Report on the optimization loops: distribute particle

Nicole Orzan

February 5, 2018

The aim of this exercise was to profile a provided code (distribute_particle.c), 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 given code is correct but it also 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, 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 the most. 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 (the add of the -pg option allows to profile with gprof)
- I ran with: ./myprog [input data]
- I created a callgraph using: gprof myprog.x | gprof2dot | dot -T png -o callgraph.png

What you get with this commands is 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 that the main function is called 100% times and calls in turn two subfunctions, ijk and MAS.

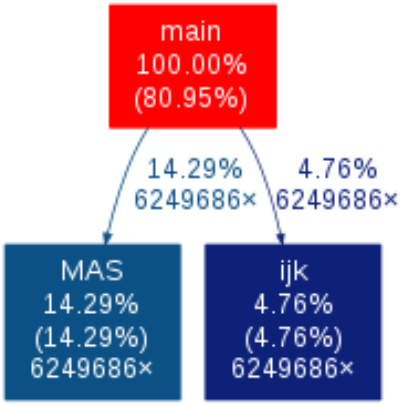

Figure 1: Callgraph of the code provided, executed with input 50 50 2.

Step	Hand opt(s)	σ	O0(s)	σO0	O1(s)	σO1	O2(s)	σO2	O3(s)	σO3
0	0.50478	0.00847	0.48829	0.00827	0.11008	0.00824	0.12199	0.00824	0.12428	0.00824
1	0.46337	0.00832	0.44301	0.00825	0.08327	0.00824	0.08229	0.00824	0.07446	0.00824
2	0.41238	0.00833	0.39659	0.00826	0.08267	0.00824	0.08194	0.00823	0.07496	0.00824
3	0.35343	0.00826	0.33669	0.00825	0.08079	0.00824	0.08161	0.00823	0.07582	0.00824
4	0.37126	0.00826	0.36678	0.00826	0.09375	0.00824	0.09346	0.00823	0.09512	0.00825
5	0.32061	0.00825	0.32203	0.00828	0.09353	0.00823	0.09364	0.00823	0.09568	0.00825
6	0.22168	0.00824	0.22327	0.00833	0.09384	0.00824	0.09555	0.00824	0.09512	0.00824

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

Optimization steps

In this section there are, summarized, the steps that I made to improve the performance the code, using as example the code optimizations given in the loop optimization folder. The optimization work have been divided into 7 different steps, which are the followings:

- step 0: starting point, worst case, given;
- step 1: use of the the squared power in order to remove roots calculus;
- step 2: replacing the "pow" function with the use of multiplications and separation of the 3D calculus of the distances with the separate use of the coordinates x,y and z;
- step 3: replacing the repeated calculus of the inverse value $1/N_g$ with a new fixed variable, in order to avoid to do it repeatedly;
- step 4: separating the multiplications of the variables in different line.
- step 5: moving the loop on the particles outside the loop on the grid;
- step 6: insertion of some "register" variables, for the most often called ones

In general those steps are the same done in the "avoid_avoidable" exercise.
For each step of the optimization it was analyzed the time spent by the program to be executed; for the analysis to be complete, for each optimization step I calculated the mean time running the program 12 times and then I computed the error as the standard deviation.

Compiler optimizations

At this point I decided to use also some compiling options, such as O0, O1, O2 and O3, to take again the measurements of the execution times and to compute the means (with the respective errors). With the -O0 flag, we prevent the compiler to use any kind of optimization, while with the other ones the compiler improves the performance trying to reduce code size and execution time (turning on different optimization flags).
You can find the table containing the mean calculated times for an execution with input 50 as number of particles, 50 as number of grid points and 2 as weight functions 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 2.

We can see from the table and the graph above that the use of O1, 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, above all in the last steps of the optimization.
We can clearly notice that from the version 3 to the version 4 there is a slowdown of the speed of the code, as it happens "avoid_avoidable" exercise. This happens maybe because we are separating the multiplications of the variables in a different line.
We can see that without optimizations and with -O0 flag the importance of writing a clean and hand-optimized code is really big! Indeed the hand optimization of the final step improves the speed of the zero step code by the 56%.

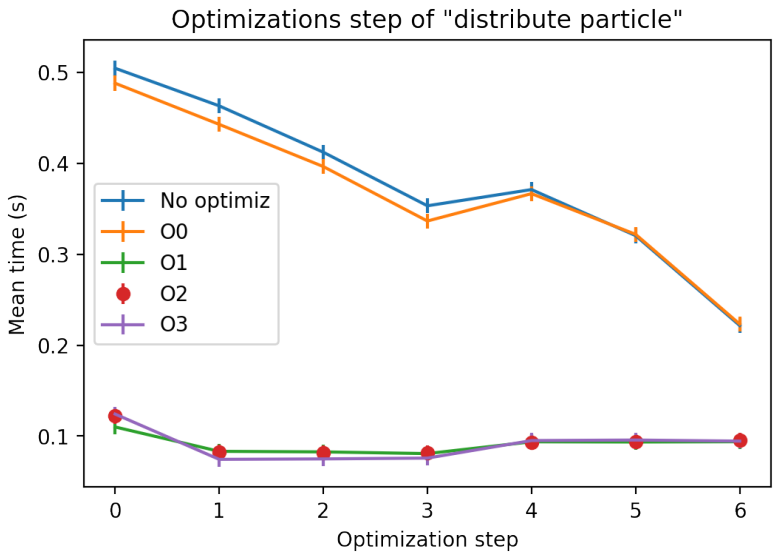

Figure 2: Mean computing times and error bars for each optimization step, with the addition of the compiler optimizations O0, O1, O2 and O3. Input: 50 as number of particles, 50 as grid points and 2 as weights.

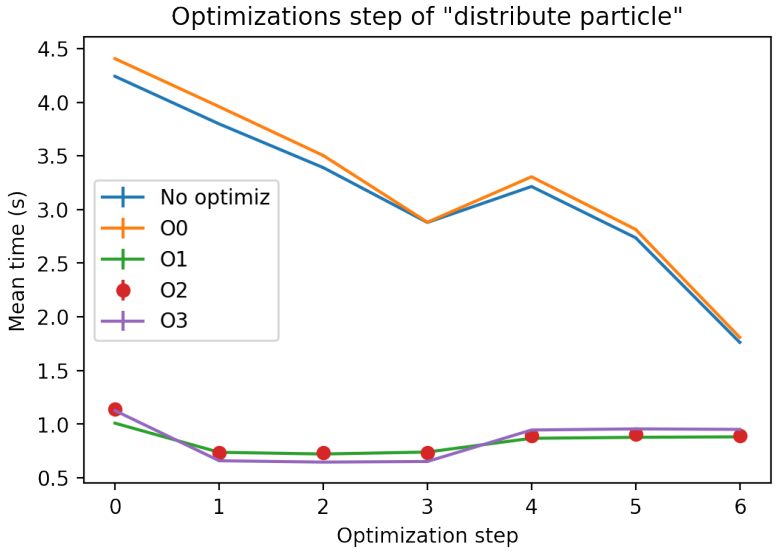

Figure 3: A new measurement with different variables. Here I used 256 as number of particles, 128 as grid points and 1 as weights. Here you can see mean computing times and error bars for each optimization step, with the addition of the compiler flags O0, O1, O2 and O3.

Conclusions

With this exercise we understood how important it is to write the code in a smart and clean way, to check it and eliminating bugs to improve its the performance. Writing a code in a careless way - for examples using roots and divisions when not necessary - can lead us to lose performance, so it is better to avoid those errors 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.