COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

Práctica 2 - Arquitecturas Avanzadas

Javier Escobar Serrano
Grupo A

Índice

Introducción

Contexto.

Segunda práctica de la asignatura de Big Data. Centrada en el tema 3 de la asignatura.

Problema.

Se trata de resolver el problema de un sistema distribuido de estructura master-slave más avanzado que la práctica 1.

Objetivos.

- Desarrollar las habilidades necesarias para resolver un caso de uso más complejo utilizando un sistema distribuido

Descripción de la práctica

- El alumno debe resolver cómo implementar el cálculo del número Pi usando el método de Montecarlo mediante un sistema distribuido
- Implementar un modelo máster-slave y realizar la comunicación entre ellos a través de sockets. El sistema distribuido a implementar consta de un nodo máster y varios nodos slave
- Utilizar el lenguaje Python para crear 2 scripts, correspondientes al código que se ejecuta en el nodo máster y el código que se ejecuta en los nodos slave
    - Máster - La ejecución se inicia en el nodo máster, preguntando cuantos slaves se van a utilizar en el cálculo y enviando trabajo a los nodos slave utilizando un hilo para cada slave (módulo threading). El nodo máster combina los resultados de los nodos slave para generar el resultado final y debe volver a estar preparado para una nueva ejecución
    - Slave - Debe estar escuchando peticiones de forma continua y abrir un hilo por cada petición que reciba. Una vez acabado su trabajo, envía el resultado de vuelta al nodo máster
- Realizar varias ejecuciones, utilizando 2, 3 y 4 nodos slave

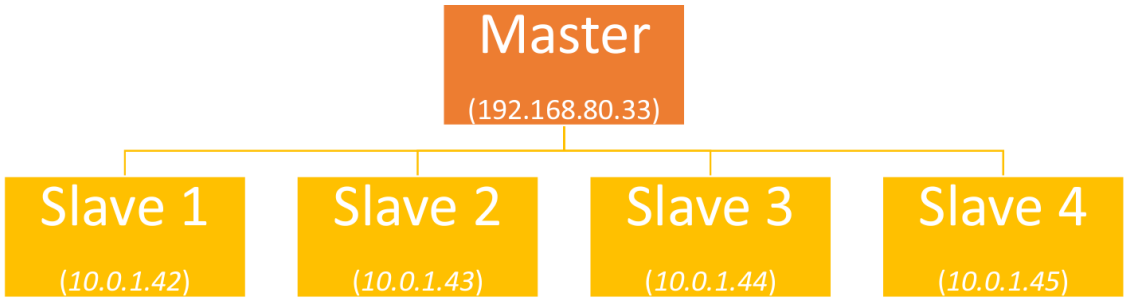
Metodología

Entorno de desarrollo.

Para esta practica he utilizado Python 3.12 junto a la librería de socket, threading, time y random.
Para ejecutarlo he utilizado el sistema distribuido del cluster de ICAI con un sistema operativo de Ubuntu.

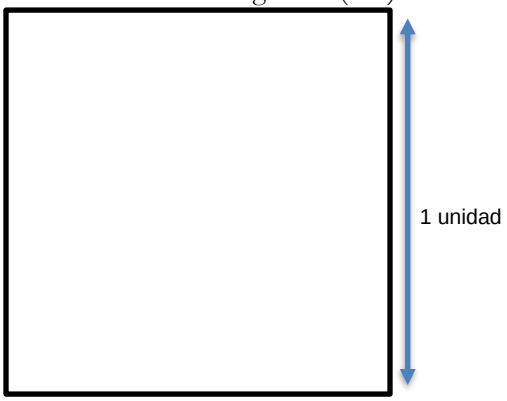
Diseño de la solución.

El sistema consta de un master que controla 2, 3 o 4 slaves.

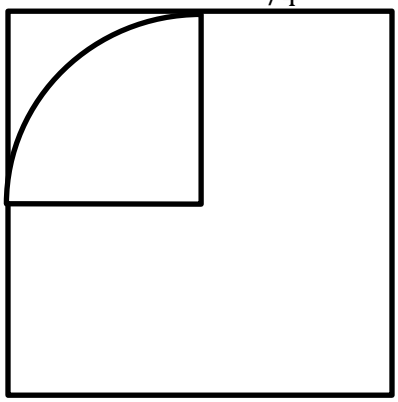


De esta manera, se separa la tarea en dos partes iguales para utilizar expansión horizontal.
El master se inicializa pidiendo cuantos nodos nodos se quieren usar cómo slaves y con cuantos puntos quiere realizar el método de Montecarlo.
El método de Montecarlo se trata de un método que utiliza aleatoriedad para resolver problemas complejos de manera más simple y con posibilidad de partir la computación.
En esta práctica lo hemos aplicado para el cálculo del número pi.
Se siguen los siguientes pasos:

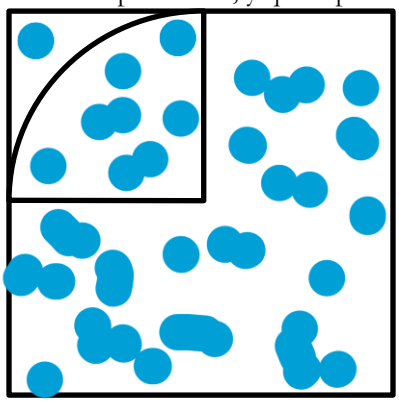
1. Tomamos un cuadrado de longitud 1 (1x1). Por tanto, tiene un área de 1 unidad cuadrada



2. Describimos un cuarto de círculo circunscrito en el cuadrado.
   Este tiene un área de $\pi/4$ $unidades\ cuadradas$



3. Creamos pares de x, y que representan puntos en el plano.



4. Vemos cuantos de estos están dentro del cuarto de círculo respecto del resto.

$$Inside = (x^2 + y^2 \leq 1)$$

5. Finalmente, calculamos pi:

$$\pi \approx 4 \times (\frac{\#Points\ inside}{\#Total\ points})$$

En la ejecución se han utilizado un millón (1 000 000) de puntos distribuidos de manera igual entre todos los nodos.
El nodo master se ejecuta y recibe cuantos puntos tiene que calcular.
Tras esto, crea hilos para cada uno de los nodos con la tarea de calcular cuantos puntos están dentro del círculo.

```
inside = 0
port = 20022
slave_n = int(input("Enter the number of slaves: "))
slave_ips = [f"10.0.1.{i}" for i in range(42, 42 + slave_n)]
sockets = [socket.socket(socket.AF_INET, socket.SOCK_STREAM) for _ in range(slave_n)]

points = int(input("Enter the number of points: "))
points_per_slave = points // slave_n

threads = []
time_ = time.time()
for i in range(slave_n):
    t = threading.Thread(target=thread_task, args=(sockets[i], slave_ips[i], port, points_per_slave))
    threads.append(t)

for t in threads: t.start()
for t in threads: t.join()
```

La tarea de cada hilo consiste en mandar al slave correspondiente el número de puntos a calcular.

```
def thread_task(sock, ip, port, points):
    global inside
    sock.connect((ip, port))
    sock.sendall(str(points).encode())
    print(f"Connected to {ip}, and sent {points} points.")
    data = sock.recv(1024)
    if data:
        inside += int(data.decode())
        sock.close()
```

En este momento, cada slave, al recibir una petición, crea un hilo para esa ejecución.

```
points = int(data.decode())
inside = 0
thread = threading.Thread(target=tarea_hilo, args=(points,))
thread.start()
thread.join()
```

La tarea de este hilo es la de calcular de manera aleatoria una x y una y e utilizar la ecuación para ver si está dentro del cuarto de círculo.

```
def tarea_hilo(n_points):
    global inside
    for _ in range(n_points):
        x = random.random()
        y = random.random()
        if x**2 + y**2 <= 1:
            inside += 1
```

Finalmente, al terminar con todos los puntos, devuelve al master el número de puntos.

```
conn.sendall(str(inside).encode())
```

El master espera a cada uno de los hilos y calcula pi con la fórmula deseada.

```
pi = 4 * inside / points
print(f"Estimated value of pi: {pi}")
print(f"Time elapsed with {slave_n} slaves: {time.time() - time_} seconds")
```

Pruebas realizadas.

Para probar el programa en local, se ha intentado usar la ip de loopback: 127.0.0.1 como valor de la host. Esta IP es una dirección especial que un host utiliza para dirigir el tráfico hacia sí mismo. De este modo, especificando HOST='127.0.0.1' tanto en master.py como en slave.py y ejecutando cada uno en una terminal, podréis simular el entorno del clúster en vuestro ordenador y testear cambios que hagáis en los scripts conforme los estéis desarrollando. De este modo, os ahorráis tener que subir el script por SCP al clúster para testear cada cambio que hagáis.

Sin embargo, para este problema no es posible ya que no se puede usar 2 veces la ip de 127.0.0.1. Por ello se comprueba directamente en el cluster.

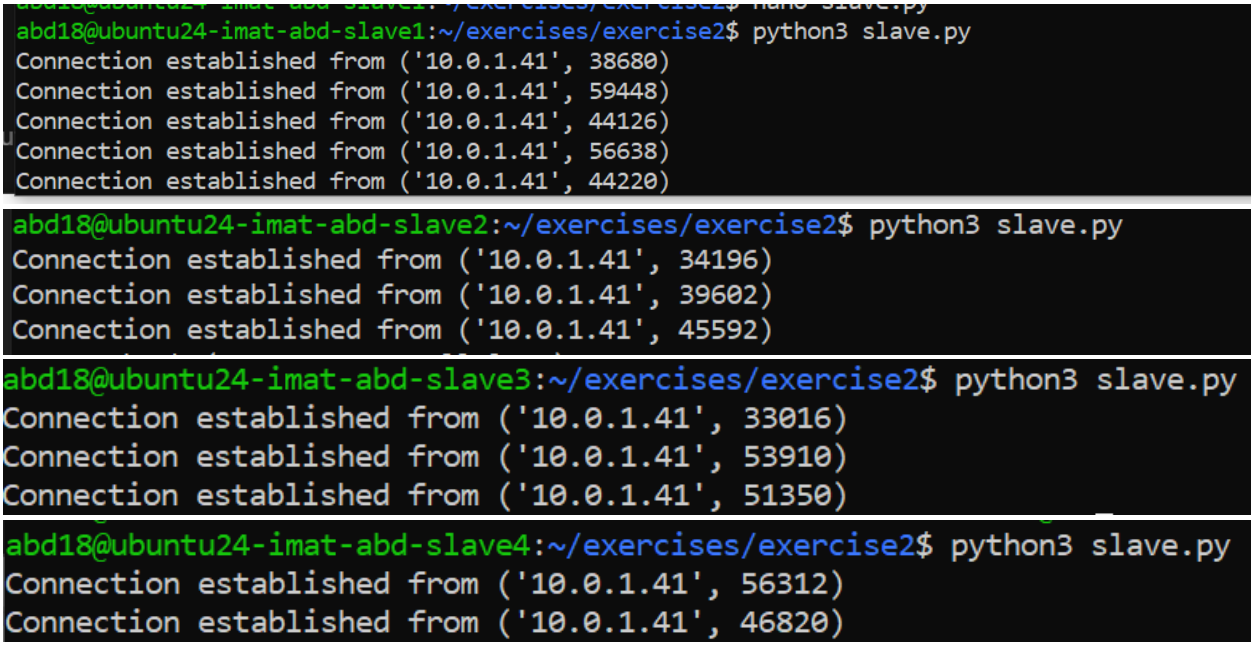
Resultados

Descripción de los resultados.

Tras ejecutar en los nodos, tenemos tres resultados.
Con 2 nodos tenemos pi = 3.13962, con un tiempo de ejecución de 0.20964 segundos.
Con 3 nodos tenemos pi = 3.13984, con un tiempo de ejecución de 0.14112 segundos.
Con 4 nodos tenemos pi = 3.14086, con un tiempo de ejecución de 0.10654 segundos.
Podemos observar que a medida que tenemos más nodos, el tiempo se reduce.
Se puede observar que la precisión es bastante similar.
Esto es debido a la aleatoriedad intrínseca del método

Pantallazos de la ejecución.

Ejecución de cada slave



Y finalmente el master:

```
abd18@ubuntu24-imat-abd1-master:~/exercises/exercise2$ python3 master.py
Enter the number of slaves: 2
Enter the number of points: 1000000
Connected to 10.0.1.42, and sent 500000 points.
Connected to 10.0.1.43, and sent 500000 points.
Estimated value of pi: 3.139624
Time elapsed with 2 slaves: 0.2096409797668457 seconds
abd18@ubuntu24-imat-abd1-master:~/exercises/exercise2$ python3 master.py
Enter the number of slaves: 3
Enter the number of points: 1000000
Connected to 10.0.1.42, and sent 333333 points.
Connected to 10.0.1.43, and sent 333333 points.
Connected to 10.0.1.44, and sent 333333 points.
Estimated value of pi: 3.13984
Time elapsed with 3 slaves: 0.14112019538879395 seconds
abd18@ubuntu24-imat-abd1-master:~/exercises/exercise2$ python3 master.py
Enter the number of slaves: 4
Enter the number of points: 1000000
Connected to 10.0.1.42, and sent 250000 points.
Connected to 10.0.1.44, and sent 250000 points.
Connected to 10.0.1.43, and sent 250000 points.
Connected to 10.0.1.45, and sent 250000 points.
Estimated value of pi: 3.140856
Time elapsed with 4 slaves: 0.10653829574584961 seconds
```

Conclusión

Tras la practica he comprendido la dificultad de utilizar los sistemas distribuidos. El código más básico se convierte en algo complejo y la necesidad de librerías adicionales le añade una capa extra de abstracción.

Sin embargo, la utilidad de usar un sistema como este es bastante visible. En esta práctica vemos la mejora en tiempo de ejecución a pesar de la creación de los sockets, mandarles la información y crear los hilos.
Esto es altamente relevante para programación que se pueda dividir en distintas tareas.