Coding challenge - DCS Computing

Dr. Sangeeth Simon

October 23, 2024

Contents

1 Task 1

1.1 Challenge

Have a look at the following function.

Task 1: Please explain what this function calculates, why is it needed?

1.2 Answer

From reading the code and the references [1, 2, 6], I think that the purpose of this function is to compute the critical time step that will be employed towards stable explicit time stepping of a DEM simulation. The aforementioned function implements two methods

to estimate this, namely the Rayleigh based time step method and the Hertizan contact based time step method.

The Rayleigh time estimate seems to be used generally with a linear contact force model, and is built upon on the idea of a Rayleigh wave: a surface wave that propagates along the surface of elastic materials. A stable estimate of Rayleigh time step ensures that the effect of this Rayleigh wave is contained within the immediate set of particles in contact with each other instead of affecting particles that are sufficiently far away. Li et al. [4] shows that an estimate for the critical time step based on Rayleigh wave can be obtained by taking into account the Rayleigh wave speed, and is given by

$$\Delta t_{crit} = \frac{\pi r}{\beta}\sqrt{\frac{\rho}{G}} \tag{1}$$

where r, ρ and G are the particle radius, density and shear modulus. The parameter β is approximated to,

$$\beta = 0.8766 + 0.163\nu \tag{2}$$

where ν is the Poisson's ratio of the particle. Eqns 1 and 2 are implemented in the line 262 of the aforementioned function in LIGGGHTS. The major known disadvantages of Rayleigh based time estimate is that they provide unstable estimates for highly damped dynamical systems with high relative particle velocities [2].

For highly dynamical systems, a popular recommendation is to use a time step that is based on a nonlinear contact force model named the Hertz-Mindlin model. The nonlinearity arises from the fact that the contact force is modeled to be primarily a function of the overlap area and hence changes with time step. Ardic et al.[3] shows that this time estimate is expressed as:

$$\Delta t_{crit} = 2.9432\sqrt[5]{\frac{25\gamma m^2}{64 v_{rel}}} \tag{3}$$

$$\gamma = \frac{9}{16R}\left(\frac{1-\nu}{G}\right)^2 \tag{4}$$

where m is effective mass, R is effective radius, G is effective shear modulus, ν is effective Poisson's ratio and v_{rel} is the relative velocity between the two spheres in contact. A con-

servative estimate based on this model is important to prevent excessive contact between particles which could lead to nonphysical forces and hamper stability of the solution.

The aforementioned function computes both of these time step estimates and synchronizes them across the MPI processes to obtain a global minimum for each of them.

2 Task 2

2.1 Challenge

Task 2: Test the code and fix any obvious bugs you find. Explain what tools you used to find the bugs.

2.2 Answer

I noticed two bugs in the given code. Both of them were related to wrong memory accesses from data containers. These problems were identified by using the GDB debugger and the valgrind debugger's memory profiling feature. The code was compiled as

```
g++ -g main.cpp
```

before being used with either of these tools.

The first problem concerned the Yeff[...][...] container defined as:

```
std::vector<std::vector<double>> Yeff(max_type, {1.25e6, 1.75e6});
```

The code that *Segfaulted* was the following:

```
for (int ti = 1; ti < max_type + 1; ti++) {
  for (int tj = 1; tj < max_type + 1; tj++) {
    const double Eeff = Yeff[ti][tj];
```

This is because of trying to access elements $Yeff[1][2]$, $Yeff[2][1]$ and $Yeff[2][2]$ which are undefined (C++ indexing for STL containers start at 0). The solution to this problem was to rewrite the access as:

```
const double Eeff = Yeff[ti-1][tj-1];
```

The second problem was concerning accessing the container named *type*[...] that seems to store the types of particles:

```
std::vector<int> type({1, 2});
```

The code that *Segfaulted* was:

```
for (int i = 0; i < nlocal; i++) {
...
 double shear_mod = Y[type[i] - 1] / (2. * (nu[type[i] - 1] + 1.));
}
```

where *type*[...] container was accessed by an index that goes over the number of particles. Thus when $i > 2$, the following access is undefined. The solution to this problem was to assume that particle types alternated between 1 and 2 and therefore to employ the modulo operator on the particle index i to effectively toggle between these types within the loop:

```
for (int i = 0; i < nlocal; i++) {
 ...
 //type access index
 typeIndex = i%type.size();
 double shear_mod = Y[type[typeIndex] - 1] / (2. * (nu[type[typeIndex] - 1] + 1.));
 }
```

The results from a valgrind memory leak check session on the unoptimized code, named 'valgrind-out.txt', is provided with this solution tarball. It confirms that no memory leaks occur in the current code after the above bug fixes.

3 Task 3

3.1 Challenge

Task 3: There are two for loops over all particles in the function "fun". Do you see any way of optimizing these? If so, implement the optimization such that the code runs as fast as possible (while maintaining correct results). Please attach the output (timings) you get for your compiled code, and also indicate what compilation command / flags you used

3.2 Answer

Since it was clarified in a separate email communication that the optimization here meant optimizing for single core performance through compiler flags and general code refactoring, I restricted myself to only such attempts. The optimizations were carried out on my personal 64 bit Lenovo laptop with a 12 core AMD Ryzen 5 8645HS.

Table 1 shows results from my attempts at such optimizations. The column titled 'Original Code' shows the runtime against various problem sizes as measured from the original bug-fixed, unoptimized code. I attempted two level of optimizations. The first level (Level 1) involved simply passing compiler flags $-O3$ and $-march = native$ while compiling the original code. The runtimes measured after this level of optimization is presented in the column titled 'Level 1: Compile Flags Only'. This simple level of optimization already provides us about 80% reduction in total runtime as seen from the third column titled Reduction %.

The second level (Level 2) of optimization was more involved and required a fair bit of refactoring to achieve. Brief details of these refactoring efforts are given below. The runtimes measured after this level of refactoring is collected in column 'Level 2: Code Refactoring and Compiler Flags'. Notice that this level provides an additional 61% reduction in runtime. Figure 1 shows these values as a histogram for easier visualization. To run these cases, follow the instructions in the Table 2.

Overview of refactoring

I briefly discuss some of the salient refactoring efforts that I carried out to achieve the speedups reported at Level 2 optimizations.

1. Explicitly instantiate the vectorMag3DSquared() and inline it:

Problem Size	Original Code	Level 1: Compile Flags Only (-O3, -march=native)	Reduction(%)(original to Level 1)	Level 2: Code Refactoring and Compiler Flags (-O3, -march=native)	Reduction (%)(Level 1 to Level 2)
10	0.007	0.001	85.71	0	100
100	0.042	0.008	80.95	0.001	87.5
1,000	0.409	0.085	79.22	0.008	90.59
10,000	1.306	0.272	79.17	0.086	68.38
100,000	12.799	2.525	80.27	0.856	66.09
1,000,000	127.494	25.118	80.29	9.287	63.03
10,000,000	1278.55	251.104	80.36	97.505	61.17
total	1418	278	80.39	106	61.87

Table 1: Comparison of runtimes observed from the original code, Level 1 optimizations, and Level 2 optimizations. The columns marked as Reduction % indicate the relative reduction in runtime for the current optimization action in comparison to its previous one.

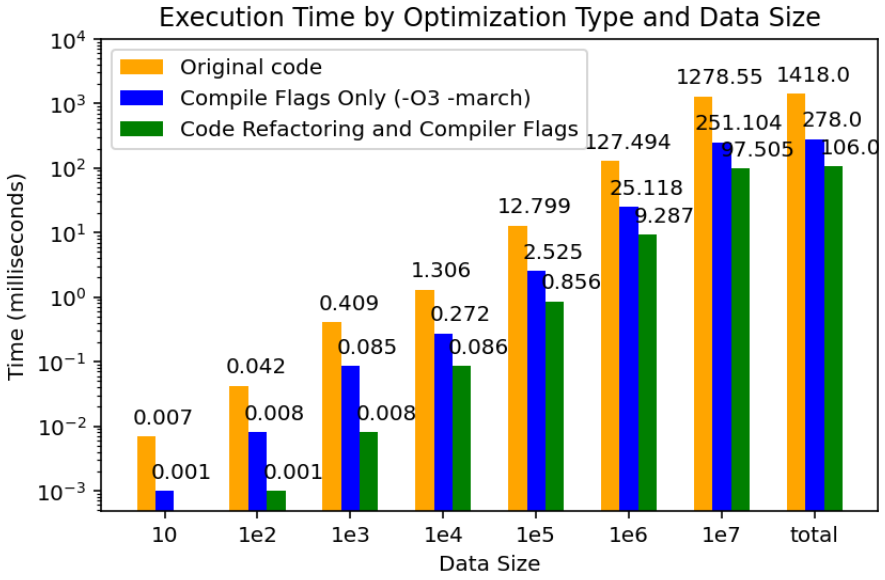

Figure 1: A histogram showing runtime comparison across two levels of optimizations.

```
//Explicit specialization
 template <> inline double MyClass::vectorMag3DSquared
 (const std::array<double,3> &v) const {
   return v[0] * v[0] + v[1] * v[1] + v[2] * v[2];
 }
```

Optimization level	Git tag name	Compilation options
Original code	level1_optimized	g++ main.cpp
Compile Flags Only (-O3, -march=native)	level1_optimized	g++ -O3 -march=native main.cpp
Code Refactoring and Compiler Flags (-O3, -march=native)	level2_optimized	g++ -O3 -march=native main.cpp

Table 2: Table showing the mapping between various version of the code in terms of their level of optimizations, their respective git tags on the master branch and their respective compilation options.

2. Make *max_type* a compile time constant:

```
constexpr int max_type = 2;
```

   Also use $std :: array$ for all containers based on the *max_type* value.

3. Pre-compute the coefficients involving ν and *shear_mod* that are used in the loop for Rayleigh time computation before entering the loop.

```
//Loop over the types of particles, precompute and store:
  //coeff involving nu
  //Shear Mod
  for (int i = 0; i < max_type; ++i) {
    coeff1_nu[i] = 1./(2. * (nu[i] + 1.));
    coeff2_nu[i]= 1./(0.1631 * nu[i] + 0.8766);
    coeff_shearMod[i] = 1.0/(Y[i]*coeff1_nu[i]);
  }
```

4. Change divisions to multiplications wherever possible.

5. Substitute branching for min, max computations within loop over particles with $std :: min$ or $std :: max$.

6. Compute *hertz_time_min* as a minimum on the refactored expression:

```
hertz_time_i = meff * meff / (reff * Eeff * Eeff * v_rel_max_simulation);
hertz_time_min = std::min(hertz_time_i, hertz_time_min);
```

and apply the $pow(hertz\_time\_min, 0.2)$ only once after exiting the loop.

7. Manually unroll the loop

```
for (int ti = 1; ti < max_type + 1; ti++) {
 for (int tj = 1; tj < max_type + 1; tj++) {
  ...
 }
}
```

to aid compiler optimization of the loop over the particles that is nested within them. I am not personally happy with how ugly the code looks after doing this especially considering that it offers only about 10ms of total runtime improvement against a code version that does not involve the loop unrolling. Nevertheless, I did it to demonstrate what is possible.

8. Move all stack & heap variable declarations outside the loops.

4 Task 4

4.1 Challenge

Bonus task 4: There is a comment saying " this is not exact...". Can you identify what this comment refers to? I.e. what is not exact?

4.2 Answer

To me, this comment looks directed towards the approximations involved in the computation of the Hertz time. The formula used in the code is definitely different from the one presented in Eqn 4. One form of approximation that is immediately noticeable to me is that the formula for $meff$ is built for a sphere of radius r which may not be perfectly valid in a simulation with non-spherical particles. Secondly, the parameters ν and G have been replaced with an effective Young's modulus $Eeff$ through some constitute relationship. The multiplication factor 2.1736 (obtained from $2.9432 * ((25/64) * (9/16))^{0.2}$) has also been replaced by 2.87 which seems to be some form of empirical fit.

Bibliography

[1] CFDEM Documentation, *fix_check_timestep_gran*, Available at: `https://www.cfdem.com/media/DEM/docu/fix_check_timestep_gran.html`

[2] S. J. Burns, P. T. Piiroinen, and K. J. Hanley, *Critical time-step for DEM simulations of dynamic systems using a Hertzian contact model*, International Journal for Numerical Methods in Engineering, 2019.

[3] Ö. Ardic, *Analysis of Bearing Capacity Using Discrete Element Method*, Ph.D. Thesis, Middle East Technical University, Civil Engineering, 2006.

[4] Y. Li, Y. Xu, and C. Thornton, *A comparison of discrete element simulations and experiments for "sandpiles" composed of spherical particles*, Powder Technology, Vol. 160, 2005, pp. 219–228.

[5] M. Otsubo, C. O'Sullivan, and T. Shire, *Empirical assessment of the critical time increment in explicit particulate discrete element method simulations*, Computers and Geotechnics, Vol. 86, 2017, pp. 67–79.

[6] C. O'Sullivan, J. D. Bray, *Selecting a suitable time step for discrete element simulations that use the central difference time integration scheme*, Engineering Computations, March 2004, pp. 278-302.