一、資料結構與方法設計：(開發環境：WSL2-Ubuntu-22.04LTS)

struct Core { // 檔案結構

string filename; // 檔案名稱

int k; // 切成幾份

int n; // 總共幾筆

double time; // 總耗時

int method; // 使用方法

vector<int> data; // 原資料/最終結果

vector<vector<int>> sublists; // 拆分後的子集/經過 bubble sort 後的子集

};

// 方法一

class Method1::core; // 全域便於繼承

class Method1::Method1{} // 讀入原始資料

class Method1::bubbleSort{} // 氣泡排序

class Method1::writeFile{} // 寫檔

class Method1::exec{} // 在 main 中執行

// 方法二繼承方法一，建構子也繼承方法一（所以 Core 才用全域）

class Method2::merge{} // 丟入要合併的兩個子集合併排序

class Method2::mergeSort{} // 丟入要合併排序的完整單一陣列，遞迴

class Method2::splitData{} // 拆分成 k 個子集

class Method2::exec{} // override 方法一的，fork 出一個 child process，並在其中依序執行氣泡排序與合併排序後，使用 Shared Memory Model 寫回 parent process

// 方法三繼承方法二，建構子也繼承方法二

class Method3::exec{} // override 方法二的，先依序產生 k 個 child process，依序將子集排序後寫進 shared memories，child process 不用互斥（個別子集不影響），但要與 parent process 互斥，所以在 parent process 中等待 k 回。K 個子集完成後依序寫入 parent

process 的 core.sublists 中，再清除 shared memories。再來，再產生 k 個 shared memories，將第一個子集先寫到第一個 shared memory。之後進行 k-1 個 child process(1~k)，每一個 child process 都先從前一個 shared memory 抓資料，先計算合併後的大小，建立目前的 shared memory，合併排序後寫入目前的 shared memory。不能平行處理，因為這個寫法必須等待上一個 child process 寫入資料後才能進行自己的 process。最後，將最後一個 shared memory 的資料寫入 parent process 的 core.data，並且清除所有的 shared memories。

// 方法四繼承方法三，建構子也繼承方法三

class Methods4::exec{} // override 方法三的，先在目前的 process 啟動 k 個 threads 平行處理 bubble sort，全部完成後再啟動 k-1 個 threads 依序執行 merge sort（threads 之間互斥）。最後寫回目前的 process 中的 core.data。

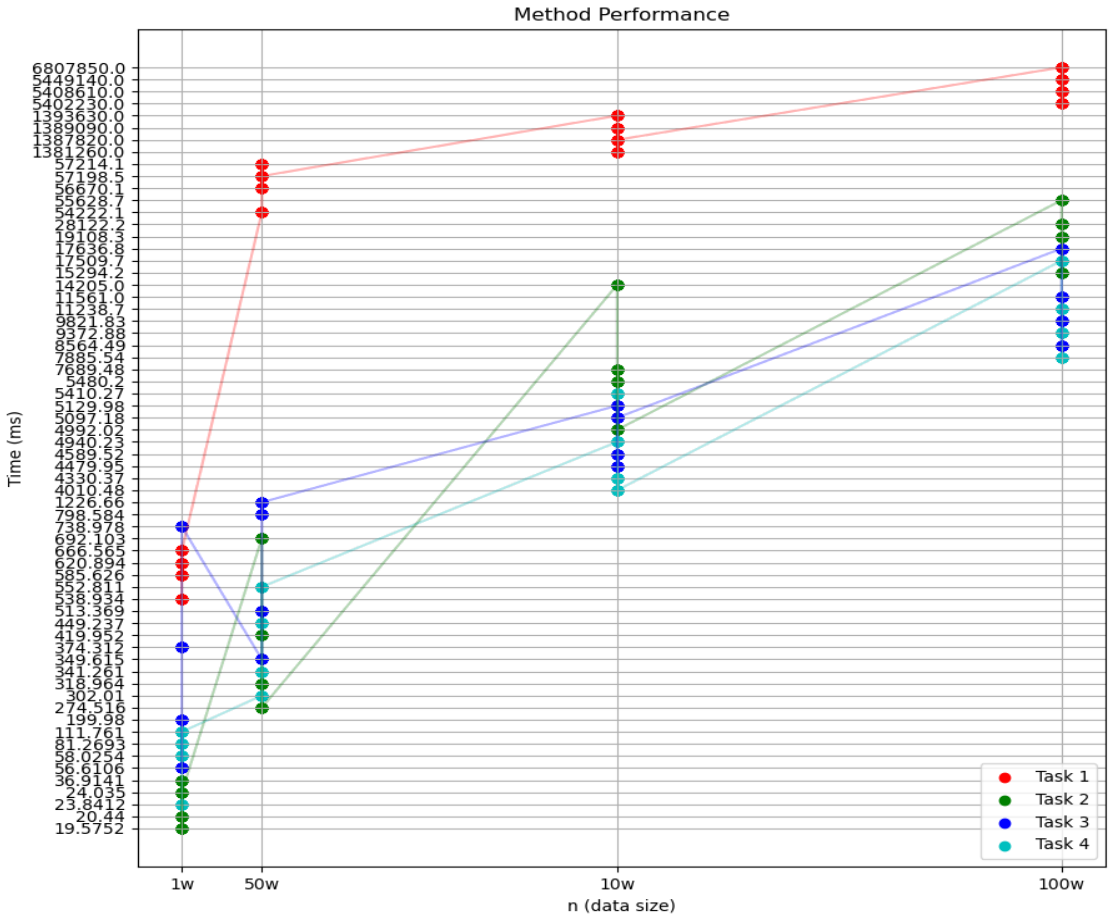
二、圖表分析：

K = {100, 200, 300, 400}	N = 1 萬	N = 10 萬	N = 50 萬	N = 100 萬
方法一	538.934, 585.626, 620.894, 666.565	54222.1, 56670.1, 57214.1, 57198.5	1393630, 1389090, 1381260, 1387820	6807850, 5408610, 5402230, 5449140
方法二	36.9141, 19.5752, 20.44, 24.035	692.103, 419.952, 318.964, 274.516	14205, 7689.48, 5480.2, 4992.02	55628.7, 28122.2, 19108.3, 15294.2
方法三	56.6106, 199.98, 374.312, 738.978	349.615, 513.369, 798.584, 1226.66	5129.98, 4479.95, 4589.52, 5097.18	17636.8, 11561, 9821.83, 8564.49
方法四	23.8412, 58.0254, 81.2693, 111.761	302.01, 341.261, 449.237, 552.811	4940.23, 5410.27, 4330.37, 4010.48	17509.7, 11238.7, 9372.88, 7885.54

表 1：實驗記錄表格（單位：ms）

N = {1 萬, 10 萬, 50 萬, 100 萬}	K = 100	K = 200	K = 300	K = 400
方法一	538.934, 54222.1, 1393630, 6807850	585.626, 56670.1, 1389090, 5408610	620.894, 57214.1, 1381260, 5402230	666.565, 57198.5, 1387820, 5449140
方法二	36.9141, 692.103, 14205, 55628.7	19.5752, 419.952, 7689.48, 28122.2	20.44, 318.964, 5480.2, 19108.3	24.035, 274.516, 4992.02, 15294.2
方法三	56.6106, 349.615, 5129.98, 17636.8	199.98, 513.369, 4479.95, 11561	374.312, 798.584, 4589.52, 9821.83	738.978, 1226.66, 5097.18, 8564.49
方法四	23.8412, 302.01, 4940.23, 17509.7	58.0254, 341.261, 5410.27, 11238.7	81.2693, 449.237, 4330.37, 9372.88	111.761, 552.811, 4010.48, 7885.54

表 2：實驗記錄表格（單位：ms）



三、檢討架構：

由圖表觀察到，耗時：方法一>>方法二>方法三>方法四。第一個方法是最耗時的，在 main process 中指做氣泡排序；方法二比他快但比方法三、四慢，因為他使用兩種排序法，但只有產生一個 process 並在裡面接連做排序，並且還要 Shared Memory；方法三更快比方法四慢，因為氣泡排序全部 k 個 process 可以平行處理，但合併排序我是採子集 i 與子集 i+1 合併後寫到下一個 shared memory, 下一個 process 再從上一個 shared memory 讀取資料後寫到下一個，主要弊端有二：其一，全部 child process 互斥，沒辦法平行執行；其二，進行了大量的 memory copy operation。假設有 8 個 processes，我的作法會是 array1 = (0+array1), array2 = (array1 + array2), …依序執行，但更好的做法是平行執行[subarray1 = array1+array2, subarray2 = array3+array4, subarray3 = array5+array6, subarray4 = array7+array8] →平行執行 [subarray5 = subarray1+subarray2, subarray6 = subarray3+subarray4] → 最後執行 final_result = subarray5+subarray6，達到同樣是 k-1 個 child processes 但能夠平行執行的效果。方法四最快的原因是，雖然它幾乎和方法三相同，但是多執行緒的比多進程的成本低了許多，因此雖然也有大量的 memory copy 等耗時的操作，但比起方法三還是會更快。

對所有方法而言，資料量愈大，操作愈多，愈發耗時，這是無庸置疑的。因此，相同 K 值之下 N 值愈大愈耗時。而 K 值對方一完全無關，因為完全沒有切資料；對方法二而言，K 值愈大愈快，因為方法二只有產生一個 child process，沒有成倍的記憶體操作，又新增了合併排序，因此耗時會愈短。對於方法三，N 值在 1 萬、50 萬時比方法二和方法四慢，表示記憶體操作與合併排序沒有平行執行所帶來的耗時非常高；但在 N 值到了 10 萬與 100 萬時，平行處理氣泡排序的優勢就被放大了，降低了耗時。最後對於方法四，一樣在 N 值為 1 萬、50 萬時耗時比方法二久，但比方法三快（上面解釋過的操作成本與多進程、多執行緒成本），到了 N 值為 10 萬、100 萬時，又變成了幾乎是最快的那個，理由同方法三。

我使用了 Unix-based 的 fork()建立 process，並且基於 Shared Memory Model 實作方法三。優點是不用像在 Windows 環境下使用極其麻煩的 createProcess 方法，更加直觀；缺點是要記得手動清除 shared memories，並且要自行計算它的大小，風險較大。在建立 thread 時，我使用 lambda 寫法呼叫 std::thread 的建構子。傳入一個 lambda 函數，它會接要建立的 process 式第”i”個，this 指向的 instance，並且在這之中執行操作。

原本使用 Python 撰寫，但效率不理想就用 C++重寫，結果還是用了暴力法。花最多的時間在：記憶體的操作，以及忘記互斥機制導致一直在對正確的記憶體操作 debug。下一次作業會寫得更好。