IPS Instituto Politécnico de Setúbal Escola Superior de Tecnologia de Setúbal	COMPUTAÇÃO PARALELA E DISTRIBUÍDA 2022/2023
	Laboratório 02: Introdução ao Paralelismo em Python

Objetivos do laboratório

Pretende-se que o aluno:

1. Conheça melhor o ambiente de execução do interpretador Python
2. Conheça as principais bibliotecas para código **multi-threaded** e **multi-processo** em Python
3. Seja capaz de decidir quando utilizar uma ou outra funcionalidade para suas aplicações

`Links para consultar:`

- https://realpython.com/python-gil/#why-hasnt-the-gil-been-removed-yet
- https://www.packtpub.com/product/parallel-programming-with-python/9781783288397
- https://docs.python.org/3/library/threading.html?highlight=threading#module-threadin
- https://docs.python.org/3/library/multiprocessing.html?highlight=multiprocessing#multiprocessing
- https://www.parallelpython.com/
- https://docs.python.org/3/library/multiprocessing.html

Nível 1

Escreva o código apresentado e responda as questões colocadas.

```
import time

COUNT = 50000000

def contar_decrescente(ls, li):

    while ls > li:

        ls -= 1

inicio = time.time()

contar_decrescente(COUNT, 0)

fim = time.time()

print(f'Tempo em segundos: {fim - inicio}')
```

- Explique a função deste programa;
- Caracterize o programa em termos de **CPU-bound** ou **I/O Bound**.

Nível 2

O código apresentado implementa um novo algoritmo do mesmo problema do exemplo anterior, mas utiliza uma estratégia baseada em **threads.** Escreva o programa e responda as questões colocadas.

```
# threads Múltiplos

import time
from threading import Thread

COUNT = 50000000


def contar_decrescente(ls, li):
    while ls > li:
        ls -= 1


inicio = time.time()

thread_1 = Thread(target=contar_decrescente, args=(COUNT, COUNT // 2,))
thread_2 = Thread(target=contar_decrescente, args=(COUNT // 2, 0,))

thread_1.start()
thread_2.start()

thread_1.join()
thread_2.join()

fim = time.time()

print(f'Tempo em segundos: {fim - inicio}')
```

- Explique a estratégia seguida neste algoritmo;
- Compara o resultado obtido, neste código, com o obtido no código do nível 1 e apresente uma explicação.

Nível 3

O código apresentado implementa um novo algoritmo do mesmo problema dos exemplos anteriores, mas utiliza uma estratégia baseada em **processos.** Escreva o programa e responda as questões colocadas.

```
import time
from multiprocessing import Pool


def contar_decrescente(ls, li):
    while ls > li:
        ls -= 1


if __name__ == '__main__':
    COUNT = 50000000
    pool = Pool(2)
    start = time.time()
    pool.apply_async(contar_decrescente, [COUNT//2, 0])
    pool.apply_async(contar_decrescente, [COUNT, COUNT // 2])
    pool.close()
    pool.join()
    end = time.time()
    print('Tempo em segundos: ', end - start)
```

- Explique a estratégia seguida neste algoritmo;
- Compara o resultado obtido, neste código, com o obtido nos exemplos anteriores e apresente uma explicação.

Nível 4

O código apresentado mede o tempo gasto na criação e ordenação de conjuntos de vetores, com diferentes dimensões, utilizando processamento sequencial e multiprocessamento. Escreva o programa e responda as questões colocadas.

```
import multiprocessing
import random
from timeit import default_timer as timer

def criar_e_ordenar(n):
    rand = random.Random(50)
    x = [rand.randint(0, 100) for _ in range(n)]
    x.sort()
    return x

if __name__ == "__main__":
    numero_CPUs = multiprocessing.cpu_count()
    print(f'Número de CPUs: {numero_CPUs}')
    vetores_a_gerar = [2, 4, 6, 15]
    dimensoes_dos_vetores = [10**2, 10**3, 10**4, 10**6]
    for numero_de_elementos in dimensoes_dos_vetores:
        print(f'Número elementos do vetor: {numero_de_elementos}')
        for qtd_vetores_a_gerar in vetores_a_gerar:
            print(f'\tQuantidade de vetores a gerar e ordenar:
{qtd_vetores_a_gerar}')
            dimensoes = []
            for i in range(qtd_vetores_a_gerar):
                dimensoes.append(numero_de_elementos)
            # dimensoes1 = [d for i in range(qtd_vetores_a_gerar)]
            # print(dimensoes ,dimensoes1)
            # Aplicar a função sequencialmente
            resultado = []
            inicio = timer()
            for d in dimensoes:
                resultado.append(criar_e_ordenar(d))
            # resultado = [createandsort(d) for d in dimensoes]
            fim = timer()
            print("\t\tTempo para ordenação sequencial: ", fim - inicio)
            # print(resultado)
            # print([createandsort(d) for d in dimensoes])

            # Utilizando multiprocessamento
            pool = multiprocessing.Pool(processes=numero_CPUs)  # Usa o número de
cores físicos da sua máquina
            inicio = timer()
            resultado = pool.map(criar_e_ordenar, dimensoes)
            fim = timer()
            print("\t\tTempo para ordenação paralela: ", fim - inicio)
```

- Diga o faz cada uma das seguintes instruções:
  - pool = multiprocessing.Pool(processes=numero_CPUs)
  - pool.map(createandsort, dimensoes)

- Observe a saída do programa e realize uma análise crítica dos resultados observados. A análise deverá ter em atenção as dimensões dos vetores, a quantidade de vetores gerados e os

tempos observados para a ordenação sequencial e a ordenação utilizando multiprocessamento.

Nível 5

O código apresentado implementa duas formas diferentes para criar **processos.** Escreva o programa e responda as questões colocadas.

```
import multiprocessing as mp
from timeit import default_timer as timer

import numpy as np


def quadrado(nums):
    pnome = mp.current_process().name
    for num in nums:
        resultado = num * num
        print(f"Processo {pnome}; o quadrado do número {num} é {resultado}.")

if __name__ == '__main__':
    numero_CPUs = mp.cpu_count()
    processos = []
    limite_superior = 200
    lista = np.array_split(range(limite_superior), numero_CPUs)
    print('Início do multiprocessamento')
    inicio = timer()
    for i in range(numero_CPUs):
        processo = mp.Process(target=quadrado, args=(lista[i],))
        processos.append(processo)
        processo.start()
    for processo in processos:
        processo.join()
    fim = timer()
    print(f"Multiprocessamento completo em {fim-inicio} segundos")
    print('Início do multiprocessamento com pool')
    pool = mp.Pool(processes=numero_CPUs)
    inicio = timer()
    resultado = pool.map(quadrado, lista)
    fim = timer()
    print(f"Multiprocessamento completo em {fim - inicio} segundos")
```

- Diga o faz cada uma das seguintes instruções:
  - lista = np.array_split(range(limite_superior), numero_CPUs)
  - processo = mp.Process(target=quadrado, args=(lista[i],))
  - processos.append(processo)
  - processo.start()
  - processo.join()

- Observe a saída do programa e verifique quantos processos foram utilizados no processamento:
  - Sem pool
  - Com pool

- Realize uma análise crítica dos resultados observados quando alterar significativamente o número de valores a calcular. A análise deverá ter em atenção a quantidade de valores, o número de processos utilizados e o impacto na performance.