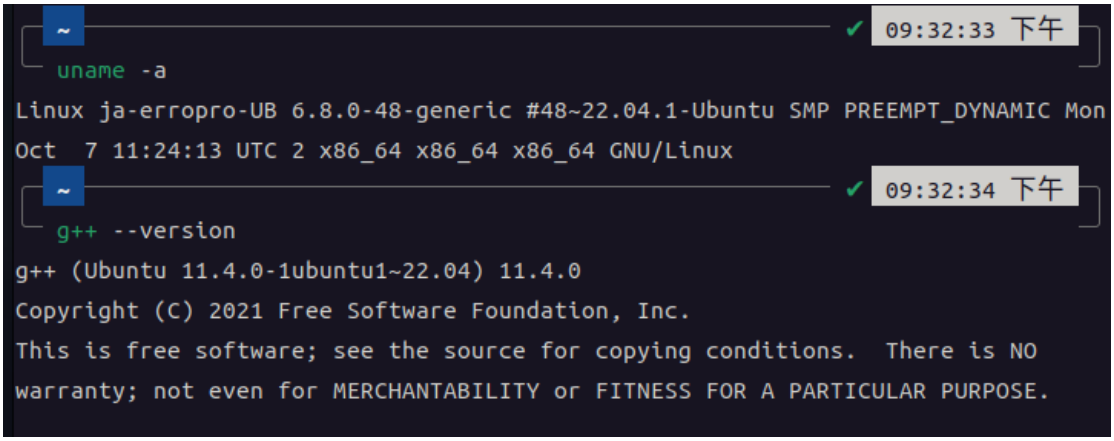
作業系統 作業一

資工三甲 11127137 黃乙家

一、開發環境



此系統直接安裝於物理硬碟，不使用任何虛擬化架構，程式語言使用 C++，執行緒方法為 POSIX Threads(pthreads)，編譯指令為 g++ -o <ELF> <Program.cpp> -lpthread

二、實作方法與流程

一開始程序進行讀檔，若成功則將讀到的資料一一存到陣列(以下的陣列皆使用 C++ Vector 實作)中，接著輸入 K、方法後依照方法分成不同流程：

(一)Pure Bubble Sorting

直接將整個陣列進行 Bubble Sorting，逐一比較相鄰兩元素，如果順序錯誤則交換，每輪會確保最後一個值位於正確的位置上，直到整個陣列都在正確的位置上。

若陣列長度為 N，需跑 N-1 次，每輪都會跑 n-i 次，總共有 n(n-1)/2 次，因此時間複雜度為 O(N^2)。

(二)Single Process, Single Thread Bubble Merge Sorting

先將資料切成 K 份，接著逐一依照(一)進行 Bubble Sorting，接著將 K 份已各自排好的陣列進行 K-way Merge Sorting，每輪選出 K 個陣列的第一個元素優先度最高的(由小到大排，數值越小優先度越高)，將其放入結果的陣列中，直到 N 個數都被選完。

若陣列長度為 N，將陣列切好後 Bubble Sorting 的時間複雜度為 O((N/K)^2)，總共有 K 份，因此為 O(N^2/K)，再進行 K-way Merge Sorting 後，最後複雜度為 O(NlgK)。

(三)Multiprocess Bubble Merge Sorting

先將資料切成 K 份，接著使用 pipe 作為行程間的通訊，利用 fork() 來啟動 process 分支來進行 Bubble Sorting，排序完後使用

write 將資料寫入 file descriptor 中，讓親代行程能夠接收到資料。

另外，父行程在 fork() 結束後，使用 read 來接收各行程排序完的資料，再進行 Merge Sorting，一樣使用 pipe 進行 IPC，fork 來新增 Process，每個 Process 將兩個陣列合併後傳回，直到只剩下一個陣列為止。

父行程在接收完所有資料後，需使用 wait 來確認所有 Process 皆已結束。

(四)Multithread Bubble Merge Sorting

與 Multiprocess 不同的地方在於因為 thread 已有共用記憶體，因此不需要 pipe，main thread 即可收到排序後的資料，只需新增完 thread 後使用 join 等待 thread 結束即可。

另外需要注意的是 C++ thread 無法接收回傳值，需使用 call by reference 或全域變數來獲得排序後的資料。

執行完以上流程後，將結果寫入檔案，完成排序。

實作的資料為 N=10000, 100000, 500000, 1000000，K=1, 10, 1000, 10000，並比較固定 N 值，不同 K 值各種方法的執行時間，以及固定 K 值，不同 N 值各種方法的執行時間。

而由於方法三中 C++ 提供的 <ctime> 只會紀錄一個 Process 的 CPU Time，而 <chrono> 並沒有提供測量 CPU Time 的函式，在親代行程等待時，clock 會暫停，最後的結果不準確，因此執行時使用 time ./<執行檔> < <輸入資料> 這道內建指令來記錄實際時間。

實驗方法每種組合皆執行三次，並記錄每次的時間，計算平均作為最終結果，對於 K = 1 的情況，因為時間複雜度相當於方法一的時間複雜度，且時間非常大，所以使用一樣的資料來分析。

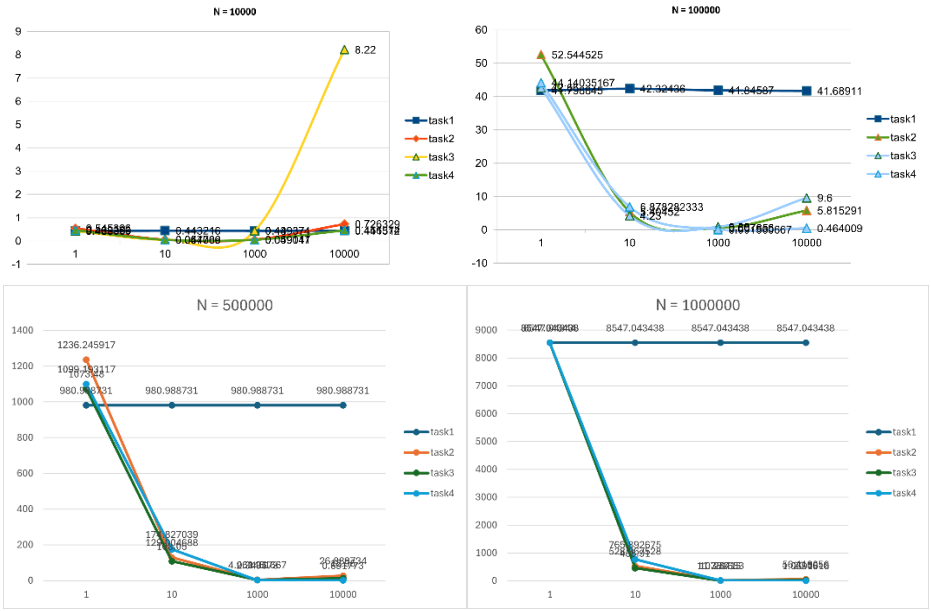
三、結果與原因

相同 N 值，不同 K 值：

K = 1, 10, 1000, 10000	10000	100000	500000	1000000
1	0.435, 0.443, 0.439, 0.441	41.798, 42.324, 41.846, 41.689	980.988, 980.988, 980.988, 980.988	8547.043, 8547.043, 8547.043, 8547.043

2	0.545, 0.054, 0.059, 0.726	52.544, 5.404, 0.607, 5.815	1236.245, 129.004, 4.061, 26.889	528.367, 11.381, 56.118
3	0.43, 0.06, 0.44, 8.22	42.66, 4.23, 0.89, 9.60	1073.48, 108.05, 3.01, 12.96	8547.043, 455.91, 7.57, 20.06
4	0.462, 0.047, 0.049, 0.456	44.140, 6.878, 0.091, 0.464	1099.193, 174.827, 2.349, 0.691	8547.043, 765.892, 10.244, 1.069

繪製成圖表後



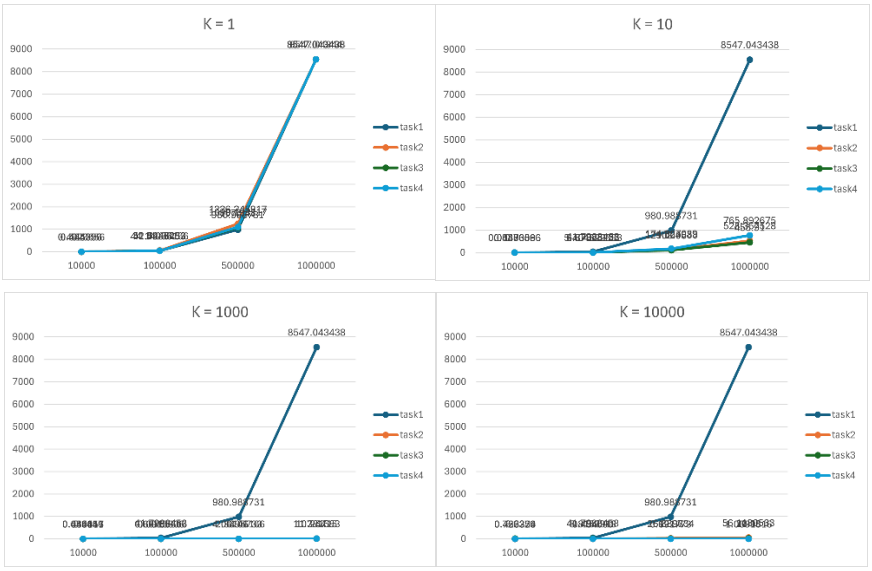
從圖表中可以觀察到，無論 K 值為多少，方法一沒有太多變化，可以推測 K 值大小不影響方法一的時間複雜度。另外可以發現當 K 很大時，方法三時間顯著提升，推測是因為 Context Switch 代價以及 Process 複製記憶體的代價相較其他方法還多。

相同 K 值，不同 N 值：

N = 1w, 10w, 50w, 100w	1	10	1000	10000
1	0.435, 41.798, 980.988, 8547.043	0.443, 42.324, 980.988, 8547.043	0.439, 41.845, 980.988, 8547.043	0.441, 41.689, 980.988, 8547.043
2	0.545, 52.544, 1236.245	0.0547667,	0.059, 0.607,	0.726,

		5.404, 129.004, 528.367	4.061, 11.381	5.815, 26.889, 56.118
3	0.43, 42.66, 1073.48, 8547.043	0.06, 4.23, 108.05, 455.91	0.44, 0.89, 3.01, 7.57	8.22, 9.60, 12.96, 20.06
4	0.462, 44.140, 1099.193, 8547.043	0.047009, 6.878, 174.827, 765.892	0.049, 0.091, 2.349, 10.244	0.456, 0.464, 0.691, 1.069

繪製成圖表後



可以發現當 K=1 時，每種方法的時間都差不多，而對於 K!=1 的情況，隨著 N 變大，方法一的上升趨勢非常明顯，當 K 很大時，無論方法 234 的差距都很小。