CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Homework 4
Due Friday March 13th, 4:30 P.M. P.S.T.

1 Overview

1.1 Goals

In class and homeworks, we have been exploring the utility of standardized container interfaces and generic algorithms. You have seen the value of iterators for navigating your graph. However so far all your work has been done sequentially. Most computations today are done in parallel. In this homework you will extend some standard `thrust` iterators to leverage **parallelism** in a simple manner. You will also implement a more efficient way to navigate through your graph to implement collision detection in `mass_spring`.

Concepts Introduced Parallelism, practice with fancy iterators.

1.2 Tasks

- Update the `makefile`. (short)
- Create a random access iterator for the `Node` class. (short)

- Update `mass_spring` to use `thrust`. (short)
- Implement the bit operators in `MortonCoder.hpp`. (short)
- Complete the `SpaceSearcher` class. (medium)
- Use the `SpaceSearcher` and parallelism to implement the `SelfCollisionConstraint`. (time intensive)

1.3 Tips

- There is a lot of started code for this assignment. If you understand what it does the actual amount of code you have to write is fairly limited.
- Look at your EX3 before implementing the iterators. A lot of the same material has been covered there.
- Thrust has great documentation and tutorials, use these to understand how to use their functions.
- We have provided you several test functions, use them to debug your code.
- In this assignment you will build on top of existing libraries so make sure you understand them first.
- Start early!
- Make sure you understand the starter code provided before you write your own code.
- You can always come to the teaching team with questions.
- Any error you encounter someone else will have had before. Stackoverflow is your best friend.
- Don't forget to pull the starter code.
- We advise committing and pushing frequently as you make progress on the assignment.
- Work together.

1.4 Starter code

- `test_omp.cpp` & `test_omp2.cpp` Used to check the successful installation and run of parallelism.
- `test_spacesearcher.cpp` Used to test your `SpaceSearcher` class.
- `MortonCoder.hpp` Contains the MortonCoder class used in SpaceSearcher. Here you implement the bit operations. Everything else has been implemented.
- `SpaceSearcher.hpp` Contains the starter code for the SpaceSearcher. A lot of the necessary functions have already been implemented.

2 Setup

Our helper code for Homework 4 extends the code from Homework 3. To retrieve it, follow these steps:

```
# Check the status of your git repository
```

```
$ git status  # Should say "on branch master"

# Otherwise, save a HW3 branch and checkout the master branch
$ git branch hw3
$ git checkout master

# Should also show no changes (all commits should already be made for HW3)
# Otherwise, commit your changes.
$ git commit -am "Stray HW3 changes"

# Load our changes into your repository
$ git fetch CME212

# Apply our changes to your repository
$ git merge CME212/master
```

Then follow git's instructions. For example, if there are conflicts, fix them and commit the result with `git commit -a`. If the merge produces too many conflicts, try `git rebase CME212/master` instead.

2.1 Update `makefile` for Thrust and OpenMP

We'll be parallelizing some of our work using OpenMP and Thrust. In order to do so, we need to use a compiler with OpenMP support (Clang 3.5 does not, but Clang 3.7 does!). First, modify the `Makefile` to use `g++` (4.7) with OpenMP:

```
CXX := g++ -fopenmp
```

You may find that the error reporting with Clang is easier to read, but we'll need the above to compile some of the code in this homework.

Additionally, we add the following flag to force Thrust to use OpenMP as it's "device system":

```
CXXFLAGS += -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_OMP
```

To install Thrust, clone the repo:

```
$ git clone git@github.com:thrust/thrust.git
```

and add the path to your `Makefile`.

3 Parallel mass-spring state update

Objectives

- Implement random access iterator for `node_iterator`.

- Parallelize the mass spring system.

3.1 Random access node iterator

If you have a collection of objects you need to operate on there is no need to go over them one-by-one unless the value of one object influences the next. For example when you update the positions of all the nodes in `mass_spring` you only need the velocity vector of each node. You do not need any other information from the graph. So in principle you could update all the node positions simultaneously. This idea is known as Divide and Conquer. The current node iterator in your Graph class has to start at the first node and keep incrementing until you find the node you are looking for, this is know as **forward iteration**. It would be much more efficient to be able to access an object in constant time regardless of its position in the container. This is known as random access. **Your task** is to implement a random access node iterator for your graph class.

Recall: Fancy Iterators from Thrust Modify your `Graph.hpp` such that `node_begin()` returns a random access iterator over nodes. Note that `node_end()` must be updated to return a corresponding iterator to the end of the range. One simple way to achieve this is to use "fancy iterators" from Thrust as demonstrated in EX3. You may find `thrust::transform_iterator` and `thrust::counting_iterator` particularly useful.

Documentation and examples can be found here:

`http://thrust.github.io/doc/group__fancyiterator.html`

and the source code itself:

`https://github.com/thrust/thrust/tree/master/thrust/iterator`

Protip: The constructor of a `thrust::transform_iterator` likely differs from the constructor of your old `node_iterator`. If you are using the `using` style in EX3, you would have to go through the code and change each instance of constructing a `node_iterator`... lame! Instead, consider inheriting from `thrust::transform_iterator` and providing your old constructor interface that simply forwards to the new interface.

```
struct NodeIterator : thrust::transform_iterator<___, ___, Node> {
  using super_t = thrust::transform_iterator<...>;
  NodeIterator() {} // Default (invalid) constructor.
  NodeIterator(const super_t& ti) : super_t{ti} {} // KLUDGE Conversion constructor.
  private:
    friend class Graph;
    NodeIterator(..., const graph_type* g) : super_t{...} {}
};
using node_iterator = NodeIterator;
```

3.2 Parallel integrator

Embarassingly Parallel Operations can be Parallelized Trivially Each step of the symplectic Euler method from HW2 involves two `for` loops over nodes. The first loop updates the node position and the second loop updates the node velocity. Updating the state of a single node is independent of the state of all other nodes; i.e. the process is embarassingly parallel. We could execute updates in arbitrary order or in parallel.

Your task is to replace the symplectic Euler `for` loops in `mass_spring.cpp` with appropriate calls to `thrust::for_each`, which will *enable* operation on the nodes in parallel.

Enforcing Multi-Core Execution Once the `for_each` loops are constructed, it is trivial to then enforce parallel execution by specifying an OpenMP execution policy. Specifically, by passing `thrust::omp::par` as the first parameter to any Thrust algorithm, that algorithm will be parallelized with OpenMP threads, *if possible*.

Parallel code $\nrightarrow$ faster code This does not mean that your for loops will *necessarily* execute faster. There is *overhead* in launching OpenMP threads, synchronizing them, and even perhaps additional work to be performed in a generic parallel algorithm as compared to a generic serial algorithm. In this case, without enough *work* (or a large enough problem size) to be performed within the loop, this overhead can dominate. Note that you may have to modify the VirtualBox settings to give the virtual machine access to more than one CPU core. In the README, document the runtime of the serial version of the code against the parallel version for various grid sizes (we've provided an additional `grid4` with one million nodes).

Note: Run your code on `rice.stanford.edu` for the timing as it has multiple core CPU's. Some laptops only have a single core so won't be able to run in parallel.

To execute `mass_spring` on all grid sizes, the following constants result in stable simulations.

```
mass = 1.0 / graph.num_nodes();   // Node mass
k = 100.0;                          // Spring constant
dt = double dt = 0.001;             // Time step
```

4 Location Optimization

Objectives

- Implement the bit operations in `MortonCoder.hpp`
- Complete the `SpaceSearcher` class.
- Implement `SelfCollisionConstraint`.
- Speed up `SelfCollisionConstraint`

Overview

A lot of the constraints we saw in the mass-spring system in HW2 only act on a small number of nodes in a defined region. However to check the constraints we still iterated over all the nodes individually. It Would be easier if we could check the nodes closest to the region the constraint acts upon first and then move to the nodes further away. This way as soon as we see that the nodes we no longer satisfy the constraint condition we can stop iterating knowing that any nodes we have not seen are further away from the constraint region than the current node.

For example: for the table constraint it would be easy if we could iterate over the nodes in order of z-coordinate. We start of at the lowest and as soon as the nodes stop touching the table we can stop iterating.

The first part of this implementation is to define a `BoundingBox` for each constraint. This defines the region on which the constraint acts, such as the z-coordinate for the table constraint.

The second part is to find a way to encode the nodes such that nodes close to each other in space have similar codes. This is done with the `MortonCoder`. Even though nodes may be far apart in the Graph if they are close in spcae they will have close Morton codes. This will be very helpful when trying to do collision detection as nodes close to the node you want to avoid will have similar Morton codes. So we can start iterating at the nodes with similar Morton codes and stop once the current node in the iteration no longer collides, knowning that all the unseen nodes are further away.

This design point motivates why we have the constraints operate on `Graph` rather than `Node` like a force does. One difference between constraints and forces in our model is that constraints are often localized in space, while forces are usually applied to every node.

Space-Searcher Let's build a class called `SpaceSearcher`:

```
template <typename T>
class SpaceSearcher
    SpaceSearcher(Box3D, TIter, TIter, PointIter, PointIter);
    SpaceSearcher(Box3D, TIter, TIter, T2Point);

    class neighboorhood_iterator;
    neighboorhood_iterator begin(const Box3D&) const;
    neighboorhood_iterator end(const Box3D&) const;
```

which is constructed on (abstractly?) two ranges: one range of values, and another range of `Points` which correspond to the location of each value. This class constructs some data structure and provides an iterator over the values whose associated `Point` fell within a given bounding box. Thus, we can associate data to points and then iterate over only the data with points contained within a box. If the iteration can be performed in sub-linear time, then each constraint application could be significantly faster. This would be super useful in the future for other localized operations as well!

4.1 Morton Coding

In order to accomplish this, we have provided you with the majority of a class that partitions space and can be used to order points using a *space filling curve*. In this case, we use the Z-order curve, or Morton coding, which has a number of interesting properties.

Morton coding divides a bounding region – a D-dimensional rectangular volume – into L levels and 2^{LD} smaller identically-sized volumes, called *cells*. There are 2^L cells on each side of the volume. The cells are numbered from 0 to $2^{LD} - 1$. But the numbering isn't row-major or column-major: it is `fractal`. In this ordering, cells that are close to one another in space often have codes that are close to one another in Morton order. For example, here is a 2D Morton coding with $L = 2$.

(x,y)	x=0 (00)	x=1 (01)	x=2 (10)	x=3 (11)
y=0 (00)	0 (0000)	1 (0001)	4 (0100)	5 (0101)
y=1 (01)	2 (0010)	3 (0011)	6 (0110)	7 (0111)
y=2 (10)	8 (1000)	9 (1001)	12 (1100)	13 (1101)
y=3 (11)	10 (1010)	11 (1011)	14 (1110)	15 (1111)

Notice that the code can be computed by interleaving the bits of the box tuple (x,y).

To compute the Morton code we need to do bit operations on the coordinates of a node.

First, check out our `MortonCoder` class. Then complete the definitions of `interleave` and `deinterleave` in `MortonCoder.hpp`. You may find some bit-twiddling hacks to be helpful.

4.2 Constructing a `SpaceSearcher`

Now, `SpaceSearcher` can use `MortonCoder` to transform `Points` into Morton codes and order data by their position along the space-filling curve. First, we need this data structure.

A `SpaceSearcher` is represented by a vector of pairs of Morton codes and data values:

```
struct morton_pair {
  code_type code_;
  T value_;
  // Cast operator so we can treat a morton_pair like a code_type
  operator const code_type&() const { return code_; }
};
std::vector<morton_pair> z_data_;
```

and the vector is sorted by the Morton codes. This allows us to iterate over the points in order of proximity in space. The cast operator let's us use the `morton_pair` as a Morton code and avoid writing trivial lambda functions everywhere to extract the `code_` member. Typically, cast operators are dangerous and awkward, but here it cleans up the code nicely (see each use of `std::lower_bound`, for example).

The representation invariant is

$$\forall\, i, j \text{ with } 0 \le i < j < \texttt{z\_data\_.size()},\quad \texttt{z\_data\_}[i]\texttt{.code\_} \le \texttt{z\_data\_}[j]\texttt{.code\_}$$

or, even more nicely,

```
std::is_sorted(z_data_.begin(), z_data_.end(),
               [](morton_pair a, morton_pair b) { return a.code_ < b.code_; })
```

For convenience, we provide two constructors for `SpaceSearcher`, which are left to you to implement. You may consider using delegating constructors or a helper function.

Tips on parallelizing your code In both cases, we want to consider parallelizing the construction. In order to do so, it would be best to perform the construction in two stages:

1. Use the range constructor of `std::vector` to initialize the data.
2. Use (optionally) parallel algorithms to sort the data.

To use the range constructor (i.e. `z_data_ = std::vector<morton_pair>(first, last)`), you'll want to write a constructor for `morton_pair` that accepts a `thrust::tuple<code_type,T>` and consider using `thrust::zip_iterator`, `thrust::transform_iterator`, and any other "fancy iterators" that you may find useful.

4.3 Efficient neighborhood search

Since your constraints operate on a `Graph`, they can now construct (or share a reference to) a `SpaceSearcher` and ask for only `Node`s that are in or near a bounding box that encapsulates the constraint's influence. Thus, it can cut down on the number of `Node`s it needs to check!

Here's a cute and easy implementation of a `SelfCollisionConstraint`, which prevents the `mass_spring` model from passing through itself. It's expensive! It requires $\mathcal{O}(\texttt{num\_nodes()}^2)$ work since it iterates over all pairs of `Node`s.

```
struct SelfCollisionConstraint {
  void operator()(GraphType& g, double) const {
    for (Node n : nodes(g)) {
      const Point& center = n.position();

      double radius2 = std::numeric_limits<double>::max();
      for (Edge e : edges(n))  // std::accumulate?
        radius2 = std::min(radius2, normSq(e.node2().position() - center));
      radius2 *= 0.9;

      for (Node n2 : nodes(g)) {
        Point r = center - n2.position();
        double l2 = normSq(r);
        if (n != n2 && l2 < radius2) {
          // Remove our velocity component in r
          n.value().vel -= (dot(r, n.value().vel) / l2) * r;
        }
      }
    }
  }
};
```

Your task is to use the `SpaceSearcher` and/or parallelism to accelerate this constraint.

Submission Instructions

Since you made `staff-cme212` "collaborators" on your git repository, we will examine your work in that repository.

Use a git *tag* to mark the version of code you want to submit. Here's how:

```
$ git add <files to track>                      Tells git which files to track
$ git status                                    View files git is tracking
$ git commit -am "Describe last few edits"      Commit files
$ git tag -am "My HW4" hw4                      Tag this commit with tag ‘‘hw4’’
$ git push --tags                               Push all commits to git repo
```

It is possible to change your tag later, too, if you discover a bug after submitting:

```
$ git tag -d hw4
$ git tag -am "My HW4" hw4
$ git push --tags
```

We will use Git timestamps on tags and the associated commits to check deadlines.

Be careful with overwriting tags – you don't want to lose the submission tag. We can handle versioned tags, such as `hw4_1`, `hw4_2`, etc.

To verify that all of the files were pushed correctly, you can click on your repository to the right of your `code.seas` account and view the "Source Tree". There may be a delay between your push and the files showing up in the browser.