NWEN303

Assignment 2

Universe

William Kilty
300473541

Task 1:

The Model class represents an instance of the particle simulator ‘world’. It has a size, which the screen is sized to fit. It has a Gravitational constant, which scales the strength of the attraction between particles. It has a light speed, which is the maximum speed a particle can travel, and therefore scales the speeds of particles. It has a frame timing scale, which determines how much movement happens between frames, with more movement leading to less accurate simulation due to particles ‘clipping’ through one another.

The model holds a list of all the particles present in the simulation, with which to manipulate, and a list of particles to draw to the screen. This distinction allows for the drawing thread to draw to the screen without threat of simulation happening during drawing, creating artefacts or throwing errors.

Particles are objects represented as a circle, with a mass, a velocity, and a position. They will move around, and attract other particles based on the gravitational equation $F = G\frac{m_1 m_2}{r^2}$. When they collide, they merge; combining their mass, position, and velocity.

The method `Model.step()` consists of four steps:

1. Interaction. Each particle is given a list of every other particle, and uses the equation declared above to adjust its velocity for each other particle. If a particle is colliding with another, it will add the other particle to its list of impacting particles.
2. Movement. Each particle will add its velocity to its position, moving the particle.
3. `mergePaticles()`. For each particle, we check if it has collided with anything. If it has, we remove it from the model. We then go through each of the removed particles, and gather them into ‘chunks’. (A chunk is a single group of particles all connected through some chain of touching). Each chunk is then turned into a brand new particle consisting of all the combined fields of the chunk particles.
4. `UpdateGraphicalRepresentation()`. We simply add all the particles to our list of particles to draw, so that the GUI class can draw the simulation state.

Bugs:

There is a bug in the mergeParticles method. The use of a hashset when merging particles means particles are merged in essentially ‘random’ order. With the merging code, this results in a new combined particle that’s slightly different depending on the order of combinations. - In order to fix this bug, replace all instances of HashSet with ArrayList. Including in Particle.java. This ensures order is retained and the maths will perform identically each time

Task 2:

The GUI.java file holds a main method which creates a model, and gives it to two threads. One holding a GUI object to handle drawing the screen, and the other holding a MainLoop object to handle running the simulation.

The MainLoop object has a while loop which attempts to perform a simulation step every 20 milliseconds. The steps, as discussed in task 1, result in writing to a volatile list of particles to draw.

The GUI object sets up the screen for drawing, and then schedules a worker to update the screen every 5 milliseconds.

Repainting the screen calls the paint method in the GUI’s canvas object which looks through the volatile list from Model, and draws each of the particles to the screen. This list is shared between both threads, making it the contended data. The contention pattern is, in this case, supposed to be ‘many reads, one write’.

This is implemented incorrectly, as writing particles to the list happens over multiple atomic actions, meaning theoretically a frame could be drawn from an incomplete list - leading to some particles not being drawn to the screen randomly.

The volatile keyword is important as it specifies that the canvas is not to cache an old version of the list between frames. A fresh read of the list will be performed every frame.

Task 3:

There are three main areas which can be parallelised within `Model.step()`.

- Interact can be `parallelStream()`ed, dividing up the loop through each particle into separate threads. This can be done in the step method, but not in the interact method for each particle, as there are unavoidable contentions.
- Move can be `parallelStream()`ed, dividing up the loop through each particle into separate threads.
- Merge Particles can also be `parallelStream()`ed. The list 'deadPs' can be constructed this way, and so could the set 'tmp' in getSingleChunk - However, this is unnecessary, as the number of particles touching at once are unlikely to be in the hundreds whereupon parallelism would be worthwhile.

This will help with interacting, merging, and moving, as these processes will all be able to happen in parallel, simultaneously, instead of looking one by one at each of the particles in the simulation.

These parallel systems are ideal, as they mostly work in isolation from one another. The only contended data would be in the form of writing to the velocities of particles in the interact method. This would need to be resolved by collating all the changes in a set, and applying all the changes after the fact. However, this would change the order of velocity changes from sequential, and therefore would not be a matching simulation.

Aliasing is not an issue here, as nothing parallelised relies on writing or recently written variables. Any time there's a 'fork', all the variables needed are set in stone. Writing only happens after a join, and where that data isn't needed in parallel anymore.

Task 4:

My design decisions were fairly straightforward following my plan. I used ParallelStreams in order to benefit from automatically scaled parallelisation. If there are few enough particles involved, it'll just use the one thread. But if there's lots of activity, many threads will work better. ParallelStreams have this functionality in addition to being compact and fast to implement, which drove me to the decision to utilise them.

Task 5:

I used the JUnit testing library to assert that both model classes resulted in identical answers. This included writing a number of test cases to test different scenarios between the models, and using JUnit's coverage feature to make sure I was covering all the elements I needed to cover.

In order to ensure that ModelParallel behaves exactly as Model in all the situations I designed the following automated testing strategy: Create two models; one parallel and one sequential; and run them against each other with the same number of step calls. The results should show that both models have identical lists of particles.

This gave me a high level of confidence because no matter what I got the models to do, if they end up with the same list, the models have to be doing the same thing as one another. Plus control testing to see if deliberate divergence results in a failed test assures me that my testing methodology is correct.

This is also where I discovered a bug in the model code. The use of a hashset when merging particles means particles are merged in essentially 'random' order. With the merging code, this results in a new combined particle that's slightly different depending on the order of combinations. I temporarily fixed this bug, and the tests all passed, indicating that this was indeed the cause of divergence.

I assure you that the parallelisation is correct.

Task 6:

I used the JUnit testing library to assert that parallel modelling was faster than sequential. This included writing a number of test cases to test different scenarios between the models.

I used a modified version of the performance test from assignment 1. This works by running the models through 1,000 times to ‘warm up’ the process (Essentially give it priority), and then take the time before and after running it through 200 more times. The difference between these two measured timings gives a fairly accurate representation of how long a model takes to run a scenario.

The results I obtained through this were:

```
On the data type RegularGrid
Sequential Model takes 59.674  seconds
  Parallel Model takes 15.728  seconds
On the data type Elaborate
Sequential Model takes 170.402 seconds
  Parallel Model takes 40.871  seconds
```

As you can see, my parallel model processes the simulation significantly faster, with speeds up to four times as fast as the sequential model.

In order to check that ModelParallel is more efficient than Model I designed the following automated testing strategy: Utilising the performance test suite from Assignment 1, and modifying it to test parallel against sequential in a regular grid dataset and an elaborate dataset.

This gave me a high level of confidence because allowing the process to warm up for a number of cycles gives an accurate measure of the speed of the process at peak efficiency, plus running it 200 times gives an aggregate score that can be used to find an average time more accurate than any single test. Given the significance of the variance in timings, it seems obvious that the parallel method is significantly faster than the sequential method.