NWEN303 Project1

Hayden Richardson 300262622

Task1 Understand Model.java

Give details on variables:

- public static final double size=900;
    - Used by Canvas class to determine the width and height for the preferred size's dimension.
- public static final double gravitationalConstant=0.1;
    - used in Particle.interact(). Weights the amount of gravitational pull between objects/mass.
- public static final double lightSpeed=10;
    - Used in Particle.interact() to scale the new speed of mass/an object.
- public static final double timeFrame=1;
    - Used in Particle.move() to scale how far an object should move in a frame based on the time per frame.
- public List<Particle> p=new ArrayList<Particle>();
    - The list of all particles in the model.
- public volatile List<DrawableParticle> pDraw=new ArrayList<DrawableParticle>();
    - Used by Canvas.paint() to draw each particle.

Explain the meaning of the four main actions in Model.step():

- for(Particle p:this.p){p.interact(this);}
    - This step calls interact on every particle in the model. The interact method calculates how the particle is affected by the gravitational pull of all of the particles in the model. Interact then updates the particle's speedX and speedY fileds based on this gravitational pull. If the particle is impacting another particle, that particle will be added to the impacting list.
- mergeParticles();
    - This method goes through each particle and adds it to the list of dead particles (deadPs) if it is impacting any other particles. Then it removes all of the dead particles from the list of particles in the model (this.p). Then for each of these dead particles, it finds all of the particles that are impacting together and adds them into one list (to be merged together). The dead particles in this clump list are removed from the deadPs list to avoid duplicates, and they are then passed as a parameter to the mergeParticles() method, which combines their mass and velocity, and returns a new particle. This new particle is then added into the list of live particles in the Model (this.p).
- for(Particle p:this.p){p.move(this);}

    - For each of the now alive particles in this.p, the particles has move() called on it. This moves the particle by modifying the particle's x and y fields. The amount to move is based on how much time passes during each frame, and the direction is based on the speed which it is travelling in the directions of x and y at.
- updateGraphicalRepresentation();
    - Now we want to update the list of Drawable Particles so that the Canvas can draw the updated state of each particle. It goes through each particle to be drawn and adds it to the list of DrawableParticle (this.d) by constructing a DrawableParticle using the particle's x, y, and mass (it also tells them to be drawn in orange).

Explain how the merging of particles work:
This method goes through each particle and adds it to the list of (soon to be) dead particles (deadPs) if it is impacting any other particles (this is calculated in the interact() method by adding other particles to a particles set if it is going to impact with them). Then it removes all of these dead particles from the list of live particles in the model (this.p). Then for each of these dead particles, it pops off the particle and finds all of the particles that are impacting together (or connected, via another impact, to particles impacting together) and adds them into one list (to be merged together). The dead particles in this clump list are removed from the deadPs list to stop looking at the particles again, and they are then passed as a parameter to the mergeParticles() method, which combines their mass and velocity, and returns a new particle. This new particle is then added into the list of live particles in the Model (this.p).

Bug:
I believe that there is a bug in the step method of Model. The positioning of the mergeParticles() is in the wrong order relative to p.interact and p.move. This is because we should calculate the speed and gravitational pull of a particle and then move that particle as one 'atomic' operation. We can then merge the particles together at the end so in the next step their now combined speed and gravitational pull are calculated as one mass. To fix this bug we simply need to move the mergeParticles() method call below the line 'for(Particle p:this.p) p.move(this);' and above the line 'updateGraphicalRepresentation();'.

Task2 Understand Gui.java

The Gui's main method firstly creates a Model. Then it shedules the main loop to begin running in 500 milliseconds. This gives the SwingUtilities.invokeLater() method 500 milliseconds to create the GUI before the particles start moving. If the GUI does take longer than 500 milliseconds to build, then it will just miss out on displaying the first frames of the simulation as it reads the latest version of the list of Particles each frame.

The BuildGui class uses the Swing library to create a JPanel Canvas on which everything is drawn. The Gui has a run method as it is added to the scheduler which is a ScheduledThreadPool. scheduler has a core pool thread size of 2 so that the Gui and MainLoop are never dropped even if idle for a period. The Gui's run method adds new Runnables to the schedular to run every 25 Milliseconds. This is essentially the frame rate of the simulation. If the MainLoop hasn't finished updating/calculating the next state of the Model in 25 milliseconds, then the next frame will simply display the same content. The MainLoop is forced to wait a full 25 milliseconds between calculations to stop it getting ahead between frames. This is done by having the MainLoop's run() method loop infinitely, calling step() on the Model every 25 milliseconds.

The way that parallelism is being used in the Gui follows the Readers/Writers contention pattern. The data being contended over is the Model and all of its particles and their states. Any contention issues are dealt with by only having one writer and many readers. The MainLoop is the writer as it updates all of the particles' positions in the Model object, in particular the list of DrawableParticles (pDraw). The BuidlGui makes lots of Runnable readers who read the state of pDraw to update the Swing canvas every 25 milliseconds.

This parallelism is properly implemented as the readers can only display the latest complete, coherent state of the Model. This is because the writer MainLoop can atomically update everything which is to be read by updating pDraw.
.

Task3 Discuss how to introduce parallelism

(a) How you plan to add parallelism in the algorithm.
In the Model class, the vast majority of the work being done is by Particle.interact() in the step() method in Particle. Therefore I intend to calculate each particle's interact() method on a separate thread using a CachedThreadPool() and getting the futures from each of the particles. The other place I considered injecting parallelism was the similar situation of calling a method on each particle - the calling of the move() method. However when looking at the move() method, there is only 2 statements to be calculated (or 8 if you want bouncing boundaries). These two simple statements involve assignment and division and are therefore only O(1). So the efficiency of where we call move() in step is already O(n), and by running each method call on a separate thread, not only are we still looking at O(n), but the overhead cost of creating all of those threads is likely more time consuming than simply doing the divisions and assignments sequentially.

(b) Why is it going to help in simulating particle moving, attracting each other and merging?
step() is currently being called sequentially and for every Particle, it has to iterate over every other particle in the Model, and do several calculations. This makes it very inefficient (to the order of $O(n^2)$). By running these method calls concurrently, the actual time taken to compute will be pushed towards O(n) depending on how many of the threads run together. Essentially because there is so much work to do in the interact() method, it is worth the time/overhead of creating a thread to perform this method alone.

(c) What kind of data contentions you will need to resolve?
The interact method reads each Particle, but it doesn't write any on the changes to the list of Particles that affects the other threads. Instead we can just wait for all of the threads to complete and then carry on with our calculations. The other methods such as mergeParticles() and getSingleChunk() cannot be made parallel in the same way as they would cause data contentions because they would have multiple readers and writers on the same Model.

(d) How are you sure that there is no hidden aliasing creating unpredicted data contentions?
Because none of the interact() method's calculations rely upon any fileds from other Particles other than their mass and x, y positions. The only fileds changed in interact() are the speedX and speedY, so there are no possible data contentions between the threads.

Task4 Explain your design decisions

I used a CachedThreadPool to manage my threads as this is a great library to use in place of doing threads myself (which would usually go very badly). I could also have used another ExecutorService such as a FixedThreadPool, but without knowing the capabilities of the machine running the program, nor how many threads would be needed, so I used a CachedThreadPool. I created a map from Particle to Future<?> rather than a List<Future<?>> as the map allows me to iterate over the key set, to check if the value (Future.get()) is ready to go. Once all of the futures are computed, then the method carries on like before. all I have to do is submit a new Runnable to call p.interact() on the model (I couldn't use this as this inside the Runnable was the Runnable not the Model, so I stored the Model in a higher scope).

Task5 ModelParallel Correctness

I used a JUnit test to run all four configurations of Model both sequentially and in parallel. I then check to see if every particle in the two Models are exactly the same in mass, position and velocity. I set the number of frames to run at 500 as this is more than enough time for particles to interact and merge (allowing time for errors in my parallel code to cause inconsistencies). I have also put a test in that runs for two sequential implementations of a Model. This gives me confidence that if all five of my tests were to fail, that it was not an error to do with my parallel implementation, but rather an inconsistency caused by a bug in the rest of the system.

Task6 ModelParallel Efficiency

To test the efficiency of the parallel implementation, I figured that all of the work we care about speeding up is done in the step() method. So I created a Model for every configuration in both sequential and parallel. These were each timed to do 1000 steps using the System time. I was able to do a large number of steps as without the 25 millisecond per frame limit of the Canvas, the calculations can be done very quickly in real time. I report the results to System.out giving values of the total time spent executing sequential and parallel Models, and then also the average time spent per parallel vs sequential model. I then report the difference between the sequential time and the parallel time to show how much time was saved by my parallel implementation of step(). Lastly I give the average time saved over all four configurations. I feel this figure is particularly useful as it takes into account four different configurations and their different features regarding what makes it more efficient. I am quite confident that my work is efficient as the amount of time saved on average was quite significant (at least 40% faster in one test). This being the larger bit of work for the whole MainLoop means that this will increase the efficiency of the simulator as a whole.