National Taipei University of Technology OS - HW1

113368545 陳昀鴻 職電子碩一

Source Code

1. First Program – A (For-Loops)

```python
import time
import numpy as np

def initialize_matrix_A(rows, cols):
    """初始化矩陣 A，元素公式為 a[i,j] = 6.5*i - 1.8*j"""
    matrix = np.zeros((rows, cols), dtype=complex)
    for i in range(rows):
        for j in range(cols):
            matrix[i, j] = complex(6.5 * i, -1.8 * j)
    return matrix

def initialize_matrix_B(rows, cols):
    """初始化矩陣 B，元素公式為 b[i,j] = (30 + 5.5*i) - 12.1j"""
    matrix = np.zeros((rows, cols), dtype=complex)
    for i in range(rows):
        for j in range(cols):
            matrix[i, j] = complex(30 + 5.5 * i, -12.1)
    return matrix

def matrix_multiply(A, B):
    """使用 for loop 計算矩陣乘法"""
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    if cols_A != rows_B:
        raise ValueError("矩陣維度不符合乘法要求")

    # 初始化結果矩陣
    C = np.zeros((rows_A, cols_B), dtype=complex)

    # 矩陣乘法
    for i in range(rows_A):
        for j in range(cols_B):
            sum_val = 0
            for k in range(cols_A):
                sum_val += A[i, k] * B[k, j]
            C[i, j] = sum_val

    return C

def main():
    # 矩陣維度
    rows_A, cols_A = 50, 80
    rows_B, cols_B = 80, 50

    # 初始化矩陣 A 和 B
    A = initialize_matrix_A(rows_A, cols_A)
    B = initialize_matrix_B(rows_B, cols_B)

    # 計時開始
    start_time = time.time()

    # 矩陣乘法
    C = matrix_multiply(A, B)

    # 計時結束
    end_time = time.time()

    # 輸出執行時間
    execution_time = (end_time - start_time) * 1000  # 毫秒
    print(f"執行時間 (for loop): {execution_time:.2f} ms")

    # 驗證結果矩陣 C
    print("結果矩陣 C 的部分內容:")
    print(C[:5, :5])  # 印出部分內容確認

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

2. Multithread(50 threads)

```python
import time
import numpy as np
from threading import Thread
import threading

def initialize_matrix_A(rows, cols):
    """初始化矩陣 A，元素公式為 a[i,j] = 6.5*i - 1.8*j"""
    matrix = np.zeros((rows, cols), dtype=complex)
    for i in range(rows):
        for j in range(cols):
            matrix[i, j] = complex(6.5 * i, -1.8 * j)
    return matrix

def initialize_matrix_B(rows, cols):
    """初始化矩陣 B，元素公式為 b[i,j] = (30 + 5.5*i) - 12.1j"""
    matrix = np.zeros((rows, cols), dtype=complex)
    for i in range(rows):
        for j in range(cols):
            matrix[i, j] = complex(30 + 5.5 * i, -12.1)
    return matrix

def multiply_row_range(start_row, end_row, A, B, C):
    """計算指定行範圍的矩陣乘法"""
    cols_B = B.shape[1]
    cols_A = A.shape[1]

    for i in range(start_row, end_row):
        for j in range(cols_B):
            sum_val = 0
            for k in range(cols_A):
                sum_val += A[i, k] * B[k, j]
            C[i, j] = sum_val

def matrix_multiply_multithreading(A, B, num_threads):
    """使用多執行緒進行矩陣乘法"""
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    if cols_A != rows_B:
        raise ValueError("矩陣維度不符合乘法要求")

    # 初始化結果矩陣
    C = np.zeros((rows_A, cols_B), dtype=complex)

    # 每個執行緒處理的行數
    rows_per_thread = rows_A // num_threads
    threads = []

    # 建立執行緒
    for i in range(num_threads):
        start_row = i * rows_per_thread
        end_row = start_row + rows_per_thread if i < num_threads - 1 else rows_A

        thread = Thread(target=multiply_row_range, args=(start_row, end_row, A, B, C))
        threads.append(thread)
        thread.start()

    # 等待所有執行緒完成
    for thread in threads:
        thread.join()

    return C

def main():
    # 矩陣維度
    rows_A, cols_A = 50, 80
    rows_B, cols_B = 80, 50

    # 初始化矩陣 A 和 B
    A = initialize_matrix_A(rows_A, cols_A)
    B = initialize_matrix_B(rows_B, cols_B)

    # 設定執行緒數量 (如 10 或 50)
    NUM_THREADS = 50  # 可以修改為 50 測試不同情況

    # 計時開始
    start_time = time.time()

    # 執行多執行緒矩陣乘法
    C = matrix_multiply_multithreading(A, B, NUM_THREADS)

    # 計時結束
    end_time = time.time()

    # 輸出執行時間
    execution_time = (end_time - start_time) * 1000  # 毫秒
    print(f"執行時間 (multithreading, {NUM_THREADS} threads): {execution_time:.2f} ms")

    # 驗證結果矩陣 C
    print("結果矩陣 C 的部分內容:")
    print(C[:5, :5])  # 印出部分內容確認

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

3. Multithread(10 threads)

```python
import time
import numpy as np
from threading import Thread
import threading

def initialize_matrix_A(rows, cols):
    """初始化矩陣 A，元素公式為 a[i,j] = 6.5*i - 1.8*j"""
    matrix = np.zeros((rows, cols), dtype=complex)
    for i in range(rows):
        for j in range(cols):
            matrix[i, j] = complex(6.5 * i, -1.8 * j)
    return matrix

def initialize_matrix_B(rows, cols):
    """初始化矩陣 B，元素公式為 b[i,j] = (30 + 5.5*i) - 12.1j"""
    matrix = np.zeros((rows, cols), dtype=complex)
    for i in range(rows):
        for j in range(cols):
            matrix[i, j] = complex(30 + 5.5 * i, -12.1)
    return matrix

def multiply_row_range(start_row, end_row, A, B, C):
    """計算指定行範圍的矩陣乘法"""
    cols_B = B.shape[1]
    cols_A = A.shape[1]

    for i in range(start_row, end_row):
        for j in range(cols_B):
            sum_val = 0
            for k in range(cols_A):
                sum_val += A[i, k] * B[k, j]
            C[i, j] = sum_val

def matrix_multiply_multithreading(A, B, num_threads):
    """使用多執行緒進行矩陣乘法"""
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    if cols_A != rows_B:
        raise ValueError("矩陣維度不符合乘法要求")

    # 初始化結果矩陣
    C = np.zeros((rows_A, cols_B), dtype=complex)

    # 每個執行緒處理的行數
    rows_per_thread = rows_A // num_threads
    threads = []

    # 建立執行緒
    for i in range(num_threads):
        start_row = i * rows_per_thread
        end_row = start_row + rows_per_thread if i < num_threads - 1 else rows_A

        thread = Thread(target=multiply_row_range, args=(start_row, end_row, A, B, C))
        threads.append(thread)
        thread.start()

    # 等待所有執行緒完成
    for thread in threads:
        thread.join()

    return C

def main():
    # 矩陣維度
    rows_A, cols_A = 50, 80
    rows_B, cols_B = 80, 50

    # 初始化矩陣 A 和 B
    A = initialize_matrix_A(rows_A, cols_A)
    B = initialize_matrix_B(rows_B, cols_B)

    # 設定執行緒數量 (如 10 或 50)
    NUM_THREADS = 10  # 可以修改為 50 測試不同情況

    # 計時開始
    start_time = time.time()

    # 執行多執行緒矩陣乘法
    C = matrix_multiply_multithreading(A, B, NUM_THREADS)

    # 計時結束
    end_time = time.time()

    # 輸出執行時間
    execution_time = (end_time - start_time) * 1000  # 毫秒
    print(f"執行時間 (multithreading, {NUM_THREADS} threads): {execution_time:.2f} ms")

    # 驗證結果矩陣 C
    print("結果矩陣 C 的部分內容:")
    print(C[:5, :5])  # 印出部分內容確認

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

Q1: Point out the *major parts* coded in the *threaded* program to highlight its differences with *for-loops*.

A1: 使用多執行緒的方式做矩陣乘法，並且是利用 **threading** 這個 **Library** 來實現。

```python
def multiply_row_range(start_row, end_row, A, B, C):
    """計算指定行範圍的矩陣乘法"""
    cols_B = B.shape[1]
    cols_A = A.shape[1]

    for i in range(start_row, end_row):
        for j in range(cols_B):
            sum_val = 0
            for k in range(cols_A):
                sum_val += A[i, k] * B[k, j]
            C[i, j] = sum_val

def matrix_multiply_multithreading(A, B, num_threads):
    """使用多執行緒進行矩陣乘法"""
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    if cols_A != rows_B:
        raise ValueError("矩陣維度不符合乘法要求")

    # 初始化結果矩陣
    C = np.zeros((rows_A, cols_B), dtype=complex)

    # 每個執行緒處理的行數
    rows_per_thread = rows_A // num_threads
    threads = []

    # 建立執行緒
    for i in range(num_threads):
        start_row = i * rows_per_thread
        end_row = start_row + rows_per_thread if i < num_threads - 1 else rows_A

        thread = Thread(target=multiply_row_range, args=(start_row, end_row, A, B, C))
        threads.append(thread)
        thread.start()

    # 等待所有執行緒完成
    for thread in threads:
        thread.join()

    return C
```

其中，將其包裝成 function，parameter 帶入 A、B 矩陣，以及 num_threads 數量。

Q2: Record your experimental results at least 3 rounds execution in the below table, and state how you can count the running time of programs in ms.

A2: （比較表呈現）

Coding Skill	No. of threads	Execution Time (ms)			Average Execution Time
For-Loops 【A】	1	26.70ms	29.08ms	28.41ms	28.06ms
Multithread 【B1】	50	31.79ms	31.78ms	31.33ms	31.63ms
Multithread 【B2】	10	30.08ms	30.23ms	30.22ms	30.17ms
Differences 【B1-A】	49	5.09ms	2.7ms	2.92ms	3.57ms
Differences 【B2-A】	9	3.38ms	1.15ms	1.81ms	2.11ms

用 Python 的 time 來計算運算時間，計算具體步驟從在執行矩陣乘法之前就開始計時，最終計算完成之後計算結束時間。

Step : 矩陣乘法前 -> （執行矩陣乘法） -> 結束矩陣乘法運算

Q3: State your discovering and comments on this exercise of coding threaded programs.

A3: 由 A2 比較表可發現差異性並不大。

這個問題有幾個主要原因：

1. 矩陣太小

目前的矩陣大小是 50x80 和 80x50，對於這麼小的矩陣，多處理程序的開銷反而大於計算時間，處理程序的創建和管理需要額外的時間成本。

2. 不應該使用 Multithread 而應該使用 Multiprocess

因為矩陣運算應該是 CPU 處理而不是 I/O 處理。

對此如果我們將矩陣放大，來驗證看看是否有把計算時間下降，如以下比較表：

以 **500x500** 的矩陣我們可以獲得以下的比較表：**(Multithread)**

Coding Skill	No. of threads	Execution Time (ms)			Average Execution Time
For-Loops 【A】	**1**	**17616.47ms**	**17444.21ms**	**17559.33ms**	**17540.00ms**
Multithread 【B1】	**50**	**16975.51ms**	**17434.56ms**	**17433.14ms**	**17281.07ms**
Multithread 【B2】	**10**	**18014.88ms**	**17774.77ms**	**17553.43ms**	**17781.02ms**
Differences 【B1-A】	**49**	**640.96ms**	**9.65ms**	**126.19ms**	**258.93ms**
Differences 【B2-A】	**9**	**398.41ms**	**330.56ms**	**5.9ms**	**244.95ms**

我們可以發現，具體其實並沒有非常顯著的改善執行時間。透過上述比較圖，我們可以發現，不一定使用 Multithread 來優化會是最佳策略。

其中 Python 當中的 Multithread 應該應用在 I/O 密集處理的操作（如：檔案操作、網路請求），所以用 Multiprocessing 比較合理，因為：CPU 密集處理（如：矩陣運算、大型數據處理），最後可以透過 share memory \ pipeline \ IPC 等方式來通信。

- **GIL 是什麼？**

GIL 是 Python 的全域直譯器鎖（Global Interpreter Lock），它是 CPython（Python 的標準實現）中的一個互斥鎖，用於限制同一時間只能有一個執行緒執行 Python 位元組碼。

重點程式碼如以下：

```python
def multiply_row_range(start_row, end_row, A, B, C_shared, rows_C, cols_C):
    """計算指定行範圍的矩陣乘法，並將結果寫入共享記憶體"""
    cols_B = B.shape[1]
    cols_A = A.shape[1]
    C = np.frombuffer(C_shared.get_obj(), dtype=complex).reshape((rows_C, cols_C))

    for i in range(start_row, end_row):
        for j in range(cols_B):
            sum_val = 0
            for k in range(cols_A):
                sum_val += A[i, k] * B[k, j]
            C[i, j] = sum_val
```

```python
def matrix_multiply_multiprocessing(A, B, num_processes):
    """使用多處理程序進行矩陣乘法"""
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    if cols_A != rows_B:
        raise ValueError("矩陣維度不符合乘法要求")

    # 初始化共享記憶體的結果矩陣
    C_shared = Array('d', rows_A * cols_B * 2)  # 每個複數需要 2 個 double（實部和虛部）

    # 每個處理程序處理的行數
    rows_per_process = rows_A // num_processes
    processes = []

    # 建立處理程序
    for i in range(num_processes):
        start_row = i * rows_per_process
        end_row = start_row + rows_per_process if i < num_processes - 1 else rows_A

        process = Process(target=multiply_row_range, args=(start_row, end_row, A, B, C_shared, rows_A, cols_B))
        processes.append(process)
        process.start()

    # 等待所有處理程序完成
    for process in processes:
        process.join()

    # 將共享記憶體結果轉換為 NumPy 矩陣
    C = np.frombuffer(C_shared.get_obj(), dtype=complex).reshape((rows_A, cols_B))
    return C
```

多執行緒當中我做了一個 C_shared memory 來做一個共享記憶體空間，並且分配每個 process 應該執行的行數，然後最終將結果回傳回來的時候轉換成 numpy 矩陣。

而矩陣計算的地方我寫成一個 function 作為包裝，parameter 有三：A 矩陣、B 矩陣、process 的個數（threads -> process）。

以 **500x500** 的矩陣我們可以獲得以下的比較表：（**Multiprocessing**）

Coding Skill	No. of threads	Execution Time (ms)			Average Execution Time
For-Loops 【A】	1	17415.47ms	17523.90ms	17359.17ms	17532.84ms
Multiprocessing 【B1】	50	4494.03ms	4096.50ms	4233.13ms	4274.55ms
Multiprocessing 【B2】	10	3859.10ms	3854.23ms	3807.99ms	3840.44ms
Differences 【B1-A】	49	12921.44ms	13427.4ms	13126.04ms	13154.96ms
Differences 【B2-A】	9	13556.37ms	13669.67ms	13551.18ms	13592.40ms

Running Screenshot

```
~/De/Y/NTUT_OS/Multithread on main ?3 python3 q1.py took 18s at 16:44:43
執行時間 (for loop): 17523.90 ms
結果矩陣 C 的部分內容：
[[-2717055. -4.17999825e+08j -2717055. -4.17999825e+08j
  -2717055. -4.17999825e+08j -2717055. -4.17999825e+08j
  -2717055. -4.17999825e+08j]
 [ 1840257.5-4.18039150e+08j  1840257.5-4.18039150e+08j
   1840257.5-4.18039150e+08j  1840257.5-4.18039150e+08j
   1840257.5-4.18039150e+08j]
 [ 6397570. -4.18078475e+08j  6397570. -4.18078475e+08j
   6397570. -4.18078475e+08j  6397570. -4.18078475e+08j
   6397570. -4.18078475e+08j]
 [10954882.5-4.18117800e+08j 10954882.5-4.18117800e+08j
  10954882.5-4.18117800e+08j 10954882.5-4.18117800e+08j
  10954882.5-4.18117800e+08j]
 [15512195. -4.18157125e+08j 15512195. -4.18157125e+08j
  15512195. -4.18157125e+08j 15512195. -4.18157125e+08j
  15512195. -4.18157125e+08j]]
```

```
~/De/Y/NTUT_OS/Multithread on main ?3 python3 q2.py at 17:00:58
執行時間 (multiprocessing, 50 processes): 4418.16 ms
結果矩陣 C 的部分內容：
[[-2717055. -4.17999825e+08j -2717055. -4.17999825e+08j
  -2717055. -4.17999825e+08j -2717055. -4.17999825e+08j
  -2717055. -4.17999825e+08j]
 [ 1840257.5-4.18039150e+08j  1840257.5-4.18039150e+08j
   1840257.5-4.18039150e+08j  1840257.5-4.18039150e+08j
   1840257.5-4.18039150e+08j]
 [ 6397570. -4.18078475e+08j  6397570. -4.18078475e+08j
   6397570. -4.18078475e+08j  6397570. -4.18078475e+08j
   6397570. -4.18078475e+08j]
 [10954882.5-4.18117800e+08j 10954882.5-4.18117800e+08j
  10954882.5-4.18117800e+08j 10954882.5-4.18117800e+08j
  10954882.5-4.18117800e+08j]
 [15512195. -4.18157125e+08j 15512195. -4.18157125e+08j
  15512195. -4.18157125e+08j 15512195. -4.18157125e+08j
  15512195. -4.18157125e+08j]]
```

環境

Hardware:

Hardware Overview:

Model Name: MacBook Pro

Model Identifier: Mac15,3

Model Number: Z1C80002RTA/A

Chip: Apple M3

Total Number of Cores: 8 (4 performance and 4 efficiency)

Memory: 16 GB

System Firmware Version: 10151.121.1

OS Loader Version: 10151.121.1

Serial Number (system): CJ492309HN

Hardware UUID: 3650AB42-B261-5140-93F0-B5A49288DA1E

Provisioning UDID: 00008122-000571C11A40001C

Activation Lock Status: Enabled

結論

在看到題目的同時，第一直覺會想用 Thread 去解決，但轉念一想他所做的事情應該是針對 CPU 處理的部分（矩陣運算）而並非 I/O 運算，而做完比較表之後，也實際採取將 Multithread 的架構改成用 Multiprocessing 的方式來做處理。

最終也確實證明在本案例當中，使用 Multiprocessing 會更為高效的表現。

Github Link : https://github.com/roger28200901/NTUT_OS （source treee & 報告原檔）