Project Report: Multiple Particle Simulation

Pavan Ravindra, phr2114@columbia.edu

December 18, 2024

1 Introduction

This report details a parallel implementation of a molecular dynamics simulation. The main task is to simulate the motion of N interacting point particles according to the velocity Verlet update rules.

In Section 2, we provide an overview of the steps in a standard molecular dynamics simulation. We then discuss how we parallelize such a simulation in Section 3. We provide some analysis and characterization of our parallel implementation in Section 4, and then we close with some final thoughts and potential future directions in Section 5.

Since the focus of this project is parallel programming, we limit our discussion to only the necessary technical physics details, but all of the simulation choices made in this project are fairly standard [1, 2].

1.1 Overview of Changes made after Presentation

I worked on a few additional things since my presentation yesterday. I summarize them here just for clarity:

1. I managed to get a trivial speedup by increasing the nursery size with the `-A32M` flag, as you had suggested. Previously, my 12 thread runtime only achieved a speedup of $\sim$2x, but with the increased nursery size of 32 megabytes, I get $\sim$5x speedup on 12 threads.
2. I also attempted to implement a version of my code that used `Data.Vector`, but unfortunately I couldn't get it to match the performance of my existing implementation in time for the deadline. The implementation works, it's just much slower than my previous implementation. You can find it in the submitted code under `vector_attempt.zip`. The general approach was to store everything as a `Data.Vector.Unboxed` collection, and then I implemented my own equivalent of `parMap` that does a map over this vector.

I have updated all of the results in this report to match the results with the increased nursery size. Hence, the runtime results here are different from those that I presented yesterday. The previous results are still in the submitted `presentationSlides.pdf` file, and they can also be reproduced by omitting the `-A32M` flag when running my code.

2 Sequential Algorithm

2.1 Overview of Molecular Dynamics

In a molecular dynamics simulation, we simulate the motion of N interacting point particles. The variables of interest are the position and velocity vectors of each particle as they evolve in time. For a particle i at time t, we will use $\mathbf{r}_i(t)$ to denote its position vector and $\mathbf{v}_i(t)$ to denote its velocity vector.

These variables are numerically integrated forward in time by a small, finite timestep Δt according to the velocity Verlet algorithm [2]:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \Delta t \mathbf{v}_i(t) + \frac{\Delta t^2}{2m_i}\mathbf{F}_i(t) \tag{1}$$

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \frac{\Delta t}{2m_i}[\mathbf{F}_i(t) + \mathbf{F}_i(t + \Delta t)] \tag{2}$$

In spirit, these update rules are analogous to integrating the equations forward in time with forward Euler, but the above update rules are known to yield trajectories that are more numerically stable than traditional forward Euler [2].

Once we choose our initial positions $\mathbf{r}_i(0)$ and initial velocities $\mathbf{v}_i(0)$ for all N particles, Eqns. 1 and 2 give us a recipe for computing the future values of our positions $\mathbf{r}_i(t)$ and velocities $\mathbf{v}_i(t)$. The one remaining thing to discuss is the calculation of the force vectors $\mathbf{F}_i(t)$ in these equations.

$\mathbf{F}_i(t)$ denotes the force acting on particle i at time t. We generally assume that our interaction potential is pairwise additive, meaning that we can compute the force acting on a particle by just summing up all of the forces from the other particles acting upon it:

$$\mathbf{F}_i(t) = \sum_{i \neq j} \mathbf{F}_{ij}(t) \tag{3}$$

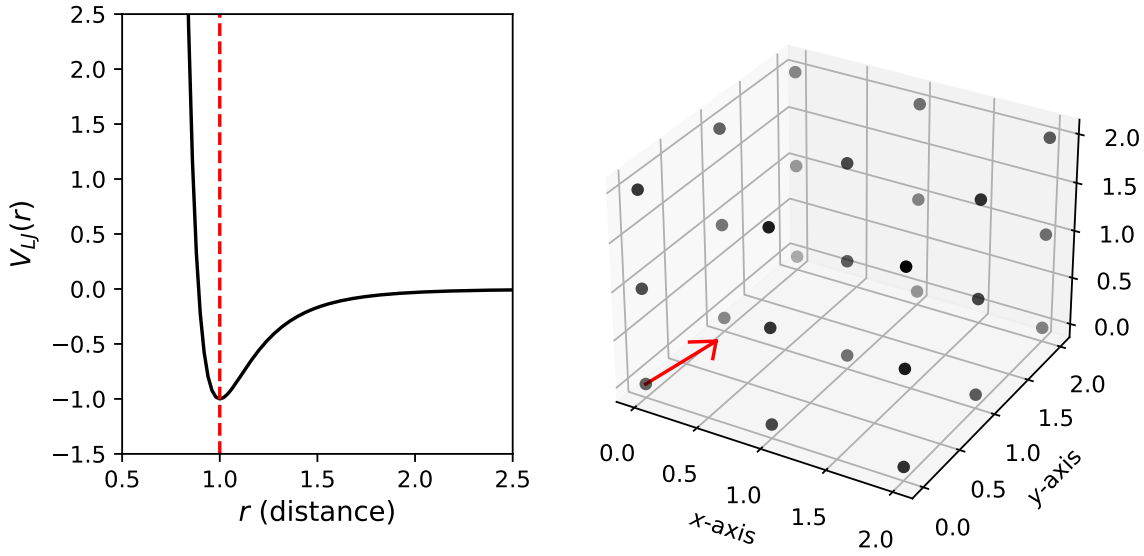

Figure 1: (Left) The Lennard-Jones potential $V_{LJ}(r)$ used in this report. (Right) An example cubic lattice used for the starting positions of our simulations. All particles are given 0 initial velocity except for one particle in the corner, whose velocity vector is indicated by the red arrow.

where $\mathbf{F}_{ij}(t)$ is the force that particle j exerts on particle i. For this project, I use the Lennard-Jones interaction potential, which is a common potential used to simulate the interactions between atoms [1]:

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

In the above, ϵ and σ are parameters that decide the exact shape of the potential. Fig. 1 shows a plot of the Lennard-Jones potential with the chosen values $\epsilon = 1$ and $\sigma = 2^{1/6}$. These are the values that we will use for all of the simulations in this report. The x-axis is the distance r between two particles. The slope of the line gives the magnitude of the force acting on the two particles due to their interaction with each other. This means that if two particles are a distance of 1 unit away from each other, there is no interaction force between the two particles. If they move away from this stable minimum in the potential, they experience a corrective repulsion or attraction that nudges the two particles back towards a separation distance of 1 unit.

If we place these particles such that they form a cubic lattice where all the immediate particle neighbors are separated by 1 distance unit, we will form a mechanically stable cubic crystal. This structure is shown in the right plot of Fig. 1. We use this arrangement of particles as our initial positions $\mathbf{r}_i(0)$ throughout all of the numerical experiments in this report. We will discuss the choice of initial velocities $\mathbf{v}_i(0)$ in Section 2.2.

To summarize, there are two main steps in running a molecular dynamics simulation:

1. Generate initial conditions: Generating the initial positions for a cubic lattice of N particles as described above takes $O(N)$ time. We'll discuss a method for generating the initial velocities for our simulation that also takes $O(N)$ time.
2. Update the positions and velocities of our particles for the desired simulation length: This step involves repeatedly applying Eqns. 1 and 2 to move the simulation forward in time. These equations require us to compute the force $\mathbf{F}_i(t)$ on each particle i. Since we are to compute the force on all N of these particles, and the calculation of each $\mathbf{F}_i(t)$ requires summing over all other particles $j \neq i$, the overall cost of this step is $O(N^2T)$, where T is the number of timesteps that we want to run the simulation for.

From this analysis, it is clear that the second step should be the major computational bottleneck in running our simulations. We verify this empirically in Fig. 2, where we run molecular dynamics simulations for 100 timesteps with varying numbers of particles. From this plot, it is clear that the simulation runtime scales quadratically with the number of particles, just as we predicted.

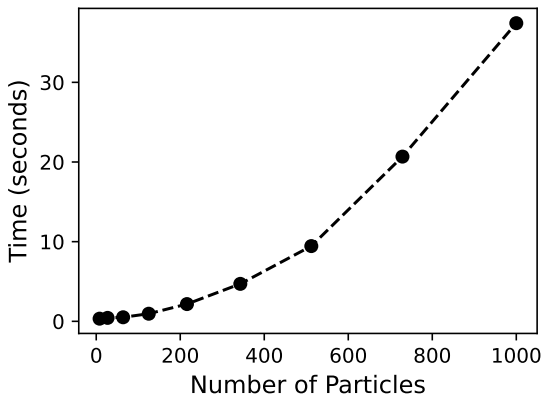

Figure 2: The scaling of the simulation runtime (single-threaded) as the number of simulated particles increases.

2.2 Simulating the Melting of a Lennard-Jones Crystal

To verify that the results of our simulations are physically reasonable, we simulate the melting of the cubic Lennard-Jones crystal that we described above. Our initial positions are that of the cubic lattice shown in the right half of Fig. 1. We then give all of the particles an initial velocity of 0 except for one particle in the corner of the lattice. This particle is given a velocity based on what we'll call the "temperature" of the simulation. The larger the "temperature" is, the faster this initial velocity will be chosen to be. As an aside: this is a physically-motivated but incorrect definition of temperature.

We then plot (the real part of) the self-intermediate scattering function $F_s(k,t)$:

$$F_s(k,t) = \sum_{i=1}^{N} e^{i\mathbf{k}\cdot(\mathbf{r}_i(t)-\mathbf{r}_i(0))} \tag{5}$$

If the initial velocity is too low, the neighboring atoms will cushion out this initial velocity, and the crystal will remain intact. In such cases, $F_s(k,t)$ will stay around 1 for all values of time. However, if the initial velocity is large enough, the moving particle will violently push the other particles out of their stable cubic lattice positions, causing the crystal to melt into a disordered phase. When this happens, $F_s(k,t)$ will decay to 0.

Fig. 3 shows this behavior for three different temperatures. At the lowest temperature, we see that $F_s(k,t)$ oscillates near a value of 1, meaning that the crystal remained intact. However, at the higher temperatures, $F_s(k,t)$ decays to 0, indicating that the crystal has melted! In fact, the higher temperature crystal melts faster than the intermediate temperature crystal. (Note that the x-axis is on a log-scale, as is standard with $F_s(k,t)$ plots. This means that the difference in melting time between these higher temperature simulations is actually more substantial than it appears at first.)

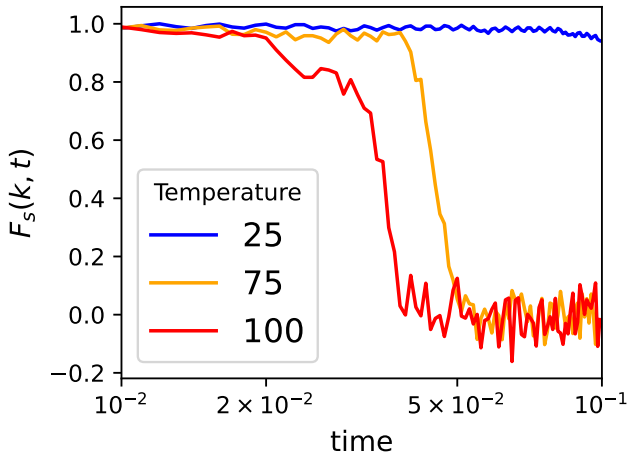

Figure 3: Demonstrating the melting of a Lennard-Jones crystal at high temperatures via the decay of the self-intermediate scattering function $F_s(k,t)$.

3 Parallelizing the Simulation

We now describe how we parallelized our implementation. We focus only on the parts of our implementation that are relevant to understanding our parallelization-induced speedup. Our full code listing is provided in the Appendix at the end of this report (and of course as part of the Courseworks submission).

As discussed in the previous section, the $\mathbf{r}_i(t)$ and $\mathbf{v}_i(t)$ vectors are what we are primarily interested in computing. We define a `MDVector` datatype that stores the 3 components of these vectors:

```
1  data MDVector = MDVector !Double !Double !Double
```

The `!`'s enforce immediate evaluation because we want to avoid laziness at this level. Very importantly, the simulations in this report use periodic boundary conditions. This means that a particle that leaves the right-hand side of the box will effectively re-enter the box immediately on the left-hand side. A similar rule applies to all 6 faces on the edge of our cubic simulation cell. This periodicity is why the `boxLength` parameter has to be passed to many of the functions that act on `MDVector`'s.

As previously discussed, the major computational bottleneck in these simulations is the calculation of the forces acting on each particle, since this requires $O(N^2)$ individual force calculations. By implementing this with one `map` call, we can make use of strategies to easily parallelize these force calculations.

```
1  -- Computes list of forces on all particles given a configuration
2  forceMatrix :: [MDVector] -> Double -> [MDVector]
3  forceMatrix rs boxLength =
4    map totalForce rs `using` parList rseq
5    where
6      -- Gets force acting on particle at r1 due to particle at r2
7      forceVector r1 r2
8        | r1 == r2 = zeroVector
9        | otherwise = vectorMultiply flj (unitVector r12)
10         where r12 = displacement r2 r1 boxLength
11               d12 = vectorNorm r12
12               sor = sigma / d12
13               flj = 24.0 * epsilon * (2 * (sor ** 12.0) - (sor ** 6.0)) / d12
14     -- Computes total force on particle at r due to all other particles
15     totalForce r = foldr vectorAdd zeroVector $ map (forceVector r) rs
```

The `rs` argument is a list of the `MDVector` positions of our particles at the current timestep. We `map` over this list with a `totalForce` function that acts on individual `MDVector`'s in this list. `totalForce` looks at this individual `MDVector` and computes the total force acting on this particle as a result of all of the other `forceVector`'s that act on this particle. This force is determined by the Lennard-Jones interaction from Eqn. 4 and is computed on Line 13 of the above code snippet. Since our implementation of the force calculation is written around this single `map` call, we can trivially parallelize this calculation using strategies, as done on Line 4.

4 Results

4.1 Speedup Results

We'll now move onto characterizing this simple parallelization scheme. Fig. 4 shows that the parallel implementation is certainly faster for the larger system sizes.

Fig. 5 shows that increasing the number of threads for a fixed system size consistently decreases the overall runtime. Although we don't achieve ideal scaling, our parallel implementation does still steadily yield improve runtimes. The obvious exception to this is the green curve corresponding to 8 particles, for which it is clearly more work than it's worth to divide the force calculation across threads.

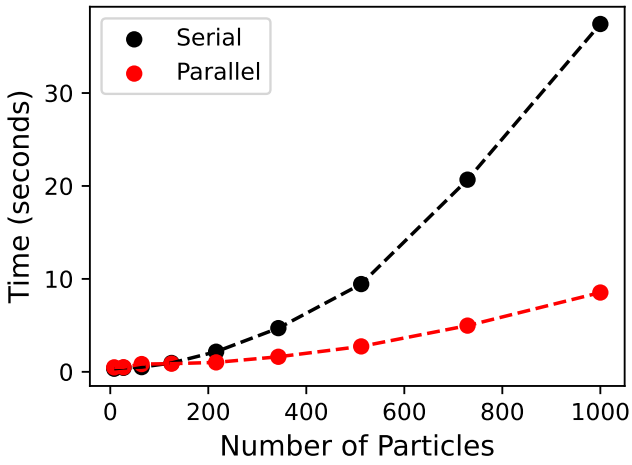

Figure 4: Comparing the runtime scaling between the serial and parallelized (12 threads) implementations.

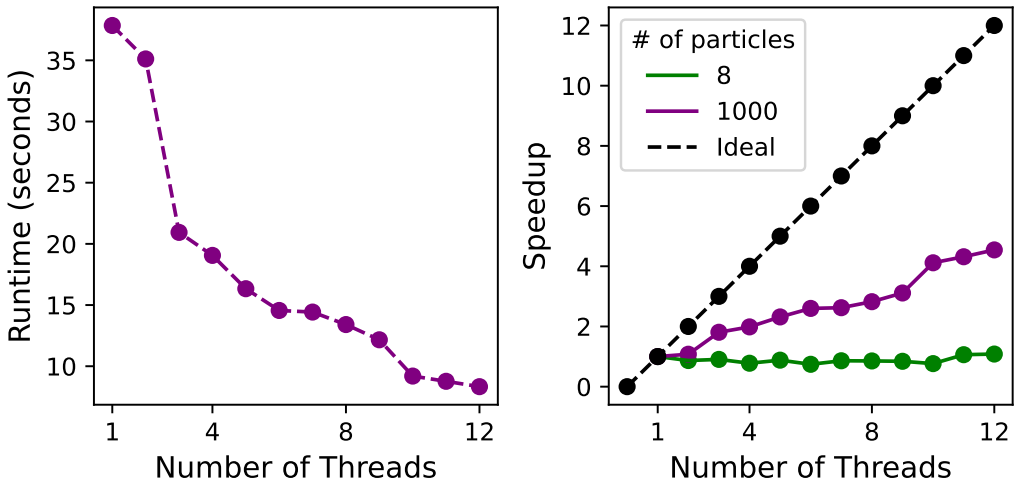

Figure 5: (Left) The total simulation runtime for 100 timesteps for a simulation size of 1000 particles. (Right) The speedup scaling with respect to number of threads for selected system sizes. The dashed black line shows ideal scaling.

4.2 Spark Fate Analysis

We can also take a closer look at what exactly is happening to all of the sparks at different simulation sizes. At larger simulation sizes, most of the sparks are successfully converted. Even for the largest simulation sizes, we never run into issues of the spark pool overflowing. Unsurprisingly, at small system sizes, the final statuses of the sparks is more depressing, with a fair amount of them getting fizzled.

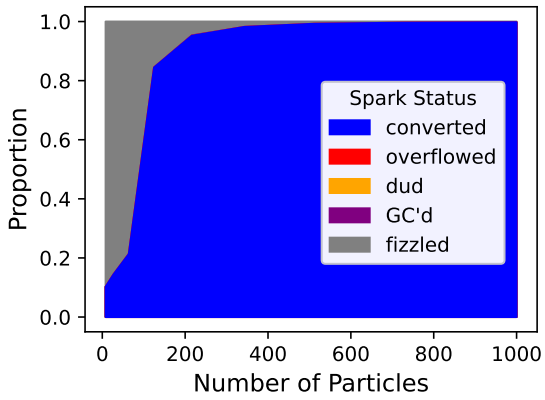

Figure 6: The proportion of sparks that end in each category based on the number of particles in the simulation.

4.3 Load Balancing Analysis

We conclude our analysis of this implementation by looking at the load balancing for the largest simulations we ran: 1000 particles across 12 threads. At a first glance, our implementation seems to smoothly balance the work load across all of the threads, as shown in Fig. 7. However, after zooming in, things don't look quite as pretty. The top image in Fig. 8 shows a slightly zoomed-in version of a single simulation timestep. You can see that the spark pool for all the force calculations is created suddenly, and then it gradually declines as the individual force calculations finish. If you look even closer at the individual thread-level activity, you can see that within a single one of these timesteps, the threads all simultaneously stop working to do some

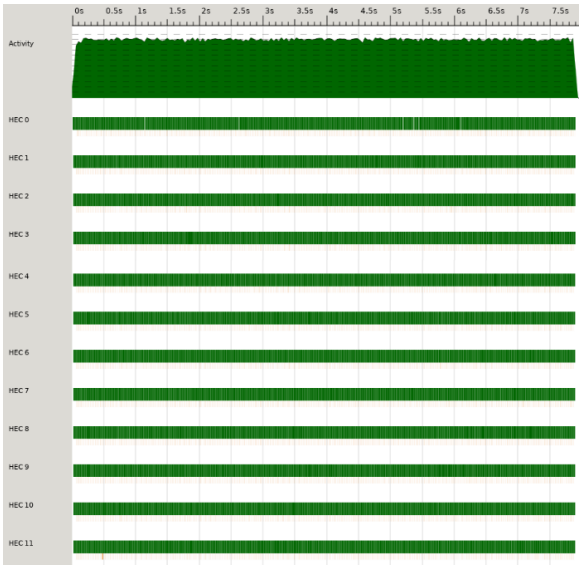

Figure 7: A zoomed out picture of the load balancing for 1000 particles across 12 threads.

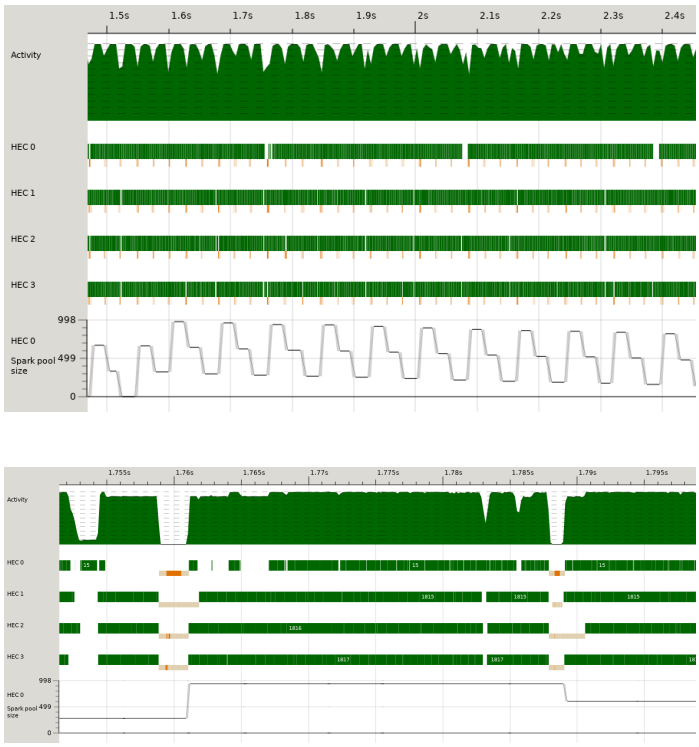

Figure 8: (Top) A slightly zoomed-in version of Fig. 7. (Bottom) An even more zoomed-in version of Fig. 7.

garbage collection. Note that we only show 4 threads in these zoomed-in images to reduce the size of the figures, but the other threads' behaviors look similar.

4.4 Using `parListChunk` instead of `parList`

Since we parallelized our force calculation in `forceMatrix` using `parList`, a natural follow-up question would be to try using `parListChunk` instead to group together certain particles' force calculations. If we want to do this, we also have to calibrate the size of the chunks that we split the particle coordinates into. We focus on our largest simulation sizes, which contain 1000 particles. The results for different chunk sizes and thread counts are in Table 1 below.

For low thread counts, all chunk sizes yielded similar runtimes, as expected. However, for the higher thread counts, using a chunk size of 1 (which is equivalent to just using `parList`) yielded the best performance. This suggests that it isn't worth the work of splitting the work up into chunks.

Total Chunk Size	-N1	-N2	-N3	-N4	-N5	-N6	-N7	-N8	-N9	-N10	-N11	-N12
1	37.5	25.3	21.8	19.5	**19.0**	19.6	**18.2**	**17.9**	**17.4**	**16.7**	**16.3**	**15.3**
2	41.0	29.6	25.6	24.6	23.7	23.1	23.3	20.9	20.5	21.0	19.9	18.7
4	39.8	28.8	25.4	23.6	22.3	22.1	22.2	21.9	20.7	19.6	20.4	17.8
5	43.4	30.8	26.3	23.8	22.8	22.4	21.6	21.1	21.5	20.8	21.2	19.7
10	**37.4**	**22.6**	**20.8**	**21.0**	20.0	19.2	18.4	18.2	17.9	17.7	18.2	18.2
20	37.7	25.3	23.1	21.3	20.2	**19.0**	19.7	19.5	20.0	19.3	19.2	21.5
50	37.9	25.8	23.3	21.5	20.4	20.2	19.7	19.9	20.7	20.9	20.1	20.1
100	40.2	27.3	26.6	24.0	21.6	22.3	22.9	23.1	23.1	20.5	22.0	21.6

Table 1: 100 timestep simulation runtimes (in seconds) for different numbers of threads and chunk sizes. Simulations contain 1000 particles.

5 Conclusions

In this report, we described our efforts to parallelize molecular dynamics simulations of Lennard-Jones particles. We parallelize the force calculation at each simulation timestep using strategies, which lowers the overall computational cost of each simulation timestep and accordingly the overall simulation runtime. As expected, our implementation did best with large simulation sizes of 1000 particles. Notably, simulations in modern computational physics literature often involve running simulations with $\sim$1000 particles [1].

Unfortunately, the threads appeared to take synchronized garbage collection breaks, which occur during the execution of individual molecular dynamics timesteps. As a next step, it would be useful to consider how we can reduce the slowdown induced by these mid-timestep garbage collection pauses.

Appendix A Code Listings

The instructions for executing our code can be found in the submitted `READMDE.md` file.

Appendix A.1 `src/MDVector.hs`

```haskell
module MDVector
  ( MDVector(..),
    zeroVector,
    vectorAdd,
    vectorSubtract,
    vectorMultiply,
    dotProduct,
    vectorNorm,
    displacement,
    distance,
    unitVector,
    matrixMultiply,
    addMatrixList
  ) where

import Data.Fixed (mod')

-- Fundamental 3-vector datatype for x, y, and z coordinates
data MDVector = MDVector !Double !Double !Double

instance Show MDVector where
  show (MDVector x y z) = "[ " ++ (show x) ++ " , " ++ (show y) ++ " , " ++ (show z) ++ " ]"

-- Allows for some floating point error when comparing MDVectors
instance Eq MDVector where
  (MDVector x1 y1 z1) == (MDVector x2 y2 z2) =
    all closeEnough rTuple
    where closeEnough (a,b) = (abs (a - b)) < 1e-10
          rTuple = [(x1,x2),(y1,y2),(z1,z2)]

-- Defining a zero vector (just for convenience)
zeroVector :: MDVector
zeroVector = MDVector 0.0 0.0 0.0

-- Adds two MDVectors together
vectorAdd :: MDVector -> MDVector -> MDVector
vectorAdd (MDVector x1 y1 z1) (MDVector x2 y2 z2) =
  MDVector (x1+x2) (y1+y2) (z1+z2)

-- Subtracts two MDVectors
vectorSubtract :: MDVector -> MDVector -> MDVector
vectorSubtract (MDVector x1 y1 z1) (MDVector x2 y2 z2) =
  MDVector (x1-x2) (y1-y2) (z1-z2)

-- Multiplies a MDVector by a scalar
vectorMultiply :: Double -> MDVector -> MDVector
vectorMultiply c (MDVector x y z) =
  MDVector (c*x) (c*y) (c*z)

-- Dot product between MDVectors
dotProduct :: MDVector -> MDVector -> Double
dotProduct (MDVector x1 y1 z1) (MDVector x2 y2 z2) =
  (x1*x2) + (y1*y2) + (z1*z2)

-- Computes the norm of an MDVector
vectorNorm :: MDVector -> Double
vectorNorm v1 =
  sqrt $ dotProduct v1 v1

-- Given MDVectors v1 and v2, computes wrapped displacement vector v_{12}
```

```haskell
displacement :: MDVector -> MDVector -> Double -> MDVector
displacement v1 v2 boxLength =
  wrapDisplacement $ vectorSubtract v2 v1
  where wrapDisplacement (MDVector x y z) = MDVector (boxMod x) (boxMod y) (boxMod z)
        halfBox = boxLength / 2.0
        boxMod c = (mod' (c + halfBox) boxLength) - halfBox

-- Computes the distance between two position vectors
distance :: MDVector -> MDVector -> Double -> Double
distance vec1 vec2 boxLength =
  vectorNorm $ displacement vec1 vec2 boxLength

-- Returns a unit vector in the direction of the provided MDVector
-- (or the zero vector if the provided vector is the zero vector)
unitVector :: MDVector -> MDVector
unitVector vec
  | vec == zeroVector = zeroVector
  | otherwise = vectorMultiply (1.0 / (vectorNorm vec)) vec

-- Multiplies a list of vectors (AKA matrix) by the provided double
matrixMultiply :: Double -> [MDVector] -> [MDVector]
matrixMultiply c m =
  map (vectorMultiply c) m

-- Adds a list of "matrices" to the provided "matrix"
addMatrixList :: [MDVector] -> [[MDVector]] -> [MDVector]
addMatrixList m0 matrixList =
  foldr (zipWith vectorAdd) m0 matrixList
```

Appendix A.2 src/MDEngine.hs

```haskell
module MDEngine
  ( forceMatrix,
    velocityVerlet,
    isf,
    mdIsf,
    mdTraj
  ) where

import Control.Parallel.Strategies (using, parList, rseq)

import MDVector

-- Simulation Parameters

epsilon :: Double
epsilon = 1.0

sigma :: Double
sigma = 2.0 ** (-1.0/6.0) -- Chosen so that r_{min} = 1

mass :: Double
mass = 1.0

dt :: Double
dt = 1e-3

-- Computes list of forces on all particles given a configuration
forceMatrix :: [MDVector] -> Double -> [MDVector]
forceMatrix rs boxLength =
  map totalForce rs `using` parList rseq
  where
    -- Gets force acting on particle at r1 due to particle at r2
    forceVector r1 r2
      | r1 == r2 = zeroVector
      | otherwise = vectorMultiply flj (unitVector r12)
```

```haskell
        where r12 = displacement r2 r1 boxLength
              d12 = vectorNorm r12
              sor = sigma / d12
              flj = 24.0 * epsilon * (2 * (sor ** 12.0) - (sor ** 6.0)) / d12
    -- Computes total force on particle at r due to all other particles
    totalForce r = foldr vectorAdd zeroVector $ map (forceVector r) rs

-- Updates positions, velocities, and forces using velocity Verlet
velocityVerlet :: [MDVector] -> [MDVector] -> [MDVector] -> Double -> ([MDVector],[MDVector],[MDVector])
velocityVerlet rt1 vt1 ft1 boxLength =
  ( rt2 , vt2 , ft2 )
  where rt2 = addMatrixList rt1 [ (matrixMultiply dt vt1) , (matrixMultiply c1 ft1) ]
        ft2 = forceMatrix rt2 boxLength
        vt2 = addMatrixList vt1 [ (matrixMultiply c2 ft1) , (matrixMultiply c2 ft2) ]
        c1 = (dt ** 2.0) / (2.0 * mass)
        c2 = dt / (2.0 * mass)

-- Computes intermediate scattering function value between two configurations
isf :: MDVector -> [MDVector] -> [MDVector] -> Double
isf k r0 rt =
  let diffMatrix = zipWith vectorSubtract rt r0 in
  let dotMatrix = map (dotProduct k) diffMatrix in
  let cosMatrix = map cos dotMatrix in
  (sum cosMatrix) / (fromIntegral (length cosMatrix))

-- Given initial configuration, velocities, number of timesteps to execute
-- and a k-vector of interest, computes the self-ISF trajectory.
mdIsf :: [MDVector] -> [MDVector] -> Int -> Double -> MDVector -> [Double]
mdIsf r0 v0 timesteps boxLength k =
  let mdIsfHelper rt vt ft steps
        | steps == 0 = []
        | otherwise  = (isf k r0 rt) : (mdIsfHelper rt2 vt2 ft2 (steps - 1))
          where (rt2,vt2,ft2) = velocityVerlet rt vt ft boxLength
  in mdIsfHelper r0 v0 f0 timesteps
     where f0 = forceMatrix r0 boxLength

-- Given initial configuration, velocities, and number of timesteps to execute,
-- computes the trajectory of the first particle.
mdTraj :: [MDVector] -> [MDVector] -> Int -> Double -> [MDVector]
mdTraj r0 v0 timesteps boxLength =
  let mdTrajHelper rt vt ft steps
        | steps == 0 = []
        | otherwise  = (head rt) : (mdTrajHelper rt2 vt2 ft2 (steps - 1))
          where (rt2,vt2,ft2) = velocityVerlet rt vt ft boxLength
  in mdTrajHelper r0 v0 f0 timesteps
     where f0 = forceMatrix r0 boxLength
```

Appendix A.3 src/InitialConditions.hs

```haskell
module InitialConditions
  ( cubicPositions,
    zeroVelocity,
    oneVelocity
  ) where

import MDVector

-- Generates list with values for each coordinate in cubic lattice
getRange :: Double -> [Double]
getRange boxLength =
  map fromIntegral [0..(numParticles-1)]
  where numParticles = floor boxLength :: Int

-- Generates cubic initial configuration
cubicPositions :: Double -> [MDVector]
cubicPositions boxLength =
  [ MDVector x y z | x <- range , y <- range , z <- range ]
  where range = getRange boxLength

-- Generates zero initial velocities
zeroVelocity :: Double -> [MDVector]
zeroVelocity boxLength =
  [ MDVector 0 0 0 | _ <- range , _ <- range , _ <- range ]
  where range = getRange boxLength

-- Generates mostly zero initial velocities but gives particle 1 some velocity based on the provided temperature.
oneVelocity :: Double -> Double -> [MDVector]
oneVelocity boxLength temp =
  (MDVector temp temp temp) : tail (zeroVelocity boxLength)
```

Appendix A.4 app/isf/Main.hs

```haskell
module Main (main) where

import System.Exit (die)
import System.Environment (getArgs, getProgName)
import Text.Printf (printf)

import MDVector
import MDEngine
import InitialConditions

main :: IO ()
main =
  do args <- getArgs
     case args of
       [timestepsString,boxLengthString,tempString] -> do
         let timesteps = read timestepsString :: Int
             boxLength = read boxLengthString :: Double
             temp = read tempString :: Double
             r0 = cubicPositions boxLength
             v0 = oneVelocity boxLength temp
             kval = 2.0 * pi * 10
             k = MDVector kval kval kval
             isfTraj = mdIsf r0 v0 timesteps boxLength k
         putStr $ unlines (map (\d -> printf "%.3f" d) isfTraj)
       _ -> do pn <- getProgName
               die $ "Usage: " ++ pn ++ " <timesteps> <boxLength> <temperature>"
```

Appendix A.5 app/traj/Main.hs

```haskell
module Main (main) where

import System.Exit (die)
import System.Environment (getArgs, getProgName)

import MDEngine
import InitialConditions

main :: IO ()
main =
  do args <- getArgs
     case args of
       [timestepsString,boxLengthString,tempString] -> do
         let timesteps = read timestepsString :: Int
             boxLength = read boxLengthString :: Double
             temp = read tempString :: Double
             r0 = cubicPositions boxLength
             v0 = oneVelocity boxLength temp
             firstTraj = mdTraj r0 v0 timesteps boxLength
         putStr $ unlines (map show firstTraj)
       _ -> do pn <- getProgName
               die $ "Usage: " ++ pn ++ " <timesteps> <boxLength> <temperature>"
```

Appendix A.6 test/vector/Vector.hs

```haskell
import System.Exit (exitFailure)

import MDVector

checkEqual :: Eq a => a -> a -> IO ()
checkEqual val1 val2 =
  if val1 == val2
  then return ()
  else exitFailure

checkDouble :: Double -> Double -> IO ()
checkDouble d1 d2 =
  if abs (d1 - d2) < 1e-10
  then return ()
  else exitFailure

main :: IO ()
main = do
  let oneVector = MDVector 1.0 1.0 1.0
      twoVector = MDVector 2.0 2.0 2.0
      threeVector = MDVector 3.0 3.0 3.0
      oneFiveVector = MDVector 1.5 1.5 1.5
      someVector = MDVector 8.0 7.0 1.0
  checkDouble 0.0 0.0
  checkEqual zeroVector zeroVector
  checkEqual oneVector (vectorAdd oneVector zeroVector)
  checkEqual twoVector (vectorAdd oneVector oneVector)
  checkEqual oneVector (vectorSubtract twoVector oneVector)
  checkEqual twoVector (vectorMultiply 2.0 oneVector)
  checkDouble 12.0 (dotProduct twoVector twoVector)
  checkDouble 3.0 (vectorNorm (MDVector 1.0 2.0 2.0))
  checkEqual oneVector (displacement twoVector threeVector 5.0)
  checkEqual (vectorMultiply (-1.0) oneVector) (displacement threeVector twoVector 5.0)
  checkEqual oneFiveVector (displacement threeVector oneVector 3.5)
  checkEqual (vectorMultiply (-1.0) oneFiveVector) (displacement oneVector threeVector 3.5)
  checkDouble 5.0 (distance oneVector someVector 10)
  checkDouble 1.0 (vectorNorm (unitVector someVector))
  checkDouble (vectorNorm someVector) (dotProduct (unitVector someVector) someVector)
```

Appendix A.7 test/engine/Engine.hs

```haskell
import System.Exit (exitFailure)

import MDVector
import MDEngine
import InitialConditions

checkEqual :: Eq a => a -> a -> IO ()
checkEqual val1 val2 =
  if val1 == val2
  then return ()
  else exitFailure

checkListEqual :: Eq a => [a] -> [a] -> IO ()
checkListEqual [] [] = return ()
checkListEqual _ [] = exitFailure
checkListEqual [] _ = exitFailure
checkListEqual (a:as) (b:bs)
  | a == b = checkEqual as bs
  | otherwise = exitFailure

main :: IO ()
main = do
  let boxLength = 5.0
      cubicCrystal = cubicPositions boxLength
      crystalForce = forceMatrix cubicCrystal boxLength
  checkListEqual crystalForce $ map (\_ -> zeroVector) crystalForce
```

Appendix A.8 test/init/Init.hs

```haskell
import System.Exit (exitFailure)

import MDVector
import InitialConditions

checkEqual :: Eq a => a -> a -> IO ()
checkEqual val1 val2 =
  if val1 == val2
  then return ()
  else exitFailure

checkDouble :: Double -> Double -> IO ()
checkDouble d1 d2 =
  if abs (d1 - d2) < 1e-10
  then return ()
  else exitFailure

main :: IO ()
main = do
  let initPos = cubicPositions 2.0
      zeroVel = zeroVelocity 2.0
      oneVel = oneVelocity 2.0 1.0
  checkDouble 0.0 0.0
  checkEqual zeroVector zeroVector
  checkEqual 8 (length initPos)
  checkEqual 8 (length zeroVel)
  checkEqual 8 (length oneVel)
  checkDouble (3.0 + 3.0 * (sqrt 2.0) + (sqrt 3.0)) $ sum (map vectorNorm initPos)
  checkDouble 0.0 $ head (map vectorNorm zeroVel)
  checkDouble 0.0 $ sum (map vectorNorm zeroVel)
  checkDouble (sqrt 3.0) $ head (map vectorNorm oneVel)
  checkDouble 0.0 $ sum (tail (map vectorNorm oneVel))
```

References

[1] Walter Kob and Hans C Andersen. Testing mode-coupling theory for a supercooled binary lennard-jones mixture. ii. intermediate scattering function and dynamic susceptibility. *Physical Review E*, 52(4):4134, 1995.

[2] Mark E Tuckerman. *Statistical mechanics: theory and molecular simulation*. Oxford university press, 2023.