開發環境

本程式在 Linux 環境下開發，使用 C++ 語言編寫。

實作方法和流程

流程

1. 印出主選單，等待使用者輸入檔案名稱、切割數量、排序方式。
2. 嘗試以檔案名稱開啟檔案，如果開啟失敗或使用者輸入的參數錯誤，則回到步驟 1。
   如果成功開啟檔案，則呼叫 readFile() 將檔案資料讀入。
3. 根據使用者選擇的排序方式來執行 method_1()、method_2()、method_3()、method_4() 中的其中一個。

- method_1() :
    1. 取得當前時間，並將該時間存放在 start 變數中。
    2. 進行氣泡排序。
    3. 取得當前時間，並將該時間存放在 end 變數中。
    4. 計算 start 與 end 的時間差，並轉換為毫秒單位，並將時間差作為函數的回傳值返回。

- method_2() :
    1. 取得當前時間，並將該時間存放在 start 變數中。
    2. 將陣列拆分成 k 個區塊。
    3. 在 for 迴圈中，對每個區塊執行 bubble sort。
    4. 當區塊數量大於 1 時，執行以下的動作：
        - 若區塊數量是奇數，標記最後一個區塊為孤立區塊 (lonely block)。
        - 對相鄰的兩個區塊進行合併排序成為一個新的區塊，不合併孤立區塊。
    5. 取得當前時間，並將該時間存放在 end 變數中。
    6. 計算 start 與 end 的時間差，並轉換為毫秒單位，並將時間差作為函數的回傳值返回。

- method_3() :
    1. 取得當前時間，並將該時間存放在 start 變數中。
    2. 建立一個共享的虛擬空間。
    3. 將陣列拆分成 k 個區塊。
    4. 使用 fork() 創建 k 個 child process，對每個區塊執行 bubble sort，將結果寫回共享的虛擬空間，並在 child process 結束時退出。
       parent process 用 waitpid() 等待所有 child process 結束。
    5. 使用 fork() 創建新的 child process，對每個需要合併的區塊執行 merge sort，將結果寫回共享的虛擬空間，並在 child process 結束時退出。

parent process 用 waitpid() 等待所有 child process 結束。

6. 複製虛擬空間中的排序結果回到原本的 arr 中，並釋放共享內存。
7. 取得當前時間，並將該時間存放在 end 變數中。
8. 計算 start 與 end 的時間差，並轉換為毫秒單位，並將時間差作為函數的回傳值返回。

- method_4() :
    1. 取得當前時間，並將該時間存放在 start 變數中。
    2. 將陣列拆分成 k 個區塊。
    3. 在 for 迴圈中，用 pthread_create() 創建 thread，用 thread 對每個區塊執行 bubble sort。
    4. 使用 for 迴圈，等待所有 thread 執行完畢。
    5. 當區塊數量大於 1 時，執行以下的動作：
        - 若區塊數量是奇數，標記最後一個區塊為孤立區塊 (lonely block)。
        - 用 pthread_create() 創建 thread，用 thread 對相鄰的兩個區塊執行 merge sort 成為一個新的區塊，不合併孤立區塊。
    6. 取得當前時間，並將該時間存放在 end 變數中。
    7. 計算 start 與 end 的時間差，並轉換為毫秒單位，並將時間差作為函數的回傳值返回。

4. 將排序所需的時間輸出到螢幕和檔案中。
5. 回到步驟 1，等待下一次 User 輸入。

資料結構

程式主要使用了向量和動態分配的 int 陣列來存儲數據，並且使用了自定義的數據結構 piece 來分割數據。

merge 方法如何運作

這個 function 接受一個 int 類型的陣列 arr，以及三個整數 left、mid、right。這個 function 的目的是將 arr 的 [left, mid] 和 [mid + 1, right] 兩個子陣列進行 merge sort，並將排序結果放回 arr 中的對應區域。

在函式一開始，定義了三個整數變數 i、j、k，分別代表左子陣列的指針、右子陣列的指針和合併結果的指針。接著創建了一個 vector<int> temp，大小為 right - left + 1，用來暫存排序後的結果。

接下來使用 while 迴圈來比較左子陣列的第 i 個元素和右子陣列的第 j 個元素，將較小的元素存入 temp[k] 中，然後指針 i 或 j 往後移動一位，指針 k 往後移動一位，直到其中一個子陣列的指針超過了 mid 或 right 的位置。

若還有元素沒有排序，那麼進入第二個和第三個 while 循環，將剩下的元素複製到 temp 中。

最後用 for 迴圈將排序好的結果 temp 複製回原陣列 arr 中的對應區域。

建立 process 和 thread 的方法

程式中實現了多進程和多線程的排序，建立 process 的方法使用了 Linux 下的 fork() 函數，建立 thread 的方法則使用了 C++11 中的 pthread_create() 創建 thread。

process 之間如何共享資料

程式中使用了 Linux 下的共享內存的概念，讓多個 process 可以共享一塊內存，從而實現 process 之間的數據共享。

輔助函數

程式中實現了一些輔助函數，如 printMenu()、readFile()、writeFile() 和 printTime() 等。

其中 printMenu() 用於接收 User 的輸入，readFile() 用於讀取檔案中的資料，writeFile() 用於將排序後的數據寫入到檔案中，printTime() 則用於輸出程式執行時間和當下時間。

探討結果與原因

K = {10, 20, 30, 40}	N = 1 萬	N = 10 萬	N = 50 萬	N = 100 萬
方法一	{707, 680, 726, 662}	{52772, 52695, 56518, 56493}	{684544, 687511, 684084, 686140}	{2043789, 2041745, 2044810, 2047876}
方法二	{66, 34, 26, 16}	{7670, 3791, 2512, 1773}	{158758, 85313, 56688, 44774}	{533951, 310914, 215053, 165589}
方法三	{34, 20, 19, 18}	{1718, 890, 625, 424}	{54160, 29306, 19955, 13929}	{191976, 106408, 66176, 54786}
方法四	{25, 18, 15, 11}	{1851, 869, 582, 439}	{47872, 28281, 22086, 15868}	{188158, 106890, 73262, 57108}

表 1、實驗記錄表格（ ms ）

N = {1 萬, 10 萬, 50 萬, 100 萬}	K = 10	K = 20	K = 30	K = 40
方法一	{707, 52772, 684544, 2043789}	{680, 52695, 687511, 2041745}	{726, 56518, 684084, 2044810}	{662, 56493, 686140, 2047876}
方法二	{66, 7670, 158758, 533951}	{34, 3791, 85313, 310914}	{26, 2512, 56688, 215053}	{16, 1773, 44774, 165589}
方法三	{34, 1718, 54160, 191976}	{20, 890, 29306, 106408}	{19, 625, 19955, 66176}	{18, 424, 13929, 54786}
方法四	{25, 1851, 47872, 188158}	{18, 869, 28281, 106890}	{15, 582, 22086, 73262}	{11, 439, 15868, 57108}

表 2、實驗記錄表格（ ms ）

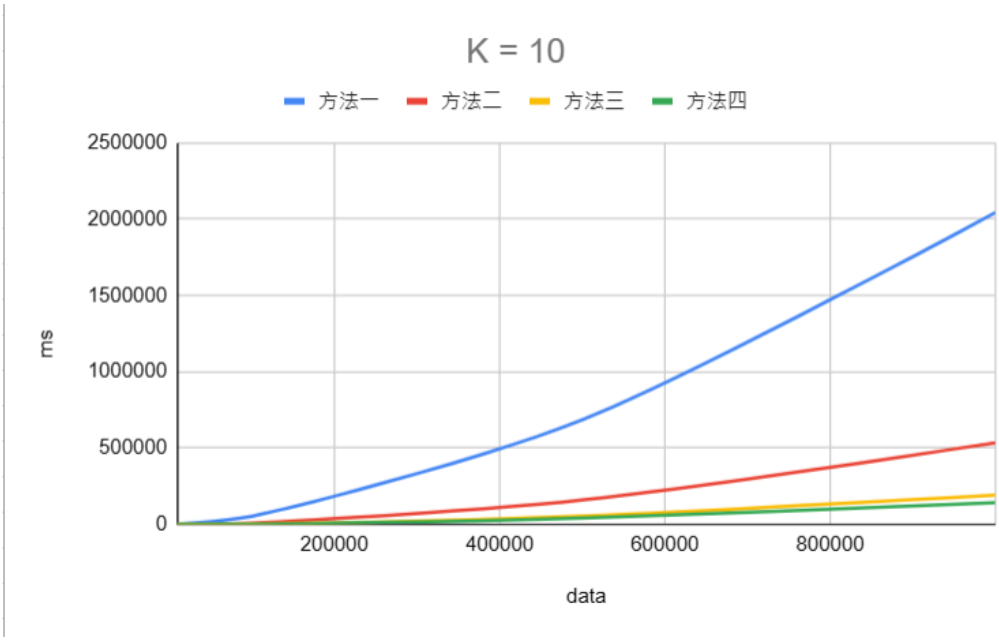

圖 1、當 K = 10 時，資料筆數與執行時間之關係

- 方法一 ：

  直接對資料進行 Bubble Sort，其處理時間與資料量成平方關係。當資料量增加 10 倍時，處理時間會增加約 100 倍。

- 方法二 ：

  由於資料被切分為 K 塊，因此其處理時間約為方法一的 1/K 倍。當資料量增加 10 倍時，處理時間會增加約 10 倍左右。

- 方法三 ：

  方法三與方法二類似，將資料切分為 K 塊，但是它使用了 K 個 process 同時進行 Bubble Sort。

  由於資料被切分為 K 塊，因此其處理時間約為方法一的 1/K 倍。因另外使用了 K 個 process，因此其處理速度比方法二更快，當資料量增加時，處理時間的增加速度會比方法二更慢。

- 方法四 ：

  方法四與方法三類似，但是它使用了 K 個 thread 而不是 process，這樣可以更有效地利用 CPU。

  與方法三相比，使用 thread 可以更快地創建和結束執行緒，因此其處理速度可能會更快。當資料量增加時，處理時間的增加速度會比方法三更慢。

綜合來看，當需要處理大量資料時，方法二、方法三和方法四都比方法一更有效率。而在方法二、方法三和方法四之間，使用 thread 的方法四可能會是最快的，但是使用 thread 也可能會增加系統的開銷。

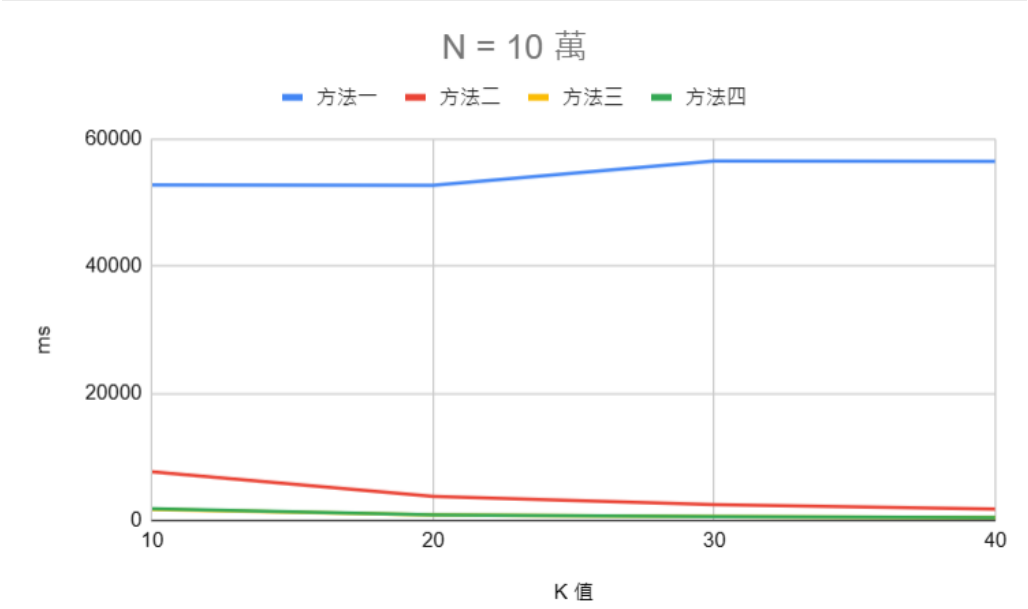

圖 1、當 N = 10萬 時，K 值與執行時間之關係

- 方法一 ：

  方法一中不需對資料進行切割，所以 K 值的變化幾乎不影響方法一的執行時間。

- 方法二:

  由於資料被切分為 K 塊，因此能顯著的減少 Bubble Sort 的執行時間，其處理時間隨著 K 值的增加而呈現下降趨勢。

- 方法三:

  同方法二，將資料切分為 K 塊，但方法三使用了 K 個 process 同時進行 BubbleSort，所以其處理速度比方法二更快。

- 方法四:

  方法四與方法三類似，但是它使用了 K 個 thread 而不是 process，這樣可以更有效地利用 CPU。

  與方法三相比，使用 thread 可以更快地創建和結束執行緒，因此其處理速度可能會更快。當資料量增加時，處理時間的增加速度會比方法三更慢。

總的來說，使用多核心處理資料可以加速處理速度，但也要注意程式的複雜度問題。在這個範例中，BubbleSort 的複雜度相對於 MergeSort 高，所以須著重於降低執行 BubbleSort 時的時間，其最直觀的方式舊時提高 K 值。