Parallel Programming Hw1

111062503 資工系碩二 吳冠志

A. Implementation Detail

本次作業透過 MPI 來實作 Odd-Even sort，我認為此問題主要有兩大塊問題需要思考與實作，一是資料如何平均的被分到各 Processor，二是 Processor 如何透過 odd-phase 和 even-phase 的 Compare and Swap 對整個序列做 Sorting，我將會分點對其描述。

p.s. 為了方便解釋，我假設某次 testcase 共有 n 筆資料，且共有 p 個 processors 可做平行化的 odd-even sort。

1. 資料分配

為了做妥善的資料分配，我們須事先得知 n/p 的商及餘數。

```
int data_size = arraySize / size;
int remaining_elements = arraySize % size;
```

Fig1. data_size 和 remaining_elements 分別代表商數及餘數

data_size 和 remaining_elements 分別代表各個 Process 會被分配到的基本 Data Size 以及會被多分配一個資料的 Process 數，我設計 rank 0~rank=remaining_elements-1 的 process 會被分到 data_size+1 的資料數，其餘被分配到 data_size 的資料數，以將全部資料妥善分配。

e.g. 下圖為分配資料範例，其中 n=17，p=5。

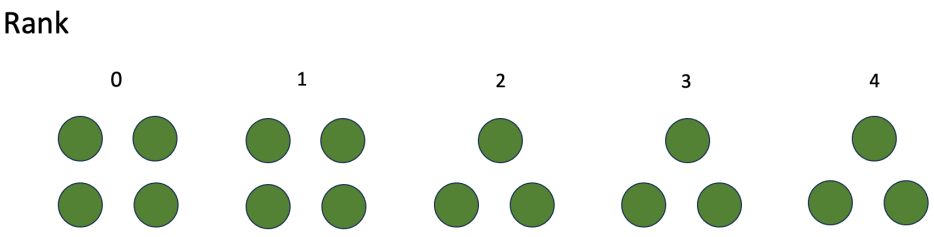

Fig 2. 資料分配示意圖，圖中綠點分別代表各 Data

2. Swapping Algorithm Design

接著說明 sorting 步驟

Step 1. 透過 spread::sort 的 floatsort 模組將各 process 的 local data 進行 local sort

Step 2. 重複指定 round 數，每 round 對各 process 執行 odd sort 和 even sort，先令各 sorting 中 rank 比較小的為 Sender，比較大者為 Receiver

在此步，我們會傳一個參數 odd，1 代表 odd-sort，0-代表 even sort

我們希望每次 odd/even sort 結束後，Sender 拿到較小的值，Receiver 拿較大的值，且兩個 process 的資料成 ascending 排序。
我們會比較 Sender 中最大的資料與 Receiver 中最小的資料，若前者小於等於後者，則代表兩 processor 無需交換，因其各自以取得自己所需之資料。
反之，將兩 process 的資料傳到彼此 process，與自己原有的資料比較取得其應取得之資料（Sender 取得小的資料，Receiver 取得大的資料），且保持 size 相同。

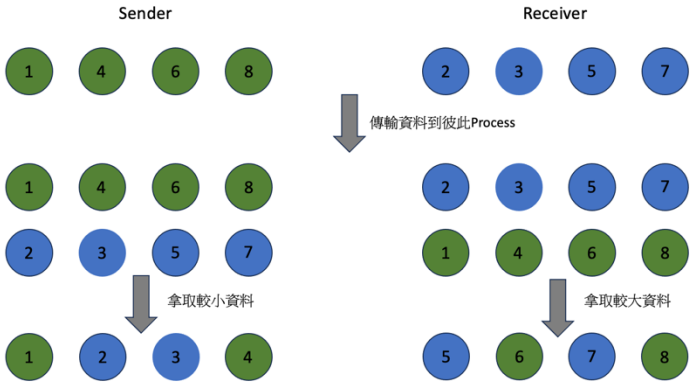

Figure 3. 兩 Process 交換資料範例

在 swap 的實作上，除了 my_data 為我們讀取的資料外，我額外 malloc 兩塊空間 received_data 和 swaping_data，分別代表從對方 process 接收到的資料以及比較後儲存結果的空間。並透過變數 mode 來確定上一輪將比較後的結果儲存在哪裡。若上輪將結果存於 swaping_data，本輪就應該要比較 swaping_data 和 received_data，並將結果存於 my_data，反之亦然。
關於重複 round 數，須根據 process size 的奇數偶數和各 process 之資料數量是否完全相同判斷。若有 p 個 process，考慮 worst case 此 array 中最大的 Data 被分配到 rank==0 之 Process，此資料要被轉移到最後一個 process 至少要 p-1 次 swap，及需要 p/2 次 even-phase swap 和 p/2 – 1 次 odd swap，此為要確保成功排序的最小交換次數。

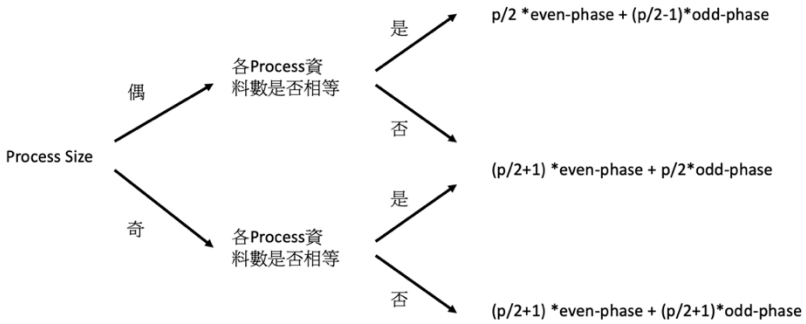

Figure 4. 不同情況下所需 swap 數，其中 p 為 process size

若 process size 為奇數，則在 worst case 時最大資料位於 0 號 process 且 0 號 process 不會參與第一次 swap，因此需要 p 次 swapping，分別為(p/2+1)次 even-phase swapping 和(p/2)次 odd-swapping，若此時各 process 所含有之 data 數量不相等時，則在 worst case 下可能需要多一次 swap，因為部分資料因為兩 process 所含 Data size 大小不同而需在原本 process 上 waiting 無法直接向目的地 process swap，下圖為範例，共有 5 個 process，各含有 6,6,5,5,5 個 Data，Data 編號代表其在此 array 之順位，1 號代表最大的，以此類推，本次舉例只寫出前 12 大的 Data 進行說明。

可以看到在第一次的 Odd Swap 中 6 因為 rank1 和 rank2 的 Data Size 相差 1 而無法成功 Swap 到 Rank2 的 Process 中需要等下一次的 Odd Swap，導致下一輪 12 同樣因為 Data Size 限制卡在 Rank0 中無法往目的地 Swap，造成分別需要 3 次的 even-phase swap 和 odd-phase swap 才能獲得完整的 Sorting Array。

Rank	0	1	2	3	4
Data Size	6	6	5	5	5
Original Data	1, 2, 3, 4, 5, 6	7, 8, 9, 10, 11, 12			
After Even Swap	7, 8, 9, 10, 11, 12	1, 2, 3, 4, 5, 6			
After Odd Swap	7, 8, 9, 10, 11, 12	6	1, 2, 3, 4, 5		
After Even Swap	12	6, 7, 8, 9, 10, 11		1, 2, 3, 4, 5	
After Odd Swap	12	11	6, 7, 8, 9, 10		1, 2, 3, 4, 5
After Even Swap		11, 12		6, 7, 8, 9, 10	1, 2, 3, 4, 5
After Odd Swap			11, 12	6, 7, 8, 9, 10	1, 2, 3, 4, 5

Figure 5. Process Size 為奇數且各 process 含有資料數不全相同範例

若 Process Size 為偶數則大家都會參與第一次的 Even-phase swapping 因此需 n-1 次 Swapping，但若各 process 所含資料數不全相等，則一樣會遇到因為 process 所含資料數不同而需在本地 process 等待的情況，需要多 Swapping 兩次。下圖為 Process Size 為偶數且各 process 所含資料數不全相等的情況。

Rank	0	1	2	3	4	5
Data Size	6	6	6	5	5	5
Original Data	1, 2, 3, 4, 5, 6	7, 8, 9, 10, 11, 12				
After Even Swap	7, 8, 9, 10, 11, 12	1, 2, 3, 4, 5, 6				
After Odd Swap	7, 8, 9, 10, 11, 12		1, 2, 3, 4, 5, 6			
After Even Swap		7, 8, 9, 10, 11, 12	6	1, 2, 3, 4, 5		
After Odd Swap		12	6, 7, 8, 9, 10, 11		1, 2, 3, 4, 5	
After Even Swap		12	11	6, 7, 8, 9, 10		1, 2, 3, 4, 5
After Odd Swap			11, 12		6, 7, 8, 9, 10	1, 2, 3, 4, 5
After Even Swap				11, 12	6, 7, 8, 9, 10	1, 2, 3, 4, 5

Figure 6. Process Size 為偶數且各 process 含有資料數不全相同範例

B. Experiment & Analysis

1. System Spec

   本次使用課程提供的 Apollo 進行本次實驗，本次主要的實驗測試資料為 38.in，Data Size 為 536831999，因其資料量夠大可明顯觀察出平行化所帶來的好處。

2. Performance Metrics

   以下解釋各 metrics 的量測方法，本次透過 MPI_Wtime()計算各 metrics 時間。

   a. CPU time(Computing time)

      在 MPI_Init 後及 MPI_Finalize 前加上 MPI_Wtime()計算差值即可得到整個程式執行時間，此執行時間扣除 Communication time 和 IO time 即可得到 CPU time。

   b. Communication time

      本次透過 MPI_Sendrecv(…)實現 process 之間的溝通，會在一開始傳輸一個資料，判斷需要 Swapping 後傳輸自己的所有資料並接收對方 process 資料進行 Sorting。我們在每次的 MPI_Sendrecv(…)前後加上 MPI_Wtime()計算本次 communication time，對其進行加總可得 total communication time。

   c. IO time

      在 MPI_File_read_at(…)前後和 MPI_File_write_at(…)前後加上 MPI_Wtime()，計算差值並進行加總計算 IO time。

   本次實驗各 process 各自計算其 CPU time、Communication time 及 IO time，reduce 到 process 0 後計算 performance

3. Plots: Speedup Factor & Profile

   a. Strong Scalability

      我們先固定 node=1，比較 performance 在不同 process 數下的表現

# of Nodes	# of process per node	Comm. Time	IO Time	CPU Time	Total Time	Speedup
1	1	0.000000	15.050703	27.192119	42.242822	1.0
1	2	0.668020	14.061236	15.191384	29.92064	1.4118288
1	4	1.348338	14.658995	8.580374	24.587707	1.7180464
1	8	1.930287	12.641657	5.594730	20.166674	2.0946846
1	12	2.564795	13.339217	4.540847	20.444859	2.0661831

Table 1. Single Node 下不同 process 數下的表現

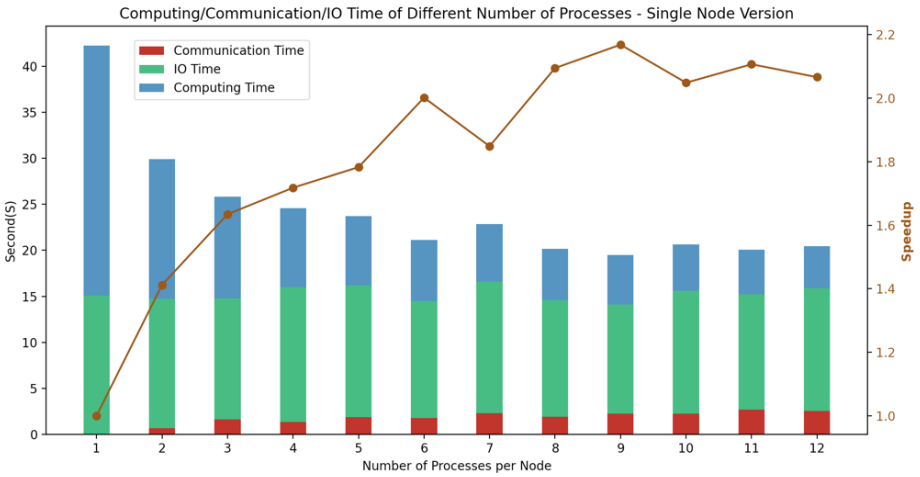

Figure 7. Single Node 下不同 process 的時間分佈及 Speedup

由 Table 1.和 Figure 7. 可以觀察到 IO time 會隨著 process 數數量提升而減少，因為每個 process 需負責讀取和寫入的資料量變少，但 process 數到一個數量後，增加 process 數反而造成 IO time 提升，推測可能是因為硬體資源已到達飽和，再增加 process 反而會增加資源競爭，造成 IO time 拉長。

CPU time 隨著 process 數量增加而遞減非常直觀，也是本次平行程式優化的最大獲益，但 process 數到 8 之後再繼續增加 process 所獲得的時間加速非常有限，反而因為 process 數量增加而上升的 communication time 會吃掉 CPU time 的效能提升，造成總執行時間不減反增。

接著看複數個 node 下不同 process 數的表現

# of Nodes	# of process per node	Comm. Time	IO Time	CPU Time	Total Time	Speedup
3	1	2.809727	12.809484	10.758029	26.37724	1.0
3	2	1.431577	14.305311	6.557174	22.294062	1.1831509
3	4	2.291034	13.186288	4.493999	19.971321	1.3207558
3	8	2.460254	14.635769	2.742485	19.838508	1.3295979
3	12	2.707608	13.272493	2.081619	18.06172	1.4603947

Table 2. Three Nodes 下不同 process 數下的表現

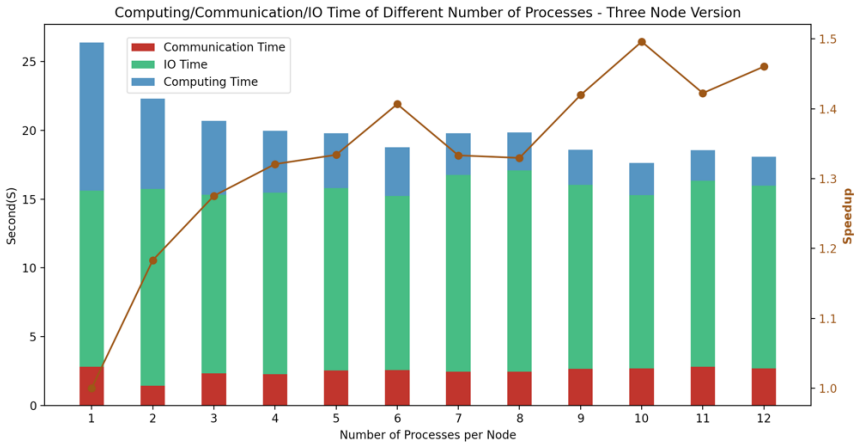

Figure 8. Three Nodes 下不同 process 的時間分佈及 Speedup

從表和圖我們可以得知其 Communication time 不隨著 process 數上升而有明顯增加，反而有增有減，代表跨 node 間的溝通因 process 數量增加造成的時間提升不明顯，這可能是由於跨 node 溝通耗時較長導致。

CPU time 因 process 數量降低而有的效能提升非常顯著，但提升到一個程度後即使單看 CPI time 的 speedup 提升仍然明顯但可能相差不到多少秒，因此在此情況不同 process 數量下的 total time 和 speedup 可能主要被 IO time 所影響，因其是這三個時間中耗時最長的且波動大。從圖表觀察 IO time 並無固定規律可能與硬體資源的分配有關。

b. Speedup Factor

這邊只計算 CPU time 的 speedup，也就是拿 single process per node 與 multi-process 比較 CPU time 並與 ideal speedup 比較

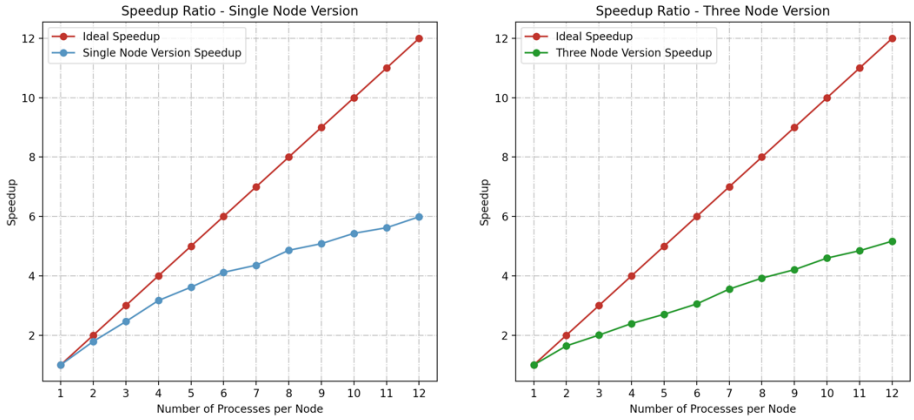

Figure 9. Single Node/ Mulit Nodes CPU time Speedup

只考慮 CPU time 的 speedup，不管是 Single Node 還是 Multi Node，其 Speedup 遠遠不及 Ideal Speedup，在 12 個 process 時 Speedup 約為 Ideal Speedup 的一半。雖然這件事在平行程式中已

司空見慣且很容易理解，但我在這邊也嘗試分析造成 Speedup 下降的可能原因。

- 有 Process 在 Swap 時 idle：在 process 數為奇數時，even/odd phase swap 均會有一個 process 不參與 Swap，而 process 數為偶數時在 odd phase swap 時會有兩個 process 不參與 swap，這些 idle 的 process 是造成 speedup 降低的一個可能原因。
- Swapping 耗時：本次 Local sort 使用的演算法為 floatsort，其概念類似 radix sort，按照特定位數進行分桶，並依照桶中數據進行排序，其不為 caparisoned-based sorting algorithm，但若有多個 process，雖然 process 內部資料仍透過 floatsort 實作，但 process 之間是透過 compare-and-swap 交換資料，造成資料移動較緩慢，speedup 不如 ideal
- Data Size 不全相同：若只有一個 process，無須考慮不同 process 的 Data size，雖然本次設計已經讓多個 process 間的資料量盡量平均，仍不免發生 process 數量無法整除資料量導致不同 process 之 Data Size 不全相同，這樣的差異將導致額外的 Swapping 次數，造成 speedup 下降

4. Compare

a. Sort Algorithm Comparison

Porcess 在彼此間交換資料前，會先在自己內部進行 local sort，選擇快速且合適的演算法很重要，以下簡單介紹我目前已知常用 sorting algorithm

Std::sort：C++標準庫提供的排序算法，根據 data 的比較來做排序，時間複雜度為 O(nlogn)最壞情況為 O(n^2)

Qsort：C 標準庫提供的排序算法，使用 quick sort algorithm，取決於數據最壞情況時間複雜度為 O(n^2)

Boost::sort(spreadsort 和 floatsort)：是 Boost C++提供的函式庫，包含 spreadsort 和 floatsort，是相對高效的非排序演算法，spreadsort 使用分不是排序，而 floatsort 則特別針對浮點數排序做優化，通常具有線性或接近線性的時間複雜度。

本次實驗透過 24 個 processes 各自取得 38.in 中自己負責的資料，並記錄其 local sort 所需時間，最後透過 MPI_Allgather 集合大家 sorting 所需時間並計算平均值，目的是為了充分考慮不同資料分布的排序效能

Sorting Algo	*Std::Sort*	*qsort*	*spreadsort*	*floatsort*
Time(s)	2.5548	4.6639	1.3843	1.3786

Table 3. 不同 Sorting Algorithm 表現

根據上表，我們最後選擇 floatsort 做為本次 local sort 實作演算法。

b. 固定 process 數量，不同 node 的 performance

固定 process 數為 12，比較不同 node 數的 performance

# of Nodes	# of process per node	Comm. Time	IO Time	CPU Time	Total Time
1	12	2.994662	15.070617	4.500326	26.37724
2	6	2.173124	14.973304	4.437393	22.294062
3	4	1.815987	13.177506	4.471689	19.971321

Table 4. 固定 process 數量下不同 Nodes 數表現

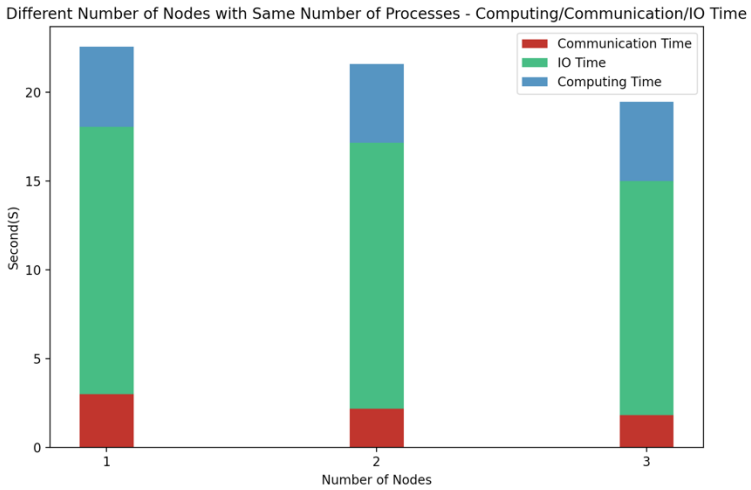

Figure 10. 12 個 processes 在不同 Nodes 下的時間分佈

從上面圖表可以看出，在相同 process 下，使用多個 node 對 CPU time 的效能幫助不大，使用 one node 和 three nodes 提升不到 0.1 秒。但使用多個 nodes 在 communication time 和 IO time 所帶來的效能差異是非常明顯的，此與我的理解相悖。我認為將 process 分散在多個節點上可能造成溝通成本提升，因為跨節點通信通常透過網路進行，可能導致較高的 communication time。推測造成此結果的原因可能與硬體設備或網路性能有關，多節點系統的拓撲結構設計、節點的獨立網路連接都有可能使得跨節點資料溝通更快速，因此將 process 平均分散在各 node 確實有可能讓 communication time 不增反減，而在做 IO 時因為不用考慮其他 process，只需讀取和寫入自己負責的資料即可，因此將 process 分散在不同 Nodes 更能提升 IO 的 performance。

5. Discuss
   a. Compare I/O, CPU, Network performance. Which is/are the bottleneck(s)? Why? How could it be improved?

綜上實驗得出幾個結論：1. 只要 process 數量提升，就可以降低 CPU

time 的部分 time cost，但此成效隨著 process 數提升而逐漸趨緩 2. Process 數量提升帶來的缺點是 communication time 也會隨之提升，甚至到後來會超越平行化為 CPU time 所帶來的效能提升 3. IO time 無固定的規律可循，可能與硬體配置等多個因素有關。

因此得到以下結論：在 process 數量少時，CPU 是主要的 bottleneck，因為無大量的溝通需求反而需要花很多時間在計算。隨著 process 數量提升，Network performance 漸漸成為 bottleneck，因此時 CPU time 隨 process 數量所降低的幅度已越來越少，此時 process 間的溝通成本已成為平行程式優化的主要障礙。要優化這兩個時間的主要方法是透過資料數量、分佈等可以預先獲得的資訊選擇一個合適的 process 數量，這個能需要大量的實作經驗以及實驗。而 I/O 我認為是這三個時間裡面最難優化的一個，其有可能為 bottleneck 但造成其耗時的原因可能有許多如磁碟速度、檔案系統效率等，甚至同一時間在 server 上的使用者數量也是影響 I/O 的主要原因，本次 I/O 各 process 只做了一次的檔案讀寫因此我認為優化空間不大，唯一嘗試的優化方法是在比較少人使用 server 時測試資料，體感來說凌晨兩點到五點使用者較少似乎有比較容易獲得好結果，但無確切證據佐證只在 Discussion 說明我的經驗。

b. Compare scalability. Does your program scale well? Why or why not? How can you achieve better scalability? You may discuss the two implementations separately or together.

為了驗證 scalability，我將比較不同 data size 在不同 process 數量下的表現，所選的資料集為 16.in,24.in,28.in,,36.in，Data Size 分別為 54923, 400009, 1000003,536869888，在不同 process 數量比較其 CPU time

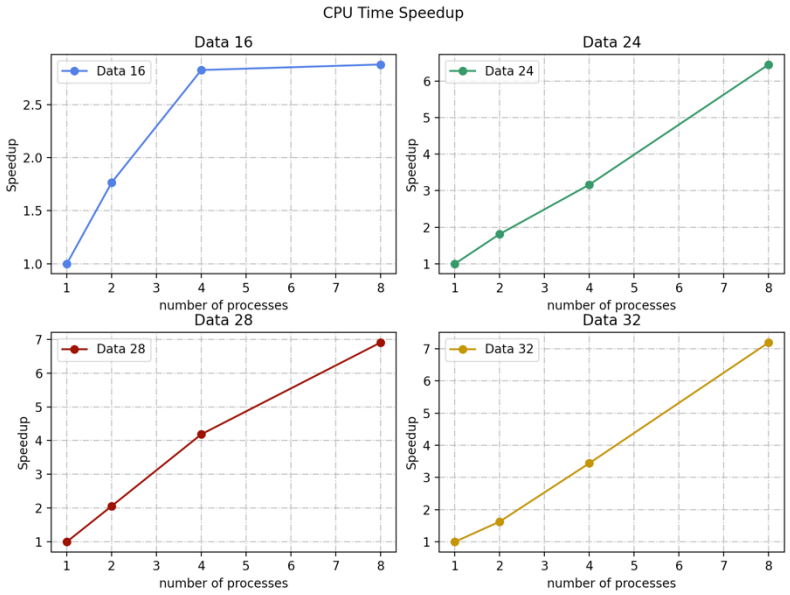

Figure 11. 不同資料及下 Speedup 表現

由上圖可看出我的方法在不同大小 Dataset 上隨著 process 數的提高其 CPU time Speedup 均有不錯的表現，不會有程式在大的資料上表現不佳的情形發生，因此，我認為我的程式具有好的 Scalability

C. Conclusion

作為平行程式這堂課的第一個作業，透過實作的過程讓我理解了很多，MPI 的使用、對 Even-Odd Swapping 的全面理解、還有細節的演算法及程式碼優化等等，也深刻理解到老師上課所說的平行程式撰寫想法上要與傳統寫程式不同，在考慮多個 process 後可能發生許多無法直觀解決的程式問題，在思考的過程也讓我對程式的運作有更深刻的理解。也為了獲得更短的程式執行時間，也需要修改以前寫程式的習慣，例如用 Bitwise Operation 取代除法，使用不一樣的 Sorting Algorithm 等等，在撰寫平行程式的同時也讓我對基本的程式設計有了不一樣的理解和寫法。

我在這個作業的實作上獲益良多，期望在未來可以有更多收穫。