Operating System Project1

資訊三乙 10727211 林彥輝

一、開發環境:

作業系統: Windows 10 Enterprise x64

使用語言: Python 3.8

測試環境:

- CPU: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (12 CPUs),~2.6GHz
- RAM: Micron DDR4 2666/32GB*1
- SDD: 1TB PCIe SSD
- OS: Windows 10 Enterprise x64
- IDE: PyCharm Community Edition 2020.2.2

二、實作方法和流程

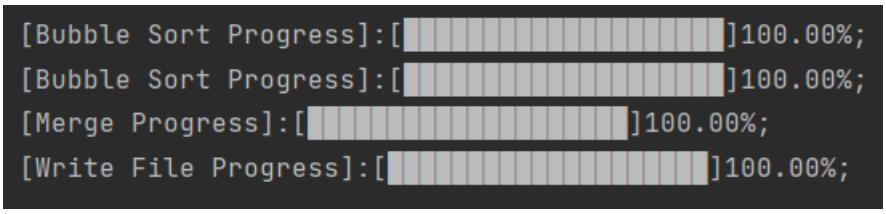
在 Task1 使用的 Bubble Sort 的演算法並非傳統教程之算法，而是在大二資料結構習得之算法，此算法相較於傳統算法之優點是在每一回合的 Sorting 過程會即時檢測是否排序完成，若過程中檢測資料排序完成就直接結束排序，減少不必要的檢查。

在 Task2 中，參考[1]設計出平行化 Thread 程式，先使用 list []將 K 份 Thread 放置其中，在使用 start()、join()，來執行、回收進行 Bubble Sort 的 K 份 Thread。隨後將 K 份 Bubble Sort 之結果兩兩一對 Merge()，(本 Project 是採用頭尾合併，每一次 Merge 都開出新的 Thread 執行)，直到 Merge 結果只剩一份。

在 Task3 中，參考[2]設計 multi-processor 程式，先使用 Pool 開出 K 份 processor 進行 Bubble Sort，隨後將 Bubble Sort 的資料分成一半，參考[3]使用 starmap()、zip()，將資料批次 Merge()，直至結果只剩一份。

Task4 的實作與 Task3 相當雷同，只是把 multi-processor 的數量調成 1，其餘內容皆與 Task3 相同。

因 Bubble Sort()、Merge Sort()、Merge()的等待時間可能因資料筆數因素導致執行時間過於冗長，故參考[4]實作了進度條(進度條的內容針對排序的 round 數而非估計時間)。

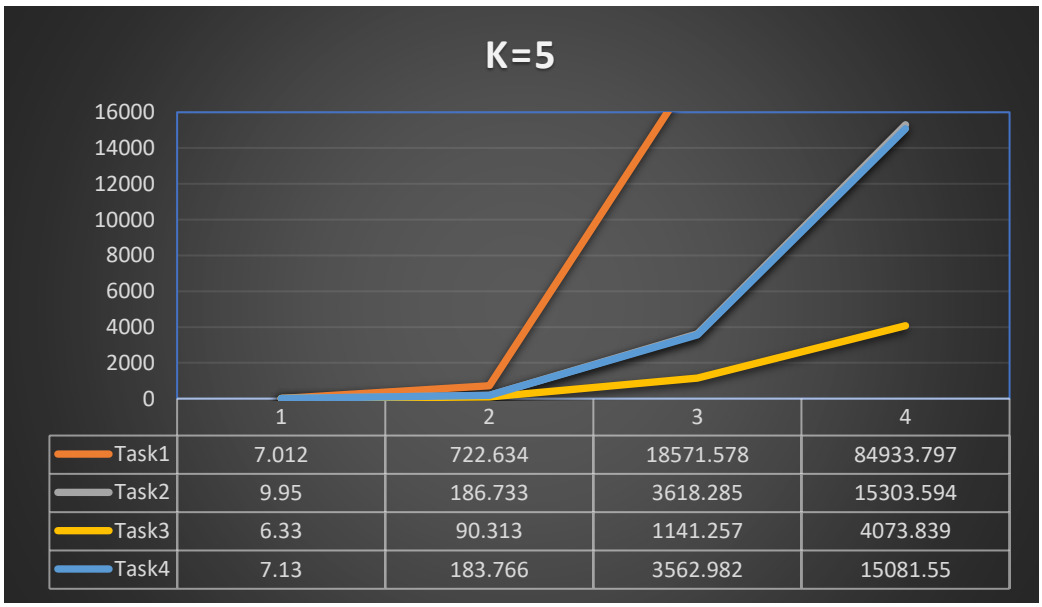


▲進度條功能

三、分析比較（time unit= second）

When K = 5 **(time unit = second)**

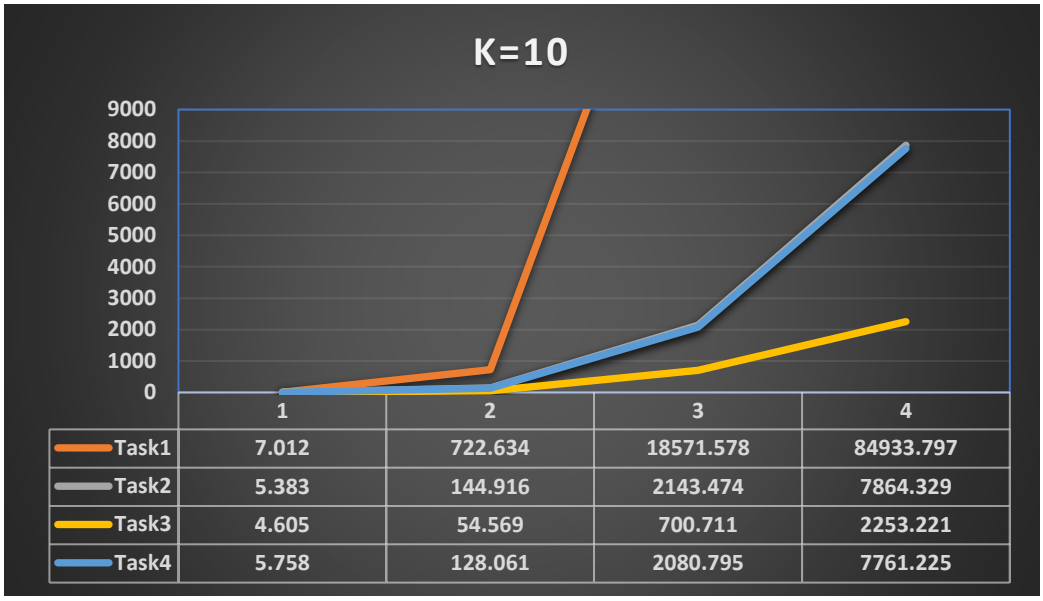
(行：花費時間、列：資料筆數(1:1 萬、2:10 萬、3:50 萬、4:100 萬))



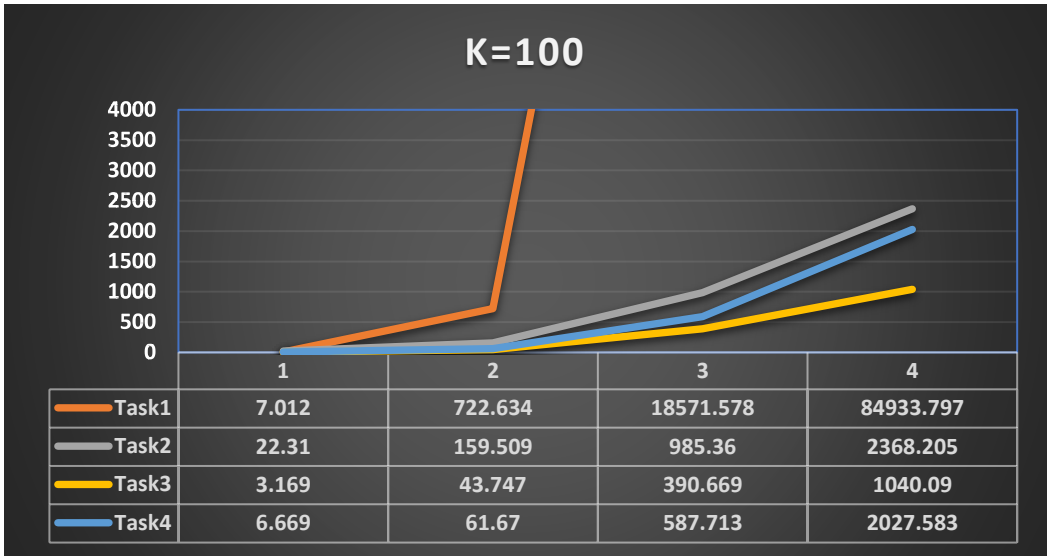
	1	2	3	4
Task1	7.012	722.634	18571.578	84933.797
Task2	9.95	186.733	3618.285	15303.594
Task3	6.33	90.313	1141.257	4073.839
Task4	7.13	183.766	3562.982	15081.55

When K = 10 、 K = 100 、 1000 **(time unit = second)**

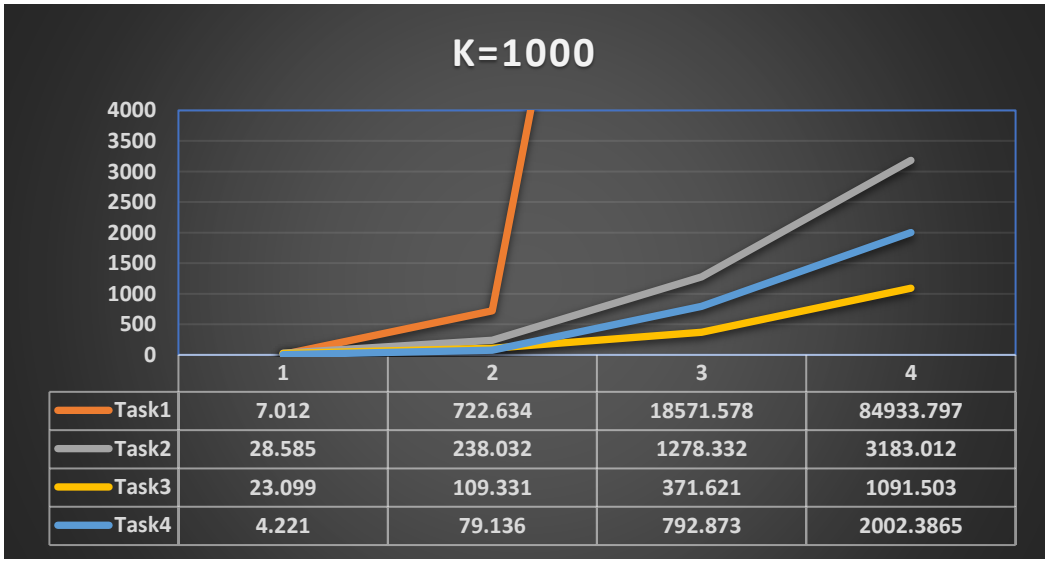
(行：花費時間、列：資料筆數(1:1 萬、2:10 萬、3:50 萬、4:100 萬))



	1	2	3	4
Task1	7.012	722.634	18571.578	84933.797
Task2	5.383	144.916	2143.474	7864.329
Task3	4.605	54.569	700.711	2253.221
Task4	5.758	128.061	2080.795	7761.225



	1	2	3	4
Task1	7.012	722.634	18571.578	84933.797
Task2	22.31	159.509	985.36	2368.205
Task3	3.169	43.747	390.669	1040.09
Task4	6.669	61.67	587.713	2027.583



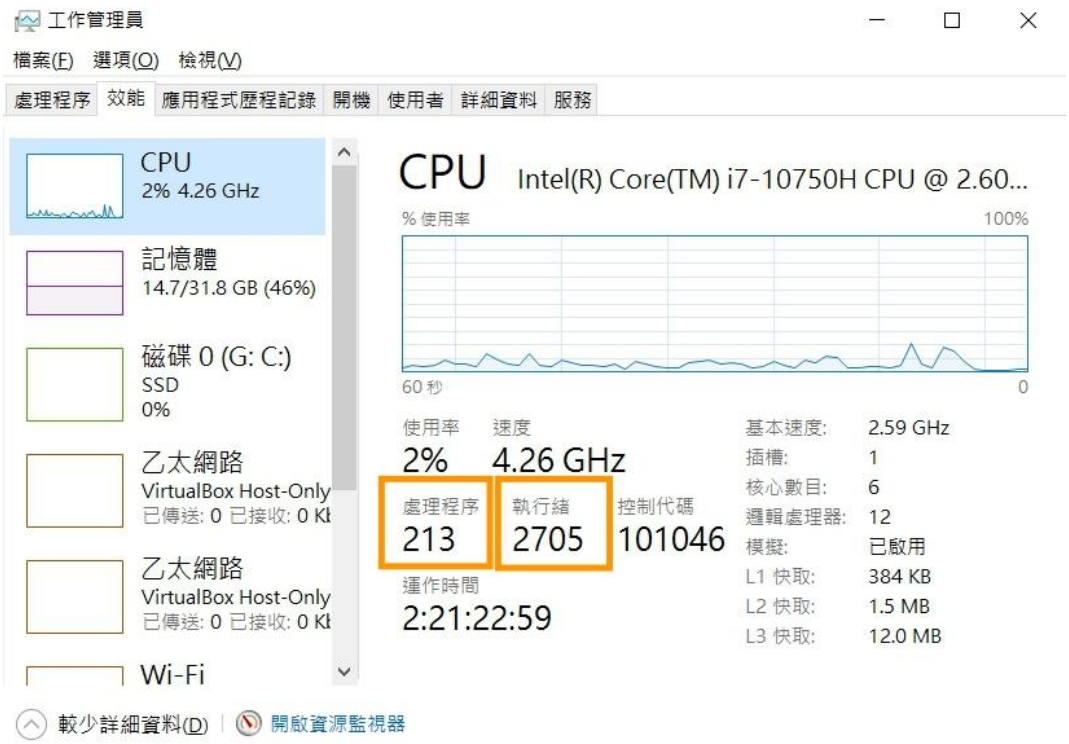
	1	2	3	4
Task1	7.012	722.634	18571.578	84933.797
Task2	28.585	238.032	1278.332	3183.012
Task3	23.099	109.331	371.621	1091.503
Task4	4.221	79.136	792.873	2002.3865

四、分析結果

在 Task1 之 Bubble Sort 中，執行時間隨資料筆數增加是無庸置疑的，結果也可以觀察到 100 萬筆的資料中需要花費到接近 1 天的時間處理。

接著換個角度觀察，從 K 的數量為度量衡進行比較，當 K 為 5 時，在資料筆數少量(1 萬)的情況下，開出 K 個 Thread 的效率並沒有比 Task4 來的優，原因可能是因為 Thread 是依附在 Process 上，K 個 Thread 共同平分一個 Process 的 Time Slice 資源，同樣的道理，K 個 Thread 效率也不會比 K 個 Process 優異。

若將 K 往上疊加，來到了 10、100、1000 的情形下，隨著 K 越大，資資料筆數越小效果反而會更差，Process 需要負擔開關 Thread 的狀況因為 K 的數量而變成了相當大的負擔，直到將資料筆數慢慢調增，才會真正顯現 K-Thread、K-Process 的威力，另外 K-Process 在實際運作時可以透過工作管理員感受 CPU 真正被消耗資源的感覺，是一次相當有趣的體驗。



▲開啟工作管理員查看 Thread、Processor 之增減狀態

五、參考資料

1. Python 多執行緒 threading 模組平行化程式設計教學

2. Multiprocessing – Process-based parallelism

3. How can I use pool.starmap and zip to combine and pass an entire list with a single element

4. Python 使用 tqdm 展示進度條，並嘗試自己寫進度條