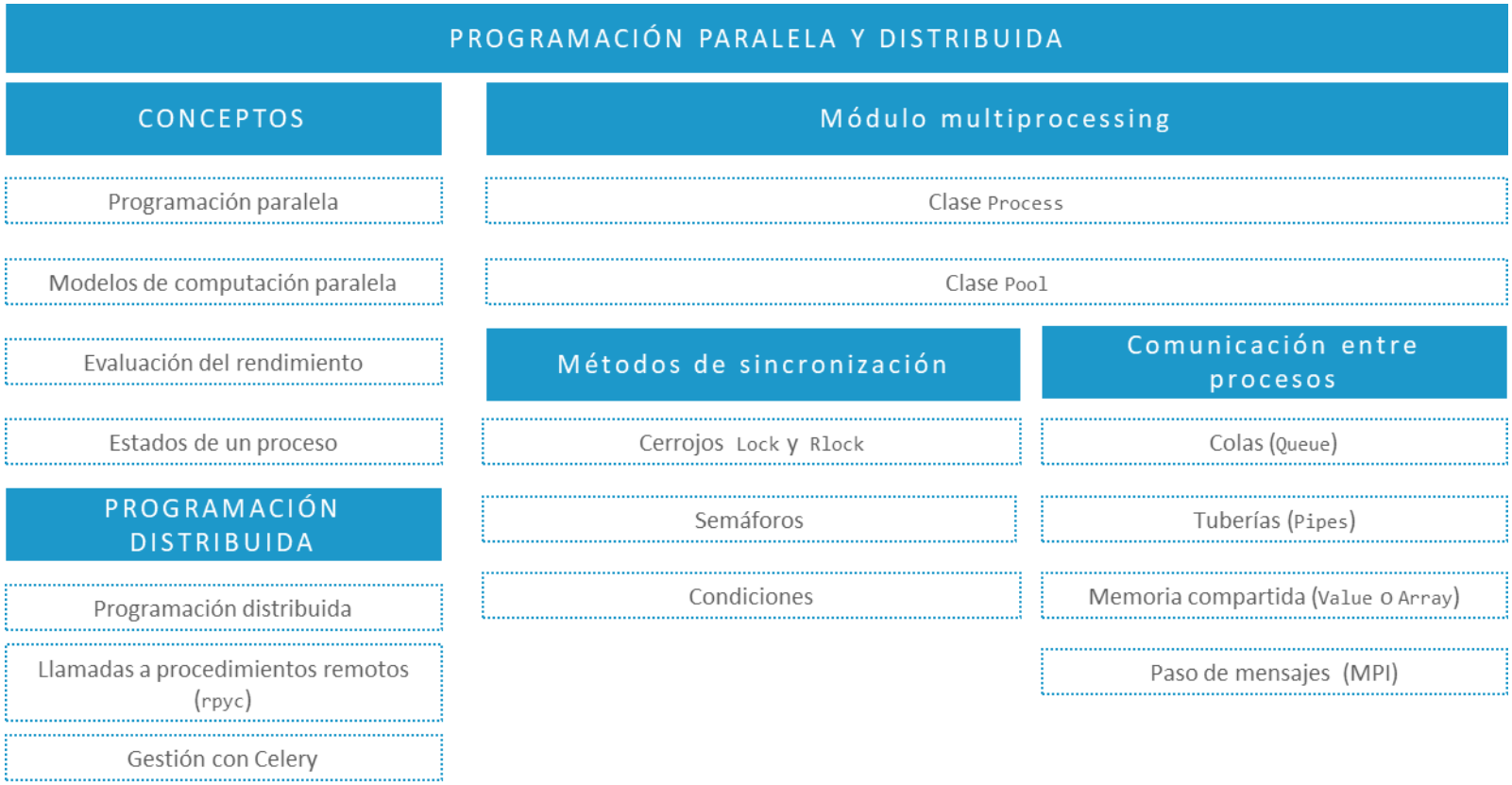
Programación Científica y HPC

Programación paralela y distribuida

Índice

Esquema



Ideas clave

8.1. Introducción y objetivos

En este tema se tratará la computación paralela, sus modos de implementación y las ventajas que aporta a la computación.

La computación paralela no ha parado de evolucionar desde los años cincuenta, con el objetivo de mejorar la eficiencia de los programas, pero, sobre todo, debido a las **necesidades de la computación de altas prestaciones en áreas como la ciencia y la industria.** Las aplicaciones que se implementan en estas áreas demandan prestaciones y velocidad computacional para poder **tratar problemas complejos en un tiempo razonable.**

En los últimos años, **la tendencia para mejorar las capacidades de procesamiento y la velocidad van dirigidas al aumento del número de núcleos integrados** en cada máquina. Conviene indicar que hasta el procesador de un teléfono móvil puede tener dieciséis o más núcleos, y existen ordenadores no demasiado costosos que cuentan con una unidad de procesamiento gráfico o **GPU** (*Graphics Processing Unit*), que **actúa a modo de coprocesador para liberar la carga sobre el procesador central**, además se dedica al procesamiento gráfico y a realizar operaciones en coma flotante, mejorando considerablemente el rendimiento de los ordenadores.

De hecho, la tendencia va dirigida a dejar de producir procesadores monolíticos, por muy rápidos que sean, y desarrollar procesadores que cuenten con varios núcleos. De forma que, actualmente, **es equiparable el concepto de** unidad central de procesamiento o **CPU** (*Central Processing Unit*), **con el de núcleo.**

De esta forma, los esfuerzos de programación van orientados a explotar las múltiples CPU ejecutando muchos procesos en paralelo, es decir, simultáneamente mediante

la ejecución de aplicaciones paralelas, en lugar de las tradicionales del paradigma secuencial y haciendo uso de nuevos modelos de programación.

Por tanto, **la unidad básica de programación serán los procesos que son entidades de ejecución independiente**, que puede sincronizarse o comunicarse con otros procesos y que cuenta con registro de activación propio para la ejecución.

Este incremento de capacidades de procesamiento **adquiere especial relevancia en la programación científica,** que necesita manejar grandes cantidades de datos y realizar simulaciones y otros cálculos intensivos en la CPU.

La programación paralela consiste, básicamente, **en el uso de varios procesadores que trabajarán de forma conjunta en la resolución de un problema computacional.** Lo habitual es que en cada procesador se ejecute una parte del problema, pero se puede permitir el intercambio de datos entre los procesadores.

Es precisamente la forma de intercambio de datos la que determina dos modelos de programación paralela, que se muestran en la Figura 1, y que se conocen como:

- **Computación paralela o multiprocesamiento,** que es un método de procesamiento que permite que una tarea sea dividida en partes, cada una de las cuales se ejecutará sobre un procesador distinto de forma simultánea. Posteriormente, todos los resultados obtenidos se combinan porque, de hecho, la memoria es compartida. Esto mejora la eficiencia en tiempo de cálculo de tareas complejas. Se puede decir, por tanto, que es una **forma de computación colaborativa mediante el uso de varios procesadores.**

- **La programación o procesamiento distribuido** es un método de procesamiento en el que las partes de las tareas se ejecutarán en procesadores distintos, pero en este caso, la memoria no es compartida.

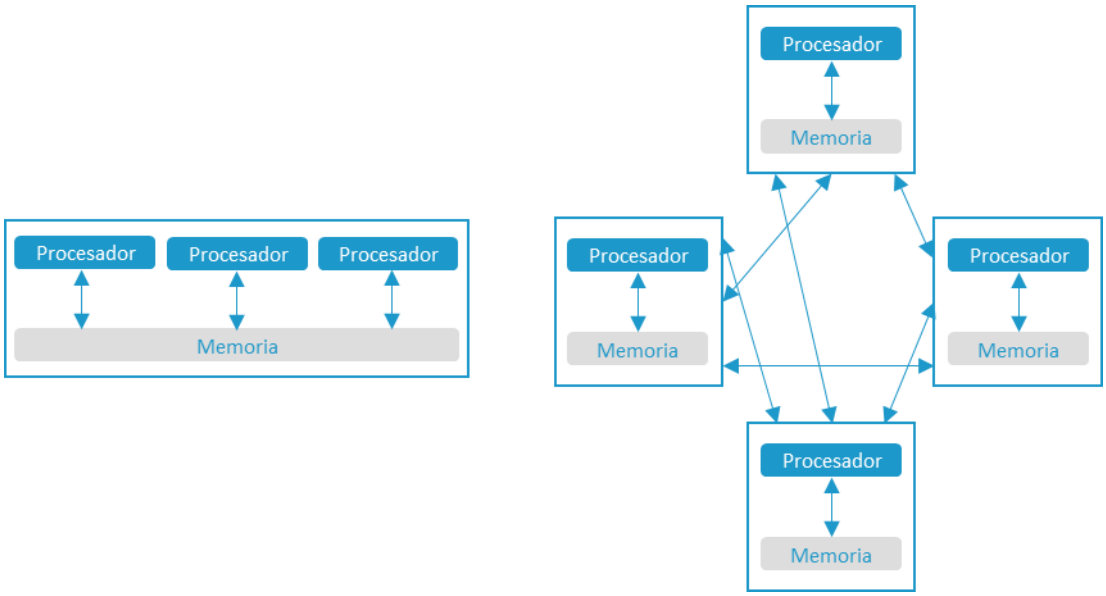


Figura 1. Sistema paralelo y sistema distribuido. Fuente: elaboración propia a partir de https://es.wikipedia.org/wiki/Computación_distribuida

Actualmente, se puede decir que la computación de altas prestaciones o **HPC** (*High Performance Computing*), **es equivalente a computación paralela.** Su objetivo principal es la **reducción del tiempo de computación de la tarea que se quiere ejecutar,** así como la resolución de problemas con modelos matemáticos complejos con una mayor precisión.

Arquitectura del sistema

Los sistemas que se utilizan en la programación paralela o distribuida son sistemas conocidos como sistemas **MIMD** (*Multiple Instruction, Multiple Data*), que están **compuestos por un conjunto de núcleos independientes con su propia unida de control y unidad aritmeticológica.** No existe dependencia temporal entre los distintos núcleos, cada uno tiene su propio reloj, su ejecución es totalmente independiente, a no ser que existan necesidades de sincronización.

En este modelo de sistema se usan dos enfoques para la paralelización:

ENFOQUES PARA LA PARALELIZACIÓN

ENFOQUE *FORK-JOIN*	ENFOQUE SMPD *(SINGLE PROGRAM, MULTIPLE DATA)*
Se divide el proceso en varias tareas que se ejecutan en paralelo, pero se espera hasta que termine mediante el uso de una instrucción `join`.	Todos los procesos ejecutan el mismo código, pero sobre datos distintos. Este enfoque requiere tener en cuenta el balanceo de la carga para cada proceso.

Tabla 1. Enfoques para la paralelización. Fuente: elaboración propia.

En este grupo de sistemas, existen los sistemas de memoria compartida y los sistemas de memoria distribuida.

8.2. Modelos de computación paralela

En cualquiera de los modelos de computación paralela hay dos aspectos fundamentales que se deben considerar:

- La **forma de paralelismo**, que afecta a la división del proceso.
- Las **necesidades de comunicación** entre las subtareas, si cooperan para obtener una solución común.

Forma de paralelismo

A la hora de implementar una aplicación paralela se debe decidir **cómo se quiere distribuir el trabajo.** Esto da lugar a dos tipos de paralelismo, el **paralelismo a nivel de tareas** y el **paralelismo a nivel de datos.**

En el primero de ellos se crean distintas tareas que contribuyen a un objetivo común, en el segundo, los datos se dividen de forma que cada proceso trabaja con los datos que le corresponden.

Formas de comunicación

Los dos paradigmas principales de comunicación son la **memoria compartida** y el **paso de mensajes,** propio de los sistemas distribuidos.

La computación paralela que utiliza el primero **permite que varias tareas o procesos accedan a zonas de memoria compartida,** de forma que estos pueden realizar operaciones de lectura sobre ella, pero el acceso para escritura se debe realizar en exclusión mutua, es decir, solo puede escribir un hilo o proceso a la vez. Por lo tanto, desde el punto de vista de la programación, es un **modelo simple y conveniente,** pero se deben considerar algunos aspectos para garantizar un acceso coherente a la memoria. En el modelo de **paso de mensajes permite las tareas o procesos, que se están ejecutando simultáneamente, se pasen datos entre sí.** Este modelo es bastante **común y se utiliza ampliamente** en la programación.

Ambos enfoques tienen sus ventajas y desventajas, pero existen algunos aspectos que se deben tener en cuenta en la computación paralela y que condicionan muchas decisiones de programación. Estos aspectos son:

- La **sincronización** de las distintas tareas que conforman un algoritmo que se ejecuta en paralelo

- El **seguimiento** de los pasos de la computación
- El **registro** o la **transmisión** de datos desde varios dispositivos de computación.

8.3. Evaluación del rendimiento en programas paralelos

La computación paralela es compleja y no siempre se consiguen los resultados esperados. Por tanto, es necesario poder evaluar y analizar su rendimiento mediante el uso de distintas métricas que se enumeran a continuación:

Aceleración

En el caso de poder implementar un programa paralelo con tantas tareas como núcleos o procesadores se tienen, se puede decir que el programa se ejecutará tantas veces más rápido que el programa ejecutado en forma secuencial como procesadores hay. Se puede definir mediante la siguiente fórmula, siendo n el número de procesadores disponibles y el tiempo correspondiente expresado mediante T:

$$Aceleración = \frac{T_{ej\_secuencial}}{T_{ej\_paralela}} = \frac{T_{ej\_secuencial}}{\frac{T_{ej\_secuencial}}{n}} = n$$

Lo que induce a pensar que la aceleración es lineal porque la relación entre los tiempos así lo es.

Sobrecostes

Unos aspectos adicionales importantes son los sobrecostes asociados al uso de varios procesadores, especialmente en programas con memoria compartida. Estos están

ocasionados por el uso de mecanismos de exclusión mutua, o los correspondientes al paso de mensajes, que crecen cuantos más procesos se ejecuten.

En este caso, el tiempo de ejecución en paralelo se puede expresar como:

$$T_{ej\_paralela} = \frac{T_{ej\_secuencial}}{n} + T_{sobrecoste}$$

Por tanto, se puede decir que no siempre existe una aceleración lineal con respecto al tiempo en paralelo.

Téngase en cuenta que, cuanto mayor es el tamaño del programa la aceleración aumenta porque, en proporción, la sobrecarga disminuye.

Porcentaje de paralelismo

El porcentaje de paralelismo es un factor que se debe considerar. Se trata de determinar **qué porcentaje de código del programa secuencial se ha podido paralelizar.**

Este porcentaje expresado en decimales se representará como $\%\boldsymbol{dp}$.

De forma que, teniendo en cuenta este porcentaje, el cálculo real del tiempo de ejecución paralela (no se ha considerado los sobrecostes) quedará como:

$$T_{ej_{paralela}} = \%dp \times \frac{T_{ej_{secuencial}}}{n} + T_{sobrecoste} + (1 - \%dp) \times T_{ej_{secuencial}}$$

De acuerdo con esto, se debe tener en cuenta una limitación que se produce cuando el programa no puede ser completamente paralelizado.

Para ello, se debe considerar la ley de Amdahl, enunciada por el arquitecto de ordenadores Gene Amdahl en el año 1967. Esa ley estudia la mejora de un sistema

cuando solo una parte de él es mejorada, es decir, se refiere a que la mejora que se obtiene en el rendimiento de un sistema, al alterar uno de sus componentes, está limitada por la fracción de tiempo que se utiliza dicho componente. (Amdahl,1967)

La fórmula es:

$$T_{mejorado} = T_{original} \times \left(\frac{Fraccion_{mejora}}{Factor_{mejora}} + \left(1 - Fraccion_{mejora}\right) \right)$$

El modelo matemático de la ley presenta la **relación** que existe **entre la aceleración esperada de la ejecución paralela de un algoritmo y su ejecución secuencial** o en serie. Por tanto, de acuerdo con la ley, la aceleración que se está tratando se puede representar como:

$$T_{ej\_paralela} = T_{ej\_secuencial} \times (\frac{\%dp}{n} + (1 - \%dp))$$

Analizando la expresión, se puede deducir que cuanto más crezca el número de procesadores, $\frac{\%dp}{n}$ tiende a 0, luego el tiempo de ejecución paralela nunca podría ser menor que $T_{ej\_secuencial} \times (1 - \%dp)$, con lo que el incremento de procesadores o núcleos no afectaría.

De hecho, se podría establecer que:

$$Aceleracion = \frac{1}{\frac{\%dp}{n} + (1 - \%dp)}$$

La aceleración que se produce en un programa paralelo está limitada por la fracción de programa que no se puede paralelizar, y es independiente del número de núcleos o procesadores.

Esta limitación es compensada por el hecho enunciado en la ley de Gustafson, que afirma que cualquier problema suficientemente te grande puede ser paralelizado de forma eficiente, de manera que:

$$Aceleracion = n - (1 - \%dp) + (n - 1)$$

Eficiencia

La eficiencia de un programa paralelo es la **aceleración dividida entre el número de procesadores.** Se puede expresar como:

$$Eficiencia = \frac{T_{ej_{secuencial}}}{n \times T_{ej_{paralela}}}$$

Es importante hacer notar que la eficiencia también aumenta cuanto más grande es el problema tratado.

8.4. El módulo `multiprocessing`

El módulo `multiprocessing` es un **paquete que soporta la generación de procesos,** utilizando una interfaz de programación similar a la del módulo de concurrencia, `Threading`. Este módulo hace uso de subprocesos. evitando el bloqueo global del intérprete, además de permitir tanto concurrencia local, como remota. Por tanto, este módulo **permite el máximo aprovechamiento de los procesadores de una máquina determinada.**

A continuación, se pasan a describir algunas de las características más importantes de este módulo para la creación y activación de subprocesos. Algunas de estas características dependen de la plataforma de ejecución.

El módulo `multiprocesing` cuenta con cuatro tipos de elementos:

- Una **serie de clases,** entre las que cabe destacar la clase `Process` para definir los procesos propiamente dichos.

- Métodos para la **creación y activación de procesos,** y otros de consulta sobre estados y otras cuestiones de interés.

- Objetos para **sincronización**, que permiten adquirir y liberar cerrojos sobre recursos compartidos.

- **Excepciones para informar de los problemas** que se producen en tiempo de ejecución sobre los procesos, para que puedan ser manejados de forma conveniente.

La clase `Process`

Un proceso en Python es un objeto de la clase `Process` al que se le asigna el código que debe ejecutar, generalmente implementado mediante una función. En esta clase se cuenta con las **operaciones básicas que se deben usar para,** entre otras cosas, **la activación de los procesos.** Los objetos de estas clases representan secuencias de código que se ejecutan de forma autónoma.

Los pasos que hay que seguir para crear procesos y activarlos en Python son:

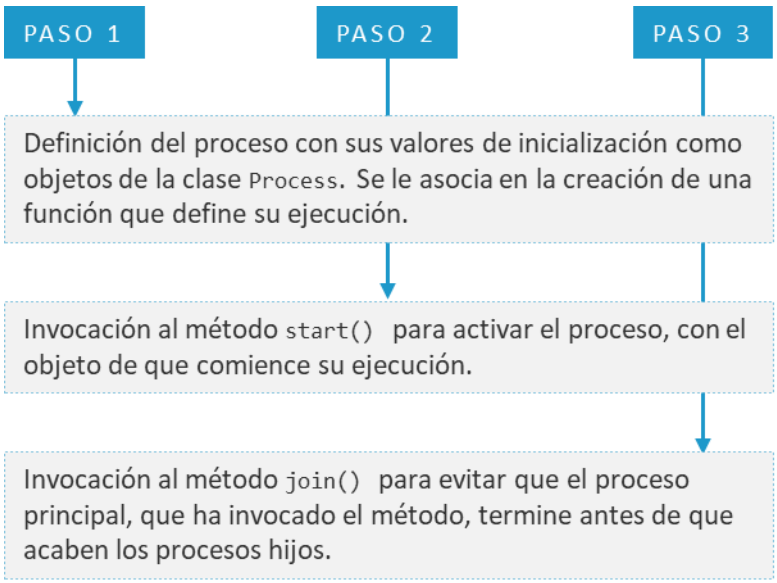


Figura 2. Pasos para la creación y activación de procesos en Python. Fuente: elaboración propia.

En la Tabla 2 se enumeran algunas de las principales funciones para manejar procesos.

Métodos	Tipo	Descripción
`is_alive()`	Método de objeto	Devuelve *True* si el proceso está vivo.
`start()`	Método de clase	Es un método que se invoca una única vez para que el proceso comience su ejecución.
`join()`	Método de clase	Bloquea la finalización de otro código hasta que el proceso para el que se llamó al método `join()` finalice.

Tabla 2. Métodos del módulo `multiprocessing` para procesos. Fuente: elaboración propia.

Se verán a continuación unos ejemplos de código en los que se compara el uso de un único hilo, el principal, con uno con varios hilos y varios procesos. Todos ejecutan la misma función. Además, se miden los tiempos para entender las formas de ejecución.

Ejemplo 1

Se ejecuta una función que imprime una información y duerme el hilo o proceso por un tiempo.

```
import os
import time
import threading
import multiprocessing

NUMERO_ACTIVIDADES = 6

def dormir():
    """ solo espera a que pase el tiempo"""
    print("IDProceso: %s, Nombre  Proceso: %s, Nombre Hilo: %s \n" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    "se obtiene el identificador del proceso actual, el nombre del proceso"
    "que se ejecuta y el nombre del hilo"
    time.sleep(1)



if __name__=='__main__':
    multiprocessing.freeze_support()
    tiempo_inicial = time.time()
    for _ in range(NUMERO_ACTIVIDADES):
        dormir()
    tiempo_final= time.time()

    print("Tiempo ejecución secuencial=", tiempo_final - tiempo_inicial)

    tiempo_inicial = time.time()
    threads = [threading.Thread(target=dormir) for _ in range(NUMERO_ACTIVIDADES)]
    [thread.start() for thread in threads]
    [thread.join() for thread in threads]
    tiempo_final = time.time()

    print("Tiempo ejecución concurrente=", tiempo_final - tiempo_inicial)


    tiempo_inicial = time.time()
    processes = [multiprocessing.Process(target=dormir) for _ in range(NUMERO_ACTIVIDADES)]
    [process.start() for process in processes]
    [process.join() for process in processes]

    tiempo_final = time.time()

    print("Tiempo  ejecución paralela=", tiempo_final - tiempo_inicial)
```

Figura 3. Código del Ejemplo 1. Fuente: elaboración propia.

La salida que se obtiene es:

Figura 4. Salida del código del Ejemplo1. Fuente: elaboración propia.

Ejemplo 2

Se ejecuta una función que realiza unos cálculos.

```python
import os
import time
import threading
import multiprocessing

NUMERO_ACTIVIDADES = 6

def  incrementar():
    """ Incrementa un número grande de veces  la variable contador """
    print("IDProceso: %s, Nombre  Proceso: %s, Nombre Hilo: %s" %  (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    "se obtiene el identificador del proceso actual, el nombre del proceso"
    "que se ejecuta y el nombre del hilo"
    x = 0
    while x < 10000000:
        x += 1


if __name__=='__main__':

    tiempo_inicial = time.time()
    for _ in range(NUMERO_ACTIVIDADES):
        incrementar()
    tiempo_final= time.time()

    print("Tiempo ejecución secuencial=", tiempo_final - tiempo_inicial)

    tiempo_inicial = time.time()
    threads = [threading.Thread(target=incrementar) for _ in range(NUMERO_ACTIVIDADES)]
    [thread.start() for thread in threads]
    [thread.join() for thread in threads]
    tiempo_final = time.time()

    print("Tiempo ejecución concurrente=", tiempo_final - tiempo_inicial)


    tiempo_inicial = time.time()
    processes = [multiprocessing.Process(target=incrementar) for _ in range(NUMERO_ACTIVIDADES)]
    [process.start() for process in processes]
    [process.join() for process in processes]

    tiempo_final = time.time()

    print("Tiempo  ejecución paralela=", tiempo_final - tiempo_inicial)
```

Figura 5. Código del Ejemplo 2. Fuente: elaboración propia.

La salida que se obtiene es:

```
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: MainThread
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: MainThread
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: MainThread
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: MainThread
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: MainThread
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: MainThread
Tiempo ejecución secuencial= 2.8272247314453125
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: Thread-141
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: Thread-142
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: Thread-143
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: Thread-144
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: Thread-145
IDProceso: 3471, Nombre  Proceso: MainProcess, Nombre Hilo: Thread-146
Tiempo ejecución concurrente= 2.766195058822632
IDProceso: 17168, Nombre  Proceso: Process-109, Nombre Hilo: MainThread
IDProceso: 17170, Nombre  Proceso: Process-111, Nombre Hilo: MainThread
IDProceso: 17171, Nombre  Proceso: Process-112, Nombre Hilo: MainThread
IDProceso: 17169, Nombre  Proceso: Process-110, Nombre Hilo: MainThread
IDProceso: 17166, Nombre  Proceso: Process-107, Nombre Hilo: MainThread
IDProceso: 17167, Nombre  Proceso: Process-108, Nombre Hilo: MainThread
Tiempo  ejecución paralela= 0.6820130348205566
```

Figura 6. Salida del código del Ejemplo 2. Fuente: elaboración propia.

Al igual que para los hilos, se pueden definir procesos como objetos de clases derivadas de `Process`, debiéndose sobrecargar el método run correspondiente.

Sincronización entre procesos

El módulo `multiprocessing` cuenta con los métodos o primitivas de sincronización equivalentes a las del módulo `threading`. Entre ellas cabe destacar:

- Cerrojos o bloqueos sobre recursos (`Lock`).
- Bloqueos o cerrojos recursivos o reentrantes (`Rlock`).
- Semáforos (`Semaphore`).
- Condiciones (`Condition`).

Los métodos asociados a estas primitivas son los mismos que los que se definen en la clase `Threading` y su función es la misma, solo que esta vez se aplican sobre procesos.

Estados de un proceso

De acuerdo con las formas de sincronización se puede determinar que un proceso pasa por los siguientes estados.

- **Creado:** se ha invocado al constructor para construir el proceso, pero todavía no ha empezado su ejecución.

- **Preparado:** se invoca al método `start()` para el proceso y queda a la espera de la asignación del procesador para su ejecución.

- **En ejecución:** se ejecuta el proceso en el procesador asignado.

- **Parado:** el proceso no avanza en la ejecución. Hay básicamente tres estados posibles:
  - El proceso está a la espera de que se cumpla una condición por la ejecución de un `wait()`, sale de esta si se produce un `notify()` o `notifyAll()`.
  - Se ejecuta un `sleep()`, volverá a estar preparado cuando pase el tiempo.
  - El proceso está a la espera de poder adquirir un cerrojo, a la espera de una operación de E/S (entrada/salida) o en una operación de sincronización.

- **Terminado:** el proceso acaba su ejecución porque completa el código de su método asociado, o bien, se produce una excepción no controlada o una interrupción.

En la Figura 7 se muestran los estados de un proceso, que si se observa son como los estados de los hilos. Solo es importante hacer la consideración de que se producirán menos salidas o entradas del procesador porque, en este caso, puede que no se comparta con otros procesos.

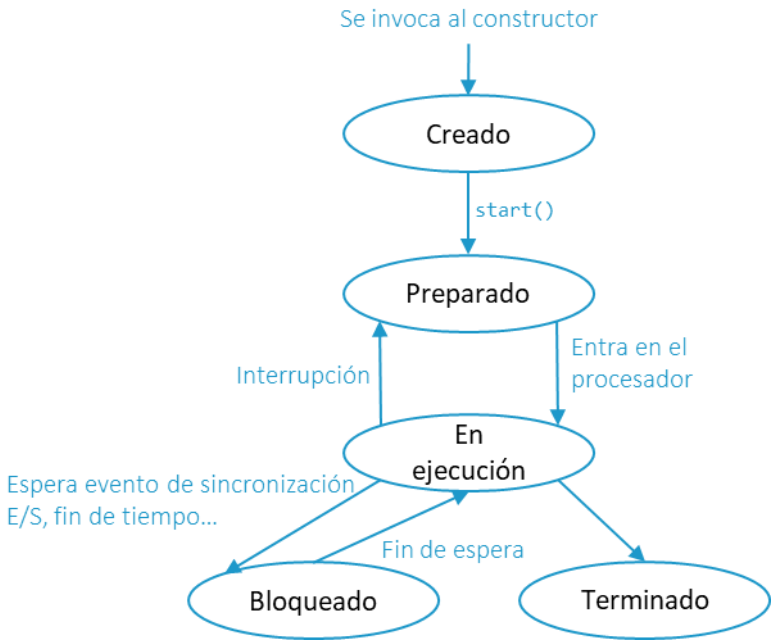


Figura 7. Estados de un proceso. Fuente: Elaboración propia

Paralelización de datos con `Pool`

En el módulo `multiprocessing` se cuenta con una **clase que permite paralelizar las sucesivas llamadas a una función con distintos datos de entrada.** Los pasos básicos que se deben llevar a cabo con `Pool` son:

- Crear un conjunto de procesos entre los que distribuye la ejecución en paralelo.
- Definir la función que se quiere ejecutar para distintos datos de entrada.
- Usar el método `map` sobre la función y los argumentos, para distribuir el trabajo entre los procesos, y que devuelve una lista con todos los resultados obtenidos.

Ejemplo 3

El siguiente ejemplo sencillo muestra cómo crear cinco procesos con `Pool` y paralelizar el cálculo de medias sobre varias listas

```
from multiprocessing import Pool
import numpy as np

def media(l):
    return np.mean(l)

if __name__ == '__main__':
    listas=[[1, 2, 3],[5, 2, 7],[4, 4, 4]]
    with Pool(5) as p:
        print(p.map(media, listas))
```

Figura 8. Código del Ejemplo 3. Fuente: elaboración propia.

La salida obtenida es:

```
[2.0, 4.666666666666667, 4.0]
```

Figura 9. Salida del código del Ejemplo3. Fuente: elaboración propia.

Para profundizar sobre el funcionamiento y el rendimiento de estos ejercicios visualiza la clase magistral **Ejecución paralela en Python con Pool y análisis de rendimiento** y descarga el notebook pool.ipynb.

Accede al vídeo

Comunicación entre procesos

En el módulo `multiprocessing` se admiten dos formas de comunicación entre procesos, las colas y las tuberías o pipes.

Colas

En este módulo se usa la clase `Queue` y no existen colas con prioridad. Estas son seguras para poder acceder por varios procesos, porque se garantiza el acceso en exclusión mutua. En el siguiente ejemplo se muestra su uso consistente.

Ejemplo 4

Se crean cinco procesos que insertan valores en una cola sin que se produzcan problemas de estados inconsistentes en ella.

```
import multiprocessing
from multiprocessing import Process, Queue
import random


def insertar_valor(cola):

    valor = random.random()
    cola.put(valor)
    print ("Proceso "+multiprocessing.current_process().name+ " inserta valor "+str(valor))


if __name__ == "__main__":
    cola = Queue()

    procesos = [Process(target=insertar_valor, args=(cola,)) for _ in range(5)]

    for p in procesos:
        p.start()

    for p in procesos:
        p.join()

    resultado = [cola.get() for _ in procesos]
    print(resultado)
```

Figura 10. Código del Ejemplo 4. Fuente: elaboración propia.

Los resultados obtenidos son:

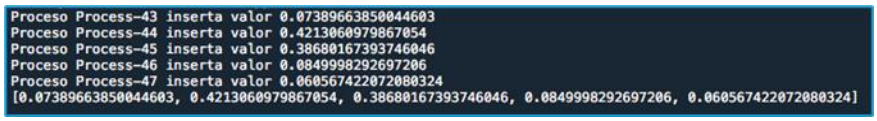


Figura 11. Salida del código del Ejemplo 4. Fuente: elaboración propia.

Pipes

Una tubería o pipe consiste en una **cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo.** Permiten la comunicación y sincronización entre procesos. Es común el uso de «búfer» de datos entre elementos consecutivos.

Las tuberías permiten la comunicación entre procesos, incluso bidireccional.

En este caso, mediante el uso de la función `Pipe()`, se obtienen dos objetos conectados por la tubería. Los métodos que se pueden aplicar sobre los objetos retornados son:

- Método `send()`: para enviar datos.
- Método `recv()`: para recibir datos

Se debe tener en cuenta que estos **métodos pueden ejecutarse de forma simultánea, pero en extremos distintos de la tubería.** Si dos procesos tratan de escribir o leer en el mismo extremo de la tubería a la vez, puede que los datos queden en estado inconsistente.

Ejemplo 5

En este ejemplo se presenta el uso de Pipe con dos procesos.

```python
from multiprocessing import Process, Pipe
def enviar(conn):
    conn.send(["num_contadores",1000, 2000])
    conn.close()
def recibir(conn):
    print(conn.recv())
    conn.close()
if __name__ == '__main__':
    conexion_receptor, conexion_emisor = Pipe()
    emisor = Process(target=enviar, args=(conexion_emisor,))
    receptor=Process(target=recibir, args=(conexion_receptor,))
    receptor.start()
    emisor.start()


    emisor.join()
    receptor.join()
```

Figura 12. Código del Ejemplo 5. Fuente: elaboración propia.

La salida que se muestra es:

```
['num_contadores', 1000, 2000]
```

Figura 13. Salida del código del Ejemplo 5. Fuente: elaboración propia.

Memoria compartida

En esta biblioteca se cuenta con una característica adicional que da mucha flexibilidad, que es la **capacidad de poder contar con una memoria compartida por los procesos, pero accedida de forma segura,** lo que garantiza el acceso en exclusión mutua y, por tanto, la consistencia de la información.

Las memorias compartidas pueden ser variables, objetos de `Value` o `arrays`, que son objetos de `Array`, clases disponibles en el módulo `multiprocessing`. La limitación que se presenta en este caso es que los valores que pueden tomar las variables y los elementos del `array` son objetos de los tipos básicos de CPython, por lo que se debe indicar el tipo del elemento que se usa. Esto quiere decir que no se pueden usar objetos de otras clases como valores.

Ejemplo 6

En el siguiente ejemplo se muestra cómo un proceso accede a una variable y `array` compartidos. En este caso se usa la variable para definir la constante pi y el `array` para pasar los valores de radios de 1 a 9. La función que no tiene interés

de implementación, hay otras formas de hacer esto, tiene como objetivo el cálculo de las longitudes de las circunferencias con los radios dados.

Primero se muestra con un proceso.

```
from multiprocessing import Process, Value, Array
def calcular_longitudes(pi, radios):
    pi.value = 3.1415927
    for i in range(len(radios)):
        radios[i] = 2*pi.value*radios[i]
if __name__ == '__main__':
    valor_cte = Value('d', 0.0)
    array_radios = Array('d', range(1,10))
    print(array_radios[:])
    p1 = Process(target=calcular_longitudes, args=(valor_cte, array_radios))

    p1.start()

    p1.join()

    print (valor_cte.value)
    print (array_radios[:])
```

Figura 14. Código del Ejemplo 6. Fuente: elaboración propia.

Y se obtiene la solución:

```
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
3.1415927
[6.2831854, 12.5663708, 18.8495562, 25.1327416, 31.415927, 37.6991124, 43.9822978, 50.2654832, 56.5486686]
```

Figura 15. Salida del código del Ejemplo 6. Fuente: elaboración propia.

8.5. Paso de mensajes con MPI

La interfaz de paso de mensajes o **MPI** (*Message Passing Interface*) **es un estándar,** cuya primera versión se aprobó y publicó en el año 1994. Es una especificación de la sintaxis y la semántica de las funciones contenidas en una biblioteca, que es **usada para comunicar datos entre procesos que se ejecutan sobre múltiples procesadores.**

Sus características principales son su **amplia funcionalidad** y su **portabilidad,** al poder usarse con entornos de multiprogramación y programación distribuida, así como en entornos heterogéneos.

La especificación detalla las funciones que se pueden utilizar y que son, entre otras:

- Funciones de inicio, gestión y finalización de procesos.
- Funciones de comunicación entre procesos o en un grupo de procesos (comunicador).

Los elementos básicos de MPI son los procesos que cuenta un identificador interno (*rank*) con espacios de memoria independientes, que intercambian información por medio del paso de mensajes. Las características y formas de implementación dependen de los lenguajes que se utilicen, nosotros veremos su implementación en Python.

MPI en Python

En MPI **se define un comunicador como una colección de procesos que se envían mensajes unos a otros.** El comunicador básico se denomina MPI_COMM_WORLD y se define mediante una macro del lenguaje C. MPI_COMM_WORLD agrupa a todos los procesos activos durante la ejecución de una aplicación.

En Python se cuenta con el módulo mpi4y del que se puede importar MPI.

Una forma básica de paraleliza con MPI se basa en tres operaciones básicas: inicialización de MPI con `MPI_Init()`, invocación a la interprete y finalización de MPI con `MPI_Finalize()`.

Se debe tener en cuenta que para poder ejecutar se puede requerir de la instalación de algunos módulos y características para que se soporte el uso de MPI. En el ejemplo siguiente se muestra la importación de MPI que se encuentra en el módulo mpi4py.

> Ejemplo 7
>
> Crear un comunicador y obtener su identificador

```
from mpi4py import MPI

comunicador = MPI.COMM_WORLD

print ("El identificador es %d de %d en ejecución..." % (comunicador.rank, comunicador.size))
```

Figura 16. Código del Ejemplo 7. Fuente: elaboración propia.

Salida:

```
El identificador es 0 de 1 en ejecución...
```

Figura 17. Salida del código del Ejemplo 7. Fuente: elaboración propia.

Se usa el comunicador por defecto, que consiste en todos los procesadores.

A continuación, se enumeran las formas de comunicación con las que cuenta este módulo.

- **Comunicación punto a punto:** mecanismos para pasar datos de un proceso a otro. Los métodos que se usan son:
  - `send(dato,destino)`: envía el dato indicando el destinatario (rank del proceso).
  - `recv(fuente)` o `Recv()`: para recibir el dato indicando la fuente de la que proviene (*rank* del proceso).

- **Difusión:** comunicación en grupo mediante el envío de una copia exacta de la información a todos los procesos de un comunicador. El comunicador lo envía haciendo uso del método `bcast(dato,raíz)` o `Bcast(dato,raíz)`.

- Operaciones de dispersión de datos, recolección y reducción con `arrays`.

Para profundizar consulten el recurso recomendado de la sección de A fondo,

Parallel Programming with MPI For Python.

8.6. Introducción a la programación distribuida. Llamadas a procedimientos remotos

En esta sección se presenta, de forma breve, algunas características de la programación distribuida y su forma de uso en Python.

El objetivo de la programación distribuida es el diseño e implementación de sistemas distribuidos. Estos se caracterizan por estar formados por una serie de ordenadores que se comunican, mediante mensajes a través de una red, para cooperar en una o varias tareas. En un sistema distribuido **no hay memoria física compartida, aunque los algoritmos pueden simularla.**

La programación distribuida es de **gran complejidad** y la tarea de escribir aplicaciones distribuidas es ardua, pero es muy adecuada para mejorar el rendimiento de programas con grandes exigencias de computación. El diseño de un programa distribuido **posibilita el uso de recursos y características de conectividad,** que permiten integrar las distintas ejecuciones de forma transparente a las plataformas de proceso y al almacenamiento de datos.

Se trata de identificar en un programa las partes que pueden ejecutarse en paralelo para su ejecución simultánea e independiente, con el objeto de mejorar el rendimiento. En este caso, esas tareas se distribuirán entre distintas máquinas.

Llamada a procedimientos remotos

Un enfoque importante en los sistemas distribuidos es el mecanismo de comunicación, que se conoce como llamada a procedimientos remotos o RPC (*Remote Procedure Call*). Esta solución se implementa mediante una interfaz que proporciona una serie de funciones que pueden ser invocadas para establecer la comunicación. El sistema **RPC gestiona todos los detalles referentes a la**

comunicación, como la codificación de los argumentos de las funciones con representaciones internas que permiten su envío por la red y se encarga de enviarlo a la máquina remota correspondiente, la cual espera la respuesta.

La llamada a procedimientos remotos es un mecanismo de comunicación síncrona, en el que se indica el receptor, y es asimétrica por la información va en un sentido.

RPyC (*Remote Python Call*)

RPyC es un **módulo transparente de Python** para llamadas a procedimientos remotos simétricos, *clustering* y computación distribuida. Para usar las llamadas a procedimientos remotos en Phyton se debe instalar `rpyc`.

Las principales características de este módulo, según se indica en la página oficial, son:

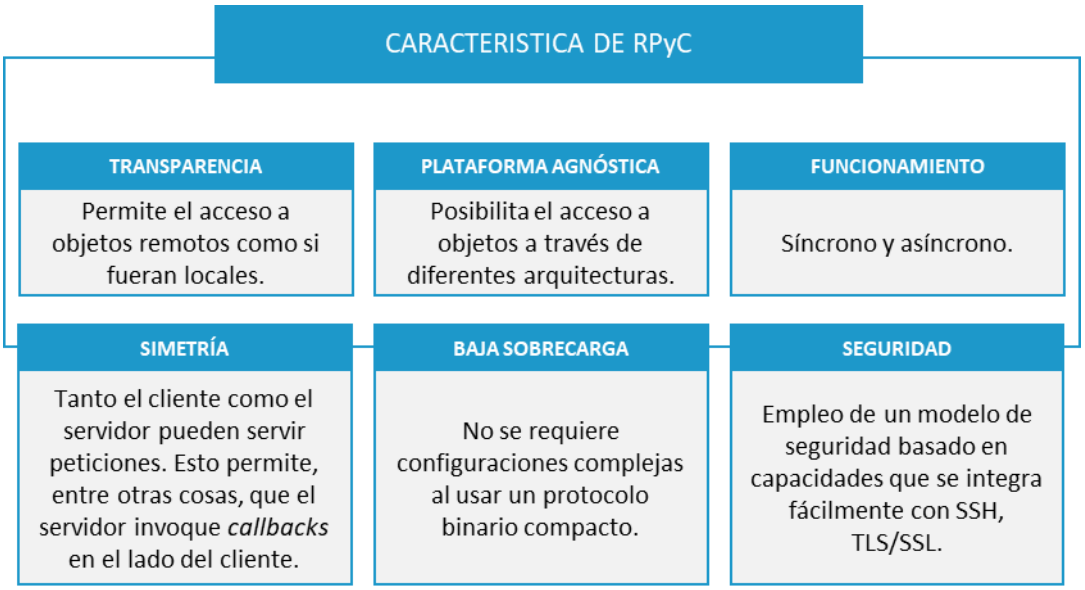


Figura 18. Características de RPyC. Fuente: elaboración propia.

Puede consultar las características, funciones y formas de instalación y configuración en la página oficial de RPyC.

8.7. Gestión con Celery

Es una biblioteca de Python para la gestión de colas distribuidas mediante el enrutamiento de tareas asíncronas, basadas en paso de mensajes distribuidos. Proporciona la capacidad de lanzar servicios gestionándolos como si fuesen tareas. Suele utilizar un «bróker» para transportar los mensajes.

Para el uso de Celery se necesita el módulo `celery` de Python. La forma básica para comenzar consiste en los siguientes pasos:

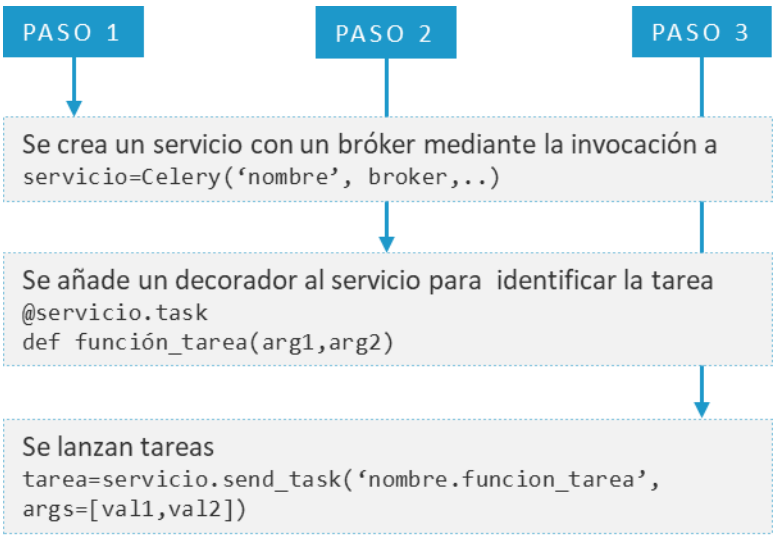


Figura 19. Pasos para utilizar `celery`. Fuente: elaboración propia.

Existen muchas cuestiones que se deben tener en cuenta, como el número de «bróker» o colas de mensajes, los destinarios de los mensajes y los protocolos de envío de mensajes o la manera de enrutarlos.

Profundiza el uso de Celery en su página y la guía proporcionada en la sección de A fondo.

8.8. Referencias bibliográficas

Amdahl, G. M. (18-20 de abril de 1967). *Validity of the single processor approach to achieving large scale computing capabilities* [Presentación en papel], pp. 483-485. Association for Computing Machinery, New York, United States. https://doi.org/10.1145/1465482.1465560

Types distributed systems (13 de diciembre de 2020). En *Wikipedia*. File:Types distributed systems.jpg - Wikimedia Commons

8.9. Cuaderno de ejercicios

Ejercicio 1. Creación de procesos con clases derivadas de `Process`

En este código solo se crean procesos que escriben su nombre y duermen dos segundos. A uno de ellos se le ha dado un nombre nuevo.

```
import time
from multiprocessing import Process


class ProcesoDurmiente(Process):

    def run(self):

        print("El proceso que va a dormir es "+ self.name)
        time.sleep(3)




if __name__ == '__main__':
    proceso1 = ProcesoDurmiente()
    proceso2 = ProcesoDurmiente(name="Proceso 2")
    proceso1.start()
    proceso2.start()

    proceso1.join()
    proceso2.join()
```

La salida producida es:

```
El proceso que va a dormir es ProcesoDurmiente-202
El proceso que va a dormir es Proceso 2
```

Figura 20. Código del ejercicio 1. Fuente: elaboración propia.

Ejercicio 2. Problema de los lectores-escritores con prioridad de los lectores

Se trata el problema clásico de lectores escritores con procesos. Se ha implementado con un objeto `Value` y el dato compartido como una cola de tamaño 1.

```
import multiprocessing
import time
import random


class LectoreEscritores():
    def __init__(self):

        self.semaforoEscritores = multiprocessing.Semaphore()
        #semaforo que bloquea el acceso a más de un escritor
        self.numLectores = multiprocessing.Value('i',0)   #contador de lectores activos con Value
        self.dato=multiprocessing.Queue(1) #acceso  al valor en exclusión mutua
        self.condicionVacio=multiprocessing.Condition()#espera por si no hay dato
        #la condición se usa para que asegurar que un lector no lee uun valor nulo


    def lector(self,dato):
        print ("ha entrado")
        while True:
            with self.condicionVacio:
                if self.dato==None:
                    self.condicionVacio.wait()#el hilo no avanza si el dato está vacío

            self.numLectores+=1       #incremento del contador de lectores, en exclusión mutua
            if self.numLectores == 1:
                #se toma el semaforo de los escritores para que no entre ninguno
                # si lo tuviese un escritor no podría avanzar
                self.semaforoEscritores.acquire()
                 #se LIBERA el semaforo para dar paso a más lectores

            self.dato.get(dato)
            print(f"El {threading.currentThread().getName()} leyendo\n")
            print(f"Hay {self.numLectores} leyendo {dato}\n")


            self.numLectores-=1   #se decrementa el contador de lectores, en exclusión mutua
            print(f"Hay {self.numLectores} leyendo\n")
            if self.numLectores == 0: #si no quedan lectores se libera  el semaforo de los escritores
                self.semaforoEscritores.release()


            time.sleep(3)

    def escritor(self):
        while True:
            self.semaforoEscritores.acquire()
            #adquiere el semaforo y ejecuta las acciones en exclusión mutua

            dato=random.random()
            print(f"El {threading.currentThread().getName()} escribiendo\n")
            print("Escribiendo datos "+str(dato))
            with self.condicionVacio:
                if self.dato==None:
                    self.condicionVacio.notifyAll()
                time.sleep(3)
            self.dato.put(dato)

            self.semaforoEscritores.release()      #se libera el semáforo
            time.sleep(3)




if __name__=="__main__":
        lecEsc= LectoreEscritores()
        dato=0

        p1 = multiprocessing.Process(target = lecEsc.lector, args=((dato),), name="Lector 1")
        p1.start()
        p2 =multiprocessing.Process(target = lecEsc.escritor,  name="Escritor 1")
        p2.start()
        p3 = multiprocessing.Process(target = lecEsc.lector, args=((dato),), name="Lector 2")
        p3.start()
        p4 = multiprocessing.Process(target = lecEsc.lector, args=((dato),), name="Lector 3")
        p4.start()

        p6 = multiprocessing.Process(target = lecEsc.lector, args=((dato),), name="Lector 4")
        p6.start()
```

Figura 21. Código del Ejercicio 2. Fuente: elaboración propia.

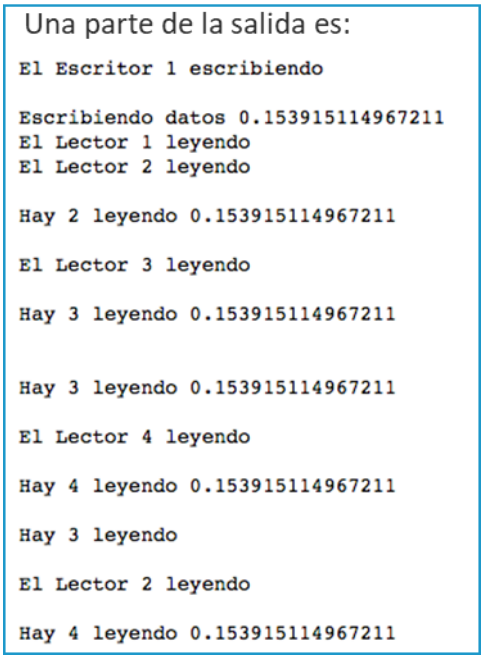


Figura 21. Código del Ejercicio 2. Fuente: elaboración propia.

Ejercicio 3. Uso de `Value` y `Array` con dos procesos

Se añade una función que calcula las áreas de las circunferencias, con radios de 1 a 9, y un proceso que la invoca. La constante pi y el `array` de radios se definen como un objeto `Value` y un objeto `Array`.

El ejercicio solo pretende mostrar como estos objetos tienen asegurado el acceso y modificación en exclusión mutua. De esta forma, como las funciones modifican el `array`, los valores resultantes tras la ejecución de los dos procesos no son exactamente los esperados.

El primer proceso que ejecuta la función lo hará sobre el `array` inicializado [1,2,3,4,5,6,7,8,9]. El segundo debe esperar a que el primero acabe de modificar el `array` y, por tanto, los valores de los radios que usa el segundo proceso serán los valores de las longitudes o las áreas, dependiendo de quien haya entrado.

Está claro que los valores calculados no son inconsistentes, pero no refleja los valores de las áreas y las longitudes con los radios dados. Una buena solución sería crear listas o `arrays` nuevos con los valores de las áreas y longitudes.

Sirve únicamente como ejemplo de `Array` y `Value` protegidos.

```
from multiprocessing import Process, Value, Array
def calcular_longitudes(pi, valores):
    pi.value = 3.1415927
    for i in range(len(valores)):
        valores[i] = 2*pi.value*valores[i]

def calcular_areas(pi, valores):
    pi.value = 3.1415927
    for i in range(len(valores)):
        valores[i] = pi.value*valores[i]**2

if __name__ == '__main__':
    valor_cte = Value('d', 0.0)
    array_valores = Array('d', range(1,10))
    print(array_valores[:])
    p1 = Process(target=calcular_longitudes, args=(valor_cte, array_valores))
    p2 = Process(target=calcular_areas, args=(valor_cte, array_valores))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print (valor_cte.value)
    print (array_valores[:])
```

Figura 22. Uso de `Value` y `Array` con dos procesos. Fuente: elaboración propia.

Si se ejecuta varias veces, se producen distintas salidas, una corresponde a que el primer proceso que entró fue el que calcula longitudes, y otra cuando el primero que entró es el que calcula áreas.

Las salidas posibles son:

```
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
3.1415927
[19.73920938538658, 78.95683754154632, 177.65288446847921, 315.8273501661853, 493.4802346346645, 710.6115378739169, 967.2212598839425,
1263.3094006647411, 1598.875960216313]
```

Figura 23. Resultado I de la ejecución del código de la Figura 22. Fuente: elaboración propia.

Entró primero el proceso que calcula áreas.

Figura 23. Resultado II de la ejecución del código de la Figura 22. Fuente: elaboración propia.

Entró primero el que calcula longitudes.

Ejercicio 4. Paralización de un `for` de llamadas a una función sobre distintos datos de entrada

El uso de `Pool` permite distribuir las distintas entradas a la misma función entre distintos procesos. De forma que el procesamiento se realiza en paralelo, mediante el uso del método `map`.

En este ejercicio se van a hacer 10 llamadas a una función que recibe una tabla de números como matriz, la transpone y devuelve una lista con la suma de sus filas. Se usan 10 tablas (con los mismos valores) de tamaño 10100x40.

El hecho de que todas tengan los mismos valores no es relevante para lo que se quiere mostrar. Se ejecutarán primero las llamadas de forma secuencia, mediante un `for` y se verá el tiempo de ejecución. Posteriormente, se ejecuta en paralelo.

El tamaño de las tablas es grande porque para tamaños pequeños no se mejora el rendimiento con el paralelismo. Por tanto, estas soluciones están pensadas para procesamiento de datos masivos.

Los resultados que se obtienen son listas con la suma de las filas de la traspuesta, es decir, cuarenta sumas.

Consulte el notebook en aula virtual y ejecútelo para ver las diferencias entre los tamaños de tablas y las formas y tiempo de ejecución.

Se muestra un extracto de una tabla que se va a usar, teniendo en cuenta que tiene 40 columnas y 10100 filas.

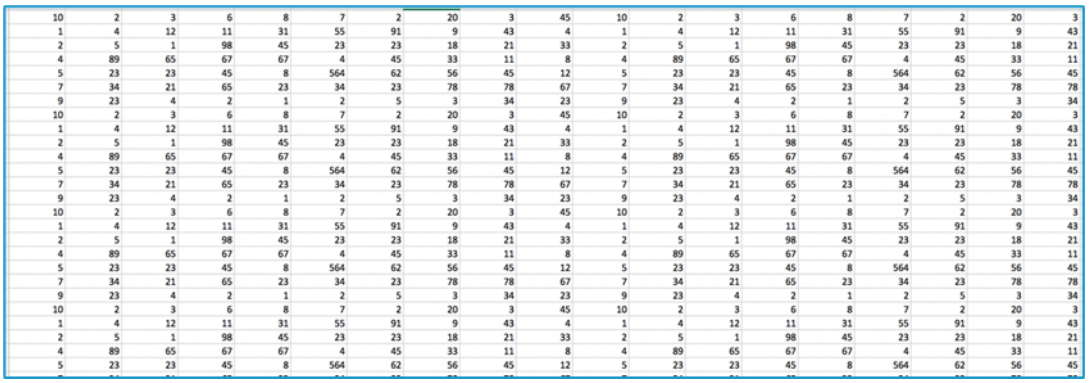


Figura 25. Tabla de datos de entrada para uso paralelo de `for`. Fuente: elaboración propia.

El código es el siguiente:

```
from glob import glob
from numpy import genfromtxt
import time
from multiprocessing import Pool

def procesar_tabla(archivo):
      tablas = genfromtxt(archivo, delimiter=';')
      traspuesta = [list(x) for x in zip(*tablas)] # tabla de datos  traspuesta
      suma_filas = [sum(fila) for fila in traspuesta]
      return suma_filas


if __name__=="__main__":
    lista=glob("*.csv")
    #carga la lista de archivos .csv
    print("Archivos procesados ")
    print(lista)
    #lista de  los archvios que se cargan
    tiempo_inicial = time.time()
    resultados = []
    for archivo in lista:
        #se ejecuta   la llamada tantas veces como archivos
        resultados.append(procesar_tabla(archivo))

    print (resultados)
    # se imprimen los resultado en una  lista de listas
    # habra tantas listas internas  como archivos y dentro de cada lista tantas sumas como columnas
    #tenía la tabla original
    tiempo_final = time.time()
    print("Tiempo usado ejecución secuencial: "+str(tiempo_final-tiempo_inicial))



    p = Pool(6) # crea un pool con 6 procesos
    tiempo_inicial = time.time()
    resultados = p.map(procesar_tabla, lista) # los hilos ejecutan el trabajo
    tiempo_final = time.time()

    print( resultados)
    print("Tiempo usado ejecución paralela: "+str(tiempo_final-tiempo_inicial))
```

Figura 26. Código para la ejecución paralela de un `for` sobre distintos conjuntos de datos mediante la llamada a una función con un parámetro. Fuente: elaboración propia.

La salida es:

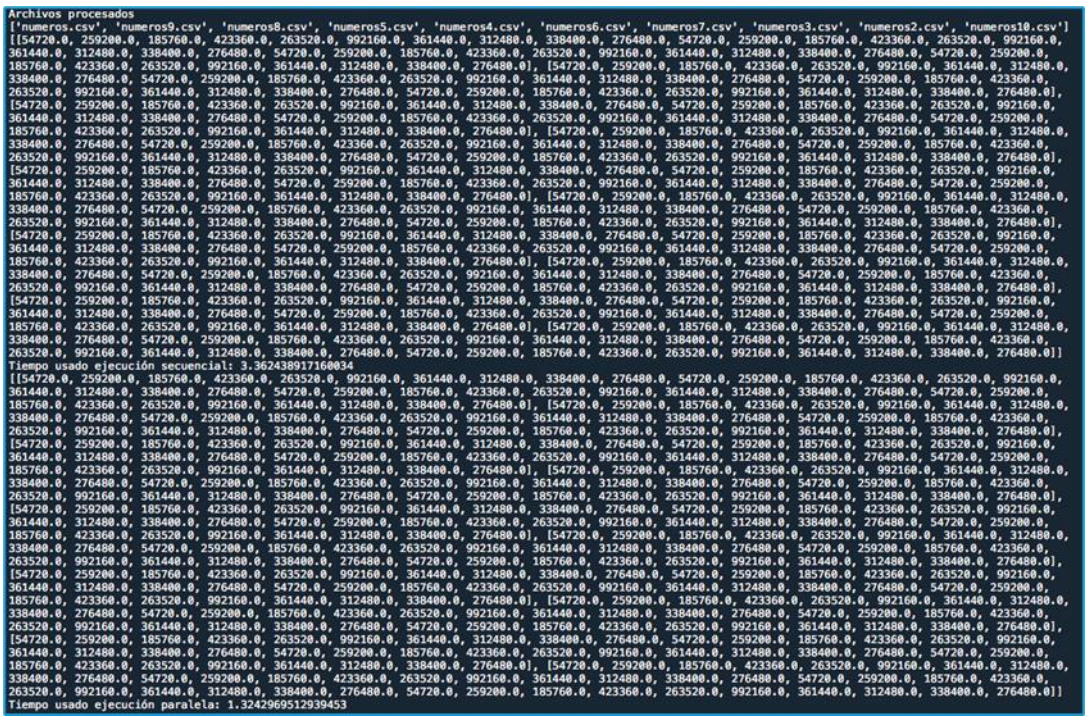


Figura 27. Resultado de la ejecución del código anterior. Fuente: elaboración propia.

Como se puede ver, el tiempo de ejecución paralela es menos de la mitad, usando seis procesos en una máquina con seis núcleos.

Ejercicio 5. Paralización de un `for` de llamadas a una función con dos parámetros sobre distintos datos de entrada

El uso de `Pool` permite distribuir las distintas entradas a la misma función entre distintos procesos. De forma que el procesamiento se realiza en paralelo, mediante el uso del método `map`. Además, en este caso, y debido a que la función tendrá más de un parámetro, se usará la función `partial` del módulo `functools`, que permite agregar argumentos para la llamada con `map`.

En este ejercicio se van a hacer 10 llamadas a una función que recibe una tabla de números como matriz, la transpone y devuelve una lista con las medias ponderadas

de sus filas (antiguas columnas). En el ejercicio se usan 10 tablas (con los mismos valores) de tamaño 10100x40.

El hecho de que todas tengan los mismos valores no es relevante para lo que se quiere mostrar. Se ejecutará primero las llamadas de forma secuencial mediante un `for` y se verá el tiempo de ejecución y, posteriormente, se ejecutará en paralelo.

El tamaño de las tablas es grande porque para tamaños pequeños no se mejora le rendimiento con el paralelismo. Por tanto, estas soluciones están pensadas para procesamiento de datos masivos.

Los resultados que se obtienen son listas con la suma de las filas de la traspuesta, es decir, cuarenta sumas.

Consulte el notebook en aula virtual y ejecútelo para ver las diferencias entre los tamaños de tablas y las formas y tiempo de ejecución.

Se muestra un extracto de una tabla que se va a usar, teniendo en cuenta que tiene 40 columnas y 10100 filas.

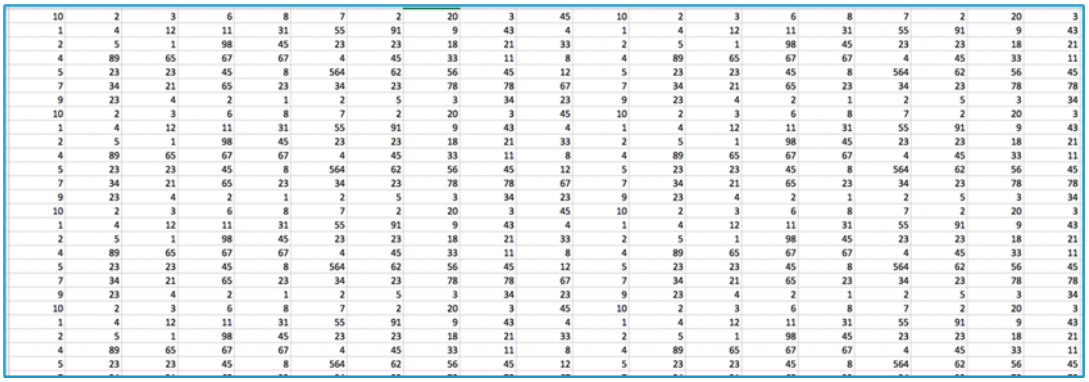


Figura 28. Tabla de datos de entrada para uso paralelo de `for`. Fuente: elaboración propia.

El código es el siguiente:

```
from glob import glob
from numpy import genfromtxt
import time
from multiprocessing import Pool
import random
from functools import partial

def procesar_tabla(pesos,archivo):
      tablas = genfromtxt(archivo, delimiter=';')
      n=len(tablas)
      transpuesta = [list(x) for x in zip(*tablas)] # tabla de datos  traspuesta
      medias=[sum(f*p for f,p in zip(fila,pesos))/n for fila in transpuesta]



      return medias


if __name__=="__main__":
    lista=glob("*.csv")
    print(lista)
    pesos = [random.random() for _ in range(0, 10080)] # crea la lista de peso con valores entre 0 y 1
    tiempo_inicial = time.time()
    resultados = []
    for archivo in lista:
        resultados.append(procesar_tabla(pesos,archivo))

    print (resultados)
    tiempo_final = time.time()
    print("Tiempo usado ejecución secuencial: "+str(tiempo_final-tiempo_inicial))



    p = Pool(6) # crea un pool con 6 procesos
    tiempo_inicial = time.time()

    procesar_tabla_con_peso = partial(procesar_tabla, pesos) # añade los pesos como argumento que se pasará a la función

    resultados = p.map(procesar_tabla_con_peso, lista) # los hilos ejcutan el trabajo



    tiempo_final = time.time()

    print( resultados)
    print("Tiempo usado ejecución paralela: "+str(tiempo_final-tiempo_inicial))
```

Figura 29. Código de la ejecución paralela de un `for` de llamadas a una función con dos parámetros sobre distintos datos de entrada. Fuente: elaboración propia.

La salida es:

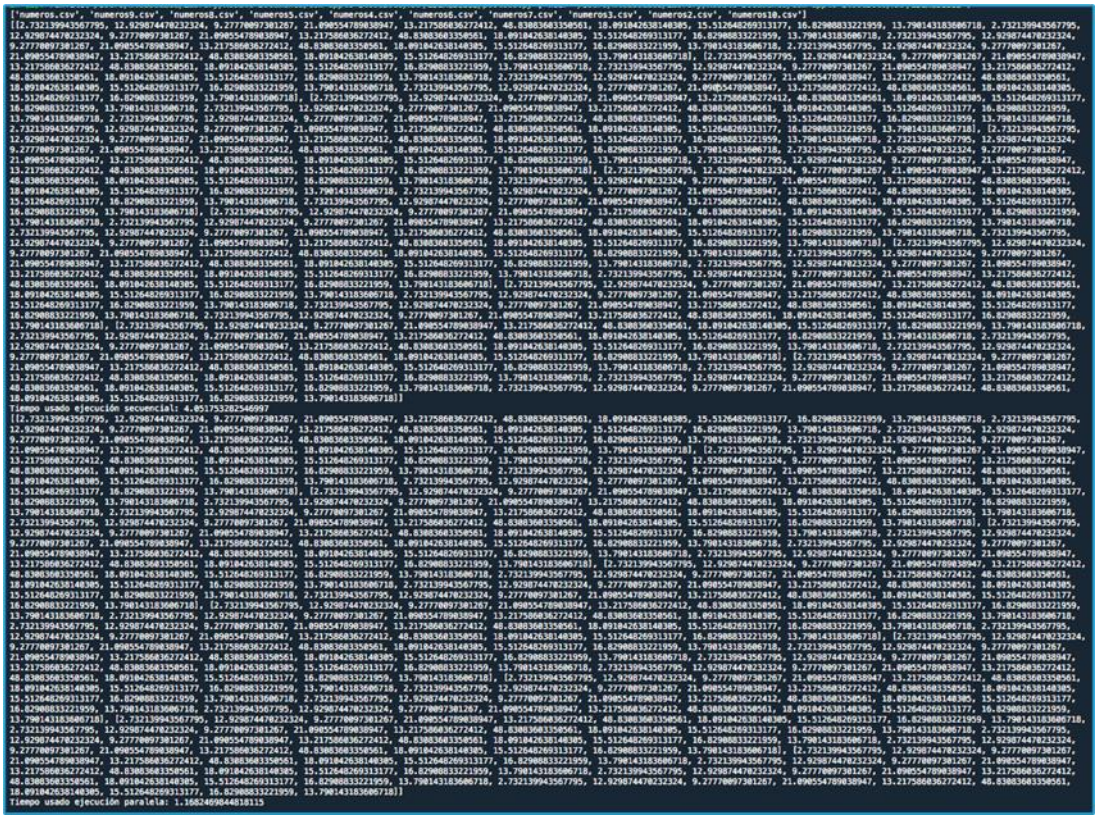


Figura 30. Resultado de la ejecución del código anterior. Fuente: elaboración propia.

Como se puede ver, el tiempo de ejecución paralela en este caso todavía ha mejorado más que en el caso anterior, usando seis procesos en una máquina con seis núcleos.

De cualquier forma, estos tiempos siempre son orientativos, porque dependen del estado de los recursos sobre los que se ejecuta.

A fondo

Parallel Programming with MPI for Python

Parallel Programming with MPI for Python (16 de noviembre de 2017). Rabernat GitHub. Parallel Programming with MPI For Python - Research Computing in Earth Sciences (rabernat.github.io)

Profundiza en la implementación y uso de MPI en Python.

Tutorial del documento oficial de RPyC 3 en español

Tutorial del documento oficial de la biblioteca Python RPyC 3 (s. f.). Programmer click. Tutorial del documento oficial de la biblioteca Python RPyC 3 - programador clic (programmerclick.com)

Este tutorial cubre todas las funcionalidades de RPyC para llamadas a procedimientos remotos.

AMQP, RabbitMQ and Celery - A Visual Guide For Dummies

Tiwari, A. (22 de enero de 2013). *AMQP, RabbitMQ and Celery - A Visual Guide For Dummies*. Abhishek Tiwari. AMQP, RabbitMQ and Celery - A Visual Guide For Dummies (abhishek-tiwari.com)

En esta guía se detalla el uso de Celery con el bróker RabbitMQ y el protocolo de paso de mensajes AMPQ (*Advanced Message Queuing Protocol*).

Test

1. La programación paralela:

A. Se basa en el uso de un único procesador que ejecutan las distintas tareas en las que se ha subdividido un programa.

B. Se basa en el uso de varios procesadores que ejecutan las distintas tareas en las que se ha subdividido un programa.

C. No realiza, en ningún caso, subdivisión de un programa.

D. Ninguna de las anteriores es verdadera

2. Un entorno de multiprocesamiento:

A. Cuenta con múltiples procesadores con memoria compartida

B. Cuenta con múltiples procesadores sin memoria compartida.

C. Cuenta con un único procesador.

D. Ninguna de las anteriores es verdadera.

3. Un proceso es:

A. Una entidad de ejecución independiente que cuenta con registro de activación propio.

B. Una entidad de ejecución independiente que no cuenta con registro de activación propio.

C. Una entidad de ejecución independiente que no puede sincronizarse con otros procesos.

D. Ninguna de las anteriores es verdadera.

4. Las formas de paralelismo que se pueden llevar a cabo son:

A. Solo a nivel de tareas.

B. Solo a nivel de datos.

C. A nivel de tareas y datos.

D. Ninguna de las anteriores es verdadera.

5. Los procesos en Python:

A. Solo se crean como objetos de la clase `Process` del módulo `multiprocessing`.

B. Solo se crean como objetos de una clase derivada de `Process` del módulo `multiprocessing`.

C. Se crean como objetos de la clase `Process` o de clases derivadas de esta.

D. Ninguna de las anteriores es verdadera.

6. Los estados principales por los que puede pasar un proceso en Python son:

A. Creado, en ejecución y terminado.

B. En ejecución y terminado.

C. Creado, preparado, bloqueado y terminado.

D. Ninguna de las anteriores es verdadera

7. `Pool` permite crear procesos que:

A. Ejecutan la misma función sobre datos distintos de forma secuencial.

B. Ejecutan la misma función sobre datos distintos de forma paralela.

C. Ejecutan la misma función sobre los mismos datos de forma secuencial.

D. Ninguna de las anteriores es verdadera.

8. Las colas del módulo de `multiprocessing`:

A. Son colas seguras para el acceso por varios procesos.

B. No son colas seguras para el acceso por varios procesos y las llamadas a sus funciones deben ir precedidas de la adquisición de un cerrojo.

C. Son colas seguras para el acceso por varios procesos, si solo tienen tamaño 1.

D. Ninguna de las anteriores es verdadera.

9. Los objetos de `Array` del módulo de `multiprocessing`:

A. Pueden contener objetos de cualquier tipo.

B. Solo puede contener objetos de tipos básicos de CPtython.

C. Pueden contener solo objetos de tipo entero.

D. Ninguna de las anteriores es verdadera.

10. La aceleración de un programa paralelo:

A. Está limitada por la fracción del programa que no se puede paralelizar.

B. Está limitada por número de procesadores.

C. Aumenta de forma continua según aumenta el número de procesadores.

D. Ninguna de las anteriores es verdadera.