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

D. Jiménez, E. Morancho and À. Ramírez

February 19, 2016

Index

1

Tools to Measure

You will analyze `pi.c` in all the exercises except the first one where you will use `popul.c`. You will have to run it without any parameter. In that case, `pi.c` computes the first 10000 decimals of the π number and writes them in the stardard output.

1.1 Accounting Tools

1. Compile `popul.c` program with `gcc` and answer the followings questions:
   (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) Repeat the experiment redirecting the output to `/dev/null`.
   (d) Do you know why you are observing some differences? (*elapsed time*, *%CPU*, *user time*,...)
   (e) What can you say regarding to the measures you have done once you have seen the results? Which is the best experimental setup for your future experiments?
2. Compile `pi.c` program with `gcc` and O0 optimization level, run the program, do timing with the GNU time command and explain the obtained result. 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.
3. Compile the `pi.c` program using `gcc` and O0 and O3 optimization level flags
   (a) Run and check that pi obtained with O0 and O3 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, for the best experimental context used in previous exercises.
   (c) Compute the speed-up of *elapsed time* of the program compiled using O3 compared to the program compiled with O0, for the best experimental context used in previous exercises.

1.2 Profiling tools

1.2.1 Profiling with gprof

4. Compile the `pi.c` program with `gcc`, O0 optimization level, `gprof` profiling option `-pg`, and the debug option (`-g`). Using `gprof`, answer the following questions:
   (a) Which is the most invoked routine by the program?

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

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

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

5. Repeat the previous experiment compiling now with O3 optimization level

   (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?

      - It may be useful for you to look at the assembler code generated using O0 and O3 optimization levels.
      - Also, observe the information given by `gprof` at source code line level.

1.2.2 Profiling with `oprofile`

6. 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 `CPU_CLK_UNHALTED` event (fist counter that appears in the output of `ophelp` command). Use values 750000 and 7500: 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?

7. 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 `CPU_CLK_UNHALTED` event is 1/7500. Compare the results obtained with this profiling to the profiling obtained in previous exercise with the same frequency. Use opreport, opannotate and ocount.

   (a) What are the differences? Why?

1.2.3 Instrumenting with system calls

8. 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 `getrusage()` system call is used instead of `times()` system call. Indicate if `getrusage()` can provide both CPU and elapsed time? Do you have less or more precision compared to “times” results?

   Considerations:

   - `struct timeval` struct uses to be defined at `/usr/include/bits/time.h` file.
   - In order to obtain time format 1.035 seconds, the `tv_sec` field of the `struct timeval` struct will have value `1` and `tv_usec` field will have value 35000 (0.035 seconds are 35000 microseconds).

2

Automatization and data managment tools

9. Create an script that automatizes the execution of the program `pi.c` for `NMIN` up to `NMAX` (with `NSTEP` step) number of decimals. The script has the following arguments:

   - Executable program of the original (no optimized) `pi.c`.
   - Executable program of the possible optimized `pi.c` (to be done in next sessions).
   - `NMIN` and `NMAX` : minimum and maximum number of decimals.
   - `NSTEP` value: loop step.
   - `NEXEC` value: number of executions to do average of execution time.

   First, the script should check the correct result for each execution of the optimized program, comparing its results to the original program results. If there is any difference the script should give a message "No correct results for N=Value" and stop the execution of the script. If everything is fine, it should run the optimized program and generate a text file with the following format for each line:

   ```
   number_of_decimals elapsed_time
   ```

   For instance:

   ```
   500 1.3454
   1000 2.1234
   1500 3.9834
   ...
   ```

   Where elapsed time is the average of the elapsed time of the `NEXEC` executions done.

   Run the script with `NMIN`= 500, `NMAX`= 10000 and `NSTEP`= 500, and do the following figures:

   (a) One figure that shows *elapsed time* (Y axis) function of the number of decimals computed (X axis) for the original `pi.c` compiled with "-O0", "-O1", and "-O3 -march=native" compiler options. You can create a figure for each case or all cases in the same figure.

   (b) Another figure that shows the *elapsed time/number of decimals computed* (Y axis) function of the number of decimals computed (X axis) for the `pi.c` program for "-O0", "-O1", and "-O3 -march=native" compiler options. You can create a figure for each case or all cases in the same figure.

   (c) Explain the execution differences and the shape of the figures.

   (d) Prepare an script (similar to a regression test) to run your script with two examples to test it:

i. First test a correct program: The original executable program should be the `pi.c` compiled with `-O0`. The optimized executable program should be the `pi.c` compiled with `-O3`. `NEXEC` value should be three. The rest of parameters can be: `NMIN`= 500, `NMAX`= 1000 and `NSTEP`= 500.

ii. Second test an incorrect program: The original executable program should be the `pi.c` compiled with `-O0`. The optimized executable program should be the `popul.c` compiled with `-O3` (YES! `popul.c`, it should be incorrect :). `NEXEC` value should be three. The rest of parameters can be: `NMIN`= 500, `NMAX`= 1000 and `NSTEP`= 500.

Note that you may want to include the generation of the figures into the script, so that everything will be automatically generated (hints: bc, jgraph, gnuplot are programs that may help you to automatize the generation of the figures).