


Aplicaciones para Ambientes Distribuidos

Laboratorio # 6 – Prof. Regis Rivera

Objetivo: Crear aplicaciones que apliquen procesamiento paralelo con Python

Nota: Los nombres de archivos de cada aplicación están dados en el icono 

Procesamiento paralelo en Python

El procesamiento paralelo puede aumentar la cantidad de tareas realizadas por su programa, lo que reduce el tiempo total de procesamiento. Estos ayudan a manejar problemas a gran escala.

En este laboratorio cubriremos los siguientes temas:

- Introducción al procesamiento paralelo
- Biblioteca Python de procesamiento múltiple para procesamiento paralelo
- Marco paralelo de IPython

Introducción al procesamiento paralelo

Para el paralelismo, es importante dividir el problema en subunidades que no dependan de otras subunidades (o menos dependientes). Un problema en el que las subunidades son totalmente independientes de otras subunidades se llama vergonzosamente paralelo.

Por ejemplo, una operación por elementos en una matriz. En este caso, la operación debe ser consciente del elemento particular que está manejando en ese momento.

En otro escenario, un problema que está dividido en subunidades tiene que compartir algunos datos para realizar operaciones. Esto da como resultado un problema de rendimiento debido al costo de comunicación.

Hay dos formas principales de manejar programas paralelos:

- Memoria compartida
En la memoria compartida, las subunidades pueden comunicarse entre sí a través del mismo espacio de memoria. La ventaja es que no es necesario manejar la comunicación explícitamente porque este enfoque es suficiente para leer o escribir desde la memoria compartida. Pero el problema surge cuando varios procesos acceden y cambian la misma ubicación de memoria al mismo tiempo. Este conflicto se puede evitar utilizando técnicas de sincronización.
- Memoria distribuida
En la memoria distribuida, cada proceso está totalmente separado y tiene su propio espacio de memoria. En este escenario, la comunicación se maneja explícitamente entre los procesos. Dado que la comunicación se produce a través de una interfaz de red, es más costosa en comparación con la memoria compartida.

Los hilos son una de las formas de lograr el paralelismo con la memoria compartida. Estas son las subtareas independientes que se originan a partir de un proceso y comparten memoria. Debido al bloqueo global de intérprete (GIL), los subprocesos no se pueden utilizar para aumentar el

rendimiento en Python. GIL es un mecanismo en el que el diseño del intérprete de Python permite que solo se ejecute una instrucción de Python a la vez. La limitación de GIL se puede evitar por completo utilizando procesos en lugar de subprocesos. El uso de procesos tiene pocas desventajas, como una comunicación entre procesos menos eficiente que la memoria compartida, pero es más flexible y explícito.

Multiprocesamiento para procesamiento paralelo

Utilizando el módulo de multiprocesamiento estándar, podemos paralelizar eficientemente tareas simples mediante la creación de procesos secundarios. Este módulo proporciona una interfaz fácil de usar y contiene un conjunto de utilidades para manejar el envío y la sincronización de tareas.

Proceso y clase de grupo (Process / Pool Class)

A-Proceso

Al subclasificar `multiprocessing.Process`, puede crear un proceso que se ejecute de forma independiente. Al extender el `__init__` método, puede inicializar el recurso y al implementar `Process.run()` el método, puede escribir el código para el subproceso. En el siguiente código, vemos cómo crear un proceso que imprima la identificación asignada:

```
import multiprocessing
import time

class Process(multiprocessing.Process):
    def __init__(self, id):
        super(Process, self).__init__()
        self.id = id

    def run(self):
        time.sleep(1)
```

Para generar el proceso, necesitamos inicializar nuestro objeto Proceso e invocar `Process.start()` el método. Aquí `Process.start()` creará un nuevo proceso e invocará el `Process.run()` método.

```
if __name__ == '__main__':
    p = Process(0)
    p.start()
```

El código posterior `p.start()` se ejecutará inmediatamente antes de que se complete la tarea del proceso `p`. Para esperar a que se complete la tarea, puede utilizar `Process.join()`.

```
if __name__ == '__main__':
    p = Process(0)
    p.start()
    p.join()
```



```

1  import multiprocessing
2  import time
3
4
5  class Process(multiprocessing.Process):
6      def __init__(self, id):
7          super(Process, self).__init__()
8          self.id = id
9
10     def run(self):
11         time.sleep(1)
12         print("Soy el proceso con el ID: {}".format(self.id))
13
14 if __name__ == '__main__':
15     p = Process(0)
16     p.start()
17     p.join()
18     p = Process(1)
19     p.start()
20     p.join()
21
22

```

B-Clase de Grupo (Pool)

La clase Pool se puede utilizar para la ejecución paralela de una función para diferentes datos de entrada. La `multiprocessing.Pool()` clase genera un conjunto de procesos llamados trabajadores y puede enviar tareas usando los métodos `apply/apply_async` y `map/map_async`. Para el mapeo paralelo, primero debe inicializar un `multiprocessing.Pool()` objeto. El primer argumento es el número de núcleos; si no se proporciona, ese número será igual al número de núcleos del sistema.

```

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

```

Veamos con un ejemplo. En este ejemplo, veremos cómo pasar una función que calcula el cuadrado de un número. Usando `Pool.map()` puede asignar la función a la lista y pasar la función y la lista de entradas como argumentos, de la siguiente manera:

```

def square(x):
    return x * x

inputs = [0, 1, 2, 3, 4]
outputs = pool.map(square, inputs)

```

Laboratorio 6.2 Aplicar multiprocessing para procesamiento paralelo, inciso grupo



Lab62.py

```
1 import multiprocessing
2 import time
3
4
5 def square(x):
6     return x * x
7
8 if __name__ == '__main__':
9     pool = multiprocessing.Pool()
10    pool = multiprocessing.Pool(processes=4)
11    inputs = [0,1,2,3,4]
12    outputs = pool.map(square, inputs)
13    print("Input: {}".format(inputs))
14    print("Output: {}".format(outputs))
```

Cuando utilizamos el método de map normal, la ejecución del programa se detiene hasta que todos los núcleos completen la tarea. Al utilizar `map_async()`, el objeto `AsyncResult` se devuelve inmediatamente sin detener el programa principal y la tarea se realiza en segundo plano. El resultado se puede recuperar utilizando el `AsyncResult.get()` método en cualquier momento como se muestra a continuación:

```
outputs_async = pool.map_async(square, inputs)
outputs = outputs_async.get()
```

Laboratorio 6.3 Aplicar multiprocessing para procesamiento paralelo, inciso grupo con async



Lab63.py

```
1 import multiprocessing
2 import time
3
4
5 def square(x):
6     return x * x
7
8 if __name__ == '__main__':
9     pool = multiprocessing.Pool()
10    inputs = [0,1,2,3,4]
11    outputs_async = pool.map_async(square, inputs)
12    outputs = outputs_async.get()
13    print("Output: {}".format(outputs))
```

`Pool.apply_async` asigna una tarea que consta de una única función a uno de los núcleos. Toma la función y sus argumentos y devuelve un objeto `AsyncResult`.

```
results_async = [pool.apply_async(square, i) for i in
range(100)]
results = [r.get() for r in results_async]
```

Laboratorio 6.4 Aplicar multiprocessing para procesamiento paralelo, inciso grupo con async II

```
1 import multiprocessing
2 import time
3
4
5 def square(x):
6     return x * x
7
8
9 if __name__ == '__main__':
10     pool = multiprocessing.Pool()
11     result_async = [pool.apply_async(square, args = (i, )) for i in
12                     range(10)]
13     results = [r.get() for r in result_async]
14     print("Output: {}".format(results))
```



Lab64.py

Marco paralelo de IPython

El paquete paralelo IPython proporciona un marco para configurar y ejecutar una tarea en máquinas de un solo núcleo y en múltiples nodos conectados a una red. En IPython.parallel, debe iniciar un conjunto de trabajadores llamados Motores (Engines) que son administrados por el Controlador (Controller). Un controlador es una entidad que ayuda en la comunicación entre el cliente y el motor. En este enfoque, los procesos de trabajo se inician por separado y esperarán los comandos del cliente indefinidamente.

Los comandos de shell de Ipcluster se utilizan para iniciar el controlador y los motores.

```
$ ipcluster start
```

Después del proceso anterior, podemos usar un shell IPython para realizar tareas en paralelo.

IPython viene con dos interfaces básicas:

- Interfaz directa
- Interfaz basada en tareas

Interfaz directa

Direct Interface le permite enviar comandos explícitamente a cada una de las unidades informáticas. Esto es flexible y fácil de usar. Para interactuar con las unidades, debe iniciar el motor y luego una sesión de IPython en un shell separado. Puede establecer una conexión con el controlador creando un cliente. En el siguiente código, importamos la clase Cliente y creamos una instancia:

```
from IPython.parallel import Client
rc = Client()
rc.ids
```

Aquí, `Client.ids` se dará una lista de números enteros que brindan detalles de los motores disponibles.

Al utilizar la instancia de `Direct View`, puede emitir comandos al motor. Hay dos formas en que podemos obtener una instancia de vista directa:

Indexando la instancia del cliente

```
dview = rc[0]
```

Llamando al método `Direct View.direct_view`

```
dview = rc.direct_view('all')
```

Como paso final, puede ejecutar comandos utilizando el método `Direct View.execute`.

```
dview.execute(' a = 1 ')
```

El comando anterior será ejecutado individualmente por cada motor. Usando el método `get` puede obtener el resultado en forma de un objeto `AsyncResult`.

```
dview.pull(' a ').get()
```

```
dview.push({' a ' : 2})
```

Como se muestra arriba, puede recuperar los datos utilizando el `Direct View.pull` método y enviar los datos utilizando el `Direct View.push` método.

Interfaz basada en tareas

La interfaz basada en tareas proporciona una forma inteligente de manejar las tareas informáticas. Desde el punto de vista del usuario, tiene una interfaz menos flexible pero es eficiente en el equilibrio de carga en los motores y puede volver a enviar los trabajos (Jobs) fallidos, aumentando así el rendimiento.

La clase `LoadBalanceView` proporciona la interfaz basada en tareas utilizando el método `load_balanced_view`.

```
from IPython.parallel import Client  
  
rc = Client()  
  
tview = rc.load_balanced_view()
```

Usando el mapa y el método de aplicación podemos ejecutar algunas tareas. En `LoadBalanceView`, la asignación de tareas depende de cuánta carga hay presente en un motor en ese momento. Esto garantiza que todos los motores funcionen sin tiempos de inactividad.