# Pràctiques de Programació Conscient de l'Arquitectura
# Lesson 2: Programming and Optimizing Tools Activities

Daniel Jiménez-González

February 21, 2021

# Index

# 1

# Tools to Measure

In this lab you will analyze `pi.c` in all the exercises but the first one, where you will use `popul.c`. You have to run them without any parameter but be carefull beacuse `popul.c` prints out binary data. `pi.c` computes the first 10000 decimals of the $\pi$ number and writes them in the stardard output. Download lab2_sessions.tar.gz from the Racó. This file contains the source code of the two programs that you will use in your laboratory 2.

## 1.1 Accounting Tools

1. Check that you have different versions of the `time` command, running, for instance, `time ls` and `/usr/bin/time ls` from different shells (`bash`, `csh`, `tcsh`,...).
2. I/O and the execution environment may affect the performance of the application. You will test that using `popul.c` program, compiled with `gcc` in your FIB account.
   (a) Run the program and do timing with the GNU `time` command, redirecting the output to a file in your FIB account: under `/home/...` and under `/dades/...` (Note: at the FIB machines, your `/home/...` account disk is mounted by NFS, and your `/dades/...` account disk is mounted by CIFS).
   (b) Repeat the experiment redirecting the output to a file located at `/tmp`.
   (c) Do you see any difference? It may be significant and the *%CPU* may be very small if several runs are done, why? Reason your answer.
3. It is time to save the golden output of `pi.c` program:
   (a) Compile `pi.c` program with `gcc` and O0 optimization level.
   (b) Run the program doing timing with the GNU time command. Remember to save the output result of the original `pi.c` program to compare its result to future optimizations of the program in a first run (without timing). Note that you may want to repeat several times the measure in order to be sure that the timing results are similar. In order to explore different execution contexts, do the experiments redirecting the output to NFS, CIFS, Local disk and `/dev/null`. Is there any big difference in the results? Justify your answer.
   (c) Keep the information of the timing executions and try to understand the obtained result: what is %CPU?.
   (d) Which another accounting tool you could use to measure elapsed time that we can indicate it the number of times to execute the program? Which another accounting tool you could use to measure the total number of cpu cycles using the hardware counters, and then cpu time?
4. Now, compile the `pi.c` program using `gcc` and O0, O1, O2 and O3 optimization level flags
   (a) Using the best execution context, run and **check** that pi obtained with O0-3 optimization levels obtain the same result.

(b) Compute the speed-up of *user time + system time* of the program compiled using O3 compared to the program compiled with O0.

(c) Compute the speed-up of *elapsed time* of the program compiled using O3 compared to the program compiled with O0.

## 1.2 Profiling tools

### 1.2.1 Profiling with `gprof` and `Valgrind`

5. Compile the `pi.c` program with `gcc`, O0 optimization level, `gprof` profiling option `-pg` for the `gprof` experiments, and the debug option (`-g`), with and without static link (`-static`). Using `gprof` and `Valgrind` answer the following questions. Note: Look at the `pid` number of the output of valgrind to know which is the callgrind.out.pid file that you should use in `callgrind_annotate` or `qcachegrind`.

   (a) Which is the most invoked routine (of the program or not) by the program?

   (b) Which is the most CPU time consuming routine of the program?

   (c) Which is the most CPU time consuming source code line of the program?

   (d) Does it appear the system mode execution time in the `gprof` output? and in the `Valgrind` output?

6. Compile the `pi.c` program with O3 optimization level and profile the execution using `gprof`

   (a) Which differences you can observe looking at the *flat profile* (significant changes on the routine weights, routines that disappear/appear,...)?

   (b) Do you know why there are those differences? Hints: look at the assembler... using objdump, gcc, etc.

### 1.2.2 Profiling with `oprofile`

7. Compile your `pi.c` program using `gcc` and O0 optimization level and the debug option. Perform two `oprofile` of the `pi.c` program varying the counter value that indicates the frequency of the sample of the event to count cpu cycles (look at the output of `ophelp` command to figure out which is the name of the event and the minimum sampling frequency). Use values 750000 and 100000: frequency is 1/counter. Compare the results obtained with `opreport -l`.

   (a) Why do you think that there are those differences in the *samples* column?

8. Compile your `pi.c` program using `gcc` and O3 optimization level and the debug option. Perform a `oprofile` of the `pi.c` program that indicates the frequency of the sample of the event to count cpu cycles is 1/100000. Compare the results obtained with this profiling to the profiling obtained in previous exercise with the same frequency. Use `opreport` and opannotate to compare them.

   (a) What are the main differences? Why?

### 1.2.3 Profiling with `perf`

9. Compile your `pi.c` program using `gcc` and O3 optimization level and the debug option. Perform a `perf` of the `pi.c` program at the maximum frequency of sampling allowed for `cycles`. Compare the results obtained with this profiling to the profiling obtained in previous exercise with the same frequency. Use `perf` report and annotate. You should see the same kynd of differences than for `oprofile`.

   (a) What are the main differences? Why?

### 1.2.4 Profiling using `pin`

10. Using `pin`, indicates the number of instructions executed when compiling `pi.c` with `gcc`, using O0 and O3 optimization levels, when obtaining the first 10000 decimals of the $\pi$ number.

| | gcc O0 | gcc O3 |
|---|---|---|
| Total Instructions | | |
| Stack Reads | | |
| Stack Writes | | |
| `IDIV + DIV` | | |
| `IMUL + MUL` | | |

11. Looking at the previous `pin` output and the profiling and timing information, it is clear that there have been a significant change between the O0 and O3 binary. Previous table provides very interesting information about the possible optimizations that the compiler could do.

### 1.2.5 Instrumenting with system calls and PAPI

12. The `pi_times.c` program uses the system call `times()` in order to show the execution time (decomposed in user mode and system mode) for each call to `calculate()`.

    (a) Observe the differences between `pi.c` and `pi_times.c` programs and how the system call `times()` is called. Indicate if the time shown by the program is CPU time or elapsed time. Can the system call `times()` provide both CPU and elapsed time?

    (b) Modify `pi_times.c` program so that `clock_gettime()` system call is used instead of `times()` system call in order to compute elapsed time. Can `clock_gettime()` provide both CPU and elapsed time? Do you have less or more precision compared to "times" results?

    (c) (if installed at Lab) Modify `pi_times.c` program so that `PAPI` is used in order to provide total number of cycles of the program `PAPI_TOT_CYC` and the CPU time of `calculate` function. Note that `papi_avail` provides information about the frequency of the processor. You may have to specify $-L$ and $-I$ information at compile time with the information of the path where the library is installed (if not in the PATH environment variable), in addition to the `-lpapi` library.

## 1.3 Tracing Tools

### 1.3.1 `strace`

13. Compute how many system calls are invoqued by the `pi.c` program. Compute how much time those system calls spend.

14. Which information show `strace -e trace=write` command, for `pi.c` program?

### 1.3.2 `ltrace`

15. Compute how many times `fprintf()` library routine is called by an execution of the program `pi.c`.

# 2

# Automatization and data managment tools

16. The objective of this exercise is to prepare a script that can help you to automatize the following steps of the optimization methodology: execution of the original code (several executions) to obtain timing information and golden output, execution of the optimized code to check the result first, and then several executions to obtain timing information of the optimized code. Finally, speedup computation.

    In order to achieve this objective, do the following steps:

    (a) Create a script that, given the name of optimized executable program (first argument of the script), executes the program $N$ times (second argument of the script), and shows the minimum, maximum and average elapsed time and the minimum, maximum and average cpu time of the $N$ executions at the end of the script.

    (b) Add a new feature to the script to accept arguments of the program.

    (c) Add a new feature to the script to accept a new argument with the original executable program. This original program will be used to generate the golden output, check the correct result of the optimized executable program, and also to compute the speedup of the optimized version compared to the original one. If the result check is ok, the script will show the minimum, maximum and average time of both programs, and the speedup of the optimized code compare to the original code, using the average time, for elapsed and cpu time. If the result check is not ok, the script should indicate this fact.

    Test your script running it in the following cases:

    (a) Run your final script using `popul.O0.g` (O0 optimization flag with debug information) as the original program, and `pi.O0.g` as the optimized program. Number of executions $N = 3$.

    (b) Run your final script using `pi.O0.g` (O0 optimization flag with debug information) as the original program, and `pi.O0.g` as the optimized program. Number of executions $N = 3$.

    (c) Run your final script using `pi.O0.g` (O0 optimization flag with debug information) as the original program, and `pi.O3.g` as the optimized program. Number of executions $N = 3$.

17. It is up to you if you want to introduce more features, use of `perf` stat, etc.