Contents

[Task 1 – Model.java 1](#_Toc69652572)

[Overview 1](#_Toc69652573)

[Fields 1](#_Toc69652574)

[The Model.step() Method 2](#_Toc69652575)

[Particle.interact(Model m) 2](#_Toc69652576)

[Model.mergeParticles() 3](#_Toc69652577)

[Model.getSingleChunk(Particle current) 3](#_Toc69652578)

[Model.updateGraphicalRepresention() 4](#_Toc69652579)

[Task 2 – Gui.java 4](#_Toc69652580)

[Overview 4](#_Toc69652581)

[Fields 4](#_Toc69652582)

[Methods and Classes 4](#_Toc69652583)

[Gui.run() 4](#_Toc69652584)

[Gui.MainLoop 5](#_Toc69652585)

[Gui.main(String [] args) 5](#_Toc69652586)

[Parallelism 5](#_Toc69652587)

[schedulerRepaint 6](#_Toc69652588)

[schedulerSimulation 6](#_Toc69652589)

[SwingUtilities invokeAndWait() and invokeLater() 6](#_Toc69652590)

[Contention Pattern 6](#_Toc69652591)

[Task 3 – Introducing Parallellism 6](#_Toc69652592)

# Task 1 – Model.java

## Overview

Model.java is in charge of handling the underlying logic of the particle simulator. It interacts with the Gui and Particle related classes to handle particle movement and merging.

## Fields

* **size**: Used by Canvas.java to create the size of the drawing canvas. Canvas.java extends JPanel and is used by the Gui.java for drawing.
* **gravitationalConstant**: Determines the force of gravity in the simulation. It is used by Particle.java to determine how strong a gravitational pull a particle should have. A higher value increases gravity, while a lower value decreases gravity. Higher gravity results in particles being pulled into each other more strongly, while lower gravity results in particles being pulled into each with less force. Usually, a high gravity will draw everything into the centre more strongly as the initial centre particle has the largest mass, and so has the strongest force of gravity.
* **lightSpeed**: Determines what constitutes the speed of light in the simulation. A smaller lightspeed results in faster particle speeds; we simulate a larger chunk of the universe because particles can cover more distance. A larger lightspeed results in slower particle speeds; we simulate a smaller chunk of the universe because particles cover less distance. Particle speed cannot exceed lightspeed.
* **timeframe**: Determines the length of time of a ‘step’ in the simulation. A larger timeFrame will result in a slower simulation, while a smaller timeFrame will result in a fast simulation.
* **p**: A list to hold all the particles (and their individual data) in the simulation.
* **pDraw**: A list to hold all the particles to draw in the simulation.

## The Model.step() Method

The primary method which runs the particle simulator in Model.java is step(). Model.step() is broken down into four main actions:

1. For each particle in the p field, have the particle interact with the model. This calls the Particle.interact() method in Particle.java. **Particle.interact()** is discussed below, but it essentially checks for any impacting particles and adjusts the movement direction of each particle based on each other particle’s force of gravity.
2. For each particle in the p field, have the particle move. This calls the Particle.move() method in Particle.java. Particle.move() is quite a simple method that updates the x and y position of a given particle taking into account the speed of the particle and the model’s timeFrame. Most of the heavy lifting to find out where a particle should move to is done in the Particle.interact() method.
3. Merge any impacting particles together. This calls the **Model.mergeParticles()** method, which is discussed below. Essentially, any particles that impact while moving will merge together and form a new particle based on their combined values (e.g combined speed and mass).
4. Finally, update the Gui by calling Model.updateGraphicalRepresentation(), which is discussed below.

### Particle.interact(Model m)

This method is used in the first part of the Model.step() method. It takes a model, m, as an argument so that this particle knows which other particles it can interact with in the simulation. The whole method loops through every particle, p, in the model. There are a few branches that the method can go down:

1. If p is this particle, then continue to the next particle in the loop.
2. If p is not this particle, then calculate the direction and distance of this particle relative to particle p. The method uses Math.signum() to find out what the direction the particle is relative to particle p in terms of x and y values. A negative dirX is to the left of particle p, while a positive dirX is to the right. A negative dirY is above particle p, while a positive dirY is below particle p. The distance between this particle and particle p is calculated using Particle.distance2(). Particle.distance2() calculates the x and y distance between two particles, squares it (i.e to the power of 2), and returns the value. The method then checks if this particle and particle p are impacting using Particle.isImpact(). Particle.isImpact() will return true if the distance between the particles is not a number or if the distance between the particles is less than the mass the particles take up (i.e the particles are overlapping). The method branches off again here:
   1. If this particle is impacting particle p, then add p to this particle’s impacting set, and continue to the next particle in the loop.
   2. If this particle is not impacting particle p, update the direction while taking into account mass, gravity, and time. The calculation for updating each direction is split into two parts. First is Mass \* gravitationalConstant which is the force at which particle p pulls in this particle. This ‘force’ \* the appropriate dir ensures the particle is being pulled in the right direction. Second is Dist \* timeFrame which takes into account the ‘speed’ of the simulation. Finally, the direction is updated to be the ‘force’ divided by the ‘speed’. A bigger timeFrame will result in a slower simulation because the force/speed number will be smaller, thus the change in direction will be smaller. After calculating the new direction of the particle, assert that the speed of the particle is less or equal to both lightSpeed and the particle’s own current speed. Then make the new speed to be the current speed plus the change in the appropriate direction. Divide the new speed by speed\*direction (a higher speed for a large distance being pulled) divided by lightSpeed (to keep the speed in proportion to the lightSpeed of the simulation). If the dirX or dirY are in fact numbers, update the speed fields of this particle.

### Model.mergeParticles()

This method is used to find particles that have impacted and then merge them together. The method starts by creating a stack, deadPs, to keep track of all the ‘dead’ particles (a dead particle being a particle that has impacted and been merged). The method then loops through every particle in the model. If the particle’s impacting field is not empty (e.g it is in fact impacting with at least one other particle) then we can add this particle to the deadPs stack. We can then remove all of the particles in the deadPs stack from the main particles list in the model, as they are now dead and about to be merged. While there are still particles in deadPs we take the following steps:

* Pop the next dead particle from the deadPs stack.
* Get a single chunk for the current particle. This calls the **Model.getSingleChunk()** method which essentially gets all of the particles that are impacting into each other in a given step (getSingleChunk) is discussed in detail further below).
* We can remove from the deadPs stack all of the particles in the single chunk, as we are handling these particles now.
* We then merge all of the particles in the single chunk, by passing the single chunk into the other Model.mergeParticles() method which takes a set of particles as an argument and returns a single particle. Particles in a set are merged together based on the combined mass and speed of all the particles in a single chunk. The merged particle is then added into the model’s list of particles.

### Model.getSingleChunk(Particle current)

The method takes in a single particle as an argument and then returns a set of all that particles that the particle is impacting with, as well as any particles that the impacting particles are impacting with. The method keeps track of all impacting particles by creating a set, impacting. It then adds the particles passed in as an argument to the set. The method then keeps running the following loop:

* Create a new temporary set, tmp, of particles.
* For each particle in the impacting set, add all particles that are impacting with the current particle to tmp.
* The method then adds al the particles in tmp to impacting, and checks if the impacting set was changed.
* If the impacting set was not changed, then break the loop as no new impacting particles have been found in this chunk. Otherwise, repeat the loop as new particles have been found and there are potentially more particles that are impacting and therefore part of the chunk.

After the loop the method will return all of the particles that are impacting on each other in a given step involving a particular particle (the particular particle is the ‘current’ particle passed into the method).

### Model.updateGraphicalRepresention()

The method updates the graphical representation of the particle simulator. It creates a new list of drawable particles, d, and a new colour with which to draw particles (the program uses the colour orange). Then, for each particle in the universe model create a new drawable particle and add it to the drawable particles list, d. After the loop, the method replaces the drawable particles list in the model with the list, d. This is an atomic update as pDraw (the model’s list of drawable particles) is a volatile variable.

# Task 2 – Gui.java

## Overview

Gui.java handles the graphical representation of the particle simulator, and also initiates the loop that runs the simulation. Help was found from the following links:

For scheduledThreadPoolExecutor: <https://liakh-aliaksandr.medium.com/java-concurrency-and-parallelism-executors-and-thread-pools-cb62764bca02>

For invokeAndWait and invokeLater: <https://stackoverflow.com/questions/5499921/invokeandwait-method-in-swingutilities>

## Fields

* **minTime:** The minimum amount of time that a step has to take in the simulation.
* **schedulerRepaint:** The ScheduledThreadPoolExecutor used to handle the repainting of objects on the Gui.
* **schedulerSimulation:** The ScheduledThreadPoolExecutor used to handle the MainLoop which runs the model.
* **m:** The model that handles the logic of the particle simulator.

## Methods and Classes

### Gui.run()

This method implements the run() method for the Gui class. The method handles the repainting of the particle simulator. First it calls a few methods that relate to Swing and setting up the Gui, essentially boilerplate (calling setDefaultCloseOperation(), setting the layout, creating a Canvas.java object, calling setDoubleBuffered() on the canvas, adding the canvas to the layout, packing the Gui, setting visibility to true). After the Swing Gui has been established, schedulerRepaint is used to call scheduleAtFixedRate(). scheduleAtFixedRate() allows us to have a task that runs at a certain rate. In this case, the method is repainting the canvas, with an initial delay of 500, a period of 5, and the time unit being in milliseconds. The code to repaint first checks if the schedulerRepaint queue is not empty. If this is true, then the code prints out the message “Skipping a frame” and returns. This means that the ‘worker’ will only pick up one task at a time and isn’t interrupted by another task while it is still repainting. If it is false that the queue is not empty, then the code calls swingUtilities.invokeAndWait() passing in repaint(). invokeAndWait() means that the given task/runnable is placed into the worker queue, and the then waits until the thread has executed it. According to a Stack Overflow response, “this should be used when a non-GUI thread needs to do something that affects the GUI, but also needs to wait until it is actually done before it can continue”. This is true for the particle simulator as the program needs the GUI to finish updating in order to work out what should happen in the next step of the model, because every single step needs to be rendered on the GUI. The method also wraps the invokeAndWait() around a try catch to catch any InvocationTargetExceptions or InterruptedExceptions.

### Gui.MainLoop

MainLoop is a class in Gui.java. It has a field m, which holds a model object, and the constructor takes a model and assigns it to m. The class implements Runnable, and so has a run() method. The run method runs the following loop:

* In each iteration the start time (called ut, or “used time”) is kept track of.
* m.step() is called. Model.Step() is discussed in Task 1 above, and handles most of the logic of the particle simulator.
* The ut is then reassigned to the be current time minus the old ut. This will get the total time used to take the step.
* A variable sleepTime is created and assigned the value of minTime minus ut (noting that **minTime** is a field of Gui.java, see above).
* It then checks that sleepTime is greater than 1. If sleepTime is 1 or less, then the step took at least the minTime. If the sleepTime is greater than 1, then the thread sleeps for the sleepTime to ensure that the step takes the minimum amount of time we want (in this case it is 20 milliseconds).
* The loop then repeats, thus running the simulation.

The loop is surrounded by a try catch to catch any Throwables.

### Gui.main(String [] args)

The main method is quite simple, but it essentially starts off the simulation. First it creates a variable m that holds a model. There are several lines of code commented out for various different models. These lines use the DataSetLoader.java class to create different data sets for the simulation (e.g models with varying particle size, particle distance, and number of particles). The method then uses schedulerSimulation to schedule the MainLoop class to run with a delay of 500 milliseconds. This is different from the scheduleAtFixedRate method because it will only run the given task/runnable once, whereas scheduleAtFixedRate will continuously run at a given task/runnable at a certain interval. The method then calls SwingUtilities.invokeLater() passing in new Gui(m). We use invoke later here because we don’t care when the method passed in is executed (in this case it is the creation of the GUI. We do not care when it is created because we are not waiting on any thread/worker to finish a task that the creation of the GUI is reliant on).

## Parallelism

Gui.java demonstrates minimal parallelism through the schedulerRepaint and schedulerSimulation fields, both of which are ScheduledThreadPoolExecutors. Both fields are initialised with a corePoolSize of 1. The class uses the SwingUtilities invokeAndWait() and invokeLater() methods (as discussed above). Help was found from the following links:

Java documentation for ScheduledThreadPoolExecutor: [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ScheduledThreadPoolExecutor.html#scheduleAtFixedRate(java.lang.Runnable,long,long,java.util.concurrent.TimeUnit)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ScheduledThreadPoolExecutor.html" \l "scheduleAtFixedRate(java.lang.Runnable,long,long,java.util.concurrent.TimeUnit))

Java documentation for SwingUtilities: <https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/SwingUtilities.html>

### schedulerRepaint

schduledAtFixedRate() is used to repaint the Gui’s canvas at a specific interval (as discussed above). scheduleAtFixedRate() returns a SchduledFuture which represents the pending completion of the series of repeated tasks (according to the documentation). A ScheduledFuture is an extension of Future. According to the documentation, “If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.” This means that a ScheduledThreadPoolExecutor will not run tasks concurrently on its own, but it will queue up subsequent tasks. However, the particle simulator relies on a step to be fully completed before the logic for the next step beings. This is why if the schedulerRepaint queue is not empty the code is returned so that ‘frames’ are skipped until the program has caught and the current step is completely finished.

### schedulerSimulation

schedule() is used to start the MainLoop with a delay (as discussed above). Schedule() also returns a ScheduledFuture like schduleAtFixedRate().

### SwingUtilities invokeAndWait() and invokeLater()

According to the documentation, the two invoke methods are used to ensure that tasks are executed on the event dispatching thread, and both methods should be used when an application needs to update the GUI. invokeAndWait() is blocking until all AWT events have been processed and doRun().run returns; this is likely for the reason discussed above that the GUI needs to be fully updated before the program can continue running the logic of the particle simulator. invokeLater() will wait until after other pending AWT events have been processed; as discussed above, the creation of the GUI is not reliant on any task so we don’t care if it is invoked later. Because the GUI tasks are executed on the seperate event dispatching thread, it is logical that the handling of the GUI and the handling of the particle simulator logic is separated in the program file structure.

## Contention Pattern

The contention pattern is Many Reads, One Write. This is because many parts of the Gui access the model, however only one part updates it. Gui.run() accesses the model to repaint the GUI. MainLoop.run() accesses the model to call Model.step(). By calling Model.step(), MainLoop.run() becomes the one writer as Model.step() will change data in the Model object. The parallelism is implemented correctly because the pDraw field in Model.java (which is ultimately the field that Gui.java is accessing) is volatile, and pDraw is updated atomically. pDraw is updated atomically because the Model.Step() method (which is the method that methods in the Gui.java class call) calls Model.UpdateGraphicalRepresentation() which creates a brand new List of drawable particles, rather than updating the values existing in the field.

# Task 3 – Introducing Parallellism

Design decisions:

- AbstractModel; Model and ModelParallel will share a lot of functionality anyway. Essentially it is just the step() method which will be different, so this method is made abstract. As a result, everywhere model is previously mentioned in the code, it will be replaced by AbstractModel. Dataset loader has been modified to create datasets for ModelParallel objects.

* ParallelStream for interact and move because interact and move are simple operations. Essentially just calling a predefined method on each particle. Initially tried to use Futures but then realised that Futures need to return an object via get(), but interact and move don’t return objects. Simulation appears to work fine when using parallel streams, will test for correctness and efficiency next.

**Correctness**

* Getting issue where for elaborate datasets the speed of particle is ever so slightly different. I’m not too worried about this since the different is very insignificant; a difference in speedX of 0.0000000000000002E-4.

Text

Description automatically generated

* Otherwise I tested for:
  + Number of particles before and after taking a step.
  + Each of the fields for each particle after taking a step.

I tested for 1000 steps on the ECS lab machines.

Tests had an assertion error for randomGrid, regularGrid, and elaborate

**Efficiency**

For each different type of dataset:

* Random set: sequential is slightly faster
* Elaborate: is very quick anyway that time different is negligible.
* Random grid: parallel is more than 2x faster.
* Random rotating grid: parallel is more than 2x faster.
* Regular grid: parallel is more than 2x faster, and nearly 3x faster.

This is tested under the following conditions:

* Only using the implementation of parallel streams with interact and move.
* There are 100 warm up steps that aren’t timed.
* There are 2000 actual steps which are timed.
* There is a min of 500 particles and max of 1500 particles in each dataset.
* Distance is 30 for datasets that take a distance.
* ModelParallel only uses parallel algorithms if the number of particles is greater than or equal to 50.

Graphical user interface

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

After running this initial test, I ran the test again on the ECS lab machines with steps being 10000, since the ECS machines are better than my laptop.

- The difference in efficiency for random set and elaborate is negligible; both datasets run very fast anyway.

- Random grid is between 4 and 5 times faster for parallel.

- Random rotating grid is about 4 times faster for parallel.

- Regular grid is about 4 and 5 times faster for parallel.









