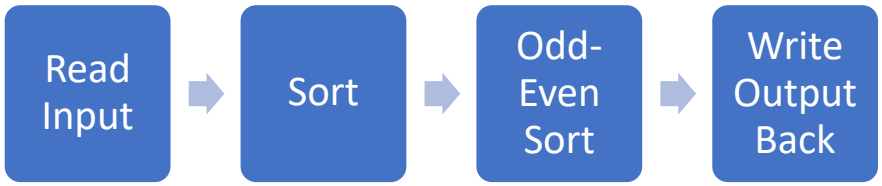
HW1: Odd-Even Sort

107062223 歐川銘

I. Implementation

在這次作業可以把我的實作部分拆分成以下四個步驟



1. Read Input

每個 process 都會有 process 數量跟自己 rank 這兩個資訊，透過這兩個資訊決定每個 process 要處理 data 裡面的哪一段資料。
在我的實作中，首先我先先判斷是否 process 數量有沒有比要處理的資料大小還有多，如果有的話就只取用跟資料大小相同的 process 數量讓每個 process 只負責一筆資料並且把多餘的 process kill 掉，至於其他資料大小比 process 數量多的情況，每個 process 會平均分配要處理哪一段的資料，遇到除不盡有餘數 m 的情況就是前面幾個 m 個 process 再多分配一筆資料，這樣可以確保 process 之間最多只差一筆資料，每個 process 再根據要處理的資料位置去讀取 file。

2. Sort

在做 odd-even sort 之前，我的做法是先把每個 process 裡的資料先做 sort 確保每個 process 現階段的資料都是 ascending order，讓之後 process 在交換資料的時候會更方便更新得到所需要的資料。
在這邊我使用的 sort 是 C++ boost sort library 提供的 float_sort，float_sort 主要是針對 floating 的資料進行優化，主要概念是用 radix sort 把 32 bits 資料每 11 bits 做一個分界，radix sort 的 time complexity 是 O(d(n+b)) 在這個 case 裡面會比一般的 sort 再快一點。

3. Odd-Even Sort

在 Odd-Even Sort 之前，我先會讓每個 process 先算好 left process 跟 right process 個別有多少筆資料，這樣資料在做交換的時候就不需要再多花一次 communication 的時間去知道對方總共有幾筆資料，再來我把 Odd-Even Sort 總共分成四個部分去做解釋資料如何做交換以及中止條件。

1. Termination

每個 process 做 odd-even sort 最多只需要做 size+1 次就可以確定全部資料一定是 sort 好的結果，因為如果在每個 process 資料 balance 的情況下，最糟情況就是某一筆資料需要做 rank 次就可以到正確的位置，但是因為現在每個 process 最多可能會差一筆，所以需要再多加一次確保正確性。

```
for(int i = 0; i < size + 1; i++)
```

2. Even Phase

在 even phase 每個 pair 裡面，rank 是偶數相對於 rank 是奇數是要拿取比較小的資料，一開始先判斷最後一個 process 是不是偶數，是偶數的話代表這個 process 這一輪不需要傳輸資料，然後其他 pair 左邊的 process 會傳給隔壁最大的一筆資料 left_big_value 跟右邊的 process 最小的一筆資料 right_small_value 去做比較，如果 left_big_value 小於等於 right_small_value 的話就代表這對 pair 目前暫時沒有要交換資料，所以兩邊不用再把整個 array 互相交換，但如果 left_big_value 大於 right_small_value 就需要傳輸兩邊的資料做 merge sort 更新。

為了增加平行度，在傳輸資料的部分我是 call MPI_Sendrecv 讓兩邊 process 都可以同時得到資料去做更新，而不是只有讓一邊處理資料而另外一邊 idle 等待資料回傳。

```
// even sort
if (rank % 2 == 1) { // process 在右邊
  // 先確認兩邊 process 資料是不是需要更新
  MPI_Sendrecv(&data[0], 1, MPI_FLOAT, rank - 1, 0, &temp_buffer[0], 1, MPI_FLOAT, rank - 1, 0, mpi_comm, &status);
  if (data[0] < temp_buffer[0]) {
    MPI_Sendrecv(data, total, MPI_FLOAT, rank - 1, 0, temp_buffer, left_total, MPI_FLOAT, rank - 1, 0, mpi_comm, &status);
    exchange_right(left_total, total);
  }
} else if (rank != size-1) { // process 在左邊除了最後一個是偶數的情況
  MPI_Sendrecv(&data[total-1], 1, MPI_FLOAT, rank + 1, 0, &temp_buffer[0], 1, MPI_FLOAT, rank + 1, 0, mpi_comm, &status);
  if (data[total-1] > temp_buffer[0]) {
    MPI_Sendrecv(data, total, MPI_FLOAT, rank + 1, 0, temp_buffer, right_total, MPI_FLOAT, rank + 1, 0, mpi_comm, &status);
    exchange_left(right_total, total);
  }
}
```

3. Odd Phase

   Odd phase 的時候跟 even phase 原理一樣，每個 pair 也需要先判斷是否要傳輸資料，唯一不同的地方是 rank 0 跟 rank 是奇數並且是最後一個 process 不需要做任何動作，其他傳輸資料的方式都跟 even phase 提到的做法一樣，都是兩邊互相交換資料，並且個別去做 merge sort 更新自己所需要的資料。

4. Merge Sort

   每個 process 都需要維護 total 筆資料，拿到 neighbor 傳來的資料之後，在 pair 左邊的 process 必須要拿取兩邊最小的 total 筆資料，而右邊的 process 必須要拿取最大的 total 筆資料，更新完成之後這個 phase 就算結束。

```
// for right
void exchange_right(int neighbor_total, int total) {
  for (int i = total-1, run_n = neighbor_total-1, run = total-1; i >= 0; i--) {
    put_buffer[i] = (run_n >= 0 && data[run] < temp_buffer[run_n]) ? temp_buffer[run_n--] : data[run--];
  }

  swap(data, put_buffer);
}

// for left
void exchange_left(int neighbor_total, int total) {
  for (int i = 0, run_n = 0, run = 0; i < total; i++) {
    put_buffer[i] = (run_n < neighbor_total && data[run] > temp_buffer[run_n]) ? temp_buffer[run_n++] : data[run++];
  }

  swap(data, put_buffer);
}
```

4. Write Output Back

這個部分就是把每個 process 的維護的 total 筆資料，根據 read file 時候從 offset 多少讀出來，就從一樣的 offset 開始寫回到 file 裡面，最後再 call MPI_Finalize 結束 Odd-Even Sort。

II. Experiment & Analysis

1. Methodology

a. System Spec

這邊我是使用課堂上提供的機器 Apollo 來做測試，並且調整每次要使用的 process 的數量來進行實驗。

b. Performance Metrics

測試 performance 的部分，我是用這次 testcases 提供的第 30, 33 筆測資，並且自己另外寫了一個 calculate_time.h 跟 calculate_time.cc 的 file 來測試時間，檔案裡面使用的是 MPI library 提供的 MPI_Wtime()，在特定程式前後加上特定的 check point 就可以計算出每個 process 的 Compute、Comm、IO 時間，最後再傳給 rank 0 的 process 做平均得出結果。

Compute time: 紀錄 data 做 sort，以及 data 交換之後做 merge sort 更新資料的時間。

```
/*
  calculate cpu time
*/
calculate_time->record_cpu();
exchange_right(left_total, total);
calculate_time->update_cpu();
```

Comm time: 紀錄 communication 所花的時間，紀錄在任何傳輸 data 給其他 process 所花費的時間。

```
/*
  calculate COMM time
*/
calculate_time->record_comm();
MPI_Sendrecv(data, total, MPI_FLOAT, rank - 1, 0, temp_buffer, left_total, MPI_FLOAT, rank - 1, 0, mpi_comm, &status);
calculate_time->update_comm();
```

IO time: 紀錄 read file 跟 write file 總共所花費的時間。

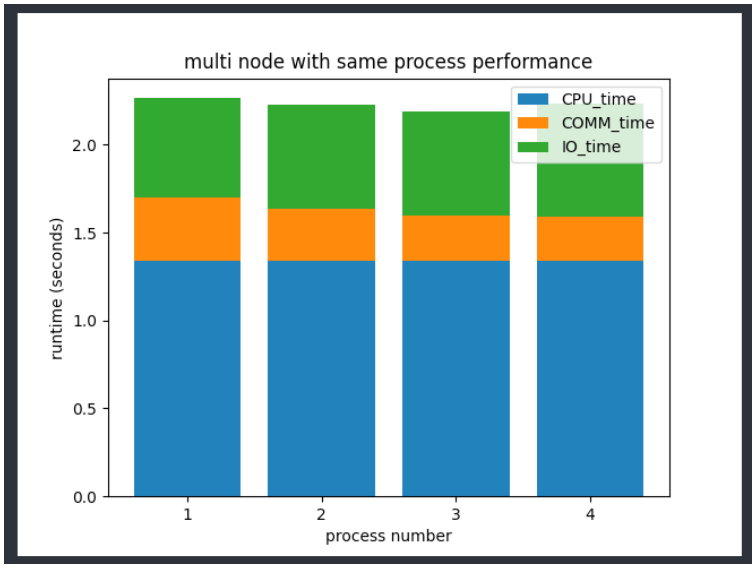
```
/*
  calculate IO time
*/
calculate_time->record_io();
// write back
check = MPI_File_open(mpi_comm, argv[3], MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &f);
if (check != MPI_SUCCESS) {
  cout << "open file error" << endl;
  MPI_Abort(MPI_COMM_WORLD, check);
}
MPI_File_write_at(f, offset, data, total, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&f);
calculate_time->update_io();
```

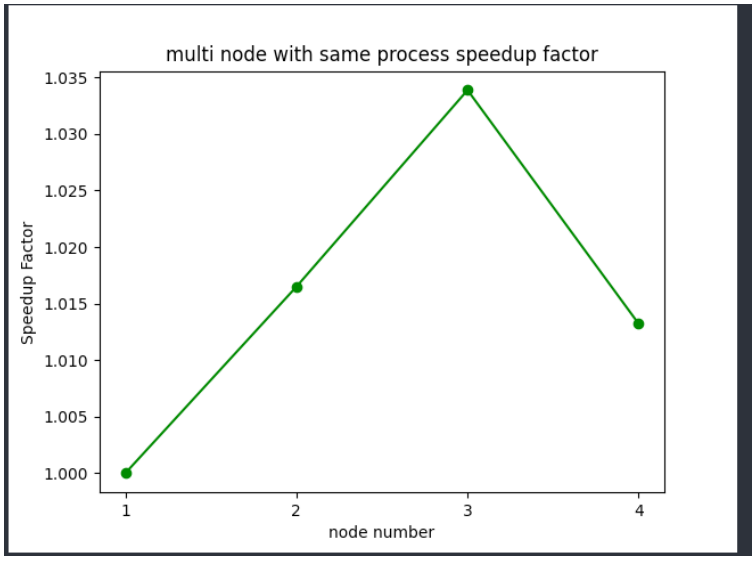
2. Plots:Speedup Factor & Time Profile

Multiple Nodes with same number of process

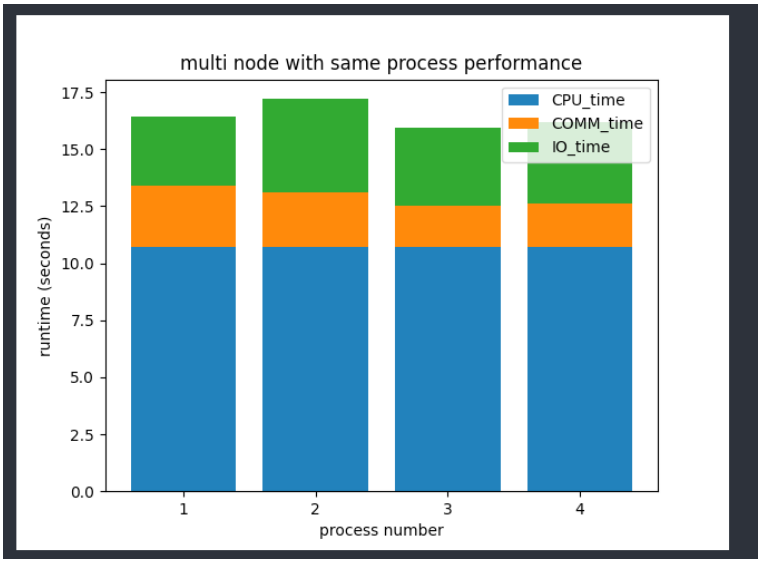
固定好每次都執行一樣數量的 process 12，再控制要執行的 node 數量，並且如果有多個 nodes 時候，每個 node 都處理一樣數量的 process。

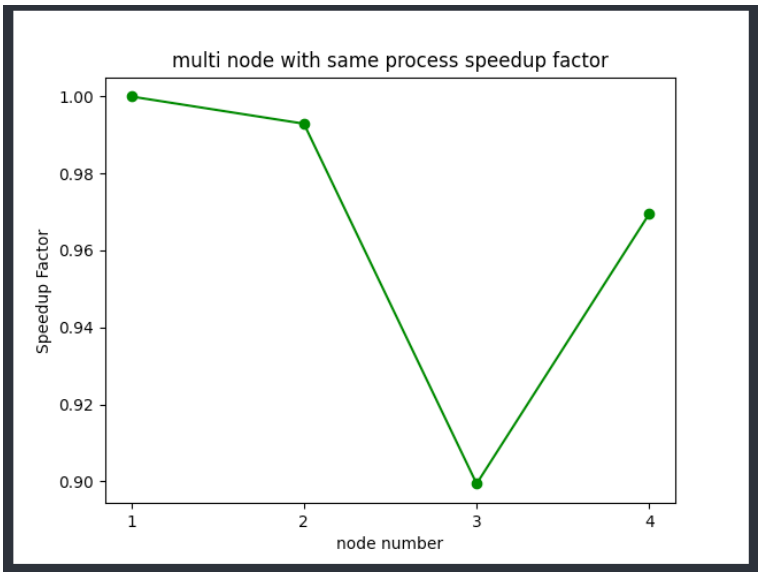
首先是第 30 筆測資，data 數量大小是 64123483，我認為在 process 一樣的情況下，速度上不會有太大的差異，所以從下圖 time profile 跟 speedup factor 中基本上看不太出來差別。





再來是第 33 筆，基本上也沒有太大的差異，頂多就是每次實驗機器的因素所導致的些微差距，跟 30 筆結果其實差不多。

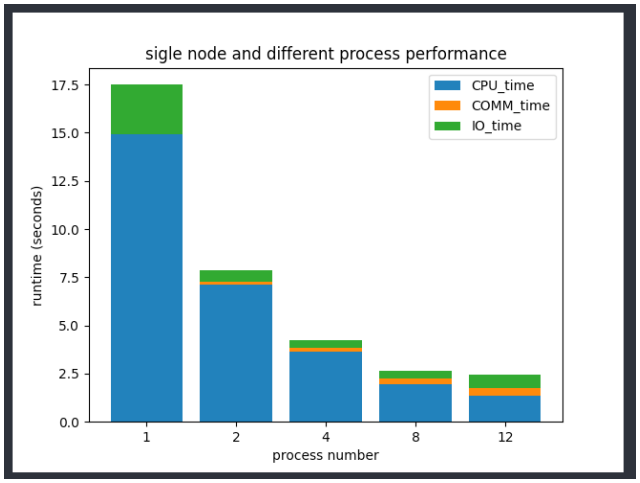


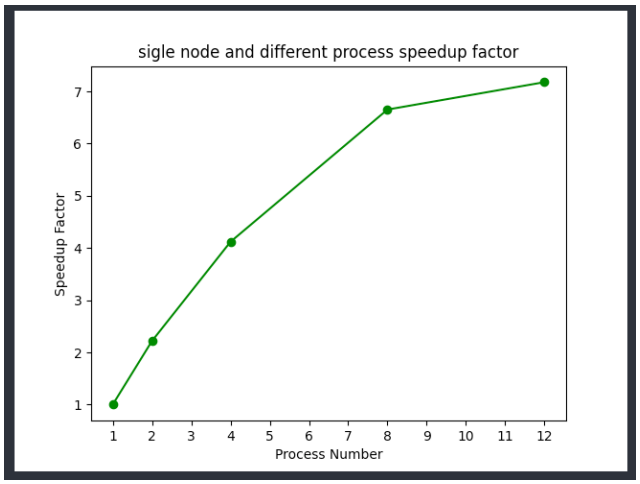


Single Node with different number of process

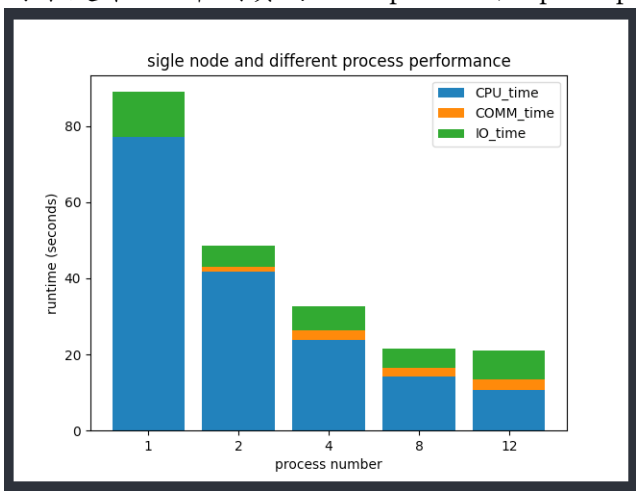
在一個 node 上面跑一樣的測資，然後 process 數量從 1 到 12 來看 process 數量增加看 performance 的變化，這個實驗也可以用來解釋 scalability 對 performance 的影響。

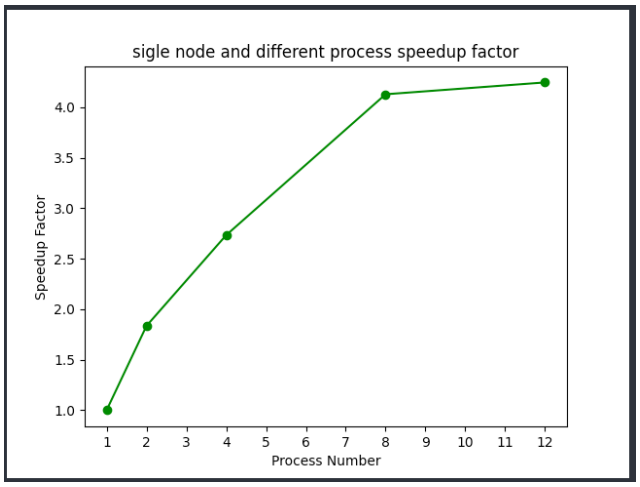
首先先是第 30 筆測資的 time profile 跟 speedup factor 的圖





再來是第 33 筆測資的 time profile 跟 speedup factor 的圖表





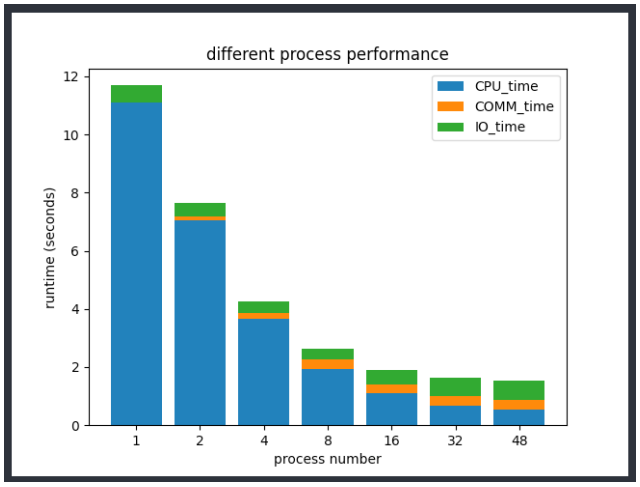
最後從以上四個圖中可以得出以下兩個結論

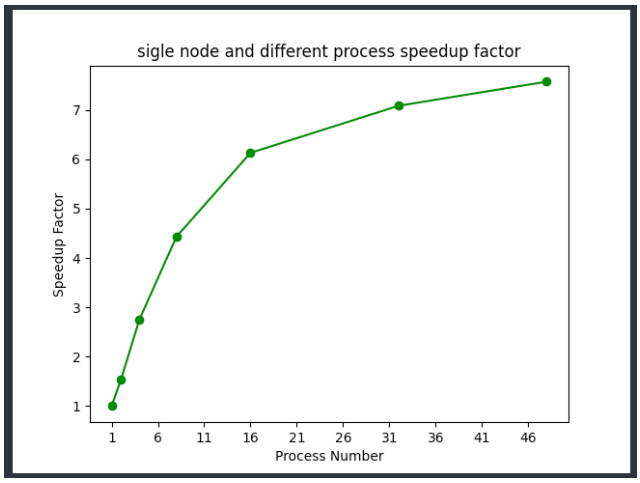
1. Process 數量越多，整體的時間會慢慢降低，但是下降的幅度會慢慢減緩，代表 scalability 伴隨著 process 數量上升，performance 可以變得更好。
2. 從圖中可以明顯看出 CPU time 在下降，所以也代表每個 process 所要處理的運算量有漸漸減少，但是 Comm time 有開始增加，這就代表 process 數量變多所要溝通的時間也變的更多，甚至很可能 process 數量越多導致 performance 下降。

Different number of process

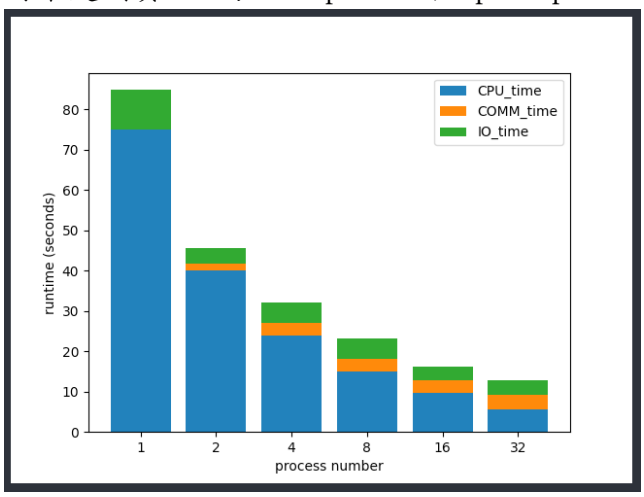
這個實驗主要就是看越多的 process 對整體的 performance 影響，看看 scalability 越大，會有什麼結果

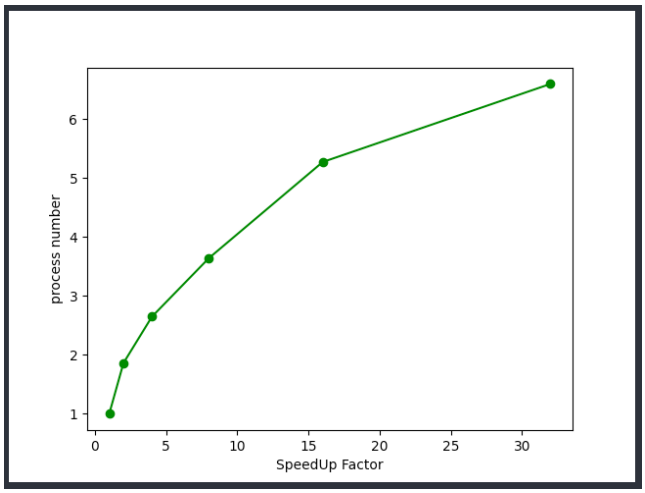
一樣是第 30 筆測資的 time profile 跟 speedup factor





再來是測資 33 的 time profile 跟 speedup factor





3. Discussion

1. Bottleneck: 在 process 數量不多的時候，從上面實驗結果可以明顯看到，compute time 原本是 bottleneck，但是 process 數量變多的時候，反而 Comm time 變成了 bottleneck，所以要降低 Comm time 的話，首要的方式就是減少 Comm 的次數，或是減少 process 之間傳輸資料的大小，那在這次的實作中，我認為減少 process 之間傳輸資料的大小是比較有可能的，因為有太多的資料是被重複傳輸的，所以如果要更近一步優化的話，我認為要針對這些重複的資料去做處理才有可能讓結果更好。

2. Compare Scalability: 在這次的實作中跟實驗結果，我認為平行度的部分有達到還不錯的效果，因為從上面各種實驗結果來看，process 數量越多，整體時間有明顯開始下降，尤其是 Compute time 更是快上非常多，但因為 process 數量越多意味著要付出更多的 Comm 時間，所以如果想要達到更好的 scalability 的話就要想辦法降低 Comm 時間，或是讓 Comm 上升的幅度小於 Compute time 跟 I/O time 下降的幅度，我想到比較好的方式就是結果 share memory 讓某些資料不要一直重複被做 memory copy，這樣比較有可能讓 scalability 再更好一點。

III. Experience / Conclusion

在這次的作業中，讓我更了解了 MPI 要如何使用，因為這次作業類型很適合嘗試去使用各種 MPI function 來看看有沒有辦法讓自己的程式變得更快，此外為了加快 process 之間的平行度來避免某個 process 因為太慢而讓整體時間大幅增加，所以也必須考慮每個 process 要處理的 data 數量，過程中也需要結合許多以前演算法的知識或是用到計算機結構的知識來加快速度，更重要的是平行程式讓我了解到 coding style 跟做好註解的重要性，因為平行程式思考模式跟之前所有的程式有不小的差異性，我要時刻告訴自己現在寫的程式將來是同時多個 process 去執行，所以要明確知道現在每個 process 是處理資料的哪一個部分，有 process 傳資料就要記得一定有 process 要收資料，因為如果沒有注意好各個細節，就會產生很多預期之外的錯誤，可能是哪邊沒有寫清楚導致某些 process 傳輸的數量跟接收的數量不同，或是某個 process 忘記寫 Recv 去接收資料，這些都是我一開始遇到的問題。

雖然過程有點艱辛，因為畢竟是第一次寫有關 MPI 的程式，但是過程中我的確也學習到蠻多知識，包括在寫 report 的過程中也需要去查一點資料驗證自己的推論，而且最後做出來的結果在 scoreboard 上也有不錯的名次，讓人成就感十足～