Task1

1.public static final double size=900;

Used by the canvas to determine width and height of the drawing area.

2.public static final double gravitationalConstant=0.002;

Used by the interact method in Particle class as a coefficient to calculate the distances with respect between particles.

3.public static final double lightSpeed=10;

Used by the interact method in particle class to calculate new speed for particles.

4.public static final double timeFrame=20;

Used in the move method to calculate the change of x and y coordinates in every movement. Speed is divided by the timeFrame so that the amount of movement is affected by the timeFrame.(the bigger timeframe is, the slower the particles will move everytime the move method is called.)

5.public List<Particle> p=new ArrayList<Particle>();

the list of all particles in the model.

6.public volatile List<DrawableParticle> pDraw = newArrayList<DrawableParticle>();

The list that stores all particles to be drawn. This is declared volatile because we want to make sure that in multi-threading, the drawing of particles in each step do not get messed up. More specifically, since the GUI uses newScheduledThreadPool, declaring the list volatile helps prevent caching problems.

Four main actions in Model.step():

for(Particle p:this.p){p.interact(this);}

This is calling interact method on every particle in this model. In every step the speed of this particle is updated based on the gravitation of all other particles. If any particle is impacting this particle, that particle is added into the impacting collection.

mergeParticles();

Creates a stack deadPs for storing dead particles. For loop all particles in the model to see if it is impacting collection is empty. If it is empty add the particle into the deadPs and remove it from the list p since it is dead. Then while the deadPs is not empty, pop a particle from the stack and get the chunk of particle impacting the popped particle and remove all these particles from the deadPs. Last, merge the chunk into a single particle and add this particle back into the list p.

for(Particle p:this.p){p.move(this);}

Updates all particles' x and y fields using the calculated speedX speedY in the interact method.

updateGraphicalRepresentation();

Creates a list containing DrawableParticles, declare a Color variable that equals Color.ORANGE , for loop all particles in the model, creates drawableParticle objects using the x, y, mass values from one of the particles in p as well as the previously defined color variable c and last, add all the DrawableParticles objects into the local list and last, make the volatile list of DrawableParticles pDraw point to the local list so that when we want to read the pDraw list, we ignore cache and read it directly from memory.

How the merging of particles works:

First, in every step, calling interact method on all particles updates the speedX and speedY fields of all particles based on the gravitational force of all particles. Second, calling move method on particles would result in the x and y fields to change based on the speedX and speedY fields. After a number of steps, the positions of the particles would continue to be affected by the gravity of each other until it comes to a point when they impact. The mergeParticles method finds out the chunks of particles that are impacting and merge those particles into one bigger particle. Notice

that the cost of finding out the chunks of impacting particles in p should be linear. This is because:

1. We first remove all particles with non-empty impacting list from p and add those particles to a deadPs stack.
2. Then while deadPs is not empty, we find out the chunks of impacting particles and remove them from deadPs.
3. No particle is repeated "traversed" as they will be removed from deadPs as soon as they are found to be in a chunk.

This makes the above steps 1 and 2 not very expensive. Thus in task 3, I do not parallelize this method, instead, I choose to parallelize the mergeParticles(ps) which is more costly.

Bugs:

1. MergeParticles() method should not be called in between interact and move. Putting it there could cause particles which are close enough to be merged together not merged and shown on the GUI. In order to fix this, I have an idea:

```
mergeParticles();// particles generated using methods in datasetLoader class may be very close
//to each other, should call merge method to merge the particle before changing the status of
//any particle. If we don't do this particles that should have been merged may move away from
//each other.
for(Particle p:this.p){p.interact(this);}
for(Particle p:this.p){p.move(this);}
mergeParticles();// this is to merge after moving all particles. If we do not do this, the gui may
//show particles close enough to be merged not merged.
updateGraphicalRepresentation();
```

2. I think the project code does not have proper **encapsulation** but of course it depends more on what the coder needs it to do.

For example the methods in the model and Particle classes are all public but the only usage of some methods are within the model class. (like the mergeParticles method in Model class and the interact and move methods in Particle class).

For the parallel version I have did some modifications to the program including creating an ModelAbstract class and let the sequential Model and a ModelParallel class extend it. I have also done some level of encapsulation in the project to hopefully help prevent encapsulation from **breaking**.

Task2

The Gui class implements Runnable, which possesses a run method, 2 static int fields frameTime and stepsForFrame and a static final field ScheduledExecutorService which points to a

newScheduledThreadPool with pool size 2. The constructor takes in a Model object and makes the m field point to that object.

The stepsForFrame determines how many steps are run for each “frame”.

The frameTime field determines how long a frame is. By default it is 20ms.

The run method of Gui creates a Canvas and submits a task to the threadpool using the “scheduleAtFixedRate” method. The task submitted runs repaint() periodically (every 25 milliseconds) method after a given delay (500ms). **The “invokeLater(repaint())” makes sure that the repaint() method only runs on the event dispatch thread.** Calling repaint method will have all components within “this” JFrame, which is the Canvas, to repaint. In our case the composite pattern has only one component, which is the Canvas JFrame itself and thus the paint method overriden in Canvas class is called to redraw the particles using the volatile List pDraw.

The inner class MainLoop implements Runnable. The run method has a while loop that runs infinitely. In the while loop, it first does step() for stepsForFrame times, which, by default, is 20. The time consumed running the steps is recorded and stored in a long variable ut. The sleepTime is then calculated using frameTime - ut and if sleepTime is greater than 1, the thread is put to sleep for sleepTime milliseconds. This is to make sure that the running time of each while loop is **at least frameTime** milliseconds. I think the main reason for doing so is that we want to visualize the particle motions in a way that the speed of the motions are human-recognizable (if the while loop goes too fast we will not be able to see the motions).

The main method of Gui has a set of Model using different specification. The default model creates 100 particles with min x, y = 100 and max x,y = 800. The schedule method submits a task that becomes enabled after a given delay. By default the delay is 500 ms.
SwingUtilities.invokeLater(new Gui()) ensures that the threads other than the event dispatch thread cannot run “new Gui()”.

In general, the Gui class utilizes a newScheduled thread pool which

About why we use invokeLater to initialize the Gui (wrapping new Gui() with invokeLater)and to repaint(), from the Swing documentation, **all UI operations should be done in a special thread called the event dispatch thread.** Since the program we are running is concurrent, we need to explicitly call SwingUtilities.invokeLater on the initialization of Gui and on all UI operations to ensure that they run on the event dispatch thread.
Moreover, all tasks performed inside a swing event handler should not be long-running tasks and methods like Thread.sleep(), Object.wait() cannot be used on the event dispatch thread as these methods would block or delay the thread. This is probably why we separate the drawing of particles from the calculations of particle movements.

The contention pattern is: many reads one write. MainLoop is the writer as only in the run method of this class the volatile field pDraw is updated. The run method of Gui submits tasks to the threadpool which reads the pDraw and updates the graph every 25ms.

Task3

(a)
I plan to parallelize parts of the step method. In specific, part of the mergeParticles() method (if it works out.), the move method and the interact method. I will be using parallelStream on the interact method and move method and perhaps, a newCachedThreadPool and a list of futures of Particle for the mergeParticles(set) method.
(b)
For interact method:
ParallelStream utilizes the forkjoinpool, which has a work stealing mechanism.
I use **parallelStream** for the interact method because:
1. Interact method is thread-safe. No write will affect any read.
2. It is simple. One line of code does the work.

For move method:
Similarly to interact, the move method is also thread-safe. It reads and updates the x and y values using fixed values speedX and speedY (when the move method is run the speedX and speedY have been updated.) If number of particles in the model is huge and the step method is run many times, parallelizing the move method can potentially speed up the program.

For mergeParticles method:
Specifically, in the mergeParticles(set) method I plan to parallelize the very last step: **this.p.add(mergeParticles(ps))** because this is a heavy step and it can be safely **decoupled** from the while loop since the "chunk" of particles "ps" has no hidden aliasing with any particles in p. We can store all chunks in a list after the while loop instead of merging them right away and merge the chunks separately in parallel to speed up the process. I have used a set of Futures to help preserve ordering because parallelization should achieve exactly the same result as the sequential one.

The reason why I choose to parallelize interact, move and mergeParticles was that these are the heaviest tasks in the program. In the run method of MainLoop, the step method is run for stepsForFrame times and in each run of step method the interact method is called on every particle in the model. The work of MergeParticles is tedious mainly because of the mergeParticles method as there are many calculations The move method is not as worth parallelizing as the interact method because it only updates the x and y fields of all particles using the calculated speedX speedY factors but I still parallelized it because if the number of particles get too large,

say 100 million, parallelizing the move method could still speed up program.

(c)

The interact and move methods follows a “many readers zero writer” pattern. All writes by different threads do not affect the reads by any thread. For example. DirX and dirY are calculated using the x and y fields of a particle in the p list but at this stage the values of x and y will not have been changed. In other words, writes on different threads will have no affect on any read on any thread.

The mergeParticles method, however, follows the “many readers many writers” pattern. All threads are reading p and all threads could modify p, which is disastrous. However going deeper into the code, I spotted that the mergeParticles(set) method can be decoupled and parallelized separately.

(d)

For the interact method no aliasing is guaranteed because no write will affect any read.

For the move method no aliasing guaranteed because of similar reason to the interact method. Updating of x,y is dependent on speedX and speedY fields but they should have been calculated before the move method.

Task4

```
@Override
public void step() {
   if(p.size() < 50)
      doParallel();
   else
      doSequential();
   updateGraphicalRepresentation();
}
```

Basically if p.size < 50 I delegate the tasks to the sequential way. This is to help improve performance because maintaining threads is costly.

For the doSequential method I just copied over the code from the step method in model class.

For doSequential():

```
//do p.interact in parallelStream
p.parallelStream().forEach(p ->p.interact(this));
mergeParticlesParallel();//parallelize mergeParticles method.
//do p.move in parallelStream
p.parallelStream().forEach(p ->p.move(this));
```

I use parallelStream on interact and move methods.

For the mergeParticlesParallel method:

```
private void mergeParticlesParallel(){

   Stack<Particle> deadPs=new Stack<>();
```

```
    List<Set<Particle>> chunksToMerge = new ArrayList<>();
    for(Particle p:this.p){ if(!p.impacting.isEmpty()){deadPs.add(p);}}
    this.p.removeAll(deadPs);
    while(!deadPs.isEmpty()){
        Particle current=deadPs.pop();
        Set<Particle> ps=getSingleChunck(current);
        chunksToMerge.add(ps);
        deadPs.removeAll(ps);
    }
    //in this way we can decouple the merging and adding of particles with the while loop and
parallelize this separately.
    List<Future<Particle>> futureTasks = new ArrayList<>();
    for (Set<Particle> chunk : chunksToMerge)
        futureTasks.add(pool.submit(() -> mergeParticles(chunk)));
    for(Future<Particle> future : futureTasks)
        this.p.add(get(future));
}
```

Lines highlighted in yellow are the modifications.

I use a list of sets of Particles to store the "ps" in each while loop. Then I use a list of futures of Particles for storing all the tasks in order. Essentially just using a parallelStream could be sufficient for getting "correct" result. However, As we know, in multi-threading, tasks can be done at any moment, which means the ordering of the mergedParticles being added back into the p list could be messed up. So if we still want to perfect the threading by preserving ordering, we could so by storing the tasks in a **list of futures** and call get method on it in order. In this way the tests which I will mention in task5 will not fail.

Task5

I plan to test correctness by comparing in every step whether the x, y, mass, speedX and speedy fields of the parallelized step and the sequential step methods are equal using a number of different models.

Summary of the test cases:

For a given number of steps(set to 600 as default):

1. **In each step Ordering in which the particles are stored in the p list in each step is tested.**
2. In each step the equality of size of particles in each model is tested.
3. In each step each particle is tested equality (I have overriden hashcode and equals methods based on all field values.)

For model configurations I set

gravitationalConstant = 0.05

timeFrame = 20

lightSpeed = 10;

These are to make the test cases run faster.

I made 10 tests each runs 600 steps using different model. This gives me solid proof that if all 10

tests pass, my parallel implementation is correct because the only place where the fields of particles are modified is in the step method and if every run of step is correct in every model setup, I think it is safe to say my parallel implementation is correct.

Task6

You can run the PerformanceTests class to see the results.
Since in task 5 the correctness of the parallel model is ensured. I can simply log the time taken to run the parallel model for a big number of steps and compare whether it is faster or slower than the sequential model.
I have noticed, however, the parallel model runs faster when the number of particles is huge and it tends to slow down as the degree of particle-merging increases. This is easy to explain since in the step method, there are multiple for loops that goes through each particles and make changes to them. With parallelization we can divide N tasks into N/(number of threads) tasks, which drastically improves performance if we are dealing with a very huge N. If N is small, the cost of maintaining the threads could exceed the benefit of parallelizing those tasks.
I have thus made my parallel implementation to check the current number of particles in the model. **If number > 50, do parallel, else delegate the tasks to sequential** to better salvage system resource.

Results:
parallel model Time taken: 142
sequential model Time taken: 51

parallel model Time taken: 293
sequential model Time taken: 705

parallel model Time taken: 579
sequential model Time taken: 1902

parallel model Time taken: 145
sequential model Time taken: 377

parallel model Time taken: 1661
sequential model Time taken: 6003

parallel model Time taken: 557
sequential model Time taken: 1917

parallel model Time taken: 1150
sequential model Time taken: 3970

parallel model Time taken: 80
sequential model Time taken: 182

parallel model Time taken: 4070
sequential model Time taken: 14483

parallel model Time taken: 182
sequential model Time taken: 375

From the results, models with **smaller particle size** run faster in sequence, whereas models with **bigger particle size** run faster in parallel. Which is exactly what I expected.