HPC Tools 2024, Exercise 1 solutions

Bruno Oliveira Cattelan, Laurent Chôné

Problem 1

In this exercise we investigate the info contained in the file `/proc/cpuinfo`. One straightforward way to proceed would be to simply print out the content of the whole file and read it out, using `cat /proc/cpuinfo`. Since we are specifically looking for the frequency, we can distill the information using `grep` (as an example below, the model name sometimes contains the base frequency, but sometimes not):

Personnal workstation (under load):

```
$ grep -E 'name|Hz' /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
cpu MHz         : 3600.000
model name      : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
cpu MHz         : 3600.000
model name      : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
cpu MHz         : 3599.999
model name      : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
cpu MHz         : 3599.999
```

Pangolin:

```
$ grep -E 'name|Hz' /proc/cpuinfo
model name      : Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
cpu MHz         : 2992.968
model name      : Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
cpu MHz         : 2992.968
model name      : Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
cpu MHz         : 2992.968
model name      : Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
cpu MHz         : 2992.968
```

Turso:

```
$ grep -E 'name|Hz' /proc/cpuinfo
model name      : AMD EPYC 7452 32-Core Processor
cpu MHz         : 2345.593
model name      : AMD EPYC 7452 32-Core Processor
cpu MHz         : 2345.593
model name      : AMD EPYC 7452 32-Core Processor
cpu MHz         : 2345.593
model name      : AMD EPYC 7452 32-Core Processor
cpu MHz         : 2345.593
```

Remark: systems such as Pangolin, Turso, Windows Subsystem for Linux, Google Colab,... are virtualised. As a consequence the frequency obtained in cpuinfo is the base frequency, not the instantaneous frequency.

Problem 2

In this problem, we want to measure the effective frequency of CPU cores on a Linux system. The program provided returns measurements of the frequencies of each CPU core every second for the number of time passed as argument. A simple way to process the measurement is collecting them in a file to be analysed later with your language of choice, like so:

```
$ ./get_freq 100 &> freq.txt
```

The statement of problem 1 emphasized the need to ensure sufficient workload during the measurement, so that the cores do not go into power saving mode. We evidence this with two different experiment. In the "idle" case, the computer is partially loaded by normal usage (IDE, web browser). In the "loaded" case, a dummy program is run in parallel on all cores.

The table and figures illustrate the results: in the "idle" case, the cores mostly work at the base frequency, occasionally entering power saving modes bringing the frequency as far down as 800 Mhz. In the "loaded" case, all cores function at the "turbo" frequency for the duration of the experiment.

f (MHz)	idle			loaded		
	min	mean	max	min	mean	max
CPU0	800.055	3041.333	3400	3599.999	3600	3600
CPU1	800.033	3014.175	3400	3599.999	3600	3600
CPU2	800.084	3199.278	3400	3599.999	3599.999	3600
CPU3	800.045	3133.521	3643.106	3599.999	3599.999	3600

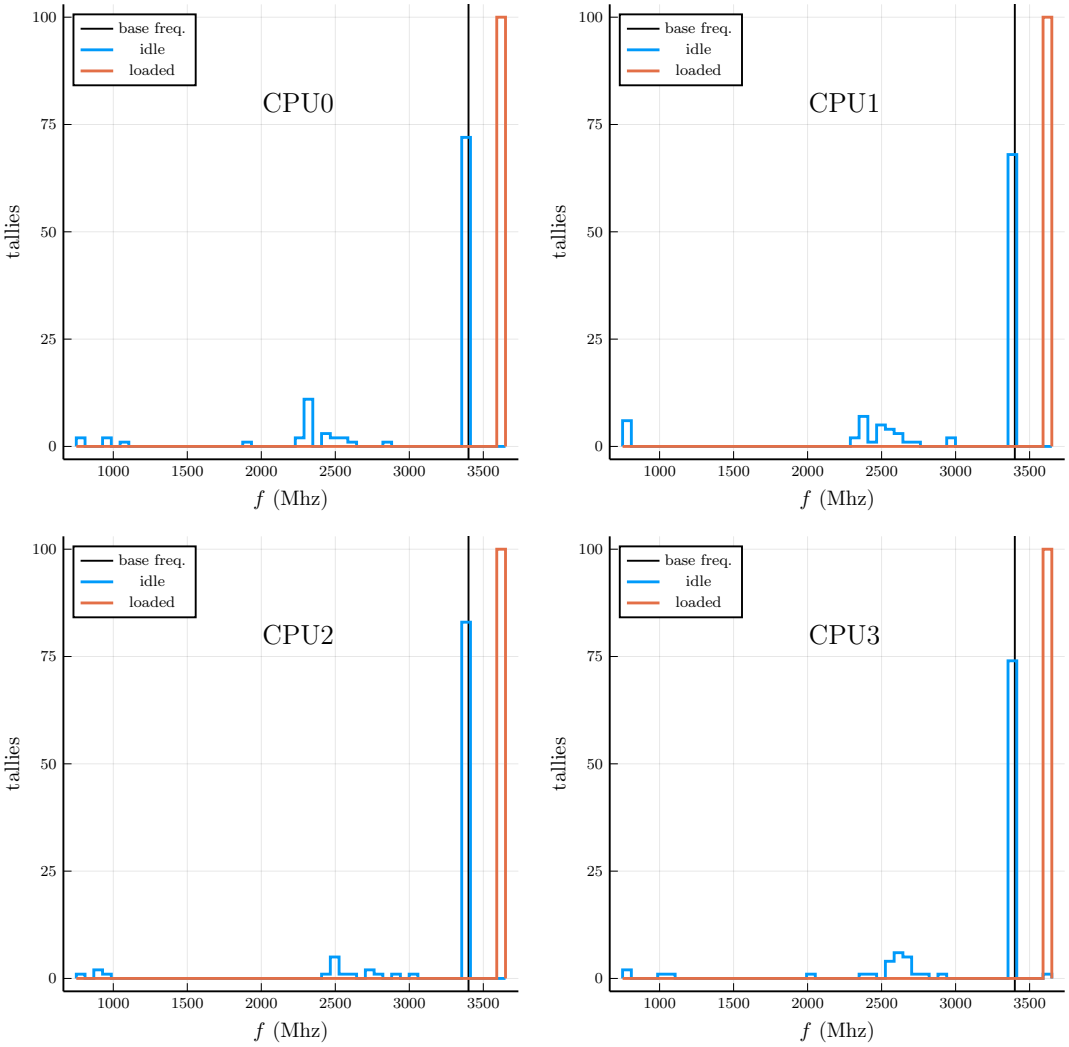

Figure 1: Histograms of the clock frequencies during the experiment, idle and under full load. The vertical black line is the base frequency announced by the vendor.

Problem 3

In this exercise we were asked to provide a makefile following the image given in the exercise sheet. In order to execute it one can simply call "make" in the folder or call directly the final command "make target".

Since no files were provided, we are also required to create the dummy files used in the earlier dependencies in this example. For that, the first two dependencies "dep_a1" and "dep_b1" make use of echo for creating the file and adding some content. One could also use touch to create the file and then populate it with some data.

Next, "dep_a2" and "dep_b2" copy the data in the files previously created to a new file for each (a2.txt and b2.txt respectively). Finally, we have the command "target" which calls upon the dependencies in order to merge both files into one. I also added a "clean" command. This is good practice, as users can easily call "make clean" in order to start from scratch again.

```
target: dep_a2 dep_b2
	cat a2.txt b2.txt > merged.txt

dep_b2: dep_b1
	cp b1.txt b2.txt

dep_a2: dep_a1
	cp a1.txt a2.txt

dep_b1:
	echo "Content␣of␣b1" > b1.txt

dep_a1:
	echo "Content␣of␣a1" > a1.txt

clean:
	rm *.txt
```

Problem 4

For this problem we are asked to demonstrate what happens when dependencies follow the order presented in the exercise sheet. Simply put, if not taken into account this type of dependencies can create an infinite loops as the dependencies cannot ever be fulfilled.

With taking into account I mean either preventing such cases or somehow identifying them before hand. If we try to create a dependency loop in our makefile, we receive the following error: "make: Circular c1 <- c3 dependency dropped". Meaning that the loop was identified by make and not executed.