Assigned:	Mon, Sep 23, 2019 11:59 PM
Due:	Wed, Oct 9 2019 11:59 PM
Last Day to Handin:	Sat, Oct 12 2019 11:59 PM

1 Overview

Before you begin, please take the time to review the course policy on academic integrity:

Academic Integrity Policy

Download the Assignment 3 starter code from the course Github using:

$ git clone https://github.com/cmu15418/assignment3-f19.git

1.1 Assignment Objectives

In this assignment, you will implement an *n-body simulation*, which is a prediction of the movements of a large number of particles that interact with one another gravitationally. You will need to finish implementing a portion of the sequential version of this simulation, and study the performance of the finished sequential version. Then you will be directed to use OpenMP to improve the programs' performance through shared memory parallelism.

You will need to instrument your code to determine where the most time is spent in computation and evaluate where optimizations are most valuable. You will also need to focus on avoiding sequential bottlenecks, memory contention, and workload imbalance.

1.2 Machines

The OpenMP standard is supported by a variety of compilers, including GCC, on a variety of platforms. Programs can be written in C, C++, and Fortran. For this assignment, you will be working in C++. You can test and evaluate your programs on any multi-core processor, including the GHC machines.

For performance evaluation, we will be using the GHC machines for testing.

1.3 Resources

There are many documents describing OpenMP, including those linked from the OpenMP home page at http://www.openmp.org. Like many standards, it started with a small core of simple and powerful concepts but has grown over the years to contain many quirks and features. You only need to use a small subset of its capabilities. **A good starting point is the document at https://www.cs.cmu.edu/~418/doc/openmp.pdf.**

2 Introduction

In this problem, we will simulate the movement of particles with varying mass and initial position. Each particle is defined as a tuple of id, mass, initial velocity and initial position. Particles are affected by gravitational forces from nearby particles.

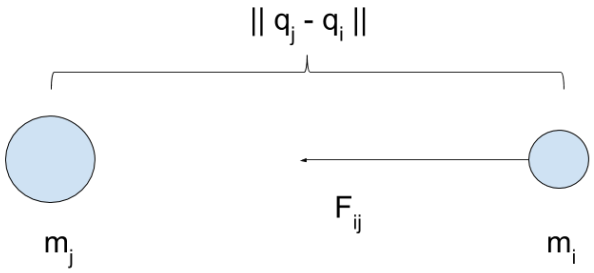

Figure 1: illustration of gravitational force between two particles

The n-body problem considers n point masses $m_i, i = 1, 2, \ldots, n$ in an inertial frame of reference in three dimensional space R^3 moving under the influence of mutual gravitational attraction. Each mass m_i has a position vector q_i. Newton's second law says that mass times acceleration $m_i \frac{d^2 * q_i}{dt^2}$ is equal to the sum of the forces on the mass. Newton's law of gravity says that the gravitational force felt on mass m_i by a single mass m_j is given by

$$\mathbf{F}_{ij} = \frac{G m_i m_j \left(\mathbf{q}_j - \mathbf{q}_i\right)}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|^3},$$

where G is the gravitational constant and $\|\mathbf{q}_j - \mathbf{q}_i\|$ is the magnitude of the distance between q_i and q_j (metric induced by the l_2 norm).

In the simulation you will write, particles only exist in a 2D world, with x and y dimensions. We will also use an approximation; past a predefined radius, particles will not affect other particles, since their gravitational force is almost negligible.

2.1 Computing Gravitational Force

In this assignment, you do not need to implement the physical formula that defines the amount of gravitational forces. We provide a function, `computeForce(Particle p1, Particle p2, float radius)`, in the file `world.h` that returns the gravitational force between particle p1 and p2. You are not allowed to modify this function. It is summarized as the following;

```
computeForce(Particle target, Particle attractor, float radius) =
       dist = distance between particles
       dir = direction from target to attractor particle
       force = dir * target mass * attractor mass * (Gravity / (dist *
           dist));
       return force
```

The `radius` parameter is a predefined input. The `computeForce` function is guaranteed to return 0 if the distance between p1 and p2 are greater than `radius`.

2.2 Simple Simulation Algorithm

We provide a simple sequential simulation algorithm, which has no performance optimizations, called 'simple simulator.' It computes the total force on every particle p_i by summing the individual forces that result from the gravitational interactions of p_i and every nearby particle p_j. However, it only considers

particles that are within a certain radius of p_i, because particles outside that radius have a negligible force effect.

The pseudo-code for this implementation is as follows:

```
for each iteration
    for each particle p:
        find nearby particles whose distance to this particle is less
            than 'radius'.
        // compute the sum of gravitational forces that need to be
            applied to this particle:
           force = 0;
           for each nearby particle p1
              force += computeForce(p, p1);
        update the position and velocity of this particle using the
            computed force.
```

This algorithm is implemented in `src/simple-simulator.cpp`, and is available for your reference. The accuracy of you quad-based sequential and parallel implementations will be checked against output generated by this code.

The interesting part of this problem, is to find the nearby particles of a given particle efficiently. A naïve approach, implemented in `src/simple-simulator.cpp`, is to test the distance of all particles in the scene. This yields an implementation with $O(n^2)$ computational complexity, where n is the number of particles in the scene. When n is very large (e.g. 1 million), the naive approach will take a very long time to finish. We can improve this significantly.

3 Quad-Tree Simulation Algorithm

To more efficiently find the number of 'near' particles to a specific particle p_i, we can create a tree structure to quickly determine which particles are potentially in the 'radius' of p_i. In this case, we will be using a simple tree structure called a 'quad-tree'.

3.1 Quad Tree

A Quad Tree is a tree data structure in which each node has exactly four children. They are useful because they can be used to partition a two-dimensional space by recursively dividing into 4 sub-spaces.

For this n-body simulation, we are interested in partitioning our 2D space into rectangles, such that each 'leaf' of the tree contains at most 'QuadTreeLeafSize' particles. Given a list of particles and their positions, we can build this quad tree by recursively bisecting the space containing those particles into nodes,

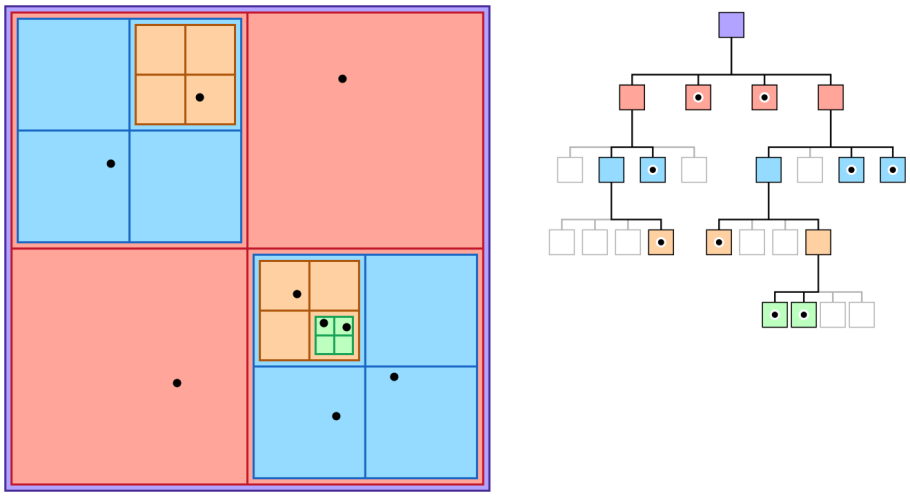

Figure 2: A Quad tree with 'QuadTreeLeafSize' = 1

while keeping track of bounds of the current square of space used. For each node, we must store the following:

1. isLeaf: whether the node is a leaf
2. particles: What particles that node contains
3. children: The four child nodes of that node

For your reference, the tree building process for particles in a 2D space might look something like this:

```
QuadTreeLeafSize = max number of particles per node
buildQuadTree(particles, range_min, range_max){
    if num(particles) >= QuadTreeLeafSize:
        return leaf node
    else:
        result = non-leaf node
        pivot = (range_min + range_max) * 0.5f;
        <assign particles to appropriate children of result>
        return result
}
```

Note that in this algorithm, a leaf node may contain 0 particles. The pseudo-code is purposefully vague, as the implementation of the 2D version is left to you in Task 1.

3.2 Simulation Algorithm

When using a quad tree structure, each simulation iteration is separated into two stages: building the quad-tree, and using the quad-tree for simulation:

```
for each iteration
    1. build quad-tree
    2. for each particle p:
        find nearby particles using the quad-tree
        compute the sum of gravitational forces that need to be applied
            to this particle:
           force = 0;
           for each nearby particle p1
               force += computeForce(p, p1);
        update the position and velocity of this particle using the
            computed force.
```

In this assignment, you will first finish implementing a sequential version of the simulation by completing a function building a quad tree, then parallelizing both the tree construction and simulation stage on a Intel® Xeon® Processor E5-1660 v4.

4 Starter Code

We provide starter code, which includes the simple simulator and an incomplete version of the sequential tree-based algorithm.

4.1 Building the Starter Code

After cloning this repository, you can make the started code using:

```
$ make all
```

This will build an executable named `nbody-release`. To perform a debug build, run:

```
$ make all CONFIGURATION=debug
```

4.2 Running the Startup Code

To run a simulation, use the following command:

```
$ ./nbody-release -i 10 -n 100
```

This will create a scene with 100 randomly positioned particles and run for 10 iterations. The resulting particle positions and velocities will be dumped to `out.txt` by default. You can use `-o out1.txt` command-line option to change the output destination.

4.3 Visualization

You can have `nbody-release` output a visualization of the particles after each iteration. This visualization will be a 2D image of particle positions, formatted as a bitmap image file (.bmp), which is viewable in most image editors. To create a visualization, use the command-line option `-fo`:

```
$ ./nbody-release -i 10 -n 100 -fo ./
```

This informs `nbody-release` to output bmp files to the current directory that visualize each simulation iteration. By default, the visualization represents a viewport of -10$\leq x \leq$10 and -10$\leq y \leq$10. To change the viewport size, use `-v` option:

```
$ ./nbody-release -i 10 -n 100 -fo ./ -v 20.0
```

This changes the viewport to visualize the -20$\leq x \leq$20 and -20$\leq y \leq$20 range.

4.4 Running the Benchmark

Running the following command will start a benchmark of your sequential and parallel implementation:

```
$ ./nbody-release -b
```

The benchmark will time your implementation on a variety of scenes ranging from 100 to 100,000 particles. It will also check the correctness of your simulation by comparing your simulation result to the reference results.

4.5 Usage

`nbody-release` has both testing capabilities and 'custom run' capabilities. A custom run creates a scene with 10 clusters of particles, with a number of options. The full usage of `nbody-release` is as follows:

Custom Run Options:

- `-i <int>`: This flag specifies the number of iterations, a positive integer 1 or greater, for a custom run. The default value is 1 iteration.
- `-n <int>`: Specifies the number of particles to generate.
- `-s <float>`: This flag specifies the space size that the particles for a custom run will be generated in. The default is 10.0, for which particles may have x positions in the range -10.0, 10.0 and y positions in the range -10.0, 10.0.
- `-in <string>`: Specifies an input file to read particle positions from in a custom run. The file must be a `.txt` type file with the format `mass, x position, y position, x velocity, y velocity`, space separated, with one line corresponding to each particle.
- `-o <string>`: Specifies the output file to write results to. The default is `out.txt`.
- `-fo <string>`: When this flag is included, a .bmp visualization file will be created for each step of the simulation, labeled by step number, in the directory <string>. For example, `-fo "./"` will create bmp files in same directory as nbody-release.
- `-v <float>`: Specifies the viewport size, which is the range of space the visualization file will use. Defaults to 10.0; in general try to keep this value to the space bounds specified with -s.
- `-seq`: Runs the sequential quad-tree version of the simulation.
- `-par`: Runs the parallel quad-tree version of the simulation.
- `-simple`: Runs the provided simple version of the simulation.
-
- `-c`: The addition of this flag specifies to run the correctness check suite for the specified implementation (-seq or -par). If none are specified, the correctness of the sequential implementation is checked by default

5 Your Tasks

Your tasks will be to complete the sequential tree-based implementation and to implement a parallel simulation implementation.

The file `simple-simulator.cpp` provides the naive implementation that checks all particles for proximity. Make sure you understand this code before proceeding to Step 1.

5.1 Task 1: (20 Points) Implement Sequential Quad-Tree Based Simulation

The file `seq-simulator.cpp` provides the function definitions for the sequential simulation. You need to implement the `buildAccelerationStructure(..)` and `simulateStep(..)` function.

The `buildAccelerationStructure(..)` function takes a vector of current particles, and should return a `QuadTree` object. See `quad-tree.h` for the definition of `QuadTree` class. **You may not delete or modify the contents of this file, but you may define new functions or structures if you believe them to be useful.**

A QuadTree has the following properties:

- root: A QuadTreeNode
- bmin: A Vec2 with <x,y> minimum bounds on the locations of particles in the tree
- bmax: A Vec2 with <x,y> maximum bounds on the locations of particles in the tree
- getParticles: A function that assigns the list of particles in this tree to a pointer provided as a parameter.

A QuadTreeNode has the following properties:

- isLeaf: Boolean labeling if this node is a leaf
- children: Array of 4 QuadTreeNode pointers representing the children of this node
- particles: List of particles at this node

The `simulateStep(..)` function implements stage 2 of the simulation. It reads particle information from the `particles` parameter, and stores the new particle status in the `newParticles` vector. It should be very similar to the simple implementation.

Evaluation

Your sequential program will be evaluated based off of correctness. You can check the correctness of your sequential quad-tree build implementation and `simulateStep` function by running:

```
$ make
$ ./checker.pl -s
```

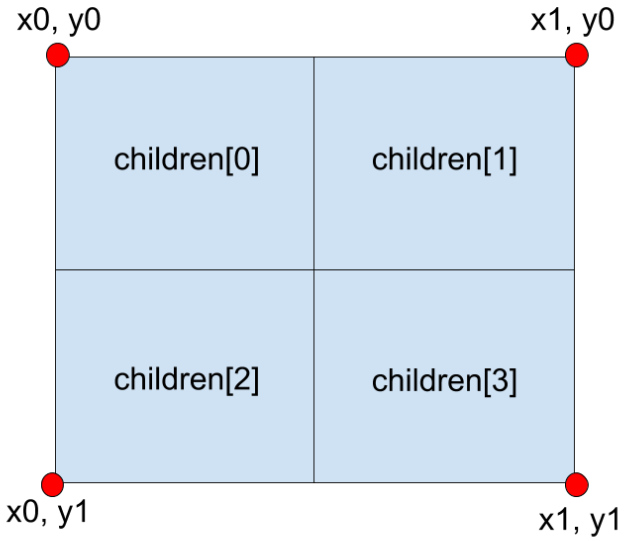

Figure 3: Diagram of children mapping of quad tree

5.2 Task 2: (60 Points) Implement Parallel Quad-Tree Based Simulation

For this task, you need to implement the `buildAccelerationStructure(..)` and `simulateStep(..)` functions in `parallel-simulator.cpp`. You may find it useful to start from your sequential implementation from Task 1.

Your implementation must perform well even in scenes with imbalanced particle distribution. It will be tested against a total 7 scenes, each shown in the gallery at the end of this writeup. It will benefit you to look at these scenes and consider them when writing your implementation.

Tips and Tricks

- Consider the use of task-based instead of data-based parallelism. What task best suits which type of parallelism? Blindly creating `#pragma omp parallel for` loops will not serve you well in this task.
- Research the different types of scheduling OpenMP provices, and choose the type that best suits the problem at hand. The following article has great resources explaining how different types of scheduling handle work.

  http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

- You may find tuning constants involved in your implementation useful.

Evaluation

The benchmark shows the performance for your parallel implementation, including speedup information and correctness errors. It also displays the target speedup for each scene.

```
$ make
$ ./checker.pl -p
```

You will be evaluated both on correctness and speed; an implementation that is not correct with not receive any points, so ensure that correctness does not fail. Although you could improve the apparent speedup by detuning the single-core performance, that will not lead to an optimal overall score, as we will look at your code. As a special case, if your single-core performance exceeds the target performance, then the target performance will be used as the denominator in the speedup calculation. This means that you don't need to artificially slow down a fast, single-threaded version in order to maximize the speedup measurement.

Scoring of your performance is broken down as follows:

Table 1: Breakdown of Points.

Function Performance	Points
buildAccelerationStructure	40
simulateStep	20

Your performance for different scenes will be weighted in your overall score. Below are the percentage weights for each scene. As an example; random-1000 will contribute $.05 * 20 = 1$ point to your total score.

Table 2: Percentage weight of different scenes.

Scene	Percent
random-1000	5
random-10000	15
random-50000	20
corner-1000	5
corner-10000	15
corner-50000	20
repeat-10000	20

6 Write-up

In your writeup, please include the following:

1. Briefly describe your sequential implementation and explain your parallel implementation in your write-up.
2. Include your reasoning for your implementation choices, including any graphs or tables that helped you make your decisions.
3. Include timings for different parts of the program, for at least the `buildTree` and `simulateStep` functions in the parallel implementation.
4. Include the benchmark output as a result of running

```
$ ./checker.pl -p
```

7 Hand-in Instructions

You will submit your code via Autolab and your report via Gradescope. For the code, you will be submitting all C++ header and source files in the src folder.

1. Submitting your code:
   (a) If you are working with a partner, form a group on Autolab. Do this before submitting your assignment. One submission per group is sufficient.
   (b) Make sure all of your code is compilable and runnable. We should be able to simply run make, then execute ./checker.pl -p. Please remove excessive print statements, if they were added.
   (c) Run the command "make handin.tar." This will run "make clean" and then create an archive of any C++ source code in /src. If you find it absolutely necessary, you may modify the Makefile to include other files you have added.
   (d) Submit the file handin.tar to Autolab.
2. Submitting your writeup:
   (a) Please upload your report as file report.pdf to Gradescope, one submission per team, and select the appropriate pages for each part of the assignment. After submitting, you will be able to add your teammate using the add group members button on the top right of your submission.

8 Gallery

Your implementation will be tested against the following scenes. They all have different distributions of points, different space sizes. Taking into account possibly uneven distributions may help your implementations' performance.

The red lines overlaid on the following images visualize the quad-tree based partition of the particles.

N = Number of particles in Scene

S = Space Size, where particles can take any position with x coordinate in (-S, S) and y coordinate in (-S, S).

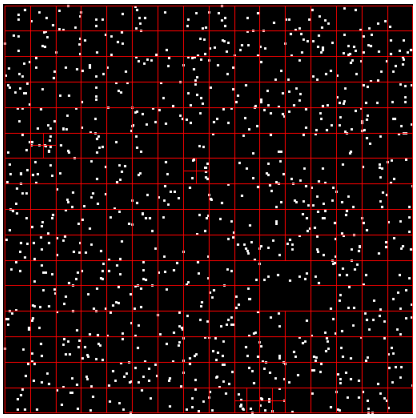

Figure 4: Random-1000: N=1000, S=10.0, randomly distributed points

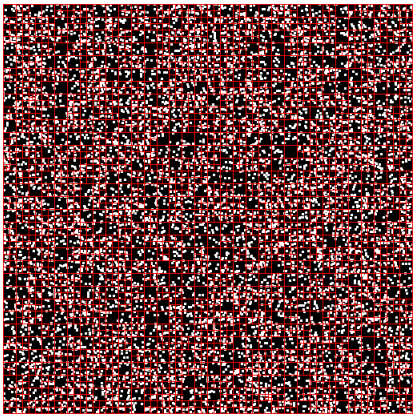

Figure 5: Random-10000: N=10000,S=100.0, randomly distributed points

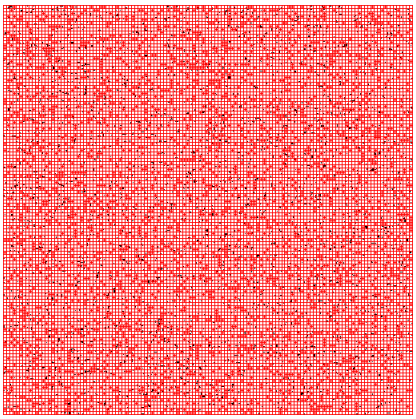

Figure 6: Random-50000: N=50000, S=500.0, randomly distributed points

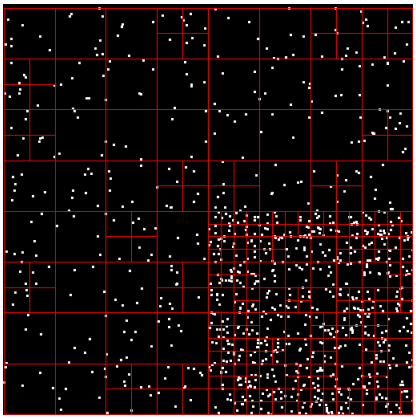

Figure 7: Corner-100: N=1000, S=10.0, Corner associated points

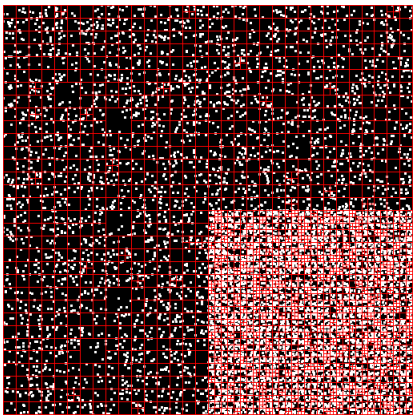

Figure 8: Corner-10000: N=10000, S=100.0, Corner associated points

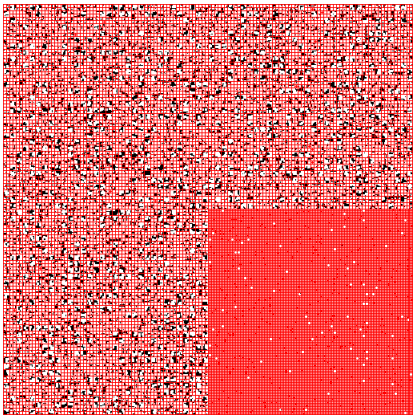

Figure 9: Corner-50000: N=50000, S=500.0, Corner associated points

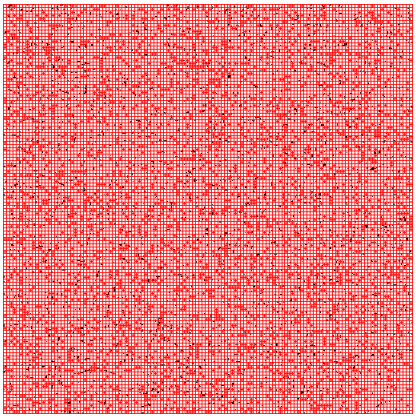

Figure 10: Repeat-10000: N=1000, S=100.0, Random distributions with many iterations (50)