HPC Lecture Notes
Simulation and the N-Body Problem

Thomas Fogal

November 18, 2014

1 Simulation

Simulation is the process of imitating some real process on a computer. This involves creating a mathematical model of the process of interest, and solving that model given a set of ***initial conditions***. The simulation is then used to model the system and understand what its state will be at some future point of time.

Simulations are used in basically any science that exists today. Weather simulations attempt to predict such values as temperatures and precipitation at a particular time. Simulations are used in the design of engines to estimate and evaluate their efficacy. Similarly, simulations are used to determine ideal fuel mixing ratios. The movie industry simulates the transport of light to develop the Hollywood blockbusters we watch. Network simulations estimating packet flow are used to improve network architectures. Stress simulations ensure building designs will be safe when acted on by external forces (e.g., an earthquake).

In some cases, it is possible to solve a model analytically. Unfortunately, such cases are rather limited in scope, and frequently are too simplified to answer questions of interest. Thus we rely on *numerical* solutions, which approximate the system using numerical integration. The ***resolution*** or fidelity of numerical solutions is the accuracy implied by the method of solution; an entire discipline has developed which specifically addresses accuracy in numerical approximations.

The basic lifecycle of a simulation consists of input, subdivision of work and load balancing, advancing the state of the system (often, 'timestepping'), and output of intermediate and ending states. Each of these components can be very complex on their own, and are frequently intertwined. For example, a spatial subdivisioning often effects which data are loaded on which processes. We will talk about each of them in turn over the course of the semester.

2 High-Performance Computing

The desire for increased fidelity and the simulation of larger systems has fueled the field of ***high-performance computing (HPC)***. As people realized that a single processor and

machine just does not have enough resources to solve a model in the desired timeframe, we have increasingly turned towards ***parallelism*** to increase performance. Using multiple processors in concert enables higher-resolution simulations, but requires extra work in synchronizing and getting the processors to work together on the problem.

At first, supercomputers were developed by companies pushing unique architectures that were especially suited to massive parallelism. Such machines boasted special SIMD ('single instruction, multiple data') instructions for performing operations on multiple elements at once[1].

Modern day supercomputing can be traced to more humble beginnings, however. ***Beowulf clusters*** emerged from a group at NASA that needed more computational power. Supercomputers put out by Cray and the like were *expensive*. To circumvent the cost issue, the group at NASA took a set of everyday workstations running Free software and connected them with off-the-shelf networking technology. The result was a cluster that could be bought at a standard electronics store; the cost savings due to mass production of the materials is enormous. Beowulf enabled a much wider set of users for high-performance computing than was previously possible.

Modern supercomputers have certainly changed, but they maintain many of the same characteristics as the original Beowulf design: distributed memory (separate machines), free software, and primarily off-the-shelf processing and networking hardware.

One of the aspects that has changed, especially in the last 5 years, is the rise of multi-scale parallelism by way of shared memory. Supercomputers now are universally composed of a set of multiprocessors instead of uniprocessors, and this trend continues as so-called 'manycore' accelerators become commonplace. Exploiting the parallelism at both scales is critical for obtaining good performance, as you will see during this course.

3 N-Body Problem

The N-Body problem is a classic problem in high-performance computing. It simulates how a set of different bodies move in space given their gravitational attraction to one another, based on Newton's classical mechanics. One example of initial values for this problem is the current configuration of planets in the milky way: if the simulation is correct, the simulation can predict the planetary configuration at any future point in time. However, one can also see that this problem quickly becomes intractable: adding the moons, stars, and asteroids of our galaxy makes the system *considerably* more computationally complex, and if we start to consider multiple galaxies ...

In this course, we use the N-Body problem for the sole reason that it is easy to explain and implement. The focus of this course is high-performance *computing*. We will give you enough information on the problem to be able to implement it, but we stop there; there are much more intelligent ways to do this. We encourage you to keep your implementations simple, especially at the start of the semester.

[1] A modern example of SIMD instructions are Intel's **SSE** instructions; indeed, 'SSE' stands for 'Streaming SIMD Extensions'

The mathematics for classical Newtonian gravitation is given by Equation 1

$$F_1 = F_2 = G\frac{m_1 m_2 r_{12}}{\|r_{12}\|^3} \tag{1}$$

where F represents a force, G is Newton's gravitational constant ($\sim 6.674\times10^{-11}N(m/kg)^2$), m is the mass of a particle, and r represents the distance between the two bodies. Unfortunately, as r goes to 0, the force goes to infinity. To combat this issue, we add a softening factor ϵ:

$$F_{ij} = G\frac{m_i m_j r_{ij}}{(\|r_{ij}\|^2 + \epsilon^2)^{3/2}} \tag{2}$$

We are uninterested in Force, per se, but rather only a particle's acceleration. However, of course F is equivalent to ma, and so we can obtain the acceleration simply by dividing by the mass. With the acceleration we can modify the velocity, and given a constant velocity and a length of time, we can compute a particle's new position from it's old position. In an N-Body simulation, we sum up the effect on a particle's acceleration by all other particles, a la Equation 3.

$$m_i\vec{p_i} = G\sum_{j=i}^{N-1}\frac{m_j m_i(p_j - p_i)}{(\|p_j - p_i\|^2 + \epsilon^2)^{3/2}} \tag{3}$$

Here $\vec{p}$ represents the acceleration of point p; other symbols are the same. Equation 3 gives the new acceleration for the i'th point; note that in a real simulation, we need to calculate Equation 3 N times!

3.1 Timestepping

Once we have obtained the acceleration for each particle, we apply that to each particle's velocity. With the velocity, we can finally advect the particle, but: *how much?* Over what time period should we assume the particle's velocity is constant?

Selecting this time period intelligently involves a lot of math, and can change the entire logic for summing up the forces. It has a *huge* effect on the running time (likely more effective than all of the methods we will learn about in this course), but is beyond the scope of this class.

Thus: for now / in this course, we'll just use a constant time in a simple leapfrog integration. If 'a' is particle's acceleration, then the new velocity v and the new position x are calculated like so:

$$a = A(...) \tag{4}$$
$$v_{i+1} = v_i + a\Delta t \tag{5}$$
$$x_{i+1} = x_i + v_{i+1}\Delta t \tag{6}$$