Homework 1: Odd-Even Sort

107062612 熊祖玲

I. Implementation

這次作業為odd-even sort的實作，在odd-even sort進行時，主要是分成兩種階段：odd phase和even phase，並且在每一個iteration交互運作，直到所有資料完成排序為止，以下分別對basic和advanced版本作詳細地介紹。

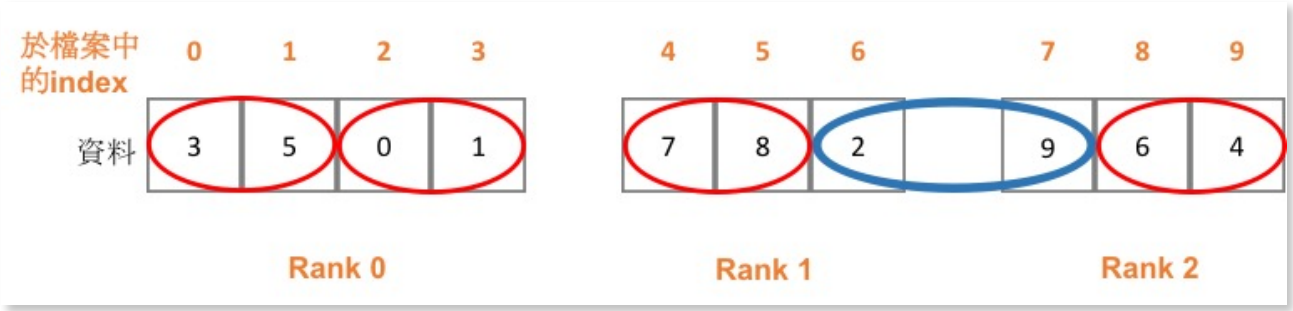
A. Basic

1. 根據程式接收到的第一個參數N（資料量）決定每個task需要處理的資料量。藉由MPI函式庫所提供的MPI_Comm_size(...)函式取得被建立的task數量T，則每個task必須處理N/T個資料，此時將會遇到兩種問題：
   a) N小於T
      當此狀況發生時，表示有些task(s)會沒有被分到資料而空等，因此我選擇透過MPI_Group_range_incl(...)和MPI_Comm_create(...)函式將需要執行運算的task(s)額外建立新的group和communicator，並且透過MPI_Finalize()將多餘的task(s)終止。
   b) N無法被T整除
      在此狀況之下，表示資料無法被平均分配，我想到的作法有兩種。第一，將剩餘的資料全部分配給rank為T-1的task，但遇到T的值很大的時候，task T-1需要處理的資料量將會是其它task(s)的數倍，最後可能成為程式的瓶頸。第二種作法是將剩餘的資料再平均分配給前N mod T個task(s)，因此只有前幾個task(s)會比其它的task(s)多一個資料，將會比第一種作法更加平均。因此我選擇使用第二種作法實作。
2. 決定好每個task需要處理的資料量後，便開始讀取資料，由於每個task各自處理所負責的資料，所以我選用MPI_File_read_at(...)函式執行，可以各自讀取需要的資料即可，不會讀取多餘的資料再捨去。除此之外，我在每個task中紀錄其第一個和最後一個資料分別在檔案中的index，作為sorting時分辨屬於odd或even的依據。
3. 接下來開始進行odd-even sort，首先必須注意：在basic版本當中所有資料僅能與相鄰的兩個資料進行交換。以下分成兩種階段討論：
   a) Even phase
      在even phase中，若原本於檔案中的index為偶數者必須與其index+1者進行比較及交換，如下圖一所示，rank 0中的資料3在檔案中的index為0，因此其必須與rank 0中的資料4在檔案中的index為1進行比較與交換，以此類推。當相鄰的兩筆資料屬於不同的task時，如圖一中index 6、7的資料，則必須透過message passing的技術交換資料，而使用non-blocking及blocking的message passing作法略有不同，以下逐一討論：
      (1) Blocking message passing
          若使用此技術，在傳遞訊息時，傳送方會等到接收方收到訊息後才會繼續做接下來的指令，反之，接收方完成先前的指令時，也需等收到傳送方的訊息後才會停止等待繼續完成接下來的指令。因此，我先將每個task中需進行message passing的資料忽略，其餘資料兩兩進行比較與交換，如圖

一，rank 1先執行資料7、8的比較與交換，rank 2先執行資料6、4的比較與交換。完成後再開始進行message passing，rank r將其最後一個index為偶數的資料m傳送至rank r+1，rank r+1將其第一個index為奇數的資料n傳送至rank r（註：若rank r為最後一個task，表示rank r+1不存在，則不允許進行message passing），rank r將m, n中較小者保留，rank r+1將m, n中較大者保留，即完成資料傳遞及比較交換。然而此技術可能會造成等待時間太長，影響程式的效能，但最後我還是使用這種方法，主要是能夠避免在撰寫上出現判斷錯誤，造成程式無限等待的問題。

(2) Non-blocking message passing

若使用此技術，在傳遞訊息時，傳送方一旦送出訊息後，就直接執行後續的指令，接收方也可以先執行其他運算再接收資料。因此，我會讓rank r, r+1（註：若rank r為最後一個task，表示rank r+1不存在，則不允許進行message passing）先將欲傳送的資料m, n送出後，開始進行task中所有資料even phase運作，最後再接收彼此的資料，rank r將m, n中較小者保留，rank r+1將m, n中較大者保留，即完成資料傳遞及比較交換。雖然接收方可能也需要等待資料到達，但只要傳送方提早傳遞資料時，還是可以節省等待時間。有些task(s)可能會需要進行頭尾兩個方向的資料傳遞，呼叫函式MPI_Isend(...)和MPI_Irecv(...)時，必須小心參數Tag的填寫，否則接收方可能會因為讀不到資料而進行無限地等待。



圖一、EVEN PHASE示意圖

b) Odd phase

在odd phase中，若原本於檔案中的index為奇數者必須與其index+1者進行比較及交換，如下圖二所示，rank 0中的資料5在檔案中的index為1，因此其必須與rank 0中的資料0在檔案中的index為2進行比較與交換，以此類推。當相鄰的兩筆資料屬於不同的task時，如圖二中index 3、4和6、7的資料，則必須透過message passing的技術交換資料，而使用non-blocking及blocking的message passing作法略有不同：

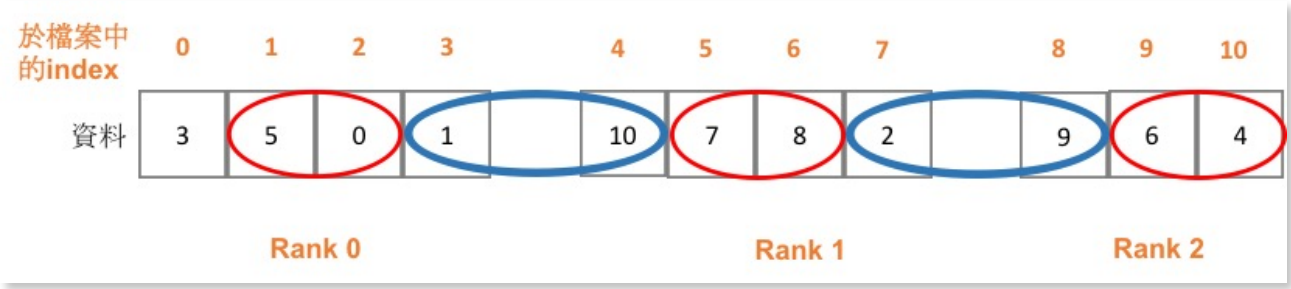
(1) Blocking message passing

若使用此技術，我先將每個task中需進行message passing的資料忽略，其餘資料兩兩進行比較與交換，如圖二，rank 1先執行資料7、8的比較與交換，rank 2先執行資料6、4的比較與交換。完成後再開始進行message passing，rank r先將其最後一個index為奇數的資料m傳送至rank r+1，rank r+1將其第一個index為偶數的資料n傳送至rank r（註：若rank r為最後一個task，表示rank r+1不存在，則不允許進行message passing），

rank r將m, n中較小者保留，rank r+1將m, n中較大者保留，即完成資料傳遞及比較交換。然而此技術可能會造成等待時間太長，影響程式的效能。

(2) Non-blocking message passing

若使用此技術，在傳遞訊息時，傳送方一旦送出訊息後，就直接執行後續的指令，接收方也可以先執行其他運算再接收資料。因此，我會讓rank r, r+1（註：若rank r為最後一個task，表示rank r+1不存在，則不允許進行message passing）先將欲傳送的資料m, n送出後，開始進行task中所有資料even phase運作，最後再接收彼此的資料，rank r將m, n中較小者保留，rank r+1將m, n中較大者保留，即完成資料傳遞及比較交換。有些task(s)可能會需要進行頭尾兩個方向的資料傳遞（如圖二藍色部分，rank 1必須與rank 0, 2交換資料），所以在呼叫函式MPI_Isend(...)和MPI_Irecv(...)時，必須小心參數Tag的填寫，否則接收方可能會因為讀不到資料而進行無限地等待。



圖二、ODD PHASE示意圖

4. 在每次iteration中，我會設定一個變數C負責紀錄從其它task傳來的資料是否有被留下來，並在每次iteration結束前收集所有tasks的C進行or運算後更新到所有tasks，若更新的結果為1則表示有task(s) 透過資料傳遞而改變其內部資料，則需要再進行下一個iteration，若更新的結果為0則表示所有tasks都沒有透過資料傳遞而改變其內部資料，則排序已完成。
5. 當排序完成之後，每個task將已排序的資料根據原本於檔案中的index 透過MPI_File_write_at(...) 寫入檔案的相對位置，而用此方法可以避免將所有資料傳到某個task再一次寫入的時間。

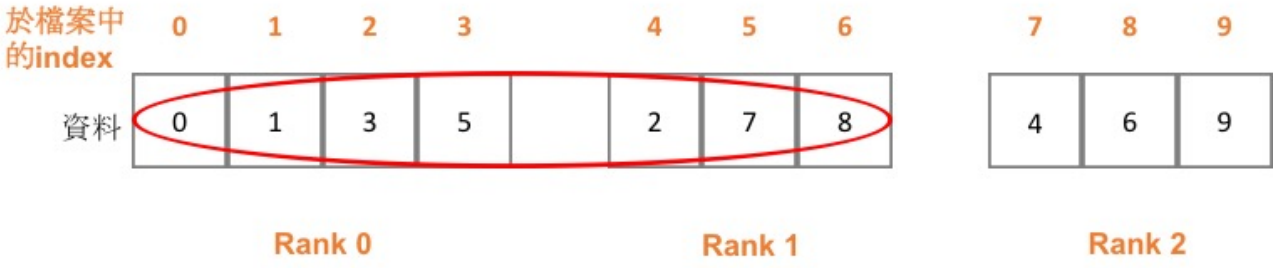
B. Advanced

1. 根據程式接收到的第一個參數N（資料量）決定每個task需要處理的資料量。實作方法與basic版本相同。
2. 決定好每個task需要處理的資料量後，便開始讀取資料，由於每個task各自處理所負責的資料，方法與basic相同，但我另外紀錄前後一個tasks的資料量以方便後續傳遞訊息的處理。
3. 接下來開始進行odd-even sort，因為少了basic版本的限制，所以我會將每個task中的資料先做quicksort後，再於odd phase與even phase的將兩個tasks的資料進行merge。以下分別討論兩階段的實作方法：

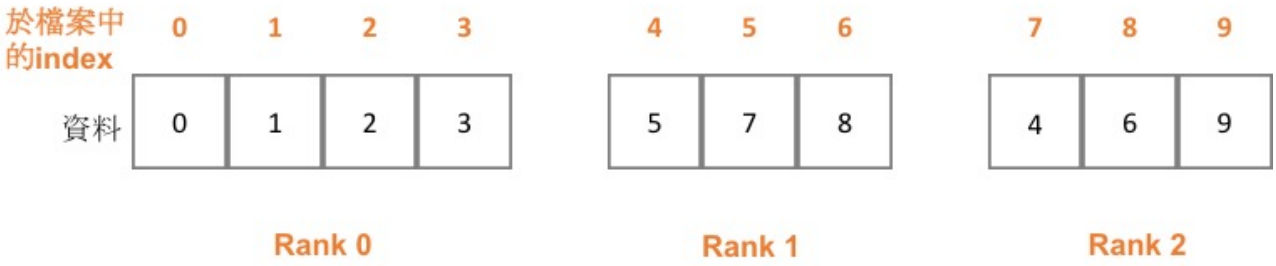
   a) Even phase

   因為一開始已經將資料分別排序，所以可直接進行資料交換，若rank r為偶數者必須將其所有的資料傳送至rank r+1，且rank r+1必須將其所有的資料傳送

至rank r（註：若rank r為最後一個task，表示rank r+1不存在，則不允許進行message passing），然而每個task的資料量有所不同，因此需要紀錄前後一個task的資料量作為傳送、接收的資料量之依據，接收到對方的資料後必須進行merge。假設rank r, r+1的資料量分別為m, n，我會在rank r中建立一個buffer next_buf[n] 用來接收rank r+1傳入的n筆資料，並在rank r+1中亦建立一個buffer pre_buf[m] 用來接收rank r傳入的m筆資料，接著rank r將其所有之m筆資料與next_buf中之n筆資料進行m次的merge，即可得到m+n個資料中前m小的資料，而其它的資料直接捨去即可；反之，rank r+1將其所有之n筆資料與pre_buf中之m筆資料進行n次的merge即可得到後n大的資料。以下圖三紅色圓圈所示，rank 0將其4筆資料傳給rank 1，rank 1將其3筆資料傳給rank 0，再進行merge，即可得到圖四的結果。



圖三、EVEN PHASE 資料交換前示意圖



圖四、EVEN PHASE 資料完成交換及MERGE示意圖

b) Odd phase
其作法與even phase相同，不同處為rank r為偶數者必須將其所有的資料傳送至rank r-1，且rank r-1必須將其所有的資料傳送至rank r（註：若rank r為第一個task，表示rank r-1不存在，則不允許進行message passing），接著再進行merge即可。

4. 在每次iteration中，我會設定一個變數C負責紀錄從其它task傳來的資料是否有被留下來，並在每次iteration結束前收集所有tasks的C進行or運算後更新到所有tasks，若更新的結果為1則表示有task(s) 透過資料傳遞而改變其內部資料，則需要再進行下一個iteration，若更新的結果為0則表示所有tasks都沒有透過資料傳遞而改變其內部資料，則排序已完成。
5. 當排序完成之後，每個task將已排序的資料根據原本於檔案中的index 透過MPI_File_write_at(...) 寫入檔案的相對位置，而用此方法可以避免將所有資料傳到某個task再一次寫入的時間。

C. Another version of advanced

1. 在advanced版本中，我另外實作了一個版本，也就是在改變步驟3中的資料交換方法，在上一個版本中，需要做交換的資料為接收方及傳送方的所有資料，假設rank r, r+1的資料量分別為m, n，則每一組溝通需要m+n筆的資料傳遞，然而這會造成程式的loading太重，因此我改成只傳送所有task中最大處理資料量之一半資料，也就是說當資料量D大於task數量T，交換的資料量就是 $\left\lceil \frac{\lceil \frac{D}{T} \rceil}{2} \right\rceil$ ，當資料量D小於等於task數量T，交換的資料量就是1。
2. 在odd phase與even phase中，除了傳遞資料量的減少外作法皆相同，雖然傳送的資料量比較少會造成iteration的增加，但task間彼此的溝通、等待時間可能會減少許多，CPU time的時間也只有小幅的成長。

II. Experiment & Analysis

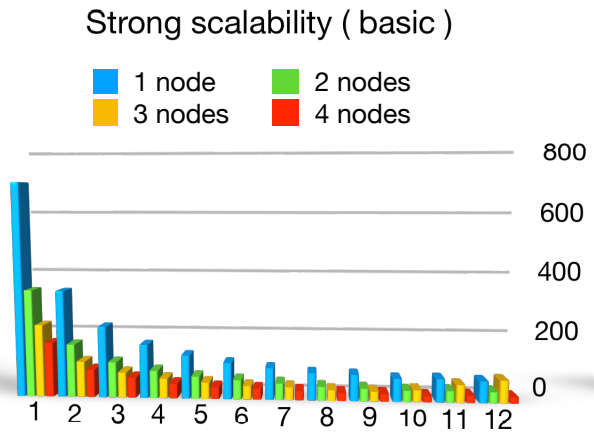
A. Methodology

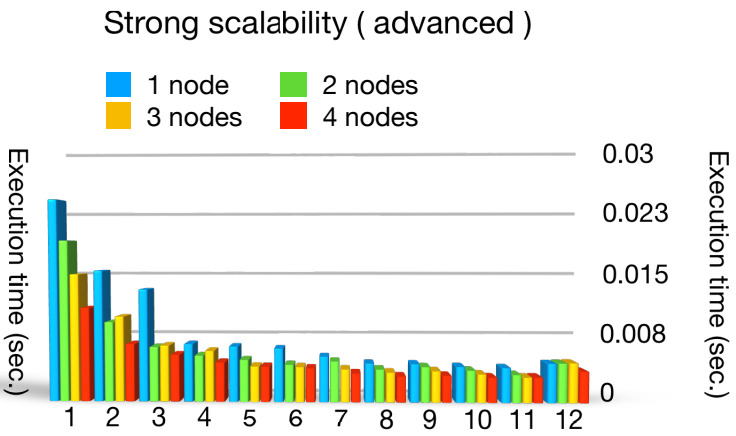
1. System Spec.
   所有程式皆於課程所提供之cluster上測試。
2. Performance Metrics
   a) Computing time
      在完成MPI_Init()之後及做MPI_Finalize()之前加上MPI_Wtime()，再算其差值得到整個程式的執行時間，用此時間扣除IO time與communication time後即為所有花費於計算的時間。
   b) IO time
      在呼叫讀檔、寫檔函式MPI_File_read_at(...)、MPI_File_write_at(…)的前後加上MPI_Wtime()，並算出其差值再加總，則可得到所有讀寫檔所花費的時間。
   c) Communication time
      在呼叫communication相關函式MPI_Sendrecv(...)、MPI_Isend(…).....等的前後加上MPI_Wtime()，並算出其差值再加總，則可得到所有溝通所花費的時間，但每個task所花費的溝通時間可能會有很大的差異，因此將所有task的溝通時間透過MPI_Allreduce(...)進行加總再做平均，作為其communication time。
   d) 其它
      所有的測試資料皆使用我寫的測資產生器產生出的1,000,000筆資料，這些資料由1,000,000開始逐一遞減1，即為1到1,000,000的數字由大到小排序，且此測資為worst case；此外，我會將每個task的所有計算、IO和溝通時間取平均值，因為每個task處理的資料量可能不同，這麼做會比較符合一般狀況。

B. Plots: Speedup Factor & Time Profile

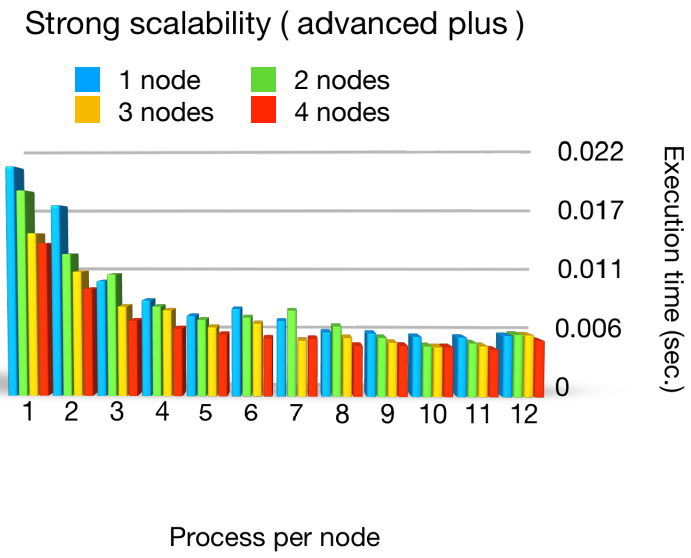
1. Strong scaleability

由下面兩張圖可以得知：無論是basic還是advanced版本中，當process的數量增加，執行時間會隨之減少，若node數量增加，執行時間也會逐漸下降，但下降至一定程度之後會停止，由此可見，將程式平行化後可以帶來的加速效果很可觀。然而，advanced版本中因為不受到限制（每筆資料只能與其相鄰之資料進行交換），所以執行時間較basic版本小很多。



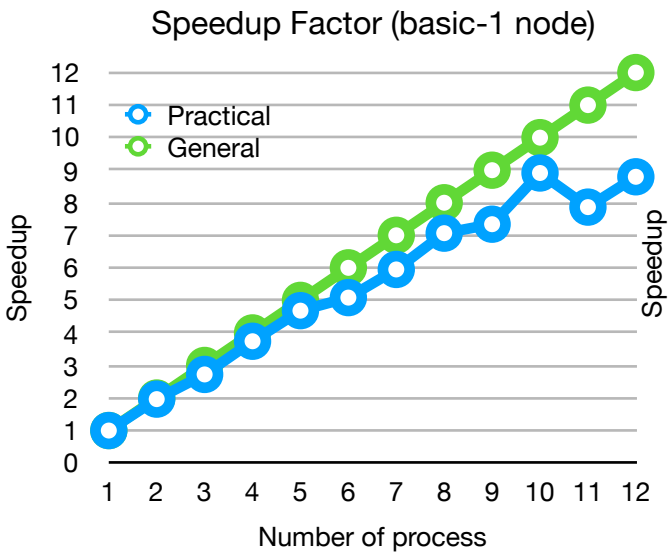


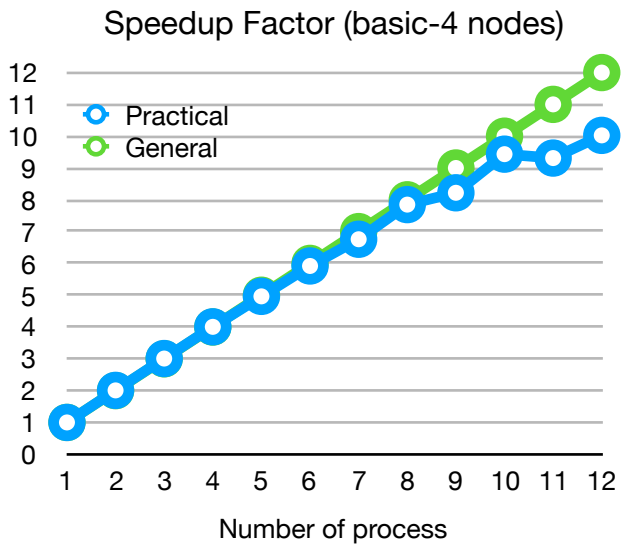
另外，在advanced plus版本中因為所傳遞的資料量為advanced版本中的一半，在進行資料merge時，computing time為原本的advanced版本之0.75倍，所以執行時間與basic版本相比又來的更小。



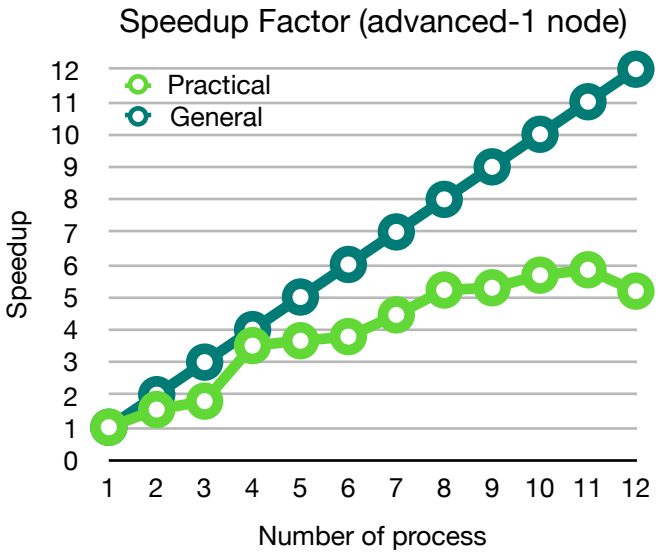
2. Speedup Factor

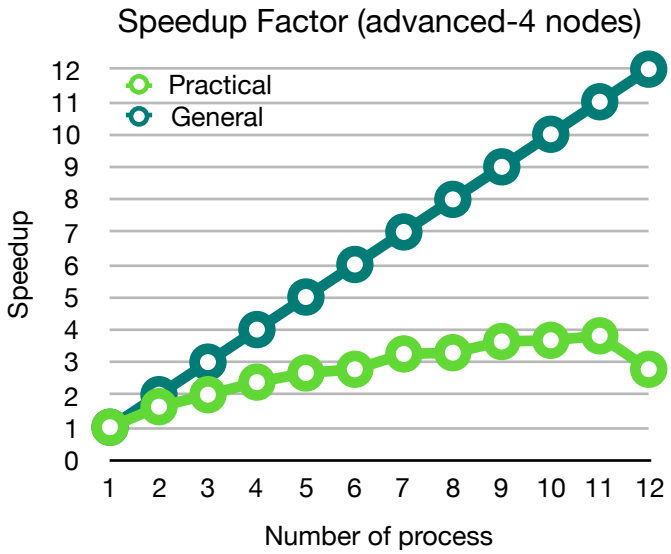
由下面兩張圖可以得知：在basic版本中，speedup會隨著process的數量逐漸上升，但在node為1時，speedup成長到一定數量時會逐漸趨緩；而node為4時，process增加到11時會稍微下降一點，process增加到12時又會再次上升。除此之外，實驗曲線（藍色）在node為4時會與理想曲線（綠色）較相符，因此node為4時加速效果較佳。



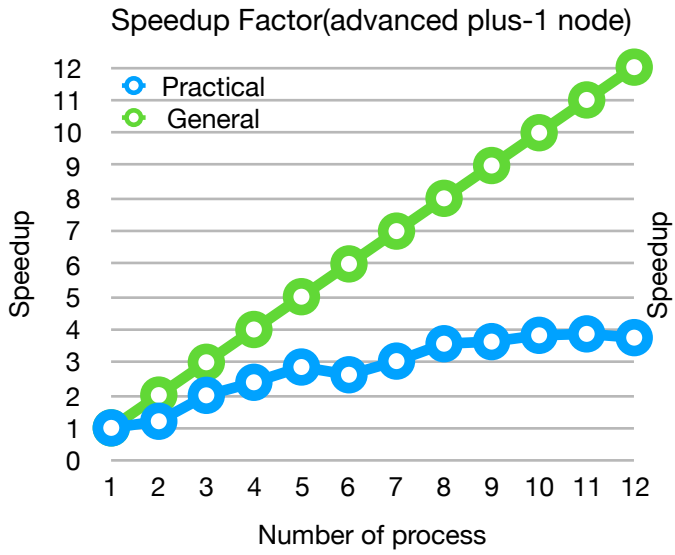


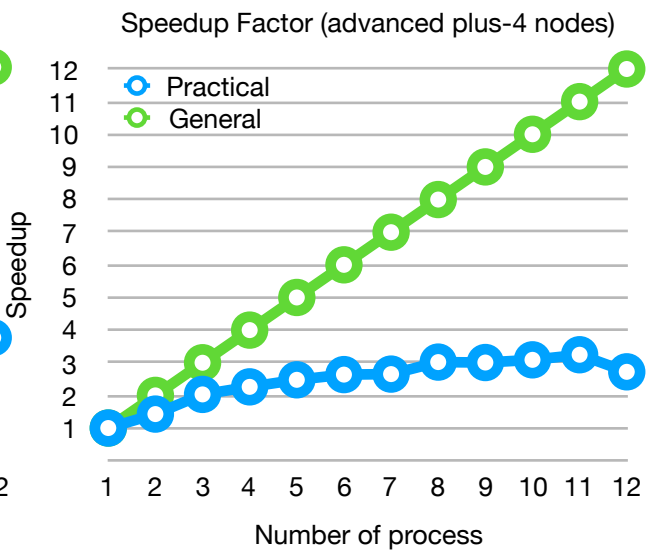
由下面兩張圖可以得知：在advanced版本中，speedup會隨著process的數量逐漸上升，但在node為4時，speedup成長到一定數量時會逐漸趨緩，speedup程度最大只有4；而node為1時，實驗曲線（淺綠色）在與理想曲線（深綠色）較相符，也就是說node為1時，加速效果較佳。這個現象和理想現象有一段差距，當node數量增加時，speedup效果理應會更好，但實驗結果卻與之相反，我認為可能是因為需要平行的計算量已經達到極限，因此就算增加process的數量也無法繼續提升效能。



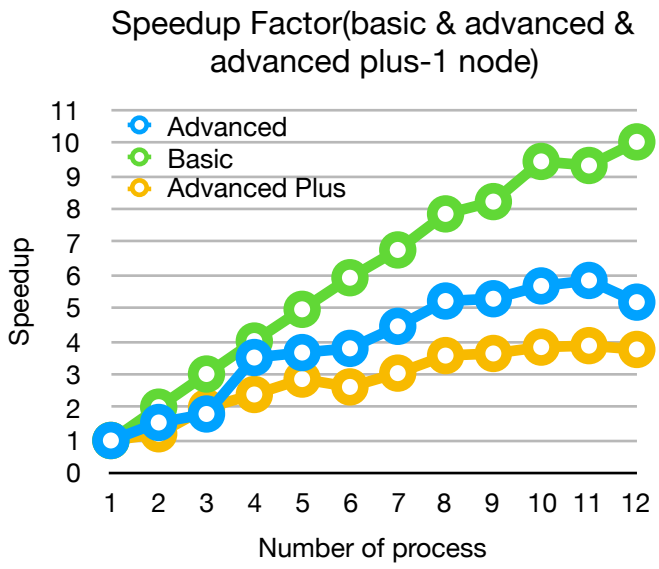


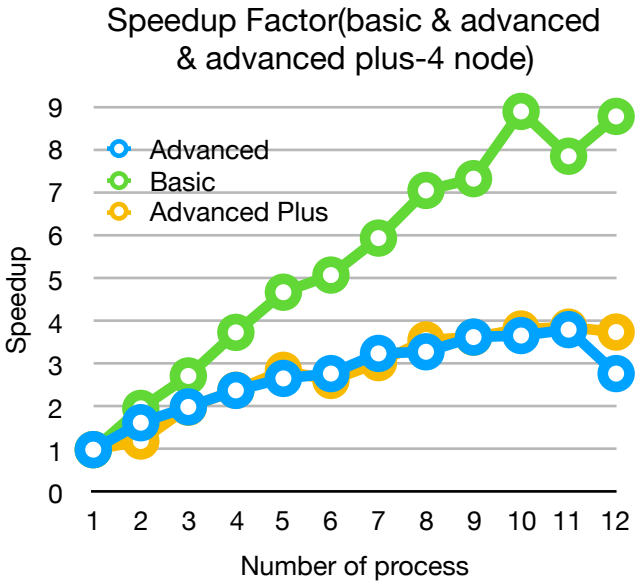
由下面兩張圖可以得知：在advanced plus版本的speedup曲線與advanced版本相似，亦會隨著process的數量逐漸上升，但在node為4時，speedup成長到一定數量時會逐漸趨緩，speedup程度最大只有4。





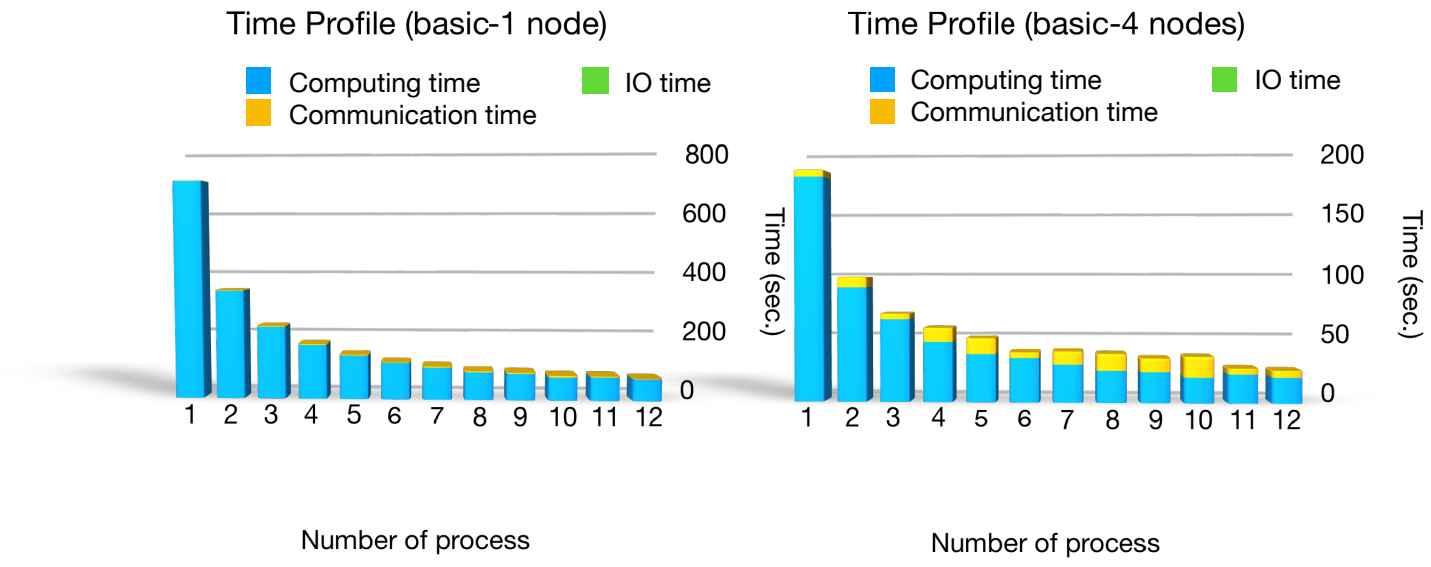
接著，我分別將1 node和4 nodes的basic、advanced與advanced plus版本放在一起做比較，此時發現：在basic的版本當中speedup的效果最好，接著是advanced，最後才是advanced plus，這表示我的advanced與advanced plus版本在效能上沒有比較好的表現。



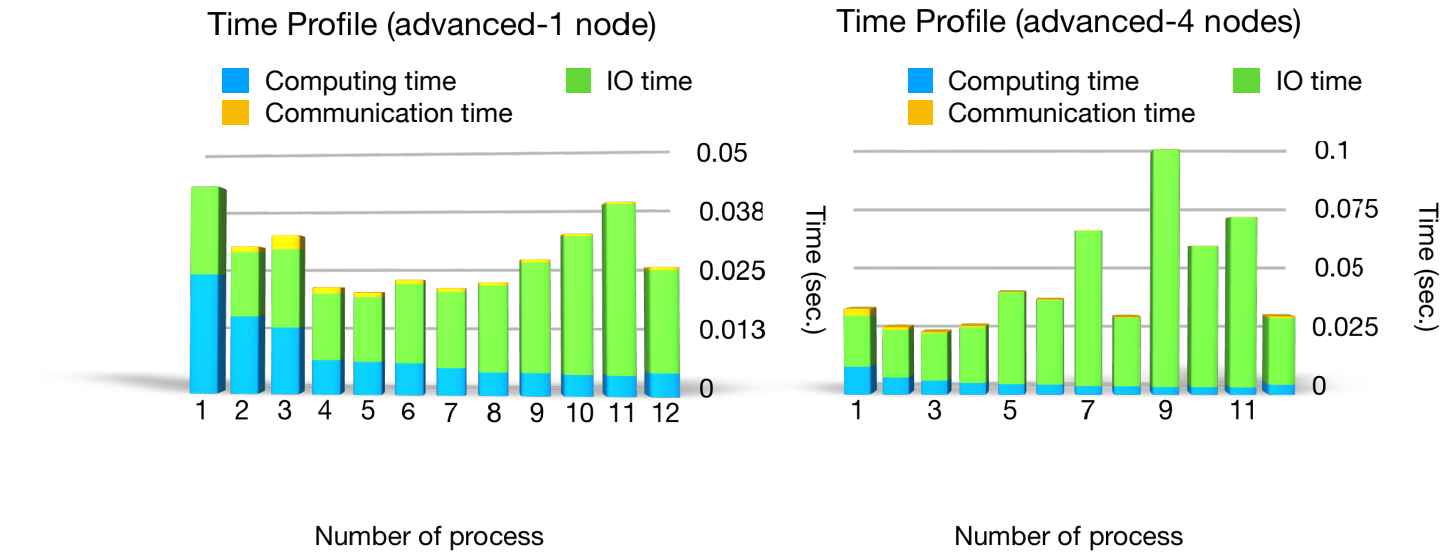


3. Time Profile

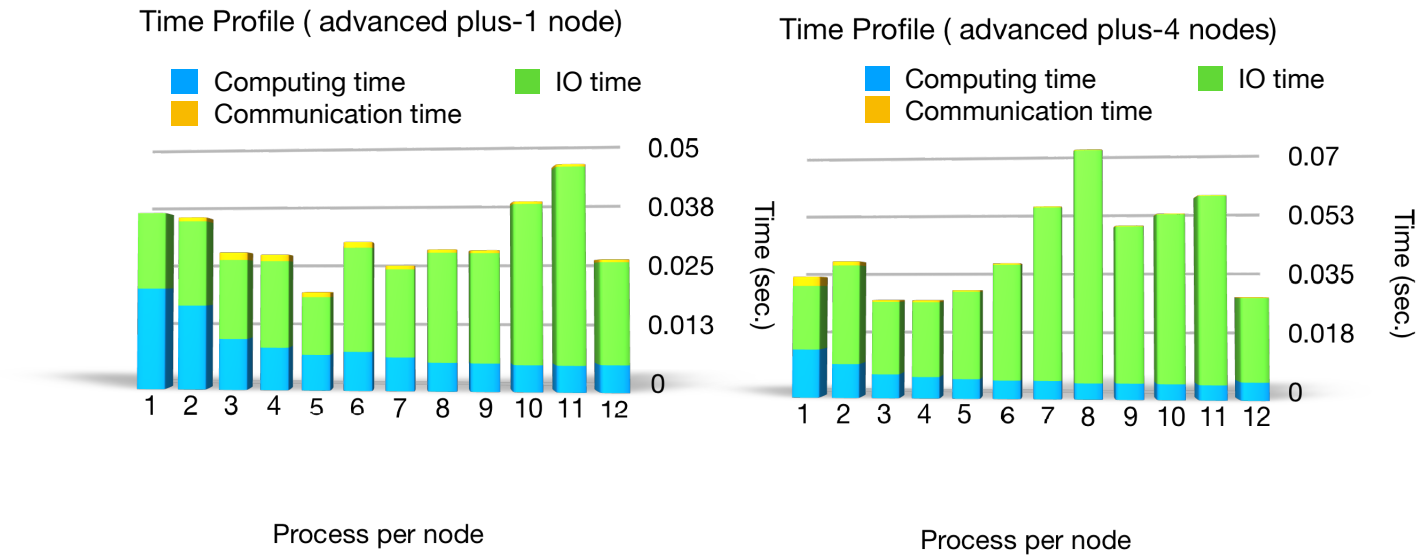
由下面兩張圖可得知：CPU time會隨著process數量增加而遞減。但在右圖當中發現：在4個node的情況下，溝通時間會隨著process數量增加而遞增，然而這是因為process的數量增加而造成process之間需要透過溝通交換資訊的時機增加，才會出現這種現象。



接著由下左圖可得知：在只有一個node時，CPU time會隨著process數量增加而遞減，而I/O time會隨之遞增，溝通時間則也會隨之增加。
由下右圖可得知：在4個node的情況下，CPU time會隨著process數量增加而遞減，而溝通時間會隨之增加，但I/O time卻會大幅增加並且遠大於CPU time。



接著由下兩圖可得知：其computing time、I/O time和communication time之間的比例與原本的advanced版本極為相似，I/O time仍然為bottleneck，但整體時間來看是有更進一步的加速。



C. Discussion

經過實驗數據可以發現：advanced版本的實作speedup並不如預期的好，甚至比basic的版本來得差，但就整體時間來說，advanced版本的效能的確是非常好的。

1. Basic版本

   若為sequential的odd even sort，其時間複雜度為$O(N^2)$，而平行化之basic版本因為是由多個tasks T平行處理資料排序，其時間複雜度為$O(\frac{N^2}{T})$，又因為我有加入判斷式紀錄每個iteration是否有資料的交換，若無則完成排序，所以若非遇到worst case，效能可能會更好。然而，I/O time對basic版本是比較小的影響因素，其loading主要是兩兩資料間比較且swap。在network performance的角度來說，因為每次傳遞的訊息為一筆資料，所以對程式的負擔也不至於太大。

2. Advanced版本

   在advanced版本中，會先將分配到的資料進行quicksort，接下來才開始做odd even的merge，quicksort時間複雜度為$O(NlgN)$，而平行化後是由多個tasks T平行處理則為$O(\left\lceil\frac{N}{T}\right\rceil lg\left\lceil\frac{N}{T}\right\rceil)$，因為是由多個task T平行處理資料排序；而每個iteration所進行的資料merge，其時間複雜度為$O(2N)$，因此整個程式被quicksort所限制，最後的時間複雜度為$O(\left\lceil\frac{N}{T}\right\rceil lg\left\lceil\frac{N}{T}\right\rceil + 2N)$，但我有加入判斷式紀錄每個iteration是否有資料的交換，若無交換則提早完成排序，因此若非遇到worst case，效能可能會更好。然而，I/O time對Advanced版本是比較大的影響因素，畢竟開檔、讀檔及關檔所耗費的時間本來就會比較大。在network performance的角度來說，因為每次傳遞的資料量都滿大的，所以對程式的負擔也不會太小。

3. Advanced改良版本

在advanced改良版本中，也會先將分配到的資料進行quicksort，接下來才開始做odd even的merge，quicksort時間複雜度為$O(NlgN)$，而平行化後是由多個tasks T平行處理則為$O(\left\lceil \frac{N}{T} \right\rceil lg \left\lceil \frac{N}{T} \right\rceil)$，因為是由多個task T平行處理資料排序；而每個iteration所進行的資料merge，其時間複雜度為$O(\frac{3N}{2})$，因此整個程式被quicksort所限制，最後的時間複雜度為$O(\left\lceil \frac{N}{T} \right\rceil lg \left\lceil \frac{N}{T} \right\rceil + \frac{3N}{2})$，但我有加入判斷式紀錄每個iteration是否有資料的交換，若無交換則提早完成排序，因此若非遇到worst case，效能可能會更好。除此之外，若資料量N很大時，效能與advanced版本會有更明顯的差異。然而，I/O time對Advanced改良版本是比較大的影響因素。在network performance的角度來說，雖然每次傳遞的資料量也很大，但較原本的advanced版本少了一半，所以會有比較好的效果。

從strong scalability圖可以發現，basic版本的結果較advanced佳，因爲process的數量增加可以幫助basic版本改善資料的兩兩比較及交換，而advanced的版本中，因為其瓶頸在於I/O time，就算增加process的數量也無法有效地改善，不過從圖中可以知道執行時間依然有明顯地改善，因此我認為我的program scale都滿好的。

III. Experiences / Conclusion

第一次處理平行化的程式遇到最大的困難莫過於改變撰寫程式的思維，必須謹慎分配每個task需要負責的工作，在進行每個task之間的溝通也需要非常小心，我在實作中曾經因為判斷式寫錯，而發生有task送出資料卻沒有task接收的狀況，執行程式時無限等待，最後等待時間超過上限被強制終止，一時間也沒有想到是這裡出錯，在debug時花了很多時間解決；除此之外，在計算buffer的offset時，也常常不小心或是沒有想清楚，而造成segmentation fault或取到其他記憶體位置的值，而花很多時間處理。不過還好有實驗室的同學們互相討論，這次作業才可以順利完成。