Parallel Programming -Odd Even Sort

106062530 張原嘉

1. Implementation.
   - Basic
     - ➢ 程式步驟：
       i. 判斷輸入合法性，若輸入參數的格式不正確，則返回錯誤。變數設定與資料處理（將在＂主要想法＂部分解釋意義）
       ii. 判斷 Process 數量是否大於輸入參數中的＂Size_of_list＂，也就是N，若大於，表示有些 Process 無資料可處理，此時使用＂MPI_Group_range_excl＂放棄這些 Process（用＂MPI_Comm_group＂分類哪些要保留哪些要捨棄的 Process）。
       iii. 變數設定與資料處理（將在＂主要想法＂部分解釋意義）
       iv. I/O 部分，使用 MPI_File_read_at / MPI_File_write_at 處理，搭配 Offset（指定各個 Process 從檔案的哪個位置開始讀取 / 寫入自己的資料）進行讀檔寫檔。
       v. Odd Even Sort 程式主體
       vi. 各 Process 分別寫回檔案
     - ➢ 主要想法：

       首先每個 Process 所要處理的資料量儲存於 dataOfperProcess，它的值來自給定的＂Size_of_list＂然後除以 numOfProcess（Process 總數），又因為會有餘數，因此把剩的資料丟給最後一個 Process 處理。接者，因為限制只能與相鄰的元素溝通，所以如何得知元素的相對位置並正確使用就顯得重要。在此利用 head、tail 作為 Odd Even Sort 判斷 Odd Phase 與 Even Phase 與元素溝通的依據。元素溝通採用 MPI_Sendrecv 函式，只要設定 Tag、Datatype、Amount、收送 Process 的 Rank 即可，具有簡潔、易寫等優點，Advanced 亦採用此函式。

       根據觀察（以 Odd phase 為例）：

       若某個 process 的 tail 是奇數，則此 process 緩衝區的最後一個資料會與下一個 process 的第一個資料做溝通（兩元素相鄰），若大於下

一個 process 的第一筆資料，則兩者交換（額外開一個 tempBuffer 當交換暫存區），繼續下一輪 sort。

反之，若某個 process 的 head 是偶數，則此 process 緩衝區的第一個資料會與前一個 process 的最後一個資料做溝通（兩元素相鄰），若小於前一個 process 的最後一筆資料，則兩者交換（額外開一個 tempBuffer 當交換暫存區），繼續下一輪 sort。

與其他相鄰的元素都執行該做的檢查後，執行 localSort，將各自 Process 內的資料進行最後排序。

因 Even phase 與 Odd phase 雷同，在此不再贅述。

不論 Odd phase 或 Even phase，有一點值得注意：在判斷完某個 Process 的 tail 是奇數/偶數後，要確定該 Process 不是最後一個 Process，若是的話，程式會報錯（因為它右邊（下一個）沒有東西）。反之，若判斷完某個 Process 的 head 是偶數/奇數後，要確定該 Process 不是第一個 Process，若是的話，程式也會報錯（因為它左邊（前一個）沒有東西）。

- Advanced
  - ➢ 程式步驟：
    i. 判斷輸入合法性，若輸入參數的格式不正確，則返回錯誤。
    ii. 判斷 Process 數量是否大於輸入參數中的 " Size_of_list " ，也就是N，若大於，表示有些 Process 無資料可處理，此時使用 " MPI_Group_range_excl " 放棄這些 Process（用 " MPI_Comm_group " 分類哪些要保留哪些要捨棄的 Process）。
    iii. 變數設定與資料處理
    iv. 利用 C++ algorithm library 的 Sort()，對各 process 內部的資料進行排序
    v. Between process 的 Odd even sort
    vi. 各 Process 分別寫回檔案

➢ 主要想法：

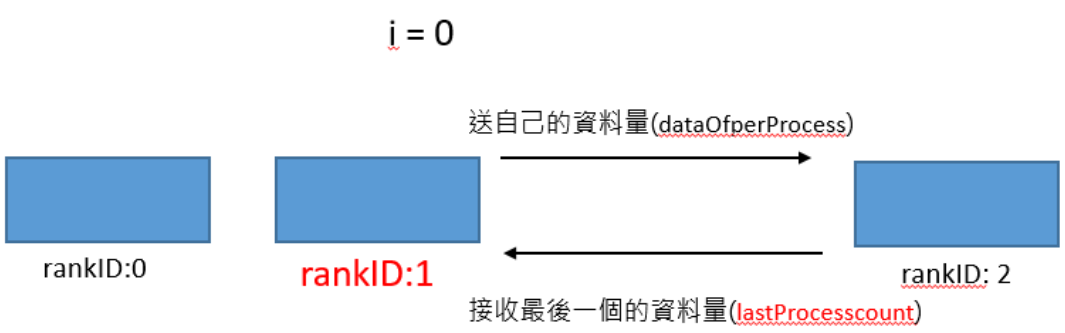
在 Advanced 中，與 Basic 版最大的不同在於各 Process 的資料量。在 Basic 中，各個 Process 取得的資料量，其計算方法為：總資料量除以總 Process，餘數丟給最後一個 Process。而 Advanced 版本，若使用上述方式，則有些 testcase 會無法順利通過。因此採用以下方式計算各 Process 的資料量：

```
dataOfperProcess      = (int)(ceil((double)atoi(argv[1])/(double)numOfProcess));
generalProcesscount = dataOfperProcess;
lastProcesscount = atoi(argv[1])- (dataOfperProcess * (numOfProcess-1));
```

GeneralProcesscount 是除了最後一個以外的 Process 資料量，lastProcesscount 則是最後一個的 Process 資料量。此方法會讓資料盡量分配給前面的 Process，而最後一個 Process 的資料量就會最少，如此可降低當遇到一些邊界條件的測資時，所可能引發的錯誤。因 Advanced 版僅限制 Process 而非 element 與左右的交換，因此先利用預設函數 sort()對各個 process 內部的資料排序（本來是用 qsort()，但後來發現 sort()的內部優化似乎比 qsort()好，且 qsort()要另外考慮參數的型態等）排序完後，再對 Process 進行類 merge Sort，保留較高(低)的部分，因為是 Process level 的 odd even sort，所以 performance 遠比 element level 的 odd even sort 還要好。

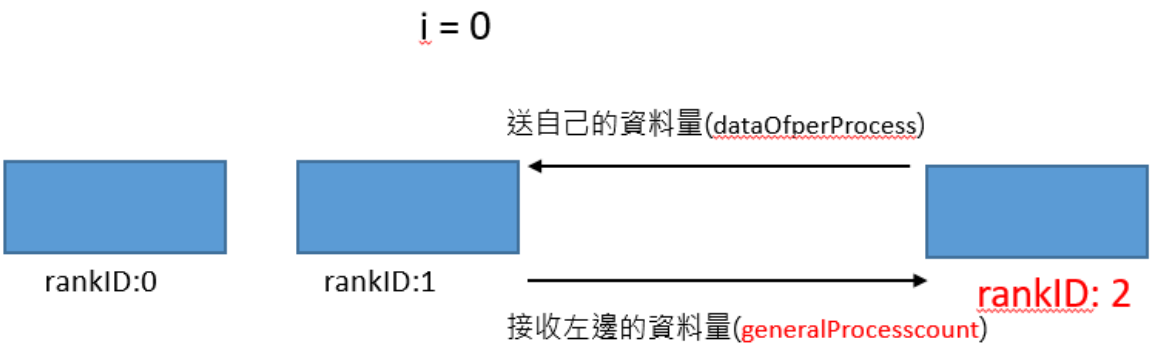
Between process 的 Odd even sort 採用臨時變數 i 紀錄 odd/even phase，而中止迴圈條件為 Process 最大數量，確保每個 Process 都被檢驗過而不會造成意外結果。

在 Odd-phase 時，若 Process 的 rankID 是奇數，且又是倒數第二個，則當它和右邊 Process（最後一個）溝通時，接收的資料量須為 lastProcesscount（因為最後一個 Process 有額外的資料量）



其他情況只要 Process 不是最後一個，就可以和右邊的 Process 溝通（呼叫 exchangeWithright()）

而若 Process 的 rankID 是偶數且為最後一個，則當它和左邊 Process 溝通時，接受的資料量須為 generalProcesscount（此變數的值與 dataOfperProcess 相同，只是特別設定方便辨別）



其他情況只要 Process 不是第一個，就可以和左邊的 Process 溝通（呼叫 exchangeWithleft()）

2. Experiment & Analysis
   - System Spec.

     課程提供之設備

     Intel(R) Xeon(R) CPU X5670 @ 2.93GHz、96GB memory、 2 x 6-core
   - Performance Metrics

     使用 MPI library 提供之 MPI_Wtime()進行時間紀錄，單位為毫秒(ms)。

     各時間記錄方式如下：

     Execution time：MPI_INIT()後到最後一個 MPI_Finalize()間的差值

     I/O time：I/O read & I/O write 各自 MPI_File_open() 到 MPI_File_close()的時間差值，兩者再相加。

     Communication time：在各個 MPI_Sendrecv()前後紀錄其時間差值，每個迴圈累加總時間，再利用 MPI_Allreduce()取平均。

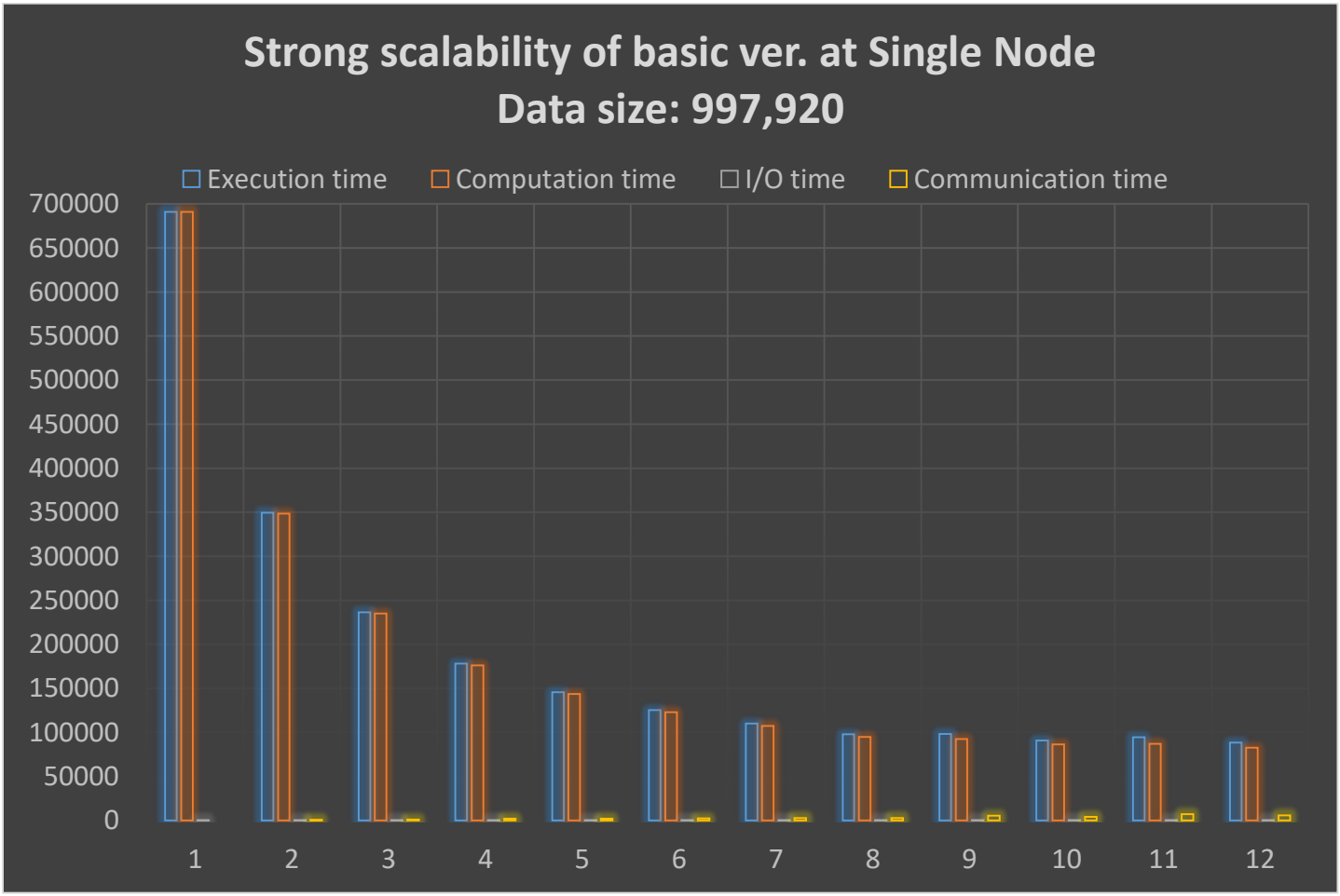
     Computation time： Execution time - I/O time -Communition time。

3. Experience / Conclusion

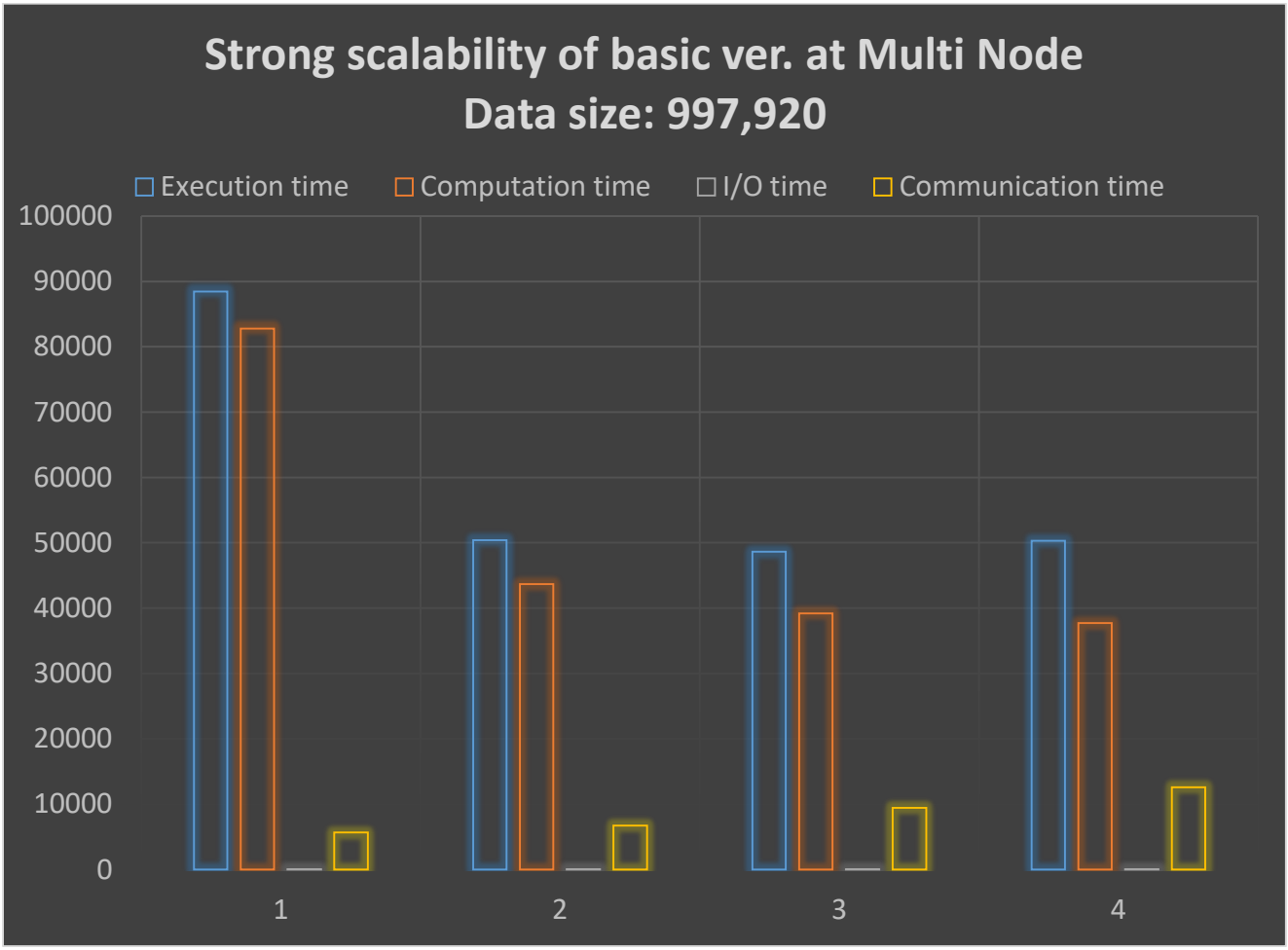
- Experience

  測試資料使用 997,920 筆資料（此數字可在各種 PPN 與 Node 組合下，把資料平均分配給各 Process），分別針對四個 Performance Metrics 在單一 Node 不同 process 數量、多個 Node 單一 process 數量的環境下進行測試。從第一張圖不難發現，隨著 process 數量增加，Execution time 及 Computation time 皆呈下降趨勢。

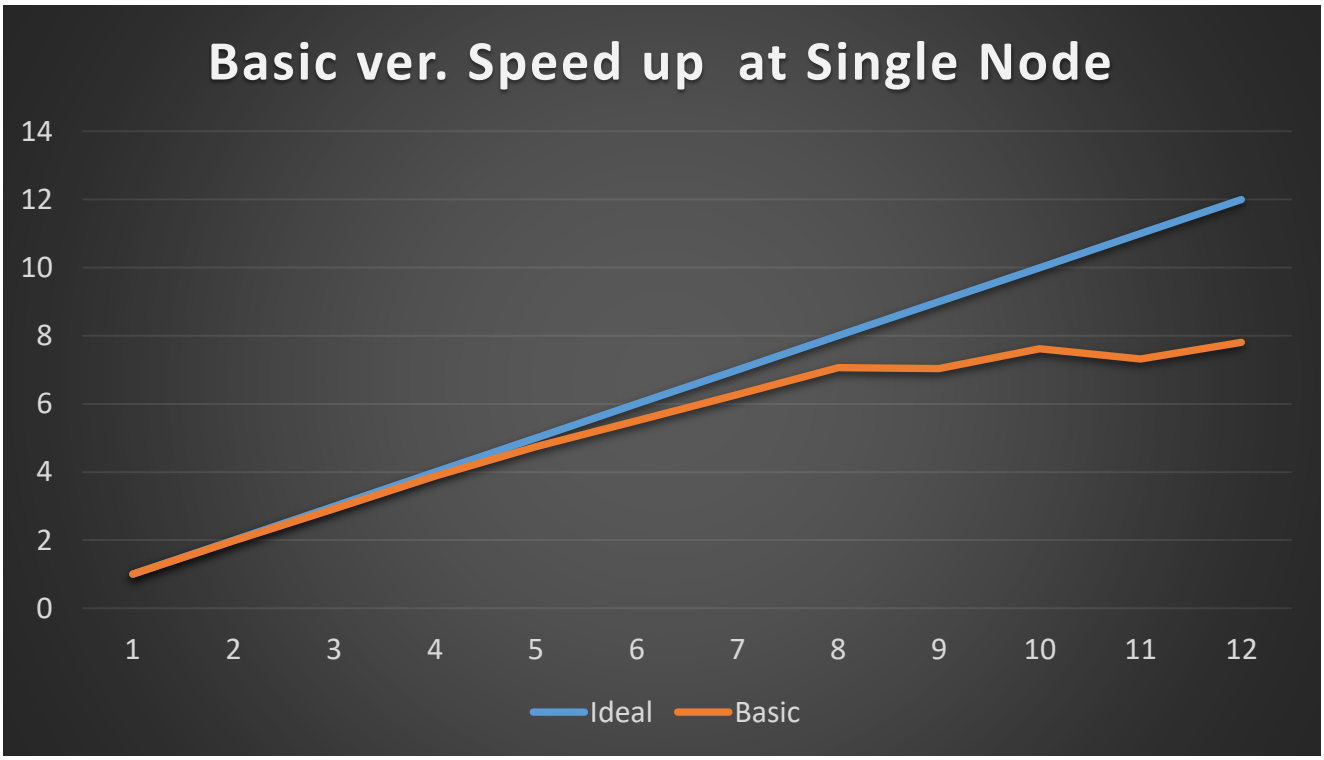
  第二張圖，隨著 Node 增加（PPN 固定 12），整體 Execution time 及 Computation time 亦呈下降趨勢。然因 Node、Process 數量皆增加，此圖中的 Computation time 增加趨勢，會比第一張圖來得顯著。



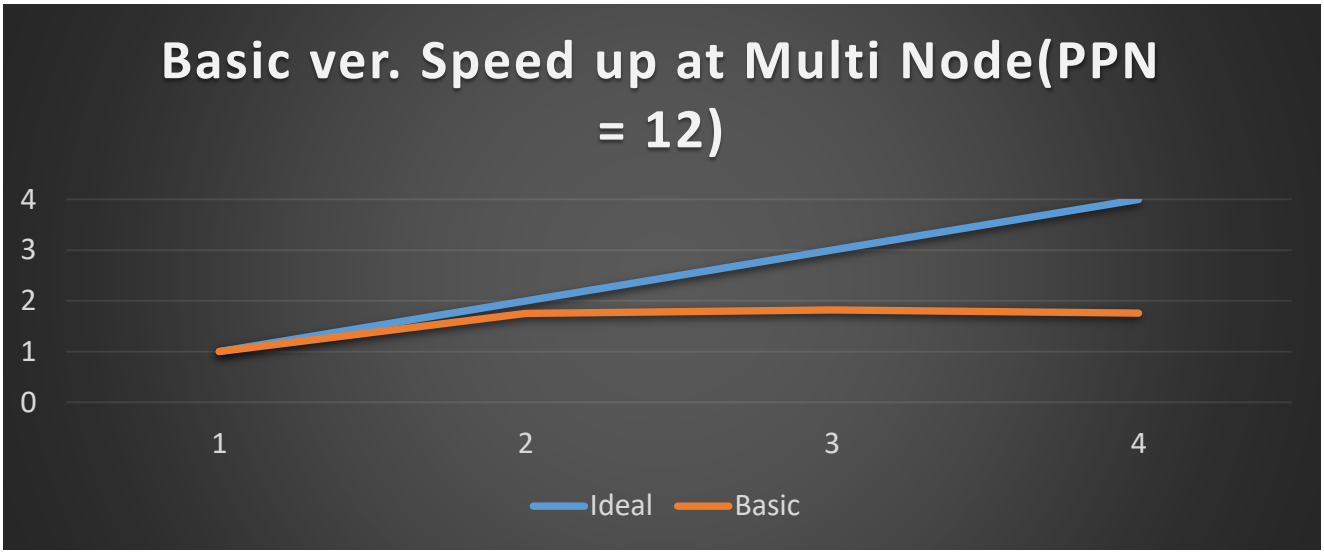
x-axis→# of process (PPN)　　y-axis→time(ms).



x-axis→# of Node y-axis→time(ms).



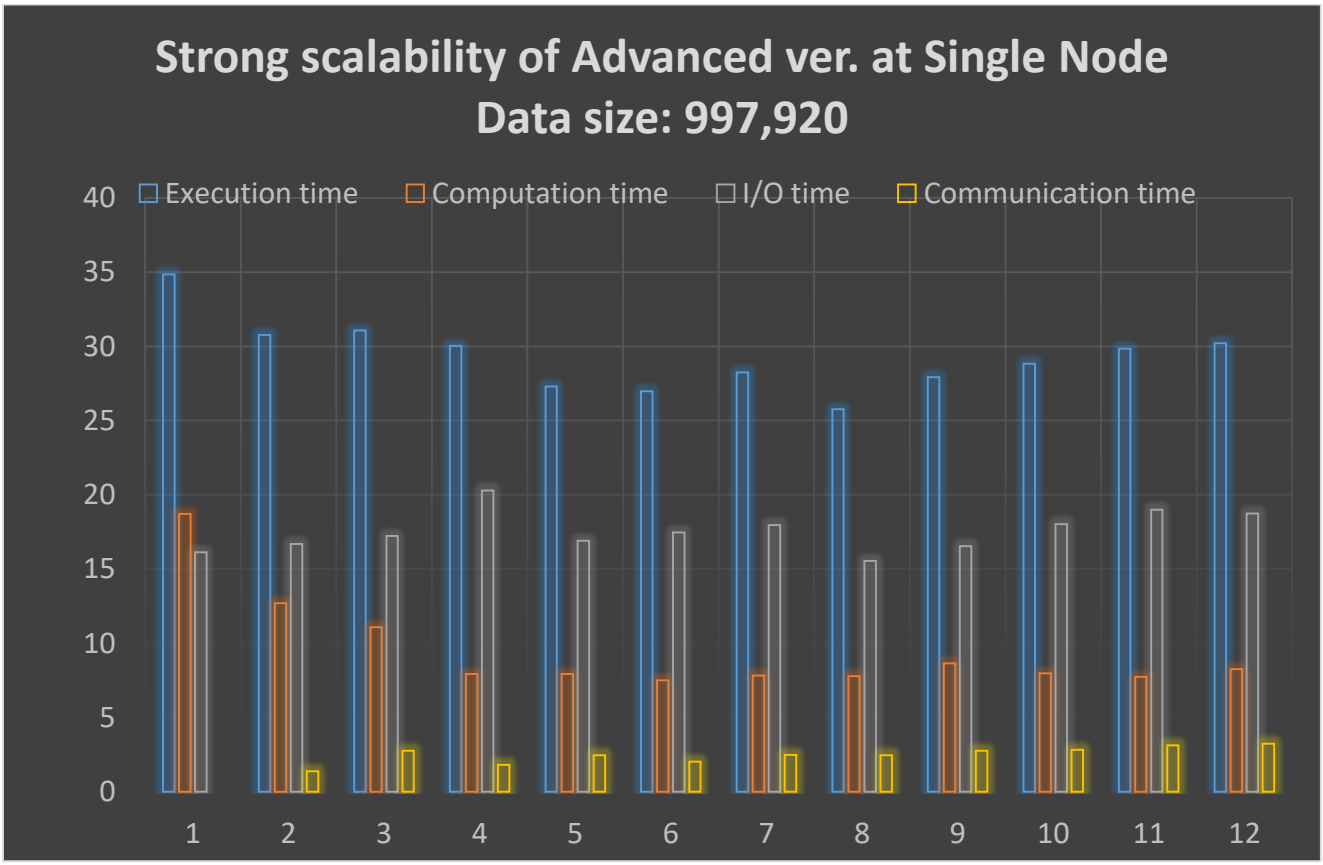
x-axis→# of process (PPN) y-axis→Speedup Factor.



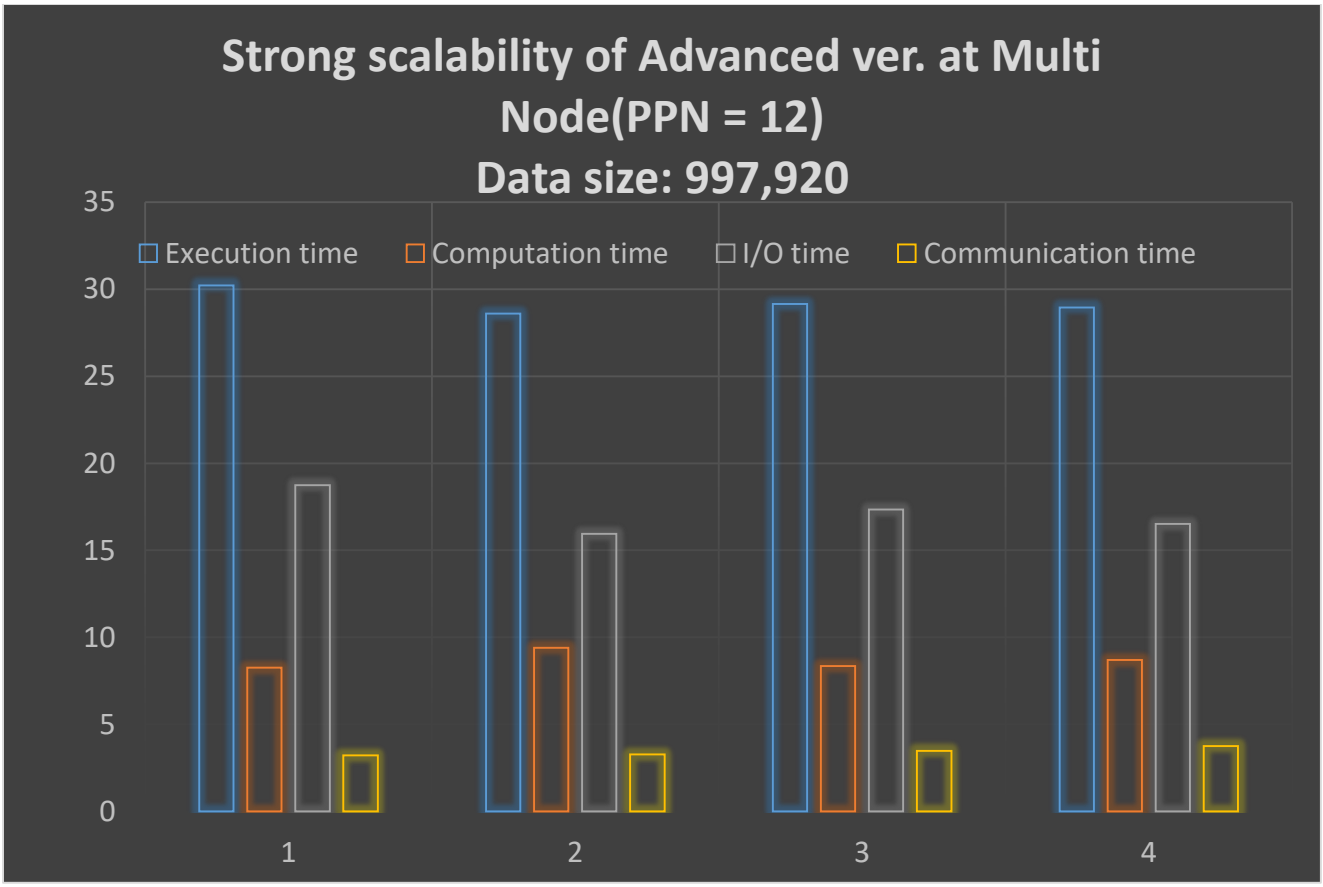
x-axis→# of Node　　y-axis→Speedup Factor.

用 Speedup factor 圖來看，雖然有些不穩定的狀態（測了幾次發現時間跳動幅度滿大的），但還是不難看出，無論 Process 提升或 Node 數提升，加速效果基本上都是顯而易見。

接著再來看 Advance 版本的測試



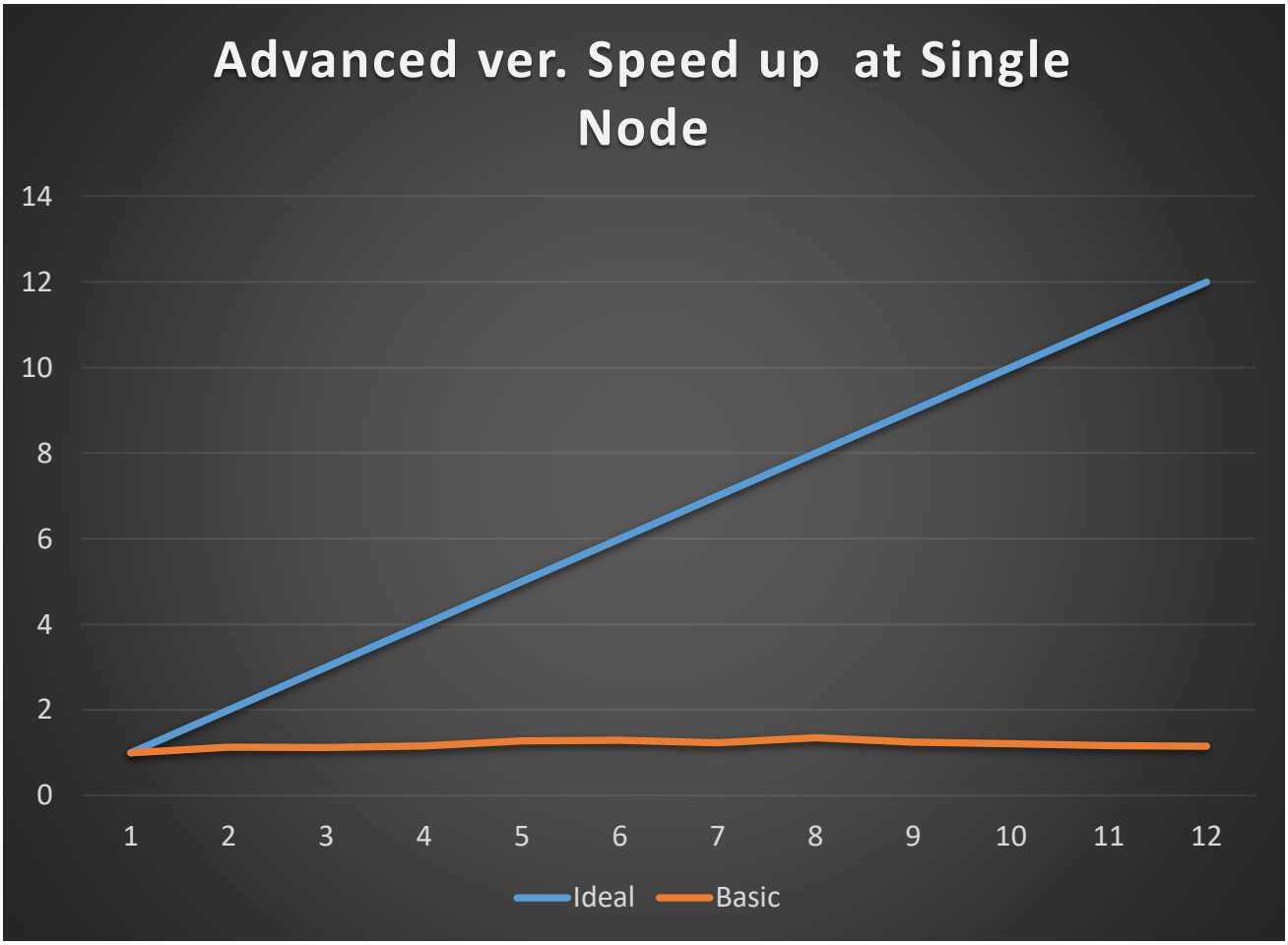
x-axis→# of process (PPN)　　y-axis→time(ms).



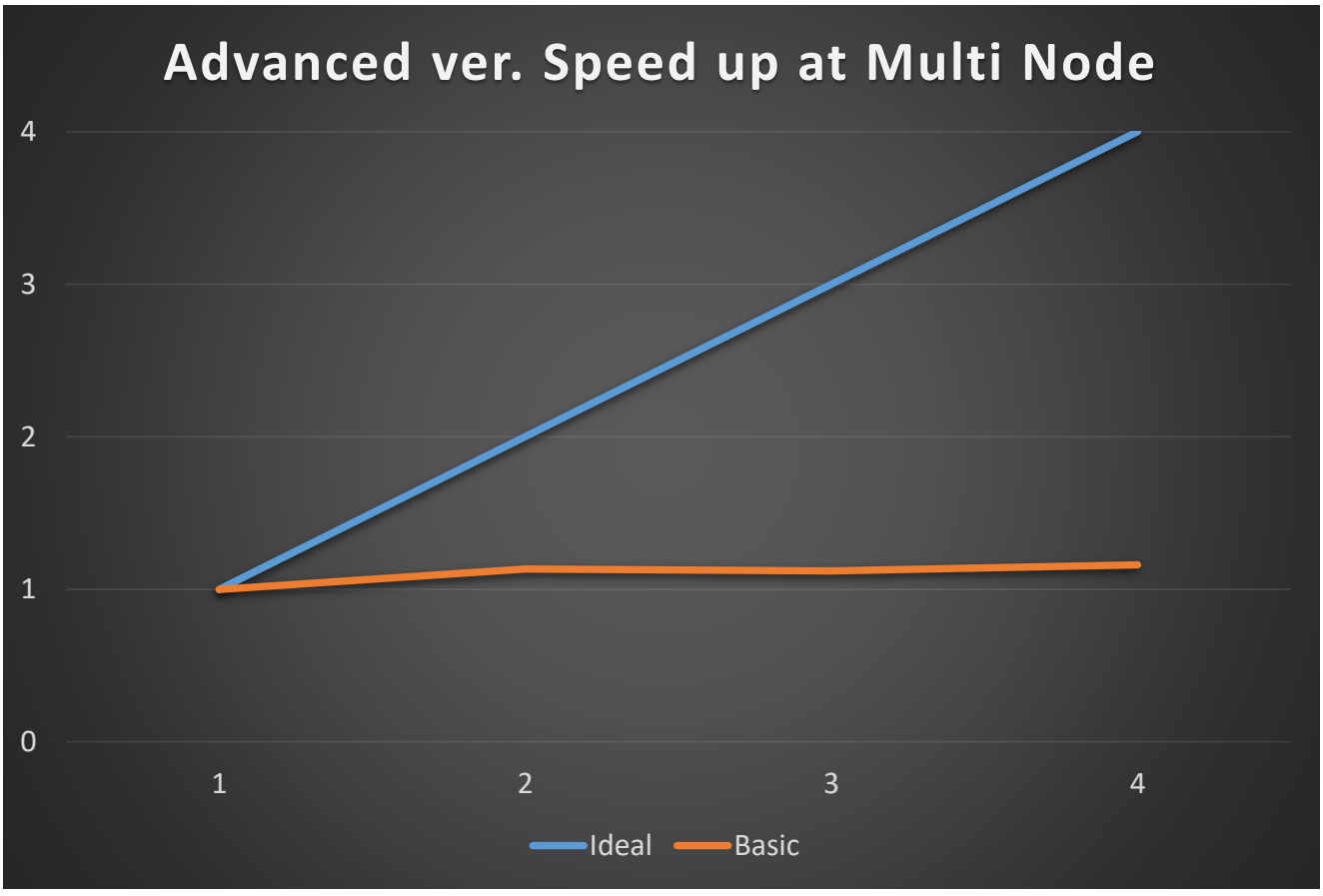
x-axis→# of Node　　y-axis→time(ms).

兩張圖皆能明顯看出 Advanced 版的執行時間與 Basic 相比，大幅縮短。與 Basic 不同的是，Advanced 中的 I/O 與 Communication 逐漸主宰程式的執行時間，第二個發現為，隨著 Process 或 Nodes 增加，Advanced 版本的 Exec.time 並沒有如 Basic 那般顯著的減少。非但沒有減少，有時可能會增加。

下兩張 Speed up 圖顯示，Advanced 版隨著 PPN 或 Node 增加，其 Speed up 並沒有顯著提高，肇因於 I/O 是主要的瓶頸，而這也是與 Basic 很大不同的地方。



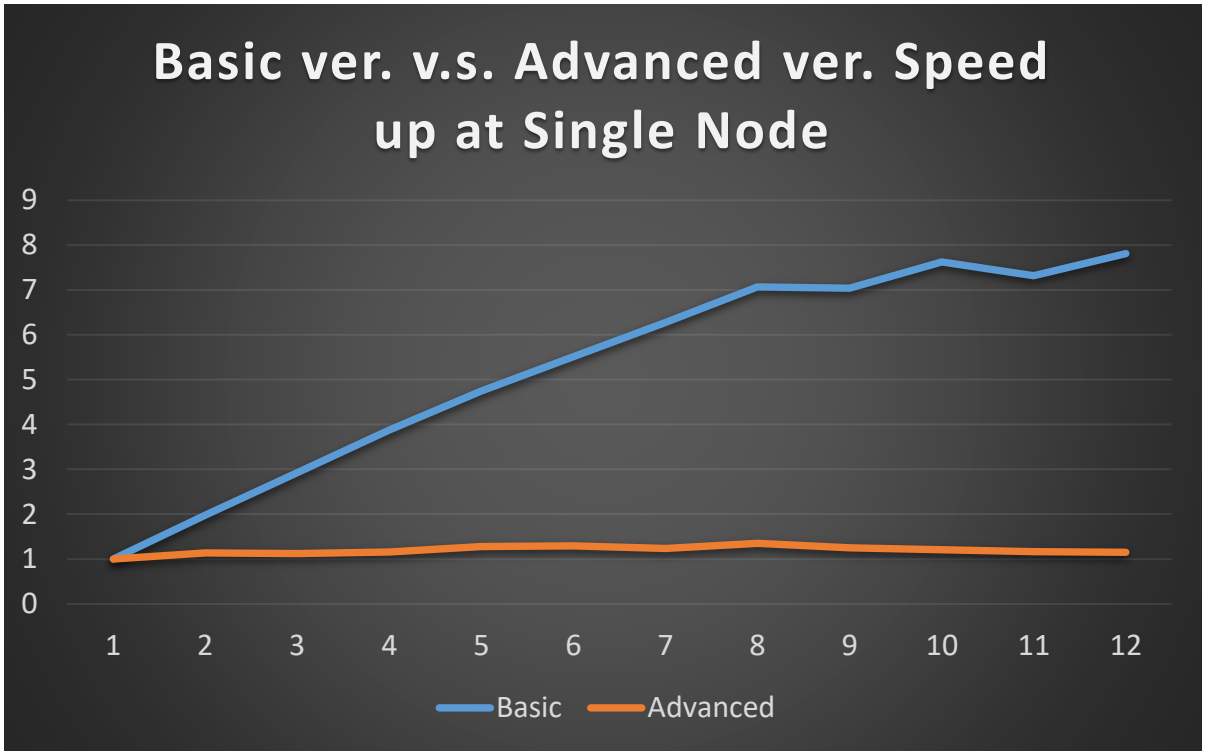
x-axis→# of process (PPN) y-axis→Speedup Factor.



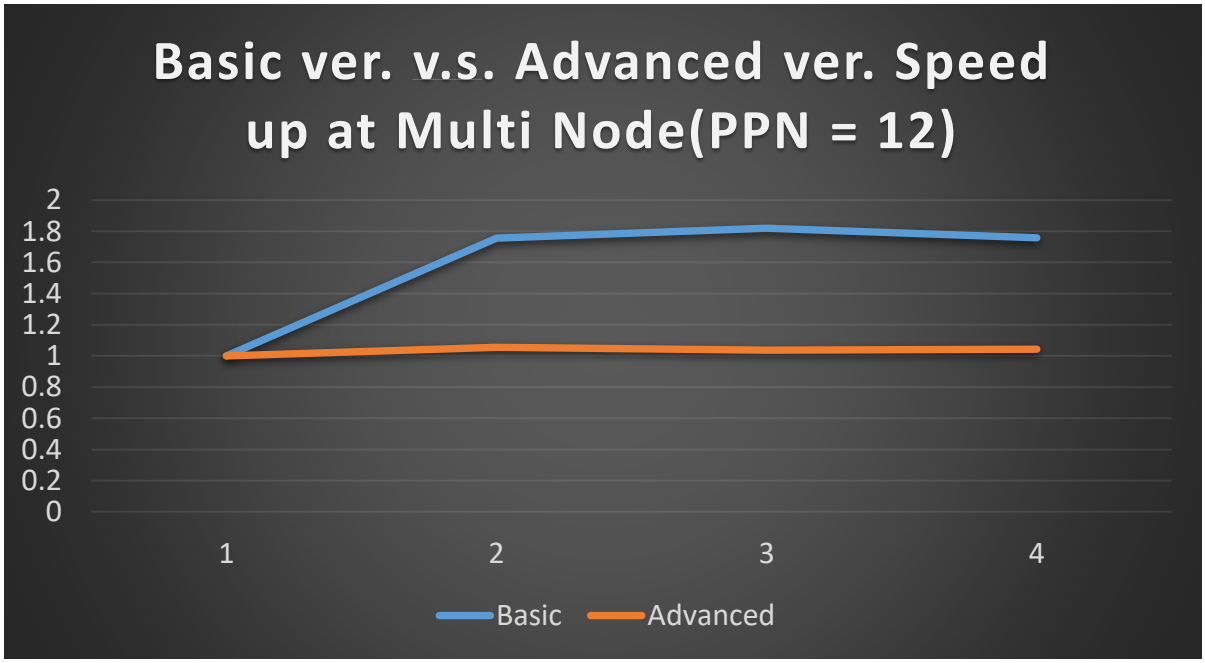
x-axis→# of Node y-axis→Speedup Factor.

- Conclusion

  Basic 版與 Advanced 版相比，Advanced 確能大幅縮短程式執行時間。而若單從版本內部來看，Basic 對於 Process 或 Nodes 數的提升，其執行時間有較顯著的縮短。Advanced 的 I/O 與 Communication，隨著 Process 或 Nodes 數的提升，逐漸主宰了程式的執行時間，尤以 I/O time 為最。這也導致了不論是在 Single Node 或 Multi Node，Advanced 的 Speedup factor 增加的情形不如 Basic 版明顯。



x-axis→Version　　y-axis→Speedup Factor.



x-axis→Version　　y-axis→Speedup Factor.