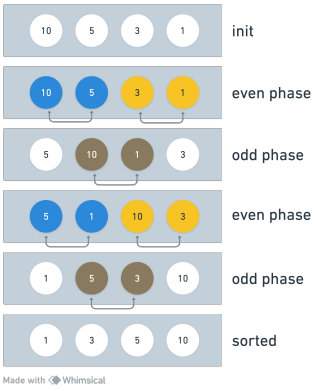
HW1 Report

資應所碩二 111065531 郭芳妤

Introduction

本次作業要求在指定數量的 node、process 下，使用平行化的方式實作 Odd-Even Sort，並盡可能地優化其效能。Odd-Even Sort 的 serial code 是其實是 bubble sort 的變化版本，會在每次 iteration 分別以 odd-index、even-index element 交替作為主要的比較對象，進行比較大小與交換位置之後可完成排序，在單一 process 的情況下，這個演算法與 bubble sort 相同，worst case 需要 $O(N^2)$ 時間，但如果是使用 N 個 process 來排序 N 個 element，則我們花費 $O(N)$ 時間可完成排序：



當然在實際情況下，我們不可能擁有無上限的運算資源，因此如何在有限的 process 下實作並優化這個演算法也是在本次作業中要處理的部分。在多數 testcases 裡，我們會遇到一個 process 需要處理多個 elements（partial array）的狀況。總結來說，實作上要處理的問題包括：

- `Load balance`：如何盡可能平均地分配 elements 給各個 process
- `Local sort (Compare time)`：每個 process 取得 partial array 之後，要先對 partial array 做 sorting。另外，在 local 排序完後，進入 odd/even phase 進行 element 交換時，也需要進行比較與交換。
- `Communication`：Odd/even phase 時，每個 process 會與 neighbor process 進行交換，在此處交換的機制（一個一個進行 send / receive 或直接建構一個 buffer 整筆傳送）也會影響總時間。
- `IO`：使用 MPI_Open()、MPI_Read()、MPI_Write() 對檔案進行讀寫

Implementation

1. **Load balance**：一旦發生 process 之間 loading 分配不均，就極可能出現已經 sort 好、或者已經完成 compare 的 process pair 需要等待（空轉）的情形，因此我們希望所有的 process 可以被分配到盡量一致數量的 elements。這邊使用了 ceil，即：

   ```
   int parti = N/size + 1;
   // parti 為每一個 process 被分到的 elements 個數，以下將使用 parti 作為 size of
   // partial array 的表述
   ```

   這個方法可以達到平均 loading，同時也可以避免使用 floor 的話最後一個 process 被分配到過多 elements 而等過久的情形（嘗試過後甚至會出現 runtime error）

2. **Initialization local sort**：在開始 odd-even phase 以前，每個 process 需要針對各自被分配到的 partial array 先進行排序，這邊嘗試以下兩種 sorting 方法，實驗起來 spreadsort 通常花費少時間，因此最終的版本選擇 spreadsort()：

```
float_sort()：138 sec (total time)
spreadsort()：132 sec (total time) --> win!
```

3. **Communication & odd/even phase sort**：本來一開始最 naive 的想法是在交換的 phase 利用 send()、receive()，不斷把小的 element 往 rank 較小的 process 進行換，但後來發現這樣太浪費時間了，後來發現可以利用 `MPI_Sendrecv()` 一口氣把 process 的 array 搬到對方的 process 上再進行比較，實作方式類似 merge sort 時 merge 的概念：
   a. **version 1 [原始版]**：每一個 process 都在一開始宣告一個 buffer，在 communication 階段把 neighbor 的 partial array 放在這個 buffer 以便在自己的 process 上進行比較；另外，我們還需要一個 tmp_array 來放比較後放入的值（大小為 2*parti，因為是兩個 array merge 之後的大小）。以 even phase 的 rank 2 和 rank 3 的 process 舉例(：
      i. rank 3 會將自己的 partial array 傳到 rank 2 事先開好的 buffer 空間，rank 2 此時就可以透過一個簡單的 for loop，依序把 rank2 和 buffer（屬於 rank3 的 partial array）按照值的大小擺放進 tmp_array，最後取前 parti 個 element，這樣 rank2 在交換好的同時也保證是 sorted 的。
      ii. **與此同時**，rank3 也會將 rank2 的 partial array 傳到自己的 buffer 空間，使用 for loop 一樣把值放入 tmp_array，唯一的差別在於：rank 3 為 rank 2 的 right neighbor，所以會保留 tmp_array 的後半段作為 sorted array，大小同樣為 parti。
   b. **version 2 [優化版]**：這一段可以透過兩個額外的判斷減少比較大小&交換的時間
      i. **先確認否真的需要 communicate**：進入 odd even phase 以後，花費時間來自 communication 和交換比較兩個 partial array，但，如果有其中兩個 process pair 其實已經是 local sorted & pair-wise sorted，則我們其實可以讓他們不進行溝通和比較，只要等尚未 sort 好的 pair 工作即可。比較的方式是把 partial array 的 head 或 tail element 傳給對方，確認兩個 process 之間是否有必要進行 communicate 和 merge，如果不用，則不做任何事。
      ii. **Phase + rank 的綜合判斷**：加速除了最後一個 process 可能因為取 ceil 而被分配到的個數較少（或者沒有分配到任何 partial array），其餘的 process 被分配到的 element 個數相同（= parti），因此我們可以在 merge 時利用兩個不同的 function（Merge_low & Merge_high），讓 pair 中的左節點使用 Merge_low() 由小 → 大取到 size = parti 時直接 return、右節點使用 Merge_high() 由大 → 小取到 size = parti (or 原 partial array 大小) 時直接 return，以此減少全部比完再放入 tmp_array 的時間。

Experiment & Analysis

1. **System Spec**：

   本次實作是利用課堂提供的 Apollo，下方的實驗是以系統的 3 個 test node 完成的

2. **Performance Metrics & 計算方式**：
   以下均使用 MPI_Wtime() 紀錄不同程式區塊的時間（單位：秒）
   a. Communication time：紀錄在 odd even phase 時所有 call MPI_Sendrecv() 的時間，包括交最大最小值以及交換一整個 partial array 的時間
   b. I/O time：程式讀寫檔案的時間，即 call MPI_Read()、MPI_Write() 的時間
   c. Computation time：這邊主要紀錄進入 odd even phase 以前 local sort 的時間，以及進入 odd even phase 的 loop 之後，執行 Merge() function 的時間
   d. Total Time：程式總時間
3. **Test Case chosen（使用大小測資驗證 scalability）**
   a. 大測資：Data 38 - 在這筆測資的 size 底下，我們算出來的 partial array 大小為 44,736,000，最後一個 process 的 partial array 大小為 44,735,999，基本上達到每個 process 的 load 平衡的狀態。

> Experiment Result（On judge：12.80 sec，setting：N = 3, Process = 12）

```
{
    "n": 536831999,
    "nodes": 3,
    "procs": 12,
    "time": 180
}
```

b. 小測資：Data 26 - 在這筆測資的 size 底下，我們算出來的 partial array 大小為 16,668，最後一個 process 的 partial array 大小為 16,645，基本上達到每個 process 的 load 平衡的狀態。

> Experiment Result（On judge：1.32 sec，setting：N = 2, Process = 24）

```
{
    "n": 400009,
    "nodes": 2,
    "procs": 24,
    "time": 60
}
```

Performance on single node

Proc	1	2	3	4	5	6
COM	0.0	0.74	1.95	1.16	1.74	1.78
IO	13.62	12.08	11.74	10.37	10.48	11.42
CPU	26.92	15.48	11.14	9.27	8.08	7.55
Total	40.54	28.30	24.83	20.80	20.29	20.74

Proc	7	8	9	10	11	12
COM	2.05	2.15	2.28	2.42	2.41	3.12
IO	9.93	9.07	10.98	10.91	11.25	9.56
CPU	6.95	6.25	6.1	5.8	5.98	5.48
Total	18.93	17.47	19.35	19.13	19.64	18.15

上表為使用單一 node 時，設定不同 process 數量的表現結果，我們得到以下觀察：

1. Total time 隨 process 數量提升而減少，但在 process 數量 ≥ 5 之後，減少的幅度趨緩，推論可能是因為 IO 不穩定以及隨著 process 數量增加，相應的 communication time 也會增加，而 CPU time 為因為平行化而獲得最顯著 speed up 的部分。
    a. 圖 1：IO time 部分並不隨著平行度提高而減少
    b. 圖 1：Communication time 確實隨著平行度提高而增加
    c. 圖 1：CPU time 隨平行度提升有最為明顯的減少
2. 圖 2 為總執行時間的 speed up factor：可以看到平行化雖然讓程式效能有所提升，但並不隨著平行度的提升而提升，換言之，在這個實作版本下，我們即使增加了 process 的數量，也沒有對總體的時間有更好的加速。
3. 圖 3 為 CPU time 的 speed up factor：相較於 total time，CPU 的 speed up 就很明顯，雖然離 ideal speed up 仍有差距，但確實可以看出「排序、compare 等計算時間」因平行化有更好的效能。

圖 1 - 不同 process 下，Data 38 的時間分佈（N = 1）

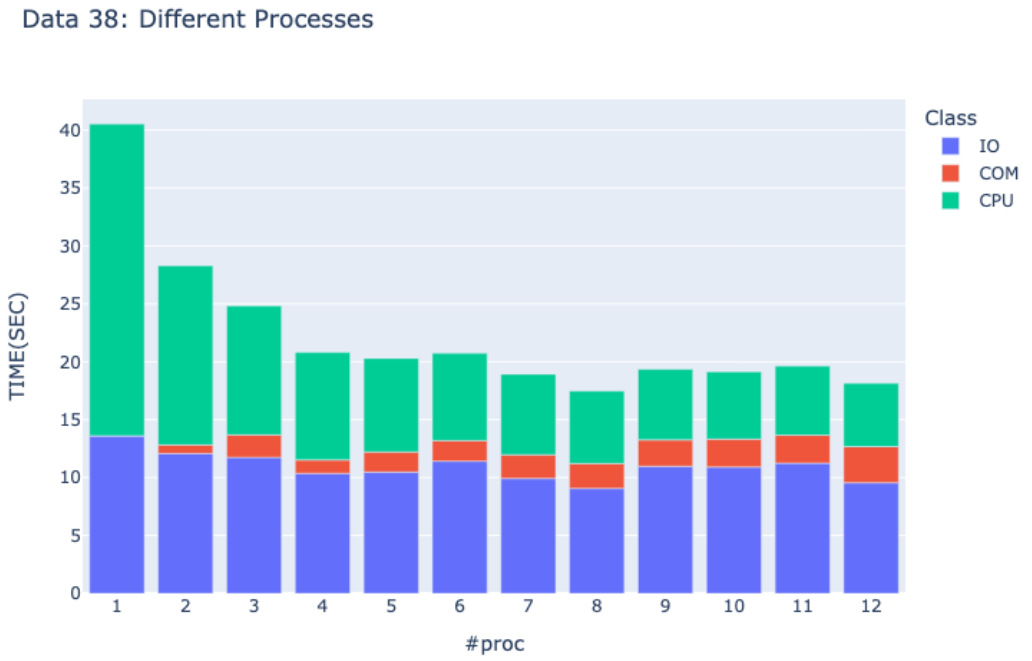


圖 2 - Total time speed up

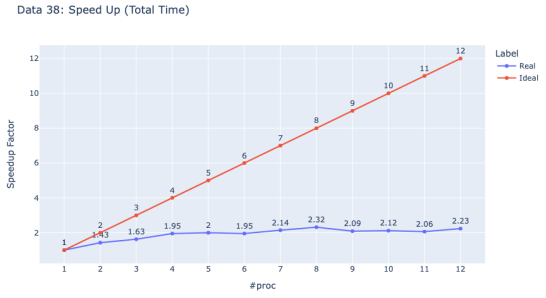
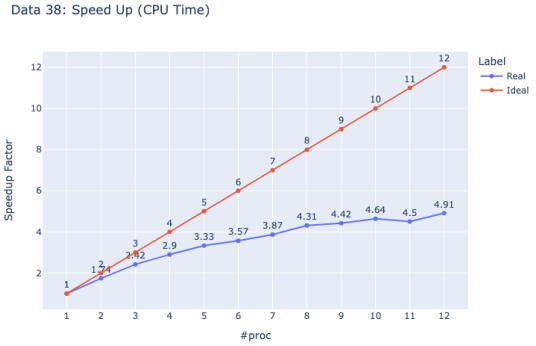


圖 3 - CPU time speed up



Performance on multiple nodes（N=3）

為了確認若 process 被分配在不同 node 上是否會對 performance 有不同的影響，以下也會看設定在 3 個 node 上時，相同 process setting（Proc = 1-12）的表現（註：Proc = 1 or 2 時，系統會強制分配在同一個 node 上）。

Proc	1	2	3	4	5	6
COM	0.0	1.37	3.3	1.66	2.2	1.79
IO	14.15	11.24	12.38	9.38	12.52	6.57
CPU	26.89	15.07	10.5	8.61	7.5	6.88
Total	41.04	27.68	26.17	19.65	22.23	15.24

Proc	7	8	9	10	11	12
COM	2.19	2.36	2.59	2.59	2.75	2.67
IO	4.47	4.37	6.29	5.52	3.54	9.83
CPU	6.56	6.17	5.9	5.54	5.55	5.24

Proc	7	8	9	10	11	12
Total	13.22	12.9	14.78	13.65	11.84	17.74

圖 4 - 不同 process 下，Data 38 的時間分佈（N = 3）

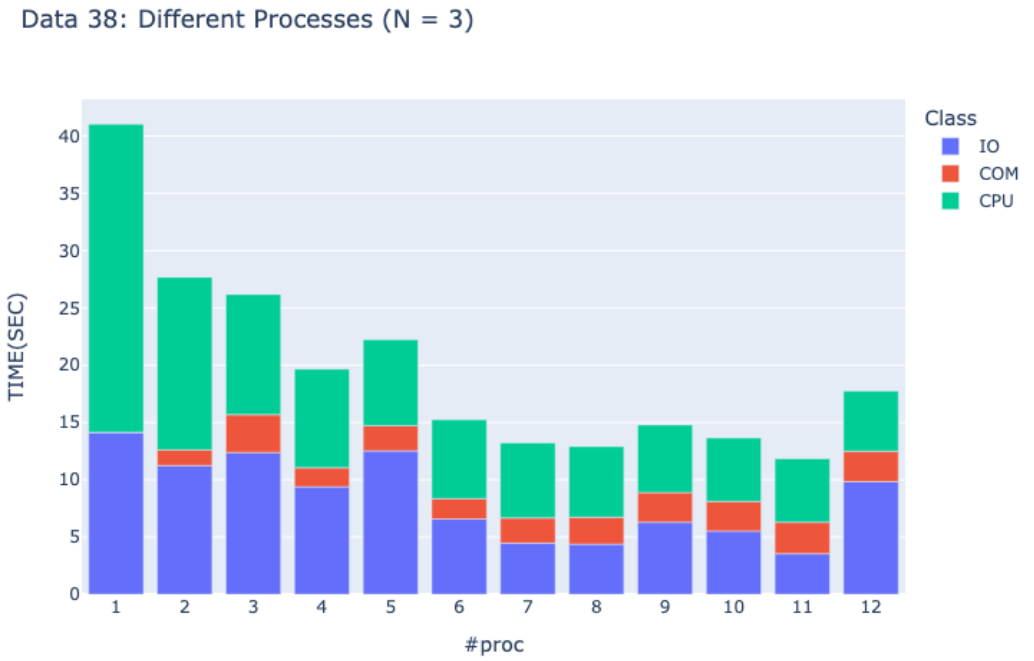


圖 5 - Total time speed up (N = 3)

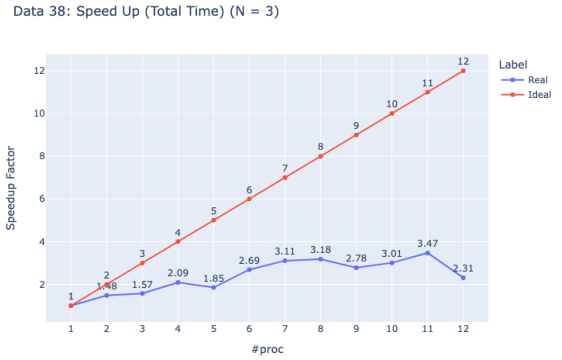
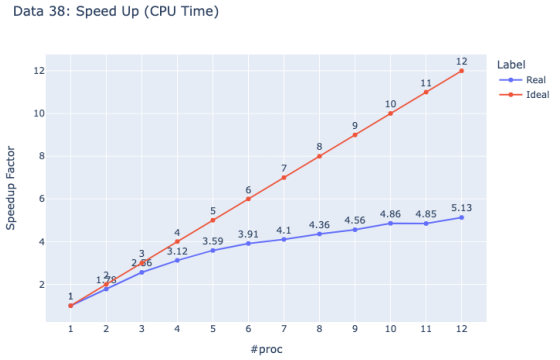


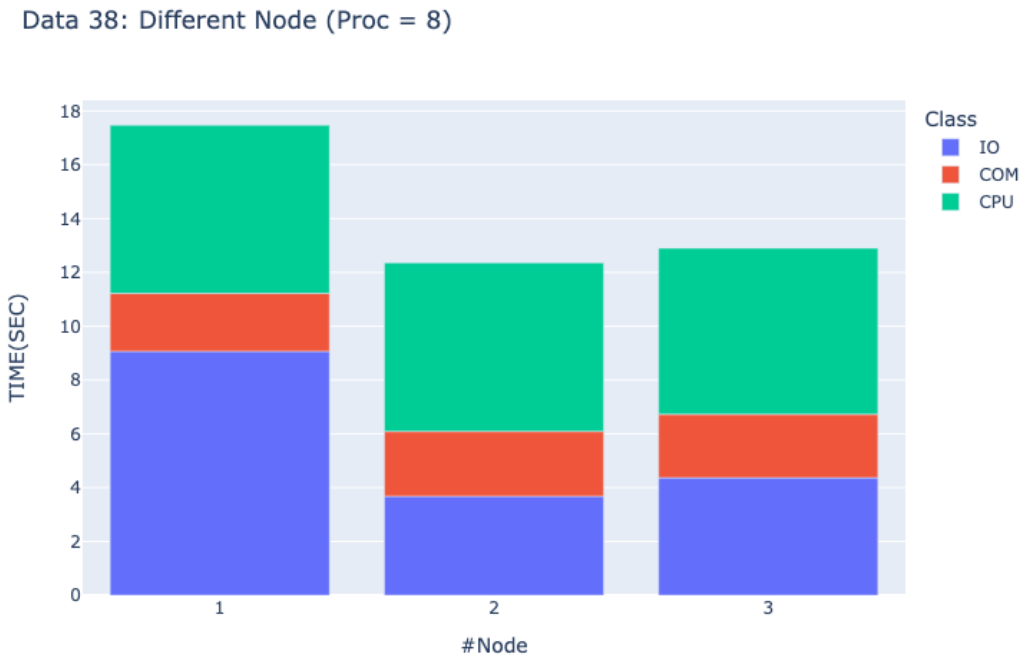
圖 6 - CPU time speed up (N = 3)



1. 觀察上述實驗數據，相較於使用同一個 node（N = 1），在 N = 3 時，CPU time 減少的幅度較大（計算上更有效率），而 IO time 雖然不能觀察到明顯地隨平行度上升而下降的趨勢，但整體而言比 N = 1 花費更少的時間（當然也不排除是 IO time 本身就不太穩定所導致），而這邊比較出乎意料的是 communication time 增加的幅度比 N = 1 還要小（表示沒有因為不同 node 而增加溝通成本）。
2. [比較圖 2 與圖 5] 相比 N = 1，Total time 在 N = 3 時有更好的 speed up，而我們可以從實驗數據和圖 4 推論出這是 IO time 差異所導致。
3. [比較圖 3 和圖 6] 相比於 N = 1，CPU time 在 N = 3 時有更好的 speed up，但整體趨勢相同。
4. 為了驗證上述的結論與猜想，我接下來看了相同 proc 數量（proc = 8）分布在不同 node 上的表現，可以看到如果分佈在單一的 node 上表現最差（花費增時間最久）。在 Communication 和 CPU time 上差異不大，表示演算法在

排序上和溝通時間不會因為跨 node 而有差異。而主要差異確實在於 IO time，可以看到在 node 數量增加時，IO 時間有很明顯的下降，這應該和每一顆 Node 需要處理的資料量有關；而 N = 2 和 N = 3 的 IO time 之所以沒有很大的差異，推論是在分配 process 時 8/2 和 8/3 在分法上沒有很大的差異。

圖 7 - 不同 node 下，Data 38 的時間分佈（proc = 8）



Different Data Size (驗證程式 Scalability)

- 將此方法驗證在小測資（testcase 26）上，所得到的結果如下：可以發現 (1) IO dominate 了整體時間 (2) CPU Time 隨著平行度上升而下降 (3) Communication time 似乎沒有隨著平行度提升有很明顯的上升
- 以 speed up 比較，除了 process 8 因為 IO time + communication time 的突然提高而導致 speed up factor 下降以外，其餘幾個 process setting 下的 speed up 都貼近 linear ideal case（尤其 proc = 4-7），相較於上圖 6 實驗在大測資上的結果：**隨著平行度提升，小測資確實有更好的 speed up 表現，但其實兩者差異並不大**，以此驗證本次實作的 scalability。

圖 8-1 - 不同 process 下，Data 26 的時間分佈（N = 3）

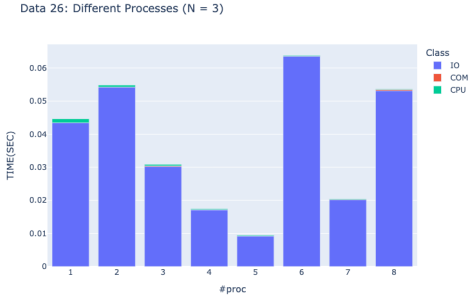


圖 9 - Total time speed up (Data 26)

圖 8-2 - 左圖去掉 IO Time 結果

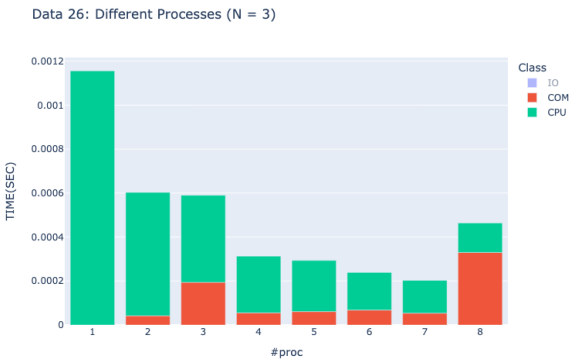
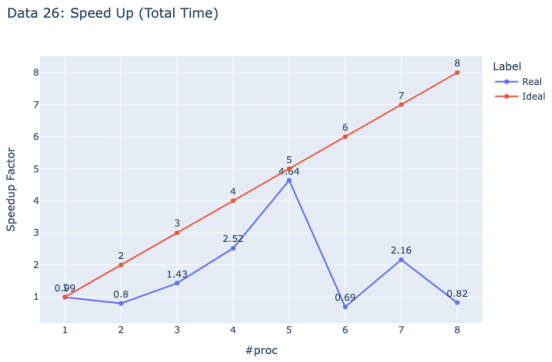
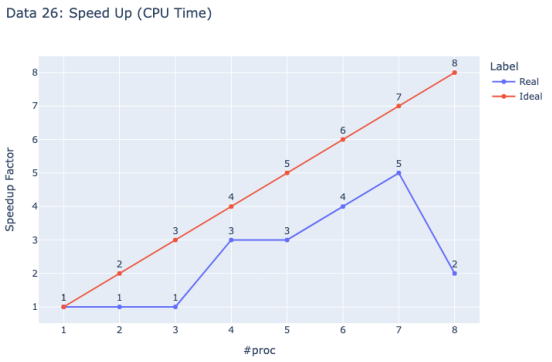


圖 10 - CPU time speed up (Data 26)



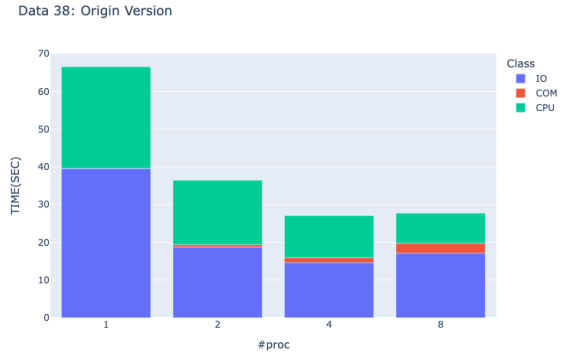


Bottleneck identification & Optimization Strategy

與原始版本（version 1，即尚未區分 merge_low()、merge_high() 的方法）相較，可以發現 bottleneck 主要在 CPU time 上，這是因為使用 naive merge 的話，我們會花費一半不必要的時間在由小到大依序放入 compare 後的值，結果如下：比較圖 12 & 圖 14，可以透過 CPU speed up 看出這邊的 bottleneck 位在 CPU time，而在這邊優化的策略就如同 implementation 所述，考慮到在不同 phase，不同 rank id 需要取用的 sorted array 會分成「較小的半邊」和「較大的半邊」，因此把這兩種情況拆開處理後，CPU time 自然就得到明顯的加速。

圖 11 - 原始版本不同 process 時間分佈比較

圖 12 - 原始版本 CPU speed up



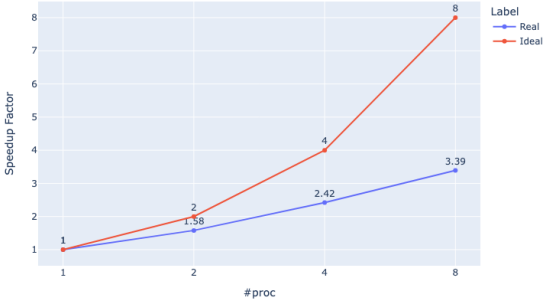
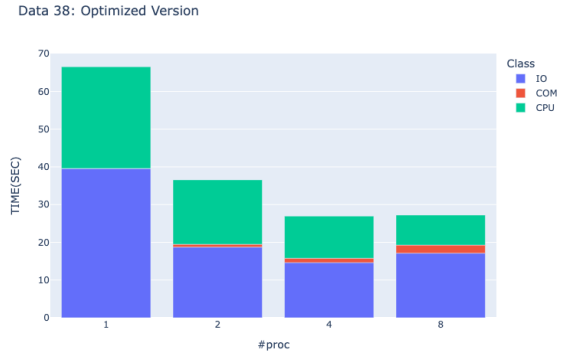
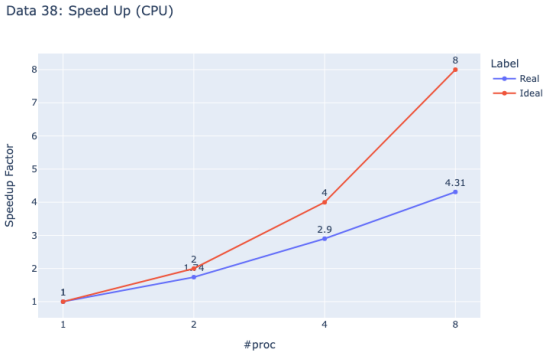


圖 13 - 優化版本不同 process 時間分佈比較

圖 14 - 優化版本 CPU speed up





Experience & Conclusion

綜合本次作業實作與實驗，以下兩點是我在這次作業裡得到的重要結論與收穫：

- 儘管被給予很多資源（比如說更的 process），平行程式效能也會受到演算法與無可避免的溝通成本影響。
- 想要提升平行程式效能，分析程式的能力與心力（包含把總時間拆解成不同的 components 如 IO, COM, CPU time）是必不可少的，因為如果不把這些數字分別算出來，我們很難得知是哪個地方還能再更好、哪些又是無法避免的效能瓶頸。

另外，在實作 odd even sort 時，感受到最困難的是從 serial 到 parallel 思考模式的轉換。在寫 code 的時候會很受到過去寫 serial code 的習慣影響，很難想像同一個演算法應用在多的 node 或 process 上具體而言是如何運作，所以一開始甚至是觀摩了好多份別人的 parallel odd-even sort 才勉勉強強寫出自己的第一份 code。另外，這也是第一次開始在乎自己程式的效能，並且仔細利用工具，分析哪裡是 bottleneck、哪裡可以再改進（以前都是有過 judge 就覺得好棒了，完全沒有想要怎麼讓他更快 XD）。

最後感謝所有超級認真的同學和網路上的資源，大家的每一次共同分析和討論都讓彼此的 code 和實驗呈現更加完善。

Reference

https://github.com/eduardnegru/Parallel-Odd-Even-Transposition-Sort/blob/master/mpi.c

https://github.com/Elven9/NTHU-2020PP-Odd-Even-Sort

http://selkie-macalester.org/csinparallel/modules/MPIProgramming/build/html/oddEvenSort/oddEven.html