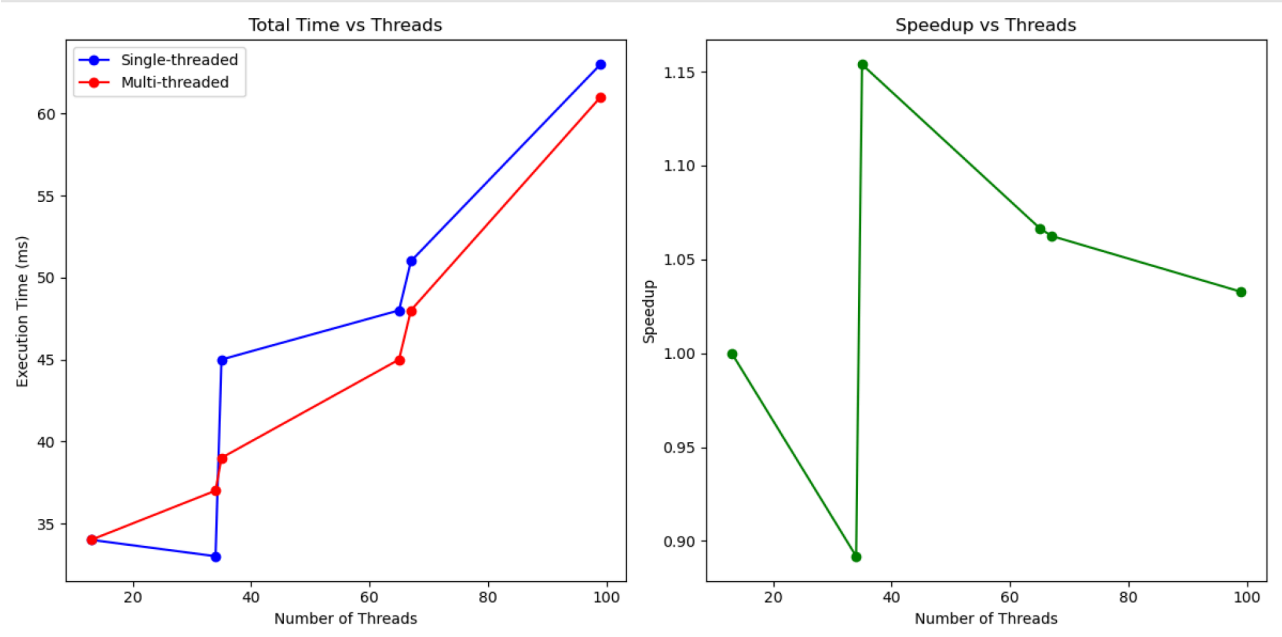
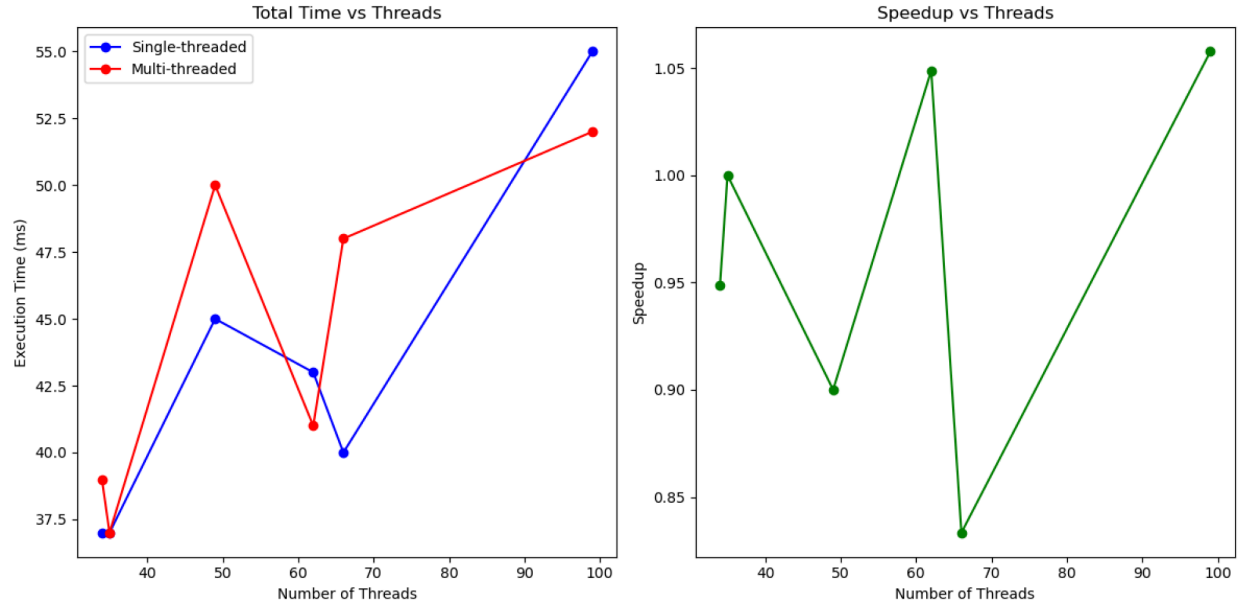
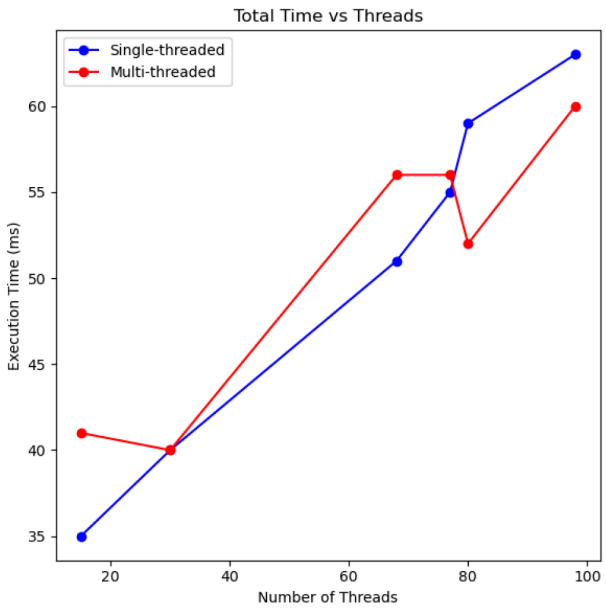
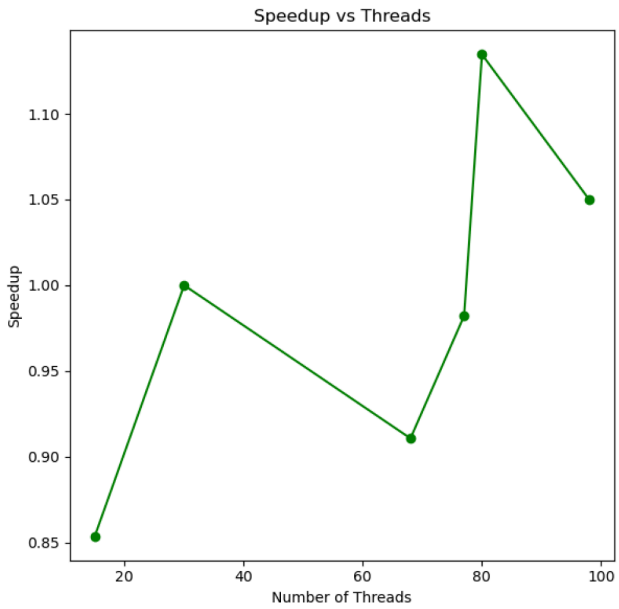
Testing q1.java with test.sh bash script

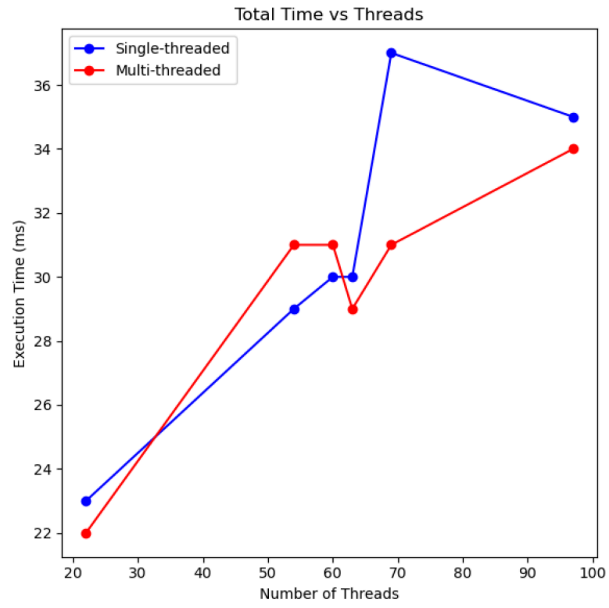
Consecutive testing on input.
Input: ./test.sh
Outputimage.png size w = 1024 and h = 1024
On default values of t = 1, 10, 20, 40, 60, 80, 100
Number of snowmen n = 100

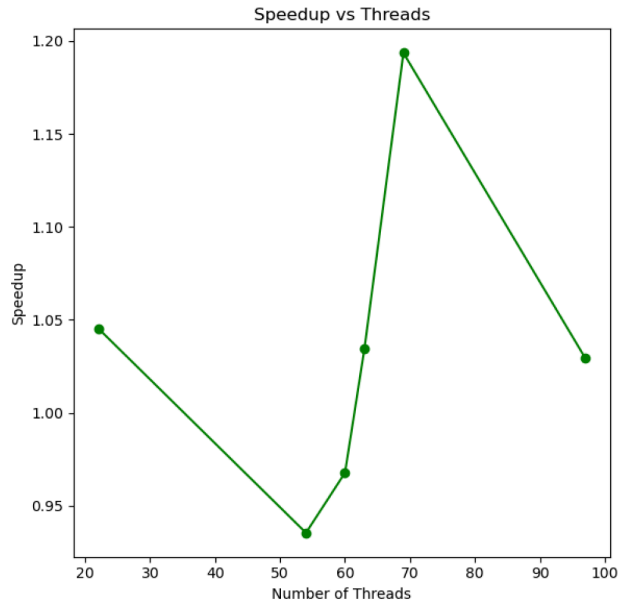












Result explanation

Strategy:

The idea was to create a method for drawing circles in the q2 class, which would then be used in the nested class of q2, the snowman class, to draw the three circles of the snowman. The drawCircle() method has a lock in the critical section where the shared variable outputimage.png is accessed and modified. In the snowman class there is a drawSnowman() method which is entirely locked because if 2 or more threads are accessing the method at a time then there will be overlap between snowmen drawn since threads would not be able to check the validity of the positions at which they choose to draw. This is because the entirety of

the process is selecting the position on the shared canvas (access to charred data) and then drawing it on the canvas with the draw circle method.
The strategy was to put a reentrant lock in the circle drawing process and a reentrant lock in the snowman drawing process at the moment in the code when shared data needs to be accessed.

```
// Reentrant lock for drawing a complete snowman
 public static ReentrantLock snowmanLock = new ReentrantLock();
 public static ReentrantLock lock  = new ReentrantLock();
```

```
if(outputimage.getRGB(pX, pY) == 0){
//reentrant lock allows one thread at a time to access to shared outputimage
        lock.lock();
        try{
```

```
//logic for drawning snowmen
public void drawSnowman(){
snowmanLock.lock(); // Lock the entire snowman drawing process
try{
        int size = 8 +rand.nextInt(20); //Random size for radius
```

Explanation of results:
Thread behavior and speedup:
Speedup = single time (ms) / multithreaded time (ms)
Overall trend represents an up and down trend of increased speed due to multithreading and loss of speedup. The results consistently show dips in speedup around 40-60 threads before the multithreading regains advantage in speedup relative to the single-threading. In fact, there is a trend in alternate periods of speedup advantage and loss following each other.
It seems like there is overhead that increases when more threads are involved when competing for the lock. Then after the trial when the speedup loss is observed as overhead increases, for some consecutive trials, with increased thread count, speedup advantage is regained.
This is probably due to some adjustments in thread local cache storage, as well as the cache storage of processors.
Notably, as seen from the graphs to the right of each sample output result, most of the time there is little thread contention unless there is an observed speedup due to multithreading or a loss of speedup due to increased overhead computations.

On a 12 core processor:

$ nproc

12

Therefore, running threads beyond 12 threads results in context switching results in overhead.

When multiple threads access shared data cache coherence mechanisms introduce stalls. Nevertheless at higher thread counts, some runs improve load balancing, which decreases idle time and allows for regains speedup. The observed alternating pattern in speedup could be due to how threads are distributed across logical cores.