Parallel Praogramming HW3 Report

102062209 邱政凱

1. Design

a. Pthread

在 Pthread 的版本中，我實作的是類似於講義第七章上所描述的 Moore's algorithm。Main thread 讀取完檔案之後開啟指定數量的 thread，每個 thread 個別檢查共用的 vertex queue 裡面是否有 vertex，有的話就把他 pop 出來，並且由該 thread 負責更新該輪該 vertex 的 outgoing neighbor 的距離。接著該 thread 再把距離有被更新的 vertex 重新放回 queue 的後面，等待其他 thread 把它取出，並且同時把被更新的 vertex 的父節點也記錄起來以供 output 使用。

在以上的過程中，對於 queue 的操作全部都必須取得 queue 的 mutex lock 之後才能進行。而在這樣的過程中整個 graph 的權重、距離、父節點、vertex queue 等等資訊是放在 heap 中由所有 thread share。而中止條件就是所有的 thread 都發現 vertex queue 裡面已經沒有元素可以取出的時候。關鍵演算法大致如下：

```
find_path(){

    while(1){

        while(vertex!=null){

            {   // critical section
                get a vertex from queue
            }
            flag[this_thread] = 1;
            if(flags of all threads are 1, then return)
        }

        flag[this_thread]=0;

        {
            for(){
            calculate distance

                { // critical
                update distance from this vertex
                put the vertex into queue
                }
            }
        }
    }

}
```

我選擇用這種方式實作 Pthread 的版本的原因是，SSSP 本身並不是個好

平行化的問題，用這種 Moore’s algorithm 的方式，所有 thread 之間只需要共用一個 queue，每回合取出 queue 中的 vertex 之後計算的部分都是各個 process 間獨立的，每個 thread 之間的相依性很低，不太會有互相等待的問題。，它可以達到較高的平行度，需要處理 race condition 的地方也只有在 queue 的操作而已，因此我認為是個有效率的演算法。

b_1. MPI_Synchronous

在同步的 MPI 版本中，首先我讓 master process(rank = 0)進行讀檔，接著用 MPI_Bcast 把 Intput 資訊廣播到所有的 process 上(傳送所有 edge 的起終點跟權重，避免用傳送二維的 weight array 以免節點數多的稀疏圖會有太多沒必要的資訊需要傳送)。Rank 為 x 的 process 在這個演算法中就負責第 x+1 個 vertex。每個 Process 收到 Input 之後先初始化對應 vertex 的 in 和 out vertices list。在 SSSP 演算法本體的部分，每個 Process 在每個 iteration 先把自己當前從原點到自己的距離用加上對應 outgoing edge 的權重的值用 MPI_Isend 送給所有的 outgoing neighbor，接著便進行(# fan in edges)次的 MPI_Recv，接收所有送給該 vertex(process)的距離(message)。這部分為了提升效率，MPI_Recv 我使用的 source 是 MPI_ANY_SOURCE，由發送方在訊息中再另外包一個參數告知接收方這個訊息是誰送的(避免固定順序的 recv)。

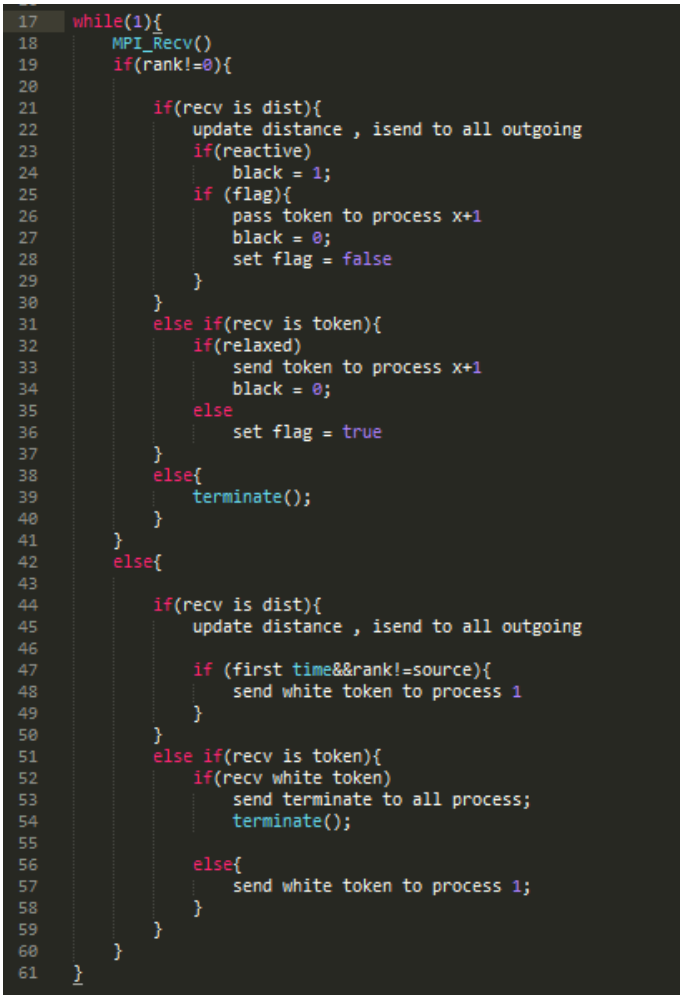
每次 recv 到一個訊息就確認是否新的距離比目前的還短，是的話就更新當前距離，並同時記錄這個距離的更新是由誰送來的(以供 output 使用)。最後在每個回合結束前用 MPI_Allgather 收集 termination condition。如果某 process 在該 iteration 中距離有被更新，就把 flag 設成 true，否則設成 false。

而 global termination 就是在每回合結束前把所有 process 的 flag 收集起來檢查是否所有 flag 都等於 false，否的話全部 Procss 都進入下一輪，是的話就跳出迴圈。最後每個 process 把最晚更新道的父節點資訊 send 給 master process，由 master process 統一輸出成指定格式。關鍵演算法大致如下：

```
while(1){

    termination = 1;
    updateFlag = 0;

    for(i=0;i<outCount;i++){

        MPI_Isend(&outBuffer[i*2],2,MPI_INT,outList[i],0,MPI_COMM_WORLD,&reqs[i]);
    }
    for(i=0;i<inCount;i++){
        MPI_Recv(&recvBuffer[i*2],2,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
        if(recvBuffer[i*2]<m_dist){
            m_dist = recvBuffer[i*2];
            father = recvBuffer[i*2+1];
            updateFlag = 1;
        }
    }



    for(i=0;i<outCount;i++){
        outBuffer[i*2 +0] = m_dist+outWeight[i];
    }

    MPI_Allgather(&updateFlag,1,MPI_INT,allFlags,1,MPI_INT,MPI_COMM_WORLD);
    for(i=0;i<process_num;i++){
        if(allFlags[i]==1){
            termination = 0;
            break;
        }
    }
    if(termination)
        break;

}
```

B_2. MPI_Asyncronous

在非同步的 MPI 版本中，除了核心 SSSP 演算法的部分外，全部都跟同步版本一樣。而在核心關鍵法的部分，因為少掉了 synchronous point，所以每個 process 必須要有辦法個別行動收發訊息而不會發生死結的狀況，這部分應該是非同步版本最需要考慮的部分。因此我使用了以下的架構：



```
while(1){
    MPI_Recv()
    if(rank!=0){

        if(recv is dist){
            update distance , isend to all outgoing
            if(reactive)
                black = 1;
            if (flag){
                pass token to process x+1
                black = 0;
                set flag = false
            }
        }
        else if(recv is token){
            if(relaxed)
                send token to process x+1
                black = 0;
            else
                set flag = true
        }
        else{
            terminate();
        }
    }
    else{

        if(recv is dist){
            update distance , isend to all outgoing

            if (first time&&rank!=source){
                send white token to process 1
            }
        }
        else if(recv is token){
            if(recv white token)
                send terminate to all process;
                terminate();

            else{
                send white token to process 1;
            }
        }
    }
}
```

也就是說，不像同步版本中所有 Process 的行為都一樣，rank = 0 的 process 跟其他 rank 的 process 的行動是不一樣的。在 rank!=0 的 process 中，先用 recv 等待訊息進入，等收到訊息後確認訊息的種類，共分為(1)距離的更新、(2)傳 token、(3)傳 global termincation condition。

若是收到距離，則確認收到的距離是否比目前此 process 儲存的距離還小，如果是，則更新距離，並且發送距離的更新訊息給所有 outgoing neighbor(如果此 process 有任何 outgoing neighbor 的 rank 比自己還小，進行完此步驟後必須把 black flag 設成 1)。

如果是收到 token，則確認自己是否進入 local termination(我的 local termination 定義為：此 vertex 曾經被 relax，也就是從 src 到它的距離不為無限)。如果已進入 local termination，則確認 black flag，為 0 則把收到的 token value 直接傳給 Process rank (x+1)%(num_process)，為 1 則傳 1 給 Process rank (x+1)%(num_process)。如果還未進入 Local termination，則為了避免距離根本還沒算出來就太提早被誤判 global termination(畢竟目前原點根本還無法走到此 vertex，照理說它不可能已經結束)，所以就先保留收到的 token，等到此 process 被 relax 之後，才依照上面的規則把 token 繼續傳下去。如果收到由 rank=0 送來的 global termination 訊息，則跳出迴圈。

在 rank=0 的 process 中，同樣先用 recv 確認收到的訊息，若是距離則行為跟其他 process 一樣。若 rank=0 的 vertex 是第一次被 relax，則同時發送一個 token 給 process 1(也是整個演算法的第一個 token)。

若是收到 token，則確認值為 1 或 0，為 1 則傳 value 為 0 的 token 給 process 1，若是 0，則發送 global termination 訊息給所有其他 process，自己也跳出迴圈。

注意在以上的這個架構中，在進入迴圈前必須由 source vertex 先發送距離更新的訊息給他的 outgoing neighbor，避免一開始所有 process 都 deadlock 在一開始的 MPI_Recv 裡。

整體看來，同步版本的優勢在於所有 Process 間的行為是整齊一致的，code 比較好寫，也因為有 synchronization point 的關係，不需要特別實作 termination detection algorithm。缺點是需要有 synchonization 的 Overhead，以及較大的 message 數量(因為每回合所有 process 為了維持一致的行為都要收發訊息)，我認為比較適合較為密集(edge 較多)的 graph。

而非同步版本的優勢則在於所有 process 間的行為相依性降低，且因為行為不需要一致，可以不用發送太多不必要的訊息。缺點我認為是 code 架構比較難規劃，且須要維護額外的 termination detection algorithm，所以 process 每次收到新的訊息後的運算量也比 synchronous 大。我認為比較適合較為稀疏(edge 較少)的 graph。

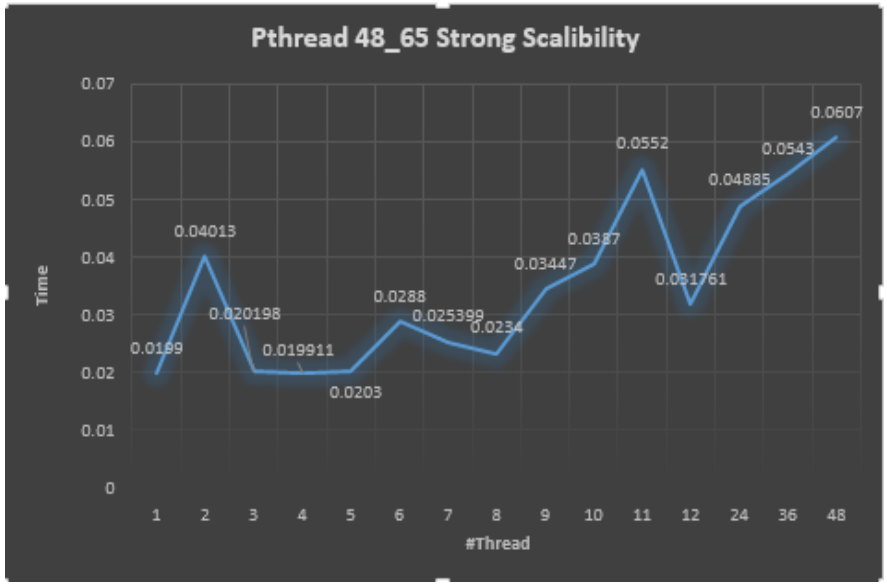
---

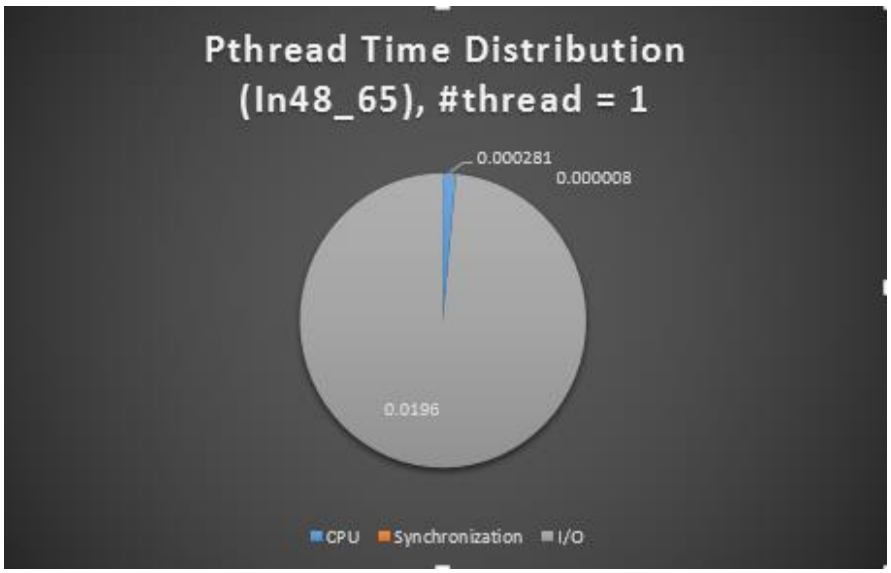
2. Experiment & Analysis

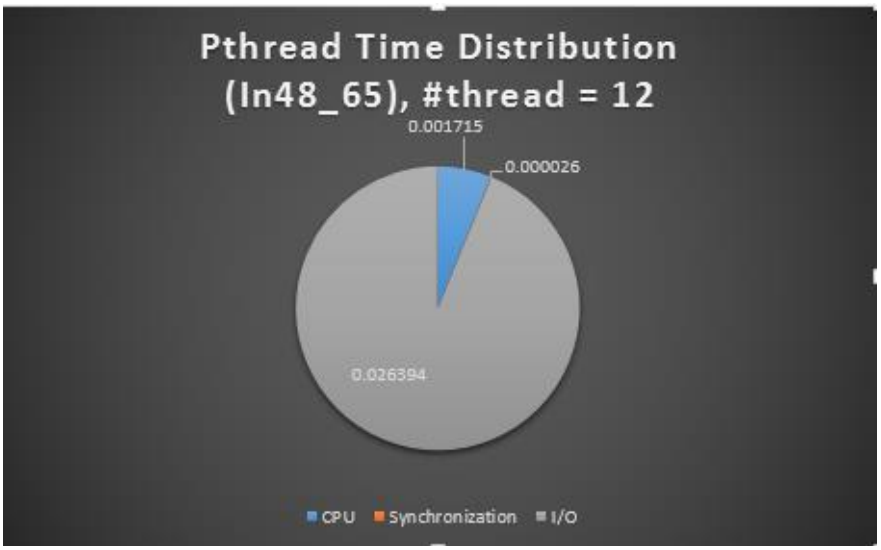
我使用的環境是課程提供的 cluster，因此規格就跟助教公布的一樣。

以下的 圖表內 X_Y 代表 Vertex 數和 edge 數(例:48_65 = 48 個 vertex 和 65 個 Edge)。在以下實驗中，各個版本都提出了 4 種 input 的 Strong scalability 的實驗結果，並且挑選其中幾種 input 做 time distribution 的分析。也因為 strong scalability 和 time distribution 一起分析會比較好理解，所以我把(a)(b)兩個 section 混在一起寫。

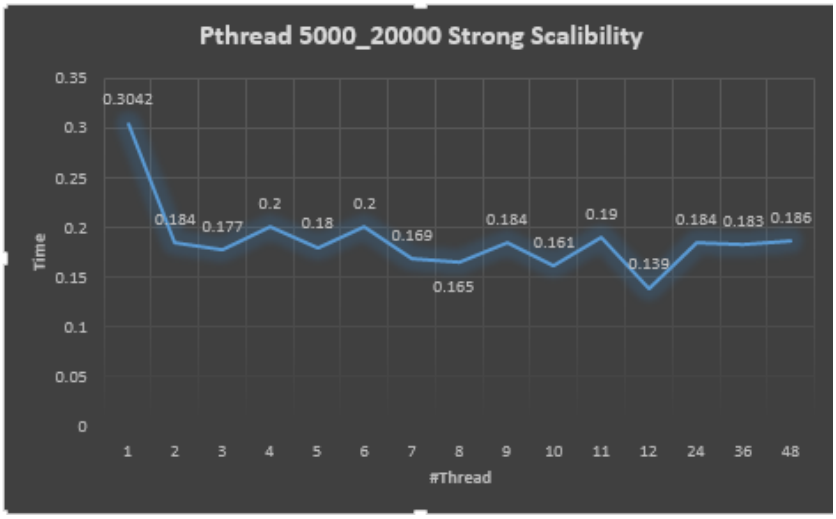
a. Pthread Strong Scalability & Time Distribution



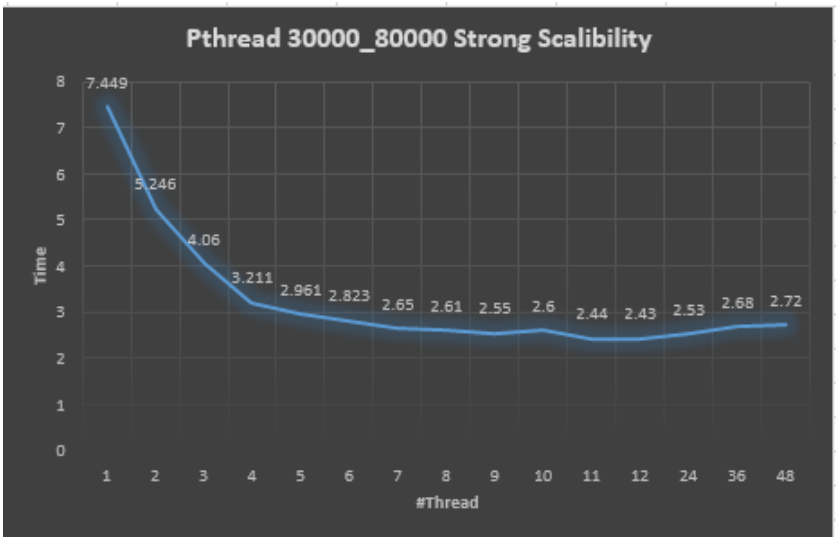


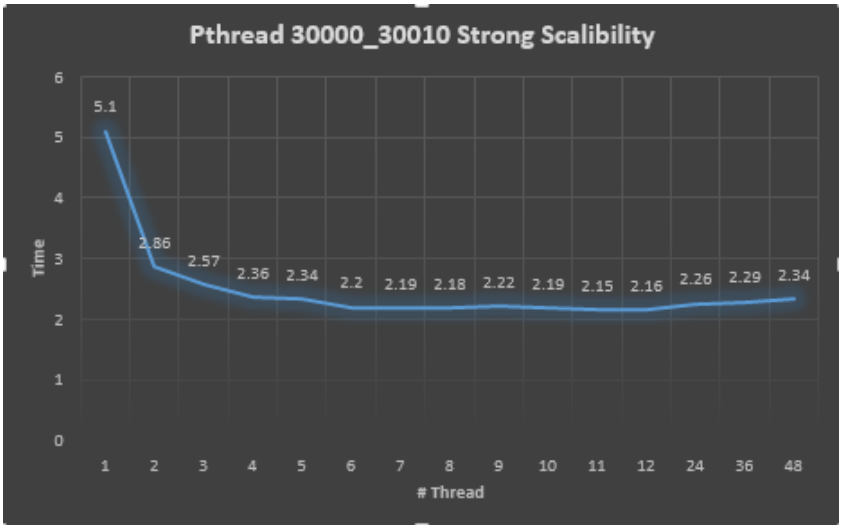


首先看到 Pthread 48_65 的 Strong Scalability 和 Time distribution，可以發現隨著 thread 數量的變多，整體效率是不增反減的。這部分的原因很單純。參照上方的 Time Distribution，我們可以發現在 vertices 數不大時，整個執行時間大部分都是花在 I/O 的 Overhead 上了。而 thread 從 1 增到 12 時，我們發現 CPU Time 還變多了，這部分我覺得是因為隨著 thread 增加維護 thread 的時間成本(pthread_create、pthread_join…etc)增加的幅度大過於因運算平行化而節省掉的時間。因此在 I/O 不變，CPU 時間增加的狀況下 Strong scalability 就會呈現上面的結果。(至於 synchronization 時間因為整體運算量非常小，似乎還不會有要等待 Lock 的狀況發生)



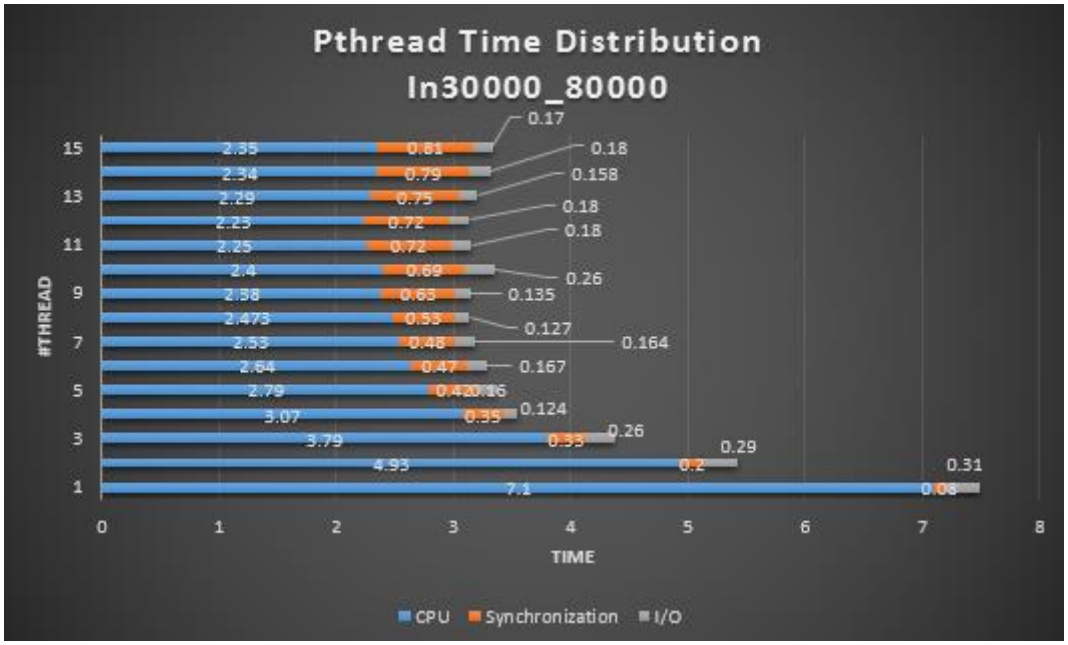
接著看到 input 5000_20000 的狀況。我們可以發現平行化的效果確實是有出來，thread 數不為 1 的狀況都比 thread 數為 1 的狀況還要快。至於為何 thread 數超過 2 以後幾乎不會再變快，我覺得可能的原因除了像上述的維護 thread 的 overhead 以外，我推測另一個可能的原因是 queue 中沒辦法隨時維持超過 2 個以上的 vertex 在內，所以很多 thread 其實就是閒閒沒事晾在那邊。





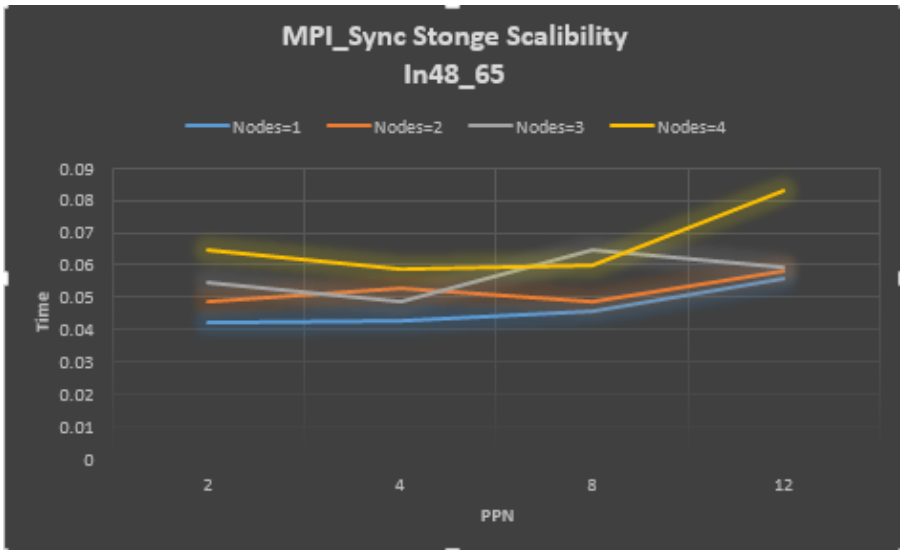
接著看到 Input30000_80000 的狀況，可以發現 Strong scalability 的效果又更好了。Thread 數一直增加到 12 以前時間都會穩定的減少，這部分原因大概就是龐大的運算量讓 queue 隨時都有不少的 vertices，所以平行化的效率可以獲得提升。然而可以發現 thread 超過 12 之後時間卻是反而開始增加的。這部分是因為實際單一機器裡的 core 最多就是 12 個，就算嘗試增加 thread 數，但是實體運算單元的上限是 12，因此也不會變快，反而還會有更多的 thread 維護跟 context switch 的 overhead 存在。

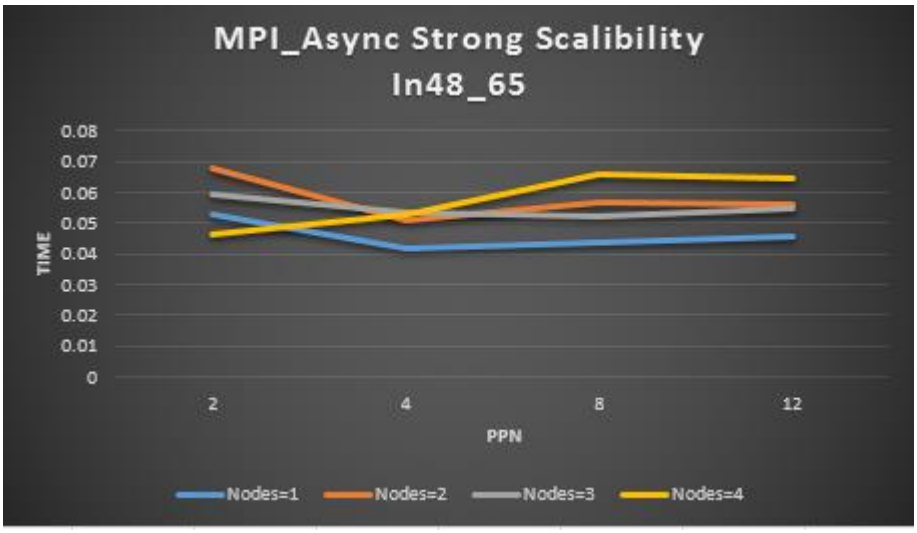
至於 Input30000_30010 的狀況，我們可以發現 Strong scalability 的狀況跟 In5000_20000 有點像，這部分我認為應該是因為這是一個很稀疏的圖，所以每個點在更新的時候幾乎只會 Push 1 個左右新的 vertex 進去 queue 裡面(因為鄰居少)，因此儘管 vertices 數多，但是仍舊會有許多 thread 因為 queue 是空的沒有工作可以做而晾在那邊。



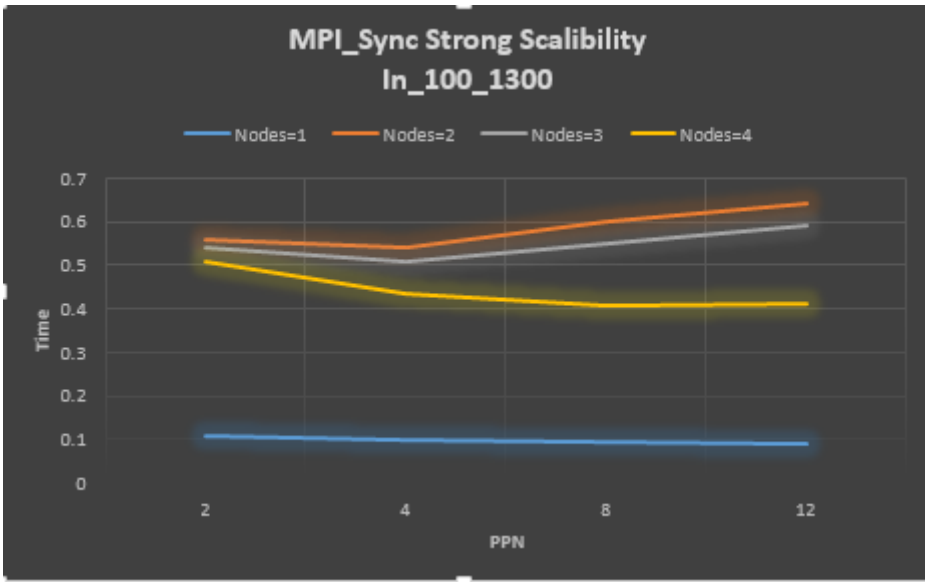
最後看到 Input30000_80000 的狀況下的 time distribution。我們可以看到絕大部分的時間是花在實際的 CPU 時間，而 I/O 大致呈現固定值(每次跑會有一點誤差)。而 Synchronization 時間，我的計算方法是把每個 thread 的總等待 lock 的時間加總(因為總工作量一樣，所以取平均的話會看不出來差異)。可以看到 synchronization time 幾乎是隨著 thread 增加成比例增加的，這部分是因為 In30000_80000 的狀況下 queue 隨時都有 vertex 在裡面，因此每個 thread 經常都會在等待取得 Mutex 以從 queue 中取得 vertex 來工作。

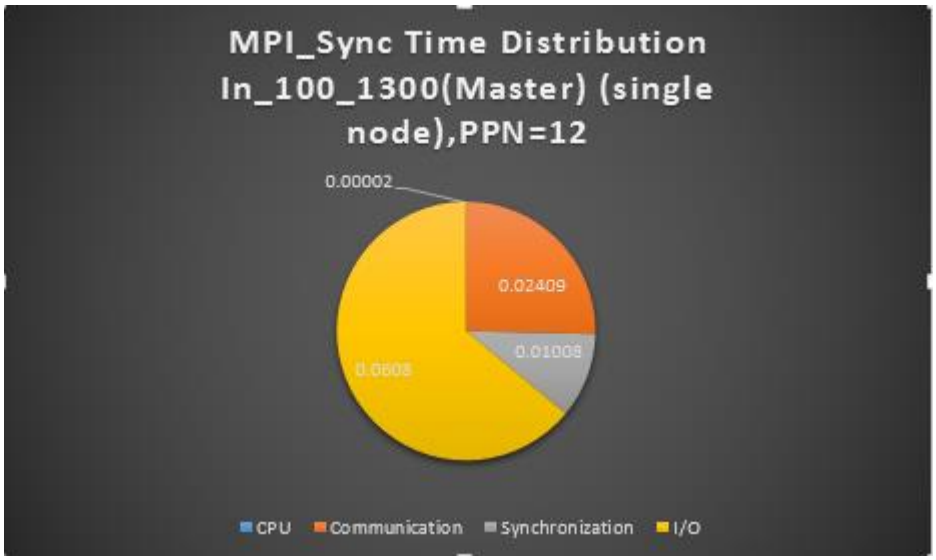
B. MPI_Synchronization / Asynchronous Strong Scalability & Time Distribution & Load Balancing:

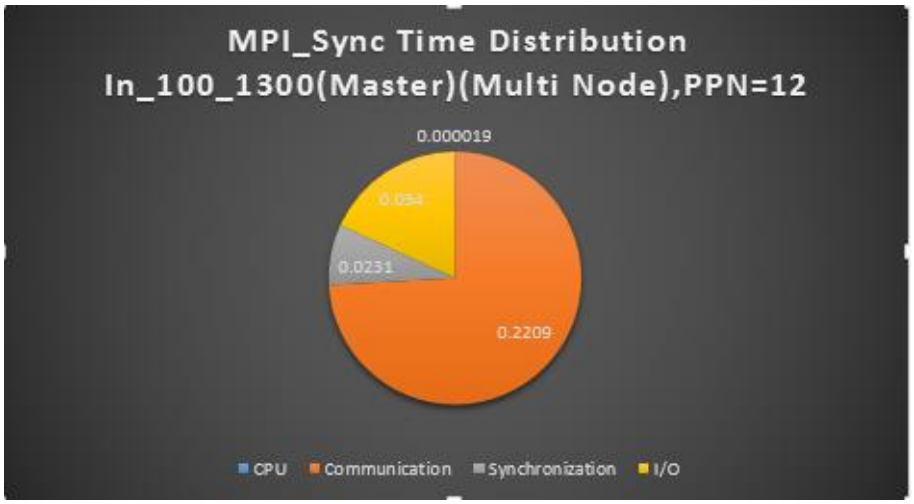


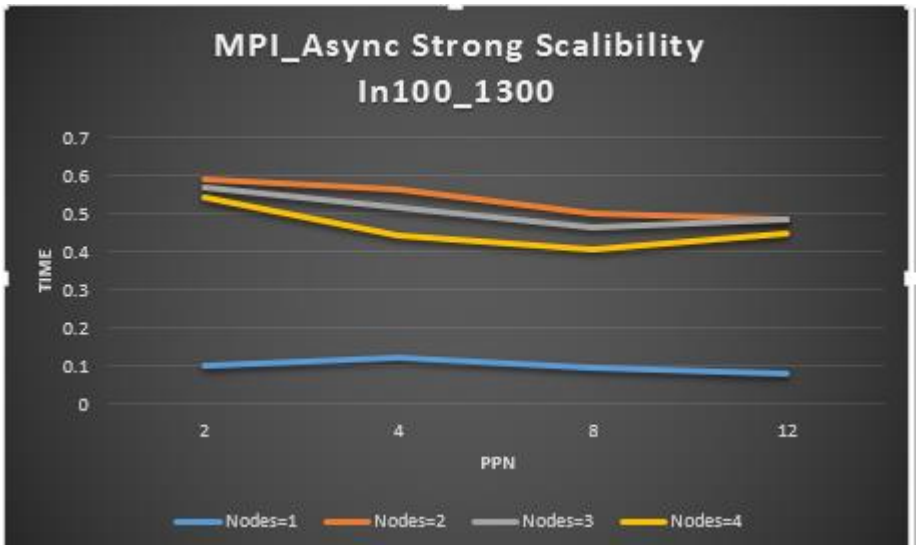


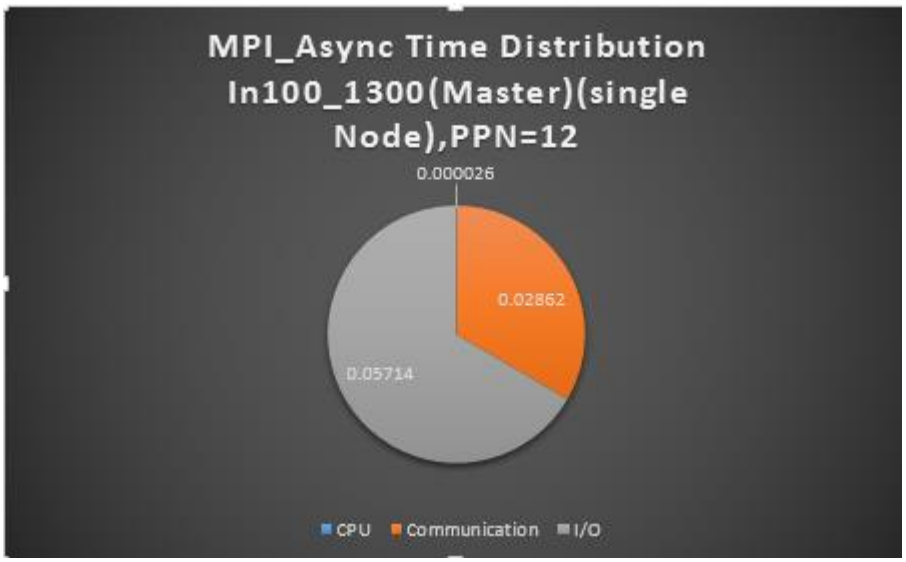
首先看到 MPI 在 In48_65 的狀況下的 Strong Scalability。可以發現隨著 PPN 增加，執行時間並不會縮短。這部分的原因我認為就跟 Pthread 的部份一樣，I/O 有個 Overhead 在，而實際的 SSSP 演算法在小的 Input 下根本看不出平行化的差異時間。隨著 NODE 數變多，時間增加的原因我認為是因為如果執行的機器增多，需要經過的 communication overhead 就會變長(因為要經過 infini band)。而 Synchronous 和 Asynchronous 的差異在這樣的 Input 下還看不出來。

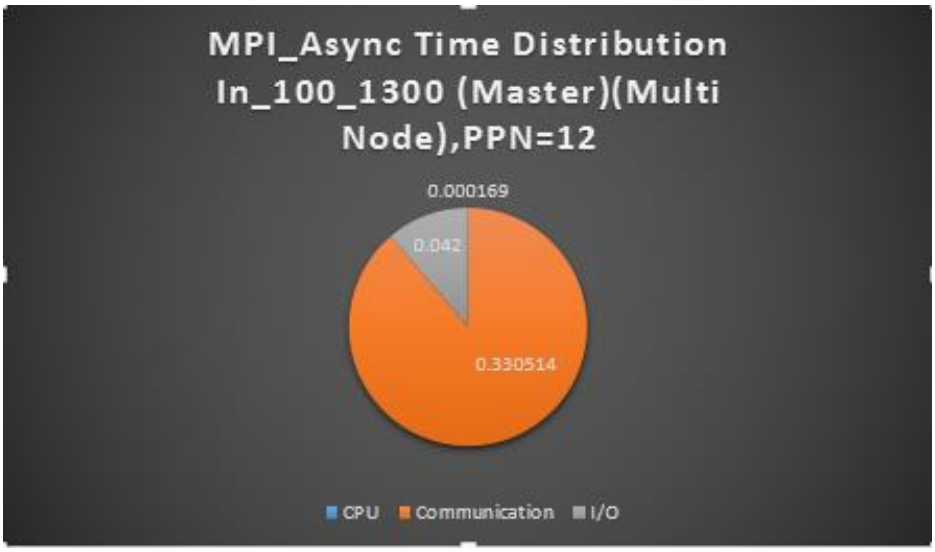





MPI_Sync Time Distribution
In_100_1300(Master)(Multi Node),PPN=12
0.000019
0.054
0.0231
0.2209
CPU
Communication
Synchronization
I/O


MPI_Async Strong Scalibility
In100_1300
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
TIME
2
4
8
12
PPN
Nodes=1
Nodes=2
Nodes=3
Nodes=4

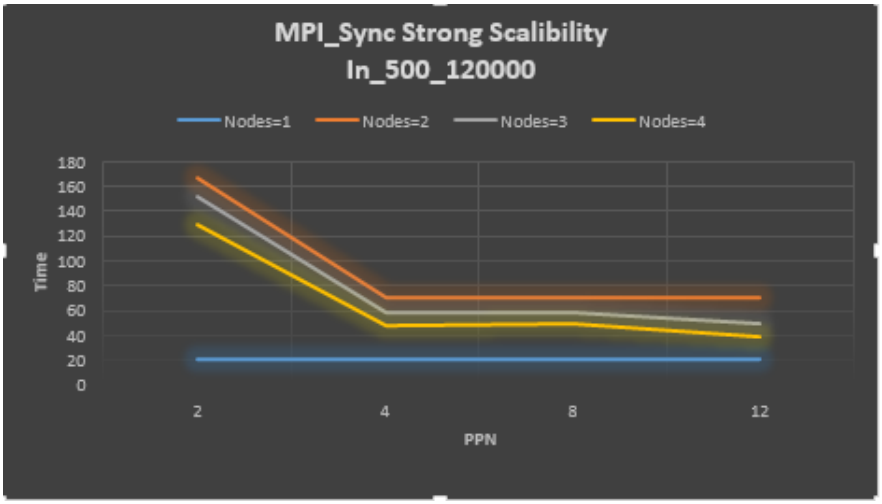

MPI_Async Time Distribution
In100_1300(Master)(single
Node),PPN=12
0.000026
0.02862
0.05714
CPU
Communication
I/O

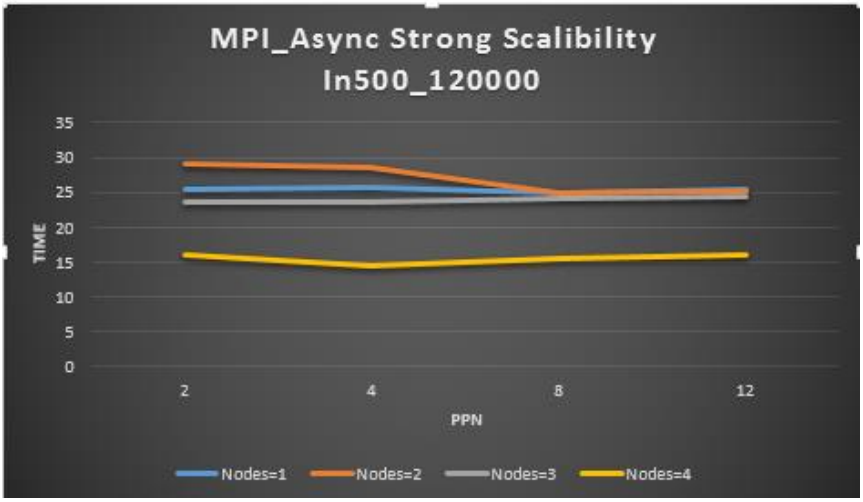


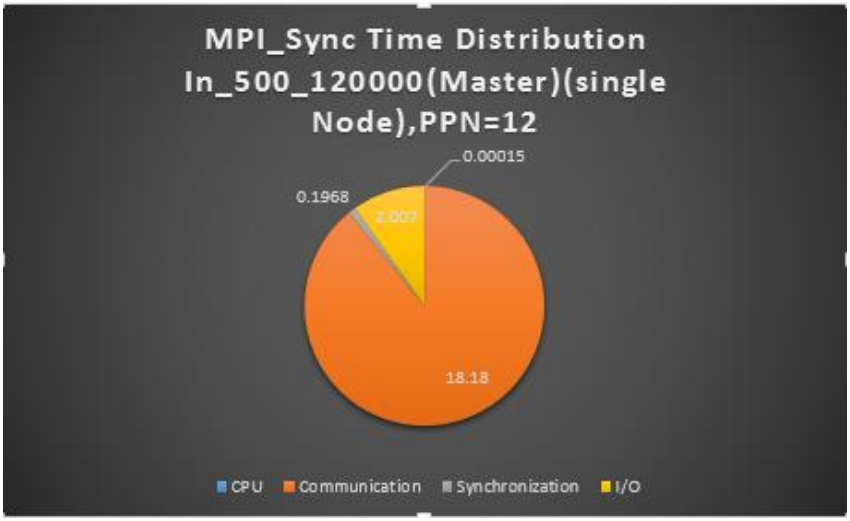
接著看到 In_100_1300 的狀況。令人意外的是結果看起來還是沒有隨著 core 數變多而變快的跡象。觀察 Time Distribution(rank=0)圓餅圖可以發現，在 single node 中大部分的時間都在做 I/O，而在 Multi Node 中大部分的時間都在做 communication 和 Synchronization，Communication 和 synchronization 的時間幾乎變成十倍左右(不論是 Synchronous 或 Asynchronous，因此可以想見訊息有沒有經過 network 的差異非常的大。而實際的 CPU 時間極少(在圓餅圖裡甚至幾乎看不到，在圓形的上方的那個數字是 CPU 時間)。

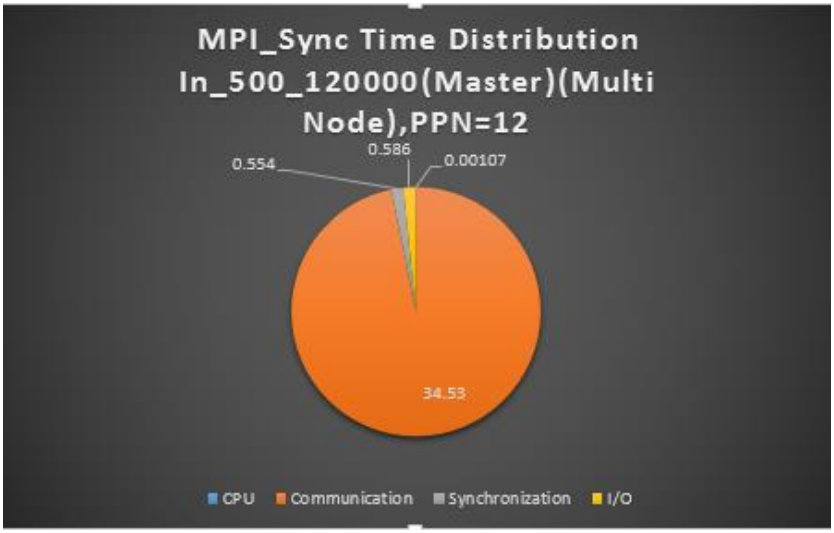
如果說在這種 fully distributed 的架構下絕大部分的時間是由 communication(&synchronization)所佔據，那 PPN 增多速度卻沒有變快我的解釋大概如下：雖然 PPN 增多呼叫 MPI_Send 和 MPI_Recv 的速度會變快(可能一次可以有好幾個 core 一起 call send/recv)，但是實際上傳遞訊息還是需要 os 的支援，因此就算 Core 變多，訊息的傳遞還是 bounded by os 的效能。因此在 fully distributed SSSP 這種問題之下，因為每次收到訊息後的運算量非常小(就是更新距離)，而需要傳送的訊息量很龐大(尤其在 vertices 數多的狀態下)，因此更多的 Core 並不必然會帶來更好的效果。

另外一點值得注意的是，其實觀察 synchronous 和 asynchronous 的圓餅圖可以發現 asynchronous 的 communication 時間比跟 synchronous 的 communication 和 synchronization 加起來的時間還要多，這點我認為也許是因為 synchronous 的訊息是一起發送的，而 asynchronous 是分開來送，因此也許在 mpi 的 message passing 機制中 pass message 有個 overhead，同時發送大量的訊息會比有一些時間差分開來發送大量的訊息還要來的有效率一些。


MPI_Sync Strong Scalibility
In_500_120000
Nodes=1
Nodes=2
Nodes=3
Nodes=4
180
160
140
120
100
80
60
40
20
0
Time
2
4
8
12
PPN


MPI_Async Strong Scalibility
In500_120000
35
30
25
20
15
10
5
0
TIME
2
4
8
12
PPN
Nodes=1
Nodes=2
Nodes=3
Nodes=4


MPI_Sync Time Distribution
In_500_120000(Master)(single
Node),PPN=12
0.00015
0.1968
2.007
18.18
CPU
Communication
Synchronization
I/O

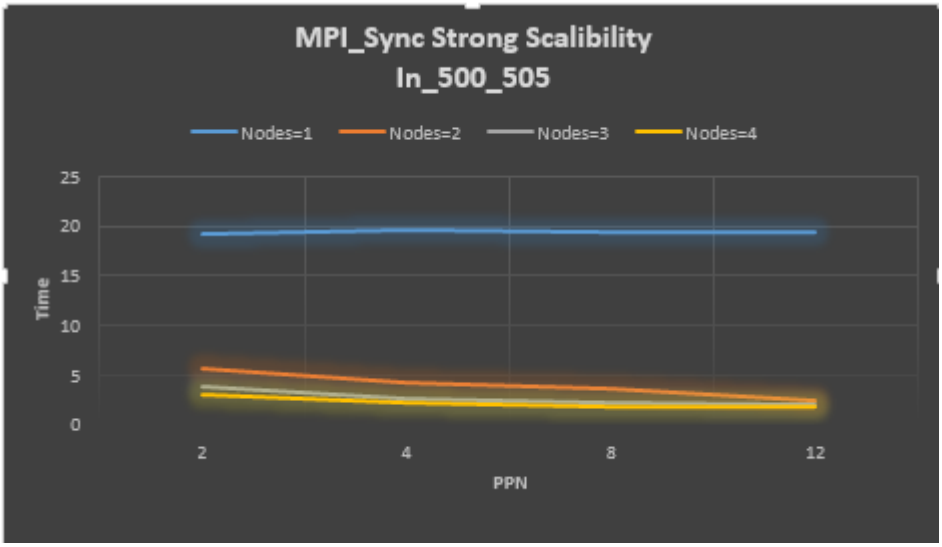


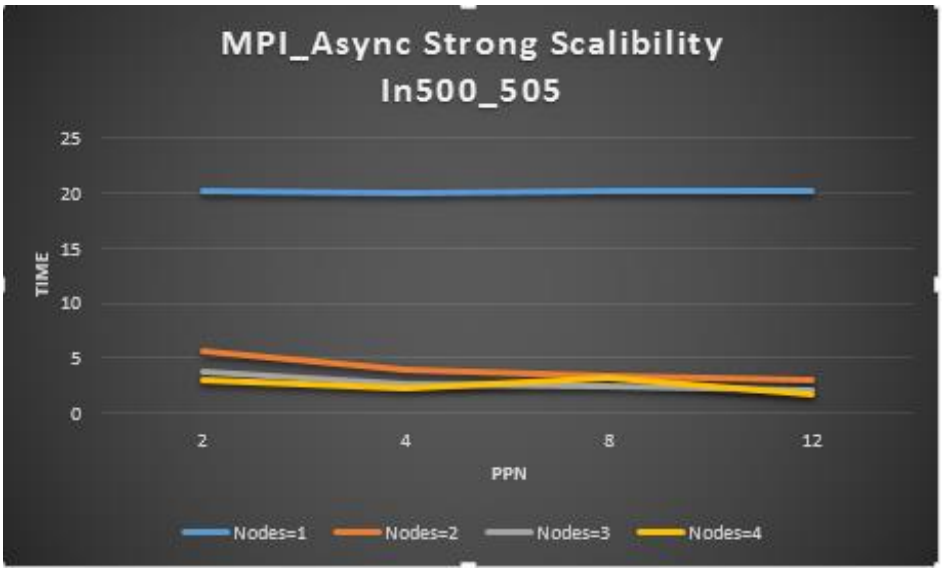
接著我們看到大測資的狀況，因為 process 數必須=vertices 數的限制，沒辦法用到破千的測資，因此用 500_120000 這樣子幾近於 Complete graph 的測資來測試。觀察 Strong Scalability，發現隨著 PPN 數增多計算時間沒有什麼變少這個假設依然維持。(有一個例外，在 multi node ppn=2 跟!=2 時效率會有差，不過我不認為這部分是因為 core 數的影響，因為 4:2 效率比 2:4 還差很多，所以我認為也許是在特定的 PPN 時 cluster 底層 OS 的排程演算法會不同和收發 message 的機制不同，可能剛好就是在 PPN=2 時使用的排程和 message passing 機制不利於這組 Input)

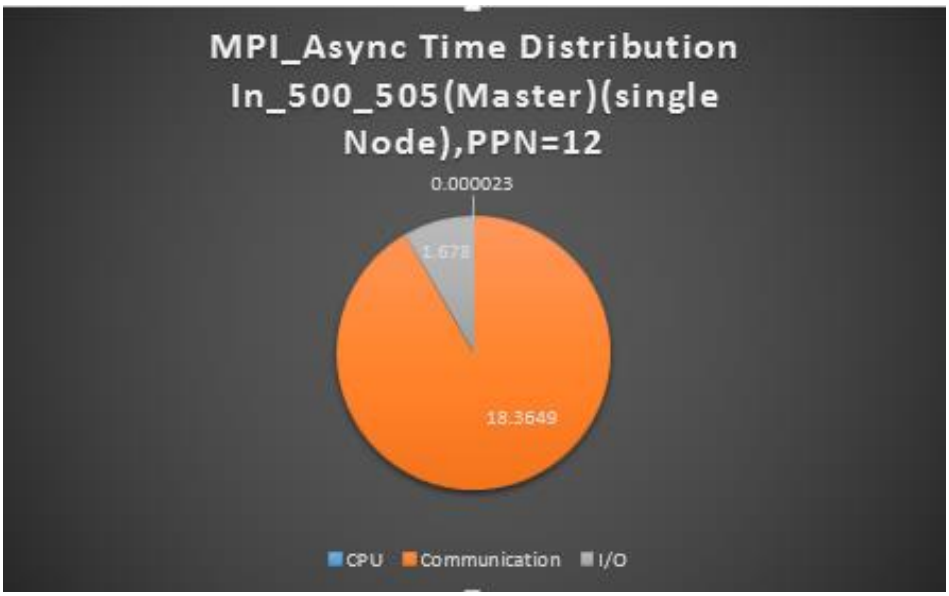
只是可以發現 Aynchronous 的演算法的效率整體是比 Synchronous 還好的，這部分我認為是因為 Synchronous 幾乎每個 process 會在相同的時間點發出(120000*2)如此多的 message，而。Asynchronous 因為 process 間彼此收發 message 的時間點會錯開來，所以比較可以避免掉 os 沒辦法負荷同時太多 message 要傳送的問題。

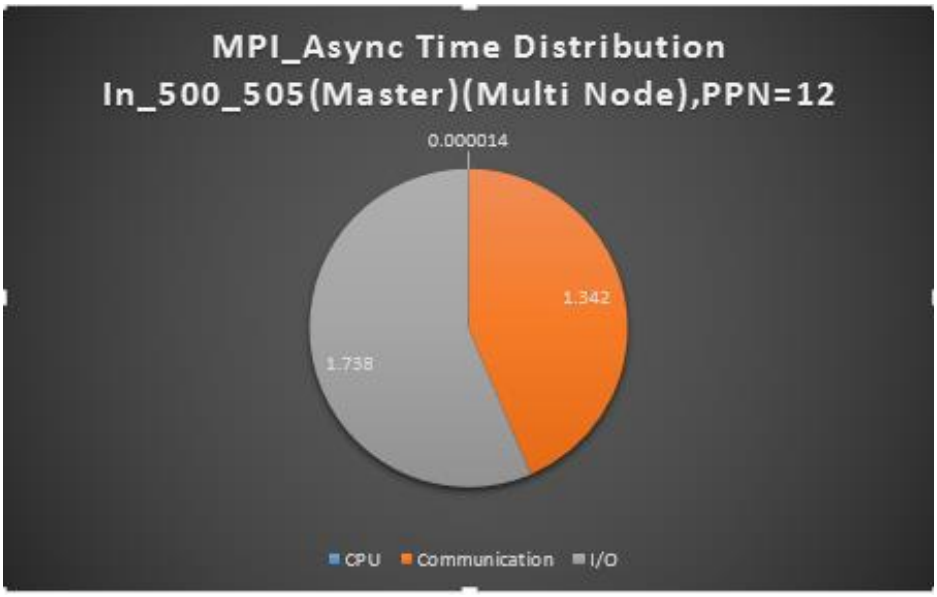
而觀察 Time Distribution，可以發現 communication time 幾乎佔據了全部的時間，而且比例比起 48_65 和 100_1300 還要多，從這點我們可以確認 Fully distributed SSSP 這個問題隨著 Input size 增加時間的增加幾乎是導因於 Communication time。(而 multi node 比 single node 的 communication time 還多的現象依然還是可以觀察的到)

最後一點值得注意的是可以發現(不論 synchronous 或 asynchronous)，雖然在 multi node 的溝通時間比 single node 還多，但是 node = 4 的溝通時間 < node=3 的溝通時間 < node=2 的溝通時間。這部分我認為是因為 node 數越多，可以處理訊息收發的 os 也越多，所以某種程度可以改善 Communication 的 overhead 的問題。


MPI_Sync Strong Scalibility
In_500_505
Nodes=1
Nodes=2
Nodes=3
Nodes=4
25
20
15
10
5
0
Time
2
4
8
12
PPN


MPI_Async Strong Scalibility
In500_505
25
20
15
10
5
0
TIME
2
4
8
12
PPN
Nodes=1
Nodes=2
Nodes=3
Nodes=4

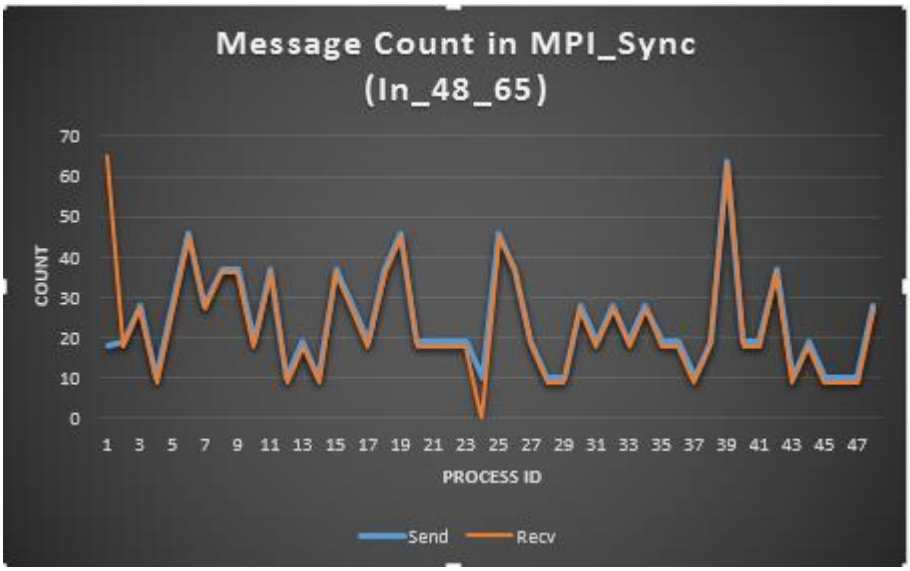

MPI_Async Time Distribution
In_500_505(Master)(single
Node),PPN=12
0.000023
1.671
18.3649
CPU
Communication
I/O

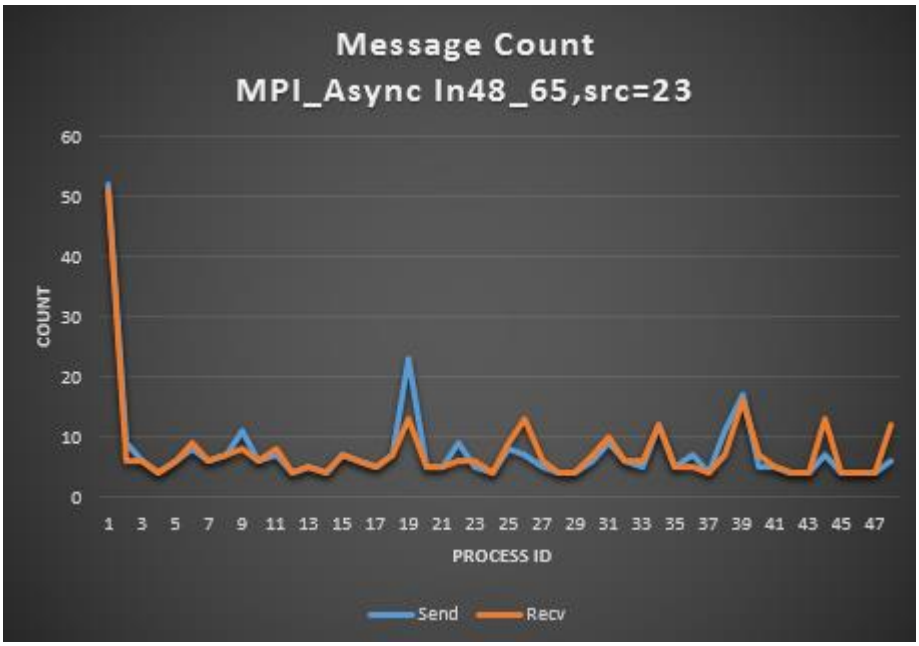


最後看到在 In500_505 這種稀疏的圖為 Input 的狀況。在這種狀況下就跟密集圖有些落差。Single Node 的效率普遍比 Multi Node 還要差。這部分我認為可能的原因可能有(1)Multi Node 因為機器數較多，可以處理訊息收發的 OS 也較多，或是(2)在 Single Node 跟 Multi Node 時，cluster 使用的排程演算法不同，而 Multi Node 時的排程演算法針對這樣的 input graph 來的有效率。

觀察 Time distribution 可以發現在 Multi node 中 communication 的時間佔的比例突然大幅下降，可以推測 network 的 overhead 在這樣的 input 中其實不是 communication overhead 最主要的組成因素，而是來自於 OS 處理收發訊息的效率。

整體而言，Fully distributed SSSP 因為某次收發訊息後的運算量(computation)都非常少，而且 Input size 因為受限於 process 數而無法太大，而且 communication 的 Overhead 又太高，因此無法看出什麼效果上優異之處。單就 SSSP 這個演算法而言，用 Pthread 做平行化會是比較好的選擇。

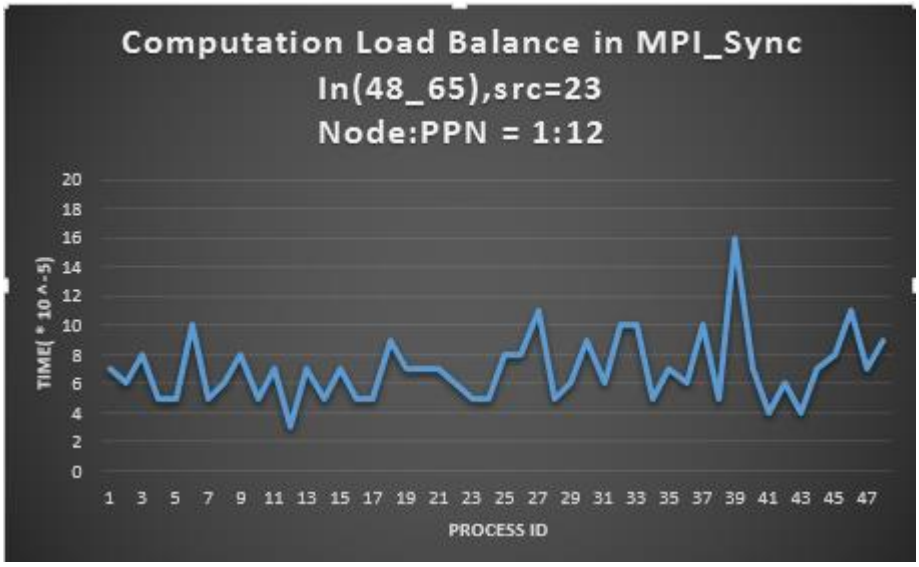


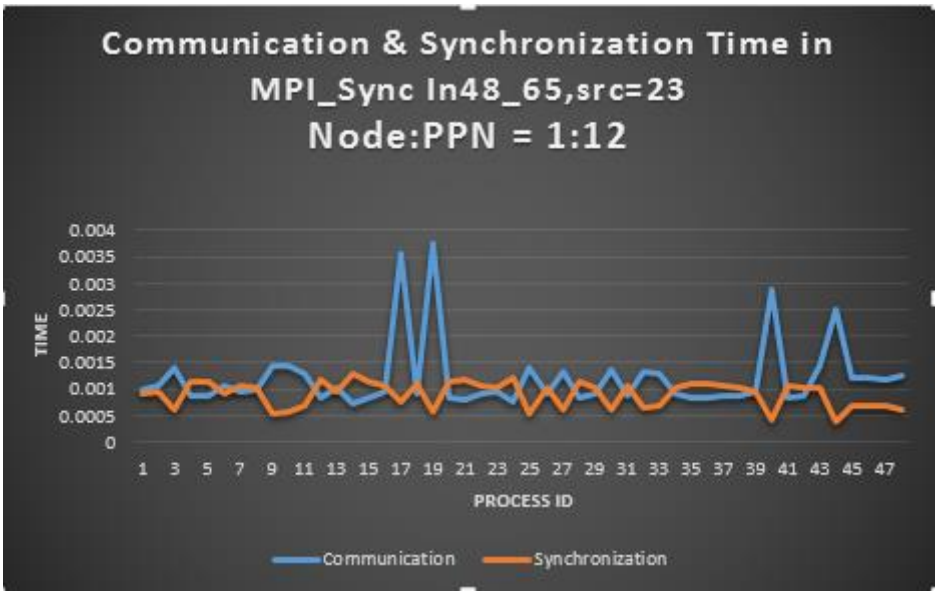


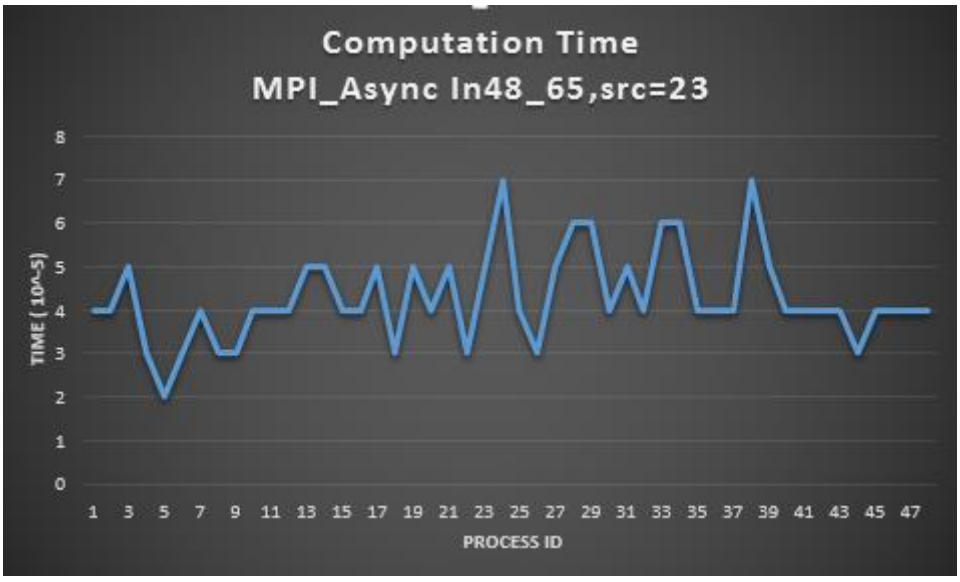
接著看到 Load balancing 的部分。以上兩張圖描述在同步和非同步的演算法時，In48_65(src=23)這組 input 的 send message 跟 receive message 的 load distribution。

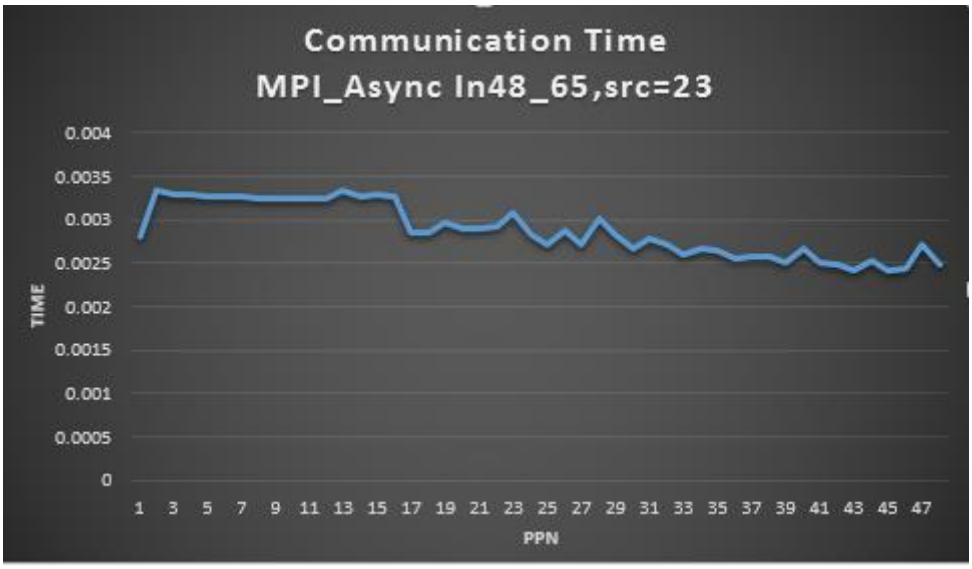
整體而言，可以看到 Asynchronous 的訊息數量平均遠低於 Synchronous 的數量。這部分就如同前面解釋的一樣，是因為 Synchronous 在每個 iteration 都必須 send 訊息給所有 outgoing neighbor，也都必須從所有 fan in neighbor recv message。而因為此 input 比較少，dual ring 演算法可能沒有經過太多輪就結束了，因此沒有造成太大的訊息量。

而因為我的 code 是由 rank=0 的 process 把 input 發送給每個 process，並且在 SSSP 計算完成之後收集所有 Process 的運算結果統一輸出。所以可以看到在 process ID = 0 的狀態下 Send 跟 Recv 值都很高。


Computation Load Balance in MPI_Sync
In(48_65),src=23
Node:PPN = 1:12
TIME(* 10 ^-5)
0 2 4 6 8 10 12 14 16 18 20
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
PROCESS ID


Communication & Synchronization Time in
MPI_Sync In48_65,src=23
Node:PPN = 1:12
TIME
0 0.0005 0.001 0.0015 0.002 0.0025 0.003 0.0035 0.004
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
PROCESS ID
Communication
Synchronization


Computation Time
MPI_Async In48_65,src=23
TIME (10^-5)
0 1 2 3 4 5 6 7 8
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
PROCESS ID

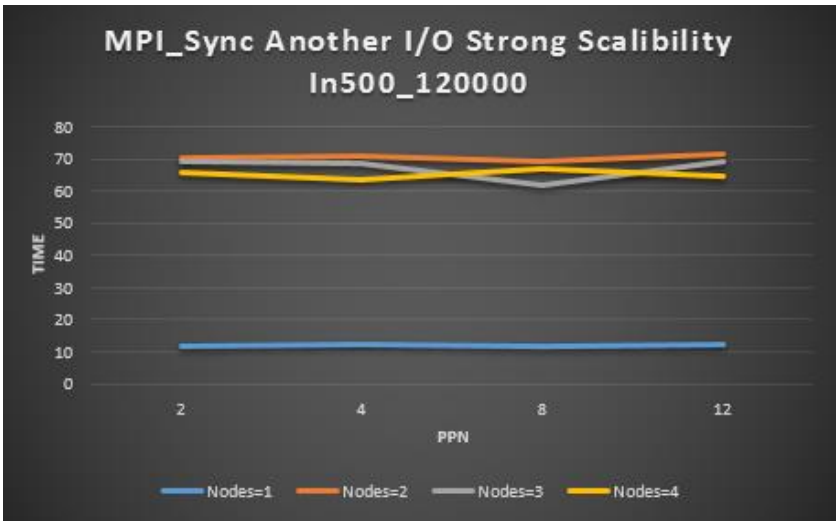


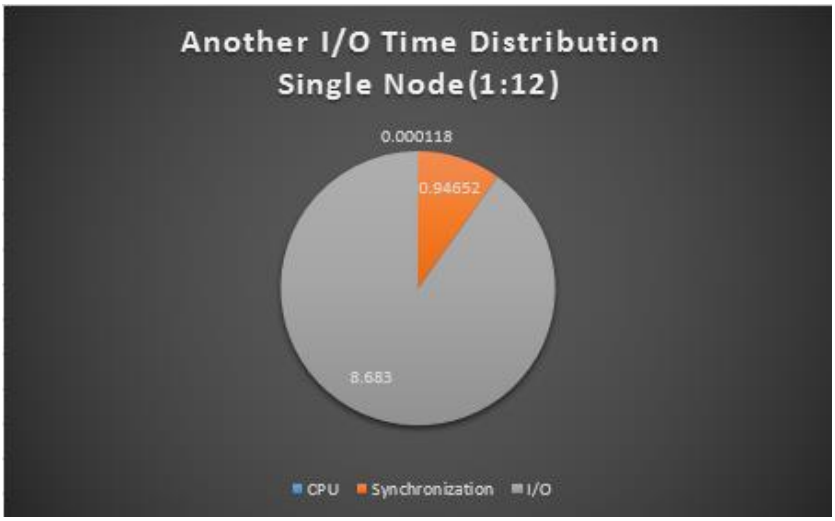
因為 total execution time 每個 process 都差不多(因為在程式末段會統一把 input send 給 rank=0 的 process)，所以 load balancing 的部分我用 CPU time balance 和 Communication time 來分析。對比於 message count 的圖可以發現，大致上收發訊息數量多的 Process 同時會有比較高的 cpu time 和 communication time。訊息量多，除了 communication time 變多以外，cpu 也必須花更多時間處理這些訊息，因此不只是 communication time 會增加。而且比起 communication，cpu time 的 load balance 的差距會更明顯(因為傳送和接收訊息的時間會相依於與他溝通的 process，而計算量不會，degree 越多的 vertex 在 cpu 的時間增加上也會比較明顯。

另外一點就是可以發現 Synchronous 版本中 Communication time 高的 process，Synchronization time 就會比較少，這部分可以解釋為比較早收發完訊息的 process 會比較早進入 barrier，因此反而 Synchronization time 會比較高。

Extra Experiment : Another I/O way

我在 MPI 的版本中進行讀取 Input graph 的方式是由 rank=0 的 process 統一讀檔之後 Broad cast 給其他的 process。然後這會遇到一個問題，因為我會把整個二維的 weight array broad cast 出去。所以隨著 vertices 增多，這個需要廣播的訊息大小會成平方倍成長。因此我嘗試了另一種 I/O 的方式，也就是讓所有 Process 自己去讀取 input，因為 POSIX fopen()在同步 read 的狀況下是不會有問題的。(輸出的部分還是統一由 rank=0 收集之後輸出)。在 In_500_120000 的 input graph 下觀察到以下結果：





對比於前面的結果，我們可以發現在 Single Node 的狀態下 Communication 時間幾乎剩下原本的 1/10，而 I/O 時間則也暴增近十倍左右。因此我們可以推測說在 Single node 的狀態下，原本的 Communication 時間有很大一部份是在一開始的時候把 Input bcast 出去的時候產生的。而在 Multi-Node 之中時間的減少對比於前面則比較不那麼明顯，我認為是因為在 Multi-Node 的狀況下 Communication 時間本來很大的比例就是在 SSSP 演算法本體中透過 network 傳送訊息時所產生的。

3.Experience & Conclusion:

、 這次的作業讓我學到了：

1 .Pthread 的使用以及 shared memory programming 的一些技巧、race condition 的處理，以及設計平行化 SSSP 演算法的經驗。雖然先前已經練習

過 OpenMP 了，但是因為 OpenMP 算是 Compiler 的 directive，很多細節沒有辦法親自處理到，所以到現在才算是比較了解一些 shared memory programming 的方式。

2 Fully distributed 的平行程式模型的練習。雖然說這次作業的問題並不是特別適合用這樣的 Model 來解決，但是卻是個很好用來理解這種模型的問題(因為計算本身單純，重點放在 process 間的溝通)。相信有許多類型的問題是更適合用這種模型來 Load balancing 的。

在這次作業中遇到的困難點：

1. MPI_Asynchronous 一開始在實作的時候怎麼樣都會發生 dead lock 的狀況，因為要同時處理距離的訊息計算和 token 的管理，一個地方順序沒弄對整個程式就會完全卡死。
2. 實驗的部分很多地方跑出來的結果都跟預期的有些落差，但是因為我也不清楚底層系統的一些詳細設定，所以很多實驗結果要想提供合理的解釋讓我花了不少時間。