The problem

In this assignment we are going to simulate THE ENTIRE UNIVERSE (dun, dun, DUN!) Well, a rectangular 2D universe made up of ~3000 circles anyway. Close enough, I guess.

So we have a few thousand objects flying around in a rectangle and are interested in their gravitational interaction according to Newton's famous law. (Well, a slightly modified version of it. Let's call it Philipp's Law.)

Sadly, those pesky physicists tell me that we have to calculate the gravitational force between each pair of objects. A quadratic algorithm for a quadratic universe?! I don't think so.

Instead we are going to exploit that a group of objects which is far enough away can be approximated by a single large object. So we do not have to calculate the force for each member of the group, but only once.

Hence we have to find an efficient scheme to determine which objects are far enough away from each other and then put them into groups. For that we are going to use a hierarchical domain decomposition using quadtrees.

Quadtrees

The concept of a quadtree is based on refinement. At the coarsest level we have a single square cell covering the whole domain. Refining it once, we end up with four (square) cells of the same size, each covering a quarter of the domain. Then we could recursively refine any number of them. The key is that we can refine 'interesting' regions (usually ones with more objects in them) more often, and get a better resolution / have each cell contain a similar number of objects.

Usually, one would adapt the decomposition after some number of steps, to adjust to objects that have moved. Instead, we are not going to do that. (For the assignment we are only doing 100 steps, so there is not a lot of movement. If you run the GUI you will find that performance deteriorates after a while, as objects move out of their original cells and cluster somewhere else.)

While we do not adapt the domain decomposition, we still have to manage the positions of objects. Some objects which are close to their cell's boundary will leave that cell and move to a neighbouring one. That means that the code has to do some bookkeeping, so that each object actually resides in the cell that contains it.

As the name implies, the whole quadtree is a tree data structure. The root of the tree is the single cell covering the whole domain. Each cell which has been refined will have four children, corresponding to the four cells it has been refined into. Cells which are not refined further are called leaves, and they contain the actual objects.

To make things simpler for you, there are always going to be 31 leaves, exactly one per process. (Why not 32?) This is of course inefficient, usually one would want to split up the domain on a much finer level and have each process manage a number of cells.

The algorithm

The algorithm is going to be the same for the parallel and sequential versions. (So it makes sense for you to take the sequental implementation as a baseline.)

We are going to call the set of objects contained in a node its 'high-resolution data'. If we compute the average position and total mass of those objects we get the 'low-resolution data'. So the low-resolution data for a node is basically just a single object representing all objects inside that node.

1. For each leaf node a
    1. For each leaf node b
    2. If a is close to b
        1. Calculate force between all pairs of objects in a and b
    3. If a is far away from b
        1. Calculate force between all objects in a and the low-resolution data of b
    4. Update the velocities of the objects in a
    5. Move the objects in a, keep track of the ones that have left
2. Put the objects that have left their node into the correct node

Compile and run

Compile the code using `make`. There will be the usual executables in the `student` directory, `nbody_par`, `nbody_seq` and `unit_test`. Run them in the usual fashion:

```
./student/nbody_seq
mpirun -np 31 ./student/nbody_par
mpirun -np 31 ./student/unit_test
```

You can adjust the parameters. If you run the code with `-h` it will list the available options and default values:

```
philipp@phllnx:~/a10$ ./student/nbody_par -h
Usage:
  ./student/nbody_par [-p <num_points>] [-l <num_leaves>]
          [-s <num_steps>] [-f <file>] [-r <file_ref>] [-g]
  -p <num_points> [default: 100]
      Initialise with (approximately) this number of points
      per leaf node.
  -l <num_leaves> [default: 31]
      Generate this many leaf nodes. Must be 1+3*n for some
```

```
    n. Only used for sequential execution!
-t <timestep> [default: 0.01f]
    How much time a single step of the simulation simulates.
-s <num_steps> [default: 20]
    How many steps of the simulation to perform.
-f <file> [default: output_par.svg]
    Name of the output SVG file.
-r <file_ref> [default: output_unit_ref.svg]
    Name of the output SVG file for the reference code.
-g [default: off]
    Whether to show the animation in a window as it runs.
```

The most interesting one is `-g` to enable the gui. You can try running `./student/nbody_seq -g` to see the simulation happening in realtime. For the parallel version only the main process will actually draw its data, so you will probably see only a fraction of the whole domain. Still, it could help with debugging. (Mostly it looks nice.)

Code structure

As usual, you only need to modify `student/nbody_par.c`, which is the file that is submitted to the server. There are the following files:

- `./nbody.h`: The header file for the data structures and all kinds of utility functions.
- `./student/nbody_par.c`: The file you need to modify
- `./nbody_seq.c`: A sequential implementation of the algorithm, which you can (and should) use as a starting point
- `./main.c`: Initialisation, argument handling, glueing everything together.
- `./nbody.c`: Implementing the data structures, initialisation, and the utility routines. You should refer to the source in here if you want to know how a certain function was implemented.
- `./Makefile`: Build the project.
- `./gui.h`, `./gui.c`: The GUI. This should not be relevant to the assignment, as the GUI is initialised from `main.c` and the frames are drawn from within `nbody.c` without your involvement. Additionally, the server compiles without GUI anyway. Still, if you ever feel the need to open a window under Linux to draw an animation, this code is the bare minimum necessary to do just that.
- `./output_*.svg`: These files will be generated when you run the simulation. They show the position of each object as well as the low-resolution information for each node. You can look at them to debug correctness issues in your code.

Your task

You have to (surprise!) parallelise the code using MPI and get a high enough speedup. It should be quite obvious how you distribute the domain across your processes. Then, you have to think about which data each process need to do its computation, and what communications need to be performed.

There are no restrictions on the kind of MPI functions you are allowed to use. In particular, you are not required to use any collective operations. (They may come in handy, though.)

You do not have to perform further optimisations regarding the algorithm, a straightforward parallel implementation should be enough to get the required speedup. Of course, you are free to further optimisations, as long as your submitted code is actually parallelised using MPI.

It is not *required* that you call any of the helper functions in `nbody.c`. But it is sensible. You might need to write your own helper functions to execute specific tasks on the tree data structure.

Good Luck!