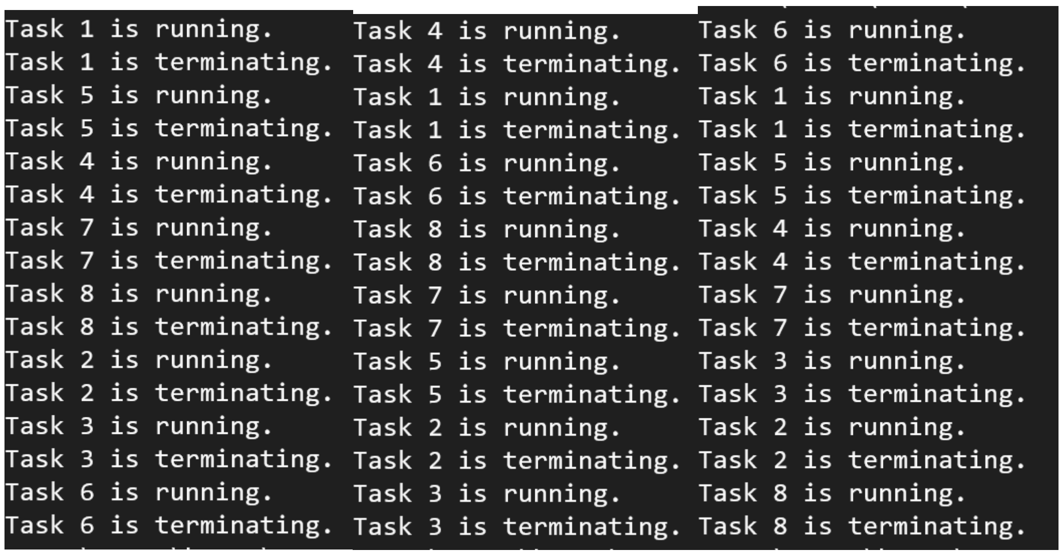
Assignment 1 - Introduction to Parallel Programming

Bernat Xandri Zaragoza

1. **Exercise 0**: “Hello, World!” in C++ (0 points; no need to submit)
   See the source code
2. **Exercise 1**: Concurrency and Non-Determinism (0.5 points)

   Run the program several times. Discuss and explain the observed output. Are there other possible outputs that you did not (yet) observe?

   We can observe that all 8 independent threads are executed, but the order in which the execution is printed out varies between executions. Also, all the threads are finished before the next one starts.



```
Task 1 is running.     Task 4 is running.     Task 6 is running.
Task 1 is terminating. Task 4 is terminating. Task 6 is terminating.
Task 5 is running.     Task 1 is running.     Task 1 is running.
Task 5 is terminating. Task 1 is terminating. Task 1 is terminating.
Task 4 is running.     Task 6 is running.     Task 5 is running.
Task 4 is terminating. Task 6 is terminating. Task 5 is terminating.
Task 7 is running.     Task 8 is running.     Task 4 is running.
Task 7 is terminating. Task 8 is terminating. Task 4 is terminating.
Task 8 is running.     Task 7 is running.     Task 7 is running.
Task 8 is terminating. Task 7 is terminating. Task 7 is terminating.
Task 2 is running.     Task 5 is running.     Task 3 is running.
Task 2 is terminating. Task 5 is terminating. Task 3 is terminating.
Task 3 is running.     Task 2 is running.     Task 2 is running.
Task 3 is terminating. Task 2 is terminating. Task 2 is terminating.
Task 6 is running.     Task 3 is running.     Task 8 is running.
Task 6 is terminating. Task 3 is terminating. Task 8 is terminating.
```

3. **Exercise 2**: Shared-Memory Concurrency (1 point)

   Compile the program using: g++ -std=c++11 -Wall -pthread and run it several times. Discuss and explain the observed output.?

   We can observe that every time that we run the program, the output that we observe changes. This is due to the nature of thread scheduling in concurrent programs, which behave in an unpredictable and non-deterministic way. This means that the order in which each thread is executed is going to vary every time.

4. **Exercise 3: Race Conditions vs. Data Races (1.5 points)**
   Study the source code of non-determinism.cpp and shared-variable.cpp. Does either program contain a race condition? Does either program contain a data race? Justify your answers.

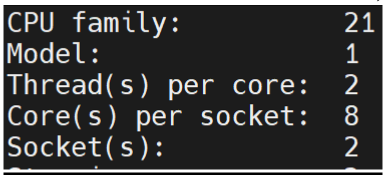
We can observe that there’s no race conditions in any of the scripts, since there’s no possible error that is dependent on the sequence of the threads.

There’s also no Data Races. In the non-determinism.cpp is obvious, since there’s no access to shared data, that a data race can not happen. In shared-variables.cpp a data race could happen, but both shared data (run and x) are protected by the mutex, ensuring that the access to it is synchronized.

5. Exercise 4: Multicore Architectures (1.5 points)

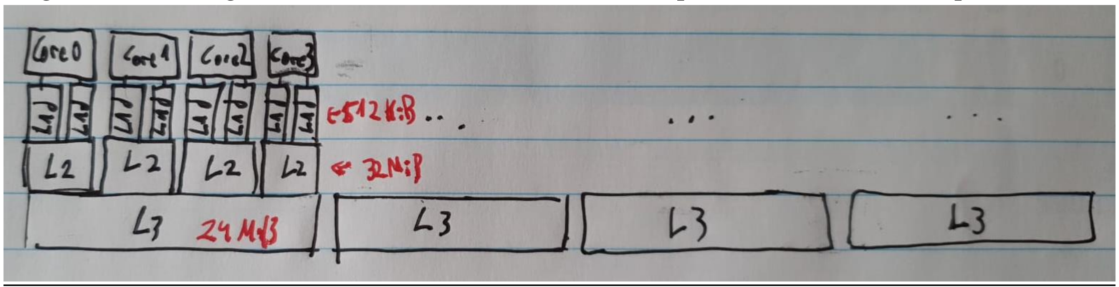
A) Use the lscpu command to obtain some information on the CPU architecture of your lab machine. How many (logical) CPUs does your machine have? How many (physical) processors (i.e., sockets) and processor cores? How many hardware threads are running on each core?

It has 32 CPUs. It has 2 sockets, to a total of 16 cores (8 per socket). It has 2 threads per core.



```
CPU family:          21
Model:               1
Thread(s) per core:  2
Core(s) per socket:  8
Socket(s):           2
```

B) The diagram below shows the CPU architecture and cache layout of an Intel I5-450M processor. However, tussilago has a processor from AMD, not from Intel. Use the information gained from lscpu and lscpu -p to draw (either by hand or with a drawing program) a similar diagram for tussilago. What are the main differences compared to an Intel-based processor?



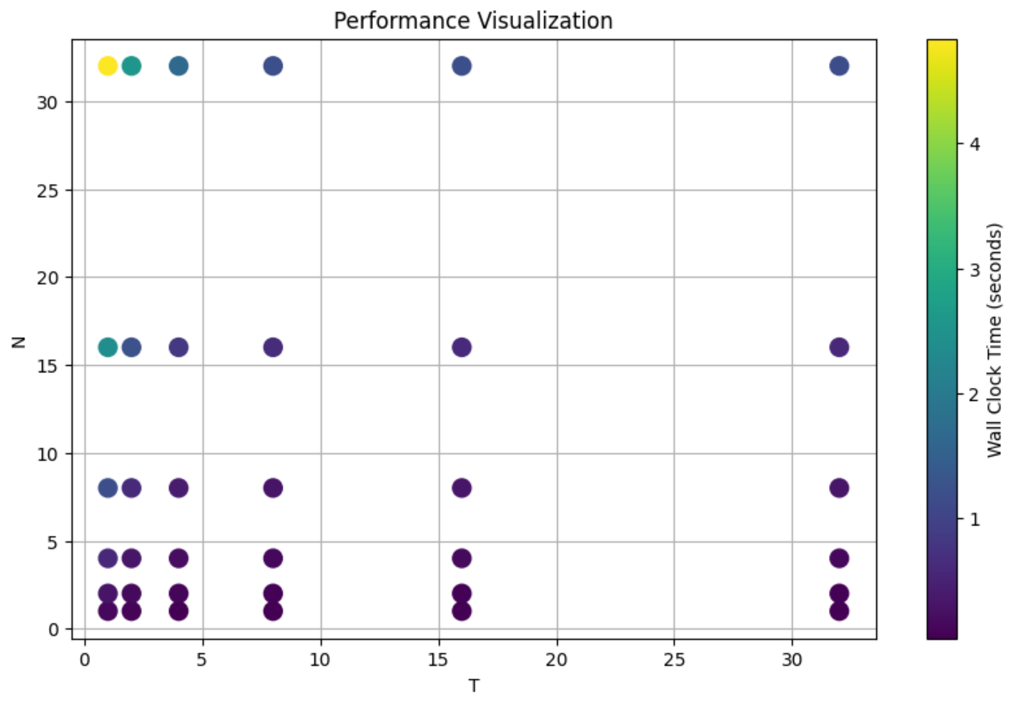
6. Exercise 5: Performance Measurements (1.5 points)

a) Using a (departmental) machine with at least 16 physical cores, run the program for each combination of T ∈ 1, 2, 4, 8, 16, 32 and N ∈ 1, 2, 4, 8, 16, 32. (You can either do this manually or write a simple script to automate it.) Note the reported run times.

T	N	Wall Clock Time (seconds)
1	1	0.14674
1	2	0.295492
1	4	0.595864
1	8	1.19468
1	16	2.39901
1	32	4.83571
2	1	0.0774055
2	2	0.153971
2	4	0.31018

2	8	0.610617
2	16	1.24127
2	32	2.5636
4	1	0.0545223
4	2	0.109392
4	4	0.194312
4	8	0.417004
4	16	0.819943
4	32	1.65617
8	1	0.0406747
8	2	0.0723469
8	4	0.149344
8	8	0.285584
8	16	0.629965
8	32	1.19089
16	1	0.038487
16	2	0.0754245
16	4	0.152206
16	8	0.301785
16	16	0.60194
16	32	1.18196
32	1	0.0387839
32	2	0.0716162
32	4	0.141155
32	8	0.303225
32	16	0.583558
32	32	1.15057

b) Visualize your measurements in a chart (graph), e.g., in a line chart or three-dimensional graph. You are encouraged to use suitable software (such as LibreOffice Calc, Excel, etc.) to produce the chart.



Plotted with python

c) Discuss and attempt to explain your measurements.

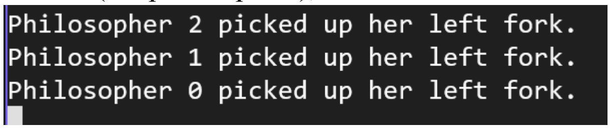
We are observing Parallel slowdown in the extreme cases, where the parallelization beyond a certain point causes the program to run slower

7. Exercise 6: Dining Philosophers (2 points)

a) Run the program several times for different values of N (2 ≤ N ≤ 10). What output do you observe? (You can use Ctrl+C to interrupt the program.) Discuss and explain (as far as you can) the observed output.

We can observe the different philosophers doing the steps described in the exercise in other to eat. The order that this progression follows is random, and varies from execution to execution, because of the reasons described before.

We observe, in all executions, that after a random period of time, it reaches a block situation where it can not continue anymore. This is caused by a problem in the synchronizations of the threads (the philosophers), that lead to a locked situation.



```
Philosopher 2 picked up her left fork.
Philosopher 1 picked up her left fork.
Philosopher 0 picked up her left fork.
```

b) Devise and implement a solution to the dining philosophers that does not suffer from the problem(s) you observed in part a).

A simple solution to this lock situations that can occur when more than one thread accesses the first resource simultaneously, and then they are all waiting for the second one (which is now accessed by another thread) would be to alternate which fork is grabbed first by each thread, ensuring this way that the lock situations observed before can not happen. This small modification breaks the topological dependency between how the accessing threads are sitting.

This way, if a thread grabs the first fork (let's say it's the left one for him), both threads next to them would start picking up the right fork, so that in this case, it can not happen that one of these threads sitting next to it would pick the fork that he's going to grab next.

After implementing this change, we don't observe locked situations anymore.