Homework 1: Odd-Even Sort Report

- **Name**: 廖思愷
- **Student ID**: 112062519

Implementation

以下會從我實作出的第一個 "能通過所有測資" 且 "滿足作業規範中的基本需求" 的版本開始描述，再進一步描述如何一步一步優化程式的效能。

Version 1

在這個初始版本中，實作了基本的 odd-even sort 演算法，並且讓此程式能夠在不同數量的 MPI processes 下運行。這一版本能夠通過所有測資，並滿足作業規範中的基本需求。

程式碼

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <sstream>

void oddEvenSort(std::vector<float>& local_data, int local_n, int rank, int size)
{
    MPI_Status status;

    std::sort(local_data.begin(), local_data.end());

    for (int phase = 0; phase <= size; ++phase) {
        int partner;
        if (phase % 2 == 0) {
            // Even phase
            partner = (rank % 2 == 0) ? rank + 1 : rank - 1;
        } else {
            // Odd phase
            partner = (rank % 2 == 0) ? rank - 1 : rank + 1;
        }

        // Adjust partner if it's invalid
        if (partner < 0 || partner >= size) {
            partner = MPI_PROC_NULL;
        }

        // Get partner's local_n
        int partner_n = 0;  // Initialize partner_n to 0
        MPI_Sendrecv(&local_n, 1, MPI_INT, partner, phase,
```

```
                    &partner_n, 1, MPI_INT, partner, phase,
                    MPI_COMM_WORLD, &status);

        if (partner != MPI_PROC_NULL) {
            // Allocate buffer for receiving data
            std::vector<float> temp(partner_n);

            // Exchange data
            MPI_Sendrecv(local_data.data(), local_n, MPI_FLOAT, partner, phase +
1,
                         temp.data(), partner_n, MPI_FLOAT, partner, phase + 1,
                         MPI_COMM_WORLD, &status);

            // Merge the received data
            std::vector<float> merged(local_n + partner_n);
            std::merge(local_data.begin(), local_data.end(), temp.begin(),
temp.end(), merged.begin());

            // Keep the appropriate part
            if (rank < partner) {
                local_data.assign(merged.begin(), merged.begin() + local_n);
            } else {
                local_data.assign(merged.end() - local_n, merged.end());
            }
        }
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc != 4) {
        if (rank == 0) {
            std::cerr << "Usage: " << argv[0] << " <n> <input_file> <output_file>"
<< std::endl;
        }
        MPI_Finalize();
        return 1;
    }

    int n = std::atoi(argv[1]);
    const char* input_file = argv[2];
    const char* output_file = argv[3];

    int local_n = n / size;
    int remainder = n % size;
    if (rank < remainder) {
        local_n++;
    }
```

```
    std::vector<float> local_data(local_n);

    // Read input file using MPI-IO
    MPI_File in_file;
    MPI_File_open(MPI_COMM_WORLD, input_file, MPI_MODE_RDONLY, MPI_INFO_NULL,
&in_file);

    MPI_Offset offset = rank * (n / size) * sizeof(float);
    if (rank < remainder) {
        offset += rank * sizeof(float);
    } else {
        offset += remainder * sizeof(float);
    }

    MPI_File_read_at(in_file, offset, local_data.data(), local_n, MPI_FLOAT,
MPI_STATUS_IGNORE);
    MPI_File_close(&in_file);

    std::cout << "before sort" << std::endl;

    // Perform odd-even sort
    oddEvenSort(local_data, local_n, rank, size);

    // Write output file using MPI-IO
    MPI_File out_file;
    MPI_File_open(MPI_COMM_WORLD, output_file, MPI_MODE_CREATE | MPI_MODE_WRONLY,
MPI_INFO_NULL, &out_file);

    MPI_File_write_at(out_file, offset, local_data.data(), local_n, MPI_FLOAT,
MPI_STATUS_IGNORE);
    MPI_File_close(&out_file);

    MPI_Finalize();
    return 0;
}
```

1. 如何處理不固定的輸入數量及 MPI processes

- **輸入數量 `n` 的分配：**
  - 程式接收三個命令行參數：`n`（要排序的浮點數數量）、`input_file`（輸入檔案名稱）、`output_file`（輸出檔案名稱）。
  - 使用 `MPI_Comm_size(MPI_COMM_WORLD, &size)` 取得 process 的總數。
  - 將 `n` 均分給每個 MPI process，並分配 `local_n = n / size` 個數值給每個 process。
  - 若 `n` 無法被 process 數整除，則將餘數 (`remainder = n % size`) 依序分配給 rank 較低的 process，使得 rank 低的 process 有 `local_n + 1` 個元素，以確保所有元素都被處理。
- **偏移量 `offset` 的計算：**
  - 根據每個 process 的 rank 計算其在檔案中的偏移量，以便從 `input_file` 讀取各自負責的區塊。
  - `offset` 計算方式：

1. **基礎偏移量計算：** 每個 process 的基礎偏移量是其 rank 乘上「每個 process 負責的資料數量」的大小（以位元組計算）。也就是說，若每個 process 負責 `n / size` 個元素，則基礎偏移量為 `rank * (n / size) * sizeof(float)`。
2. **考慮餘數的調整：** 如果總元素數 `n` 無法被 process 數 `size` 整除，則會產生餘數 `remainder`。這些多出的元素會依序分配給 rank 較低的 processes，因此每個 rank 小於 `remainder` 的 process 還需額外偏移 `rank * sizeof(float)`，以確保偏移量正確反映各個 process 負責的資料區塊。

```
MPI_Offset offset = rank * (n / size) * sizeof(float);
if (rank < remainder) {
    offset += rank * sizeof(float);
} else {
    offset += remainder * sizeof(float);
}
```

- 該偏移量會讓每個 process 正確地從檔案中讀取其負責的資料，並確保資料沒有重疊或遺漏。

2. 程式中的排序機制

- **Odd-Even Sort 邏輯：**
  - 此演算法交替執行 **even phase** 和 **odd phase**，每個 phase 中相鄰的 process 會交換資料以完成排序。
  - 每個 process 會在一開始進行一次 local sort
  - **Even Phase：**
    - 偶數 rank 的 process 與右邊的奇數 rank process 交換資料。
    - 奇數 rank 的 process 與左邊的偶數 rank process 交換資料。
  - **Odd Phase：**
    - 奇數 rank 的 process 與右邊的偶數 rank process 交換資料。
    - 偶數 rank 的 process 與左邊的奇數 rank process 交換資料。
- **交換 partner 的元素數量：**
  - 在每個 phase 開始時，兩個 process 先交換彼此 `local_n` 的數量資訊，以確定接下來要傳遞的元素數量。
  - 每個 process 在 partner 數量確定後，使用 `MPI_Sendrecv` 交換各自的 `local_data`，然後使用 `std::merge` 函數將自己的資料和 partner 的資料合併並進行排序。
- **合併後的資料篩選：**
  - 當 partner 資料接收並合併後，透過比較 rank 值決定要保留哪一部分資料。
  - 若 rank < partner，則 process 保留合併後的前半部分（較小的 `local_n` 個數據），否則保留後半部分（較大的 `local_n` 個數據）。

```
if (rank < partner) {
    local_data.assign(merged.begin(), merged.begin() + local_n);
} else {
```

```
        local_data.assign(merged.end() - local_n, merged.end());
    }
```

3. MPI IO 與邊界 process 的處理

- **MPI I/O 操作：**
    - 程式使用 `MPI_File_read_at` 和 `MPI_File_write_at` 函數從 `input_file` 讀取並寫入 `output_file`。
- **邊界 process 的處理：**
    - 每個 phase 根據奇偶性來分配 partner，並使用 `MPI_PROC_NULL` 處理無效的 partner，像是 rank 0 沒有左邊的 鄰居，最後一個 rank 沒有右邊的鄰居，以避免邊界 process 進行無效通訊。

```
if (partner < 0 || partner >= size) {
    partner = MPI_PROC_NULL;
}
```

實驗結果

使用 `hw1-judge` 測試後，Version 1 對於 40 測資的總執行時間為 **187.12 秒**。

Version 2

假設

在 Version 1 中使用了 `std::vector` 作為資料容器，但 `std::vector` 會有額外的記憶體管理和初始化開銷，特別是進行大量資料的交換和合併時。假設將動態分配方式改為 C-style 的 `malloc` 和 `free` 來手動管理記憶體，應該可以降低這些額外負擔，提高運行效率。

實驗結果

使用 `hw1-judge` 測試後，Version 2 的總執行時間為 **165.26 秒**，相比 Version 1 的 **187.12 秒**，整體效能有所提升。

Version 3

假設

根據 `spreadsort` 的官方文件，在特定情況下，`spreadsort` 比 `std::sort` 更快，尤其是在以下幾種情況：

1. **大量元素的排序**：當排序元素數量較多時（如 N >= 10,000），`spreadsort` 相較於 `std::sort` 更快速。
2. **較慢的比較函數**：例如浮點數排序在 x86 處理器上比較耗時，而 `spreadsort` 對浮點數進行了優化，預期能加快排序速度。

3. **已排序資料：**`spreadsort` 有優化機制，在遇到已排序的資料時可提早退出排序程序。

在此假設下，將程式的排序部分從 `std::sort` 改為 `spreadsort` 應能利用上述優勢，加速排序並降低總執行時間。由於這個版本針對大量浮點數資料進行排序，符合 `spreadsort` 在處理大量浮點數時的最佳化情境，預期將進一步優化效能。

實驗結果

使用 `hw1-judge` 測試後，Version 3 的總執行時間為 **145.65 秒**，相比 Version 2 的 **165.26 秒**，整體效能有所提升。

Version 4

假設

在前面的版本中，執行每個 phase 的資料交換後會將 `local_data` 和 partner 的資料合併排序並保留結果。然而，針對 `oddEvenSort` 的最後階段操作，實際上不需要排序整個合併後的陣列，而是僅需保留 `local_n` 個適當的元素。假設在每個 phase 的最後階段僅排序並保留 `local_n` 個元素，可以進一步減少不必要的排序運算量，提升整體運行效能。

實驗結果

使用 `hw1-judge` 測試後，Version 4 的總執行時間為 **117.75 秒**，相比 Version 3 的 **145.65 秒**，整體效能有所提升。

Version 5

假設

在前面的版本中，每個 phase 的排序會直接交換相鄰 process 的 `local_data` 資料。然而，如果左邊的 process 之最大元素小於等於右邊 process 的最小元素，即代表這兩個區段已排序完成，無需進行資料交換。因此，假設先交換邊界元素 (左邊 rank 傳最大元素，右邊 rank 傳最小元素) ，並檢查是否需要進行合併，可以在不必要時略過交換操作，進一步提升效能。

實驗結果

使用 `hw1-judge` 測試後，Version 5 的總執行時間為 **103.53 秒**，相比 Version 4 的 **117.75 秒**，整體效能有所提升。

Version 6

假設

在 Version 5 中，每個 phase 都會在資料交換時動態分配緩衝區來儲存 partner 傳遞過來的資料，並在合併後分配另一個緩衝區來儲存合併結果，然後將結果更新到本地的資料陣列中。然而，這些分配與釋放操作在每次 phase 中重複進行，帶來了額外的記憶體管理開銷。如果在整個排序過程中只進行一次這兩個緩衝區的分配，並在每個 phase 直接重複利用已分配的緩衝區，即可減少頻繁分配和釋放記憶體的開銷，從而提升效能。

實驗結果

使用 `hw1-judge` 測試後，Version 6 的總執行時間為 **95.34 秒**，相比 Version 5 的 **103.53 秒**，整體效能有所提升。

Version 7

假設

在 Version 5 中，程式在每個phase交換local data前，檢查自己與 neighbor process 的邊界元素，如果左邊 process 的最大元素小於等於右邊 process 的最小元素，則不需進行合併。在 Version 7 中，進一步假設：如果左邊 process 的最小元素大於等於右邊 process 的最大元素，則左邊的資料全部大於右邊的資料，因為它們已經各自排序完成。這表示兩個 process 的資料彼此間無需再進行合併，只需交換位置即可。但是需注意兩個 process 各自擁有的 local data 數量為多少，以我分配資料給各 process 的策略：

- 若 input size `n` 無法被 process 數整除，則將餘數依序分配給 rank 較低的 process，使得 rank 低的 process 有 `local_n + 1` 個元素

所以兩個 process 擁有的元素數量有兩種情況：

1. 數量一樣
2. rank 低的 process 比 rank 高的 process 多 1 個元素

當兩個 process 擁有相同數量的元素時，可以直接將 neighbor data 的元素複製更新為 local data；若兩者相差一個元素數量時，仔細處理多出來的那個元素即可，這種優化能避免不必要的資料比較和合併操作。

實驗結果

使用 `hw1-judge` 測試後，Version 7 的總執行時間為 **90.88 秒**，相比 Version 6 的 **95.34 秒**，整體效能有所提升。

最終程式碼

```
#include <mpi.h>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <cstdlib>
#include <boost/sort/spreadsort/spreadsort.hpp>

void oddEvenSort(float*& local_data, int local_n, int rank, int size, float*&
partner_data, float*& merged_data) {
    MPI_Status status;

    boost::sort::spreadsort::spreadsort(local_data, local_data + local_n);

    for (int phase = 0; phase <= size; ++phase) {
        int partner;
        if (phase % 2 == 0) {
            // Even phase
```

```
            partner = (rank % 2 == 0) ? rank + 1 : rank - 1;
        } else {
            // Odd phase
            partner = (rank % 2 == 0) ? rank - 1 : rank + 1;
        }

        // Adjust partner if it's invalid
        if (partner < 0 || partner >= size) {
            partner = MPI_PROC_NULL;
        }

        if (partner != MPI_PROC_NULL) {
            // Exchange boundary elements to check if merge is necessary
            float local_first, local_last;
            float partner_boundary_first, partner_boundary_last;
            int partner_n = 0;

            // Get partner's local_n
            MPI_Sendrecv(&local_n, 1, MPI_INT, partner, phase,
                         &partner_n, 1, MPI_INT, partner, phase,
                         MPI_COMM_WORLD, &status);

            if (rank < partner) { // left rank
                local_last = local_data[local_n - 1];
                MPI_Sendrecv(&local_last, 1, MPI_FLOAT, partner, phase + 1,
                             &partner_boundary_first, 1, MPI_FLOAT, partner, phase
+ 1,
                             MPI_COMM_WORLD, &status);

                if (local_last <= partner_boundary_first) {
                    // Skip merge
                    continue;
                }

                local_first = local_data[0];
                MPI_Sendrecv(&local_first, 1, MPI_FLOAT, partner, phase + 2,
                             &partner_boundary_last, 1, MPI_FLOAT, partner, phase
+ 2,
                             MPI_COMM_WORLD, &status);

                if (local_first >= partner_boundary_last) {
                    // Exchange data
                    MPI_Sendrecv(local_data, local_n, MPI_FLOAT, partner, phase +
3,
                                 partner_data, partner_n, MPI_FLOAT, partner,
phase + 3,
                                 MPI_COMM_WORLD, &status);

                    if (local_n == partner_n) {
                        std::memcpy(local_data, partner_data, local_n *
sizeof(float));
                        continue;
                    } else { // local_n - partner_n == 1
                        std::memcpy(local_data, partner_data, partner_n *
```

```
sizeof(float));
                        local_data[local_n - 1] = local_first;
                        continue;
                    }
                }
            } else { // right rank
                local_first = local_data[0];
                MPI_Sendrecv(&local_first, 1, MPI_FLOAT, partner, phase + 1,
                             &partner_boundary_last, 1, MPI_FLOAT, partner, phase +
1,
                             MPI_COMM_WORLD, &status);

                if (local_first >= partner_boundary_last) {
                    // Skip merge
                    continue;
                }

                local_last = local_data[local_n - 1];
                MPI_Sendrecv(&local_last, 1, MPI_FLOAT, partner, phase + 2,
                             &partner_boundary_first, 1, MPI_FLOAT, partner, phase
+ 2,
                             MPI_COMM_WORLD, &status);

                if (local_last <= partner_boundary_first) {
                    // Exchange data
                    MPI_Sendrecv(local_data, local_n, MPI_FLOAT, partner, phase +
3,
                                 partner_data, partner_n, MPI_FLOAT, partner, phase
+ 3,
                                 MPI_COMM_WORLD, &status);

                    if (local_n == partner_n) {
                        std::memcpy(local_data, partner_data, local_n *
sizeof(float));
                        continue;
                    } else { // local_n - partner_n == -1
                        std::memcpy(local_data, partner_data + 1, local_n *
sizeof(float));
                        continue;
                    }
                }
            }

            // Exchange data
            MPI_Sendrecv(local_data, local_n, MPI_FLOAT, partner, phase + 2,
                         partner_data, partner_n, MPI_FLOAT, partner, phase + 2,
                         MPI_COMM_WORLD, &status);

            int i = 0, j = 0;
            if (rank < partner) {
                // Keep the smallest local_n elements
                int count = 0;
                while (count < local_n) {
                    if (local_data[i] < partner_data[j]) {
```

```
                        merged_data[count++] = local_data[i++];
                    } else {
                        merged_data[count++] = partner_data[j++];
                    }
                }
            } else {
                // Keep the largest local_n elements
                int count = local_n - 1;
                i = local_n - 1;
                j = partner_n - 1;
                while (count >= 0) {
                    if (local_data[i] > partner_data[j]) {
                        merged_data[count--] = local_data[i--];
                    } else {
                        merged_data[count--] = partner_data[j--];
                    }
                }
            }

            // Update local_data to point to merged_data
            float* temp_ptr = local_data;
            local_data = merged_data;
            merged_data = temp_ptr;
        }
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc != 4) {
        if (rank == 0) {
            std::cerr << "Usage: " << argv[0] << " <n> <input_file> <output_file>"
<< std::endl;
        }
        MPI_Finalize();
        return 1;
    }

    int n = std::atoi(argv[1]);
    const char* input_file = argv[2];
    const char* output_file = argv[3];

    int local_n = n / size;
    int remainder = n % size;
    if (rank < remainder) {
        local_n++;
    }

    float* local_data = (float*)malloc(local_n * sizeof(float));
```

```
    float* partner_data = (float*)malloc((local_n + 1) * sizeof(float));
    float* merged_data = (float*)malloc(local_n * sizeof(float));

    // Read input file using MPI-IO
    MPI_File in_file;
    MPI_File_open(MPI_COMM_WORLD, input_file, MPI_MODE_RDONLY, MPI_INFO_NULL,
&in_file);

    MPI_Offset offset = rank * (n / size) * sizeof(float);
    if (rank < remainder) {
        offset += rank * sizeof(float);
    } else {
        offset += remainder * sizeof(float);
    }

    MPI_File_read_at(in_file, offset, local_data, local_n, MPI_FLOAT,
MPI_STATUS_IGNORE);
    MPI_File_close(&in_file);

    // Perform odd-even sort
    oddEvenSort(local_data, local_n, rank, size, partner_data, merged_data);

    // Write output file using MPI-IO
    MPI_File out_file;
    MPI_File_open(MPI_COMM_WORLD, output_file, MPI_MODE_CREATE | MPI_MODE_WRONLY,
MPI_INFO_NULL, &out_file);

    MPI_File_write_at(out_file, offset, local_data, local_n, MPI_FLOAT,
MPI_STATUS_IGNORE);
    MPI_File_close(&out_file);

    free(local_data);
    free(partner_data);
    free(merged_data);

    MPI_Finalize();
    return 0;
}
```

Experiment & Analysis

Methodology

System Spec

使用課程提供的 Apollo Cluster

Performance Metrics

本次實驗使用 **NVIDIA Nsight Systems** 來進行記錄與分析，以深入探討 Odd-Even Sort 演算法在 MPI 環境下的效能瓶頸與執行效率。具體的測量過程與指標如下：

1. **記錄與轉檔：**
    - 每次執行測試時，使用 `nsys profile` 指令進行記錄，將每個 process 的運行過程記錄成 `.nsys-rep` 格式的檔案。
    - 針對產生的 `.nsys-rep` 檔案，執行 `nsys stats -r mpi_event_trace --format csv` 指令，將其中 MPI 通訊事件的相關資訊轉換成 `.csv` 格式的檔案，以便後續進行資料處理和視覺化分析。
2. **數據分析與指標提取：**
    - 針對每個進程的 `.csv` 檔案，分別計算程式執行中的**計算時間**、**通信時間**和**I/O 時間**，以量化每個指標的平均耗時。
    - 將各進程的數據整理並計算平均值，得到不同進程數下的計算、通信和 I/O 的時間分布，進而揭示效能表現與瓶頸。
3. **效能指標定義：**
    - **計算時間（CPU Time）**：計算時間反映在排序及合併資料過程中所花費的時間，包含各 process 進行 local data 排序和 phase 間資料合併所需的計算操作。從總執行時間（由 `MPI_Init` 開始到 `MPI_Finalize` 結束的時間差異）中扣除通信與 I/O 時間，以得到計算時間。
    - **通信時間（Communication Time）**：通信時間主要測量 process 間在進行資料交換時的耗時，包括各 process 與其 partner 進行 `MPI_Sendrecv` 通訊的時間。將每次通訊的執行時間匯總，並取平均值以展示不同進程數下的通信開銷。
    - **I/O 時間（I/O Time）**：I/O 時間測量程式從檔案讀取及寫入資料的耗時，通過 `MPI_File_open`、`MPI_File_read_at`、`MPI_File_write_at` 及 `MPI_File_close` 等操作進行記錄。將所有進程的 I/O 時間平均後呈現在圖表中，以反映檔案 I/O 的負載。
4. **圖表呈現：**
    - **執行時間分佈圖**：將單節點與多節點下不同進程數的計算、通信和 I/O 時間，分別以堆疊長條圖形式展示，以直觀對比各階段的耗時分布。
    - **Speedup 圖**：根據 baseline（最小進程數下的執行時間）計算出不同進程數的加速比（Speedup Factor），繪製理想加速線與實際加速線，展示 Odd-Even Sort 在 MPI 並行環境下的可擴展性。

Plots: Speedup Factor & Profile

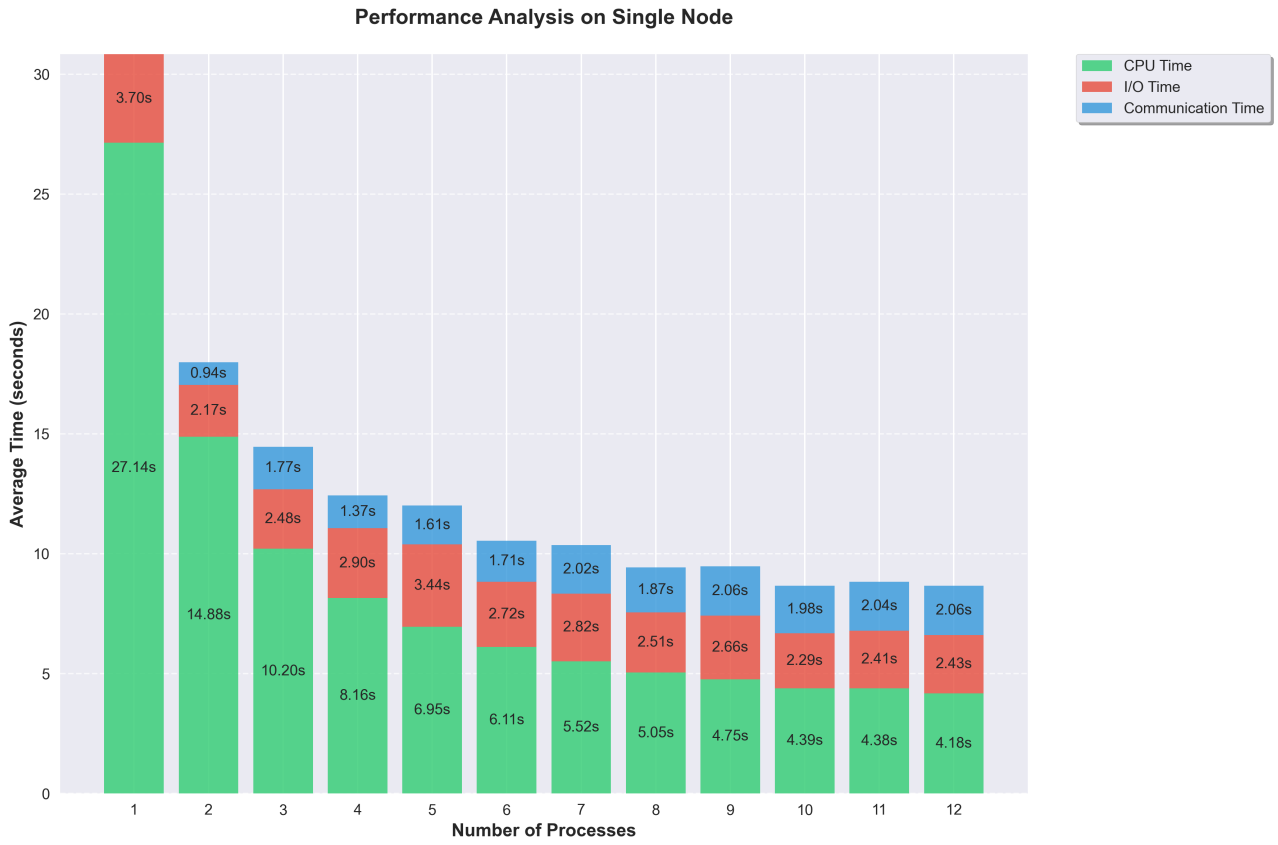
Experimental Method

- Test Case Description
    - 本次實驗選擇使用作業提供的第 39 筆測資進行測試，因為該測資擁有最大的資料量，共 536,870,864 筆浮點數，適合作為 strong scalability 實驗的測試案例，以分析程式在不同進程數下的效能表現。
    - 使用 Version 7 的程式進行測試
- Parallel Configurations
    - 為了測試程式的並行效能，本次實驗分別進行了單節點和多節點配置。實驗使用了以下的並行配置：

- **單節點配置**：在單節點上逐步增加進程數（從 1 至 12 個），每個進程在相同的 node 上運行。
- **多節點配置**：在三個節點上進行實驗，並逐步增加進程數（3、6、9 等到 36 個）。

Analysis of Results

- 實驗一：測試在單節點上設置不同的 process 數 (1 ~ 12 個)，程式的 CPU time、communication time、I/O time 會有什麼變化，以及程式整體的 speedup 是否符合理想情況。

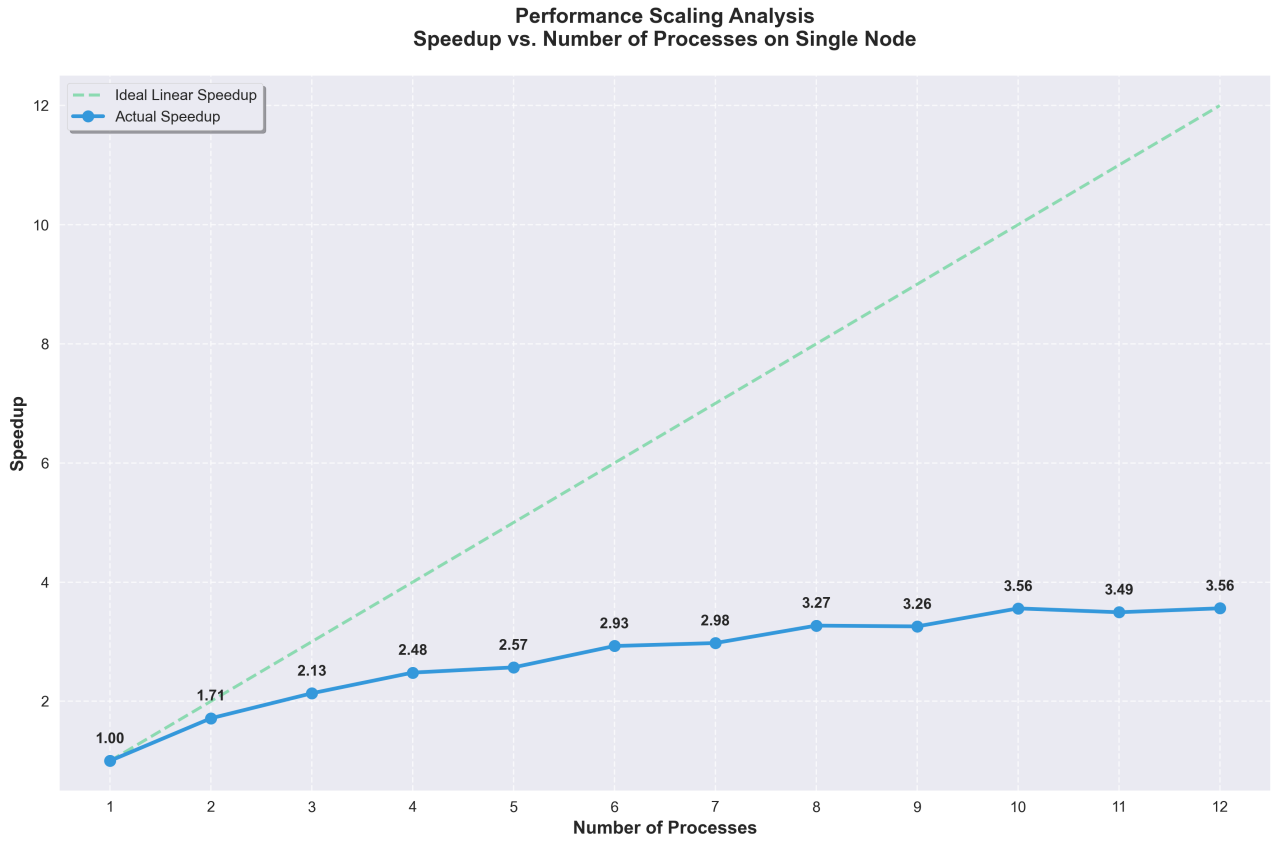


  - CPU Time:
    - 隨著 process 數量的增加，CPU time（綠色區塊）大幅度減少，這符合預期，因為更多的 processes 應該能分攤排序和合併資料的計算工作。然而，在較高的 process 數量下，CPU 時間的下降趨緩，表示計算已經達到某種瓶頸，並非線性縮減。
    - 在較高的 process 數量下，為何難以達到線性縮減？
      - 我推測是因為當 process 數增加時，雖然每個 process 被分配到的 local data 減少了，但是，需要跟 neighbor 交換 data 以及 merge 的次數變多了，增加的 merge 次數會逐漸抵銷資料量減少帶來的好處，導致難以達到線性縮減。
  - I/O Time:
    - 這邊原本的預期是，I/O time (紅色區塊) 應該要隨著 process 增加而下降，因為每個 process需要讀取跟寫入的資料變少了，但令人意外的是，並沒有觀察到明顯的下降趨勢，圖表呈現出略微波動的趨勢，我推測可能的原因如下：
      - **MPI-IO 的實作效率：**
        - 雖然 BeeGFS 是一種高效的平行文件系統，但 OpenMPI 的 MPI-IO 實作可能無法充分利用 BeeGFS 的性能。不同 MPI 實作對文件系統的支援程度不同，

OpenMPI 在使用 MPI-IO 時可能無法完全達到 BeeGFS 的理論性能。這會導致 I/O 操作的瓶頸，尤其是在多 process 同時執行 I/O 操作的情況下。

- **文件系統的同步與鎖機制：**
  - 在 BeeGFS 中，雖然理論上每個 process 都可以平行地進行 I/O 操作，但如果某些 I/O 操作需要文件系統層面的同步或鎖，可能會導致額外的等待時間。隨著 process 數量增加，這些同步或鎖開銷也可能增加，抵消了資料量減少的好處。
- **單節點上的 I/O 頻寬限制：**
  - 如果在單節點上運行所有 process，單個節點的 I/O 頻寬可能會成為瓶頸。即使 BeeGFS 支援多個 process 的平行讀寫操作，但單節點的磁碟或網路頻寬無法同時支援多個高效 I/O 操作，也會導致 I/O 時間無法隨 process 增加而線性下降。但是在後續的多節點實驗中也有觀察到類似現象，所以我想這個應該不是主要原因。

- Communication Time:
  - Communication Time（藍色區塊）隨著 process 數的增加而呈現增加的趨勢，這符合預期。因為在 Odd-Even Sort 中，進程間需要大量的 MPI_Sendrecv 通訊來交換資料，這些通信操作會隨著 process 數量的增加而增加。



根據這張 Speedup 圖，我們可以觀察到 Odd-Even Sort 在單節點上的 scalibility，以及隨著 process 數量增加，實際加速效果逐漸趨於平緩的現象。這裡的分析如下：

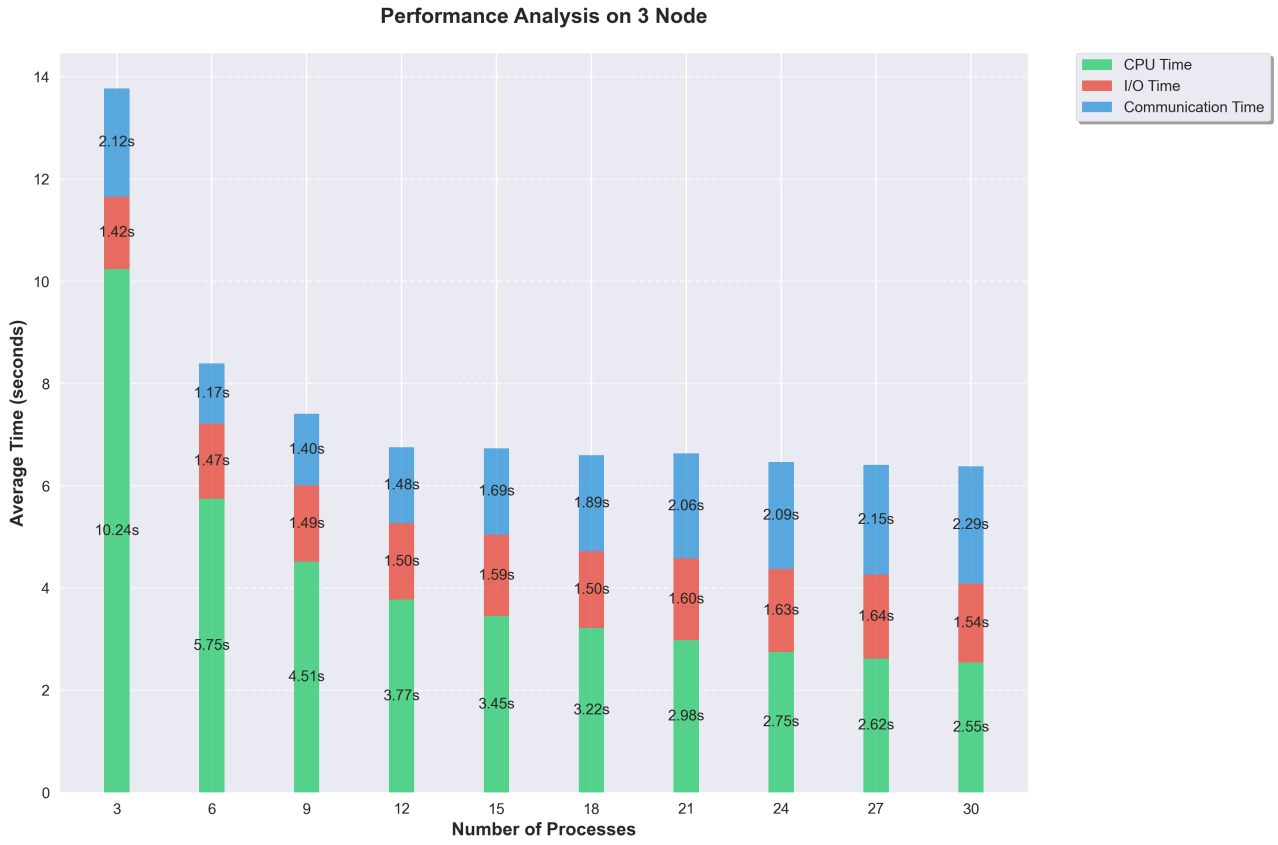
- **理想加速線與實際加速線的差距：** 理想情況下，隨著 process 數增加，速度應該以線性比例提升（即理想線性加速線，綠色虛線）。然而，圖中顯示，實際加速線（藍色實線）明顯低於理想線性加速線，顯示出無法達到完美的線性加速。這意味著隨著 process 增加，某些開銷逐漸成為性能瓶頸，限制了整體的加速效果。

- **通訊與同步開銷的影響：** 根據之前的分析，隨著 process 數增加，每個 process 的資料量雖然減少，但 Odd-Even Sort 中需要與相鄰 process 交換資料的次數增加了，這意味著通訊開銷隨 process 增加而不斷上升。此外，MPI_Sendrecv() 需要等待 neighbor process 同步完成才能繼續執行，這種同步開銷會對加速效果產生顯著影響，特別是在 process 數量多時，通訊與同步的負擔顯得更為顯著。

- **I/O 開銷的影響：** 根據之前的 I/O 時間分布圖，I/O 開銷並未隨 process 數顯著減少，這可能進一步限制了加速效果。

- **單節點的限制：** 在單節點上執行多個 process 時，系統的資源（例如 CPU 快取、記憶體頻寬和 I/O 頻寬）會被多個 process 共享。隨著 process 數增加，這些資源的競爭加劇，可能會導致計算、通訊和 I/O 操作的效率降低。特別是在多 process 間頻繁交換資料的情況下，這種資源競爭會進一步影響加速效果。

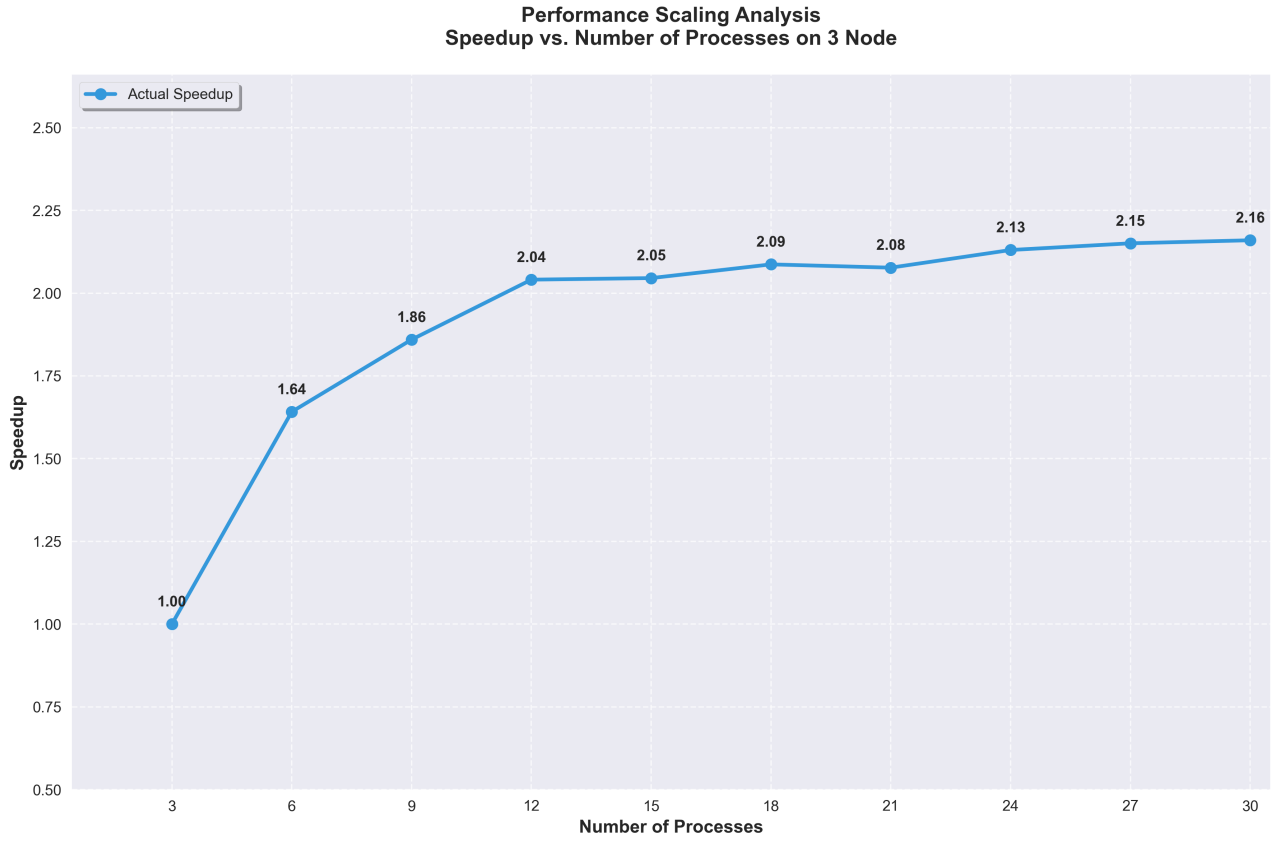
- 實驗二：測試在多節點上（三個 nodes）設置不同的 process 數（3、6、9 等到 36 個），程式的 CPU time、communication time、I/O time 會有什麼變化，以及程式整體的 speedup 是否符合理想情況。



多節點的分布情況整體跟單節點很像，分析如下：

- CPU Time:
  - 隨著 process 數量的增加，CPU time（綠色區塊）逐漸減少，這符合預期，因為每個 process 負責的資料量隨之減少。
  - 但是，在 process 數增加到 18 以後，CPU time 的減少速度趨於緩慢。這表明增加更多 process 並未顯著提高計算效率，與單節點的情況相同，可能是因為當 process 數增加時，雖然每個 process 被分配到的 local data 減少了，但是，需要跟 neighbor 交換 data 以及 merge 的次數變多了，增加的 merge 次數會逐漸抵銷資料量減少帶來的好處，導致難以達到線性縮減。

- I/O Time:
  - I/O time（紅色區塊）並未隨 process 數增加而顯著下降，反而在較高的 process 數時呈現出緩慢上升的趨勢。
  - 推測可能與以下原因有關：
    - **BeeGFS 的同步開銷**：在多節點環境中，BeeGFS 可能需要執行更多的同步操作，尤其是當多個 process 同時進行 I/O 操作時，文件系統的鎖機制會增加額外的等待時間。
    - **OpenMPI 的 MPI-IO 效率限制**：OpenMPI 的 MPI-IO 實作可能無法充分利用 BeeGFS 的平行性能，在多節點環境下，這一限制更為顯著，導致 I/O 時間未隨 process 數增加而顯著下降。
- Communication Time:
  - 與單節點的情況相同，Communication Time（藍色區塊）隨著 process 數的增加而呈現增加的趨勢，這符合預期。因為在 Odd-Even Sort 中，進程間需要大量的 MPI_Sendrecv 通訊來交換資料，這些通信操作會隨著 process 數量的增加而增加。



上圖是在多節點的環境下，增加 process 數的 speedup 情況。在理想情況下，隨著 process 數增加，speedup 應該呈現線性增長。然而，實際加速比（藍色實線）在 process 數量超過 12 後趨於平緩，並最終在約 2.1 倍左右達到飽和。這表明隨著 process 數量增加，效能的提升受到瓶頸限制，原因如同以上對於分布圖的分析，難以達到理想的線性加速。

- 單節點與多節點的比較 根據觀察單節點與多節點的圖表，我們可以比較兩者在 process 數為 3、6、9、12 時的各項耗時情況。以下是詳細比較和分析：

Process 數	節點	CPU Time	I/O Time	Communication Time

Process 數	節點	CPU Time	I/O Time	Communication Time
3	單節點	10.20s	2.48s	1.77s
	多節點	10.24s	1.42s	2.12s
6	單節點	6.11s	2.72s	1.71s
	多節點	5.75s	1.47s	1.17s
9	單節點	4.75s	2.66s	2.06s
	多節點	4.51s	1.49s	1.40s
12	單節點	4.18s	2.43s	2.06s
	多節點	3.77s	1.50s	1.48s

- **CPU Time 的比較：**
  - 在 process 數較高的情況下（6、9 和 12），多節點環境下的 CPU Time 似乎低於單節點，這可能是因為多節點中每個節點使用較少的計算資源，減少了資源競爭。
  - 當 process 數為 3 時，單節點和多節點的 CPU Time 差異不大，這可能意味著在 process 數較少的情況下，單節點的計算資源尚未成為瓶頸，兩者的執行效率相近。
- **I/O Time 的比較：**
  - 在多節點環境中，I/O Time 大多低於單節點，尤其在 process 數增加的情況下。這可能與 BeeGFS 的平行處理能力有關，BeeGFS 能夠在多節點間分擔 I/O 負載，從而減少 I/O 時間。不過這只是推測，實際情況可能還取決於 BeeGFS 與 MPI-IO 的配合效率。
- **Communication Time 的比較：**
  - 多節點環境下的 Communication Time 在 process 數較高時（6、9 和 12）似乎略低於單節點。這可能是因為多節點間的資料傳輸在 HPC 平台上通常使用 InfiniBand 等高速網路，其傳輸速率可能優於單節點內的記憶體總線。

Optimization Strategies

基於以上的實驗分析，針對 Odd-Even Sort 演算法，我提出以下可能的優化策略：

1. **透過指標操作進行資料交換**
   - 在 Odd-Even Sort 的實現中，我目前使用 `std::memcpy` 將 partner 資料複製到本地陣列。如果能夠改用指標操作，讓 `local_data` 直接指向目標 array，便可避免多餘的複製操作。
   - 這種改進的好處在於避免了不必要的記憶體拷貝。當 partner 與本地資料大小相等時，將 `local_data` 指向 `partner_data` 可直接使用已接收的資料，省去複製開銷。
2. **使用 SIMD 指令加速 local sort**
   - SIMD（Single Instruction, Multiple Data）指令可以用來優化本地排序部分。透過實作 SIMD-friendly 的 local sort，可以加速本地資料的排序過程。

- 例如，若實作一個簡單的 SIMD-based radix sort，可以使用 Intel AVX、SSE 等指令來處理浮點數的批次排序。這樣的操作可以顯著加快排序，尤其是在每個 process 資料量大的情況下。

3. **異步化通訊和計算**

- 使用 MPI 的非阻塞通訊（如 `MPI_Isend` 和 `MPI_Irecv`），可以實現通訊和計算的重疊。當 process 在等待 partner 的資料時，可以先執行一些不依賴於 partner 資料的本地計算。

4. **通訊路徑優化**

- HPC 系統的 InfiniBand 通訊架構中，不同路徑的延遲可能有所差異。透過 `MPI_Cart_create`，可以設計 process 的拓撲結構，讓相鄰 process 的通訊經由較低延遲的路徑，這樣可以進一步減少通訊延遲。
- 例如，將 process 排列為一維拓撲結構，以使相鄰 process 排在物理網路的近處。

5. **啟用 NUMA 支援**

- 在多節點環境下，每個節點的計算資源通常是多個 NUMA（Non-Uniform Memory Access）節點。透過 OpenMPI 將 process 綁定到特定的 NUMA 節點，可確保每個 process 優先使用本地記憶體區域，從而減少記憶體存取延遲。

Experience & Conclusion

這次作業是我第一次實作並優化 MPI 平行程式，我發現要實作出一個能通過所有測資的版本並不難，難的是要如何優化程式，還有如何使用 profiler 工具來分析程式的效能，有時用 profiler 測量出來的數據會跟預期差很多，這時候就需要思考各種可能的原因並一一排查，這個部分需要花大量的時間，我覺得我前面花了太多時間在思考並實作一些可能可以優化程式效能的方法，太晚開始熟悉 profiler 工具還有做實驗，不然應該可以做更多實驗來分析程式的瓶頸，我覺得 spec 上要求的測量 CPU time、communication time 以及 I/O time 還是太粗略了，很難找到程式真正的瓶頸。不過我還是在這過程中學到非常多有關 MPI 的知識以及實作方法，感謝老師跟助教這麼用心的出這份作業！