Parallel Programming

C Programming and Performance Delivery

Alfredo Hernández & Alejandro Jiménez

27 October 2017

Contents

Goals

- Understand the algorithm and its computational model (complexity, problem size).
- Performance engineer the code: optimize the serial execution and understand where is the performance bottleneck.
- Identify opportunities of vectorization (use of SIMD instructions) and improvement of the memory accesses.

Detailed Schedule

1. Select one of the applications described in this document. Define the operation related to the application that will be used as a unit to measure the performance (operations/second). Estimate the algorithmic complexity of the application: i.e., find a formula to compute the total number of operations performed as a function of the problem size (one or more parameters related to the input of the application).
2. Compile the base version of the program and measure relevant performance metrics (time, instructions executed, IPC). Select a problem size that is big enough for the execution to take several seconds. Determine a way to check that the output of the program is correct: remember that the discrete nature of the mathematical operations using real numbers (floating-point) implemented in the computer means that the associative property does not hold for additions and multiplies. Therefore, changes in the order of the operations may generate slight variations in the results that can grow if the accumulated error diverges.

3. Measure execution time to check the effect of the problem size on performance. Identify the time taken by the initialization stages and the time taken by the main part of the algorithm. Ideally, the initialization part should be relatively less and less important as the problem size grows. By selecting the data type used for real numbers, either float or double, you can test the effect both on performance and functionality.
4. You can check the performance differences of the code generated by different compilers (different versions of the GNU `gcc` or Intel `icc`) and using different compilation flags or options.
5. Improve the program implementation of the serial (or single-thread) baseline code in order to achieve better performance. There are several classical optimizations that you should consider: loop interchange (or reordering), code motion, strength reduction, … Ask for help to your teacher. Explain your results, both when the optimizations succeed and when they fail.
6. Improve the program implementation of the serial code in order to take advantage of the SIMD or vector instructions available in the processor. Ask the teacher if you need help.
7. Find out the performance bottleneck of the program (using `perf`). Ask the teacher if you need help.

Introduction

When executing this kind of simulations, the first thing to do is to define the problem *size*. The problem *size* is about the dimensionality of the simulation we are carrying on. In this case - a diffusion problem - the dimensionality has to do with the *size* of the cube where the simulation is running on. In this program the cube is expressed as a 3-dimensional matrix. Since one of the objectives is to measure the performance of the code, we can carry the program over different problem *sizes*.

Another point worth considering is the comparison between the performance of different compilation flags. For this purpose, we have considered two different compilation flags: `O2` and `O3`. Another popular possibility was the flag `Ofast`, but we discarded because it is not so different from `O3` and it enables some non-standard compliant optimisations and can lead to some problems.

One decision that we have also to make is about the compiler that we are going to use. There are many compilers to consider, each with its own advantages, but we limited our scope to the more popular ones. Bibliographical research has lead us to decide on `gcc v7.2.0` (the latest version available) over the rest of them. Even though the differences measured in performance were small and with heterogeneous results, the tests presented here will be compiled with `gcc`.

From `.c` to `.csv`

The two bash scripts down below compile different versions of our code over different problem *sizes* and they export the performance results using `perf stat`. This lets us output raw data with the performance and time data of the execution of the code into a beautifully simple `.csv` file.

```bash
#!/bin/bash

perfbin=do-perf.sh; executable=run;
source=$1; perfoutfile=$2;

chmod +x $perfbin

if [ -f ${perfoutfile}-base.csv ]; then
    rm ${perfoutfile}-base.csv
    echo "Removed previous results."
fi

echo "Looping with floats..."
```

```
gcc -lm ${source}.c -o $executable
for (( i = 25; i <= 125; i +=25 )); do
    bash $perfbin $i $i $i $executable ${perfoutfile}-base.csv
done
```

```
#!/bin/bash
NX=$1; NY=$2; NZ=$3; executable=$4; perfoutfile=$5
outfile=out.txt

if [ ! -f $perfoutfile ]; then
    echo "iter,init,acc,nx,ny,nz,cycl,inst,time,br,brmiss" > $perfoutfile
fi

perf stat -o $outfile -e cycles,instructions,branches,branch-miss ./$executable \
$NX $NY $NZ | awk '/Running/ { print $8; print $10} /Accuracy/ { printf("%e,", $3) }' \
ORS="," >> $perfoutfile
awk '/stats/ { printf ("%d,%d,%d,", $6, $7, $8) }' $outfile >> $perfoutfile
awk '/cycles/ { gsub(/\,/,""); print $1 }' ORS="," $outfile >> $perfoutfile
awk '/instructions/ { gsub(/\,/,""); print $1 }' ORS="," $outfile >> $perfoutfile
awk '/elapsed/ { print $1 }' ORS="," $outfile >> $perfoutfile
awk '/branches/ { gsub(/\,/,""); print $1 }' ORS="," $outfile >> $perfoutfile
awk '/branch-miss/ { gsub(/\,/,""); print $1 }' $outfile >> $perfoutfile

rm $outfile
```

Cleaning variables with R

Using R we can easily do calculations with the results obtained with `perf stat` (in `.csv` files):

```
# Accuracy + init
df.float <- read_results("perfstats-base-float.csv", "float")
df.double <- read_results("perfstats-base-double.csv", "double")
# Flags
df.no <- read_results("perfstats-flags-no.csv", "base")
df.o2 <- read_results("perfstats-flags-o2.csv", "O2")
df.o3 <- read_results("perfstats-flags-o3.csv", "O3")
# Optimisations
df.bs <- read_results("perfstats-opts-base.csv", "base")
df.nt <- read_results("perfstats-opts-notime.csv", "no time")
df.nt.gl <- read_results("perfstats-opts-notime-goodloop.csv", "no time + good loop")

# Rbinds
df.acc <- rbind(df.float, df.double)
df.flags <- rbind(df.no, df.o2, df.o3)
df.opts <- rbind(df.bs, df.nt, df.nt.gl)
```

Performance tests and optimisation process

Defining performance

Now that we have clean data, we start the analysis of the results. Using `R` scripts, we adequately merge results from different versions of the code over different problem *sizes*.

One interesting point about the data analysis of the performance results is the different definitions we can consider about the concept of **performance**. A classical and straight-forward way to define performance is *time*. The *time* needed to execute the program is the fastest and easy-to-understand magnitude to measure the efficiency of a code; assuming a program does everything we want it to do, the fastest the better. However reasonable this may sound, it does not lack of some flaws and is easy to trick it into apparent high performance. In fact, there is no perfect way to measure the performance. Taking all of that into consideration, a safe way to measure and compare the performance is using a variable known as **operations** per second, which is derived from easily obtainable variables using a profiler such as `perf`:

$$\frac{Operations}{Second} = \frac{Operations}{Instruction} \times \frac{Instructions}{ClockCycles} \times \frac{ClockCycles}{Second}$$

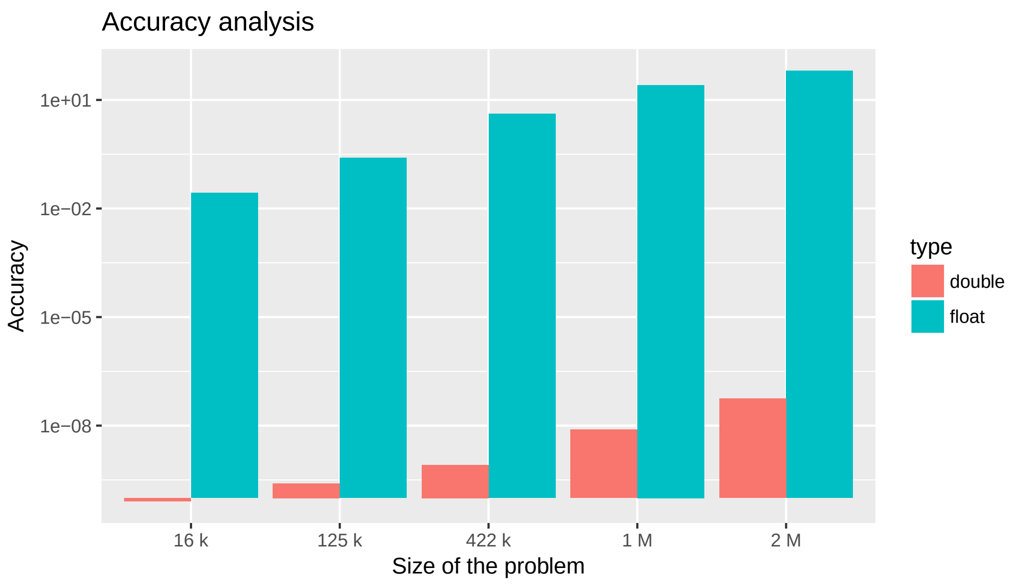
The most controversial thing about the definition of *Operations* is that it lacks of a formal definition. In our case, we have considered that the number of operations escalates with the number of loops of the code. Taking a look into the code, one can notice three triple loops on nx, ny and nz and one quadruple loop on nx, ny, nz and $iter$. Thus, we define the number of operations as $operations \propto n_x \times n_y \times n_z \times (3 + iter)$.

```
read_results <- function(file, typ) {
    data <- read_csv(file)
    data <- data %>%
        mutate(type = typ) %>%
        mutate(brpmiss = brmiss/br) %>%
        mutate(op = nx * ny * nz * (3 + iter)) %>%
        mutate(size = nx * ny * nz) %>%
        mutate(ipc = inst/cycl) %>%
        mutate(clck = cycl/time/10^9) %>%
        mutate(opps = op/time/10^6) %>%
        mutate(acc = abs(acc))
    return(data.frame(data))
}
```

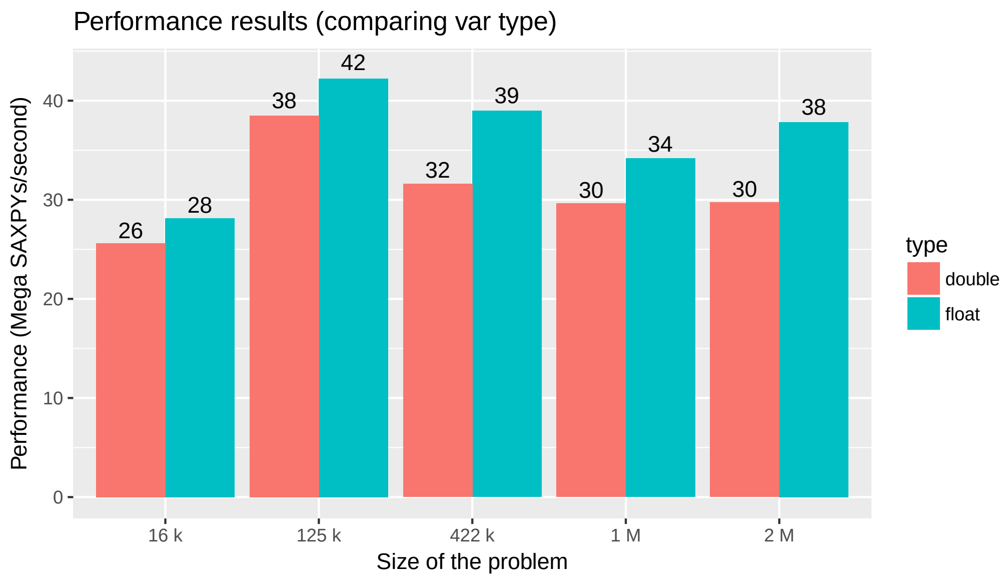
Influence of the variable type

In all this process, it is wise to keep in mind that making changes in a code can lead to unexpected results, and this can compromise the functionality of our code. What does functionality mean? The functionality is not altered if the code does exactly what it used to in the exact same way. For instance, if we change all `double` variables with `float` variables, we could spare some memory space and improve the performance, but we would lose some decimals in our variables, this leads to a loss in **accuracy**. Actually, since all decimal variables present in the code are declared as `float`, we have made the opposite change and studied its consequences:

```
get_acc_plots(df.acc)
```



```r
get_perf_plots(df.acc, "(comparing var type)")
```

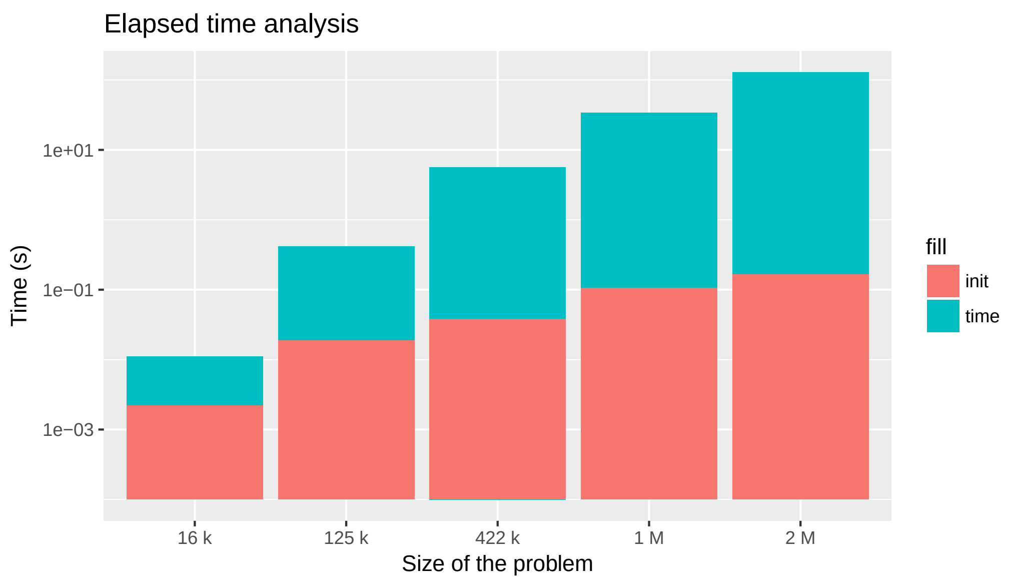


Taking a look at the results we can notice how the performance of the program has been deteriorated a bit, but the accuracy has been greatly improved. Thus, we chose to use `double` variables and accept this small trade-off as we will be optimising the code anyway.

Concerning time

A crucial point considering the measurement of performance is that we are always using **time**. Not as easy as it may seem, the *time* we measure is a concept to be well defined. The matter about time is that we have the time needed to initialize the program in one hand, and the total time of execution in the other. In order to take a decision we first take a look at the dependence of both *times* on the *size* of the problem. Note that the scale is logarithmic.

```r
get_time_plots(df.double)
```

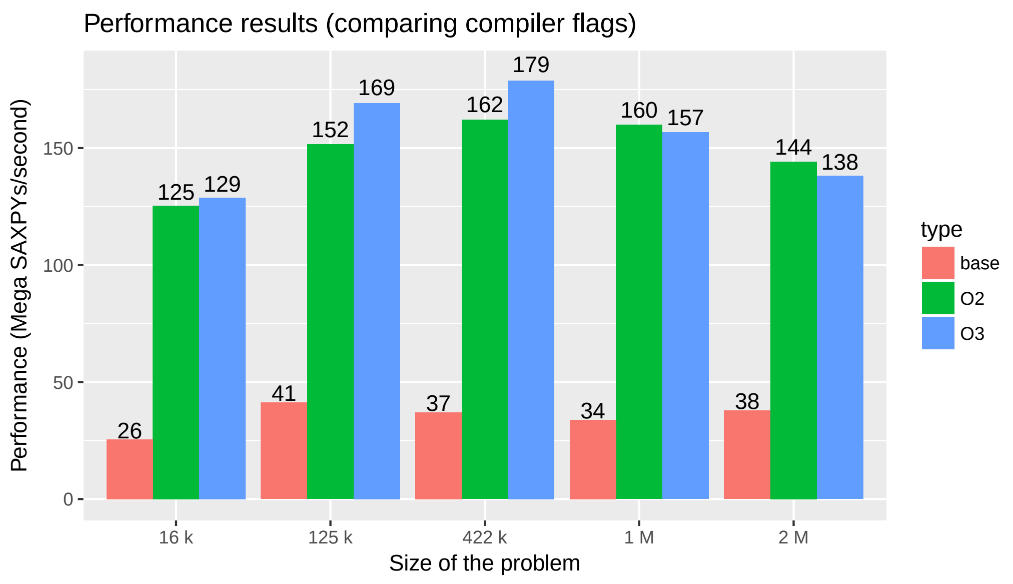


Here we can notice how irrelevant the *init* time becomes when we make the problem big enough, and since we prioritise the measures taken from the biggest problems, we don't need to make any major optimisations of the initialisation process.

Choosing compiler flag

Now that we know how we will measure the performance and we understand how the time is measured, we just need to decide which compiler flag is more suitable for our purpose. Now we execute the initial version of the code using different `gcc` flags.

```
get_perf_plots(df.flags, "(comparing compiler flags)")
```



Then we can decide which compiler flag is the best option for us. Is not an easy choice, for we can notice that for some problem *sizes* the highest performance is obtained using `O3`; but for the bigger problem *sizes*, the best performance is obtained using `O2`. We don't know enough about `gcc` to be sure this is just an statistical outlier or a real pattern. Taking into account the scalability of the problem and prioritising the measures of the bigger *sizes*, we have decided to use the compiler flag `O3` henceforth regardless.

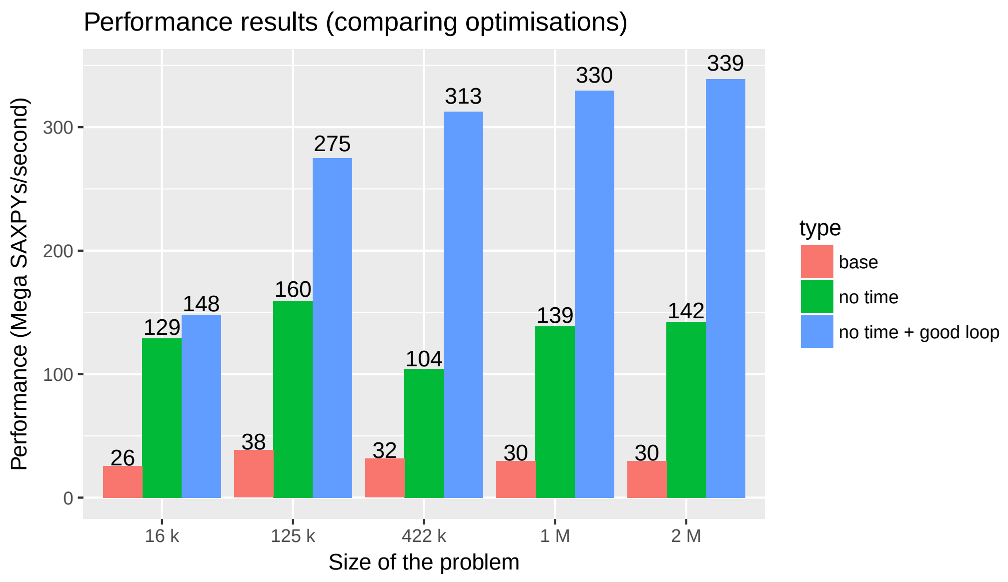
Classical optimisations

At this moment we should be able to measure the performance of our code in a pretty reliable way. Is at this point where we find useful and interesting to optimise some parts of our code, for we can now measure precisely its effects.

For this purpose, we prepared different versions of the original code:

- **Original version**: self-explanatory.
- **No-time version**: exploring the original code, we noticed that a variable called `time` was declared and set to 0, then this variable was used in many functions and computations. Most of this computations were expendable knowing that the variable *time* was always zero. Removing this computation and setting its results as constants without computation simplified the code and optimized its performance without modifying the results.
- **Good-loop version**: one easy way to trick the performance of a code is to permutate the index of multiple loops. We have played with many different combinations of loops and found the fastest one (z nested inside x, all nested inside y), which is the one used in this version.
- **Good-loop + no-time version**: this version involve both changes applied in the previous versions.

```
get_perf_plots(df.opts, "(comparing optimisations)")
```



Conclusions

The performance of a program is always a crucial point in the development of a code. A small change in the code can lead to impressive differences in the time of execution, differences that can make a difference between a truly useful program and an impractical bunch of code. Is important to take into account that a change in the code can implies a trade-off between performance and functionality, and this means that these changes have to be thoroughly thought over. The good measurement of the performance is thus a key point of this decision making process and an indispensable step in a optimization task.

References

- `https://openbenchmarking.org/result/1707017-TR-CLEARLINU79`
- `https://stackoverflow.com/questions/14492436/g-optimization-beyond-o3-ofast`