平行程式 Parallel Programming – HW3: All Pair Shortest Path

Name：蕭子謦　　Student ID：103062372

1、 Design

(a) Pthread 的 version 是採用平行化的 Floyd-Warshall Algorithm，原因是 Floyd-Warshall Algorithm 比較好平行化，實作起來也比較輕鬆。而 Thread 的工作分配方法是參照 HW2 Mandelbrot Set, Static 的作法，從 Thread ID 開始，每次跳 Thread count 個，直到超出範圍則停止。用這個作法的原因是因為工作分配可以很平均，而且不需要任何額外的工作分配計算。

(b) Synchronous 與 Asynchronous 版本也是使用 Floyd-Warshall Algorithm，Synchronous 版的優點是易於實作，藉由每一回合的 Allreduce 可以很輕鬆得知所有 Node 的更新概況來決定是否完成。Asynchronous 版則相對於 Synchronous 版，在實作上面就非常複雜了，且時間上也和 Synchronous 相差無幾。此外，為了達到更好的效率，我使用的 Termination Detection 方法是利用 Spanning Tree 來回傳送來確保所有 Node 已停止更新。詳細實作方法請參考 4、Appendix (a) Asynchronous Implementation。

(c) Synchronous 版與 Asynchronous 版相比的話，很明顯光是 Synchronous 版易於實作就已經是很大的優勢，再加上兩者時間相比其實相差甚少。我覺得其實沒有必要選擇 Asynchronous 版實作，除非有必要性，像是整體的 Scale 非常龐大且本身 Node 與 Node 間就不易更新的狀況下，Asynchronous 就會有意義了，例如：Routing。

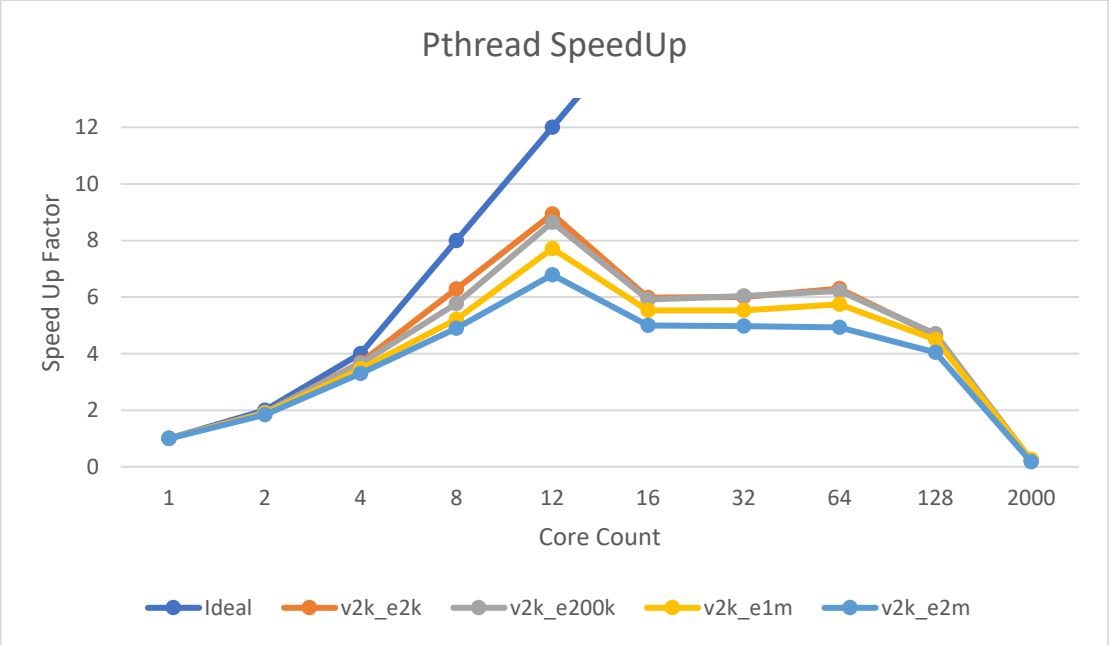
(d) Hybrid 版 – 不公開。

(e) 另外，為了讓程式的速度達到極致，我也有在 IO 上面做特殊的處理。由於 IO 速度緩慢，在運算上面仍會佔有一部分 Bottleneck，尤其是當 Call IO 的次數愈多時愈緩慢。因此我的解決方法是，在讀檔的時候會先將一整個檔案讀到 Stringstream 後再慢慢讀取使用，寫檔時則是先將所有東西都寫到 Stringstream 後再整個直接 Dump 到 File 內，實測出來發現當 Edge 規模有破百萬時至少能夠節省 0.2~0.4 秒左右，這在 Pthread 版上，因為整體執行時間很少超過 3 秒以上，因此算是有滿大的優勢。

2、 Performance analysis

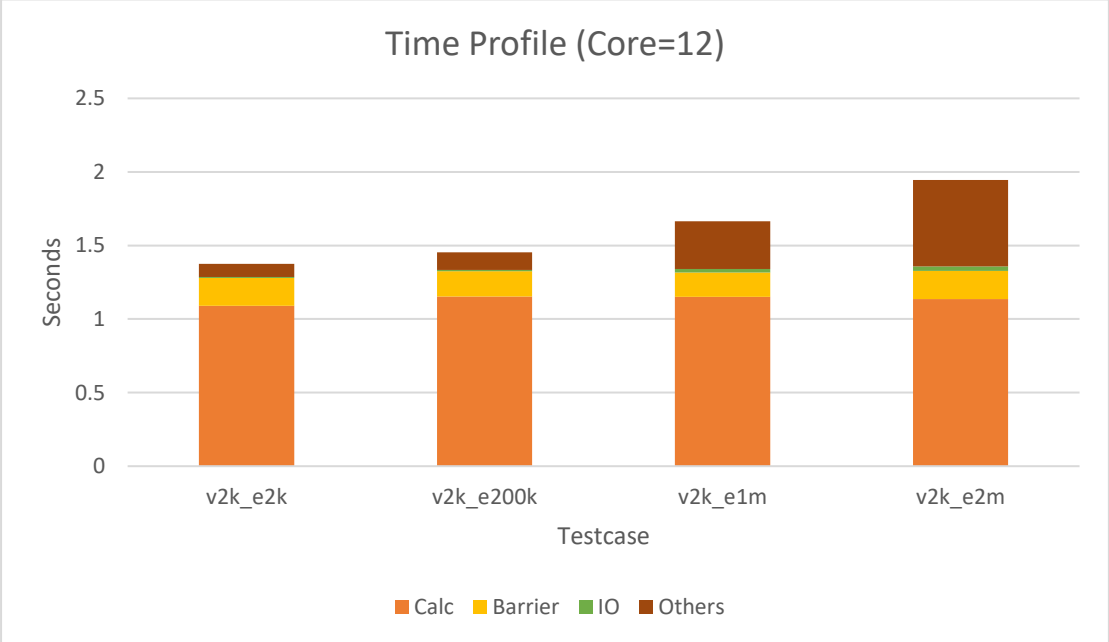
(a) Pthread Scalability

Pthread 使用最純的 Floyd-Warshall Algorithm 計算，由於只靠 Vertex 與 Vertex 間的資料做計算，因此本身複雜度只會跟 Vertex 的數量有關係，加上每個 Thread 運算之間不具有太多的 Dependency，如此可以達到很好的 Scalability。如下圖：

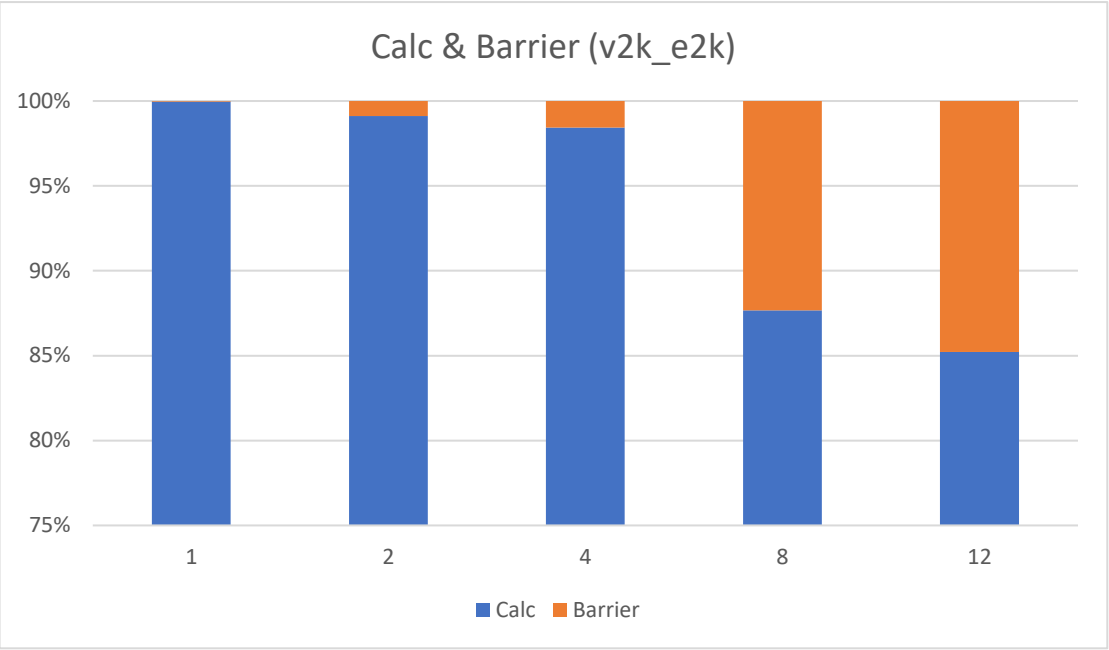


其中 v2k_e2k 代表 2000 個 Vertex 與 2000 個 Edge。以此類推

可以看到在突破 Physical Core (12) 前的 Speed Up 都還算可以，突破之後就會急遽的下降。另外，透過各種不同的測資可以發現，Pthread 的執行時間與 Edge 數量不大有關係，但是 SpeedUp Factor 還是會降一點點，原因其實就是在讀檔上面，Edge 愈多讀得愈慢。



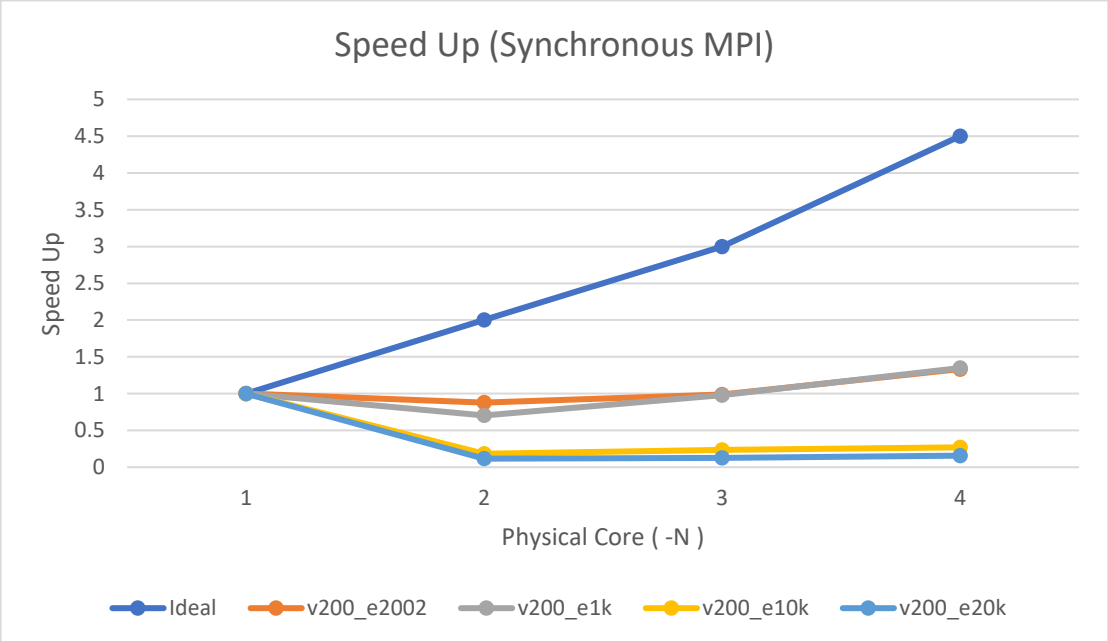
由上圖可以發現，在不同測資下 Calc (計算) + Barrier 時間幾乎都相同，唯一差就差在 IO 與前置處理與後處理 (Others) 上。由於在 IO 上面有做特殊處理，所以整體來講節省了很多 IO 的時間。



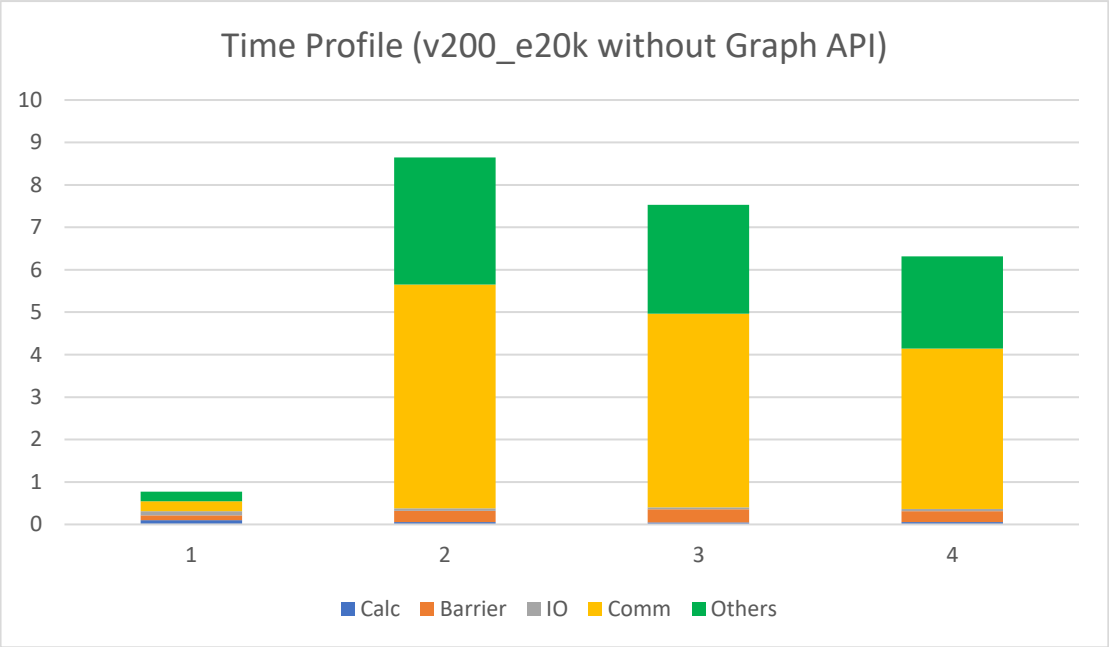
另外，在相同測資下，隨著 Core 數的增加，除了整體速度會變快之外，也可以很明顯看出 Calc (計算) 與 Barrier 之間比例會有大幅度的變化（注意垂直座標的刻度）。

(b) Synchronous MPI

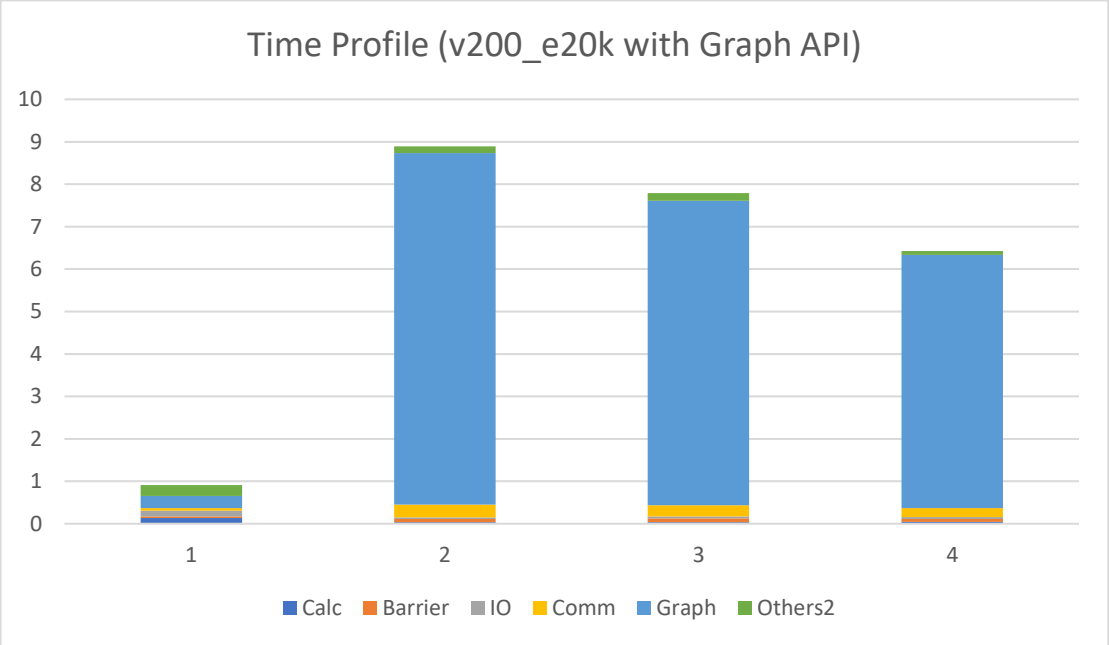
下圖為 Synchronous MPI 在各種測資下，不同數量的 Physical Core 下的加速情形。



很明顯可以看的出來，增加 Physical Core 數量似乎沒有任何改善，有些甚至更糟糕。由下圖可以很明顯看的出原因，主要是在於 Comm 時間上的增加，導致整體速度沒有提升反而下降。

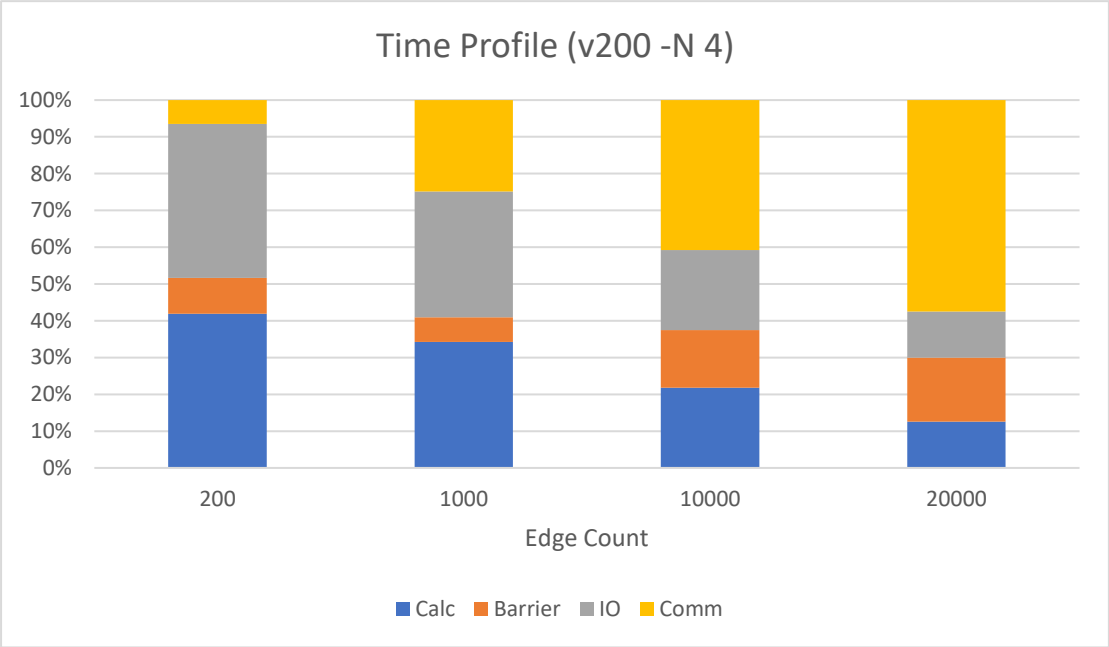


另外，根據小道消息指出使用 Graph API 可以提升速度，因此我也實際測試了 MPI 提供的 Graph API 功能，主要是使用 MPI_Dist_Graph_create()來建立 Graph 的 Communicator。



經實測後發現，Graph API 確實能夠大幅度降低 Comm 的時間，但由於建立 Graph 也會需要耗時，加上跨電腦的關係，有時不是很穩定，導致整體速度可能有些微變快，也有可能變慢，完全就要看運氣。

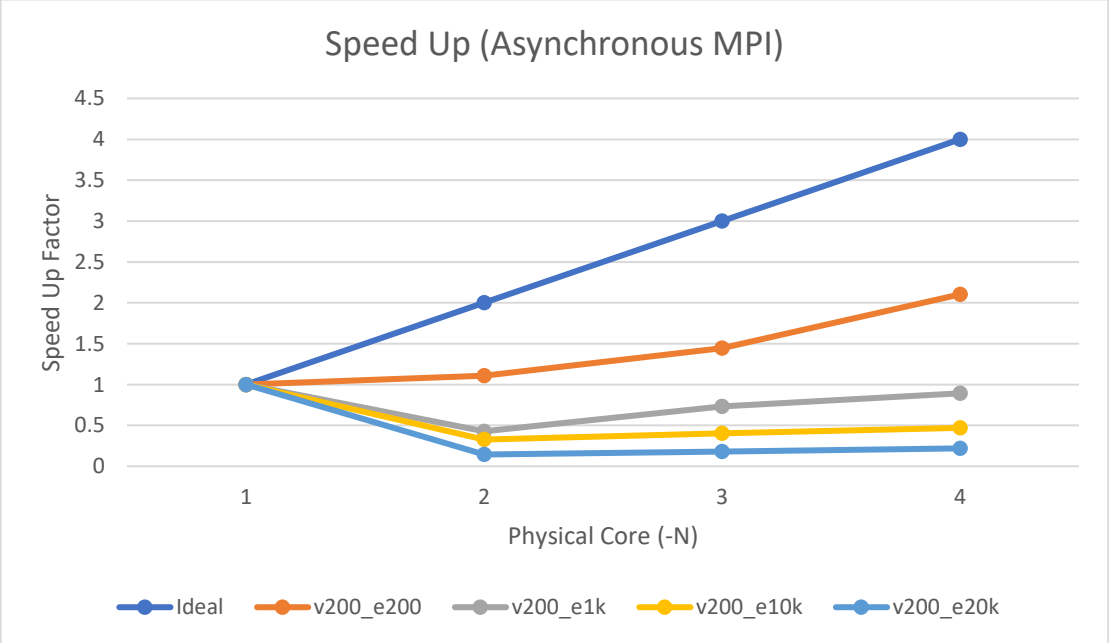
在相同 Node 不，不同 Edge 數下，由下圖可以看出，不同於 Pthread 版本 Synchronous MPI 版因為 Vertex-centric 的關係，速度會受到 Edge 數量的影響，原因是當 Edge 增加時，除了增加 IO 的時間外，Neighbor 的數量增加，Comm 的次數也會大幅增加，導致 Comm 的時間所佔的比例變多。



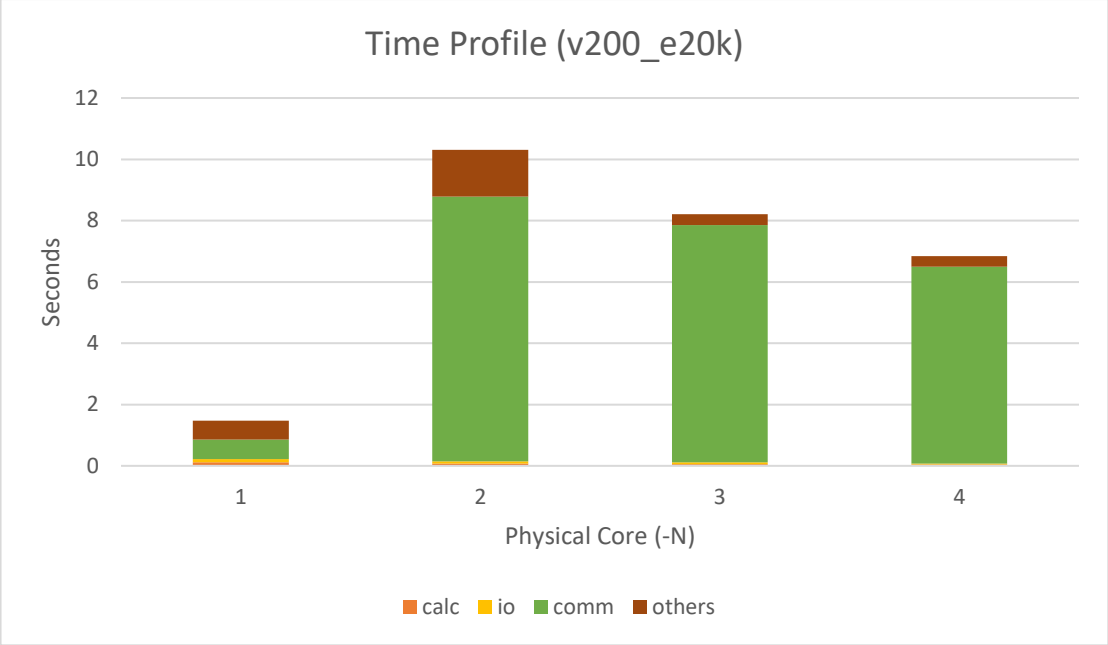
當 Edge 數量到達 20000 時，Comm 已經佔了大部分的程式執行時間。

(c) Asynchronous MPI

Asynchronous MPI 版也同樣使用 Graph API 實做。下圖為在不同 Physical Core 數量下的 Speed Up Factor。



由圖可知，與 Synchronous 版本相同的情形在此也有出現。



同樣 Graph API 的問題在 Asynchronous 也有發生，在此就不多作分析，將他列入 Comm 裡面。同樣可以發現幾乎看不到優化後的 IO。

3、 Experience and conclusion

(a) 藉由這次作業讓我了解到，實際在計算完全分散式的最短路徑時的工作其實是非常複雜的，像是生活中常見的網路 Routing 之類的應該也都是在利用類似的算法進行計算。這次因為 Spec 上面有特別限制只能與 Neighbor 溝通，我才特地自己想了一個 Spanning Tree 的 Termination Detection 方法，結果後來助教又說 Termination Detection 不限制只能與 Neighbor 溝通，說實在其實滿難過的。雖然速度上與 Duel-ring 相差無幾，但是至少我證明了我自己想的辦法是行得通的，還是有一些些成就感。

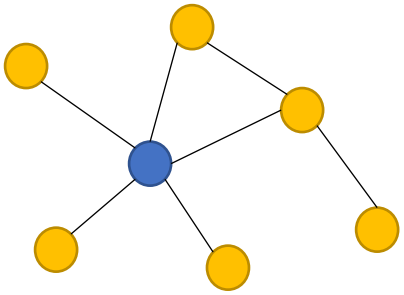
(b) 這次遇到的問題還是一樣，MPI 的 Document 寫得有夠爛，很慶幸這次應該會是最後一次翻閱 MPI 的 Document 了。

(c) 我覺得 Apollo 的時間限制可以再降低一點，現在設定的 30 分鐘說實在有點久，一般作業程式也很難跑到 30 分鐘還沒出來的。甚至截止日前幾天還大塞車，我丟一個 6 秒就可以完成的 Task 卻等了將近 10 分鐘才結束。根據可靠消息指出，有發現有些同學利用 batch 大量提交 Task，而且每個 Task 都將近 20、30 分鐘才結束，這樣其實滿影響其他同學使用資源的權利的，所以希望助教如果有看到這段的話可以把 Apollo 的時間上限調低。

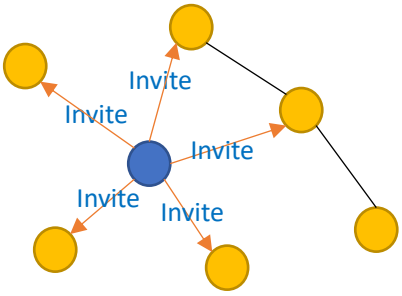
4、 Appendix

(a) Asynchronous Implementation

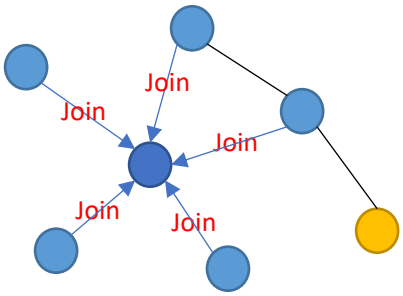
Spanning Tree 的創建過程如下：



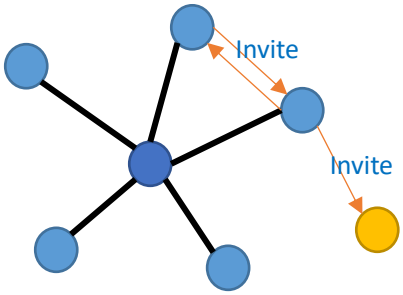
1、Graph 初始狀態



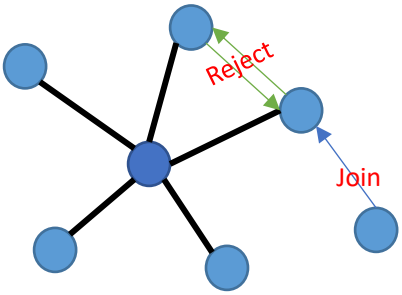
2、Root 向外發送邀請



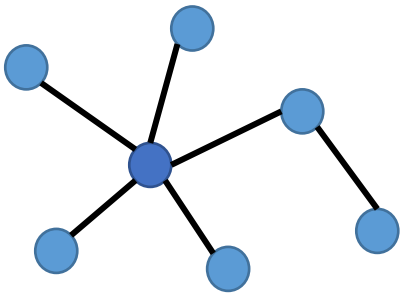
3、Neighbor 收到邀請後會檢查自己是否已經有 Parent，無則接受邀請



4、同時剛加入 Tree 的 Node 也會向所有除了 Parent Node 以外的 Neighbor 發送邀請

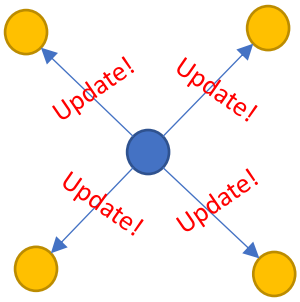


5、如果收到邀請的 Node 已經有 Parent 則會拒絕邀請。

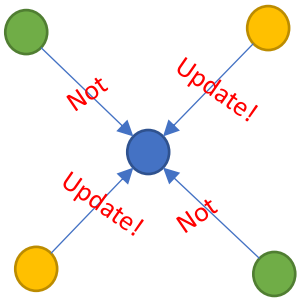


6、最後得到 Spanning Tree

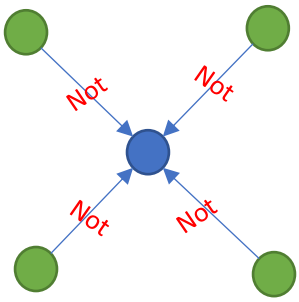
更新過程如下：



1、若中心 Node 有更新的話，會發送一份更新給所有 Neighbor

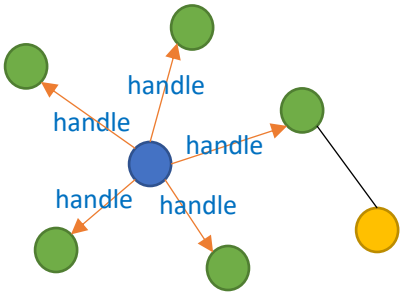


2、若沒有更新則會回送無更新訊息，同時中心 Node 會將該 Neighbor 標示為沒有更新

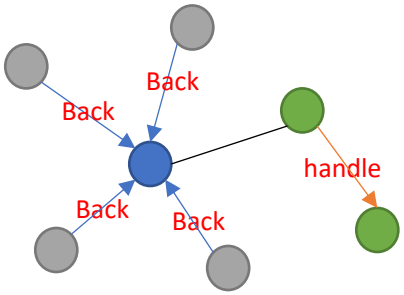


3、若所有 Neighbor 都不再更新則會觸發 Termination 事件

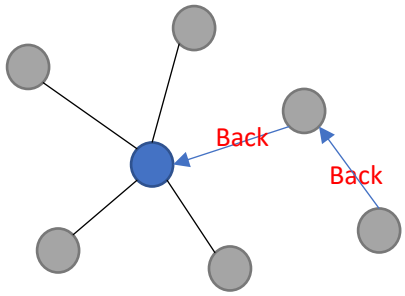
終止過程如下：



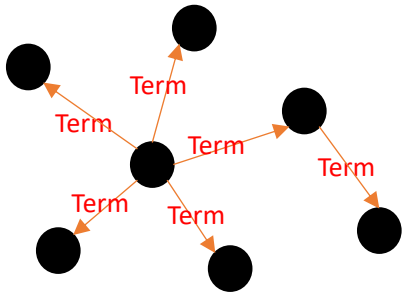
1、若所有 Neighbor 都不再更新則由 Root Node 開始向所有 Child 發送 Termination handle



2、若 Node 發現自己是 Leaf 的話就會直接回傳 Back，否則同樣等待到所有 Neighbor 都終止再發送 Termination handle 給 Child



2、當 Node 收齊了所有 Child 所發送的 Back 之後，Node 會再往自己的 Parent 回送 Back



2、當 Root 收齊所有 Child 發送的 Back 之後就會發送 Terminate 訊號，拿到的 Node 就會往所有 Child 發送後自動結束工作

(b) X