Assignment 0

High Performance Parallel Computing 2022

Kimi Cardoso Kreilgaard & Linea Stausbøll Hedemark

January 2022

1 Task 1: Create a Struct-of-Arrays version of the program

Following the examples given in the assignment, we restructured the code to a SoA type. The general logic in doing this was changing the loops in the three functions, `UpdateBondForces`, `UpdateAngleForces`, and `UpdateNonBondedForces`, such that the innermost loop in each function was the loop with the most iterations. In `UpdateNonBondedForces` we had to think more abstract and utilised the fact that the molecules have a lot of identical properties, and made it such that the function loops over types of atoms instead of all molecules. We also had to reformulate the latter part of the code, so that they would work with the updated functions. We made changes to the functions: `System`, `UpdateBondForces`, `UpdateAngleForces`, `UpdateNonBondedForces`, `Evolve`, `MakeWater` and `WriteOutput`. Notice that the output file created is now in the format: `no_mol` lines with O-atoms, `no_mol` lines with H1-atoms and `no_mol` lines with H2-atoms, instead of `no_mol` set of three lines describing O, H1 and H2 for each molecule. The Python script used for visualising is invariant about the order of the output file, so this is fine. As a final test we removed the singular Atom and Molecule classes, and managed to make the code run successfully.

2 Task 2: Investigate the performance of the code with a profiler

The results of our gprof analysis is seen in table 1. For four molecules it's clearly the `UpdateNonBondedForces` that takes up most of the runtime. As we increase the number of molecules the percentage of time it takes to run `UpdateNonBondedForces` increases noticeably, and it's due to this functions runtime increasing as n^2 whereas the others increase linearly as n^1. It is therefor clear that to effectively increase performance, one should focus on improving the `UpdateNonBondedForces` part of the code.

To compare the SoA and AoS setups we used the same number of steps for the two algorithms (though different for the number of molecules, to avoid it taking forever for 128 molecules). For two molecules (2000000 steps) we get the runtime 1.315 s for the vectorised code and 1.21 s for the sequential code. For 128 molecules (100000 steps) we get 108.4 seconds for both codes. So there doesn't seem to be a difference in their performances, even though we expected the vectorised version to be better than the sequential.

No. of mol	`UpdateBondForces`	`UpdateAngleForces`	`UpdateNonBondedForces`	`Evolve`	Total time
2	22.07%	19.13%	47.09%	11.77%	1.194 s
4	8.88%	11.22%	76.22%	3.74%	3.28 s
16	2.66%	2.75%	93.45%	1.17%	37.95 s
128	0.39 %	0.57%	98.91 %	0.19%	2169 s

Table 1: Percentage of run time for each function as given by gprof. Values are for $2 \cdot 10^6$ time steps.

3 Task 3: Adding OpenMP SIMD pragmas to the code

The most important function was `UpdateNonBondedForces` so we inserted our first pragma there, using the command line `pragma omp simd` with no clause, which let's the compiler decide the best way to do it. Because the assignment asked for four pragmas we also inserted pragmas in each of the functions mentioned in table 1, since these were the most time consuming. For 16 molecules it improved the runtime with 1.4%. This seemed low, so we tested for 20 molecules, but still got a very modest improvement.

4 The Code

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <cassert>
#include <math.h>
#include <chrono>

/* Define constant pi/180 for changing degs to radians */
const double deg2rad = acos(-1)/180.0;

/* Ininialising accumulated values at zero */
double accumulated_forces_bond = 0.; // Checksum: accumulated size of forces
double accumulated_forces_angle = 0.; // Checksum: accumulated size of forces
double accumulated_forces_non_bond = 0.; // Checksum: accumulated size of forces



// ===========================================================
// Define a class for handling math operations of 3D vectors
// ===========================================================

class Vec3 {

// Define access-specifier: members are accessible from outside the class
public:

    // The attributes accecible will be the three components
    double x, y, z;

    // Initialization of vector
    Vec3(double x, double y, double z): x(x), y(y), z(z) {}

    // Define various vector operations
    double mag() const{
        return sqrt(x*x+y*y+z*z);
    }

    Vec3 operator-(const Vec3& other) const{
        return {x - other.x, y - other.y, z - other.z};
    }

    Vec3 operator+(const Vec3& other) const{
        return {x + other.x, y + other.y, z + other.z};
    }

    Vec3 operator*(double scalar) const{
        return {scalar*x, scalar*y, scalar*z};
    }

    Vec3 operator/(double scalar) const{
        return {x/scalar, y/scalar, z/scalar};
    }

    Vec3& operator+=(const Vec3& other){
        x += other.x; y += other.y; z += other.z;
        return *this;
    }

    Vec3& operator-=(const Vec3& other){
        x -= other.x; y -= other.y; z -= other.z;
        return *this;
    }

    Vec3& operator*=(double scalar){
        x *= scalar; y *= scalar; z *= scalar;
        return *this;
    }

    Vec3& operator/=(double scalar){
        x /= scalar; y /= scalar; z /= scalar;
        return *this;
    }
};


Vec3 operator*(double scalar, const Vec3& y){
    return y*scalar;
}

Vec3 cross(const Vec3& a, const Vec3& b){
    return { a.y*b.z-a.z*b.y,
             a.z*b.x-a.x*b.z,
             a.x*b.y-a.y*b.x };
}

double dot(const Vec3& a, const Vec3& b){
    return a.x * b.x + a.y * b.y + a.z * b.z;
}


// ===============================================================
// Define classes describing different physical aspects
// ===============================================================

/* A class for the bond between two atoms U = 0.5k(r12-L0)^2 */
```

```
class Bond {
public:
    double K; // force constant
    double L0; // relaxed length
    int a1, a2; // the indexes of the atoms at either end
};

/* A class for the angle between three atoms U=0.5K(phi123-phi0)^2 */
class Angle {
public:
    double K;
    double Phi0;
    int a1, a2, a3; // the indexes of the three atoms, with a2 being the centre atom
};

/* atom class, represent N instances of identical atoms */
class Atoms {
public:
    // The mass of the atom in (U)
    double mass;
    double ep; // epsilon for LJ potential
    double sigma; // Sigma, somehow the size of the atom
    double charge; // charge of the atom (partial charge)
    std::string name; // Name of the atom
    // the position in (nm), velocity (nm/ps) and forces (k_BT/nm) of the atom
    std::vector<Vec3> p,v,f;
    // constructor, takes parameters and allocates p, v and f properly to have N_identical elements
    Atoms(double mass, double ep, double sigma, double charge, std::string name, size_t N_identical)
    : mass{mass}, ep{ep}, sigma{sigma}, charge{charge}, name{name},
      p{N_identical, {0,0,0}}, v{N_identical, {0,0,0}}, f{N_identical, {0,0,0}}
    {}
};

/* molecule class for no_mol identical molecules */
class Molecules {
public:
    std::vector<Atoms> atoms; // list of atoms in the N identical molecule
    std::vector<Bond> bonds; // the bond potentials, eg for water the left and right bonds
    std::vector<Angle> angles; // the angle potentials, for water just the single one, but keep it a list for generality
    int no_mol;
};



// ===================================
// Define classes handling the simulation. This is where we have made changes.
// ===================================

/* system class */
class System {
```

```
public:
    Molecules molecules; // all the molecules in the system
    double time = 0; // current simulation time
};

class Sim_Configuration {
public:
    int steps = 10000; // number of steps
    int no_mol = 4; // number of molecules
    double dt = 0.0005; // integrator time step
    int data_period = 100; // how often to save coordinate to trajectory
    std::string filename = "trajectory.txt"; // name of the output file with trajectory
    // system box size. for this code these values are only used for vmd, but in general md codes, period boundary
        ↪ conditions exist

    // simulation configurations: number of step, number of the molecules in the system,
    // IO frequency, time step and file name
    Sim_Configuration(std::vector <std::string> argument){
        for (long unsigned int i = 1; i<argument.size() ; i += 2){
            std::string arg = argument.at(i);
            if(arg=="-h"){ // Write help
                std::cout << "MD -steps <number_of_steps> -no_mol <number_of_molecules>"
                          << " -fwrite <io_frequency> -dt <size_of_timestep> -ofile <filename> \n";
                exit(0);
                break;
            } else if(arg=="-steps"){
                steps = std::stoi(argument[i+1]);
            } else if(arg=="-no_mol"){
                no_mol = std::stoi(argument[i+1]);
            } else if(arg=="-fwrite"){
                data_period = std::stoi(argument[i+1]);
            } else if(arg=="-dt"){
                dt = std::stof(argument[i+1]);
            } else if(arg=="-ofile"){
                filename = argument[i+1];
            } else{
                std::cout << "---> error: the argument type is not recognized \n";
            }
        }

        dt /= 1.57350; /// convert to ps based on having energy in k_BT, and length in nm
    }
};


// Given a bond, updates the force on all atoms correspondingly
void UpdateBondForces(System& sys){
    Molecules& molecules = sys.molecules;
    // Loops over the (2 for water) bond constraints
    for (Bond& bond : molecules.bonds){
        auto& atom1=molecules.atoms[bond.a1];
        auto& atom2=molecules.atoms[bond.a2];

        // Loop over the molecules indices
        #pragma omp simd
        for (int i = 0;i<molecules.no_mol;i++) {
            Vec3 dp = atom1.p.at(i)-atom2.p.at(i);
            Vec3 f = -bond.K*(1-bond.L0/dp.mag())*dp;
            atom1.f.at(i) += f;
            atom2.f.at(i) -= f;
            accumulated_forces_bond += f.mag();
        }
    }
}

/* Iterates over all bonds in molecules (for water only 2: the left and right).
And updates forces on atoms correpondingly */
void UpdateAngleForces(System& sys){
    Molecules& molecule = sys.molecules;

    // Loop over the angles
    for (Angle& angle : molecule.angles){
        //==== angle forces (H--O---H bonds) U_angle = 0.5*k_a(phi-phi_0)^2
        //f_H1 = K(phi-ph0)/|H1O|*Ta
        // f_H2 = K(phi-ph0)/|H2O|*Tc
        // f_O = -f1 - f2
        // Ta = norm(H1O x (H1O x H2O))
        // Tc = norm(H2O x (H2O x H1O))
        //=============================================
        auto& atom1=molecule.atoms[angle.a1];
        auto& atom2=molecule.atoms[angle.a2];
        auto& atom3=molecule.atoms[angle.a3];

        // Loop over molecules indices
        #pragma omp simd
        for (int i = 0; i < molecule.no_mol; i++){
            Vec3 d21 = atom2.p.at(i)-atom1.p.at(i);
            Vec3 d23 = atom2.p.at(i)-atom3.p.at(i);

            // phi = d21 dot d23 / |d21| |d23|
            double norm_d21 = d21.mag();
            double norm_d23 = d23.mag();
            double phi = acos(dot(d21, d23) / (norm_d21*norm_d23));

            // d21 cross (d21 cross d23)
            Vec3 c21_23 = cross(d21, d23);
            Vec3 Ta = cross(d21, c21_23);
            Ta /= Ta.mag();

            // d23 cross (d23 cross d21) = - d23 cross (d21 cross d23) = c21_23 cross d23
            Vec3 Tc = cross(c21_23, d23);
            Tc /= Tc.mag();

            Vec3 f1 = Ta*(angle.K*(phi-angle.Phi0)/norm_d21);
            Vec3 f3 = Tc*(angle.K*(phi-angle.Phi0)/norm_d23);

            atom1.f.at(i) += f1;
            atom2.f.at(i) -= f1+f3;
            atom3.f.at(i) += f3;

            accumulated_forces_angle += f1.mag() + f3.mag();
        }
    }
}

// Iterates over all atoms in both molecules
// And updates forces on atoms correpondingly
void UpdateNonBondedForces(System& sys){
    /* nonbonded forces: only a force between atoms in different molecules
       The total non-bonded forces come from Lennard Jones (LJ) and coulomb interactions
       U = ep[(sigma/r)^12-(sigma/r)^6] + C*q1*q2/r */

    Molecules& molecule = sys.molecules;

    // Loop over types of atoms
    for (auto& atom1 : molecule.atoms)
        for (auto& atom2 : molecule.atoms)

            // Loop over molecules indices (only looking forward)
            #pragma omp simd
            for (int i = 0; i < molecule.no_mol; i++)
                for (int j = i+1; j < molecule.no_mol; j++){

                Vec3 dp = atom1.p.at(i)-atom2.p.at(j);

                double r = dp.mag();
                double r2 = r*r;
                double ep = sqrt(atom1.ep*atom2.ep); // ep = sqrt(ep1*ep2)
                double sigma = 0.5*(atom1.sigma+atom2.sigma); // sigma = (sigma1+sigma2)/2
                double q1 = atom1.charge;
                double q2 = atom2.charge;

                double sir = sigma*sigma/r2; // crossection**2 times inverse squared distance
                double KC = 80*0.7; // Coulomb prefactor
                Vec3 f = ep*(12*pow(sir,6)-6*pow(sir,3))*sir*dp + KC*q1*q2/(r*r2)*dp; // LJ + Coulomb forces
                atom1.f.at(i) += f;
                atom2.f.at(j) −= f;

                accumulated_forces_non_bond += f.mag();
                }

}

// integrating the system for one time step using Leapfrog symplectic integration
void Evolve(System &sys, Sim_Configuration &sc){

    Molecules& molecule = sys.molecules;

    // Loop over types of atoms
    for (auto& atom : molecule.atoms)

        // Loop over molecule indices
        #pragma omp simd
        for (int i=0; i < molecule.no_mol; i++) {

            atom.v.at(i) += sc.dt/atom.mass∗atom.f.at(i); // Update the velocities
            atom.f.at(i) = {0,0,0}; // set the forces zero to prepare for next potential calculation
            atom.p.at(i) += sc.dt∗ atom.v.at(i);
        }

    // Update the forces on each particle based on the particles positions
    // Calculate the intermolecular forces in all molecules
    UpdateBondForces(sys);
    UpdateAngleForces(sys);
    // Calculate the intramolecular LJ and Coulomb potential forces between all molecules
    UpdateNonBondedForces(sys);

    sys.time += sc.dt; // update time
}

// Setup one water molecule
System MakeWater(int N_molecules){
    //===========================================================
    // creating water molecules at position X0,Y0,Z0. 3 atoms
    // H−−−O−−−H
    // The angle is 104.45 degrees and bond length is 0.09584 nm
    //===========================================================
    // mass units of dalton
    // initial velocity and force is set to zero for all the atoms by the constructor
    const double L0 = 0.09584;
    const double angle = 104.45∗deg2rad;

    // mass ep sigma charge name
    Atoms Oatom(16, 0.65, 0.31, −0.82, ”O”, N_molecules); // Oxygen atom
    Atoms Hatom1( 1, 0.18828, 0.238, 0.41, ”H”, N_molecules); // Hydrogen atom
    Atoms Hatom2( 1, 0.18828, 0.238, 0.41, "H", N_molecules); // Hydrogen atom

    // bonds beetween first H-O and second H-O respectively
    std::vector<Bond> waterbonds = {
        { .K = 20000, .L0 = L0, .a1 = 0, .a2 = 1},
        { .K = 20000, .L0 = L0, .a1 = 0, .a2 = 2}
    };

    // angle between H-O-H
    std::vector<Angle> waterangle = {
        { .K = 1000, .Phi0 = angle, .a1 = 1, .a2 = 0, .a3 = 2 }
    };

    System sys;

    // Loop over molecule indices
    for (int i = 0; i < N_molecules; i++){
        Vec3 P0{i * 0.2, i * 0.2, 0};
        Oatom.p.at(i) = {P0.x, P0.y, P0.z};
        Hatom1.p.at(i) = {P0.x+L0*sin(angle/2), P0.y+L0*cos(angle/2), P0.z};
        Hatom2.p.at(i) = {P0.x-L0*sin(angle/2), P0.y+L0*cos(angle/2), P0.z};

        std::vector<Atoms> atoms {Oatom, Hatom1, Hatom2};

        // Declare atoms, bonds, waterbonds, waterangle, N_molecules, i.e. class version of pushback
        sys.molecules.atoms = atoms;
        sys.molecules.bonds = waterbonds;
        sys.molecules.angles = waterangle;
        sys.molecules.no_mol = N_molecules;
    }

    // Store atoms, bonds and angles in Water class and return
    return sys;
}

// Write the system configurations in the trajectory file.
void WriteOutput(System& sys, std::ofstream& file){

    // Loop over atoms, then over molecules
    Molecules& molecule = sys.molecules;
    for (auto& atom: molecule.atoms)
        for (int i = 0; i < molecule.no_mol; i++) {

            // Write positions and others stuff to file. Order doesn't matter for the Python script
            file << sys.time << " "
                 << atom.name << " "
                 << atom.p.at(i).x << " "
                 << atom.p.at(i).y << " "
                 << atom.p.at(i).z << '\n';
        }
}


//==========================================
//============= Main function ==============
//==========================================
int main(int argc, char* argv[]){
    Sim_Configuration sc({argv, argv+argc}); // Load the system configuration from command line data

    System sys = MakeWater(sc.no_mol); // this will create a system containing sc.no_mol water molecules
    std::ofstream file(sc.filename); // open file

    WriteOutput(sys, file); // writing the initial configuration in the trajectory file

    auto tstart = std::chrono::high_resolution_clock::now(); // start time (nano-seconds)

    // Molecular dynamics simulation
    for (int step = 0;step<sc.steps ; step++){

        Evolve(sys, sc); // evolving the system by one step
        if (step % sc.data_period == 0){
            //writing the configuration in the trajectory file
            WriteOutput(sys, file);
        }
    }

    auto tend = std::chrono::high_resolution_clock::now(); // end time (nano-seconds)

    std::cout << "Elapsed time:" << std::setw(9) << std::setprecision(4)
              << (tend - tstart).count()*1e-9 << "\n";
    std::cout << "Accumulated forces Bonds   : " << std::setw(9) << std::setprecision(5)
              << accumulated_forces_bond << "\n";
    std::cout << "Accumulated forces Angles  : " << std::setw(9) << std::setprecision(5)
              << accumulated_forces_angle << "\n";
    std::cout << "Accumulated forces Non-bond: " << std::setw(9) << std::setprecision(5)
              << accumulated_forces_non_bond << "\n";
}
```