CSE 6220 N-Body Problem (Barnes-Hut) – OMP

In this lab, you will implement a parallel version of the Barnes-Hut algorithm for solving n-body problems. The efficiency of your code will be compared to a reference implementation to help determine your grade.

Getting Started

Begin by obtaining the starter code from the github repository.

```
git clone --recursive https://github.gatech.edu/omscse6220/lab2.git
```

Note that this is the GT github server, so you will need to authenticate with your GT credentials.

Optionally, you may choose use a git hosting service for your code. As always, please **do not make your repository public** on Github or another git hosting service. Anyone will be able to see it. If you feel the need to use a hosting service for your project, please keep your repository private. Your GT account allows you to create free private repositories on the GT github server.

Part 1: Barnes-Hut Approximation for the N-Body Problem

In classical mechanics, the n-body problem consists of predicting the motion of celestial objects interacting through gravity. That is, given the instantaneous positions and velocities of a group of objects at a given time, predict the gravitational forces and use these to predict the position and velocities for all future times.

In a computational approach to making these predictions, the forces at a given time are calculated, and then numerical integration is used to predict the positions and velocities a short time in the future. Repeating this process makes it possible to generate accurate predictions for celestial motion far into the future even when closed-form solutions are not possible.

Task 1: N-Body Exploration

In the first part of the lab, you are invited to explore various data sets and visualization techniques for this N-body problem. The repository contains a complete, though woefully slow, implementation. Later in the lab, you will work to improve its performance.

- An instance of the **Body** class represents one of the n bodies. It stores the body's position, velocity, mass, and it provides functions for exerting gravitational forces and performing leapfrog integration given forces on the object to update its position and velocity.

- The **ForceNaive** class implements the abstract class **ForceCalculator**. It takes an array of n bodies, say B as an argument to its constructor. The operator() then takes a body as an argument and applies the gravitational forces from the given set of bodies B.
- **UniverseState** is a convenience class that contains an array of n bodies, a time value, and operations for file I/O.
- The **Integrator** class handles the process of alternately computing forces and integrating positions given the number of steps and the step size. After each time step it calls a user-defined callback.
- The python files **barneshut_convert.py** and **nbody_convert.py** convert data from file formats used in a couple of labs from Princeton University into the UniverseState format. You can find files with a small number of bodies here and those with a large number of bodies here.
- The main file **nbody.cc** accepts a file containing the initial state as a command line argument, and it outputs as csv file representing the universe state over time. Type `./nbody --help or -h` to get more arguments and their default values.
- The file **visualize.py** uses python's matplotlib and pandas libraries to read the output csv data and produce a visualization.

To illustrate the usage of this code, we'll visualize the so-called spiral system. You should first acquire the data by downloading spiral.txt. Then run

> Note: You may need to rename the file from jZJa3X to spiral.txt if you download it using something like wget.

```
python nbody_convert.py spiral.txt > spiral.us
```

to convert the file into the UniverseState format. Then compile the nbody main file with

```
make nbody
```

and run it with

```
./nbody -f spiral.us -n 50000 -s 0.01 -r 500 > spiral.csv
```

Finally, you can visualize the results with

```
python visualize.py spiral.csv
```

There are no deliverables for this portion of the lab, but understanding this portion thoroughly is extremely important for Part 2. You are encouraged to explore, share tips on things like appropriate time step, share results, and share code for improved visualizations.

Task 2: Barnes-Hut Implementation

The 'naive' computational approach to the n-body problem computes the gravitational force induced by every body to every other body. This results in $O(n^2)$ work to compute all forces for a single timestep. The Barnes-Hut algorithm reduces the complexity to $O(nlog(n))$, which makes it tractable to closely approximate solutions for problem sizes that would otherwise be prohibitively expensive.

The central idea is grouping together bodies that are sufficiently close to each other and sufficiently far from the force calculation point, and approximating the group of bodies as one body with the total group mass located at the group center of mass. 'Sufficiently close' and 'sufficiently far' are defined by a simulation parameter called the multipole acceptance criterion (MAC), usually denoted by θ, which is a ratio of the group 'diameter' d and the distance r between the force calculation point and the group center of mass. In the figure below, diagram B represents a Barnes-Hut approximation of diagram A.

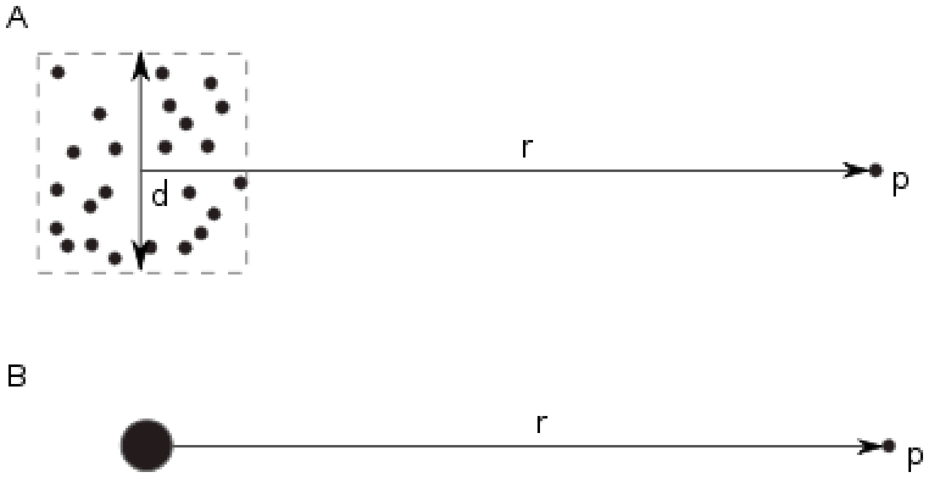

Figure 1: MAC

Generally, there is a tradeoff between accuracy and speedup. The lower the MAC, the more accurate but more expensive the computation, because only smaller groups (and hence a greater number of total groups) can be used in the approximation.

Quadtrees

To find suitable sets of bodies to group together, a Quadtree data structure is used. This approach recursively divides square regions of R^2 into 4 equal sub-squares. Thus, the root node corresponds to a square region encompassing all of the bodies and it's children correspond to the four sub-squares of equal size. Squares with no bodies in them are pruned from the tree.

In calculating the force on a particular body, the bodies within a square are grouped together, and if approximating these bodies as a single body meets the MAC (smaller than θ), the approximation is used. Otherwise, the contributions from the four sub-squares are summed together.

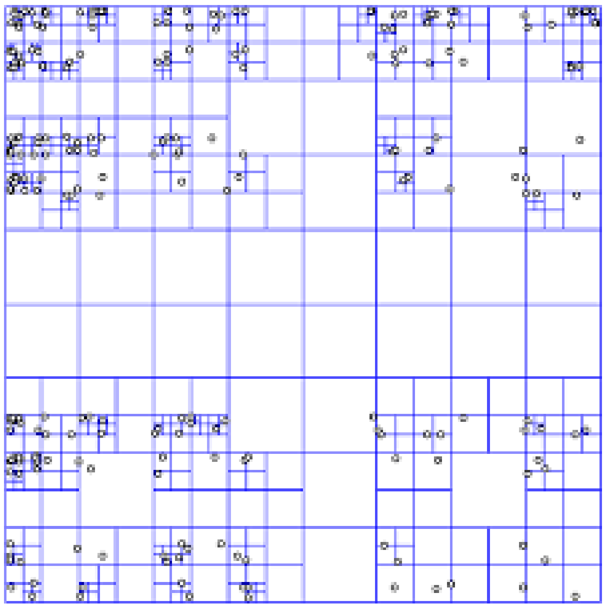

Figure 2: Quadtree example

For 3D data, there is the analogous octree data structure.

Deliverables

Your task for this part of the lab is to implement the Barnes-Hut approximation with a quadtree in the files **ForceBarnesHut.hh** and **ForceBarnesHut.cc**. Your ForceBarnesHut class should inherit from ForceCalculator and define the operator () so that it can be used in place of the ForceNaive class. You should construct the quadtree in the constructor and then use it to quickly compute the forces in the () operator member function. The MAC parameter is specified as the variable θ to the constructor.

Part 2: Parallel Barnes-Hut

What could be faster than the Barnes-Hut approximation? A parallel version, of course! And, indeed, creating a parallel implementation of the Barnes-Hut algorithm is the next part of the lab.

Given the quadtree, parallelizing the force calculation is a simple parallel loop. Parallelizing the construction of the tree, however, is a little more challenging. At first, the tree construction may seem like a curious thing to optimize. In the N-body simulations that you will have done, the dominant part of the computation was the computation of the forces *after* the construction of the tree. As the number of bodies grows, however, the share of computational time due to the construction increases. In addition, for other applications it may be important to compute the gravitational effect of a large number of bodies, not on each other, but on a smaller set of bodies, perhaps even in real-time. The real-time nature of the problem makes a fast Barnes-Hut approximation necessary, and it would be nice to compute the quadtree quickly so as to be sure to have the tree ready when needed.

Unfortunately, building a quadtree in parallel is not merely a matter of using a parallel for loop. Simply dividing the points among the cores would have them all attempting to modify the same quadtree creating locking or contention problems. One could have each core build its own quadtree, but that approach leaves the problem of merging the tree together, and it's not clear why merging the trees should be much faster than constructing them in a serial fashion.

The key to fast merging of the trees is to avoid having them overlap. If a subtree is present in one tree but entirely absent in an another, then the subtree can be added simply by marking it as a child of the appropriate node. We want to have as much of the merge happen in this way as possible. To achieve this, we use a MortonKey ordering of the bodies.

Task 3: Morton Ordering

Morton order is a space-filling curve that follows a 'Z' or 'N' pattern microscopically and macroscopically. This order helps us construct our quadtree efficiently. In our code, and as is common, the pattern has the shape of an 'Z'.

Consider the following image where points reside at each integral coordinate and are ordered in a Morton order starting from (7,7). The curve shape is that of a 'Z'.

You will notice that curves between individual points have a 'Z', but also the pattern between square regions of points are also 'Z'-shaped. The four quadrants of the whole diagram go bottom-right, bottom-left, top-right, then top-left.

We can discover the Morton order by forming Morton 'keys' associated with each point, and then sorting these keys. Each key can be determined by taking integer representations of the x and y coordinates (implemented as functions `x` and `y` in class `MortonKeyCalculator`), and interleaving the bits of these integer representations. The exact values of the integer representations do not really matter, just their relative values to the other points in the domain. If our

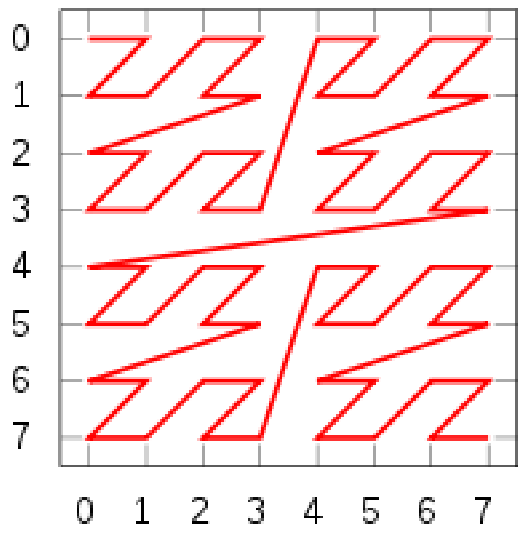

Figure 3: Morton order

integer representations of x and y for a particular point are 10001 and 01110, respectively, then with interleaving we get the Morton key of 0110101001. Bit *i* of the x integer and bit *i* of the y integer form pair *i* in the Morton key. If x and y integers are both 32-bit, then its Morton key will be 64-bit. You can get a clue of how to interleavely construct a Morton key from the given `printKey` function in class `MortonKeyCalculator`. Actually there is a smart way to compare the Morton keys without explicitly generating them, think carefully! Please keep in mind that y coordinate is more significant. We also provide the `cellWidth` function in `MortonKeyCalculator` class to help calculate the maximum value of width and length for a given level's sub-squares.

After sorting the Morton keys, the points (quadtree leaves) are in the order they would appear in the quadtree.

Deliverables

Using the ideas above, complete the implementation of the MortonKeyCalculator class in the files **MortonKeyCalculator.cc** and **MortonKeyCalculator.hh**. You should make your implementation parallel. Thankfully, gcc makes this easy with its builtin parallel mode. You should use the method of "Using Specific Parallel Components." The main file **mortonsort.cc** may be useful for debugging, which can be compiled by `make mortonsort` and run independently.

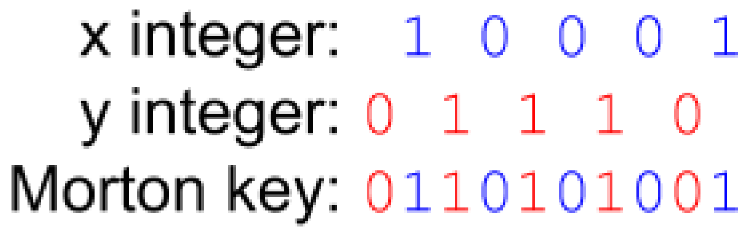

Figure 4: Bit interleaving

Task 4: Merging Quadtrees

As explained above, once the bodies are sorted, the sequence can broken down into consecutive chunks, and each chunk assigned to its own core. If your serial implementation allocates nodes for the quadtree as they are needed, you may want to consider pre-allocating them for your parallel implementation. Because the address space is shared among the several worker threads, memory allocation can be a bottleneck. Regardless of your choice, the sorting by Morton keys will ensure that the resulting trees will have small overlap and can be merged quickly.

Deliverables

Employ this strategy in an updated implementation of the **ForceBarnesHut** class in **ForceBarnesHut.hh** and **ForceBarnesHut.cc** files. You can use `"_OPENMP"` macro to separate OpenMP parallel codes with sequential codes you implement for Task 2. If you wish to use the **MortonKeyCalculator** you implement previously, you will have to include its source (both classes and codes) in these files.

Summary

After successfully completing this lab, you will have solid code base that will allow you to quickly approximate the net forces exerted by n bodies on an arbitrary mass in space. This will allow you to quickly solve the n-body problem for every large instances, and your implementation will be parallel-friendly allowing it benefit where multiple processors are available.

Benchmarking

Performance Testing

You should measure the performance of your code on the Deepthought cluster. The file **benchmark.cc** performs the tests and compile it with

```
make benchmark
```

The file **test.sub** serves as an example for how to use sbatch. Thus, to measure the performance of you code you should run

```
sbatch test.sub
```

from your Deepthought login node.

Output format

The benchmark executable outputs timing results in JSON format (`marks.json`). Each of the field names are described below.

```
"nbodies": input parameter, number of gravitational bodies in the simulation.
"nqueries": input parameter, the number of single-body force computations.
"theta": input parameter, the multipole acceptance criterion.
"seed": input parameter, random seed.
"max_nthreads": should be set to 32 for benchmarking purposes.
"time_setup": The time in seconds required to sort the Morton keys and
              construct the quadtree.
"time_queries": The time in seconds required to perform 'nqueries' computations.
```

Expected Timings

Since reference sequential Barnes-Hut implementation were not provided, only target parallel timings will be provided. You can test the naive implementation to check out the benefit of Barnes-Hut algorithm. The following timings were obtained with “nqueries” >= 10000, “theta” = 0.1, “seed” = 5, “max_nthreads” = 32.

```
time_setup:
  "nbodies" = 1e6: ~ 0.15 seconds
  "nbodies" = 1e7: ~ 1.75 seconds

time_queries:
  "nbodies" = 1e6: ~ 50 microseconds / query
  "nbodies" = 1e7: ~ 80 microseconds / query
```

Submitting Your Code

Once you have completed and tested your implementations, please submit using the **submit.py** script,

```
python submit.py
```

which will do a quick correctness test. You may submit as many times as you like before the deadline. At the deadline, the TA will download the code and perform some timing runs. These results along with a manual inspection of the code will determine your grade.