HW1 Odd-Even Sort

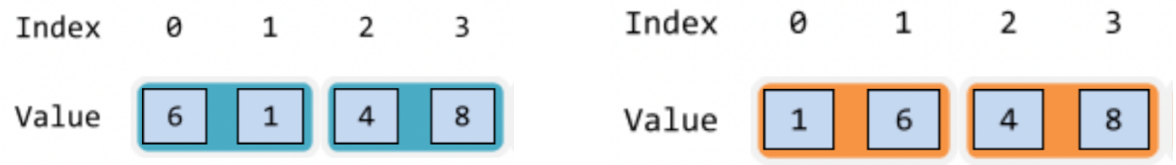
111062613 蔡鎮宇

1. Implementation

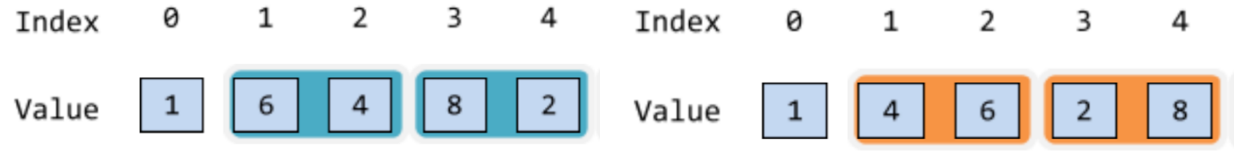
Basic Explanation of Algorithm:

我的做法是先將整個 input array 平均分攤給所有的 process，每個 process 先對自己的 local array 進行 c++ algorithm 中的 sort，時間是 O(nlogn)等級的。

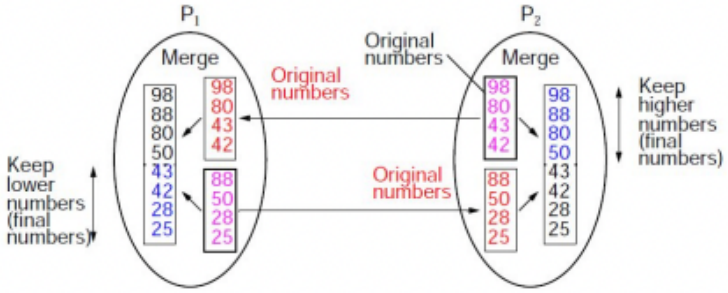
接著進行 Odd-Even Sort，以下圖為例 index 為偶數的 process (index = 0)會和比他 index 大的 process (index = 1)做 merge。這個 merge 就像是 merge sort 中的概念，把兩個 array 的數都拿出來放在一個臨時 array，把比較小的數都分到左邊 array，較大的數分到右邊 array。merge 完之後可以發現 process 0 內的 local array 所有數字會小於 process 1 的所有數字，且兩個 local array 中的數是 sorted 的狀態。此為 Even Phase。



而 Odd Phase 也是如下圖的方式操作，Loop 這兩種 phase 直到終止條件。



如果不論奇偶數 index 的話，可以歸納成下圖。P1 和 P2 的所有數字可以 merge 成一個 sorted array，而 P1 拿到最小的 4 個數，而 P2 得到最大的 4 個數。



為了要找到終止條件，我們要定義 Swap。Swap 的定義是如果左邊的 local array 的 element 在 merge 的過程後跑到了右邊的 array，代表有 swap 發生。我將 Even Phase 和 Odd Phase 合併看成一個 Round，當某 Round 結束而所有的 merge 都沒有發生 swap，則 Odd-Even Sort 可以結束。

Load Balance:

Input size 不一定會整除 process 數，但在一開始我們就要先規定哪個 process 擁有哪些 global element。因此我就用兩個 array 紀錄每個 process 擁有 input array 的起始 index 和結束 index。

假設有 8 個 element 和 3 個 process，則 process 0, 1 會有 3 個 element，process 2 只有 2 個 element。Process 0 一開始拿到的是 index = 0, 1, 2 的 element，Process 1 拿到的是 index = 3, 4, 5 的 element，依此類推。

Sorting Details:

1. 因為避免重複 allocate memory，所以在兩個 process a, b 要 sort 時，我們從 load balancing 的階段可以知道 process a 或 b 的 partition 大小是多少，所以我們可以事先就 allocate 一個固定大小的 array，接著再使用 MPI_Send, MPI_Recv，獲取另一個 partition 的數字們。
2. 此外，這時候 myMerge() function 會依照 process rank 把 element 依照大小放在左邊的 process 或是右邊的 process。此時我們在 merge 的時候不需要真的把兩個 partition 合併成一個大 array，我們只需要 merge 到該 process 所需要的數量即可。
3. 使用 MPI_Allreduce 來計算到底有沒有發生 Swap，用一個變數 swap_result 記錄。一開始寫了一個 if else，如果 swap_result = 0，則 break。
4. 假設原本的 process a 有 x, y array，x 是擁有從 file 讀取、local sort 後的數字，y 有 merge 後的數字。為了省去把 y 的數字再傳送給 x array，MPI_Send(), MPI_Recv()的參數在 Odd 或 Even phase 是相反的，有時候會讓 x 成為裝載和其他 process 溝通 merge 後的數字(亦即 x, y 角色互換)。
5. 要注意有些情況下 rank 0 或是 rank 最大的 process 不會參與 merge，這時候要記得把他們的值填到和上述(4)一樣名稱的 array，以便下個 phase 的 merge。
6. Phase 總數最多不超過 Process 總數 + 1，可以把此條件設定在 while()裏面。

Other Efforts:

1. Loop 的每個 iteration 會跑 Even phase 和 Odd phase 各一次，此時才結算 MPI_Allreduce()，這樣可以讓 MPI_Allreduce()的次數砍半。
2. 使用 boost::sort::spreadsort::spreadsort 取代 C++ <algorithm>的 std::sort()可以讓原本的速度減少 13 秒
3. 使用 MPI_Sendrecv()取代 Send 和 Recv，可以再將時間優化 17 秒。

2. Experiment & Analysis

i. Methodology

(a). SystemSpec

使用的是課程規定的 cluster。

(b). Performance Metrics

How do you measure the computing time, communication time and IO time? How do you compute the values in the plots?

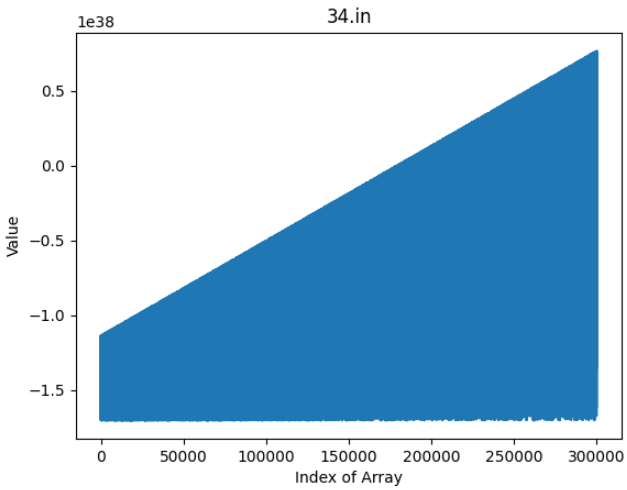
因為最容易計算的為 Wall time、communication time(簡稱 Comm time)和 I/O time，因此會把總時間扣掉 Comm time 和 I/O time，剩下的就是 Computing time。

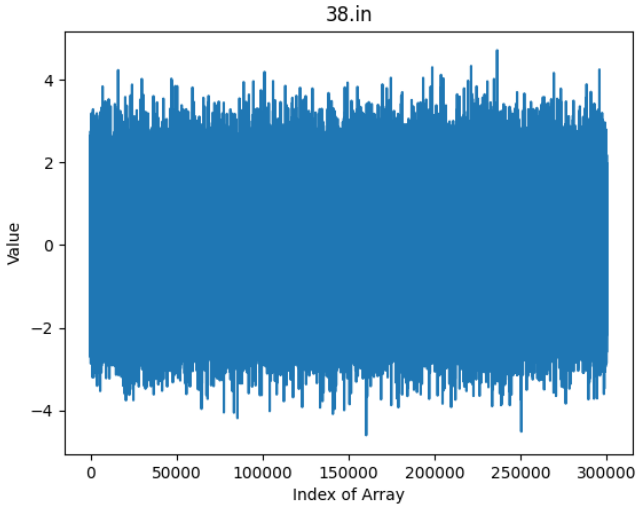
(c). Testcase Selection

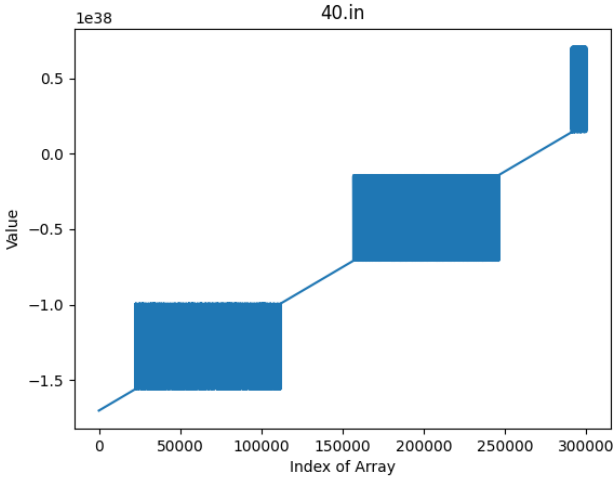
一開始挑選了 34, 37, 38, 39 40 等五個 testcase 來做挑選，最終的選擇會是以 testcase **38**，以下主要拿 34, 38, 40 的圖來做比較，而 37 和 39 的原因也會在章節的末段講解。

首先要 size 夠大，皆超過 5 億個數字，因此選了這 5 個 testcase，這樣之後在做實驗時，可以較容易去顯示 scaling 的差距。接著我會依照兩個標準，1. 數字的排序，2. 數字的分佈作為選擇的標準。

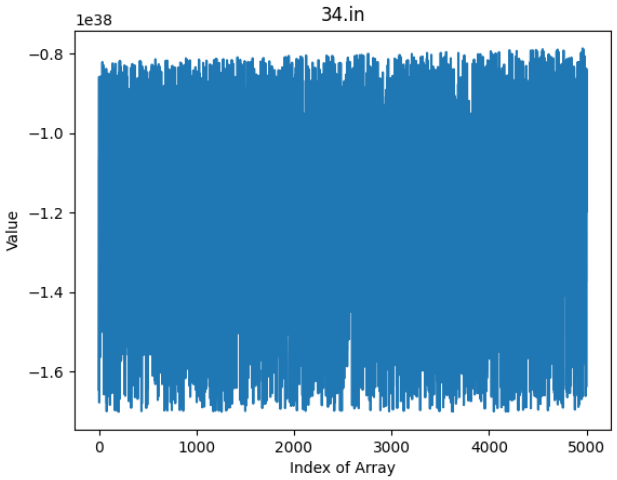
我從每個 testcase 中抽樣出了 300,000 個數，平均分佈在 0 ~ 300,000,000 中間，然後把這 30 萬個數依照 index 的大小畫在折線圖上(下圖)。從 34、40 可以看出，這些 data 都有向上排序的趨勢，而 38 的則是在-4~4 之間震盪。

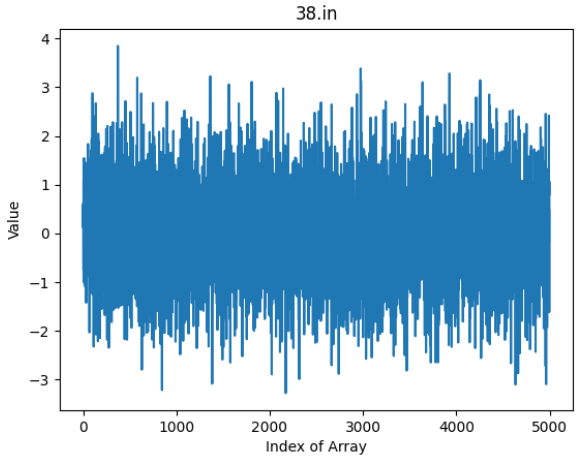


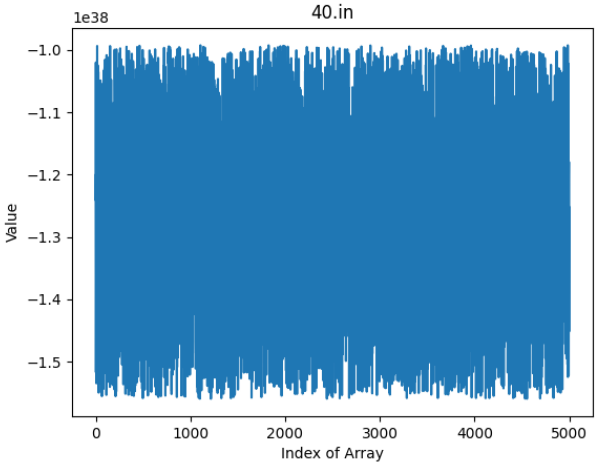




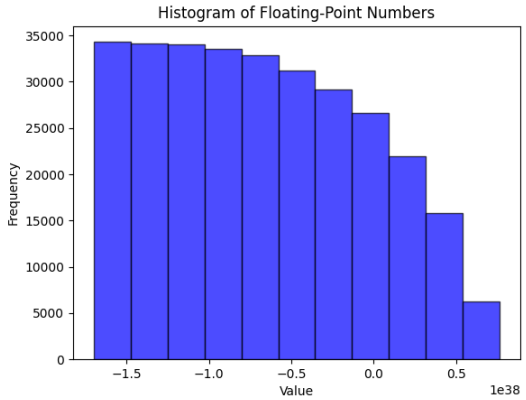
再更進一步細看下左圖，34 雖然看起來是已經排序過，但從挑選其中連續的五千個數字中，他是有震盪的狀況。40 如果把它上面長得像是正方形的區間取出成下圖，也是震盪的。38 取五千個數(下右圖)，跟原本的震盪差不多。

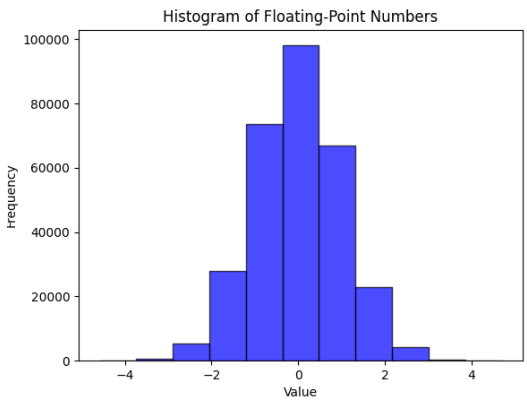


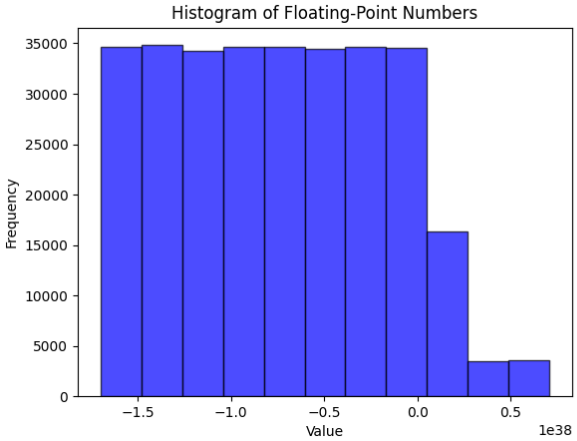




接著我們從 histogram 看數字的分佈。下圖從左上、右上、下是 34, 38, 40。可以發現 38 最符合 normal distribution。







結論：

剩下兩個 testcase 37, 39，37 的分佈類似 40，而 39 的資料分佈不像 38 一樣是常態分佈。我認為大量的資料時，通常會遵守常態分佈，不會有一個趨勢向上的排序，也不會有完全排序好的狀態。我們的程式應該是要能解決最常發生的情況而非特例的 best case 或是 worst case，因此選擇 38 做為之後實驗的資料集。

ii. Performance Measurement

- Use a profiler (like mpiP or IPM) for performance analysis

  本次實驗的時間測量會以 IPM 和 MPI_Wtime()作為主要衡量 performance 和時間的方式。

- Provide basic metrics like execution time, communication time, IO time, etc.

  Communication time 會用下述的程式碼包住 MPI_Sendrecv 和 MPI_Allreduce 進行加總來求得總時間。I/O time 也會運用相同方式包住 MPI_File_open、MPI_File_read_at 等 function 來計算總時間。而整個程式的總時間是從 MPI_Init 開始計算直到與 output 結果的最後一行程式碼；再扣掉上述的 Comm time 和 I/O time 就是 execution time。

```
start_comm_time = MPI_Wtime();
MPI_Sendrecv(partition, individual_load, MPI_FLOAT, rank+1,
sendrecv_ttl_time += (MPI_Wtime() - start_comm_time);
```

  由於使用 hw1-judge -i 來做 unit test 時的 I/O 會比 srun 跑 MPI_Wtime()的時間長度有很大的落差，導致 unit test 的 execution time 會比 srun 跑出來的 execution time 少很多。因為 unit test 無法針對不同的 node 和 process 做調整，所以在最後的實驗時，會以 srun 的結果為主。

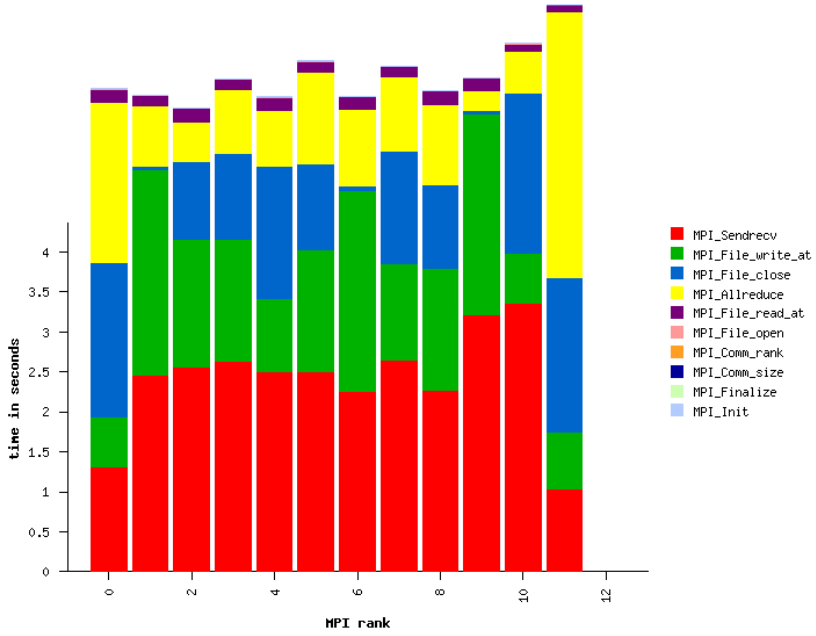
- Load Balance

  我先以 hw1-judge 上對 38.txt 做 unit test。由 IPM 的圖可以發現每個 process 的總時間並沒有差異到很多，但每個 process 在不同 function 上所花的 MPI time比例並不相同，以 rank

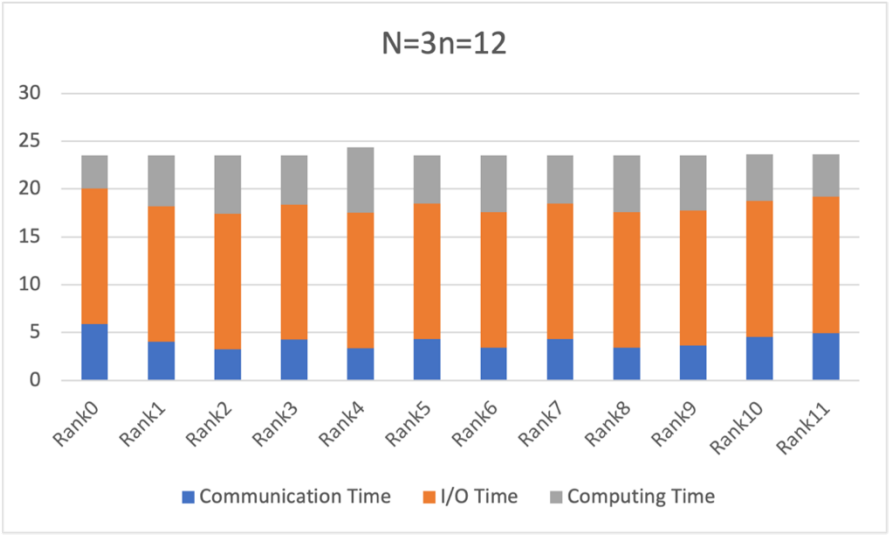
0 和 11 來說，他們花最多的是黃色的 MPI_Allreduce，而其他 process 則是花比較多的時間在紅色的 MPI_Sendrecv。這和我的演算法有關，rank 0 和 rank11 本來就會比較少參與 merge，因此不會一直使用 MPI_Sendrecv；而當他們快速做完自己的工作後，卻要在做 MPI_Allreduce 等待其他 process 做完 merge 然後才做 MPI_Allreduce，如此一來大大增加了這方面的比例。

第二可以理解到 I/O 主要是藍色和綠色，雖然每個 process 看起來不同，但是如果把藍色的長度和綠色相加，就會發現是非常接近的，因此可以認為在 I/O 方面的 Load Balance 是平均的。

下圖表格和三色堆疊圖則是 srun 跑出來的 I/O time，從中可以發現 I/O 的時間非常接近，皆在 14.15 秒左右，故可以得出 I/O 的 Load Balance 是平均的結論。



N=3n=12	Rank0	Rank1	Rank2	Rank3	Rank4	Rank5	Rank6	Rank7	Rank8	Rank9	Rank10	Rank11
I/O Time	14.1537	14.1542	14.1528	14.1545	14.1539	14.1542	14.1536	14.1545	14.1531	14.1524	14.2645	14.2665



- Identify Performance Bottlenecks

  從 IPM 的圖可以看出 Communication 中，除了 MPI_Sendrecv 外，MPI_Allreduce 佔了不少時間，我認為 MPI_Allreduce 是屬於 Broadcast 的行為，例如 rank10 會去等待 Rank2，這樣會浪費時間，因此我決定從這方面進行 Optimization。

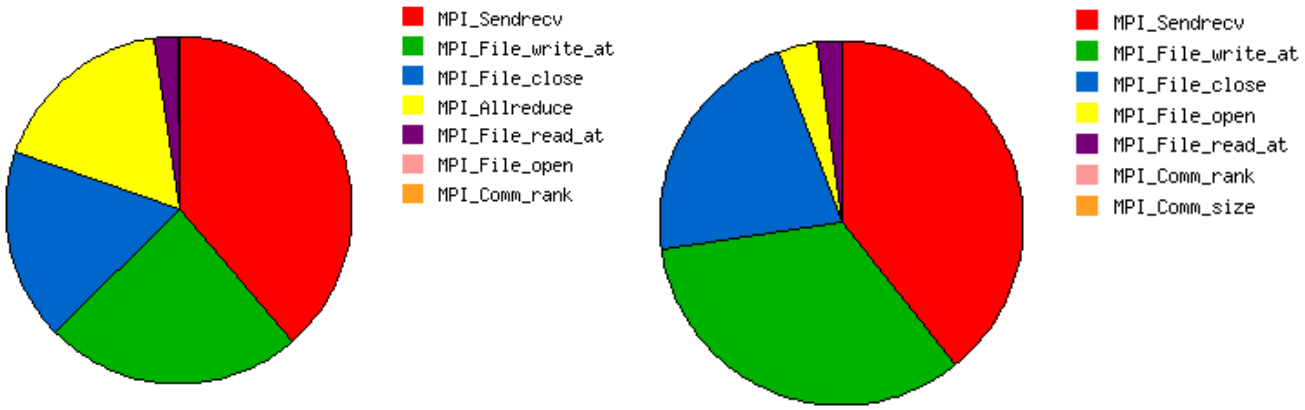
  此外，我認為 I/O 的時間無法省下來，故只能靠 Load Balance 的方式讓每個 process 能寫入相同數量的 output 為解決方法，因此增加 process 可以應該減少這方面的時間。

- Optimzation Strategies

  1. Optimzation 的方法是讓相鄰的 process 多進行一次 MPI_Sendrecv()，內容是確定左邊 array 最大 element 要小於右邊 array 的最小 element，如果為真，則不需要進行 MPI_Sendrecv()，如此一來，若發生所有 element 已經排序完成時，每個相鄰的 process 都不會進行下一步的 merge，可以省下 MPI_Sendrecv() local array 的時間。以傳送 1 個數字而換取不用傳送整個 array。
  2. 而且把 Other Efforts(1)這個條件捨去，改成每個 phase 都進行 iteration，利用 iteration 的 variable 決定是 Even Phase 或 Odd Phase。如此一來有機會省掉最後一次的 Odd Phase(要看總共有多少個 process，如果是奇數的話則沒省到)。

     Comparison:

     以下圖而言 MPI_Allreduce 被右圖的 MPI_Sendrecv 取代了。



從 IPM 表格可以看到使用 MPI_Allreduce 的演算法時，因為每個 phase 都要 merge，所以 MPI_Sendrecv 要使用的 Buffer Size 非常大且要執行 154 次。而使用 Optimized 演算法時，由於已經用 Buffer Size = 4 的 MPI_Sendrecv 確認過需不要 Swap，因此大 Buffer Size 的 MPI_Sendrecv 可以被省下來。很可惜由於 static IPM 只能對 node = 3, process = 12 的情況下做 profile，不然如果改變 scale 的話可以進一步看到更多成果。

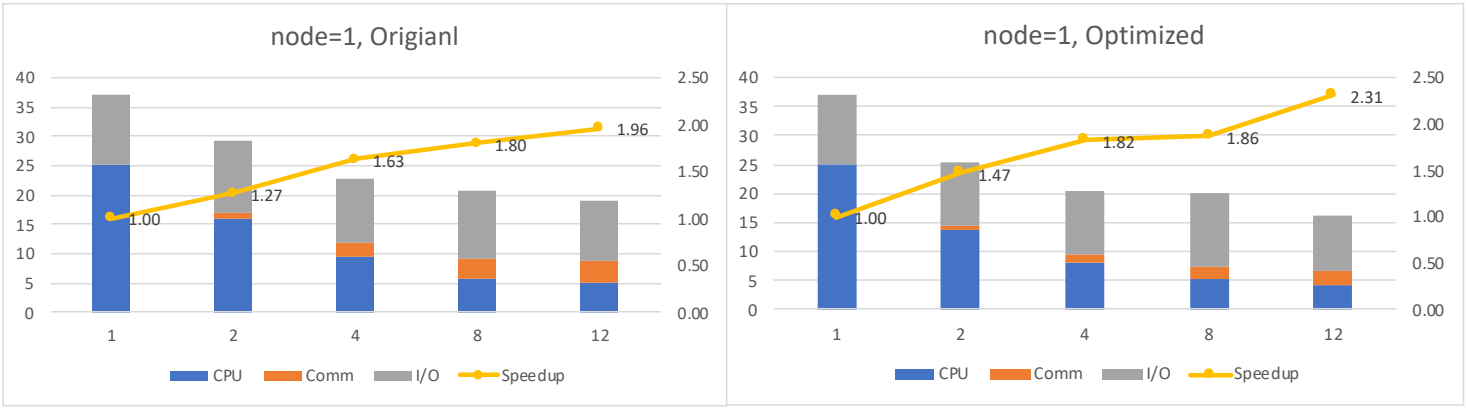
Original Algorithm	Buffer Size	Ncalls	Total Time
MPI_Sendrecv	167772160	154	28.775
MPI_Allreduce	4	84	12.854

Optimized Algorithm	Buffer Size	Ncalls	Total Time
MPI_Sendrecv	167772160	136	21.165
MPI_Sendrecv	4	144	10.688

iii. Plots: Speedup Factor & Profile

- **Single-Node Scalability**

下圖為不同演算法在 node = 1, process 不同時的 runtime 分佈和 Speedup。可以看到優化的版本有更佳的 Speedup，Comm time 稍微少了一點。Speedup 如預期，因為有 sequential code、communication 或是 I/O 的關係沒有和 process 增加等比例降低。意外的是，I/O time 並沒有顯著地因為 process 數增加而降低。

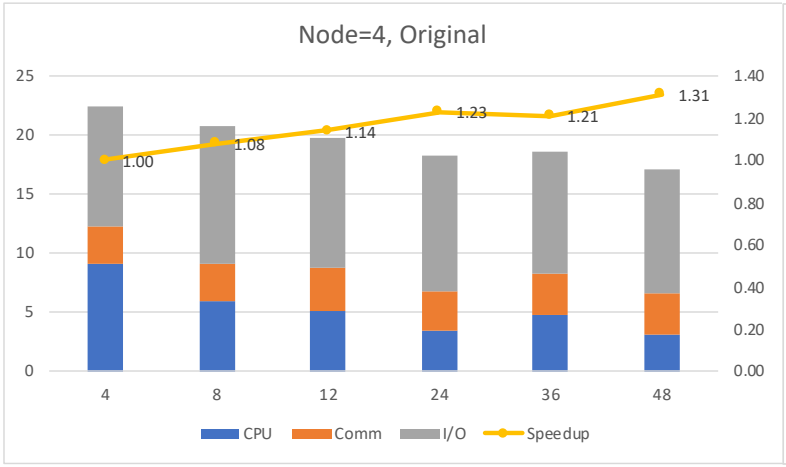


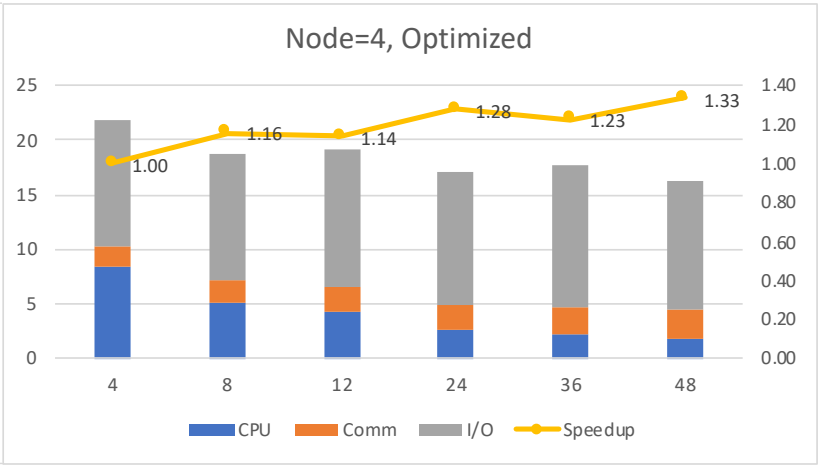
- **Multi-Node Scalability**

下圖和表格為當 node=4 時，不同演算法下的不同 process 數下的 runtime 分佈和 speedup。隨著 process 增加，Comm time 會增加，也可以發現 Comm time 並不定隨著 process 增加而必然繼續往上升，而是卡在一個數值上不去。Speedup、I/O time 沒有等比例的增高和降低。

process=	4	8	12	24	36	48
CPU	9.12	5.99	5.13	3.40	4.83	3.06
Comm	3.16	3.11	3.66	3.36	3.49	3.49
I/O	10.17	11.70	10.91	11.47	10.20	10.55
Speedup	1.00	1.08	1.14	1.23	1.21	1.31
Total	22.45	20.80	19.70	18.23	18.52	17.10

process=	4	8	12	24	36	48
CPU	8.50	5.13	4.34	2.59	2.15	1.85
Comm	1.79	2.00	2.29	2.30	2.54	2.57
I/O	11.50	11.68	12.54	12.13	13.07	11.94
Speedup	1.00	1.16	1.14	1.28	1.23	1.33
Total	21.79	18.81	19.17	17.02	17.76	16.35



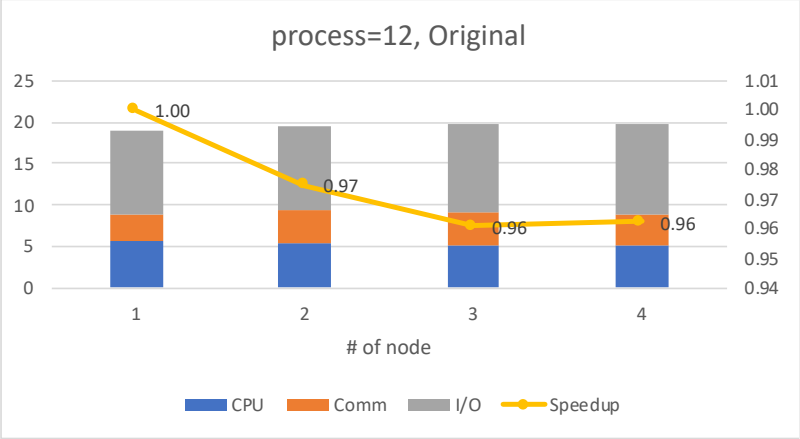


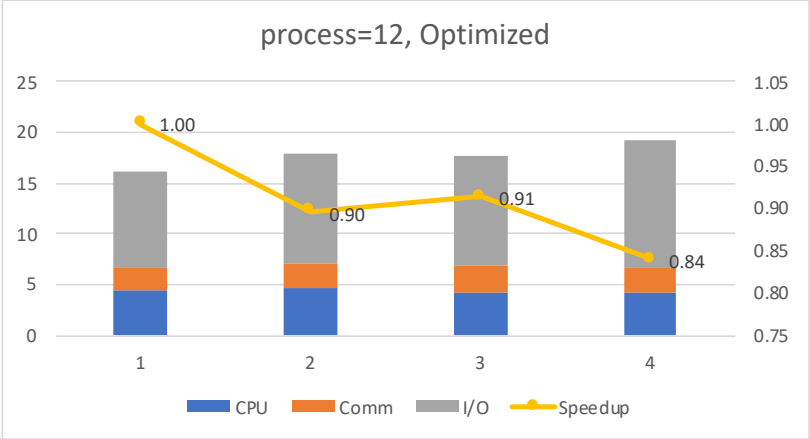
- **Different Scale in Node**

  下圖為不同演算法在 node 不同時，process = 12 的 runtime 分佈和 Speedup。可以發現當 Node 數增加時，Comm time 也會增加。

n=12,Node=	1	2	3	4
CPU	5.55	5.53	5.20	5.13
Comm	3.21	3.85	3.93	3.66
I/O	10.20	10.07	10.60	10.91
Speedup	1.00	0.97	0.96	0.96
Total	18.96	19.45	19.73	19.70

n=12,Node=	1	2	3	4
CPU	4.41	4.59	4.29	4.34
Comm	2.25	2.45	2.65	2.29
I/O	9.45	10.93	10.68	12.54
Speedup	1.00	0.90	0.91	0.84
Total	16.11	17.98	17.62	19.17





iv. Discussion (Must base on the results in your plots)

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

  Network 和 I/O 會是 bottleneck。從 Multi-Node Scalability 和 Different Scale in Node 的表格可以看到當 process 數增加時，Comm time 會增加，當 4 個 process 增加到 48 個時，Comm time 增加 10.4%和 43.5%。如果想要增加平行度使得排序時間更短，我們勢必要增加更多 node 和 process，但此時 Comm time 也增加，這樣會讓我們的加速效果減少。

  此外 I/O 幾乎是定值。但在我們的程式碼中，每個 process 都會從他們指定的 offset 開始寫起，因此不會有 data dependency 的問題，是屬於 Embarrassingly Computations，可是在 performance 卻沒有什麼 Speedup，因此我認為 I/O 是 bottleneck。

  改善的方法可從硬體方面改善，如果讓一個 node 擁有更多 core，這樣可以少用到網路，使得 Comm time 不要增加太多。或是從 Algo 層面來減輕 Comm 的負擔，我覺得 Odd-even sort 有個問題是第一個 process 和最後一個 process 常常不會參與 merge 和 MPI_Sendrecv，使得他們的 workload 是稍微比其他少的，他們也常常在等待其他人做好工作，所以卡在下一個 iteration 的 MPI_Sendrecv。如果能想到一個方法讓他們能更平均的參與 merge，或許可以發揮更大的平行效益。

- 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.

  設 data size 為 n，process 數為 p，w 為 MPI_Allreduce 所多花的時間倍數，x 為 MPI_Sendrecv 所花的時間倍數，y 為寫入 I/O 的時間倍數，z 是讀取 I/O 所多花的時間倍數。

- 原本的演算法時間複雜度：

  一開始花 O(p)的時間計算其他 process 和自己的 workload；花 z * O(n/p)讀取 local element。接著做一次 local sort O((n/p) * (log n/p))；之後進入 p 次 loop，每次 merge 花 O(n/p)，再加上 MPI_Sendrecv n/p 個 element 的時間，還要進行一次的 MPI_Allreduce O(pw)等所有 process 都做好，共為 p * (x * O(n/p) + O(w))；最後還有寫入的時間 y * O(n/p)。所以總共的時間複雜度為 (px + y + z + (log n/p)) O(n/p) + O(p) + (p/2)*O(w)。

- 優化的演算法時間複雜度：

  在 loop 之前的 code 跟上面一樣。進入 p 次 loop，每次 merge 花 O(n/p)，再加上 MPI_Sendrecv 1 + n/p 個 element 的時間，依舊為 p * (x * O(n/p))；最後還有寫入的時間 y * O(n/p)。所以總共的時間複雜度為 (px + y + z + (log n/p)) O(n/p) + O(p)。

從時間複雜度分析可以發現如果 px + y + z 和 w 決定了我們 speedup 的幅度。不需要任何平行程式的話，我們需要花 O(nlogn)的時間排序完畢。而如果不需要 I/O、Comm time 的話，可以達到理想的 Speedup p。

在改善了 MPI_Allreduce 後我覺得程式有顯著提升 Scalability。從 Single-Node Scalability 的圖來看可以發現 Speedup 明顯更高。因為 MPI_Sendrecv 是 point-to-point 的 Communication，而 MPI_Allreduce 是 Collective，如果我們之後繼續增加 process 時來獲得 Speedup 時，程式跑到 MPI_Allreduce 要等待的 process 數量就會增加，也就是時間 (p/2)*O(w)會增加。而用優化的演算法只要網路不要 congested，可以不用擔心時間複雜度分析會有一項是不減反增的。

3. Experiences / Conclusion

- Conclusion

  在平行程式中，演算法的設計會跟 single process 的寫法稍有不同。還要考量到 Comm 和 I/O 這兩個更花時間，且不一定能真的做到平行化的因子。一開始我是使用比較直觀的 MPI_Allreduce 來做檢查，後來才突然想到可以用其他方法來代替，也算是慢慢更懂得怎麼利用 MPI 的工具來進行優化。

- Difficulties

  在平行程式中 Debug 變得比較困難，雖然 vscode 已經是非常好的工具，但在寫平行程式時，無法像過去一樣可以開 debug 快速看到自己的變數發生了什麼改變，只能不停的 print 出來。

  要使用 hw1-floats 去檢視資料一開始也花費了不少時間才讓我熟悉整個系統，當然到後面優化程式碼時這件事就沒那麼困擾了。

- Feedback

  我覺得有些人占用系統資源的時間太多了，如實驗所示，當 process 數為 1 時，都只要花不到 40 秒就可以跑完 5 億筆資料，但 timed out 的限制卻是 5 或 10 分鐘。有些人就讓他的程式卡在系統也不主動去 cancel，這樣會讓其他使用者損失不少時間。