Programación Paralela: Práctica 4

15 – 19 abril 2024

Las respuestas deben entregarse en un único documento (formato pdf). Este documento debe incluir el nombre completo de todos los integrantes del grupo. Si el ejercicio es de implementación, el código fuente se puede entregar en ficheros .py independientes dentro de un archivo comprimido.

Ejercicio 1

El paralelismo basado en hilos de Python está limitado por el GIL. Para realizar una auténtica ejecución paralela es necesario utilizar procesos. En los siguientes ejercicios vamos a paralelizar con procesos el programa de las prácticas anteriores que descarga todos los ficheros referenciados por una página HTML. Partiremos de esta versión secuencial:

```
import os
import urllib.request
import urllib.parse
import re
import shutil
import time
import multiprocessing

fich_total = 0
bytes_total = 0

def bajar_fichero(direccion):
    global fich_total, bytes_total
    id = multiprocessing.current_process().name
    fich_total += 1
    print(id, fich_total, direccion)
    s = direccion.split('/')
    fichero = 'download/' + s[-1]
    try:
        i = urllib.request.urlopen(direccion)
        with open(fichero, 'wb') as f:
```

```
            shutil.copyfileobj(i, f)
            bytes_total += f.tell()
    except Exception as err:
        print(err)

def bajar_html(url_raiz, urls):
    r = urllib.request.urlopen(url_raiz)
    html = r.read().decode('utf-8', 'ignore')
    for m in re.finditer(r'src\w*=\w*"([-_./0-9a-zA-Z]*)"',
html, re.I):
        direccion = urllib.parse.urljoin(url_raiz,
m.group(1))
        urls.append(direccion)

def main():
    url_raiz = 'http://www.uv.es'
    if not os.path.exists('download'):
        os.mkdir('download')

    urls = []
    t1 = time.time()
    bajar_html(url_raiz, urls)
    for direccion in urls:
        bajar_fichero(direccion)
    t2 = time.time()
    tiempo_total = t2 - t1

    print('----------------------')
    print('Tiempo', tiempo_total)
    print('Ficheros', fich_total)
    print('MBytes', bytes_total / (1024.0 ** 2))
    print('Ancho de banda (MBit/s)', (bytes_total * 8 / (1024.0
** 2)) / tiempo_total)

if __name__ == "__main__":
    main()
```

La librería `multiprocessing` proporciona para procesos un interfaz similar al de `threading`. Por ejemplo, para iniciar un proceso se utiliza la rutina `Process` que es similar a `Thread`

```
p = multiprocessing.Process(target=..., args=...)
```

La variable `p` contiene un objeto de tipo `Process` que contiene los métodos `start()` y `join()` con la misma funcionalidad que en `Thread`. Un detalle importante es que la parte correspondiente al proceso principal debe estar dentro de un condicional como:

```
if __name__ == '__main__':
```

para evitar que los procesos hijos ejecuten ese código. Al igual que con hilos, hay que guardar los procesos hijos en una lista y arrancarlos con `start()`. También el proceso principal debe esperar a todos sus hijos con una llamada a `join()`.

- Ejecuta el programa y comprueba que funciona mirando el contenido del directorio `download`. Anota el tiempo de ejecución.

- Haz una copia del programa anterior pero haciendo que cada llamada a `bajar_fichero` se ejecute en un proceso distinto con la función `multiprocessing.Process()`. Compara los tiempos de la versión secuencial con la paralela.

Ejercicio 2

Un problema del ejercicio anterior es que crea tantos procesos como ficheros descargados. Esto es aún menos eficiente que la versión con hilos ya que los procesos son bastante más costosos de crear. Al igual que con hilos, la solución es crear un número fijo de procesos y que el productor coloque los datos en una cola de donde los van cogiendo los consumidores. Típicamente el número de procesos es igual al número de procesadores del sistema que se puede obtener con la función `os.cpu_count()`.

La clase `multiprocessing.Queue` implementa la cola sincronizada para procesos. La cola debe estar compartida entre todos los procesos, para ello se crea en el proceso principal y se pasa como parámetro a los procesos hijos.

Una diferencia importante entre `queue.Queue` y `multiprocessing.Queue` es que la última no tiene los métodos `task_done` y `join`. En su lugar necesitamos un mecanismo para avisar a los consumidores que el trabajo ha terminado. La solución más sencilla es insertar (después de que el productor haya acabado) un valor especial en la cola (por ejemplo, la cadena vacía). De esta forma los consumidores deberán obtener el valor de la cola (con get()) y comprobar si es o no la cadena vacía. Si es la cadena vacía, como ya no quedan ficheros

que descargar deben terminar el proceso y si no continúan descargando el fichero.

- Modifica el programa anterior para crear un número fijo de procesos y las descargas se repartirán entre los procesos utilizando la clase `multiprocessing.Queue`. Recuerda que el proceso principal debe insertar tantas cadenas vacías como procesos para que estos terminen.
- Compara los tiempos de esta versión con las anteriores.

Ejercicio 3

El modelo de ejecución con procesos también se conoce como *paralelismo de memoria compartida*. Cada ejecución tiene su propia copia de la memoria, por eso las variables `fich_total` y `bytes_total` del proceso principal se quedan a cero. Python proporciona las clases `multiprocessing.Value` y `multiprocessing.Array` para compartir datos entre procesos. Estas clases incluyen los bloqueos necesarios para evitar conflictos de acceso. Nota: para hacer operaciones aritméticas con estas variables es necesario usar el atributo `value`.

- Modifica el programa anterior para poder compartir variables entre procesos. Para ello, quita las variables globales `fich_total` y `bytes_total` y cámbialas a parámetros de `bajar_fichero`. Ahora estas variables se deben construir en el proceso principal con:

```
fich_total = multiprocessing.Value('i', 0)
bytes_total = multiprocessing.Value('i', 0)
```

Ejercicio 4

Para implementar paralelismo de procesos en Python existe la clase Pool, un mecanismo es de más alto nivel que Thread. La clase `Pool` representa un conjunto de procesos al que se les pueden enviar trabajo (llamadas a funciones) de diferentes maneras. El reparto del trabajo y la comunicación entre proceso es automática. La clase se crea con el constructor (si no se especifica el número de procesos se utiliza `os.cpu_count()`):

```
pool = multiprocessing.Pool(processes=4)
```

Una de las funciones más sencillas es `apply_async` que lanza una llamada a función con sus correspondientes parámetros. El trabajo se asigna

a uno de los procesos creados por `Pool` y se ejecuta de forma asíncrona. Cuando se ha terminado de enviar trabajos al `Pool` hay que llamar al método `close` y luego al método `join` para esperar a que todos los procesos terminen.

- Haz una versión paralela del programa secuencial del ejercicio 1 usando `Pool`. La llamada para enviar trabajo es similar a la utilizada para crear un proceso:

  ```
  pool.apply_async(bajar_fichero, (direccion, ))
  ```

- Compara los tiempos de la versión secuencial con la paralela.

Ejercicio 5

Una ventaja de la clase `Pool` es que las funciones ejecutadas por los procesos trabaja-dores pueden devolver resultados al proceso principal. Para ello, el método `apply_async` devuelve un objeto de la clase `ApplyResult` de la que se puede extraer con `get` los re-sultados que la función envió con `return`. Obviamente, estos resultados no están disponibles hasta que la función termine.

- Guarda los objetos `ApplyResult` que devuelve `apply_async` en una lista y calcula las sumas totales en el programa principal.

Ejercicio 6

Otro interfaz para ejecutar en paralelo con la clase Pool es la operación `map`. Al igual que la versión secuencial, `map` toma como argumentos una función y un iterable y ejecuta la función con cada uno de los elementos como parámetro. Los valores de retorno se devuelven como una lista.

La operación `map` se espera a que los procesos terminen, por lo que ya no hace falta llamar a `join`. Pero si que es necesario llamar a `close` o `terminate` para destruir los procesos trabajadores. Esto se puede automatizar usando `with` con lo que la llamada a `map` queda así:

```
with multiprocessing.Pool() as pool:

        result = pool.map(bajar_fichero, urls)
```

- Modifica el programa del ejercicio anterior para utilizar `map` en lugar de `apply_async`