Mathieu Lajoie 40026331
Multicore Programming
Assignment #2

Introduction

The assignment consists of re-implementing assignment #1 to use Intel's Thread Building Blocks. We are to make use of data and task parallelism to redesign the control and computation threads that we had implemented in the first assignment.

Architecture

The assignment uses OpenGL version 3.3 in immediate mode for simple drawings. The main source file *main.cpp* contains most of the implementation logic. It creates the particles, initialises the state and performs the drawing. In order to conform to the requirement of thirty frames per second, I implemented the following system.

```
while(!glfwWindowShouldClose(window)) {

    this_time = clock();
    time_counter += (double)(this_time - last_time);
    last_time = this_time;

    if(time_counter > (double)(NUM_SECONDS * CLOCKS_PER_SEC)) {
        time_counter -= (double)(NUM_SECONDS * CLOCKS_PER_SEC);
        processInput(window);
        updateAndDraw();

        glfwSwapBuffers(window);
        glfwPollEvents();
    }
}
```

The loop is the main draw loop called by GLFW. The code inside calculates if it has been 1/30th of a second since the last time the code inside the if-statement was run. This method allows me to stay in control of the loop, and perhaps run other things while it does not run, as opposed to using a sleep function. Every interval, as set by the num_seconds constant, the code inside the if-statement is run. It should be noted that this method requires the use of some time library of C++.

The galaxies could be generated procedurally very easily, but for the purpose of the demo I decided to hard-code them for a best result. Two galaxies are created through simple random generation of points in a circle using trigonometric functions. An orbital velocity is then applied so that particles spin around the center particle, representing a galaxy. I opted

to not scale the particle drawing to its mass, as that could cause the display to turn to a weird state.

I implemented a class called Particle which contains all pertinent information about a particle. This includes, position, velocity, acceleration and mass. It also has many utility functions to be used in conjunction with the TreeNode class, which represents the QuadTree that is used in a Barnes-Hut algorithm. Constants for calculations are stored in the constants.h file. Every cycle, the tree is deleted, then rebuilt and the forces are recalculated.

The source code is available in the joined files and is accompanied by informative comments.

Parallelizing

The multithreading aspect of the program was done through Intel's Thread Building Blocks as was required of us in the assignment requirements. To accomplish this, I made us of a parallel loop, and a task group. Below is an image of the code.

```
void ParallelTBB() {
    tbb::parallel_for(tbb::blocked_range<size_t>(0, TOTAL_PARTICLES - 1),
        [&](tbb::blocked_range<size_t>& r)
    {
        for (int i = r.begin(); i < r.end(); ++i)
        {
            Vec2D force = root->calculateTreeForce(*particles[i]);
            particles[i]->acc_x = force.x / particles[i]->mass;
            particles[i]->acc_y = force.y / particles[i]->mass;
            particles[i]->computeVelocity();
            particles[i]->calculatePosition();
        }
    });
}
```

As you can see, I do not set a grain size as TBB does a great job balancing everything by default, as their documentation explains. The scheduler will assign threads by itself when they are available.

I also decided to use a task group, because I do perform my rendering through task parallelism later on in the code. Using this group allows me to organize the parallel code much better than without it.

```
tbb::task_group g;
g.run([&] {ParallelTBB(); });
g.wait();
```

I can then wait for the loop to finish, and then start rendering using OpenGL calls afterwards. It makes everything more neat and organized, as opposed to what it would look like otherwise.

```
tbb::task_group r;
r.run([&] {
    for (int i = 0; i < TOTAL_PARTICLES; ++i) {

        // Check if particle is outside the bounding box
        if (!(particles[i]->x > windowWidth || particles[i]->x < 0 || particles[i]->y > windowHeight ||
            particles[i]->y < 0)) {

            // Set color as specified in Particle object
            glColor3f(particles[i]->color.red, particles[i]->color.green, particles[i]->color.blue);

            // Draw particle
            glVertex3d(particles[i]->x, particles[i]->y, 0.0);
        }
    }
});
r.wait();
```

As you can see above, rendering in parallel is not actually difficult. This code works flawlessly with one exception. It seems that accomplishing this task in parallel causes a slight stutter in the rendering, as it does cut to black once in a while. I will still showcase this in the demo for the purpose of getting a point.

A visible improvement from the sequential algorithm was observed when switching to the parallel implementation. It is showcased in the table below (table 1).

Table 1: Maximum particles without stutter per method at theta = 1.

	Sequential	Parallel [TBB]
Maximum Particles	~6000	~18000

Switching from sequential to parallel resulted in a speedup of around 150%.

Difficulties

The article provided by the professor was not very clear, nor did it provide any details on accomplishing what the video the author posted performed. It simply solved the n-body problem. I also had trouble doing the proper calculations, as my physics classes are very far in my memory. It took me over three hours to realize that I had forgotten about f = ma, and was just setting the force to be the acceleration, which is wrong. Many libraries that the professor recommended in class actually turned out to be dead, forcing me to find alternatives on the web.

Using TBB in CLion on Windows turned out to be nightmare. After over 8 hours of searching, I was forced to change to Visual Studio and use TBB through their package manager, called Nuget. Subsequently, the assignment went much more smoothly.