Parallel Programming

Homework 3 Report

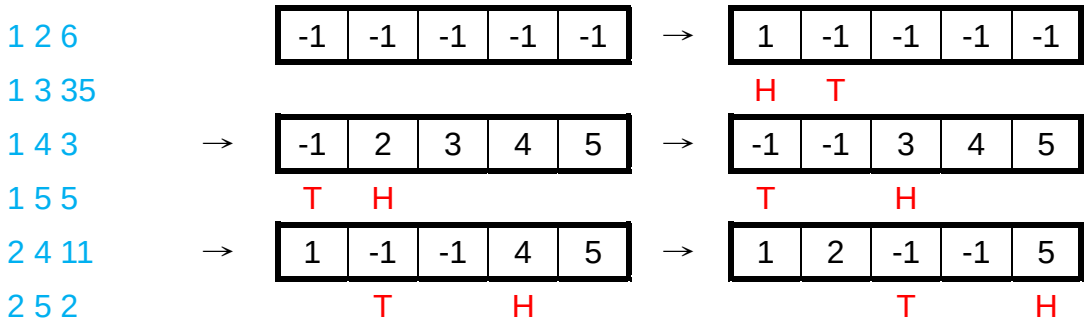
Student ID: 105062635
Name: 吳浩寧

I. DESIGN

1.) SSSP_Pthread

採用老師上課中所介紹的 Moore's Shortest Path Algorithm，使用一個 working queue 來記錄要被計算的點。程式碼中使用 array 來實作，分別用 head、tail 兩個變數指向下個要被計算的點，與要更新的點放入的位置，而 queue 中的初始值和計算完成後的值都為-1。

舉輸入檔案 In_5_7 為例，假設 source vertex 為 1，H、T 分別為 head、tail 變數、則 working queue 陣列中數值的變化如下。



另外我為每一個 vertex 建立一 struct 物件，分別記錄該點的鄰點數量、前行 vertex、最短距離、鄰點陣列、weight 陣列。

每個 thread 從 queue 中取出要計算的點後，即算出每個鄰點的新距離，若比鄰點的原距離短，即更新鄰點的最短距離值，並檢查該點是否 queue 中，若沒有，就將它放到尾端。當所有的點都不需要再更新，意即 queue 中所有的值都為-1，terminate 就變為 1，並中止程式。

2.) SSSP_MPI_sync

每個 MPI process 負責一個 vertex，每一輪接收所有鄰點傳過來的新距離，和自己原距離比較，選出最小值，再傳送距離給所有鄰點，每輪計算完後，會用 MPI_reduce 判斷所有的點是否在這輪有更新距離，若全無則送出終止信號。

3.) SSSP_MPI_asyn

與同步版本不同之處在於每個 MPI process 每輪只接收一個鄰點傳來的新距離，並傳送有更新的距離給對應鄰點，且每輪計算後沒有同步的 collective function call。終止判斷則用要求的 Dual-Pass Ring termination algorithm。

2. PERFORMANCE ANALYSIS

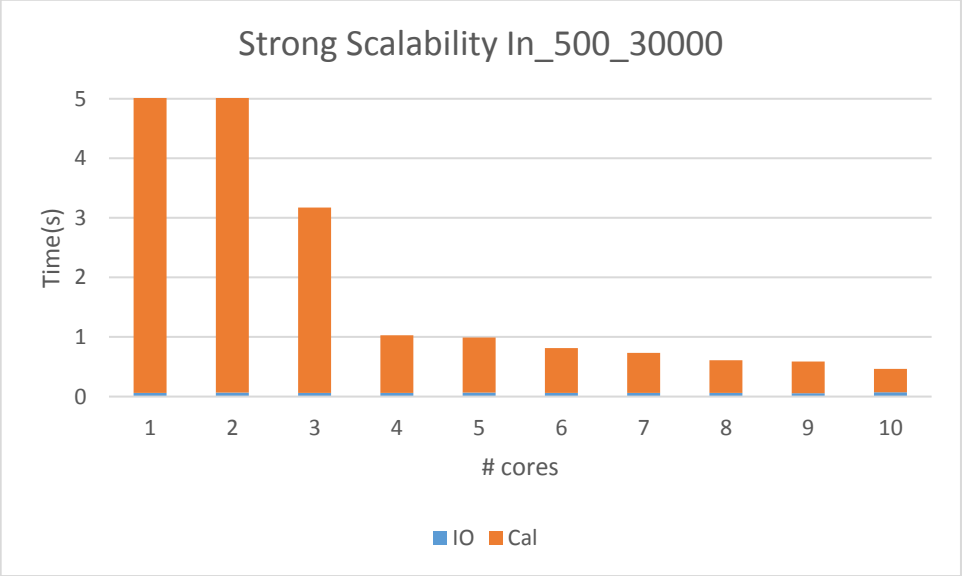
使用的 Input 皆為 graph_ge 產生的 connected graph，max weight 皆設為 10000，使用各種 edge/vertex 比例的測資，比如 edge 比重較低的 In_1000_10000 和接近 complete graph 的 In_250_30000。

Strong Scalibility

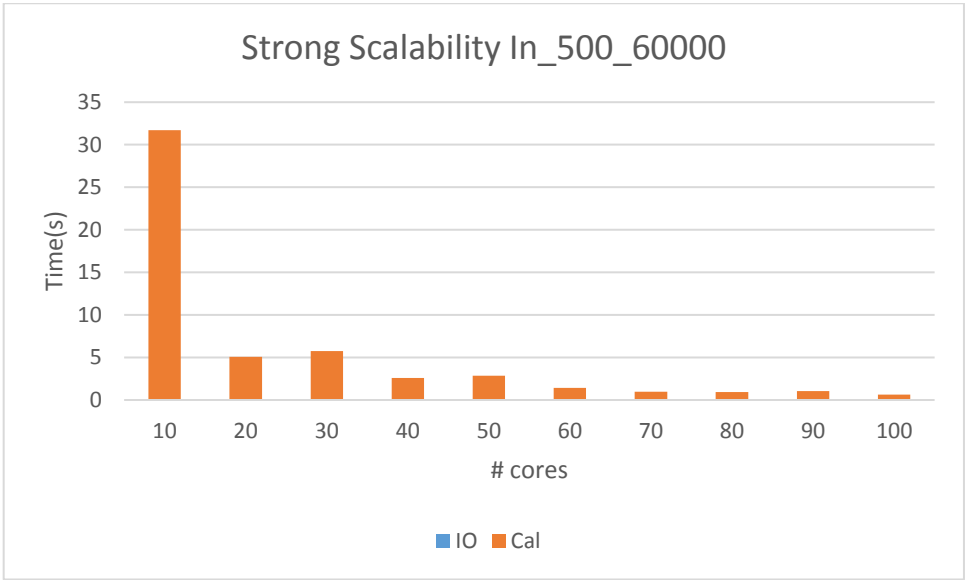
MPI 兩個版本 cores 的排列組合如下：

# nodes	1	1	2	2	3	3	4	4
ppn	6	12	9	12	10	12	10	12
# cores	6	12	18	24	30	36	40	48

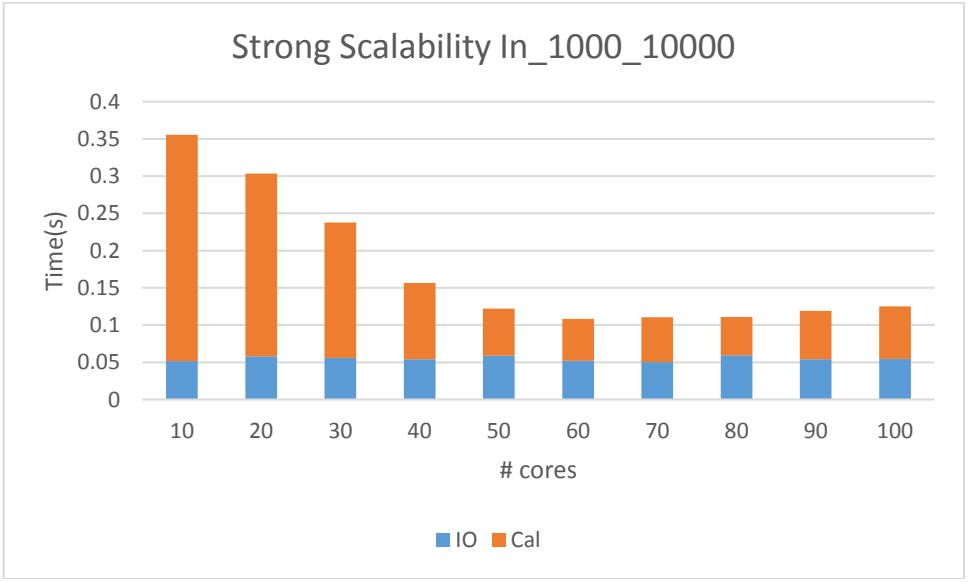
Pthread Version



※t(1)= 264.8201, t(2)= 167.3773


Strong Scalability In_500_60000
35
30
25
20
15
10
5
0
Time(s)
10
20
30
40
50
60
70
80
90
100
cores
IO
Cal


Strong Scalability In_250_30000
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Time(s)
10
20
30
40
50
60
70
80
90
100
cores
IO
Cal

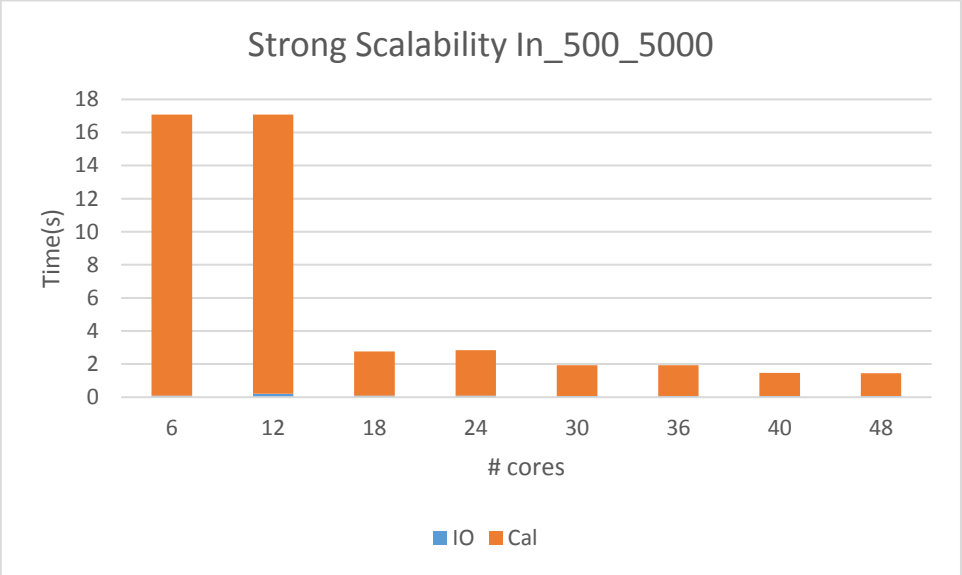

Strong Scalability In_1000_10000
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Time(s)
10
20
30
40
50
60
70
80
90
100
cores
IO
Cal

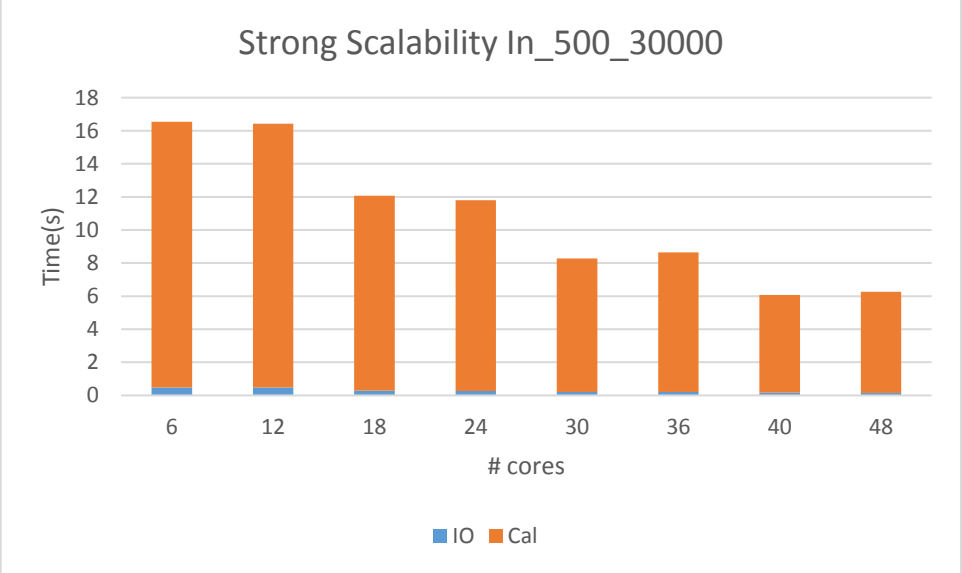
經我實驗發現，pthread 的運行的時間非常短、浮動非常大，因此不易觀察出其特性。比較 In_500_30000 和 In_500_60000，一開始都有急遽下降的趨勢，然後幅度瞬間緩和，而這個趨勢經觀察推測會隨 edge 比重增加而延後。

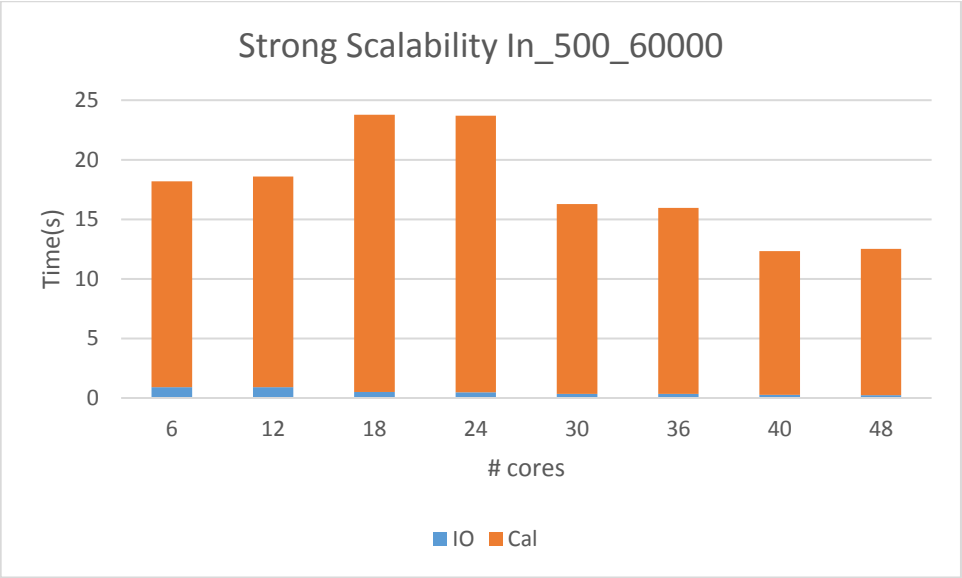
另外，從各種組合可以看出，所花的時間主要受到 edge 數目的影響，一來每個 thread 計算量和自己的鄰點數成正比，二來因為我的終止條件要遍歷整個 working queue，而其大小和 edge 數量又成正相關。

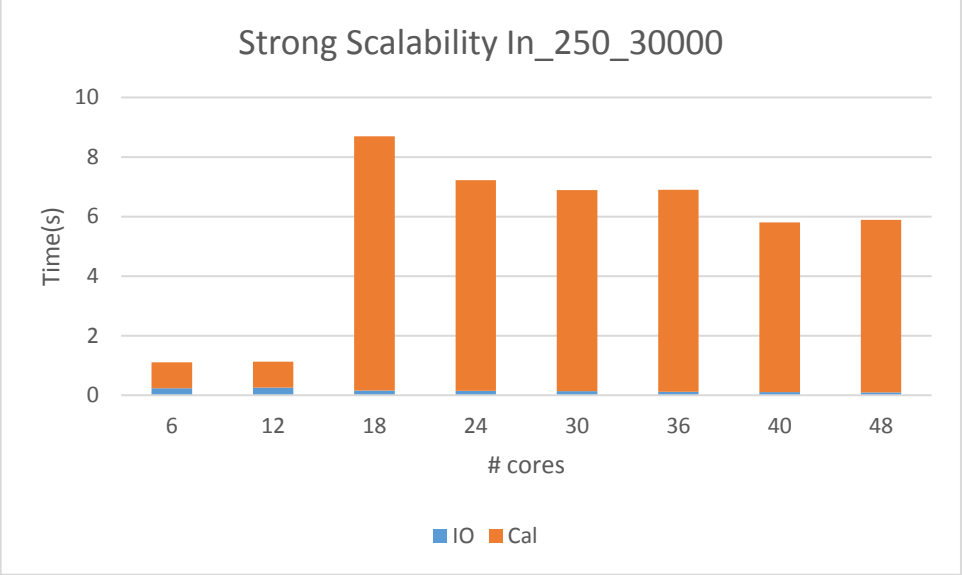
比較意外的是在 In_500_30000 和 In_250_30000 圖表中，使用 20 個 threads 實所我的時間都最多，不曉得原因為何。

MPI_sync Version



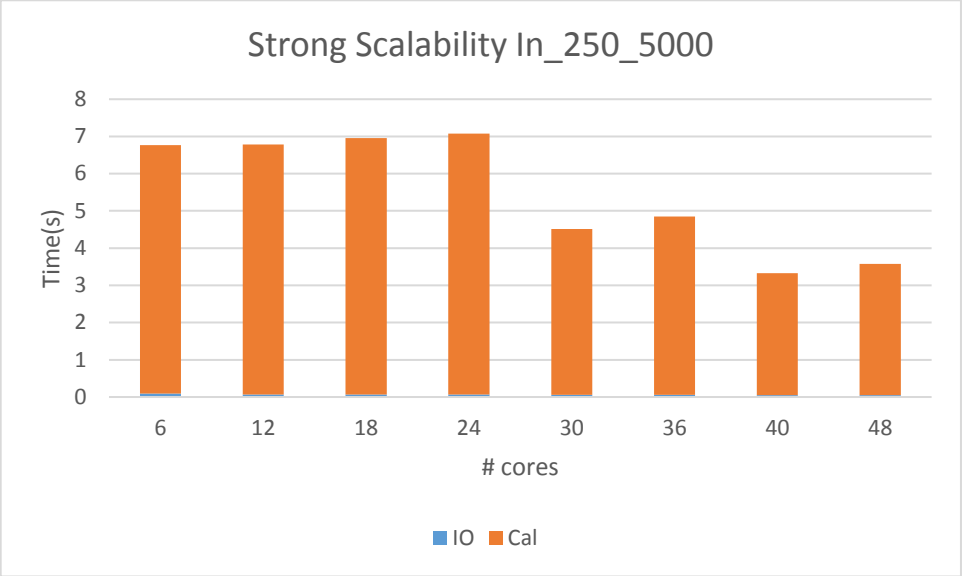


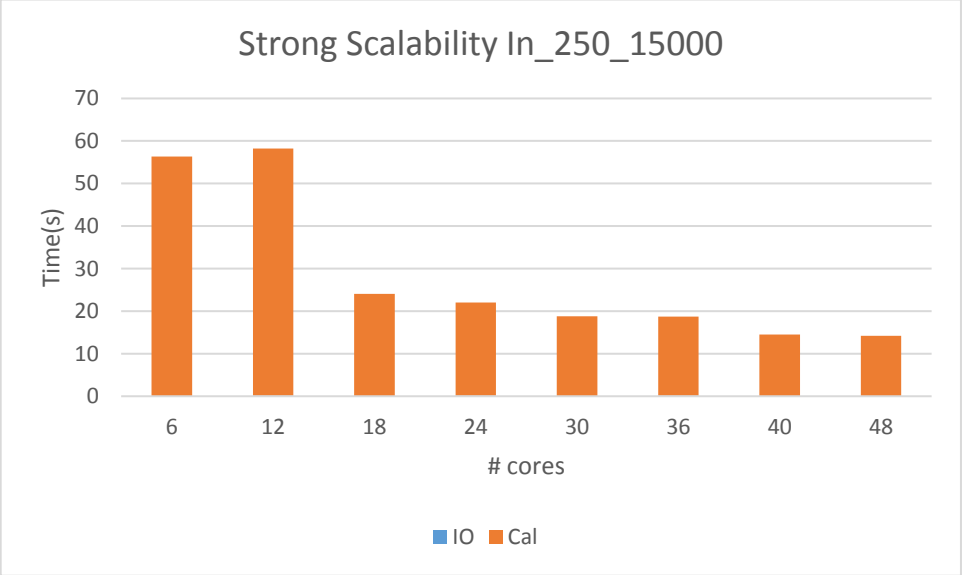


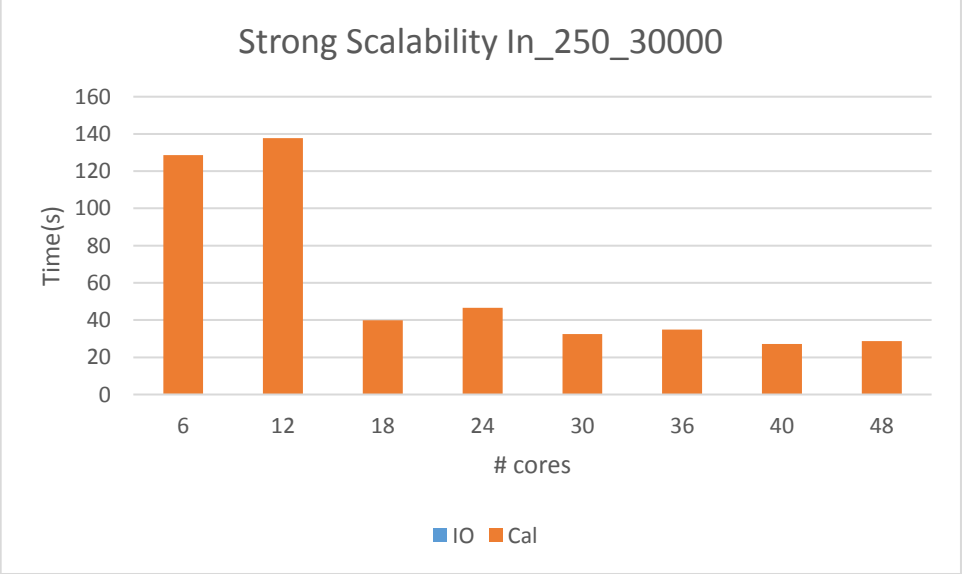


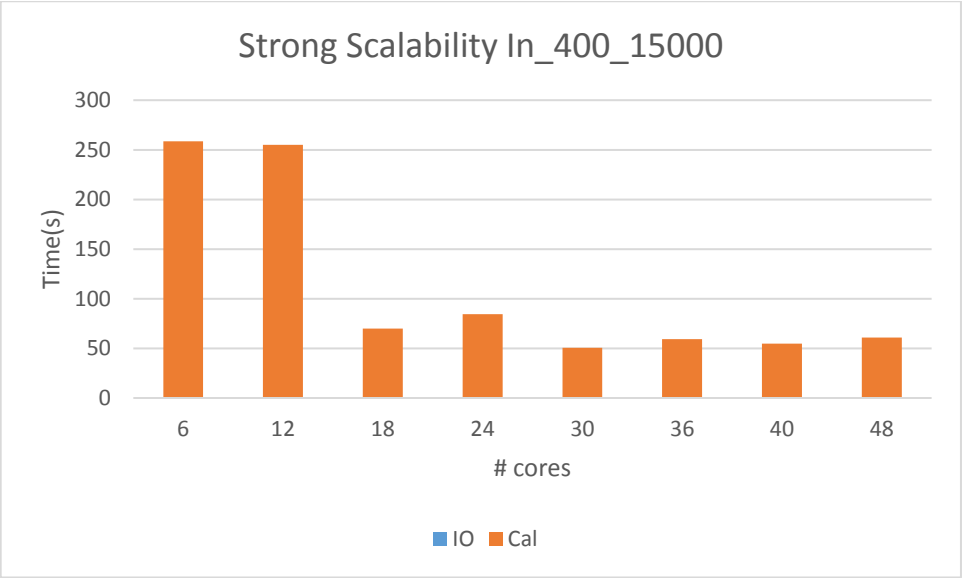
從 vertex=500 的三個圖表來看，可以發現 edge 越多，node=1 執行的速度幾乎不受影響，使用超過 1 個 node 的執行時間則越來越慢，但成線性關係。推測可能是越多 edge 產生 cross-node 的 communication 的機率就越高，synchronous 版本中，最多鄰邊的 vertex 會成為 overhead，因而產生此現象，令我猜測前面 pthread 的現象是否也因為使用到不同 CPU core 的關係。

MPI_async Version







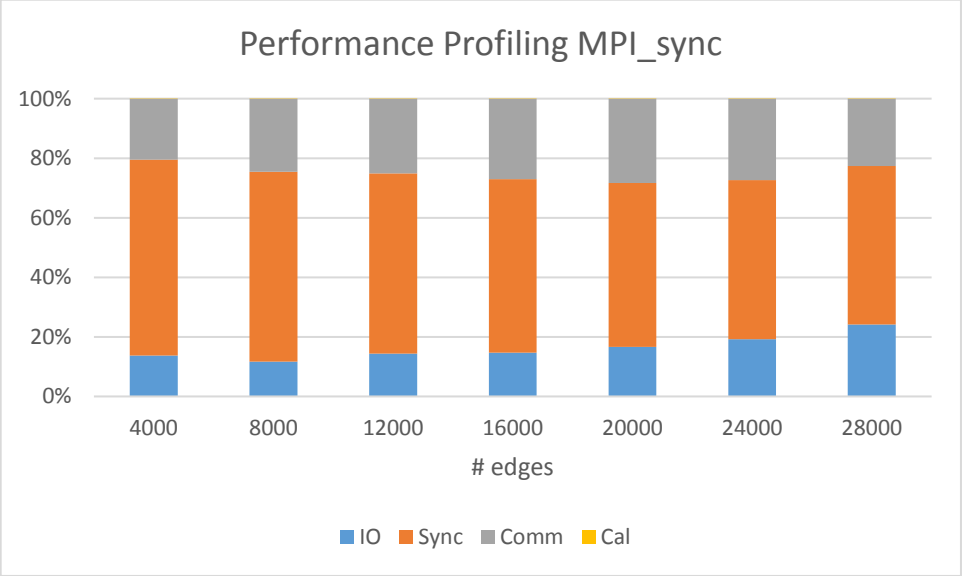


令人訝異的是，MPI_async 的速度是最慢的，而且圖表也比較看不出有線性的加速現象，可能是由於傳送 token 占 communication 的比重太高導致，MPI_sync 版本雖然有同步點，但一次就收集所有 process 的資訊，可能因此較快。

Performance Profiling

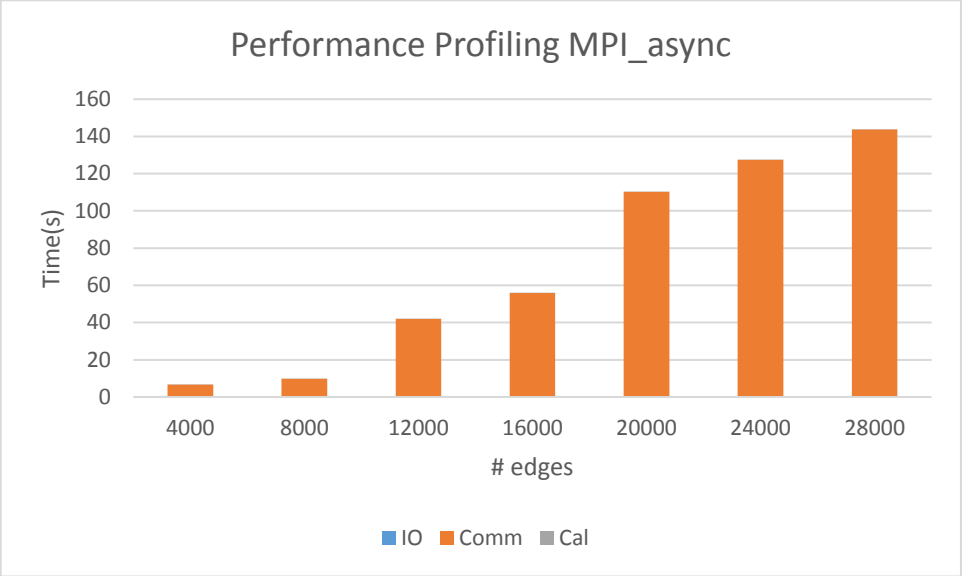
使用 250 個 vertex 的測資；nodes=1；ppn=12，數值皆採用所有 process 的平均值。

MPI_sync Version



MPI_sync 版本中，隨著 edge 數增加，花在 IO 讀檔上佔的比例增加，花在等待同步上的時間，因為鄰邊最多的 vertex 會成為瓶頸，而鄰邊最多都是 vertex 數減 1，因此從數值上來看幾乎都差不多。

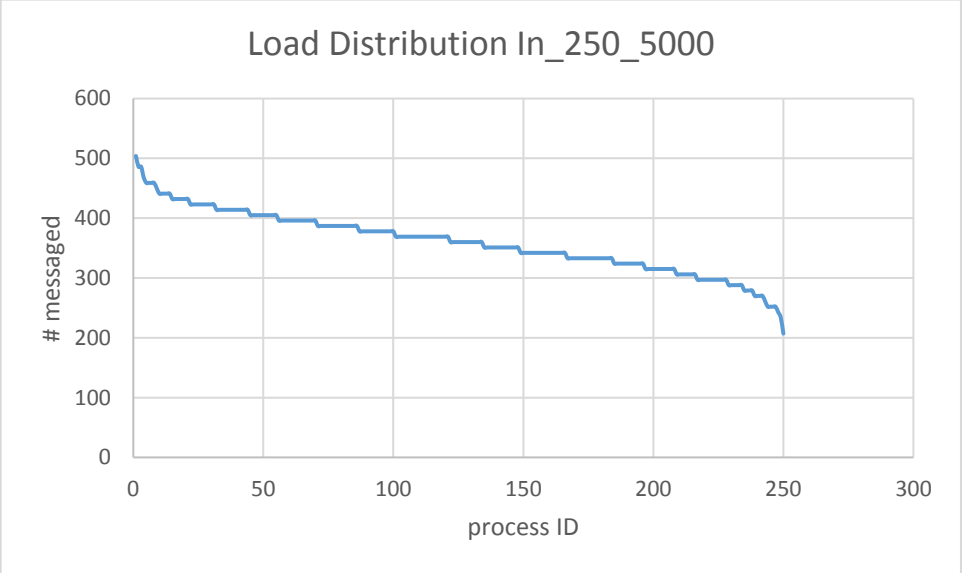
MPI_async Version

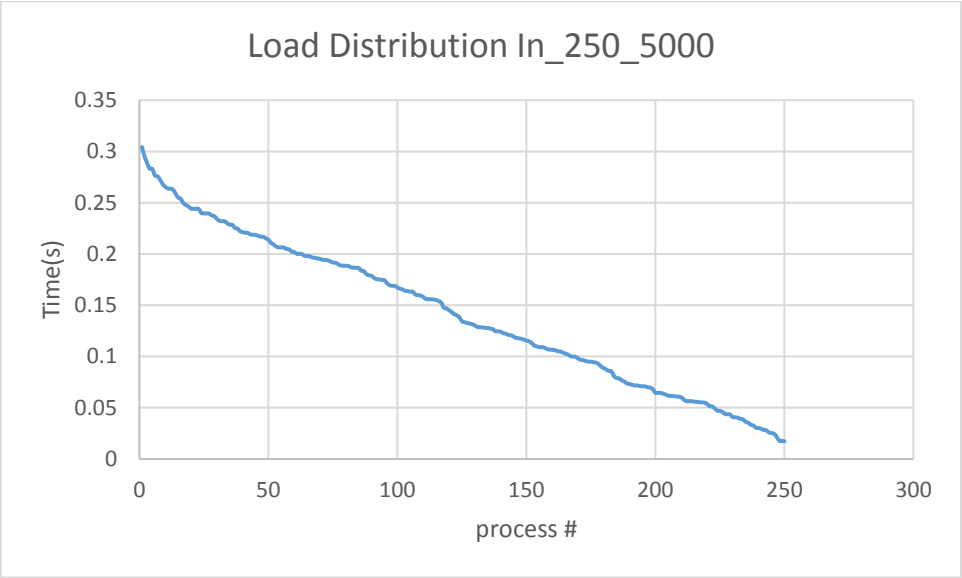


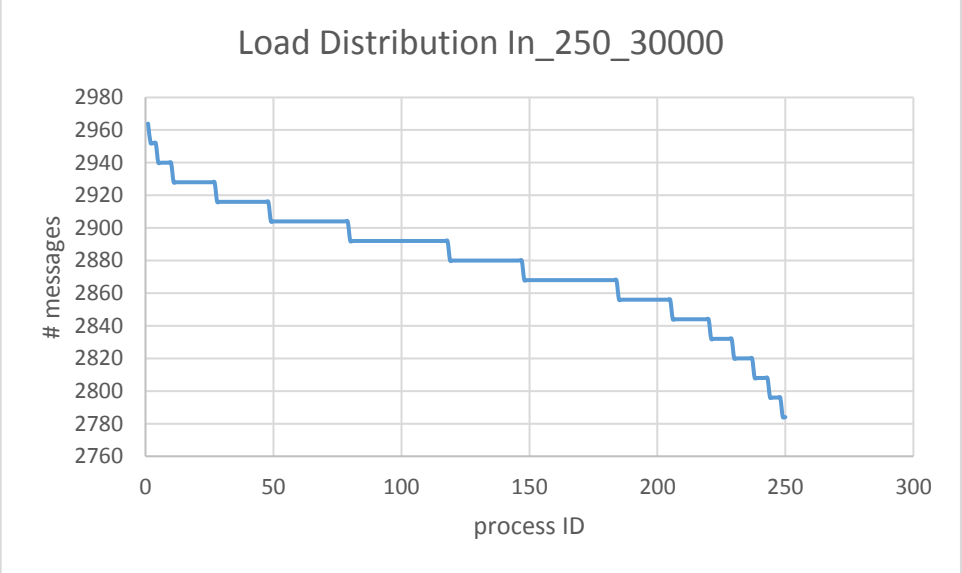
MPI_async 版本由於花的時間太久，communication、IO、計算時間數量級差距太大，所以比較看不出時間分布的比例。

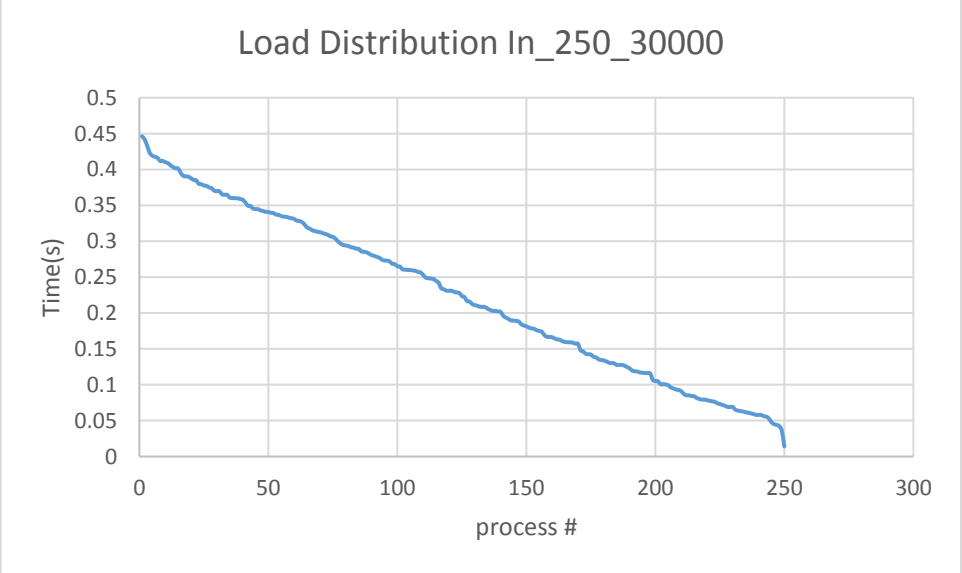
Load Distribution

MPI_sync 版本中時間計算為扣掉 IO、等待同步時間



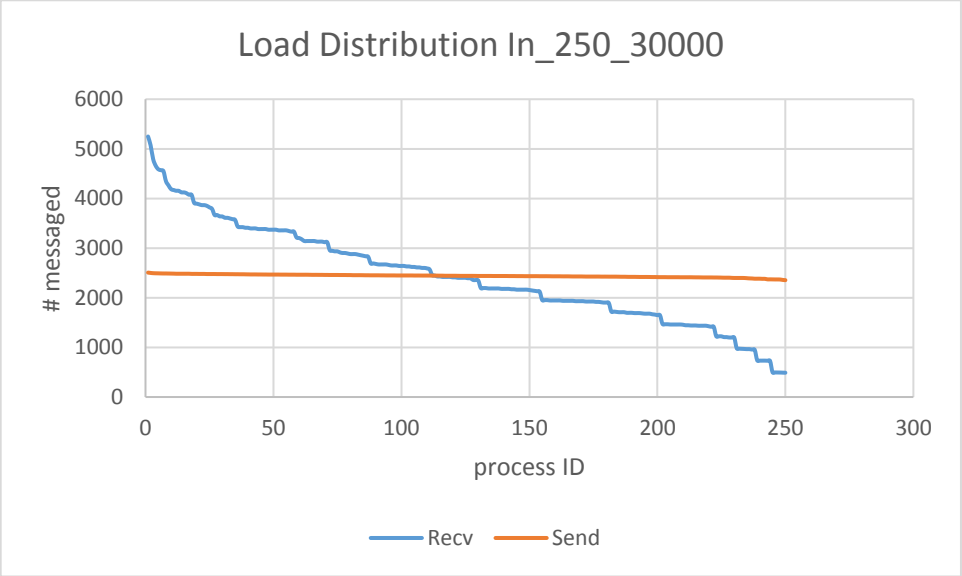


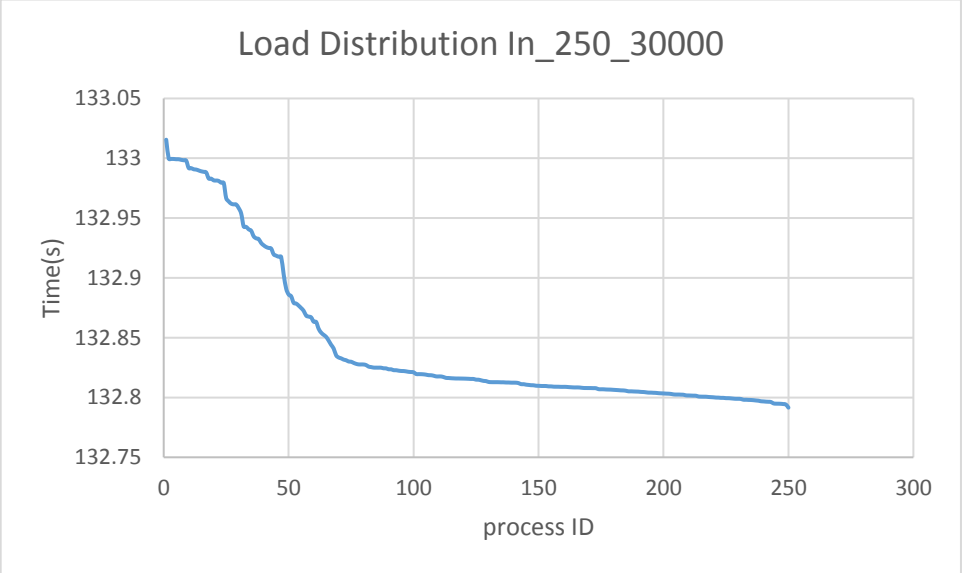


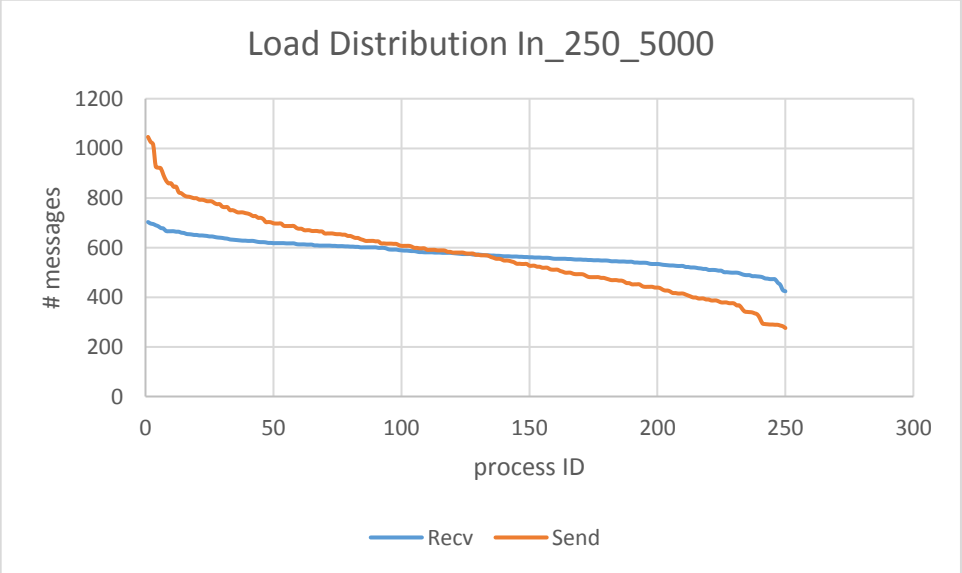


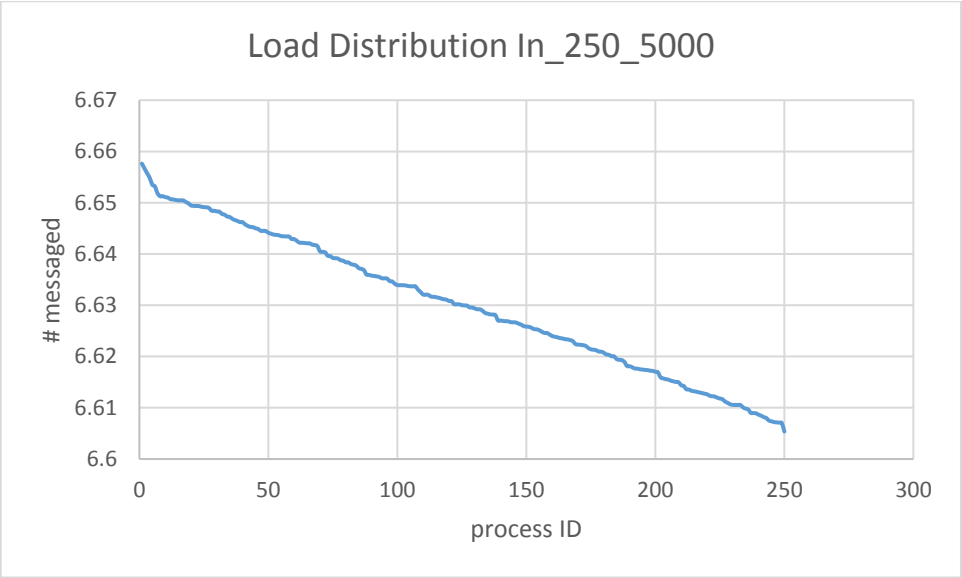
由於 MPI_sync 版本各個 process 傳送、接收的訊息量都一樣，且都等於鄰邊數量，因此合在一起繪製成圖表。

MPI_async


Load Distribution In_250_30000
6000
5000
4000
3000
2000
1000
0
messaged
0
50
100
150
200
250
300
process ID
Recv
Send


Load Distribution In_250_30000
133.05
133
132.95
132.9
132.85
132.8
132.75
Time(s)
0
50
100
150
200
250
300
process ID


Load Distribution In_250_5000
1200
1000
800
600
400
200
0
messages
0
50
100
150
200
250
300
process ID
Recv
Send



原本猜測 MPI_sync 版本的時間分布，會比 MPI_async 版本大很多，因為前者每個 process 的 loading 就是鄰邊數量，而後者做完計算繼續便會繼續等待新訊息或終止信號。從結果看來 MPI_sync 版本只稍微大了一點，可能由於運作時間短很多，造成差異並不顯著，而分布圖大致呈線性

比較值得觀察的是 MPI_async 版本在 edge 比重高的時候，每個點都要傳資料給其他所有鄰點，使得 send message 分布很平均，反之當 edge 比重低的時候，send message 分布差距甚至大於 receive message，因為 edge 數少的時候需要更新的次數也少很多。

3. EXPERIENCE

這次作業讓我更了解 pthread 的優勢，由於它屬於比較低階的 API，加上使用 shared memory 方式共享資料，使得計算的速度快了非常多，但是在寫程式時就得額外注意是否有同時存取一個變數，進而產生 race condition，舉這次作業為例，就是在修改 working queue 內容，和 struct vertex 中 MIN distance 時，須加上 mutex，以確保同一時間只有一個 thread 修改該變數，判斷終止情況時，也必須將整個 working queue lock 住，檢查每個項目的值，是主要的 overhead 來源，而決定 working queue 的大小也試了很久，太小很容易發生因為一個 thread 計算過久，導致 tail 追上 head 而產生錯誤，太大則會使 complexity 增加。此外也學會用<time.h>中的 struct timespec 來取得較精準的貓 wall time。