities that conserve momentum. Where L is the side length of the system and R_c is the cutoff distance. We also ran the simulations for 250,000 time steps in order to make sure that the simulation took a substantial amount of time. By doing this, we hoped that we would be able to clearly see differences in runtime between different numbers of processors used. In order to run simulations with more atoms, we made copies of liquid256 and translated them by L. For liquid512, $L_x = L_z = 6.8$ and $L_y = 2 \times L_x$. The same process was used to create liquid1024 and liquid2048.

5 Results

The results of the force decomposition simulation program were verified against a known working single-core program. In general, a good check to verify that an MD simulation has run without errors is to plot the total system momentum over time: momentum should be conserved in all three directions on the order of 10^{-13} LJ units or lower. In Figure 7, we see that momentum is indeed conserved. Additionally in Figure 8, we see that the calculated values for temperature agree very well with the single-core simulation results. They are not exactly the same due to randomized initial velocities in every simulation run.

The raw runtimes and relative speedup for 1, 2, 3, and 4 cores at each system size are shown in Figure 9. As expected, the runtimes grew exponentially with larger systems sizes, and decreased when run on more cores. The plot of runtime speedup versus number of cores additionally shows that larger systems sizes are able to take better advantage of the increased number of cores.

Recalling Amdahls Law, the runtime data at each system size was used to find best fit values for P, the parallel portion of the code. The results are shown in Figure 10. After getting the best fit values, P was plugged back into Equation 1 and the theoretical equation for runtime speedup is shown along the experimental data points in Figure 10. We see that the percent of parallelized computation in each simulation increases as the system size

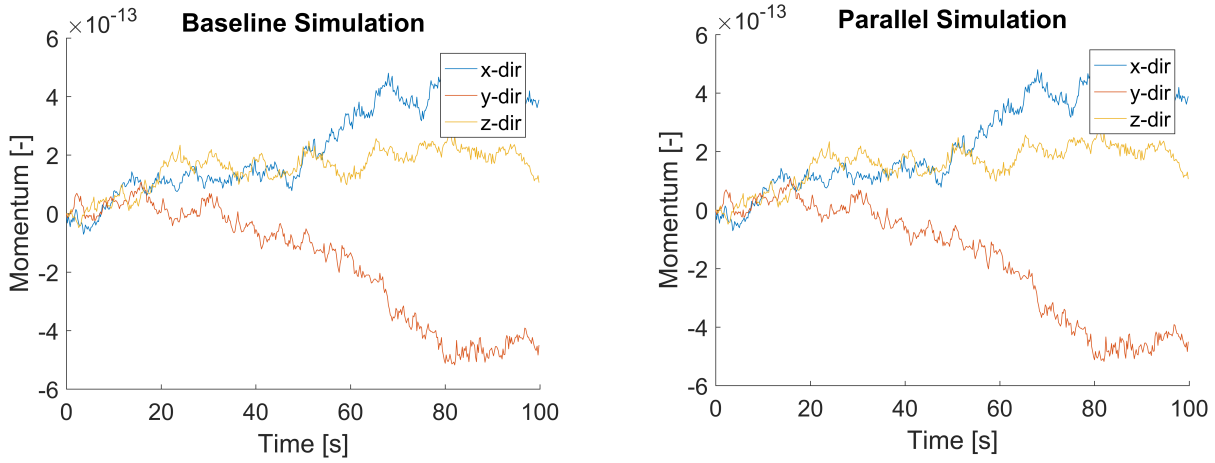

Figure 7: Validation of code, via momentum conservation. Left: simulation results for the baseline non-parallel code. Right: simulation results for the force decomposition parallel code.

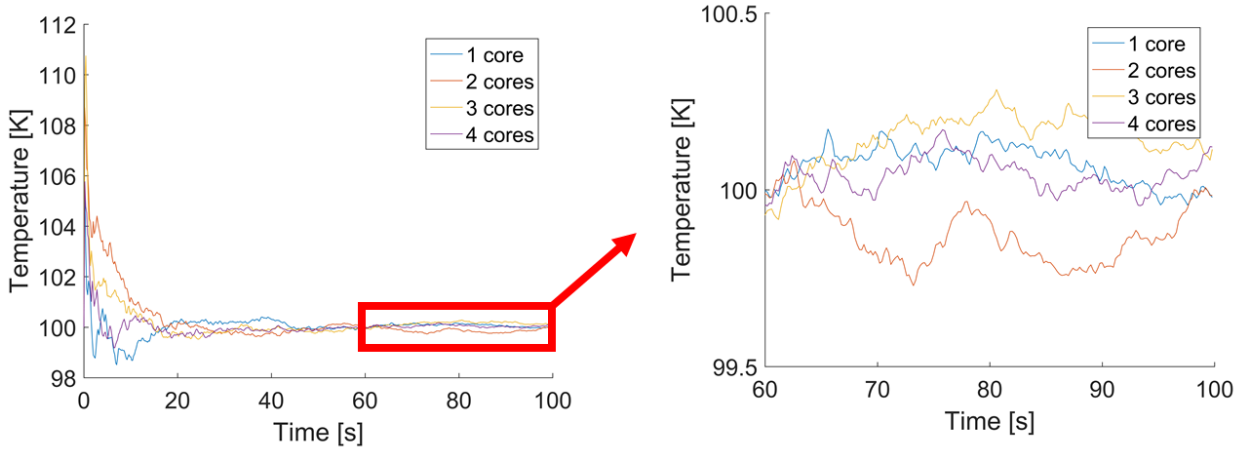

Figure 8: Validation that the code is properly simulating the NVT ensemble. A Nose-Hoover thermostat was set to T=100 K, and the results show the cumulative average temperatures approach this set value.

increases. With larger system sizes, it's expected that more computation time is spent inside the parallelized compute forces subroutine compared to the rest of the program, which is run serially. Additionally in Figure 10, the parallelized percentage appears to plateau short of 95% as the system size grows. This relates to the earlier discussion about the benefits and drawbacks of the various parallel algorithms for MD simulations, when it was noted that the force decomposition algorithm has a high communication cost relative to other algorithms.

To show this concept of diminsishing returns for using additional cores if the parallelization percentage is not near 100%, the best fit values for P from Amdahl's Law were used for extrapolation. In Table 1, the predicted runtime speedup is shown for each system size,

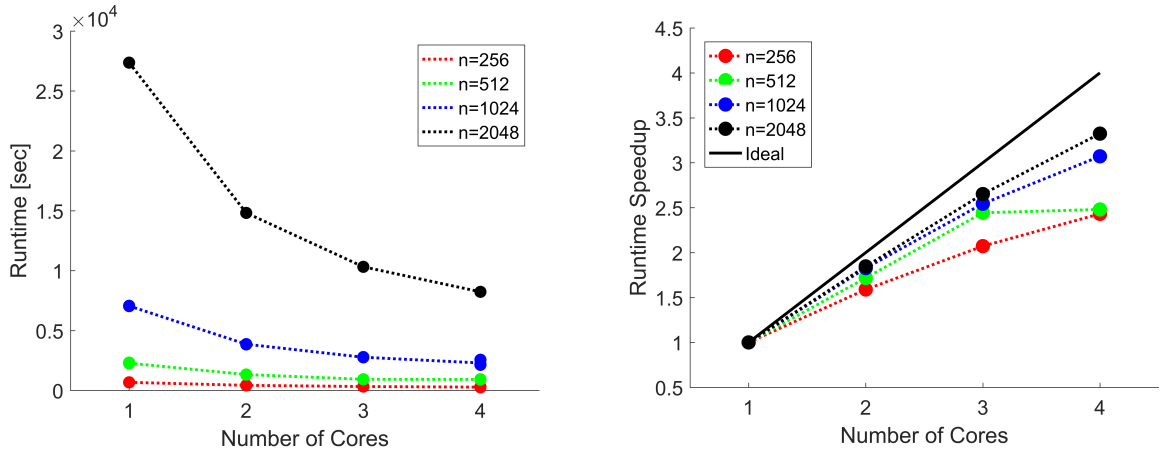

Figure 9: Simulation results for four systems sizes, run for 250,000 timesteps each. Left: runtime results. Right: runtime speedups, normalized by their respective single-core runtimes. The ideal speedup for a 100% parallelized code is shown for reference.

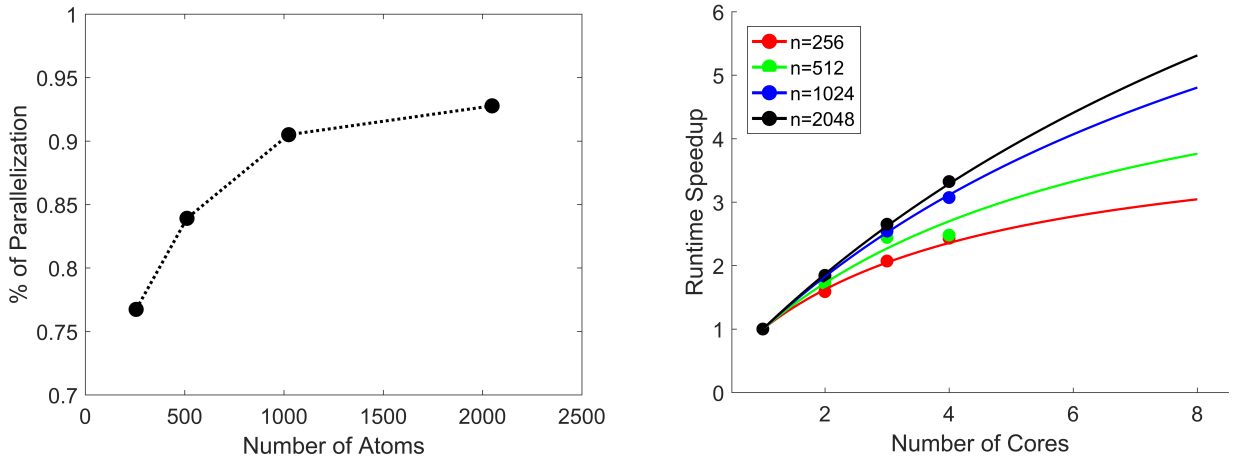

Figure 10: Fitting the simulation results to Amdahl's Law. Left: the percentage of the program that runs in parallel as a function of system size. Right: runtime speedup, using Amdahl's Law to extrapolate the predicted speedup out to eight cores.

Table 1: Predicted speedups from Amdahl's Law.

N	4 cores	100 cores	1000 cores	100,000 cores
256	2.431	4.161	4.285	4.299
512	2.480	5.907	6.183	6.2147
1024	3.071	9.611	10.427	10.525
2048	3.324	12.258	13.656	13.829

Table 2: Predicted runtimes from Amdahl's Law, for 250,000 timesteps.

N	4 cores	100 cores	1000 cores	100,000 cores
256	279.27 sec	163.13 sec	158.44 sec	157.82 sec
512	919.90 sec	386.19 sec	368.95 sec	367.07 sec
1024	2296.80 sec	733.96 sec	676.52 sec	670.22 sec
2048	8233.25 sec	2232.41 sec	2003.87 sec	1978.81 sec

for number of cores ranging from 100 to 100,000. In Table 2, the predicted runtimes are applied to our simulated system with 250,000 timesteps to predict the actual runtime for each system size and number of cores. Also, the real speedup and runtime with four cores is shown for reference. In Figure 9, we saw substantial runtime improvements when the number of cores was increased from one to four. In Tables 1 and 2, we also see that there is a clear performance benefit from four to one hundred cores (although running a simulation on one hundred cores is likely unfeasible in most cases). However, in all cases there is negligible benefit to using more than one hundred cores. To make worthwhile use of a high-end supercomputer with 100,000+ cores, a much more efficient parallel algorithm would need to be developed.

6 Conclusion

Parallel algorithms for molecular dynamics simulations were investigated. There are three common algorithms for simulating large systems in parallel: force decomposition, atom decomposition, and spatial decomposition. A force decomposition algorithm was implemented in the NVT ensemble, capable of running on up to four cores. System sizes of 256, 512, 1024, and 2048 were simulated to investigate the scalability of the force decomposition algorithm. Larger percentages of parallelization were seen in larger system sizes. The largest system size of 2048 atoms was determined to have 92.77% parallelization efficiency, and a runtime speedup of 3.32 was seen running on four cores compared to one. From our results we see immediate and substantial runtime improvements with running molecular dynamics simulations in parallel. The reader is strongly encouraged to incorporate parallel algorithms in their own MD simulations, in order to make larger and more interesting systems more accesible for study.

References

[1] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, San Diego, California, 2002.

[2] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Jouranl of Computational Physics*, 117:1–19, 1995.

[3] Steve Plimpton. Overview of molecular and particulate dynamics. Presented at the 85th Meeting of the Society of Rheology, 2013.

[4] Arthur F. Voter. Parallel replica method for dynamics of infrequent events. *The American Physical Society*, 57(22):985–988, 1998.