Jeremiah Hanson
N-Body Collision Report

For this project I first created a sequential program that creates the required body objects. It sets the speed of the objects randomly and the direction towards the middle of the field. Then the program runs through the calculations to determine where each body will go for the next time step. If collisions are detected then it will change the velocities of both objects based on the given formulas then continue to calculate the remaining positions. After all bodies have been given new positions, the current positions and velocities are stored in an array held by each body object.

The parallel program creates the bodies in a similar manner but the calculations are done differently. It requires at least two threads because one thread is dedicated to calculating the collision. I did it this way because collisions are not always going to happen every time step and when they do happen its not always going to have to calculate collisions for every object. Then then remaining threads are given out bodies to calculate positions for distributed evenly.

For both programs, the results are written to a file and the times are printed to stdout. None of this is part of the timing. to test the functionality of the programs I created a simple gui that draws each step and is controlled by two buttons, one for going forward and one for going back. This gui is run by adding an argument "true" to the end of the first 4 arguments.

Following are some of the results of my testing, which I did on Oxford.

```
test1: java SequentialCollision 1 30 50 1000
                 time: 0.189
         java ParallelCollision 4 30 50 100000
                 time: 0.292
         java ParallelCollision 5 30 50 1000
                 time: 0.356
         java ParallelCollision 9 30 50 1000
                 time: 0.417
         java ParallelCollision 25 30 50 1000
                 time: 0.602
         java ParallelCollision 32 30 50 1000
                 time: 0.652

test2: java SequentialCollision 1 30 50 10000
                 time: 1.064
         java ParallelCollision 4 30 50 10000
                 time: 1.239
         java ParallelCollision 5 30 50 10000
                 time: 1.598
         java ParallelCollision 9 30 50 10000
                 time: 1.528
         java ParallelCollision 25 30 50 10000
                 time: 5.553
         java ParallelCollision 32 30 50 10000
```

```
                 time: 6.753

test3: java SequentialCollision 1 30 50 100000
                 time: 9.320
         java ParallelCollision 4 30 50 100000
                 time: 6.986
         java ParallelCollision 5 30 50 1000000
                 time: 11.252
         java ParallelCollision 9 30 50 100000
                 time: 19.038

test4: java SequentialCollision 1 40 50 100000
                 time: 15.429
         java ParallelCollision 4 40 50 100000
                 time: 9.466
         java ParallelCollision 5 40 50 1000000
                 time: 10.754
```

From these results I have come to the conclusion that for only a few time steps its more efficient to run this program sequentially. However, once the time steps get closer to 100,000 running with four processes was faster. Then by added 10 more bodies, running with 5 processes became faster than only 1 but slower than 4. Since Oxford is running with 4 cores it makes sense that the four thread version would be the best of the parallel programs since it can run one thread per core. This reminded me that my main thread is still active so I ran test 4 again with only 3 threads and the time was 8.280. I think to improve this I would need to put the main thread to sleep till all the others have finished.