Parallel Programming Assignment #1

姓名：胡展維
系所：電機所
學號：106061527

A. Implementation

Basic Part

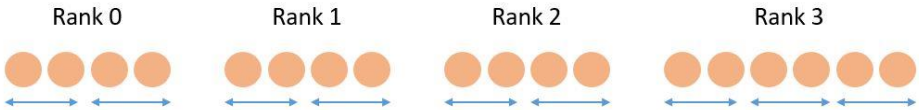
此次作業利用 MPI 來實作 Odd-Even sort。首先，有多個 processor 可用 (假設為 P 個)，我先將輸入的 N 個 elements 並分配至各個 processor，也就代表每個 processor 理論上要有 N/P 個 elements。但是有兩種情況會發生：

- 當 N<P 時，可能會有些許 processor 並未被分配到 element (代表不需要這些多餘的 processor)。
- 當 N/P 無法整除時，其中一個 processor 的 element 數量會與其他 processor 不同。

除此之外，也要考慮當 N/P 為奇偶的情況。以下考慮各個不同情況：(這裡假設有四個 processors, 也就是 P=4)

在 Even Phase 時：

(1) 當 N/P 為偶數，rank P-1 的 process 中 element 也為偶數：

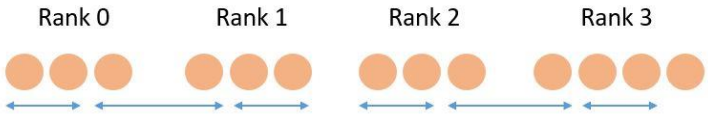


Processor 之間不需溝通，僅需在 local 做 even phase sort。

(2) 當 N/P 為偶數，rank P-1 的 process 中 element 為奇數：

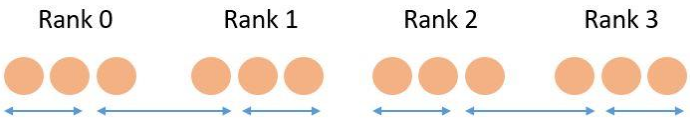
如同上述，processor 之間亦不需溝通，僅做 even phase sort。

(3) 當 N/P 為奇數，rank P-1 的 process 中 element 為偶數：



從例圖看出當 rank 為偶數時，其 tail 會與相鄰的 processor 的 head 比較；除此之外，當 rank 為偶數時其 local even sort 是從第一個 index 開始，而 rank 為奇數則從第二個 index 開始。

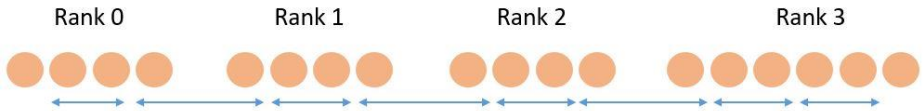
(4) 當 N/P 為奇數，rank P-1 的 process 中 element 也為奇數：



情況與(3)相同，不贅述。

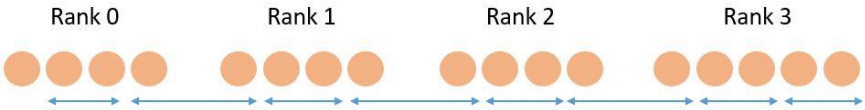
在 Odd Phase 時：

(1) 當 N/P 為偶數，rank P-1 的 process 中 element 也為偶數：



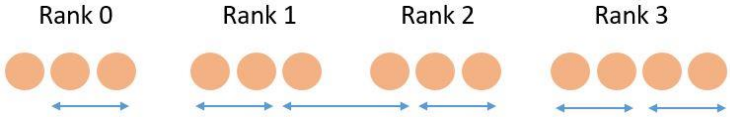
當 rank 不為 0 或 P-1 時，其頭尾都要與相鄰的 processor 做比較，且所有 rank 的 local odd sort 都從第二個 index 開始。

(2) 當 N/P 為偶數，rank P-1 的 process 中 element 為奇數：



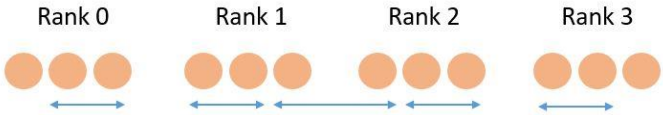
情況如上(1)，不贅述。

(3) 當 N/P 為奇數，rank P-1 的 process 中 element 為偶數：



當 rank 為偶數時(不為 0)，其 head 會與前個 partner 的 tail 比較，且 local odd sort 從第二個 index 開始。而奇數 rank 的 local odd sort 則從第一個 index 開始。

(4) 當 N/P 為奇數，rank P-1 的 process 中 element 也為奇數：



與(3)類似，不贅述。

對以上情況做整理以後即可找到規律如講義。在我的程式中，若第 i 個 processor 的 tail 需要與第 i+1 個 processor 的 head 做比較，則第 i 個 processor 會先 send tail 的值給第 i+1 個 processor，當第 i+1 個 processor 收到值以後會與自己的 head 做比較，若較大則 swap 值並傳送較小的值回到第 i 個 processor，第 i 個 processor 收到後會觀察收到的值與傳送的值是否相同，若不同，則 swap 收到的值與自己的 tail (概念類似 Fig.1)。在此部份我設計了一個簡易的 tag 用來分辨 message 的傳輸，哪個 processor ”主動”傳資料就用哪個 processor 的 rank 做為 tag。例如: 第 i 個 processor 先傳自己的 tail value 給第 i+1 個 processor，所以 tag 為 i，而第 i+1 回傳資料是回傳到 i，其 tag 亦為 i。如此即可保證資料不會傳輸錯誤。

而對於 N<P 的情況，則需要刪除一些無用的 processor (因為他們不會被分到 element)。我利用 MPI 中 communication group 的概念，重新創建一個 group，並讓 processor 在這個 group 裡可以重新做溝通，而那些不會用到的 processor 其 MPI_COMM 參數會變為 MPI_COMM_NULL，這樣即可將這些 processor finalize。

藉由以上觀念，即可 handle 任意 N 值與 processor 數目。

- P_1 send A to P_2, this compares B with A and sends to P_1 the $min(A, B)$.

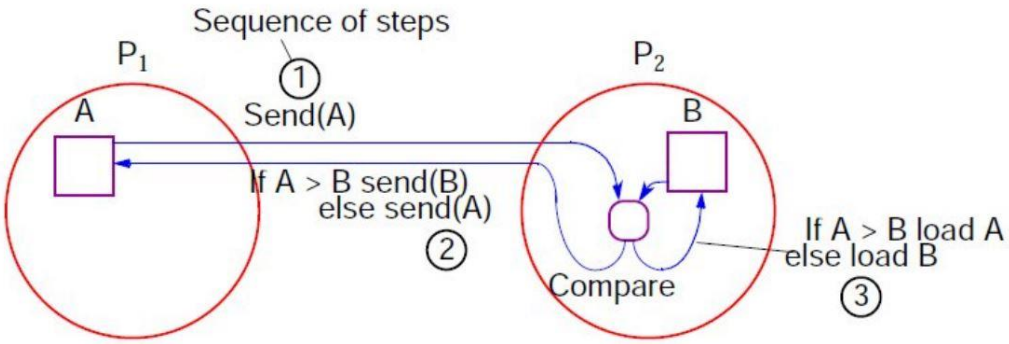
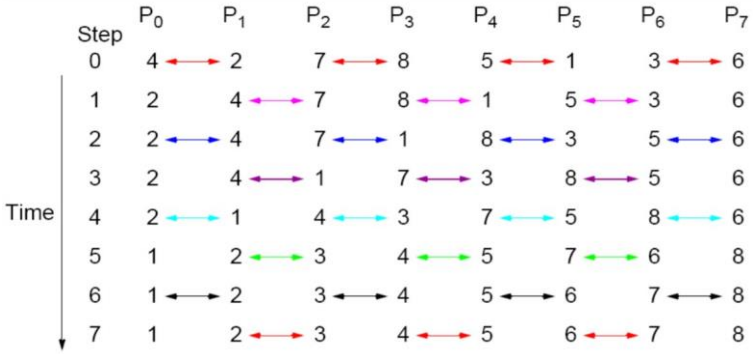


Fig.1

Advanced Part

以下幾點簡單敘述流程:

1. 利用 C library 提供的 qsort()先對每個 processor 做 local sort。
2. 在完成 local sort 以後，做 processor level 的 odd-even sort。如以下示意圖:



3. 接下來，我會先傳送各 processor 的 local data 到鄰近的 partner。以在 odd phase 的 processor rank=1 與 2 來做說明: P1 會傳送分配到的 data 到 P2，而 P2 也會傳送至 P1，再來就是個別 processor 要自行計算要留下哪些資料，在 P1 中會將自己的資料與接收的資料做 merge sort，只做到 P1 所需的 lower numbers 即停止，而 P2 則做到所需之 higher numbers 就停止 (參考 Fig.2)。

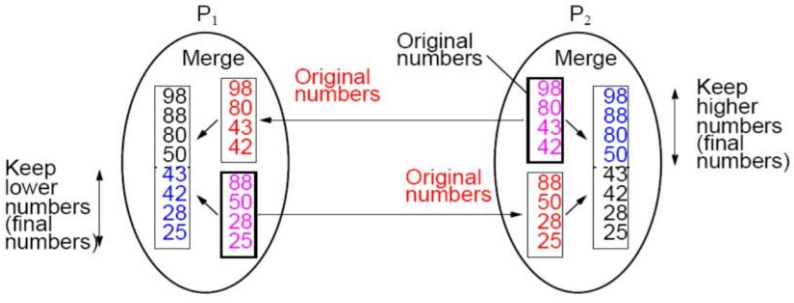


Fig.2

4. 重複 odd 與 even phase 直到沒有 exchange 發生。

B. Experiments and Analysis

System Spec.

使用課程提供之 server。

Performance metrics

我使用 glibc 的 clock_gettime()做為評量 I/O, communication time 與 computation time 的依據。即便已經有諸多更簡易的 function 例如: MPI_Wtime(), clock()等等可以使用，但其精確度仍有可能不足。使用此 function 的好處是其可以精確到 nano second，在比較微小差異時可以精確表達效能。
另外，由於同時執行多個 processor，必須確保每個 processor 都完成當下的任務才是真正的完成。我利用 MPI 提供的 MPI_Barrier()來達成，每個 processor 都必須執行到 MPI_Barrier()這行程式時才會記 end time，否則會等待其他 processor 完成。
使用 clock_gettime()的範例程式碼如下:

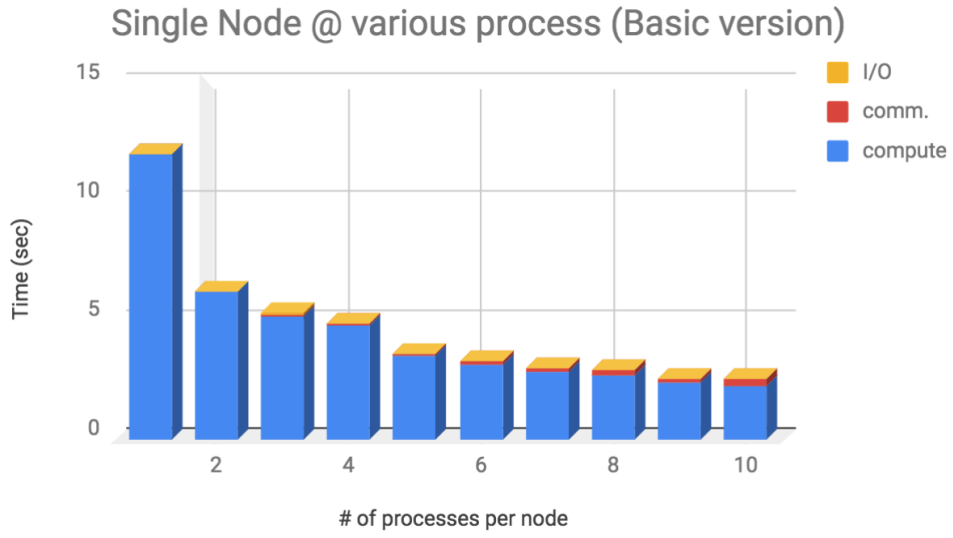
```
#include <time.h>

struct timespec start, end;
clock_gettime(CLOCK_REALTIME, &start);
// Measurement you want
clock_gettime(CLOCK_REALTIME, &end);
printf("Time elapsed: %ld", end.tv_nsec-start.tv_nsec);
```

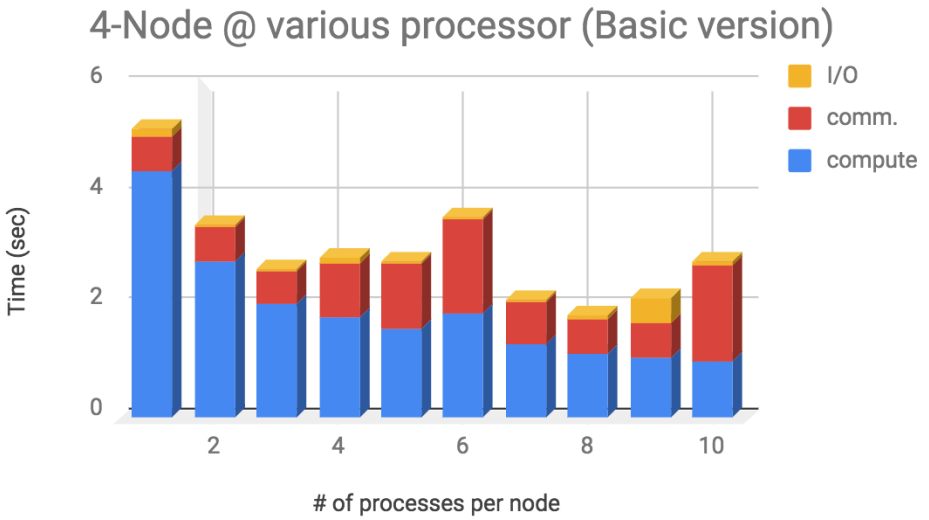
計算出來的時間會 log 到一個 performance log 中，用以判斷在某設定底下的效能資料。

Strong Scalability experiments and Discussion

1. **Basic Version (測資 N = 10^5)**
   表格資料僅以部分表示趨勢！詳細資料可以參考：
   https://docs.google.com/spreadsheets/d/1cr4yA1MMtFlbzE5aQQqB_zlDTFQO4ldyvvNggV9BsZo/edit?usp=sharing



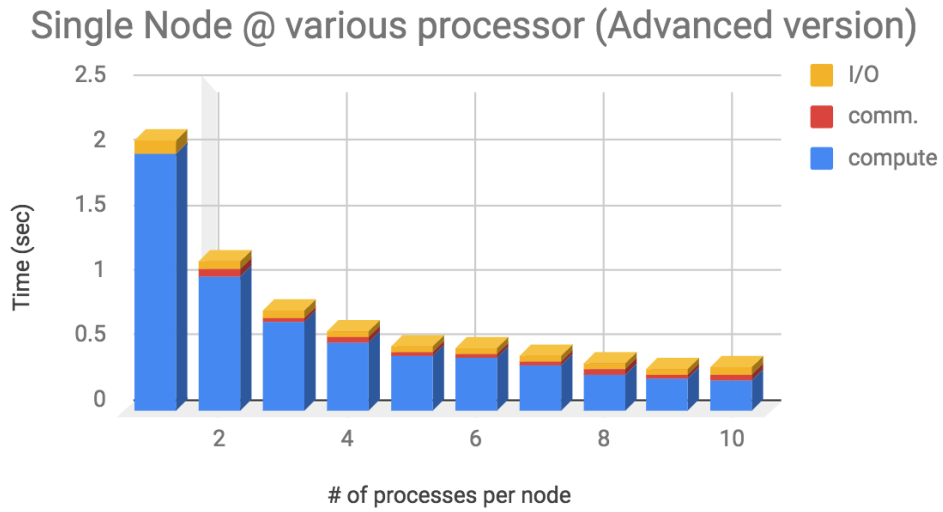
Process per node	1	3	5	7	9
I/O time (sec)	0.004738	0.018255	0.004639	0.00306	0.004454
Computing (sec)	12.028212	5.179465	3.481223	2.844971	2.359502
comm. (sec)	0	0.082247	0.118049	0.153019	0.177217



Process per node	1	3	5	7	9
I/O time (sec)	0.146007	0.044261	0.054066	0.021347	0.460435
Computing (sec)	4.457311	2.074093	1.598871	1.333917	1.087797
comm. (sec)	0.615978	0.559571	1.179417	0.767986	0.621967

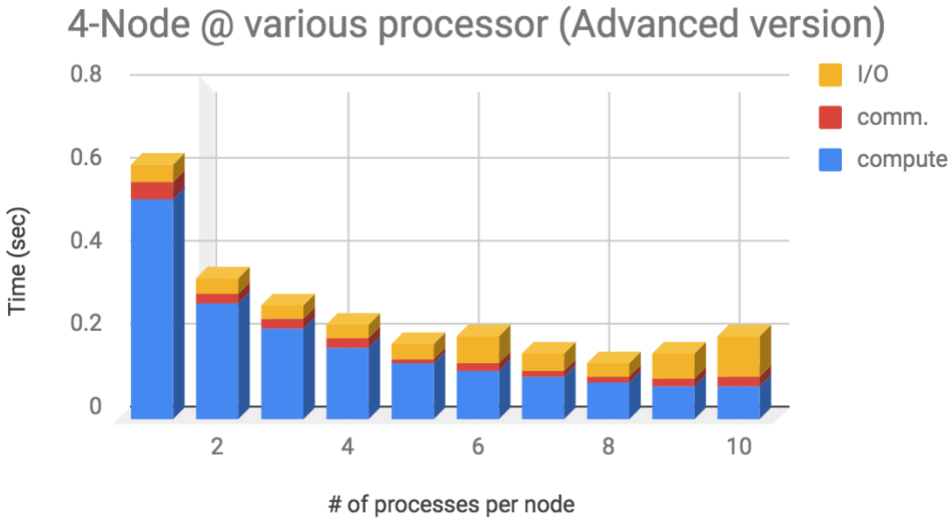
由上述實驗可知， 當 process per node 數量越多時，node 數增多會讓 communication time 佔總執行時間的比例越高。這蠻符合我的期待，因為跨 node 的通訊時間肯定比單一 node 要久，甚至可能造成 total execution time 有一半都是在執行通訊。

2. **Advanced Version (測資 N = 10^7)**



Process per node	1	3	5	7	9
I/O time (sec)	0.095711	0.049987	0.050658	0.045125	0.044736

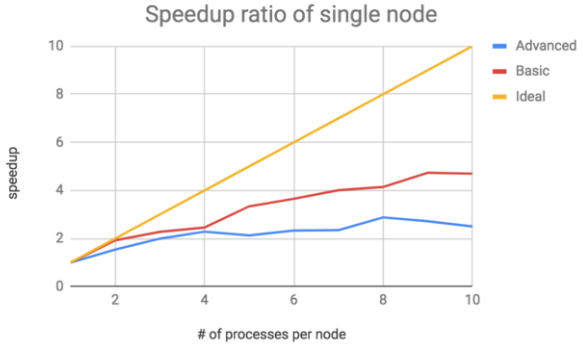
Computing (sec)	1.985947	0.692641	0.421945	0.349601	0.253577
comm. (sec)	0	0.031503	0.02914	0.027001	0.026027

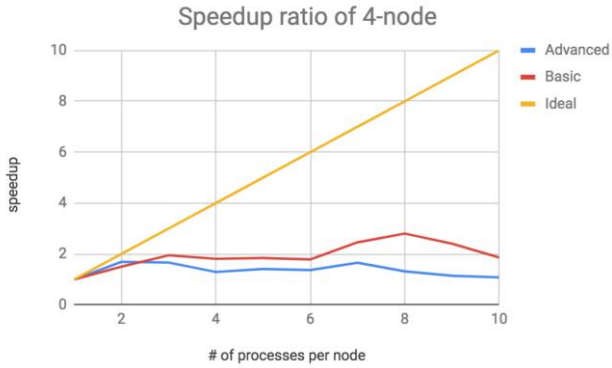


Process per node	1	3	5	7	9
I/O time (sec)	0.041786	0.032594	0.037509	0.041821	0.06042
Computing (sec)	0.529871	0.216296	0.132443	0.102177	0.079399
comm. (sec)	0.041688	0.023434	0.010609	0.015112	0.015501

從以上實驗數據來看，觀察出對 advanced version 來說，node 值上升會使 I/O 佔整體執行時間的比例上升，推測可能原因是真正 computing 時間越來越小，而因為在 multi-node 中每個 node 都要自己完成 I/O，造成花費比較多時間，所以即使 computing speed 可以倍數增加，礙於 I/O delay 的因素，使得增加更多 node 或 process 的效果大大減低。

3. Basic vs. Advanced speed up comparison





從以上比較發現 advanced version 的 speedup ratio 沒有 basic version 來得顯著，我分成兩種情況討論：

Single node:

在 single node 中，我的 basic version speedup 比較顯著，我推測的原因是當只有一個 processor 時，basic version 會 sequentially 做 odd-even sort，使得

時間複雜度為$O(N^2)$。當增加 process 數量時，每個 process 只會被分到 N / # of process 個 element，大大減少單一 process 執行的時間，但由於增加 process 會額外多 communication time，所以也不能無限制增加 processor。而在 advanced version 中，若只有單一 processor，會直接利用 quick sort 做 sorting，時間複雜度為$O(NlogN)$。當增加 processor 時，在每個 time step 時 merge 兩個 sorted list 時會增加 $O(N)$，所以 advanced version 的多 process 運算複雜度為$O(NlogN + N)$，這還是未考慮 communication time 的情況。最終當 N 越來越大時，運算複雜度會趨近$O(NlogN)$，就比較難 speed up。

Multi-node:
對 basic version 來說，從繪出的長條圖表中可以看出 communication time 佔了很大一部分。即使 computing time 線性下降，communication time 與 I/O time 仍然佔了快一半，導致無法大幅度增加 performance。在 advanced version 也是相同問題，不過由於 advanced version 所需 communication time 不多 (因為我的作法是直接將 local sorted array 全部傳給 partner process 之後就不等待回傳，不需像 basic version 要等待 partner process 回傳)，所以大部分時間卡在 I/O 的部分，最後導致 speedup 曲線無法上升甚至幾乎持平。

C. Compare

Advanced vs. Basic

將測資訂在 N=10^5，比較 single node 中不同 process 個數對二者 performance 的比較：

		Basic		
processor per node	node num	I/O (sec)	computing (sec)	comm. (sec)
1	1	0.004738	12.028212	0
2	1	0.004621	6.186895	0.058096
3	1	0.018255	5.179465	0.082247
4	1	0.006599	4.796263	0.092078
5	1	0.004639	3.481223	0.118049
6	1	0.00675	3.147912	0.138883
7	1	0.00306	2.844971	0.153019
8	1	0.008836	2.662294	0.231164
9	1	0.004454	2.359502	0.177217
10	1	0.005667	2.229095	0.326629

		Advanced		
processor per node	node num	I/O (sec)	computing (sec)	comm. (sec)
1	1	0.004127	0.014464	0
2	1	0.004306	0.00755	0.000232
3	1	0.004066	0.005034	0.000215
4	1	0.004083	0.003809	0.000244
5	1	0.005362	0.003133	0.000227
6	1	0.005124	0.002586	0.00026
7	1	0.00538	0.002247	0.000291
8	1	0.004078	0.002046	0.000333
9	1	0.0047	0.001792	0.000347
10	1	0.005511	0.001687	0.000236

觀察出 I/O time 差異不大，在 computing time 的部分 advanced version 比 basic version 快上幾乎 1000 倍！原因是 basic version 的每個 process 都是做 sequential shifting，而 advanced version 每個 process 都是做 quick sort 再 merge，加上 advanced version 所需的 communication time 很少，僅需做(# of process – 1) 次。

Bottleneck

綜合以上結果，對於 basic version 來說，主要 bottleneck 為 communication time，I/O time 相對影響不大。原因是在每個時間點時，每個 processor 必須傳送自己的 head (或 tail) 給目標 process，並等待回覆，當資料量大的時候，達到 terminal state 時所需 step 呈線性上升，造成 communication time 線性增加。以下做一個小實驗來驗證 (以 single node 與 10 個 processor 為基準)：

N (# of element)	communication time (sec)
1000	0.003418
10000	0.030789
100000	0.2611
1000000	3.893713

由此可知，主要 bottleneck 為 communication time，會造成當 process 增加時不能很理想的增加 performance。減少這 communication time 的方法我認為是利用 non-blocking communication 來做，讓 computing time 與 communication time 有 overlap，使整體執行時間降低。在我的程式中我使用較為安全的 blocking communication 確保資料傳輸不會錯誤，可能是因為這個原因導致我的 basic version 會卡在 communication time。

而對於 advanced version 來說，由於通訊時間與 processor 數量較有關 (影響到需要多少 steps 完成 sorting)，所以不太會是 bottleneck，以下實驗驗證(以 single node 與 10 個 processor 為基準)：

N (# of element)	communication time (sec)
1000	0.00002
10000	0.00004
100000	0.000364
1000000	0.000148

隨著 N 越大，communication time 會稍微上升，但並非線性成長，我推測是因為傳送資料較多，需要較多時間。

對 advanced version 來說，我認為主要 bottleneck 為 I/O time。從上面的 experiment 來看，computing time 越來越小，但 I/O time 會約略維持在 0.04 秒，造成不管 process 數增加或 node 數增加都無法有太大 speedup。

Scalability

在 basic version 中，我的程式並未有太大 speedup，至多 speedup 約 5 倍左右。而在 advanced version 中，也未有太大 speedup 甚至持平。原因已經敘述在 Part-B (3)中。

D. Conclusion

此次作業者要利用簡易的 MPI 實作 sorting algorithm，利用資料可以分別作 sorting 再合起來的特性，使用多個 processor 可以加速這運算。

這次作業老實說我做得有點辛苦，因為是自己不是資工本科，對底層運算的理解還尚在幼兒階段，甚至還要回去複習一下 pointer 的用法之類。隨著時間推移，漸漸了解該怎麼實作這次作業，統整課程 slide 與自行上網 google 的資訊慢慢一行一行刻出 code，最後看到兩個 version 在 judge 時都 accept 頓時覺得滿滿成就感！這次作業我卡得比較久的部分並非 MPI 的使用，而是對於整個 pointer 的理解重新大翻新，例如：memory 怎麼 allocate 比較有效率，怎麼搞 function pointer 等等，讓我對 C 有比較深的理解！

很早以前就一直期望能修平程增進我對系統的理解，很多同學都說平程很硬，不斷勸退，但是為了求知，我還是點了下去，期望在後面的作業能有更多收穫！