HW3 - All-Pairs Shortest Path (CPU)

106062322 江岷錡

Implementation

本次使用 `Pthread` 作為主要的 threading library，以下將分別講解各自部分的實作：

ReadData

在 ReadData 的部分，仿照 `validator.cc` 的寫入，使用 `ifstream` 進行檔案讀取。

```
std::ifstream f(argv[1]);
if (not f) {
    throw reprintf("failed to open file: %s", argv[1]);
}
f.seekg(0, std::ios_base::end);
f.seekg(0, std::ios_base::beg);
int E;
f.read((char*)&V, sizeof V);
f.read((char*)&E, sizeof E);
```

值得一提的是，由於本次的 matrix 為 2-d 陣列，為了加快讀取速率，我使用 `C++ Vector` 進行 matrix 的存取。

```
vector<int> row;
row.assign(V, INF);//配置一個 row 的大小
dist.assign(V, row);
for (int j=0;j<V;j++) {
  dist[j][j] = 0;
}

for (int i = 0; i < E; i++) {
  int e[3];
  f.read((char*)e, sizeof e);
  dist[e[0]][e[1]] = e[2];
}
```

APSP Algorithm

本次使用 `Floyd Warshall Algorithm` 進行實作，在原始的 algorithm 下，會試著窮舉出所有的 `i, j, 以及第三點 k` 的可能性，複雜度為 `O(V³)` 。

在我的實作中，我參考了一篇 < Rectangular Algorithm > 的 Paper 進行改寫。我在第 `i` 層進行平分，讓每個 thread 拿到的計算量相似。切分的方法，為了確保 Memory Access Locality ，是以相臨的 `i` 做切分（假設平均後一個 thread 拿到 `k` 個 col ， 第 0 thread 拿到 `0` → `k-1` 的資料，第一個 thread 拿到 `k` → `2k-1` 的資料 ...）。

Paper Link: https://reader.elsevier.com/reader/sd/pii/S0893965911002928?token=99F343875CA7D42311EE44C85AF13054D9AD82A71A3452F28A69D2F42B671E56E7754B98DA4BE8FB2F4483

同時，進行 Early Stop ，因為 matrix 的 diagonal 不需計算，且當任一 row 及任一 col 出現無限值時，該 row 或 col 也不須計算。透過上述特性，即可減少部分計算量。

而因為 `Floyd Warshall Algorithm` 是有序性的 algorithm ，下一個 stage `k` 需等前一個 stage `k` 結束才可計算，因此我使用 `pthread_barrier` 進行排程控制。

```
int iLower = V * realThreadId / ncpus;
int iUpper = V * (realThreadId + 1) / ncpus;

for (int k=0; k < V; k++) {
  // For Locality
  for (int i = iLower; i < iUpper; i++) {
    if (i == k || dist[i][k] == INF) continue;
    for (int j = 0; j < V; j++) {
      if (dist[i][k] + dist[k][j] < dist[i][j]) {
        dist[i][j] = dist[i][k] + dist[k][j];
      }
    }
  }
  pthread_barrier_wait(&barr);
}
```

Complexity

由於基底是 `Floyd Warshall Algorithm` ，但有用 Pthread 進行均分，大致會趨近 `O(V³ / P)` 。不過因為有 early stop 的設計，因此執行效率上會再稍微提高一些，下方有實驗證明。

Write Back

同寫入方式，我使用 `std::ofstream fout` 的方式進行寫回。

```
std::ofstream fout(argv[2]);
for(int i=0; i<V; i++){
  for(int j=0; j<V; j++){
      fout.write((char *)(&dist[i][j]), sizeof(int));
  }
}
fout.close();
```

Generate Test Case

由於在 `Floyd-Warshall algorithm` 的寫法下，耗時共可分成三個部分：

- Input：O(E)
- Computation：O(V³)
- Output：O(V²)

因此，若要製造出最耗時的 testcase ，便是最大化 `V` 與 `E` 。我製作出的 testcase 中，企圖讓 V 達到極限值 `6000` 。相對應的，也創造出 dense 的 Matrix ，讓 E 也達到極限值 `35994000` （ `V² - V` ，且因為是 directed ，任一點又都不可自連）。

同時，在 `Floyd-Warshall algorithm` 這個演算法下，迴圈內每次所執行的項目為：
1. 判斷新路線是否快過舊路線
2. 若是，則賦與新值

也因此，若要設計出最多次計算的測資，就是增加多一些 `第二步賦值` 動作。實作上，我會讓 row index 漸增時， weight 逐步下降，確保新路線會快於舊路線。經過測試，新測資約需運行 `28s` 。

Experiment & Analysis

Methodology

System Spec

使用學校提供的 Apollo Cluster。

Performance Metrics

本次使用 `clock_gettime` 進行採樣 time。由於用此 function 可達成 nanosecond 級的採樣，得以更細緻的進行表現差異的分析。

一個標準的 `clock_gettime` 範例如下：

```
struct timespec start, end, temp;
double time_used;
clock_gettime(CLOCK_MONOTONIC, &start);

....  something to be measured  ...
```

```
clock_gettime(CLOCK_MONOTONIC, &end);
if(end.tv_nsec < start.tv_nsec){
  output = ((end.tv_sec - start.tv_sec -1)+(nano+end.tv_nsec-start.tv_nsec)/nano);
} else {
  output = ((end.tv_sec - start.tv_sec)+(end.tv_nsec-start.tv_nsec)/nano);
}
time_used = temp.tv_sec + (double) temp.tv_nsec / 1000000000.0;

// Get what we want to evaluate
printf("%f second\n", time_used);
```

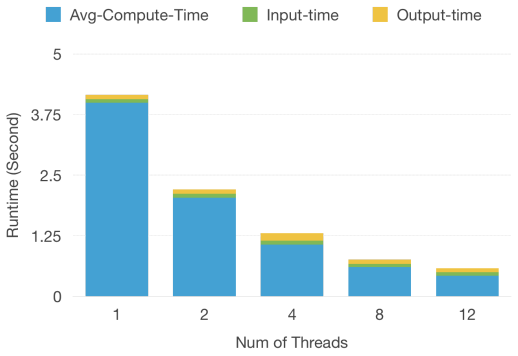
Time Profiling

為了有效驗證本次實驗的效率，我使用 testcase c16（較小） 與 c20 （較大） 當作測試測資。 c16 共有 `1500` 個點與 `1085102` 個邊，而 c20 有 `5000` 個點與 `7594839` 個邊。同時，共設定 5 種 thread : `1, 2, 4, 8, 12` 作為測試。

同時，因為 pthread 會平行處理 computation ，我計算 average computation time 作為參考 metric 。

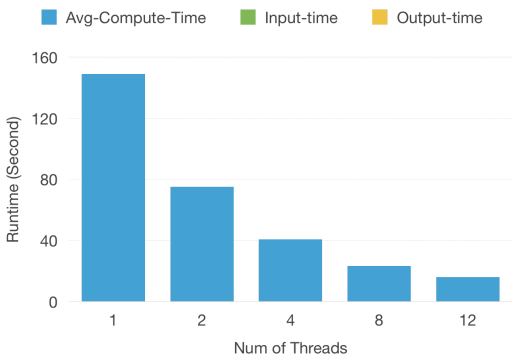
Strong Scalibility c16

	1	2	4	8	12
Avg-Compute-Time	3.995495	2.031748	1.071598	0.605274	0.427173
Input-time	0.075740	0.086741	0.075499	0.064512	0.073511
Output-time	0.084985	0.085512	0.155055	0.085444	0.078074



Strong Scalibility c20

	1	2	4	8	12
Avg-Compute-Time	148.980377	75.186832	40.803805	23.266811	15.974583
Input-time	0.062232	0.062343	0.075129	0.064512	0.075499
Output-time	0.086741	0.079432	0.084234	0.076741	0.084985



從上兩圖可見，因為 `input size` 沒有改變，所以所有情況下 `IO-Time` 皆類似。然則， `Average Computational Time`，可大致隨著 thread 數量變多，有較好的效能加速。下方的 Strong Scalibility 會有更詳細的解說。

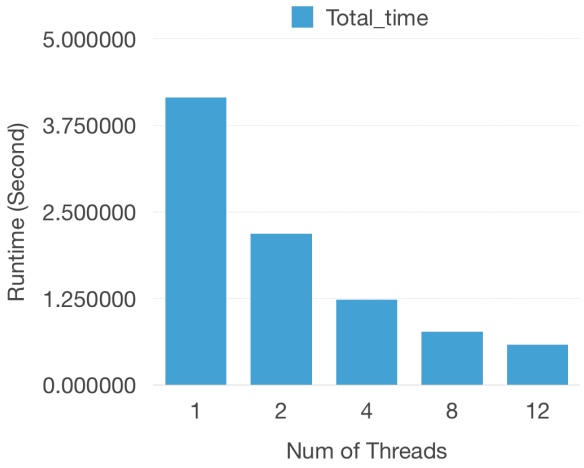
Strong Scalibility

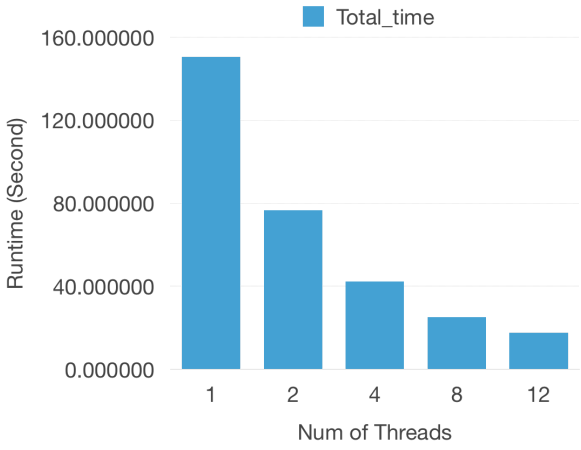
Strong Scalibility - c16

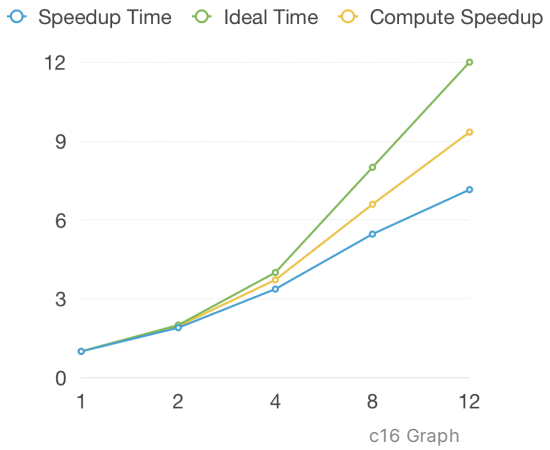
	1	2	4	8	12
Total_time	4.150630	2.183093	1.230083	0.768176	0.581009

Strong Scalibility - HW3-c20

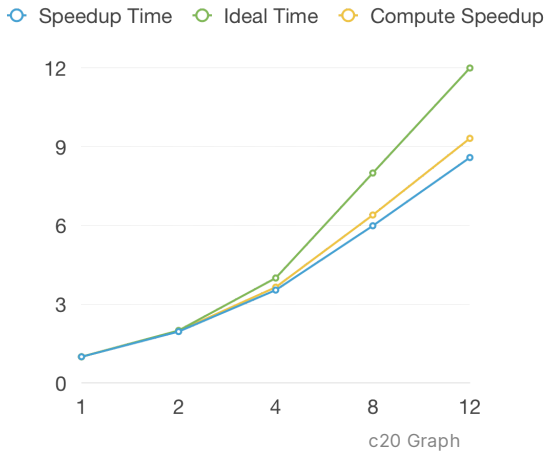
	1	2	4	8	12
Total_time	150.569550	76.791931	42.450867	25.168787	17.512062







c16 Graph



c20 Graph

從上圖可見，當我們去除掉 IO-Time 的因素，computation 的成長效率，會更貼近理想的狀況。而在 c16 情況下，因為 computation 較少，導致效率成長上受 IO_Time 的影響較大。然則，在 c20 情況下，因為 computation 較大，導致效率成長上受 IO_Time 的影響較小。

而 computation time 無法有效的跟隨 threads 數量提升，可能原因有:

1. **memory access 的議題**：在本次實作中，我試著讓 matrix 存成 vector ，在最終效能上有大幅提升，這表示 memory access 是本次效能的重大 bottleneck ，若能找出更有效的 matrix 資料分配方法，應該表現會再更好。
2. **Pthread 互卡**：遷就於演算法的特性，所有的 thread 都必須執行完 `k` round 後，才可一起進入下一次 `k+1` round 。較快完成的 pthread ，就必須等待較慢完成的 pthread ，這可能也是 bottleneck 的主要原因。

Correlation Between Time & Matrix Pattern

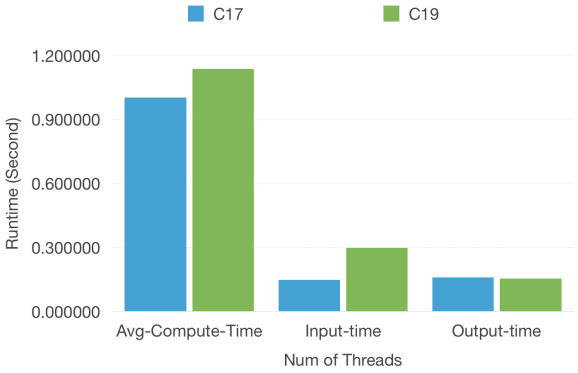
由於本次實驗中，我們的設計理論上能達成：

- Input：O(E)
- Computation：$O(V^3/P)$
- Output：$O(V^2)$

由此可見，理論上當點的數量固定， Graph 更 Dense 時，Input 的執行時間應該要較高。我使用 `c17` 與 `c19` 兩個測資驗證這個想法。 `c17` 共有 `2100` 個點與 `1805699` 個邊，而 `c19` 也有 `2100` 個點，但 `3862222` 個邊。

Correlation between time and input pattern

	C17	C19
Avg-Compute-Time	1.001610	1.136383
Input-time	0.147319	0.298913
Output-time	0.159190	0.154452



從圖中得出以下結論：

1. **邊的數量與 input-time 成絕對關係**：由於 C19 的邊數量高過於 C17 ，所以 input time 也合理的較長。然則，因為 output time 是以 $O(V^2)$ ，所以當點數量相同時，時間相近。
2. **Early Stop 有其效果**：在原本的演算法中，理論上 Computation Time 只會和 `v` 的數量有關係，因此上述兩個測資，computation time 應該要相同。但在我的實作中，有進行 Early Stop 的操作，當 graph 較為 sparse 時，會進行較少的 for-loop cycle ，使其有較短的執行時間，此實驗也驗證其結果。

Experience & conclusion

這次體驗了當助教的生測資的過程，也重新喚起 Shortest Path 的記憶，重溫習了 Data Structure 課，是個蠻好的學習經驗。上完平程，也比較會做實驗 present 自己的成果。

不過蠻好奇那些做很快的人，是使用其他的演算法，還是用其他的 threading 方法（目前已試過 OpenMP \ std::thread ，似乎沒有現在的效果好），期待助教的公布，辛苦了！