Parallel Programming

Homework 1 Report: Odd-Even Sort

108062605 呂宸漢

A. Implementation

1. 呼叫 MPI_Comm_size(…)取得當前建立的 process 數(T)，並讀取引數 argv[1] 取得 input data 數(N)，利用兩者決定如何分配及處理資料。
   處理任意數量的 input data 及 processes 可以分為三種情況：
   i. N < T：
      一開始我用 rank 判斷該 process 是否有被使用，若沒有被使用則呼叫 MPI_Finalize()中止 process，可是被中止的 process 似乎仍包含在 MPI_COMM_WORLD 的 communication group 中，導致溝通錯誤。因此，我用 MPI_Comm_group(…)建立 MPI_COMM_WORLD 的 group，並用 MPI_Group_range_incl(…)在 MPI_COMM_WORLD 的 group 中蒐集需要的 process，再搭配 MPI_Comm_create(…)建立一個新的 communication，如此一來沒有在新 communication 中的 process 就會顯示 MPI_COMM_NULL，即使呼叫了 MPI_Finalize()後，也不會影響新的 communication。處理完多餘的 process 後，data 數會等於 process 數，所以一個 process 分配一筆 data 即可。其實當 data 量不多時，只用一個 process 可以更快 sort 完，可是題目表明要遵守 odd-even sort principal，因此仍分配 data 給所有 process。
   ii. N = T：
      一個 process 分配到一筆 data。
   iii. N > T：
      當 T 可以整除 N 時，每個 process 就分配到 $\frac{N}{T}$ 筆 data。若 T 無法整除 N 時，則有兩種方法可以分配 data：第一種方法是前 N − 1 個 process 分配 $\left\lceil \frac{N}{T} \right\rceil$ 筆 data，最後的 process 則接收剩餘的 data，這個方法雖然直觀，可是容易造成 process 間 loading 不平均，最後一個 process 與前面的 process 分配到的 data 數差距可能較大。第二種方法是先分配 $\left\lceil \frac{N}{T} \right\rceil$ 筆 data 給每個 process，再將剩餘的 data 平均分配給前面幾個 process，如此一來 process 間最多只差一筆 data，雖然在處理上比較麻煩，可是可使每個 process 間的 loading 較平均，在平行處理上可以有更好的 performance。

2. 在讀取 input file 時，我採用 MPI_File_read_at(…)讀取 file，只要設定好每個 process 要讀取 file 的 offset 及 data 數，即可讓每個 process 平行讀取檔案，不必一次讀取整個檔案，節省 I/O 的時間。
3. 在 sorting 的部分，我先對每個 process 的 local-array 進行 sort，一開始我採用<algorithm>的 sort，後來發現<boost>有專為 float 設計的 sort，因此改用 boost::sort::spreadsort::float_sort 加速 local-array 排序的過程。
4. 接著便利用 odd-even 的方式進行 merge：
   i. Even phase：
      rank 為 even 的 process 必須和 rank + 1 的 process 互傳 data，在接收到 data 後便與 local-array 進行 merge，rank 為 even 的 process 將自己的 local-array 替換成 merge 前半的 data，rank + 1 的 process 則將自己的 local-array 替換成 merge 後半的 data。rank 為 even 且是最後一個 process，不需要傳遞資料。
   ii. Odd phase：
      rank 為 odd 的 process 必須和 rank + 1 的 process 互傳 data，在接收到 data 後便與 local-array 進行 merge，rank 為 odd 的 process 將自己的 local-array 替換成 merge 前半的 data，rank + 1 的 process 則將自己的 local-array 替換成 merge 後半的 data。第一個 process 與 rank 為 odd 且是最後一個 process，不需要傳遞資料。

在 data 傳遞的方面有 blocking 及 non-blocking 的兩種方法，由於講義上有詳細描述並比較過了，在此便不贅述，我選擇用 MPI_Sendrecv(…)傳遞及接收資料，簡化撰寫及維護的麻煩。

在 merge 的方面有兩種方法：

i. 第一種方法是用<algorithm>的 merge，先將 local-array 與接收到的 data-array merge 成一個 merge-array，再依照 process 的需要複製 merge-array 的前半或後半到自己的 local-array。然而這種方法除了額外 merge 了 local-array 不需要的 data，還需要多做一個迴圈複製 merge-array 到 local-array，增加了許多不需要的時間。
ii. 第二種方法就是自己寫一個 merge，在 merge local-array(稱為 a)與接收到的 data-array(稱為 b)時，我會額外再 new 一個 array(稱為 t)。當 t 為空時，比較 a 及 b 何者較小，當 a<b 時，則不進行交換，反之，則將 a 存入 t 中，再將 b 存入 a；當 t 不為空，則比較 b 及 t 何者較小，當 t<b 時，比較 a 與 t 何者較小，當 t<a 時，則將 t 存入 a，反之，則不進行交換，當 t>b 時，比較 a 與 b 何者較小，當 b<a 時，先將 a 存入 t，再將 b 存入 a，反之，則不進行交換。
    雖然這種方法比較繁瑣，需要很多判斷式輔助，因為 merge 完的結果

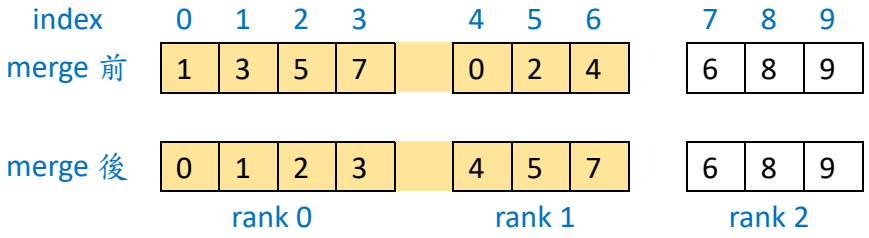
是存在自己的 local-array 裡面，在某些情況下，可以減少 data copy 的次數，無須將原本不需要移動的 data 另外移動到其他陣列，可以加快一些速度。

5. 當 odd phase 與 even phase 皆沒有 data 在 merge 階段交換時，代表 data 已經 sorting 完成，即可進入 data output 的階段。在程式中，我讓 merge 回傳變數紀錄是否有 data 交換，並用 MPI_Allreduce(...)收集所有 merge 的回傳變數，決定是否停止 merge。
6. 與讀取 input file 雷同，在寫入 output file 時，我採用 MPI_File_write_at(...)寫入 file，只要設定好每個 process 要寫入 file 的 offset 及 data 數，即可讓每個 process 平行寫入檔案，不必一次寫入整個檔案，節省 I/O 的時間。

基於上述的架構我寫了兩個版本：

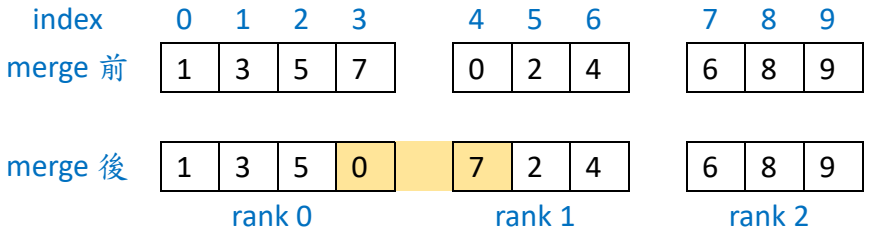
full version：

在 odd phase 及 even phase 傳遞 data 時，傳送完整的 local-array 給對方，雖然這可以完整 merge rank 與 rank + 1 的 local-array，iteration 也比較少，可是每次需要傳送的資料量是 $\left\lceil \frac{N}{T} \right\rceil$ (N 為 data 數，T 為 process 數)，執行時間被網路傳輸時間所限制，也有機會傳送不需要的 data。



half version：

在 odd phase 及 even phase 在傳遞 data 時，只傳送一半的 local-array 給對方，雖然 iteration 會增加，merge 的次數也會增加，不過每次需要傳送的資料量剩下 $\left\lceil \frac{\lceil \frac{N}{T} \rceil}{2} \right\rceil$ (N 為 data 數，T 為 process 數)，可以降低傳送不必要 data 機率，也可以讓網路傳輸時間下降，加快執行速度。



B. Experiment & Analysis

i. Methodology

a. System Spec

所有程式皆在課程所提供的 cluster 上進行測試。

b. Performance Metrics

1. Computing time

在 MPI_Init(…)後與 MPI_Finalize()前加上 MPI_Wtime()，再計算其差值即可得到整個程式的執行時間，用執行時間減掉 communication time 及 IO time 剩下來的時間就是 computing time。

2. Communication time

在 communication 相關函式，如：MPI_Send(…)、MPI_Isend(…)、MPI_Receive(…)、MPI_Ireceive(…)或 MPI_Sendrecv(…)的前後加上 MPI_Wtime()，在取得其差值後再加總，即可得到該 process 用在 communication 上所花費的時間。然而，不同的 process 會有不同的 communication time，所以還要用 MPI_Allreduce(…)加總所有 process 的 communication time，再除以 process 數，即可得到該程式的平均 communication time。

3. IO time

在 MPI_File_read_at (…)與 MPI_File_write_at(…)的前後加上 MPI_Wtime()，取得其差值後加總，即可得到該 process 讀寫檔案的 IO time。

4. Other

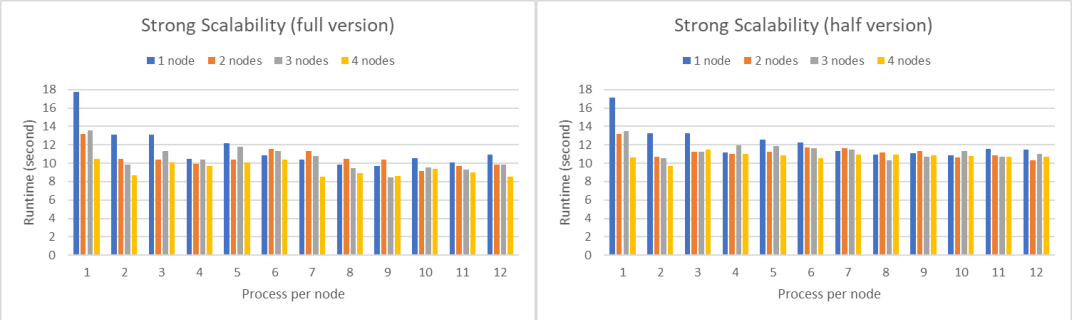
為了使測試結果更準確，我自己產生了一個 worst case 的 test file，file 中包含 536869888 筆浮點數，原本想要用作業 spec 上寫的 536870911 筆 data 做測試，可是只要產生超過 536869888 筆 data，該 file 就無法正確顯示浮點數，因此我只用 536869888 筆 data 做測試，數值的範圍是從 2684349 到-2684349.75，每個數字間相差 0.01 模擬浮點數。

由於每次測試程式時，cluster 的狀況及 runtime 皆不相同，因此我選擇將每種測試的組合都測試 3 次，並將 3 次所得到的時間取平均，降低不同狀態的 cluster 對實驗數據的影響。

ii. Plots: Speedup Factor & Time Profile

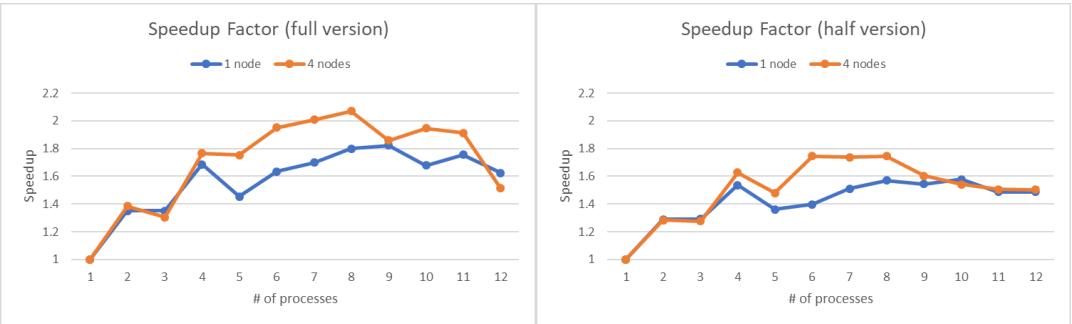
1. Strong Scalability

由下圖可見，full version 及 half version 在不同 process 上所花費的時間曲線大致相同，process 數及 node 數愈多，程式執行時間愈短，但是執行時間下降到某個範圍時就停滯了，沒有隨著 process 數及 node 數線性減少。不過，full version 在 node 數增加時的加速效果較明顯，half version 則在 process 數變多後，node 數增加就不影響執行時間了。我認為 full version 會加速較多是因為他的 iteration 較少，即使是 worst case，最多只需要比 process 數再多一點的 iteration 就可以完成了，而 half version 則是即使每個 iteration 要處理的 data 變少，因為 iteration 變成非常多，導致時間無法下降。



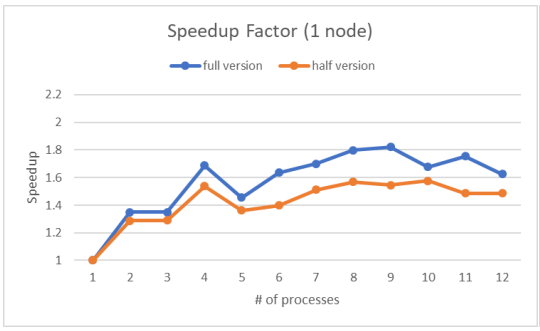
2. Speedup Factor

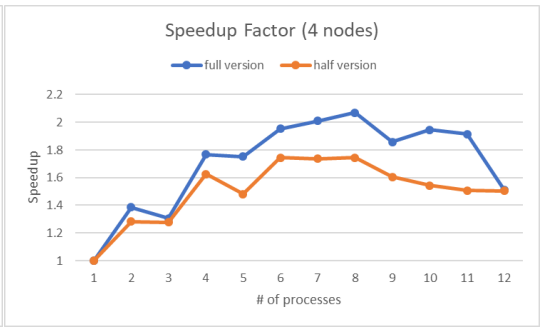
由下圖可見，無論是 full version 或 half version，他們的 speedup 大多是正成長的，4 node 的 speedup 幾乎都大於 1 node 的 speedup。不過 speedup 在 4 個 processes 後就趨緩了，甚至在 8 個 processes 後反而呈現負成長。理想的 speedup factor 應當是多一個 process 就快一倍，不過由下圖得知，最大的 speedup 不過是 2 倍再多一些，我認為是 communication time 及 IO time 將 runtime 限制住了，再因為要遵循 odd-even principle 的緣故，不易降低 iteration，speedup factor 才趨緩。



下圖分別呈現 full version 與 half version 在 1 node 與 4 node 下的 speedup factor，無論是 1 node 或 4 node 的情況下，full version 的 speedup 皆較 half version 好，這個結果與我用 hw1-judge 的結果相違背，我認為是因為 worst case 最終需要交換大多數的 data，僅有少量的 data 留在正確 local-array 中，因此 full version 直接交換所有的 data

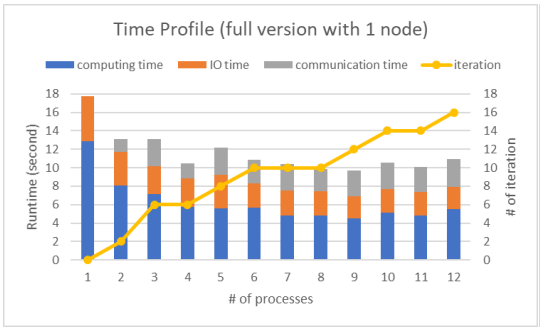
會較 half version 只交換一半的 data 來的更有效率。

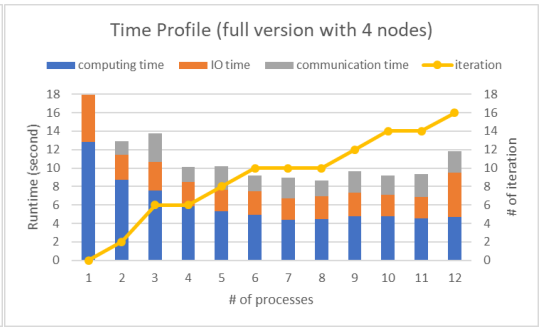




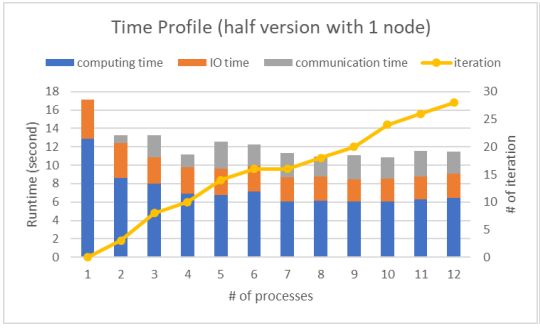
3. Time Profile

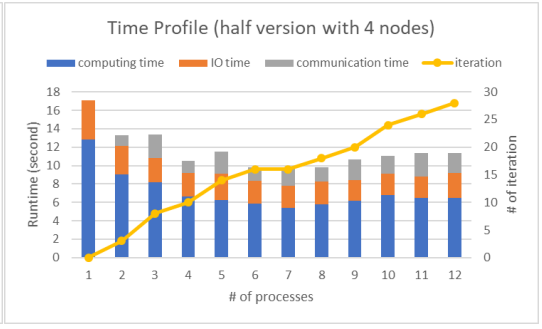
由下圖可見，full version 的 computing time 與 process 數成反比，process 數越多，computing time 越小，然而 computing time 降低的速度隨著 process 數漸多而漸趨平緩，代表程式的平行化已達極限，再多的 process 也無法繼續減低 computing time。由 IO time 分析得知，除了單一一個 process 的 IO time 較長之外，其餘的測試組合即使 process 數增加，IO time 也沒有下降，代表 IO 是這個程式平行化的瓶頸之一。由 communication time 及 iteration 數分析，雖然 process 增加，每個 process 的 local-array 變小，每次交換的 data 數變少，理論上 communication time 應該要降低，不過由於交換量變少，所需要的 iteration 數變多，因此 communication time 總和的差距才變小。





由下圖所見，half version 的結果與 full version 的結果雷同，值得注意的是 half version 的 iteration 數是 full version 的兩倍之多，雖然在一般情況下，half version 傳遞不需要 data 的機率較小，理論上表現會較佳，然而我的測資卻是對 half version 較不利的 worst case，因此才會變成 full version 比 half version 的 performance 更佳的假象。





iii. Discussion

綜合以上測試結果與圖表顯示，full version 與 half version 的 strong scalability、speedup factor 及 time profile 的趨勢雷同，若以 worst case 而言，full version 在各方面的表現較好，但以 general case 而言，卻是 half version 可以有較好的 performance。

1. full version：(N 為 data 數，T 為 process 數)

   在 full version 的版本中，每個 process 會先對自己的 local-array 排序，由於每個 process 會分配到 $\left\lceil \frac{N}{T} \right\rceil$ 筆 data，排序的時間複雜度最快為 $O(nlgn)$，因此 process 排序 local-array 的時間複雜度為 $O(\left\lceil \frac{N}{T} \right\rceil lg \left\lceil \frac{N}{T} \right\rceil)$。由於排序後需利用 odd-even principle 進行 merge，full version 每次需傳遞 $\left\lceil \frac{N}{T} \right\rceil$ 筆 data，又因為 merge 改成只 merge 一半資料的 merge，其時間複雜度為 $O(\left\lceil \frac{N}{T} \right\rceil)$，其完整的時間複雜度為 $O(\left\lceil \frac{N}{T} \right\rceil lg \left\lceil \frac{N}{T} \right\rceil + \left\lceil \frac{N}{T} \right\rceil + \left\lceil \frac{N}{T} \right\rceil)$，況且我有加入判斷是判斷每個 iteration 是否有 data 交換，用來決定是否排序完成，因此若非遇到 worst case，程式會提早結束，效能可能會更好。如上述討論，IO 及 communication 是此程式的 bottleneck，若需找到更好的 IO 系統或 communication 方式。

2. half version：(N 為 data 數，T 為 process 數)

   在 half version 的版本中，每個 process 會先對自己的 local-array 排序，由於每個 process 會分配到 $\left\lceil \frac{N}{T} \right\rceil$ 筆 data，排序的時間複雜度最快為 $O(nlgn)$，因此 process 排序 local-array 的時間複雜度為 $O(\left\lceil \frac{N}{T} \right\rceil lg \left\lceil \frac{N}{T} \right\rceil)$。由於排序後需利用 odd-even principle 進行 merge，half version 每次需傳遞 $\left\lceil \frac{\lceil \frac{N}{T} \rceil}{2} \right\rceil$ 筆 data，又因為 merge 改成只 merge 一半資料的 merge，其時間複雜度為 $O(\left\lceil \frac{N}{T} \right\rceil)$，其完整的時間複雜度為 $O(\left\lceil \frac{N}{T} \right\rceil lg \left\lceil \frac{N}{T} \right\rceil + \left\lceil \frac{\lceil \frac{N}{T} \rceil}{2} \right\rceil + \left\lceil \frac{N}{T} \right\rceil)$，況且我有加入判斷是判斷每個 iteration 是否有 data 交換，用來決定是否排序完成，因此若非遇到 worst case，程式會提早結束，效能可能會更好。如上述討論，IO 及 communication 是此程式的 bottleneck，communication 因為改成只傳一半資料已經有所改善，其餘只需要有更好的 IO 系統或是不同 IO 的方式，才可再增加 performance。

總使 full version 及 half version 在 strong scalability 表現不如預期，不過在 speedup factor 及 time profile 的方面而言，我覺得 performance 還不錯。

C. Experiences / Conclusion

這是我第一次寫平行程式，雖然程式架構容易設計，可是在 sort 及 merge 的部分，卻要花許多時間去思考，一開始優化很容易，可以很輕易地找出 bottleneck 及多餘的部分，在修改程式後，可以輕易地降低 runtime，可是當 runtime 變小的時候，就不容易優化程式，很多時候只改一些小地方，就有可能讓 runtime 增加，或是當時 cluster 的狀態不好，也會影響每次測試的 performance，實在花了很多心力在優化上面。

同時這也是我第一次測試程式並寫 report，之前在大學的時候都沒有寫過這類的 report，在寫的時候很常覺得自己的用字遣詞怪怪的，卻又不知道要怎麼敘述或說明，在做圖表的時候也是，很難做成自己腦中所想的圖樣，在寫 report 的方面也遇到很多困難，幸好有研究室的同學可以一起討論，才讓我完成了這次的程式作業。