Parallel Programming –APSP

106062530 張原嘉

1. Design.

Algorithm、Pros and Cons、Other effort.

✓ASAP_Pthread

這次 Pthread 版使用類 Johnson’ s algorithm 實作。其演算法如下

```
Johnson’s Algo.(G){
    compute G’, where V[G’]=V[G]∪{s} and E[G’]=E[G]∪{(s,v):v∈V[G]
    if Bellman-Ford(G’,w,s)=False
        then print “∃ a neg-weight cycle”
    else
        for each vertex v ∈V[G’]
            set h(v)=δ(s,v) computed by Bellman-Ford algo.
            for each edge (u,v)∈E[G’]
                w’(u,v)=w(u,v)+h(u)-h(v)
            for each vertex u ∈V[G]
          run Dijkstra(G,w’,u) to compute δ’(u,v)
            for each v ∈V[G]
                duv=δ’(u,v)-h(u)+h(v)
     return D
}
```

簡言之

1. 圖上新加一個點，此點到全部的點之距離為 0
2. 跑 Bellman-Ford
3. Reweight（除去負邊）：將每個點設一個高度 h(v)，且調整邊的 weight function w(u,v) 成為 w’ (u,v)=w(u,v)+h(u)-h(v)
4. 跑 Dijkstraδ ‘(u,v)，則全員最短路徑為 δ(u,v)=δ ‘(u,v)-h(u)+h(v)

至於此次作業的步驟為：

1. 將 input 2D array 轉成 1D array（應 Algo 要求）
2. 將 1D array（儲存邊的資訊供 thread 共用）傳入 mainFunction（ ），建立、初始化資料，建立 thread 執行 secondPhase（ ）
3. secondPhase（ ）由初始 thread 資料及跑 Dijkstra 組成，各 thread 自行跑 Dijkstra。Dijkstra 所需的起點位置則以 threadID 為起始位置，thread 數為間隔，總點數為中止條件，將結果放在 tmp 陣列，再回傳給 mainFunction。for (int i = id; i < vertices; i += threads)

(10 個點， 2 個 thread)

4. 取得最短路徑的 1D array（result 陣列）並輸出。

值得一提的是，Dijkstra 實作的資料結構原本為相鄰矩陣，Time complexity 為O（|V|^2），後來改成 priority_queue，因此變為 O(|E| + |V| log |V|)，在速度上有些微提升。

✅ASAP_MPI_sync

MPI_Sync 版相對 Pthread 精簡，具體做法是將該輸入的資料輸入後，開始跑類 Dijkstra。因限制一個點就是一個 Process，因此在吃輸入檔時，直接把點與 rankID 做對映，如此一來每個 Process 的 ID（0,1,2,3...）就代表點的編號。進入 all_pair_shortest_path（ ），用迴圈中的 j 判斷是否為 rankID（是的話代表此輪是自己，自己跟自己是 0 所

以直接進下一輪），接著判斷相鄰陣列是否為 MAX（有無邊相連），有的話則更新 needUpdated [ ]（此陣列儲存自己點的資訊），再用 MPI_Sendrecv 將此陣列傳到 neighbor [ ]，供鄰居判斷是否須更新共享陣列（result）裡面的值。最後使用 MPI_Barrier、MPI_Allreduce 收集 localDone、allDone 參數，判斷是否要繼續進行。輸出部分是用 MPI_Gather 到 output [ ]，供 rankID=0 之 Process 進行寫檔動作。

✅ASAP_MPI_async

Async 版與 Sync 版基本概念不變，唯在判斷全域終止的方式有些不同。程式用 localAdjancy [ ] 儲存輸入檔內容，並多了 token [ ] 用以實作 Dual Ring Termination Algorithm。all_pair_shortest_path（ ）以 while 迴圈內的 MPI_Recv 持續接收資訊，MPI_ANY_SOURCE 接收任何來源，而來源資訊用 MPI_ANY_TAG，包含三種可能，CALC、TOKEN（black/white）、TERMINATE。首先跑個迴圈，判斷是否當下 i 值為 rankID，若是則繼續下一輪（此步驟與 Sync 版類似），接著判斷若當下 i 值大於 0（代表有其他點與之相連），則針對與之相連的點，計算區域最短距離，並將其結果送給 while 迴圈的 MPI_Recv 進行後續動作。與此同時若發現該點（該 Process）小於當前 process 之 rankID，則當前 Process 的 token 須改為 black（代表之前的 Process 還沒被更新，整個迴圈需再執行下一輪）。至於三種 TAG 所包含的狀況簡述如下：

1. CALC：若全員最短路徑 allSource [ ]（內放全圖的最短路徑）

2. TOKEN：先檢查 change 是否為 true（代表之前有更新過），之後再依照前述將 token 設為 black 或保持不變（rankID 是否為 0 分別討論），並重新一輪 Tag 為 CALC 的計算。
3. TERMINATE：

   若 rankID 不等於 vertices – 1（代表最後一個，因為接收者計算式為 rankID+1 % numOfprocess，若為最後一個，則表 rankID=0 的接收者），在 Send " TERMINATE " 的 Tag 後離開迴圈。

2. Design.

Strong scalability、*Performance profiling*、*Others*

✅System Spec.

課程提供之設備：

Intel(R) Xeon(R) CPU X5670 @ 2.93GHz、96GB memory、 2 x 6-core

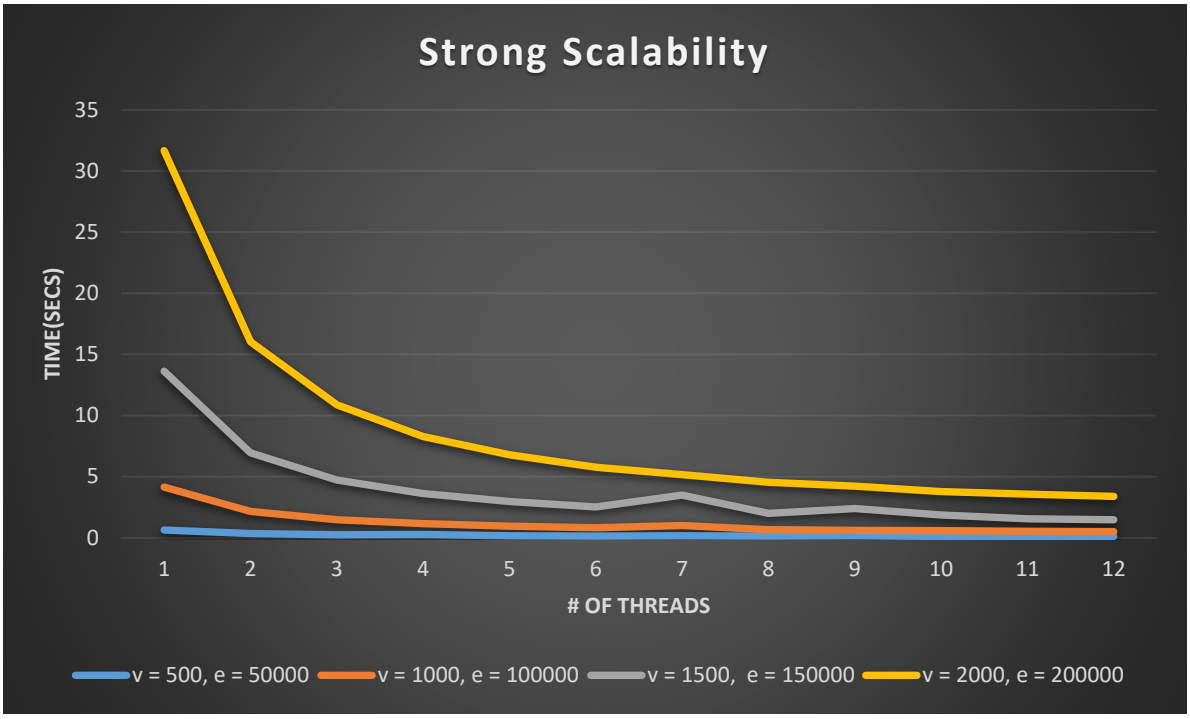
✅Performance Metrics

Pthread 部分採用 gettimeofday （ ）進行 Computation、Join、I/O 等時間之紀錄，單位為毫秒。

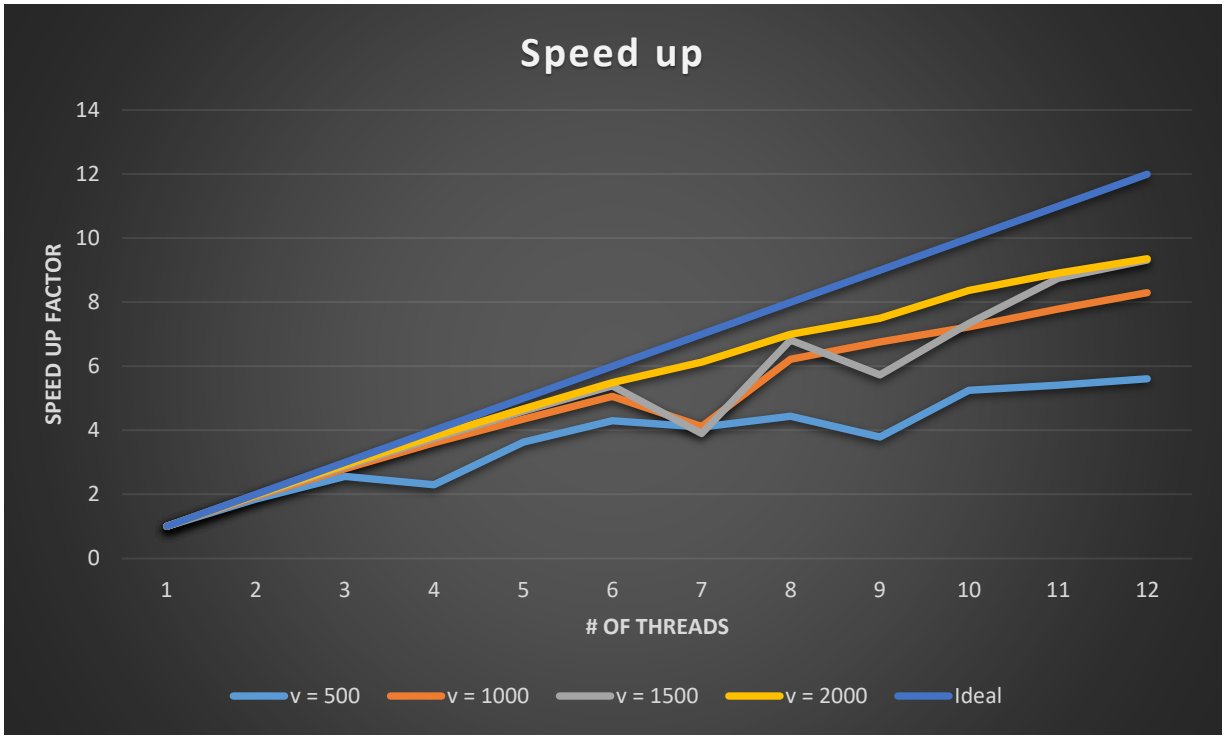
MPI 部分採用 MPI_Wtime()進行 Communication、Computation、synchronization(or token)、I/O 等時間紀錄單位，為毫秒。

其餘部分將與圖表一併解釋

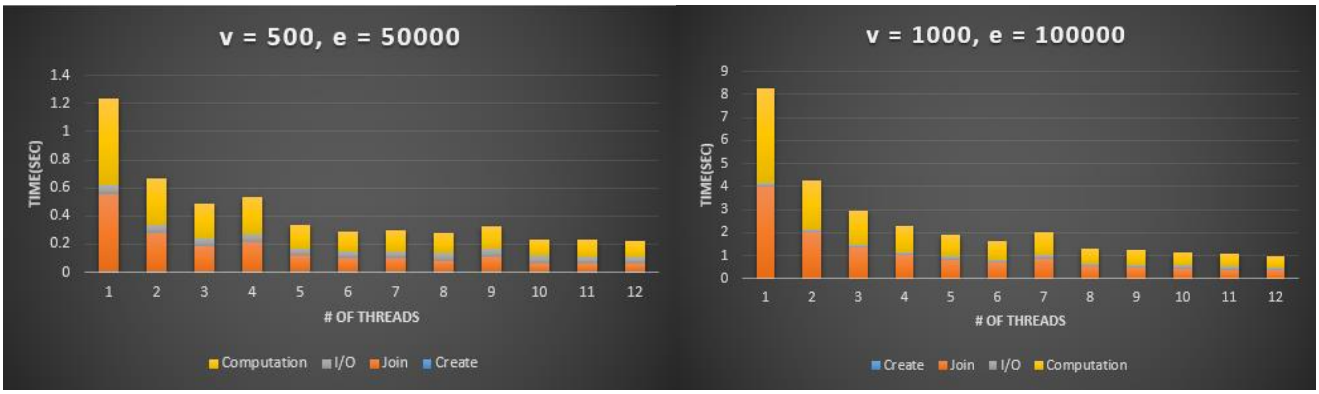
✅Strong Scalability - Pthread

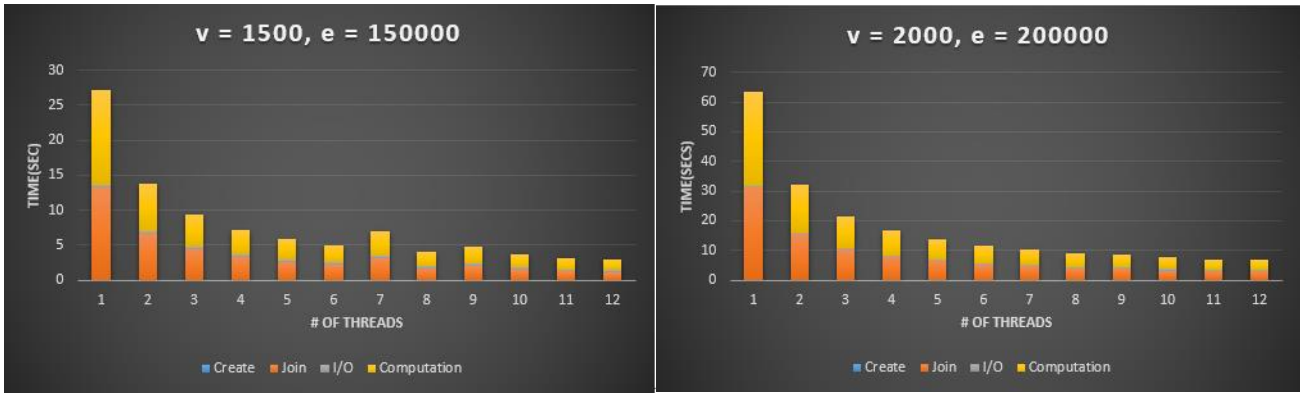


圖為 Thread 數 1 到 12，input size 從 500(vertex) / 50000(edges)到 2000(vertex) / 200000(edges)的執行時間變化。可看出，Pthread 版執行速度很快，縱使點數為 2000，邊數達 200000，其執行速度依然理想。下方 Speed up 圖亦可顯現，測資越大，Speed up 越接近理想值。



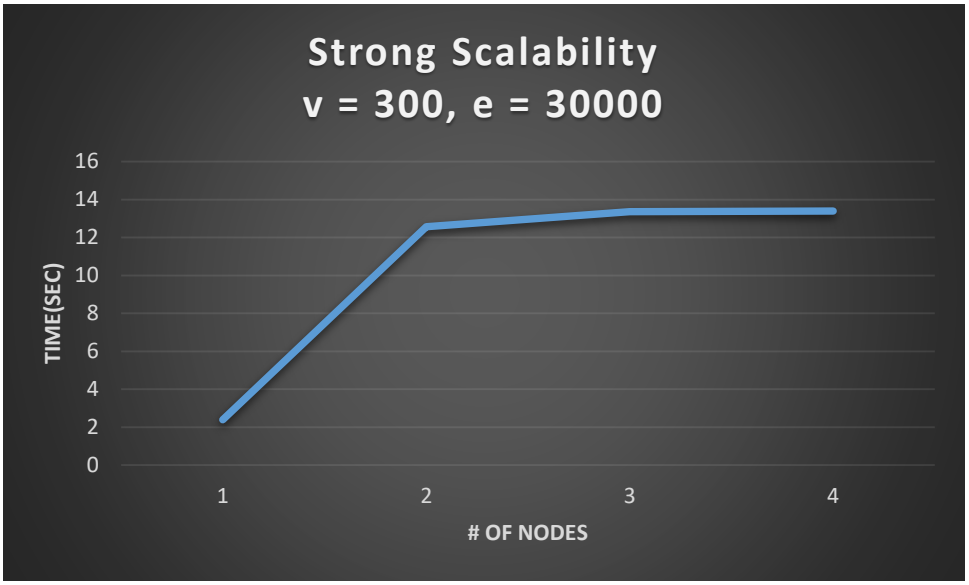
✓Time distribution - Pthread





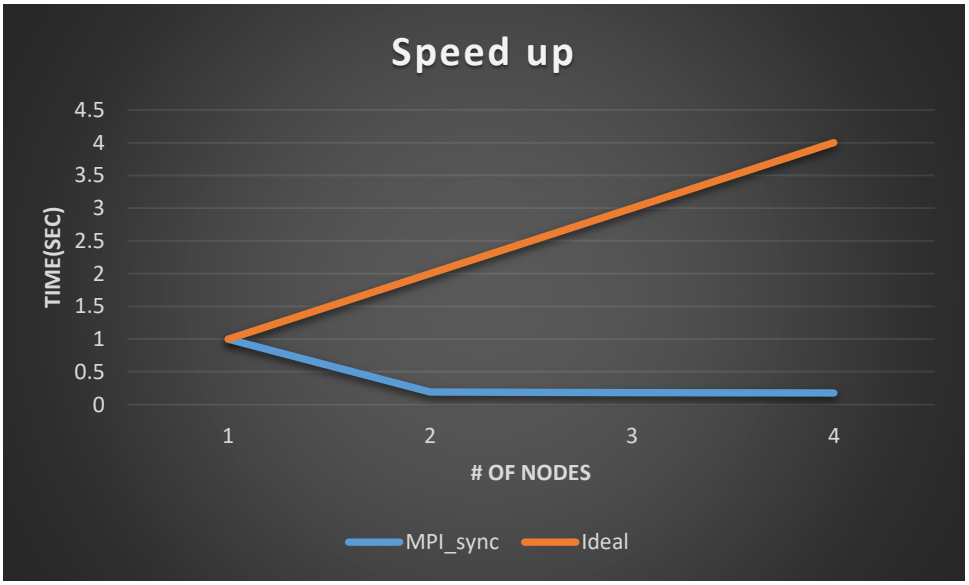
Time distribution 的圖很容易看出，Pthread 版的 bottleneck 多在 thread 的 Join、Computation。當 input 不大時，I/O 在某些 Thread 數下可能會是 bottleneck，而當 input 增大時，Join、Computation 反而成為 bottleneck，由此可知，若是較簡單的計算（如 Dijkstra 只比較大小並更新），利用 Pthread 反而會成為執行時間的 bottleneck。儘管如此，Pthread 在各個版本的實作中還是有較高 C/P 值的。

✅Strong Scalability – MPI_sync

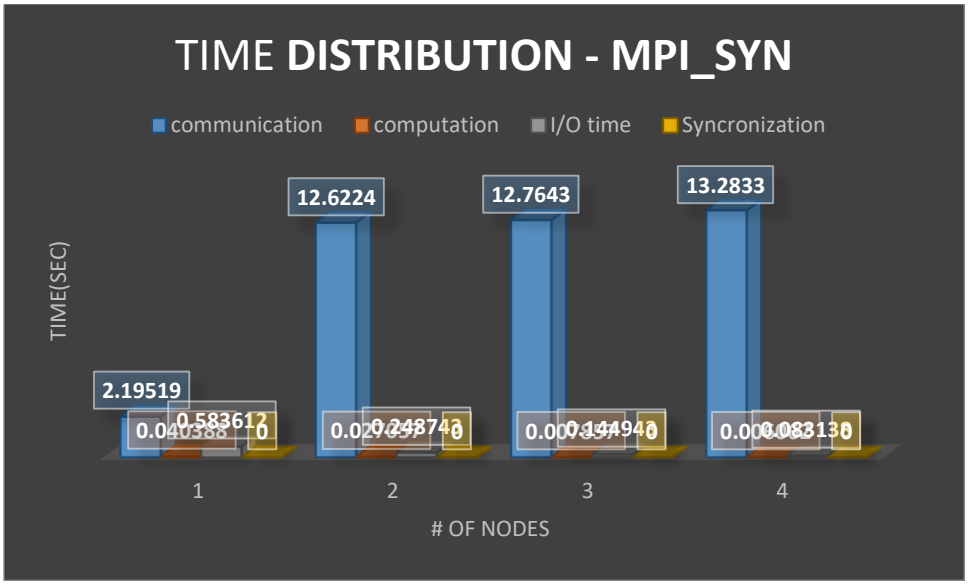


MPI_sync 在 Node 數增加同時，執行時間不減反增，主要是因為 Vertex centric 的架構導致 Process 間溝通頻繁，在跨 Node 情形亦加嚴重。稍後 Time distribution 即可看出此情形。

因此，Speed up 有別以往，其 Speed up 成反比情形，與 Pthread 版有很大差異。

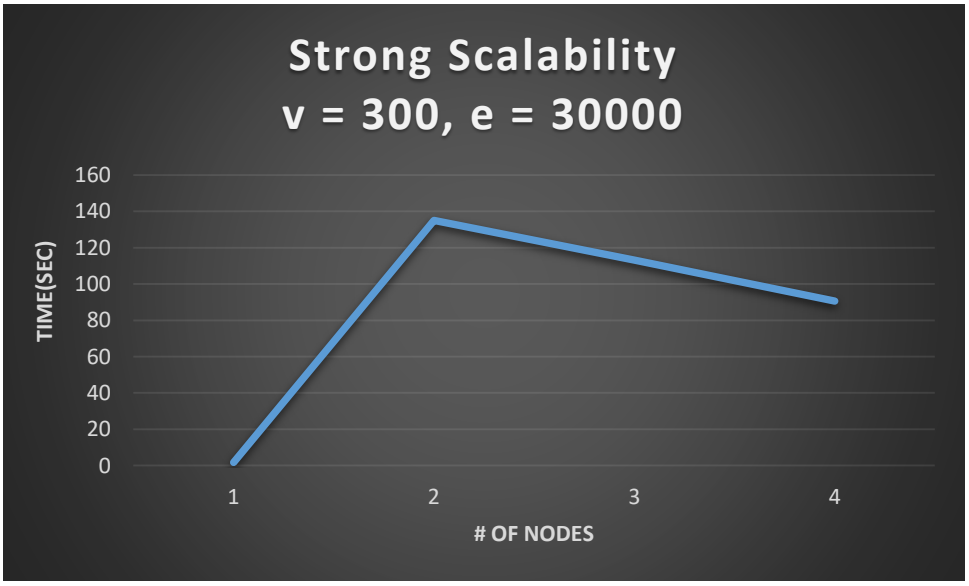


✅Time distribution – MPI_sync

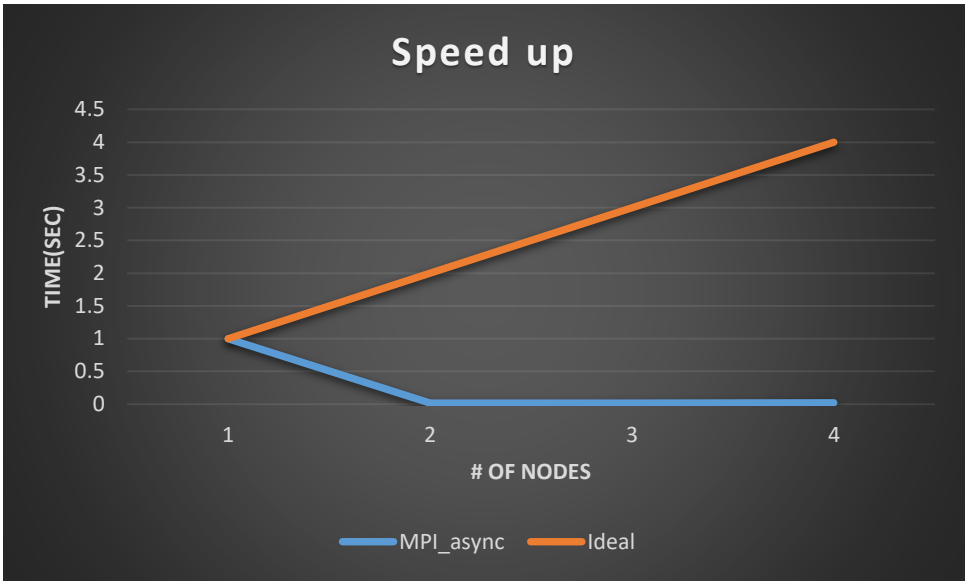


上圖可清楚看出，MPI_sync 版本下，不管是幾個 Node，Communication Time 都佔了很大一部分，表示 Communication Time 是主要 bottleneck，原因即為，圖上每個點就是一個 process，而點與點彼此需要頻繁溝通所導致。

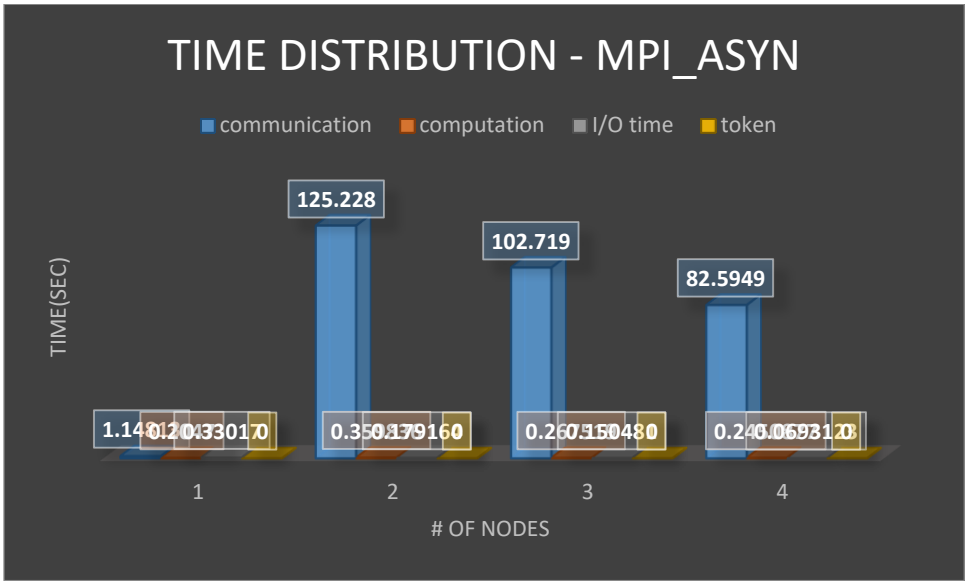
✅Strong Scalability – MPI_async



MPI_async 版本的 Strong Scalability 圖、Speed up 圖與 Sync 版本類似，都是隨著 Node 數增加，Performance 未必會變好的版本，推測可能原因與 Sync 相似，都是因 Process 數太多以致需要頻繁溝通所導致。



✅**Time distribution – MPI_async**



此版本與 Sync 亦類似，bottleneck 也是在 Communication，而其中用來當作 Barrier 的 Token time，在整體時間中佔的比例並沒有很多。總體而言，MPI 最主要的 bottleneck 即是 Communication。

3. Experience / Conclusion

Pthread 採 Shared memory，因此有執行速度快等優點，然在使用 Pthread 時計算過程不宜太簡易，否則 Crate、Join 等過程會超越實際計算時間，而成為效能的 bottleneck。

因採 vertex centric，每個點都是一個 Process，造成 MPI 無論是 Sync 或 Async，Communication time 皆為其效能 bottleneck。MPI 兩個版本在跨 Node 溝通時亦困難，導致 Speed up 隨著 Node 數上升，呈現下降趨勢。

本次作業著實學到了許多新東西，其中印象最深刻的是 Dual Pass Algorithm，在實作中其實要考量許多面向，也遇到了不少的問題，幸好受惠同學耐心地協助才得以完成此次作業，