Report on the optimization loops: distribute particle

Nicole Orzan

December 16, 2017

The aim of this exercise was to profile a code provide (distribute particle), identify the hotspots and optimize it. The provided code is a loop over a number Np of particles with the aim of distributing the mass of the particles over a grid with an appropriate weight function, so that to have a resulting density field instead of a particles distribution. The code given is correct but it also is highly inefficient in several aspects, so I had to optimize it everywhere possible, cleaning and redesigning some parts.
As input data I used 50 as number of particles, 50 as number of grid points and 2 as value for the weight function.

Call-graph

Before starting to optimize the code I mapped it using a call-graph (or call-tree), that is a control flow graph showing the relationships between the various parts of the code, like the main function and its subroutines. The profiler uses information collected during the actual execution of the program, and allows the user to understand where the program spent its time. This information can show to the user which pieces of the program are slower than expected, and helps in this way to find the hotspots to rewrite.
To create a call-graph I used gcc as compiler and gprof, a GNU tool to profile the code. The steps I made were:

- I compiled with: gcc -pg myprog.c -o myprog.x -lm
- I ran with: ./myprog (input)
- I created a callgraph using: gprof myprog.x | gprof2dot | dot -T png -o callgraph.png

What I obtained was a tree-made graph where each node represent a procedure, and each arch indicates the called functions.
In figure 1 it is shown the call-graph for the given code without optimizations (step 0 of the optimizations). We can see, as expected, that the main function in called 100% times. The main calls in turn two subfunctions, ijk and MAS.

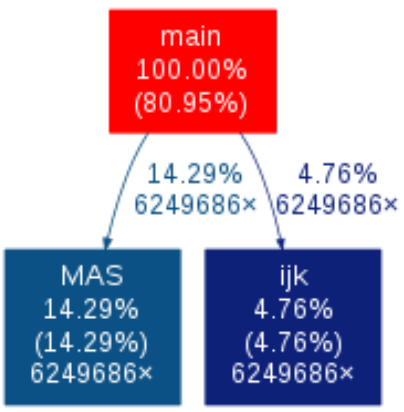

Figure 1: Callgraph of the code provided, without optimizations.

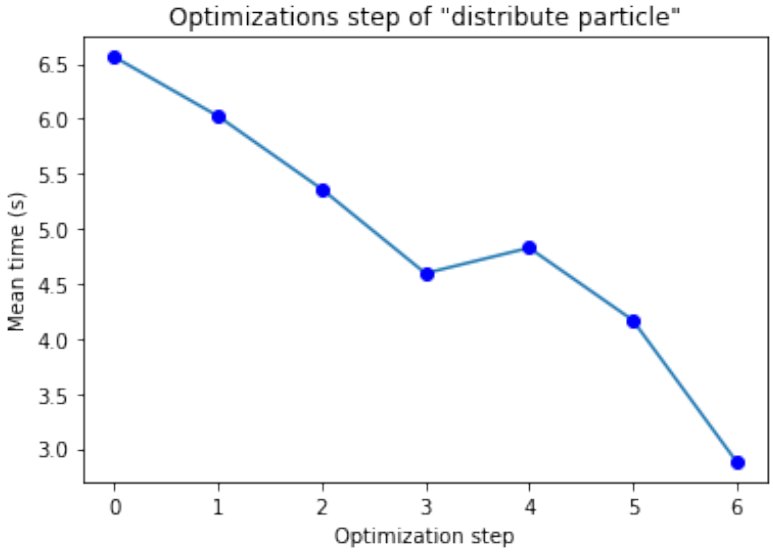

Figure 2: Mean computing time for each optimization step

Optimization steps

In this section I summarize the steps done to optimize the code, using as example the code optimizations given in the loop optimization folder. The optimization have been divided into 7 different steps, which are the followings:

- step 0: starting point, worst case, given;
- step 1: some changes inside the calculus of the distance: I used the squared power of the distance, in order to remove roots calculus;
- step 2: I separate the calculus of the distance (which is in 3D) calculating the distance for every coordinate x,y and z; I replaced also the calculus of dx^2 with $dx * dx$;
- step 3: replacing the calculus of $1/N_g$ with a new variable, in order to avoid to do it repeatedly;
- step 4: I moved the loop on the particles outside the loop on the grid;
- step 5: insertion of some "register" variables, for the most often called ones;
- step 6: insertion of new variables, in order to replace the use of x, y and z to work in 3D. Removal of the use of the functions, putting everything inside the main.

For each step of the optimization it was analyzed the time spent by the program to be finished. The input used were: 50 as number of particles, 50 as number of grid points and 2 as weight function.
For the analysis to be complete, for each optimization step was calculated the mean time running the program 12 times saving the results in different files, which were then opened to calculate means.
Then I used jupyter-notebook to create the graph of the mean execution times computed for every step, which you can see in figure 2.

Compiler optimizations

At this point I decided to use also some compiling options, such as O0, O1, O2 and O3, to take the same measures and compute the mean times. 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 a 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 I made; you can see it in figure 3

Optimiz Step	Hand optim	O0	O1	O2	O3
0	6.56	6.35	1.43	1.59	1.61
1	6.02	5.80	1.08	1.07	0.97
2	5.28	5.15	1.07	1.06	0.97
3	4.54	4.38	1.05	1.06	0.98
4	4.82	4.76	1.22	1.22	1.23
5	4.16	4.19	1.21	1.21	1.24
6	2.88	2.90	1.21	1.24	1.22

Table 1: Mean computing time for each optimization step, without compiler options and with compiler options O0, O1, O2 and O3 added.

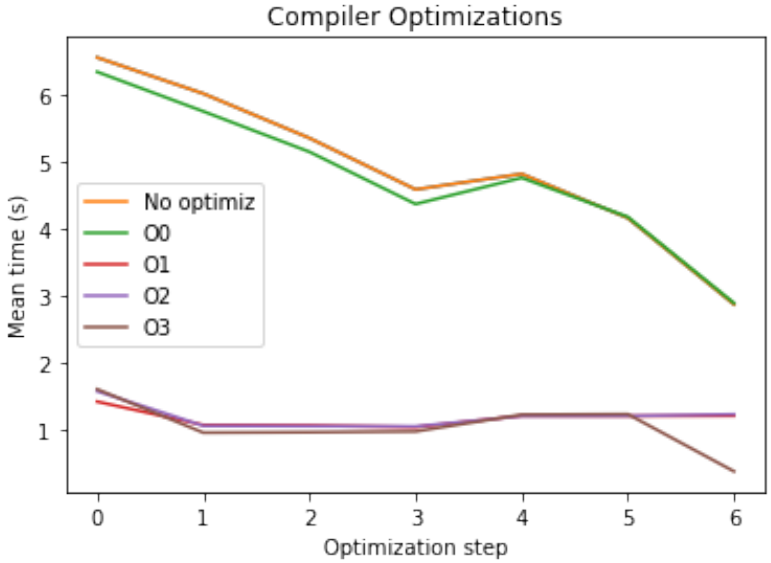

Figure 3: Mean computing time for each optimization step, with the addition of the compiler optimizations O0, O1, O2 and O3.

We can see from the table and the graph above that there is not a big difference between the case without compiler optimization and the case with O0 optimization, while the use of)1, O2 and O3 options gives us different trends, lowering a lot the computing time required. The trends of those three curves are quite the same, even if we can notice from the table that the option O3 gives slightly better results.

Conclusions

With this exercise we understood how important it is to write the code in a smart and clean way, eliminating bugs, to improve its the performance. A silly way to write the code - for examples the use of roots and divisions - can make our code much worse, so it is better to avoid them when possible.
Obviously we can help ourselves using also compiler optimizations, which gave us a good help in decreasing the execution time of the application. We can see, by the way, that our hand-made optimization give a big difference also in the performance obtained by O1, O2 and O3 optimizations, so its importance is not to be underestimate!