F74072251 資訊系 111 級 蔡哲平

作業系統 作業一

開發環境 ：

(1) OS: Windows 10 家用版

(2) CPU: intel(R) Core(TM) i7-8750H CPU@ 2.20GHz 2.20GHz

(3) Memory: 8.00GB (7.85GB 可用)

(4) Programming Language(version): C++11

程式執行時間 ：

因在 windows 環境下跑 1G 的測資會跑非常久，故在測試各 thread 時間時只有使用 100MB 的測資，但在 ubuntu 就有使用 1G 的測資因為滿快，可以看到在 windows 下 100MB 花的時間甚至比 ubuntu 下 1G 還久。

1	Win10	100MB	Linux	1GB
2	thread	time	thread	time
3	1	33.13	1	50.458
4	2	33.778	2	49.7426
5	4	50.256	4	50.1435
6	8	63.107	8	50.5733
7	16	65.617	16	51.5822
8	32	75.636	32	50.3857
9	64	83.029	64	50.5385
10	128	80.82	128	50.6083
11	256	80.209	256	50.476
12	512	82.788	512	50.3132
13	1024	78.707	1024	50.8169
14	2048	76.163	2048	51.1432

程式開發與使用說明 ：

(1) 程式開發過程 ：

這份程式我更改了許多次，不斷嘗試不同方法，光是轉換成 json 格式的函式我就寫了五個版本，最後是使用下面這個，因為速度最快

```
7  void convertStr(string& inputStr, bool last = false) {
8      // convert input string into JSON format
9      string resultStr = "\t{\n\t\t\"col_1\":";
10     int cnt = 1;
11     int findSplit = inputStr.find("|");
12     while(findSplit != string::npos) {
13         cnt++;
14         resultStr += (inputStr.substr(0, findSplit) + ",\n\t\t\"col_" + to_string(cnt) + "\":");
15         inputStr = inputStr.substr(findSplit + 1);
16         findSplit = inputStr.find("|");
17     }
18     if(last)
19         resultStr += (inputStr + "\n\t}\n");
20     else
21         resultStr += (inputStr + "\n\t},\n");
22     inputStr = resultStr;
23 }
```

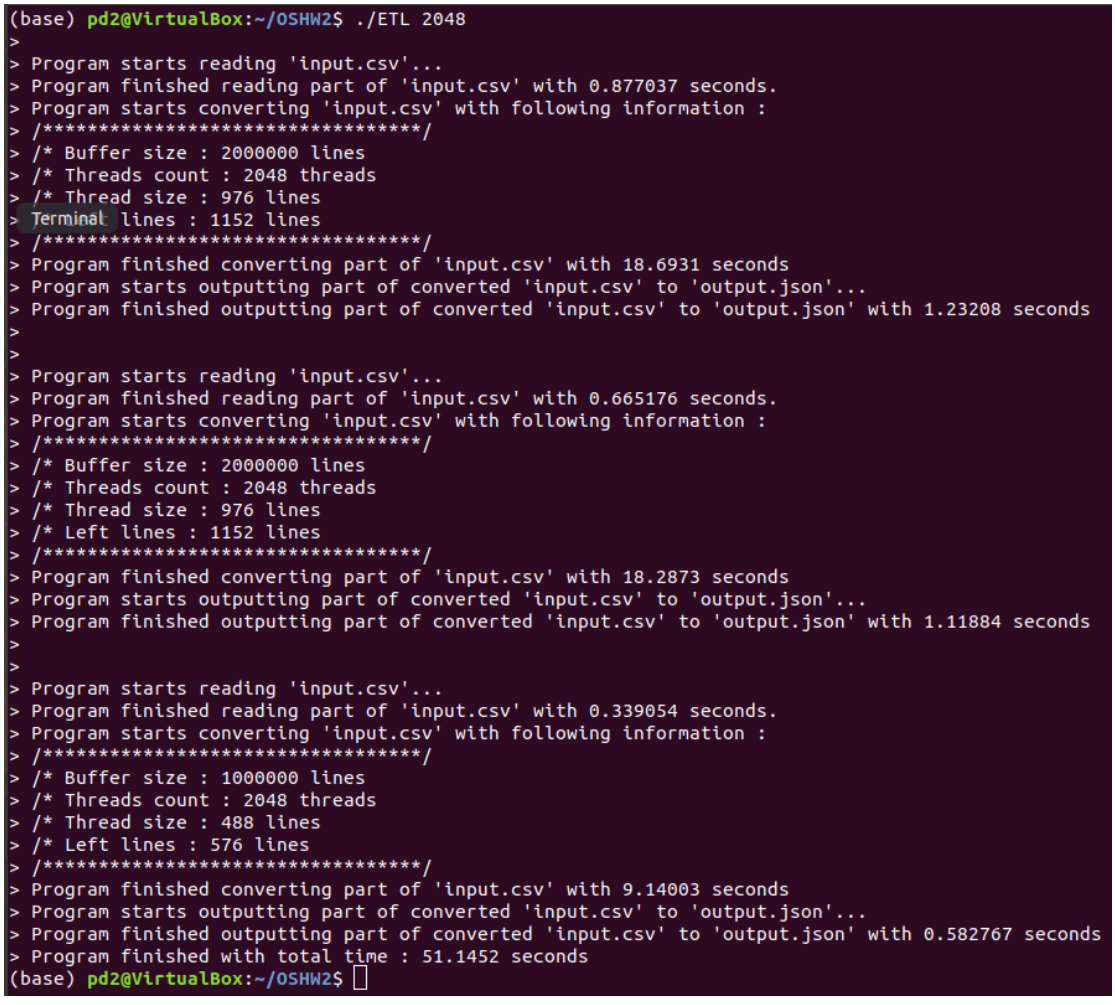
函式傳入 csv 格式字串的 reference，會直接將那個字串改成 json 格式。而我的程式邏輯就是先將 input.csv 讀入且存進一個 vector，再依照使用者輸入的執行緒個數去分配各個執行緒所負責的部分，假設 input.csv 有 100 行，總共 5 個執行緒，那第一個執行緒就是負責 inputVec 的第 0 到 19 項，其餘以此類推，而各執行緒要負責的就是將字串轉成 json 格式存回 inputVec，最後當所有執行緒都處理完成後再 inputVec 的結果輸出至檔案。第一個版本我很快就寫完，但隨這測試檔案的大小愈大，終於在測試 500M 的時候發生了 std::bad_alloc，原先我是 input 跟 output 各一個 vector，而這也是導致 memory leak 的原因，所以後來我改成直接將轉好的字串存回 inputVec，但一樣還是爆掉，所以就決定採用作業一的方式，利用一個 buffer 去處理，當 buffer 滿了就輸出，再繼續讀剩餘之 input。

(2) 在 Ubuntu 環境下程式使用方法 ：

程式編譯 ： g++ ETL.cpp -O3 -o ETL -l pthread

程式執行 ： ./ETL [thread num]

Buffer 的大小設為 200 萬，也就是一次最多能處理 200 萬行，1G 差不多是 500 萬行，程式就會自己重複跑三次才能處理完所有測資，執行結果如下圖



效能分析報告 ：

(1) I/O bound or CPU bound ?

由上面那張圖可得知，在一次讀檔、資料處理、輸出的過程中，讀檔跟輸出

分別只有大概一秒，而資料處理也就是轉換成 json 格式卻佔用了 18 秒，換句話說，整個過程中 I/O 只佔了約 10%的時間，而資料處理就佔了近 90%，由此可知這次的作業是屬於 CPU bound。下圖是我原先所寫的輸出函式：

```
8   void printVec(const vector<string>& stringVec, ofstream& fout, bool last = false) {
9       int line = 1;
10      for(int i = 0; i < stringVec.size(); i++) {
11          if(line == 1) {
12              fout << "\t{\n";
13              fout << "\t\t\"col_" << line << "\":" << stringVec.at(i) << ",\n";
14              line++;
15          }
16          else if(line == 20) {
17              fout << "\t\t\"col_" << line << "\":" << stringVec.at(i) << "\n";
18              fout << "\t}";
19              if(i == stringVec.size() - 1 && last) {
20              }
21              else {
22                  fout << ",";
23              }
24              fout << "\n";
25              line = 1;
26          }
27          else {
28              fout << "\t\t\"col_" << line << "\":" << stringVec.at(i) << ",\n";
29              line++;
30          }
31      }
32  }
```

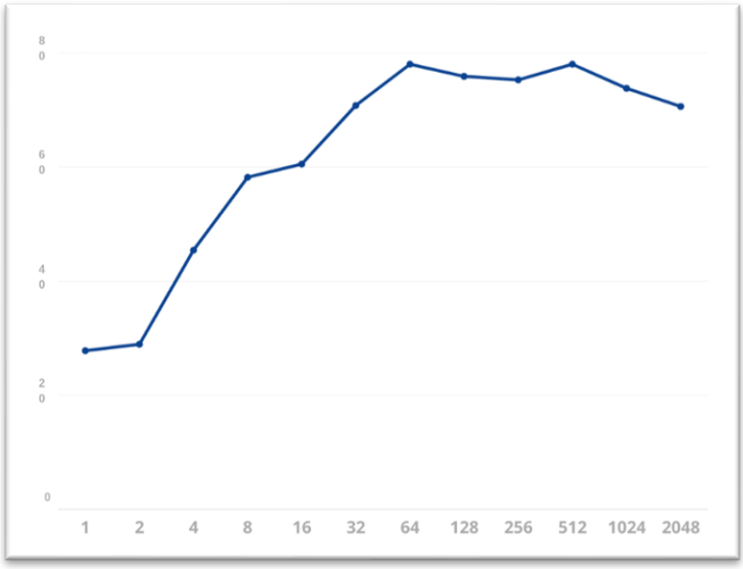
可看出我在函式中頻繁的使用 fout，而不是直接輸出一整串轉好的 json 格式至檔案，實測後發現這種寫法會拖慢速度，我的理解是因為這樣會使作業系統頻繁的在 cpu 跟 I/O 之間做切換，但其實可以先利用 cpu 將整串完全轉好成 json 格式，再一次整串輸出至檔案，就可以解決此問題，以下是我的程式碼：

```
146     int outputStart = clock();
147     if(!bufferFull)
148         fout << "[\n";
149     for(auto& i: inputVec)
150         fout << i;
151     int outputEnd = clock();
```

這樣輸出就是一次一整行，可大幅減少 I/O 時間。

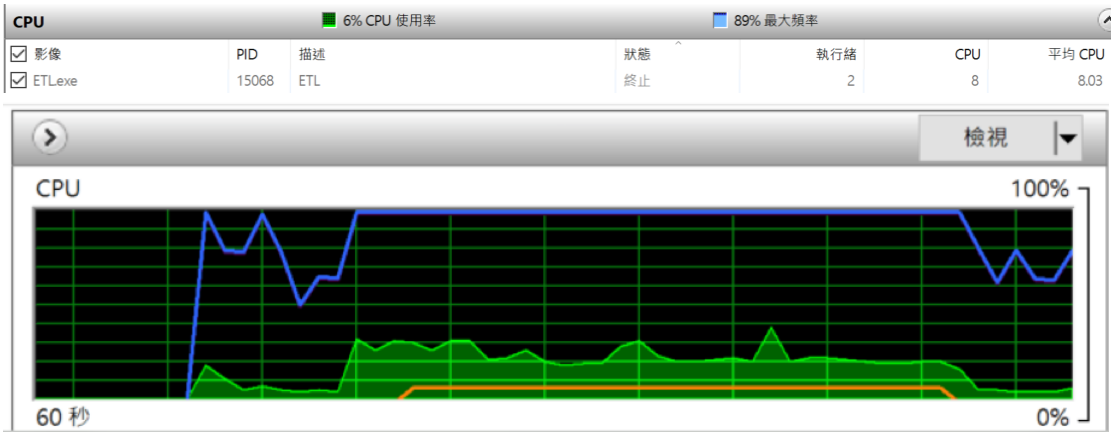
(2) 不同 thread 數量的比較

hread	time
1	27.804
2	28.925
4	45.418
8	58.192
16	60.516
32	70.782
64	77.997
128	75.872
256	75.261
512	77.997
1024	73.787
2048	70.602

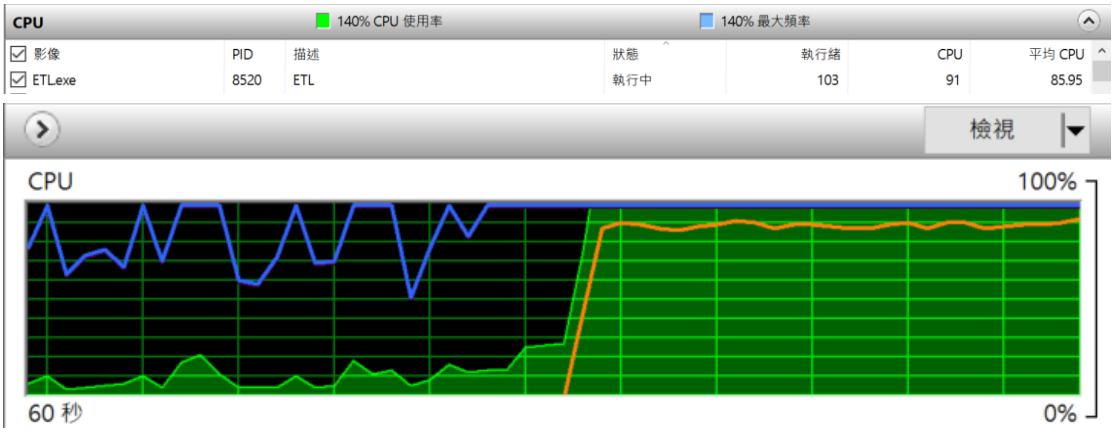


上圖是在 windows 環境下測試 100M 的資料處理部分所花的時間，由於檔案輸入及輸出，在不同 thread 數量時差距很小，故把資料處理的部分拿來討論，可以發現 thread 個數越多，資料處理的速度反而越來越慢，由下圖可看到在 1 個執行緒及 2048 個執行緒時 cpu 使用量差異是極大的，

Thread : 1 / CPU : 8%

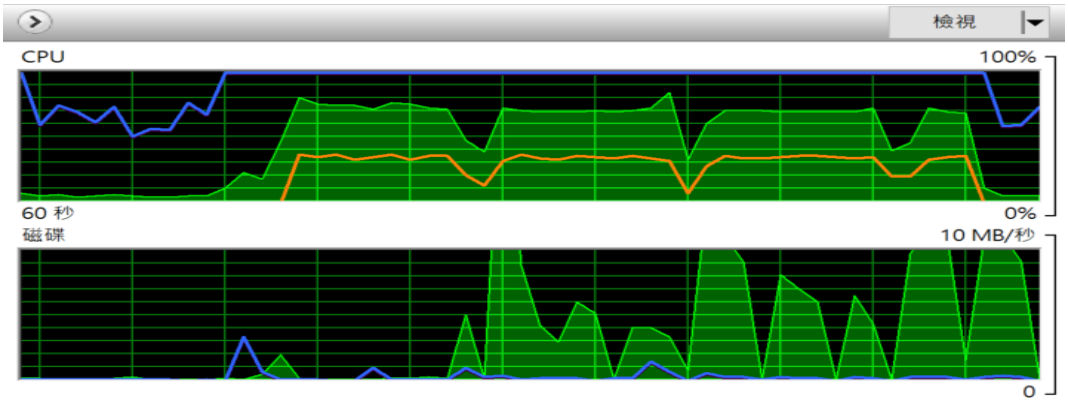


Thread : 1024 / CPU : 90%up



雖然 1 個執行緒只使用了 8%的 cpu，但是卻比使用了 90%以上 cpu 的 2048 個執行緒速度快上非常多，上網查了一下資料後，發現其實各執行緒是不會同時執行的，一顆 CPU 同時只能執行一項任務，為了讓使用者感覺這些任務正在同時進行，作業系統的設計者巧妙地利用了時間片輪轉的方式，CPU 給每個任務都服務一定的時間，然後把當前任務的狀態儲存下來，在載入下一任務的狀態後，繼續服務下一任務。任務的狀態儲存及再載入，這段過程就叫做上下文切換。時間片輪轉的方式使多個任務在同一顆 CPU 上執行變成了可能，但同時也帶來了儲存和載入的直接消耗，而這也導致了更多的執行緒不見得會使速度變快。

(3) CPU 在程式執行中的使用狀況



上圖是我使用 8 個 thread 的程式完整生命週期，這次測試是讀進 50 萬行，buffer 存 15 萬行，所以 cpu 那邊共有三次起伏，可以觀察到在在資料處理的部分 cpu 使用量很高，而因為本次作業資料量不大，所以 I/O 時間極短，cpu 閒置的情況不嚴重。

作業系統的優化：

雖然這次作業在 I/O 上的時間並不多也並非決定執行時間的主要因素，但我認為若能在讀入和輸出至檔案的時候也採用多執行緒的方法，在處理更大的資料量時勢必能加快速度，很遺憾我並沒有找到方法是能做到這種事，在 I/O 時都只能使用一個執行緒去執行，所以我認為要是作業系統也能在檔案讀寫時採用多執行緒的方式，且不會造成寫入檔案時順序亂掉，那勢必是個很棒的優化。雖然實測後果發現多執行緒並不依定能使執行時間加速，但我認為多執行緒的採用是為了發揮 CPU 的最大效能，提高資源使用效率，比如你的應用程式需要訪問網路，因為網路有延時，如果在介面執行緒訪問，那麼在網路訪問期間介面將無法響應使用者訊息，這時就應該使用多執行緒，但我認為作業系統在各執行緒的切換時仍需做優化，已我得測試結果為例，1 個執行緒及 2 個執行緒的執行時間比較上，2 個執行緒還是輸 1 個執行緒，這就表示執行緒切換所需的時間，並沒有比使用多執行緒節省的時間還要來的多，這是讓我覺得十分驚訝的部分。