Lab G

Exercise 1:

Solution:

impl Particle {

    fn new(x: f32, y: f32) -> Self {

        Self { x, y }

    }

    fn move\_randomly(&mut self) {

        let mut rng = rand::thread\_rng();

        let dx = (rng.gen::<f32>() - 0.5) \* 2.0;

        let dy = (rng.gen::<f32>() - 0.5) \* 2.0;

        self.x = (self.x + dx).clamp(0.0, ENCLOSURE\_SIZE);

        self.y = (self.y + dy).clamp(0.0, ENCLOSURE\_SIZE);

    }

    fn collide(&self, other: &Particle) -> bool {

        let dx = self.x - other.x;

        let dy = self.y - other.y;

        let distance = (dx \* dx + dy \* dy).sqrt();

        distance < COLLISION\_THRESHOLD

    }

}

   fn detect\_collisions(&self) -> usize {

        let mut collisions = 0;

        let n = self.particles.len();

        for i in 0..n {

            for j in (i + 1)..n {

                if self.particles[i].collide(&self.particles[j]) {

                    collisions += 1;

                }

            }

        }

        collisions

    }

fn simulate\_movement(&mut self, duration\_secs: u64) {

        let start\_time = Instant::now();

        while start\_time.elapsed().as\_secs() < duration\_secs {

            self.move\_particles\_threaded();

            self.display\_particles\_grid();

            // Collision detection using a separate thread pool with 1 thread.

            let collision\_count = {

                let mut collision\_pool = Pool::new(1);

                let mut count = 0;

                collision\_pool.scoped(|scope| {

                    scope.execute(|| {

                        count = self.detect\_collisions();

                    });

                });

                count

            };

            println!("Collision count: {}", collision\_count);

            thread::sleep(Duration::from\_millis(500)); // Delay between iterations.

        }

    }

Reflection:

In my code no locking is required as long as you don’t increase the number of threads from 1. Currently there is no simultaneous reading as collision detection happens once the movements have happened as we use a scoped pool. Currently there are no race conditions either as there is no simultaneous reading and writing. If I changed the number of threads to check collisions in parallel or decided to run both movement and collisions concurrently then there would be issues. To optimise the code, I could use a spatial data structure to check for collisions as currently I’m just checking every pair of particles. I could also make collision detection calculations run in parallel which would require locking.

Exercise 2:

struct ParticleSystem {

    particles: Vec<Particle>,

    collision\_count: AtomicUsize,

}

 fn detect\_collisions\_atomic(&self) -> usize {

        self.collision\_count.store(0, Ordering::SeqCst);

        let n = self.particles.len();

        let mut pool = Pool::new(NUM\_OF\_THREADS as u32);

        let chunk\_size = (n + NUM\_OF\_THREADS - 1) / NUM\_OF\_THREADS;

        pool.scoped(|scope| {

            for chunk\_start in (0..n).step\_by(chunk\_size) {

                let chunk\_end = (chunk\_start + chunk\_size).min(n);

                let collision\_counter = &self.collision\_count;

                let particles = &self.particles;

                scope.execute(move || {

                    for i in chunk\_start..chunk\_end {

                        for j in (i + 1)..n {

                            if particles[i].collide(&particles[j]) {

                                collision\_counter.fetch\_add(1, Ordering::SeqCst);

                            }

                        }

                    }

                });

            }

        });

        self.collision\_count.load(Ordering::SeqCst)

    }

Reflection:

By using an atomic counter, we can use multiple threads for the collision detection logic. In my code we create the counter and set it to 0 and then create a thread pool of 4 threads. Then the threads are given four equal chunks and then they are scoped. The atomic counter will not let more than one thread write to it at a time, so it is safe from race conditions. The fetch add method increments the counter. The Ordering::SeqCst makes atomic updates behave as one global sequence ensuring that every increment is seen in the same order by all threads, and thus preventing bugs.

impl ParticleSystem {

    fn new() -> Self {

        let mut rng = rand::thread\_rng();

        let mut particles = Vec::with\_capacity(PARTICLE\_COUNT);

        for \_ in 0..PARTICLE\_COUNT {

            let x = rng.gen\_range(0.0..ENCLOSURE\_SIZE);

            let y = rng.gen\_range(0.0..ENCLOSURE\_SIZE);

            particles.push(Particle::new(x, y));

        }

        Self {

            particles,

            collision\_count: AtomicUsize::new(0)

        }

    }

    fn detect\_collisions(&self) -> usize {

        let n = self.particles.len();

        let mut collisions = 0;

        for i in 0..n {

            for j in (i + 1)..n {

                if self.particles[i].collide(&self.particles[j]) {

                    collisions += 1;

                }

            }

        }

        collisions

    }

    fn detect\_collisions\_atomic(&self) -> usize {

        // Reset the counter

        self.collision\_count.store(0, Ordering::SeqCst);

        let n = self.particles.len();

        let mut pool = Pool::new(NUM\_OF\_THREADS as u32);

        let chunk\_size = (n + NUM\_OF\_THREADS - 1) / NUM\_OF\_THREADS;

        pool.scoped(|scope| {

            for chunk\_start in (0..n).step\_by(chunk\_size) {

                let chunk\_end = (chunk\_start + chunk\_size).min(n);

                let collision\_counter = &self.collision\_count;

                let particles = &self.particles;

                scope.execute(move || {

                    for i in chunk\_start..chunk\_end {

                        for j in (i + 1)..n {

                            if particles[i].collide(&particles[j]) {

                                collision\_counter.fetch\_add(1, Ordering::SeqCst);

                            }

                        }

                    }

                });

            }

        });

        self.collision\_count.load(Ordering::SeqCst)

Reflection:

I created a new system that randomly places the particles and then calls both versions of the collision detection. It is then timed as to how quickly the an answer is given. I also double checked that the order of the method being called was not effecting things at all.

Non threaded called first:

Non-threaded collision detection: count = 26, time = 267.9µs Threaded collision detection: count = 26, time = 398.8µs

Non threaded called second:

Threaded collision detection: count = 37, time = 420.8µs

Non-threaded collision detection: count = 37, time = 271.5µs

This is because of the overhead required with threading. We would need a much larger particle size for the benefits to be seen.