Informatik und Angewandte Kognitionswissenschaft
Lehrstuhl für Hochleistungsrechnen

Thomas Fogal
Prof. Dr. Jens Krüger

High-Performance Computing

http://hpc.uni-due.de/teaching/wt2014/nbody.html

Exercise 2 (70 Points)

All assignments should be pushed to your personal Git repository. Assignments are due at midnight on the due date. No late assignments will be accepted.

All assignments must include a `makefile` for compiling your assignments. Assignments which do not compile will receive 0 points. Assignments that do not satisfy the test inputs will receive 0 points.

Please **do not include output** other than what was requested by the assignment details.

Your assignment will be graded on the `duecray.uni-due.de` supercomputer. It does not matter if your program runs correctly on another machine; it must run correctly on `duecray` to receive credit.

1 N-Body Simulation

In this assignment your task will be to write a serial n-body simulation.

1.1 Quick Reminder

In an n-body simulation we try to simulate the gravitational effects that multiple objects have on each other. The necessary computational workload grows fast with the number of objects. This is due to the fact that we have to compute how every single object is being affected by every other object.

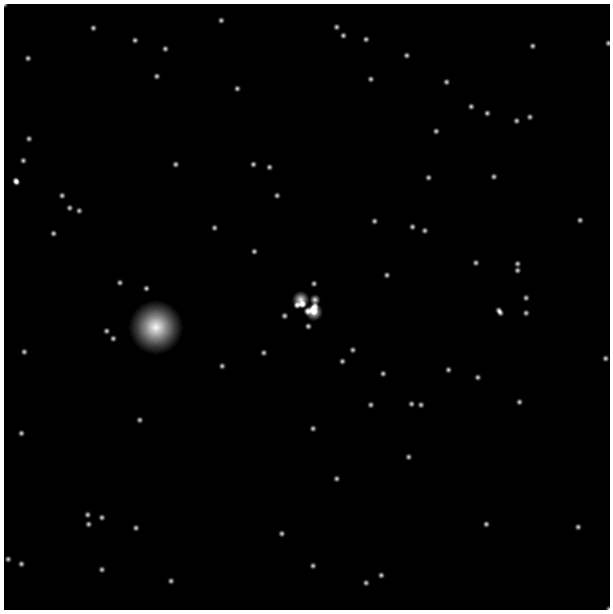

Figure 1: Example visualization from an n-body simulation like the one you will develop.

1.2 Relevance

In reality we should calculate the gravitational influence for every possible point in time—a continuous variable. Given the difficulty this poses, we will settle for discrete timesteps. This assignment will be the foundation for upcoming assignments. The calculations that have to be done within every timestep offer themselves for parallelization and future assignments will therefore be about adapting the code to be executed in parallel. Having a proper architecture to begin with will save you time in the future.

1.3 Overview

Figure 2 might be helpful to give you a better understanding of what needs to be done.

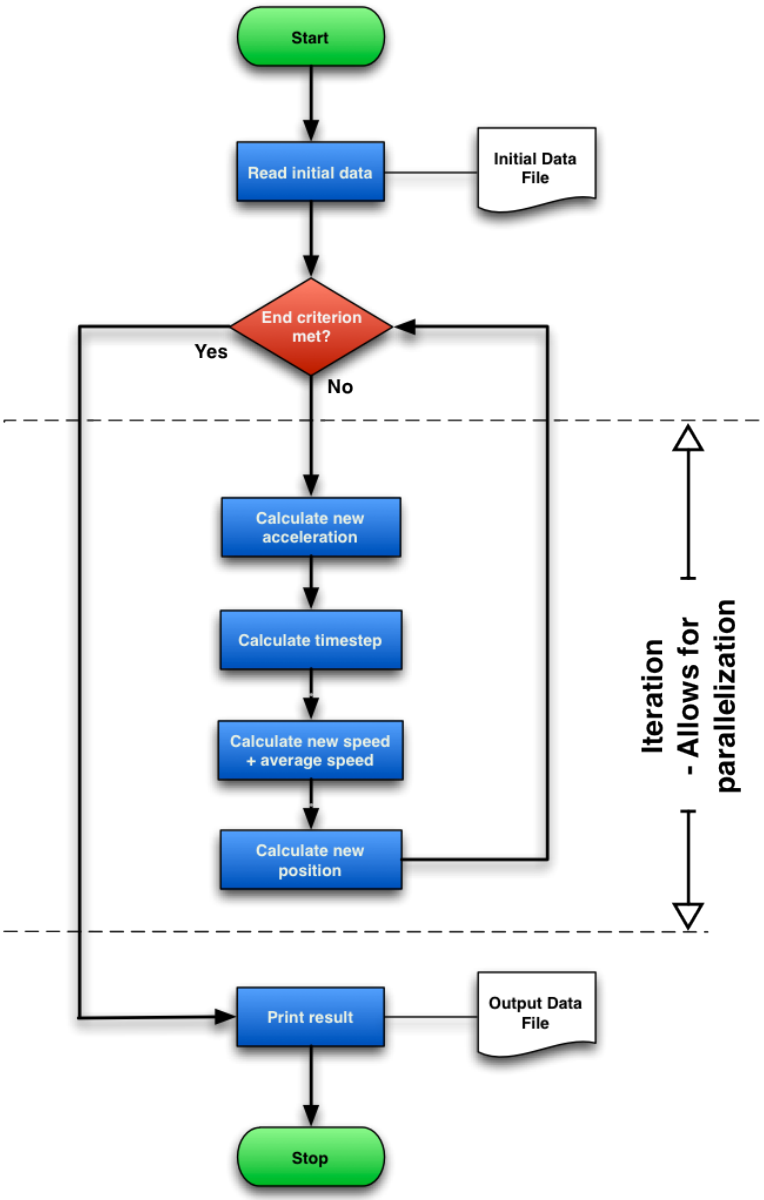

Figure 2: Overview

2 Command line arguments

Your program must accept a number of command line arguments. These arguments are:

- `-i` (string): the filename for the input file.
- `-n` (integer): the total number of objects.
- `-d` (float): an "end time criterion": when the simulation should stop.
- `-t` (float): timestep. The Δt to utilize.
- `-e` (float): epsilon. The 'softening factor' used to prevent infinite accelerations.
- `-o` (integer): number of timesteps between outputs.

Options may not necessarily appear in that order. I suggest you use the `getopt` function for parsing command line options.

3 Input File Format

The input files that your simulation must read follow the format given in Listing 1:

```
"mass", "x", "y", "z", "vx", "vy", "vz"
 200.0,  1.0,    1.0,    1.0,  00.00,  00.00, -00.00
2000.0,  10.0,   10.0,  10.0, -01.00, -00.30, -00.90
2000.0, -10.0,  -10.0, -10.0,  01.00,  02.00,  00.02
9000.0, -150.0,  20.0,   0.0,  12.20, -01.95,  00.12
  70.0,  150.0, -15.0,   0.0, -06.40,  00.70,  00.02
```

Listing 1: Simple input file.

Note the quotes in the header line, and variable number of spaces separating items. This format is the same as your output format; more detail is given in §6.

4 Iterator

Once reading and writing files works, we can take a look at the main part of your program; you will have to write code that starts and executes the iteration loop. What we want to calculate is where every object is going to

be after every iteration step. For a hint at the necessary steps take another look at Figure 2. Some of those steps require additional precautions, which are described in this section.

4.1 Calculate new acceleration

The basics of this step cause most of the computational load. You will have to calculate the acceleration that every object will be affected by in the current timestep. To get this total acceleration of one object you will have to accumulate the acceleration influence of all other objects on your current object. If p is a particle's position, then $\vec{p}$ is the particle's velocity, and $\vec{\vec{p}}$ is that particle's acceleration. Then the acceleration is given by Equation 1:

$$\vec{\vec{p_i}} = G \sum_{j=0}^{N} \frac{m_j(p_j - p_i)}{(\|p_j - p_i\|^2 + \epsilon^2)^{3/2}} \tag{1}$$

Note that $\|var\|$ is the vector norm[1] of the vector var. The $p_j - p_i$ quantities are point distances, not simple subtractions (the position is a 3-element array, not a scalar!).

For this course, you should use $6.673e-11$ for the gravitational constant G.

4.2 Calculate new speed, average speed and position

The integration scheme we will use for this course is similar to leapfrog integration, with some small modifications to make it easier to calculate.

$$a = A(...) \tag{2}$$
$$v_{i+1} = v_i + a\Delta t \tag{3}$$
$$x_{i+1} = x_i + v_{i+1}\Delta t \tag{4}$$

A is a function which computes Equation 1.

[1] https://en.wikipedia.org/wiki/Norm_(mathematics)#Euclidean_norm
https://de.wikipedia.org/wiki/Vektor#L.C3.A4nge.2FBetrag_eines_Vektors

5 File Handler

We will supply you with some example input and output files. Your first step should be to write

- a simple data structure in which to keep the data stored in the input files,
- code that helps you with reading the input files, and
- code that helps you with writing your results back out into a file.

6 Output Format

You will need some way to verify your code is working as expected. For that, your program should output a CSV (comma-separated values) file for a timestep. You should produce an output every `-o` timesteps. Use code similar to Listing 2 to calculate this, where `outputs` comes from parsing the `-o` command line option.

```
if((++ts) % outputs == 0) {
  write_data(...);
}
```

Listing 2: Sample code to determine if you should create an output.

More details on CSV files can be found by searching the web, but they are simple; your CSV file should have the following format:

```
  "mass", "x", "y", "z", "vx", "vy", "vz", "scalar1-name", ...
  M, X0, Y0, Z0, VX0, VY0, VZ0, scalar0, ...
  M, X1, Y1, Z1, VX1, VY1, VZ1, scalar1, ...
  M, X2, Y2, Z2, VX2, VY2, VZ2, scalar2, ...
  M, X3, Y3, Z3, VX3, VY3, VZ3, scalar3, ...
```

Listing 3: CSV output file format

where $X0, Y0, Z0$ gives the location of the first particle, $VX0, VY0, VZ0$ give the velocity of that particle, and $scalar0$, is some scalar associated with it. You do not need to output more than mass, position, and velocity, but you are welcome to include other outputs as long as they satisfy the CSV restrictions. Every line must have the same number of values, and a column should have the same meaning in each line.

The first line of this file is a 'header' line, which simply gives a name to each column. You must have a name for every field, otherwise common tools will not be able to read the file; likewise, you cannot have more fields in the header than you have in the data lines.

In most cases, these values should be floating point numbers. You can output them using (e.g.) "`%7.4f`" in your C program.

As a concrete sample of the output format, see Listing 4. This file describes 4 particles, each with 3 associated scalars (presumably the 3 components of the particle's velocity).

```
"mass", "x", "y", "z", "vx", "vy", "vz"
1000.000, 07.3713,   0.3218,   3.1043,  1.1833, 2.0134,   2.2291
1000.000, 10.2746,  26.4729,  70.5028,  8.0630, 8.4932,   6.0171
1000.000, 02.2911,  09.1276,  10.8019,  6.3234, 1.1237,  15.5061
1000.000, 24.1717,  10.4758,  15.2072,  2.0334, 4.7838, 101.1451
```

Listing 4: Sample CSV output file

If the file describes (z.B.) timestep 164, it would make sense to store it in a file named "`164.csv`".

7 *sine qua non*

As always, submit your assignment by committing your code to your personal repository. Every assignment utilizes a new repository. The name for this assignment is **as2-*username***, where *username* is your name on our git server.

All assignments must include a makefile to compile your program. No makefile, no points!