CS542200 Parallel Programming Homework 3:

Single Source Shortest Path

104062703 曾若淳

Implementation

PThread

由於沒有weight < 0 的邊，所以我選擇時間複雜度是 $O(|V|^2)$ 的 Dijkstra 演算法，比複雜度是 $O(|V|^2)$ 的Bellman-Ford 快。
而 Dijstra 的最短路徑算法，必須分iteration 做最短路徑的更新，所以我能平行加速的部份是，每個 iteration 內的兩個地方：

一、找出還沒看過又是和source vertex 路徑最短的點 v

二、更新source vertex 到 v 的neighbor 的最短路徑

這兩個可平行的地方，也是有順序的：必須做完一、找出路徑最短的點，才能做二、。因此可平行的地方都必須重新用 pthread_create 給定數量的 thread，thread 做完task後就要做pthread_join。

MPI

因為每個process 代表一個vertex，而且只有local view，所以我選擇用 Bellman-Ford 演算法，因為每個process 只需要存自己和source vertex的最短距離以及neighbor 的weight，就能各自平行計算了。

Synchronous

按照Bellman-Ford 演算法，分iteration 做更新。
在每個iteration裡，每個process 會用source 到自己的最短距離 d_self + w[self][neighbor] 來計算 source 到 neighbor 的最短距離d_neighbor，並將d_neighbor 用MPI_send 傳給 neighbor。接下來，每個process 會收到來自neighbor 計算出的它的最短路徑，比自己紀錄的小就更新，假如做了更新就表示還沒做完done = 0，最後會用MPI_Allreduce on done 來偵測是否有process在這個iteration 做了更新，沒有的話就結束。

Asynchronous

對Bellman-Ford 算法做了點修改，改成只要有neighbor 傳message 做比較更新最短距離，而不需要分iteration。因為不分iteration了，所以必須偵測什麼時候該停止，實做方法是，每個process 任何時候都可以收message，而message 分兩種：

一、來自neighbor，更新source 最短距離的message

如同synchronous版本，比較neighbor算出的是否比自己紀錄的還要小，假如比較小就更新，然後計算neighbor 的最短路徑，並傳更新最短距離的message 給neighbor。

二、來自rank-1，偵測停止的dual-passed-ring message

dual-passed-ring 為了偵測停止，必須按照一個順序傳中止message，我使用從rank 小的process 傳給下一個 rank+1 直到到後一個。

在實做時，每個process 還另外用了一個變數 color 表示狀態，color = 白色 = 1，表示process 完成，color = 黑色 = 0，表示process 傳了更新最短距離的message 給 rank 比自己小的process。

而收到dual-passed-ring message時，會拿出message內的token，然後計算new_token = token & color，並往下傳new_token （即：假如自己狀態是完成，那會原封不動將token往下傳，假如自己狀態是有傳task 給rank 較小的process，則會傳黑色的token）

dual-passed-ring 必須要由rank 最小的process 開始傳，每個process 另外還有個變數 passed 紀錄目前是第幾次收到dual-passed-ring message，passed=1 表示才傳過一輪，可能還有process 沒做完，當passed=2 而token 又是白色時，表示所有process 都做完，並且rank 小的已經停止了，因此這個process 傳給下一個rank+1 的process 後就能安心停止。

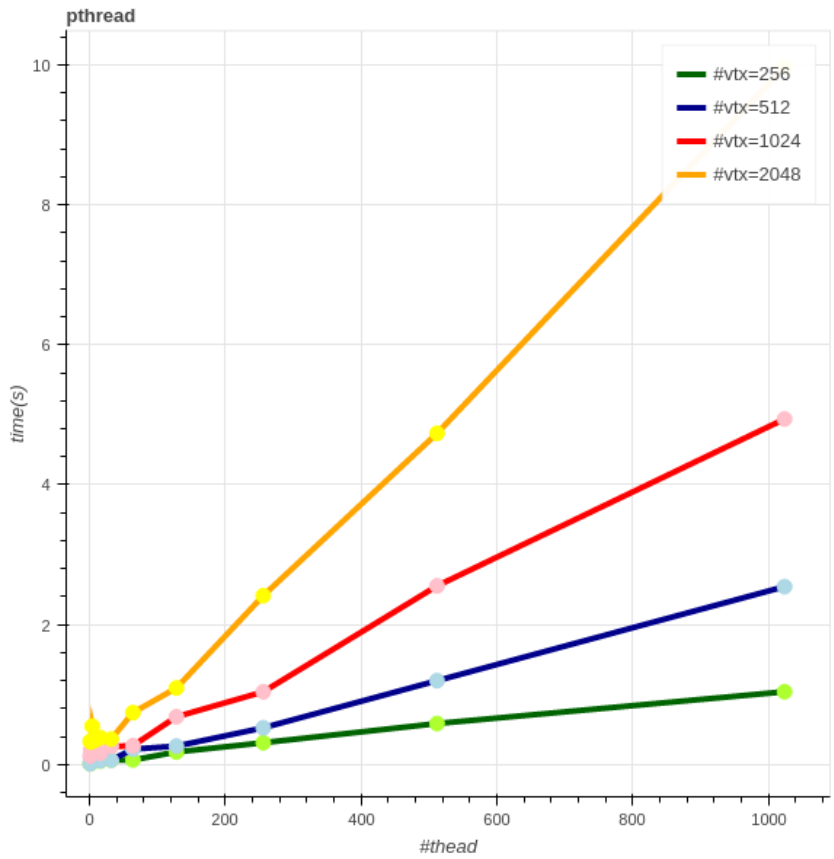
一旦 dual-passed-ring 的token 變黑色就必須要重頭開始傳，我實做reset 的機制是假如任何一個process 收到黑色token 那passed = 0表示重傳，而token 顏色在傳回rank 最小的process 時會被reset 成白色。

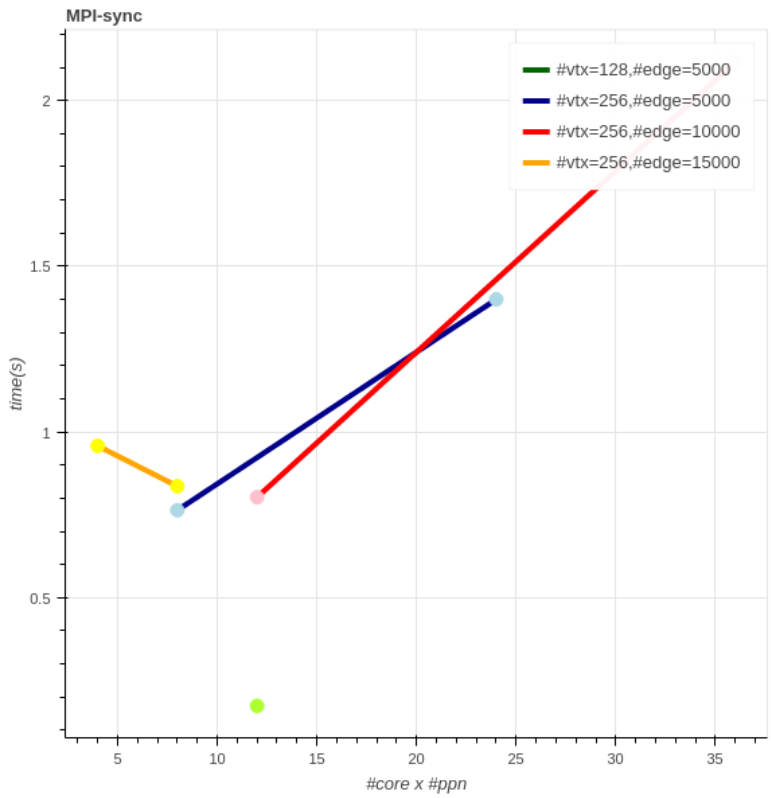
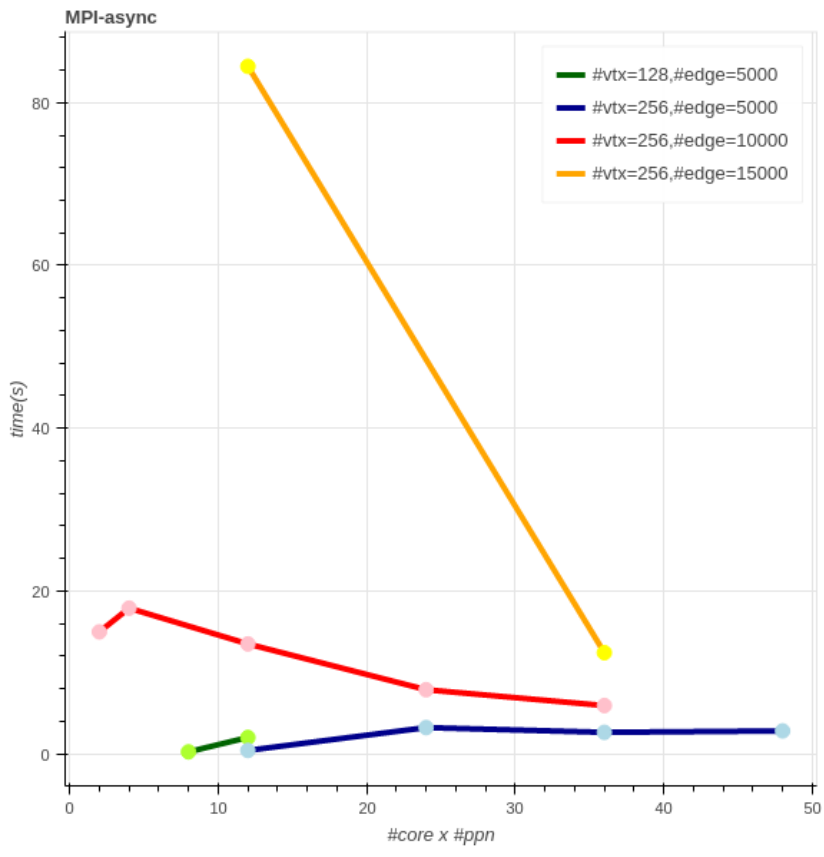
最後，當所有process 都結束後，會將parent（backtrack 印出路徑用的資訊）傳給代表source vertex 的process 做輸出。但假如source vertex 還沒跳出能收任意 message 的階段將會出問題，所以我的實做方法是另外再創一種message 專門收集parent，並讓source vertex 晚一點結束。

Performance Analysis

實驗環境為課程提供的 batch cluster：1 sequencer node + 10 worker node，每個node 為2x6 Interl(R) Xeon(R) 的CPU 配備96GB 的memory 和 2TB HHD。

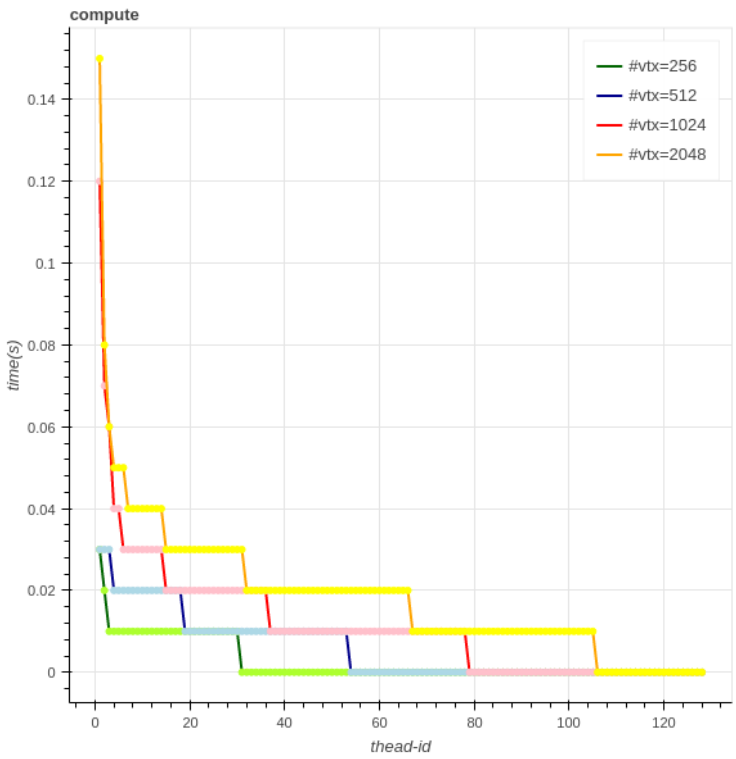
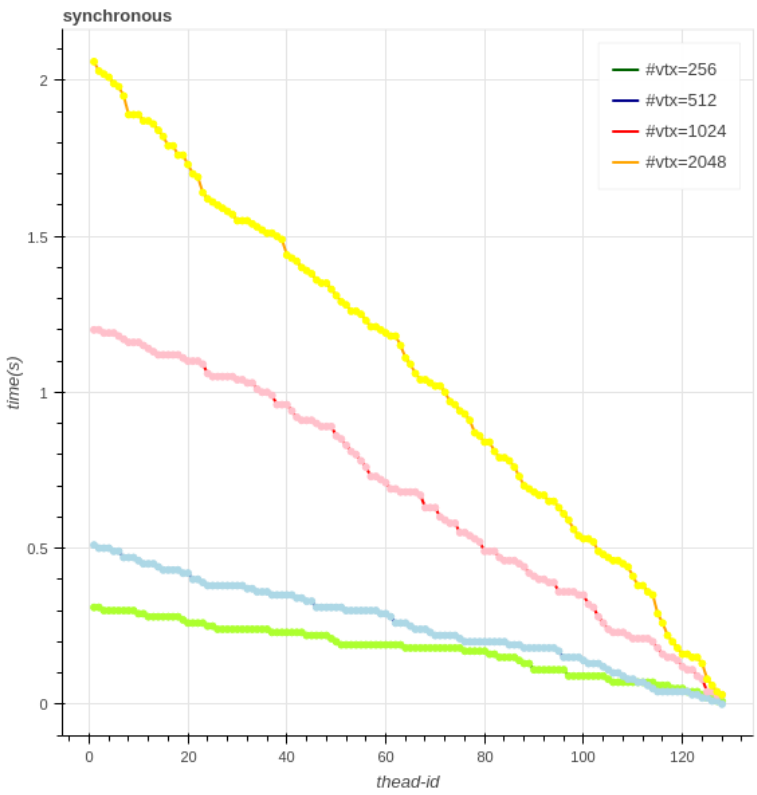
Strong Scalability

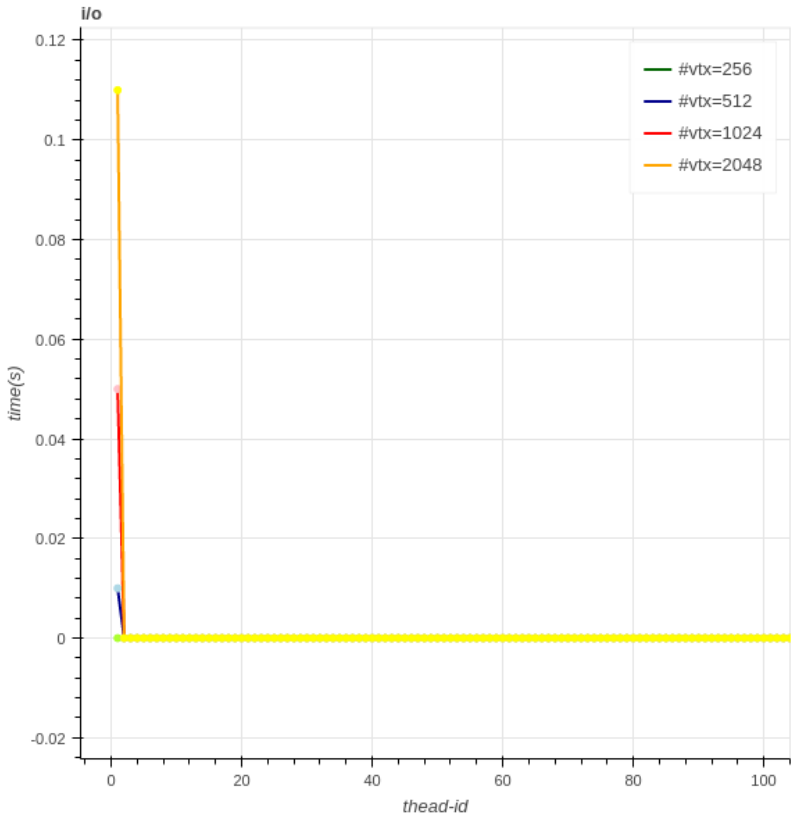
PThread 在相同problem size下，是三者之中最快的，但是scalability 很差。	

MPI synchronous 版本在跑實驗時許多組(#core, #ppn) 無法執行，但唯一能看出增加#MPI-process有用的是problem size最小的情況。	
MPI asynchronous版本是三者之中跑得最慢的，但是在增加#MPI-process 能明顯看出有幫助。	

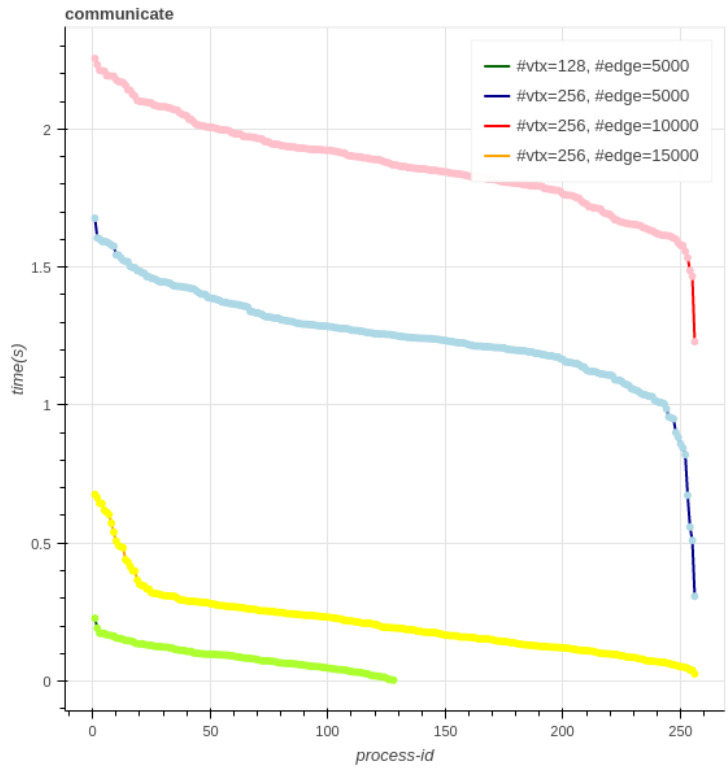
Time Distribution

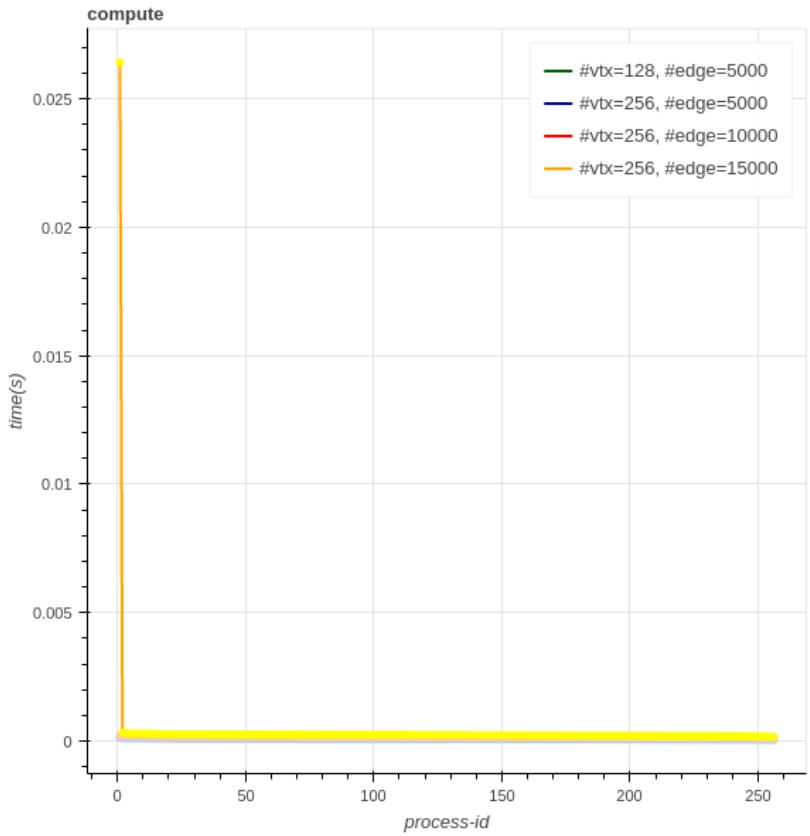
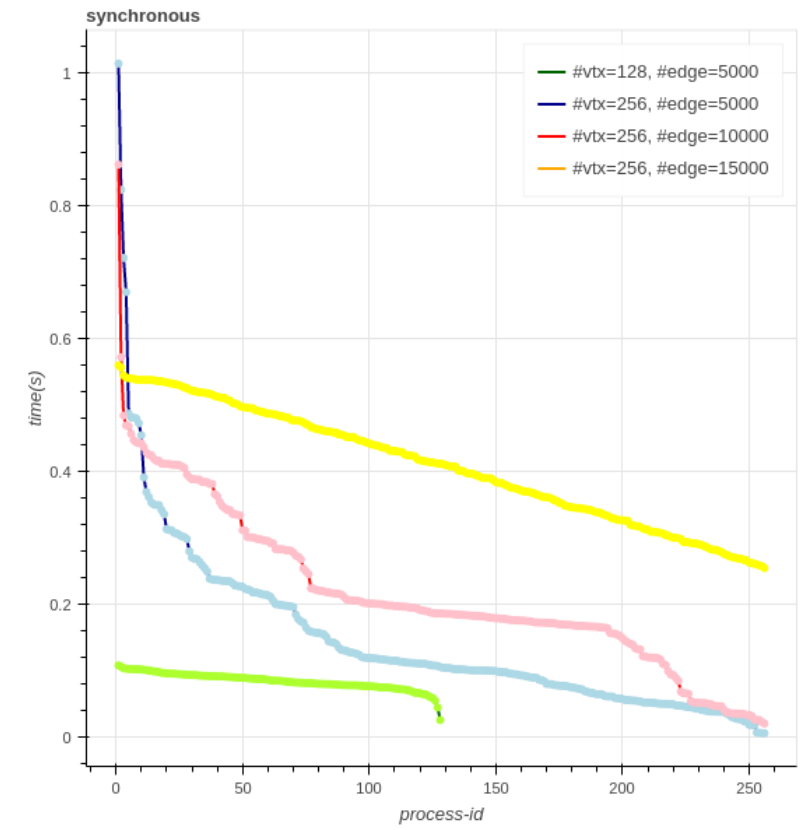
PThread

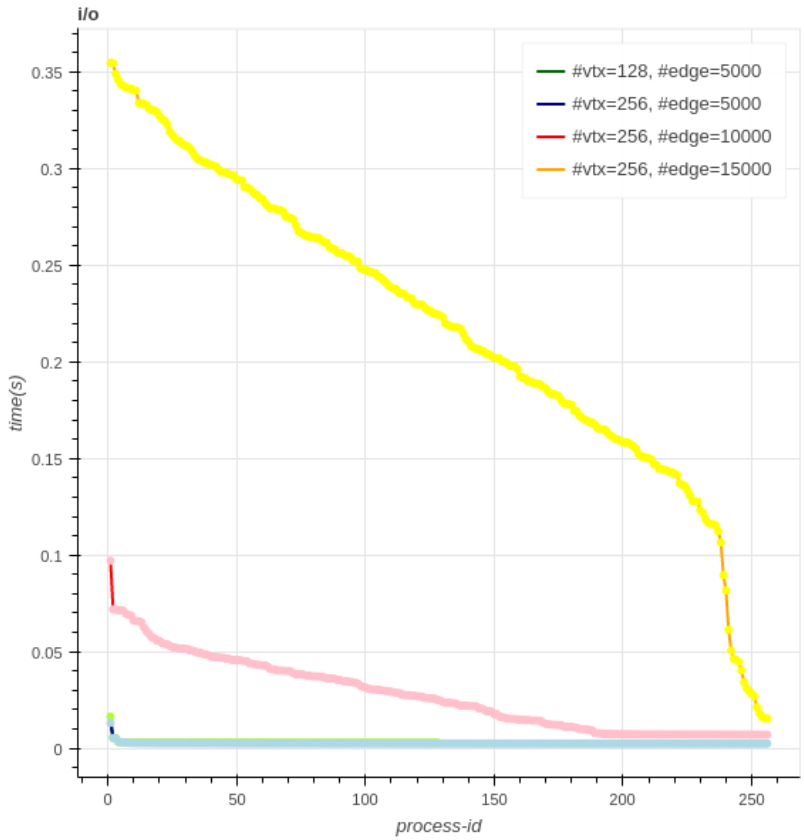
PThread版本的compute 分佈滿不平均的，而總體執行時間會由最慢的thread dominate，所以這也許是難以得到strong scability的原因之一。	
難以達到strong scalability 原因之二是synchronous 時間的差距problem size越大，差距越大。	

由於PThread 版本的I/O 是在 parallel region 外只由 main thread 做，這部份對整體執行時間影響較少。	

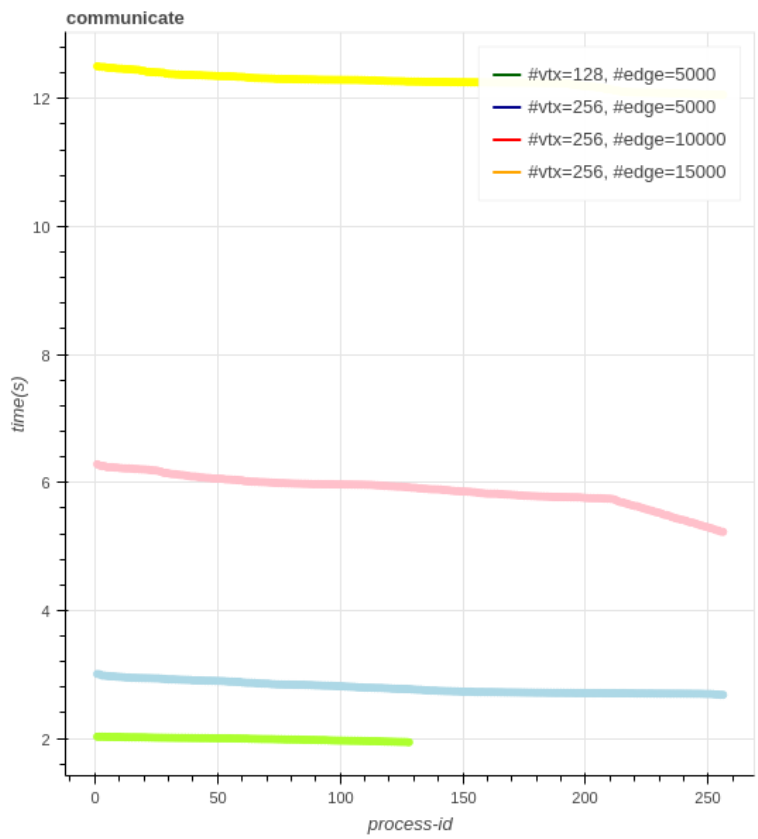
MPI Synchronous

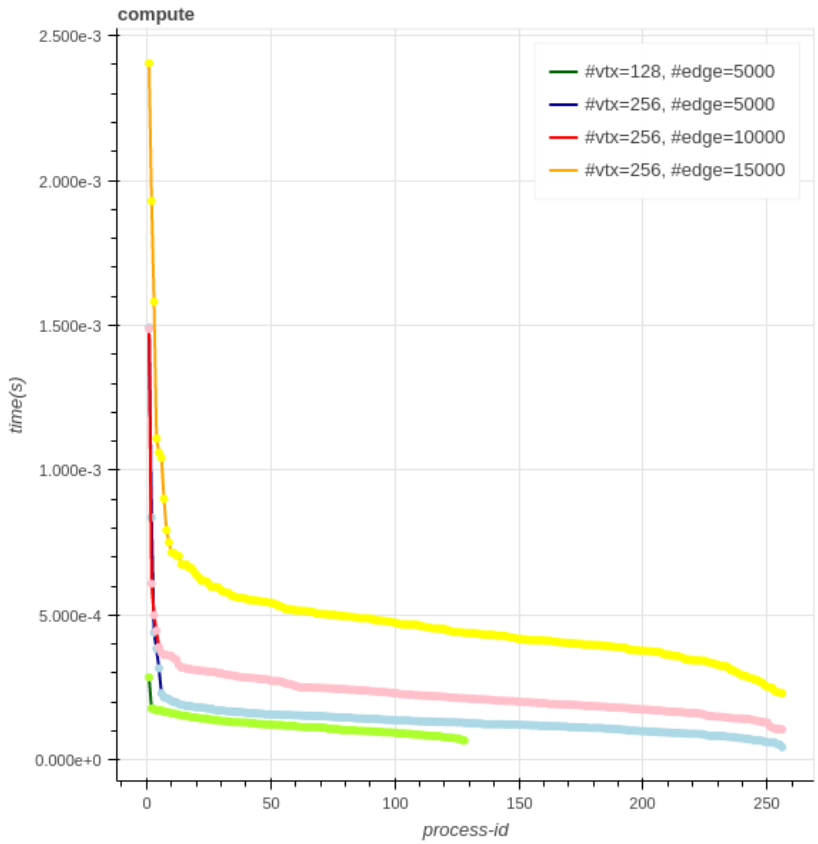
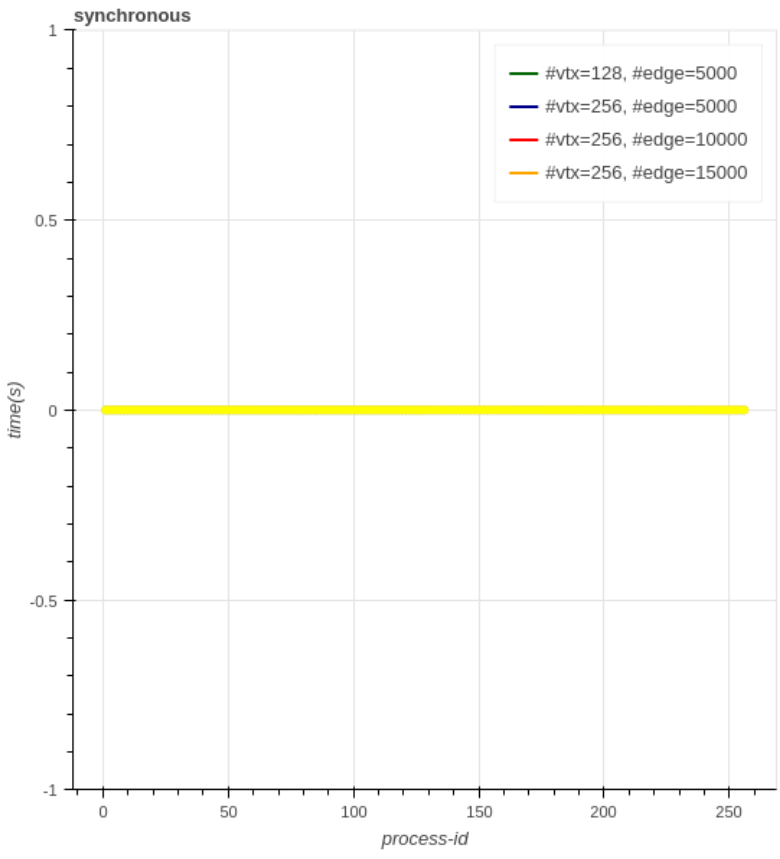
MPI-Synchronous 版本花在 communication的時間大致算平均。	

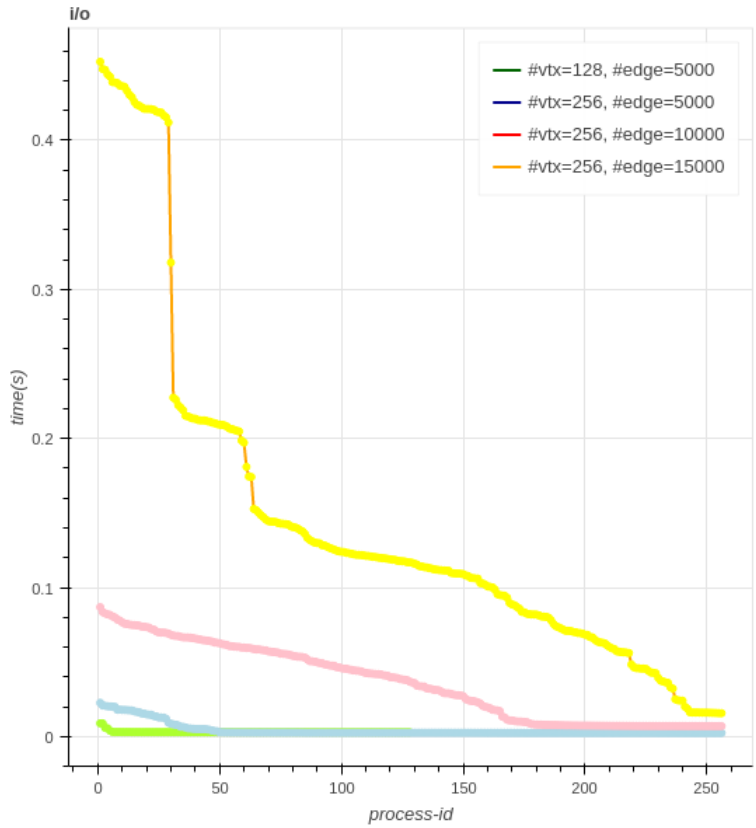
MPI-Synchronous 版本的compute 時間分佈，除了少數一兩個 process 計算特別久以外，其他都很快，猜測原因是只有少數的 node 有很多邊。	
MPI-Synchronous 版本的由於每個 iteration 結束前必須放置barrier 確保每個process 都執行完畢，所以可預期分佈不會差太多，在 #vertex 相同但增加#edge時（假如平增加的話）synchronous時間分佈應該會越來越平均。	

MPI-Synchronous版本I/O時間在problem size分佈比較不均勻，可能是因為我沒有使用MPI_IO function的關係。	

MPI Asynchronous

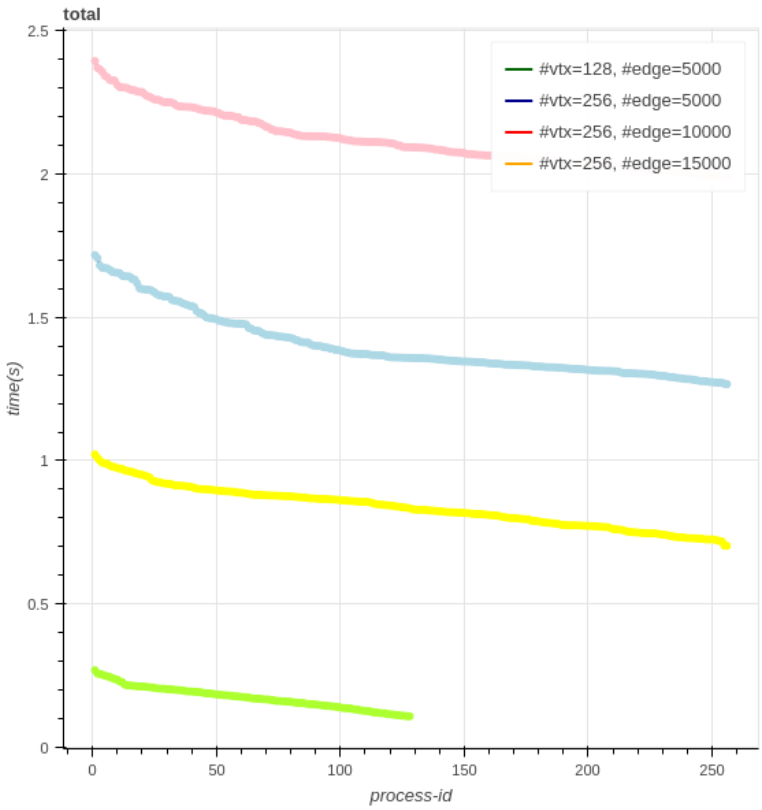
MPI-Asynchronous版本的communicate時間非常平均，因為每個process 都可以接受任意數量的message，而且總是有dual-passed ring message在傳。	

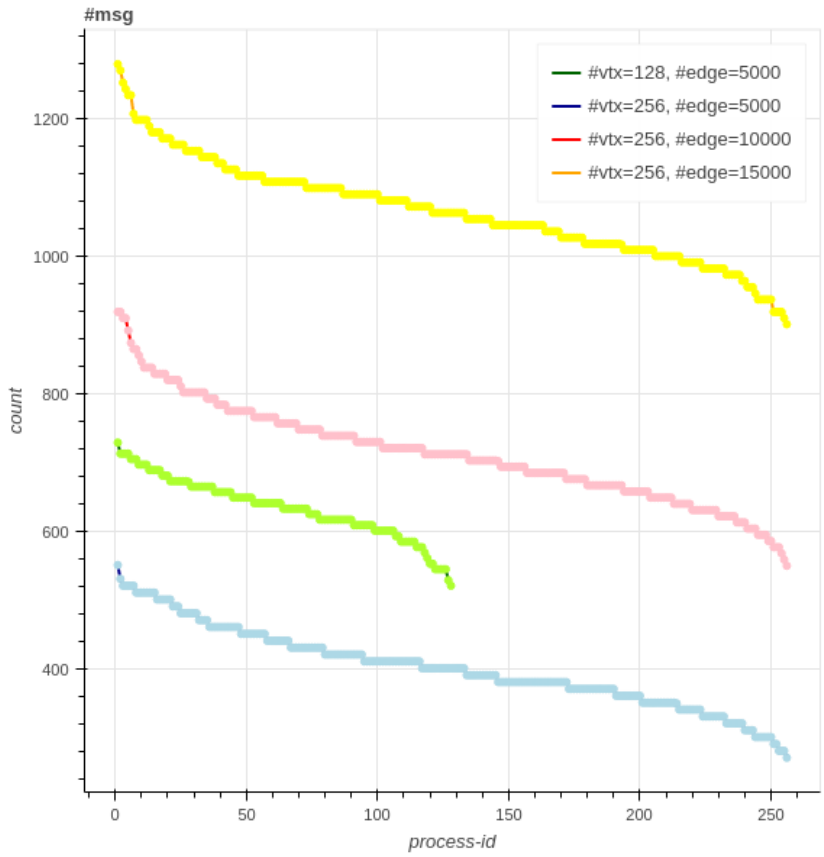
MPI-Asynchronous版本計算時間分佈和MPI-Synchronous版本一致，也猜測原因是只有少數node有很多edge的關係。	
由於MPI-Asynchronous版本沒有barrier，所以不需要做synchronous。	

MPI-Asynchronous版本的I/O時間也和MPI-Synchronous版本觀察到一樣的現象，在增加problem size分佈比較不均勻，也猜測都是因為沒用MPI_IO function造成locking的問題。	

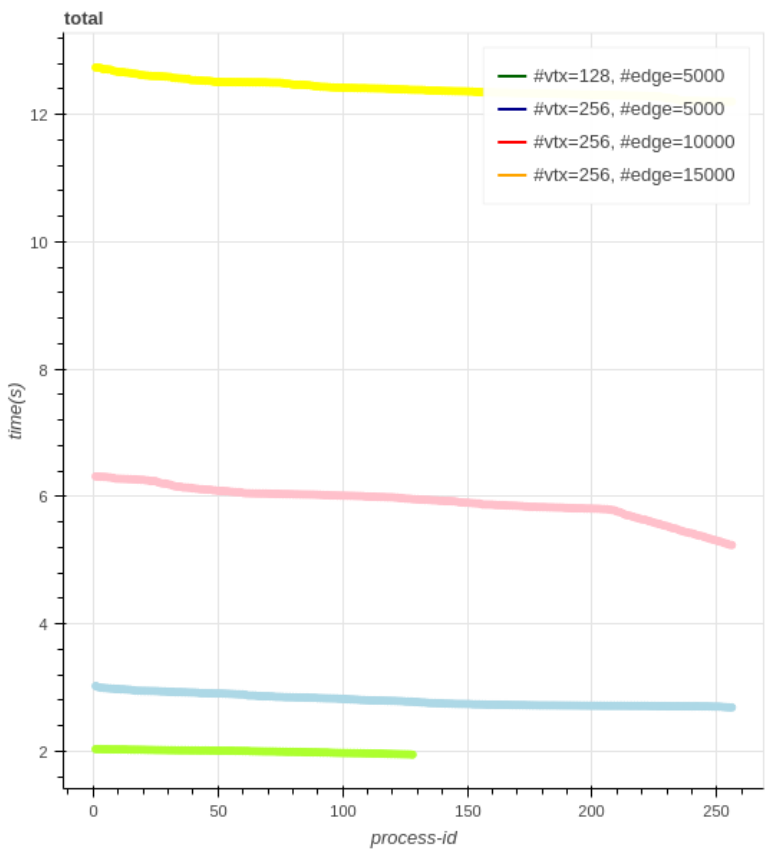
Load Distribution

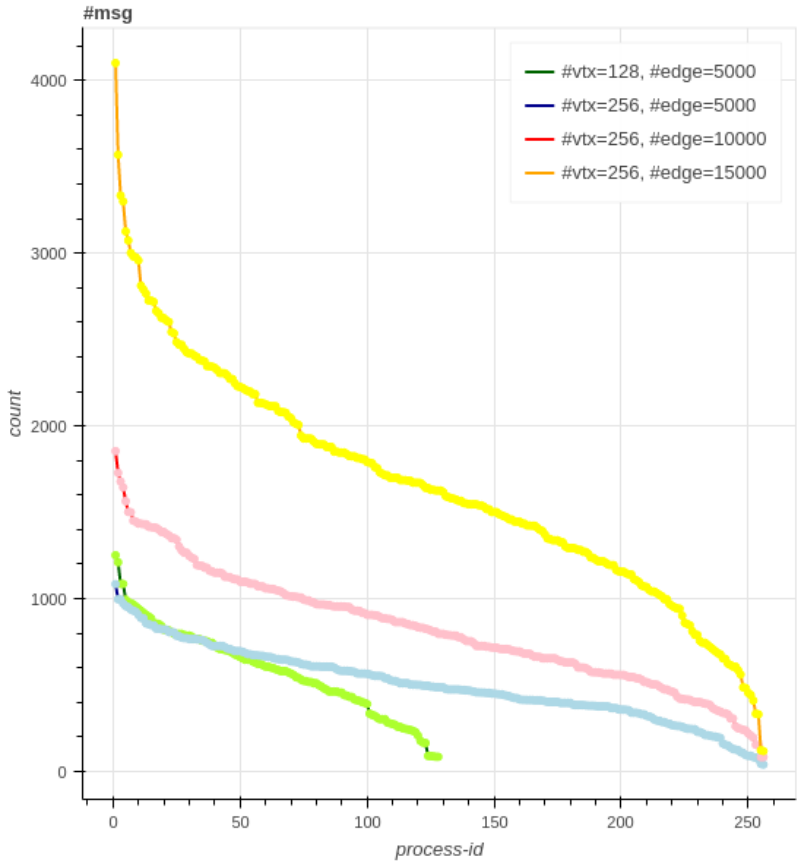
MPI Synchronous

MPI-Synchronous版本總體執行時間分佈非常平均，原因是有使用barrier，因此會由最慢的process dominate。	

MPI-Synchronous版本的message數量大致相同，但node 邊的分佈差別很大，所以我認為dominate message數量的主要是MPI_Allreduce這類barrier的message，而非compute message。	

MPI Asynchronous

MPI-Asynchronous版本總體執行時間差不多，符合預期因為必須要等到dual-passed ring傳過一輪確定可以中止才會按照順序一個一個中止，因此時間差可以預期非常小。	

MPI-Asynchronous版本的message數量差別很大，跟MPI-Synchronous版本比較，反應了兩件事：一、MPI-Asynchronous版本叫沒效率，需要比較多的message。二、MPI-Asynchronous版本的分佈和computation時間分佈大致一致：只有少量的process的message數量非常多或非常少，猜測是vertex 間的#edge 分佈導致的。	

Conclusion

很希望能用做更大規模的觀察，因為在這三者之中竟然是PThread版本最快，而最複雜的MPI Asynchronous版本最慢，慢了一個數量級。但是當#vertex 增加時，可能的#edge 是指數級成長，所以我認為應該在#vertex 非常大的時候才觀察到MPI vertex-centric 的好處。