

Assignment 2: Molecular dynamics model of a collection of water molecules

Practical information

Deadline: Saturday 14/2 23.59

Resources:

- ERDA for file storage
- Jupyter for the Terminal to access DAG
- DAG for testing, visualization, and benchmarks

Hand-in:

- A report of up to 3 pages in length (excluding the code)
- Your vectorized (Struct-of-Array) code both as source file as well as in pdf
- An up to 3-minute video where one person from the group scrolls through the code and explains what you have done.
- Use the template on Absalon to include your code in the report

Introduction

N-body simulations are common in physics and chemistry. The basic model has $O(N^2)$ complexity (for N bodies the number of calculations needed is proportional to N^2), where each body is updated by applying the forces of all other bodies. In the case of molecular dynamics (MD) the goal is to obtain the physical movements of the interacting molecules. The atoms of the molecules interact by certain energy potentials, and the system evolution is obtained numerically by integrating Newton's equations of motion. This method is a powerful investigatory tool in physics, chemistry, materials science, and biology. We need to define certain intramolecular and intermolecular potentials to perform a MD simulation.

While mathematically the intermolecular forces affecting each atom depends on every other atom, the vast majority of the force is caused by the few nearest molecules. For this reason, we will use the approximation of only considering the $n_closest$ (set to 8) molecules, the “neighbours”. Since this changes very slowly compared to e.g. the oscillations on bonds, we update the list of neighbours only once every 100 steps

Modern MD simulation software such as GROMACS and NEMD use certain approximations to reduce this problem to an $O(n)$ problem using a much more sophisticated technique than our “neighbours” method, and they are used for simulating systems of millions of particles for millions of time steps.

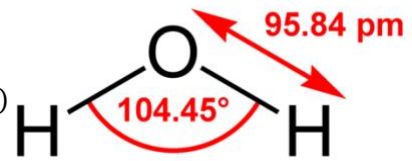
In this tutorial, we will make a simple MD program to simulate N water molecules. Three atoms represent each water molecule, and therefore the system contains $3 N$ particles. These particles interact with one another with a set of potentials.

For atoms belonging to the same molecule, two potentials exist. Harmonic and angle potentials. The Harmonic one is to model the covalent bond.

$$V_{bond}(r) = \frac{k_b}{2}(r - l_0)^2 \quad (1)$$

The angle potential is an interaction between three particles

$$V_{angle}(\phi) = \frac{k_a}{2}(\phi - \phi_0)^2 \quad (2)$$



where k_a, k_b are force constants, $l_0 = 95.48$ pm the bond length, and $\phi_0 = 104.45^\circ$ the bond angle.

Non-bonded potentials describe interactions between any pair of particles. Non-bonded potentials are only applied among the particles that do not interact with one another through bonded potentials, because they are already included in the bonded forces. In this MD code, we use Lennard Jones and Coulomb potentials for the non-bonded potentials. The r^{-6} term in LJ potentials models the Van der Waals force. What does r^{-12} model?

$$V_{LJ}(r) = \varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r} \right)^{12} - \left(\frac{\sigma_{ij}}{r} \right)^6 \right]$$

$$\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}$$

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}$$

For the Coulomb potential, in many popular MD forcefields (e.g. CHARMM, AMBER) q_1 and q_2 are often smaller than one elementary charge. *How can the charges be less than an elementary charge?*

$$V_E(r) = k \frac{q_1 q_2}{r}$$

Using the potentials, we can obtain the total force acting on each particle and we can integrate the equation of motion by means of some integrator algorithm; here we are using the leap-frog integrator.

$$v\left(t + \frac{1}{2}\Delta t\right) = v\left(t - \frac{1}{2}\Delta t\right) + \frac{\Delta t}{m}F(t)$$

$$r(t + \Delta t) = r(t) + \Delta t v\left(t + \frac{1}{2}\Delta t\right)$$

How to use the code

For this tutorial we need g++ and python, which is provided on ERDA. Spin up a Jupyter session on DAG. Important! Select the “HPC notebook” notebook image. This is the image we use for the course. In the launcher, launch the Terminal.

In the terminal (or the folder view on the right side) you can see two folders: `erda_mount` and `hpc_course`. `erda_mount` contains your own storage area. This is where you save files. In the folder `hpc_course` the exercises are stored. If you have not done so already, start by making a new folder in your storage area for the course. Then copy the exercise to the folder and enter into the folder. You can write ‘`ls`’ to get a file listing of the folder.

```
cd erda_mount
mkdir HPPC
cd HPPC
cp -a ~/hpc_course/module2 .
cd module2
ls
```

To be able to edit the files for the exercise navigate to the same folder in the file view. Here you can open four files:

- Makefile
- Water_sequential.cpp
- Water_vectorised.cpp
- Water_visualizer.ipynb

Before you can run the code, you need to compile it. This can be done with make. You should see something like this in the terminal:

```
$ make
g++ Water_sequential.cpp -O1 -Wall -march=native -g -std=c++17 -o seq
g++ Water_vectorised.cpp -O1 -Wall -march=native -g -std=c++17 -fopenmp-simd -o vec
$
```

Two binaries are produced: `seq` and `vec`. `seq` is the binary of the reference code, while `vec` is the binary of your code. To begin with the two C++ source code files are very similar.

You can use the `seq` binary file to run a simulation. In the command line you can change some default system configurations. Use `seq -h` to get a terse line with the command options:

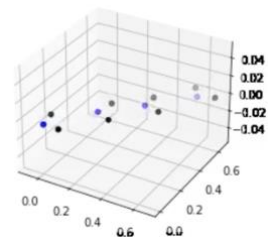
```
$ seq -h
MD -steps <number of steps> -no_mol <number of molecules> -fwrite <io frequency> -dt
<size of timestep> -ofile <filename>
```

Try run the command. It should be very quick and produce something like:

```
./seq
Accumulated forces Bonds      : 3.4061e+08
Accumulated forces Angles    : 2.6864e+08
Accumulated forces Non-bond: 5.7982e+09
Elapsed total time:          1.7036
```

The default values of the input parameters are defined in the “Sim_Configuration” class. Please read the source. The output tells us how long time it took to execute total and three checksums. The checksums pertain to each part of the force update, and is a good cross check for errors. If the checksums are not the same for the `seq` and the `vec` binary, something has gone awry.

When the simulation is finished, you can use the python notebook to visualize the trajectory file “trajectory.txt” and check the system evolution. The movie will show frames like the one shown on the right with the 3D position of the molecules (and atoms): Initial system is a bunch of water on a sphere which will fall in together, almost like a collapsing air bubble, and some small droplets will then be slung off.



Code structure:

The main function reads all the common line options and pass them to a function on the `Sim_Configuration` class to update the input parameters. Then in the main function, a system containing a set of water particles is returned by the `MakeWater` function. The configuration of this system is evolved by `Evolve` function. Every `data_period` time steps, the position of the atoms is stored, they can later be read by notebook.

Objects in the system

The code is built with a class-oriented Array-of-Structures layout corresponding to the different physical entities that are relevant for the problem. The **System** class contains a vector of molecules and the global time. Then **Molecule** class contains vectors of atoms, bonds, angles and the neighbours.

The **Bonds** and **Angles** classes contain the constants relevant for the bonds and angles with respect to the atoms in the molecules. The **Atom** class contains positions, velocities, forces, name, charge and LJ potential coefficients. Finally, **neighbours** are a list of the indexes of up to nClosest neighbors with which we will calculate intermolecular forces. To evolve the atoms forward in time we implement the leap-frog integrator in the **UpdateKDK** function. To calculate the forces and update the neighbours, we use four functions **UpdateBondForces**, **UpdateAngleForces**, **UpdateNonBondedForces** and every 100 timesteps, **BuildNeighborList**.

Task 1: Create a Struct-of-Arrays version of the program (amenable to vectorization) [Points: 4]

The sequential version of the program uses a data layout with the data for each atom collected in to a structure. This has two advantages: i) the code is simple, compact, and expressive. ii) different data describing a single body are close to each other in memory, resulting in optimal cache usage when handling one “object” at a time. This layout is called “array-of-structs” or AoS. The drawback is that when looping over elements of different atoms (as in intermolecular forces) the data of interest is interleaved on the cache lines of the other info, wasting precious memory space and throughput. In a language like python this would be much worse, since each element would be allocated separately scattered around memory. To optimize the code and make it amenable for execution with SIMD units (or on a GPU or other data parallel accelerators), the data has instead to be laid out in an orthogonal format, called “struct-of-arrays” or SoA.

In the vector version of the code, `Water_vectorised.cpp`, that is ready for OpenMP SIMD parallelisation, the `Atom` class has been changed in to an `Atoms` class containing multiple atoms / vectors to follow the SoA data layout, and the `Molecule` class has become a `Molecules` class containing `no_mol` identical molecules.

Bonus: You can take it even further and store each component of the `Vec3`’s separately to further ease vectorization; feel free to explore this for the very most optimal performance.

To accomplish the task you will have to restructure the loops calculating the forces such that the innermost loop is iterating over all the identical atoms in the system (compared to the original version where the outer loop was iterating over all the molecules). You also have to make small changes to the initialization of the water molecules, to the output routine and to the leap-frog integrator loop. Finally, in the `System` class instead of using a vector of `Molecule`:

```
std::vector<Molecule> molecules;           // all the molecules in the system
```

you can now use a single copy of the new `Molecules` class:

```
Molecules molecules;                       // all the molecules in the system
```

Typically, a loop that before looked like (from the leap-frog update):

```
for (Molecule& molecule : sys.molecules)
    for (auto& atom : molecule.atoms){
```

will now become

```
Molecules& molecule = sys.molecules;
for (auto& atom : molecule.atoms)
    for (int i = 0; i < molecule.no_mol; i++){
```

Notice how we used a reference to `sys.molecules` to keep the variable names the same.

When your code is complete, you should be able to remove the `Atom` and `Molecule` classes and obtain the same results and checksums with the “seq” and “vec” executables. Verify!

To complete this task, you have to do two things:

- **Submit your code both as source as well as in pdf**
- **Create and upload an up to three-minute video done by one group member, who explains the code (e.g. briefly what it does in different part of the source code), and show that your code gives the correct checksum.**

The TA has to look through ≈ 20 of these videos, and it is therefore important that you keep the 3 minute limit, to make it feasible.

Task 2: Investigate the performance of the code with a profiler [Points: 4]

Optimising a code too early can result in a lot of lost time. A powerful tool for investigating how well a code performs is a profiler. The alternative to a profiler is to include timings in the code. You can already see how to do this with the chronos library in the main routine.

In this exercise you will use the simple text-based profiler *gprof* that is available on ERDA, and indeed on most Linux machines. *gprof* requires the code to be compiled with a certain flag “-pg”.

Open the Makefile in a text editor and you can see that there already are three suggested combinations of compile flags specified with the variable `OPT`. `make` will always use the last definition in the Makefile. Edit the Makefile to select the flag combination for profiling, remove the binaries and recompile. This should result in a terminal output like this:

```
jovyan@...$ make clean
rm -fr seq vec
jovyan@...$ make
g++ Water_sequential.cpp -O1 -pg -Wall -march=native -g -std=c++17 -o seq
g++ Water_vectorised.cpp -O1 -pg -Wall -march=native -g -std=c++17 -fopenmp-simd -o
vec
jovyan@...$
```

To get an idea of the basics of profiling spend 5 minutes reading through:

<https://www.thegeekstuff.com/2012/08/gprof-tutorial/>

After reading, try executing the new binary doing a simulation with four molecules for a large enough number of steps that the full execution takes around a second, and then try get a table of the most demanding functions using the command “`gprof -p -b ./seq gmon.out`”. This should result in an output similar this:

```
jovyan@2f2b00cb86e1:~/erda_mount/Teaching/2022 HPPC/hpc_course/module2$ ./seq -steps 100000 -no_mol 4 -fwrite 1000000
Elapsed time: 0.8439
Accumulated forces Bonds : 1.2924e+08
Accumulated forces Angles : 1.7177e+08
Accumulated forces Non-bond: 8.5669e+08
jovyan@2f2b00cb86e1:~/erda_mount/Teaching/2022 HPPC/hpc_course/module2$ gprof -p -b ./seq gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self           self      total
time  seconds    seconds   calls  us/call  us/call   name
68.36    0.28     0.28    100000    2.80    3.27  UpdateNonBondedForces(System&)
14.65    0.34     0.06  14000000    0.00    0.00  operator*(double, Vec3 const&)
 7.32    0.37     0.03    100000    0.30    0.33  UpdateBondForces(System&)
 4.88    0.39     0.02    100000    0.20    0.30  UpdateAngleForces(System&)
 2.44    0.40     0.01  1200000    0.01    0.01  cross(Vec3 const&, Vec3 const&)
 2.44    0.41     0.01    100000    0.10    4.10  Evolve(System&, Sim_Configuration&)
 0.00    0.41     0.00  4000000    0.00    0.00  dot(Vec3 const&, Vec3 const&)
 0.00    0.41     0.00    42         0.00    0.00  void std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char>>::__M_construct<char*>(char*, char*, std::forward_iterator_tag)
 0.00    0.41     0.00    11         0.00    0.00  void std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char>>::__M_construct<char const*>(char const*, char const*, std::forward_iterator_tag)
 0.00    0.41     0.00    3         0.00    0.00  int __gnu_cxx::__stoa<long, int, char, int>(long (*)(char
const*, char**, int), char const*, char const*, unsigned long*, int)
 0.00    0.41     0.00    3         0.00    0.00  void std::vector<Molecule, std::allocator<Molecule>
>::__M_realloc_insert<Molecule>(__gnu_cxx::__normal_iterator<Molecule*,
std::vector<Molecule>>, Molecule&&)
 0.00    0.41     0.00    2         0.00    0.00  WriteOutput(System&, std::basic_ofstream<char, std::char_traits<char>
>&)
 0.00    0.41     0.00    1         0.00    0.00  _GLOBAL_sub_I_accumulated_forces_bond
 0.00    0.41     0.00    1         0.00    0.00  MakeWater(int)
```

```

0.00      0.41      0.00      1      0.00      0.00      0.00      0.00
Sim_Configuration::Sim_Configuration(std::vector<std::__cxx11::basic_string<char,
std::allocator<char> >, std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
> > >)
0.00      0.41      0.00      1      0.00      0.00      0.00      0.00
std::vector<Molecule, std::allocator<Molecule> >::~vector()
0.00      0.41      0.00      1      0.00      0.00      0.00      0.00
std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>,
std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
> > >::vector<char**, void>(char**, char**, std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > > const&)
0.00      0.41      0.00      1      0.00      0.00      0.00      0.00
std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>,
std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
> > >::~vector()

```

It is clear that the bulk of the time is used by the forcing functions and the vector operators. Evolve also takes some time because it does the leap-frog integration loop. The exact amount of time each function takes of course depends on the implementation, but also on the chosen parameters.

To get a better idea of the real performance of the code, please select the optimal compilation options “OPT=-O3 -ffast-math”, but add the “-pg” flag. This will dramatically improve the runtime (by x10), because “-O3” tries to inline (reimplement directly in the body of functions) the vector operations and because “-ffast-math” removes a lot of checks for e.g. sqrt of negative numbers, divide by zero etc. Now use this to investigate how the performance changes.

Questions: (all using the optimal settings OPT=-O3 -ffast-math -pg)

- **Using 100 molecules, which functions (and with which percentage) contributes most to the runtime**
- **How does it change if modelling 10 molecules, 100 molecules, 1000 and 40000 molecules. Explain why the percentages change. Think about your choice of number of timesteps to balance good statistics and reasonable runtime.**
- **Based on the above results, which function(s) are most important to get good performance?**
- **How does the vectorised (SoA) version perform compared to the original sequential (AoS) version for each of the system sizes? Discuss the observations; both vectorization, reuse of data in cache (L1, L2, and L3), and overall memory usage affect your results.**

Task 3: Adding OpenMP SIMD pragmas to the code [Points: 2]

During the exercise you have developed a good idea about which are the main loops, in terms of performance, *in your vectorized code*. Add OpenMP SIMD pragmas to them. In the current code the performance should not change much (though this will depend on the exact implementation), but it is still good practice to add them. There are four loop blocks where it is relevant to add pragmas. Verify that you still get the right checksums when recompiling with “OPT=-O3 -ffast-math”. The highest impact should be seen in runs with many molecules.

- **Write a few lines in the report about where you have placed the pragmas, the benchmark results using 16 molecules and why you chose your pragmas as you did.**
- **Do a quick timing test, seeing whether each of your pragmas improve performance**