Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

31 de julio de 2022

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro

ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: https://github.com/facundobatista/libro-pyciencia

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Parte I
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Procesamiento en paralelo

Desde la invención de la computadora nos hemos acostumbrado a procesar más y más información a través de las mismas. Este procesamiento no sólo fue aumentando en cantidad, sino también en complejidad.

Hay dos formas de poder aumentar la capacidad de procesamiento, de forma "vertical" u "horizontal", términos que son simples metáforas para expresar en el primer caso el aumento de capacidad del procesador en sí (por ejemplo, aumentando su frecuencia de reloj, o memoria *cache* interna), o en el segundo caso hacer referencia a poner varios procesadores uno al lado del otro y distribuir el procesamiento entre ellos.

A esto último lo llamamos "procesamiento en paralelo", y puede hacer referencia tanto a distintos *cores* (o núcleos de silicio) dentro del mismo procesador, distintos procesadores en una computadora, distintas computadoras en un centro de datos, distintos centros de datos, etc.

Cuando el procesamiento de la información implica esperar eventos fuera del procesador (de forma genérica "entrada o salida", que puede ser escribir en disco, consultar un dato a través de la red, etc.) es útil otro concepto que implica repartir el mismo *core* o procesador en distintos procesamientos, aprovechando los "tiempos muertos" de ellos. Aunque con este método logramos concurrencia, no es realmente procesamiento en paralelo.

En las siguientes secciones nos adentraremos en estos conceptos, primero definiremos más extensivamente qué es la concurrencia y mostraremos dos formas de lograrla (a través de *hilos* (o *threads*) y de procesamiento *asincrónico*. La segunda sección versa sobre el procesamiento paralelo real, mostrando distintas técnicas y formas de lograrlo.

1.1. ¿Qué es la concurrencia?

La *concurrencia* es la composición de procesos ejecutados independientemente, logrando un efecto similar a que esos procesos se sucedan al mismo tiempo sin que realmente eso suceda. La concurrencia **no es paralelismo** (que sería la ejecución simultánea real).

¿Hay un recurso que se agota? Necesitamos paralelismo. ¿Pero podemos manejar distintos recursos con un mismo controlador? En este caso podemos usar concurrencia.

Pongamos un ejemplo para diferenciar mejor un caso del otro. Supongamos una cocina de un restaurant.

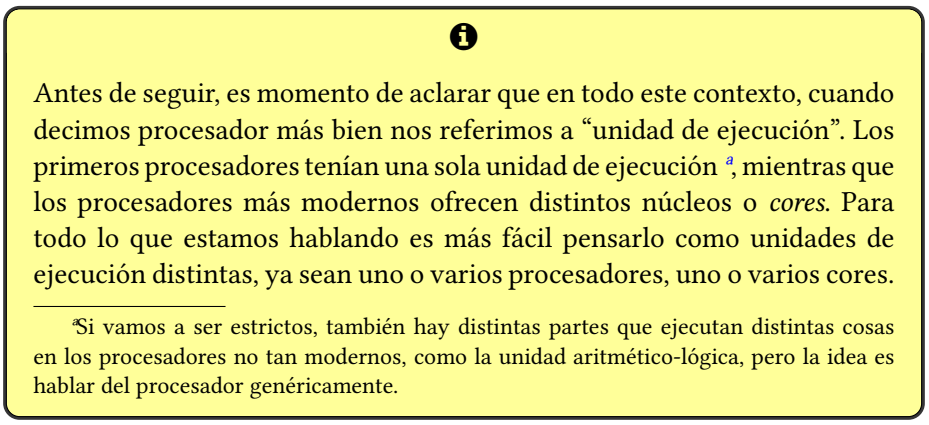
Hay que lavar muchísimos platos por hora, y un lavaplatos no alcanza, aunque esté lavando todo el tiempo. No nos queda otra que poner varios lavaplatos en paralelo, muchas personas lavando platos simultáneamente, una al lado de la otra. Esto es paralelismo.

Por otro lado, tenemos un par de hornos, una freidora, varias hornallas, algún microondas; como a cada una de esas unidades hay que esperarlas, podemos tener una persona (*le chef*) manejando todo eso, ya que por ejemplo se puede poner algo en el microondas y mientras este trabaja, la persona puede retirar algo del horno. Esto es concurrencia.

Llevándolo de nuevo para el lado de la computación, para paralelismo necesitamos varias

unidades de ejecución (las distintas personas lavaplatos en las distintas bachas), mientras que la concurrencia se logra administrando qué proceso entra en ejecución (horno, freidora) en el mismo procesador (chef).

En Python la única forma de usar paralelismo real es con múltiples procesos (vamos a charlar de esto más adelante en la sección de procesamiento en múltiples procesadores 1.4), y aunque tiene toda su complejidad no hay otro mecanismo conceptual que explicar. Por el contrario, para que obtengamos concurrencia, corriendo muchos procesos en el mismo procesador, alguna "magia" tiene que haber, y vamos a proceder a explicarla.



Antes de seguir, es momento de aclarar que en todo este contexto, cuando decimos procesador más bien nos referimos a "unidad de ejecución". Los primeros procesadores tenían una sola unidad de ejecución [a], mientras que los procesadores más modernos ofrecen distintos núcleos o *cores*. Para todo lo que estamos hablando es más fácil pensarlo como unidades de ejecución distintas, ya sean uno o varios procesadores, uno o varios cores.

[a] Si vamos a ser estrictos, también hay distintas partes que ejecutan distintas cosas en los procesadores no tan modernos, como la unidad aritmético-lógica, pero la idea es hablar del procesador genéricamente.

Entonces, ¿cómo logramos concurrencia? Si tenemos un sólo procesador o unidad de ejecución, ¿quién se encarga de administrar qué proceso debe ejecutarse en cada momento?

En el modelo clásico de múltiples procesos (o hilos, que son una versión especial de procesos, como veremos más adelante 1.2), hay una combinación de funcionalidad entre el procesador y el sistema operativo específicamente diseñada para lograr este efecto. Los procesadores tienen implementados[1] mecanismos que el sistema operativo aprovecha, entonces, de manera que si un proceso necesita servicios del sistema operativo porque tiene que realizar entrada o salida, o pedir o liberar memoria, o cualquier llamada al sistema (o *syscall*), el procesador mismo dejará de ejecutar dicho proceso y pasará a ejecutar al sistema operativo, para que sirva esa llamada.

Esta no es la única manera en que un proceso puede ser "sacado" del procesador. También puede ser por cualquier interrupción del sistema que tenga que ser atendida por el sistema operativo (por ejemplo, porque llegó información por la red y debe ser leída), o incluso porque hace mucho tiempo que el proceso está ejecutándose. De cualquier manera, al proceso se lo saca del procesador *forzadamente*, sin que este haga nada para ser sacado, y en cualquier punto de ejecución del mismo. Este modelo se llama multitarea apropiativa (del inglés *preemptive multitasking*).

En ambos casos hay un costo muy importante a considerar, que es el de cambio de contexto (en inglés *context switch*), ya que se deben adecuar muchas estructuras dentro del procesador para que entre o salga un proceso cualquiera o el sistema operativo, lo cual puede implicar accesos a memoria o incluso a disco. El otro costo en este modelo es que los procesos o hilos ocupan memoria en mayor o menor medida. La suma de ambos hace que un sistema no escale en cantidad de procesos o hilos como quisiéramos: está bien cuando hay 10 trabajando, o 100, pero al crecer en cantidad en algún momento los cambios de contexto y la ocupación de memoria van a hacer que el sistema se vuelva ineficiente.

El otro modelo para lograr concurrencia, totalmente distinto al anterior, es el asincrónico. En este caso no tenemos múltiples procesos o hilos para lograr el efecto deseado, sino algo que se llama *corrutina*, que no es más que una función del programa, pero con la diferencia conceptual que sabemos que estas funciones/corrutinas van a estar ejecutándose de forma concurrente en

[1] No todos, claro, por ejemplo en la linea x86 de Intel esta funcionalidad recién apareció en el modelo 80386 (año 1985).

el mismo proceso (alguna en un momento, otra luego, otra o la anterior más adelante). Todo esto está manejado por otra sección del código que se llama *reactor* o *event loop*, el cual normalmente no necesitamos implementar sino que usamos algún *framework asincrónico* que nos lo provea.

Este reactor entonces va a manejar los recursos según necesiten ser atendidos. Si una corrutina está esperando información de disco, el reactor ejecutará otra mientras tanto, y cuando esa información esté disponible continuará la corrutina anterior.

Todo esto sucede dentro del mismo proceso (con lo cual no existen los cambios de contexto que mencionábamos para el caso de multitarea apropiativa). Pero entonces ¿cómo es que se puede alternar entre las distintas corrutinas y el reactor, si no hay nada que "interrumpa"? De una única manera: la corrutina tiene que "soltar" el procesador cuando tiene que esperar que suceda algo externo (el *cómo* se hace eso en el código depende exclusivamente del framework asincrónico que estemos usando). Es por eso que este modelo se llama "cooperativo", porque cada función/corrutina tiene que estar escrita de forma de cooperar con el framework asincrónico.

Tengamos en cuenta que todo lo explicado hasta ahora no es específico de Python, sino genérico a toda la informática. En las siguientes secciones profundizaremos un poco en cada concepto y sí entraremos en cómo utilizar ambos modelos desde Python.

Para facilitar el entendimiento práctico mostraremos en cada caso la versión con concurrencia del siguiente ejemplo integrador, que ejecuta la típica tarea de bajar distintos archivos de un servidor:

```
1  from urllib.request import urlopen
2
3  BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
4  TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]
5
6
7  class Adder:
8      def __init__(self):
9          self.total = 0
10
11     def add(self, value):
12         self.total += value
13
14
15 def downloader(tex_name, adder):
16     url = BASE + tex_name
17     u = urlopen(url)
18     length = len(u.read())
19     adder.add(length)
20
21
22 adder = Adder()
23 for tex_name in TEX_NAMES:
24     downloader(tex_name, adder)
25 print("Done:", adder.total)
```

En este código secuencial encontramos a la función `downloader` que baja un archivo, calcula el largo, y llama a un acumulador para ir guardando el total. Por fuera (desde la linea 22) instanciamos el acumulador, iteramos una lista de nombres de archivos `.tex`, llamamos al `downloader` por cada archivo, y finalmente mostramos el total. Este pequeño programa tiene una estructura que a priori suena excesiva, pero es para minimizar las diferencias con los códigos posteriores, donde tendremos los nuevos conceptos a aprender.

```
$ python3 concur_no.py
Done: 288998
```

1.2. Threading

Los hilos (en inglés llamados *threads*) son una especie de procesos livianos que comparten entre sí la mayor parte de las estructuras internas, particularmente la memoria a la que tienen acceso. El concepto de *hilo* a nivel de sistemas operativos es casi tan antiguo como el de *proceso*, recomendamos este artículo de Wikipedia para más información.

La gran ventaja de los hilos es que nos permite lograr concurrencia o paralelismo de forma sencilla, aunque implica tener algunos cuidados que mencionaremos más adelante. Hagamos un ejemplo sencillo para explicar mejor algunos conceptos de forma más tangible.

```python
import random
import threading
import time

waits = []


def waiter():
    wait = random.randint(0, 100)
    time.sleep(wait / 100)
    waits.append(wait)


all_threads = []
for _ in range(5):
    th = threading.Thread(target=waiter)
    th.start()
    all_threads.append(th)

for th in all_threads:
    th.join()

print(waits)
```

En este código tenemos una función `waiter` que calcula un número al azar entre 0 y 100, duerme esea cantidad de milisegundos, y agrega el valor a una lista global. Esta función es la que ejecutaremos en distintos hilos, de forma concurrente. Es clave en este ejemplo la diferencia entre hilos y procesos, ya que la función accede a una lista global, y eso lo puede hacer porque los hilos comparten el espacio de memoria entre sí (incluyendo al "hilo principal" que los genera); si hubiésemos ejecutado la función en distintos procesos, este modelo no serviría porque la función accedería a *copias* de `waits`, no logrando el efecto que buscamos.

Luego tenemos un bucle `for` en el que creamos un nuevo hilo (indicando que la función a ejecutar en ese hilo es `waiter`), arrancamos dicho hilo, y lo agregamos a una lista que necesitaremos luego.

Antes de terminar nuestro proceso necesitamos esperar a que los hilos generados terminen. Para eso llamamos a `.join` en cada hilo, lo cual bloqueará el hilo principal hasta que ese hilo se le una. Luego finalmente mostramos las esperas que se sucedieron.

```
$ python3 ej_hilos_simple.py
[77, 97, 6, 9, 59]
```

Si de ese código eliminamos el segundo bucle con los `.join` a cada hilo, la lista mostrada estará vacía porque no esperamos a los hilos que terminen[2]. Les dejamos este cambio para que experimenten ustedes.

[2] En realidad, eso tampoco es 100 % determinístico, porque como mencionábamos antes, en la multitarea apro-

En sistemas con más de un procesador (lo cual es normal en estos días), los hilos pueden ejecutarse en distintos procesadores, segun determine el sistema operativo. En lenguajes como C o C++ esto implica que los distintos hilos *podrían* ejecutarse simultáneamente, logrando efectivamente paralelismo. En Python[3] esto no es así, ya que incluso cuando los hilos se ejecutan en distintos procesadores, por como trabaja internamente el lenguaje esos hilos se ejecutarán no simultáneamente.

Expliquemos eso un segundo. Python tiene determinadas estructuras internas (en especial el conteo de referencias a los objetos, parte de la administración dinámica y automática de la memoria[4]) que no pueden modificarse en cualquier momento desde cualquier hilo: para evitar que dos hilos accedan simultaneamente a esas estructuras se bloquea su uso de una forma particular (veremos estos conceptos con detalle más adelante).

Si ese bloqueo y desbloqueo sucede cada vez que se acceden a las estructuras mencionadas, su utilización sería demasiado ineficiente, especialmente en el caso en que la aplicación no esté usando hilos. Python resuelve esto teniendo un bloqueo global a todo el intérprete (*Global Interpreter Lock*, o *GIL*), que es mucho más eficiente, pero tiene como resultado colateral que los distintos hilos no se ejecutarán simultaneamente, incluso cuando estén en distintos procesadores.

Esto no es para nada un problema en programas que no usan hilos (la mayoría), o incluso en programas con muchos hilos que se basan en esperar entradas/salidas (acceso a disco, a la red, o en general a cualquier periférico o dispositivo fuera del procesador), pero sí es una limitación en programas *cpu-bound*[5]. Por eso muchas bibliotecas de procesamiento numérico escritas en lenguajes de bajo nivel (como NumPy), liberan el GIL al ejecutarse, logrando el paralelismo real que mencionábamos antes.

1.2.1. Usando hilos

La forma más sencilla de ejecutar una función en un hilo distinto es la que mostramos arriba, donde creamos al hilo indicándole dicha función (y opcionalmente que argumentos pasarle al ejecutarla).

Hay otra forma que no es tan directa pero nos permite armar estructuras más complejas y flexibles, que es escribir una clase que hereda de `threading.Thread`.

```
1  import random
2  import threading
3  import time
4
5
6  class Waiter(threading.Thread):
7
8      waits = []
9
10     def run(self):
11         wait = random.randint(0, 100)
```

piativa no tenemos control sobre en qué momento los hilos se ejecutan o dejen de ejecutarse (a menos que usemos herramientas de sincronización como "locks" o "semáforos"), entonces puede suceder que en alguna ejecución algún hilo lanzado sí haya terminado antes que el hilo principal.

[3] Estos conceptos de bajo nivel y otros explicados luego como el GIL son inherentes a cPython (la implementación de Python en C), otras implementaciones pueden variar.

[4] Python maneja la memoria de forma dinámica basándose en el conteo de referencias: por cada objeto creado Python lleva la cuenta de todas las referencias a ese objeto. Cuando esas referencias se pierden (el contador baja a cero), Python libera la memoria usada por ese objeto.

[5] Se denominan *cpu-bound* a aquellos programas que usan (casi) exclusivamente el procesador, sin tener ninguna entrada o salida, por ejemplo haciendo cálculos matemáticos, procesamiento de imágenes, manejo de datos científicos, etc.

```
        time.sleep(wait / 100)
        self.waits.append(wait)


all_threads = []
for _ in range(5):
    th = Waiter()
    th.start()
    all_threads.append(th)

for th in all_threads:
    th.join()

print(Waiter.waits)
```

Ahora encapsulamos nuestra funcionalidad dentro de la clase, la cual debe heredar `threading.Thread` para ser un hilo. El código en si es bastante parecido a la versión más simple, porque son ejemplos sencillos, pero tenemos que darnos cuenta que pasamos de tener sólo una función a poder aprovechar todo el potencial de la programación orientada a objetos.

Cuando le hagamos `.start()` al hilo, luego de correr alguna maquinaria interna terminará ejecutando el método `.run` que es el punto de entrada a nuestro código; el hilo termina cuando este método termina. El único cuidado que tenemos que tener es que si definimos el método `__init__` en nuestra clase debemos recordar de llamar al método de inicialización de su clase padre `Thread`, caso contrario no estará listo para arrancar con el `.start`.

Volviendo a la forma más directa de usar hilos, veamos nuestro ejemplo integrador (que baja archivos y suma sus longitudes) pero adaptada a esta forma de usar hilos:

```
import threading
from urllib.request import urlopen

BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]


class Adder:
    def __init__(self):
        self.total = 0

    def add(self, value):
        self.total += value


def downloader(tex_name, adder):
    url = BASE + tex_name
    u = urlopen(url)
    length = len(u.read())
    adder.add(length)


adder = Adder()
all_threads = []
for tex_name in TEX_NAMES:
    th = threading.Thread(target=downloader, args=(tex_name, adder))
    th.start()
    all_threads.append(th)

for th in all_threads:
    th.join()
print("Done:", adder.total)
```


VERSIÓN PRELIMINAR

El código es muy similar al secuencial, sólo que ahora en vez de llamar a `downloader` directamente, la ejecutamos dentro de un hilo separado (y sí, eso implica que hay que crear el hilo, arrancarlo, guardarlo para luego esperarlo... no podemos escaparnos de este pequeño costo de infraestructura).

Si lo ejecutamos, notamos que es bastante más rápido que el secuencial, y (a priori, ver luego) nos da el mismo resultado:

```
$ time python3 code/concur_no.py
Done: 288998

real    0m0,824s
user    0m0,108s
sys     0m0,011s
$ time python3 code/concur_hilos.py
Done: 288998

real    0m0,245s
user    0m0,065s
sys     0m0,024s
```

1.2.2. Modificando estructuras en sistemas multithreading

Más allá de la forma utilizada (llamando a las funciones en un hilo, o escribiendo nuestros hilos con clases), siempre debemos tener en cuenta que no tenemos control de cuando se ejecutan cada uno de los hilos creados, y debemos modificar nuestros programas para hacerlos seguros con respecto a eso.

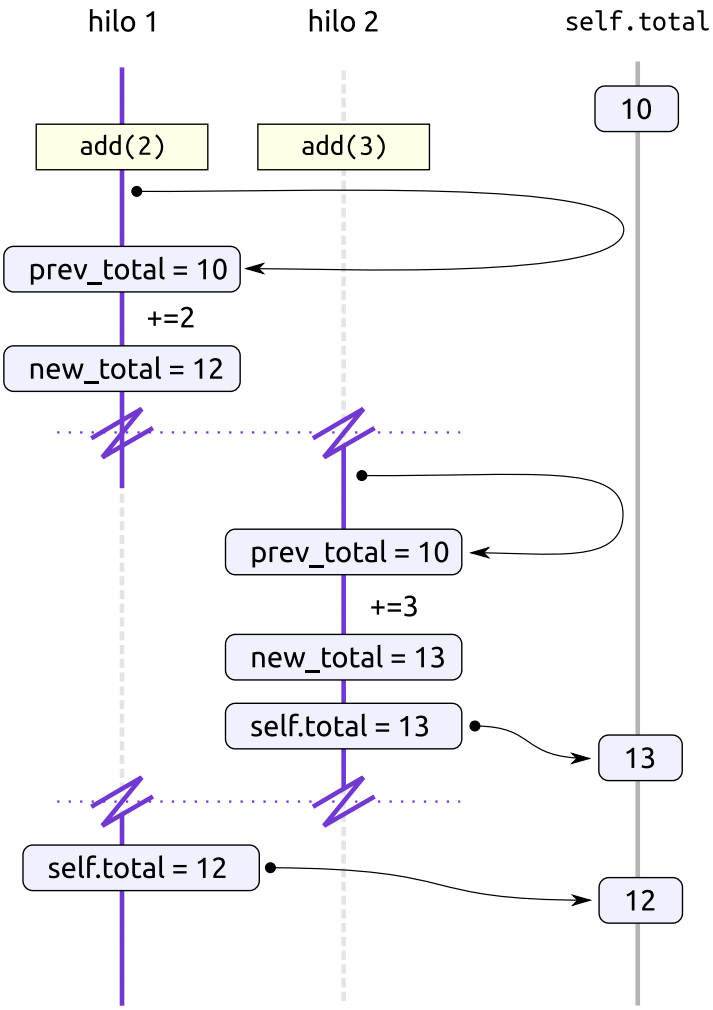
Y eso no es trivial. Aparece la gran trampa de usar hilos en nuestros programas (más allá del lenguaje utilizado): es muy difícil pensar y darse cuenta en todo el código de los potenciales problemas al no tener control de qué hilo se está ejecutando.

Analicemos en detalle, por ejemplo, el programa que mostramos antes. Tiene un punto de falla en el método `Adder.add`, donde hace `self.total += value`. Para entender mejor por qué, "abramos" esa linea en los tres pasos que realmente suceden:

```
1 def add(self, value):
2     prev_total = self.total
3     new_total = prev_total + value
4     self.total = new_total
```

Recorramos de forma detallada una secuencia que "explote" el problema. Supongamos que tenemos un valor de `10` en `self.total`, y desde dos hilos se llama a ese método con los valores `2` y `3`. Si se ejecutaran en momentos distintos, el resultado final sería el correcto, `15`.

Pero supongamos una ejecución particularmente problemática, mostrada en el esquema a continuación. En este caso vemos que empieza ejecutándose el primer hilo, guarda el total hasta ese momento y le suma lo que recibió, quedando `new_total` en `12`. En ese momento su ejecución se interrumpe y entra el otro hilo, que lee `10` de `self.total` (porque todavía no cambió), le suma `3`, y termina dejando `self.total` en `13`. Entra nuevamente el primer hilo, continúa con su proceso y guarda en `self.total` el `12` que tenía en `new_total`. En este punto estamos con los dos hilos que ya ejecutaron el `Adder.add`, y el valor final de `self.total` es incorrecto.



La probabilidad de que eso suceda en ese código es baja (pero no cero), lo cual en realidad es peor aún, ya que lleva a bugs muy difíciles de reproducir y encontrar. Incluso si ejecutamos varias veces el programa con esa modificación, veremos que nos da el valor correcto:

```
$ python3 code/concur_hilos_explicito_1.py
Done: 288998
```

Para aumentar la probabilidad de que entremos en la secuencia problemática, hagamos que el hilo "duerma" entre esas lineas (ya que la llamada a `time.sleep` implica un cambio de hilos):

```
1 def add(self, value):
2     prev_total = self.total
3     time.sleep(.01)
4     new_total = prev_total + value
5     time.sleep(.01)
6     self.total = new_total
```

Ahora sí encontramos el inconveniente:

```
$ python3 code/concur_hilos_explicito_2.py
Done: 209628
```

Incluso con estos `time.sleep` arbitrarios, que fuerzan a que el hilo suelte el procesador (lo cual incrementa la chance de que entre algún otro hilo), depende mucho de los tiempos y las velocidades de procesamiento. Por ejemplo, si en lugar de dormir 10 milisegundos como mostramos arriba, lo ponemos a dormir 1 milisegundo, casi siempre se ejecuta correctamente (porque tengamos en cuenta que los otros hilos están ocupados accediendo a la red, lo cual lleva proporcionalmente bastante tiempo).

Esta situación se denomina "condición de carrera" (*race condition* en inglés), una expresión usada en electrónica y en programación para señalar cuando la salida o estado de un proceso es dependiente de una secuencia de eventos que se ejecutan en orden arbitrario, y al trabajar sobre un mismo recurso compartido se pueden producir errores cuando dichos eventos no se ejecutan en el orden esperado.

Además, tengamos en cuenta que este código era sencillo y pequeño. Extrapolemos la dificultad si tenemos miles o decenas de miles de lineas de código, ¿cuanto podemos asegurar que cada una y todas las lineas son seguras para un entorno multi-hilo?

La forma de evitar el problema es usar un bloqueo para que distintos hilos **no puedan ejecutar al mismo tiempo** la parte del código que puede ser problemática. Eso se logra usando una herramienta que tanto en inglés como en castellano denominamos "lock".

Los *locks* se toman y se liberan. Antes de entrar a la zona problemática del código se toma el lock, y al salir se libera. Si tenemos el lock tomado y el sistema operativo deja de ejecutar nuestro hilo y entra otro, y ese otro hilo llega a la misma zona problemática, al intentar tomar el lock se bloqueará y sólo se desbloqueará cuando el primer hilo libere el lock.

Podríamos representar el uso del lock con el siguiente código:

```
mylock = threading.Lock()
mylock.acquire()
... # zona problemática del código
mylock.release()
```

Sin embargo ese código es incorrecto y peligroso: si la zona problemática del código genera una excepción nunca se liberará el lock, y eventualmente nuestro sistema se bloqueará para siempre (porque los distintos hilos estarán esperando que se libere un lock que nunca se liberará).

Podemos evitar el problema asegurándonos que **siempre** se ejecute el release, sin importar si hubo un error o no:

```
mylock = threading.Lock()
mylock.acquire()
try:
    ... # zona problemática del código
finally:
    mylock.release()
```

Los locks de Python se ofrecen en modo administrador de contexto**??** que nos asegura el mejor funcionamiento, así que el código es aún más sencillo:

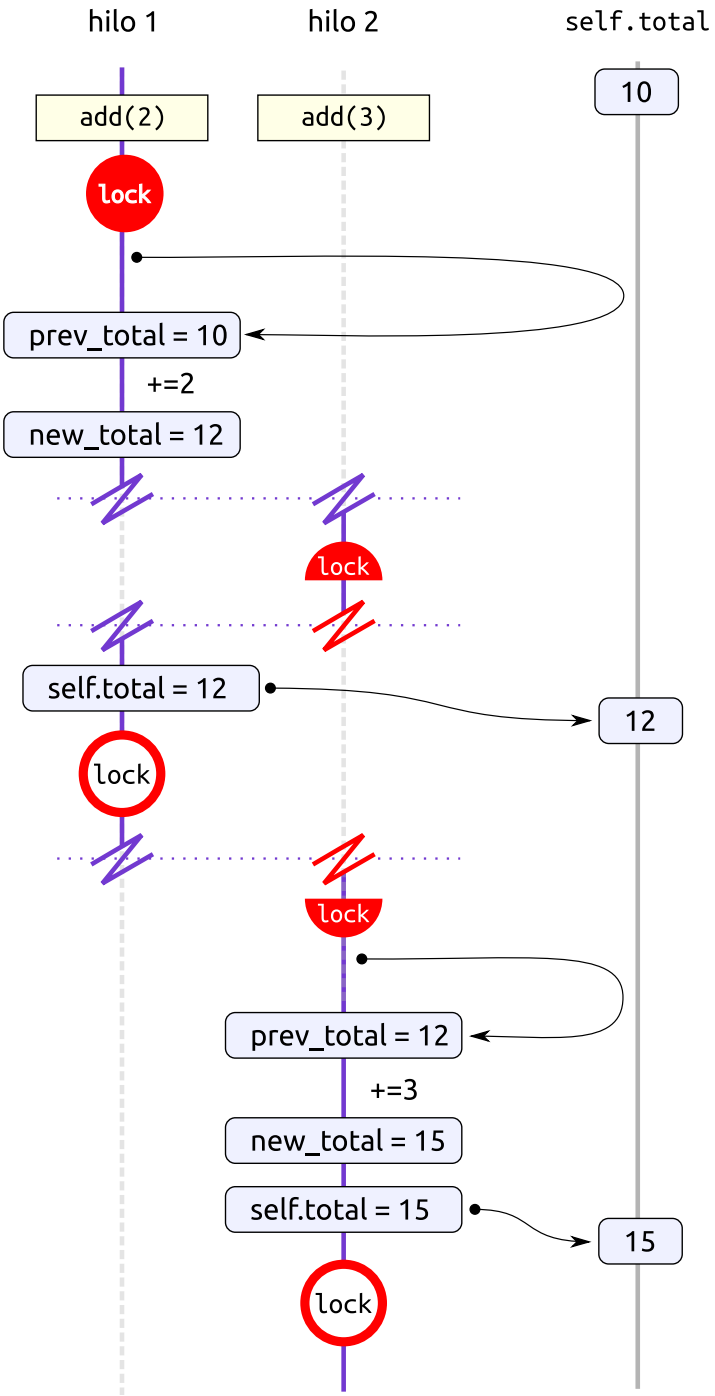
```
mylock = threading.Lock()
with mylock:
    ... # zona problemática del código
```

Veamos el uso del lock en el código problemático que traíamos:

```python
import threading
import time
from urllib.request import urlopen

BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]


class Adder:
    def __init__(self):
        self.total = 0
        self.lock = threading.Lock()

    def add(self, value):
        with self.lock:
            prev_total = self.total
            time.sleep(.01)
            new_total = prev_total + value
            time.sleep(.01)
            self.total = new_total


def downloader(tex_name, adder):
    url = BASE + tex_name
    u = urlopen(url)
    length = len(u.read())
    adder.add(length)


adder = Adder()
all_threads = []
for tex_name in TEX_NAMES:
    th = threading.Thread(target=downloader, args=(tex_name, adder))
    th.start()
    all_threads.append(th)

for th in all_threads:
    th.join()
print("Done:", adder.total)
```

```
$ python3 code/concur_hilos_explicito_3.py
Done: 288998
```

En el siguiente diagrama vemos el flujo de ejecución:



Aquí vemos que aunque el primer hilo es interrumpido igual que en la situación anterior (donde terminábamos con el valor incorrecto), cuando el segundo hilo quiere obtener el lock se traba porque el mismo ya está tomado. Allí el sistema operativo volverá a cambiar de hilos y entrará (eventualmente) el primero, que continuará con su proceso y liberará el lock al final. En este momento el segundo hilo pasa a estar disponible y (eventualmente) se ejecutará, terminando de tomar el lock, procesando la suma, y liberando el lock también al final.

¿Dónde tenemos que usar locks? Siempre que tengamos que proteger una secuencia de operaciones que si son interrumpidas y ejecutadas en parte por distintos hilos simultáneamente podrían generarnos problemas. Uno de los más grandes inconvenientes al analizar programas

propios y ajenos es reconocer estas secuencias de operaciones.

Para el ejemplo que traíamos, la zona problemática era una sola linea que hacía `self.total += value`. Pero esta linea no es una operación "atómica" (indivisible) sino que realmente es una secuencia de operaciones. La forma de detectar esto, en Python, es analizar el bytecode generado:

```
>>> import dis
>>> def add(self, value):
...   self.total += value
...
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (self)
              2 DUP_TOP
              4 LOAD_ATTR                0 (total)
              6 LOAD_FAST                1 (value)
              8 INPLACE_ADD
             10 ROT_TWO
             12 STORE_ATTR               0 (total)
             14 LOAD_CONST               0 (None)
             16 RETURN_VALUE
>>>
```

La única linea que teníamos a nivel lenguaje Python en realidad se traduce en esa secuencia de operaciones internas de la máquina virtual de cPython, que sí son atómicas (recomendamos la documentación del módulo `dis` para profundizar sobre el bytecode de Python). Ahí vemos que los pasos son varios, parecidos a los que simulamos en los códigos ejemplos que mostramos antes.

La complejidad de tener que analizar todo el código y entender qué operaciones son atómicas y cuales no, al momento de modificar estructuras de datos en situación de hilos múltiples, está presente en todos los lenguajes (por ejemplo, en C una linea se transforma en muchas operaciones en assembler cuando compilamos). Como resultado, la mayoría de los programs multi-hilos tienen situaciones problemáticas escondidas, lo que lleva a bugs esporádicos muy difíciles de erradicar.

1.3. Async

Como mencionamos antes en la introducción, el modelo asincrónico se basa en tener varias corrutinas (en lugar de procesos) que se ejecutan de forma concurrente, siendo administradas por un reactor o loop de eventos (en lugar del sistema operativo con ayuda del procesador).

Esta administración necesita sí o sí de la ayuda de las mismas corrutinas: el reactor va a decidir qué funciones o corrutinas ejecutar teniendo en cuenta qué eventos hay para las mismas, pero no podrá interrumpirlas una vez que estén ejecutándose, y dependerá que esas corrutinas liberen el control cuando necesitan ejecutar una operación bloqueante (recordemos que todas las corrutinas y el reactor corren en el mismo proceso). Es por esto que se lo llama modelo cooperativo.

Las corrutinas entonces correrán hasta que necesiten ejecutar una operación que puede tardar un tiempo significativo en ejecutarse. En ese momento le indicarán al reactor que necesitan el resultado de esa operación, en efecto liberando el uso del procesador. Este simple código muestra el modelo descripto (explicaremos en detalle cómo se usa async en 1.3.1):

```
import asyncio
import time


async def timings():
```

```
    print("A:", time.time())
    await asyncio.sleep(1)
    print("B:", time.time())

asyncio.run(timings())
```

Esta función de Python (en realidad una corrutina, porque está definida con el `async` adelante) va a mostrar dos tiempos con un segundo de diferencia entre ellos. La corrutina no debe dejar pasar el segundo con un `time.sleep(1)` clásico porque eso bloquearía a todo el proceso, sino que en vez le indica al reactor que quiere dejar pasar ese tiempo. Luego, debe interrumpir su ejecución esperando que ese evento suceda, lo cual se realiza con el **`await`** que está delante.

Por supuesto, esta corrutina tiene que ser ejecutada por el reactor, lo cual se logra de forma sencilla en la última linea del código. Hay formas más complejas y versátiles de usar o crear loops de eventos; usar `asyncio.run` es la forma más simple y es suficiente por ahora.

En la llamada a `asyncio.sleep(1)` el reactor creará un evento internamente que se ejecutará luego de un segundo, y cuando eso suceda enviará el resultado a la corrutina, la cual no se ejecutará desde el principio (como si fuese una función normal llamada nuevamente) sino que continuará desde el punto que largó el procesador (un comportamiento más parecido a los generadores que vimos en **??**).

Una simplificación casi absurda del código del reactor podría ser el siguiente:

```
while not self.stop:
    events = self.get_events()
    for event in events:
        self.dispatch(event)
```

Por supuesto que los loops de eventos reales son más complejos que eso, pero la idea es la misma: estar todo el tiempo escuchando si suceden eventos, y cuando hay nuevos despacharlos a las corrutinas correspondientes. Los eventos pueden ser de varios tipos: un timer (como vimos recien), que un recurso del sistema operativo está listo para ser leido o escrito (un archivo, un socket de red, etc.), e incluso eventos que no son comenzados por una corrutina sino que provienen de periféricos o acciones del usuario (como el click de un botón del mouse).

El tener el reactor en el mismo proceso es una de las razones por las que el modelo asincrónico escala en cantidad de operaciones de entrada/salida concurrentes mejor que el modelo basado en hilos. También es un factor que cada elemento que está esperando un resultado es simplemente una función y no una estructura mucho más compleja como un hilo o un proceso.

El modelo asincrónico tiene varias otras ventajas. Al estar todo el sistema dentro del mismo proceso es muy fácil compartir objetos y estructuras de datos, como sucede también en el caso de los hilos; pero a diferencia de estos el acceso a esas estructuras no puede ser interrumpido en cualquier momento por un agente externo, entonces no tenemos que protegerlas con locks (y no tenemos el riesgo de no darnos cuenta que esos locks eran necesarios, lo cual resulta en programas más robustos). Otro punto de vista sobre esta característica de no ser interrumpidos en cualquier momento de ejecución es que los programas son determinísticos, lo cual los hace mucho más fáciles de testear y depurar: si estamos analizando un programa podemos estar seguros que luego de la linea N se ejecutará la linea N+1, a menos que explícitamente liberemos el procesador.

No todo son rosas, claro. El modelo tiene también sus desventajas.

El principal es que al ser un modelo cooperativo, tenemos que tener la precaución de no bloquear nunca la ejecución del proceso. Siempre utilizar una llamada preparada para trabajar asincrónicamente para todo aquello que bloquee (usar la red, por ejemplo), o incluso abrir un hilo

para esa operación (veremos más adelante esto). En algún punto estamos cambiando la "preocupación por evitar condiciones de carrera" del modelo de múltiples hilos a la "preocupación por no bloquear", la diferencia sustancial es que aquí no estamos afectados por la aleatoriedad del orden de ejecución y dónde se interrumpen los hilos (en tiempo de ejecución), sino que el código se ejecuta de forma determinística y podemos detectar cuando estamos bloqueando incorrectamente al estudiar el código (al momento de escribir el código, no cuando se ejecuta).

Otro gran detalle es que al suceder todo dentro del mismo proceso, efectivamente este corre en el mismo procesador, por lo cual no tiene sentido para ser usado en procesos cpu-bound. En otras palabras, si lo que queremos es realizar en paralelo tareas intensivas de procesamiento, este modelo directamente no aplica.

Y no hay que desestimar la complejidad de que sea un concepto todavía no muy distribuido en el mundo del software (aunque no es un modelo nuevo). La mayoría de los desarrolladores no están acostumbrados a "pensar en modo asincrónico" y lleva un aprendizaje que sólo se logra a través del tiempo y la experiencia.

1.3.1. Usando async

La idea base del modelo asincrónico es hacer todo lo que se necesita hacer, pero nunca bloquear. Según esta idea, podemos separar todas las actividades que hacemos en nuestro programa en dos grandes secciones: lo que nos va a bloquear y lo que no.

Si necesitamos algo que sabemos que va a tardar por su naturaleza misma (hacer un request HTTP, traer algo de una base de datos, etc.), se lo pedimos a alguna librería asincrónica o directamente al reactor y esperamos el resultado. Esto es lo que hace la linea 7 del código ejemplo que mostramos al arrancar la sección. En vez de usar `time.sleep(1)`, que bloquea a todo el sistema asincrónico hasta que pase ese segundo, hace `asyncio.sleep(1)` y se pone a esperar el resultado con **`await`**. La corrutina se bloquea, sí, pero sólo esa corrutina: todo el sistema asincrónico sigue operando sin problemas. Y cuando haya pasado ese segundo el mismo sistema desbloqueará la corrutina, que continuará su ejecución.

Por otro lado, si necesitamos algo que sabemos que es rápido e inmediato (multiplicar dos números, formatear una cadena, instanciar una clase, etc.), lo hacemos y ya.

Claro, tenemos que poder distinguir ambos casos, y no siempre es tan fácil. Por ejemplo, si tenemos que leer un archivo de 2 kb de disco, ¿eso va a tardar o no? En realidad es siempre rápido... pero es potencialmente bloqueante: nos puede pasar que ese archivo esté en un sistema de archivos montado a través de la red y una lectura que a priori es casi instantánea tarde varios segundos, y no queremos bloquear todo el sistema durante esos segundos. Entonces tenemos que acceder a los archivos usando, por ejemplo, la biblioteca `aiofiles`.

Empecemos a ver cómo usar el sistema asincrónico para lograr todo eso.

Ya vimos que nuestro concepto fundamental son las corrutinas (más allá del reactor o loop de eventos, que usamos pero no definimos nosotros), que son nuestras "funciones asincrónicas" y nos permiten liberar el procesador, esperando eventos sin bloquear todo, haciendo que el sistema en sí funcione correctamente.

Explorémoslas un poco más en detalle.

La definición de una corrutina es parecida a la de una función, pero se le debe agregar el **`async`** adelante:

```
1 >>> async def corrut():
2 ...   print("Hola")
3 ...
4 >>> corrut()
5 <coroutine object corrut at 0x7f8716110140>
```

El hecho de "ejecutar la función" no pone la corrutina a correr, sólo la crea (de nuevo, muy parecido a lo que pasa con los generadores). Para que la corrutina "entre en el sistema asincrónico" y pase a ser ejecutada cuando el reactor de ese sistema lo disponga, tenemos tres mecanismos: pasársela a `asyncio.run` como punto de entrada de nuestro programa, esperarla con **`await`**, o crear una tarea a partir de la corrutina (a través de `asyncio.create_task`, que nos devuelve una `Task` alrededor de una corrutina con el agregado de algunos mecanismos útiles de control).

Hagamos lo primero:

```
>>> import asyncio
>>> asyncio.run(corrut())
Hola
```

Dentro de las corrutinas habrá código que se ejecuta inmediatamente y código que implica esperar por un resultado. Como mencionamos antes, tenemos que hacer que la corrutina espere ese resultado sin bloquear todo el sistema, y para ello tenemos el **`await`**. Vimos al principio de la sección un ejemplo mínimo que lo usaba, veamos ahora uno más complejo con algunos elementos que acabamos de ver:

```
import asyncio
import random
from datetime import datetime


async def sleeper(sleeper_id):
    delay = random.random() * 5  # random float between 0 and 5
    print(f"Arranca {sleeper_id} a las {datetime.now():%X.%f}")
    await asyncio.sleep(delay)
    print(f"Termina {sleeper_id} a las {datetime.now():%X.%f}")
    return delay


async def main():
    slept = await sleeper(1)
    print(f"Durmió: {slept:.3f}s")

asyncio.run(main())
```

Vemos en el código dos corrutinas. A `main` se la pasamos como punto de entrada al sistema. Esta corrutina va a llamar/esperar a otra, `sleeper` y luego mostrará cuanto durmió, para lo cual recibe un valor en la linea del **`await`**. Acá vemos la otra forma de poner a correr la corrutina, que es esperándola.

Por su lado, `sleeper` calcula un valor al azar entre 0 y 5, hace un `sleep` con ese valor, mostrando la hora antes y después, y finalmente devuelve ese valor usado. El `sleep` también es asincrónico, por eso hay que esperarlo, pero no nos interesa guardar el valor que devuelve.

```
$ python3 code/seq_async_1.py
Arranca 1 a las 16:15:51.819157
Termina 1 a las 16:15:55.991726
Durmió: 4.168s
```

Veamos de forma más explícita la separación entre crear una corrutina y ponerla a correr:

```
14 async def main():
15     coroutine1 = sleeper(1)
16     coroutine2 = sleeper(2)
17     print("Corrutinas creadas")
18     slept = await coroutine1
19     print(f"Durmió 1: {slept:.3f}s")
20     slept = await coroutine2
21     print(f"Durmió 2: {slept:.3f}s")
```

El código crea dos corrutinas (que casi lo primero que harían sería mostrar la hora de comienzo), pero antes de "esperarlas" (que las hace disponibles para el sistema asincrónico) avisa que las corrutinas fueron creadas. Luego las espera, mostrando cuanto durmió cada una:

```
$ python3 code/seq_async_2.py
Corrutinas creadas
Arranca 1 a las 17:45:12.703133
Termina 1 a las 17:45:15.223655
Durmió 1: 2.519s
Arranca 2 a las 17:45:15.223749
Termina 2 a las 17:45:15.533578
Durmió 2: 0.309s
```

Como decíamos, el "Corrutinas creadas" aparece antes que cualquier mensaje de las corrutinas porque las mismas no se ponen a correr hasta que se esperan.

Otro efecto interesante que vemos aquí (relacionado con lo que decíamos) es que aunque estamos en un sistema asincrónico y usando corrutinas, las mismas se ejecutaron de forma *secuencial*. Esto es porque en el código de `main`, aunque creamos las dos corrutinas al principio, luego esperamos una (se pone a correr esa, ¡pero no la otra!), y cuando termina, luego del `print`, esperamos la segunda (y recién ahí se pone a correr).

Veamos cómo lograr efectivamente concurrencia al utilizar la tercera forma de poner a correr una corrutina: crear una tarea con ella.

```
14 async def main():
15     task1 = asyncio.create_task(sleeper(1))
16     task2 = asyncio.create_task(sleeper(2))
17     print("Corrutinas creadas")
18     slept = await task1
19     print(f"Durmió 1: {slept:.3f}s")
20     slept = await task2
21     print(f"Durmió 2: {slept:.3f}s")
```

Este tercer código es muy parecido al anterior, sólo que ahora creamos dos tareas que esperamos luego.

```
$ python3 code/seq_async_3.py
Corrutinas creadas
Arranca 1 a las 17:46:52.575249
Arranca 2 a las 17:46:52.575287
Termina 2 a las 17:46:53.342483
Termina 1 a las 17:46:55.811887
Durmió 1: 3.234s
Durmió 2: 0.765s
```

Aquí sí notamos que ambas se ejecutan concurrentemente, porque las corrutinas están listas para ejecutarse en el momento en que las tareas fueron creadas. Envolver corrutinas en tareas

no es útil sólo por lo mencionado: también nos agrega más funcionalidad que nos permite escalar en complejidad en nuestro sistema, como por ejemplo revisar si la tarea está completada o cancelarla).

Hay dos efectos interesantes para observar. El primero es que "Corrutinas creadas" sigue apareciendo primero, porque aunque estamos creando las dos tareas en las lineas 15 y 16, realmente todavía no le pasamos el control al reactor, así que `main` sigue ejecutándose... luego viene el `print` en cuestión, y luego la linea 18 donde pasamos a esperar a `tarea1`, y es allí en ese punto donde explícitamente con el **`await`** le damos el control al reactor, que pone ambas tareas a correr.

El segundo efecto es que aunque por azar la segunda tarea tuvo un *delay* menor y terminó antes, el mensaje de "cuanto durmió" sigue apareciendo en segundo lugar. Esto es porque más allá de la ejecución concurrente de las corrutinas, nosotros en `main` las estamos esperando secuencialmente, primero a `tarea1` en la linea 18 y luego a `tarea2` en la 20.

Esto que en el ejemplo rompe un poco el efecto de concurrencia que estamos buscando en realidad no es algo que nos tiene que preocupar. En códigos reales pocas veces se esperan las tareas de esa manera si se tienen varias ejecutando de forma concurrente (lo cual además sería engorroso para el caso de tener decenas de tareas, sería ridículo tener decenas de lineas haciendo **`await`**).

Justamente para esto es que el módulo **`async`** provee una función `gather`:

```
14 async def main():
15     tasks = [asyncio.create_task(sleeper(n)) for n in range(3)]
16     print("Corrutinas creadas")
17     results = await asyncio.gather(*tasks)
18     print("Durmieron:")
19     for idx, slept in enumerate(results):
20         print(f"{idx:4d}: {slept:.3f}s")
```

Cambiamos un poco todo el código para esta versión, para hacerlo más genérico. Ahora en vez de crear dos corrutinas y tareas por separado, manualmente, lo hacemos dentro de una *list comprehension* (tres en total, que podrían ser mil, pero corremos el ejemplo con pocas para que la salida que mostramos luego no quede fea en el libro).

También generalizamos la forma de mostrar los valores luego de las esperas, aprovechando una característica de las respuestas del `gather` que asegura que el órden de los resultados corresponde al de las tareas indicadas.

```
$ python3 code/seq_async_4.py
Corrutinas creadas
Arranca 0 a las 09:57:42.549920
Arranca 1 a las 09:57:42.549979
Arranca 2 a las 09:57:42.550000
Termina 1 a las 09:57:42.670305
Termina 0 a las 09:57:43.451375
Termina 2 a las 09:57:46.989842
Durmieron:
   0: 0.900s
   1: 0.120s
   2: 4.438s
```

El `gather` también nos permite controlar qué pasa si una de las corrutinas genera una excepción: cancelar todas las corrutinas que todavía no terminaron y propagar la excepción para arriba, o considerar esas excepciones como "resultados" y acumularlos en la lista que nos entrega junto con otros resultados.

Como vemos, el mundo asincrónico nos ofrece muchas opciones, pero siempre volvemos al talón de Aquiles de que nada puede bloquear. ¿Cómo hacemos entonces cuando tenemos algo que sí o sí bloquea?

Veamos el siguiente código de ejemplo, que nos expone a una situación de bloqueo que no podemos evitar. Para ver qué sucede en ese caso, el programa tiene una corrutina que se pone a correr al comienzo que muestra un *spinner* en la terminal: si vemos girar al spinner es que el sistema asincrónico está corriendo sin problemas.

```
import asyncio
import itertools
import sys
import time


async def spinner():
    ticks = itertools.cycle("|/-\\")
    while True:
        print(f"{next(ticks)}\r", end="", flush=True)
        await asyncio.sleep(.2)


async def generic_work_async():
    await asyncio.sleep(3)
    return 5


def generic_work_blocking():
    time.sleep(3)
    return 7


async def main(option):
    print("Comienzo")
    spinner_task = asyncio.create_task(spinner())

    if option == "1":
        print("Durmiendo modo async, ¡perfecto!")
        result = await generic_work_async()
    elif option == "2":
        print("Bloqueando todo, está mal :(")
        result = generic_work_blocking()
    else:
        print("Bloqueando, pero en un hilo (no traba todo)")
        result = await asyncio.to_thread(generic_work_blocking)

    spinner_task.cancel()
    print(f"Final; {result=}")

if len(sys.argv) != 2 or sys.argv[1] not in ["1", "2", "3"]:
    print("Usar: python3 ej_async_blocking.py {1|2|3}")
    sys.exit()

asyncio.run(main(sys.argv[1]))
```

El código tiene cuatro grandes bloques. La primer corrutina es el spinner en sí, va dibujando eternamente en la terminal algo que aparenta girar. Luego tenemos dos “tareas genéricas”, una que es asincrónica y otra bloqueante (en cada caso sólo durmiendo, porque son ejemplos). Despues viene la corrutina principal que avisa del comienzo, arranca una tarea con el spinner, ejecuta alguna “tarea genérica” de alguna manera, y luego cancela al spinner y avisa del final.

Para terminar tenemos un par de lineas que valida que al ejecutar el programa hayamos pasado una opción válida y ejecuta la función principal dentro de un reactor.

Los tres casos que podemos elegir al correr el programa muestran los distintos comportamientos. En todos los casos simulamos con el `sleep` lo que sería una operación que puede tardar mucho, como un request HTTP o acceder a una base de datos. Con el `asyncio.sleep` ejemplificamos el caso de hacer esa operación utilizando una biblioteca asincrónica, y con el `time.sleep` mostramos qué pasaría si utilizamos una biblioteca sincrónica, bloqueante.

El primer caso es muy directo: como utilizamos una biblioteca asincrónica, todo funciona correctamente. En la terminal deberíamos ver el "Comienzo", el aviso del caso 1, el spinner girando durante tres segundos, y el "Final".

Si corremos al programa eligiendo el segundo caso, directamente bloquea todo. El programa también tarda tres segundos en terminar, pero nunca vemos el spinner girar porque el sistema asincrónico está trabado.

El tercer caso es el que nos ocupa para aprender a llamar algo que inevitablemente nos va a bloquear (al ejecutar el programa deberíamos ver el spinner girando correctamente). La solución es decirle al reactor que ejecute esa función en un hilo aparte, y que nos desbloquee cuando esa función termine. De esta manera la función en sí, que es bloqueante, se va a ejecutar sin problemas, y no nos va a bloquear todo el sistema asincrónico. Lo simple de esta solución es que no tenemos que manejar realmente los hilos desde nuestro programa, pero sí tenemos que tener el cuidado de que la función ejecutada en ese hilo no acceda recursos compartidos (porque si no volvemos a tener todo el problema de posibles condiciones de carrera que vimos para el caso de múltiples hilos).

El mundo asincrónico en general y `asyncio` tiene muchos otros conceptos, pero ya se empiezan a escapar del alcance de este libro. Los mencionamos brevemente para darles una idea de los diferentes temas, de manera que si lo desean puedan profundizar en los mismos:

- Streams: manejar conexiones de red (escribiendo y leyendo flujos de datos) es tan típico en los sistemas asincrónicos que hay herramientas para simplificar el proceso. En la mayoría de los sistemas asincrónicos estas herramientas son los "protocolos" y "transportes", pero el módulo `asyncio` (además de esas herramientas) tiene el concepto de *streams*, que nos permite trabajar con conexiones de red y enviar y recibir datos de forma más simple.
- Primitivas de sincronización: más allá que la ejecución de cada linea es determinística en el mundo asincrónico, si tenemos código ejecutándose concurrentemente que accede a recursos compartidos, es posible que necesitemos sincronizar esos accesos, para lo cual tenemos locks, eventos, semáforos, etc., como en el mundo de múltiples hilos, pero adaptados para sistemas asincrónicos.
- Subprocesos: para manejar asincrónicamente, sin bloquear ni tener que mandar a otro hilo, todo lo que es ejecución de subprocesos y esperar sus resultados o ir leyendo sus salidas.
- Colas: con un comportamiento similar a `queue.Queue`, el módulo `asyncio` nos provee colas que trabajan asincrónicamente, y son la herramienta ideal para armar un modelo donde algunas corrutinas pueden ser productoras de información y otras consumidoras.

Para terminar, veamos nuestro ejemplo integrador (que baja archivos y suma sus longitudes) pero adaptado a la forma asincrónica:

```
1 import asyncio
2
3 import aiohttp  # fades
```

```
4
5  BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
6  TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]
7
8
9  class Adder:
10     def __init__(self):
11         self.total = 0
12
13     def add(self, value):
14         self.total += value
15
16
17 async def downloader(tex_name, adder):
18     url = BASE + tex_name
19     async with aiohttp.ClientSession() as session:
20         async with session.get(url) as resp:
21             content = await resp.read()
22     adder.add(len(content))
23
24
25 async def main():
26     adder = Adder()
27     all_coroutines = []
28     for tex_name in TEX_NAMES:
29         coroutine = downloader(tex_name, adder)
30         all_coroutines.append(coroutine)
31     await asyncio.gather(*all_coroutines)
32     print("Done:", adder.total)
33
34
35 asyncio.run(main())
```

La estructura es muy similar a la de los mismos ejemplos hechos con multiples hilos o sin concurrencia. Antes de entrar en el código así, notemos que estamos utilizando un módulo que no es de la biblioteca standard, `aiohttp`, para hacer pedidos HTTP asincrónicos; lo tenemos comentado con `fades` para que este nos arme el entorno virtual que corresponda, como vimos en el capítulo de Entornos ??.

Luego tenemos al Adder, que es igual a los otros en su forma simple, porque como estamos seguros que nada va a interrumpir a las corrutinas cuando suman, no necesitamos *locks*.

El downloader en sí tiene una estructura similar, pero parece más complejo a simple vista. Sí, es asincrónico, pero también aparecieron dos administradores de contexto (también asincrónicos). Esto es porque interactuar con la red tiene varios pasos que son bloqueantes, y la biblioteca está diseñada para exponer eso de forma explícita.

La sesión inicializa recursos, que luego cierra automáticamente al salir del contexto. El `.get()` recibe la primer respuesta del server (y también es un administrador de contexto, porque luego hay recursos para liberar), pero esta primer respuesta tiene los headers solamente, NO trae todo el contenido de la respuesta. La acción de leer el cuerpo de la respuesta también es un paso bloqueante; en este caso el `read()` trae todo el cuerpo de la respuesta poque sabemos que es relativamente chico, pero si tuviesemos que bajar un archivo muy grande deberíamos usar la API de streaming de aiohttp.

En realidad nosotros no estamos haciendo el mejor uso de la interfaz de aiohttp; deberíamos compartir la sesión (y no crear una en cada llama a downloader) para optimizar recursos. En este código ejemplo es practicamente lo mismo (y lo dejamos así por razones pedagógicas), pero en un sistema real, si quisiéramos escalar a centenas o miles de direcciones web, lo mejor es crear la sesión una sola vez y usar siempre esa.

Finalmente agrupamos el bloque de código principal dentro de una corrutina, que es la que le indicamos al reactor para que ejecute inicialmente, pero más allá de ese detalle la estructura es muy similar a la de hilos, cambiando "threads" por "coroutines", y aprovechando el `gather` que nos simplifica esperar varias corrutinas a la vez en lugar de tener que hacer un `.join` por cada hilo.

Cabe destacar que como en el modelo de múltiples hilos (donde a veces es dificil detectar potenciales condiciones de carrera), en el modelo asincrónico también es difícil a veces detectar potenciales situaciones bloqueantes; en el código ejemplo tenemos el caso de que todo es manejado asincrónicamente, excepto la resolución del nombre del servidor... o sea, la traducción de `raw.githubusercontent.com` a la dirección IP correspondiente. Para que este paso sea automáticamente asincrónico, también deberíamos instalar la biblioteca `aiodns`.

1.4. Procesamiento en múltiples procesadores

En esta sección charlaremos acerca de la ejecución de funciones o procesamiento de datos en distintos procesadores. En otras palabras, de lograr "paralelismo real", no simular paralelismo con concurrencia como era el caso de los sistemas asincrónicos o de múltiples hilos en Python.

Tengamos en cuenta que aquí, como en las otras secciones y como explicamos en detalle al inicio de la sección sobre concurrencia 1.1, cuando decimos procesador más bien nos referimos a "unidad de ejecución", ya que en todo lo que estamos hablando es más fácil pensarlo como unidades de ejecución distintas, ya sean uno o varios procesadores, uno o varios cores.

Exploraremos entonces algunas formas de lograr paralelismo a mano, utilizando algunas funcionalidades de Python provistas en la biblioteca estándar.

1.4.1. Introducción

A nivel de ejecutar otros procesos, Python nos ofrece las herramientas básicas sobre las que se construye todo el resto.

Entrar en el detalle de cuales son y sus alcances sería engorroso y se escapa al alcance de este libro, pero como ejemplos mencionaremos dos que están disponibles en plataformas POSIX. Por un lado tenemos al `fork`, que crea un nuevo proceso duplicando el actual; el nuevo se llama "proceso hijo", mientras que el que ejecutó el `fork` es el "proceso padre" (los dos procesos se ejecutan en espacios de memoria separados que al momento del `fork` tienen el mismo contenido). Por otro lado tenemos la familia `exec*`, que reemplazan el proceso actual con uno nuevo, inmediatamente, y no se vuelve al original (las distintas funciones de la familia, `execl`, `execve`, `execvpe`, etc., difieren en cómo se pasan argumentos a ese nuevo proceso).

La gran ventaja de Python es que nos ofrece módulos para trabajar con multiprocesamiento que nos abstrae de esas herramientas de bajo nivel y nos permite ejecutar funciones en otros procesos de forma muy sencilla. Estos módulos se llaman `concurrent.futures` y `multiprocessing`. Arranquemos por el primero.

El `concurrent.futures` nos permite ejecutar en distintos procesadores una misma función sobre una cantidad de datos: automáticamente creará un pool de procesos e irá ejecutando la función sobre las distintas entradas a medida que se vayan desocupando del cálculo anterior.

Veamos el siguiente ejemplo donde tenemos una función que factoriza números. Más allá del algoritmo en sí que no vamos a explicar ahora porque no es el foco, la función muestra cuanto tarda en ejecutarse. Luego tenemos una serie de números que serán factorizados, y al final se termina mostrando el tiempo total y los resultados.

```
1 import time
```

```
2  from concurrent.futures import ProcessPoolExecutor
3  from itertools import chain
4
5
6  def factorizer(number):
7      tini = time.time()
8      orig_number = number
9      results = []
10     for factor in chain([2], range(3, number // 2 + 1, 2)):
11         while number % factor == 0:
12             results.append(factor)
13             number = number // factor
14         if factor > number:
15             break
16     tdelta = time.time() - tini
17     print(f"{orig_number:16d}: {tdelta:7.2f}s")
18     return results
19
20
21 inputs = [
22     8919372543,
23     2429192056233,
24     23286190456122,
25     51734095734,
26     25301356,
27     353061047130563,
28     537334713453,
29 ]
30 print("Ejecutando todo:")
31 tini = time.time()
32 with ProcessPoolExecutor(max_workers=4) as executor:
33     all_results = executor.map(factorizer, inputs)
34 tdelta = time.time() - tini
35 print(f"Tiempo total: {tdelta:.2f}s")
36
37 print("Resultados:")
38 for number, result in zip(inputs, all_results):
39     print(f"{number:16d}: {result}")
```

La funcionalidad interesante a explicar es el `ProcessPoolExecutor`, usado en la linea 32. Es un administrador de contexto que nos da un `executor` que a su vez tiene un método `map`. Este método es similar a la función `map` integrada en Python que aplica una función a todos los elementos de un iterable, pero con la gran diferencia en este caso que esa función va a ser ejecutada en otros procesos. ¿En cuantos procesos y en qué orden? El `ProcessPoolExecutor` arma una cantidad de procesos (por default la cantidad de procesadores que tengamos, para el ejemplo forzamos 4) y ejecuta en esos procesos la función; cuando termina una se le pasa un nuevo valor para que vuelva a ejecutarse, sin importar lo que suceda en otros procesos.

Los resultados, como con el `asyncio.gather` que vimos antes, estarán en el mismo orden que los valores de entrada, por eso podemos relacionarlos y mostrarlos con el `zip` al final.

Veamos cómo se ejecuta el ejemplo, considerando lo que aprendimos de cómo funciona:

```
$ python3 code/ej_paralelo_futures.py
Ejecutando todo:
   2429192056233:    0.20s
        25301356:    0.37s
 353061047130563:    0.42s
  23286190456122:    0.47s
    537334713453:    4.36s
     51734095734:   20.11s
```

```
      8919372543:  154.63s
Tiempo total: 154.64s
Resultados:
      8919372543: [3, 2973124181]
   2429192056233: [3, 71, 2843, 4011487]
  23286190456122: [2, 3, 11, 39989, 8822953]
     51734095734: [2, 3, 17, 507197017]
        25301356: [2, 2, 6325339]
 353061047130563: [659, 73859, 7253723]
    537334713453: [3, 1831, 97821721]
```

Los tiempos de ejecución que se muestran al principio son cuando termina cada función, así que no nos sorprende que estén los más rápidos primero. Los resultados, por otro lado, están en el orden original.

Veamos de forma explícita cómo sucedió el procesamiento. Especificamos que la cantidad máxima de trabajadores sea cuatro, entonces el sistema arrancó procesando los números `8919372543`, `2429192056233`, `23286190456122`, y `51734095734`. La primer función, en alguno de los procesos, termina a los 200 milisegundos, entonces el sistema pasa a ejecutar en ese proceso la función sobre el quinto número. También termina rapidamente, y sigue con los números de entrada. En paralelo, claro, algunos procesos siguen con los números difíciles de factorizar.

Eventualmente todos terminan, el administrador de contexto termina, y mostramos el tiempo total de cálculo, que es prácticamente igual al del número más "costoso".

También tenemos la opción de crear procesos, ejecutar funciones en esos procesos, y esperar que esos procesos terminen, todos los pasos "a mano", como mostramos antes con los hilos (y de forma muy similar a ese código). Este sería el cambio con respecto al programa que mostramos antes, solamente pasamos a usar `multiprocessing.Process` en lugar de `threading.Thread` y llamar apropiadamente a los objetos:

```
for i in range(n_cores):
    proc = multiprocessing.Process(target=downloader, args=(tex_name, adder))
    proc.start()
    processes.append(proc)

for proc in processes:
    proc.join()
```

Por otro lado, también podemos usar `concurrent.futures` para los hilos, en aquellas situaciones donde no nos interesa levantar algún hilo en particular para alguna tarea específica (como es bastante normal en el caso de los hilos), sino más bien ejecutar la misma función para una serie de entradas. Dejamos como tarea para les lectores el adaptar el código que usando hilos baja varios archivos y suma sus longitudes (mostrado en 1.2.1) pero usando `concurrent.futures.ThreadPoolExecutor` (que funciona igual al `ProcessPoolExecutor` que recien vimos, pero usando hilos).

Todo este modelo funciona de manera simple con dos condiciones: si el tiempo de procesamiento para una determinada entrada es relativamente grande, y si la entrada y salida para cada uno de los procesamientos es relativamente chica. Es lo que sucedía en el ejemplo, donde la entrada es un número y la salida una lista corta de enteros, mientras que el tiempo de procesamiento en los distintos casos fue de 200 milisegundos a los dos minutos y medio.

En cambio, si la entrada o la salida del procesamiento necesario tienen un tamaño considerable tenemos que considerar algunas alternativas o modificaciones para que el costo de enviar y recibir mucha información entre los procesos no implique un costo excesivo.

Si el corpus grande de datos a procesar puede ser manejado en paralelo, podemos particionarlo y que los distintos procesos trabajen sobre una parte de ese corpus grande. Entonces, la

VERSIÓN PRELIMINAR

información de entrada que viajará a los distintos procesos serán los límites de comienzo y fin a procesar. Pero claro, de alguna manera los distintos procesos tienen que acceder al corpus grande de información. Y también, si los resultados son grandes, dejarlos a disposición del proceso central que tiene que acceder al resultado completo.

Una alternativa típica es trabajar con archivos. Podemos bajar el corpus grande de datos a disco y que los distintos procesos lean la información de allí. Otra opción, especialmente útil cuando ya tenemos la información en memoria (por cómo la obtuvimos) y luego debemos tener el resultado en memoria (para un procesamiento posterior), es compartir memoria entre el proceso principal y los otros procesos

Veamos un ejemplo donde tenemos dos listas de valores extremadamente grandes y tenemos que realizar una operación compleja y costosa entre cada elemento de esas dos listas, guardando cada resultado en una tercera. Por razones pedagógicas, sin embargo, acotemos el tamaño de las listas a 12 valores y la operación a una simple suma, para que sea más fácil de entender el resto.

Incluso veamos como primer paso un código que está conceptualmente equivocado, como excusa para aclarar algunas nociones:

```
from concurrent.futures import ProcessPoolExecutor

# dos secuencias de datos increiblemente grandes y complejas
input_a = [1] * 12
input_b = [3] * 12

# donde guardaremos los resultados
results = [None] * 12


def operator(limits):
    lim_inf, lim_sup = limits
    print(f"Operando entre {lim_inf} y {lim_sup}")
    for i in range(lim_inf, lim_sup):
        results[i] = input_a[i] + input_b[i]
    print("Resultado parcial:", results[lim_inf:lim_sup])


all_limits = [
    (0, 4),
    (4, 8),
    (8, 12),
]
with ProcessPoolExecutor() as executor:
    list(executor.map(operator, all_limits))

print("Resultado final:", results)
```

Primero se define las tres inmensamente grandes listas, las dos de entrada de largo 12 llenas de unos y tres respectivamente, y la de salida con 12 Nones (que indican que no se calculó nada en esa posición todavía. Luego tenemos la función que recibe los límites inferior y superior dentro de los que va a operar, y efectivamente aplica una operación compleja y costosa elemento por elemento, mostrando luego la parte del resultado sobre la cual se trabajó.

Luego tenemos los límites (escritos a mano, por simplicidad), y el `ProcessPoolExecutor` que nos va a distribuir los límites en los distintos procesos para realizar todo el trabajo. Notemos el detalle de hacer un `list()` alrededor del executor.map; esto es solamente para iterar sobre las operaciones, lo cual obligará al `ProcessPoolExecutor` a efectivamente esperar que todas terminen.

Finalmente, mostramos el resultado cuando todo terminó:

```
$ python3 code/ej_paralelo_equivocado.py
Operando entre 0 y 4
Resultado parcial: [4, 4, 4, 4]
Operando entre 8 y 12
Resultado parcial: [4, 4, 4, 4]
Operando entre 4 y 8
Resultado parcial: [4, 4, 4, 4]
Resultado final: [None, None, None, None, None, None, None, None, None, None, None, None]
```

En la salida de la ejecución vemos que la función trabajó correctamente sobre las entradas para dentro de los límites recibidos en cada caso, pero al final, la lista de resultados está intacta, lo cual es incorrecto.

Ya sabemos que el `ProcessPoolExecutor` nos va a ejecutar la función `operator` en distintos procesos, pero lo que tenemos que entender aquí es cómo se generan esos distintos procesos: son una copia del proceso original al crearse, utilizando una zona de memoria separada pero con el mismo contenido que el proceso original al momento de creación.

Entonces, cuando `operator` accede a los objetos `input_a` y `input_b`, estos tienen los valores correctos, aunque sean locales al nuevo proceso, porque son parte de lo que se copió. Y al acceder a `results` (que también existe como una lista llena de Nones), también se está accediendo a la "copia local". Entonces, se escriben allí los resultados, que luego se muestran correctamente en la linea del "Resultado parcial".

Pero luego los distintos sub-procesos terminan, volvemos al proceso original, y allí el `results` nunca fue modificado, por eso encontramos el problema que vemos en la ejecución.

La solución es relativamente sencilla: tenemos que tener una zona de memoria compartida. El módulo `multiprocessing` nos da herramientas para lograr esto de manera directa. Por ejemplo, si queremos comprartir listas tenemos `ShareableList`, con la cual podemos compartir facilmente algunos tipos de datos nativos de Python (`int`, `float`, `bool`, `str`, `bytes` y `None` (con algunas restricciones de tamaños para `str` y `bytes`):

```
from concurrent.futures import ProcessPoolExecutor
from multiprocessing import shared_memory

# dos secuencias de datos increiblemente grandes y complejas
input_a = shared_memory.ShareableList([1] * 12)
input_b = shared_memory.ShareableList([3] * 12)

# donde guardaremos los resultados
results = shared_memory.ShareableList([None] * 12)


def operator(limits):
    lim_inf, lim_sup = limits
    print(f"Operando entre {lim_inf} y {lim_sup}")
    for i in range(lim_inf, lim_sup):
        results[i] = input_a[i] + input_b[i]


all_limits = [
    (0, 4),
    (4, 8),
    (8, 12),
]
with ProcessPoolExecutor() as executor:
    list(executor.map(operator, all_limits))

print("Resultado final:", results)

```

```
for shared in (input_a, input_b, results):
    shared.shm.close()
    shared.shm.unlink()
```

En este nuevo código vemos que ahora no usamos listas simples, sino que tenemos las estructuras en una zona de memoria compartida. Y no sólo para poner el resultado final, sino también las de entrada, para evitar que se copien esas estructuras (que recordemos, sin inmensamente grandes, con lo cual el costo de copiado sería algo a considerar).

Otro detalle importante es que luego de terminar con los sub-procesos que usan esa memoria compartida, tenemos que cerrar y liberar esos recursos, lo cual hacemos en las últimas tres lineas.

```
$ python3 code/ej_paralelo_compartida.py
Operando entre 0 y 4
Operando entre 8 y 12
Operando entre 4 y 8
Resultado final: ShareableList([4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4], name='psm_65c91d2a')
```

La estructura `ShareableList` es muy práctica, porque nos esconde la serialización y deserialización de los objetos, y aunque nos provee algunas utilidades más, tenemos que tener en cuenta que tiene sus limitaciones.

Para trabajar con memoria compartida de forma más genérica, Python nos ofrece la estructura `multiprocessing.shared_memory.SharedMemory`, que nos permite trabajar directamente sobre la memoria compartida. Claro, nos tendremos que encargar de la serialización y deserialización, pero ya no tendremos límites sobre qué estructuras o secuencias de datos podemos compartir. Veremos más adelante un ejemplo que usa esta estructura.

1.4.2. Trabajando exclusivamente con números

Para el caso puntual de tener que sólo compartir números, que es algo muy común en el ámbito científico a la hora de procesar datos, tenemos algo más eficiente en el uso de memoria: la estructura `Array`. Y si sabemos que nuestro código no estará teniendo acceso a las mismas zonas de memoria compartida desde distintos procesos, incluso podemos usar `RawArray`, que no fuerza a los procesos a estar sincronizados en el acceso a la memoria compartida, siendo por lo tanto también más rápido.

```
from concurrent.futures import ProcessPoolExecutor
from multiprocessing import RawArray

# dos secuencias de datos increiblemente grandes y complejas
input_a = RawArray('B', [1] * 12)
input_b = RawArray('B', [3] * 12)

# donde guardaremos los resultados
results = RawArray('B', [0] * 12)


def operator(limits):
    lim_inf, lim_sup = limits
    print(f"Operando entre {lim_inf} y {lim_sup}")
    for i in range(lim_inf, lim_sup):
        results[i] = input_a[i] + input_b[i]


all_limits = [
    (0, 4),
```

```
    (4, 8),
    (8, 12),
]
with ProcessPoolExecutor() as executor:
    list(executor.map(operator, all_limits))

print("Resultado final:", list(results))
```

Si comparamos este último programa con el anterior veremos que son prácticamente iguales. Una diferencia importante es que a `RawArray` le tenemos que pasar el tipo de datos que va a soportar (porque arma la memoria compartida para una cantidad fija de bytes que alcanza para almacenar exactamente eso), podemos ver la lista de los distintos tipos de datos en la documentación de `array` (que es la misma que se usa aquí).

La otra diferencia es que ahora no tenemos que liberar los recursos al final, es automático. Esto es porque `RawArray` sólo sirve para compartir memoria entre el proceso principal y sus sub-procesos (entonces la limpieza es automática), en contraste con la memoria compartida de `ShareableList` o `SharedMemory`, a la que podemos acceder desde cualquier otro proceso (sabiendo el nombre usado al momento de creación).

```
$ python3 code/ej_paralelo_array.py
Operando entre 0 y 4
Operando entre 8 y 12
Operando entre 4 y 8
Resultado final: [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
```

Por otro lado, el elefante en la habitación que estamos esquivando desde que arrancamos el capítulo es que para el procesamiento de datos en general usamos NumPy y otras bibliotecas que nos permiten operar de forma extremadamente eficiente. ¿Como compaginamos esto con el multiprocesamiento?

Es muy sencillo, ya que por diseño NumPy nos expone la zona de memoria sobre la que trabaja, y eso nos permite integrarnos con las estructuras de memoria compartida de forma bastante directa.

```
import numpy as np
from concurrent.futures import ProcessPoolExecutor
from multiprocessing import shared_memory

# el tamaño de nuestras secuencias de datos increiblemente grandes y complejas
size = 12

# creamos toda la memoria compartida que vamos a usar (para los tres arreglos)
array_size = np.int8().size * size
shared_memory = shared_memory.SharedMemory(create=True, size=array_size * 3)

# los dos arreglos de entrada (con sus valores), y el de los resultados
input_a = np.ndarray((size,), dtype=np.int8, buffer=shared_memory.buf)
input_a.fill(1)
input_b = np.ndarray((size,), dtype=np.int8, buffer=shared_memory.buf, offset=array_size)
input_b.fill(3)
results = np.ndarray((size,), dtype=np.int8, buffer=shared_memory.buf, offset=array_size * 2)


def operator(limits):
    l_inf, l_sup = limits
    print(f"Operando entre {l_inf} y {l_sup}")
    results[l_inf:l_sup] = input_a[l_inf:l_sup] + input_b[l_inf:l_sup]

```


VERSIÓN PRELIMINAR

```
25
26 all_limits = [
27     (0, 4),
28     (4, 8),
29     (8, 12),
30 ]
31 with ProcessPoolExecutor() as executor:
32     list(executor.map(operator, all_limits))
33
34 print("Resultado final:", results)
35
36 # cerramos y liberamos los recursos de memoria compartida
37 shared_memory.close()
38 shared_memory.unlink()
```

En este caso creamos sólo una zona de memoria compartida en la que ubicaremos los tres arreglos que manejamos: los dos de entrada y el de los resultados. Creamos las estructuras de NumPy usando `ndarray` que nos permite indicarle el buffer de memoria y la posición dentro del mismo.

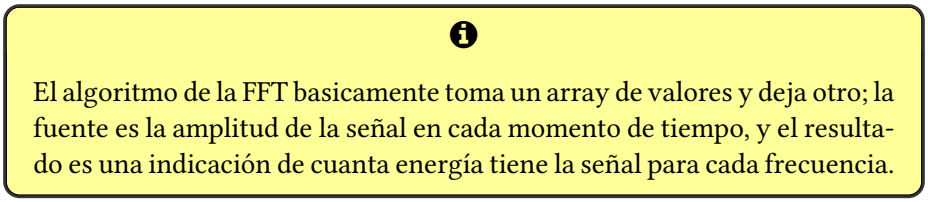
Luego `operator` se vuelve muy simple, ya que al estar trabajando con NumPy podemos directamente sumar los arreglos, lo cual sumará uno a uno sus elementos. Por supuesto su ejecución en cada uno de los procesos sumará sólo parte del arreglo general, que es justamente el efecto que estamos buscando.

Finalmente, mostramos el resultado y liberamos los recursos de memoria compartida, lo cual siempre tenemos que hacer luego de *aprovechar* los resultados (en el caso de este ejemplo mostrándolos por pantalla), ya que una vez liberada la memoria compartida no tendremos más acceso a su contenido.

1.4.3. Ejemplo práctico

Cerremos la parte de procesamiento paralelo con un ejemplo práctico.

El desafío es hacer una transformada (rápida) de Fourier (*Fast Fourier Transform*, o FFT)sobre alguna señal obtenida de alguna máquina o sensor.



El algoritmo de la FFT basicamente toma un array de valores y deja otro; la fuente es la amplitud de la señal en cada momento de tiempo, y el resultado es una indicación de cuanta energía tiene la señal para cada frecuencia.

Para nuestro experimento o ejemplo práctico usaremos una fuente que ya tenemos en disco, en formato WAV, de manera que les sea fácil poder ejecutar los distintos scripts que usaremos a continuación sobre la entrada que deseen. Nosotros usamos los Conciertos de Brandeburgo de Johann Sebastian Bach, completos. Es más de una hora y media de audio, y el archivo pesa apenas más de un gigabyte.

El siguiente es el código secuencial (no paralelo) que logra el objetivo planteado:

```
1 import sys
2
3 import scipy.io.wavfile as wavfile
4 from matplotlib import pyplot as plt, ticker
5 from scipy.fft import fft, fftfreq
6
```


VERSIÓN PRELIMINAR

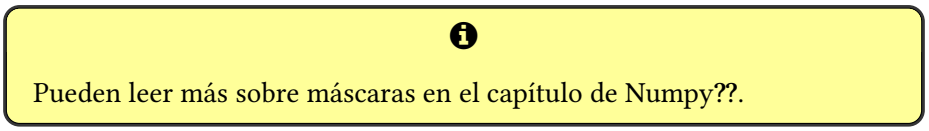
```
FREQ_MIN, FREQ_MAX = 20, 20000
TICKS = [27.5, 55, 110, 220, 440, 880, 1760, 3520, 7040, 14080]

src = sys.argv[1]
print("Cargando", src)
fs_rate, signal = wavfile.read(src)
print("Frecuencia de muestreo:", fs_rate)

q_channels = len(signal.shape)
print("Canales:", q_channels)

if q_channels == 2:
    signal = signal.sum(axis=1) / 2
assert signal.ndim == 1
Ts = 1.0 / fs_rate
print("Duración [s]:", len(signal) * Ts)

FFT = abs(fft(signal))
half_len = len(FFT) // 2
FFT_side = FFT[1:half_len]  # positive-frequency terms, excluding zero
freqs = fftfreq(signal.size, Ts)[1:half_len]
print("FFT terminado")

audible = (FREQ_MIN < freqs) & (freqs < FREQ_MAX)
FFT_side = FFT_side[audible]
freqs = freqs[audible]
assert len(FFT_side) == len(freqs)
print("Audible seleccionado, cantidad de puntos:", len(FFT_side))

(fig,) = plt.plot(freqs, FFT_side, linewidth=0.3)
plt.xscale("log")
plt.xticks(TICKS)
plt.xlim([FREQ_MIN * 0.9, FREQ_MAX * 1.1])
plt.xlabel('Frecuencia (Hz)')
fig.axes.xaxis.set_major_formatter(ticker.ScalarFormatter())
plt.savefig("fft-simple.pdf")
print("Listo")
```

Luego de los *imports* de rigor, lo primero que tenemos son unas constantes: como estaremos trabajando con audio, nos pareció interesante graficar la parte que corresponde al rango audible (entre 20 Hz y 20 kHz), y por la misma razón especificamos en qué frecuencias (las correspondiente a la nota LA en las distintas octavas) queremos los *ticks* del eje X.

Luego cargamos el archivo indicado en la linea de comandos, verificamos que sea monofónica (y si es stereo juntamos ambos canales, recuerden que estamos queriendo simular una señal simple), y calculamos la duración en segundos multiplicando la cantidad de valores por el período de la señal (la inversa de la frecuencia).

Luego sí pasamos a calcular la FFT propiamente dicha, tomamos sólo un "costado" del resultado obtenido (porque la función `fft` nos deja primero el cero, luego los valores positivos y finalmente los negativos, entonces para quedarnos con la parte positiva menos el cero tomamos desde el segundo valor hasta la mitad del array), y para terminar calculamos con `fftfreq` la frecuencia que corresponde a cada valor.

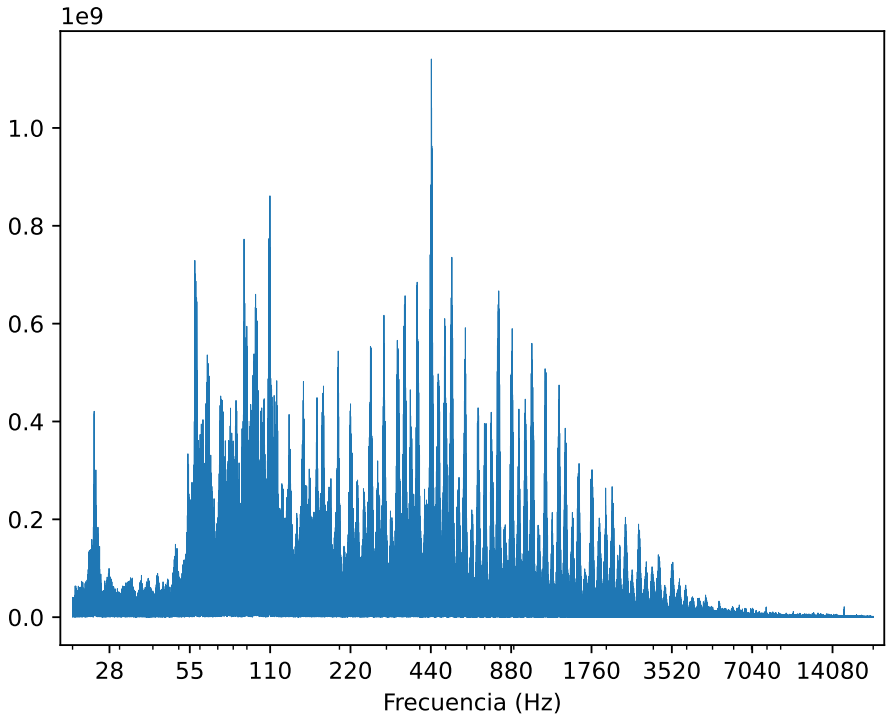
En este punto tenemos todo lo necesario para el gráfico: los valores para el eje Y (en `FFT_side`) y sus correspondientes posiciones en el eje X (en `freqs`). Pero habíamos dicho que queríamos sólo los valores correspondientes al rango audible, entonces calculamos una máscara (`audible` es un vector del largo de la señal con valores `True` y `False` según la frecuencia sea mayor que la mínima y menor que la máxima) y luego aplicamos esa máscara a los dos vectores en los que tenemos toda la información. Y como últimos pasos graficamos y guardamos la imagen resultante.



Pueden leer más sobre máscaras en el capítulo de Numpy??.

Como también vemos en el código, la ejecución nos va dando información de la señal y los pasos del procesamiento:

```
$ python code/wav-fft-simple.py bach.wav
Cargando bach.wav
Frecuencia de muestreo: 44100
Canales: 2
Duración [s]: 6058.8
FFT terminado
Audible seleccionado, cantidad de puntos: 121054823
Listo
```



En las lineas de arriba no se nota, pero la ejecución tarda demasiado. Nos gustaría que fuese más rápido, necesitamos optimizarlo.

Una regla de oro a seguir cuando necesitamos optimizar código es que necesitamos medir para poder saber qué secciones del mismo necesitamos optimizar.

Como ya tenemos el código que va imprimiendo por pantalla los distintos pasos, podemos obtener una medición rápida de los mismos usando la utilidad `ts` que nos agrega fecha y hora a cada linea de la salida de un programa.

```
$ python -u code/wav-fft-simple.py bach.wav | ts
2022-05-29 13:49:00.702611 Cargando bach.wav
2022-05-29 13:49:02.037465 Frecuencia de muestreo: 44100
2022-05-29 13:49:02.037580 Canales: 2
2022-05-29 13:49:09.187465 Duración [s]: 6058.8
2022-05-29 13:49:28.619166 FFT terminado
```

```
2022-05-29 13:49:29.242778 Audible seleccionado, cantidad de puntos: 121054823
2022-05-29 13:49:47.284587 Listo
```

Expliquemos brevemente cómo cambió la linea de ejecución. Ejecutamos nuestro programa con Python pero luego tenemos al *pipe* (`|`) que conecta la salida del programa de la izquierda con la entrada del de la derecha. En nuestro caso los prints de nuestro programa no van a pantalla sino que son leidos por `ts`, que les agrega la fecha y hora a la izquierda y ahí sí los muestra. Para lograr que las lineas de nuestro programa lleguen inmediatamente a `ts` (y no se queden retenidas en el buffer de entrada/salida que tienen los procesos para mejorar performance) ejecutamos a Python indicándole que no use ningún buffer (modo `unbuffered`, de allí la opción `-u`).

En la salida ahora notamos que el FFT en sí tarda bastante, pero también construir el gráfico al final. Esto es porque es demasiada la cantidad de puntos con los que estamos armando el gráfico: más de 121 millones, no hace falta tanto.

Entonces, la primer optimización es sencilla. En vez de armar el gráfico con todos ellos, usamos uno de cada cien (armando un gráfico con más de un millón de puntos, lo cual es más que suficiente), modificando sólo la linea donde pedimos el gráfico:

```
1     (fig,) = plt.plot(freqs[::100], FFT_side[::100], linewidth=0.3)
```

No hace falta mostrar todo el resto del código acá ya que no tiene cambios, pero queda almacenado en el archivo `code/wav-fft-optimiz.py`.

```
$ python -u code/wav-fft-optimiz.py bach.wav | ts
2022-05-29 14:02:15.010729 Cargando bach.wav
2022-05-29 14:02:15.545837 Frecuencia de muestreo: 44100
2022-05-29 14:02:15.545931 Canales: 2
2022-05-29 14:02:22.795551 Duración [s]: 6058.8
2022-05-29 14:02:40.845734 FFT terminado
2022-05-29 14:02:41.438602 Audible seleccionado, cantidad de puntos: 121054823
2022-05-29 14:02:41.905461 Listo
```

Ahora vemos como el tiempo que tarda en construirse el gráfico no influye en el total.

Es momento de paralelizar la transformada rápida de Fourier en sí, y acá nos encontramos con un gran problema. Como explicábamos arriba, la transformada rápida de Fourier nos pasa del dominio del tiempo a la frecuencia, y en su cálculo discreto debemos tener en cuenta toda la señal para calcular cada valor del vector resultante.

Es por esto que no podemos cortar la señal original "en partes" y procesar cada una de esas partes en procesadores separados. En otras palabras, si tenemos una señal de 100 segundos de largo, no podemos analizar por separado 10 bloques de 10 segundos cada uno, ya que para calcular cada valor del vector de frecuencias resultantes necesitamos siempre analizar los 100 segundos completos.

En realidad, matemáticamente, sí hay forma de paralelizar el cálculo, pero implica que implementemos FFT nosotros mismos para lograr esa paralelización, lo que ya es otra escala de complejidad y se escapa de este libro.

Si no tenemos en cuenta lo antedicho, podemos inocentemente tratar de cortar la señal en partes y luego sumar los resultados de cada análisis; por completitud implementamos esto en `code/wav-fft-paralelo-mal.py` donde analiza la señal en dos partes (el código permite extenderlo

sencillamente a muchas) y termina con un resultado que ya se nota incorrecto con una simple inspección visual del gráfico generado. Dejamos a les lectores el ejercicio de ejecutar el código y compararlo con el resultado correcto.

Algo podemos hacer, sin embargo. Teniendo en cuenta que para cada valor del resultado necesitamos sí o sí recorrer toda la señal, lo que podemos hacer es re-muestrear la señal cada N elementos en cada procesador, y luego sí sumar los vectores resultados. Supongamos que paralelizamos el procesamiento en 3 partes, entonces mandaremos los valores de las posiciones 0, 3, 6… al primer procesador, los de las posiciones 1, 4, 7… al segundo, y los de las posiciones 2, 5, 8… al tercero.

Ahora, sabiendo que tenemos que procesar la señal en distintos procesos, es momento de ocuparnos de cómo va a estar disponible la señal para los distintos procesos y cómo cada uno va a dejar su resultado para el proceso principal. De forma muy similar a lo que explicamos antes, usaremos memoria compartida tanto para la señal de entrada como para el resultado.

Implementamos todo esto en el `code/wav-fft-paralelo-ok.py`, del cual comentaremos aquí sólo las partes que cambiamos, para ayudar a entenderlas.

```
28 signal_bytes_length = signal.itemsize * signal.size
29 signal_mem = shared_memory.SharedMemory(create=True, size=signal_bytes_length)
30 shared_signal = np.ndarray(signal.shape, dtype=signal.dtype, buffer=signal_mem.buf)
31 shared_signal[:] = signal
32 del signal
33
34 result_size = shared_signal.size // MULTPARTS // 2 - 1
35 result_bytes_length = shared_signal.itemsize * result_size
36 result_mem = shared_memory.SharedMemory(create=True, size=result_bytes_length)
37 shared_result = np.ndarray((result_size,), dtype=shared_signal.dtype, buffer=result_mem.buf)
```

El primer cambio es pasar a usar memoria compartida así cada proceso no tiene que tener una copia de las estructuras de datos de señal y resultado.

En ambos casos la idea es la misma: instanciamos `SharedMemory` con la cantidad de bytes necesarios, y luego construimos un array de NumPy con la forma y tipos de nuestra señal pero indicándole que la zona de memoria a utiliza sea la compartida.

Para el caso de la señal de entrada, luego de obtener la memoria compartida tenemos que copiar allí la señal misma, objeto que luego borramos justamente para liberar esa memoria y que no se copie a cada nuevo proceso. Para el caso del resultado es suficiente con crear el array arrancando desde la memoria compartida, ya que así tendrá todos sus valores en cero.

```
40 def calculate_partial_fft(offset):
41     global shared_result
42
43     FFT = abs(fft(shared_signal[offset::MULTPARTS]))
44     half_len = len(FFT) // 2
45     useful = FFT[1:half_len]  # positive-frequency terms, excluding zero
46     with result_lock:
47         shared_result += useful
48
49
50 result_lock = Lock()
51 with ProcessPoolExecutor() as executor:
52     executor.map(calculate_partial_fft, list(range(MULTPARTS)))
```

Aquí tenemos al procesamiento en paralelo en sí. Usamos `ProcessPoolExecutor` para ejecutar nuestra función en distintos procesadores, trabajando en cada caso con un offset distinto.

Como guardamos el resultado final en `shared_result` sumando los valores de cada resultado parcial a esa estructura, tenemos que acceder a la variable global indicándolo a Python que justamente es global (podríamos tener una estructura más compleja para no usar una variable global, usando una clase que tenga el objeto a acceder y nuestra función como método interno, pero sacrificamos un poco de pureza en función de hacer el ejemplo más legible y conciso). También tenemos que usar un lock para no caer en el problema de que funciones en distintos procesos modifiquen esa estructura al mismo tiempo.

Debemos prestar atención al *slice* que se le hace a la señal al calcular la transformada; supongamos que al ejecutarse la función recibió el offset 3, eso querrá decir que esa función en ese procesador usará los valores 3, 11, 19, etc, porque arranca en 3 (el offset) y va tomando valores de a 8 (por MULTPARTS). El resto de la función toma la mitad útil del resultado (como habíamos explicado para el caso simple), para devolverlo al proceso principal.

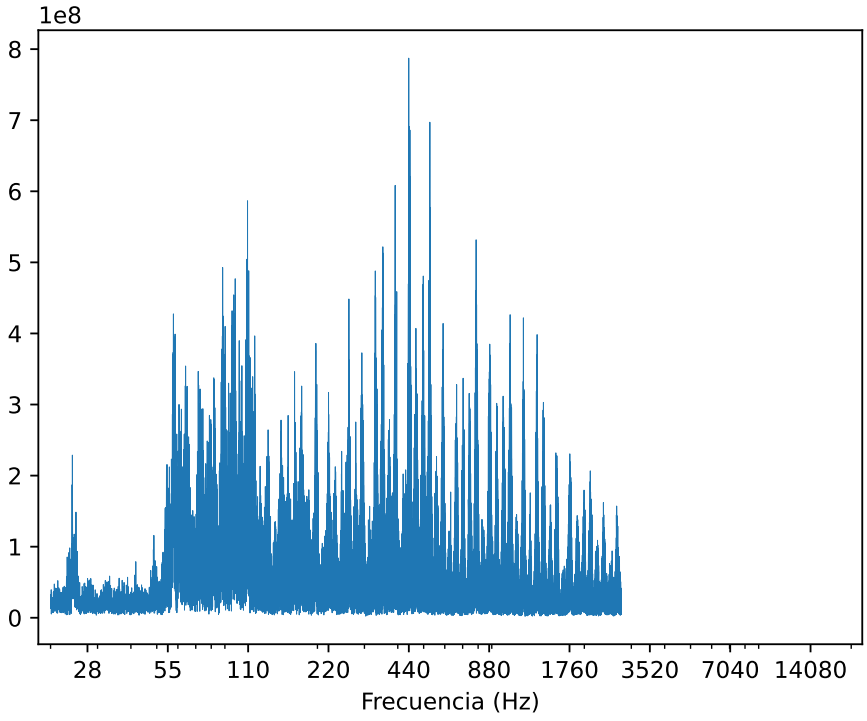
```
71 for mem in (signal_mem, result_mem):
72     mem.close()
73     mem.unlink()
```

Y luego al final del script limpiamos los recursos de la memoria compartida.

Podemos ver que la ejecución es más rápida que en el caso secuencial:

```
$ python -u code/wav-fft-paralelo-ok.py bach.wav | ts
2022-05-29 14:21:25.566173 Cargando bach.wav
2022-05-29 14:21:25.920670 Frecuencia de muestreo: 44100
2022-05-29 14:21:25.920731 Canales: 2
2022-05-29 14:21:31.560241 Duración [s]: 6058.8
2022-05-29 14:21:46.718089 FFT terminado
2022-05-29 14:21:46.857300 Audible seleccionado, cantidad de puntos: 16578390
2022-05-29 14:21:47.354224 Listo
```

Y el gráfico resultante es correcto... hasta cierto punto:



Vemos que entre las frecuencias 1760 y 3520 la curva se trunca. Esto se explica por el teorema de Nyquist-Shannon, que nos indica que sólo vamos a tener información sobre una señal hasta la mitad de la frecuencia de muestreo. En el diagrama original teníamos datos hasta 20 kHz sin problemas porque la frecuencia de muestreo del WAV es 44.1 kHz, pero en nuestro código paralelizado hicimos que cada uno de los 8 procesadores recorra la señal cada 8 valores, reduciendo efectivamente la frecuencia de muestreo a 44100 / 8 = 5512.5 Hz, por eso luego tenemos datos hasta la mitad de ese valor.

El corolario de este experimento es que tenemos que entender que paralelizar el cálculo científico no sólo tiene el inconveniente de cómo implementamos esa paralelización en uno u otro lenguaje, sino que muchas veces (la mayoría) implica tener un conocimiento específico del dominio con el que estamos trabajando.

Los distintos lenguajes y sus bibliotecas estarán más o menos preparados para lograr un paralelismo del cálculo o la tarea, y nos costará más o menos trabajo lograr el resultado final, pero nunca tenemos que perder de vista que el elemento clave en la paralelización que queremos obtener es el entender las limitaciones intrínsecas de lo que queremos paralelizar (como el caso de la transformada rápida de Fourier que acabamos de ver, o por ejemplo si partes del cálculo dependen de otras partes previas) y las limitaciones al implementarlas (cuanto se puede enviar a otros procesadores o no, cuantos datos se deben compartir en memoria, etc).

Y también es tarea del desarrollador o la científica entender y elegir cuales batallas pelear. Los códigos de arriba son genéricos con respecto al tipo de WAV que aceptan, y si reciben uno estéreo mezcla los canales para tener una sola señal con la que trabajar. Pero vemos (por ejemplo en la última ejecución) que esa operación de mezclado de los canales (el `signal = signal.sum(axis=1) / 2`) tarda casi 6 segundos, llevándose el 25 % del tiempo total de ejecución, ¡lo cual es muchísimo! ¿Vale la pena tratar de optimizar esto? Seguramente no podremos mejorar la suma que hace NumPy internamente para mezclar los canales. ¿E implementar nuevamente la función de lectura del WAV para dejar directamente una señal mezclada? ¿O limitar el script genérico para WAVs monoaurales? Este tipo de preguntas se tendrán que contestar analizando cada caso y cada necesidad en particular.

Parte II
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [2].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.