CSE 628 Term Project Report

Threading Optimizations in a Simple Particle Physics Simulation

Christopher Moore

07-29-2023

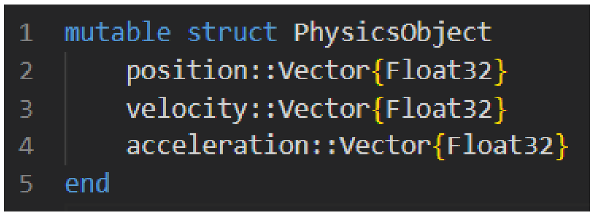
Objective

The main Objective is to explore parallel computing concepts and conduct performance analysis on parallel vs non-parallel computing problems. A simple particle physics engine is implemented using Julia as a computing problem that can be enhanced with parallel computing. All elements of the physics will be implemented by hand and done in a sequential manner. The particle system will be enhanced using Julia’s built in threading functions. The time to compute discrete physics steps (frames) will be compared between the non threaded and threaded solutions.

General Description

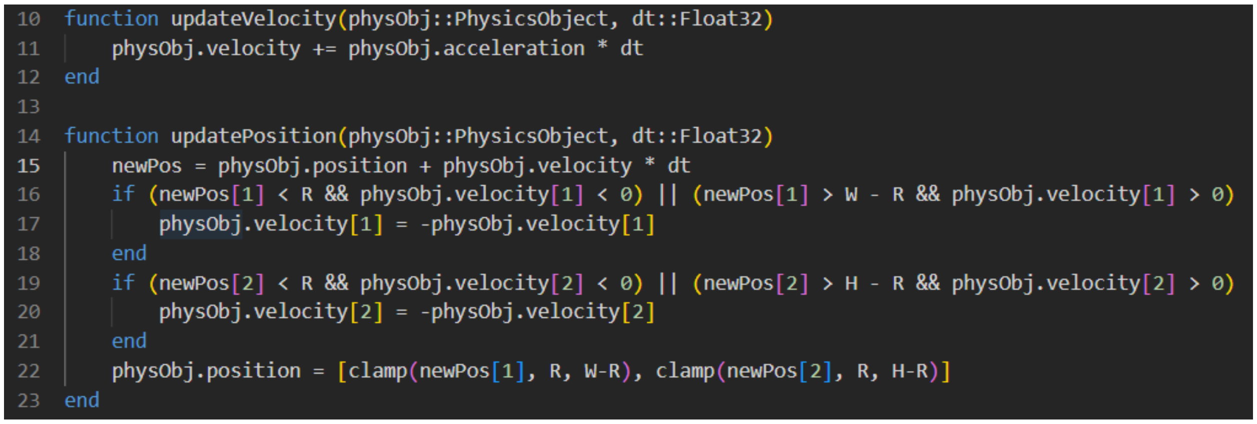
As mentioned previously, Julia is the sole language used in the experiment. To handle the final rendering of the physics simulations, the Makie Julia package (Specifically CairoMakie) is used to record animations of continuously updated figures. Julia has built in threading with Base.Threads. The ChunkSplitters package is used to distribute workload amongst the threads. The LinearAlgebra package is used for its vector dot product function. All code was run on my Dell Inspiron 7586 with an 8th gen Intel i-7 Processor.

Implementation Details



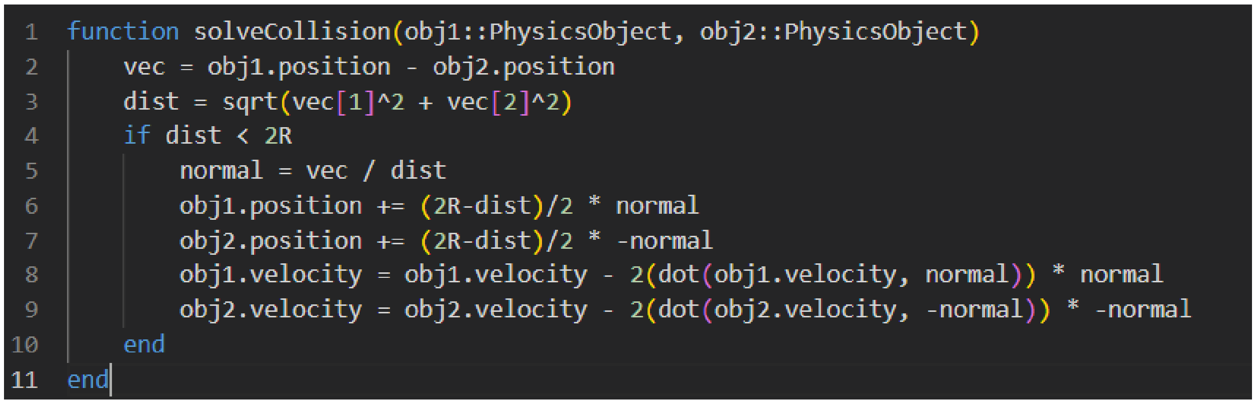
```
mutable struct PhysicsObject
    position::Vector{Float32}
    velocity::Vector{Float32}
    acceleration::Vector{Float32}
end
```

At the heart of the physics engine is a custom struct *PhysicsObject* that holds a particle's position, velocity, and acceleration each as a two dimensional Vector of Float32 numbers. The kinematic equations are implemented in the *updateVelocity* and *updatePosition* that both take a *PhysicsObject* and the change in time. *updatePosition* is a little more complicated to prevent particles from leaving the bounds of the physics simulation. The constant *R* is the particle radius; *H* and *W* are the height and width of the simulation.



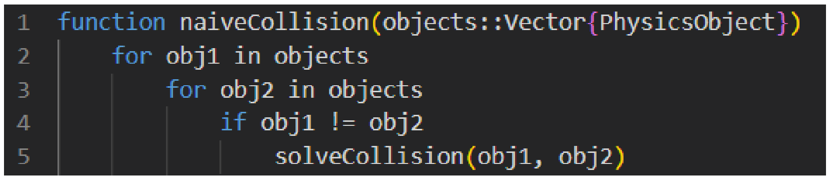
```
function updateVelocity(physObj::PhysicsObject, dt::Float32)
    physObj.velocity += physObj.acceleration * dt
end

function updatePosition(physObj::PhysicsObject, dt::Float32)
    newPos = physObj.position + physObj.velocity * dt
    if (newPos[1] < R && physObj.velocity[1] < 0) || (newPos[1] > W - R && physObj.velocity[1] > 0)
        physObj.velocity[1] = -physObj.velocity[1]
    end
    if (newPos[2] < R && physObj.velocity[2] < 0) || (newPos[2] > H - R && physObj.velocity[2] > 0)
        physObj.velocity[2] = -physObj.velocity[2]
    end
    physObj.position = [clamp(newPos[1], R, W-R), clamp(newPos[2], R, H-R)]
end
```

The *solveCollision* function handles particle collisions. Two particles are colliding if the distance between their centers is less than the sum of their radii. The distance is found by getting the magnitude of the difference of the two particle’s position vectors. If a collision is detected, the particles are adjusted to no longer be overlapping by moving each particle half the overlap distance away from each other. Each particle's velocity is then reflected tangentially to the other particle.



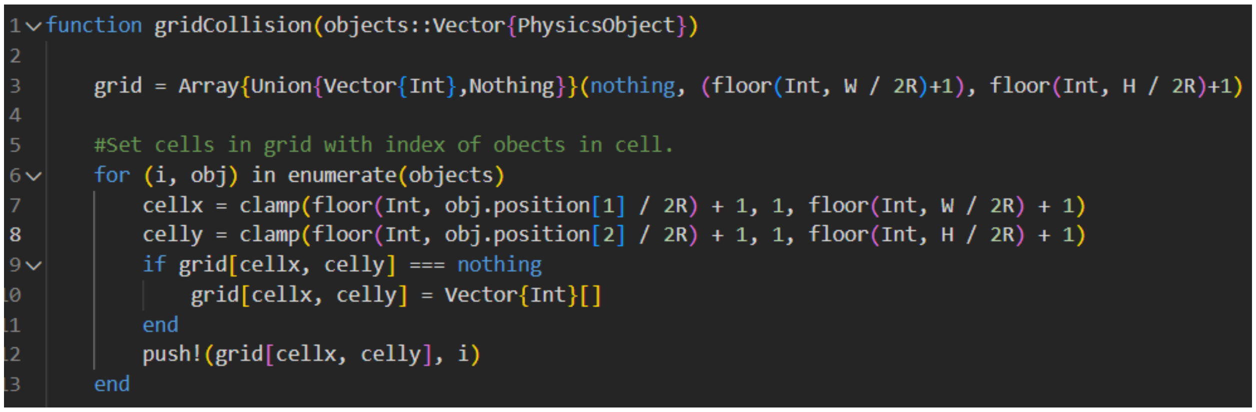
```
function solveCollision(obj1::PhysicsObject, obj2::PhysicsObject)
    vec = obj1.position - obj2.position
    dist = sqrt(vec[1]^2 + vec[2]^2)
    if dist < 2R
        normal = vec / dist
        obj1.position += (2R-dist)/2 * normal
        obj2.position += (2R-dist)/2 * -normal
        obj1.velocity = obj1.velocity - 2(dot(obj1.velocity, normal)) * normal
        obj2.velocity = obj2.velocity - 2(dot(obj2.velocity, -normal)) * -normal
    end
end
```

Detecting collisions is an expensive computing problem. The naive solution is to, for each particle, check to see if it is colliding with every other particle in the simulation. This approach is implemented in *naiveCollision.*



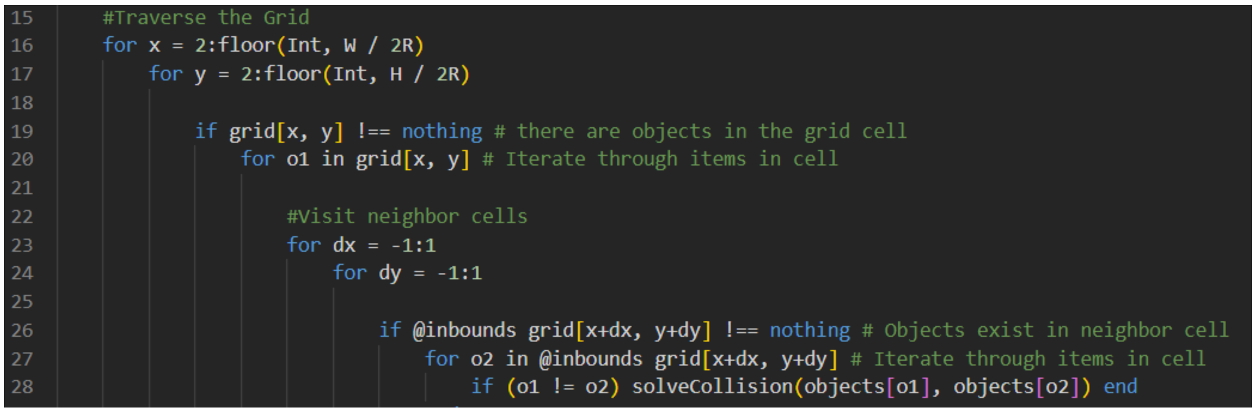
```julia
function naiveCollision(objects::Vector{PhysicsObject})
    for obj1 in objects
        for obj2 in objects
            if obj1 != obj2
                solveCollision(obj1, obj2)
```

A better approach is to divide the simulation space into a grid with cells that have the same dimensions as our particles. Collisions are then resolved cell by cell; checking to see if particles are colliding with particles in the same or neighboring cells. This approach is implemented in the *gridCollision* function. *gridCollision* starts by creating a 2 dimensional array of appropriate size and adding the indices of particles to their corresponding grid cell in the array.



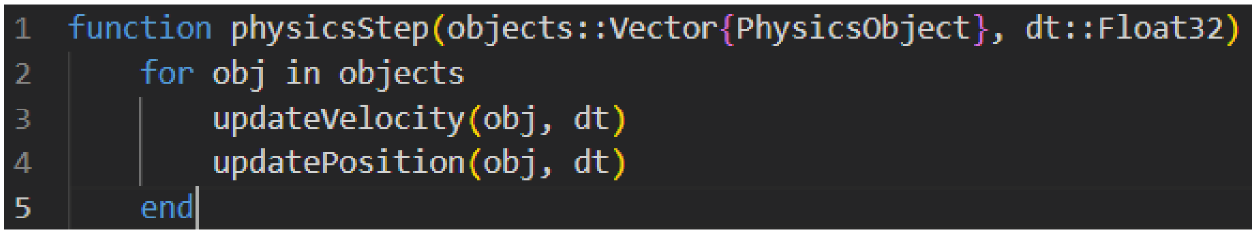
```julia
function gridCollision(objects::Vector{PhysicsObject})

    grid = Array{Union{Vector{Int},Nothing}}(nothing, (floor(Int, W / 2R)+1), floor(Int, H / 2R)+1)

    #Set cells in grid with index of obects in cell.
    for (i, obj) in enumerate(objects)
        cellx = clamp(floor(Int, obj.position[1] / 2R) + 1, 1, floor(Int, W / 2R) + 1)
        celly = clamp(floor(Int, obj.position[2] / 2R) + 1, 1, floor(Int, H / 2R) + 1)
        if grid[cellx, celly] === nothing
            grid[cellx, celly] = Vector{Int}[]
        end
        push!(grid[cellx, celly], i)
    end
```

The next step is to iterate through the grid and solve collisions within the cell and with neighboring cells.

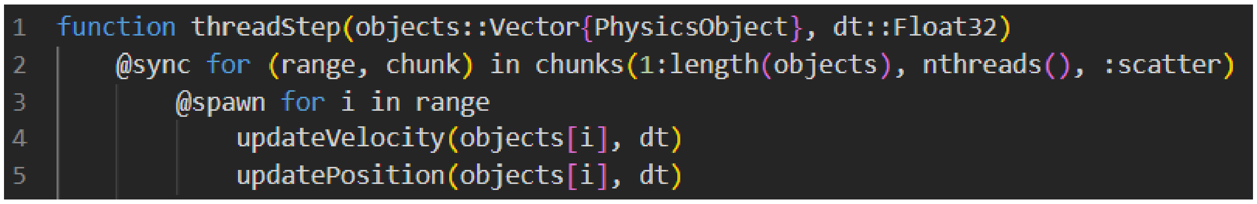


```julia
    #Traverse the Grid
    for x = 2:floor(Int, W / 2R)
        for y = 2:floor(Int, H / 2R)

            if grid[x, y] !== nothing # there are objects in the grid cell
                for o1 in grid[x, y] # Iterate through items in cell

                    #Visit neighbor cells
                    for dx = -1:1
                        for dy = -1:1

                            if @inbounds grid[x+dx, y+dy] !== nothing # Objects exist in neighbor cell
                                for o2 in @inbounds grid[x+dx, y+dy] # Iterate through items in cell
                                    if (o1 != o2) solveCollision(objects[o1], objects[o2]) end
```

There are two places where threading can be used to enhance performance. Instead of calling *updateVelocity* and *updatePosition* on object by object, the list of objects can be divided amongst threads and each thread can handle updating its portion of objects. This change is implemented in its own function called *threadStep*

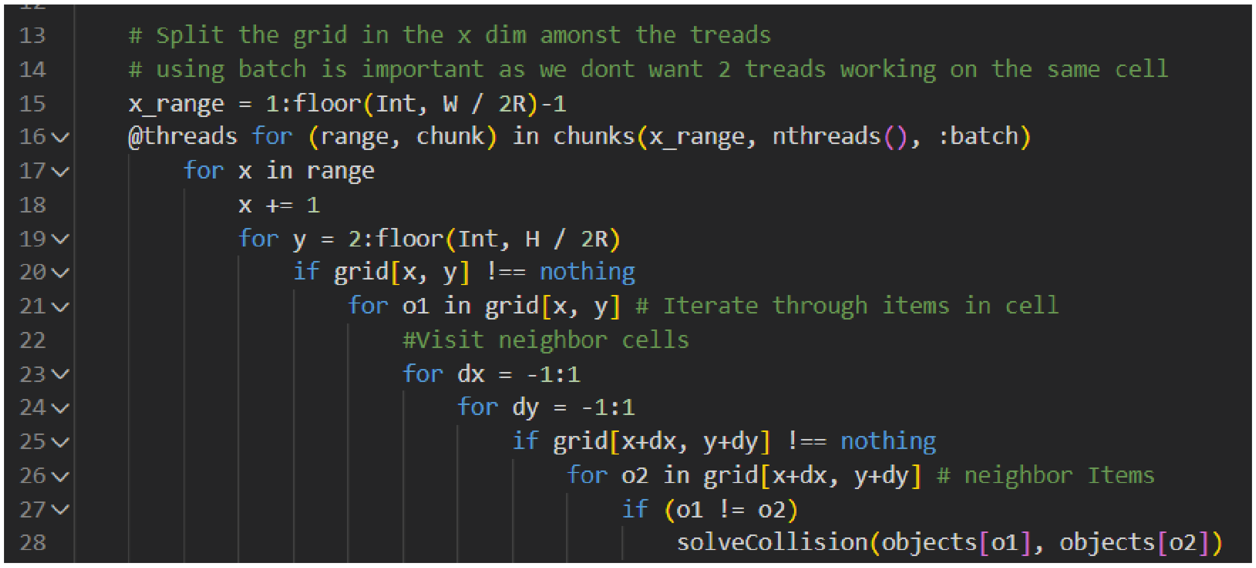


```julia
function physicsStep(objects::Vector{PhysicsObject}, dt::Float32)
    for obj in objects
        updateVelocity(obj, dt)
        updatePosition(obj, dt)
    end
```



```julia
function threadStep(objects::Vector{PhysicsObject}, dt::Float32)
    @sync for (range, chunk) in chunks(1:length(objects), nthreads(), :scatter)
        @spawn for i in range
            updateVelocity(objects[i], dt)
            updatePosition(objects[i], dt)
```

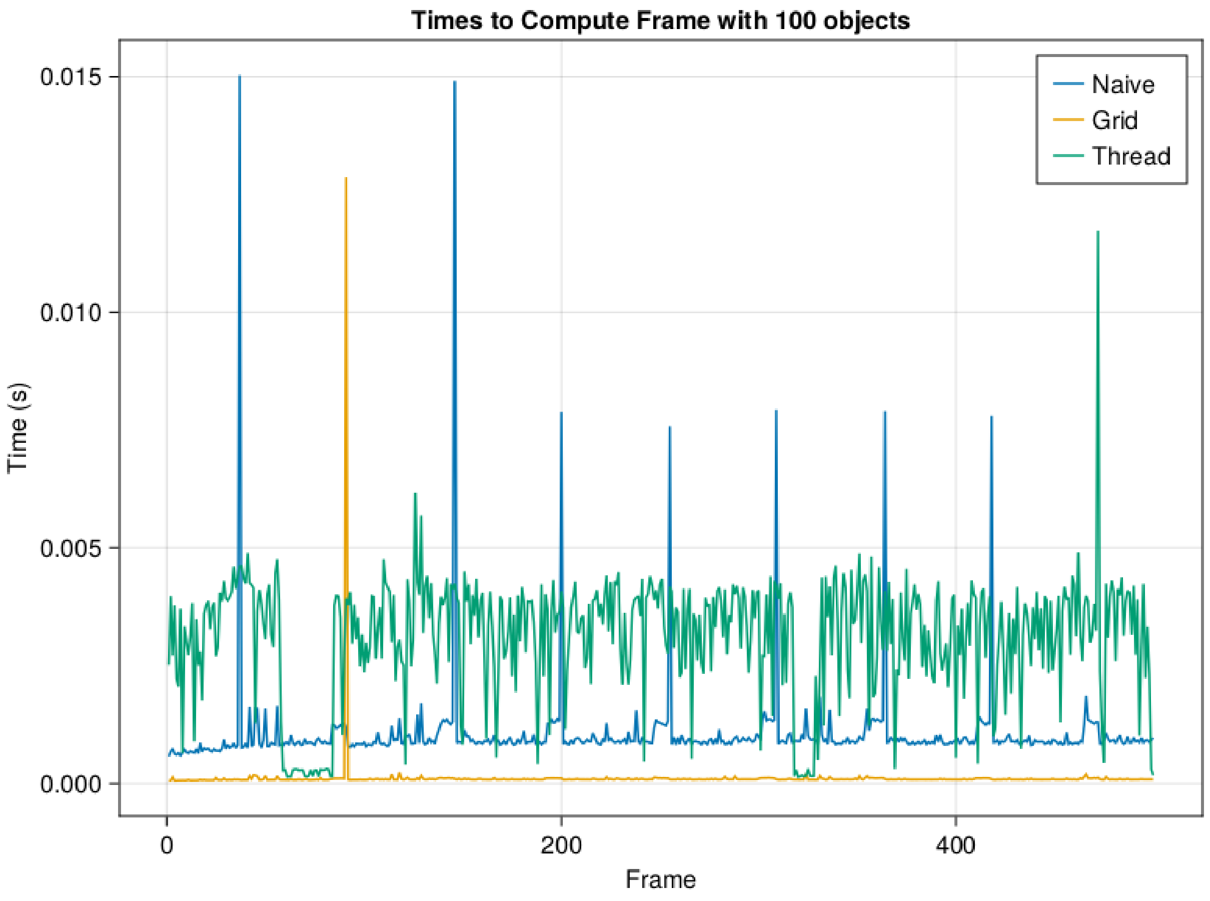
The next improvement is in the *gridCollision* function where we divide the grid in the x dimension among the threads. This is implemented in *threadCollision.*



```julia
    # Split the grid in the x dim amonst the treads
    # using batch is important as we dont want 2 treads working on the same cell
    x_range = 1:floor(Int, W / 2R)-1
    @threads for (range, chunk) in chunks(x_range, nthreads(), :batch)
        for x in range
            x += 1
            for y = 2:floor(Int, H / 2R)
                if grid[x, y] !== nothing
                    for o1 in grid[x, y] # Iterate through items in cell
                        #Visit neighbor cells
                        for dx = -1:1
                            for dy = -1:1
                                if grid[x+dx, y+dy] !== nothing
                                    for o2 in grid[x+dx, y+dy] # neighbor Items
                                        if (o1 != o2)
                                            solveCollision(objects[o1], objects[o2])
```

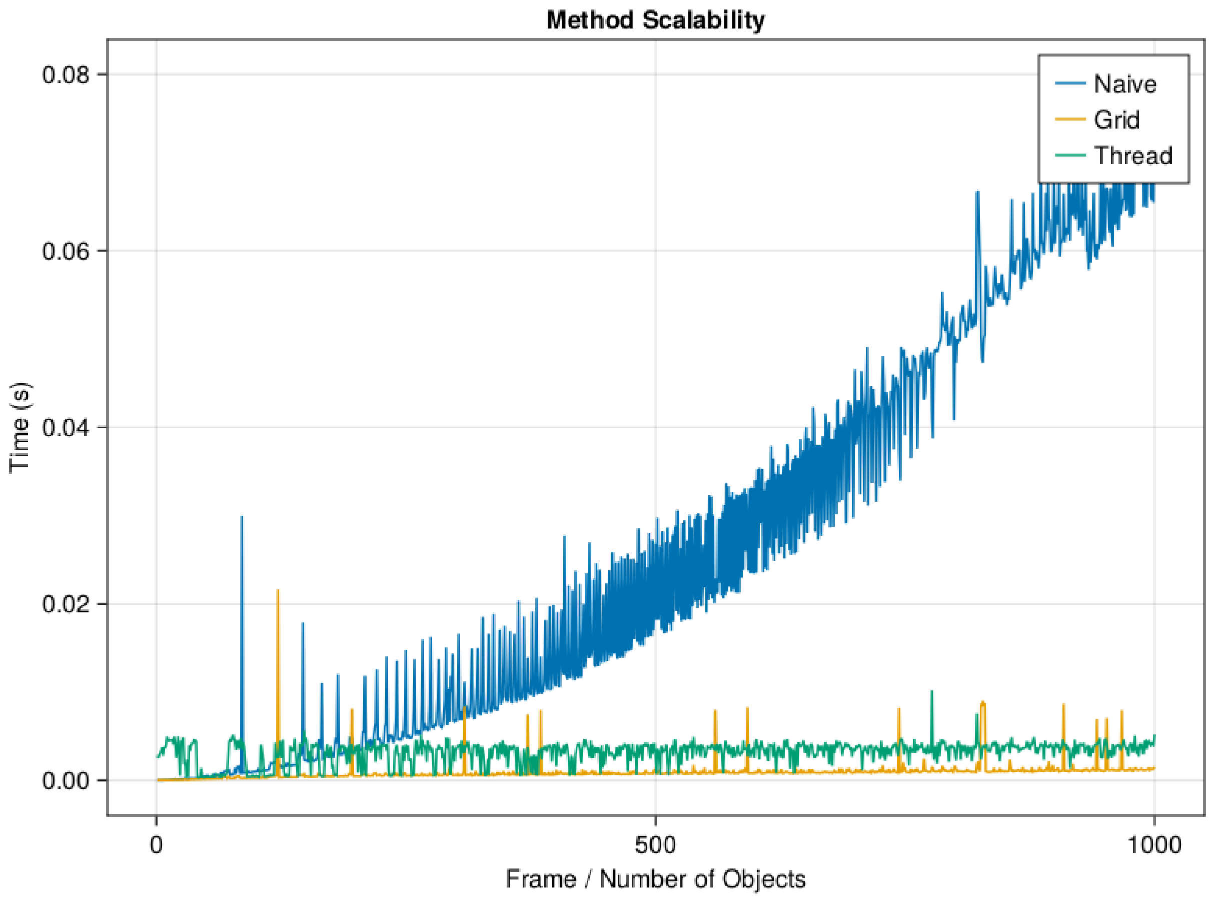
Time Performance Analysis

There are 3 physics calculation methods to test. The naive method where collision detection is brute forced, the grid method which implements the spatial subdivision, and the thread method using threaded object updates and threaded spatial subdivision. To compare the methods a list of 100 physics objects is created with random spacing and starting velocity. We then perform 500 frames of physics with each of the methods on a deep copy of the original list. The time to calculate the frame for each method is saved to an array. Plotting the times gives us the following figure.



The average times are as follows; Naive Avg: 0.0016511591 Grid Avg: 0.0001333916 Thread Avg: 0.0038975868

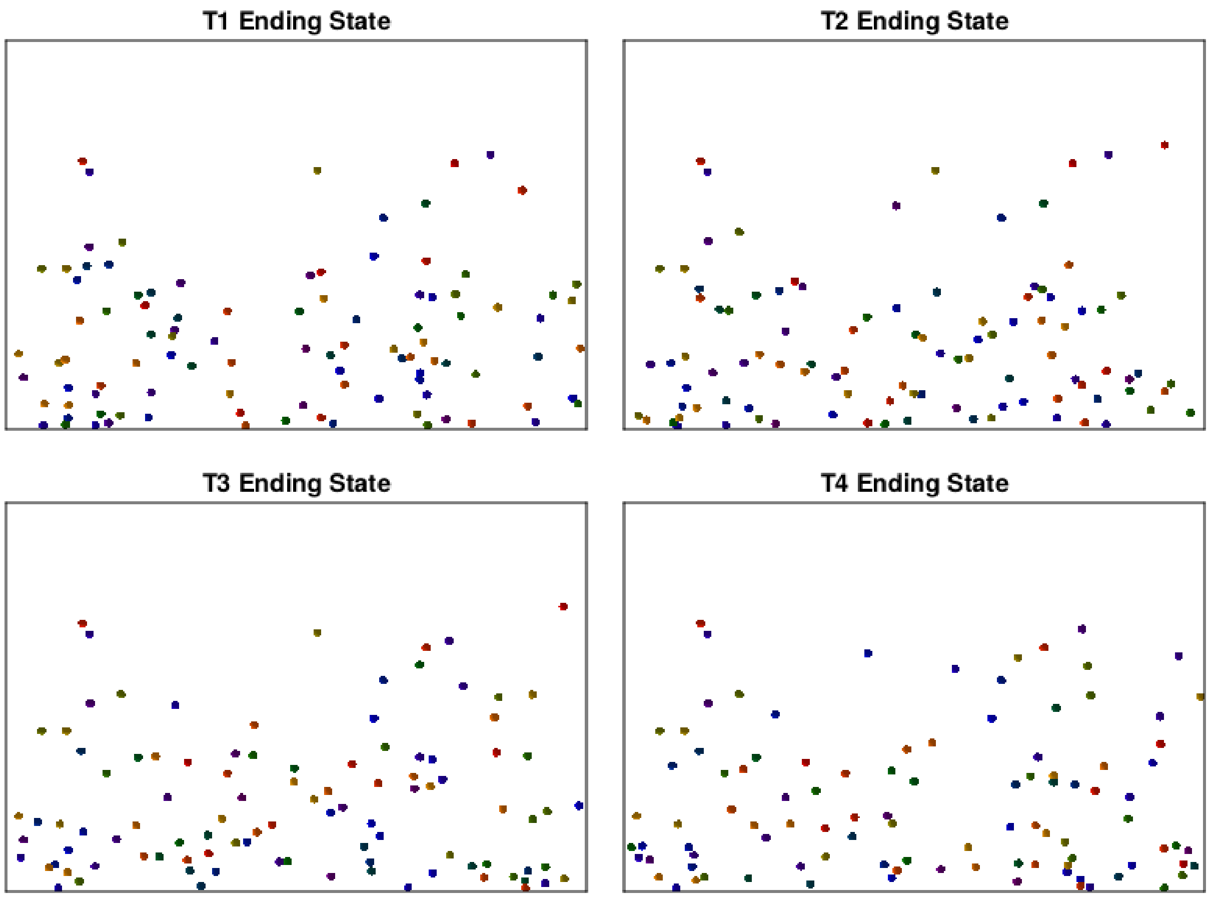
The figure above is deceptive in showing that the threads method is worse than the naive method. This is because the number of objects is very low, so solving grid collisions is similar in time to brute forcing it. To see how each system performs at scale, a new *PhysicsObject* is added after every frame. We run each method for 1000 frames which leads to 1000 objects in the simulation at once. Plotting the time it takes to calculate each frame for each method gives the figure below.



This figure shows that the time to calculate a naive frame increases exponentially with the number of objects in the simulation, whereas the other two methods increase linearly. However, the threaded approach performs worse than the grid method. I have to assume that the overhead to setup the threading is more costly than just doing it sequentially. The average time for each method are as follows; Naive Avg: 0.03282245 Grid Avg: 0.00120433 Thread Avg: 0.0044691083

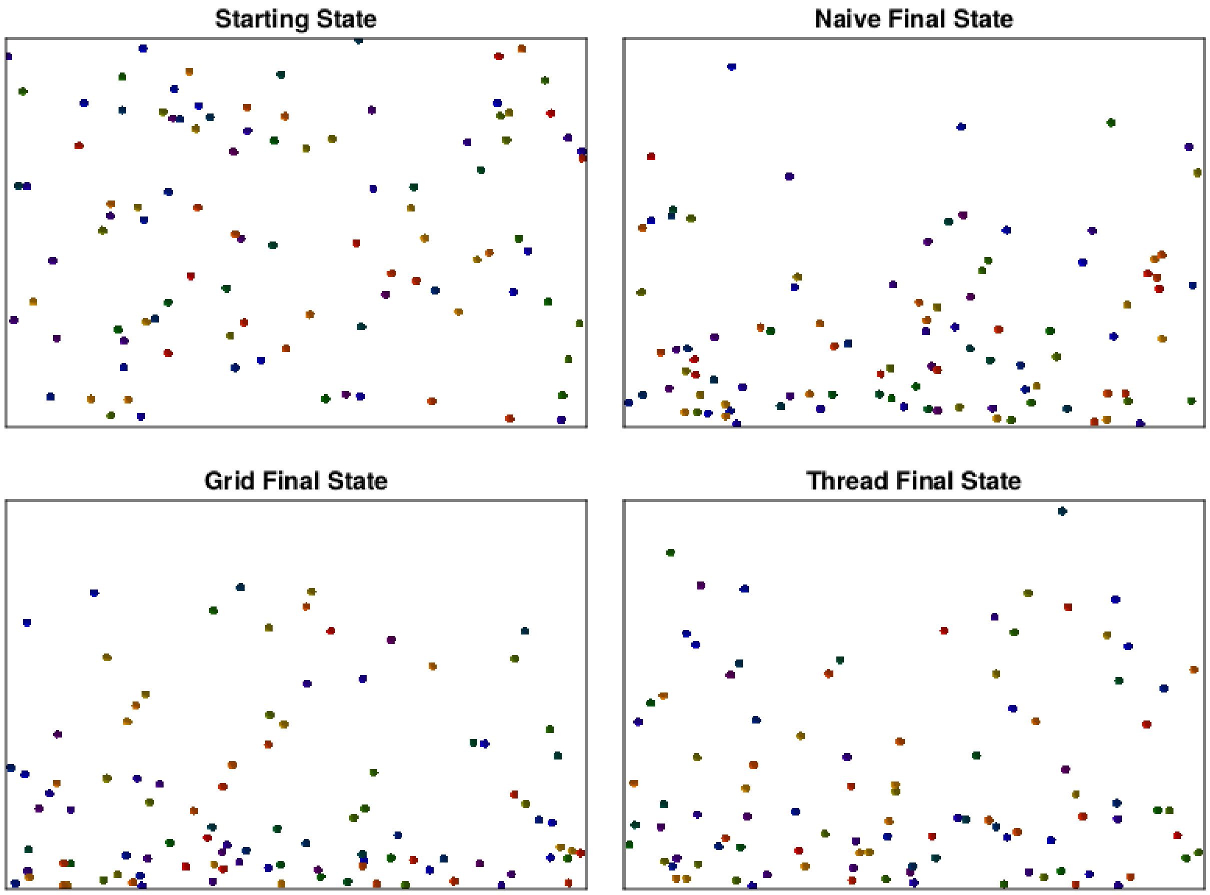
Results

The physics system should be deterministic. Meaning that given an input, the results should be the same every time for that given input. For the naive and grid method, the results will always be the same because the code runs sequentially. However, with the threaded approach we have to prevent race conditions where different threads act upon the same particle. The following figure shows the output of 4 different simulations (500 frames with 0.01 dt) on the same initial state



Although the layout of the particles are similar, they are not exactly the same. This has to be because of the order in which the particles are updated.

The following figure shows the final states of the 3 methods compared to an initial state. The difference between naive and the others make sense as the objects are updated in index order vs the grid method which updates column by column in the grid. The difference between the grid and thread method must be due to race conditions.



References

Writing a Physics Engine from scratch
https://youtu.be/lS_qeBy3aQI

Writing a Physics Engine from scratch - collision detection optimization
https://youtu.be/9IULfQH7E90

Building Collision Simulations: An Introduction to Computer Graphics
https://youtu.be/eED4bSkYCB8

Making animations and interactive applications in Makie.jl
https://youtu.be/L-gyDvhjzGQ

https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-32-broad-phase-collision-detection-cuda

https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-32-broad-phase-collision-detection-cuda