Ramez Nahas
I.D#: 26718108
Assignment 2 Report October 21, 2019

Overview:

Since the design of my program in Assignment 1 had a mixture of embarrassingly parallel patterns, which could map onto Intel TBB's basic algorithms (parallel_for), and tasks that were not easily mappable, the idea was to create TBB tasks (representing the tasks not easily mappable) that would themselves execute the embarrassingly parallel tasks. This way thread synchronization is handled by the TBB scheduler, which will also optimize parallelization when it can.

Tasks:

I created a namespace "tasks" that includes all the different task structs that extend "tbb::task" and which are needed to accomplish the bouncing ball simulation. They include tasks::update, tasks::wall_collision, tasks::ball_collision and tasks::draw.

The logic is: control thread creates tasks::update. Its job is updating the balls' velocities and centers with the new values. Once tasks::update is done, it creates tasks::wall_collision which computes all wall collisions and once done, creates tasks::ball_collision. This task handles the ball collisions and once done, it calls tasks::draw to draw the balls.

The parallelization of the tasks is handled by the TBB task scheduler. More specifically, each task must be handled in the order specified above, however the computation in the task itself is parallelized, using the structs in the namespace "parallel".

Parallel:

This namespace includes all the different structs that handle the embarrassingly parallel computations. Tasks (above) call parallel_for using parallel::update, parallel::wall_bounce and parallel::ball_bounce, depending on which task is being executed. Since tasks::draw is serial, no parallel::draw exists.

There are two more parallel tasks, parallel::init_balls (creates random balls in parallel), and parallel::init_pairs (creates the unique ball pairs in parallel).

Optimization:

All optimization I could control was done in Assignment 1. The rest is handled by Intel TBB.