Emily Crabb
6.338/18.337 Final Project

Molecular Dynamics Simulations with Julia

I. Project Overview

This project consists of one serial and several parallel versions of a molecular dynamics simulation in the Julia programming language. These implementations are contained in IJulia notebooks. The entire project is publicly available on Github at: https://github.com/ejc44/MD. Specific details regarding each version are included in each notebook, and separate timing notebooks are provided that use the BenchmarkTools.jl package. There are also three Testing folders containing the parameter and data files used for the timing tests. The last folder, called Incomplete, contains a not fully implemented GPU version of the code. My raw timing data, class presentation, and this report are all also included in the repository. Some of the information contained in this report is also included in the repository README file, the individual IJulia notebooks, or the class presentation.

This report first gives a general overview of molecular dynamics simulations. It then describes the most significant details for this project's serial and parallel implementations of a molecular dynamics code in Julia. Finally, the report compares the performance of the serial and parallel versions using the BechmarkTools.jl package and draws some conclusions about the different parallelization methods.

II. Molecular Dynamics

Molecular dynamics is a type of deterministic N-body simulation method. It is used to model the evolution of a system of particles over a fixed time period. In typical scientific applications, the particles in the simulations represent atoms or molecules, and the interparticle forces are calculated using potentials or force fields. However, other types of systems can be modeled with an appropriate implementation of the forces between particles. As such, molecular dynamics is a very flexible simulation method. Given the force on each particle, the positions, trajectories are calculated by numerically solving Newton's equations of motion $\vec{F} = m\vec{a}$. There are many methods available to numerically integrate Newton's equations, but one popular method is the Velocity Verlet algorithm. This algorithm involves solving the following equations at each time step [1]:

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2$$

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

$$\vec{a}(t + \Delta t) = \vec{f}(t)/m$$

This numerical integration algorithm is the one used in this project.

Figures 1 and 2 below show the initial and final positions of forty particles in a simple system. There are two types of particles. Interactions between particles of the same type are repulsive, while interactions between particles of different types are attractive. They are confined in a three-dimensional box with periodic boundary conditions. The simulation is run for 1000 time steps with Δt set to 0.01. This example is simple, but the particle motion is evident.

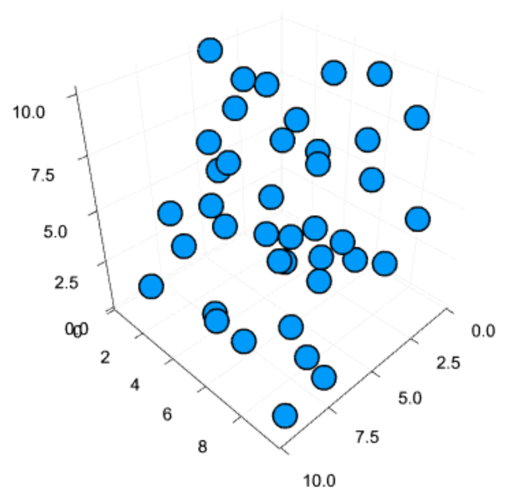


Figure 1: Initial particle configuration

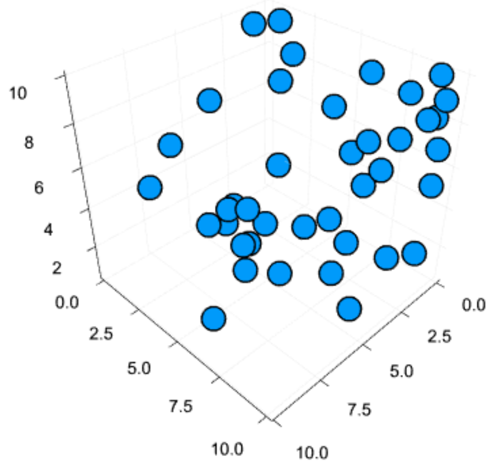


Figure 2: Final particle configuration

III. Implementation Details

This section of the report first describes some important features common to all versions of the code. It then describes the specific implementation details of the different parallel versions.

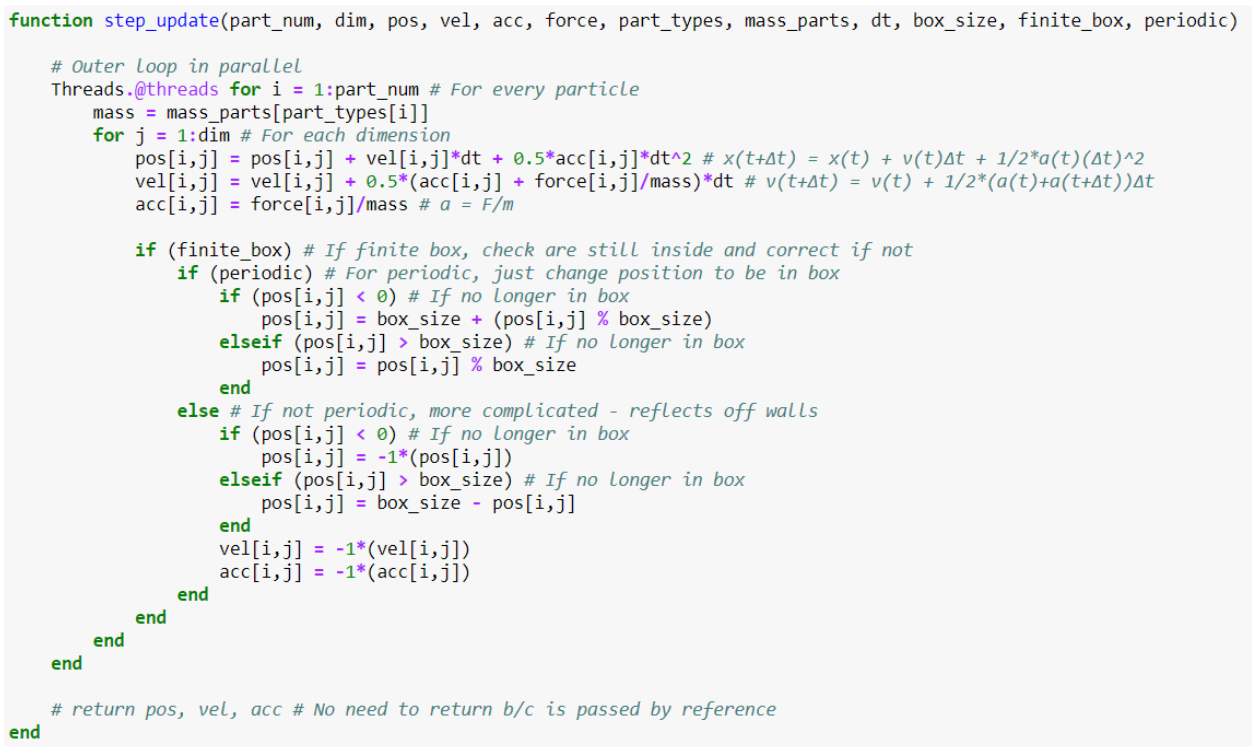
A. *Serial Implementation - Features Common to All Versions*

   a. The option to read parameters such as the number of simulation steps and/or the starting configuration for the position, velocity, acceleration, and forces from external files. These files must be in the same folder as the notebook and have the correct file names as specified in the code. Also, the user cannot pick and choose: if one parameter is read from a file, all the parameters must be read from files and if one type of starting configuration data is read from a file, all the starting configuration data must be read from files. Note: If the parameters and/or initial data is read from files, the program assumes the files are compatible (i.e. the dimension in the dim.txt file matches the dimensions of the particles' positions, etc.). This is not a problem if the user is restarting from previously saved files but could be a problem if the user has made changes to the files or generated them in some other way.

b. The option to directly specify the parameters in the notebook. Note: These parameters are all constants, so one must restart the kernel to redefine them.
c. The option to save the parameters and final configuration data to external files. This is especially useful if the user may want to later continue the same simulation.
d. The option to model finite and infinite systems.
e. The ability to make finite systems periodic or non-periodic.
f. If the initial configuration data is not specified in a file, it can be generated in the code. For example, the starting positions are currently randomly generated within the specified box size. The user can modify any of the initial conditions by altering the initialize() function.
g. The forces are currently all of the form $1/_{r^2}$. The strength of the force between any two types of particles is randomly generated, interactions between particles of the same type are repulsive, and interactions between particles of different types are attractive. This was chosen for simplicity. However, the user can easily modify the form and strength of the forces by altering the find_forces() and gen_interaction() functions.
h. For 2 or 3 dimensional simulations, the system can be visually displayed in a plot. For non-Windows systems, the user can also use the Interact package to manipulate the plot to see the movement of the particles in the system over time. One of the parameters sets the frequency with which the program saves the particles' positions.

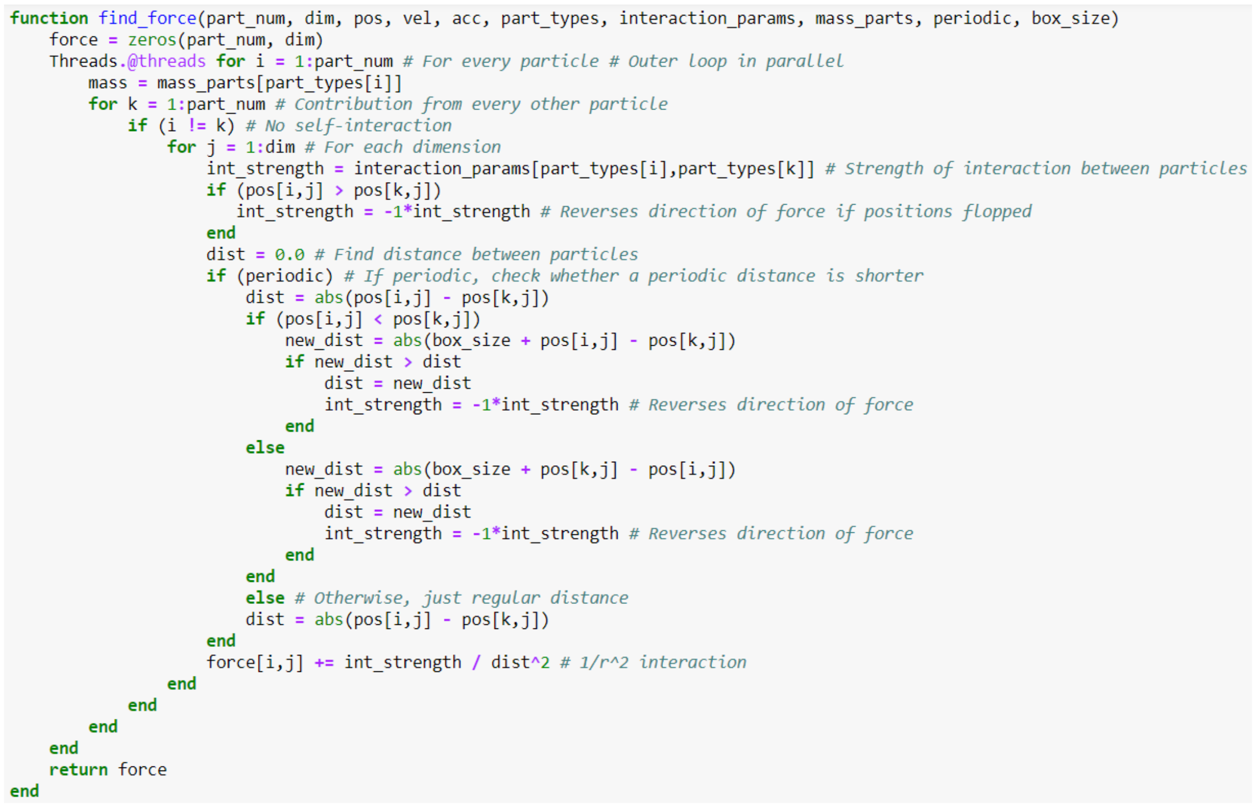
B. Threads Implementation

This version of the molecular dynamics code uses the Threads.@threads macro to implement parallelism. This macro was used for the outermost for loop for both the step_update() and find_force() functions, as shown in Figures 3 and 4 below, which divides the work in this for loop between the available threads. The step_update() function performs the Velocity Verlet integration to update the positions, velocities, and accelerations, while the find_force() function finds the new forces once the particles have all been moved. In each function, the outermost for loop iterates over the particles. This can be parallelized in the step_update() function because the Velocity Verlet algorithm for each particle only depends on that particle's properties, and it can be parallelized in the find_force() function because the forces only depend on the distances between particles and the types of the particles. Note that to achieve parallelism, it is necessary to first set JULIA_NUM_THREADS to a number greater than 1 before starting Julia.



```julia
function step_update(part_num, dim, pos, vel, acc, force, part_types, mass_parts, dt, box_size, finite_box, periodic)

    # Outer loop in parallel
    Threads.@threads for i = 1:part_num # For every particle
        mass = mass_parts[part_types[i]]
        for j = 1:dim # For each dimension
            pos[i,j] = pos[i,j] + vel[i,j]*dt + 0.5*acc[i,j]*dt^2 # x(t+Δt) = x(t) + v(t)Δt + 1/2*a(t)(Δt)^2
            vel[i,j] = vel[i,j] + 0.5*(acc[i,j] + force[i,j]/mass)*dt # v(t+Δt) = v(t) + 1/2*(a(t)+a(t+Δt))Δt
            acc[i,j] = force[i,j]/mass # a = F/m

            if (finite_box) # If finite box, check are still inside and correct if not
                if (periodic) # For periodic, just change position to be in box
                    if (pos[i,j] < 0) # If no longer in box
                        pos[i,j] = box_size + (pos[i,j] % box_size)
                    elseif (pos[i,j] > box_size) # If no longer in box
                        pos[i,j] = pos[i,j] % box_size
                    end
                else # If not periodic, more complicated - reflects off walls
                    if (pos[i,j] < 0) # If no longer in box
                        pos[i,j] = -1*(pos[i,j])
                    elseif (pos[i,j] > box_size) # If no longer in box
                        pos[i,j] = box_size - pos[i,j]
                    end
                    vel[i,j] = -1*(vel[i,j])
                    acc[i,j] = -1*(acc[i,j])
                end
            end
        end
    end

    # return pos, vel, acc # No need to return b/c is passed by reference
end
```

Figure 3: step_update() function using Threads.@threads macro

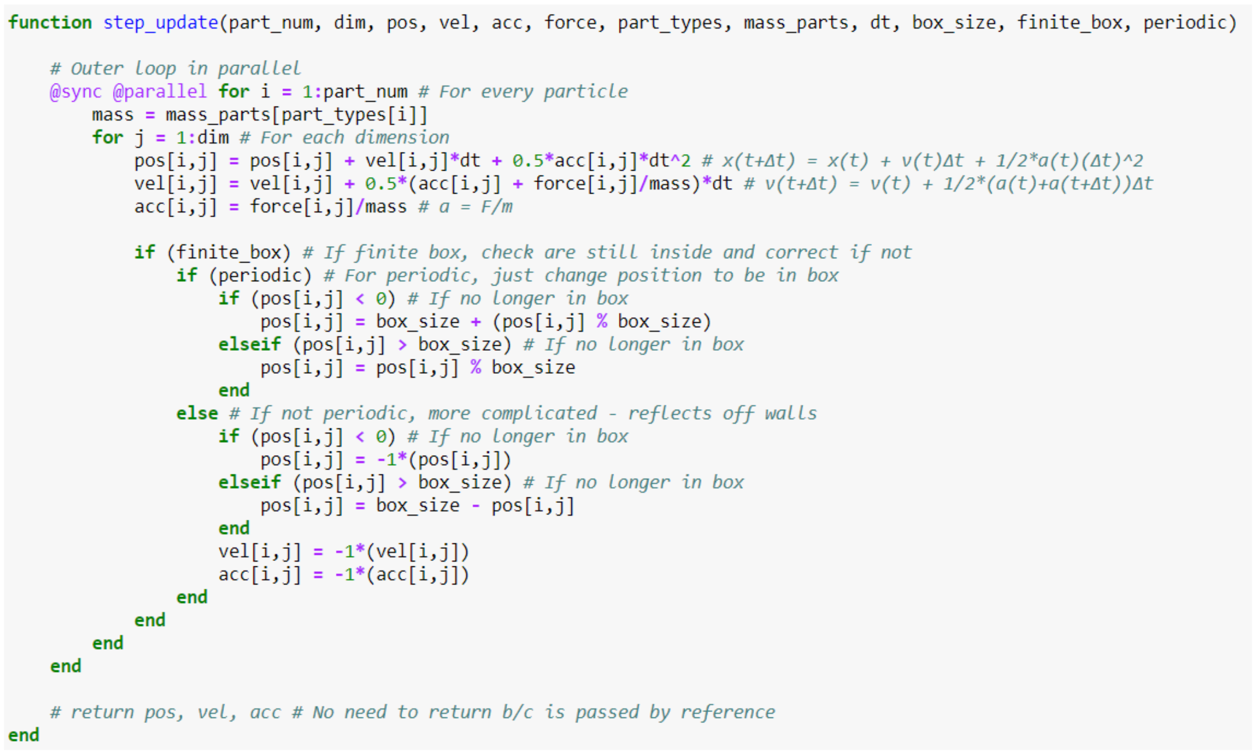


```julia
function find_force(part_num, dim, pos, vel, acc, part_types, interaction_params, mass_parts, periodic, box_size)
    force = zeros(part_num, dim)
    Threads.@threads for i = 1:part_num # For every particle # Outer loop in parallel
        mass = mass_parts[part_types[i]]
        for k = 1:part_num # Contribution from every other particle
            if (i != k) # No self-interaction
                for j = 1:dim # For each dimension
                    int_strength = interaction_params[part_types[i],part_types[k]] # Strength of interaction between particles
                    if (pos[i,j] > pos[k,j])
                       int_strength = -1*int_strength # Reverses direction of force if positions flopped
                    end
                    dist = 0.0 # Find distance between particles
                    if (periodic) # If periodic, check whether a periodic distance is shorter
                        dist = abs(pos[i,j] - pos[k,j])
                        if (pos[i,j] < pos[k,j])
                            new_dist = abs(box_size + pos[i,j] - pos[k,j])
                            if new_dist > dist
                                dist = new_dist
                                int_strength = -1*int_strength # Reverses direction of force
                            end
                        else
                            new_dist = abs(box_size + pos[k,j] - pos[i,j])
                            if new_dist > dist
                                dist = new_dist
                                int_strength = -1*int_strength # Reverses direction of force
                            end
                        end
                    else # Otherwise, just regular distance
                        dist = abs(pos[i,j] - pos[k,j])
                    end
                    force[i,j] += int_strength / dist^2 # 1/r^2 interaction
                end
            end
        end
    end
    return force
end
```

Figure 4: find_force() function using Threads.@threads macro

C. *SharedArray @parallel Implementation*

This version of the molecular dynamics code uses SharedArrays and the @sync @parallel macro to implement parallelism. The SharedArray data type is used to hold the positions, velocities, accelerations, and forces so that each process/worker has access to the whole array. This is important because while each worker should only write to the portion of the array it is processing, the workers may need to read data from other parts of the arrays, such as when calculating the distance between two particles in the find_force() function. The @sync @parallel macro was used for the outermost for loop for both the step_update() and find_force() functions, with the step_update() function shown in Figure 5 below (the find_force() function is similarly modified). This divides the work in these outermost loops amongst the workers. The reasons these loops can be parallelized is discussed in the Threads.@threads section above. To get multiple processes/workers, it is necessary to use the addprocs() function. However, because Julia currently schedules all the workers on one CPU thread, changing the number of threads should not affect the timing significantly [2].

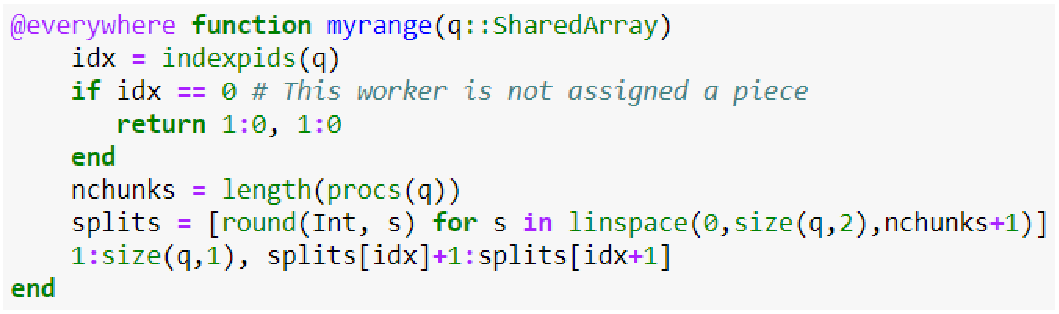


```julia
function step_update(part_num, dim, pos, vel, acc, force, part_types, mass_parts, dt, box_size, finite_box, periodic)

    # Outer loop in parallel
    @sync @parallel for i = 1:part_num # For every particle
        mass = mass_parts[part_types[i]]
        for j = 1:dim # For each dimension
            pos[i,j] = pos[i,j] + vel[i,j]*dt + 0.5*acc[i,j]*dt^2 # x(t+Δt) = x(t) + v(t)Δt + 1/2*a(t)(Δt)^2
            vel[i,j] = vel[i,j] + 0.5*(acc[i,j] + force[i,j]/mass)*dt # v(t+Δt) = v(t) + 1/2*(a(t)+a(t+Δt))Δt
            acc[i,j] = force[i,j]/mass # a = F/m

            if (finite_box) # If finite box, check are still inside and correct if not
                if (periodic) # For periodic, just change position to be in box
                    if (pos[i,j] < 0) # If no longer in box
                        pos[i,j] = box_size + (pos[i,j] % box_size)
                    elseif (pos[i,j] > box_size) # If no longer in box
                        pos[i,j] = pos[i,j] % box_size
                    end
                else # If not periodic, more complicated - reflects off walls
                    if (pos[i,j] < 0) # If no longer in box
                        pos[i,j] = -1*(pos[i,j])
                    elseif (pos[i,j] > box_size) # If no longer in box
                        pos[i,j] = box_size - pos[i,j]
                    end
                    vel[i,j] = -1*(vel[i,j])
                    acc[i,j] = -1*(acc[i,j])
                end
            end
        end
    end

    # return pos, vel, acc # No need to return b/c is passed by reference
end
```

Figure 5: step_update() function using SharedArrays and @sync @parallel macro

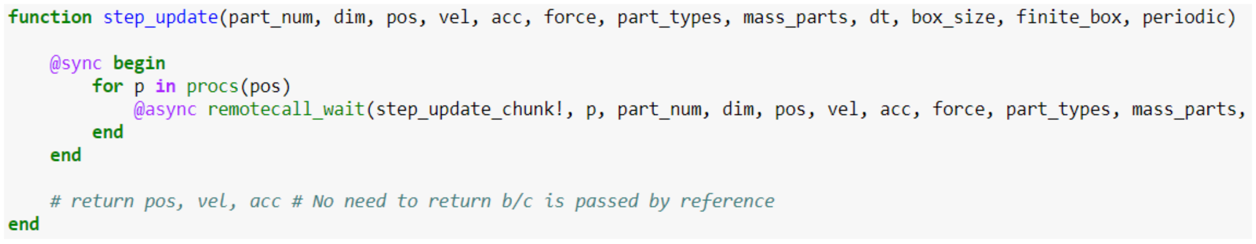
D. *Shared Array Chunks Implementation*

This version of the molecular dynamics code also uses SharedArrays to implement parallelism. However, instead of splitting the work between the workers using the @sync @parallel macro, this version of the code uses manual indexing to split the arrays into chunks for each worker to process. The indexing function is in Figure 6 and was taken from the SharedArrays example in the Julia documentation [2]. Using this function, the outermost for loops in the the step_update() and find_force() functions can again be parallelized. The parallelization of the step_update() function is shown in Figures 7 and 8 below. To get multiple processes/workers, it is necessary to use the addprocs() function. Also, if more workers are added, the indexing myrange() function and the step_update_chunk!() functions must be recompiled so that they are defined on each process using the @everywhere macro. However, because Julia currently schedules all the workers on one CPU thread, changing the number of threads should not affect the timing significantly [2].



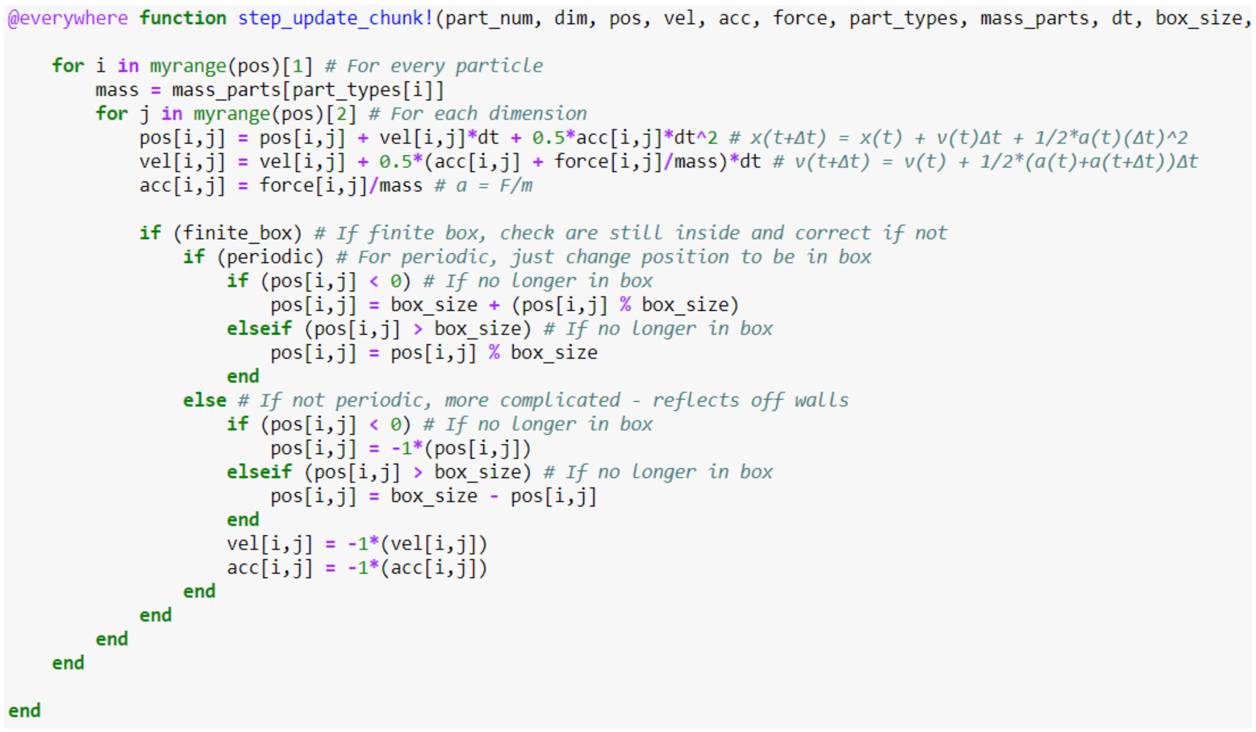
```julia
@everywhere function myrange(q::SharedArray)
    idx = indexpids(q)
    if idx == 0 # This worker is not assigned a piece
       return 1:0, 1:0
    end
    nchunks = length(procs(q))
    splits = [round(Int, s) for s in linspace(0,size(q,2),nchunks+1)]
    1:size(q,1), splits[idx]+1:splits[idx+1]
end
```

Figure 6: Indexing function for SharedArrays taken from Julia documentation [2]



```julia
function step_update(part_num, dim, pos, vel, acc, force, part_types, mass_parts, dt, box_size, finite_box, periodic)

    @sync begin
        for p in procs(pos)
            @async remotecall_wait(step_update_chunk!, p, part_num, dim, pos, vel, acc, force, part_types, mass_parts,
        end
    end

    # return pos, vel, acc # No need to return b/c is passed by reference
end
```

Figure 7: step_update() function



```
@everywhere function step_update_chunk!(part_num, dim, pos, vel, acc, force, part_types, mass_parts, dt, box_size,

    for i in myrange(pos)[1] # For every particle
        mass = mass_parts[part_types[i]]
        for j in myrange(pos)[2] # For each dimension
            pos[i,j] = pos[i,j] + vel[i,j]*dt + 0.5*acc[i,j]*dt^2 # x(t+Δt) = x(t) + v(t)Δt + 1/2*a(t)(Δt)^2
            vel[i,j] = vel[i,j] + 0.5*(acc[i,j] + force[i,j]/mass)*dt # v(t+Δt) = v(t) + 1/2*(a(t)+a(t+Δt))Δt
            acc[i,j] = force[i,j]/mass # a = F/m

            if (finite_box) # If finite box, check are still inside and correct if not
                if (periodic) # For periodic, just change position to be in box
                    if (pos[i,j] < 0) # If no longer in box
                        pos[i,j] = box_size + (pos[i,j] % box_size)
                    elseif (pos[i,j] > box_size) # If no longer in box
                        pos[i,j] = pos[i,j] % box_size
                    end
                else # If not periodic, more complicated - reflects off walls
                    if (pos[i,j] < 0) # If no longer in box
                        pos[i,j] = -1*(pos[i,j])
                    elseif (pos[i,j] > box_size) # If no longer in box
                        pos[i,j] = box_size - pos[i,j]
                    end
                    vel[i,j] = -1*(vel[i,j])
                    acc[i,j] = -1*(acc[i,j])
                end
            end
        end
    end

end
```

Figure 8: step_update_chunk!() function that uses the indexing function so that each worker only performs the updates on its part of the arrays

E. *Attempt at GPU Implementation*

A cursory attempt was made at adapting the molecular dynamics for use on a GPU. However, the step_update() and find_force() functions cannot easily be translated to broadcast functions because of the many conditionals they contain. This could be a future area of research.

IV. Performance Data

To compare the performance of the different versions of the molecular dynamics code, separate benchmarking notebooks were created that eliminated the functionality related to plotting. Only the loop that updated the configurations using the step_update() and find_force() functions was timed. The parameters and initial configurations were read in from files, so that each version of the code did the exact same calculations (as molecular dynamics is deterministic). The codes were run for 1000 time steps with 100, 500, and 1000 particles and with both four and eight threads. All other parameters were kept constant, so they are not particularly relevant for comparison purposes. However, all the testing files are stored in the Github repository in the three testing folders, so any parameter can be examined, and these tests can be replicated. These tests were performed on a laptop with an Intel Core i7-7700HQ CPU that contains four physical cores and eight threads due to hyperthreading. Whether hyperthreading is useful was discussed frequently in class, so simulations were run with both four and eight threads to see if adding more threads resulted in speedup. The minimum times for each code found using the @benchmark macro in the BenchmarkTools.jl package are shown in Figure 9 below, and the results are discussed in the following section. The minimum times were chosen based on the in-class discussions that the minimum time best reflects how quickly the code can run when other background processes are not competing for the CPU. However, there were only small differences between the minimum and average times.

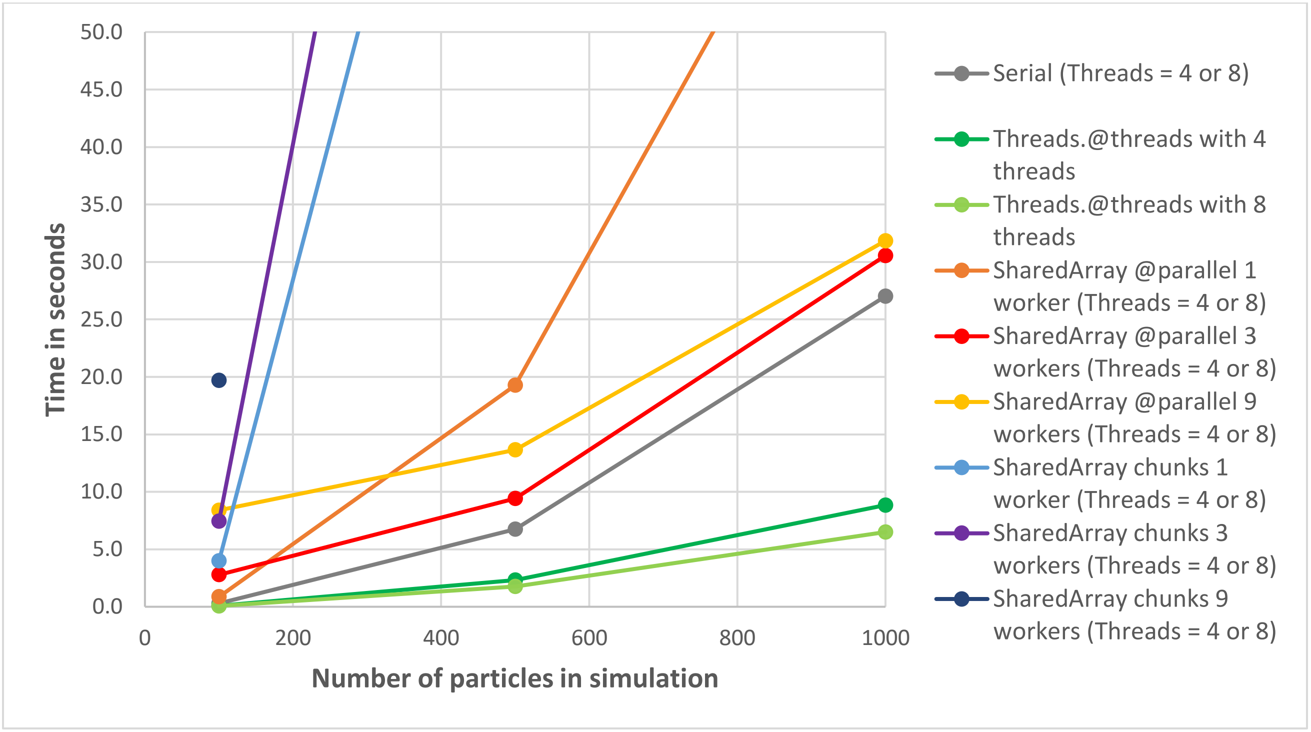


Figure 9: Timing data for different versions of the molecular dynamics simulation code when 100, 500, and 1000 particles were used

A. Notes on the Timing Plot
   a. The time axis is cut off at 50.0 seconds to make the most relevant data easily visible.
   b. For each of the SharedArray implementations, one, three, and nine workers were used.
   c. As expected, the number of threads only affected the implementation that used the Threads.@threads macro. The serial version was not affected as expected. Also, because Julia currently schedules all the workers on one CPU thread [2], changing the number of threads did not affect the timing for the various SharedArray implementations.
   d. No data was collected for the SharedArray chunks with nine workers for 500 or 1000 particles because the code ran so slowly

V. Results and Conclusions

The Threads.@threads parallel implementation was the only version faster than the serial version. When four threads were used, the threads version achieved about 3x speedup compared to the serial version for all three numbers of particles. When eight threads were used, it achieved about 4x speedup compared to the serial version. Thus, doubling the number of threads did increase the speedup but not be a factor of two. This implies hyperthreading does result in some improvement in performance, but it would be necessary to test other numbers of threads to find the optimal value.

Neither of the SharedArray versions was ever faster than the serial version. The version that manually divided the arrays into chunks was much slower than the version that used the @parallel macro. For the manual chunks version, increasing the number of workers increased the runtime. This may be because the overhead of the additional function calls dominates the runtime. For the @parallel version, using only one process was slowest while using three workers was fastest when 500 or 1000 particles were used. The three workers runtime was only about 13% slower and the nine workers runtime was only about 18% slower than the serial version for 1000 particles. Also, the difference between the serial and parallel runtimes decreased for the @parallel version with three or nine workers, so it is possible the parallel version might be faster than the serial version for even larger numbers of particles. It would be interesting to further experiment with the number of particles and the number of workers to see if it is possible to achieve speedup compared with the serial version using the SharedArray @parallel implementation. However, based on the timing data in Figure 9 above, it seems likely that the Threads.@threads version would remain the fastest.

References

1. “Verlet Integration.” From https://www.saylor.org/site/wp-content/uploads/2011/06/MA221-6.1.pdf
2. “Parallel Computing.” The Julia Language. From https://docs.julialang.org/en/stable/manual/parallel-computing