High Performance Parallel Computing Assignment 2

António Maschio ; Dimitrios Anastasiou ; Georgios Sevastakis

February 2024

1 Objective

The objective is to optimise a given code on molecular dynamics by vectorising the code and subsequently, by using the OpenMP API (Application programming interface). A profiler was implemented to investigate the performance and pragmas were added to force vectorization the code.

2 Introduction

In this assignment, we were given a code simulating a system of N water molecules (their interactions) carried through with the leap–frog integrator. In this $N-$body simulation, the time–boost four potentials were included; For atoms belonging to the same molecule, there were the bond potential and the angle potential simulating the oscillating fashion of the bond and the angle, accordingly, while for the rest of the atoms, the Lennard–Jones (LJ) and the Coulomb potentials were taken into account.

As far as the LJ potential is concerned, the r^{-12} term models the strong repulsive forces indicated by the Pauli exclusion principle, while the r^{-6} term models the Van der Waals force. Regarding the Coulomb potential, it is interesting to note that in many MD models the charges can be less than 1. This is due to the fact that we are dealing with bounded particles that cannot exist as free particles (like quarks) or that we are dealing with quasiparticles, meaning particles that are not real but they behave like if they were.

3 Task 1: Create a Struct-of-Arrays version of the program (amenable to vectorization)

In order to transition to Struct–of–Arrays the loops were modified accordingly. For example, the below code snippet shows the four for loops of the UpdateNonBondedForces function that they now iterate through the position, velocity and force vectors instead of the different molecules. The way the data are accessed can also be seen in the last line of the snippet. The complete code can be found in Appendix A.

```
for (auto& atom1 : sys.molecules.atoms){
    for (auto& atom2 : sys.molecules.atoms){
        for (long unsigned int i = 0;   i < sys.molecules.no_mol; i++){
            for (long unsigned int j = i+1; j < sys.molecules.no_mol; j++){
                Vec3 dp = atom1.p[i]-atom2.p[j];
```

Modified UpdateNonBondedForces function in the SoA format.

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

4.1 Using 4 molecules, which functions contribute most to the runtime?

When running the simulation for 100000 steps and 4 molecules, the following profiling was obtained using the gprof profiler:

% Time	Name
83.33	UpdateNonBondedForces
11.11	UpdateAngleForces
5.56	UpdateBondForces
0.00	Evolve
0.00	Vector allocator
0.00	Write
0.00	Make Water

Table 1: Function Time Percentages for the case of 4 molecules with steps = 100000.

From this table, we can deduce that the functions that contribute the most to the runtime of this simulation are the UpdateNonBondedForces and UpdateAngleForces.

4.2 How does it change if modeling 2, 16, and 128 molecules? Explain why the percentages change.

The results obtained are summarized in the following table:

Name	2 mol.	4 mol.	16 mol.	128 mol.
UpdateNonBondedForces	100%	63.64%	94.02%	99.26%
UpdateAngleForces	-	36.36%	3.42%	0.45%
UpdateBondForces	-	-	1.71%	0.19%
Evolve	-	-	0.85%	0.09%

Table 2: Function Time percentages for 4 different simulations with steps = 100000.

We need to look at how the code works to understand why the percentages change as seen at the table above. First of all, let's have a look at the for–loops of the individual functions, since the contribute the most to the complexity of the code:

Name	Number of iterations
UpdateNonBondedForces	$A^2 \cdot \frac{M(M-1)}{2}$
UpdateAngleForces	M
UpdateBondForces	$B \cdot M$
Evolve	$A \cdot M$

Table 3: Number of iterations of each individual function, where M, A and B stand for the number of molecules, atoms and bonds respectively.

In the code, the number of atoms A and that of the bonds B remain constant and are equal to 3 and 2 respectively, meaning that all functions but the UpdateNonBondedForces run M times, while the UpdateNonBondedForces M^2. This justifies the prevalence of the UpdateNonBondedForces function for larger M values.

In the $M = 2$ and $M = 4$ cases, the number of iterations is not the most important factor contributing to the complexity. In the former one, UpdateNonBondedForces completely dominates due to the A^2 term (100% of the elapsed time), while in the latter, it prevails due to the combination of M and $A(\approx 60\%$ of the elasped time). For $M = 4$, we also notice that 1/3 of the elapsed time is consumed by the UpdateAngleForces. This is due to the main body of this particular function; it includes many function calls and declarations and that is also the reason it preserves the second place for greater $M-$values. As far as the UpdateBondForces and Evolve functions are concerned, they share the same number of iterations, but the Evolve is simpler than UpdateBondForces, resulting in the former coming last.

4.3 Which function is most important to get good performance

The most important function to be optimized is the UpdateNonBondedForces as stated in the previous subsection for the aforementioned reasons.

4.4 How does the vectorized (SoA) version perform compared to the original sequential (AoS) version running with 2 molecules and with 128 molecules. If there is a difference in the relative performance, please discuss why this could be the case.

Name	**2 mol. Seq**	**2 mol. vec**	**128 mol. seq**	**128 mol. vec**
UpdateNonBondedForces	0.03 s	0.01	162.60 s	173.28 s
UpdateAngleForces	-	0.01	0.57 s	0.72 s
UpdateBondForces	0.02 s	0.03	0.58 s	0.38 s
Evolve	0.01 s	-	0.29 s	0.15 s
Total	0.05646 s	0.05794	164.4 s	174.9 s

Table 4: Time in seconds for each simulation. steps = 100000

From the table above, we notice that the vectorized version performs a bit worse than the sequential one and Memory access patterns is to blame.

The sequential code was making good use of the cache since with one memory call more data would be stored in cache, which would subsequently be called right away. On the other hand, in the vectorised version the SoA format resulted in many cache–misses and therefore, worse performance.

5 Adding OpenMP SIMD pragmas to the code

The pragmas were placed according to the results of the code profiler. The most time consuming functions of our code are the ones updating the forces along with the Evolve function. We chose to place the pragmas on those functions and on the loops that iterate through each molecule since the operation done inside these loops can be vectorised with simd(i.e. updating multiple data with one instruction).The clauses of the pragmas were the reduction used on the accumulated forces and collapse of the two for loops in the UpdateNonBondedForces function.

Appendices

A C++ code

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

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;
}

/* 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
};

// ===============================================================================
// Two new classes arranging Atoms in a Structure-of-Array data structure
// ===============================================================================

/* 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;
};

// ===============================================================================

/* 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

///////   *******UPDATED********     ////////

void UpdateBondForces(System& sys){
    // Loops over the (2 for water) bond constraints
      for (Bond& bond : sys.molecules.bonds){
        auto& atom1=sys.molecules.atoms[bond.a1];
        auto& atom2=sys.molecules.atoms[bond.a2];
        #pragma omp simd reduction(+:accumulated_forces_bond)
        for(int i=0 ; i < sys.molecules.no_mol ; i++ ){
            Vec3 dp  = atom1.p[i]-atom2.p[i];
            Vec3 f   = -bond.K*(1-bond.L0/dp.mag())*dp;
            atom1.f[i] += f;
            atom2.f[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

////////   *******UPDATED*********     /////////

void UpdateAngleForces(System& sys){
    Angle& angle = sys.molecules.angles[0];
        auto& atom1=sys.molecules.atoms[angle.a1];
        auto& atom2=sys.molecules.atoms[angle.a2];
        auto& atom3=sys.molecules.atoms[angle.a3];
        #pragma omp simd reduction(+:accumulated_forces_angle)
        for(int i=0 ; i < sys.molecules.no_mol ; i++ ){
            //====  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))
            //=============================================================

            Vec3 d21 = atom2.p[i]-atom1.p[i];
            Vec3 d23 = atom2.p[i]-atom3.p[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[i] += f1;
            atom2.f[i] -= f1+f3;
            atom3.f[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 */
    for (auto& atom1 : sys.molecules.atoms){
        for (auto& atom2 : sys.molecules.atoms){// iterate over all pairs of atoms, similar as well as dissimilar
            #pragma omp simd reduction(+:accumulated_forces_non_bond) collapse(2)
                for (long unsigned int i = 0;   i < sys.molecules.no_mol; i++){
                    for (long unsigned int j = i+1; j < sys.molecules.no_mol; j++){
                        Vec3 dp = atom1.p[i]-atom2.p[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[i] += f;
                        atom2.f[j] -= f;

                        accumulated_forces_non_bond += f.mag();
                    }
                }
        }
    }
}

// integrating the system for one time step using Leapfrog symplectic integration

////////   *******UPDATED*********     /////////

void Evolve(System &sys, Sim_Configuration &sc){

    // Kick velocities and zero forces for next update
    // Drift positions: Loop over molecules and atoms inside the molecules
    for (auto& atom : sys.molecules.atoms){
        #pragma omp simd
        for(int i=0 ; i < sys.molecules.no_mol ; i++ ){
            atom.v[i] += sc.dt/atom.mass*atom.f[i];    // Update the velocities
            atom.f[i]  = {0,0,0};                       // set the forces zero to prepare for next potential calculation
            atom.p[i] += sc.dt* atom.v[i];              // update position
        }
    }

    // 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
```

```

////////   *******UPDATED*********     /////////

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 atoms
    Atoms Hatom1( 1, 0.18828, 0.238, 0.41, "H",N_molecules); // Hydrogen atoms
    Atoms Hatom2( 1, 0.18828, 0.238, 0.41, "H",N_molecules);

    // 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;
    #pragma omp simd
    for (int i = 0; i < N_molecules; i++){
        Vec3 P0{i * 0.2, i * 0.2, 0};
        Oatom.p[i] = {P0.x, P0.y, P0.z};
        Hatom1.p[i] = {P0.x+L0*sin(angle/2), P0.y+L0*cos(angle/2), P0.z};
        Hatom2.p[i] = {P0.x-L0*sin(angle/2), P0.y+L0*cos(angle/2), P0.z};
    }

    sys.molecules.atoms.push_back(Oatom);
    sys.molecules.atoms.push_back(Hatom1);
    sys.molecules.atoms.push_back(Hatom2);
    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 all atoms in model one molecule at a time and write out position
    for (auto& atom : sys.molecules.atoms){
        #pragma omp simd
        for(int i = 0 ; i < sys.molecules.no_mol ; i++){
            file << sys.time << " " << atom.name << " "
                << atom.p[i].x << " "
                << atom.p[i].y << " "
                << atom.p[i].z << '\n';
        }
    }
```

```cpp
}

//===========================================================================================
//======================== 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";
}
```

Complete code.