# CS542200 Parallel Programming
# Homework 1: Odd-Even Sort

Bo-Wei Lee (李柏葳)

111065525

## 1 程式建置

### 1.1 元素分配

在這次作業中，我們總共拿到了 n 個元素，並且根據 Odd-Even Sort 的規則，將元素分配給節點們去做 Odd-Even phases。那在這邊我所運用到的策略是將元素平均分配給每一個節點，如果沒辦法平均分配，前面 $Floor(n/NPROC) + a$ 還會再被分配到一個元素。這樣子做的原因是在做 local elements 的排序以及 Odd-Even phases 裡產生新的陣列階段時，其複雜度至少爲 O(nlogn)。如果有一個節點拿到太多元素，其處理時間會太長，而且會造成其他 process 空轉。

### 1.2 預處理

在進入 Odd-Even phases 之前，我們主要需要依序做以下四件事情。

1. 計算左右節點的元素數量。在節點拿到自己的 rank 編號後，可以利用其來推算左右兩邊的元素數量。
2. 配置新的記憶體。在這裡，程式主要會配置四個陣列，一個是存放節點本身的元素，兩個是存放跟左右節點溝通的 buffer，還有一個是用來存每次 Odd-Even phase 排序後的結果。
3. 使用 MPIIO 來讀取資料。
4. 做本地元素的排序，在這邊是使用 C++ Boost library 裡的 Spread sort。

### 1.3 Odd-Even 階段

在這個階段，每個節點都與它的相鄰節點進行比較，並根據需要進行交換。在程式裡，總共會進行跟節點總數一樣多次的迭代，以確保最後的結果排序是對的。在每一次的迭代，每個節點會先跟相鄰的節點使用 MPI_Sendrecv() 的函式交換資料，然後再將自身的資料以及拿到的資料去做比較排序，產生出新的資料。在這邊特別注意的是假設在這個階段，節點拿到的是左半部的資料，那它就從最大的值往下排序至與其本身相同個元素爲止。如果是拿到右半邊的資料，那就是從小到大排序至與其本身相同個元素。在全部的排序結束後，程式會使用 MPI_File_write_at_all() 對其負責的部分進行輸出。

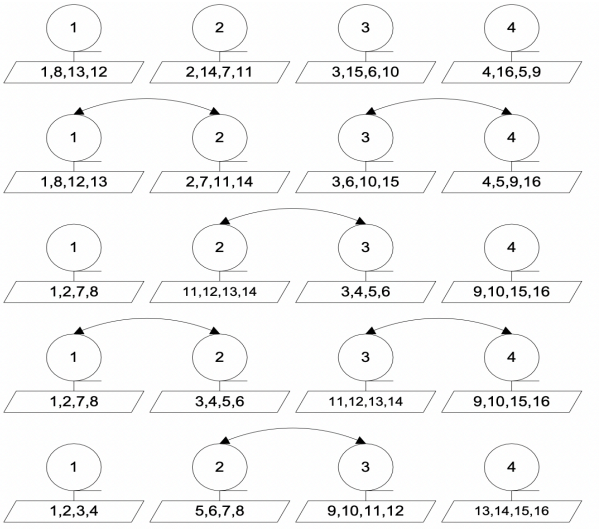


Figure 1: Data sorting steps of Odd-Even Sort method

### 1.4 Efforts used to speed up

1. 能夠 allocate 的部分盡量在 for 迴圈以外的地方先宣告好或是 allocate 好，已減少宣告變數所浪費的時間。

2. 在 Odd-Even 階段中使用 Sendrecv 來取代分開的 Send 及 Recv，這部分感覺是 MPI 的 library 有優化過了，所以在溝通的速度上會比還快一點。

3. 在 Odd-Even 的排序階段中，只排序較小或較大的半個陣列 (跟本身原本的元素加隔壁拿到的元素數相比)。並且在結束後，直接對調結果跟原本節點持有元素的指標，省去在複製元素回去的時間。

4. 使用 -O3 以及 -march=native 已使得編譯成 binary 檔時，使效能能最大化。

5. 在作業中，我有嘗試在交換之前讓節點先去欲交換之節點的最大 (小) 值，假設其最大 (小) 值比原本的節點的值還小 (大)，就不去做交換。但是在實作後發現這樣子的效能反而比不先去做比較還慢，所以後來就把程式碼有關這部分去掉。比較慢的原因主要是因爲交換最大或最小值時也會需要一個 MPI Call，而這個 MPI Call 的 Overhead 反而比有可能可以免去交換所有元素做比較的時間還長所致。

## 2 實驗與分析

### 2.1 實驗用資料與裝置描述

在以下實驗中，我選用的是公開資料集裡的 38.in 做爲輸入，其裡面共有 536831999 個元素，會選用這筆資料是因爲其數據量夠大，較可以凸顯平行程度帶來的優勢。所使用的機器則爲課程所提供的 Apollo 伺服器。計時方式式使用 MPI_Wtime() 在各個不同總類的函式，並計算總和。

### 2.2 Node 及 Process 的數量對於執行時間的影響

圖 2 是以 Odd-Even 排序所使用的程序對 Speedup factor 的圖。虛線代表程式要達到 Strong scaling 需要達到的 Speedup factor。這張圖上所有的 process 都是在同一個節點進行運算，降

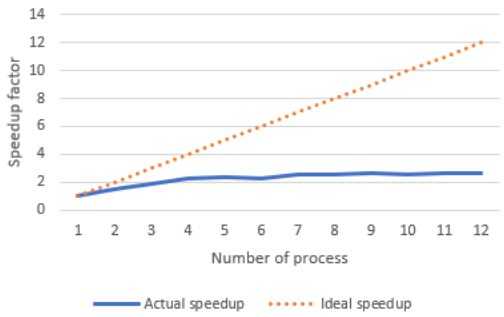


Figure 2: Process 數量對 Speedup factor 的圖

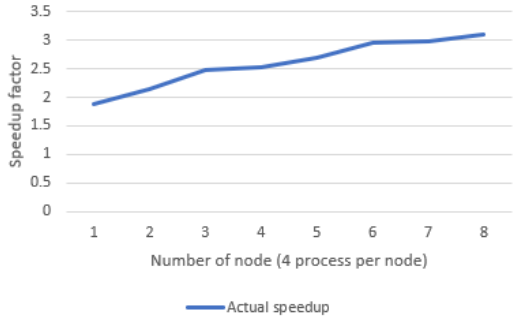


Figure 3: Process 數量對 Speedup factor 的圖

低資料傳輸的不穩定性。在這張圖中可以看到，隨著 process 數量的提升，Speedup 的斜率越來越低，不像虛線可以保持斜率在 1 。代表這個程式離完全的 Strong scaling 還有一段距離。圖 3 則是使用 Node 數量的多寡對於 Speedup factor 的圖，每一個 Node 會有 4 個 process。可以看到即使使用多達 32 個 process，Speedup factor 也卡在大約 3 左右。

## 2.3 Process 數量對各項執行時間的圖

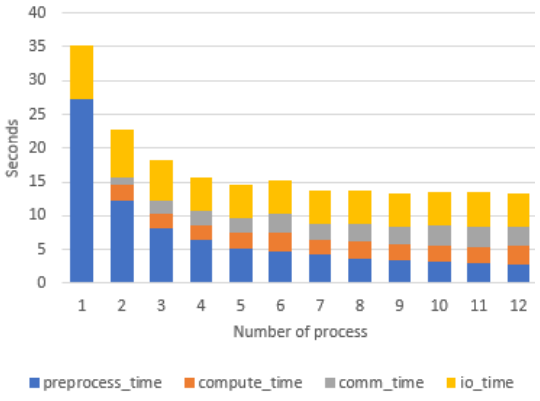


Figure 4: Process 數量對 Speedup factor 的圖

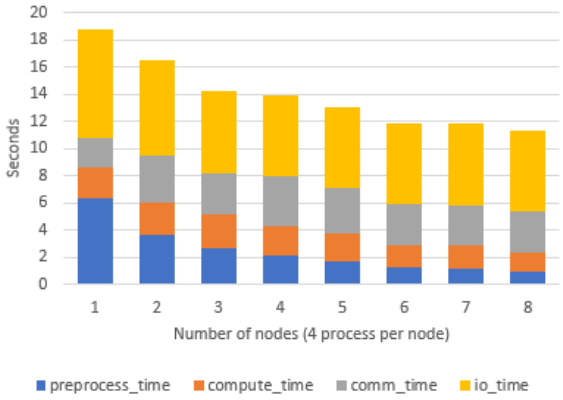


Figure 5: Process 數量對 Speedup factor 的圖

圖 4 爲 process 數量對各項執行時間的圖，其中 preprocess time 如 1.2 所寫，compute time 爲在 Odd-Even 階段排列的時間和，communicate time 爲 Odd-Even phases 中耳個節點互相傳遞資料的總和，io time 爲資料寫入與寫出的時間總和。圖 5 則是 Node 數量對各項執行時間的圖。在這兩張圖中，可以明顯的看到 Preprocess 的時間會隨著 process 的數量越來越少，這是因爲 Local 排序的時間會隨著負責的元素減少呈現指數性下降，compute 時間變化不大，因爲每個 Node 排序的數量乘上執行的 Odd-Even 數量剛好爲定值。Communicate time 則隨著 process 的增加剛開始有上升的趨勢，後來則趨向固定，應該是因爲節點數量增加同時每個節點需要傳送的數量減少所致，IO 時間則是都差不多，可能是因爲每個 process 都是透過同一個地方輸出，沒有平行化。

# 3 結論

這一次的題目我感覺能發揮的地方沒這麼多，因爲題目本身要求只能跟相鄰的節點交換資訊，所以其實對於演算法本身，有很少地方能進行改良。主要能發揮的地方是在於實作細節上。不過在實作程式時也是蠻有挑戰的，例如平常用來抓蟲的 GDB 在平行化的程式很難使用，所以需要用 printf 大法慢慢去排除造成輸出錯誤的原因。