1. 開發/測試環境:

- Windows 10
- Anaconda python 3.8
- CPU: AMD Ryzen7 2700x 8 核 16 緒 / 32G Ram 1667MHz*2

2. 實作方法:

Task 1:

一般的 bubble sort，只需讀檔、排序、寫檔即可。

分組: (Task 2, 3, 4)

分組的部分選擇讓使用者不能輸入大於總資料筆數 1 半，因這樣分組上沒辦法做很好的分配，方案大概就是分成 n 組，前 n-1 組都只有 1 個元素，剩下的 1 組拿到剩下的所有元素，或許有其他更好的方案，但都會喪失分組省時的目的。若是等於資料筆數，相當於每組只有 1 個元素，一樣失去了分組的意義。

至於可接受的輸入，若能整除資料筆數，則每組就包含整除結果的元素，若不能整除，就各組能多放 1 個元素，剩下的一組再放剩餘的元素。

Task 2:

運用 python 內建函式庫 threading，建立 thread。

在分組完成後，一邊計算每組的起始 index 與末端 index，一邊建立

thread 讓每組做 bubble sort，之後用迴圈讓每個 thread 開始執行，再來用迴圈 join 每個 thread，使主 process 在各 thread 排序完前阻塞。

(※以下合併的操作 task2~4 是差不多的，差在使用 thread、process 或不使用。)

排好以後進入到 merge 的環節，利用兩個迴圈處理，外迴圈負責判斷是否合併到只剩一組，內迴圈負責做組之間兩兩合併的動作，主要細節在內迴圈，有三個工作要做:

1. 處理組別為奇數的狀況，最後一組利用 list 存起來
   (※以下記為 L[])，並移出目前的主 list。
2. 計算每次各組別中的起始位置與末端位置，因兩兩合併，每次的兩個位置都要重新計算
3. 組與組兩兩合併，並扣除總組數 1，因每次合併都會讓組別減半。

以上的第一步會這樣操作，是因為總組數若為奇數，最後一組的合併會很難處理，較簡單的方式是統一跟最後一組做合併，這樣即使最後一組元素個數特別多，也不會影響計算各組的起始、末端位置，但這樣的方式若以 thread 或 process 的方式處理，要等前面都 merge 完成後才能開新的 1 個 thread/process 去處理，加上除非總組數經常在迴圈中連續為奇數，不然另存最後一組元素的 list 很小，merge 起來不會影響太多運用

thread/process 帶來的效益，所以乾脆都不用 thread/process 在迴圈中間做處理了，以降低程式撰寫複雜度。

最後的一步就是用主 process 合併上述的 L[]，完成排序。

Task 3:

同 Task 2，只是中間的 thread 替換成由 python 內建函式庫中的 multiprocessing module 的 process 來完成。

Task 4:

同 Task 2，只是中間的 thread 拿掉，改為一般的函式呼叫來執行。

3. 各作法理論上的比較:

Task 1 是一般的 bubble sort，時間複雜度大約是 $O(n^2)$，理論上排序時間會非常的久，並會明顯地隨著原資料增長，排序時間只比指數成長好一點，應該會是最花時間的方式。

Task 2 運用了多 thread 做分組排序以及合併，由於 bubble sort 本身是一個隨著資料增長，運算時間急速上升的演算法，因此分組排序再合併這件事本身就能夠大幅縮減運算時間，加上 thread 可以平行運算，理論上會大幅降低運算時間，然而 python 有著全域直譯器鎖(GIL)，只允許

python 在同一時間只能有一個 thread 在執行，也就是多核心/多處理器，是没有太大的意義的，因為同時間只能執行一個 thread，並且 process 拿到的 time slice 是所有 thread 共享，在這些 thread 之間快速切換，來達到 concurrent 而不是 parallel，等同於單核心/單處理器，但還是有點不同，就是 GIL 並没有禁止 thread 在不同核心上執行，只是限制它們同一時間下只能有一個在 running，所以還是有機會因為不同核心/處理器晶片的體質，速度比單核心/單處理器快，否則理論上會跟單 process(Task 4)速度差不多。

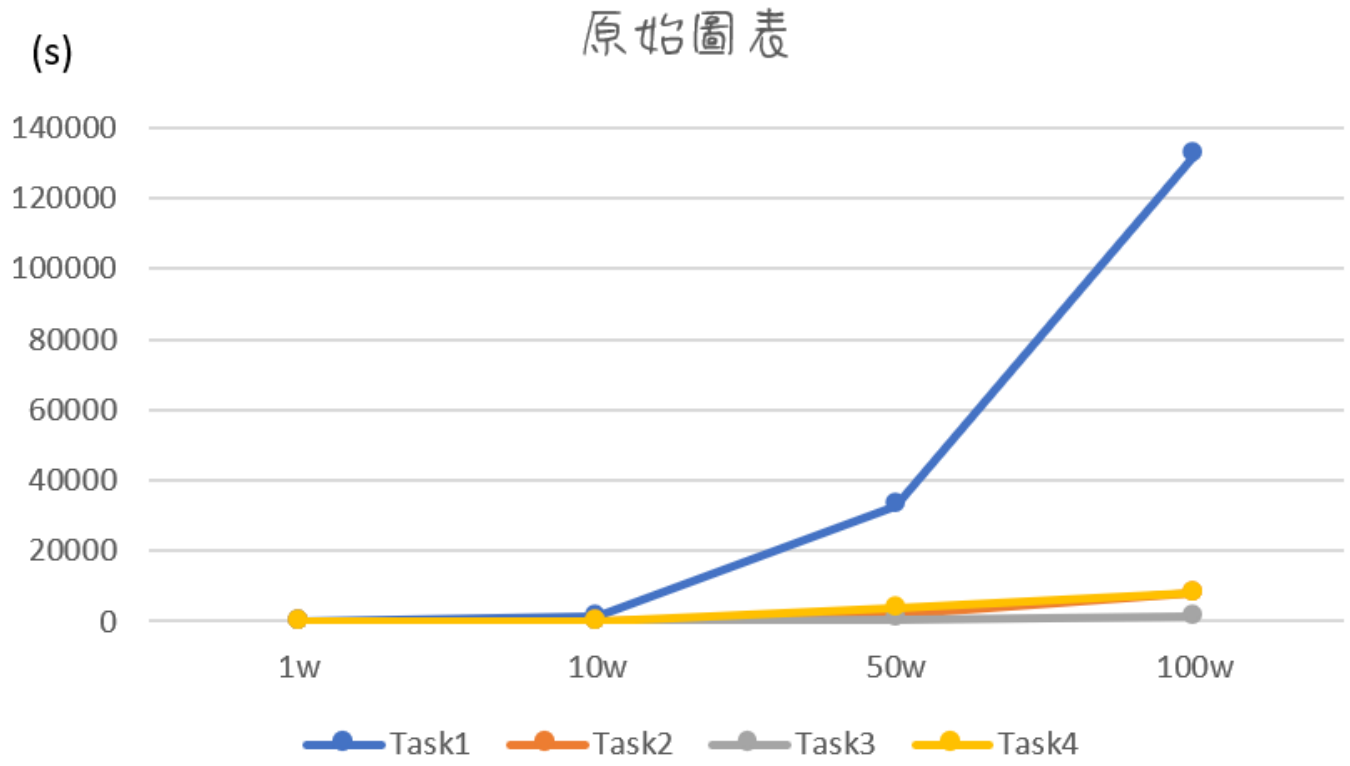
Task 3 運用了多 process 做分組排序及合併，在多核/多處理器下，是真的平行運算，只要分的組數 <= 電腦的邏輯處理器，就可以讓每個核心/處理器同時做排序/合併，並且每個 process 有自己的完整 time slice，不須像 thread 一樣，跟其他 thread 分享，綜上所述，理論上這個方式是最快能運算完成的。

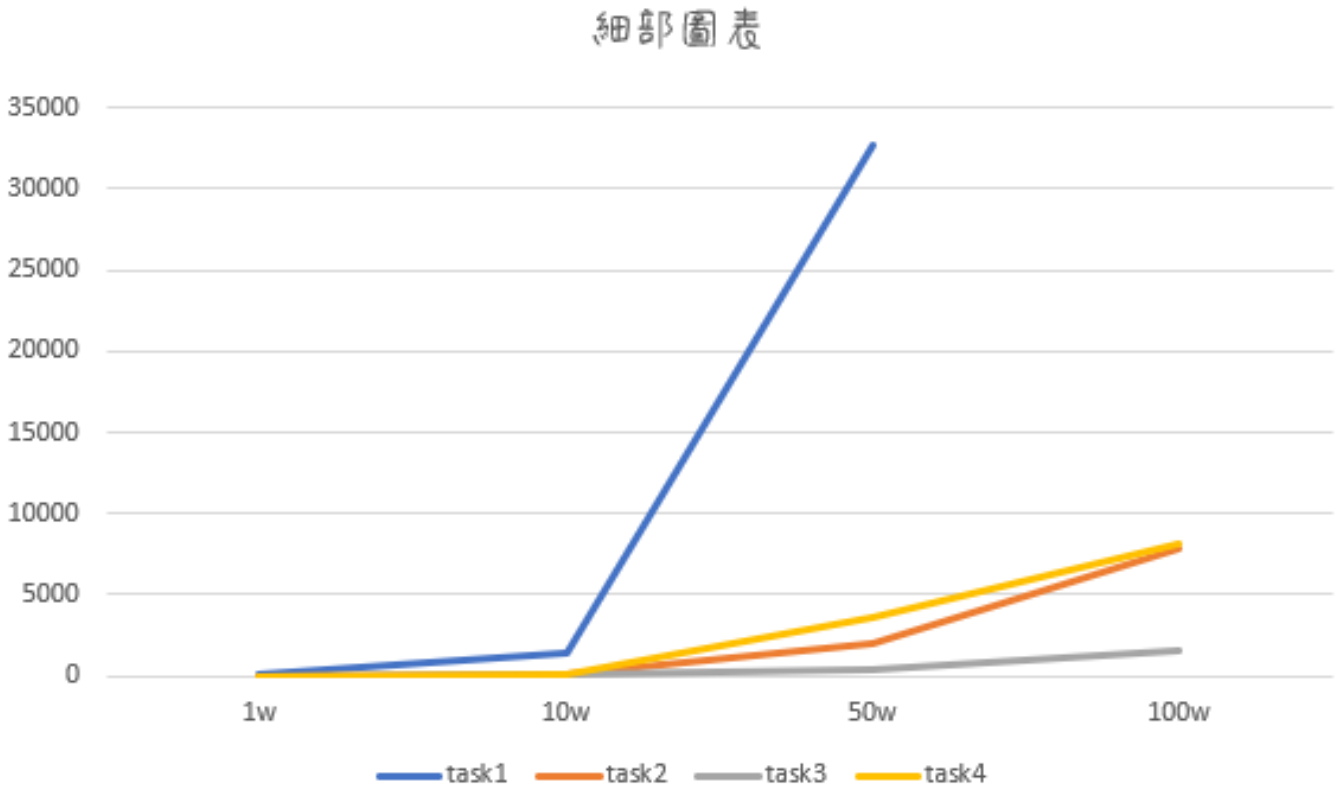
Task 4 只單純的做分組，然後各自 bubble sort，因此理論上速度只比不分組快，然而因為 python GIL，thread 没辦法發揮本來的效用，所以 task 4 的速度可能會跟 task 2 差不多。

綜上所述，理論上執行速度會是 3 > 2 > 4 > 1，
但因 python GIL 的限制，可能會是 3 > 2 >= 4 > 1。

4. 結果分析:

執行結果很明顯的表示 task 1 的執行時間隨著資料量增加，極大幅度的上升，task 3 的執行時間一直都是最快的，表示分工確實是同時進行，來達到加速的效果，至於 task 2 跟 task 4，task 2 的速度都快了那麼一點，我想是因為理論提到的 - 選到的多個核心體質比 task 單一核心好，速度飆得快了一點，因為 thread 被 GIL 鎖住，無法發揮原本的效能，用編譯語言比較容易解決這項問題。




細部圖表
35000
30000
25000
20000
15000
10000
5000
0
1w
10w
50w
100w
task1
task2
task3
task4