NWEN303 Assig 2 Report

Task 1 – Explaining the Model class:

Fields:

gravitationalConstant: This value effect how strongly two bodies will attract in the universe.

lightSpeed: This is the maximum speed of anything in this universe, as object approach this speed, it becomes harder to speed up. As shown by this simplified version of the formula used in the particle interact method: newSpeed /= (1 + (speed / lightSpeed)).
This shows that as speed increases, it creates a larger number on the denominator of the new speed. When the light speed is decreased, it gives the effect of viewing a larger chunk of the universe from further away.

timeFrame: This is the time between physics calculation frames. This is separate from the UI refresh rate because physics engines usually run much faster than the refresh rate to ensure that collisions happen accurately and aren't slowed down by slow drawing instructions (Though in this case the physics engine is slower than the refresh rate. This may be a bug).

p: This list stores all the particles in the universe and allows physics actions to be applied on them.

pDraw: This list stores visual representations of all the particles so they can be drawn on by the UI. This is kept separate from the list of particles and only updated atomically from the model thread to ensure that there are no errors caused by updating it while trying to draw the particles on the UI.

Actions in "step" method

Particle Interactions

In this step, every particle interacts with every other particle in the simulation. The interaction involves checking for impact/collision and applying gravitational forces between the particles.

Impacts/collisions are stored in the particles impact set, then later used when merging particles.

Particle Movement

This step applies the velocity of each particle to its position. The velocity is divided by the timeframe to ensure the movement is constant for different divisions of time.

This step also has the capability to bounce particles off the boundary when it is enabled.

Particle Merging

See "mergeParticles()" method

Graphics Update

See “updateGraphicalRepresentation()” method

“updateGraphicalRepresentation()" method:

The program has a sort of buffer that holds the locations and sizes of particles at a single moment in time. This is used so that the drawing is separated from the physics of the simulation. The “DrawableParticle” class represents one particle in the universe to be drawn on the screen, it holds the position, size and colour of the particle.

The updateGraphicalRepresentation method creates a new buffer to hold the DrawableParticles then loops over all the particles in the universe, adds them to the buffer and then atomically updates the buffer that the GUI will render.

The atomic update of the particle buffer ensures that there are no issues when other threads are accessing the list. If the buffer is updated when the GUI is looping over the array list, the GUI will finish the loop on the object as it still has the reference, but on the next loop it will use the new object reference from the pDraw field and therefore the new ArrayList object.

“mergeParticles()” method:

This method takes chunks of particles that are colliding with each other (from the particles impacting set), adds them to a set of particles that need to be merged.
This set of particles is then divided into chunks of colliding particles so multiple different chunks colliding at the same time are not merged into one particle.

The particles in each chunk are then combined into one bigger particle.

See “mergeParticles(Set<Particle> ps)” method

“getSingleChunck(Particle current)” method:

This method finds all particles impacting with the specified particles, and any particles impacting with those particles, i.e. it finds a chunk/group of particles that are impacting with each other.

This is achieved by iterating through all the particles colliding with the particles currently in the impacting set, then adding those particles to the set. When no more particles are added to the set, all impacting particles have been found.

“mergeParticles(Set<Particle> ps)” method:

This method returns a single particle created by combining multiple particles into one big one.

It ensures that the centre of momentum remains constant by adding the mass, position (weighted by mass), and the velocity of the smaller particles together to form the values for the larger particle.

Bugs in particle simulator:

Not using real time measurement

The move and interact methods inside the particle class use the timeframe variable to ensure that when the time frame is changed, the velocity remains constant. The problem is that they use a fixed value when they should be using a measured time from the last physics frame. If something prevents the method from being run at intervals equal to timeFrame (slow computer, low process priority etc), the physics of the simulation will be inaccurate and not constant.
Note: Since the timeFrame value is being used as a speed setting for the universe, it could be multiplied with the measured time to produce a more accurate version of the same effect.

Interact and move in wrong order

Inside the step method inside the model class, interact should be performed after move is performed, this will ensure that particles are not waiting for another time frame before a collision is registered. It also means that the user will perceive the gravity coming from the wrong place because the particle has moved from where the gravity has been applied. This is not noticeable for small values of the time frame, but larger values will cause issues.

Not checking for NaN value before using in calculations

The interact method inside the particle class should check that dirX is not NaN before calculating newSpeedX, and that dirY is not NaN before calculating newSpeedY. This means that the CPU time is not wasted calculating values that are not used.

Physics loop running slower than screen refresh

The redraw of the universe is scheduled to run every 5ms, but the physics loop has a minimum delay of 20ms. There is no point in making the screen refresh rate faster than the speed of the physics engine because the application will waste resources redrawing the same thing since the particles have not moved or changed at all. The physics engine should be sped up providing a more accurate simulation or the screen refresh rate should be slowed down so that it is the same speed or slower than the physics engine.

Task 2 – Explaining the parallelism for provided GUI class:

Explanation

Inside the main method, the GUI is created and run on the Event Dispatch Thread using the SwingUtilities.invokeLater method.

The GUI class implements the Runnable interface so after the object is used to create a thread, the run method gets called. The run method initializes the GUI and adds all the controls to the JFrame. It also uses a scheduled thread pool executor to schedule the simulation repaint every 5ms with a start-up delay of 500ms.
The repaint runnable is run on a thread inside the shedulerRepaint pool but it calls SwingUtilities.invokeAndWait in order to do the painting on the Event Dispatch Thread, this ensures that waiting for 5ms does not freeze the UI but also that the code that interacts with UI Swing elements is run on the UI thread (Interacting with non-thread safe Swing components from outside the Event Dispatch Thread will throw an exception)

The main method also schedules the MainLoop task to be run on another thread inside the schedulerSimulation pool after 500ms.

The MainLoop class implements the Runnable interface so when the object is used to create a thread, the MainLoop.run method is called which contains the physics loop for the simulation. The physics loop measures the time taken to step the model, then subtracts that from the minimum loop time and then waits for that loop time if it is larger than 1ms. This is used to set a speed limit on the loop.

Data contention

The only data passed between threads is the list of DrawableParticle pDraw inside the Model class. This data is written inside the Mode class on the physics thread and then read/drawn inside the Canvas class on the UI Event Dispatch Thread.

Even though the data is shared, no contention occurs because object aliasing ensures that if the UI is reading from the pDraw list and then the physics loop updates it, the UI will continue to read from the old list until the next time it redraws when it will grab the new object alias. (The assignment seems to imply that contention is occurring but the only definitions I can find say the contention is when a thread must wait for another to use a resource. See IBM thread contention)

This is the many reads one write pattern discussed in the lectures.

The parallelism is implemented correctly as the data is written to a temporary list inside the physics thread and then the pDraw object alias is swapped atomically to the new list of DrawableParticle. The pDraw object is also marked as volatile to avoid issues with CPU caching.

Task 3 – How to introduce parallelism:

How I plan to add parallelism:

I plan to add parallelism by replacing the steps where particles are iterated over sequentially with a parallel iteration. For example, I will be replacing the interaction loop with a function that does the interaction in parallel. The program will wait for each step to be completed before starting the next step to ensure that the writing of the particle data is done before the next step reads it.

Why will it help?

This will help because the CPU can utilise multiple cores while calculating the values for the particles, thus making the simulation run on the CPU more efficiently and utilise more processing power when needed.

Currently the model only runs on one core because it only calculates the values for one particle at a time.

What kind of data contentions will need to be resolved?

The list of particles will be shared over multiple threads so I will need to ensure that there are no data contentions here.

The particles also have a set of impacting particles inside them so I will need to make sure that particles do not modify any particles except themselves, and that the other particles do not rely on the data being modified in the same step.

How am I sure that there is no hidden aliasing created?

I made sure to create new particles using values from the old particles when modifying any particles in parallel.

Task 4 – Implement ModelParallel:

The main resource I used were parallel Java Streams. I was able to get the algorithm with regular sequential streams, then add the .parallel() method onto the stream to make it run in parallel.

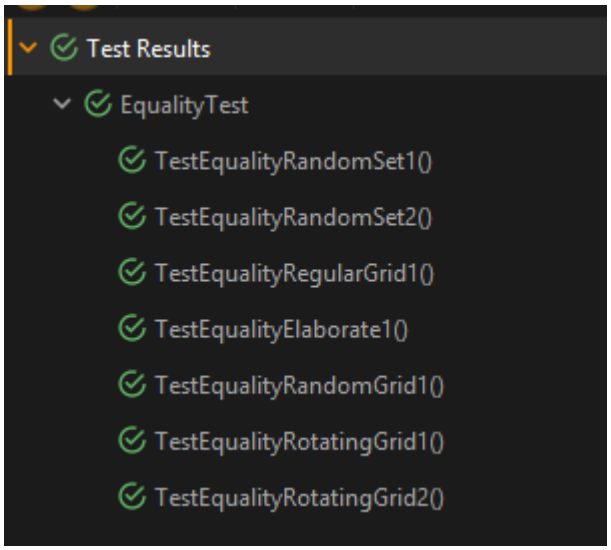
Task 5 – Testing equality/correctness:

In order to ensure that ModelParallel behaves exactly as Model in all the situations, I designed the following automated testing strategy:

I created unit tests that create a ModelParallel and a Model with the same data set then step them both 10,000 times. At each step, the particles in the model are compared to check if they are equal. I ran into issues with very small differences in the floating point values of the particles so I created a method called similar(Particle p) which checks if another particle has all of its values within a threshold of the current particle.

I created multiple of these tests using various data sets.

This gave me a high level of confidence because the testing shows that the models behave the same with a wide range of input and over many steps.



Task 6 – Testing performance:

In order to check that ModelParallel is more efficient that Model, I designed the following automated testing strategy:

I used code from assignment 1 that measures the time taken to run something.
I was not able to get the parallel version of the program to be faster than the sequential version. Even using a regular stream with foreach was over 10 times slower than just writing for(type a: b). Maybe this is because the computer I am using does not have many cores? I have no idea.

References:

IBM Thread Contention:
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzahw/rzahwconco.htm