Parallel Programming Homework3

106065509　林庭宇

1. Implementation

1.1. APSP_Pthread

- Algorithm: 這邊我實作了兩個版本的Dijkstra，分別是Binary heap和Adjacency matrix。Dijkstra是做single source shortest path，而要做all pair shortest path的方式就是對每一個點做一次的Dijkstra。

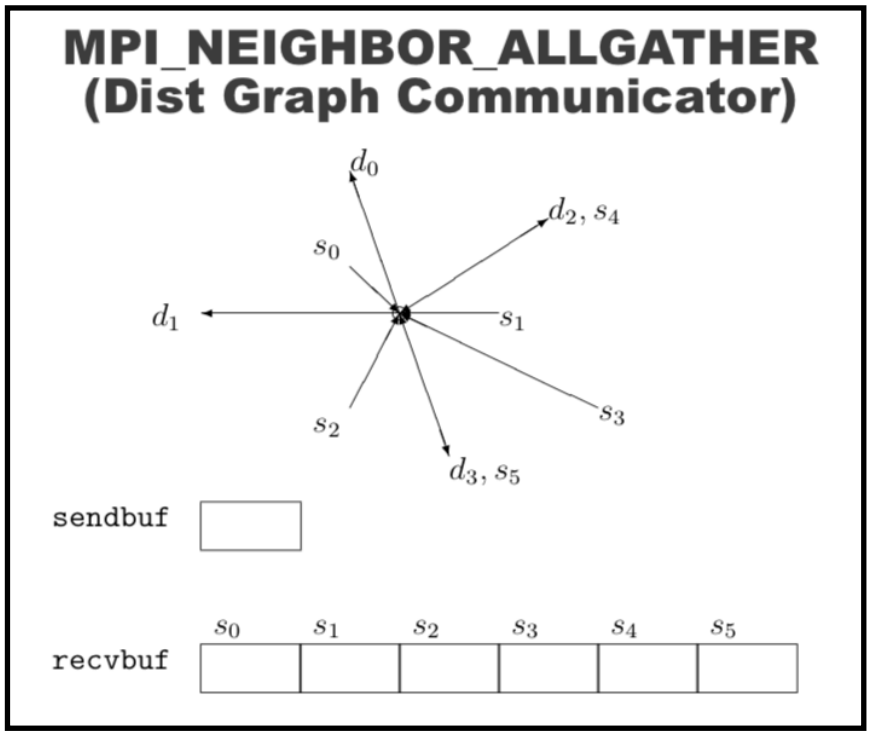
  實作兩個演算法是因為兩者的time complexity是O(ElgV)和O(V^3)，所以當graph為sparse/dense時，便有各自的優勢，且效能至少會跟Floyd Warshall依樣快。而為了整合兩者的優點，我有去實測兩個演算法的performance，並以performance相交的點作為切換的分界，後面的實驗有數據和圖。最後的切換點為$ElgV = V^2 * 0.6$。

1.2. APSP_MPI_sync

- Design/Algorithm: 每一輪vertex會和自己的neighbor做一次溝通，把自己的shortest path(sp) set傳出去，並得到每一個neighbor對於其他vertex的sp set，接著做Floyd Warshall的iteration來更新自己的sp。

  接著會做一次MPI_Allreduce，等所有vertices都溝通完畢後，檢查有無更新，有的話就進行下一輪更新，沒有的話代表Floyd Warshall完成。之後用一個MPI_Gather把所有vertex的sp set合併到rank = 0的process，由他負責最後的write out。

- Methodology: 使用MPI_Graph_create來建立一個MPI的Graph，這樣的好處是可以設定每個vertex的neighbor，之後在溝通的時候就可以直接call



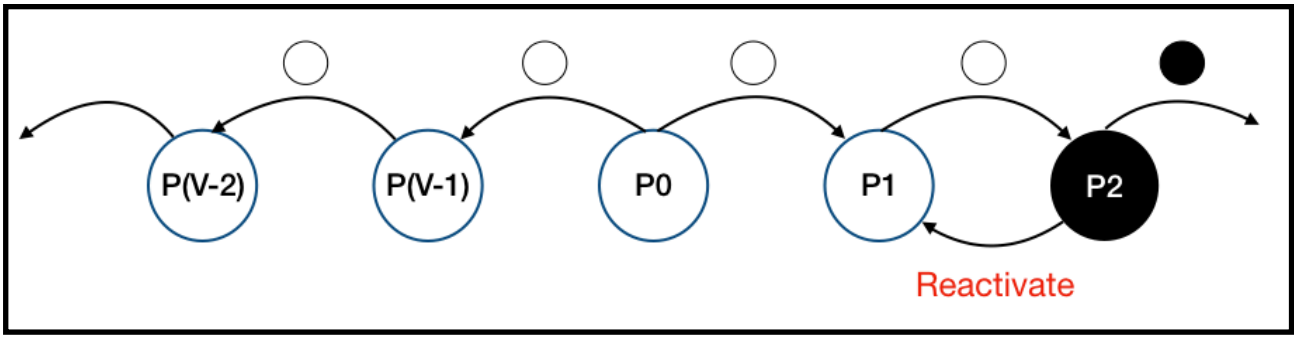
MPI_Neighbor_allgather和自己的neighbor溝通，非常簡潔。

(MPI_Neighbor_allgather: https://www.open-mpi.org/doc/v2.0/man3/MPI_Neighbor_allgather.3.php)

1.3. APSP_MPI_async

- Design: 這部分的algorithm和sync的版本一樣便不多贅述，然而因為是async的關係，必須implement好的termination detection mechanism，我使用的是講義上的dual-pass ring架構。

  不過因為一開始我有些誤會dual ring的架構，以為傳完token後還有可能被reactivate導致token來不及更新，所以我的dual ring其實一次傳了兩個token，分別從rank=0的process向左、向右傳，當兩個都繞了一圈回來皆為白色，且rank=0本身也是白色才結束。



  後來因為準確率一直沒辦法提高，我又換回了原來的Dual-pass ring，並做了一些調整來提升準確率，詳細的內容在2.7。

- Methodology: 因爲實作dual-pass ring的架構，每一個process可能會收到token和sp set兩種訊息，因此我使用的是MPI_Iprobe來確認每個incoming message的tag，0代表是sp set而1代表token，針對不同的tag做相對應的動作。

1.4. APSP_Hybrid

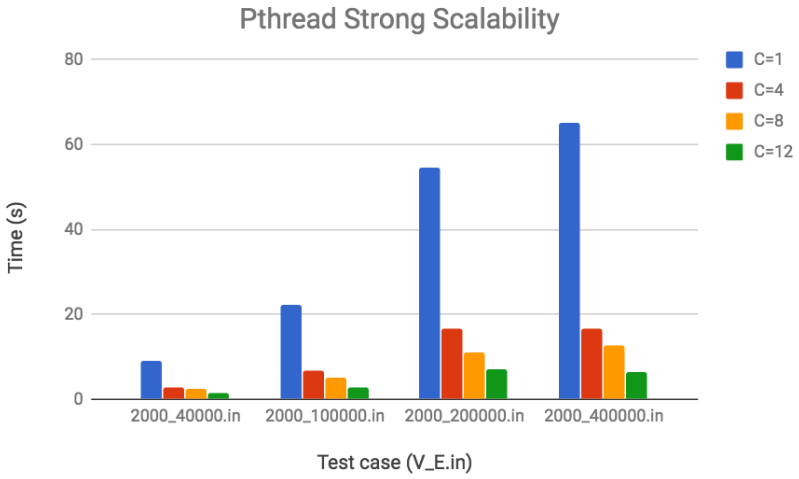
- Design: 我是採用shared-memory的寫法，說白了就是APSP_Pthread的擴充版。先把原本的graph切成許多subgraph分給每個process，再讓process去開threads平行做各個vertex的Dijkstra。

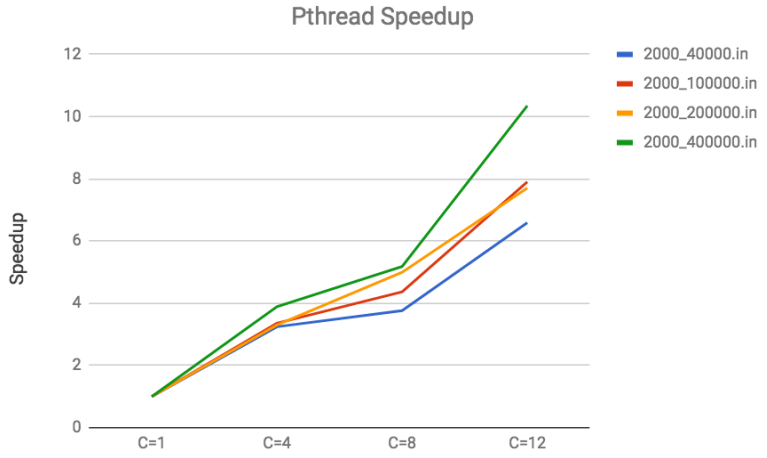
  這樣的好處是可以比APSP_Pthread多開更多的thread以提升平行度。然而按照最直接的寫法就會開V個threads，但是開太多thread反而會拉低performance，我推測這是因為context switch的關係。因此我最後是限制一個process最多只能開到12個thread，符合apollo的架構。

2. Performance analysis

2.1. APSP_Pthread

- Strong Scalability: 測試的時候我都是指定-n1 -cX，以免涉及不同process間的同步問題，在這樣的情況下Pthread的版本相對單純許多，不需要考慮跨Node和process的問題，因此有很好的Scalability。
- Speedup: 並非完全線性，但還是可以明顯看出上漲的趨勢，估計是因為CPU bound加上test case不夠大，所以4個core到8個core的成長幅度有限。



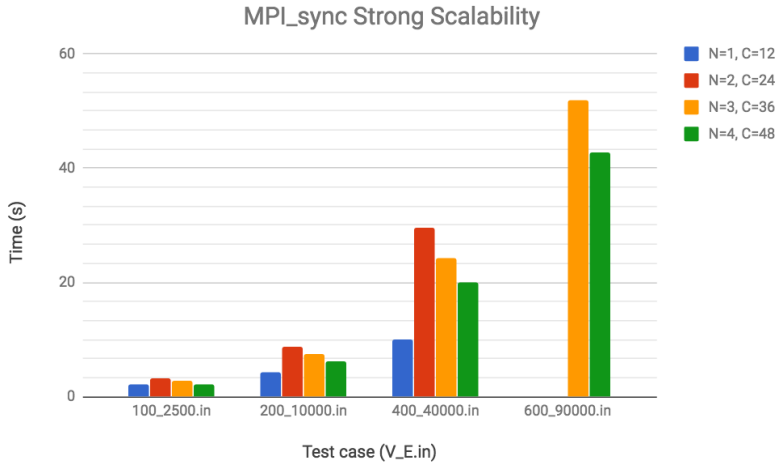


2.2. APSP_MPI_sync

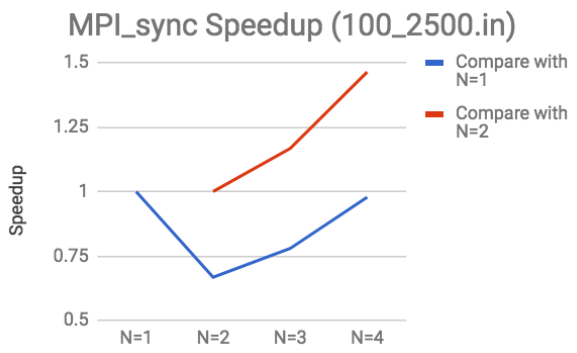
- Strong Scalability: 測試的指令是-N{1~4} -nV，因為V >> 12，所以每一個Node的core都會開好開滿，再藉由增加Node來提升core數。而測試出來的結果實在讓人冷汗直流，N=2(C=24)的執行時間居然是N=1(C=12)的兩倍，沒有變快反而還變慢，一來一往是4倍的差距啊！

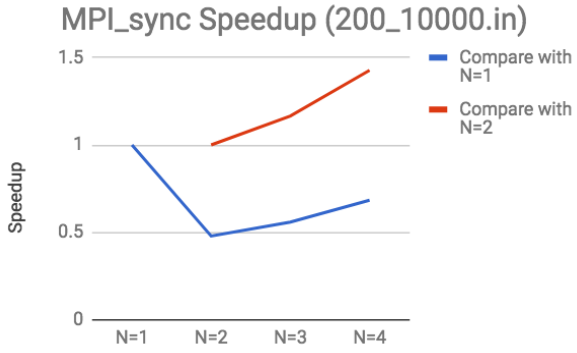
後來仔細想想其實也不是沒有道理，跨Node的溝通會很慢，加上我們都是用

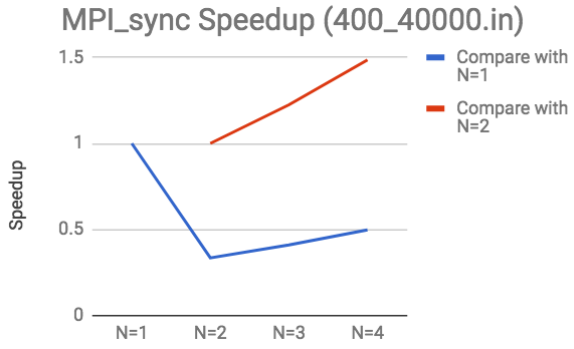
overcommit的方式一次開超過core數的process，溝通慢又要context switch的結果就是超級慢...



- Speedup: 因為N=1和N=2~4不太能比，所以我另外又針對N=2~4做了一個speedup的曲線，就可以看出其實是有在變快的只是斜率頗低，原因是Communication bound，之後的profiling就可以明顯看出來。



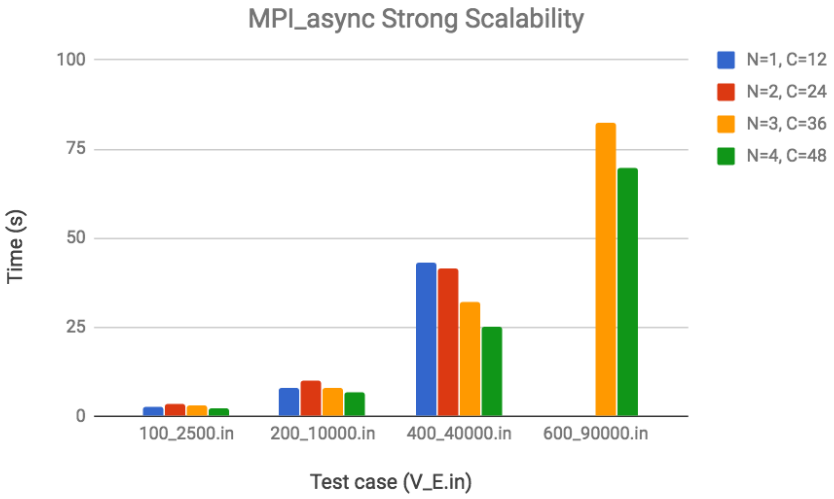




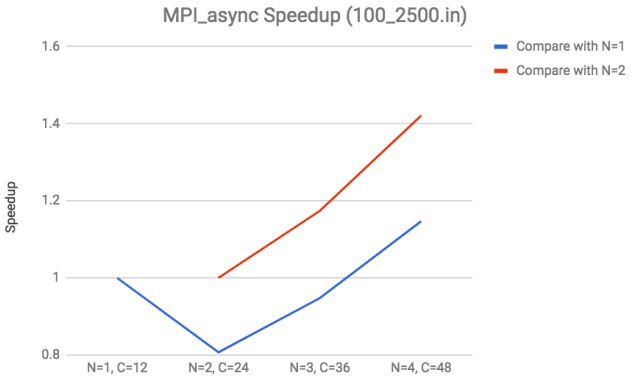
2.3. APSP_MPI_async

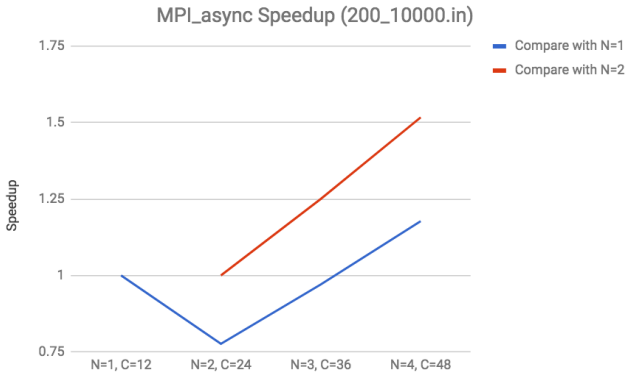
- Strong Scalability: 測試的指令同MPI_sync，大致的狀況也相同，但是因為async的關係，N=1已經沒有像MPI_sync那樣比N=2~4快那麼多了，甚至在跑

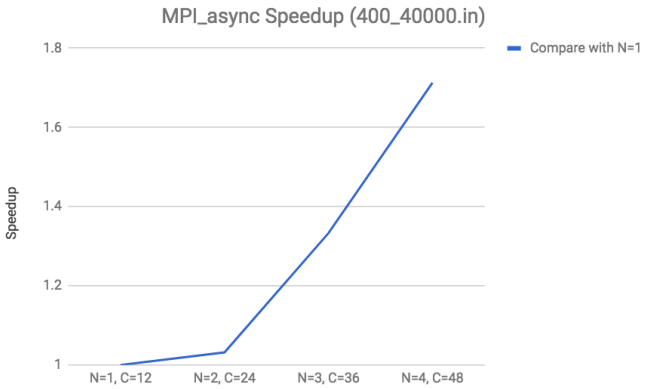
400_40000.in這筆測資時，N=2~4都成功超越N=1的速度！間接證明了async的寫法，可以讓每個process各自獨立和neighbors溝通，所以較能受益於core數的增加！不過由於是利用dual ring來判斷結束條件，傳token的時間仍拖慢了整體的執行時間。



- Speedup: 前兩張圖跟MPI_sync差不多，但是在400_40000.in就可以看到一條完整的上升曲線，雖然斜率仍是不太理想，但在Communication bound的情況下已經是很不錯的了！





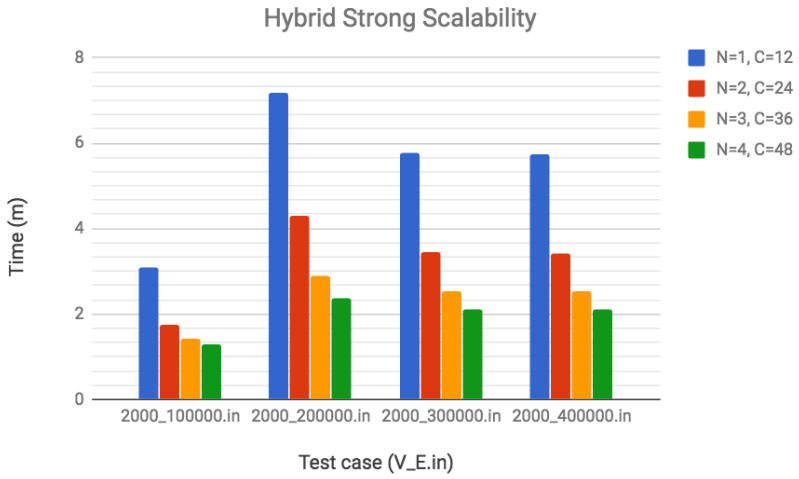


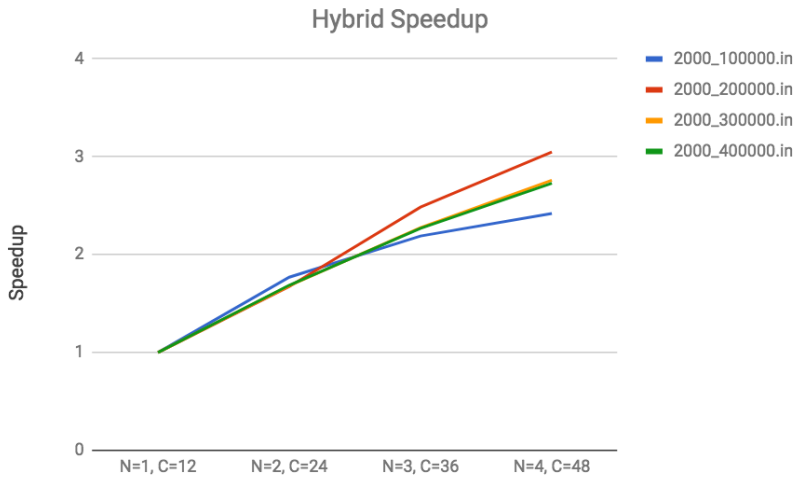
2.4. APSP_Hybrid

- Strong Scalability: 因為有限定一個process最多只能開12個thread，所以能夠避免不必要的thread scheduling和store/restore state等等，因此和APSP_Pthread有一樣好的Scalability。

  事實上因為跟APSP_Pthread是在做一樣的事情，所以當core數一樣的時候，執行時間幾乎是一樣的，甚至會因為要做Process間的溝通而慢了一些。

- Speedup: 明顯的線性上升且斜率趨近理論值。





2.5. MPI Profiling Distribution Comparison

首先說明量測的項目包含：Init, I/O, CPU, Comm, Sync五類時間:

- Init: 初始化APSP edge wight matrix，呼叫MPI_Init()、MPI_Graph_create()等等初始化變數和環境的動作。

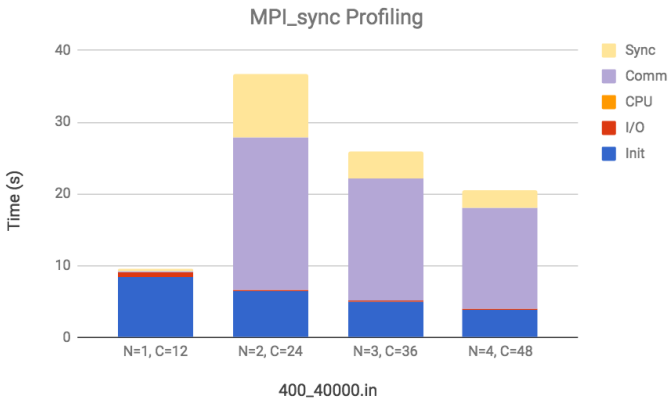
- I/O: 讀input file以及最後把APSP edge wight matrix寫進output file。

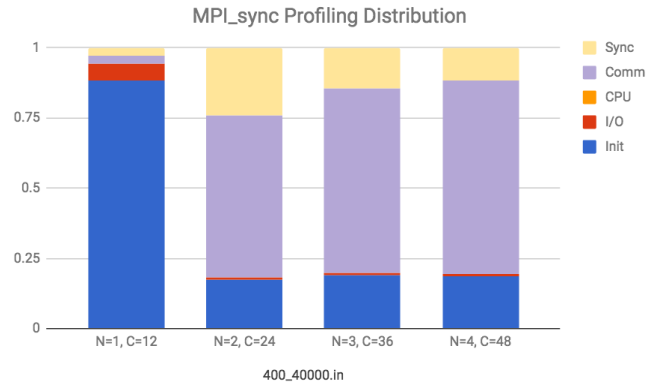
- CPU: Process收到來自neighbor的sp set後，進行比較判斷是否更新自己的sp set。

- Comm: 與Neighbor之間send/recv message的時間，在MPI_sync中MPI_Neighbor_allgather屬於此類，而在MPI_async中則包括了MPI_Iprobe/Isend/Irecv。

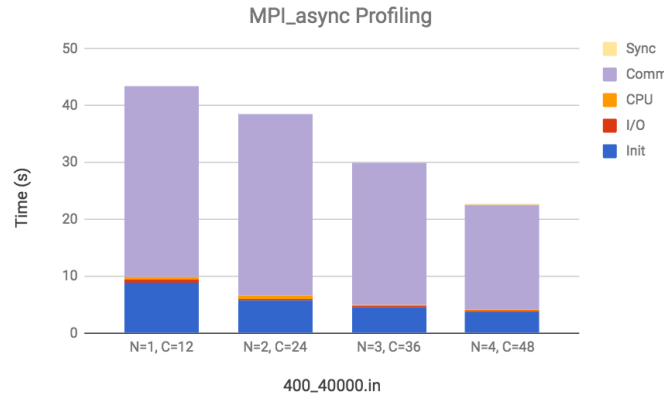
- Sync: 沒有溝通的意圖，純粹把資訊整合。在MPI_sync中包含每一個iteration判斷有沒有process更新時的MPI_Allreduce()和最後把所有process的sp set結合的MPI_Gather()，而MPI_async中只有最後call的MPI_Gather()屬於此類。

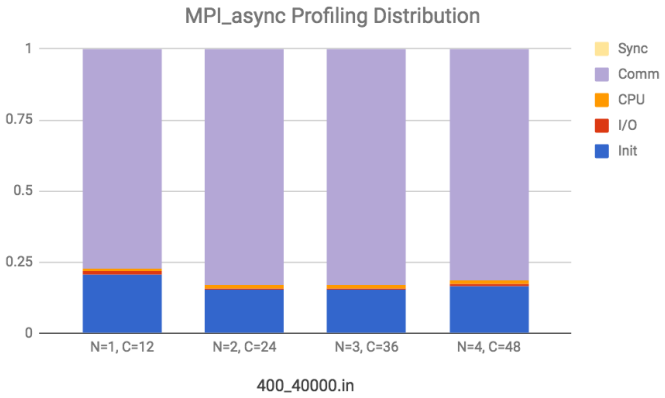
- MPI_sync:





- MPI_async:





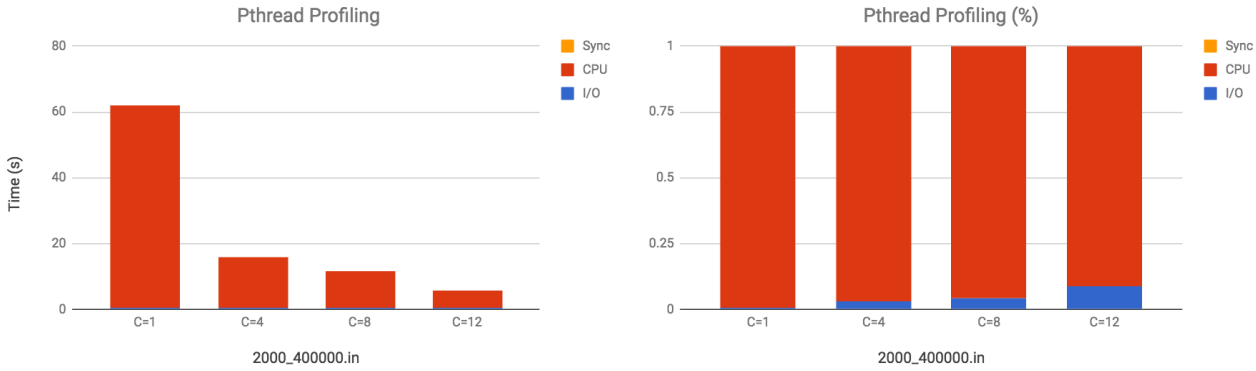
- Comparison: 四張圖都非常的紫，代表著MPI_sync和MPI_async都是Communication bound，CPU和I/O的佔比實在是微乎其微。

  值得注意的是Init time佔了幾乎1/5的時間甚至更多，我推測這是因為overcommit所致。另外MPI_async幾乎不見Sync time，因此不會需要花時間等所有的thread一起做某件事，也間接比較能受益於core數的增加。

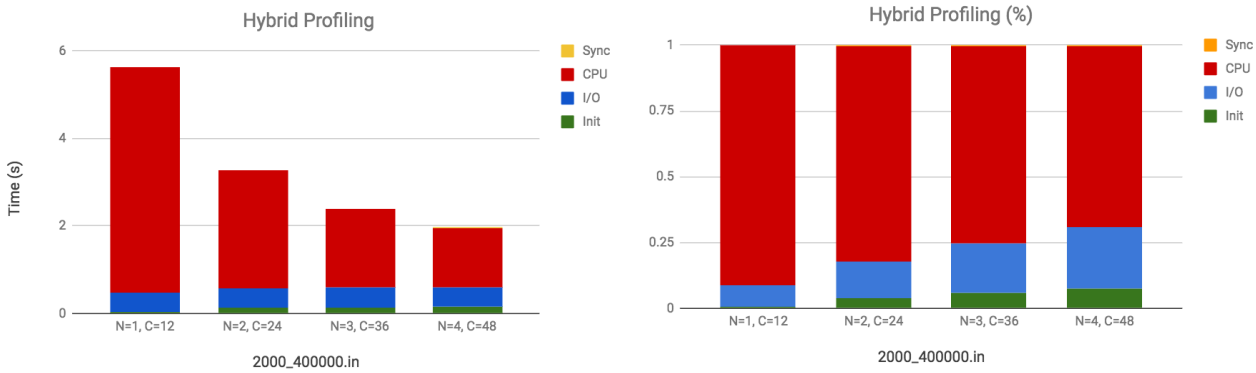
2.6. Pthread Profiling Distribution Comparison

因為實作上Hybird和Pthread很相近，因此在這邊一起比較。首先仍先說明四個量測的項目：Init, I/O, CPU, Sync。

- Init: MPI_Init()的時間，只有在Hybrid的版本有測。
- I/O: 讀input file以及最後把APSP edge wight matrix寫進output file。
- CPU: 包括pthread_create以及每個thread計算Dijkstra的時間。
- Sync: pthread_join的時間減去每個thread計算Dijkstra的時間。
- Pthread:



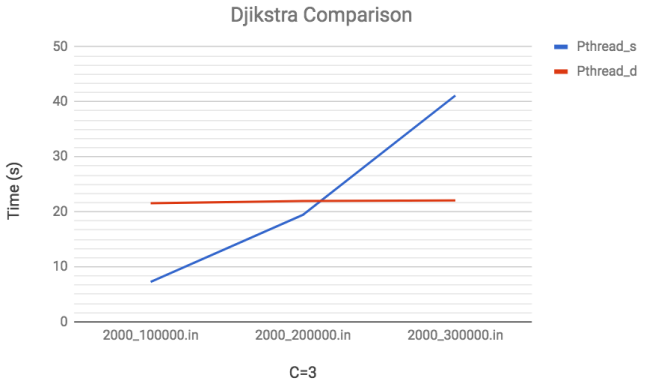
- Hybrid:

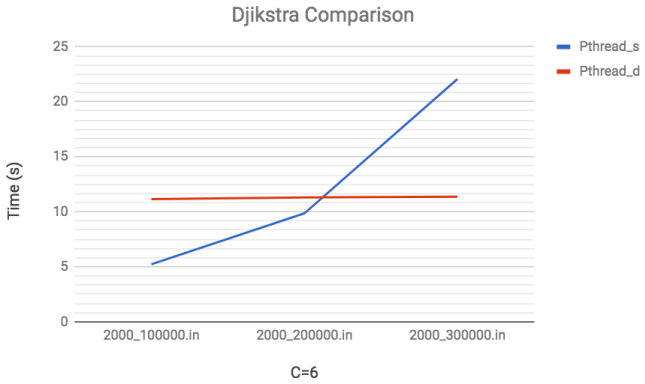


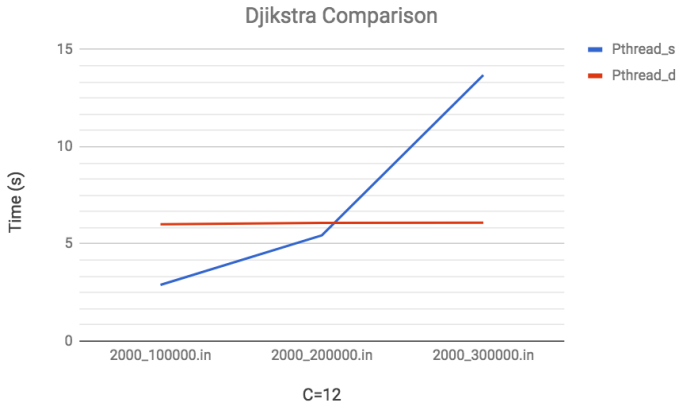
- Comparison: 四張圖表紅一片代表著Pthread和Hybrid皆為CPU bound，較為理想也是效能可以提升的主因。在Hybrid的版本中，隨著N越大，MPI_Init()的時間佔比也越大，可以視為MPI的overhead。

2.7. Other Experiments

- Dijkstra comparison: 這部分主要是關於Pthread版本的實驗，我針對C=3, C=6, C=12這三個參數，分別試驗3個不同的測資，試著找到Sparse/Dense的交界點。



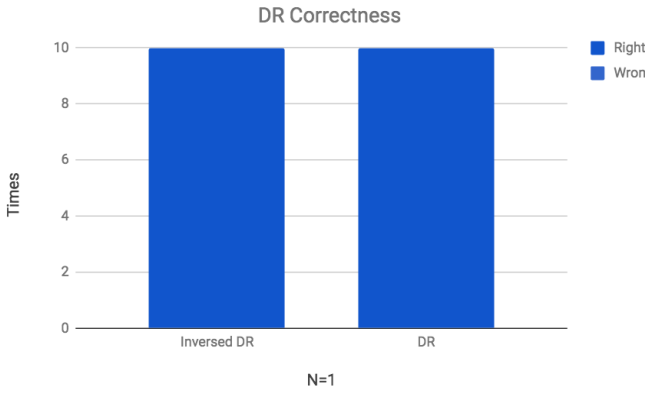


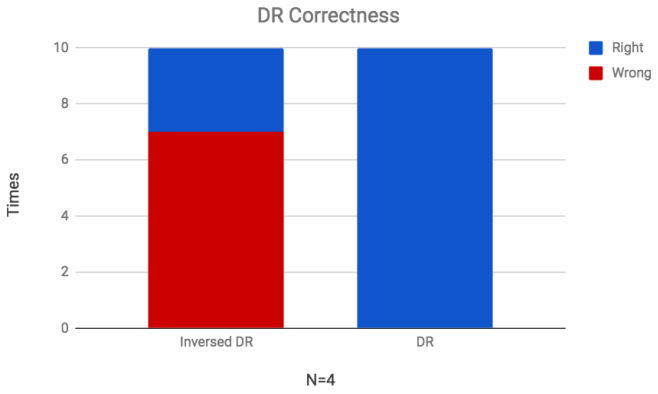


- 可以看到Pthread_d的執行時間因為只和V有關，所以是持平的狀態。兩者的時間曲線關係和C幾乎無關，相聚點則大約都是在E = 220000左右，因此我後來取 $ElgV = V^2 * 0.6$。

- Dual-pass Ring (DR) Correctness: 如同前面所述，原本因為誤解了DR的意思，而自己實作了同時往左往右傳token的Inverse DR。但根據我的理解，多傳一個token應該會讓正確率更高才對，然而在跑了hw3-judge後，準確率竟然慘不忍睹。

  在尋找可能的原因時我發現了一個有趣的線索，在N=1的時候Inverse DR的準確率幾乎是百分之百，但是在N=4的時候卻掉到只有3成的準確率。





起初我認為是token傳太快的緣故，畢竟排除自己把code寫壞之外，DR會錯的唯一可能就是token傳的比data快，導致太快結束。因此我把起始傳token的時間改成：Rank 0要收到Neighbor次的data且都沒有更新後才開始傳。

但並沒有如預期的提升準確率，令我更摸不著頭緒了，於是乎我只好捨棄Inverse DR，換回原本的DR寫法，結果準確率還是起不來！！！這實在令人匪夷所思。後來和許多同學以及助教討論的結果發現，也許我們該把矛頭指向Iprobe。

助教表示Iprobe頗雷，有時候會收不到東西，或者不按順序收東西等等，因此我判斷在這樣的風險下我又同時傳兩個token，似乎無形之中讓process就更收不到data了，於是提早結束的情況才會一直發生。

且我發現在hw3-judge的時候我只有4.in和5.in這兩個會有Wrong Answer的情況，我推測是因為這兩個testcase的vertex較少，加上又是sparse graph，因此data的量本身就較少，導致token可能傳太快的緣故。

最後我把結束的條件設為：Rank 0要收到10次的白token且自己也都是白色，才開始傳結束的token。這樣一來雖然有些犧牲Performance但是準確率就幾乎是100%了！

3. Experience and conclusion

這次的實作最令我挫折的就是Async版本的termination mechanism了，明明就implement對的寫法，卻因為MPI本身的瑕疵而改了好久，希望老師可以花點時間講解一下這些Function的底層，不然老實說MPI的document真的很少啊ＱＱ

不過因為花了很多時間所以也學到了很多，且這次在實作前有記取上次的教訓，先從演算法的角度分析，因此不是單純使用Floyd Warshall，而是加入了Dijkstra的優勢。