Report on the optimization loops: avoid avoidable

Nicole Orzan

February 5, 2018

In this report you can find a description about the steps done to optimize the code provided avoid_avoidable.c This exercise was ran and measured on my laptop, which is a Lenovo with Intel CPU: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHzi7. For each step of the optimization it was analyzed the time spent by the run of the program to be finished. For the analysis to be complete, for each optimization step was calculated the average execution time of the program, computed on a sample of 12 runs.

The optimization of the code was divided into 8 different steps, of which the first 6 will be examined. Optimizing the code means rewriting some parts of the code and clean some others. The steps done are listed below, where I also wrote some parts of the result of the "diff" command between the steps we're in and the one before.

- step 0: starting point;
- step 1: remotion of the roots calculations (sqrt), substituting it with the use of squared powers, everywhere possible;

```
avoid_avoidable_loop0.c
<       dist = sqrt( pow(x[p] - (double)i/Ng + half_size, 2) +
<                    pow(y[p] - (double)j/Ng + half_size, 2) +
<                    pow(z[p]  - (double)k/Ng + half_size, 2) );
<       if(dist < Rmax)
<                    dummy += dist;
---
avoid_avoidable_loop1.c
>       dist = pow(x[p] - (double)i/Ng + half_size, 2) +
>                    pow(y[p] - (double)j/Ng + half_size, 2) +
>                    pow(z[p]  - (double)k/Ng + half_size, 2);
>
>       if(dist < Rmax2)
>                    dummy += sqrt(dist);
```

- step 2: replacing the "pow" function with the use of multiplications;

```
avoid_avoidable_loop2.c
>       double dx, dy, dz;
>                    dx = x[p] - (double)i/Ng + half_size;
>                    dy = y[p] - (double)j/Ng + half_size;
>                    dz = z[p] - (double)k/Ng + half_size;
>
>            dist = dx*dx + dy*dy + dz*dz;
```

- step 3: replacement of the repeated calculations of an inverse with a new fixed variable, separation of the 3D calculus of the distances with the separate use of the coordinates x,y and z;

```
avoid_avoidable_loop3.c
>   double Ng_inv = (double)1.0 / Ng;
>            dx = x[p] - (double)i * Ng_inv + half_size;
>            dy = y[p] - (double)j * Ng_inv + half_size;
>            dz = z[p] - (double)k * Ng_inv + half_size;
```

- step 4: taking outside form the inner loops the parts that do not depend on them in order to avoid repeated useless calculations.

```
avoid_avoidable_loop4.c
for(p = 0; p < Np; p++)
    for(i = 0; i < Ng; i++)
      {
        double dx2 = x[p] - (double)i * Ng_inv + half_size; dx2 = dx2*dx2;

        for(j = 0; j < Ng; j++)
          {
            double dy = y[p] - (double)j * Ng_inv + half_size;
            double dist2_xy = dx2 + dy*dy;

            for(k = 0; k < Ng; k++)
              {
                double dz;
                dz = z[p] - (double)k * Ng_inv + half_size;

                dist = dist2_xy + dz*dz;
        ...
```

- step 5: separating the multiplications of the variables in different line.

```
avoid_avoidable_loop5.c
>       double dy2 = y[p] - (double)j * Ng_inv + half_size;
>       dy2 = dy2*dy2;
>       double dist2_xy = dx2 + dy2;
---
>       double dz2;
>       dz2 = z[p] - (double)k * Ng_inv + half_size;
>       dz2 = dz2*dz2;
---
>       dist = dist2_xy + dz2;
```

- step 6: insertion of some "register" variables, for the most often called ones. Those variables are stored in the CPU registers, and so they can be stored and accesses quickly.

```
avoid_avoidable_loop6.c
>       double register Ng_inv = (double)1.0 / Ng;
---
>       double register dy = y[p] - (double)j * Ng_inv + half_size;
>       double register dist2_xy = dx2 + dy*dy;
---
>       double register dz = z[p] - (double)k * Ng_inv + half_size;
```

At this point I ran the different codes and I plotted the mean execution times.
I decided to use also some compiling options, such as O0, O1, O2 and O3, to take again the measurements and to compute the mean times, to examine the possible differences. With the -O option, the compiler tries to reduce code size and execution time, and turns on a lot of optimization flags.
You can find the table containing the mean calculated times in figure 1. I provide also a graph to see the trend of the different compiler optimizations for the different optimizations steps; you can see it in figure 1.

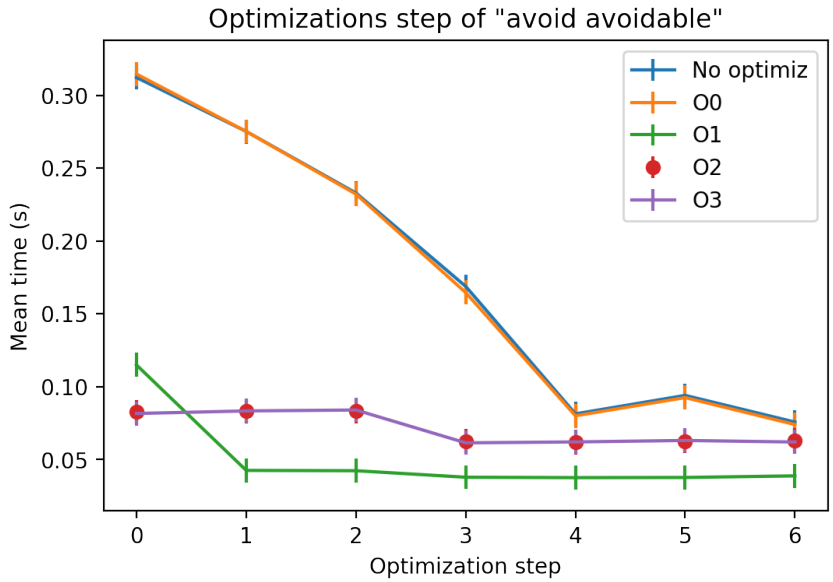

Figure 1: Mean computing time and errors for optimization steps from 0 to 6, without and with compiler optimizations O0, O1, O2 and O3 on. Input: 50 as number of particles, 50 as grid points, 2 as weight.

Step	Hand optim	σ	O0	σO0	O1	σO1	O2	σO2	O3	σO3
0	0.31220	0.00826	0.31458	0.00826	0.11488	0.00824	0.08270	0.00824	0.08158	0.00824
1	0.27518	0.00825	0.27520	0.00826	0.04251	0.00823	0.08319	0.00825	0.08335	0.00825
2	0.23293	0.00825	0.23216	0.00825	0.04230	0.00824	0.08311	0.00824	0.08397	0.00824
3	0.16883	0.00826	0.16461	0.00825	0.03783	0.00824	0.06269	0.00824	0.06144	0.00824
4	0.0813	0.00825	0.08002	0.00825	0.03755	0.00824	0.06181	0.00824	0.06206	0.00824
5	0.09400	0.00824	0.09251	0.00825	0.03767	0.00824	0.06247	0.00824	0.06306	0.00824
6	0.07570	0.00826	0.07399	0.00825	0.03876	0.00823	0.06304	0.00824	0.06200	0.00825

Table 1: Mean computing times and errors for each optimization step, with and without compiler options O0, O1, O2 and O3 on. Input of the run: 50 as number of particles, 50 as grid points.: 50 as number of particles, 50 as grid points, 2 as weight.