Assignment 2

Jonas Ølshøj Pedersen (wlq622), Mikkel Møller Mødekjær (zkm716)
and Thomas Bukholt Hansen (lkf436)

Sunday 19th February, 2023

Task 1

We present a program that simulates the interactions of water molecules, in a structure-of-arrays (SoA) version, instead of array-of-structures (AoS) version we were handed, in order for the code to be vectorized instead of running sequentially. We were therefore given two new classes (SoA) to replace the old classes (AoS), which we would then incorporate. This vectorized version of the code is presented in appendix A. We have used Jonas' code.

Task 2

We will now test the newly created vectorized vesrion with settings OPT=-O3 -ffast-math -pg. This just means that we run a compiler with 4rd level optimization, using ffast-math, and also allow us to use the *gprof* profiler on our executables. All tests are done using 100000 time-steps.
We first ran the code for 4 water molecules. After profiling, we found that the function with the highest impact on performance was the class UpdateNonBondedForces, using up 80% of the total runtime. This makes sense, since this function has to run roughly $\mathcal{O}(N^2)$ flops where others only run $\mathcal{O}(N)$ (with N being the number of particles), owing to the fact that the non-bonded forces is calculated between every atom in the system. The other noteworthy functions were UpdateBondForces, UpdateAngleForces and Evolve, which each used up 6.67% of the total runtime. All other functions had negligible impact.
We next ran the program with 2, 16 and 128 molecules, where we see that UpdateNonBondedForces uses a higher percent of run time for higher numbers of molecules as shown below:

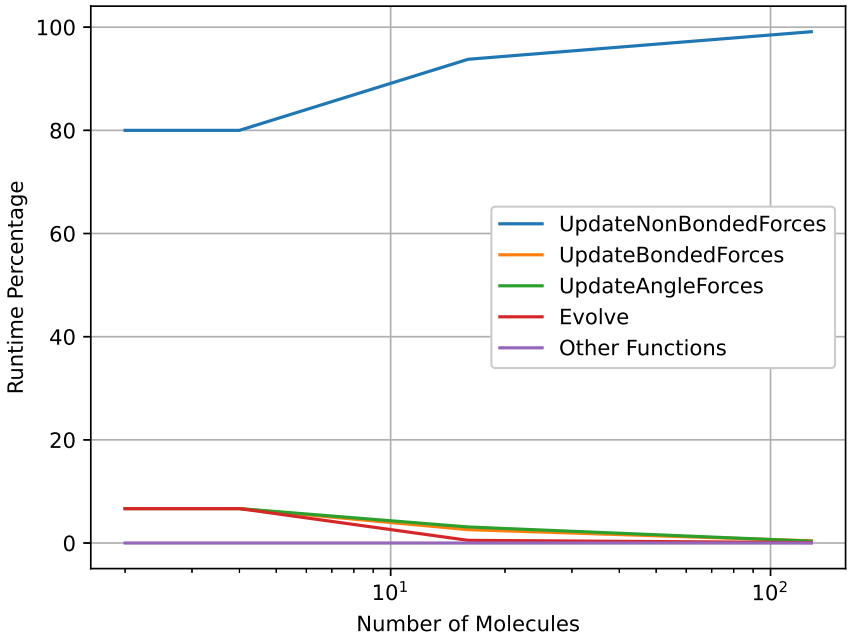

Figure 1: Plot of the runtime percentage for different functions of the program, as a function of the number of molecules in the system. We see that the class UpdateNonBondedForces (blue) has the highest runtime percentage that increases with the number of molecules, while the other functions decrease in relative runtime.

This increase in relative runtime is to be expected, for the same reasons mentioned before about the amount of flops being of order $\mathcal{O}(N^2)$ (N again being the number of molecules). All other functions use up less and less of the total performance, as UpdateNonBondedForces takes up more and more space in the computations.
We can therefore with confidence say, that the most important function to optimize would be UpdateNonBondedForces, and the importance only scales with the system size. The other functions in Figure 1, could also be optimized a bit, but is not as high a priority.
When comparing performance of the program written in AoS versus SoA, the effect is summarized in Table 1:

	2 molecules	128 molecules
AoS	0.07262	138.9
SoA	0.08117	123.8

Table 1: Table showing the runtime in seconds of the program using structure-of-arrays (SoA) and array-of-structures (AoS) for 2 and 128 molecules. We see that SoA is faster for 128 particles but a bit slower for 2 particles than AoS.

We see that SoA is faster for many water molecules but a bit slower for a small number, reflecting the strengths and weaknesses of this version.

Task 3

Lastly, we added five OpenMP SIMD pragmas to certain parts of the code, in order to further parallelize the program. Tests were all done using 64 molecules and 10000 timesteps. These pragmas have been added to for loops in the functions Evolve, UpdateBondForces, UpdateAngleForces, UpdateNonBondForces and

MakeWater. The first four functions are called each update step, where the first three of those contain only one for loop, looping over molecules, which should be able to be parallelized. In the fourth function there are two nested loops that are parallelizable, where we chose to parallelize the outermost loop. As explained in task 2, this is the most important function to optimize. Contrary to the other functions, MakeWater only runs once as it initializes the system, but in that function we have a loop which we can parallelize for small "performance" gains. These loops were chosen such that we could make use of as many threads as possible. These for loops should also be parallelizable without a race condition.

All of the pragmas and the loops they affect can be seen in the code snippets shown underneath. Pragmas are added by writing "#pragma omp for simd" above the loop. This does not however parallelize the code, as it will only run on one thread, due to a problem in the structure of the Atoms class. However, using a non SIMD OpenMP, we got it to run on multiple threads, where, on 32 threads, the runtime would increase by a factor of 8-9, with more threads only making the code run even slower. We did this by using the compiler flag "-fopenmp" instead of "-fopenmp-simd", then used "omp_set_num_threads(32)" and lastly the pragma "#pragma omp parallel for".

For the SIMD we have verified that the code does not run on multiple cores using the process viewer *htop*. We also confirmed that the checksums do not change. We experimented with a bunch of different pragmas while trying to parallelize the code, but none of the tested options improved the runtime while using SIMD. In conclusion, we did not succeed in parallelizing the code as much as we wanted, but this should mostly be because of the aforementioned structural problem in the Atoms class. We would suspect that if this was fixed, we would be able to parallelize the code using the pragmas below.

Listing 1: Evolve

```
1       for (auto& atom : molecule.atoms)
2       #pragma omp for simd
3       for (long unsigned int i = 0;   i < molecule.no_mol; i++){
```

Listing 2: UpdateBondForces

```
1       #pragma omp for simd
2       for (long unsigned int i = 0; i < molecule.no_mol; i++)
3       for (Bond& bond : molecule.bonds){
```

Listing 3: UpdateAngleForces

```
1       #pragma omp for simd
2       for (long unsigned int i = 0; i < molecule.no_mol; i++)
3       for (Angle& angle : molecule.angles){
```

Listing 4: UpdateNonBondForces

```
1       for (auto& atoms1 : sys.molecules.atoms)
2       for (auto& atoms2 : sys.molecules.atoms) // iterate over all pairs of atoms, similar as
            well as dissimilar
3       #pragma omp for simd
4       for (long unsigned int i = 0;   i < sys.molecules.no_mol; i++)
5       for (long unsigned int j = i+1; j < sys.molecules.no_mol; j++){
```

Listing 5: MakeWater

```
1       System sys;
2       #pragma omp for simd
3       for (long unsigned int i = 0; i < N_molecules; i++){
```

Appendix

A Code

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

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

class Vec3 {
public:
    double x, y, z;
    // initialization of vector
    Vec3(double x, double y, double z): x(x), y(y), z(z) {}
    // size of vector
    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;
}

```

```
64  /* a class for the bond between two atoms U = 0.5k(r12-L0)^2 */
65  class Bond {
66  public:
67      double K;    // force constant
68      double L0;   // relaxed length
69      int a1, a2;  // the indexes of the atoms at either end
70  };
71
72  /* a class for the angle between three atoms  U=0.5K(phi123-phi0)^2 */
73  class Angle {
74  public:
75      double K;
76      double Phi0;
77      int a1, a2, a3; // the indexes of the three atoms, with a2 being the centre atom
78  };
79
80  // ================================================================================
81  // Two new classes arranging Atoms in a Structure-of-Array data structure
82  // ================================================================================
83
84  /* atom class, represent N instances of identical atoms */
85  class Atoms {
86  public:
87      // The mass of the atom in (U)
88      double mass;
89      double ep;            // epsilon for LJ potential
90      double sigma;         // Sigma, somehow the size of the atom
91      double charge;        // charge of the atom (partial charge)
92      std::string name;     // Name of the atom
93      // the position in (nm), velocity (nm/ps) and forces (k_BT/nm) of the atom
94      std::vector<Vec3> p,v,f;
95      // constructor, takes parameters and allocates p, v and f properly to have N_identical
            elements
96      Atoms(double mass, double ep, double sigma, double charge, std::string name, size_t
            N_identical)
97      : mass{mass}, ep{ep}, sigma{sigma}, charge{charge}, name{name},
98        p{N_identical, {0,0,0}}, v{N_identical, {0,0,0}}, f{N_identical, {0,0,0}}
99      {}
100 };
101
102 /* molecule class for no_mol identical molecules */
103 class Molecules {
104 public:
105     std::vector<Atoms> atoms;         // list of atoms in the N identical molecule
106     std::vector<Bond> bonds;          // the bond potentials, eg for water the left and
            right bonds
107     std::vector<Angle> angles;        // the angle potentials, for water just the single one
            , but keep it a list for generality
108     long unsigned int no_mol;
109 };
110
111 // ================================================================================
112
113
114 /* system class */
115 class System {
116 public:
117     Molecules molecules;          // all the molecules in the system
118     double time = 0;                          // current simulation time
119 };
120
121 class Sim_Configuration {
122 public:
123     int steps = 10000;      // number of steps
124     int no_mol = 4;         // number of molecules
125     double dt = 0.0005;     // integrator time step
126     int data_period = 100; // how often to save coordinate to trajectory
127     std::string filename = "trajectory.txt";   // name of the output file with trajectory
```

```
128     // system box size. for this code these values are only used for vmd, but in general md
            codes, period boundary conditions exist
129
130     // simulation configurations: number of step, number of the molecules in the system,
131     // IO frequency, time step and file name
132     Sim_Configuration(std::vector <std::string> argument){
133         for (long unsigned int i = 1; i<argument.size() ; i += 2){
134             std::string arg = argument.at(i);
135             if(arg=="-h"){ // Write help
136                 std::cout << "MD␣-steps␣<number␣of␣steps>␣-no_mol␣<number␣of␣molecules>"
137                           << "␣-fwrite␣<io␣frequency>␣-dt␣<size␣of␣timestep>␣-ofile␣<
                                  filename>␣\n";
138                 exit(0);
139                 break;
140             } else if(arg=="-steps"){
141                 steps = std::stoi(argument[i+1]);
142             } else if(arg=="-no_mol"){
143                 no_mol = std::stoi(argument[i+1]);
144             } else if(arg=="-fwrite"){
145                 data_period = std::stoi(argument[i+1]);
146             } else if(arg=="-dt"){
147                 dt = std::stof(argument[i+1]);
148             } else if(arg=="-ofile"){
149                 filename = argument[i+1];
150             } else{
151                 std::cout << "--->␣error:␣the␣argument␣type␣is␣not␣recognized␣\n";
152             }
153         }
154
155         dt /= 1.57350; /// convert to ps based on having energy in k_BT, and length in nm
156     }
157 };
158
159 // Given a bond, updates the force on all atoms correspondingly
160 void UpdateBondForces(System& sys){
161     Molecules& molecule = sys.molecules;
162     // Loops over the (2 for water) bond constraints
163     for (long unsigned int i = 0; i < molecule.no_mol; i++)
164     for (Bond& bond : molecule.bonds){
165         auto& atom1=molecule.atoms[bond.a1];
166         auto& atom2=molecule.atoms[bond.a2];
167
168         Vec3 dp  = atom1.p[i]-atom2.p[i];
169         Vec3 f   = -bond.K*(1-bond.L0/dp.mag())*dp;
170         atom1.f[i] += f;
171         atom2.f[i] -= f;
172         accumulated_forces_bond += f.mag();
173     }
174 }
175
176 // Iterates over all bonds in molecules (for water only 2: the left and right)
177 // And updates forces on atoms correpondingly
178 void UpdateAngleForces(System& sys){
179     Molecules& molecule = sys.molecules;
180     for (long unsigned int i = 0; i < molecule.no_mol; i++)
181     for (Angle& angle : molecule.angles){
182         //====  angle forces  (H--O---H bonds) U_angle = 0.5*k_a(phi-phi_0)^2
183         //f_H1 =  K(phi-ph0)/|H1O|*Ta
184         // f_H2 =  K(phi-ph0)/|H2O|*Tc
185         // f_O = -f1 - f2
186         // Ta = norm(H1O x (H1O x H2O))
187         // Tc = norm(H2O x (H2O x H1O))
188         //============================================================
189         auto& atom1=molecule.atoms[angle.a1];
190         auto& atom2=molecule.atoms[angle.a2];
191         auto& atom3=molecule.atoms[angle.a3];
192
193         Vec3 d21 = atom2.p[i]-atom1.p[i];
```

```
194         Vec3 d23 = atom2.p[i]-atom3.p[i];
195
196         // phi = d21 dot d23 / |d21| |d23|
197         double norm_d21 = d21.mag();
198         double norm_d23 = d23.mag();
199         double phi = acos(dot(d21, d23) / (norm_d21*norm_d23));
200
201         // d21 cross (d21 cross d23)
202         Vec3 c21_23 = cross(d21, d23);
203         Vec3 Ta = cross(d21, c21_23);
204         Ta /= Ta.mag();
205
206         // d23 cross (d23 cross d21) = - d23 cross (d21 cross d23) = c21_23 cross d23
207         Vec3 Tc = cross(c21_23, d23);
208         Tc /= Tc.mag();
209
210         Vec3 f1 = Ta*(angle.K*(phi-angle.Phi0)/norm_d21);
211         Vec3 f3 = Tc*(angle.K*(phi-angle.Phi0)/norm_d23);
212
213         atom1.f[i] += f1;
214         atom2.f[i] -= f1+f3;
215         atom3.f[i] += f3;
216
217         accumulated_forces_angle += f1.mag() + f3.mag();
218     }
219 }
220
221 // Iterates over all atoms in both molecules
222 // And updates forces on atoms correpondingly
223 void UpdateNonBondedForces(System& sys){
224     /* nonbonded forces: only a force between atoms in different molecules
225        The total non-bonded forces come from Lennard Jones (LJ) and coulomb interactions
226        U = ep[(sigma/r)^12-(sigma/r)^6] + C*q1*q2/r */
227     for (auto& atoms1 : sys.molecules.atoms)
228     for (auto& atoms2 : sys.molecules.atoms) // iterate over all pairs of atoms, similar as
        well as dissimilar
229     for (long unsigned int i = 0;   i < sys.molecules.no_mol; i++)
230     for (long unsigned int j = i+1; j < sys.molecules.no_mol; j++){
231         Vec3 dp = atoms1.p[i]-atoms2.p[j];
232
233         double r  = dp.mag();
234         double r2 = r*r;
235         double ep = sqrt(atoms1.ep*atoms2.ep); // ep = sqrt(ep1*ep2)
236         double sigma = 0.5*(atoms1.sigma+atoms2.sigma);  // sigma = (sigma1+sigma2)/2
237         double q1 = atoms1.charge;
238         double q2 = atoms2.charge;
239
240         double sir = sigma*sigma/r2; // crossection**2 times inverse squared distance
241         double KC = 80*0.7;          // Coulomb prefactor
242         Vec3 f = ep*(12*pow(sir,6)-6*pow(sir,3))*sir*dp + KC*q1*q2/(r*r2)*dp; // LJ +
            Coulomb forces
243         atoms1.f[i] += f;
244         atoms2.f[j] -= f;
245
246         accumulated_forces_non_bond += f.mag();
247     }
248
249 }
250
251 // integrating the system for one time step using Leapfrog symplectic integration
252 void Evolve(System &sys, Sim_Configuration &sc){
253
254     // Kick velocities and zero forces for next update
255     // Drift positions: Loop over molecules and atoms inside the molecules
256     Molecules& molecule = sys.molecules;
257     for (auto& atom : molecule.atoms)
258     for (long unsigned int i = 0;   i < molecule.no_mol; i++){
259         atom.v[i] += sc.dt/atom.mass*atom.f[i];    // Update the velocities
```

```
260         atom.f[i]  = {0,0,0};                   // set the forces zero to prepare for next
                potential calculation
261         atom.p[i] += sc.dt* atom.v[i];             // update position
262     }
263
264     // Update the forces on each particle based on the particles positions
265     // Calculate the intermolecular forces in all molecules
266     UpdateBondForces(sys);
267     UpdateAngleForces(sys);
268     // Calculate the intramolecular LJ and Coulomb potential forces between all molecules
269     UpdateNonBondedForces(sys);
270
271     sys.time += sc.dt; // update time
272 }
273
274 // Setup one water molecule
275 System MakeWater(long unsigned int N_molecules){
276     //=========================================================
277     // creating water molecules at position X0,Y0,Z0. 3 atoms
278     //                          H---O---H
279     // The angle is 104.45 degrees and bond length is 0.09584 nm
280     //=========================================================
281     // mass units of dalton
282     // initial velocity and force is set to zero for all the atoms by the constructor
283     const double L0 = 0.09584;
284     const double angle = 104.45*deg2rad;
285
286     //          mass    ep    sigma charge name
287     Atoms Oatoms(16, 0.65,    0.31, -0.82, "O", N_molecules);
288     Atoms Hatoms1( 1, 0.18828, 0.238, 0.41, "H", N_molecules);
289     Atoms Hatoms2( 1, 0.18828, 0.238, 0.41, "H", N_molecules);
290
291     // bonds beetween first H-O and second H-O respectively
292     std::vector<Bond> waterbonds = {
293         { .K = 20000, .L0 = L0, .a1 = 0, .a2 = 1},
294         { .K = 20000, .L0 = L0, .a1 = 0, .a2 = 2}
295     };
296
297     // angle between H-O-H
298     std::vector<Angle> waterangle = {
299         { .K = 1000, .Phi0 = angle, .a1 = 1, .a2 = 0, .a3 = 2 }
300     };
301
302     System sys;
303     for (long unsigned int i = 0; i < N_molecules; i++){
304         Vec3 P0{i * 0.2, i * 0.2, 0};
305         Oatoms.p[i]  = {P0.x, P0.y, P0.z};
306         Hatoms1.p[i] = {P0.x+L0*sin(angle/2), P0.y+L0*cos(angle/2), P0.z};
307         Hatoms2.p[i] = {P0.x-L0*sin(angle/2), P0.y+L0*cos(angle/2), P0.z};
308     }
309     std::vector<Atoms> atoms {Oatoms, Hatoms1, Hatoms2};
310
311     sys.molecules = {atoms, waterbonds, waterangle, N_molecules};
312
313     // Store atoms, bonds and angles in Water class and return
314     return sys;
315 }
316
317 // Write the system configurations in the trajectory file.
318 void WriteOutput(System& sys, std::ofstream& file){
319     // Loop over all atoms in model one molecule at a time and write out position
320     Molecules& molecule = sys.molecules;
321     for (auto& atom : molecule.atoms)
322     for (long unsigned int i = 0;   i < molecule.no_mol; i++){
323         file << sys.time << "␣" << atom.name << "␣"
324             << atom.p[i].x << "␣"
325             << atom.p[i].y << "␣"
326             << atom.p[i].z << '\n';
```

```
327     }
328 }
329
330 //
        ================================================================================================================
331 //======================== Main function
        ================================================================
332 //
        ================================================================================================================
333 int main(int argc, char* argv[]){
334     Sim_Configuration sc({argv, argv+argc}); // Load the system configuration from command
          line data
335
336     System sys  = MakeWater(sc.no_mol);   // this will create a system containing sc.no_mol
          water molecules
337     std::ofstream file(sc.filename); // open file
338
339     WriteOutput(sys, file);    // writing the initial configuration in the trajectory file
340
341     auto tstart = std::chrono::high_resolution_clock::now(); // start time (nano-seconds)
342
343     // Molecular dynamics simulation
344     for (int step = 0;step<sc.steps ; step++){
345
346         Evolve(sys, sc); // evolving the system by one step
347         if (step % sc.data_period == 0){
348             //writing the configuration in the trajectory file
349             WriteOutput(sys, file);
350         }
351     }
352
353     auto tend = std::chrono::high_resolution_clock::now(); // end time (nano-seconds)
354
355     std::cout <<  "Elapsed␣time:" << std::setw(9) << std::setprecision(4)
356               << (tend - tstart).count()*1e-9 << "\n";
357     std::cout <<  "Accumulated␣forces␣Bonds␣␣␣:␣"  << std::setw(9) << std::setprecision(5)
358               << accumulated_forces_bond << "\n";
359     std::cout <<  "Accumulated␣forces␣Angles␣␣:␣"  << std::setw(9) << std::setprecision(5)
360               << accumulated_forces_angle << "\n";
361     std::cout <<  "Accumulated␣forces␣Non-bond:␣"  << std::setw(9) << std::setprecision(5)
362               << accumulated_forces_non_bond << "\n";
363 }
```