# CS542200 Parallel Programming HW1 Report


- 學號：108022138
- 姓名：楊宗諺


## Implementation

### Data preprocessing

在進到主要的 sorting phase 之前，這邊做了以下幾件事情：

- 將 input array partition 平分給每個 process 來處理。假設 input items 的數量為 n，平行度為 m，且考慮比較 general 的 case（n 不整除 m），那麼每個 process 分到的 subpartition 大小為 `floor(n / m) + (rank < n % m ? 1 : 0)`。
- 透過 MPI I/O，將該 subpartition 的資料從 input file 中讀出，並存放在動態配置的陣列裡。
- 利用 `std::sort()` 先排序 subpartition 內部的 local elements。

### Odd-Even sorting phase

在原先 sequential version 的 Odd-Even Sort 中，每個 element 會需要不斷地跟它的 neighbors 比大小，如果大的在小的前面就做交換，直到整個 input array 被排序好為止。而在 parallel version 中，我們可以將每個 process 視為一個 sequential version 中的 element，在 sorting 的過程中讓每個 process 跟它的 neighbor processes 比大小做交換，一樣執行到所有 processes 都被排序好為止。

為了之後討論方便，這邊定義每個交換 pair 中 rank 較小的 process 為 receiver process，較大的為 sender process。

- How to make two processes sorted ?
    - 先比較兩個 processes 的大小關係，如果 receiver process 的最後一個 element 小於等於 sender process 的第一個 element，代表兩個 processes 的 local elements 排列起來會是 sorted，這種狀況就不用做任何的交換。反之，則需要進行交換。
    - 交換的過程中，sender process 透過 message passing 將其 local elements 傳給 receiver process。receiver process 將其 local elements 和 sender process 傳來的 elements 作排序，最後再將其中較大的 elements 回傳給 sender process，這樣即可確保交換後兩個 processes 會是 sorted。
    - 值得注意的是，由於兩個 processes 的 local elements 在交換之前分別已是排序好的狀態，所以在交換時，我們可以在線性時間內將兩個 processes 的 elements 合併排序好。
- Sorting termination check

  在每一次 odd phase 的結尾，我利用 rank 0 的 process 將其餘所有 processes 在最近一次的 even phase 以及 odd phase 裡有沒有做交換的資訊收集起來，並且判斷是否完成排序，最後將判斷的結果透過 `MPI_Bcast()` 廣播給所有 processes。

### After sorting

結束 parallel version 的 Odd-Even Sort 後，透過 MPI I/O 將每個 subpartition 的內容寫回 output file 中，並且釋放程式運行過程中動態配置的記憶體。

# Experiment & Analysis

## Methodology

這邊我採用的系統配置為課堂提供的 Cluster。在計算時間的部分，我是利用 MPI Library 提供的 `MPI_Wtime()` 去夾在要計算的區塊來得到時間差。

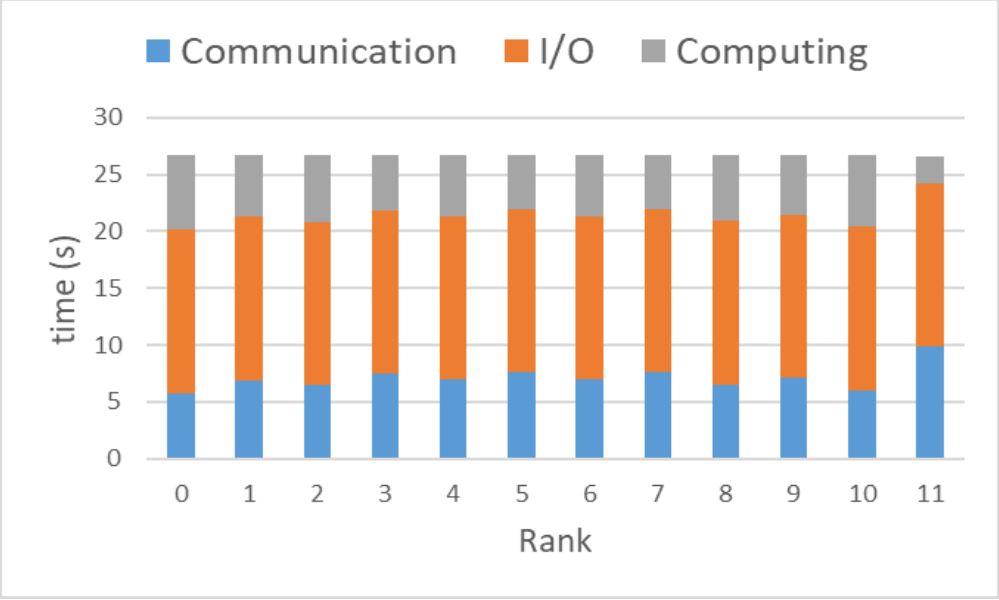
- 在計算 computing time 上，我是先將兩個 `MPI_Wtime()` 分別放在 `MPI_Init()` 和 `MPI_Finalize()` 前後得到 process 的總執行時間，最後再減去 I/O time 和 communication time 得到大概的 computing time。
- 在計算 I/O time 上，我將所有 MPI I/O 的操作前後都分別加上 `MPI_Wtime()`，最後再將所有時間加總起來得到 I/O time。
- 而計算 communication time 的方式也跟前面一樣，在所有 MPI Communication 的指令前後加上 `MPI_Wtime()` 去計算時間差，最後將所有時間加總起來得到 communication time。

然而，因為每個 process 所花費的 computing time、I/O time 以及 communication time 不盡相同，所以在收集數據上我會將所有 processes 的相關執行時間取平均來作效能分析，同時因為 cluster 並不是十分穩定，所以針對單筆實驗會進行多次測試並取最小執行時間的那次，嘗試降低環境的影響。

## Profile

這邊我使用的測資為 33.txt，input array 的大小為 536869888，使用的 processes 數量為 12 個，分布在一個 node 上，每個 process 所使用的 core 數為一個。

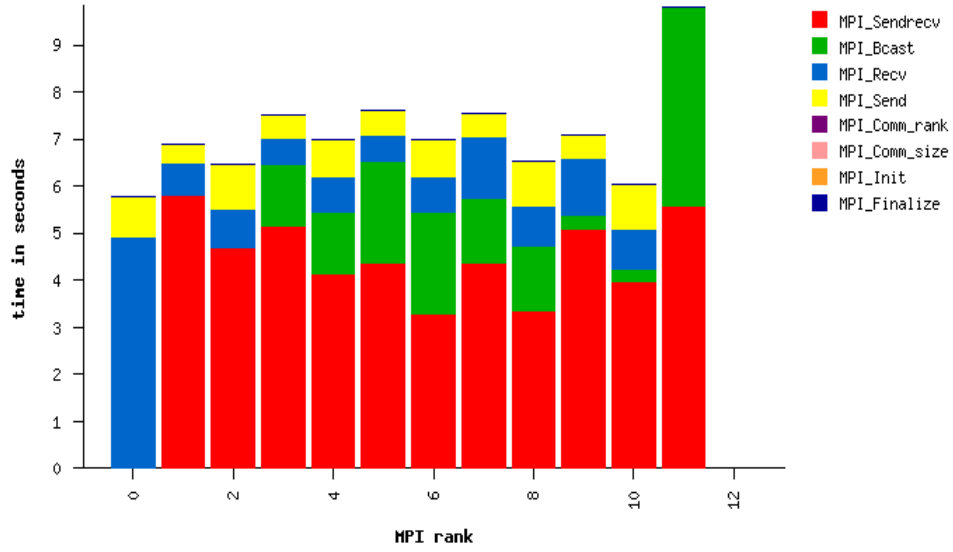
以下是我針對該測資所做的 load balancing 實驗：




其中可以發現每個 process 在 I/O 的處理上花費的時間基本上相同，說明了資料的分配滿平均的。比較有差異的是 communication time 和 computing time，根據我的 implementation，可以發現 sender process 只會傳訊息給 receiver process 並等待 receiver process 回傳結果，這段時間裡其實 sender process 會一直 idle 在 communication 上，造成這兩種時間的分布比較不平均。

此外，termination check 的 communication 也是造成這兩種時間分布不均的原因，在 odd phase 裡先做好的 processes 會需要等待所有的 processes 都完成比較和交換之後才能繼續往下走，這個時間差其實就在 idle。

接著觀察一下該實驗的 communication pattern：

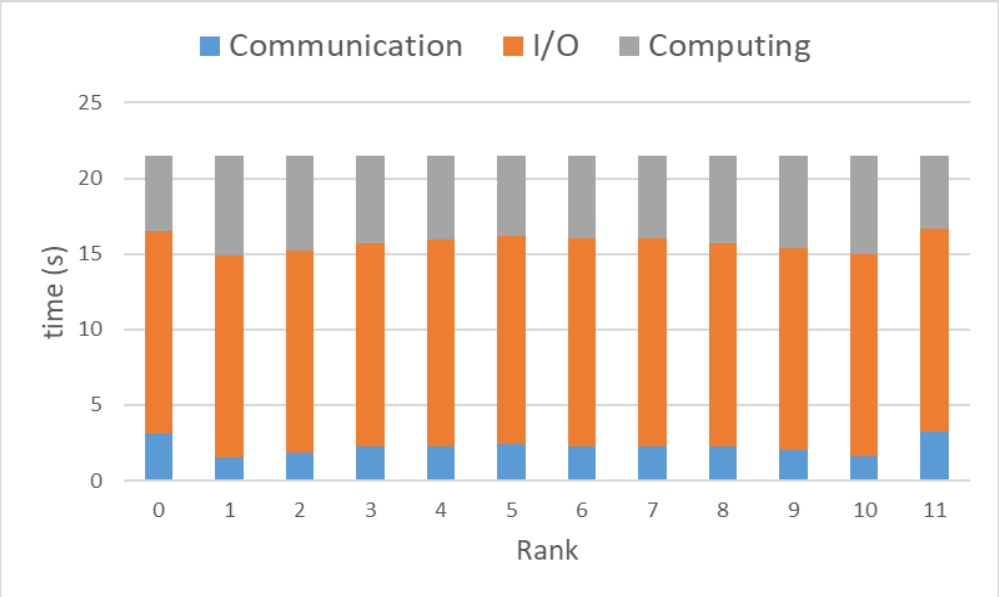



可以發現 `MPI_Bcast()` 的分佈相較於其他 MPI communication 的操作來說較為不均勻，說明了有些 processes 的確 idle 在 termination check 的 communication 上較久，有些則否。

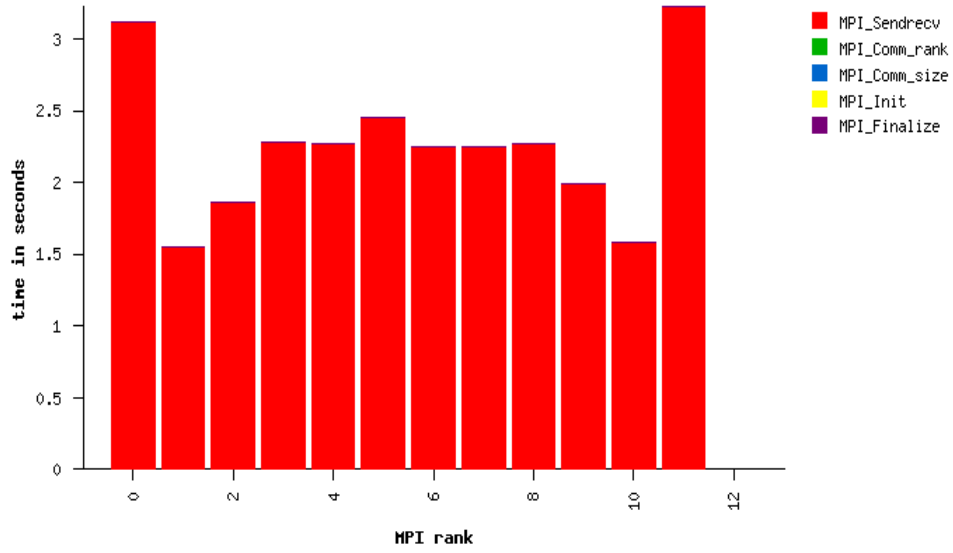
因此，我做了一些調整，原先我是透過每個 process 在 odd phase 的結尾向 rank 0 的 process 報告自己的排序狀況來做 termination check，但仔細思考過後可以發現 Odd-Even sort iteration 次數的 upper bound 為使用的 processes 的數量，所以事實上可以不用透過溝通的方式來做檢查。

此外，為了解決單向訊息溝通會造成 sender process 在 idle 的問題，我試著採用了雙向溝通的方式，交換 pair 中的 processes 會互傳其 local elements 給對方，然後同時作排序，前面的 process 取較小的部分，後面的則取較大的部分，這樣就不會有人在 idle 且可以都改用 `MPI_Sendrecv()` 來提升 message passing 的速度。

以下是修正過後的實驗結果：



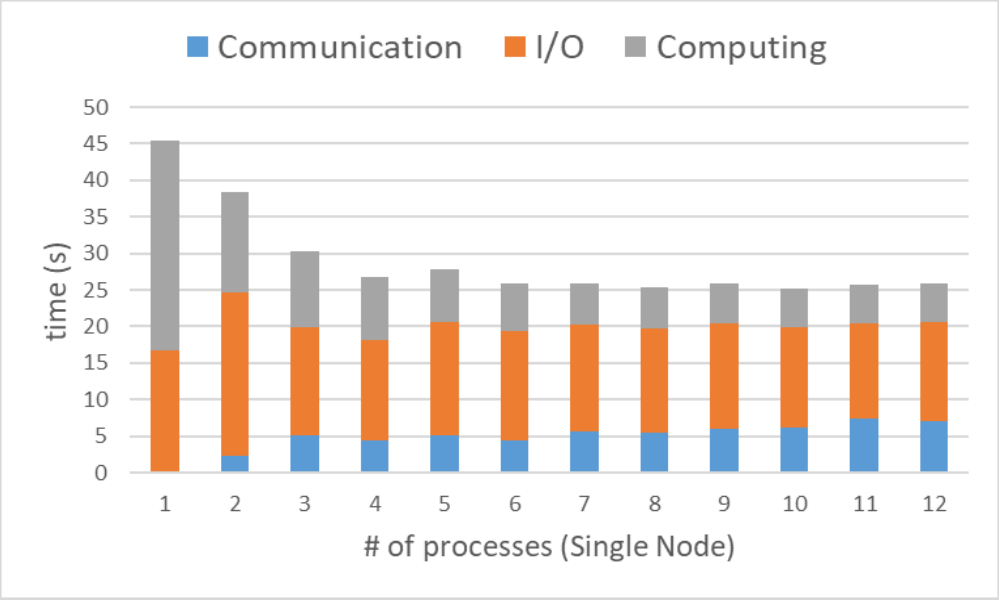

可以發現優化過後的總執行時間有明顯的下降，同時 communication time 和 computing time 的分布也變得比較均勻。




## Another Experiments

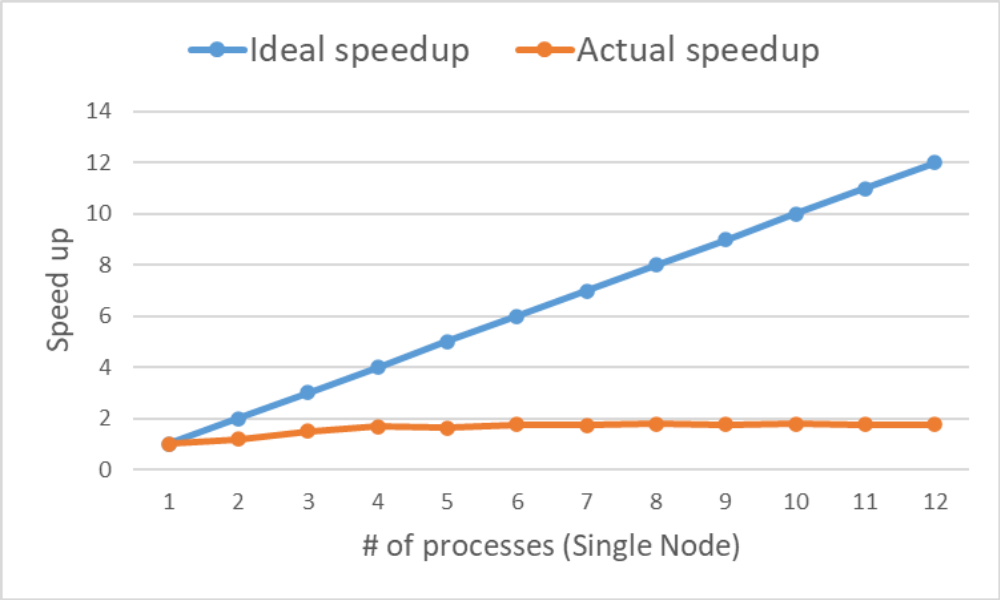
以下實驗使用原先的 implementation，為尚未優化過的版本。

- 在 Single node 上測試不同平行度



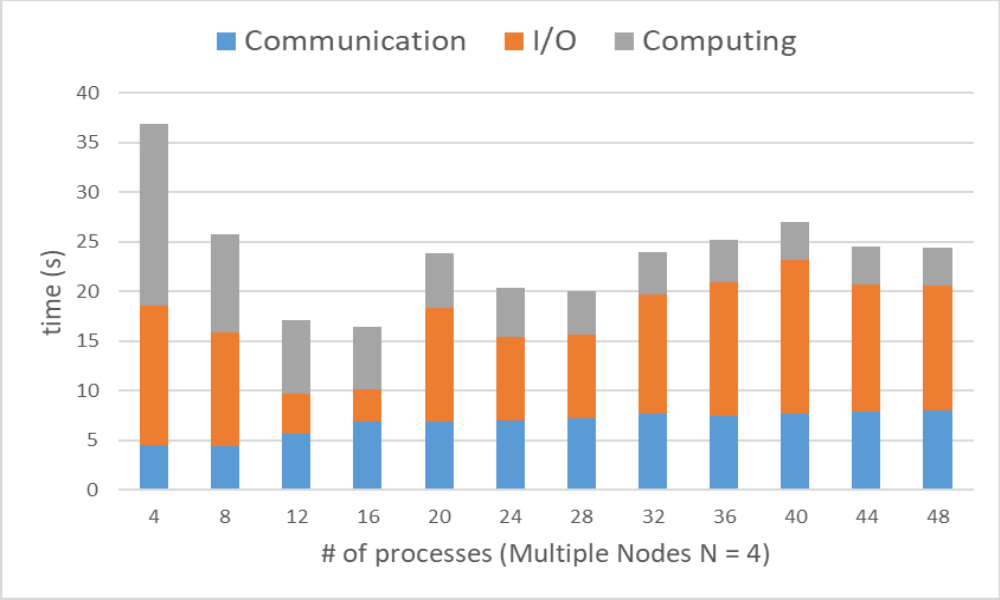

可以觀察到 I/O time 以及 computing time 都會隨著平行度上升而下降，原因是每個 process 被分配到的資料量變少，所以不管是在讀寫資料或是資料交換的速度上都會有顯著的提升。然而，唯一相反的是 communication time，可以觀察到 processes 之間溝通的時間成本會隨著平行度上升而跟著提升。

接著看一下實驗的 Speed up：



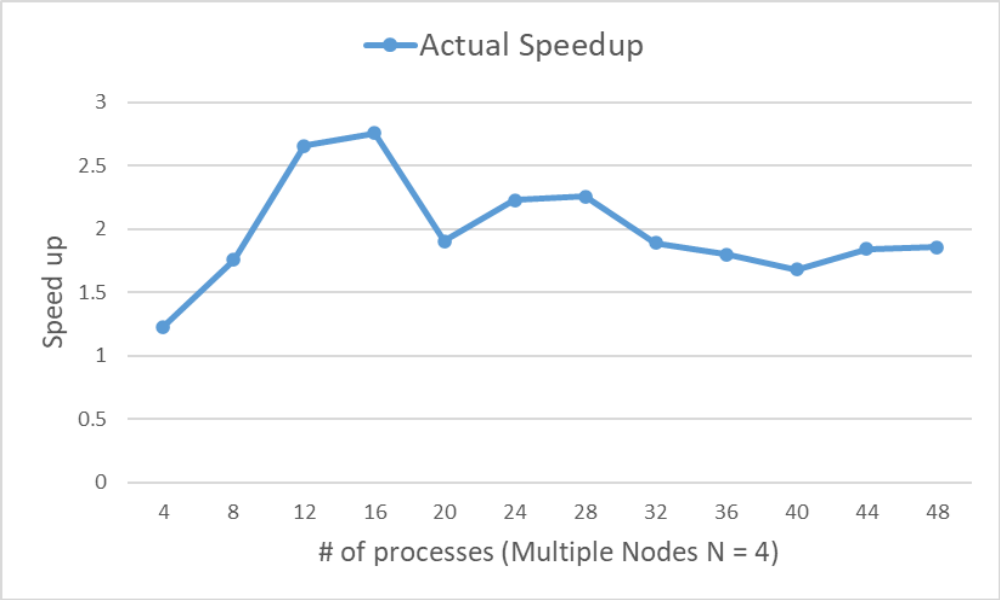

從 Speed up 的趨勢來看，可以發現 strong scalability 其實沒有很好，甚至沒有一個平行度達到兩倍以上的提速，詳細的分析留在之後的討論。

- 在 Multiple nodes 的環境下測試不同平行度（N = 4）



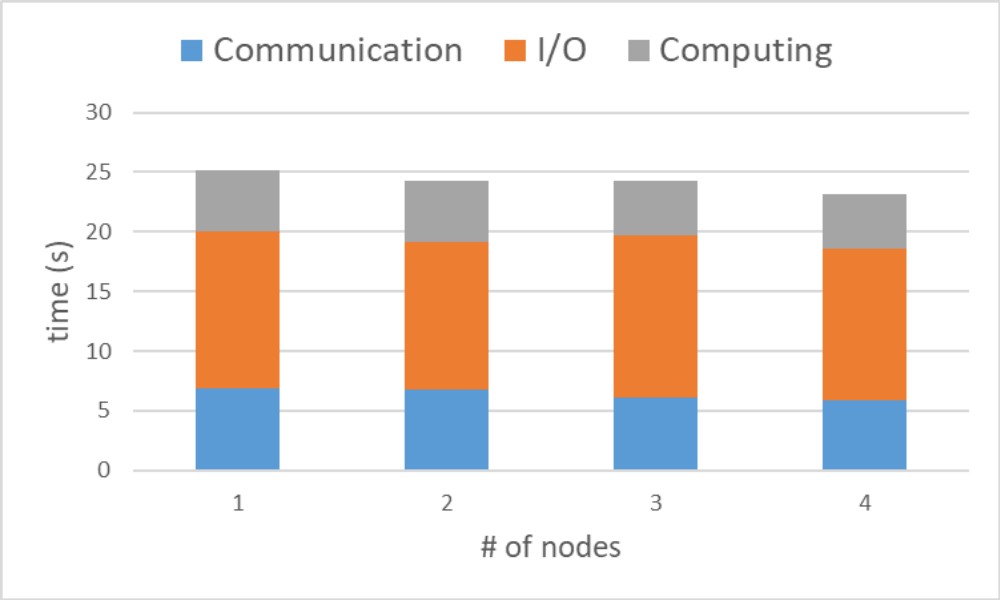

可以觀察到 computing time 基本上還是隨著平行度上升而下降，因為處理的資料量變少，而 processes 和 nodes 之間的 communication overhead 也可以很明顯的看出上升的趨勢。至於比較 tricky 的是 I/O 的 performance，我們可以觀察到 I/O performance 的 pattern 沒甚麼規律，直覺上應該要隨著平行度的上升而下降，但在圖中觀察不到這樣的趨勢。

觀察一下實驗的 Speed up：

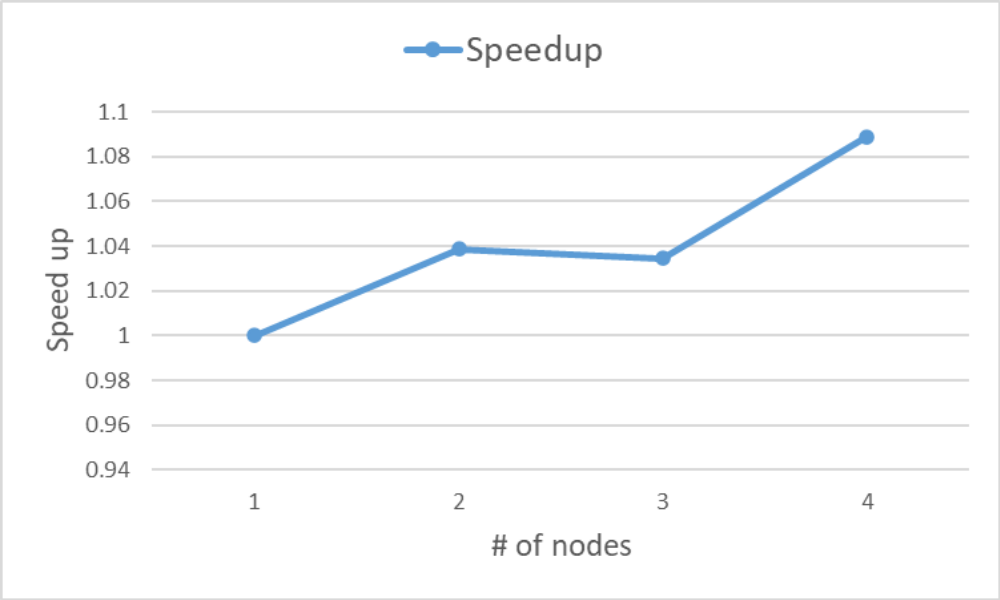



可以看到因為 I/O performance 的不穩定，加上隨著使用的 processes 數量越多，communication 所帶來的 overhead 影響越大，造成了整體 Speed up 的趨勢變得相對曲折，真要說可以看出上升趨勢的地方，大概就是 4, 8, 12, 16 這幾個 processes 數量的表現。而針對 strong scalability 來看，整體提速大致落在三倍以下，再次驗證了整個 program 的 scalability 並沒有很好，不管是在 Single node 或是 Multiple nodes 的環境下。

- 固定平行度的情況下測試使用不同數量的 nodes（n = 12）

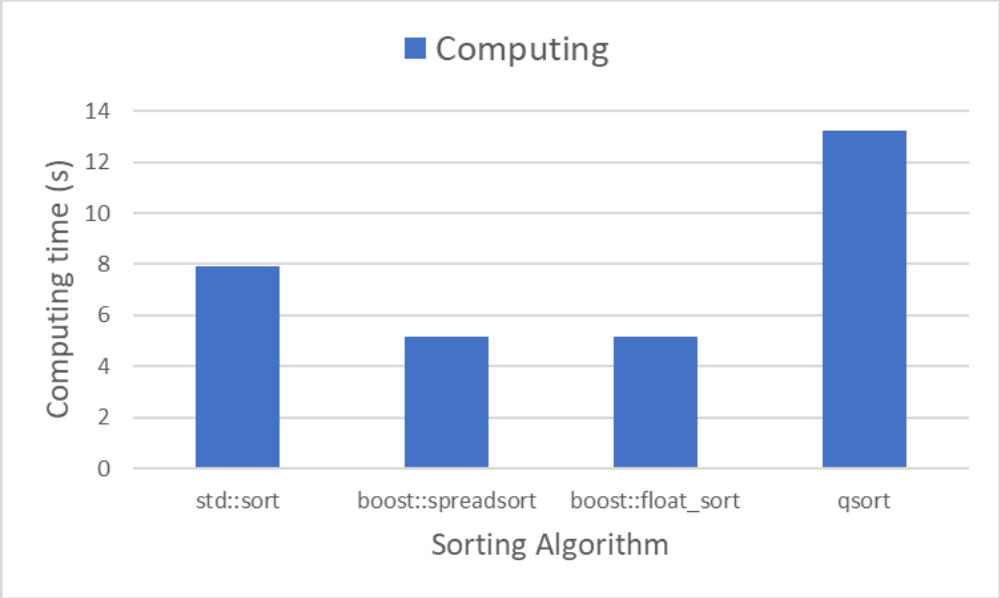



從上圖我們可以觀察到隨著使用的 nodes 數量增加，總執行時間有些微的下降，這可能是因為使用多個 nodes 在 resource contention 上相較於只使用一個 node 的情況競爭比較不會那麼激烈。而從 Speed up 來看，可以很明顯的看出上升的趨勢。




- 在 local sort 時嘗試不同 sorting algorithm

  由於 local sort 的時間會被算在 computing time 裡，使用適合的 sorting algorithm 可以有效幫助減少 computing time，尤其是在 problem size 很大的時候。




這邊我測試了一些公認比較快的 sorting method，像是 `std::sort()` 以及 `qsort()`，至於 `boost::spreadsort()` 以及 `boost::float_sort()` 則是因為 input array 的 data type 為 float，根據 boost C++ library 的 document，這兩種 sorting method 在 sorting 的對象為浮點數時，速度會比 `std::sort()` 快很多，從上面的結果來看也是如此。

## Discussion

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

  從結果來看，communication 和 I/O 所花的時間相較於 computing time 來的多。就尚未優化的版本來說，communication 的時間很大一部份是來自於等待其他 processes 回傳結果的 idle。為了避免這種狀況，我在前文中有嘗試使用雙向溝通的方式來做資料的交換排序，同時也利用平行度作為 iteration 次數

的 upper bound 來達成不溝通的 termination check，讓整體執行時間以及 communication 的 overhead 有明顯的下降。

至於 I/O 的 performance，主要是因為我在實驗的時候，output file 的位置我是寫在 home 目錄底下，使用的 disk 分區和 judge 不同，所以造成 I/O 花費的時間特別的久，這部分我倒是不太清楚要怎麼不寫在 home 目錄下做測試，也不太確定如何做優化。

- 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.

  從結果來看，整體 program 的 strong scalability 並沒有很好，甚至沒有任何平行度有超過三倍的提速，造成這種狀況的原因可能有以下兩種：

  - 因為我做實驗的時間比較接近 deadline，所以 cluster 並不是很穩定，針對同一個平行度在測 I/O 的時間時，短則可能會出現 4 秒，長則會出現一分鐘。儘管我已經嘗試針對同一筆實驗做多次測試並取最小執行時間，但仍然有可能取到的結果並不是最好。
  - Communication 的 overhead 依舊太大，雖然優化時有嘗試更好的 communication pattern，但是整體的 speed up 依舊離 ideal speed up 很遠。

  針對第一種，唯一的解決方式就是趁 cluster 比較穩定的時候早點開始做實驗，或是針對單筆實驗嘗試更多次，這樣應該可以很好的降低環境的干擾。而針對第二種，可以再思考看看有沒有更好的 communication pattern，只不過這部分我目前沒有甚麼想法。

# Conclusion

這算是我第一次寫這種平行的 program，之前都是寫 sequential code，沒有類似的經驗。藉由這次的作業，我發現在平行程式的世界裡如果想要讓效能達到極致，需要考量的方面實在是很多，像是 communication overhead 以及 I/O performace，然後在 debug 上面也不像 sequential code 那麼好 trace，需要在腦中模擬程式的運行以及 processes 之間的 communication pattern，實在是十分的燒腦，在 debug 上面也花了不少的時間，不過幸好還是順利地完成了這份作業，同時也學到了很多。