# HW 1: Odd-Even Sort


## 105062600 Yi-cheng,Chao 02:10, October 23, 2016


---

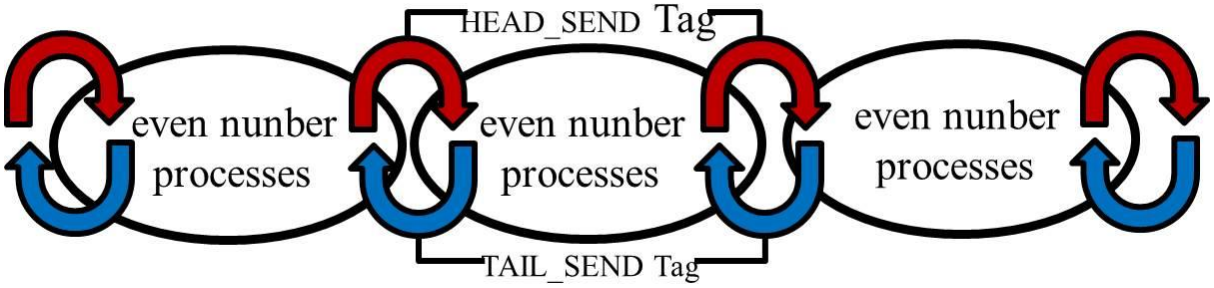
## Design Concepts - Basic

1. 在一開始為了排除 process 數大於我的資料數 N 的情況，因此我使用了 MPI_Group_range_excl 指令來去掉冗餘的 process，並且 assign N 到 size，如此一來便不會使用到多餘的 process，此外若執行指令完後發現資料數為 0，則直接 MPI_Finalize()離開程式。
2. 因為限制在 element-level 且只能跟 adjacent element 溝通的 odd-even sort，因此 index ID 就相當重要,歸納後發現有以下依據：

   在 odd phase，index 為偶數者會與前一個 index 的 element 溝通，奇數者則會與後一個 index 的 element 溝通；

   在 even phase，index 為奇數者會與前一個 index 的 element 溝通，偶數者則會與後一個 index 的 element 溝通。

   根據這點，我在每個 rank 都記錄他的第一個 element 的 index，稱之為 head，而最後的一個 element 的 index 稱之為 tail，只要判斷每個 process 的 head 及 tail 的 index 即可知道他是否要與前一個 rank 做 point-to-point 的 communication，然後再由 head 去判斷 processor 內的 local sort 起始位置。
3. 此外在 process 內資料數為 even number 時，會發生頭尾都要進行 point-to-point communication 的情況，為了使 process 判別是 Head 還是 Tail 送來的資料，我設計了兩個 Tag 來進行判別，如此就算同時收到兩筆 data 也可以由 Tag 去進行判斷 source 是 Head 還是 Tail，如下圖所示：




4. 在 MPI-IO 的部分上，使用 MPI_File_read/write_at 來進行讀檔及寫檔，只要設定好 offset，每個 process 便可以從指定位置開始讀取/寫入檔案，目的也是精簡執行時間，增加我的 performance。
5. 我資料分法有兩種版本，一種是把餘數都丟到最後的 process，一種則是把餘數從第一個 processor 開始配發，後者方法在資料量大的時候並沒有很顯著的提升 performance，反而在判斷 offset 及 head/tail 會花掉一些計算時間，即使是 worst case，最後一個 processor 也僅比其他 processor 多拿了 size-1 筆資料，因此差距在資料越來越大時會顯得越沒影響，因此最後繳交的版本決定使用前者方法來進行資料的分割。

6. 在 worst case 中，至多做 N 次的 odd-even sort(即最小值在最後一個 index 的 case)，但從執行的 round 來做實在 not efficiency，在這個部分我宣告了 sorted 和 all_sorted 兩個 integer，並且 define true 為 1，false 為 0，在每一次的 sort 都先給定 sorted 為 true，如果執行期間有發生任何 swap，則設 sorted 為 0，最後再去把所有 process 的 sorted 去做 logical and 的 MPI_Allreduce 為 all_sorted，每次執行只要判斷 all_sorted 是否為 1 便可以知道是否排序完成，只要排序完成便會跳脫迴圈，藉以達到精簡執行時間的作用。

# Design Concepts - Advanced

1. 大致上概念和 Basic 相同，但因為沒有了僅能與 adjacent index 溝通的限制，現在只限制在 adjacent process，因此我先使用 C++ algorithm library 中的 sort 指令把每個 processor 先做好自己的 local sort，最後再使用 merge sort 去做 process level 的 odd-even sort，平行演算法概念參考至[1]中的 Baudet–Stevenson odd–even merge-splitting algorithm，如下圖所示：

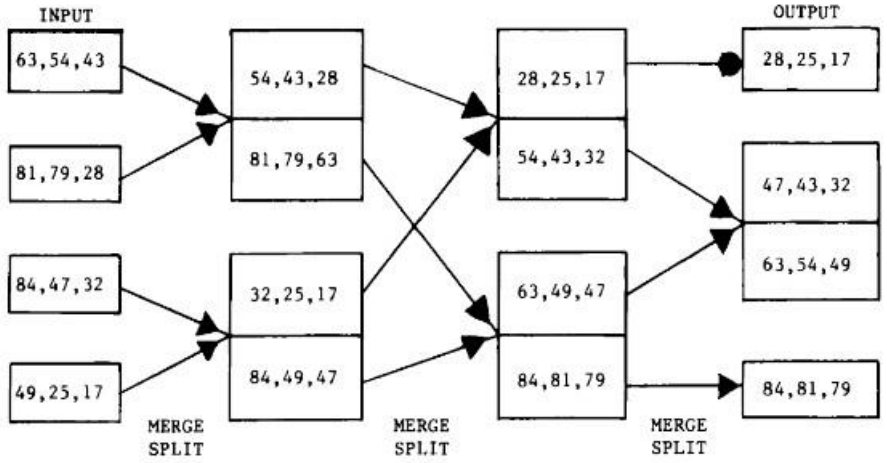


FIG. 14. Merge-splitting version of Batcher's odd–even sorting algorithm.

利用 process 間的 merge 及 split 對 data 進行 merge sort 分出較大和較小的半邊，即可從 Basic 版本到 Advanced 版本獲得非常顯著的 performance 提升，大致上概念就是從 element level 拓展到 process level 的 odd-even sort。

2. 資料分割的部分與 Basic 版本一樣，然而做 merge sort 的方法則是把要做 merge 的 processors 分為左半邊和右半邊，左半邊把全部資料送到右半邊，由右半邊做完 merge sort 再把較小的半邊送回左半邊，根據上圖 algrithm 可以知道 worst case 會執行 size 數的次數，但這時就沒辦法用有沒有交換去判斷是否排序完成了，因此我使用最原始的方法，執行 size 數的次數後就當作排序完成，便可以利用 Basic 提到的概念做已排序資料的產生與寫入。
3. 沒有使用 Non-Blocking 的 Isend()及 Irecv()是因為我在資料送出後並沒有作任何計算，也必須等資料完整收到才能去做 merge sort，為了確保資料的正確傳輸，我還是選用 Blocking Message Passing 的機制來進行 process 間的傳遞。

# Experiment & Analysis

## System Environment

執行程式使用課程提供的 Batch Cluster。

## Time Measurement

指令部分使用 MPI_Wtime() 來進行時間的量測，精度達小數點後六位。

Execution time 量測方法為取 MPI_Init() 到 MPI_Finalize()的 diff。

I/O time 則先求每次 MPI_File_open() 到 MPI_File_close()的 diff，最後再做 I/O read time 和 I/O write time 的 sum。

Communication time 在 Basic 版本中為在使用到 point-to-point communication 指令的前後先取 diff，每次迴圈都累加計算出每個 process 的 communication time，再使用 MPI_Allreduce() 送到 total communication time 內取平均，即為平均的 communication time。

Computing time 則為 Runtime 去掉 I/O time 和 Communition time 的剩餘部分。

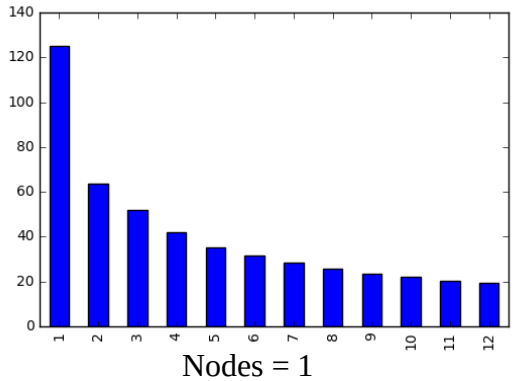
## Performance Measurement

再測資的部分，為了不讓 process 分配到的 data 數的不平均影響我的 performance，資料總數我取 Node[4] = {1, 2, 3, 4}與 ppn[12] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}乘積所產生的 48 種可能組合的最小公倍數 332640，如此一來便可以保證在任何 process 總數下都能整除我的資料總數。

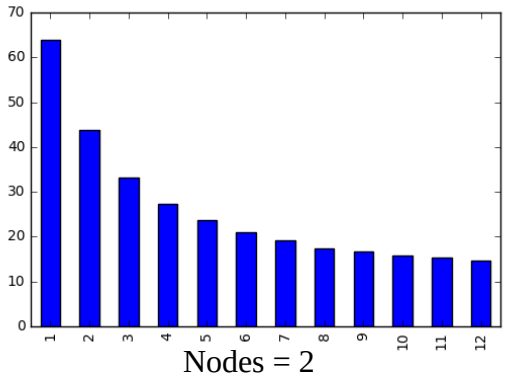
## Strong Scalability Analysis

### Basic (N = 332640) Strong Scalability

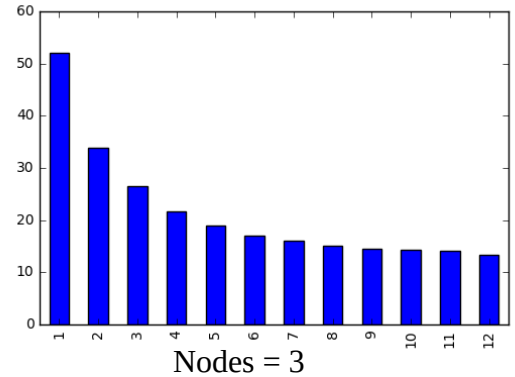
### (x-axis : # of process per node; y-axis : Execution time (s) )



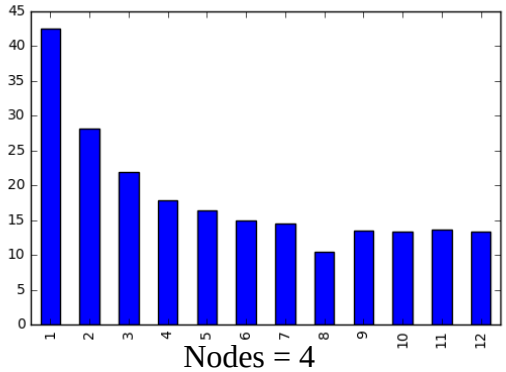

Nodes = 1




Nodes = 2

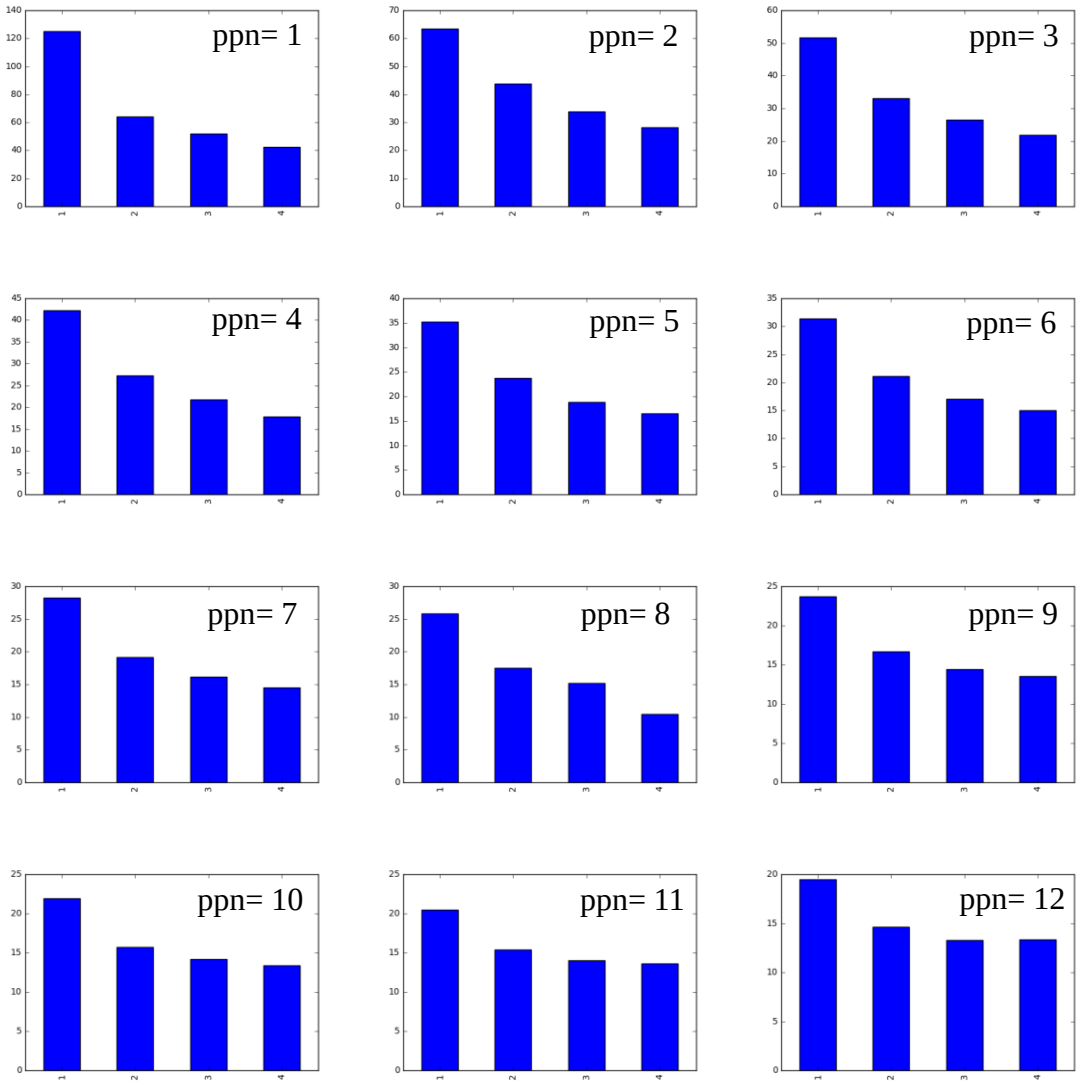



Nodes = 3




Nodes = 4

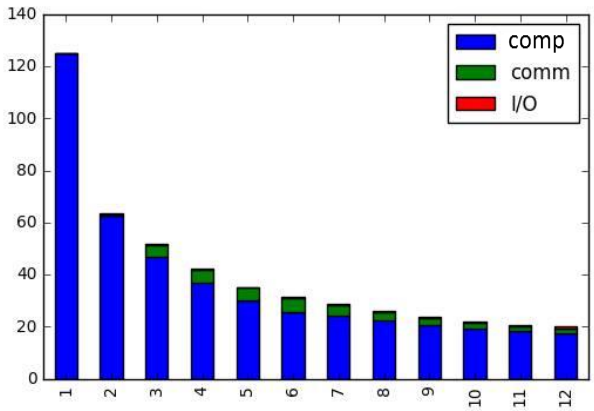
**Basic (N = 332640) Strong Scalability**
**(x-axis : # of nodes; y-axis : Execution time (s) )**



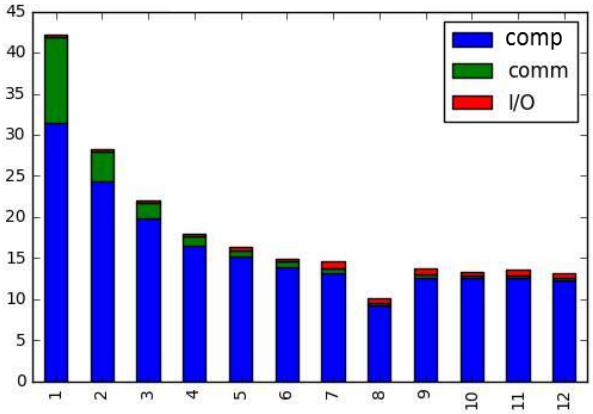

Basic 的圖形結果出來還滿讓我滿意的，由圖形可以看到隨著平行的 process 數增加，執行時間都會逐漸變小，只是 Execution time 在 process 越來越大的情況，加速的效果逐漸不明顯，在超過一定 process 數後，他的 Execution 便會降部下去。比較需要注意的是 process 數為 8 的倍數的時候 communication 會突然縮小很多，這部份尤其在{node, ppn}= {4, 8} 特別明顯，在第一組圖的 nodes= 4 狀況可以明顯看到 Execution time 突然下掉，我推測可能是和在 Tuning 時遇到的 P、Q 值有關，可能在{node, ppn}= {4, 8}的狀況剛剛好等於 P 乘 Q 值，使他的 performance 突然增加不少。

下一組圖則是對 node=1 及 node=4 去分離出 compution, communication 及 I/O time，從圖中我們可以清楚看到各種執行時間的比重。根據 node=4 的圖形我們可以看到，I/O time 的負擔會隨著 process 數的增加而上升，而在 process=1 的情況下幾乎沒有 I/O time 的延遲。

**Basic (N = 332640) Strong Scalability – Time Distribution Analysis**
**(x-axis : # of process per node; y-axis : Execution time (s) )**

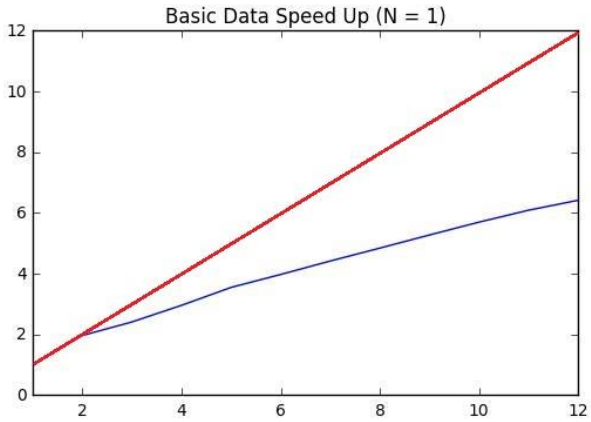


Single node (node = 1)

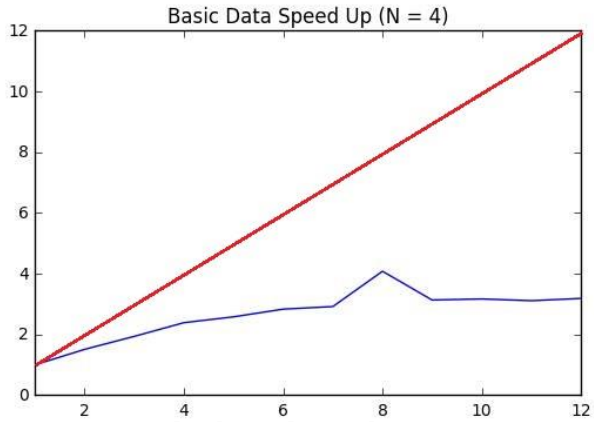


Multiple node (node = 4)

**Speedup Factor Analysis**
**Basic (N = 332640) (x-axis : # of process per node; y-axis : speedup factor )**



Single node (node = 1)



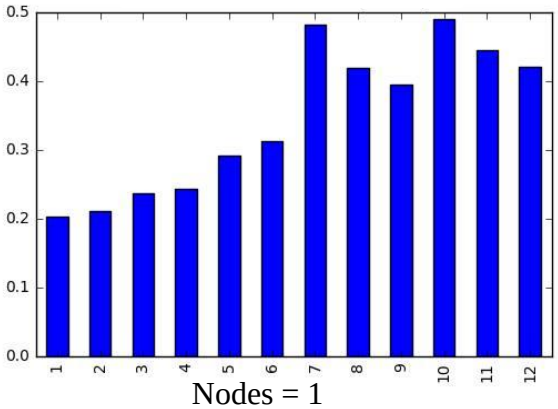
Multiple node (node = 4)

由剛剛前面的資料繪出 speed factor – process 圖，可以更清楚的看到上面所得的結論，加速效果會在超越一定的 process 數後就逐漸平穩，而 multiple noed 上面出現的 peak 則是剛剛我們提及過的{node, ppn}= {4, 8}的狀況。

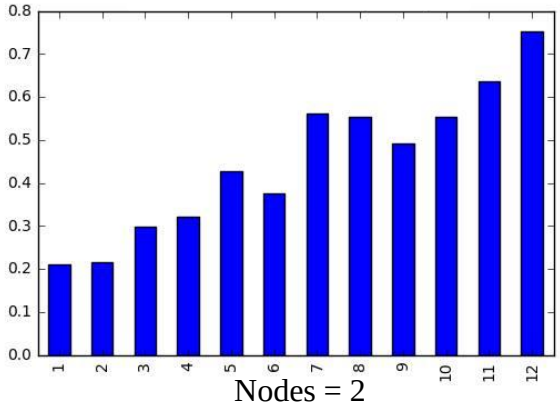
**Strong Scalability Analysis**

**Advanced (N = 332640) Strong Scalability**

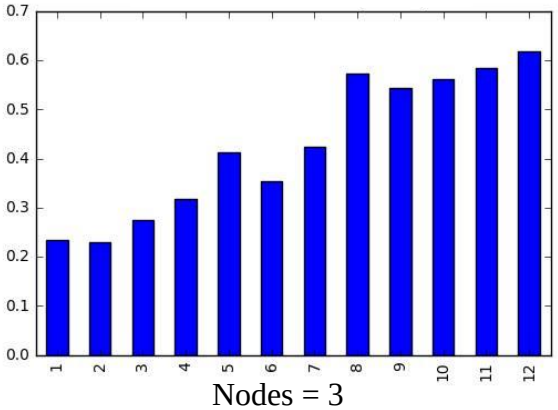
**(x-axis : # of process per node; y-axis : Execution time (s) )**



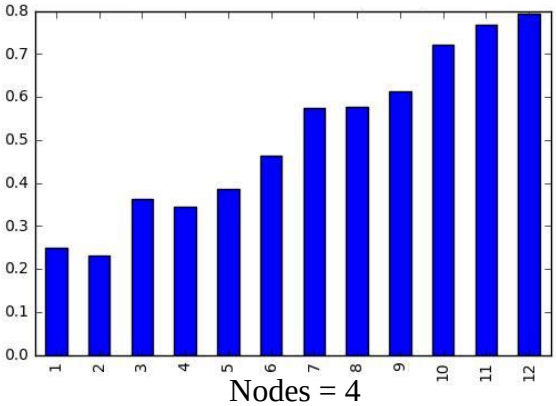
Nodes = 1



Nodes = 2



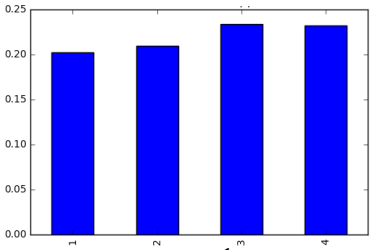
Nodes = 3



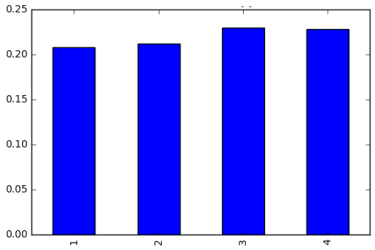
Nodes = 4

**Advanced (N = 332640) Strong Scalability**

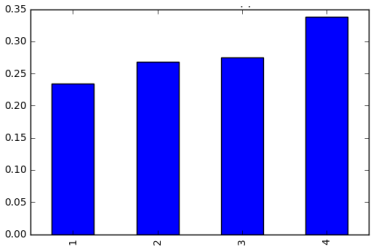
**(x-axis : # of nodes; y-axis : Execution time (s) )**



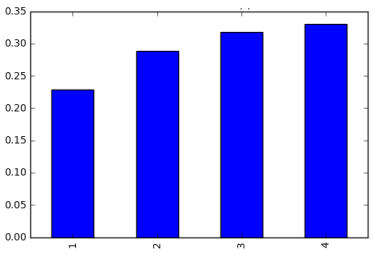
ppn= 1



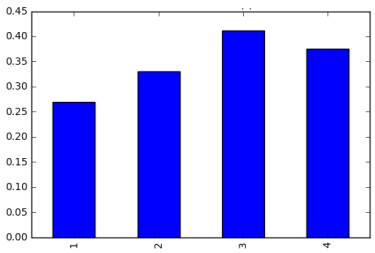
ppn= 2



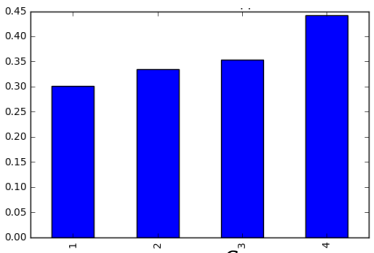
ppn= 3



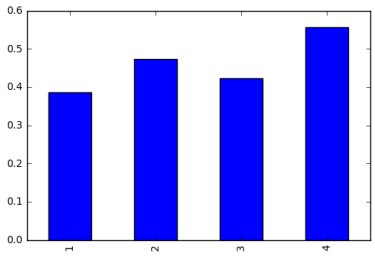
ppn= 4



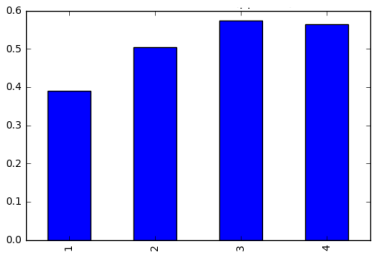
ppn= 5



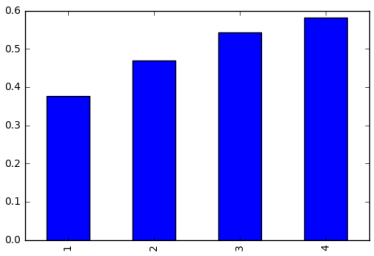
ppn= 6



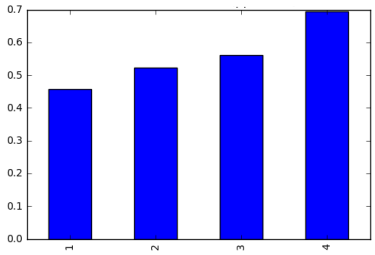
ppn= 7



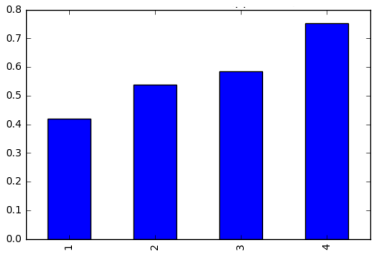
ppn= 8



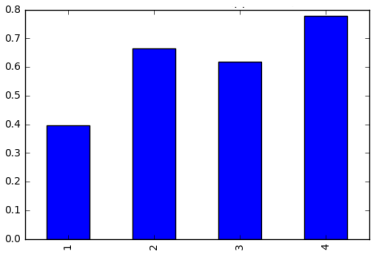
ppn= 9



ppn= 10

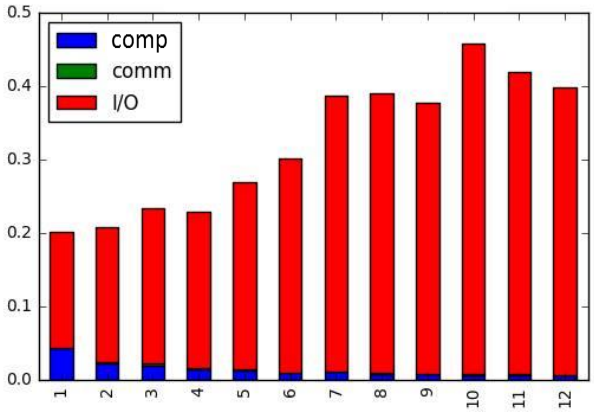


ppn= 11

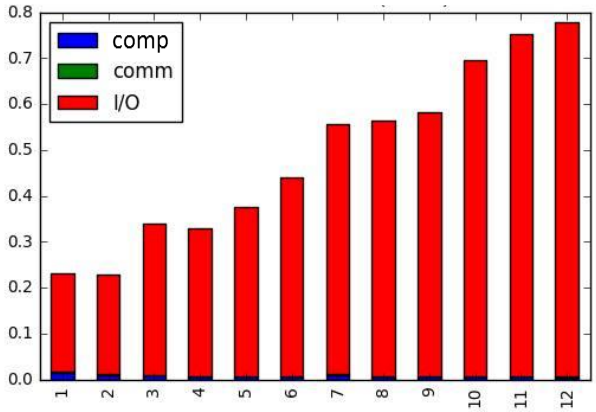


ppn= 12

**Advanced (N = 332640) Strong Scalability – Time Distribution Analysis (x-axis : # of process per node; y-axis : Execution time (s) )**

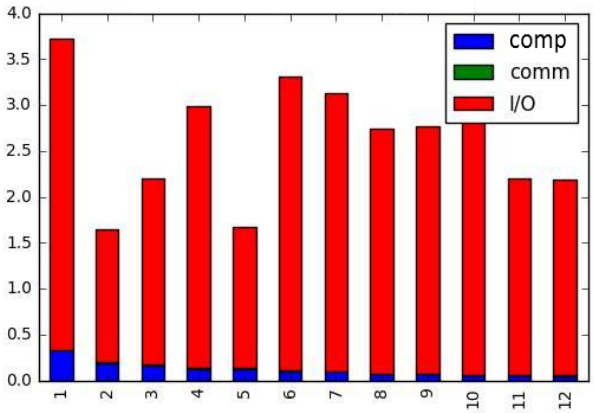



Single node (node = 1)

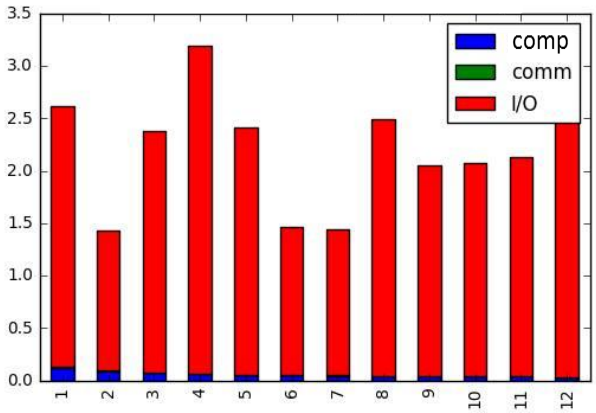



Multiple node (node = 4)

**Advanced (N = 3326400) Strong Scalability – Time Distribution Analysis (x-axis : # of process per node; y-axis : Execution time (s) )**



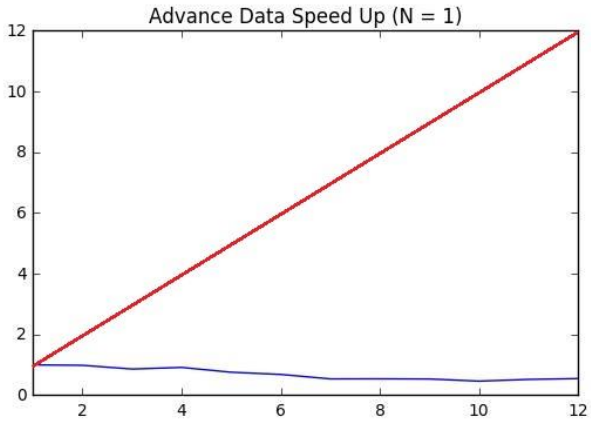

Single node (node = 1)



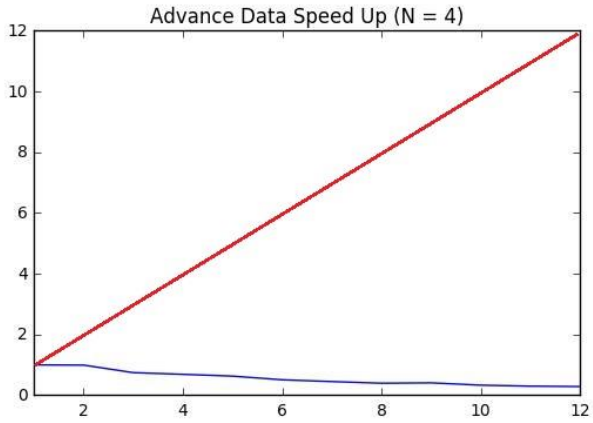

Multiple node (node = 4)

**Speedup Factor Analysis**

**Basic (N = 332640) (x-axis : # of process per node; y-axis : speedup factor )**




Single node (node = 1)

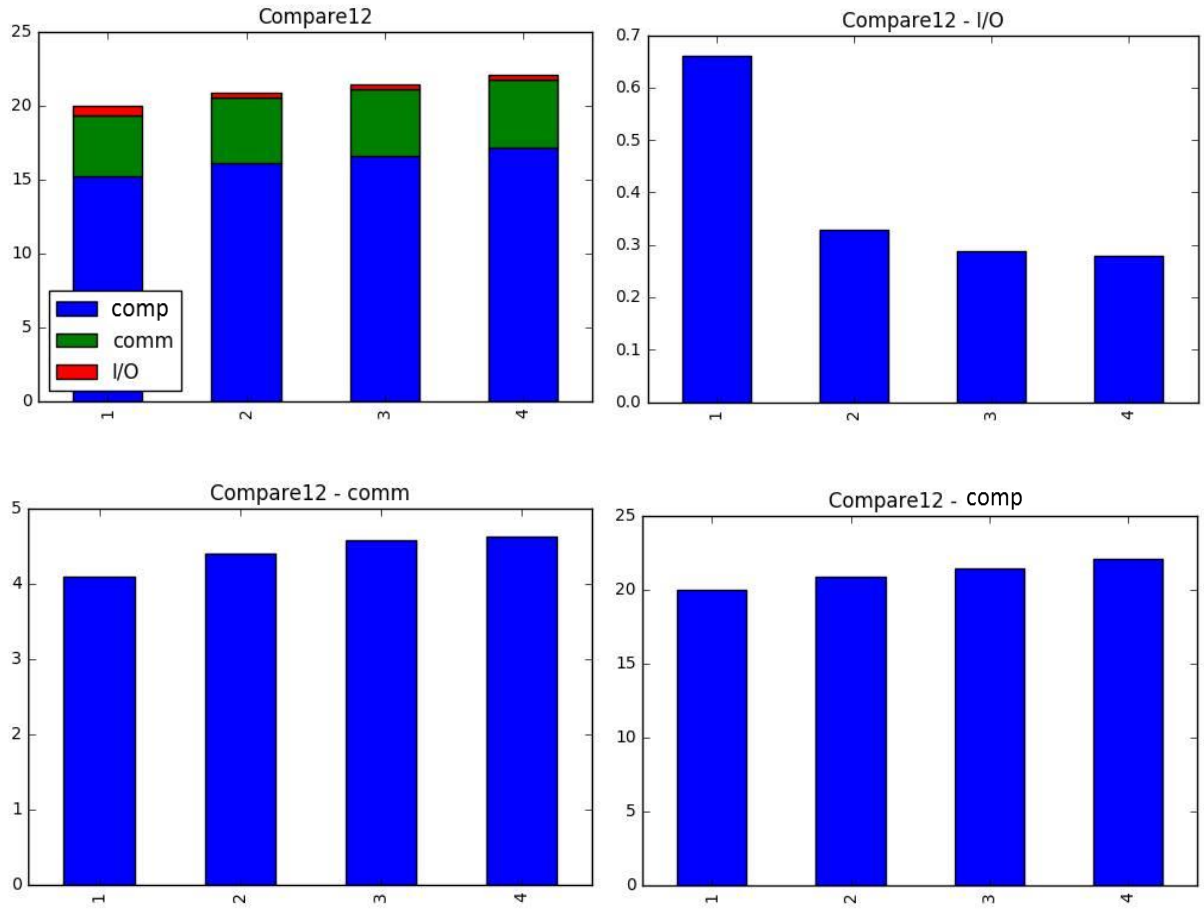



Multiple node (node = 4)

Advanced 則出現了預期以外的結果－隨著 process 數的增加，execution time 則卻是增加，我會在結論討論這點！由下面的時間分布圖也可以看到幾乎都是在做 I/O time，即使增加了十倍的資料量仍沒有改善，幾乎由 I/O time 主宰了整個 execution time，speed factor – process 圖也可以看到隨著 process 的增加卻出現 speed down 的結果！

# Conclusion

Basic 版本的結論用圖形來說明，基本上可以根據以下四張圖：




四張圖都是固定 process 數，差別在於 n 和 ppn 的組合不同，分為{node, ppn}= {1, 12}、{node, ppn}= {2, 6}、{node, ppn}= {3, 4}、{node, ppn}= {4, 3}，但時間分布卻有些微的不同，我們可以理解成每個 process 具有相同資料的基礎上，不同組合規格在 MPI 去做平行的表現。

如果是都集中在同一個 node 的話，會需要到比較多的 I/O time 去做資料的讀寫，我猜想可能是會先由 node 進行讀取在分配下去給每個 node 內的 process，但是因為在同一個 node 內所以溝通時間會比較少，整體的 execution time 也會小很多，因此如果在限制固定的 process 下，應該要盡可能集中在同一個 node 內，這樣的 performance 比較好。

然而 Advanced 圖表很不符合期望，且 execution time 也幾乎都花在 I/O time 上，我歸因於輸入的資料量太小，因為從圖表中可以看到 compution time 微乎其微，可能要在多一點的 data 去執行才可以看到他的效果吧！因為我測試的資料量頂多不到 12MB，不過從 element level 拓展到 process level 的 odd-even sort，兩者的 performance 根本不在同一檔次，如果要說得到 advanced 的效果，也確實得到了！

# Homework Reviews & Comments

由於非資工本科畢業，我在程式撰寫上遇到很多困難，儘管自己可以想像在概念上該怎麼去操作，在程式撰寫初期，因為實作的經驗尚不足，以及對 C++本身的了解尚淺，處於一種有想法但卻不知道怎麼去實行的地步，大概花了很多時間去複習指標的使用、取址之類的東西，不然就是看一些基本操作，最後能在./judge 執行後跑出 all accepted 真的非常開心，有種超越自我極限的感覺。

當初修課時打聽到，這門課 Loading 不是普通重，不過摸索的過程外加整理數據真的會實實在在地了解這個程式，雖說邊做作業心裏都在抱怨，不過老實說再打心得的時候是真的有滿滿的收穫的，對 MPI 整個程式的些微了解，還有他各項的 performance 之類的，也了解到 SPMD model powerful 的地方，在撰寫程式的同時也會在腦中描繪類似程式的平行世界，只是允許溝通，說真的還滿酷的呢！

因為從大學養成的壞習慣，不到 Deadline 臨頭不碰作業，這次真的爆炸了，在最後一天大家瘋狂的送資料，Queue 擠得跟什麼一樣，如果要說遺憾，大概就是為什麼不早點寫好去取得完美的數據來進行分析吧。

在程式撰寫的一路上求助於很多人，很感謝大家讓我問一些腦殘問題，也在此特別感謝陳振群助教讓我三番兩次的叨擾，有些問題雖然不懂但很怕覺得問的是很基本的問題不敢到討論區上直接發文詢問，感謝助教都會能替我解惑！

最後算是建議的部分，最後一天還是看到很多人一次送很多檔案到 Queue 中，而且是動輒就要執行 20 分鐘以上的 large-testcase，可是.sh 檔我記得在 tuning 有講到在 Queue 中不會去看.sh 檔的規格，會在 execution 狀態時才會去讀，雖然對實驗的數據精準度有很大的幫助，只是我覺得這樣瓜分掉資源有點不妥，希望可以設定最多只能再 Queue 中等待的時間限制，若超過就 delete 掉吧！舉 10 月 22 日晚上 10 點到半夜 4 點為例，幾乎使用 vim 去編輯 submit.sh 都跑不動，可能更改一個參數就要花 20 分鐘左右了不誇張，那時還很崩潰的跟助教問起了遲交的 penalty，還希望助教體諒。

# References

[1] Marshall C.Yovits ,“Advances in Computers”, vol. 23, pp. 333