hw1_112062532

姓名: 溫佩旻　**學號:** 112062532

Implementation

本次實作主要分成下面六個部分

1. 決定每個process所需要負責的資料量

由於每筆測資的 node、process、data 數量都不一樣，為了加快平行程式的速度，我採取的方法是將每筆 data 平均分配給每一個 process，以此做到 load balancing 的效果，若遇到無法平均分配的時候，例如20筆資料分給3個 node，會從 rank 0 開始依序多拿到一筆 data，也就是說在這個例子中，rank 0 會拿 index 0-6; rank 1 會拿 index 7-13; rank 2 會拿到 index 14- 19 的data，用這樣的方法能盡量平均每個process所負責的資料量。

2. 使用MPI-IO進行讀檔案

這部分使用 `MPI_File_read_at` 來做讀取，每個process讀取自己所負責的範圍的資料，而且每個process讀取的區間不重複，可以達到平行讀取的效果。

3. 每個 process 將自己負責的資料先做排序

spec中有特別提到可以使用任意方法排序local element，這邊我有嘗試使用sort()、並行的sort，但效果都不是很好，最後是查到一個基於 Radix Sort 的排序方法，這個方法非常適合用在排序大量的浮點數上，有接近線性的時間複雜度，相比於一般的sort()可以快1.14倍，效果很不錯!!!

```
////////// sort local data //////////
// std::sort(localData, localData + partLen);
// std::sort(std::execution::par, localData, localData + partLen);
// tbb::parallel_sort(localData, localData + partLen);
boost::sort::spreadsort::spreadsort(localData, localData + partLen);
```

4. 紀錄 proccess neighbor 的 rank 及其所負責的資料量

在正式進入 odd-even sort 前，需要先做一下紀錄一些待會會用到的資，因為proccess 在做 odd-even sort 時，odd phase 和 even phase 對應到的 neighbor process 會不同，所以要記錄他們的 rank ，以及 neighbor 負責的資料量長度為多少，在等一下開始做 odd-even sort 交換資料時會用到。這邊要注意第一筆資料和最後一筆資料，因為在邊界所以在odd phase 或 even phase 時可能會沒有 neighbor ，因此當計算 neighbor rank發現在邊界值(0~size-1)外時，要把它的 rank 設為 MPI_PROC_NULL，這樣在後續 MPI_SendRecv 交換資料時才會忽略此筆。

> ⓘ 註

> size = MPI_COMM_WORLD 的 process 總數

5. odd-even sort

接著開始做 odd-even sort ，這邊可以先想一下worst case需要跑幾次，當資料最大的兩個數字初始被排在 index 0 和 index 1 的位置(或是初始資料是由大排到小)，而我們想要的是資料由小到大排序，那就會需要 size+1次才可以將前兩個數值換到最後兩個，因此這邊我設的是跑 size + 1 次 for 迴圈，每次根據 i%2 來看這次是 odd phase 還是 even phase，透過 MPI_Sendrecv 來和 該 phase 的 neighbor 交換資料。

在拿到彼此的資料後會進行資料大小比對，也就是說，2個 process 的資料透過 MPI_Sendrecv 交換後，process 會各自拿自己 data 和 neighbor 的 data 從頭開始做大小比對，此時rank 小的 process 會從自己和 neighbor 資料中挑較小的前 k 筆來更新自己的資料，rank 大的process 則將自己的資料更新成自己和 neighbor 資料中較大的前 k 筆。

> ⓘ 註:
>
> - size = MPI_COMM_WORLD 的 process 總數
> - k = 當前 process 被分配到的資料量
> - 這邊 process 每一回合更新資料我是用swap()的方式，直接講pointer是最快的方式，可以減少memory copy的時間。

6. 每個process分別寫入自己的資料。

這部分使用`MPI_File_write_at` 來做寫入，每個process寫入自己所負責的範圍的資料，而且每個process寫入的區間不重複，可以達到平行寫入的效果。

Optimization

- 每個process local的sort方法
  原本是用STD的sort來排序local data，後來有查到
  `boost::sort::spreadsort::spreadsort`，這個很適合用來處理浮點數類型的排序，是用radix sort來排序，在需要處理大量的浮點數的情況下效能比原先的STD sort好非常多，這是我第一個優化的方法，大概快了15秒左右。
- 將mpich改成openmpi的module
  openmpi在process間的communication有做優化，因此在sendRecv的時間會比較短，整體時間也會有些許增加。
- send/recv data 的流程/方法
  原本在做odd-even sort時，會直接把local的data send給他的左鄰或右鄰，並receive他們的data再merge、swap等等操作，這部分滿耗時間的，因此我後來改成先傳自己的邊界值給對方，rank大的值傳自己的最小值，rank小的傳自己的最大值，例如：假設rank 1的process現在要和rank 2的process交換資料，他不會一次就把所有資料傳過去，而是先將自己local data 的最大值(也就是最後一個)傳給rank 2的process，同時接收rank 2的process傳來的他的最小值

(rank 2 process的第一筆資料)，接著比大小，若rank 1的最大值 < rank 2的最小值，代表兩邊並沒有重疊，不需要重新做merge和swap，這樣不但可以讓mpi間不用傳那麼大量的資料，也可以減少做無意義的merge和swap的時間。

Experiment & Analysis

- **Methodology**
  - **System Spec:** 一樣是在課堂提供的cluster上跑
  - **Performance Metrics:**
    採用 `Nsight System` 來計算時間
    - **IO time:** 在讀取和寫入檔案的 `MPI_File_open` 前和 `MPI_File_close` 後都加上 `nvtx`，最後將讀取檔案時間+寫入檔案的時間就會是IO time。
    - **communication time:** 在 `MPI_Sendrecv` 的前後加上`nvtx`，最後把全部加總起來就是communication time。
    - **computation time:** 在 `MPI_Init` 後面和 `MPI_Finalize` 前面加上 `nvtx` 紀錄程式總共執行時間，在扣除 IO time 和 communication time，剩下的就是computation time。

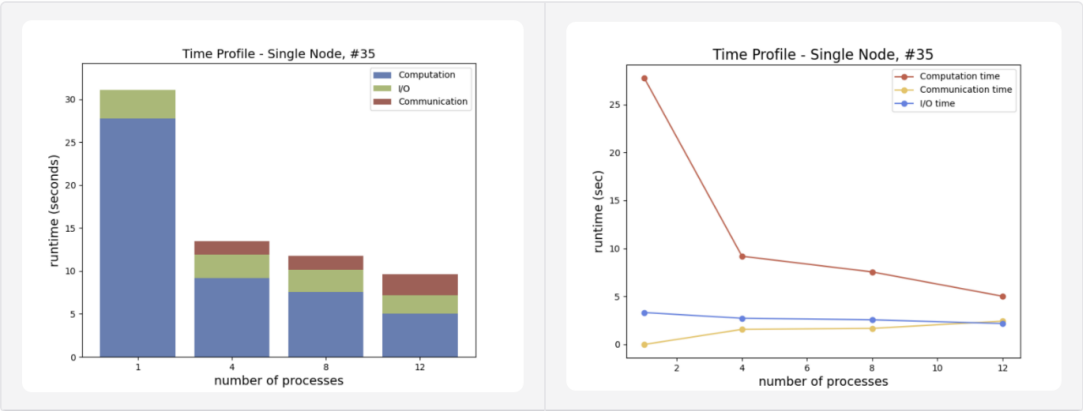
    > ⓘ 註:
    >
    > 這邊因為每個process都會有自己個computation time、IO time、communication time，所以這邊我是採用將每個process的這三種時間分別相加後做平均，得到此次執行的3個時間。

- **Plots**

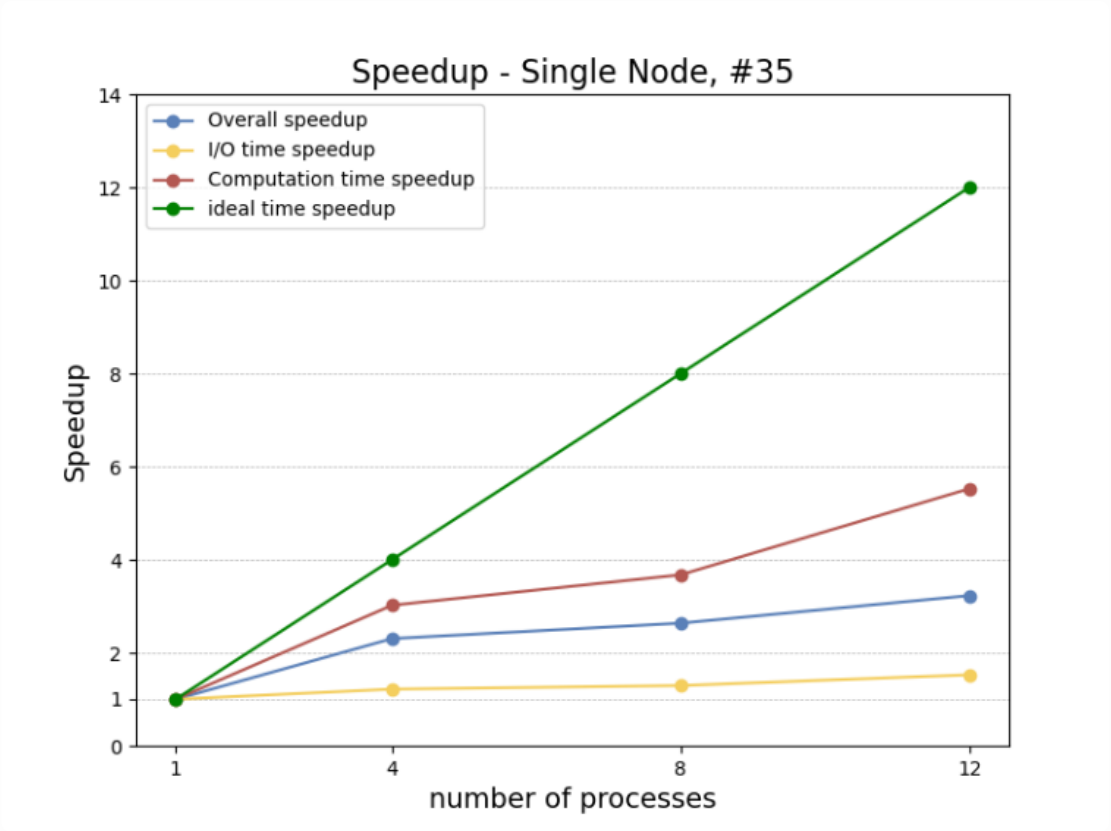
  Single Node

  - **Experimental Method:** 在40筆測資當中我的第35筆跑的時間最慢，因此使用第35筆測資(data 數量 = 536869888)在做實驗，測試在singel node的狀況下，1、4、8、12個process對程式效能的影響如何，得出下圖。

  - **Time Profile:**



- **Speed up:**



- **Analysis of Results**

  這邊主要針對圖中的3個metric做分析

  - computation time: 可以很清楚地透過上圖看到，隨著process的數量增加，計算時間有最明顯的減少，代表透過將資料分散給多個process做處理，能有效的平行處理，進而減少計算時間。不過可以注意的是，雖然計算時間明顯下降，但仍然還是整個程式中最花時間的部分，也就是我這隻程式的bottleneck。
  - I/O time: 可以從上圖發現I/O time隨著process的數量增加，有緩慢的下降，我認為應該是因為我是使用MPI_File_write和MPI_File_read來讀寫，每個process在讀寫上都是平行獨立的，不需要等待其他在做I/O的process，所以有稍微省下時間。

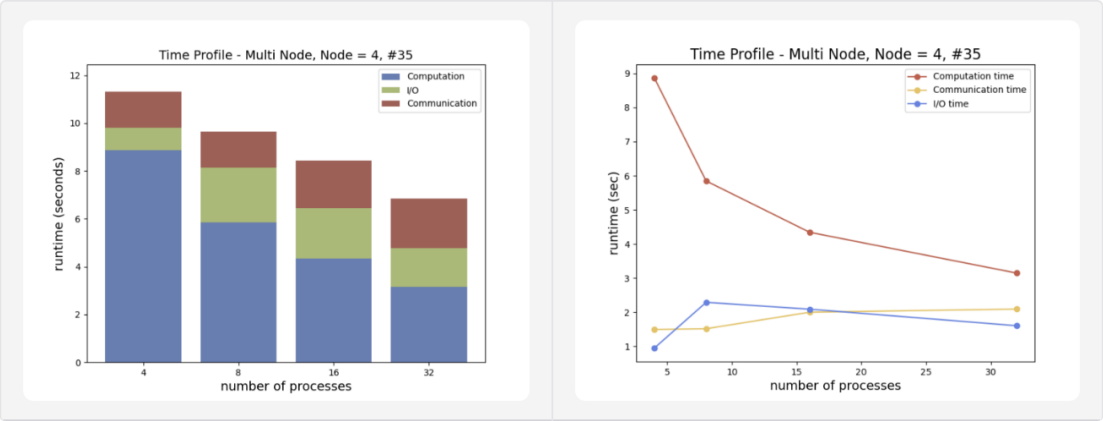
- communication time: 從上圖會發現，當process的數量增加的時候，process間的communication time不減反增，不過相差不會太大，只要大於1個process，process間就會有需要溝通的開銷，不過看起來差別並不會太大。

另外在Speedup的部分，可以明顯看到整體的speedup遠不及ideal的speedup，我想這部分是因為程式中還是會包含一些sequential code的部分，所以整體加速不會像想像中那麼完美，但從圖中還是可以清楚看到整體時間及計算時間有明顯的隨著process增加而減少。

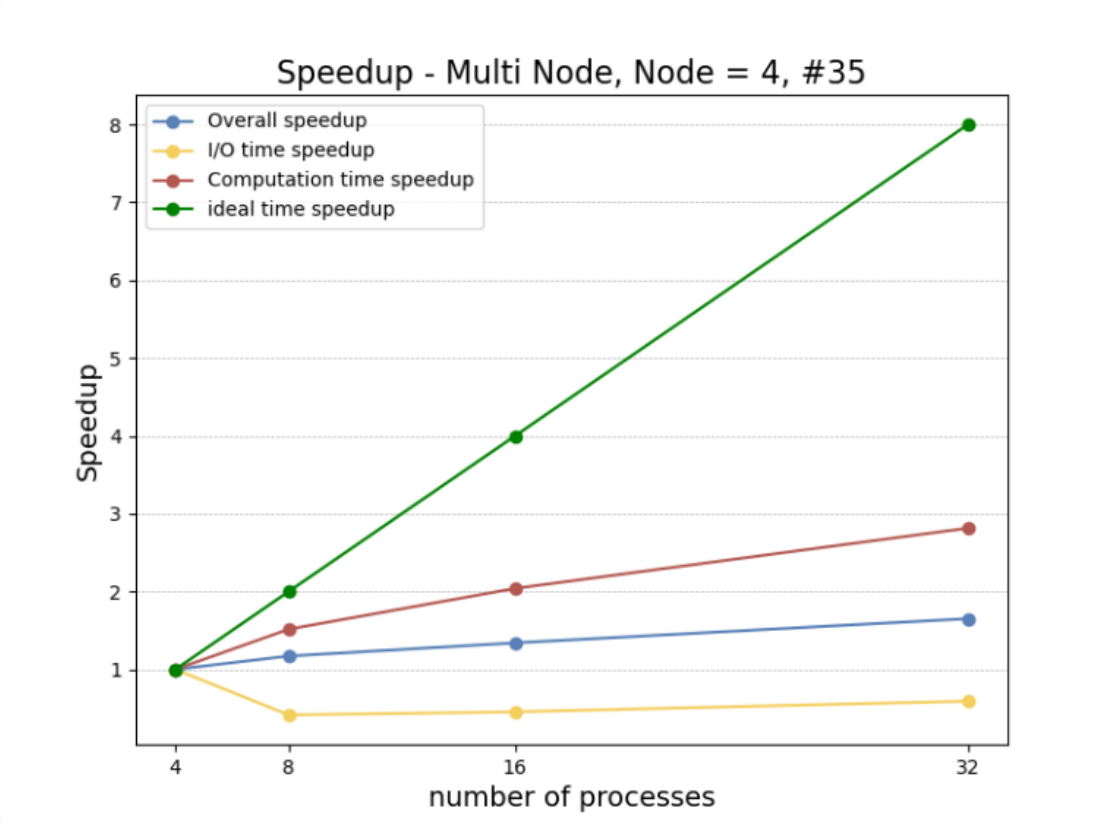
Muti Node

- **Experimental Method:** 在40筆測資當中我的第35筆跑的時間最慢，因此使用第35筆測資(data 數量 = 536869888)在做實驗，測試在4個node的狀況下，4、8、16、32個process對程式效能的影響如何，得出下圖。

- **Time Profile:**



- **Speed up:**



- **Analysis of Results**

  這邊主要針對圖中的3個metric做分析

  - computation time: 這邊和single node的情況相似，會因為process數量增加而有效減少計算時間，但仍然是程式中的bottleneck，應該還可以再想辦法優化。
  - I/O time: 從上圖會發現在Multi Node中，只有4個process的所花費在I/O上的時間反而是最少的，這邊我猜測是因為各個node分配到的process不均所造成的，例如8個process可能是node 1 分到5個process，其餘3個node各被分配到1個process所造成的。
  - communication time: 這邊和single node最大的不同就是多了node和node間的溝通，不過增長的幅度其實不太明顯，這邊我猜測是因為他會先把process(連續rank)都盡量分配到同一node上，而不用每個都涉及到node間的溝通，所以增長幅度不大。

  speedup的部分，和single node一樣，computation time的時間是減少幅度最大的，甚至比整體時間還有明顯改善，這應該是因為整體時間會受I/O、communication time因為跨不同node而額外導致的overhead影響。另外也可以看到整體加速遠小於ideal的加速，這也是因為相比於single node，還需要加上node間溝通的開銷等等，還有部分sequential code的影響所導致。

- **Optimization Strategies**

  從上述圖表來看，我認為我的程式可能還有兩點可以再優化:

1. 將`boost::sort::spreadsort::spreadsort`不用call library的方式，而是自己手刻radix sort，這樣或許可以再嘗試把sort用SIMD完成，成功的話可以達到更大平行化應該會有大幅的躍進。
2. I/O可能還有其他寫法能更有效率，像是我目前用`MPI_File_read_at`和`MPI_File_write_at`這些independent I/O來讀寫檔案，可能可以嘗試改成Collectiv I/O的方式，這在讀取大量data時可能會比較快。

- **Discussion**

1. 從上面圖表可以發現我的bottleneck應該是在cpu，也就是我的計算時間，雖然speedup已經是看似最好的，但距離ideal還是有滿大一段差距，應該還有地方可以在優化更好，像是我在上方Optimization Strategies中提到的，手刻radix sort，並用SIMD達到更大的平行化，在效能上應該會有滿好的提升。
2. 有實驗圖表可以看到，雖然我的實作方法在不同node和process數量的環境下，整體speedup都和process有線性關係，但還是可以看出我的scalability還有很大的進步空間，原因在上述的Analysis 及 Discussion第一點都有提到，而improvement的部分也如我上面Optimization Strategies提到的一樣，可以考慮再從I/O及computation time去做進一步優化

Conclusion

這次作業對於讓我MPI程式的運作更加了解，也讓我更體會到平行程式比sequential程式還需要注意到很多細節，特別是兩個process在交換資料時，一個沒有注意到就會導致原本的input資料換一換被換不見或變成NaN這種詭異的情況，加上之前沒有寫過平行程式的經驗，更容易踩到一些雷點，而且平行程式的debug也好麻煩!!幸好最後都有成功debug成功。

另外我覺得最困難的是優化的部分，基本上要寫出一個能夠過全部測資的code不難，但後續要怎麼提升效能好難!!最後全部做的優化是優化local sort方法、交換data時直接用swap交換pointer而不是用memory copy的方式、將mpich換成openmpi(openmpi有針對communication做優化，所以整體時間也快一點)，其他地方很難找到突破點，像是IO時間也有嘗試做調整，但都沒有成功，真的很好奇前面那些速度比我快幾十秒的到底怎麼做到的，希望之後能夠提供這些同學的想法讓我們參考。