Proyecto Individual - Paralelizar con Python

Universidad de Costa Rica
Escuela de Matemáticas
CA0305 - Ciencia de datos II
Estudiante: Javier Hernández Navarro **C13673**

> 🖊 Concepto importante:
>
> Un **proceso** es la ejecución de un programa. El siguiente es un ejemplo de un proceso:
>
> ```
> print('Hola mundo')
> ```

Programación estructurada

Inicialmente, la forma en que se aprende a programar sigue una estructura selectiva, iterativa y **secuencial**. En términos de programación, una estructura selectiva hace referencia al uso de condicionales, por ejemplo, los conocidos `if`, `else`, `else if` o `swith`. Por otro lado, una estructura iterativa hace uso de ciclos, como lo son los `for`, `while` o `do while`.

Ahora bien, la estructura **secuencial** no es más que la ejecución, en secuencia, de múltiples procesos. Estos procesos serán ejecutados **uno tras otro**, todos en un orden previamente definido. Un proceso **no** podrá ejecutarse antes de otro si en el programa no fue indicado de esa manera.

Considere la siguiente función:

```
def realizar_simulaciones(n_simulaciones):
    '''
    Función que realiza simulaciones de manera estructurada.
    Obs: las simulaciones son con base a los datos de stock.

    Parámetros:
        n_simulaciones(int): la cantidad de simulaciones que se quiere
realizar.

    Devuelve:
        simulaciones:(list): lista con todas las simulaciones.
    '''
```

```
    simulaciones = []

    for n in range(n_simulaciones):
        simulacion = np.random.normal(media_muestral, desv_muestral,
len(stock['Adj Close']))
        simulaciones.append(simulacion)

    return simulaciones
```

Note que la función anterior ejecuta el código de manera decreciente, esto se debe a que primero crea la lista, una vez creada la lista procede a obtener arreglos de valores aleatorios, uno por uno, de manera iterada. Es evidente que este es un programa estructurado, ya que cada proceso inicia una vez finalizado el anterior.

Por otro lado, la CPU es la encarga de ejecutar estos procesos, pero, dado que se ejecutan uno a la vez, no se está usando el máximo potencial de la CPU, ya que ella *"espera"* a que se finalice una tarea para empezar la siguiente, acción que no es muy demandante y por ende se ejecuta en un único núcleo. Para lograr aprovechar mejor este recurso, surgieron los procesos paralelos.

Procesos paralelos

Los procesos paralelos le permiten a un programa ejecutar procesos simultáneamente usando múltiples unidades de procesamiento (varios núcleos), esto con el objetivo de reducir el tiempo de ejecución y maximizar el uso de los componentes del computador. En este caso, lo que está sucediendo es que cada núcleo está realizando los procesos de manera independiente.

A continuación se explica como realizar este proceso:

```
import multiprocess as mp # librería para usar procesos paralelos
```

```
n_simulaciones = 200_000
nucleos = mp.cpu_count() // 2 # defino la cantidad de núcleos a usar (4)
simulaciones_por_nucleo = n_simulaciones  // nucleos

# Iniciador de proceso paralelizado
if __name__ == '__main__':

    procesos = []

    # Crea los procesos y les asigna la función con sus respectivos
parámetros
    for n in range(nucleos):
```

```
        proceso = mp.Process(target=realizar_simulaciones, args=
(simulaciones_por_nucleo,))
        procesos.append(proceso)

    # Inicia los procesos de forma paralelizada
    for proceso in procesos:
        proceso.start()

    # Espera a que todos los procesos terminen
    for proceso in procesos:
        proceso.join()
```

El problema acá es que **no se puede acceder directamente a los resultados.**

Solución

Para solucionar el problema anterior, hay que recurrir al objeto `Manager` de la clase `multiprocess`, a su vez fue necesario definir la siguiente función:

```
def simulaciones_paralelizadas(n_simulaciones, nucleo):
    '''
    Función que realiza simulaciones con procesos paralelos.
    Obs: no devuelve nada, modifica directamente la lista que recibe.

    Parámetros:
        n_simulaciones(int): la cantidad de simulaciones que se quiere
realizar.
        nucleo(manager.list): lista paralelizada para guardar los
resultados.
    '''

    for n in range(n_simulaciones):
        simulacion = np.random.normal(media_muestral, desv_muestral,
len(stock['Adj Close']))
        nucleo.append(simulacion)
```

A continuación se explica como realizar el proceso y obtener los resultados:

```
n_simulaciones = 200_000
nucleos = mp.cpu_count() // 2 # defino la cantidad de núcleos a usar

simulaciones_por_nucleo = n_simulaciones  // nucleos

# Iniciador de proceso paralelizado
if __name__ == '__main__':
```

```
    # Crea un objeto manager para poder acceder a los resultados
    manager = mp.Manager()

    # Debe crearse uno por nucleo
    nucleo1 = manager.list()
    nucleo2 = manager.list()
    nucleo3 = manager.list()
    nucleo4 = manager.list()

    # Crea los procesos y les asigna la función con sus respectivos
parámetros
    proceso1 = mp.Process(target=simulaciones_paralelizadas, args=
(simulaciones_por_nucleo, nucleo1))
    proceso2 = mp.Process(target=simulaciones_paralelizadas, args=
(simulaciones_por_nucleo, nucleo2))
    proceso3 = mp.Process(target=simulaciones_paralelizadas, args=
(simulaciones_por_nucleo, nucleo3))
    proceso4 = mp.Process(target=simulaciones_paralelizadas, args=
(simulaciones_por_nucleo, nucleo4))

    # Inicia los procesos de forma paralelizada
    proceso1.start()
    proceso2.start()
    proceso3.start()
    proceso4.start()

    # Espera a que todos los procesos terminen
    proceso1.join()
    proceso2.join()
    proceso3.join()
    proceso4.join()
```

Alternativa: Procesos concurrentes

Los procesos concurrentes tienen la capacidad de ejecutar múltiples procesos al mismo tiempo en el mismo sistema informático (un solo núcleo de CPU) de manera superpuesta. Si bien es cierto, no estamos usando más núcleos, pero se está logrando más que solo *"esperar"* a que termine una acción para iniciar otra.

A continuación se explica realizar este proceso:

```
import threading # librería para usar procesos concurrentes
```

```
resultados_hilos = []

n_simulaciones = 200_000
n_hilos = 10 # ojo, puedo crear más hilos dado que no son núcleos

simulaciones_por_hilo = n_simulaciones // n_hilos

hilos = []

# Crea los hilos y les asigna la función con sus respectivos parámetros
# Inicia inmediatamente cada hilo después de crearlo
for n in range(n_hilos):
    hilo = threading.Thread(target=simulaciones_hilos, args=
(simulaciones_por_hilo,))
    hilos.append(hilo)
    hilo.start()

# Espera a que todos los hilos terminen
for hilo in hilos:
    hilo.join()
```

Resultados

Para obtener los resultados, se tomó el promedio de 5 ejecuciones de cada proceso. Es necesario recalcar que cada simulación genera 2.627 valores aleatorios y a su vez se realizaron 200.000 simulaciones, es decir, cada proceso se probó para realizar 525.400.000 cálculos.

Proceso utilizado	Tiempo promedio de ejecución
Estructurado	15.002s
Pararelizado sin resultados	4.895s
Paralelizado con resultados	25.263s (+ 53.977s)
Concurrente	20.061s

Conclusiones

- Paralelizar los programas adecuados efectivamente mejora los tiempos de ejecución, el problema está si se quieren obtener los resultados, ya que esto prácticamente empeora el código.
- De manera general, son más los costos que los beneficios que proporciona paralelizar, esto debido a su complejidad.

- El uso de hilos es medianamente más fácil que el uso de procesos, sin embargo, los resultados no son tampoco lo suficientemente convincentes como para usarlos.
- La programación paralelizada es más propensa a errores.

Recomendaciones

- No cualquier proceso se puede paralelizar, hay ciertos requisitos que se deben cumplir.
- ¡**No** usar todos los núcleos disponibles! Esto debido a que el sistema operativo también necesita espacio.
- Si se busca mejorar la eficiencia de un programa, lo mejor es hacerlo por medio de vectorización. Totalmente de acuerdo.

Bibliografía

- **Torres, S. R., & Torres, S. R. (2024, March 5).** *Núcleos de un procesador. ¿Qué son y cuantos necesitas?* Impacto TIC. https://impactotic.co/micrositios-tic/chipset-2023/chipset-nucleos-y-por-que-es-importante-su-cantidad/#
- **Verma, P. (2022, August 22).** *Comprender el multiprocesamiento y los subprocesos múltiples en Python*. HackerNoon. https://hackernoon.com/es/entender-multiprocesamiento-y-multihilo-en-python#
- ***Aplicación de la programación concurrente y paralela en Python*. (s.f.).** MyApp. https://www.codeauni.com/comunidad/blog/89/#:~:text=Es%20la%20capacidad%20de%20un,de%20CPU)%20de%20manera%20superpuesta.
- **JavierCastilloGuillen. (s.f.).** *GitHub - JavierCastilloGuillen/Quantitative_Toolbox: On this repository you'll find tools used for Quantitative Analysis and some examples such: MonteCarlo Simulations, Linear Regression, General Data Visualiztions, Time-Series Analysis, etc.* GitHub. https://github.com/JavierCastilloGuillen/Quantitative_Toolbox
- **códigofacilito. (s.f.).** Threads y procesos. https://codigofacilito.com/articulos/threads-procesos#