Parallel Programming HW1 Odd-Even Sort

112065528 洪巧雲

Code Implementation

- Key: Memory optimization
- 如何減少傳輸的資料量、速度會是整體程式的優化關鍵。

Methodology & Optimization

- 問題: 做 odd-even sort 的演算法
  - Odd-even sort 為 comparison-based sorting，主要為以下兩個階段:
    - Even-phase: 先對 index 為偶數的 value 去做 compare-and-swap 的 sorting
    - Odd-phase: 再對 index 為奇數的 value 去做 compare-and-swap 的 sorting

> 在每個階段，演算法的功能是做 compare-and-swap 這個操作，直到 input 的 array is sorted。

- 使用 MPI API 拿到 rank 與 size 後，我的分法是先將 `data數量 / size(Process數)` 此時會得到變數 `m` ，接下來會去判對目前執行的 rank ID 是否為最後一行 `m += (rank == size - 1)? (temp_idx % size) : 0` ，若是最後一行則需要接收 `data數量 / size(Process 數)` 的餘數，之所以是這樣的分法原因在於這樣比較好處理與相鄰 Process 交換資料的部分。

- 會需要先判斷這個 rank 是否存在左鄰居 or 右鄰居，因為 `rank 0` 沒有左鄰居、 `rank == size - 1` 沒有右鄰居，事先判斷可以省去進入 while loop 做資料交換時發生 runtime error。

- 再來是需要去考量若為倒數第二個 rank 其右鄰居 m 的 size 需要加上 餘數:

```
// 判斷條件: 若為倒數第二個 rank，其右進程 m 需要加上 (temp_idx % size)
int rProc = (temp_idx % size != 0 && rank == size - 2)? m + (temp_idx % size) : m;
rProc = (rank == size - 1)? 0 : rProc;
```

- 最後一個需要考量的重點是 testcase 中存在特別 edge case，這些 case 的 data size 會大於 process 數量，如果不設條件判斷的話會有很多 testcase 過不了，這也是我在實作程式碼時解最久的 bug，最後是用以下方式解決:

```
if( temp_idx < size){
    if(rank < temp_idx){
        m = temp_idx / temp_idx;
        start_idx = m * rank;
        rProc = (rank == temp_idx-1)? 0 : m;
        lProc = (rank == 0) ? 0 : m;
    }else{
        m = 0;
        start_idx = 0;
        rProc = 0;
        lProc = 0;
    }
}
```

也就是當 data size 大於 process 數量時，此時會分兩個條件判斷，條件一是當此時的 rank ID < data size時，m 會等於 1，起始的 index 就會等於 rank ID，若 rank ID >= data size時，m 會等於 0。

- 處理完各個不同狀況下的 data size 後，就可以進入 while 迴圈開始做 merge、swap等計算，我將 while loop 的終止條件設為 跑 `size + 1` 次結束，這樣可以省去很多時間。
- 在一開始，我會先去判斷這個 rank 是否為可以安全的進行 Odd Phase or Even Phase，因為若此 rank 不存在左 process，則不可以進入 Even Phase，反之亦然，因為會出現 runtime error。我的 Odd Phase 是把當下的 process 與右 process 進行資料的交換排序，Even Phase 則會讓當下的 process 與左 process 進行資料的交換排序。

Sort before entering the loop

- 在我的程式碼中有實作一個特別的步驟讓整體時間省下很多，就是在進行大量資料排序與交換前，我會先讓程式去判斷需不需要做排序，在進入 while loop 之前，我利用 `boost::sort::spreadsort::float_sort(a, a + m)` 將這個 process 的資料先做排序，然後才進入 while loop 迴圈，所以只需要在 Odd Phase 時去判斷這個 process 的最後一個元素是否小於右 process 的第一個元素，是的話代表不需要再進行資料的排序與交換，不是的話才會進入排序與交換；在 Even Phase 時去判斷這個 process 的第一個元素是否大於左 process 的最後一個元素，這樣就能省去很多多餘的資料交換與排序。

- `mergeSort_increasing` & `mergeSort_decreasing`：在一開始寫程式的時候只寫了一般的 sorting method，也就是 `mergeSort_increasing` 這個 void function，但在出現 wrong answer 後才發現這次的作業跟一般的 Odd-even sorting 不一樣，奇偶排序法在平行化下的概念應該是針對奇數 rank 與偶數 rank 做 sorting 排序，其中牽涉到利用 `MPI_Sendrecv` 做資料的交換與傳輸，在接收左邊or右邊的 rank data 後，才會 call `mergeSort_increasing\mergeSort_decreasing` 進入排序的 void function，在做完這一輪 Odd or Even 的排序時，會有切換 Phase 的條件式，`oddPhase = (oddPhase == 1)? 0 : 1`，代表在同一個 process 中，都會用到這兩個 mergeSort 才能完成排序，在切換 phase 後，假設前一次 sorting 是 Odd Phase 的 `mergeSort_increasing`，那切換後就會進入 Even Phase，資料的 sorting 就會變成 `mergeSort_decreasing`，會由最後面的 index 往前做排序，做完後才代表這個 process 完成 sorting。
    - 我認為這一部分是我實作這份作業時遇到的最大瓶頸，因為一開始沒搞懂 Odd-even sort 的真正意義，但在弄懂後後面的部分就完成得差不多了。

- 除此之外還有發現一些小技巧能讓程式快幾秒:

1. `if()` 多層條件式判斷變成多個 `if()` -> 會變快
2. if(反義bool)會比 if(true) 來得慢
3. 改 compiler -> precise
4. Different Module & Compiler
    - icc + Open MPI

Experiment & Analysis

NVIDIA Night Sight

Use `nvtx` to measure the execution time

- 在使用 Nsight 時會需要使用 tag 去 add `nvtx` 指令，並且使用完後需要 `nvtxRangePop()`，使用這個指令後會讓程式抓到這段 `nvtxRangePush()` ~ `nvtxRangePop()` 時間的區間。
- 我總共在程式中抓了 4 個不同類型的時間:

1. IOTime(file): 主要用來記錄程式在跑的時候，`MPI_read_at` 、 `MPI_File_open` 、 `MPI_File_close` 、 `MPI_File_Write_at`

```
MPI_File in_file;
nvtxRangePush("IOTime(file)");
MPI_File_open(MPI_COMM_WORLD, argv[2], MPI_MODE_RDONLY, MPI_INFO_NULL, &in_file);
MPI_File_read_at(in_file, sizeof(float) * start_idx, a, m, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&in_file);
nvtxRangePop(); // IOTime(file)
// 進行排序
```

```
// 寫入排序後的結果到輸出文件
MPI_File out_file;
nvtxRangePush("IOTime(file)");
MPI_File_open(MPI_COMM_WORLD, argv[3], MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &out_file);
MPI_File_write_at(out_file, sizeof(float) * start_idx, a, m, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&out_file);
nvtxRangePop(); // IOTime(file)
```

2. ComputationTime(MergeSort): 主要用來測量程式當中的計算時間，在我的程式中，是使用類似於 MergeSort 的 Merge 方法去做 不同 rank 之間的數字排序以及 Swap。

```
nvtxRangePush("ComputationTime(Merge & Swap)");
mergeSort_increasing(a, rProcArray, temp, m, rProc);
std::swap(temp, a);
nvtxRangePop(); // ComputationTime(Merge & Swap)
```

3. CommunicationTime(SendRecvTime): Communication Time 的時間主要來自 `MPI_Sendrecv()` ，這個 MPI function 主要是用來傳遞不同 rank 的資料，以下截圖中的每個參數代表的意思是跟 `rank - 1` 交換 data (左鄰)。若是 `rank + 1` 的話是右鄰交換資料。交換的資料會存在 `lProcArray` 中。

```
nvtxRangePush("CommunicationTime(SendRecv)");
MPI_Sendrecv(
    a, 1, MPI_FLOAT, rank - 1, 0,
    lProcArray + lProc - 1, 1, MPI_FLOAT, rank - 1, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
nvtxRangePop(); // CommunicationTime(SendRecv)
```

4. TotalTime

```
int main(int argc, char **argv)
{
    nvtxRangePush("TotalTime");
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Group WORLD_GROUP, USED_GROUP;
    MPI_Comm USED_COMM = MPI_COMM_WORLD;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int temp_idx = atoi(argv[1]);
```

```
    // 釋放資源並結束 MPI
    MPI_Finalize();
    nvtxRangePop(); // TotalTime
}
```

Performance Metric

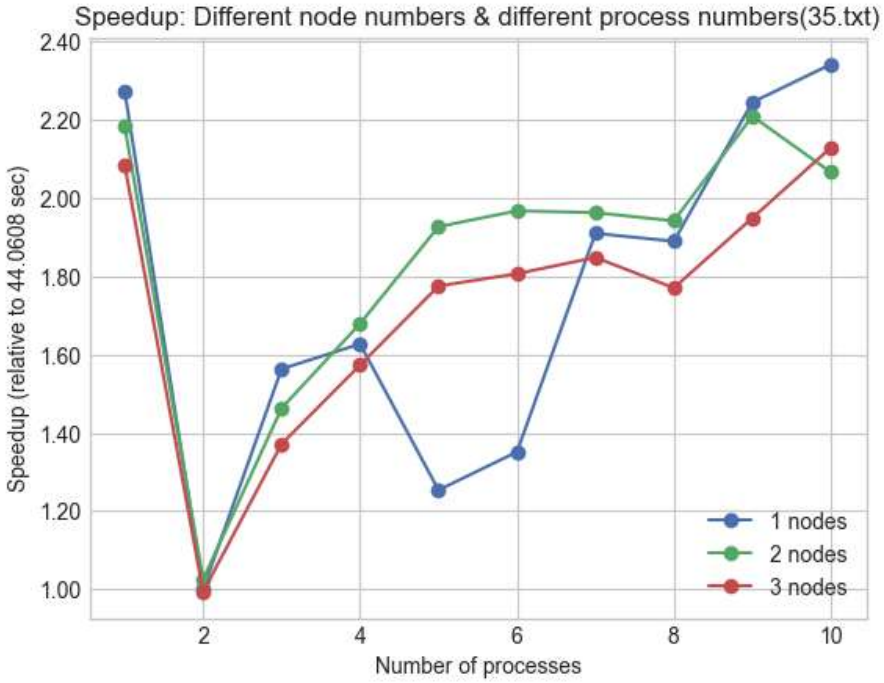
`35.txt`

Speedup

- 這裡的 Test case 是使用 `35.txt`：資料量 `n = 536869888`
- Speedup的部分我是使用 `N1 n1` sequential 的時間當作 base 去計算，公式如下:

$$\text{Speedup} = \frac{T_1}{T_p}$$

其中 `T1 = sequential time` 。



- 由圖表可以發現到 Speedup 大致趨勢有符合 processes 越多效能越好的傾向，雖然有一些 outlier。
- Node = 2, process = 1; Node = 3, process = 1,2,
  - 以上三個點的數據皆有跑出一些 error message，原因是在 process 數量 > Node 數量時，它都會以 1-Node 去跑。

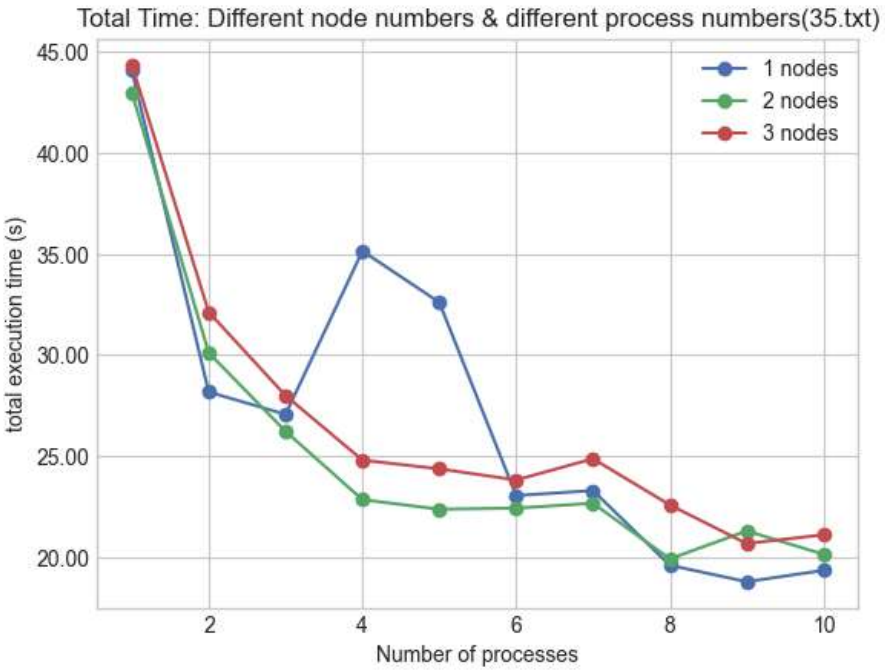
Scalability

- Scalability 的部分，我是使用不同數量的 Node (1-3)、並且開不同數量的 process (1-10)去做比較:

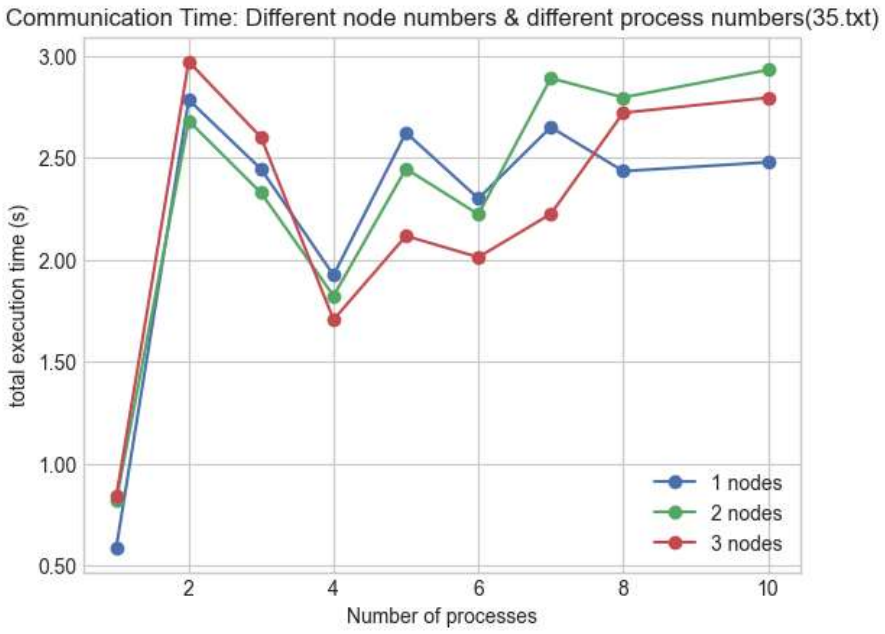
- Different Node numbers & Different Process numbers

- 可以發現圖表明顯展現出 process 越多 total time 越少的趨勢。

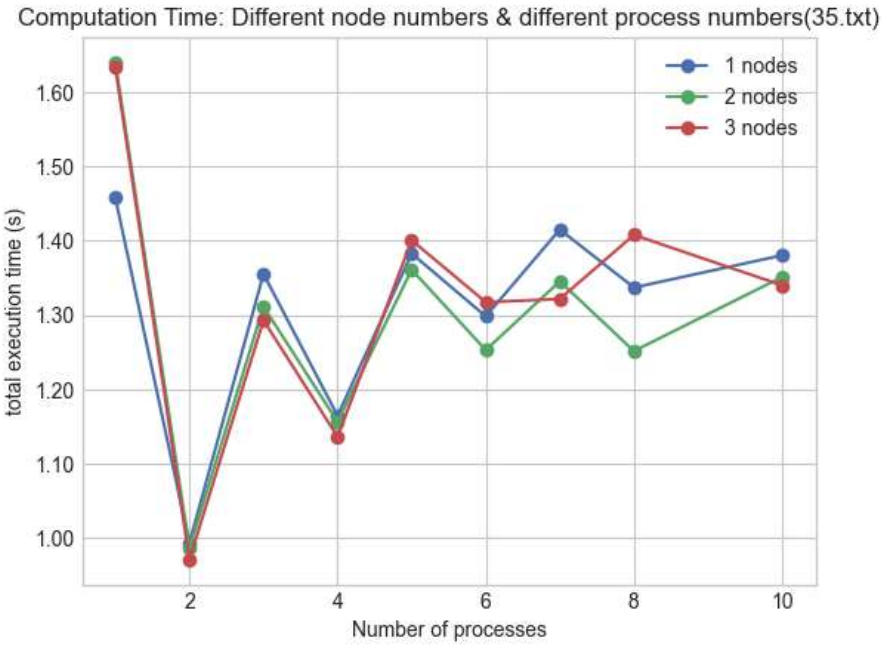
Nodes: 1-3, Processes: 1-10, testcase: 35.txt

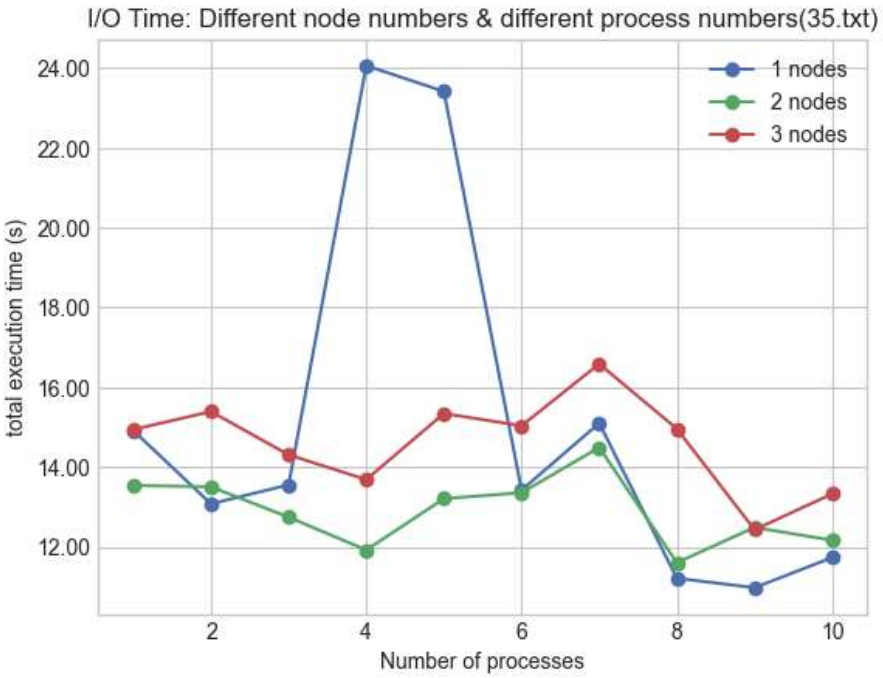


- Communication Time 則是隨著 Process 越多時間也越多，這也符合程式的邏輯 (process 越多會需要有更多 Communication time)。



- Computation Time 與 I/O Time 則是比較不規律的圖表，Outlier 也相對多很多:

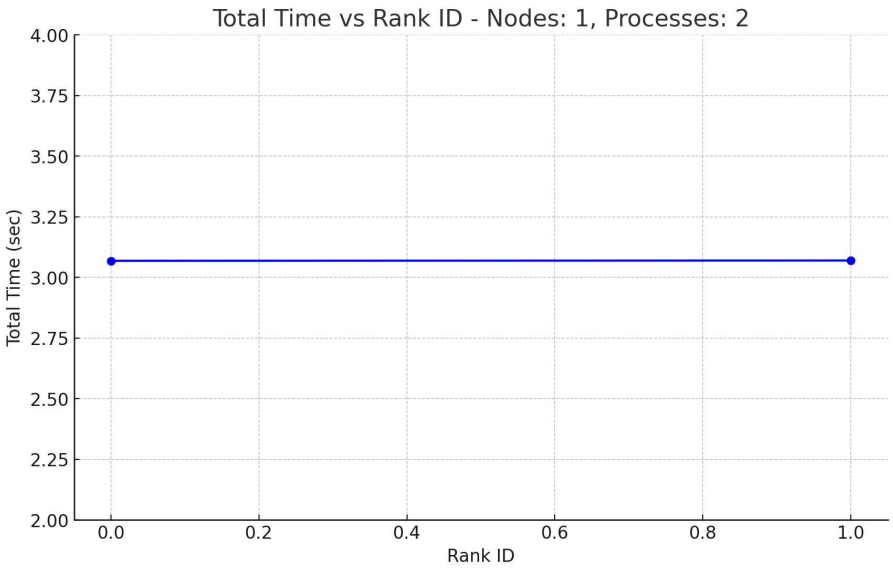


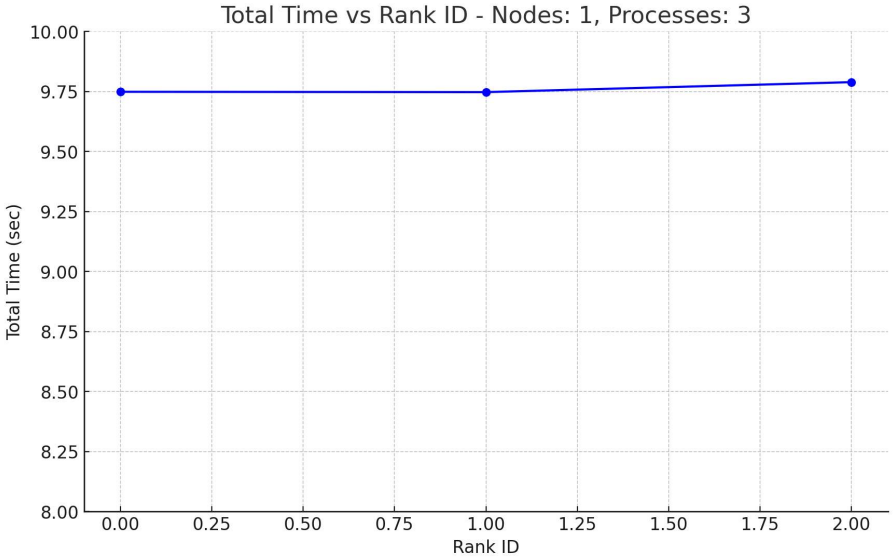


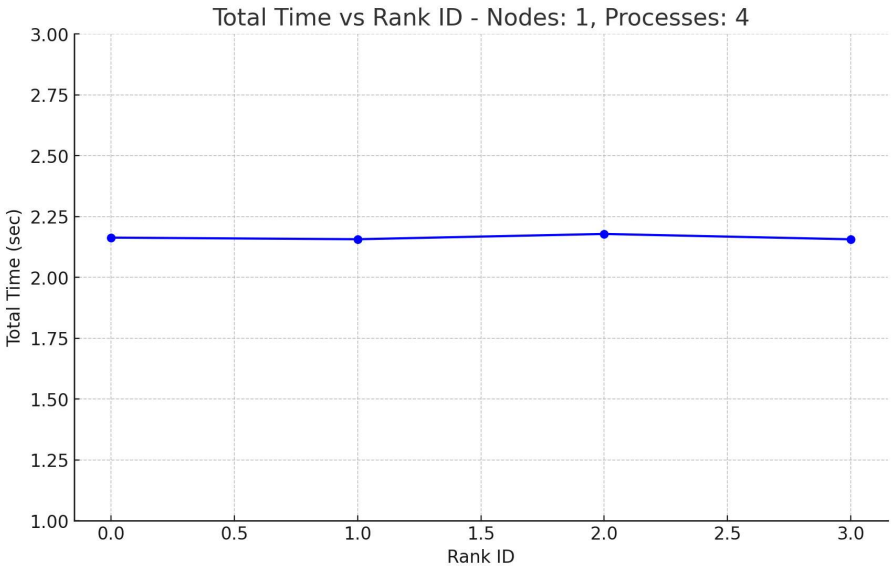
Load Balance

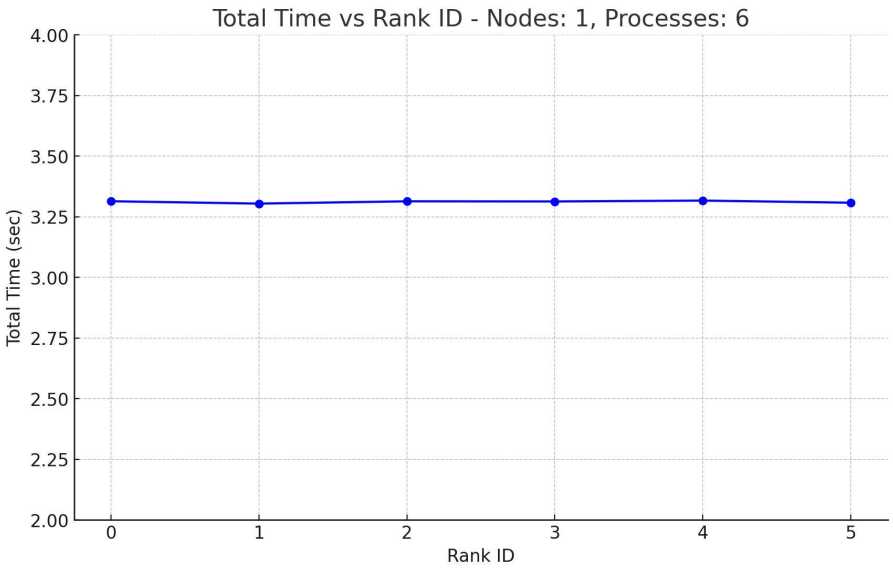
`30.txt`

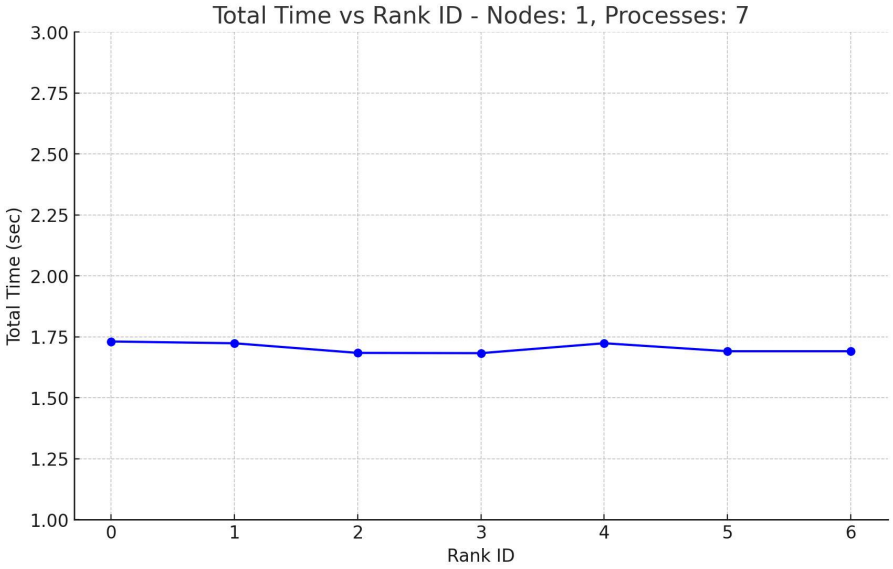
- 這裡的 Test case 是使用 `30.txt`：資料量 `n = 64123483` ，實驗的變數為: Node 固定在 1 個，使用不同數量的 Process 去看它每個 rank ID 的 execution time。
  Nodes: 1, Rank ID: 1~10: 可以發現每個 rank 的執行時間都差不多，load balance 效能不錯。

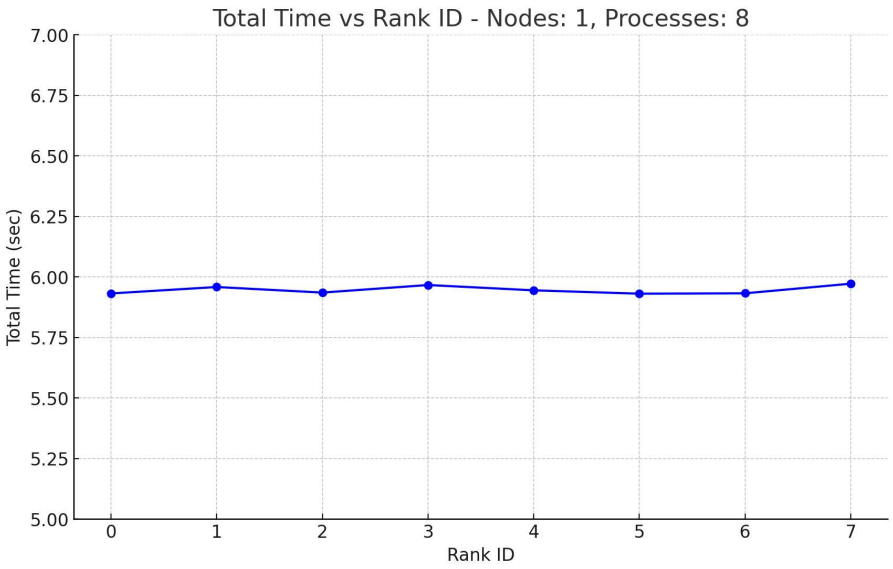

Total Time vs Rank ID - Nodes: 1, Processes: 2
Total Time (sec)
Rank ID
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0.0
0.2
0.4
0.6
0.8
1.0

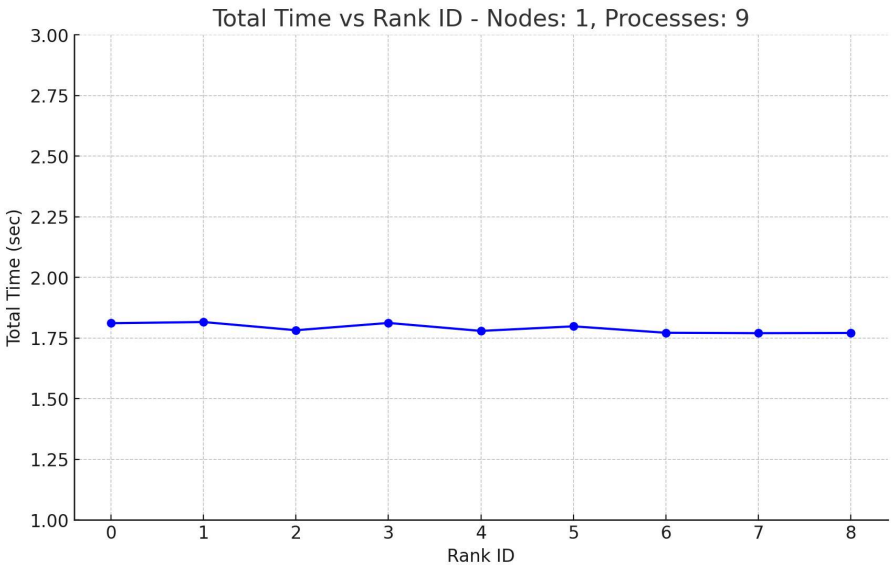

Total Time vs Rank ID - Nodes: 1, Processes: 3
Total Time (sec)
Rank ID
10.00
9.75
9.50
9.25
9.00
8.75
8.50
8.25
8.00
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

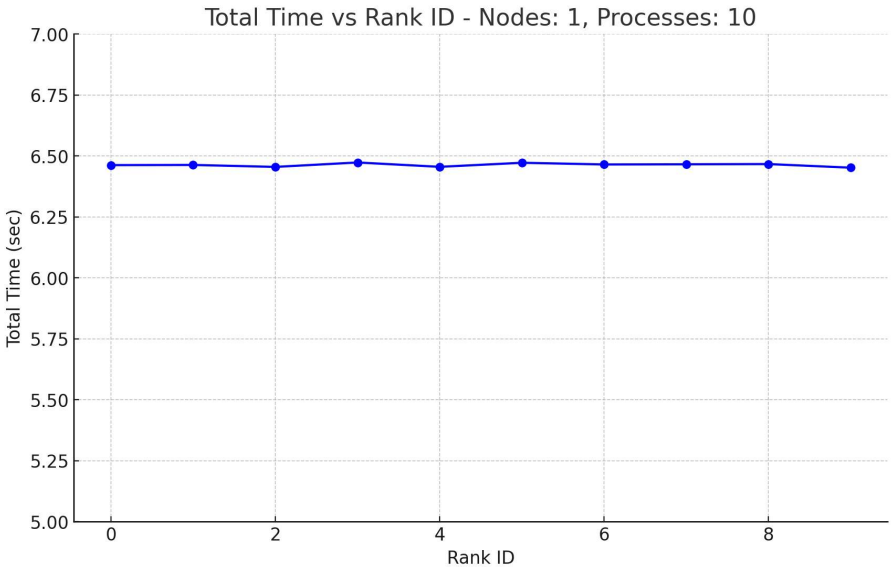

Total Time vs Rank ID - Nodes: 1, Processes: 4
Total Time (sec)
Rank ID
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0


Total Time vs Rank ID - Nodes: 1, Processes: 6
Total Time (sec)
Rank ID
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0
1
2
3
4
5


Total Time vs Rank ID - Nodes: 1, Processes: 7
Total Time (sec)
Rank ID
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0
1
2
3
4
5
6

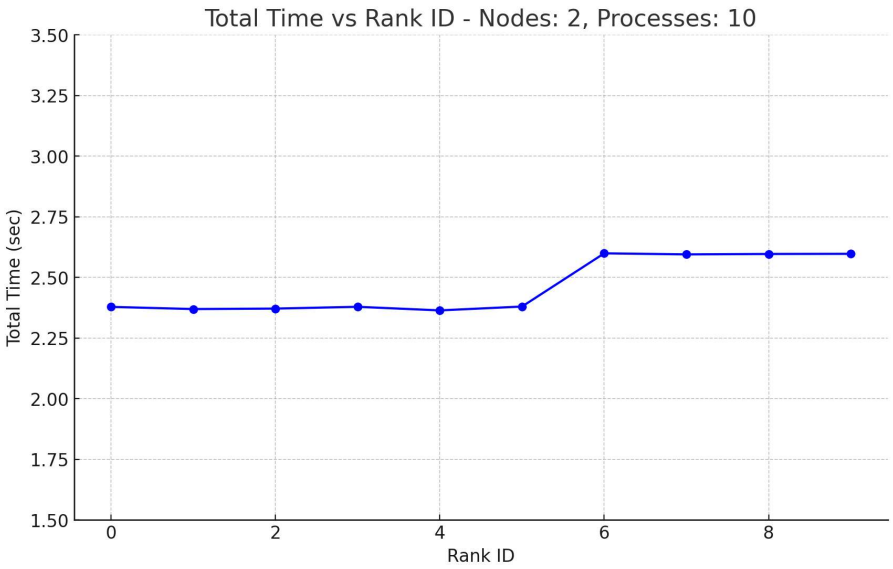

Total Time vs Rank ID - Nodes: 1, Processes: 8
Total Time (sec)
Rank ID
7.00
6.75
6.50
6.25
6.00
5.75
5.50
5.25
5.00
0
1
2
3
4
5
6
7



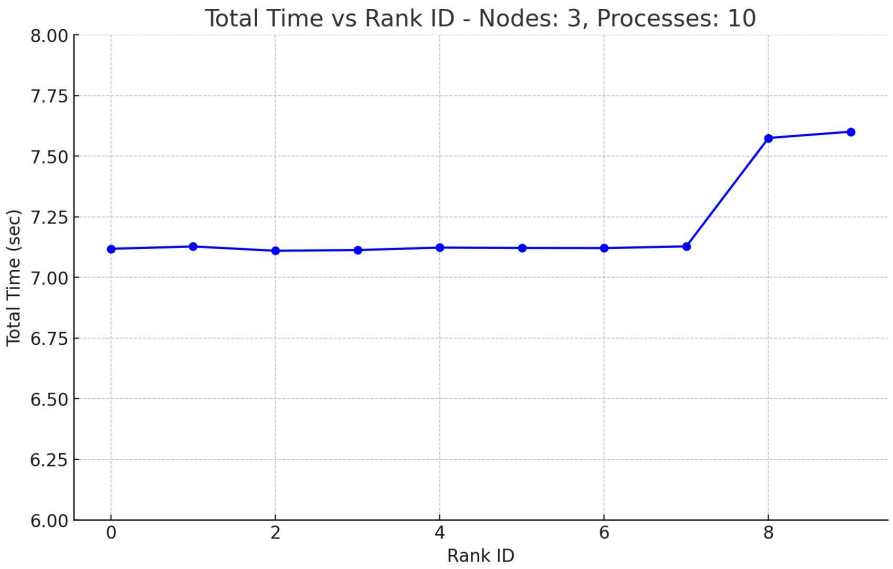


- 以下為 Multi-node 的 load balance 結果:

  Nodes: 2, Rank ID: 1~10

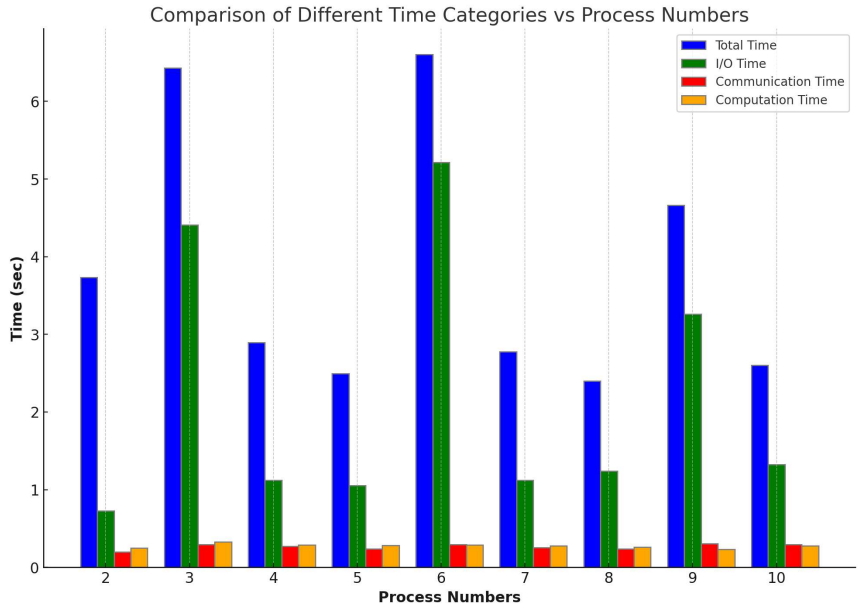


Nodes: 3, Rank ID: 1~10



可以發現 Node 數量越多，在 Process 同樣都是 10 的情況下，並不會讓 load balance 的情況越趨穩定。

- *Different Process numbers / Same Node number: Communication Time、I/O Time、Computation Time*
  下圖為不同的 Process Number 與各自的 Communication Time、I/O Time、Computation Time 綜合比較柱狀圖，由此圖可以發現程式大部分的時間是花在 IO time 上面的，第二多的是 Computation Time，但是在 Process 數量大於 9 後，Communication Time 就變成第二多的時間了。



Discussion

在實驗數據的部分，出現很多 outlier，估計是因為在做這份作業的實驗時離繳交期限剩不到一周，因為 server 上較壅塞，故有些數據的狀況不太理想，但整體趨勢依舊是符合 load balance 與 scalability 的。其實在跑實驗的時候有發現實驗數據有些特別的情況，就是 在 single node single process 的狀況下沒有 Computation Time (也沒有

Communication Time 但沒有這個很正常因為是 single process 不會使用到 `MPI_Sendrecv`)，之所以沒有 Computation Time 是因為它不會進入排序中，它在前面 `boost::sort::spreadsort::float_sort(a, a + m)` 就會 sort 完了，而我的 NVTX tag 放在 call merge sort 的 function 前，故不會測到這個的時間。其實在看了自己實作的程式數據後才可以理解很多之前沒考慮到的事，像是我在做完 performance metric 後才發現我程式的 bottle neck 應該是 Communication time，主要因為 I/O time 與 Computation time 的優化都很有限，而我的 Communication time 隨著 process 數量變多，增加了很多時間，雖然是正常的，但是或許可以讓它幅度不要增長這麼大，也是可以優化的方向，目前的想法是每個 process 要如何分 data size，會是可以優化的部分，能更 load balance 一些，scalability 也會更好。

Conclusion

作業一有許多可以時做自己的 sorting 演算法的空間，相對於作業二，作業依可以優化的空間較多，但也有很多地方需要去考量，像是使用 `MPI_Sendrecv` 會需要先去找出各種可能情況下的 data size，這些很考驗邏輯以及 debug 的能力，我一開始遇到最大的瓶頸就是一直 runtime error，是因為沒有考慮好交換資料的 data size 大小才會出現這樣的錯誤。在理解 Odd-Even sort 後，才知道原來作業一要求的 Odd-Even Sort 與網路上查到的有一些區別，主要因為需要去平行化程式，故 "Odd" 跟 "Even" Phase 在這個作業中與網路上查到的東西有所出入，網路上所說的奇偶排序是指在一串 array 中跳著 index 做 sorting，但在作業一當中的奇偶指的是 rank ID，透過與隔壁 process 做資料交換、排序，才能完成整體資料的排序，在理解完後會發現實作起來其實蠻抽象的，因為會不太確定資料是否有正確的排序好，並且很難去 debug，在實作過程我遇到過很多 wrong answer，最後是寫一個 print `.out` 的程式碼做 debug 才把測資跑過的，這個 bug 就是先前提到的當 data size n < process 的情況。這次的作業除了讓我學會怎麼使用 MPI 去時做平行化程式，還讓我學會一些不同 module 的切換也可以使得效能更好。但是在程式整體 speedup 我認為還有很多的進步空間，判斷應該是 process 分法不夠好，才導致某些 testcase outlier 很多。