Universe

Task 1.

How the class works:

The Model class is the main functioning brain of this program. The global parameters control the size of the canvas (900 in this case), a gravitational constant of 0.001 which is used for updating the direction in relation to the gravitational pull, light speed which controls the chunk of the universe which is being simulated, and finally time frame which controls the speed/accuracy of the simulation. There is also a list of particles which is all particles for this model. It is updated as the simulation runs. The final parameter is a separate list of DrawableParticels which is updated during simulation. A drawable particle is a copy of a particle with a square root of the mass and a method that takes the Graphics2D object as a parameter that can be drawn on.

Model.step()

Action 1:

The first action is the for-each loop iterating each particle stored in the list of Particles p. It then calls to interact with each particle. This method checks for any impacts with the other particles which if true adds the impacting particle to its own ArrayList of impacting particles. If there are no impacts it will create a new speed value for x and y.

Action 2:

The second for each loop also iterates each Particle and calls the move method. This simply moves the particle by incrementing its x and y value by a stored speed x and y value.

Action 3:

The next method calls merges and particles that have been impacted. See below for a detailed explanation of how this method works

Action 4:

Finally, the graphical display is redrawn with a method call. This method updates the ArrayList of drawable particles.

Particles merge

Firstly a new stack is created for the dead particles. This checks each particle's impacting list doesn't contain any other particles. If it is not empty the particle is added to the deadPs stack. The dead particles are then removed from the main list of particles for this model. A particle from the stack of dead particles is then popped off the end, getSingleChunck method is called with the particle as a parameter. getSingleChunck returns a new set of all impacting particles which includes neighboring particles impacting list. The returned set of particles is removed from the deadPs stack and passed to the mergePaticles call that takes the set of returned particles as a parameter. This method uses the information from the stack of particles to create a new particle with a new mass speed and location. This new particle is the result of the impact of all given particles and is returned to be added to the list of perticles for this model.

Task 2.
The Gui class uses 2 ScheduledThreadPoolExecutors. One is used for the repainting and one is used for the simulation, each is initialized with the value that represents the size of the corePoolSize in this case the value 1.
There is also a SwingUtilities.invokeAndWait call. This places the given runnable into the event dispatch thread which is a special GUI thread. This is used in specific cases when a non GUI thread needs to do something that affects the GUI. In our case, it needs to update the particle information before it can re-draw each particle. We can also see a SwingUtilities.invokeLater call is done in the main method. This executes the Runnable Gui object on the AWT event dispatching thread. This prevents concurrent access issues as the GUI class is a long-running action.

schedulerRapaint
This scheduledThreadPoolExecutor is used for repainting each particle on the board. The scheduleAtFixedRate call creates and executes a periodic action that becomes enabled first after the given initial delay.

schedulerSimulation
This scheduledThreadPoolExecutor is created for running the MainLoop class which runs the Model simulation. This is run in its own thread. The schedule call creates and executes a one-shot action that becomes enabled after the given delay. After the schedule, an if statement checks the queue and prints out “Skipping a frame” if it is empty.

There is a many reads to one write data contention. This occurs as the Drawableparticles are written to and read from two separate threads. The scheduleSimualtion thread updates the information of each particle through the step method (many reads). The updateGraphicalRepresentation then does a single write of all the new drawable particles (one write). The problem of information being accessed at the same time is averted as only one thread is updating information and one thread is reading.

This parallelism is properly implemented as the thread pools are correctly set up. The schedulerSimulation is only writing to the list of particles and the schedulerRepaint is only reading from it for drawing.

Task 3.
(a)
I plan on using java Collections.parallelStream in the step method contained in my new ModelParrellel.java.
(b)
The reason I am using this method on the particle.interact() inside the step method is because the cost is O(n^2) as each particle has to check every other particle. This is going greatly reduce the time for checking particle interactions.

(c)
There will be no data contention as the interact method checks for impacting particles and updates any speed/direction constants but does not change the particles location which is the only shared information in this step.

(d)
I am sure there is no hidden aliasing as the data is not being manipulated by multiple threads.

Task 4.
See ModelParallel.step()

Task 5.
In order to ensure that ModelParallel behaves exactly as Model in all the situations I designed the following automated testing strategy. I created JUnit5 testing suite with a run and compare method that firstly takes the two models and steps through each one 1000 times. I then for loop through each model's particles and compare them to each other using a custom compare method inside the particle class. If all particles match exactly I know the two models are doing the exact same thing. This gave me a high level of confidence because after 1000 steps enough moving and merging has occurred that any variance would cause a false outcome. To ensure the two models didn't contain the same list (hidden aliasing) I created a deep clone of the list of particles from the first model in the second model. To ensure this worked correctly I made a test case that removes one element and checks the lists do not match.

Task 6.
In order to check that ModelParallel is more efficient than Model, I designed an automated Junit5 test suite that tested on all the available data sets from the DataSetLoader.java file. Using the same deep clone method as stated in task 5 I created a Model class and a ModelParallel class with a deep clone of the first model's list. Using a calcTime method I passed in the model being tested the number of warmup cycles and the number of testing cycles. After running javas garbage collector I speed the Model the number of times stated by the warmup cycle input. Finally, I took the system's current time in milliseconds and stepped the model the number of times stated by the runs parameter. The finishing time is then recorded and the difference between starting and finishing times is returned. This gave me a high level of confidence because I also include a testWarmup test case that runs the calcTime with identical Models. This ensures the warmup cycle is working correctly as the resulting times should be very similar. I also reduced the number of warmups to ensure the simulation does not finish during testing and only used 5000 testing steps for the same reason.