STDISCM - Technical Report

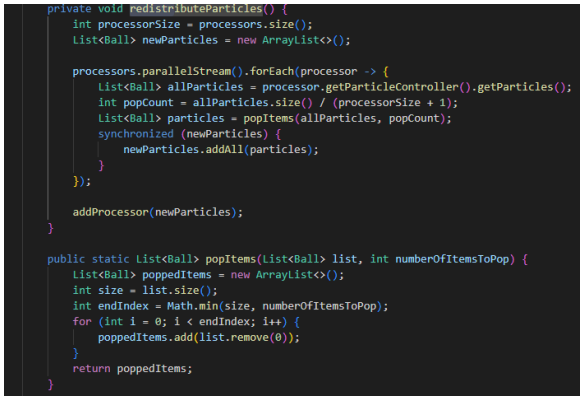
1. Load Balancing of Computations

In balancing the load of computing particle movements, we developed the ThreadController class to manage the threads used for updating particle positions. Contrary to a fixed thread pool, this class utilizes a dynamic ForkJoinPool, which not only allows for flexibility in the number of threads but also enhances efficiency through its work-stealing capabilities. This ensures that we have a consistent number of active threads and minimizes the overhead of creating and destroying threads.

```
public class ThreadController {
    private List<ParticleProcessor> processors = new CopyOnWriteArrayList<>();
    private ForkJoinPool executorService = new ForkJoinPool(); // Using ForkJoinPool
    private int canvasWidth, canvasHeight;
    private int particleSize = 0;

    private long lastAverageProcessingTime = 0;
    private List<Long> processingTimesHistory = new ArrayList<>();
    private static final int PROCESSING_TIME_HISTORY_SIZE = 20;

    private int lastParticleSizeAtThreadAddition = 0;
    private boolean isPaused = false;

    public ThreadController() {
        prewarmThreads();
    }
```

Each particle in our simulator is represented by the Ball class. This class includes properties for the particle's position and velocity. The updatePosition method updates these properties based on the particle's velocity and handles wall collisions by reversing the velocity when particles hit the boundaries of the canvas.

The ThreadController class is central to coordinating the entire simulation, dynamically managing multiple ParticleProcessor instances. These processors are responsible for handling subsets of particles. New particles are assigned to processors based on the current load, ensuring that the workload is evenly distributed among available threads.



```
private void redistributeParticles() {
    int processorSize = processors.size();
    List<Ball> newParticles = new ArrayList<>();

    processors.parallelStream().forEach(processor -> {
        List<Ball> allParticles = processor.getParticleController().getParticles();
        int popCount = allParticles.size() / (processorSize + 1);
        List<Ball> particles = popItems(allParticles, popCount);
        synchronized (newParticles) {
            newParticles.addAll(particles);
        }
    });

    addProcessor(newParticles);
}

public static List<Ball> popItems(List<Ball> list, int numberOfItemsToPop) {
    List<Ball> poppedItems = new ArrayList<>();
    int size = list.size();
    int endIndex = Math.min(size, numberOfItemsToPop);
    for (int i = 0; i < endIndex; i++) {
        poppedItems.add(list.remove(0));
    }
    return poppedItems;
}
```

Load balancing is then achieved by dividing the list of particles among multiple ParticleProcessor instances, which are then executed in parallel by the ForkJoinPool. This ensures that each thread processes an approximately equal number of particles, and threads can be added dynamically based on the load.

2. Computation of Frame Rate

The 'FPS' class handles the FPS calculation and initializes three variables: 'lastTimeCheck', which records the last timestamp of the last FPS update, 'frameCount', which counts the rendered frames since the last update, and 'fps', holds the current FPS value.

During each frame render the 'update' method performs these following tasks: retrieve the current time in milliseconds, calculate the elapsed time since the last calculation, and increase the 'frameCount' by one for each rendered frame. FPS is updated when at least 500 milliseconds has elapsed since the last update. The update involves doubling the frameCount to account for the frames rendered over the half-second interval. After the update, the framecount is reset to zero and the 'lastTimeCheck' is updated with the current timestamp. Finally, the method 'getFPS()' returns the 'fps' value.

3. Load Balancing of Rendering

To balance the load of rendering the particles, first we have the checkAndAdjustThread() function periodically check if the system needs to add a new thread (processor) to handle the increased load of particles. It calls the shouldAddThread() method to determine if the conditions for adding a new thread are met. If they are, it then calls redistributeParticles() to balance the load among the available processors.

```
public void checkAndAdjustThread() {
    if (shouldAddThread()) {
        redistributeParticles();
    }
}

private boolean shouldAddThread() {
    if (processingTimesHistory.isEmpty()) {
        return false; // Skipping if not enough data or particles
    }

    long currentAverageProcessingTime = processingTimesHistory.get(processingTimesHistory.size() - 1);
    boolean processingTimeIncreasing = currentAverageProcessingTime > lastAverageProcessingTime;
    boolean significantParticleIncrease = particleSize >= lastParticleSizeAtThreadAddition * 1.10;

    return processingTimeIncreasing &&
        processors.size() < Runtime.getRuntime().availableProcessors() &&
        significantParticleIncrease;
}
```

Second, we have the redistributeParticles() function and much like its name suggests, redistributes particles among the available processors to ensure a balanced load. It collects all particles from existing processors, calculates how many particles each processor should handle, and distributes them evenly. It then creates a new ParticleProcessor with a portion of these particles and adds it to the list of processors. This is done using ForkJoinPool which allows tasks to be done in parallel allowing for improved performance for multicore processors [1].

```
private void redistributeParticles() {
    int processorSize = processors.size();
    List<Ball> newParticles = new ArrayList<>();

    processors.parallelStream().forEach(processor -> {
        List<Ball> allParticles = processor.getParticleController().getParticles();
        int popCount = allParticles.size() / (processorSize + 1);
        List<Ball> particles = popItems(allParticles, popCount);
        synchronized (newParticles) {
            newParticles.addAll(particles);
        }
    });

    addProcessor(newParticles);
}
```

Additionally, the drawParticles() method plays a critical role in the rendering aspect. This method is called within the graphical user interface to visually represent the particles on the canvas. Each ParticleProcessor is responsible for drawing its subset of particles. By balancing the number of particles among processors, the rendering load is also balanced, which helps in maintaining a consistent frame rate.

Lastly, the addProcessor(List<Ball> particles) function creates a new ParticleProcessor initialized with a list of particles and adds it to the list of processors. It then executes this new processor using the ForkJoinPool.

```
private void addProcessor(List<Ball> particles) {
    ParticleProcessor processor = new ParticleProcessor(canvasWidth, canvasHeight, particles);
    processors.add(processor);
    executorService.execute(processor);
    lastParticleSizeAtThreadAddition = particleSize;
}
```

Group 4:

[P1] Russel Campol, **[P2]** Andres Clemente, **[P3]** Paolo Flores, **[P4]** Miel Geoffrey Racela, and **[P5]** Kenn Michael Villarama 06/21/2024

Activity	P1	P2	P3	P4	P5
Topic Formulation	25	25	25	12.5	12.5
Machine Definition	25	25	25	12.5	12.5
Formal Language and Computational Power	25	25	25	12.5	12.5
Applications	5	5	5	42.5	42.5
Raw Total	80	80	80	80	80
TOTAL	20	20	20	20	20

References

[1] Harischandra, G. (2023, November 10). Java Fork-Join pool - Gathila Harischandra - Medium. *Medium*. https://medium.com/@gathilaharism/java-fork-join-pool-with-an-example-320a0d3d5b4c

[2] Stack Overflow. (2009). Calculating frames per second in a game. *Stack Overflow*. https://stackoverflow.com/questions/87304/calculating-frames-per-second-in-a-game