Multiprocesamiento con python

1. OBJETIVOS

* Analizar el tiempo de ejecución de correr scripts en paralelo y de manera secuencias
* Ver las librerías que nos permiten realizar el procesamiento en paralelo.
* Instalacion y configuracion de conda, con las diferentes librerías de python.

2. INTRODUCCIÓN

Multiprocesamiento y Multihilo las principales diferencias entre estos dos términos son.

El **multiprocesamiento** son múltiples procesos que se ejecutan utilizando uno o más procesadores.

**Multihilo** es el proceso en el que múltiples hilos se ejecutan al mismo tiempo en un proceso.

En la práctica se utilizará el paquete de python **multiprocessing**, dicho paquete admite de generación utilizando una API similar al **threading** módulo. El **multiprocessing** paquete ofrece tanto concurrencia local como la concurrencia remota, de manera efectiva, de forma paralela. Debido a esto, el módulo de multiprocesamiento permite al programador aprovechar al máximo los múltiples procesadores en una máquina determinada. Se ejecuta en UNIX y Windows.

Este también introduce un módulo de multihilos. Un buen ejemplo es el **pool**  objeto que ofrece un medio conveniente para paralelizar la ejecución de una función a través de múltiples valores de entrada, distribuyendo los datos en la entrada a través de procesos (paralelismo de datos). El siguiente ejemplo demuestra la práctica común de definir tales funciones en un módulo para que los procesos secundarios puedan importar ese módulo con éxito.

3. EJERCICIOS Y/O PROGRAMAS IMPLEMENTADOS

Practica N°1

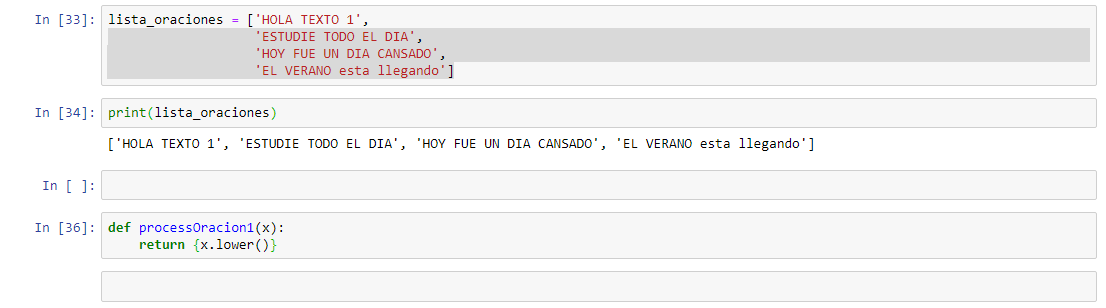


Fig 1. Muestra la creación de una lista de oraciones.

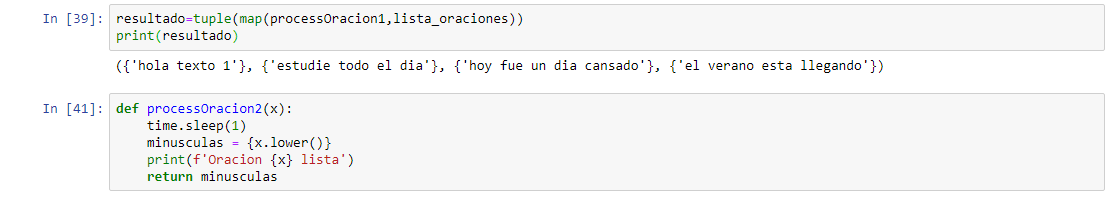


Fig 2. Muestra la definición al recorrer y modificar las oraciones con map() y la función processor Oración 2 calcula el tiempo en cambiar a minúsculas la lista.

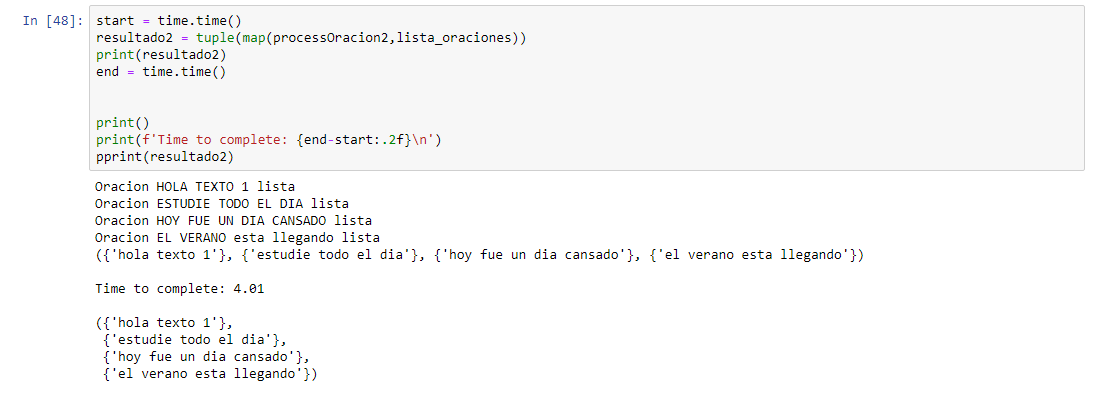


Fig 3. Muestra tiempo de ejecución de manera secuencial, dicho tiempo es mayor.

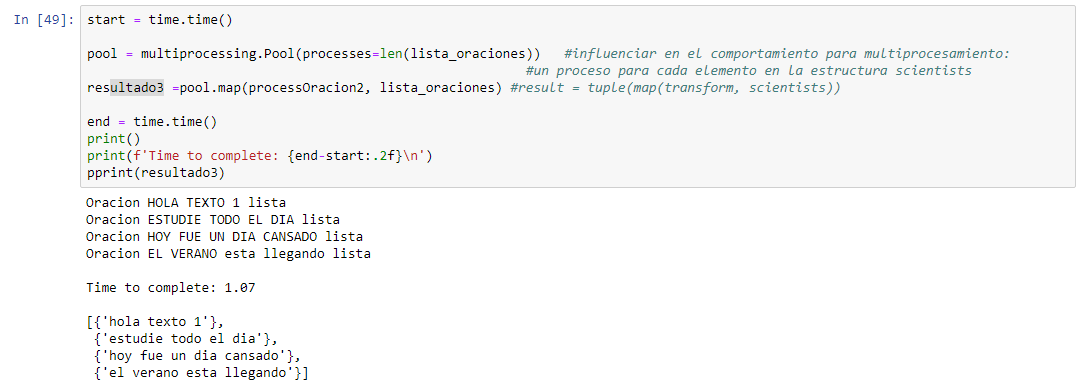


Fig 4. Muestra la ejecución de manera paralela con un tiempo de ejecución menor.

|  |
| --- |
| import collections  from pprint import pprint  import time  Scientist = collections.namedtuple('Scientist', [  'name',  'field',  'born',  'nobel',  ])  scientists = (  Scientist(name='Ada Lovelace', field='math', born=1815, nobel=False),  Scientist(name='Emmy Noether', field='math', born=1882, nobel=False),  Scientist(name='Marie Curie', field='physics', born=1867, nobel=True),  Scientist(name='Tu Youyou', field='chemistry', born=1930, nobel=True),  Scientist(name='Ada Yonath', field='chemistry', born=1939, nobel=True),  Scientist(name='Vera Rubin', field='astronomy', born=1928, nobel=False),  Scientist(name='Sally Ride', field='physics', born=1951, nobel=False),  )  pprint(scientists)  print()  def transform(x):  print(f'Processing record {x.name}')  time.sleep(1)  result = {'name': x.name, 'age': 2019-x.born}  print(f'Done processing record {x.name}')  return result  start = time.time()  result = tuple(map(transform, scientists))  end = time.time()  print()  print(f'Time to complete: {end-start:.2f}\n')  pprint(result) |

Código procesamiento secuencial.

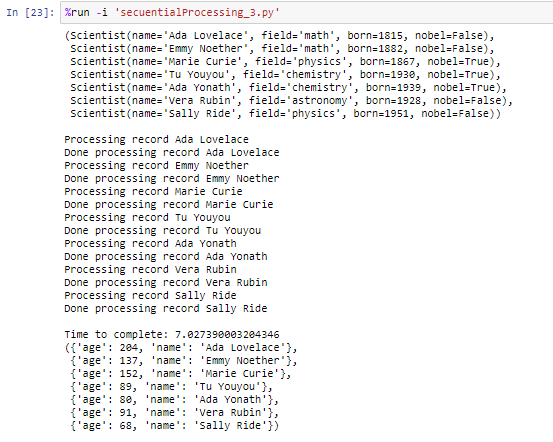


Fig 5. Muestra el tiempo de ejecución de manera secuencial.

|  |
| --- |
| import collections  from pprint import pprint  import time  import multiprocessing  import os  Scientist = collections.namedtuple('Scientist', [  'name',  'field',  'born',  'nobel',  ])  scientists = (  Scientist(name='Ada Lovelace', field='math', born=1815, nobel=False),  Scientist(name='Emmy Noether', field='math', born=1882, nobel=False),  Scientist(name='Marie Curie', field='physics', born=1867, nobel=True),  Scientist(name='Tu Youyou', field='chemistry', born=1930, nobel=True),  Scientist(name='Ada Yonath', field='chemistry', born=1939, nobel=True),  Scientist(name='Vera Rubin', field='astronomy', born=1928, nobel=False),  Scientist(name='Sally Ride', field='physics', born=1951, nobel=False),  )  pprint(scientists)  print()  def transform(x):  print(f'Process {os.getpid()} working record {x.name}') #cada proceso obtiene un identificador o número de proceso  #se puede ver cómo se trabaja en múltiples cores  time.sleep(1)  result = {'name': x.name, 'age': 2019-x.born}  print(f'Process {os.getpid()} done processing record {x.name}')  return result  start = time.time()  pool = multiprocessing.Pool(processes=1) #influenciar en el comportamiento para multiprocesamiento:  #único procesador que realiza todo el cómputo  pool.map(transform, scientists) #result = tuple(map(transform, scientists))  end = time.time()  print()  print(f'Time to complete: {end-start}')  pprint(result) |

Código procesamiento paralelo.

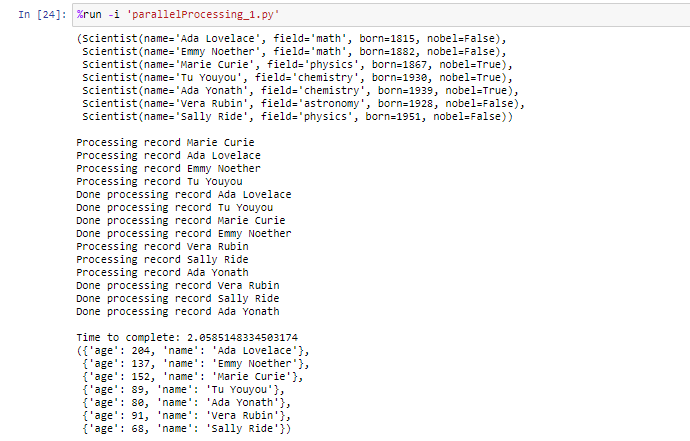


Fig 6. Muestra el tiempo de ejecución de manera paralela.

Practica N°2

Instalar numba

Instalar cudatoolkit

Instalar accelerate

GPU Nvidia Geforce

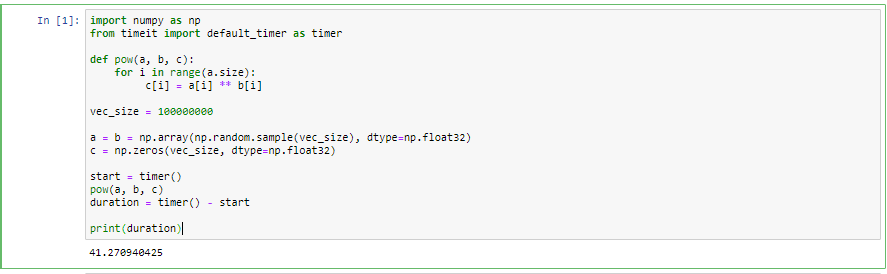


Fig 6. ejecución de algoritmo en paralelo.



FIg 7. Error al ejecutar numba.

### Desarrollar, optimizar e implementar aplicaciones aceleradas por GPU

NVIDIA® CUDA® Toolkit proporciona un entorno de desarrollo para crear aplicaciones aceleradas por GPU de alto rendimiento. Con el kit de herramientas CUDA, puede desarrollar, optimizar e implementar sus aplicaciones en sistemas integrados acelerados por GPU, estaciones de trabajo de escritorio, centros de datos empresariales, plataformas basadas en la nube y supercomputadoras HPC. El kit de herramientas incluye bibliotecas aceleradas por GPU, herramientas de depuración y optimización, un compilador de C / C ++ y una biblioteca de tiempo de ejecución para implementar su aplicación.

Las bibliotecas de CUDA aceleradas por GPU permiten la aceleración en varios dominios, como el álgebra lineal, el procesamiento de imágenes y videos, el aprendizaje profundo y el análisis gráfico. Para desarrollar algoritmos personalizados, puede utilizar las integraciones disponibles con los idiomas y paquetes numéricos más utilizados, así como las API de desarrollo bien publicadas. Sus aplicaciones CUDA se pueden implementar en todas las familias de GPU de NVIDIA disponibles en las instalaciones y en las instancias de GPU en la nube. Mediante el uso de capacidades integradas para la distribución de cómputos en configuraciones de múltiples GPU, los científicos e investigadores pueden desarrollar aplicaciones que escalan desde estaciones de trabajo de GPU individuales hasta instalaciones en la nube con miles de GPU.

Para comenzar, navegue por los recursos de inicio en línea, guías de optimización, ejemplos ilustrativos y colabore con la comunidad de desarrolladores en rápido crecimiento.

4. CONCLUSIONES

* El tiempo de ejecución de un proceso en paralelo es mucho menor que el tiempo de ejecución realizado de manera secuencial.
* multiprocessing de python nos permite un trabajo ágil en la implementación de script de manera paralela.
* En la actualidad muchas empresas están utilizando en multiprocesamiento en paralelo para las siguientes aplicaciones.