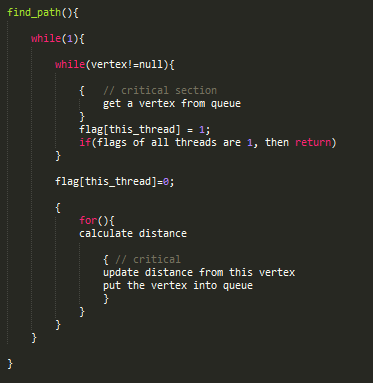
Parallel Praogramming HW3 Report

102062209邱政凱

1. Design
2. 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：



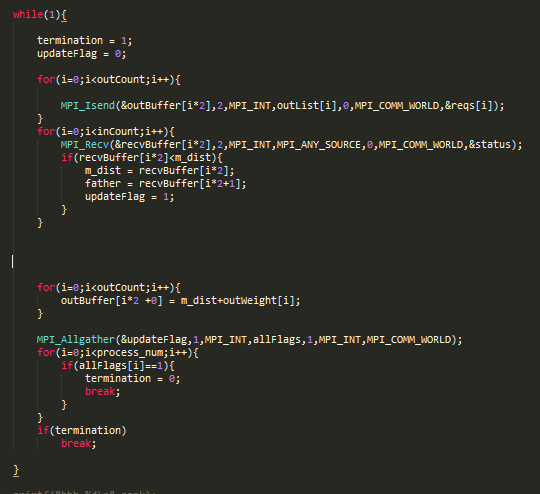
我選擇用這種方式實作Pthread的版本的原因是，SSSP本身並不是個好平行化的問題，用這種Moore’s algorithm的方式，所有thread之間只需要共用一個queue，每回合取出queue中的vertex之後計算的部分都是各個process間獨立的，每個thread之間的相依性很低，不太會有互相等待的問題。，它可以達到較高的平行度，需要處理race condition的地方也只有在queue的操作而已，因此我認為是個有效率的演算法。

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。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)。

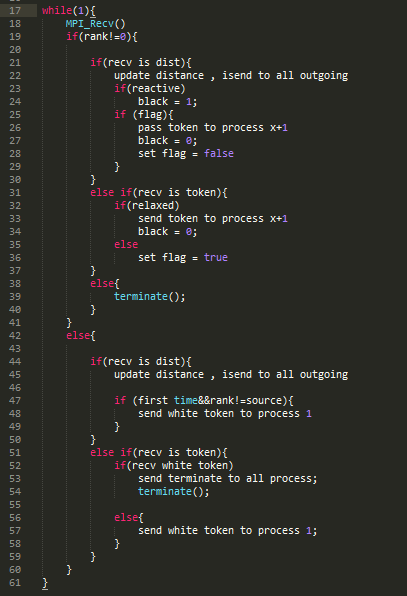
每次recv到一個訊息就確認是否新的距離比目前的還短，是的話就更新當前距離，並同時記錄這個距離的更新是由誰送來的(以供output使用)。最後在每個回合結束前用MPI\_Allgather收集termination condition。如果某process在該iteration中距離有被更新，就把flag設成true，否則設成false。

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：



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：



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。

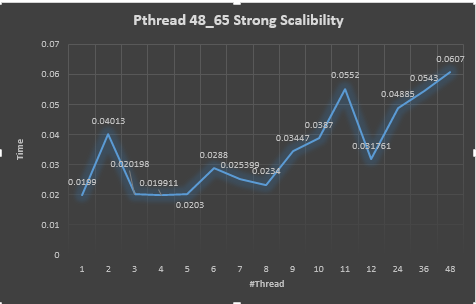
而非同步版本的優勢則在於所有process間的行為相依性降低，且因為行為不需要一致，可以不用發送太多不必要的訊息。缺點我認為是code架構比較難規劃，且須要維護額外的termination detection algorithm，所以process每次收到新的訊息後的運算量也比synchronous大。我認為比較適合較為稀疏(edge較少)的graph。

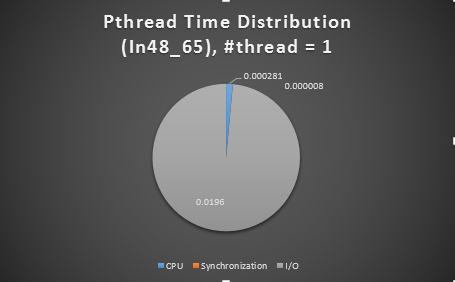
1. Experiment & Analysis

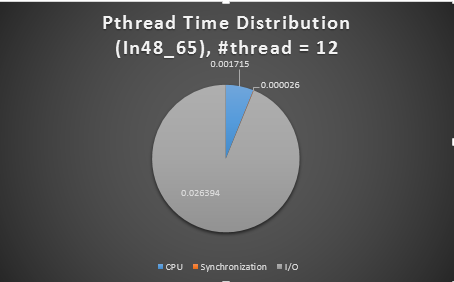
我使用的環境是課程提供的cluster，因此規格就跟助教公布的一樣。

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。I/O時間就是計算每個I/O函式個別的時間加總，Communication時間就是把所有MPI\_Send/Recv/Bcast的時間加總。Synchronization的時間就是process等在synchronous point函式的時間。(pthread中指的則是取得mutex所花的時間)。CPU時間則是把總時間撿到以上所有時間。

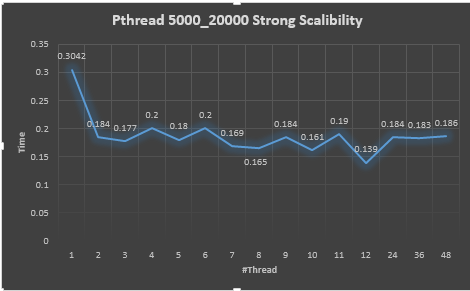
1. Pthread Strong Scalability & Time Distribution



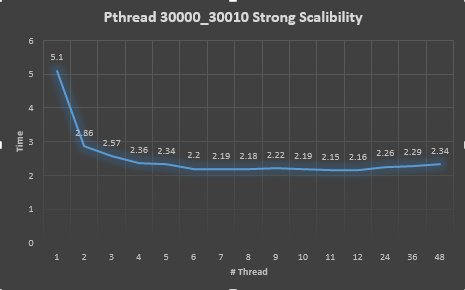
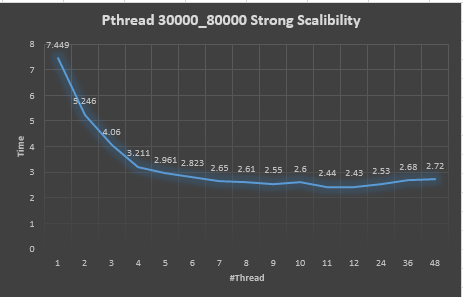




首先看到Pthread 48\_65的Strong Scalability和Time distribution，可以發現隨著thread數量的變多，整體效率是不增反減的。這部分的原因很單純。參照上方的Time Distribution，我們可以發現在vertices數不大時，整個執行時間大部分都是花在I/O的Overhead上了。而thread從1增到12時，我們發現CPU Time還變多了，這部分我覺得是因為隨著thread增加維護thread的時間成本(pthread\_create、pthread\_join…etc)增加的幅度大過於因運算平行化而節省掉的時間。因此在I/O不變，CPU時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

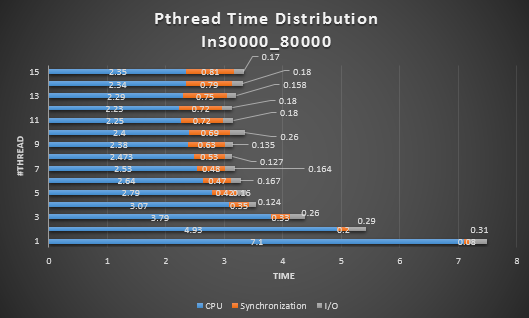


接著看到input 5000\_20000的狀況。我們可以發現平行化的效果確實是有出來，thread數不為1的狀況都比thread數為1的狀況還要快。至於為何thread數超過2以後幾乎不會再變快，我覺得可能的原因除了像上述的維護thread的overhead以外，我推測另一個可能的原因是queue中沒辦法隨時維持超過2個以上的vertex在內，所以很多thread其實就是閒閒沒事晾在那邊。



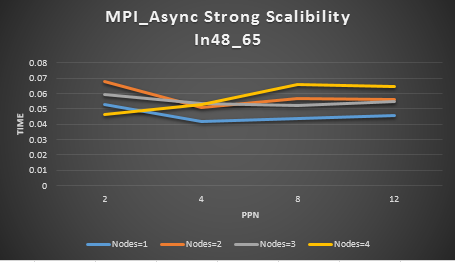
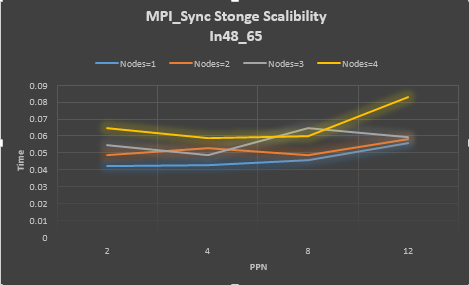
接著看到Input30000\_80000的狀況，可以發現Strong scalability的效果又更好了。Thread數一直增加到12以前時間都會穩定的減少，這部分原因大概就是龐大的運算量讓queue隨時都有不少的vertices，所以平行化的效率可以獲得提升。然而可以發現thread超過12之後時間卻是反而開始增加的。這部分是因為實際單一機器裡的core最多就是12個，就算嘗試增加thread數，但是實體運算單元的上限是12，因此也不會變快，反而還會有更多的thread維護跟context switch的overhead存在。

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。

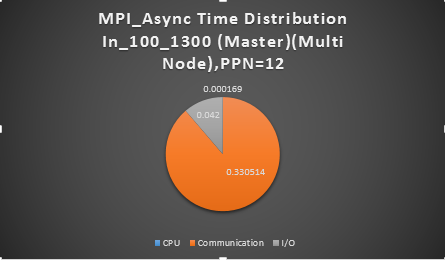
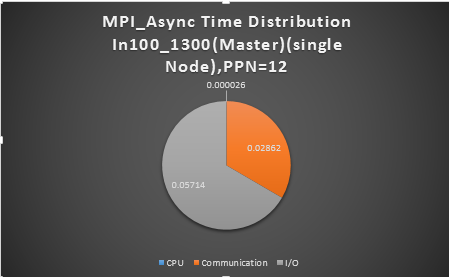
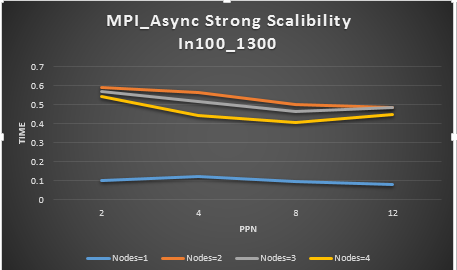
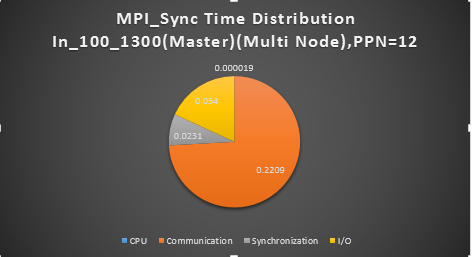
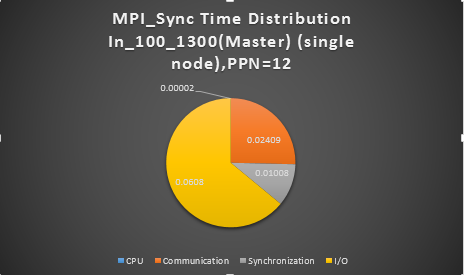
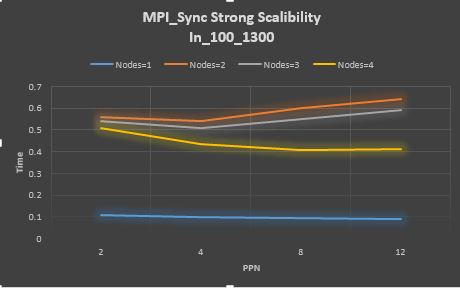


最後看到Input30000\_80000的狀況下的time distribution。我們可以看到絕大部分的時間是花在實際的CPU時間，而I/O大致呈現固定值(每次跑會有一點誤差)。而Synchronization時間，我的計算方法是把每個thread的總等待lock的時間加總(因為總工作量一樣，所以取平均的話會看不出來差異)。可以看到synchronization time幾乎是隨著thread增加成比例增加的，這部分是因為In30000\_80000的狀況下queue隨時都有vertex在裡面，因此每個thread經常都會在等待取得Mutex以從queue中取得vertex來工作。

B. MPI\_Synchronization / Asynchronous Strong Scalability & Time Distribution & Load Balancing:



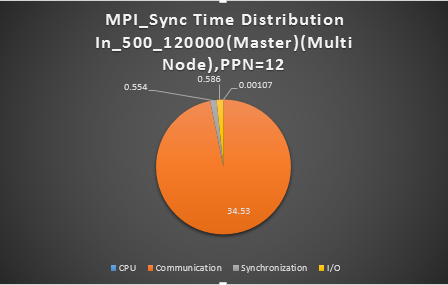
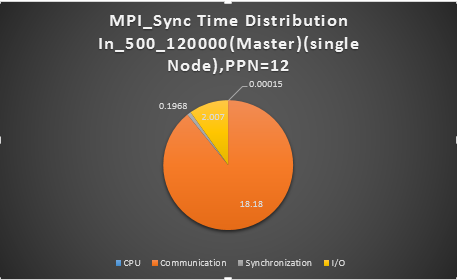
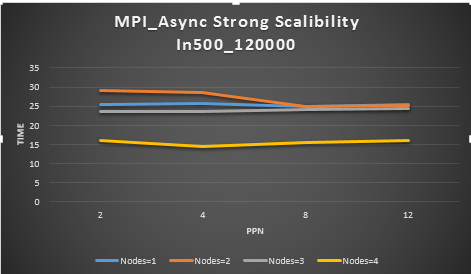
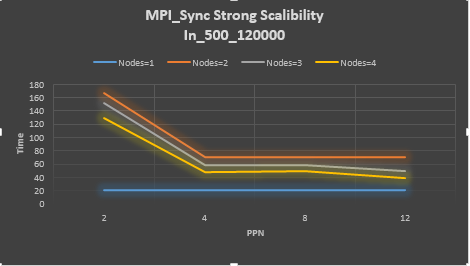
首先看到MPI在 In48\_65的狀況下的Strong Scalability。可以發現隨著PPN增加，執行時間並不會縮短。這部分的原因我認為就跟Pthread的部份一樣，I/O有個Overhead在，而實際的SSSP演算法在小的Input下根本看不出平行化的差異時間。隨著NODE數變多，時間增加的原因我認為是因為如果執行的機器增多，需要經過的communication overhead就會變長(因為要經過infini band)。而Synchronous和Asynchronous的差異在這樣的Input下還看不出來。



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。

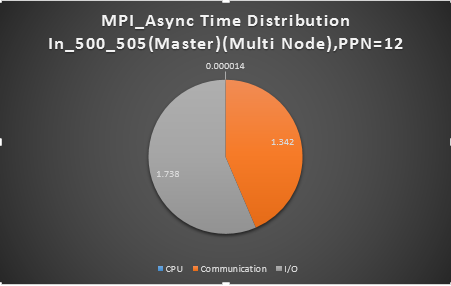
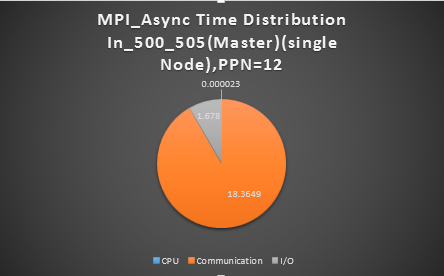
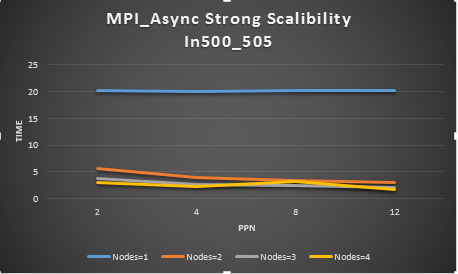
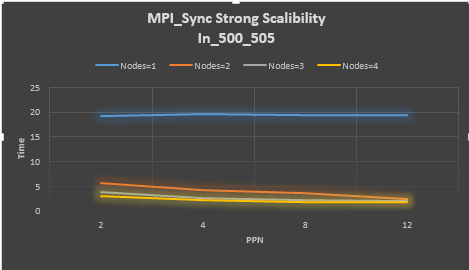


接著我們看到大測資的狀況，因為process數必須=vertices數的限制，沒辦法用到破千的測資，因此用500\_120000這樣子幾近於Complete graph的測資來測試。觀察Strong Scalability，發現隨著PPN數增多計算時間沒有什麼變少這個假設依然維持。(有一個例外，在multi node ppn=2跟!=2時效率會有差，不過我不認為這部分是因為core數的影響，因為4:2效率比2:4還差很多，所以我認為也許是在特定的PPN時cluster底層OS的排程演算法會不同和收發message的機制不同，可能剛好就是在PPN=2時使用的排程和message passing機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

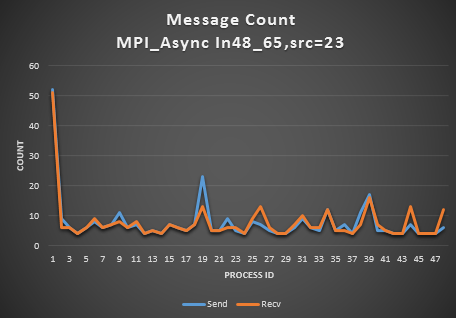
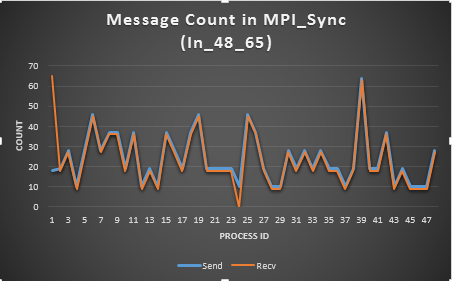
最後一點值得注意的是可以發現(不論synchronous或asynchronous)，雖然在multi node的溝通時間比single node還多，但是 node = 4的溝通時間 < node=3的溝通時間 < node=2的溝通時間。這部分我認為是因為node數越多，可以處理訊息收發的os也越多，所以某種程度可以改善Communication的overhead的問題。



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。

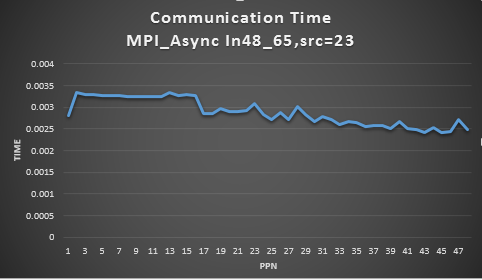
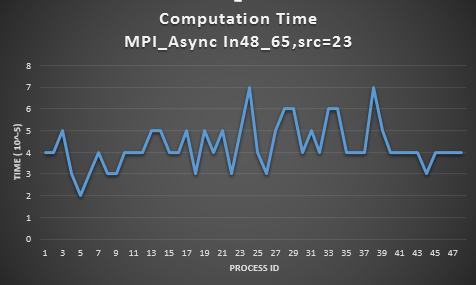
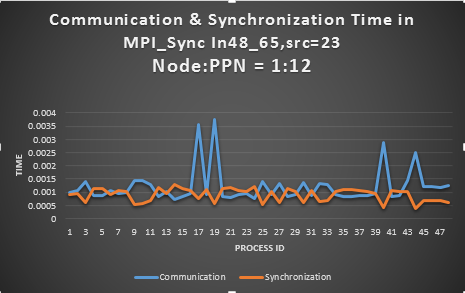
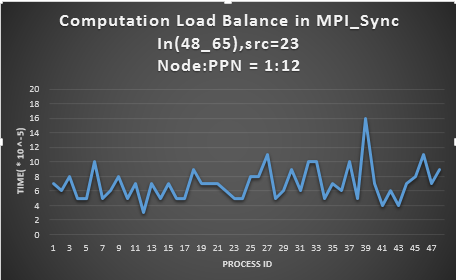
整體而言，Fully distributed SSSP因為某次收發訊息後的運算量(computation)都非常少，而且Input size因為受限於process數而無法太大，而且communication的Overhead又太高，因此無法看出什麼效果上優異之處。單就SSSP這個演算法而言，用Pthread做平行化會是比較好的選擇。



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。

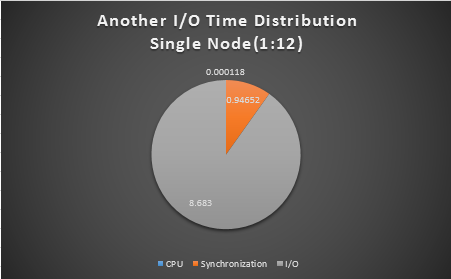
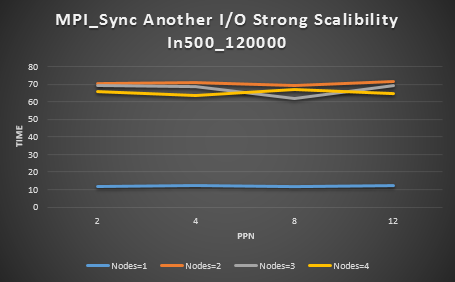


因為total execution time每個process都差不多(因為在程式末段會統一把input send給rank=0的process)，所以load balancing的部分我用CPU time balance和Communication time來分析。對比於message count的圖可以發現，大致上收發訊息數量多的Process同時會有比較高的cpu time和communication time。訊息量多，除了communication time變多以外，cpu也必須花更多時間處理這些訊息，因此不只是communication time會增加。而且比起communication，cpu 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。