Design and implementation of a celestial object simulation using numerical methods in a clustered environment

Scott Lemke
Computer Science Department
University of Wisconsin – Eau Claire
lemkesr@uwec.edu

Abstract

As hardware becomes cheaper, and clustering technology becomes easier, programmers will most likely find themselves programming for a clustered environment at some point in their career. Through the use of a clustered environment a programmer can solve a problem in many ways and with many different techniques, such as concurrent execution, parallelization of an algorithm, multiple processes, multiple threads, or a combination of all of the above. With all of these possibilities there are many design and implementation considerations.

I will take you through the software development process of an ongoing celestial object simulation that calculates the forces, positions, and velocities of an object passing through the Earth-Moon system. It consists of a multi-process, multi-threaded back-end application written in C/C++ for a hybrid openMosix/Beowulf cluster and a PostgreSQL database for data storage, and a front end data analysis tool that allows the researcher to visualize and inspect individual objects and data, written in Java.

Introduction

Dr. Paul Thomas of the Physics department at the University of Wisconsin – Eau Claire had been using Maple to simulate celestial objects moving through the Earth – Moon system and determine the forces acting upon them (N-body simulation). He had several issues with Maple; it was slow and inaccurate in its computations, it was very difficult to automate several hundred runs, and the output was difficult to use.

I offered to rewrite his application to utilize the University of Wisconsin – Eau Claire Student Interest Group Research's cluster to improve his data creation time and ability to analyze that data. The cluster is a 7 node openMosix cluster running on a Linux kernel with varying hardware configurations and 100BT Ethernet network connections.

This paper will focus on the design and implementation of the computation application and storage considerations of this project.

Problem description and algorithm

Given a list of objects with their mass, velocity, position, and acceleration, calculate an objects new velocity, position, acceleration, delta t, and check its Roche limit and collision status at any time t.

To overcome the problems, such as inaccurate data and the $O(N^n)$ runtime, that come with moving each object at the same time intervals, I will use an individual time step for each object. The general algorithm for this is to calculate the forces on an object from each object in the system, permutate them to a 3rd order Taylor series integration, and recalculate the new delta t. Then the object with the next time period up will be calculated and put back into the list.

The final form of the algorithm, including the individual time step and a needed bootstrap sequence is as follows:

Bootstrap

- *Calculate the forces on an object*
- *Calculate the Taylor series*
- *Calculate initial time steps*
- *Order the objects in a list by time step*

Main computation

- *Find the next object to compute*
- *Calculate the new position, velocity, acceleration*
- *Check for Roche limit*
    - *If it is past the limit, split the object and add those into the list, repeat bootstrap for all new objects*

- *Check for collision*
    - *If we do, remove object from list, and flag it.*
- *Update all data for object*
- *Check to see if we are* done with simulation based on maximum time
- Repeat

Requirements analysis

- In the analyzation of the problem, I identified the following list of requirements: It needed to be modular and have a generic base. There are many different N-body algorithms, and I wanted a way to essentially "plug and play" different algorithms without the need to rewrite a large portion of the supporting code.

- It needed an efficient way to store and retrieve data. I chose to use a PostgreSQL database on an external server that the cluster and the data visualization client could both connect to. This is preferred over text/data files because it allows Dr. Thomas and any others that may have interest in this data to not only examine a single objects run, but to compare and contrast several objects at once without the need for multiple open files. It can also lend itself well to data mining to look for patterns in the data.

- The data visualization/analyzation front end needed to be able to run on multiple platforms and have a standard GUI. I chose Java because it allows me quick database connectivity, an easy GUI toolkit, and runs on a multitude of operating systems.

- It had to be maintainable. This is what limited my language choices to C/C++. A large number of numerical simulation applications, including N-body, are written in FORTRAN, mostly because engineers are the ones writing these simulations and FORTRAN is what they know. In my case there are only a handful of individuals that understand FORTRAN, and none who would be able to maintain this code. C/C++, on the other hand, is widely known and has a large list of possible maintainers

Database Design and Schema

Object Table

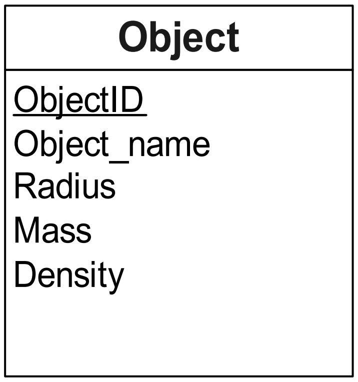

Figure 1: Object Table

The Object table is used to store static data for objects that have been used in a simulation run. This table is to allow users to reuse objects in other simulations without having to re-enter all the data for the object.

Run Table

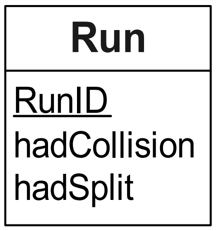

Figure 2: Run Table

The Run table is used to store information that is indicative of a simulation run. HadCollision is a Boolean value to signify if there were any object collisions during the run, and hadSplit is a Boolean to signify if any object passed its Roche limit and split into multiple particles.

Timestep Table

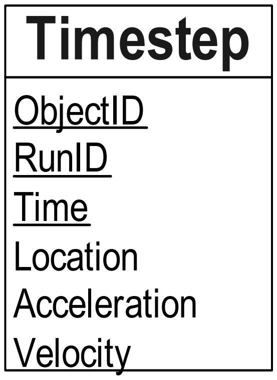

Figure 3: Timestep Table

The Timestep table is the actual data that is created during the run. Location, acceleration, and velocity are all 3 value doubles corresponding to x, y, and z.

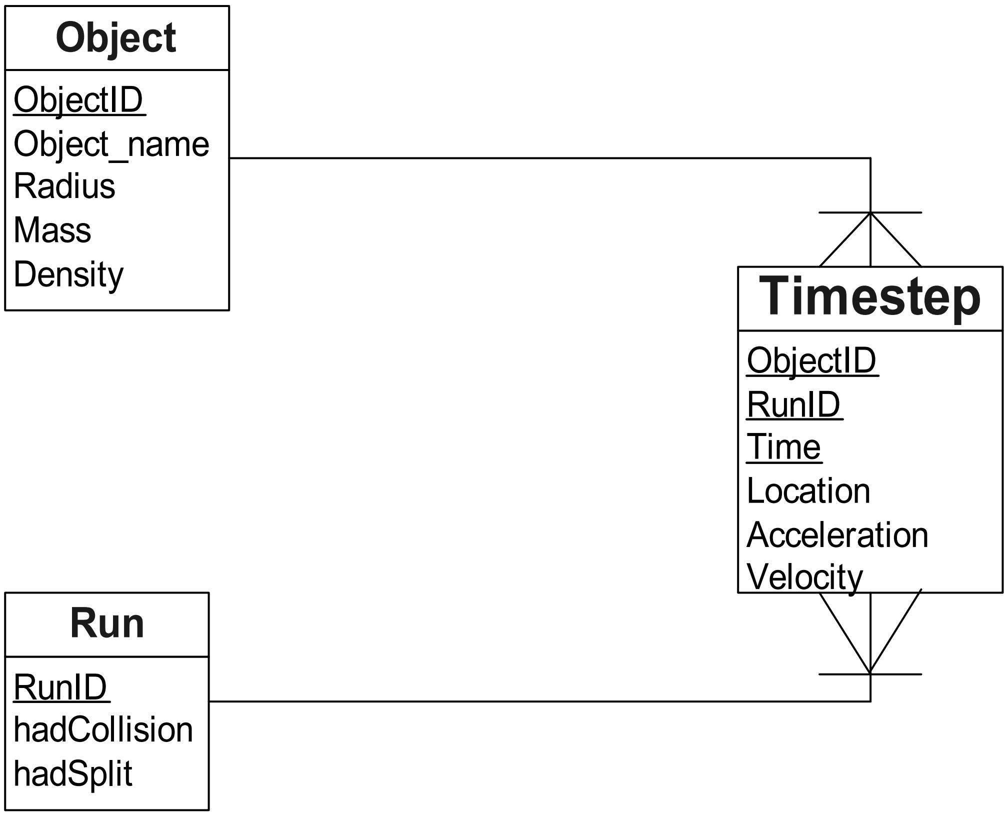

Figure 4: Database Schema

The schema is actually very simple; there is 1 Object and 1 Run per timestep, but multiple Timesteps per Object and Run.

Object Model/Implementation

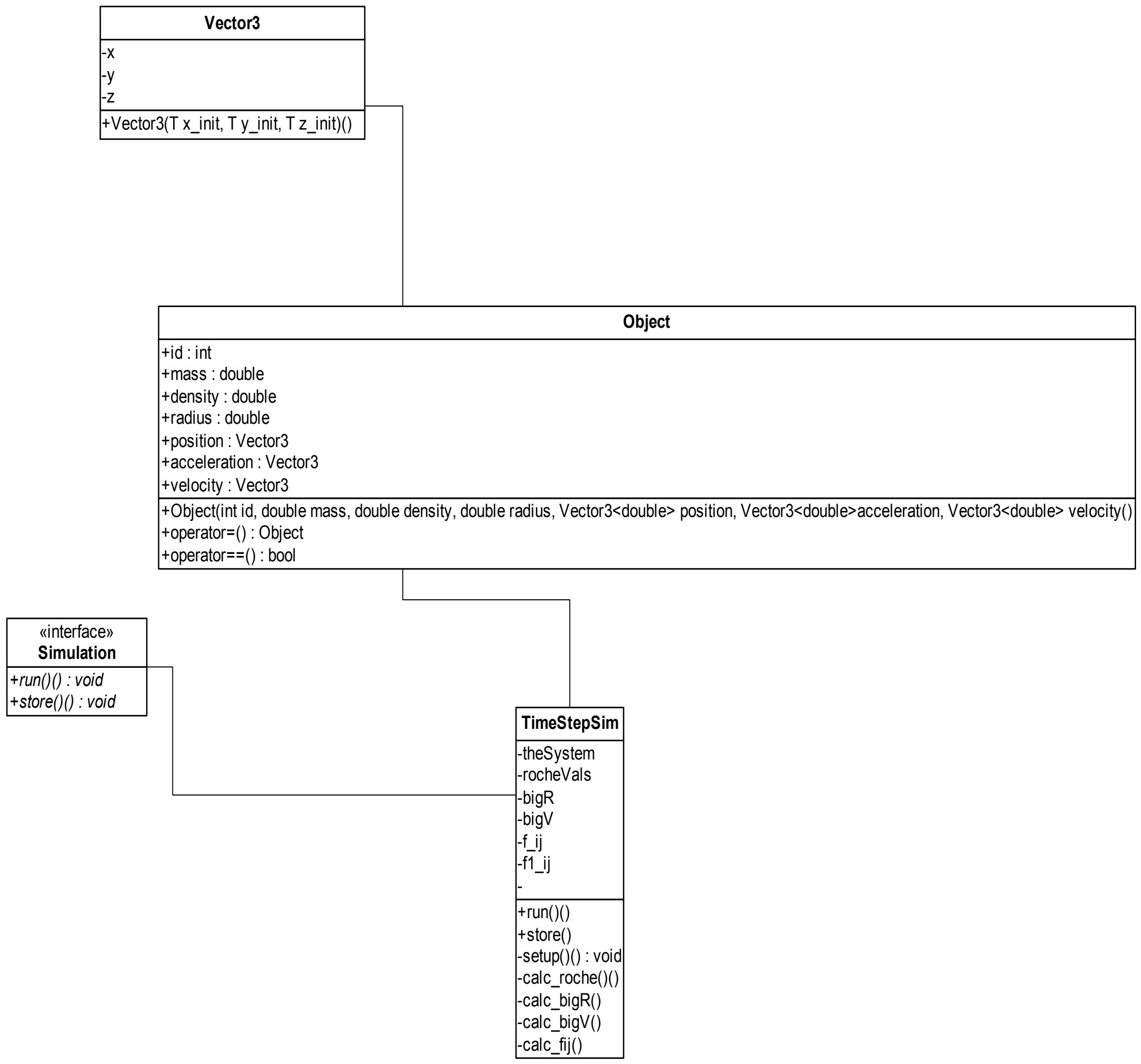

Figure 5: Class Organization

I tried to keep my objects simple as to not confuse myself and any future maintainers of this code. I also tried to follow good object oriented design as much as possible; the only notable deviation is that the member data of Object had to be made public in order for the simulation to work. I found that the physical world does not map perfectly to object oriented techniques because the forces that act on objects depends on both objects, and as is stated in Newton's Laws, most notably his 3rd law of motion, objects are affected by every object and force around them, and in turn affect those objects and forces. Because of this, when Object Oriented design and implementations are followed strictly it causes classes to be extraordinarily large because of the large number of calculations they must accommodate for (basically all their calculations and all calculations that others would do for you) or the classes degenerate into nothing but a few methods and getters and setters for each member data item, which is on the road to public data.

In addition to the final object oriented design and implementation, I tried a straight C and procedural implementation, modularizing at the file level. The first version of this code came out quite well, but it was only to test maximum data creation speed and implemented a very small part of the overall algorithm. When I started to add the other parts of the algorithm to this code I found it becoming increasingly hard to remember where code was, what code was doing, and the different structures to pass around, all very typical of a C program.   While this code would have been possible, as has been historically shown, and it would have been efficient code, it would have been hard to maintain, and more important, hard to reuse in the future for similar projects.

The final design and implementation is a mixture of object oriented and procedural styles, and I am finding it works very well. The object oriented aspects give me the ability to create generics and a framework that can be easily used and reused. Object oriented also gives me the ability to hide some of the inner workings and create a black box that could be extended and reused.

The procedural style gives me the concepts of structs, data that is linked but not necessarily a full class but is passed around as an atomic unit; the Object class is essentially a struct with an equal and equality operator.

Application workflow

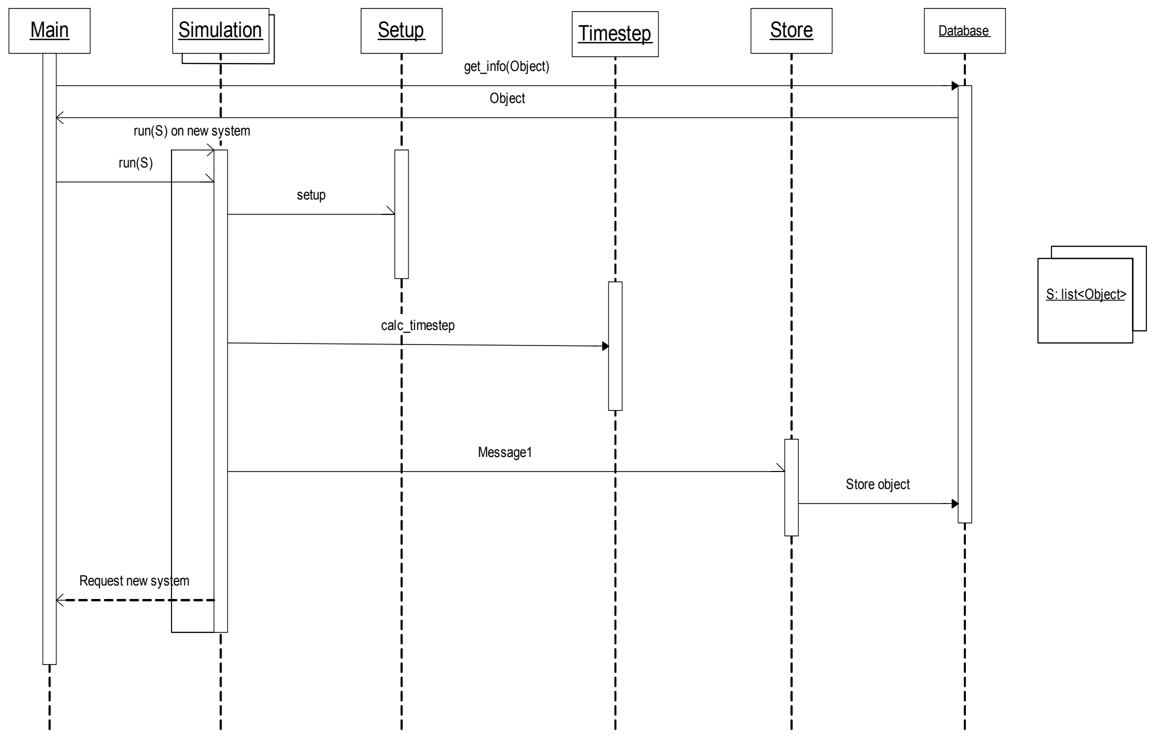

Figure 6: Sequence Diagram

The basic architecture and flow of the application is that of master/slave. The master, or main, gathers lists of objects, specified by the input parameters, and their information from the database to create systems. From here the main begins to fork off slave simulations to create the data from that system. Each simulation goes through a setup phase to calculate some initial data, and then begins to go through the time step algorithm. After each time step, the object data that was calculated is passed off to storage and the next time step is calculated. This continues until an exit condition is met, such as time exhausted, at which time the simulation does not exit, but rather signals to the master that it needs a new system to calculate. If the master still has a system in its queue to calculate, it passes this to the slave which begins its simulation process again, if there is nothing left to do, the master signals for the slave to exit.

I chose this pattern, instead of the constant creation and exiting of new processes, because it minimizes overhead and maximizes my running time. By creating a slave process only once and reusing it I gain the cycles it would have taken to fork a new process, I also gain the reduced bandwidth of the head node, which the master resides on, pushing off slave processes to other nodes, I now only have to push out a small list. This gives me more time for calculation and less overhead. It will also allow me in the future to create a sub-master on each node that will take a list of systems and be able to manage a group of slaves on its own node locally, which will give me even more processing time and less overhead.

Future Work

At the writing of this paper I am in the beginning of testing the methods for mathematical correctness on an individual basis, after which I will move into testing the overall algorithm for correctness, and the first data for analysis should be being created in 2-3 weeks. Further improvements to this application include implementing more N-body algorithms, increasing parallelization in both the master and slave pieces of the algorithm, and adding in a fluid particle simulation for the splitting and collision of objects, possibly making that a service on the cluster that is constantly running.

References

Graps, A. (2000). *N-Body / Particle Simulation Methods*. Retrieved December 1, 2003, from http://www.amara.com/papers/nbody.html.