HW1 document　　資訊三甲 10827139 黃云潔

1. 開發環境

- 開發平台：
  - OS : Windows11
  - CPU：Intel® Core™ i5-8265U
  - RAM : 8GB
- 使用開發環境：Visual Studio Code 1.61
- 使用的程式語言：C/C++

2. 實作方法與流程

實作方法

```
fileAmount = fileAmount*10000;
numberlistsize = fileAmount;
numberlist = new int[fileAmount];
```

本次作業我的儲存結構使用指標指向宣告的 array，每讀入一份檔案時會從資料名稱中判斷資料筆數，並以指標宣告 int 陣列。若選擇方法二~四，會需將資料切成 k 份，若遇上無法整除的狀況，會將餘數部分加到最後一筆。

- 方法一，將讀入的 N 個數字直接進行 BubbleSort

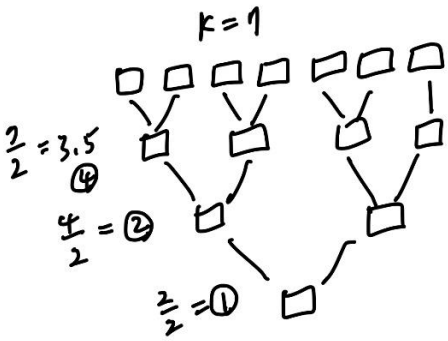
```
static void FastBubble( long start, long end ) {    // 起始位址, 結束位址
    bool hasChanged = false ;
    long total = end-start+1 ;
    int temp = 0;
    for ( int pass = 1; pass < total && !hasChanged ; ++pass ) {
        hasChanged = true;
        for ( int i = 0; i < total-pass; ++i ) {
            if ( numberlist[start+i] > numberlist[start+i+1] ) {
                temp = numberlist[start+i];
                numberlist[start+i] = numberlist[start+i+1];
                numberlist[start+i+1] = temp;
                hasChanged = false ;
            } // if
        } // for
    } // for
} // FastBubble
```

使用此 function 需給定起始位址和結束位址，方便後面的方法使用。在判斷條件中加入布林值 hasChanged，表示若排序已提早完成，則程式不需判斷到最後。

- 方法二，將讀入的 N 個數字切成 k 份，在同一個 process 內對 k 份資料進行 BubbleSort 後，再於同一 process 中將 k 份資料作 Merge

```
static void MergeKArray(int k) {
    long tempk = k, start = 0, last = 0;
    long amount = numberlistsize/tempk;
    double tempk_d = 0;
    while ( tempk/2 >= 1 ) {
        tempk_d = tempk;
        for ( long i = 0; i < ceil(tempk_d/2); i++ ) {
            start = i*amount*2;
            last = start+amount*2-1;
            if ( i < ceil(tempk_d/2) - 1 )
                Merge(start, start+amount-1, last);      //將左右兩個以排序好的子陣列合併
            else {
                if ( tempk % 2 == 0 )
                    Merge(start, start+amount-1, numberlistsize-1);
            } // else
        } // for
        tempk = ceil( tempk_d / 2 );
        amount = amount*2;
    } // while

} // MergeKArray
```



將 k 份資料使用方法一的 Bubble Sort 完成後，將 k 份資料兩兩作 Merge，若此時資料數為奇數則最後一份先保留。完成一輪的 Merge 後資料數為(原資料數/2)取頂值，直到最後 Merge 成一份就完成。

- 方法三，將讀入的 N 個數字切成 k 份，在 k 個 processes 內對資料進行 BubbleSort 後，再使用 processes 將 k 份資料作 Merge

```
    if (!CreateProcess("Merge.exe", output_par, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
        cout << "ERROR: Failed to create process" << GetLastError() << endl;
        return;
    } // if
    lpHandles[filenum] = pi.hProcess;
    lpThreads[filenum] = pi.hThread;
    filenum++ ;
} // while

WaitForMultipleObjectsExpand( lpHandles, (DWORD)filenum );
```

因為我的電腦環境為 Windows11，因此我使用**<windows.h>**函式庫中的**CreateProcess()**來建立 process。將 BubbleSort 和 Merge 方法寫成執行檔後，產生 process 來執行此執行檔。因 process 無法共享 address space，因此我在每一輪皆採用寫檔的方式，並傳遞參數告訴該 process 要對哪一個檔案作 Merge，等待所有 processes 完成後主程式再將結果讀回。

```
static void WaitForMultipleObjectsExpand(HANDLE * handles, DWORD count) {
    DWORD index = 0;
    DWORD result = 0;
    DWORD handleCount = count;
    /// 每64個HANDLE分爲一組
    while (handleCount >= MAXIMUM_WAIT_OBJECTS) {
        WaitForMultipleObjects(MAXIMUM_WAIT_OBJECTS, &handles[index], TRUE, INFINITE);
        handleCount -= MAXIMUM_WAIT_OBJECTS;
        index += MAXIMUM_WAIT_OBJECTS;
    } // while
    if (handleCount > 0) {
        WaitForMultipleObjects(handleCount, &handles[index], TRUE, INFINITE);
    } // if
} // WaitForMultipleObjectsExpand
```

等待所有 processes 完成的方法原為 **WaitForMultipleObjects()**，但由於此 function 有限制最多同時只能執行 64 個 process，因此需將 processes 分為 64 個一組進行呼叫。

- 方法四，將讀入的 N 個數字切成 k 份，在 k 個 threads 內對資料進行 BubbleSort 後，再使用 threads 將 k 份資料作 Merge

```
while ( tempk/2 >= 1 ) {
    vector<thread> threads;
    tempk_d = tempk;
    for ( long i = 0; i < ceil(tempk_d/2); i++ ) {
        start = i*amount*2;
        last = start+amount*2-1;
        if ( i < ceil(tempk_d/2) - 1 )
            threads.push_back(move(thread(Merge, start, start+amount-1, last)));
        else {
            if ( tempk % 2 == 0 )
                threads.push_back(move(thread(Merge, start, start+amount-1, numberlistsize-1)));
        } // else
    } // for

    for (int i = 0; i < threads.size(); i++) {
        threads[i].join();
    } // for

    threads.clear();
```

採用**<threads>**函式庫來宣告 thread，將 BubbleSort 和 Merge 方法及參數傳入執行緒後，便開始執行。因 threads 可共享 address space，因此多個 threads 可存取同一 process 中的資料，每一輪結束後 **join()**等待所有 threads 完成，其他同方法二。

流程

```
請輸入檔案名稱(0: end of program)：
input_100w
請輸入方法編號(1, 2, 3, 4)：
3
請輸入要切成幾份(>0)：
200

CPU Time: 6.666seconds
Output Time: Sun Apr 24 14:40:15 2022
```

輸入檔案名稱，檔案名稱需含資料筆數(如 input_100w 為一百萬筆)，接著輸入方法一~四，若輸入為方法二~四，則需再輸入欲切成幾份。

3. 特殊機制考量與設計

- **資料結構**：一開始我是採用 vector 的方式儲存，因為使用建構函式比較方便，但經實驗發現改用 array 儲存結構，測試第二個方法做一百萬筆資料切一萬份排序(僅計時排序部分)，如下表可見速度快了**兩倍**左右。

vector	array
0.947 s	0.418 s

- **BubbleSort**：我在 BubbleSort 中加入了一個判斷條件，若前面已經不需再繼續做 swap(排序提早完成)，便提早結束排序，但此方法會根據資料順序不同，提升的效率也會有所不同。
- **讀、寫檔**：因為方法三我會使用到大量的讀檔、寫檔，而此部分花費太多時間，我原本採用<fstream>中的 getline()一個一個數讀入，後來使用了<stdio.h>的 fread()一次讀入二進制檔案。測試方法三對一百萬筆資料切一百份排序(僅計時排序部分)，如下表可見速度快**四倍**左右。

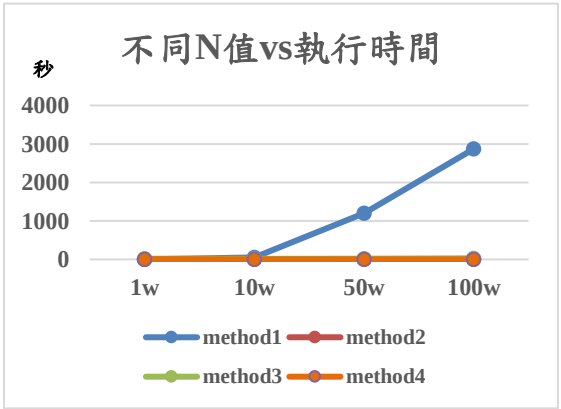
getline()	fread()
27.354 s	6.567 s

4. 分析結果和原因

（單位:秒）

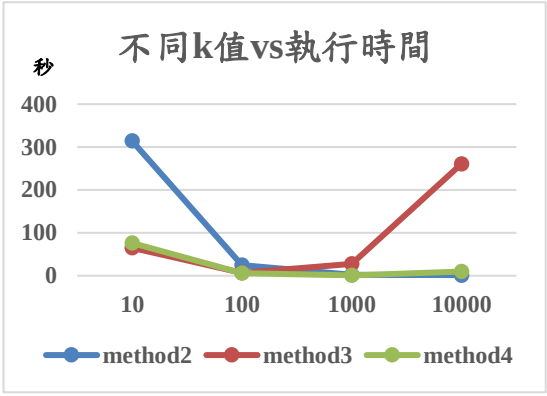
切割份數	1w	10w	50w	100w
method1				
	0.328	47.827	1202.46	2869.6
method2				
10	0.027	2.683	76.027	313.898
100	0.01	0.27	5.31	24.409
1000	0.01	0.128	0.673	1.968
10000	0.015	0.083	0.287	0.643
100000	x	0.419	0.18	0.397
method3				
10	0.186	0.631	13.192	64.798
100	3.363	1.192	3.475	6.567
150	3.829	1.693	2.503	5.238
1000	14.252	13.437	15.326	27.828
10000	151.653	189.112	152.504	260.711
method4				
10	0.058	0.623	14.873	76.153
100	0.308	0.163	1.475	5.734
500	1.58	0.377	0.7	2.909
1000	4.925	0.695	0.889	0.765
10000	26.363	25.136	24.443	9.552

➢ **比較不同 N 值與執行時間**(方法二~四取相同 k 值=1000)



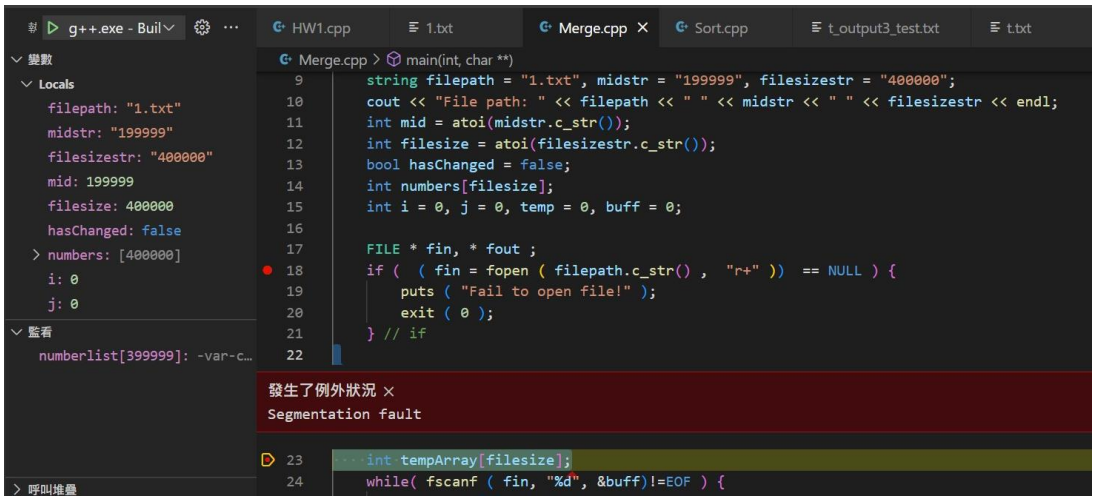
- 從圖表趨勢來看，N 越多，各方法的排序執行時間越多
- **方法一**因為沒有使用分割效率最差，BubbleSort 時間複雜度為 $O(n^2)$
- **方法二~四**使用分割方法效率較佳，將分割的資料排序好後，再使用 Merge 將資料合併，MergeSort 時間複雜度為 O(nlgn)
- 方法整體效率比較：**方法二>四>三>>一**
- **方法四**因為使用 thread，在同一 process 中並行執行，因我寫的方法在 merge 時需等每一輪兩兩 merge 完，所有 threads 都結束後，再執行下一輪，因此效能略比方法二差一些
- **方法三**使用 process 雖然可以平行執行，但由於不共享地址空間，需要寫檔及讀檔，效能比方法四更差

➢ **比較不同 k 值與執行時間**(取相同 N 值=100w)



- 根據圖表可見，各個方法花費時間最短的 k 值皆不同
- 上方表格中的紅字為不同資料筆數的最短執行時間，可見**不同方法有不同資料筆數與切割份數最適合的比例**
- **方法二** 資料筆數:切割份數=10:1
- **方法四** 資料筆數:切割份數=1000:1
- **方法三** 因讀檔、寫檔的時間較排序時間多，較看不出比例
- **方法四**因使用 thread，在 process 中有 thread scheduler 會進行 thread 的排程，節省等待時間，因此相較於方法二可明顯提升每份切割的資料處理量

5. 遇到的 bug 及解決方法



在實作 process 時，子 process 中發生了 segmentation fault，因為不會在主程式中報錯，而是直接結束此 process，所以直接造成了排序結果錯誤，我一直以為是主程式的邏輯有誤，後來才發現是宣告的問題。使用 array 的宣告方式，此變數會在 stack 記憶體上，而 stack 記憶體大小有限，當資料筆數過多，會導致溢位發生，因此需改為使用指標 new 出陣列，動態配置 heap 記憶體，使用完後再自行刪除。