CS 422 Parallel Project – Free Body Problem

Eric Evans

Spring 2017

**Introduction**

The Free Body Problem involves two fundamental kinematic scenarios: gravitational attraction and elastic collisions. Bodies, point masses with a defined radius, are placed in a two-dimensional plane. The program solution, written in Java 1.8, divides the computation between 1 to 32 threads. This report explains the details and empirical findings of this program. An interactive graphical user interface is paired with the program to display the free bodies in motion.

**Programs**

*To Compile*: javac Body.java FreeBodies.java FreeBodyGUI.java SpringUtilities.java Universe.java Worker.java

*To Run:*

* java FreeBodies <number of threads> <number of bodies> <size of body (mass)> <number of steps>
  + This runs a non-GUI version with bodies indicated by command line arguments.
* java FreeBodies <number of threads> <number of bodies> <size of body (mass)> <number of steps> --gui
  + This runs a GUI version with initial bodies indicated by command line arguments.
* java FreeBodies --gui
  + This runs a GUI version with no initial bodies.

A sequential version was used in the initial stages of the program’s development. To run this version, use “1” as the command line argument indicating the number of workers. The initial positions of the bodies (by command line) are randomly placed in unoccupied locations in a field 10000x10000 for the non-GUI version and 600x750 for the GUI version. All initial bodies have a radius of 20.

Calculating gravitation forces and moving each body is done by multi-threading. This was necessary for gravitational calculations as sequentially it is O(n2). Moving each body benefits from multiple threads as it is computationally expensive. Splitting up the responsibility of the computation was done by striping and a dissemination barrier was used. Calculating collisions is unlike the other two processes. Not nearly as many collision calculations occur in the program. About 40000 collisions happen in a simulation of 320 bodies over 70000 iterations. By optimizing the intersection detection method Body.doIntersect() (see Other Experiments and Extensions), this could reasonably be left as a sequentially invoked process. In the multi-threaded methods, getter method calls were limited and these values were stored as temporary local variables.

**Verification**

The correctness of the physics calculations in this problem was verified by the GUI. This was made possible by creating the GUI first. Many different scenarios, both random and calculated, were tested and repeated for signs of abnormalities and/or bugs. By adjusting values (mass, radius, position, and velocity), of specific bodies, via the GUI, many tests and observations could be made in a relatively short period. Adjusting the frames per second value and gravitational constant allowed for tests among the entire group of bodies.

**Timing Experiments**

The following tables and plot show timing data of the number of threads versus computation time in seconds for the following command:

java FreeBodies <number of threads> 320 50 70000

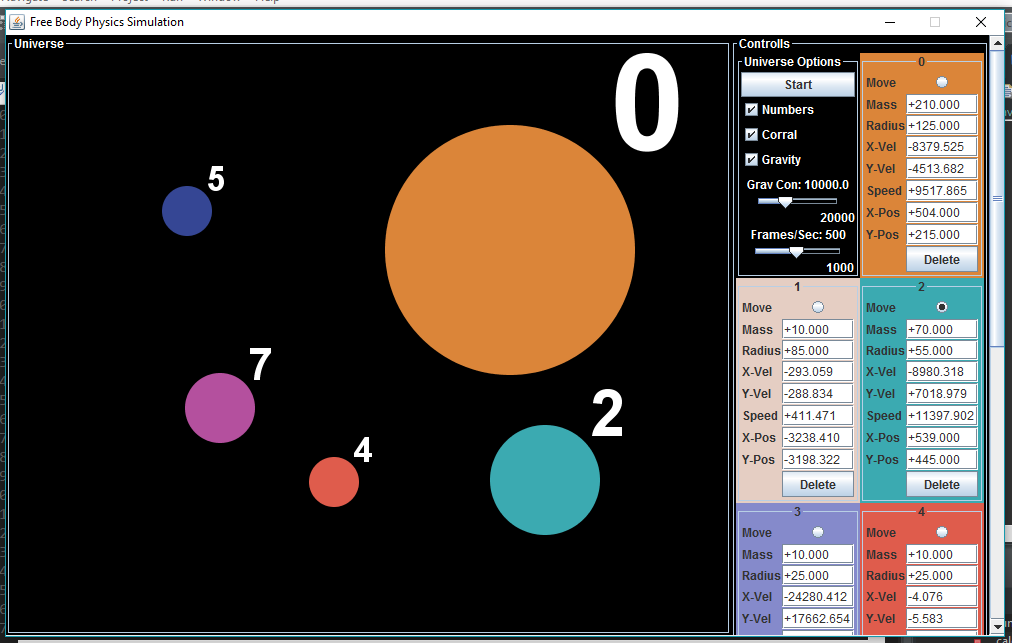
These same tests were run on oxford.cs.arizona.edu and my local machine.



As seen in the data above, computation time generally decreases as the number of threads increases. This follows logically, as the computation workload is split among the threads. The discrepancies in the oxford tests are most likely due to heavy than desired traffic on the server which cause delays among the multiple threads. My local machine’s processor limitations are responsible for the up-tick in average computation time at 32 threads.

**Other Experiments and Extensions**

*- GUI and Variable Options*



As an additional extension/ experiment, an interactive GUI was made to vary real-time body and simulation data. Features include variable body mass, radius, velocity, position, play/pause feature, gravity switch, number labelling, gravitational constant, frames per second, corral feature, deletion of body, click-to-move body feature, and additional body feature. This allows real time experimentation and analysis. The corral feature confines the bodies to the window and turns around bodies that have escaped. All but the numbering option are disabled when the simulation is “play”, as to not interfere.

*- Predictive Collisions*

Collision detection was a key experiment. The effects of this are best seen in the GUI. Instead of detecting collisions after two bodies have overlapped (tangential collisions are very rare), they are predicted and elastically collide. Each body has the following predicted position: Px = position.x + velocity.x \* 1.5 / fps, Py: position.y + velocity.y \* 1.5 / fps). Velocity is extrapolated by 50% as to compensate for the acceleration due to gravity. This value was found after experimentation in the GUI. To keep this computation reasonable each predicted position is only calculated once per frame/ step and remains as a private instance variable.

**Conclusion**

This program solved the kinematic free body problem by way of multi-threading and predictive collisions. To keep computation time to a minimum, method calls were kept low and temporary local variables were utilized, as a space-time trade off. The GUI allows for real-time experimentation and visualization. As the program developed many more questions arose, such as limiting overlap, fine tuning the number of frames per second, and experimenting with orbits. A growing knowledge of vector physics was a key role in making many of the GUI options possible.