# hw1 Odd-Even Sort

**ID: 112062520 / Name: 戴維恩**

## Implementation

### 1. original

#### Data Distribution

在程式的一開始，根據測資指定的 process 數量分配資料。分配方式為 $n/size$，$n$ 為資料總長度，$size$ 為 process 總數；若有餘數 $r$ 則平均分配給開頭 $r$ 個 process，如此一來每個 rank 最多資料量為 $n/size + 1$。

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

#### Read Data

當每個 rank 需要處理的資料被決定好，接下來就是要從測資檔案中讀取特定位置、長度的資料。首先根據每個 rank 預計處理的資料長度，計算各自要從哪個位置開始讀取，接著使用 `MPI_File` API。

```
int offset = (s_n * rank + (rank < remainder ? rank : remainder)) * sizeof(float
float* local_data = (float*) malloc(local_n * sizeof(float));
MPI_File input_file, output_file;
MPI_File_open(MPI_COMM_WORLD, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &i
MPI_File_read_at(input_file, offset, local_data, local_n, MPI_FLOAT, MPI_STATUS_
MPI_File_close(&input_file);
```

#### Local Data Sorting

每個 rank 讀取自己要處理的 data 後，先對其做 local 排序（由小至大）。

```
std::sort(local_data, local_data + local_n);
```

#### Odd-Even Sort

接下來就要進入作業重點，每個 rank 各自與左右相鄰 rank 交換 data 並重新排序。偶數 rank 在 `Odd Phase` 時要與自己左邊的 rank 交換資訊；`Even Phase` 時要與自己右邊的 rank 交換資訊。奇數 rank 則相反。

```
int r_odd, r_even;
int even_rank = (rank % 2 == 0);
```

```
r_odd = even_rank ? rank - 1 : rank + 1;
r_even = even_rank ? rank + 1 : rank - 1;
```

另外要計算相鄰兩邊的 rank 會傳多少長度的 data 過來。其中的 `l_n` 是 $n/size + 1$，而 `s_n` 則是 $n/size$。

```
int n_odd, n_even;
n_even = r_even < remainder ? l_n : s_n;
n_odd = r_odd < remainder ? l_n : s_n;
```

接著處理頭尾 rank 在某些 Phase 會沒有跟任何 rank 交換 data 的情況，將這種情況設定為 $-1$，在接下來的迴圈裡排除。

```
if (r_odd >= size) r_odd = -1;
if (r_even >= size) r_even = -1;
```

接下來進入迴圈的部分，每一輪要先判斷這輪是 `Odd Phase` 還是 `Even Phase`，並根據判斷結果選擇這輪要溝通的對象 rank；若是頭尾 rank 會有某一輪需要休息（沒有溝通對象），這時候就略過這一輪。決定好這輪的溝通對象後，使用 `MPI_Sendrecv` 進行資料的傳輸；傳輸完後根據自己和溝通對象的相對位置，決定應該要取前半部分還是後半部分的資料。

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

    int even_phase = (i % 2 == 0);
    int adj_n = even_phase ? n_even : n_odd;
    int adj_r = even_phase ? r_even : r_odd;
    if (adj_r == -1) continue;


    MPI_Sendrecv(
        local_data, local_n, MPI_FLOAT, adj_r, 0,
        local_recv, adj_n, MPI_FLOAT, adj_r, 0,
        MPI_COMM_WORLD, &status
    );


    if (rank < adj_r) choose_min(local_data, local_recv, local_n, adj_n, tmp);
    else choose_max(local_data, local_recv, local_n, adj_n, tmp);

}
```

若自己的 rank 小於溝通對象的 rank ，取前半部分資料（`choose_min`）；反之則取後半部分資料（`choose_max`）。`choose_min` 函式從兩方資料的前端（較小的部分）開始 merge，而 `choose_max` 則從資料後端開始進行融合。

```
void choose_min(float* local, float* adj, int local_n, int adj_n, float* tmp)
{
```

```
    int i = 0, j = 0, cnt = 0;
    while (cnt < local_n)
    {
        if (j >= adj_n || (i < local_n && local[i] < adj[j])) tmp[cnt++] = local
        else tmp[cnt++] = adj[j++];
    }

    memcpy(local, tmp, local_n * sizeof(float));

}


void choose_max(float* local, float* adj, int local_n, int adj_n, float* tmp)
{

    int i = local_n - 1, j = adj_n - 1 , cnt = local_n - 1;
    while (cnt >= 0)
    {
        if (j < 0 || (i >= 0 && local[i] > adj[j])) tmp[cnt--] = local[i--];
        else tmp[cnt--] = adj[j--];
    }

    memcpy(local, tmp, local_n * sizeof(float));

}
```

## 2. optimized - cpu time

### Local Data Sorting

上網查發現 `C++` 有專門在做浮點數排序的函式庫，實際使用下來比 `std::sort` 快了非常多。

```
#include <boost/sort/spreadsort/float_sort.hpp>
boost::sort::spreadsort::float_sort(local_data, local_data + local_n);
```

### Combine Odd & Even Phase

如果將 `Odd Phase` 以及 `Even Phase` 寫入同一個迴圈的 iteration ，可以省下每一輪開頭的判斷環節，將迴圈改為每輪 `i += 2` 。另外，將 `choose_min` 以及 `choose_max` 裡面的 `memcpy` 拿掉，直接在這裡使用 `std::swap` 更新資料，速度會快許多。

```
for (int i = 0; i < size + 1; i += 2)
{
    // even phase
    if (r_even != -1)
    {
        MPI_Sendrecv(
            local_data, local_n, MPI_FLOAT, r_even, 0,
            local_recv, n_even, MPI_FLOAT, r_even, 0,
            MPI_COMM_WORLD, &status
        );
        // use choose_min or choose_max to merge data ...
        std::swap(local_data, tmp);
    }
```

```
    // odd phase
    if (r_odd != -1)
    {
        MPI_Sendrecv(
            local_data, local_n, MPI_FLOAT, r_odd, 0,
            local_recv, n_odd, MPI_FLOAT, r_odd, 0,
            MPI_COMM_WORLD, &status
        );
        // use choose_min or choose_max to merge data ...
        std::swap(local_data, tmp);
    }
}
```

#### Bit Operation

在進行計算時用 `Bit Operation` 取代一般運算符號，例如：

```
int even_rank = (i % 2 == 0);
```

改為：

```
int even_rank = ((i & 1) == 0);
```

## 3. optimized - communication time

#### Odd-Even Sort

在迴圈中，某些情況其實是不需要進行資料交換的，例如 rank $0$ 與 $1$ 在真正傳遞整個資料陣列之前，可以檢查 $(0$ 的最後一個資料數值 $)$ 是否 $\leq (1$ 的第一個資料數值$)$，若 `true` 則代表這一輪根本不需要排序，進而也無需傳遞資料。

因此在每一輪開頭使用 `MPI_Sendrecv` 傳遞一個浮點數資料，根據判斷結果進行下一步真正的資料陣列傳遞。

```
// inside for-loop
float send_val, recv_val;
if (r_even != -1)
{
    send_val = (rank < r_even) ? local_data[local_n - 1] : local_data[0];
    MPI_Sendrecv(
        &send_val, 1, MPI_FLOAT, r_even, 0,
        &recv_val, 1, MPI_FLOAT, r_even, 0,
        MPI_COMM_WORLD, &status
    );
    need_exchange = rank < r_even ? (send_val > recv_val) : (send_val < recv_val)
    if (need_exchange)
    {
        // send local data & receive neighbor data ...
        // use choose_min or choose_max to merge data ...
        // swap local data & merged data
    }
}
if (r_odd != -1)
```

```
{
    // do the same as even phase ...
}
```

# Experiment & Analysis

## 1. Methodology

**System Spec:** 實驗使用課程提供的 `apollo cluster`。
**Performance Metrics:**

- Testcase: 使用助教提供測資中的 `37.txt` 作為實驗使用測資，資料長度為 $536869888$。
- Method: 僅計算從 `MPI_Init` 後至 `MPI_Finalize` 之間的執行時間，使用 `nsys / nvtx` 分割 `IO time / CPU time / Communication time`，程式中使用到 `MPI_File...` 等函式處分類為 `IO time`，使用到 `MPI_Sendrecv` 處分類為 `Communication time`，剩餘時間為計算時間 `CPU time`。

## 2. Plots: Original Implementation

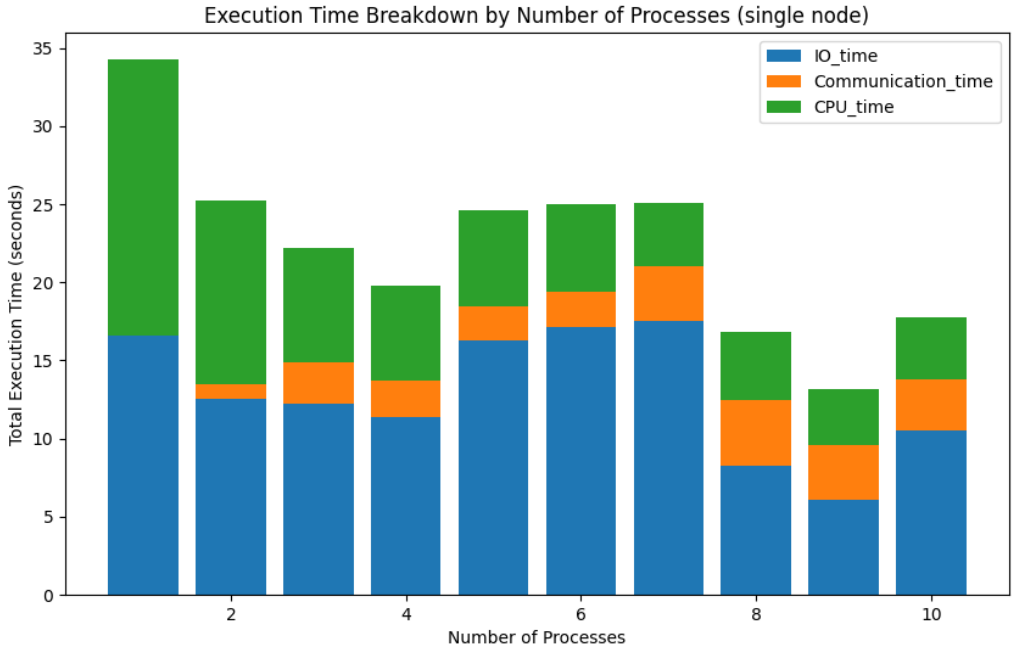
**Description**
用優化前的程式進行實驗，分別進行 `Single-node` 以及 `Multi-node` 兩組實驗。 `Single-node` 固定使用 `-N1`，並分別測試 process 數量由 $1$~$10$ 的執行時間結果，執行時間又細分為 `IO / Communication / CPU` 三種類型； `Multi-node` 固定 process 數量為 $12$，並測試 node 數量由 $1$~$4$ 的執行時間結果，一樣分類為 `IO / Communication / CPU` 三種時間區段。

**Single-node Table**

| #Processes | IO_time (s) | Communication_time (s) | CPU_time (s) | Total_time (s) |
|---|---|---|---|---|
| 1 | 16.623000 | 0.000000 | 17.652030 | 34.275030 |
| 2 | 12.516000 | 0.978069 | 11.768971 | 25.263040 |
| 3 | 12.241000 | 2.651588 | 7.301436 | 22.194024 |
| 4 | 11.394000 | 2.286457 | 6.123564 | 19.804021 |
| 5 | 16.296000 | 2.151354 | 6.148668 | 24.596022 |
| 6 | 17.124000 | 2.299843 | 5.585179 | 25.009022 |
| 7 | 17.489000 | 3.571891 | 4.016132 | 25.077023 |
| 8 | 8.257000 | 4.200365 | 4.380664 | 16.838029 |
| 9 | 6.080593 | 3.506382 | 3.546645 | 13.133620 |

| | | | | |
|---|---|---|---|---|
| 10 | 10.532040 | 3.274775 | 3.978314 | 17.785129 |

### Single-node Time Profile




### Single-node Discussion

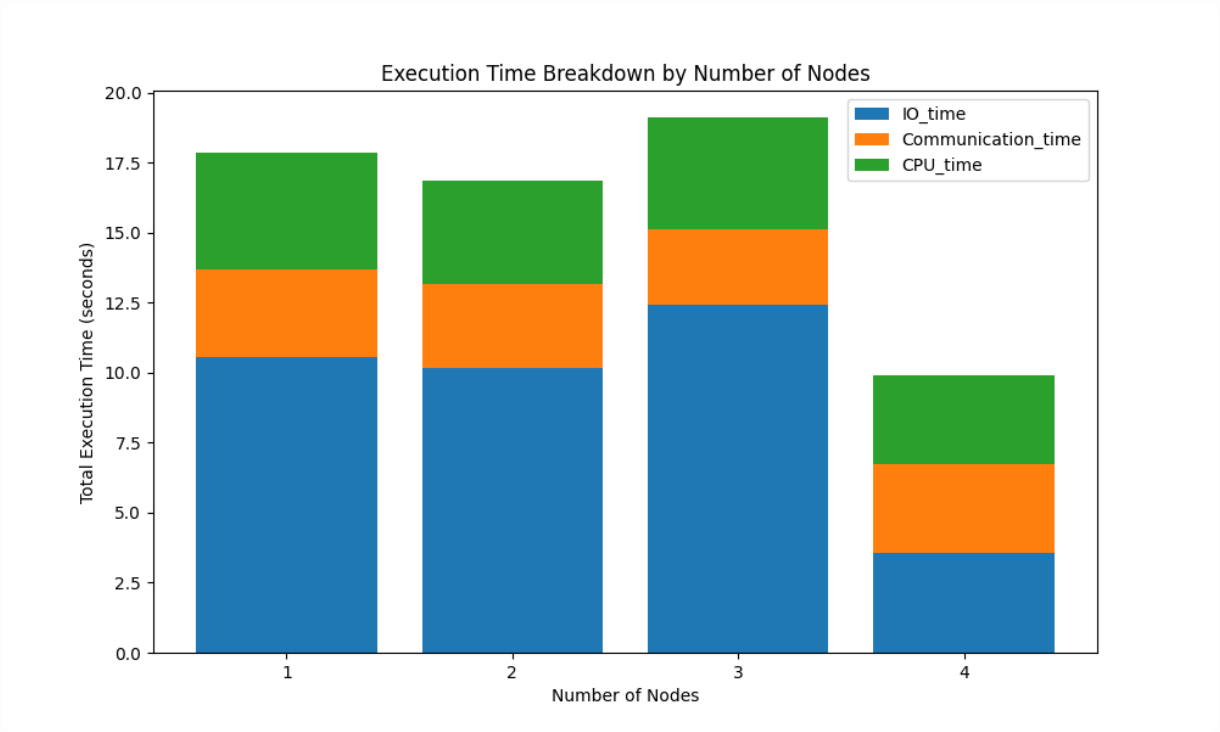
- `IO_time`
  在較少 process 數量時佔據執行時間的主導地位，隨著 process 數量增加而逐漸減少，推測是因為每個 process 所需讀取的資料量隨著平行力度增強而減少。整體來說所佔時長比例最大，為總執行時間的主要瓶頸。
- `Communication_time`
  在 process 數量少時所佔時間不大，隨著平行化增強（ process 數量增加 ）而漸漸加大佔比，顯示當 process 數量變大，process 之間的溝通也會帶來一定的 overhead。預估當 process 數量成長到一定的程度可能會抵消平行化帶來的效益。
- `CPU_time`
  在 process 數量逐漸減少的前段，時間佔比急速下降；到後期趨於平緩，顯示當 process 數量達到一定規模時，溝通所形成的 overhead 可能會抵銷計算時間平行化所帶來的效益。

### Multi-node Table

| #Nodes | IO_time (s) | Communication_time (s) | CPU_time (s) | Total_time (s) |
|---|---|---|---|---|
| 1 | 10.567910 | 3.133879 | 4.156156 | 17.857945 |
| 2 | 10.174416 | 2.982089 | 3.702938 | 16.859443 |
| 3 | 12.435681 | 2.676381 | 4.004648 | 19.116710 |

| | | | | |
|---|---|---|---|---|
| 4 | 3.553903 | 3.194901 | 3.156125 | 9.904929 |

### Multi-node Time Profile




### Multi-node Discussion

由結果來看，雖然 `IO_time` 佔比最大，依舊是執行時間的主要瓶頸，不過隨著 node 數量增加，`IO_time` 的佔比有顯著的下降。推測是由於隨著 node 數量增加，每個 node 要執行的 process 數量減少，因此 IO workload 也隨之下降。相較於 `IO_time` 的明顯差異，`Communication_time` 以及 `CPU_time` 的變化則相對小了許多。`Communication_time` 有些微上升趨勢，推測是由於 node 數量增加，溝通所帶來的 overhead 也隨之上升。

## 3. Plots: CPU Time Optimized

### Description

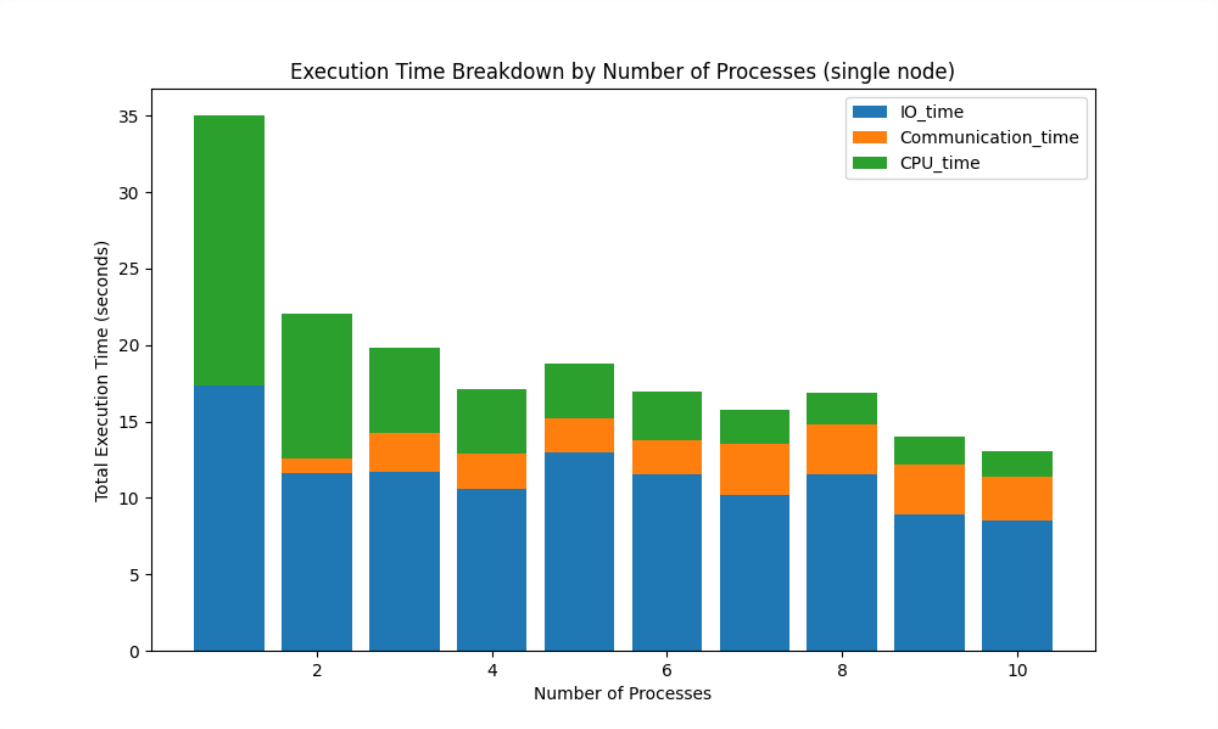
用優化 `CPU_time` 後的程式進行實驗，一樣進行 `Single-node` 以及 `Multi-node` 兩組實驗。`Single-node` 固定使用 `-N1`，並分別測試 process 數量由 $1$~$10$ 的執行時間結果；`Multi-node` 固定 process 數量為 $12$，並測試 node 數量由 $1$~$4$ 的執行時間結果。兩組實驗結果皆分類為 `IO / Communication / CPU` 三種時間區段。

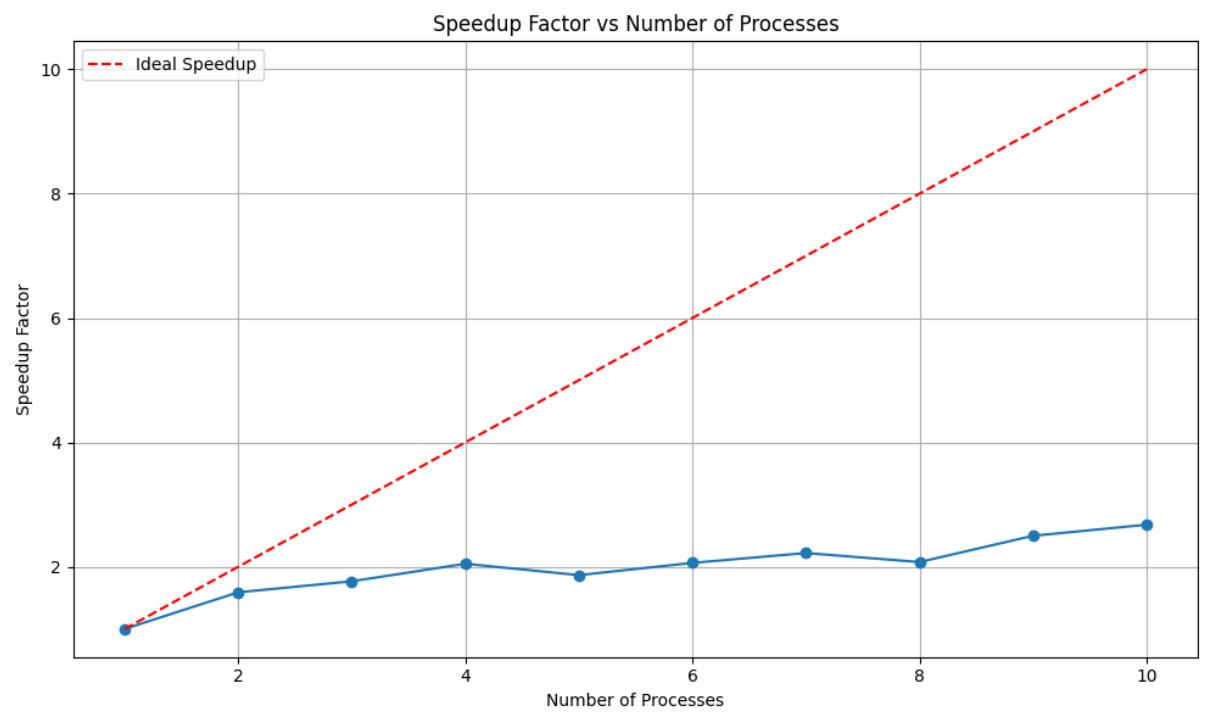
### Single-node Table

| #Processes | IO_time (s) | Communication_time (s) | CPU_time (s) | Total_time (s) |
|---|---|---|---|---|
| 1 | 17.355000 | 0.000000 | 17.664037 | 35.019037 |
| 2 | 11.625000 | 0.952307 | 9.450041 | 22.027348 |

| | | | | |
|---|---|---|---|---|
| 3 | 11.699000 | 2.575184 | 5.543857 | 19.818041 |
| 4 | 10.615000 | 2.316812 | 4.159211 | 17.091023 |
| 5 | 12.959000 | 2.246920 | 3.559116 | 18.765036 |
| 6 | 11.558000 | 2.214870 | 3.203153 | 16.976023 |
| 7 | 10.183000 | 3.322116 | 2.256907 | 15.762023 |
| 8 | 11.558000 | 3.245465 | 2.062561 | 16.866026 |
| 9 | 8.952482 | 3.198053 | 1.862968 | 14.013503 |
| 10 | 8.551583 | 2.838069 | 1.692161 | 13.081813 |

### Single-node Time Profile & Speedup

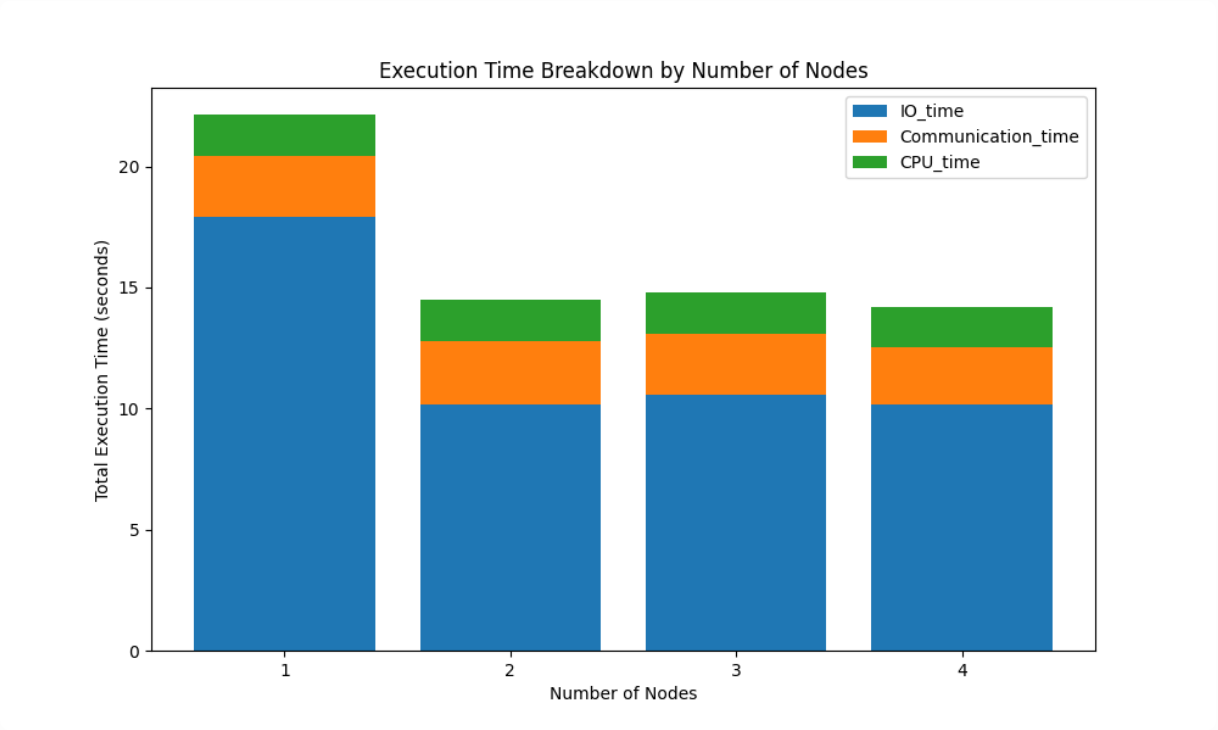


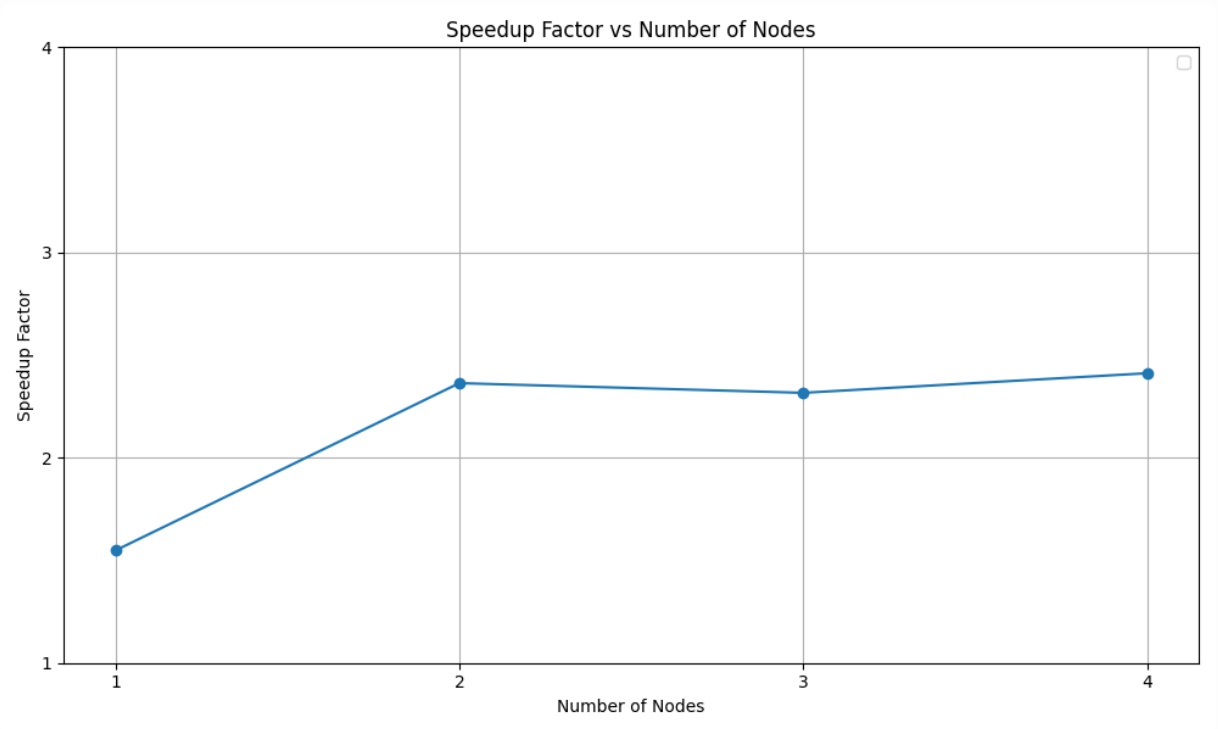



### Multi-node Table

| #Nodes | IO_time (s) | Communication_time (s) | CPU_time (s) | Total_time (s) |
|---|---|---|---|---|
| 1 | 17.892364 | 2.557506 | 1.689568 | 22.139438 |
| 2 | 10.152761 | 2.637220 | 1.712400 | 14.502381 |
| 3 | 10.583375 | 2.507442 | 1.706203 | 14.797020 |
| 4 | 10.142054 | 2.377754 | 1.692715 | 14.212523 |

### Multi-node Time Profile & Speedup




### Optimized Program Discussion

優化後的執行總時長明顯下降許多，仍然可以看到 `IO_time` 所佔時間比例最大，且隨著 process / node 數量增加，`IO_time` 有小幅度的降低；`Communication_time` 的部分則是些微呈現提升的趨勢，同樣也是 process 之間溝通所帶來的 overhead 所致；相較於優化前的結果，最明顯的差異在於 `CPU_time` 的時間佔比大幅度的下降，主要做優化的部分以計算時間為主，因此也是符合想像。

從 Speedup Factor 可以看出，單純優化 `CPU_time` 的部分能夠帶來的效益並不完全符合預期，由於主要的瓶頸在於 `IO_time` 以及 `Communication_time` ，因此優化後 Speedup 最多大概只能到達 Sequential Code 的兩倍左右（process 數量為 $10$ 時），若想要得到更好的 Speedup 結果，可能必須想辦法從優化 `IO_time` 以及 `Communication_time` 入手。

## 4. Plots: Communication Time Optimized

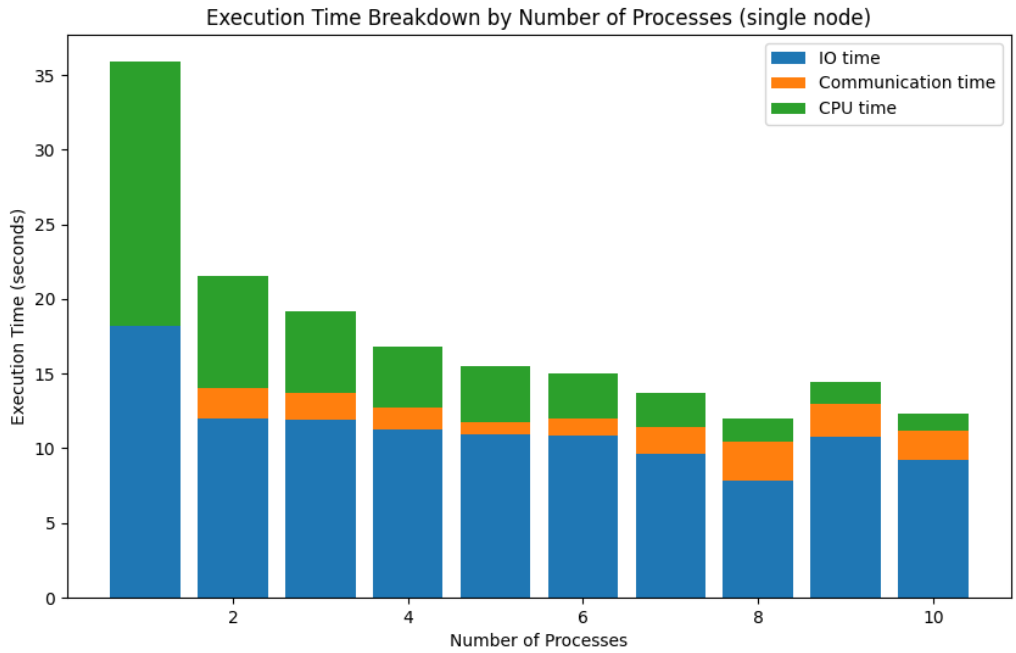
### Description

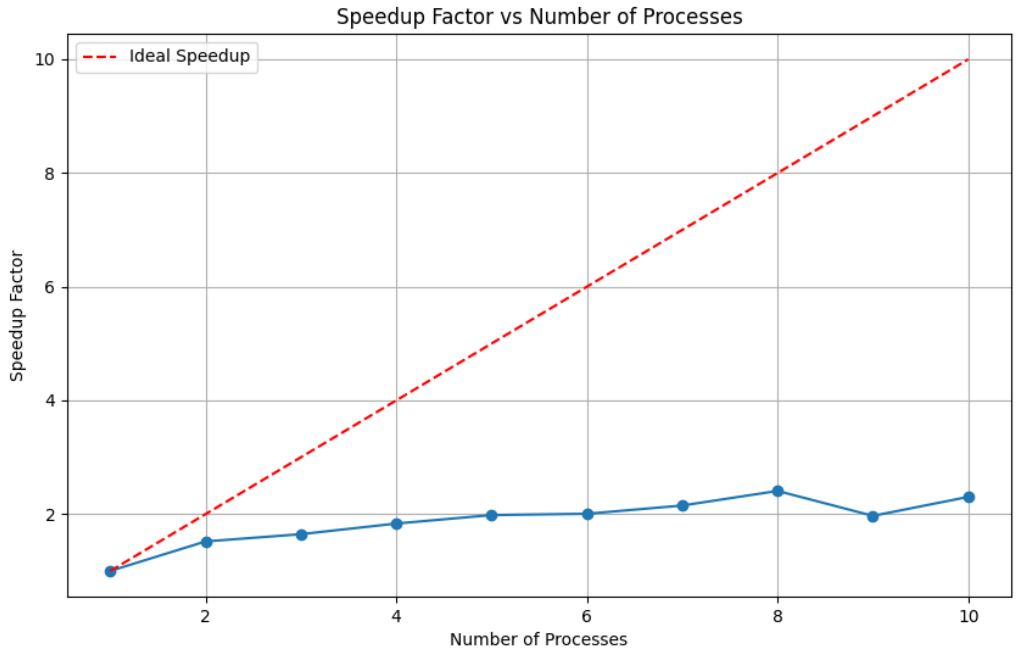
用優化 `Communication_time` 後的程式進行實驗，實驗設定與之前相同。

### Single-node Table

| #Processes | IO_time (s) | Communication_time (s) | CPU_time (s) | Total_time (s) |
|---|---|---|---|---|
| 1 | 18.176000 | 0.000000 | 17.731000 | 35.907 |
| 2 | 12.008000 | 2.059012 | 7.447988 | 21.515 |
| 3 | 11.917000 | 1.769489 | 5.453511 | 19.140 |
| 4 | 11.268000 | 1.441299 | 4.061701 | 16.771 |
| 5 | 10.914000 | 0.844959 | 3.746041 | 15.505 |
| 6 | 10.870000 | 1.105392 | 2.999608 | 14.975 |
| 7 | 9.622000 | 1.788078 | 2.317922 | 13.728 |
| 8 | 7.867000 | 2.561128 | 1.599872 | 12.028 |
| 9 | 10.765000 | 2.249911 | 1.425089 | 14.440 |
| 10 | 9.189507 | 1.977495 | 1.143998 | 12.311 |

### Single-node Time Profile & Speedup



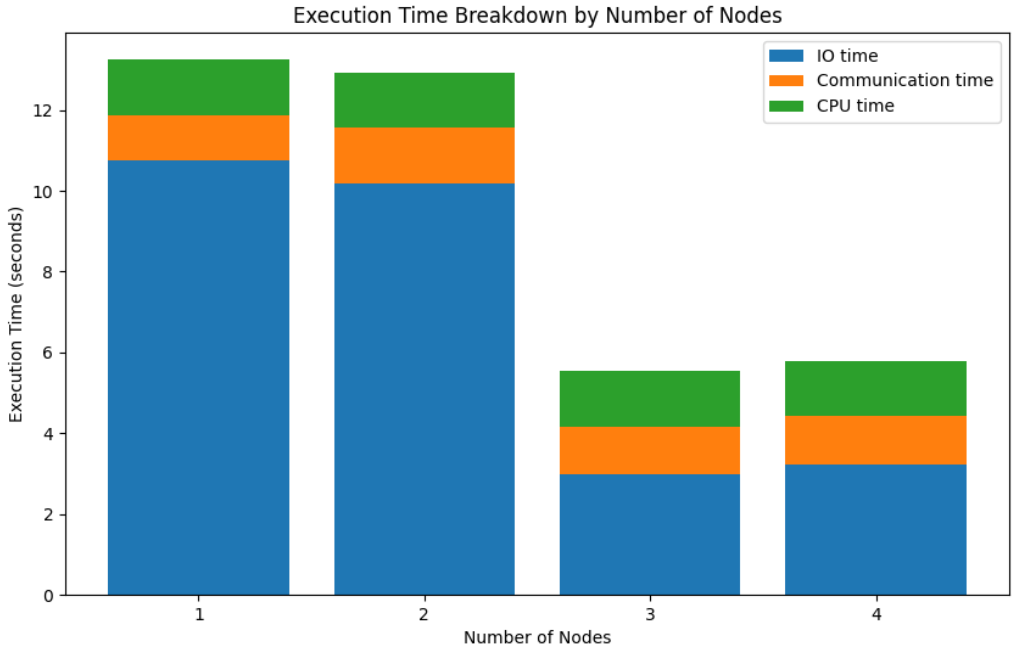



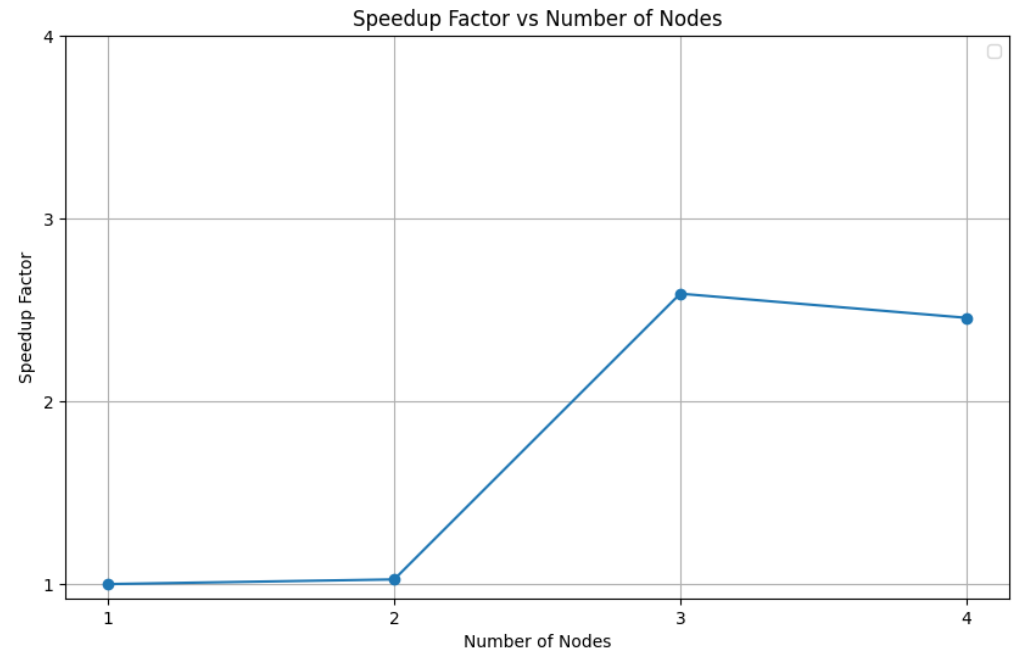

### Multi-node Table

| #Nodes | IO_time (s) | Communication_time (s) | CPU_time (s) | Total_time (s) |
|---|---|---|---|---|
| 1 | 10.759772 | 1.115942 | 1.382286 | 13.258 |
| 2 | 10.190327 | 1.380240 | 1.368433 | 12.939 |
| 3 | 2.985053 | 1.185638 | 1.367309 | 5.538 |

| | | | | |
|---|---|---|---|---|
| 4 | 3.226397 | 1.212457 | 1.353146 | 5.792 |

### Multi-node Time Profile & Speedup




### Optimized Program Discussion

經過優化後，整體執行時間顯著縮短，藉由分析各個時間組成部分可以注意到，`Communication_time` 相較於之前的結果時間大幅減少，顯示在這方面的優化帶來頗大的效益；不過在 process 數量較多的情況會有小幅度的上升的現象，這很有可能是優化方式造成的些微影響。整體來說，提前檢查資料大小確實能有效減少 process 之間在不必要的溝通上所耗費的時間。

## Conclusion & Possible Improvements

這是我第一次嘗試撰寫平行程式，接觸到很多 MPI 相關函式以及其使用方法。我認為作業遇到的第一個瓶頸是如何選擇符合自己需求的 API 來使用，例如像這次作業如果使用最陽春的 `MPI_Send` 和 `MPI_Recv` 會大大降低程式執行速度，後來上網查才發現有另外一個 API `MPI_Sendrecv` 可以同時進行雙向的資料傳輸。

另一個重點是如何收集有意義的實驗結果，以及 profiler 工具的使用。在使用 profiler 收集數據之前，很難憑空想像程式執行時間的佔比，像我在一開始嘗試優化程式時，以為最大的時間瓶頸會是數值計算（`CPU_time`）的部分，但最後用 profiler 做實驗才發現原來最大的 bottleneck 在於檔案讀寫的時間（`IO_time`）。另外，在做實驗時每次執行結果會有些微差異，這些差異也會影響我對實驗結果的詮釋，這是一個蠻大的困難點。

整體來說，我認為這次作業讓我對程式執行原理以及平行化程式有更深入的了解，不管是 API 還是 profiler 工具的使用都讓我學到許多平常不太可能接觸到的東西。以下是我認為若有多餘時間，我可以對 hw1 實作繼續嘗試的方向：

- 使用 non-blocking send / receive 以減少 communication overhead
- 使用 `MPI` 所提供的 Collective IO 函式，例如 `MPI_Scatter` 等，嘗試一次讀取所有資料再統一分配給所有 process ，而非每個 process 各自讀取，這樣也許能有效減少 `IO_time`。