Parallel Programming HW3

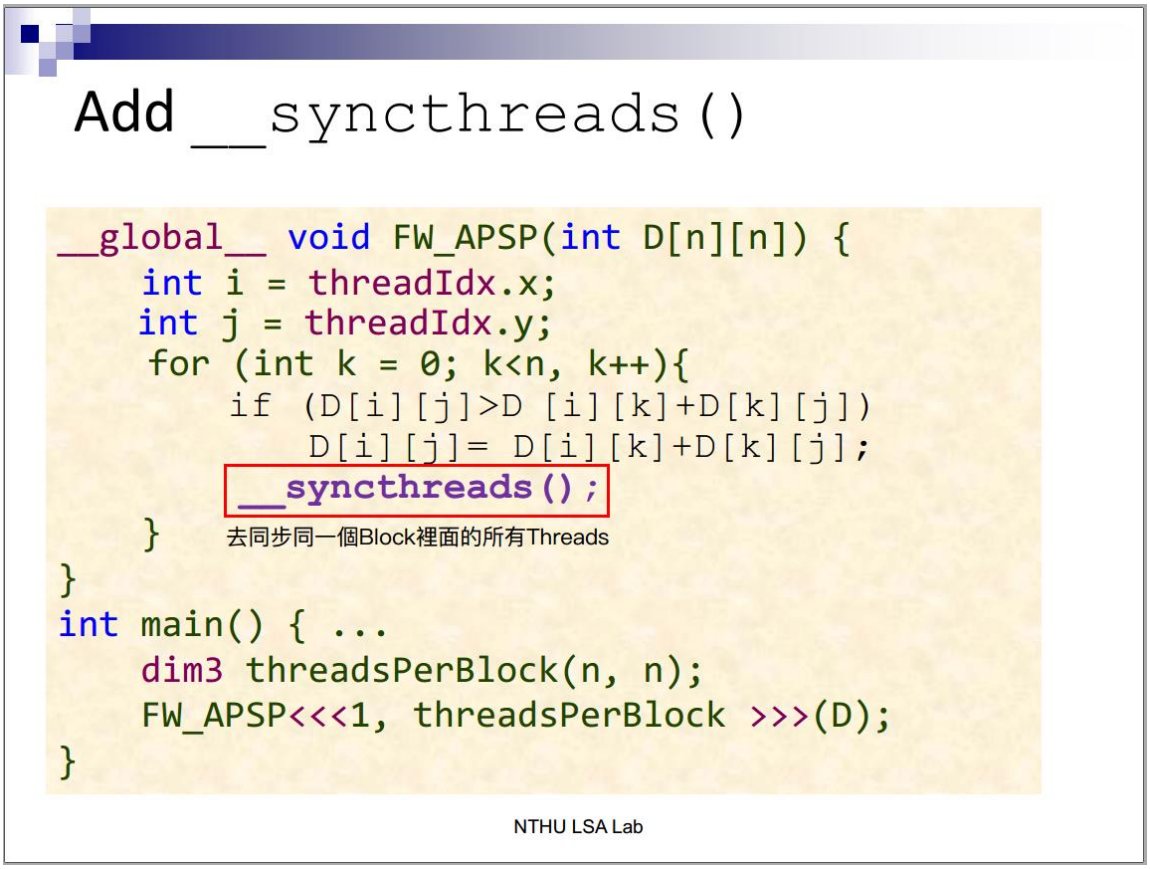
All Pair Shortest Path

ssh: pa128427359、程祥恩

A. Design

(a) Pthread

使用 Floyd-Warshall 演算法來實作，因為他方便做平行化，也有不錯的效能(O(n^3))。主要的實作方式是參考投影片上的虛擬碼，如圖。每一條 thread 都會去執行一次 FW_APSP 函式，系統必須確保 k 是按照由 0 到 n 的順序去執行的，因此在 syncthread 方面，我呼叫了 pthread_barrier_wait(&barr)來等待 k 的所有 thread 都執行完畢後，才執行下一輪 k+1。



(b) Fully distributed Synchronous vertex-centric MPI

使用了 MPI 預設的 Graph 方法，包含：

MPI_Graph_create: 輸入一張圖的 Vertex 和 Edge，在 MPI 初始化這張圖

MPI_Comm_rank: 回傳這台機器在 Graph Communicator 的 rank

MPI_Graph_neighbors_count: 回傳每個 rank(分別代表每個節點)的鄰居數

MPI_Graph_neighbors: 回傳每個 rank(分別代表每個節點)的鄰居們，放到 array

MPI_Neighbor_allgather: 每個 rank 對自己的鄰居們進行資料的收發

每一個 rank 的 send_buf 儲存該節點到所有其他節點的最短距離

每一個 rank 的 recv_buf 儲存該節點所有鄰居到其他節點的最短距離

每一個 rank 接受到 recv_buf 時，就會執行 Floyd-Warshall 的距離判斷，如果有更短的距離就取代掉原本在 send_buf 裡面的值，一旦所有節點的距離跑過一輪卻都沒被更新時，就把所有節點的資訊以 MPI_Gather 存到 rank0 後結束程式。

(c) Fully distributed Asynchronous vertex-centric MPI

參考<<Distributed Computing: Principles, Algorithms, and Systems>>P.153 的虛擬碼進行實作。

```
(local variables)
int LEN[1..n]          // LEN[j] is the length of the shortest known
                       // path from i to node j.
                       // LEN[j] = weight_ij for neighbor j, 0 for
                       // j = i, ∞ otherwise
int PARENT[1..n]       // PARENT[j] is the parent of node i (myself)
                       // on the sink tree rooted at j.
                       // PARENT[j] = j for neighbor j, ⊥ otherwise
set of int Neighbors ⟵ set of neighbors
int pivot, nbh ⟵ 0

(message types)
IN_TREE(pivot), NOT_IN_TREE(pivot),
PIV_LEN(pivot, PIVOT_ROW[1..n])
    // PIVOT_ROW[k] is LEN[k] of node pivot, which is LEN[pivot, k] in
                                                  // the central algorithm.
                // the PIV_LEN message is used to convey PIVOT_ROW.
(1)   for pivot = 1 to n do
(2)       for each neighbor nbh ∈ Neighbors do
(3)           if PARENT[pivot] = nbh then
(4)               send IN_TREE(pivot) to nbh;
(5)           else send NOT_IN_TREE(pivot) to nbh;
(6)       await IN_TREE or NOT_IN_TREE message from each neighbor;
(7)       if LEN[pivot] ≠ ∞ then
(8)           if pivot ≠ i then
(9)               receive PIV_LEN(pivot, PIVOT_ROW[1..n]) from
                    PARENT[pivot];
(10)          for each neighbor nbh ∈ Neighbors do
(11)              if IN_TREE message was received from nbh then
(12)                  if pivot = i then
(13)                      send PIV_LEN(pivot, LEN[1..n]) to nbh;
(14)                  else send PIV_LEN(pivot, PIVOT_ROW[1..n])
                              to nbh;
(15)          for t = 1 to n do
(16)              if LEN[pivot] + PIVOT_ROW[t] < LEN[t] then
(17)                  LEN[t] ⟵ LEN[pivot] + PIVOT_ROW[t];
(18)                  PARENT[t] ⟵ PARENT[pivot].
```

Algorithm 5.8 Toueg's asynchronous distributed Floyd–Warshall all-pairs shortest paths routing algorithm. The code shown is for processor P_i, $1 \le i \le n$.

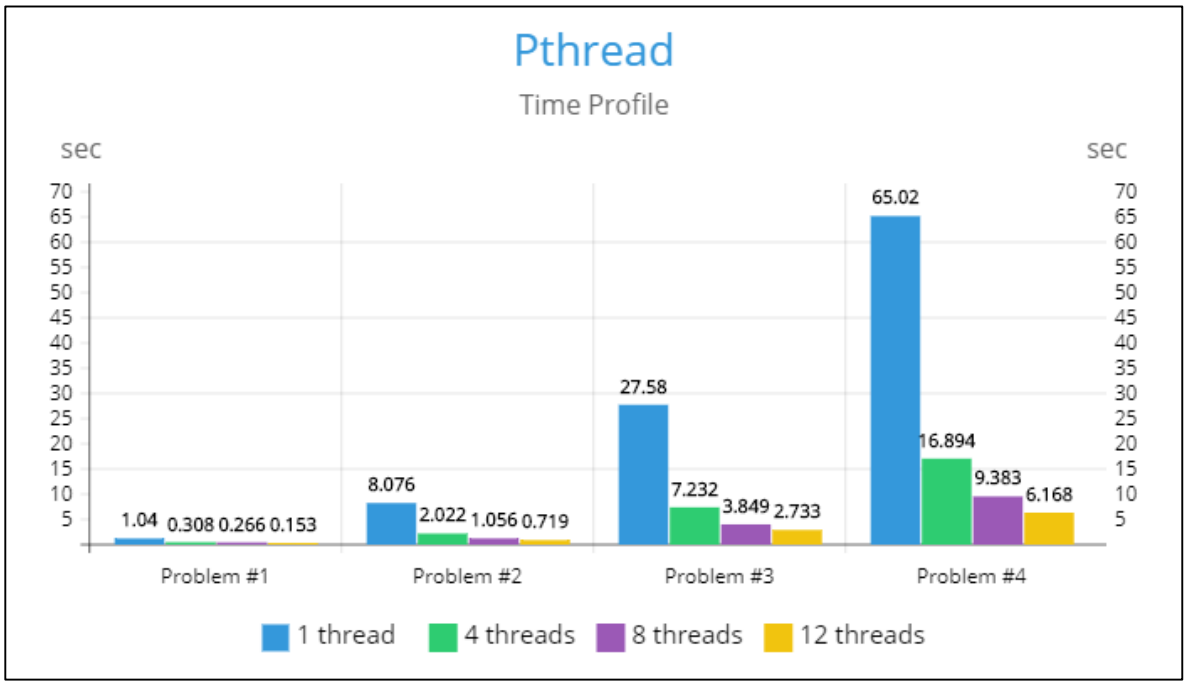
B. Performance analysis

	VERTICES	EDGES
PROBLEM #1	500	50,000
PROBLEM #2	1,000	200,000
PROBLEM #3	1,500	450,000
PROBLEM #4	2,000	800,000
PROBLEM #5	50	500
PROBLEM #6	100	2000
PROBLEM #7	200	8000
PROBLEM #8	400	32000

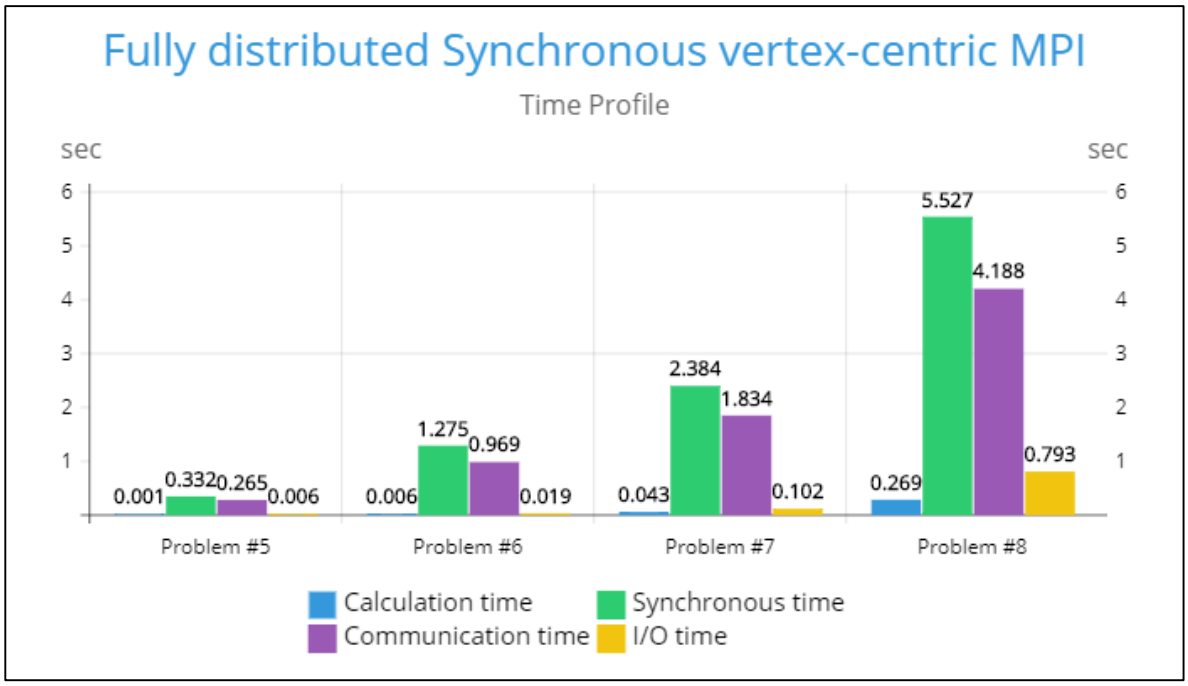
(a) Strong scalability chart

i. Pthread

從圖中可以發現，節點數在 500 到 2000 之間時擁有良好的 Scalability，在 speedup factor 上也都有接近線性平行化的表現。



ii. Fully distributed Synchronous vertex-centric MPI



iii. Fully distributed Asynchronous vertex-centric MPI

(b) Performance profiling

i. Pthread

(unit: sec)

Problem #1

Num of threads	Calculation time	Synchronous time	I/O time	Total time
1	0.99	0	0.05	1.04
4	0.253	0.005	0.05	0.308
8	0.16	0.056	0.05	0.266
12	0.098	0.005	0.05	0.153

Problem #2

Num of threads	Calculation time	Synchronous time	I/O time	Total time
1	8.055	0	0.021	8.076
4	1.993	0.008	0.021	2.022
8	1.016	0.019	0.021	1.056
12	0.684	0.014	0.021	0.719

Problem #3

Num of threads	Calculation time	Synchronous time	I/O time	Total time
1	27.15	0	0.43	27.58
4	6.762	0.04	0.43	7.232
8	3.372	0.047	0.43	3.849
12	2.27	0.033	0.43	2.733

Problem #4

Num of threads	Calculation time	Synchronous time	I/O time	Total time
1	64.24	0	0.78	65.02
4	16.082	0.032	0.78	16.894
8	7.993	0.61	0.78	9.383
12	5.337	0.051	0.78	6.168

從以上四張表格可以發現，Thread 數量為 8 的時候會花費最多時間在等待，但等待時間其實都很短，並不足以影響效能。I/O 時間則會根據問題的大小而有所改變。最後在計算時間方面，都有達到接近線性平行的效率。

ii. Fully distributed Synchronous vertex-centric MPI

(unit: sec)

Problem #	Calculation time	Synchronous time	Communication time	I/O time	Total time
5	0.001	0.332	0.265	0.006	0.604
6	0.006	1.275	0.969	0.019	2.269
7	0.043	2.384	1.834	0.102	4.363
8	0.269	5.527	4.188	0.793	10.777

從以上四張表格可以發現，比起 Pthread 版本，MPI 版本所花費的時間會提升許多，主要時間會花在同步和溝通上，推測是因為不同機器之間需要進行多次資料傳輸導致，隨著問題大小的提升，此版本距離線性平行越來越遙遠，因為花在溝通和同步的時間又更多了。

iii. Fully distributed Asynchronous vertex-centric MPI

C. Experience and Conclusion

從 Pthread 版本可以發現，程式的平行化效率很高，花在同步和等待的時間也比預期還要少許多，因此有不錯的效能。不過不知道是什麼原因，Thread 數為 8 的時候會有較高的同步時間。而 MPI 的 sync 版本，原本一直想不到該怎麼實作，正當要放棄的時候，無意間看到了 MPI 竟然已經有寫好的 Graph 方法，因此就直接拿這些方法來實作了。MPI 版本在執行上，會花費非常大量的時間，也會花費許多資源（機器），是很沒有效率的實作方法。