HW1 Odd-Even Sort

student ID: 112062619 **name:** 陳航希

Implementation

因為實驗部分分為了original、cpu優化版與communication優化版，以下分別說明實作方式:

1. Original

首先將整個陣列平均分配給各個 process。
因為有可能無法整除，所以將多出來的元素分配給前 `remainder` 個 process。
每個 rank 根據計算結果取得自己的資料數量 (`local_num`) 與檔案偏移量 (`byte_off`)，並利用 `MPI_File_read_at` 從檔案讀取到對應的記憶體區段。

```
int base = num / size;
int remainder = num % size;

int local_num = base + (rank < remainder ? 1 : 0);
MPI_Offset elem_off = (MPI_Offset)base * rank + std::min(rank, remainder);
MPI_Offset byte_off = elem_off * (MPI_Offset)sizeof(float);

std::vector<float> local_partition(local_num);

MPI_File_open(MPI_COMM_WORLD, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &i
MPI_File_read_at(input_file, byte_off, local_partition.data(), local_num, MPI_FL
MPI_File_close(&input_file);
```

1. 決定 odd / even partner

- 若 `rank` 是奇數（ `rank & 1` 為真）：
  - **odd partner** = 左邊的 rank (`rank - 1`)
  - **even partner** = 右邊的 rank (`rank + 1`)
- 若 `rank` 是偶數：
  - **odd partner** = 右邊的 rank (`rank + 1`)
  - **even partner** = 左邊的 rank (`rank - 1`)

```
int odd_partner, even_partner;

if (rank & 1) {
    odd_partner = rank - 1;
    even_partner = rank + 1;
} else {
```

```
    odd_partner = rank + 1;
    even_partner = rank - 1;
}
```

2. 接著計算自己左右鄰居的大小

由於整體資料大小 `num` 可能無法被 `size` 整除，所以前 `remainder` 個 `rank` 會多拿一個元素（即 `base+1`）。
`rank` 如果在左右兩端就會沒有 `odd_partner` / `even_partner` 所以要設成-1。

`partner_partition(base+1)`：用來接收鄰居傳來的資料（因為最多會是 base+1 個）。

`merged_partition(local_num)`：用來存放合併後的結果，大小等於自己的 partition。

```
if (odd_partner<0 || odd_partner>=size) odd_partner = -1;
if (even_partner<0 || even_partner>=size) even_partner = -1;

std::vector<float> partner_partition(base + 1);
std::vector<float> merged_partition(local_num);
```

接著，每個 process 對自己擁有的資料進行排序，使用 C++ STL 提供的 std::sort 函式。這是一個基於比較的排序，時間複雜度為 $O(n \log n)$，在大量資料時會成為主要的計算瓶頸。

```
std::sort(local_partition.begin(), local_partition.end());
```

Odd-Even Sort Iteration

進入排序階段後，每個 process 會執行多輪迴圈（共 size 輪），交替進行 even phase 與 odd phase。

- **even phase**：偶數 rank 與右鄰交換，奇數 rank 與左鄰交換。
- **odd phase**：偶數 rank 與左鄰交換，奇數 rank 與右鄰交換。
  在每一輪，程式會決定 partner 與其資料大小，並呼叫 `exchange_with_partner()` 進行交換與合併。

Exchange with Partner

此函式負責與鄰居交換資料並進行合併：

1. 利用 `MPI_Sendrecv` 雙向傳輸，交換兩個分區的元素。
2. 根據 rank 與 partner 的相對位置，選擇保留較小的一半或較大的一半。
3. 合併結果存到暫存區，再與本地資料交換。

這樣確保整個過程逐步趨近全域排序。

Merge Operation

合併函式負責將本地資料與 partner 的資料進行歸併：

- 若需要保留較小的一半，則從兩個陣列的開頭往後比較。
- 若需要保留較大的一半，則從兩個陣列的尾端往前比較。

最終得到一個已排序好的分區，長度與本地資料一致。

2. CPU Optimized

做了兩種優化：

1. **更快的排序**
   將 `std::sort` 改為 `boost::float_sort`，將時間複雜度從 $O(n \log n)$ 降到 $O(n)$。

```
boost::sort::spreadsort::float_sort(local_partition.begin(), local_partition.end
```

2. **合併 odd/even phase**
   在同一個迴圈中同時進行 even 與 odd phase，以此降低迴圈數量。
   每個 phase 使用不同的 tag，避免 MPI 訊息衝突：

```
for (int step = 0; step < size; step += 2) {
    int base_tag = 1000 + step * 10;
    if (even_partner != -1)
        exchange_with_partner(rank, even_partner, even_partner_size,
                              local_partition, partner_partition, merged_partiti
                              MPI_COMM_WORLD, base_tag + 10);

    if (odd_partner != -1)
        exchange_with_partner(rank, odd_partner, odd_partner_size,
                              local_partition, partner_partition, merged_partiti
                              MPI_COMM_WORLD, base_tag + 20);
}
```

3. Communication Optimized

在進行 Odd-Even Sort 的交換時，若兩個分區本來就有序，完整的資料傳輸與合併會造成不必要的開銷。
因此，我們在每一輪交換前，僅比較邊界元素：

- 若 `rank < partner`，只需比較 `local.back()` 與 `partner.front()`。
- 若 `rank > partner`，只需比較 `local.front()` 與 `partner.back()`。
- 若邊界已經正確，則整個分區必定有序，不需交換即可提前跳過。

這樣的設計能顯著減少不必要的 MPI 通訊與合併計算。

```
bool exchange_with_partner(...){
    // 邊界檢查 (只交換一個元素)
    float send_val = (rank < p_rank) ? local.back() : local.front();
    float recv_val;
    MPI_Sendrecv(&send_val, 1, MPI_FLOAT, p_rank, tag,
                 &recv_val, 1, MPI_FLOAT, p_rank, tag, comm, &status);

    bool need_exchange = (rank < p_rank) ? (send_val > recv_val)
                                         : (send_val < recv_val);
    if (!need_exchange) return false;

    // 若需要，才交換整段資料並合併
    ...
}
```

Experiment & Analysis

1. Methodology

- **System Spec:**
  所有實驗皆於課程提供之 Apollo HPC Cluster 上執行。

- **Performance Metrics:**
  使用作業測資中的 37.in 作為實驗資料，資料長度為 536869888。

  實驗統計 `MPI_Init` 至 `MPI_Finalize` 之間的時間。透過 `MPI_Wtime()` 手動標記程式中各個區段，將總執行時間細分為以下三類：

  - **I/O Time**：涵蓋所有 MPI-IO 相關操作，例如 `MPI_File_open`、`MPI_File_read_at`、`MPI_File_write_at`、`MPI_File_close`。
  - **Communication Time**：涵蓋所有 `MPI_Sendrecv` 的通訊操作，包括邊界值交換與完整資料區塊的交換。
  - **CPU Time**：其餘純計算部分，包括初始排序（`boost::float_sort`）與每一輪合併排序（`merge` 函式及 `std::swap`）。

  時間紀錄方式如下：

  ```
  double t0 = MPI_Wtime();
  // 某一段操作
  t_cpu += MPI_Wtime() - t0;
  ```

  最後，使用 `MPI_Reduce` 將各 rank 的時間資料彙整至 root process 進行統一輸出：

  - **I/O、CPU、Communication Time**：對所有 rank 的時間取平均（average over all ranks），反映整體的平均計算與通訊負擔。

- **Total Time**：對所有 rank 的總時間取最大值（maximum over all ranks），作為全程的 wall-clock time，反映系統中最慢的 process 對整體效能的影響。

因為三段時間皆為各 rank 的平均值，而 total time 是取最大值，因此其總和有可能會小於 total time。

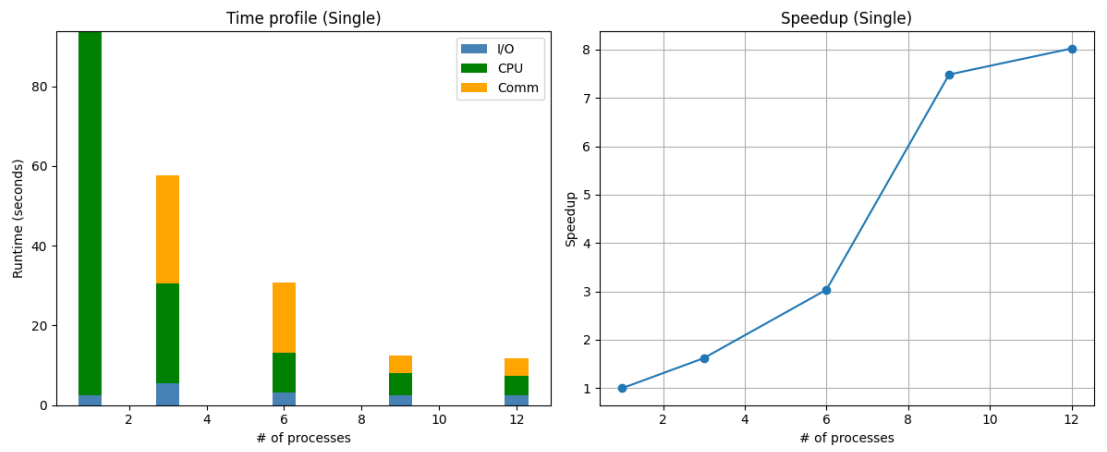
- **Experiment Setup:**
  共進行三組實驗，分別對應三個版本：**original**、**CPU optimized**、**fully optimized(包含 cpu, communication optimized)**。每組實驗均包含以下兩類測試：
  - **Single-node 測試**：固定使用 `-N 1`，調整 process 數量為 1、3、6、9、12，量測各版本在不同規模下的執行時間，並細分為 I/O、Communication、CPU 三種時間類型。
  - **Multi-node 測試**：固定 process 數量為 12，分別在節點數 1、2、3、4 的情況下執行，同樣記錄並分析 I/O、Communication、CPU 三種時間區段。

2. Plots: original

- **Single node**
  - Table

Process數	IO Time(s)	CPU Time(s)	COMM Time(s)	Total Time(s)
1	2.465788	91.292123	0.000000	93.757913
3	5.565827	24.925905	27.208091	57.806786
6	3.169446	10.065692	17.640793	30.877426
9	2.541307	5.544191	4.359775	12.528549
12	2.446096	4.795447	4.442920	11.687863

  - Profile & Speedup figure



- Discussion
  由表格與圖表可觀察到以下趨勢：

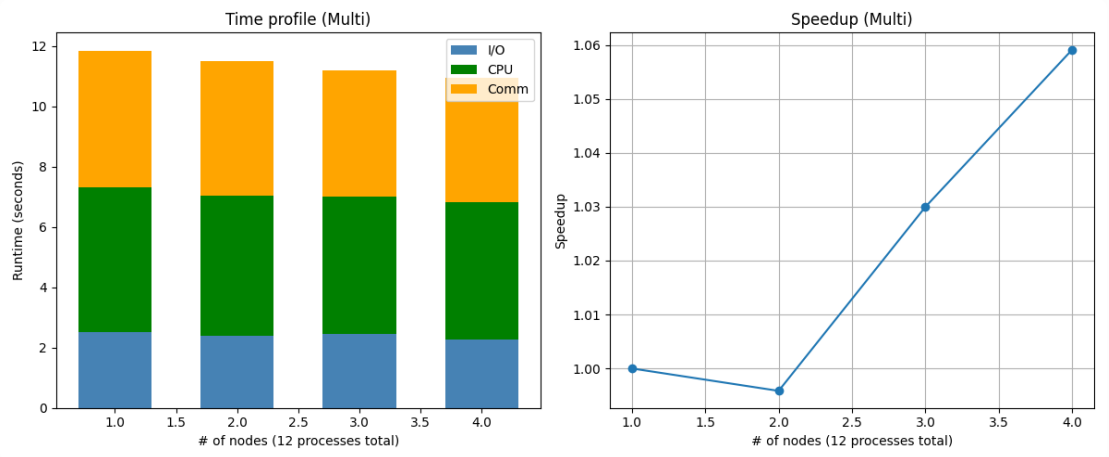
  - **CPU Time** 隨 process 數增加快速下降，顯示計算部分平行化效果良好。
  - **Communication Time** 在少量 process 幾乎可忽略，但隨平行度提升逐漸增加，並在高 process 數時成為主要 overhead。
  - **I/O Time** 大致穩定，對總時間影響不大。
  - **Total Time** 由 93 秒降至約 11 秒，加速顯著，但在 9–12 個 process 後趨於飽和。

  整體而言，在單節點環境下，CPU 計算是最主要的瓶頸並可透過平行化顯著改善，但隨著 process 數增加，Communication 逐漸成為新的限制因素。

- **multi node**
  - Table

Nodes 數	IO Time(s)	CPU Time(s)	COMM Time(s)	Total Time(s)
1	2.501567	4.801131	4.541426	11.846681
2	2.378410	4.651393	4.472242	11.896079
3	2.455183	4.543446	4.189222	11.502132
4	2.281513	4.536355	4.134613	11.185574

  - Profile & Speedup figure



  - Discussion

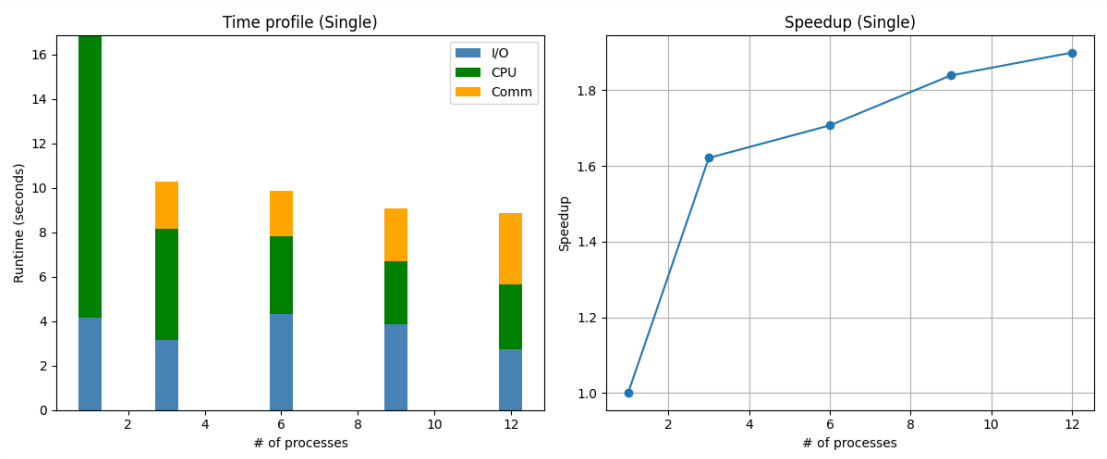
    - **CPU / I/O** 幾乎不變，增加節點對其影響有限。
    - **Communication** 約 4 秒，隨節點數僅小幅下降。
    - **Total Time** 維持在 11–12 秒，加速效果有限，顯示 scalability 受通訊與 I/O 限制。

3. Plots: CPU Optimized

- **Single node**
  - Table

Process數	IO Time(s)	CPU Time(s)	COMM Time(s)	Total Time(s)
1	4.176172	12.674732	0.000000	16.850906
3	3.165841	4.993221	2.130883	10.394385
6	4.340780	3.479623	2.049930	9.871018
9	3.872157	2.838566	2.372012	9.162275
12	2.724606	2.929329	3.216379	8.871686

  - Profile & Speedup figure

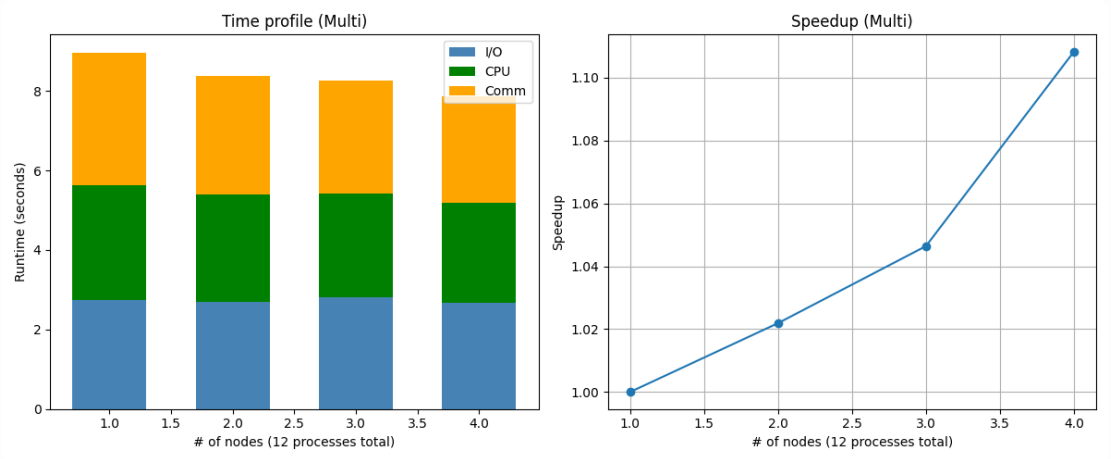


  - Discussion
    - **CPU Time**：
      - Original 在單 process 下需要 ~91 秒，CPU opt 只需 ~12 秒，下降了約 7 倍。
      - 隨 process 增加，兩者 CPU time 都逐步下降，但 CPU opt 的基準更低，使得總時間明顯縮短。
    - **Communication Time**：
      - 在 original 中，COMM 在 3 process 時高達 27 秒，隨後雖然下降但仍維持在 4–18 秒的區間。
      - 在 CPU opt 中，COMM 最多約 3 秒，顯示計算更快 → 分區更早完成 → 通訊量與等待時間同步減少，因此 CPU 優化也對 COMM 開銷帶來改善。
    - **I/O Time**：兩者皆維持在 2–5 秒間，差異不大，非主要瓶頸。

- **multi node**
  - Table

Nodes 數	IO Time (s)	CPU Time (s)	COMM Time (s)	Total Time (s)
1	2.739018	2.891396	3.336203	8.968296
2	2.683609	2.700990	3.001437	8.776066
3	2.819704	2.604792	2.832762	8.570448
4	2.673236	2.502257	2.682200	8.091947

  - Profile & Speedup figure



  - Discussion
    - Original 與 CPU opt 在多節點下的總時間皆維持在 11–12 秒左右，而 CPU opt 能進一步壓低至約 8 秒。
    - 差異主要來自計算與通訊成本的降低，而 I/O 保持不變。

- **綜合**
  - CPU 優化不僅讓排序加速，也讓通訊時間減少，因為 process 間交換與等待的成本下降。
  - Single node 下加速幅度最明顯，總時間從 93 秒 → 8.9 秒。
  - Multi node 下雖然I/O沒有明顯差別，但仍讓 12 processes 的總時間從 ~11 秒降至 ~8 秒。

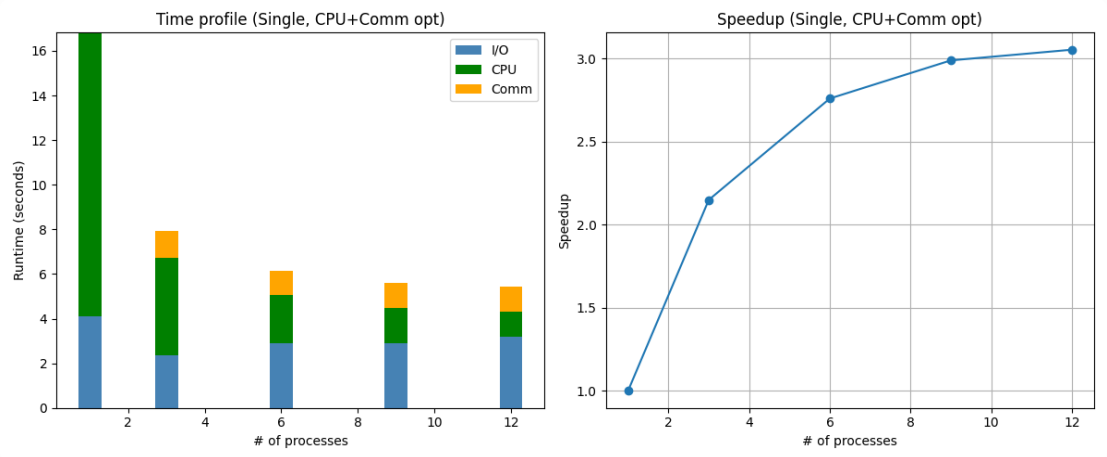
3. Plots: ALL Optimized

此為有cpu, communication optimized情況

- **Single node**
  - Table

Process數	IO Time(s)	CPU Time(s)	COMM Time(s)	Total Time(s)
1	4.086651	12.728994	0.000000	18.489364
3	2.377635	4.339005	1.210473	8.604104
6	2.919950	2.140674	1.090507	6.701919
9	2.906083	1.562215	1.122505	6.187293
12	3.209607	1.114204	1.094249	6.057116

- Profile & Speedup figure

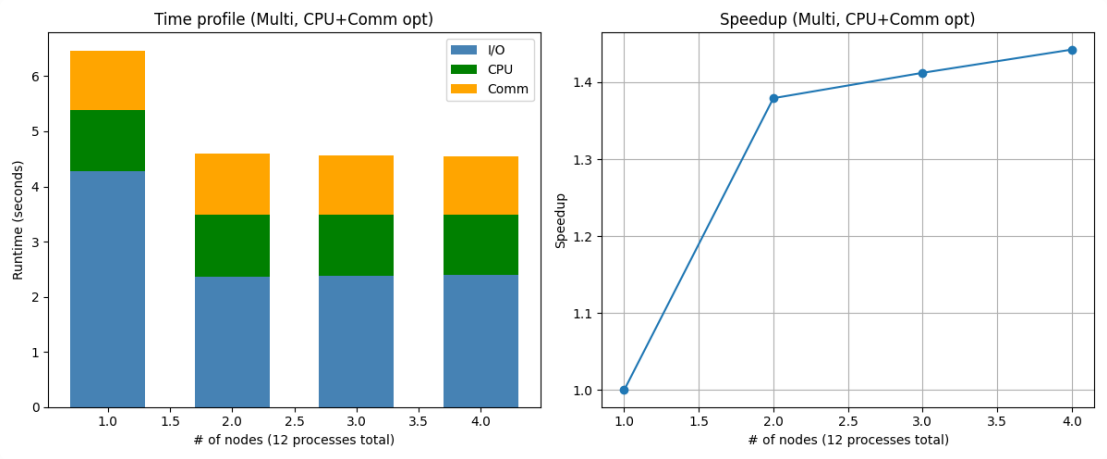


- Discussion
  - Original 在單 process 時約 93 秒，CPU opt 降至 16.8 秒，CPU+Comm opt 到 18.5 秒，可能是因為額外增加的運算導致出現一些多餘的等待。
  - 在多 process 下差異則相當明顯：
    - 12 processes 時，Original 約 11.7 秒，CPU opt 約 8.9 秒，而 CPU+Comm opt 只需 6.0 秒。
    - 通訊時間從 Original 的 4–27 秒大幅下降到 ~1 秒，顯示邊界檢查有效避免不必要的交換。

- **multi node**
  - Table

Nodes 數	IO Time (s)	CPU Time (s)	COMM Time (s)	Total Time (s)
1	4.271747	1.122473	1.073412	7.094876
2	2.362442	1.121190	1.114588	5.144108
3	2.384499	1.112171	1.071661	5.024058
4	2.402328	1.083787	1.058989	4.918962

- Profile & Speedup figure



- Discussion
  - Original 在 4 nodes 時約 11.1 秒，CPU opt 在相同條件下降至 8.1 秒，而 CPU+Comm opt 進一步縮短至 4.9 秒。
  - 在多節點環境下，通訊時間維持在 1 秒左右，遠低於 Original (~4 秒) 與 CPU opt (~2.6–3.3 秒)，從這裡也可以看到邊界檢查有效避免了不必要的交換。
  - I/O 仍然穩定在 2–4 秒，是主要瓶頸。

Experiences / Conclusion

在這次作業中，我有機會實際研究與使用 MPI，這對我來說是很不錯的經驗，因為在平常課程或研究中不太會接觸到如此底層的平行通訊實作。

從實驗數據也可以看到，當平行化程度提升時，CPU time 會隨之下降，但 Communication time 則會逐漸上升，使得總執行時間的下降幅度趨於平緩。這充分顯示了平行程式設計中的 **trade-off**：計算加速與通訊開銷必須平衡，否則平行化效益會受到限制。

在實驗過程中，我也觀察到一個意外的現象：當 CPU time 縮短後，comm time 也隨之降低。這顯示計算加速不僅直接提升了排序效能，還能間接減少通訊的等待與交換成本，真的相當有趣。

不過比較可惜的是 I/O time 無法明顯改善。我也嘗試過使用 `MPI_Alltoall` 等 collective I/O 的方式來測試，但在這個問題規模下並沒有帶來顯著效益，推測可能是資料量還不夠大，無法展現這類方法的優勢。

總結來說，這次作業讓我更了解了 MPI 在不同面向的效能瓶頸，也體驗到從演算法到系統優化的完整過程。