CS380P: Parallel Systems

Lab #5: MPI Barnes-hut

In this assignment, you will solve an N-body problem using the Barnes-Hut algorithm. You will be forced to deal with an irregular data structure, i.e. something other than an array, and you will have to deal with dynamic changing data as the bodies move during the simulation. We encourage you to think carefully about your data structure, because your choice will be central to your solution.

All your implementations (source code and Makefiles) should be organized in the `~/workspace` directory, which should be the same directory where you see this guide.

You will find all inputs in the `input` directory and references in the `reference` directory.

MPICH is already installed in the project box. If you are using your own machine, you can download source from: https://www.mpich.org/ and install it with the following commands:

```
./configure; make; sudo make install
```

1. Background

The particular N-body problem that you will solve will be an astrophysical simulation that models the continuous interaction of each body with every other body. In this case the interactions are caused by the gravitational force that each body exerts on every other body, where each body could represent a star, a planet, or a galaxy. As your simulation proceeds, the forces will cause the bodies to move, so this is a dynamic computation that requires an irregular data structure. [1] has a very good introduction about astrophysical N-body simulations.

1.1 Barnes-Hut Algorithm

There are several algorithms for solving the N-Body problem. The simplest solution directly computes the n*n interactions that exist between every pair of bodies. The complexity of this approach is O(n*n), where n is the number of bodies in the system. You will instead use a superior solution, the Barnes-Hut method, which takes O(n log n) time; this solution computes direct interactions for pairs of bodies that are sufficiently close; for the effects of far away bodies, the algorithm approximates a group of bodies by a single body whose center of mass is the same as the center of mass of the group of bodies.

Read section 4.1.1 of [2] for a basic introduction of the Barnes-Hut algorithm

The algorithm consists of two phases. The first phase constructs a tree that represents a spatial partitioning of the bodies in the system. Each interior node in the tree represents the center of mass of the bodies in the subtree rooted at that node. The second phase computes the force interactions for each body in the system by using the MAC (multipole acceptance criteria), which determines whether the bodies in a subtree are sufficiently far away that their net force on a given body can be approximated by the center of mass of the bodies in a subtree. The velocity and position of the body is then calculated via a Leapfrog-Verlet integrator. Since we are approximating the effect of group of bodies by their center of mass, this algorithm takes O(n log n) steps.

1.1.1 MAC (Multipole Acceptance Criteria)

As stated in [2], whether a body is far away from a subtree depends on the distance d between the body and the center of mass of the subtree, the length l of the side of the subtree, and a threshold theta. You should be able to compute d and l and theta is a command line argument to your program.

1.1.2 Force

The gravitational force between two bodies (or one body and a subtree) are given in [1] as

```
F = G*M0*M1*d / d^3
```

where M0 and M1 are the mass of the two bodies (or one body and a subtree) and d is the distance between the two bodies (or one body and the center of mass of a subtree). However, when two bodies are too close, the force computed using the above formula can be infinitely large. To avoid infinity values, set a rlimit value such that if the distance between two bodies are shorter than rlimit, then rlimit is used as the distance between the two bodies. Note that the force computed here is a vector, which means forces from different bodies (or subtrees) cannot be added directly. What you need is to project the force on x and y axis and then the projected forces on each axis are addable. The projected force on each axis can be computed as

```
Fx = G*M0*M1*dx / d^3
Fy = G*M0*M1*dy / d^3
```

where dx and dx is the distance between the two bodies (or one body and the center of mass of a subtree) on the x and y axis.

1.1.3 Leapfrog-Verlet Integration

When you have the total force on a body you can compute the new position (Px', Py') and velocity (Vx', Vy') of the body given its current position (Px, Py) and velocity (Vx, Vy) as follows

```
ax = Fx/M0
ay = Fy/M0
Px' = Px + Vx*dt + 0.5*ax*dt^2
Py' = Py + Vy*dt + 0.5*ay*dt^2
Vx' = Vx + ax*dt
Vy' = Vy + ay*dt
```

where dt is the timestep.

1.1.4 Parameters

You might use the following values for some of the parameters.

- `G`: 0.0001

- `rlimit`: 0.03
- `dt`: 0.005

2. Submission Guide

2.1 Input/Output

2.1.1 Flags

Your program should accept the following four command line parameters, with types indicated in parentheses:

- `-i inputfilename` (char *): input file name
- `-o outputfilename` (char *): output file name
- `-s steps` (int): number of iterations
- `-t \theta`(double): threshold for MAC
- `-d dt`(double): timestep
- `-V`: (OPTIONAL, see below) flag to turn on visualization window

Both the input and output files will have the following format:

- The number of bodies (n)
- A list of n tuples:
  - index
  - x position
  - y position
  - mass
  - x velocity
  - y velocity

You may assume that all initial velocities are 0 in the input file.

Note that the index of each particle is considered as the particle's name. Therefore, in the output file, the order of particles does not matter since each particle will be tracked by its index. You should keep the index of each particle safely with the particle because the autograder will check the correctness of the output file based on the index of each particle.

You should prepare a `Makefile` so that your program can be compiled by invoking `make` without any arguments. Name the executable `nbody`, which is the name used by the autograder.

2.1.2 Performance

Your program should also output the elapsed time for the simulation in the following form

```
xxxxxx
```

Do not print out anything else (such as the unit). Make sure that the unit of the output time is second because it is assumed that you are using `MPI_Wtime()` to measure the time. You can also use other methods, but make sure to convert the result to seconds.

2.2 Domain Boundary

When the position of a particle is out of the boundary, the particle is considered to be lost. For lost particles, you should set the particle's mass to -1 and do not include the particle in the force calculation any more. Therefore, the autograder can understand that the particle is lost by checking its mass.

The positions and masses of particles in the input are randomly generated. The x and y coordinates of each particle are within the range (0, 4). Therefore, you should set the domain as the square (0,0) -- (4,4).

3. Visualization

You can use openGL to visualize the movement of the bodies in the domain. The required packages and virtual desktop are setup in the project. If you are using your own machine, here is a list of packages and instructions about installation (tested on Ubuntu 18.04):

NOTE: The content in this section is not strictly required. Our own experience is that visualization is critically important to debugging, but ultimately, the autograder does not perform any checks of correctness of your visualization. If you do implement a visualization, you should add an additional -V flag to your program that when provided, turns on the visualization, but defaults to not showing the visualization.

- sudo apt install
  `cmake make g++ libx11-dev libxi-dev libgl1-mesa-dev libglu1-mesa-dev libxrandr-dev libxext-dev libxcursor-dev libxinerama-dev libxi-dev libglew-dev libglm-dev freeglut3-dev`
- `GLFW 3.3` download source from: https://www.glfw.org/
  install: `mkdir build; cd build; cmake ../; make; sudo make install`

Note that all visualization code should be executed on the main processor.

3.1 Headers

Include the following headers

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

```
#include <glm/gtc/type_ptr.hpp>
#include <GL/glu.h>
#include <GL/glut.h>
```

3.2 Initialization

Before going to the main loop, you need to initialize the packages

```
/* OpenGL window dims */
int width=600, height=600;
GLFWwindow* window;
if( !glfwInit() ){
    fprintf( stderr, "Failed to initialize GLFW\n" );
    return -1;
 }
// Open a window and create its OpenGL context
window = glfwCreateWindow( width, height, "Simulation", NULL, NULL);
if( window == NULL ){
    fprintf( stderr, "Failed to open GLFW window.\n" );
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window); // Initialize GLEW
if (glewInit() != GLEW_OK) {
    fprintf(stderr, "Failed to initialize GLEW\n");
    return -1;
}
// Ensure we can capture the escape key being pressed below
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
```

3.3 Coordinate Transformation

Before drawing anything in the window, you should know that you are restricted to draw anything whose coordinate is within [-1, 1]. However, the coordinates of particles and tree size can be out of this range. The simplest solution is to map all particles to the range (-1, -1) -- (1, 1) linearly. Here is an example.

Say you have a particle whose position is (x, y). x and y are within the range (0, maxX) and (0, maxY). The new coordinates of the particle when you draw it in the window should be

```
x_window = 2*x/maxX -1;
y_window = 2*y/maxY -1;
```

In the rest of this section, all coordinates are mapped coordinates in the window.

3.4 Draw Bodies and Tree

At the end or beginning of each iteration, use the following code to refresh the window and draw bodies and tree.

```
glClear( GL_COLOR_BUFFER_BIT );
drawOctreeBounds2D(node);
for(p = 0; p < _nparticles; p++)
    drawParticle2D(x_window[p], y_window[p],radius[p],colors[p]);
// Swap buffers
glfwSwapBuffers(window);
glfwPollEvents();
```

`drawOctreeBounds2D(node)` draws horizontal and vertical lines in the window according to the partition info given by `node`. Here is a sample code

```
void drawOctreeBounds2D(node) {
    int i;
    if(!node || node is external)
        return;
    glBegin(GL_LINES);
    // set the color of lines to be white
    glColor3f(1.0f, 1.0f, 1.0f);
    // specify the start point's coordinates
    glVertex2f(h_start_x, h_start_y);
    // specify the end point's coordinates
    glVertex2f(h_end_x, h_end_y);
    // do the same for verticle line
    glVertex2f(v_start_x, h_end_y);
    glVertex2f(v_end_x, v_end_y);
    glEnd();
    for each subtree of node
        drawOctreeBounds2D(subtree);
}
```

The data structure used for `node` should contain the information about the size of this `node` and how this node is partitioned. If this `node` is further divided, you should obtain the coordinates of the dividing segments from the `node` data structure.

`drawParticle2D(x_window[p], y_window[p],radius[p],color[p])` draws a circle in the window. The circle is filled with color determined by `color[p]`, which contains three float values between 0 and 1 indicating the strength of red, green, and blue. The origin of the circle is determine by the position of the body, whose coordinates are `(x_window[p], y_window[p])`. The radius of the circle is determined by `radius[p]`. Here is a sample code for drawing a 2D particle.

```
void
drawParticle2D(double x_window, double y_window,
           double radius,
           float *colors) {
    int k = 0;
    float angle = 0.0f;
    glBegin(GL_TRIANGLE_FAN);
    glColor3f(colors[0], colors[1], colors[2]);
    glVertex2f(x_coord, y_coord);
    for(k=0;k<20;k++){
```

```
        angle=(float) (k)/19*2*3.141592;
        glVertex2f(x_window+radius*cos(angle),
           y_window+radius*sin(angle));
    }
    glEnd();
}
```

4 Optimizations

Optimizing n-body computations is a well-explored field, and we encourage you to explore ideas. No additional optimizations are explicitly required, but if you do implement some, the effort will be rewarded with extra credit. This is the case even if your optimization attempts do not yield performance improvements, as long as your writeup explains the optimization, why you think it should help, and why it other does or does not help in the environment where you do your measurements.

5. Writeup

You should include a report that explains any important details of your approach, additional optimizations you implemented or attempted to implement, and includes some performance measurements and analysis.

- Unlike previous labs for this course, we have intentionally left requirements for performance analysis very open-ended. There are two reasons. First, at this point in the course, the metrics of interest and methodologies for obtaining them should hopefully be clear. Second, a large number of factors can impact the range of inputs over which performance artifacts are interesting and we want to leave freedom for you to adjust to them. Concretely, if the only environment in which you are able to obtain performance measurements is Codio, the achievable scalability will be very different from what you would observe if you have access to a large multi-core machine or multiple machines. The simplest way to directly satisfy this requirement is to generate a graph that shows performance as a function of data size using the three sample inputs by fixing the number of steps (-s parameter) and the number of nodes (-n parameter to mpirun). A more thorough approach would explore other data sizes and parameters such as the number of MPI nodes, and is encouraged but not required.
- Report the hardware details and OS version on the machine where you did your measurements if you measured in any environment other than Codio; if you just used Codio for measurements, it's fine to just report that fact.
- How much time did you spend on the lab?

6. References

[1] M. S. Warren and J. K. Salmon. "Astrophysical N-body simulations using hierarchical tree data structures," In Proceedings of the 1992 ACM/IEEE conference on Supercomputing (Supercomputing '92), Robert Werner (Ed.). IEEE Computer Society Press, Los Alamitos, CA, USA, 570-576.

[2] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, "Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity," Journal of Parallel and Distributed Computing, Volume 27, Issue 2, 1995, Pages 118-141.

[3] M. S. Warren and J. K. Salmon, "A parallel hashed Oct-Tree N-body algorithm," In Proceedings of the 1993 ACM/IEEE conference on Supercomputing (Supercomputing '93). ACM, New York, NY, USA, 12-21.

[4] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh, "Scalable parallel formulations of the barnes-hut method for N-body simulations," In Proceedings of the 1994 ACM/IEEE conference on Supercomputing (Supercomputing '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 439-448.