***NWEN 303 Project 1 Report***

**Task 1: How the model.java class works**

Beginning with the fields at the top of the class, 'size' represents the size of the GUI window in width and height. It's default value is 900, i.e therefore a 900 x 900 window is created using this field.

The fields gravitionalConstant, lightSpeed and timeFrame are all used in the logic of the particle in the 'interact' method.

The gravitionalConstant constant field represents the attraction that particles have to other particles. It is a constant acceleration force.

The lightspeed constant how quickly the particles move in a certain direction accelerated by the gravitionalConstant. Thus these two work in a counter balance effect.

The timeFrame constant controls the speed of the particles, i.e how far each particle can move in one given timeFrame. The higher the value the shorter the timestep.

P is the list of Particles in the model simulation. The particles themselves contain business logic that is used in the model to determine how they move, act and collide with other particles.

pDraw is a list of DrawableParticles. Each entry in this list is equivalent to the same indexed Particle in list 'p'. Each DrawableParticle has unique visual attributes relevant to the GUI but not relevant to the model. This list is listed as volatile, this is because of the structure of the mainLoop which calls step() when pDraw is updated. Essentially because there are 2 threads running the simulation, one for the GUI and one for the model, we need to make sure that the data that the GUI is using to draw is not cached. This is why the pDraw is atomically updated at the end of updateGraphicalRepresentation() too to avoid timeSlicing issues where the data drawn and the data stored in 'p' do not match.

The step() method operates like a tick method. It is called via the MainLoop class inside the GUI class and runs on its own thread. It has four purposes. In order, it first calculates new speed and direction values for each particle (including collision information). It then merges those particles that are now colliding in the model. It then moves (adjusts their position) of all particles based upon their speed and direction in a timeStep and finally updates the list of pDraw particles to reflect the changed data in 'p' so when pDraw particles are drawn, they reflect the state of the model data (p).

**How the algorithm works for combining particles that collided:**

It works by making each particle interact with every other particle (not inc. itself) in the simulation via a nested loop (second loop contained in particle class). Going deeper into this, if we look at the interact() method we can see new speeds and directions etc are calculated for each particle. One thing that is important that is calculated is if any other particles are impacting with us (our particle), if they are, we add them to our own personal impacting list for our particle.

Stepping back out to the step() method, we merge the particles. In the previous section of step() every particle touching any other particle was added to their impacting list, thus this meant they were impacting something. In mergeParticles() we look through every particle and look at their impacting list, if it's not empty they're impacting with another so we add it to a 'deadPs' stack and after iteration is complete we remove all of these from 'p', the main list of particles in the simulation model.

We then call getSingleChunk() on every element in deadPs, this returns a set of Particles to which any element that is touching the popped element is returned. Also any element that is touching any element that is touching the popped element is also returned etc. These particles are then removed from the 'deadPS' set of dead particles. Once removed, we have all the particles that need to be combined. We called the mergeParticles() method which takes into account masses and speeds of all the particles and then we add this combined particle to ‘p’, our main list of particles.

\_\_\_\_\_\_\_

**Task 2: How does GUI.java work?**

Gui.java contains an inner private class called MainLoop too. This description will include both of their workings.

**What does it do?**

It splits the task of displaying the View part of the MVC and the Model into two separate threads. Thus no contention between threads occcurs as the first thread's (gui) only purpose is to redraw the panel and the second (MainLoop) is to update the model.

Gui is a JFrame class that implements runnable. This means that it can be run on a thread defined by 'scheduler', the 'newScheduledThreadPool'. This Gui extending JFrame means it can support Swing views and JPanels so things can visually be drawn to a window. This is done by creating a 'canvas' class which is just a Swing JPanel, which has a simple paint method to iterate through the DrawableParticles of the model and draw ovals in their positions. Then repaint() is called on a 'scheduleAtFixedRate' applied via the scheduler. Schedule at fixed rate works by executing it every n time whether or not the last repaint has finish or not taking into account how long the last repaint took.

MainLoop is a little different. It implements Runnable only so has the ability to be run on a separate thread. It's running on the same ExecutorService that the 'gui' class is running on but on a different thread. MainLoop has a reference to the model (which holds the simulations data). The run method is run forever on that thread. The run method handles the timing for updating the model data so that this takes the same amount of time each time it loops. For instance, it takes the current time, then calls model.step() stepsForFrame times. It then calculates the time elapsed doing this. If the length of each frame minus the time taken to update the model is > 1 (i.e updating model took less time than the constant time required for each frame), then we sleep the thread for that long. This loops forever.

**What data is contended and what is the contention pattern?**

Access/data contention is simply when one or more threads try to access the same data resource. What happens depends on the type of contention that is occurring. If the contention is ‘one’, it means only one thread is performing operations (read or write) on this shared resource. If the contention is ‘many’, it means multiple threads are performing operations on this shared resource. The slides only showed 4 contention patterns, but there is also one read one write (which is what the contention pattern is in this case; however out of the 4 given in the slides (assuming many means one or more threads) it’s **many read one write**.

MainLoop is run only on one thread and is writing to pDraw (DrawableParticle list). Thus this is one write. Gui is also running on one thread and reads from pDraw. Gui doesn’t write anything to pDraw, it only reads from its one thread, thus this is one. Therefore the contention pattern is one read one write.

**Is this parallelism properly implemented?**

What defines properly implemented? Let’s define it as meaning parallel code that doesn’t promote race conditions, where CPU caching is avoided between shared resources, where data contentions are properly handled, deadlocks are avoided, and loads are balanced between threads.

It seems like the loads are balanced by separating GUI handling and model updates (writing) so that the rate at which GUI updates doesn’t affect the speed at which the model data updates. For instance, the O(n^2) complexity for particle interactions don’t affect the speed of the GUI updates, putting this on a separate thread solves this nicely and balances the work.

All shared resources are named volatile, so CPU caching shouldn’t occur. I.E pDraw has been given the attribute volatile so no two threads will ever have different values stored in their own cache and use that for their own writes (MainLoop).

The only data contention between the contending running threads is to do with pDraw (as shown in the section above). As the contention pattern is one read one write volatile should be enough to handle this.

What seemed well implemented to me was the fact that the scheduler.schedule(new MainLoop .. etc) only runs once, but has a while(true) inside it so the thread that is assigned to it at the start of the schedule persists to that process till the end of the program. This means that the only available thread leftover is the second one which will be used by the GUI to repaint. This is why we had a long delay of 500ms on scheduler.scheduleAtFixedRate() for the GUI so that we made sure that both threads don’t get assigned to repainting the GUI and none to the model...

Thus, yes, I think it’s correctly implemented.

**Task 3: How to introduce parallelism:**

1. How do you plan to add parallelism in the algorithm?
   1. By using parallelStreams as much as I possibly can (easy mode). They use ForkJoin library under the hood and handle all the problems with forking and joining for me. All I need to worry about is data contentions on certain collections and create race conditions on non-volatile structures.
      1. I’ll add them to the O(n^2) particle.interact() method call in step() as this will be where the highest speedup will occur. I imagine this will always be useful even at relatively small dataset sizes as O(n^2) sequentially is very slow, doing this is parallel to the outer loop should speed it up significantly, better than doing it on the inside loop at least.
   2. Making all shared resources volatile to remove multiple thread CPU caching problems
   3. I plan on operating on thread safe structures that don’t allow concurrent modifications, so no race conditions exist, and I don’t need to implement my own synchronisation or locking/unlocking (LinkedBlockingQueue, Collections.synchronisedList()) etc.
   4. Make try as many atomic actions to shared resources as possible to make sure I’m not doing ‘many’ modifications.
2. Why is it going to help in simulating particle moving, attracting each other and merging?
   1. Simply, distributing the load of the model update calculations onto different threads should improve efficiency as we have multiple workers churning through the calculations in parallel instead of one by one in sequential. Because of this, when simulating this, if the time taken to do all these calculations is lower then
3. What kind of data contentions will you need to resolve?
   1. Ones with ‘p’ and ‘pDraw’ being written to by multiple threads. ‘p’ will be both written to and read by multiple threads, will try to make it so that the critical sections will only contain read or write.
4. How will you make sure there is no hidden aliasing creating unpredicted data contentions?
   1. Using thread safe structures that don’t let more than one thing modify the contents of that structure, only read certain indexes. I could use ReadWriteLocks and manually do it for each structure, but it would expose unnecessary complications and make the code bug prone when valid libraries and implementations already exist using this technology. It’s the easiest option.

**Task 4: Design Designs:**

Before I could design the parallel implementation, I needed to refactor the given code to support multiple types of execution (sequential, parallel).

I followed an abstract factory model pattern for this but then realised I could just use Java8 and pass my execution methods to a Supplier<Model> which would assign the model.

Model m = DataSetLoader.getRandomSet(100, 800, 1000, **ModelParallel::new**);

And getting the model just required mSupplier.get();

No other changes to Gui were required so it made it really simple.

With regards to the actual implementation, I didn’t really follow a design as such. I just tried to use parallel streams as much as I could because the level of abstraction used is higher than other lower level counterparts (RecursiveTasks, ForkJoin etc) so it’s in theory ‘safer’.

**Task 5: ModelParallel Correctness**

**Task 6: ModelParallel Efficiency**

My benchmarking methodology was to fabricate datasets that were all identical that could be applied to different execution styles. As a result, the number of interactions would all be the same, the size of the simulation would be the same, the place of all the particles at the start would be the same. Essentially ensuring that no one execution style has an easier path through the code than the others.

Once I was happy with my consistency I created tests and helpers to compare my two implementations (ModelParallel and ModelParallelOptimised) against the sequential algorithm given to us. ModelParallel is overly parallelised to show the problems with this. ModelParallelOptimised is the result of testing what needs to be parallelised to create the most efficient parallel algorithm.

I created tests to run on all the abstract Model methods. This includes step(), updateGraphicalRepresentation() and mergeParticles(). This outputs how each algorithm does against each and which one was the fastest at any given simulation size. This made it easy to be confident that one execution style was the fastest at a particular size.