資訊三甲 10827145 黃霈昕 作業系統 HW1 document

1. 開發環境
   CPU: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz
   HDD:931GB
   RAM:8GB
   OS: Windows 10 - 21H1
   系統類型:64 位元作業系統，x64 型處理器
   程式編輯器: Visual Studio Code
   程式語言:python(3.9)

2. 實作方法和流程
   BubbleSort : 重複地走訪過要排序的 list，一次比較兩個數，若順序較前者比順序較後者到就做交換，直到沒有再需要交換。

   Merge:使用 2-way Merge，依序將兩個排序好的 array 中相對小的數字放入(append)新的 array 中合併為一個 array。

   方法一：直接做 BubbleSort。

   方法二:將放讀入資料的 list 切成 k 個 array,開啟一個 process 對這 k 個 array 進行 BubbleSort，再用 2-way Merge 合併至剩一個 array 才結束 process。

   方法三：將放讀入資料的 list 切成 k 個 array，將這 k 個 array 分別用不同的 process 進行 BubbleSort,再分別用不同的 process 進行 2-way Merge 合併至剩一個 array。

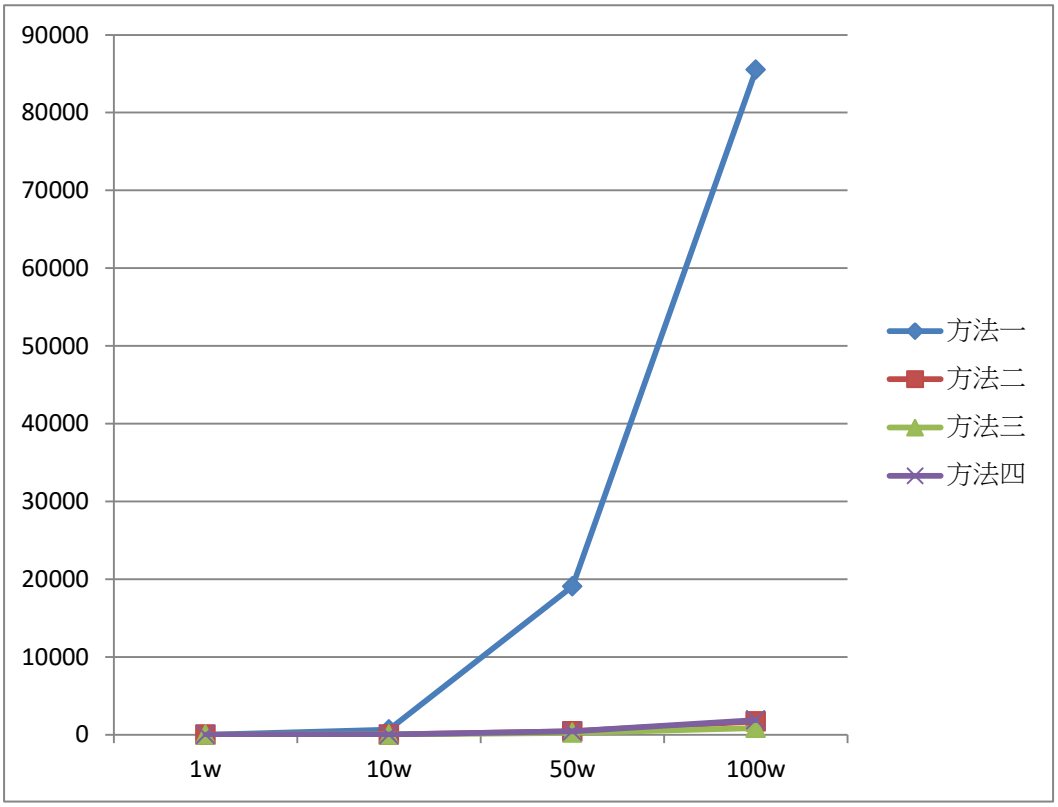
   方法四：將放讀入資料的 list 切成 k 個 array，將這 k 個 array 分別用不同的 thread 進行 BubbleSort，再分別用不同的 thread 進行 2-way Merge 合併至剩一個 array。

3. 特殊機制考量與設計
   多個 process 分別進行完 BubbleSort 或 Merge 後,其他 procees 會需要用到更新後的 array，所以需要一個容器來放這些更新後的 array，且這個容器需要能讓多個 process 都能使用，也就是需要所謂的共享資料，python 的 multiprocessing 模組中有提供程序間資料通訊模組 Manager 來解決這種情況，我使用了 Manager 提供的 Queue 資料結構來作為容器放更新後的 array 供需要用到新 array 的 process 能共享這個容器內的資料。

4. 分析結果和原因

K = 100

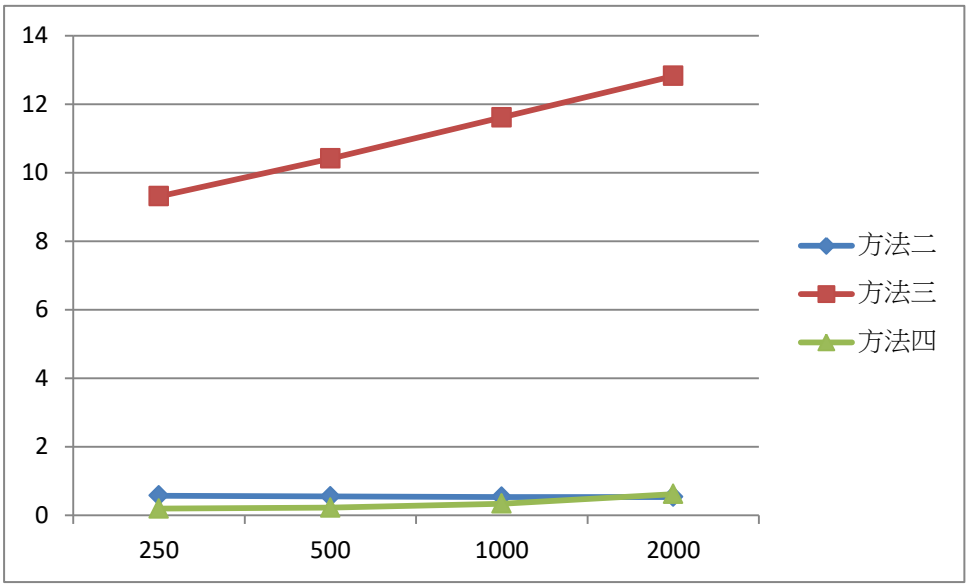
	1w	10w	50w	100w
方法一	5.885622	665.865542	19074.003006	85510.881809
方法二	0.709800	19.419975	427.529138	1698.569564
方法三	5.572300	19.625049	237.499454	871.968608
方法四	0.253466	15.957875	430.519188	1892.142035



由上表中可以看到，在資料筆數較少(一萬、十萬)時，方法四所花費的時間較少，而當資料筆數多(五十萬、一百萬)時，方法三所花費時間變明顯少於其他方法。multiprocessing 在資料傳遞上，會因為需要將資料轉移至其他 CPU 上進行運算，因此會需要考慮資料搬運的時間，所以在資料筆數少時反而花費較方法二、四多的時間，沒達到平行處理預期能達到的效率，但當資料筆數多時，多核心 CPU 實現平行運算的功能，明顯表現出平行處理所帶來的效率。方法四使用 multithreading，將 CPU 分配到的時間分給不同 thread 執行，當一個 thread 必須停下來等待需要佔據長時間處理的程序，其他 thread 還是可以繼續運作，在資料筆數少時可以提高處理效能，但當資料筆數多時，因為 OS 需要在它們之間做 Context Switch，使得程式執行速度反而比沒有使用 multithreading 的方法二更慢。

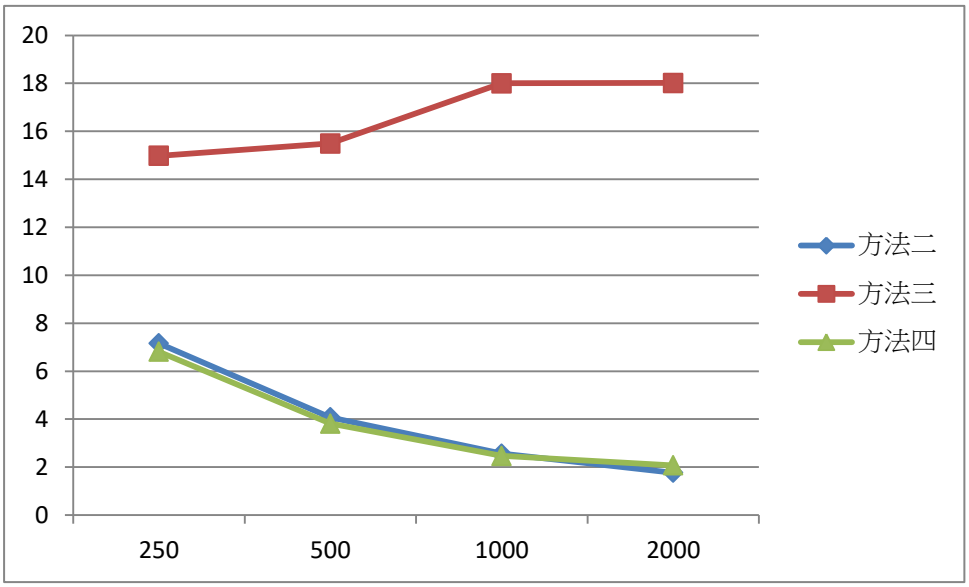
N = 1w

	250	500	1000	2000
方法二	0.573691	0.550735	0.538441	0.535169
方法三	9.308633	10.410352	11.613405	12.822633
方法四	0.197174	0.222251	0.342373	0.622491



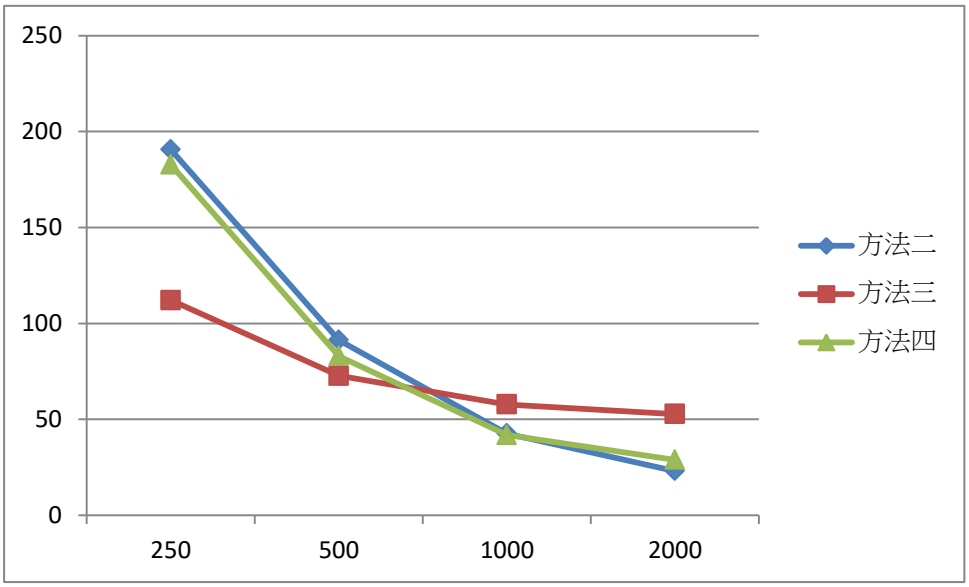
N = 10w

	250	500	1000	2000
方法二	7.157874	4.078286	2.559355	1.767261
方法三	14.980922	15.492561	17.999429	18.018164
方法四	6.809728	3.816800	2.470400	2.072284



N = 50w

	250	500	1000	2000
方法二	190.618310	91.285681	42.649786	23.071986
方法三	112.057641	72.727819	57.777140	52.805957
方法四	182.724814	83.002131	41.839509	28.912492



由上面三個表可看出，在資料筆數少(一萬、十萬)時，方法三切越多份(K 越大)所花費的時間反而更多，因為需要將資料轉移至其他 CPU 上進行運算，因此會需要花費較多時間在資料搬運上。方法二、四及資料筆數多(五十萬)時的方法三，切越多份(K 越大)所花費的時間越少，由第一個圖表可看出資料筆數越多 BubbleSort 所需要的時間也越多，因此切越多份，要進行一次 BubbleSort 的一組資料也越少，在 BubbleSort 上花費的時間也越少，所以提高了整體的效率。

5. 撰寫程式時遇到的 bug 及相關的解決辦法

(1) 在寫 multiprocessing 時使用 for 迴圈一次開始多個 procees 再用另一個 for 迴圈一次結束多個 process，在 K 值太大(500)時 VS Code 當掉(沒有錯誤訊息)

解決辦法:

猜測是因為一次生成太多 process 造成管道的壅擠而使 VS Code 當掉，改用 multiprocessing 提供的 Pool 模組，依 CPU 核數量讓系統自動分配資源，避免因一次生成太多 process 造成管道的壅擠而使 VS Code 當掉。

(2) 在寫 multiprocessing 時，使用中斷點一行一行執行不會出問題，但直接一次執行會使 VS Code 當掉(沒有錯誤訊息)

解決辦法:
因為一行一行執行不會出問題,因此猜測是由於平行處理在使用共享資料 Queue 中的資料時造成了 Race Condition，merge 要取共享資料 Queue 中的資料，而 Queue 中所放資料不是當次 merge 要取的資料，因此將取資料的程式碼放到該次 merge 的 process 都執行結束之後，避免因 Race Condition 造成 VS Code 當掉。