HW1 – Report

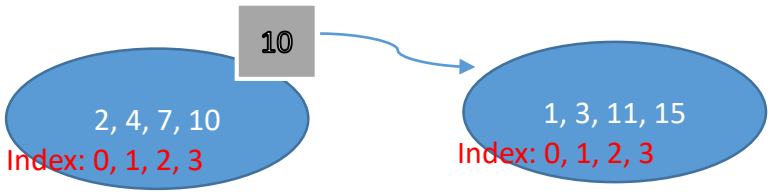
1. **Name:** 徐嘉駿, **Institude:** 資應所, **Student ID:** 107065528

2. Implement

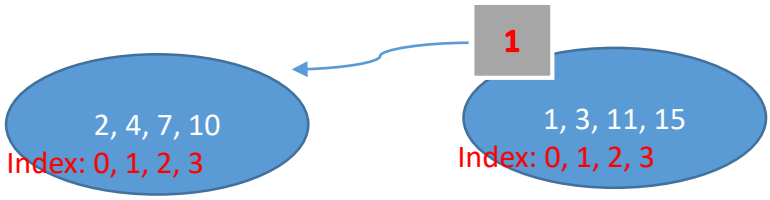
- 將 input 平均分給每個 process (最後一個 process 可能分得比較少)，令分到的 number 數量為 n_sub。
- 每個 process 開始自己 sort 被分到的部分的 numbers。(用 c library “algorithm” 做 sort)
- 與隔壁 ID 的 process 溝通，process 會在每輪會被分為 sending group 與 receiving group (利用(rank + i)%2==0, i ϵ [0, the size of processes])
  (1) Sending group 裡的 process 會先將自己擁有的最大值傳給 rank+1 的 process
  (2) Receiving group (rank+1)裡的 process 收到最大值後，找出此值應該放在自己 sorted numbers 的位置 index，並將此 index 傳回去。
  (3) Sending group 裡的 process 收到 index 後，將自己(n_sub – index)大的 number 傳給 rank+1 的 process。
  (4) Receiving group (rank+1)裡的 process 收到 numbers 後，也將自己(n_sub – index)小的 number 傳回去。
  (5) Sending group 裡的 process 從最大值開始將與收到的 numbers 比較，留下較小的值。Receiving group 裡的 process 從最小值開始將與收到的 numbers 比較，留下較大的值。

以下為兩個 process 交換的圖例:

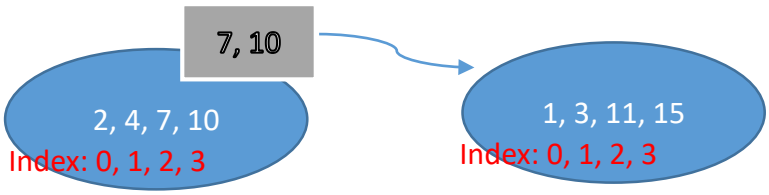
(1)



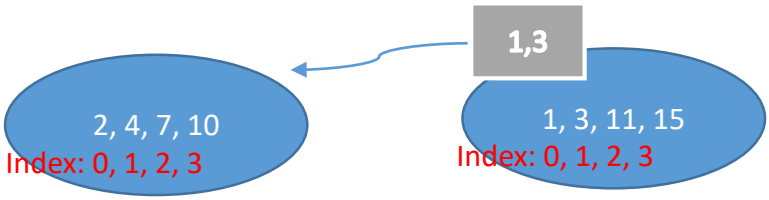
(2)



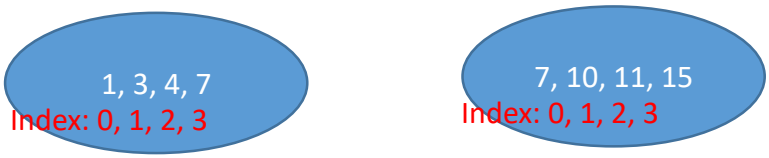
(3)



(4)



(5)



3. Experiment & Analysis

- **Methodology:**

Performance Metric:

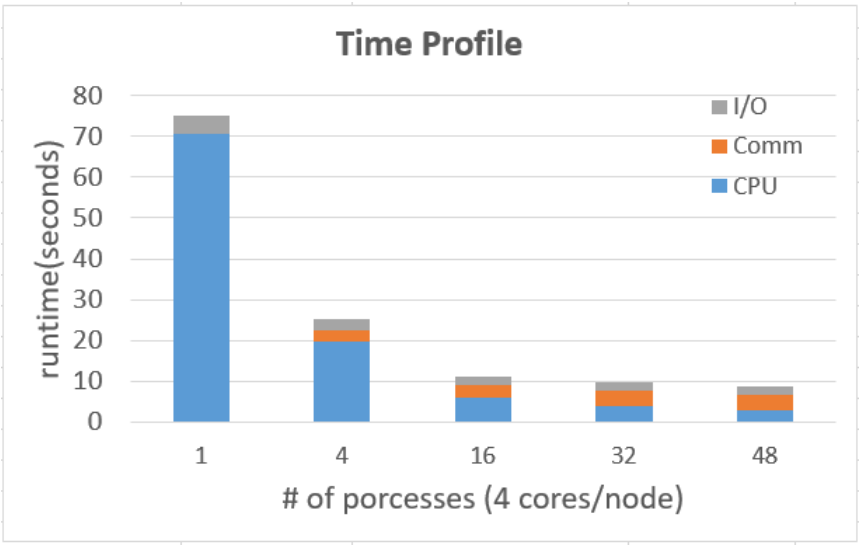
(1) **Computing time:** 有利用到 cpu 動作的時間都算在內，例: sorting，array 給值等。

(2) **Communication time:** 有用到 MPI message function 的時間都算在內。例: MPI_Send, MPI_Recv 等。

(3) **I/O time:** 有用到 MPI I/O function 的時間都算在內。例: MPI_File_open, MPI_File_write_at, MPI_File_close 等。

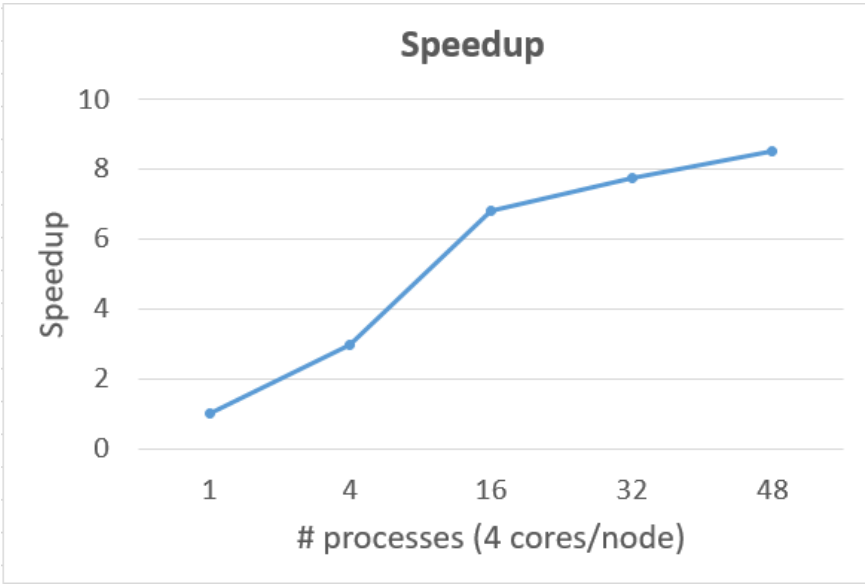
- **Time Profile & Speedup Factor:**

(1) Single Node

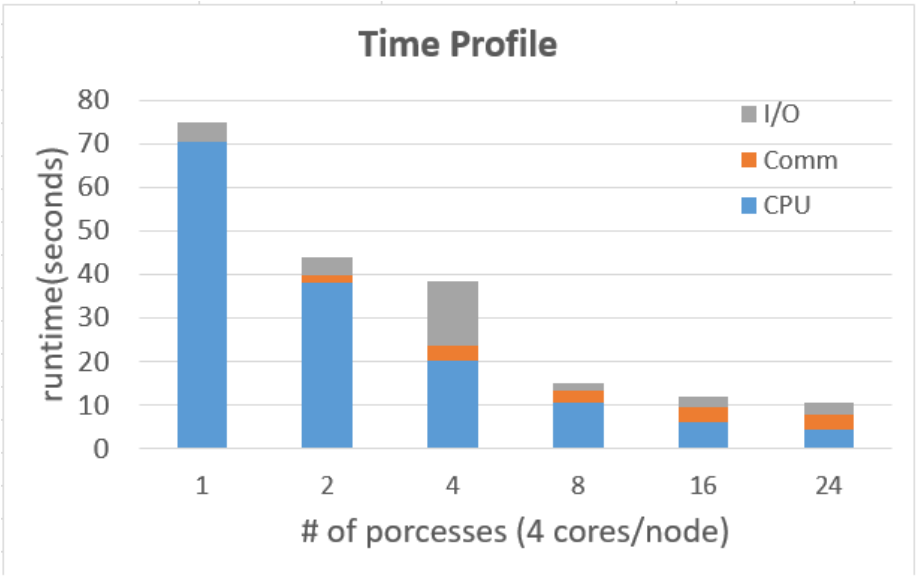


數值:

process	CPU	Comm	I/O	total time
1	70.478291	0	4.588245	**75.066536**
4	19.659207	2.802807	2.638848	**25.100862**
8	11.021839	2.955086	2.427384	**16.404309**
16	6.052372	3.126269	1.808715	**10.987356**
24	4.538694	3.160431	2.810888	**10.510013**
32	3.785957	3.953271	1.957766	**9.696994**
48	2.899924	3.84846	2.068501	**8.816885**

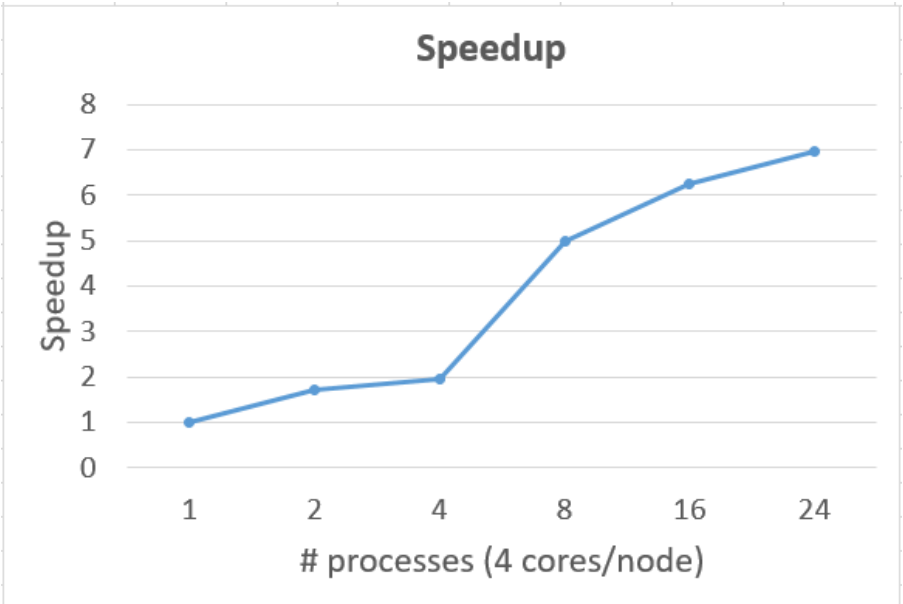


(2) 2 – Node

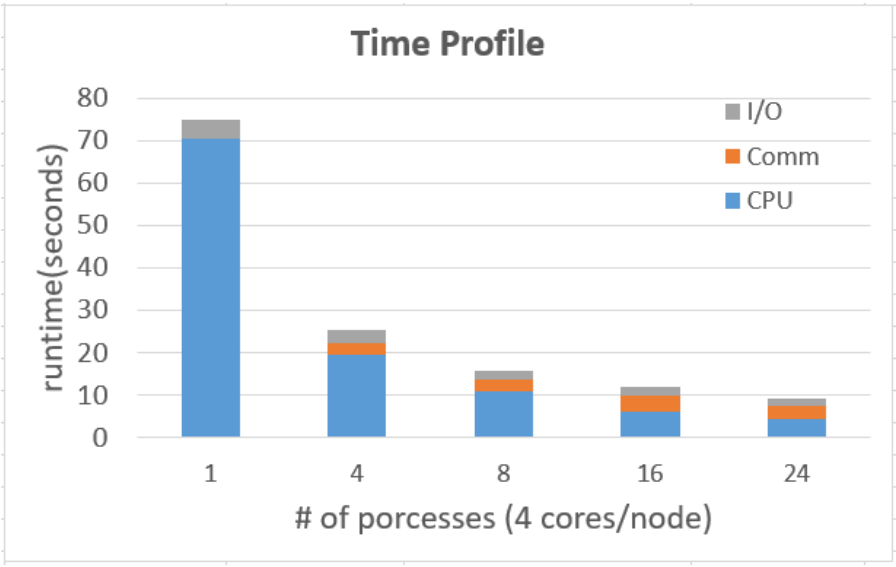


數值:

process	CPU	Comm	I/O	total time
1	70.478291	0	4.588245	**75.066536**
2	38.071786	1.84582	4.015034	**43.93264**
4	20.160243	3.554008	14.924856	**38.639107**
8	10.770981	2.752589	1.527352	**15.050922**
16	6.243737	3.369889	2.365943	**11.979569**
24	4.57504	3.156951	3.037967	**10.769958**

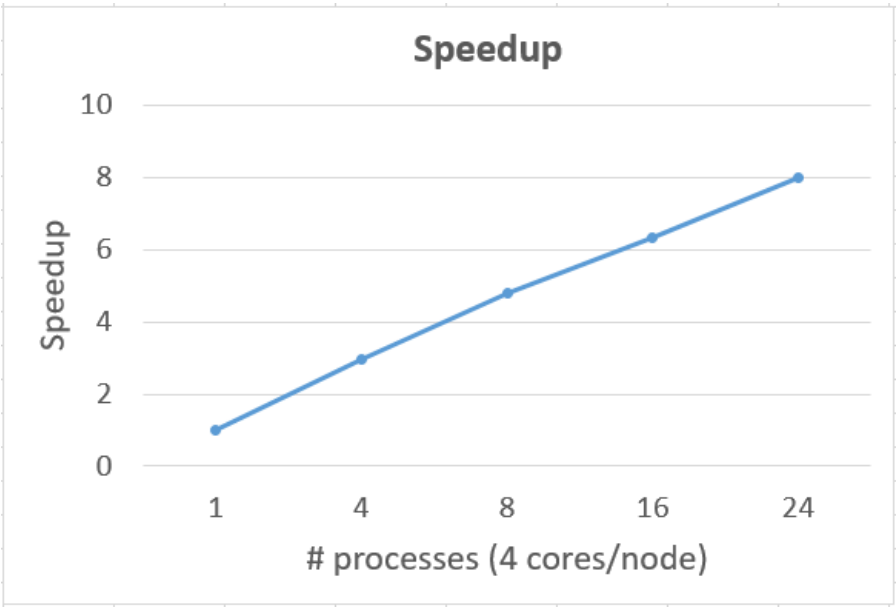


(3) 4 – Node



數值:

process	CPU	Comm	I/O	total time
1	70.478291	0	4.588245	75.066536
4	19.559909	2.786292	3.076214	25.422415
8	11.032228	2.725114	1.936369	15.693711
16	6.31837	3.454113	2.12743	11.899913
24	4.399852	3.011455	1.965647	9.376954



- **Discussion:**

  (1) 由圖可知，bottleneck 應該是在 processes 間做 communication 時是最耗時間的部分。隨著 process 越多，communication time 越長，找到適合數量的 process 或許是一個不錯的解法。

  (2) **Strong Scaling:** 由圖可知，隨著 process 數上升，總體的數字量不變，每個 process 需要處理的量越來越少，parrallel 處理下來，total time 也因此不斷減少，我的做法，如圖 **Speedup** ，比較接近 Strong Scaling 的概念。

  (3) **Weak Scaling:** 拿 test case 中的 8 及 19 為例，test case 8 有 100,000 筆數字，test case 19 有 400,000 筆數字，數字量及 process 量都增加 4 倍，由以下圖測試畫面可知，完成時間也是能幾乎達到一樣，也有接近 weak scaling 的概念。

```
[pp19s96@apollo31 hw1]$ srun -n 2 ./hw1 100000 cases/08.in 08.out
total_cpu_time: 0.001356
total_com_time: 0.000006
total_io_time: 0.004799
total_time: 0.006652
[pp19s96@apollo31 hw1]$ srun -n 8 ./hw1 400000 cases/19.in 19.out
total_cpu_time: 0.002201
total_com_time: 0.000503
total_io_time: 0.004428
total_time: 0.007792
```

4. Conclusion

這次的作業讓我知道及如何使用 OpenMPI，process 之間的溝通寫成程式並不是那麼容易。一開始寫時，最常遇到的 bug 為有 send message 但卻忘記寫 receive，導致不會報錯並且在那邊空等。最困難的是當遇到邏輯錯誤時(想 process 如何溝通)，bug 通常最難解。