n-body problem - Gravitation

Optimization training - Adrien Cassagne

1 Problem overview

The n-body problem is a classic one from the Newtonian mechanics: it consists in resolving gravitation equations. However, this problem can be generalized (from the algorithmic point of view) and we can often find it in numerical simulation: this is why it is a good real case study.

At the problem beginning (the time t), for each body i, its position $\vec{q_t}(i)$, its mass m_i and its velocity $\vec{v_t}(i)$ are known. The applied force between two bodies i and j, at the t time, is defined by:

$$\vec{F_t}(i,j) = G.\frac{m_i.m_j}{(D_{i,j})^2}.\vec{u}, \tag{1}$$

with G the gravitational constant ($G = 6,67384 \times 10^{-11} m^3.kg^{-1}.s^{-2}$), $D_{i,j}$ the distance between i body and j body and $\vec{u}$ the unit vector pointing from i to j. The resulting force for a given i body, at the t time, is defined by:

$$\vec{F_t}(i) = \sum_{j \neq i}^{n} \vec{F_t}(i,j), \tag{2}$$

with n the space bodies number. The acceleration characteristic for a given i body, at the t time, is defined by:

$$\vec{a_t}(i) = \frac{\vec{F_t}(i)}{m_i}. \tag{3}$$

The i body velocity characteristic at the $t+dt$ time depends on the velocity and the acceleration at the t time:

$$\vec{v_{t+dt}}(i) = \vec{v_t}(i) + \vec{a_t}(i).dt. \tag{4}$$

At the end, the i body position q at the $t+dt$ time depends on the position, the speed and the acceleration at the t time:

$$\vec{q_{t+dt}}(i) = \vec{q_t}(i) + \vec{v_t}(i).dt + \frac{\vec{a_t}(i).dt^2}{2}. \tag{5}$$

Thanks to Eq. 1,2, 3, 4 and 5, it is now possible to compute the new position and the new velocity for all i bodies at the $t+dt$ time.

1.1 dt time step choice

The easier way to determine dt is to select it as a constant for all the simulation iterations. For some visualization concerns we will choose to use a constant dt but, for simulation precision interests, it is better to compute a new time step for each iterations depending on the distance between the bodies. The following equation describes the variable dt calculation:

$$\|\vec{v_t}(i).dt + \frac{\vec{a_t}(i)}{2}.dt^2\| \leq 0.1 \times D_{i,j}, \tag{6}$$

with j the nearest body to the i body. For each i body, a time step is calculated and the smallest one is chosen. Eq. 6 traduces that the distance between i_t and i_{t+dt} must be below 10% of the $[i_t, j_t]$ distance. This equation assures that two masses cannot be closest than 20% between t time and $t + dt$ time. However, Eq. 6 is not directly usable: this is a 4^{th} degree polynomial equation in dt. It's why we will use the triangle inequality witch allows us to determine a new condition:

$$\|\vec{v_t}(i)\|.dt + \frac{\|\vec{a_t}(i)\|}{2}.dt^2 \leq 0.1 \times D_{i,j}. \tag{7}$$

Eq. 7 is a 2^{nd} degree equation: this is more reasonable in term of computational time.

1.2 Algorithmic

From an algorithmic point of view, we are calculating each body depending on all the other bodies. We can traduce that by 2 overlapping `for-loops` from 0 to n bodies. In a `C-like` language we can express the problem this way:

```
body b1[N]; // array which contains all the bodies (at t time)
body b2[N]; // empty array
for(unsigned int iBody = 0; iBody < n; ++iBody)
  for(unsigned int jBody = 0; jBody < n; ++jBody)
    if(iBody != jBody)
      b2[iBody] = compute(b1[jBody]);
```

Listing 1: n-body pseudo-code algorithm

Alg. 1 shows an important characteristic of the n-body problem class: for n given bodies, the algorithmic complexity in term of computational time is approximatively $O(n^2)$. It exists other methods to approximate and to resolve the problem in $O(n \log n)$ time but this is not in the range of this lab (see Barnes-Hut simulation).

2 Aim of this lab

The purpose of this lab is to teach you how to proceed in the optimization process. So, the code size is bigger than previous exercises. Here are the main aims of this lab:

1. Quickly understand a real simulation code.
2. Detect hotspots in the code.
3. Apply tools introduced in the lesson 2 on the hotspots (Gflop/s, Roofline model, etc).
4. Optimize the code kernel and reduce the restitution time.
5. Make a parallel code adapted to modern CPU architectures.
6. And of course... have fun!

The given code simulates bodies gravitation in 3D. The visualization will be achieved by OpenGL and you will see the bodies rendering in real time.

3 Compile and execute the code

3.1 Compile the executable

For some practical reasons the project use `cmake` instead of a standard `Makefile`. Don't panic this is not very complicated and this section is made to help you. First thing to do is to generate the `Makefile` with the `cmake` system:

```
> cmake .
```

This command will generate the `Makefile` file in the current directory. But before using the `Makefile` we will configure the generated `CMakeCache.txt` file. This file is the configuration file for specify our favourite compiling options. Here is the recommended configuration:

```
//Choose the type of build, options are: None(CMAKE_CXX_FLAGS or
// CMAKE_C_FLAGS used) Debug Release RelWithDebInfo MinSizeRel.
CMAKE_BUILD_TYPE:STRING=Release

//Flags used by the compiler during all build types.
CMAKE_CXX_FLAGS:STRING=-DNBODY_FLOAT -march=native -fopenmp
```

Now we can use the `Makefile` with this command:

```
> make
```

After that, the binary file will be generated in `bin/Release/nbody`. Launch the executable like this from the root folder:

```
> ./bin/Release/nbody
```

3.2 Run the executable

There is two ways to launch the initial n-body code. The first one initializes bodies from an input file:

```
Usage: ./bin/Release/nbody -f inputFile -i nIte [-h] [-v] [-w outputFiles]

  -f    the bodies input file name to read.
  -i    the number of iterations to compute.
  -h    display this help.
  -v    enable the verbose mode.
  -w    the base name of the body file(s) to write.
```

The `data/in` folder contains some test cases, you can launch them like this:

```
> ./bin/Release/nbody -f data/in/8bodies.dat -i 200 -w data/out/8bodies
```

This command will run the 8bodies test case with 200 iterations and will write the solution at each iteration in `data/out/8bodies.*.dat` files.

There is not big enough test cases to really fill the CPU capacity. This is why there is an other way to launch the n-body code based on the number of bodies we want to simulate:

```
Usage: ./bin/Release/nbody -n nBodies -i nIte [-h] [-v] [-w outputFiles]

  -n    the number of bodies randomly generated.
  -i    the number of iterations to compute.
  -h    display this help.
  -v    enable the verbose mode.
  -w    the base name of the body file(s) to write.
```

You can launch the n-body code like this:

```
> ./bin/Release/nbody -n 500 -i 1000 -v
```

This command will launch the simulation with 500 bodies randomly generated (1000 iterations). Note that the verbose mode is enabled, so it will display some information for each iteration in your shell.

There is a lot of other options you can use with this code: try the `-h` option to display them. If you do not want to run OpenGL rendering you can put the `--nv` (no visualization) in order to disable it.

4 Work to do

4.1 Part 1: standard optimizations

Go into the `part1_standard_opti` folder: this will be the root folder for the next instructions. You will find the source code in the `src` directory. The entry point of the code is in the `src/main.cpp` file. Bodies simulation code is in the `src/Space.h`, `src/Space.hxx` module. The `src/utils` directory contains different tools in order to reduce the `main` code size. The `src/ogl` directory contains the OpenGL code for the bodies visualization.

1. Play with the code, take some time to understand the executable options.
2. Profile the code with `gprof` profiler in order to determine witch part of the code is time consuming.

Help:

`gprof` profiling can be enabled by adding `-pg` compiling option in the `CMakeCache.txt`:

```
//Choose the type of build, options are: None(CMAKE_CXX_FLAGS or
// CMAKE_C_FLAGS used) Debug Release RelWithDebInfo MinSizeRel.
CMAKE_BUILD_TYPE:STRING=Debug

//Flags used by the compiler during all build types.
CMAKE_CXX_FLAGS:STRING=-DNBODY_FLOAT -march=native -fopenmp -pg
```

Here, we compile in `Debug` mode because it is easier to understand the traces with this type of compiling. `-pg` option generates extra code to write profile information suitable for the analysis program `gprof`. You must use this option when compiling the source files you want data about, and you must also use it when linking.

After the `CMakeCache.txt` modification, think to re-compile the executable:

```
> make
```

Now you have to launch the debug executable in order to generate a profiling trace:

```
> ./bin/Debug/nbody -n 500 -i 1000 --nv
```

This command will generate a `gmon.out` file containing a non human readable profiling trace. Now if you want to create a human readable trace you have to run the `gprof` command:

```
> gprof -p -b ./bin/Debug/nbody gmon.out > trace.gprof
```

You can open the `trace.gprof` file with your text editor and see the time consuming routines. Once you have detected the code hotspots you can re-do the profiling in `Release` mode in order to confirm the previous `Debug` profiling conclusions.

3. Count the number of floating-point operations (flops) in the most time consuming routine (for one iteration). Report this value in the `main.cpp` file line `174` in the `flopsPerIte` variable: now you will have Gflop/s dynamically for each iteration and at the end of the execution.
4. Estimate the number of memory accesses (memops) in the most time consuming routine (for one iteration).
5. Calculate the attainable performance of this n-body code with the Roofline model.
6. Is this a memory-bound code or a compute-bound code?
7. Launch the code with different problem sizes and keep the results into a file. What can you say about the CPU performances evolution?
8. Reduce the number of operations in the most time consuming routine and try to make the faster code as you can.

Help:

It could be a good idea to write the calculations on a peace of paper and try to factorize things. Remember that the divisions come with a big cost (in time). You can also use compiling options in order to reduce the restitution time. Be creative, do not hesitate to try things.

9. Make a multi-threaded code with OpenMP.

Help:

Try to start by the most time consuming part of the code.

10. What can you say about the code speed up with the OpenMP version? Why?

4.2 Part 2: vectorization

Go into the `part2_vectorization` folder: this will be the root folder for the next instructions. `src/Space` module have been modified a little bit for an easier vectorization of the code. The arrays have been manually split in vectors.

1. Try to auto-vectorize the most time consuming routine.

> **Help:**
>
> Remember the no-aliasing pointer qualifier with the `__restrict` keyword. You can also help the compiler in this task with the `const` keyword (alias read-only). And of course do not forget to check the vectorizer reports with the `-ftree-vectorizer-verbose=2` compiling option. You can re-direct vectorizer reports into a file like this:
>
> ```
> > make > vec_report.txt 2>&1
> ```

2. Vectorize the code with intrinsic calls. You can use the `myIntrinsics.h` wrapper as well as real intrinsic calls.

> **Help:**
>
> Check the `myIntrinsics.h` wrapper documentation in the `doc` directory.

3. Make a multi-threaded code with OpenMP.
4. What can you say about the code speed up with the OpenMP version? Why?