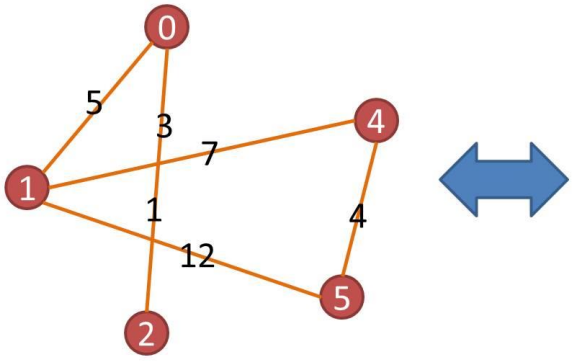
HW 3: Single Source Shortest Path

105062600 Yi-cheng,Chao 16:32, December 18, 2016

Design Concepts – SSSP_Pthread

1. Pthread 使用 shared memory 作為資料共享，圖形為 connected undirected graph(spec 上 input graph 為 weighted directed graph，但每條 path 為 bidirectional 且雙向 weighted 相同的情況下幾乎等同於 undirected graph)，在沒有 negative weighted 的情況，我使用了時間複雜度為 $O(V^2)$的 Dijkstra's algorithm，也因為是 Greedy 法我認為 performance 應該是最佳的。
2. 在讀取完 input file 第一行的頂點數 V 與邊數 E 後，為了防止輸入圖形不符合規格而導致程式出錯，判斷 V-1 是否大於 E，若成立則代表輸入圖形非 connected graph，則直接中指程式。
3. 資料儲存方式我使用 adjacency martix 記錄每條 path weighted，並且放在 shared memory 中供各 thread 存取使用，若無法到達則給值 infinity(infinity 定義為-1)，如下圖所示：



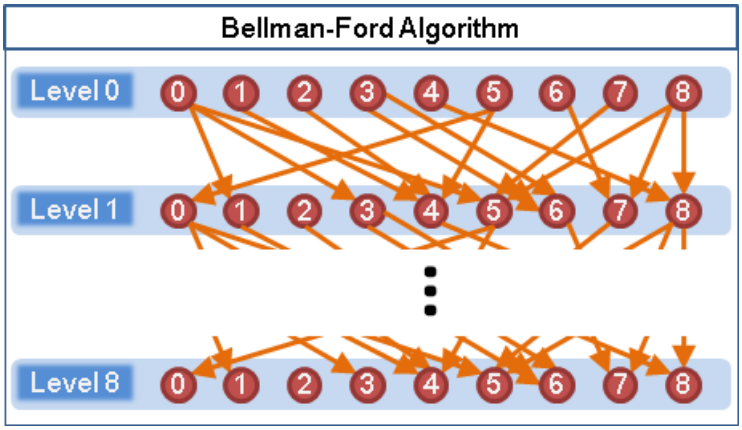
Adjacency matrix

	0	1	2	3	4
0	-1	5	3	-1	-1
1	5	-1	-1	7	12
2	3	-1	-1	-1	-1
3	-1	7	-1	-1	4
4	-1	12	-1	4	-1

4. 對於 connected graph 來說，找出 single source shortest path 的問題就是找出該 connected graph 的 minimum spanning tree，因此對於路徑的輸出我使用一個矩陣 parent 紀錄該 vertex 的 parent，如此一來輸出結果時只要使用 stack 反向輸出結果即可符合輸出檔案需求。
5. 使用矩陣 distance 紀錄當前起點到各點的距離，預設距離為最大 32bit 整數 2147364847，而計算的初始條件為起點到自身距離為 0。
6. 各線程主要平行的工作為：
   a. 先由單一程式搜尋 distance[]陣列裡頭的數值，尋找一個目前不在最短路徑樹上而且離起點最近的點，選定後將此點標記為已拜訪過，令剛剛加入的點為 a 點，以窮舉方式，根據線程數分配找一個不在最短路徑樹上、且與 a 點相鄰的點 b，把 distance[a] + adjacency[a][b]存入到 distance[b]當中(即邊 ab 進行 relaxation)。
7. 終止條件為如果點 a 沒有再被更改過，代表與起點有連通的最短路徑皆已找完，即跳脫迴圈結束計算。

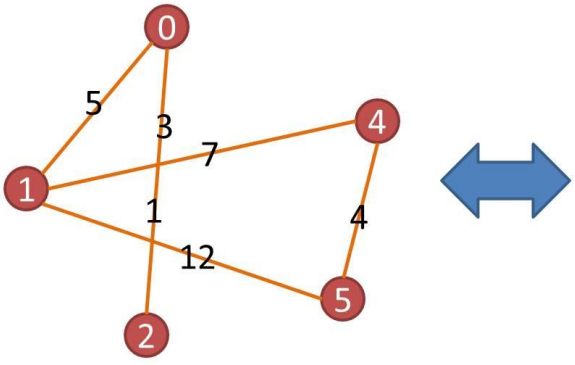
Design Concepts – SSSP_MPI_sync

1. 因為沒有 negative weighted，因此我使用平行化版本的 label correcting algorithm，即 Bellman-Ford algorithm，時間複雜度為 $O(V^3)$，因為是 vertex-centric model，每個 MPI process 各自 hold 自己的 parent 以及起點到自己的距離 distance，每個 iteration 都各自修正與鄰點的最短路徑長度，直到沒有路徑的修改則終止計算，大致上運作如下圖所示：



(圖片引用自演算法筆記-http://www.csie.ntnu.edu.tw/~u91029/Path2.html)

2. 資料儲存方式我使用 adjacency martix 記錄每條 path weighted，與 Pthread 不同的部分為使用一維陣列僅儲存該 vertex 到其他鄰點的 weighted，若無 path 相連則給值 infinity(infinity 定義為-1)，adjacency matrix 的分割如下圖所示：



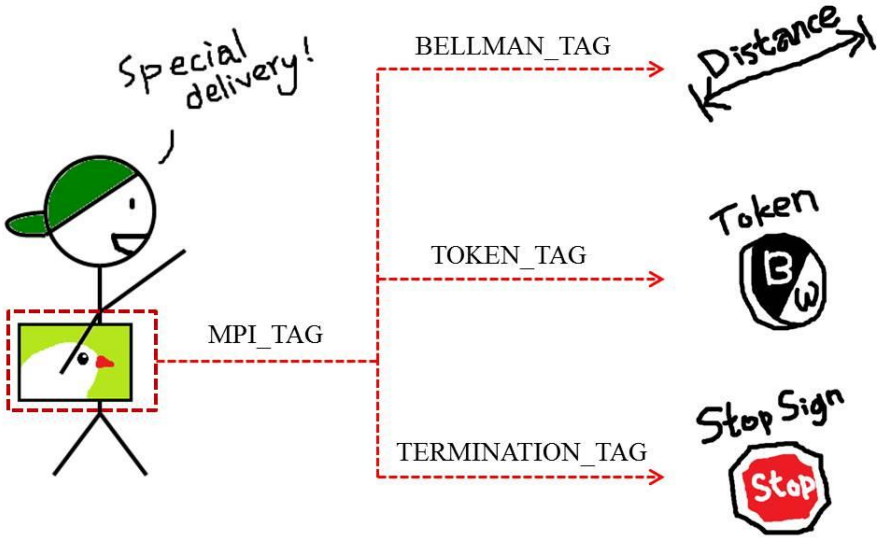
Adjacency matrix

	0	1	2	3	4
Rank = 0	-1	5	3	-1	-1
Rank = 1	5	-1	-1	7	12
Rank = 2	3	-1	-1	-1	-1
Rank = 3	-1	7	-1	-1	4
Rank = 4	-1	12	-1	4	-1

3. Synchronous communication 部分使用 MPI_Sendrecv 傳遞訊息，在每個 interation 接收一次鄰點傳來的 distance，比對是否要更新更短的距離值，若有更新的話則把終止條件的 sorted 參數改為 false，並且把 parent 更新為該鄰點，在每次 iteration 結束後使用 MPI_Allreduce 蒐集 sorted 參數，判斷使否要進行下一次 iteration。
4. 運算完後使用 Rank0 當作 master processor，把各自 hold 的 parent 使用 MPI_Gather 蒐集至 master processor，並使用與 Pthread 版本相同的方式由 master processor 輸出結果。

Design Concepts – SSSP_MPI_async

1. 大致上架構與演算法使用皆與同步化版本相同，僅因為 specification 規定在計算時需要用非同步架構操作，因此取消每次的 iteration control loop，使用 while loop 持續使用 MPI_Recv，並且利用 MPI_ANY_TAG 和 MPI_ANY_SOURCE 接受任何標記與來源來接收包裹，再由 status.MPI_TAG 來判斷該包裹內的訊息為何，如下圖示意：



2. 使用 Dual-Pass Ring termination algorithm 作為終止條件的偵測，對於 Rank0 判斷接收到的 token 為何種顏色，若為白色 token 則送出標記 TERMINATION_TAG 的包裹，並且自己使用 break 跳出 while loop，而若是黑色 token 則送出標記 TOKEN_TAG 的白色 token 包裹，目的地為下一個 Rank；而對於其他非 Rank0 的 process 則持續送出接收到的 token 給下一個 process，若接收到 TERMINATION_TAG 則繼續傳遞該訊號給下一個 process，且自己也在送出後使用 break 跳出 while loop。
3. 無論是 Bellman-Ford algorithm 或 Dual-Pass Ring termination algorithm 都會先送出第一批訊號 engage 兩個 algorithm，如此才可以持續地進行接收包裹，而除非是使用 break 跳出 while loop，否則會持續等待包裹的寄來。

Experiment & Analysis

System Environment
執行程式使用課程提供的 Batch Cluster。

Time Measurement
Pthread 版本指令部分使用<sys/time.h>library 的 gettimeofday()函數，其餘兩個 MPI 版本指令部分則使用 MPI_Wtime 來得到時間，兩者時間精度皆為微秒。

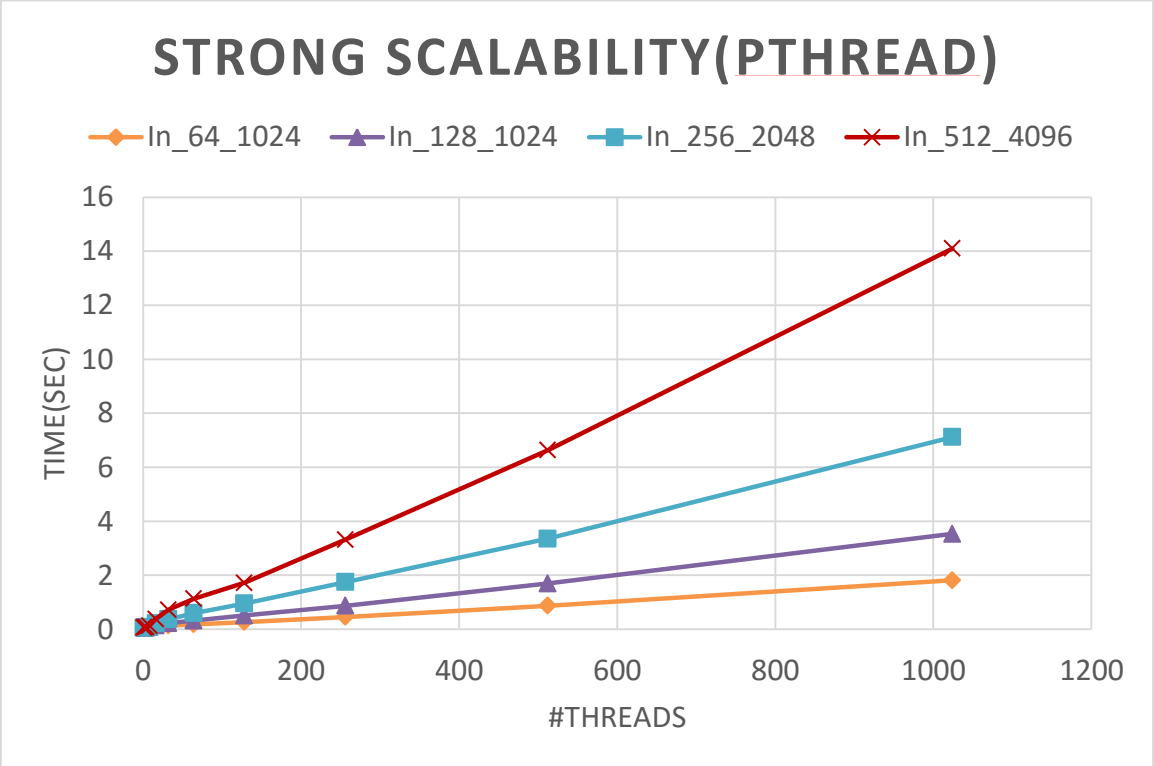
Performance Measurement
測資隨我的分析方法而改變，因此列在每個測資的下方。

Pthread-Version Anaylsis

因為 Pthread 版本方法的使用與其他兩者 MPI 版本有很大的不同，因此報告要求的分析上獨立 Pthread 版本進行分析。

Strong Scalability Analysis
(x-axis: # of Threads; y-axis: Execution time (s))
Input parameter:
./executable (2^0~2^{10}) Input_File Output_File Source

Scalability to number of threads(Problem size is fixed)，default source id is 30 for all testcases.

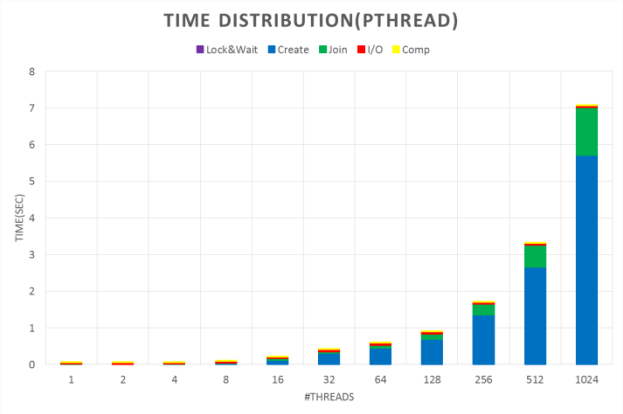


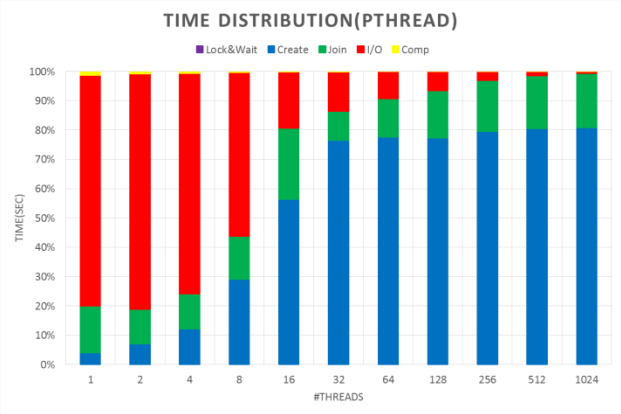
Pthread 版本有非常差的 strong scalability，甚至產生了隨著進行平行化的 threads 數增加然而執行時間卻不增反減的情況，推測是在 Pthread 的使用成為了 Bottleneck，然而接下來我會使用 time distribution 來進行佐證，此外因為使用演算法為 Dijkstra's algorithm，其時間複雜度為 $O(V^2)$，

數據上也顯示每當 vertex 數變大執行時間會隨之增長，但每當 vertex 數增加一倍，執行時間僅亦隨之翻倍，並沒有產生執行時間在預期上應為四倍的結果。

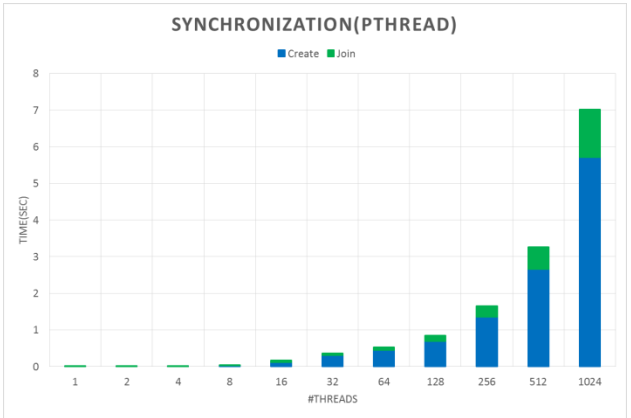
Time Distribution Analysis
(x-axis: # of Threads; y-axis: Time Cost (s))
Input parameter:
./executable (2^0~2^{10}) In_256_2048 Output_File Source

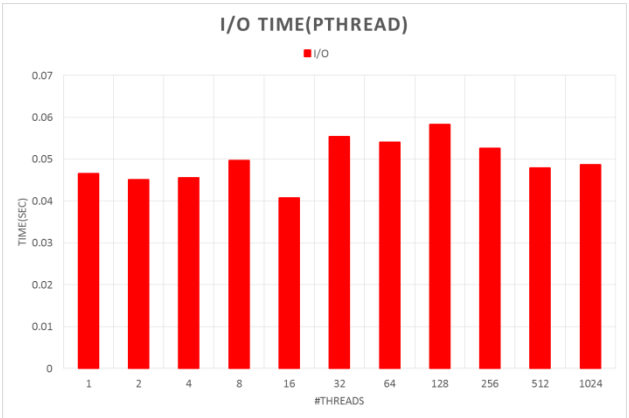
Partition execution time by {(lock&wait)x(pthread_create)x(pthread_join)x(I/O)x(computation)} and anaylsis, input testcase is 256 vertexes with 4096 edges.



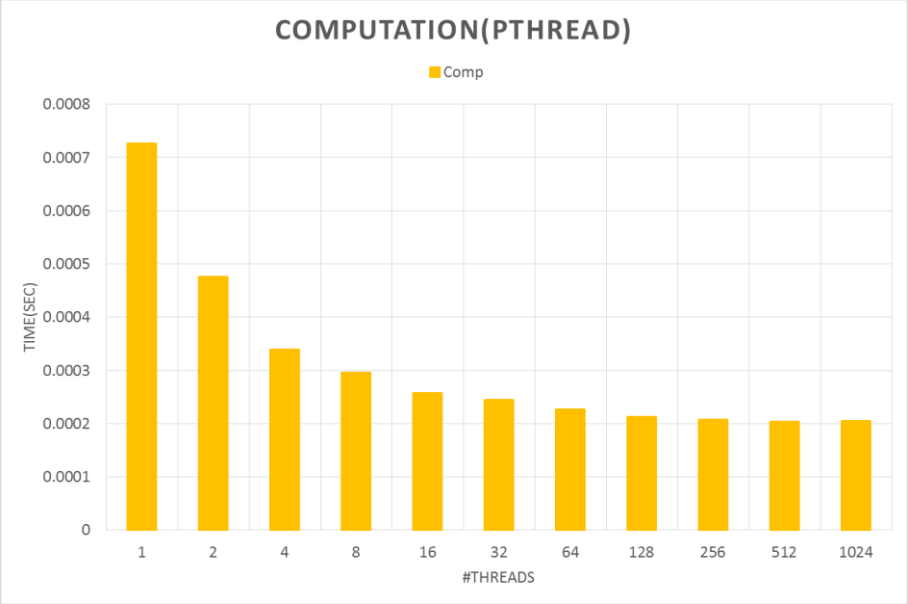


在時間分布圖上我們可以清楚的看到，隨著 parallel thread 數的增加，bottleneck 從原本的 I/O time 變為 thread 在 create 及 join 的時間，然而因為每次新路徑的選擇及計算上並不複雜，導致平行去處理計算在 computation time 並沒有得到太多的效益，在接下來每項時間拆開的時間分析上我們可以更清楚的看到，此外因為我沒有使用到 critical section，因此沒有 lock&wait time。



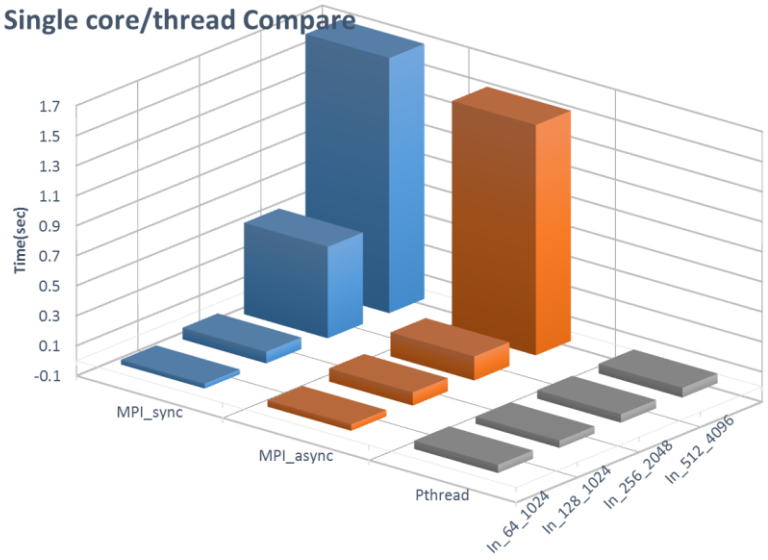


I/O time 隨著平行的 Pthread 數增減上僅是誤差內的浮動值，然而 pthread 的 create 及 join 隨著 Pthread 數的增加上值興時間也與其幾乎指數倍率的上升，因此我們可以知道對於 Pthread 的使用上如果僅是簡單的運算，反而在 thread 的產生幾乎會成為 execution time 的 bottleneck，而得不到預期上 performance 的增加，而本末倒置。



如果從時間分布圖中獨立抽出 computation time 來看便可以看到隨著 thread 數的增加上而減少運算時間的 strong scalability，但就算是最久的 computation time 也僅花 0.7ms 而已，因此也是為什麼在 Pthread 的 strong scalability 圖上看不到認知中的結果。

Single Core/thread Anaylsis (Pthread x MPI_sync x MPI_async)



此外如果比較單一資源下三種版本的程式執行時間，我們可以觀察到沒有特別 Specification 的 Pthread 執行時間是最好的，僅在執行 In_64_1024 這組 pattern 上小於 MPI 版本，而 MPI_sync 版本受圖的 vertex 數影響非常大 (在使用 In_512_4096 這組 pattern 上時間為 9.65 秒，上圖有超出圖案邊界的部分)，我認為是因為 Dijkstra's algorithm 原理上使用 Greedy 法，因此會比使用 Dynamic Programing 法的 MPI 版本在執行時間上來得更快。

MPI-Version Anaylsis

因為在執行的時候部分會遇到硬體或 scheduler 上的限制導致無法正確蒐集到數據，因此分析實體核心數僅使用{nodes = 1, ppn = 1~12}以及{nodes = 1~4, ppn = 1}等兩種組合來進行分析。

Strong Scalability Analysis

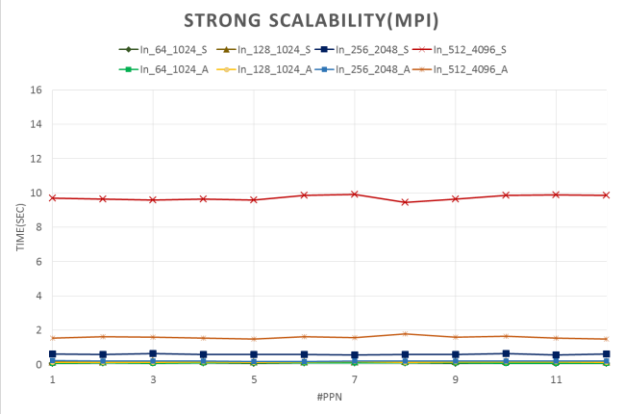
(x-axis: # of Entity cores (# of ppn/ # of nodes); y-axis: Execution time (s))

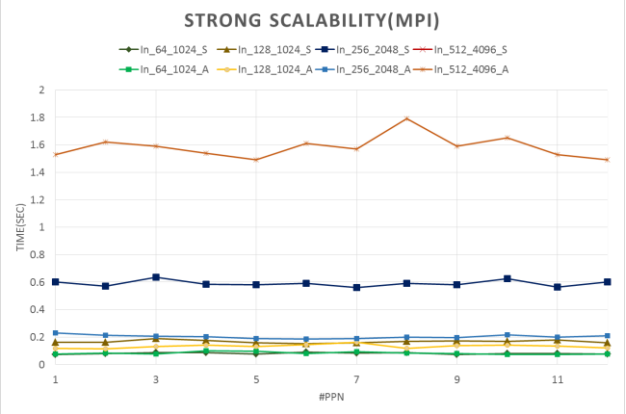
Input parameter:

./executable (unused) Input_File Output_File Source

Scalability to number of entity cores of process per node, default source id is 30 for all testcases.

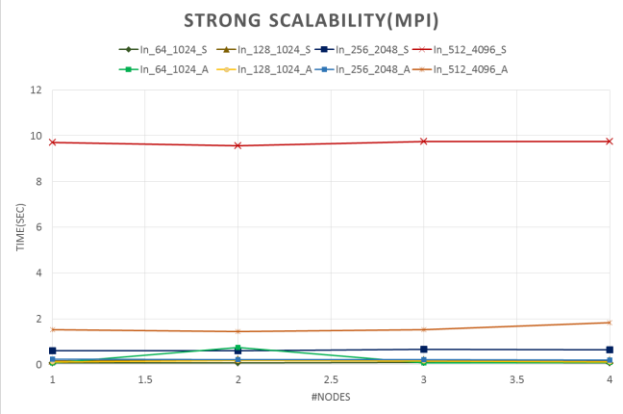
左圖為{4 組測資, 2 種版本}的 strong scalibility 分析圖，右圖則僅去掉 In_512_4096_S 的折線以便更好觀察到較小測資間的關係。

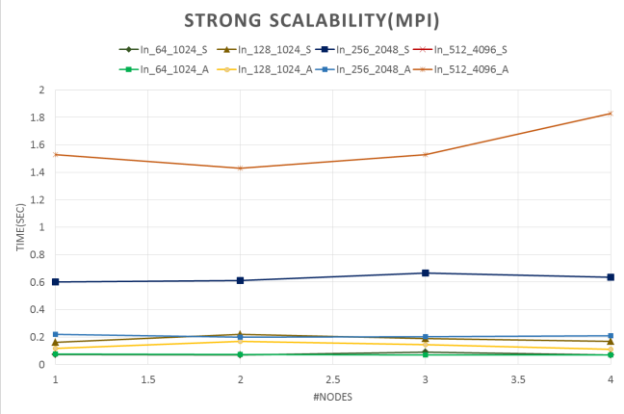




(x-axis: # of Entity cores (# of nodes); y-axis: Execution time (s))

Scalability to number of entity cores of nodes, default source id is 30 for all testcases.





無論是同步還是非同步版本都沒有很好的 strong scalability，由圖上結果顯示可以看出 execution time 是和 ppn 的數量上沒有相關的，比較值得注意的是在使用 vertex 數為 256 以下的三組測資可以看到 synchronous 版本有較好的 performance，而當 vertex 數在提高到 512 時，執行時間就會遠大於 asynchronous 版本，推測是因為使用 blocking 的 MPI_Sendrecv 指令，導致在 vertex 數數量大的時候彼此 rank 間的 communication 會成為 bottleneck，而這個現象在 vertex 數小的時候較不明顯，在之後會有個 rank ID 的時間分析會更能清楚地進行分析。

Time Distribution Analysis

因為cores distribution分析圖上可以看到execution time對於實體cores數上並沒有太大的相關，因此我的 time distribution 主要使用四種 testcase 來進行分析。

(x-axis: type of testcase; y-axis: Time Cost (s))

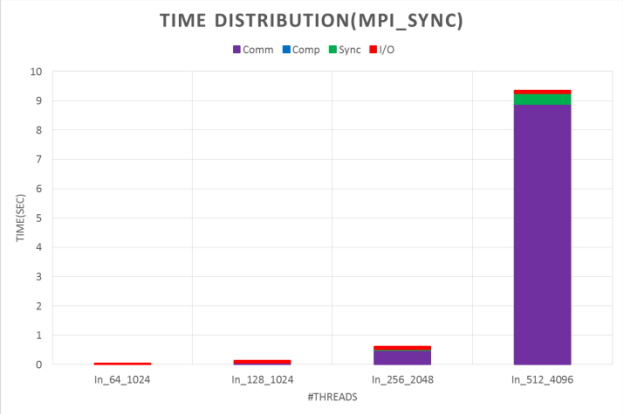
Input parameter:

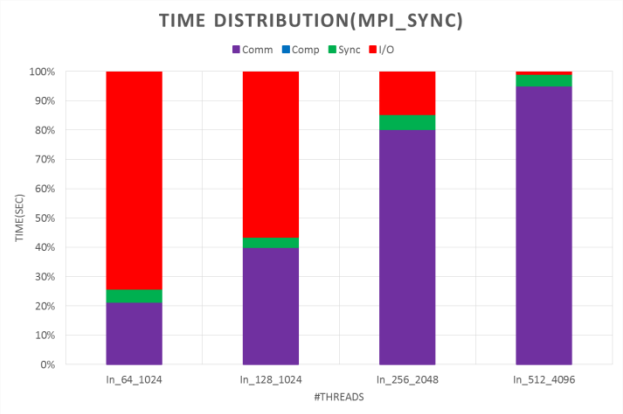
./executable (unused) Input Output_File Source

Partition execution time by :

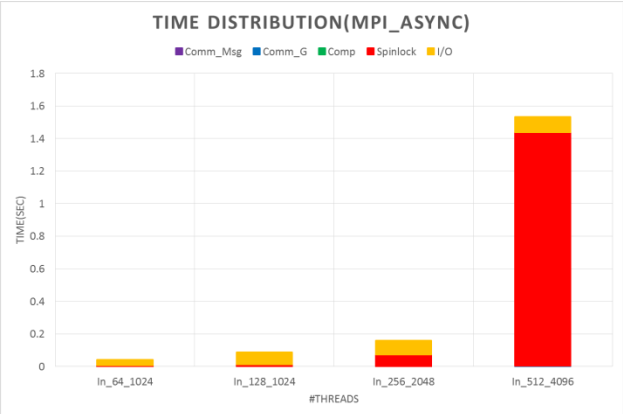
{(communication)x(computation)x(synchronizayion)x(I/O)} for synchronous version

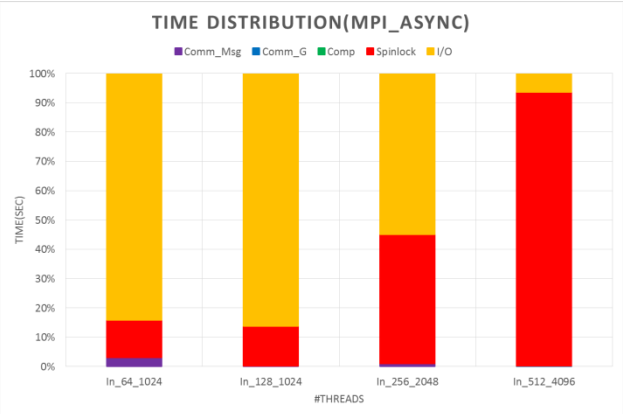
{(MPI_Send/Recv)x(MPI_Gather)x(computation)x(spinlock)x(I/O)} for asynchronous version





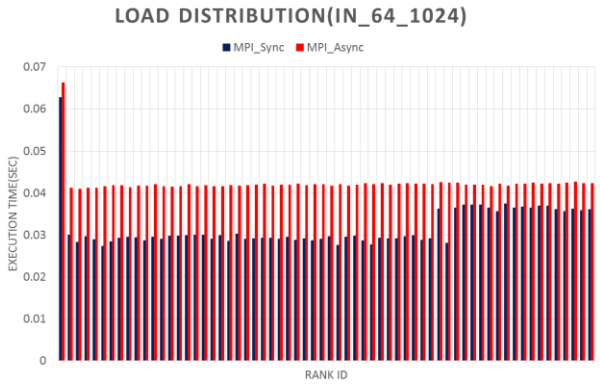
對於 Synchronous 版本由於我的作法都是使用 MPI_Sendrecv 指令來進行鄰點間的溝通，每次的距離更新都會由於 blocking 的 message passing，在前面的溝通還沒完成時後面的就會被迫等待，導致等候時間非常長，當 vertex 數一多起來由 time distribution 的百分比例圖中可以看到 communication time 幾乎佔了所有的 execution time 形成 bottleneck。

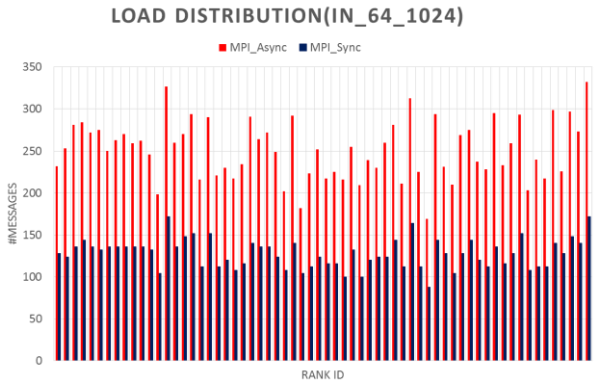


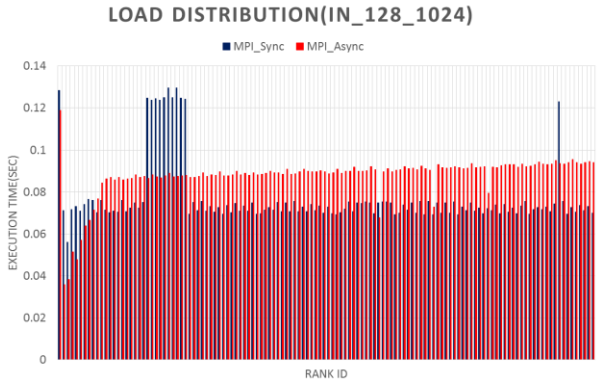


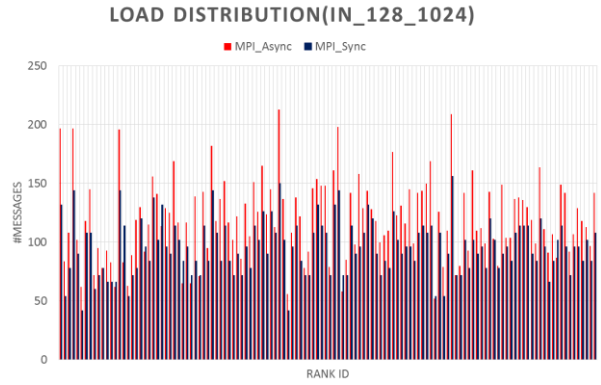
對於 asynchronous 版本雖然隨著 vertex 數的上升並沒有出現如 synchronous 版本中 vertex 數從 256 變到 512 時這麼大的 gap，但仍然會有等候的情況發生，由於我的寫法是在 while 的判斷式內進行MPI_Recv，若接收成功才做接下來的判斷package，因此會有點類似spinlock的概念，當沒有人送出 package 時便會卡在 MPI_Recv 的指令上直到收到 package，我擷取這段時間稱為 spinlock time，由上圖可以知道當 vertex 數一多的時候，I/O time 幾乎僅有微幅的上升，但 spinlock time 卻會因為各 MPI processor 都是非同步的獨立運作，有些 vertex 當前狀態會在沒有運算的時候持續 idle 直到有 package 送往，此時 bottleneck 就是這個 spinlock 所等候的時間。

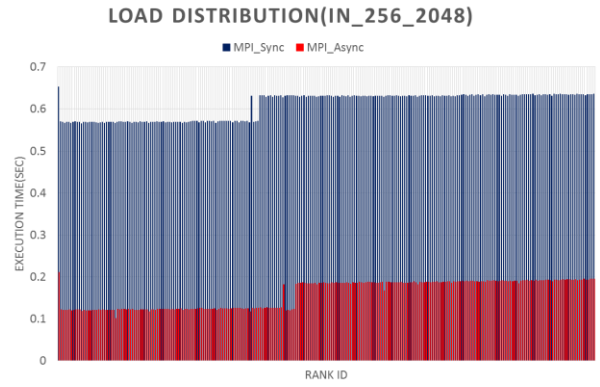
Loading Distribution Analysis
(x-axis : MPI Task ID; y-axis : Execution time(sec)/ # of messages)

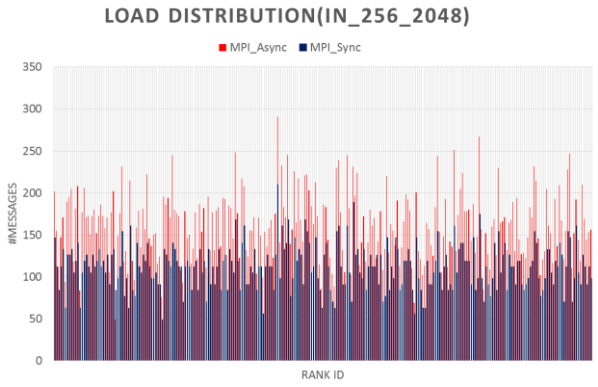


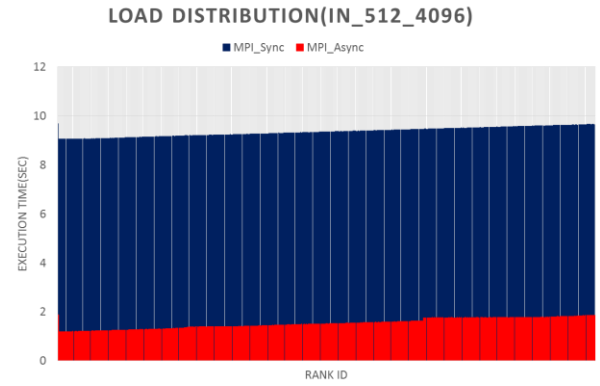


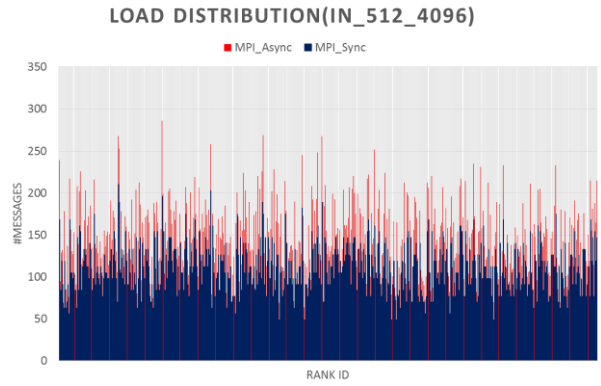








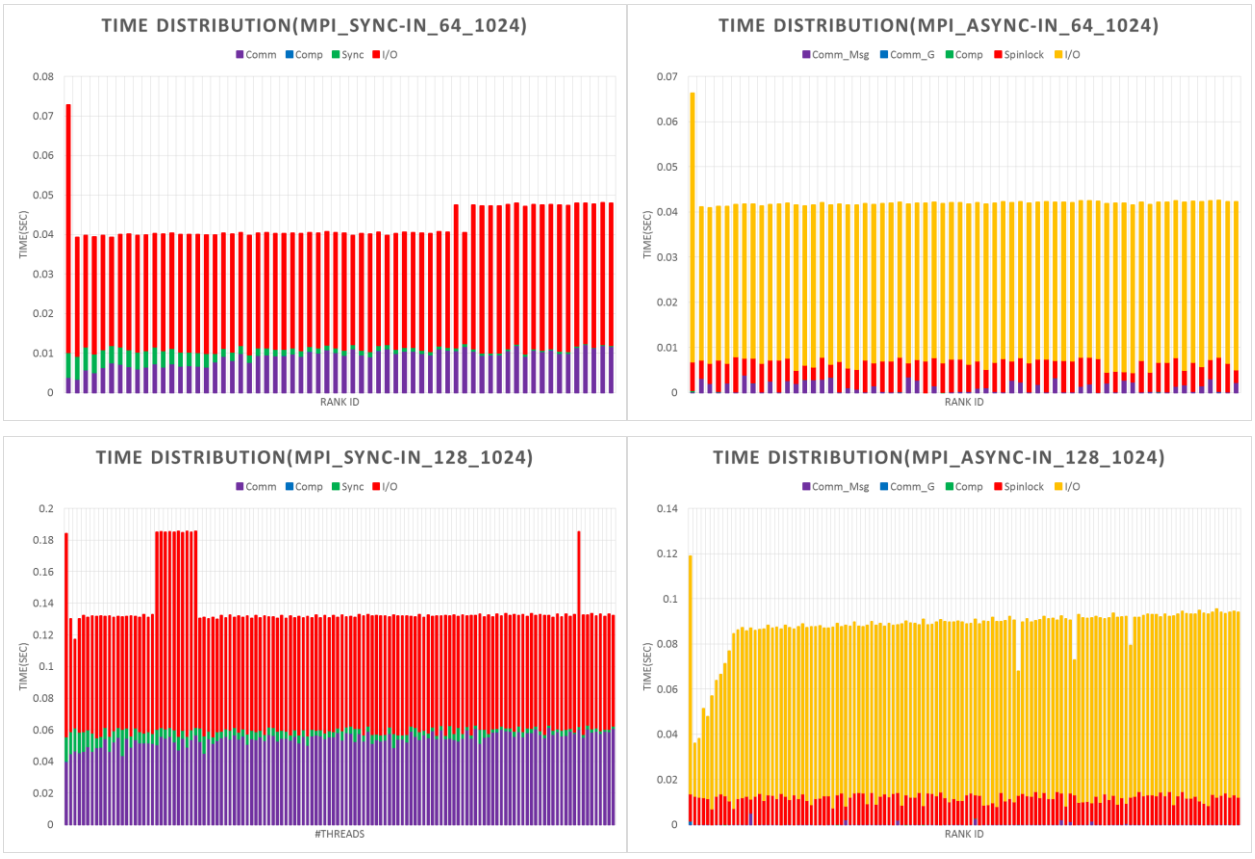


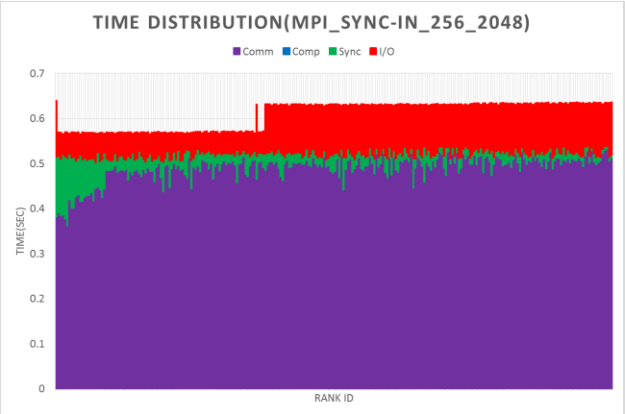


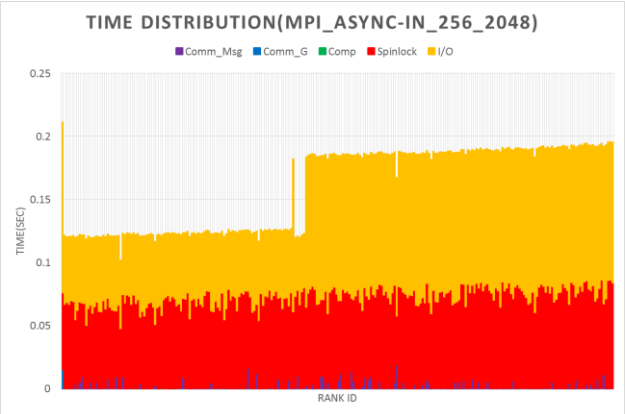
在 loading distribution 分析圖中我們可以看到 rank 0 processor 的 excecution time 特別高，原因是在於我是用 rank0 去進行寫檔，因此大家完成各自的計算結束後，其他 process 會使用 MPI_Gather 指令來把自己的 parent vertex 資訊送往 rank 0 進行資料的蒐集，此外也還有在寫檔時的 I/O cost。
而其他的 execution time 呈現由左到右 execution time 越來越大原因則是因為上述說過我都是使用 blocking 的 message passing 方式，加上我是使用 for loop 來進行一次次的 MPI_Send，因此越前面 rank 的訊息便會越快送出去，而後面 rank 的 processor 就必須等候前面的訊息溝通完成後才能進行傳送，因此產生如圖片上顯示的結果。
而 message passing number 我是在每次有使用 MPI_Recv 接收 message 的時候便+1，我們可以看到由於 asynchronous 版本必須多傳送 dual-ring pass algorithm 所要使用的 token 以及 termination condition 所要判斷的 termination tag，因此對於每個 prcessor 都會接收到比 synchronous 更多的 message。

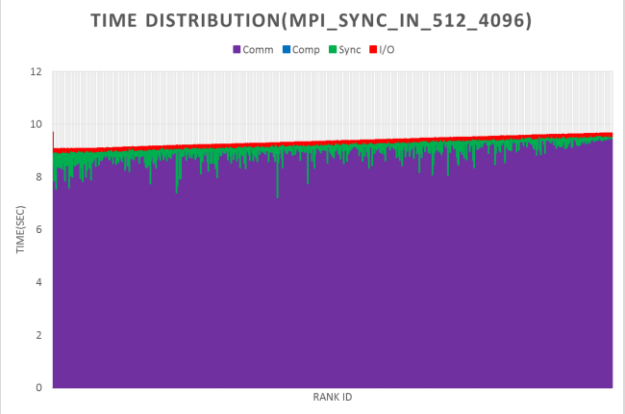
此外為了能更了解實際上每個 MPI task 的實際時間花費，我亦拿 loading distribution 去做 time distribution 的分析。

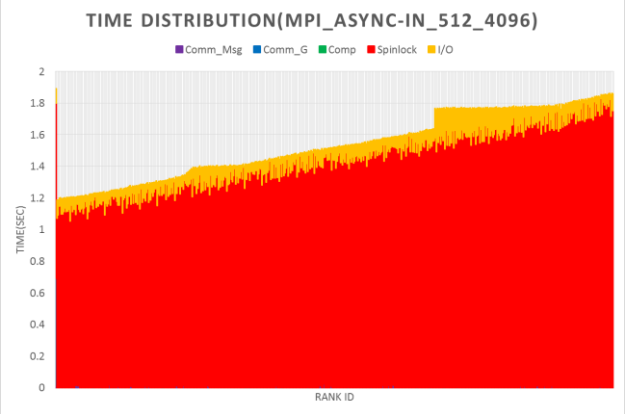
Time Distribution of Loading Distribution Analysis
(x-axis : MPI Task ID; y-axis : Time Cost (sec))
Input parameter:
./executable (unused) Input Output_File Source











我們可以由圖中清楚看到上個分析中 loading distribution 在其他 rank 的部分為什麼會產生如階梯般有浮動的結果，主要原因為因為使用的 testcase 並不是 full-connected graph，因此隨機產生下每個點所連接出去的 path 數不會是相同的，由於我記錄 adjacency matrix 的方式是每個 processor 只記錄自己的鄰點資訊，因此階梯狀原因便是後面的 processor 所代表的 vertex 連接較多的 path，因此他的紀錄鄰點資訊在自己的 adjacency matrix 陣列中的時間便比較多導致這樣的結果，但在 vertex 數一多後，I/O time 影響便逐漸變小，因此我們便可以看到較平滑逐漸上升的 loading distribution 的結果。

根據這個分析我們更可以印證上述分析組合的結果，由圖可以看到 rank 0 由於要進行寫檔的動作因此有最高的 I/O time，還有上述分析的兩種版本的 bottleneck，因為使用 Bellman-Ford algorithm 來當運作原理之下，execution time 受 vertex number 數量影響非常大，只要 vertex 數一多，幾乎時間都是花費在訊息的傳遞或者等候訊息的傳遞上。

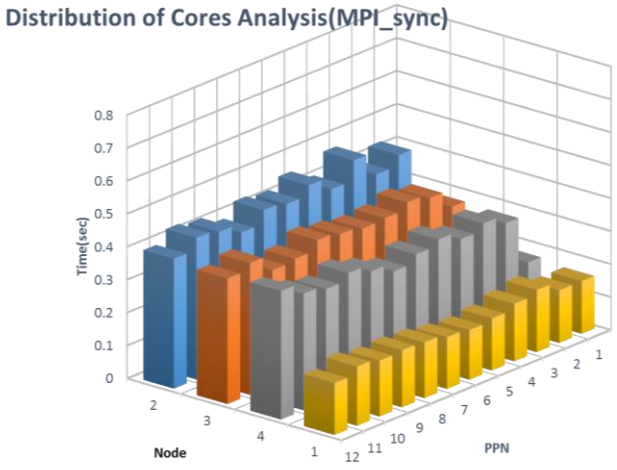
Cores Distribution Analysis

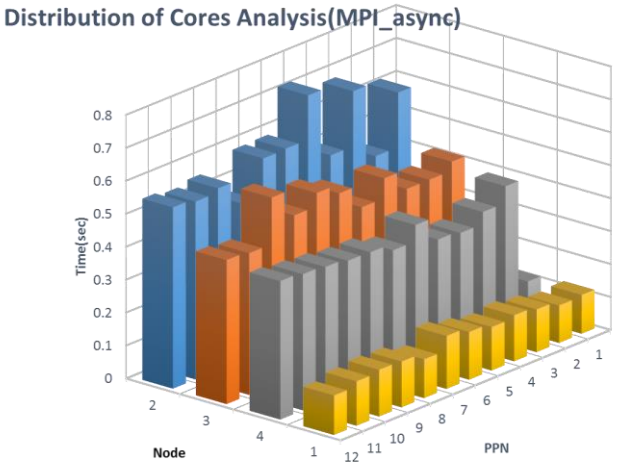
上述因為硬體或者 scheduler 上的限制導致無法正確執行的情況會在 vertex 數較小時而比較不會產生，因此我選用 testcase: In_128_1024，對於所有可能出現的{node, ppn}的 48 種組合進行 execution time 的分析。

(x-axis: #ppn ; y-axis: #Nodes; z-axis: Execution time(sec))

Input parameter:

./executable (unused) In_128_1024 Output_File Source





我們可以看到在 ppn 上對 execution time 並沒有很大的影響，對於 node 的使用上平均來看為 execution time: nodes = 1 < nodes = 4 < nodes = 3 < nodes = 2，nodes = 1 的時候執行時間會相對快很多，推測只要一有跨平台(nodes≧2)的溝通變要付出額外的 message passing cost 所導致，而只要有跨平台的溝通的話，因為皆有該 cost 的負擔，因此則變為越多 nodes 會有越好的 performance。

Difficulty Encounter

在量測使用測資 node 大於 2，ppn 不等於 1 的時候很常出現如下左圖的錯誤訊息：

```
pp18:UCM:727d:5f9dc740: 43 us(43 us):  open_hca: dev open failed for mlx4_0, err=Cannot allocate memory
pp18:SCM:727d:5f9dc740: 59 us(59 us):  open_hca: dev open failed for mlx4_0, err=Cannot allocate memory
pp18:SCM:727d:5f9dc740: 41 us(41 us):  open_hca: dev open failed for mlx4_0, err=Cannot allocate memory
pp18:CMA:727d:5f9dc740: 46 us(46 us):  open_hca: getaddr_netdev ERROR:Cannot assign requested address. Is ib0 c
pp18:CMA:727d:5f9dc740: 32 us(32 us):  open_hca: getaddr_netdev ERROR:No such device. Is ib1 configured?
pp18:SCM:727d:5f9dc740: 34 us(34 us):  open_hca: device mthca0 not found
pp18:SCM:727d:5f9dc740: 33 us(33 us):  open_hca: device mthca0 not found
pp18:SCM:727d:5f9dc740: 34 us(34 us):  open_hca: device ipath0 not found
pp18:SCM:727d:5f9dc740: 34 us(34 us):  open_hca: device ipath0 not found
pp18:SCM:727d:5f9dc740: 34 us(34 us):  open_hca: device ehca0 not found
pp18:CMA:727d:5f9dc740: 32 us(32 us):  open_hca: getaddr_netdev ERROR:No such device. Is eth2 configured?
pp18:UCM:727d:5f9dc740: 41 us(41 us):  open_hca: dev open failed for mlx4_0, err=Cannot allocate memory
pp18:UCM:727d:5f9dc740: 38 us(38 us):  open_hca: dev open failed for mlx4_0, err=Cannot allocate memory
pp18:UCM:727d:5f9dc740: 31 us(31 us):  open_hca: device mthca0 not found
pp18:UCM:727d:5f9dc740: 30 us(30 us):  open_hca: device mthca0 not found
```

```
SessID
------
 28024
 11712
 27313
 14430
 27456
 14626
 27604
```

但有時候又可以得出結果，而且還有根據 session ID 就知道能不能得到正常結果的現象，若看上右圖來舉例，session ID 為 27xxx~28xxx 都是會出現碩誤訊息的，而其餘則是能正常產生結果，不曉得有沒有什麼關係呢?

Conclusion&Experience

Pthread 僅能傳送該一個型別的一個參數進去子 function 中，因此若想傳遞多個參數的話便需要用到 struct，由於非本科系且當初資料結構學得很不紮實，在使用 struct 屢屢碰壁，除了去借了一本資料結構的書來複習，最後還問了大二學弟該怎麼去使用以及也重新複習了”->”的用法(也就是省略(*node).四個符號)，每次寫平程都有新的 C/C++的收穫真不知道該說慶幸還是該慚愧，而 MPI 的部分上因為上課時總覺得何必使用 Dual-Ring Pass Algorithm 總覺得好麻煩，但在做數據蒐集上我們可以清楚看到與不使用這個方法的 synchronous 版本在 performance 上有

的極大的差距，若 vertex 數再更多的話這個差距只會越來越大而不會變小，有實做過後也更能體會整個演算法的原理，非常受用，美中不足的是原本想用更簡潔的 Linked List 來建立 adjacency list，也對整個記憶體的使用較少，但最後只能變成空有想法而無法實作出來的窘境。