

# HW1 - Odd-Even Sort

106062322 江岷錡

## Implementation

本次實作中，強調 Odd-Even Sorting 的跨 Node 排序。首先，由於有多個 Process 可使用，因此我們進行以下處理：(假設有  $N$  筆資料與  $K$  個 Process )

1.  $N < K$  時：由於目前 Process 數量大於擁有的資料，我們首先讓前面的 Process 分到 1 筆資料。剩餘的 Process ( $K > N$ )，則將他們丟離 MPI\_COMM。透過 stackoverflow 的解說，我試著重新創建新的 communicate group `mpi_comm`，並讓其餘未使用的 process，脫離這個 group，並將其 MPI\_COMM 參數會變為 MPI\_COMM\_NULL。
2.  $N \geq K$  時：當大於 process 數量時，仍須考慮到無法整除的問題。根據嘗試兩種方法後，發現以：先將  $N$  除以  $K$ ，得出的商  $s$  與餘數  $m$ 。接著，讓  $m$  以前的每個 process 多分一筆資料，即可完美均分資料。

透過完整切分資料後，便可直接使用 `MPI_File_read_at`，每個 process 找出自己的 read offset 後，將資料取出。

接著，再進行 local data 的 sorting。在實作中，我嘗試了兩種作法：`merge sort`、`quick sort`。

### Sorting Method

<u>Aa</u> Algorithm	<u>≡</u> Best case	<u>≡</u> Worst case	<u>≡</u> Average
<u>Quick Sort</u>	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
<u>Merge Sort</u>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

由於在 Worst case 情況下，`merge sort` 有較低的複雜度，實際在極大測資下，也驗證出相同結果。因此最終以 merge sort 進行 local sorting。

接著，便進入本次 Odd-Even sorting 的實作環節。若總共有  $K$  個 Process，Odd-Even Sorting 最多只需執行  $K$  次，即可排序完成。因此，我設計一個 for-loop 迴圈，每個 phase 為一個 iteration，並讓其最多執行  $K$  次。odd-even sort 的原型如下所示：

Unsorted								
9	7	3	8	5	6	4	1	Phase 1(Odd)
7	9	3	8	5	6	1	4	Phase 2(Even)
7	3	9	5	8	1	6	4	Phase 3(Odd)
3	7	5	9	1	8	4	6	Phase 4(Even)
3	5	7	1	9	4	8	6	Phase 5(Odd)
3	5	1	7	4	9	6	8	Phase 6(Even)
3	1	5	4	7	6	9	8	Phase 7(Odd)
1	3	4	5	6	7	8	9	
Sorted								

在每次 iteration 中，會先找出自己的 partner rank id，並進行雙方的資料溝通。經過測試，我們使用 `MPI_Sendrecv` 作為溝通方式，降低 Communication overhead。同時，單次的資訊交換，就是各自將 **全部擁有的 data** 一次傳給對方，降低溝通次數。當雙方分享完各自的 **全部資料**後，每個 Process 便會開始內部的再 sorting。在任一 process pair 中，process 若位在左側的位置，則代表需要收集前半部的資料。

在任一 process pair 中，process 若位在右側的位置，則代表需要收集後半部的資料。

為了節省時間，此部分的 sorting 方式，則是以 `找滿了就停止` 的 sorting 技巧。舉例來說，假設我們要找前半部較少的資料，則透過遍歷以下步驟：

1. 比較自己的最小資料，和收到的最小資料
2. 將兩者中最小的資料塞入 `temp array`，並進行 `index++`
3. 遍歷多次後，若 `temp array` 尚未塞滿，則分別檢查 `my_data` 與 `other_data`，將剩餘的資料補齊。

```

int my_idx = 0, other_idx = 0, out_idx = 0;
while(out_idx < my_size) {
    if (my_idx < my_size && other_idx < other_size) {
        if(my_data[my_idx] < other_data[other_idx]) tmp[out_idx++] = my_data[my_idx++];
        else tmp[out_idx++] = other_data[other_idx++];
    }
    else if (my_idx < my_size) tmp[out_idx++] = my_data[my_idx++];
    else tmp[out_idx++] = other_data[other_idx++];
}

```

在我實作中，為了加快排序進程，我實作了 **early stop** 的技巧。事實上，每次要進行 sorting 前，會先檢查左邊的 **Process** 擁有的最後一筆資料，是否小於右邊 **Process** 擁有的第一筆資料。

若有此情況，則代表本次不需要進行 sorting，也就不會再進 sorting function。而當該 phase 沒有發生任何 sorting 時，則會直接 break 迴圈，讓 Odd-Even Sort 提早結束。檢查方式是將自己有沒有 sorting 的訊息，用 **MPI\_Allreduce** 分享給所有 process，用一次 MPI 溝通完成確認。

```

for (...) {
    int is_swap = 0;
    ...
    if (local_data[local_size - 1] > my_temp[0]) {
        merge(local_size, right_size, local_data, my_temp, LOW);
        is_swap = 1;
    }
    ...
    int sum_up = 0;
    MPI_Allreduce(&is_swap, &sum_up, 1, MPI_INT, MPI_SUM, mpi_comm);
    if (sum_up == 0 && !is_first_time) break;
}

```

最後，再透過 **MPI\_File\_write\_at**，將各自的資料寫回 output file，完成本次作業要求。

## Some improvements & trials

1. fewer MPI is perfect: 越少的溝通次數，包括用 MPI\_Allreduce 取代 MPI\_reduce + MPI\_Bcast，對於整體的效能有極大的幫助。

2. 降低效率的計算：`%` \ 乘除 都對計算上產生極大效能衝擊，找出替代方法進行計算，可有效提高效率。
3. Non-blocking 的疑慮：在本次實驗中，MPI 互動總共試過了 `MPI_Send` + `MPI_Recv` / `MPI_Isend` + `MPI_Irecv` / `MPI_Sendrecv`，第二種 (Non-blocking) 的方法，在某些時候會導致傳輸的值 overflow，並讓結果出錯。也因此，本次實作便不使用任何 Non-blocking 的方法。
4. Early declare function：由於本次實驗中有額外寫多個 Function 供呼叫，透過實驗發現將 merge sort function 用以下兩種寫法：
  1. 移到 main function 前
  2. 移到 main function 後，main function 前只做基本 function 定義方法一會比方法二快上 6 seconds 左右時間。

## Experiment & Analysis

### Methodology

#### System Spec

使用學校提供的 cluster

#### Performance Metrics

本次使用 `clock_gettime` 進行採樣 computation time, I/O time 及 communication time。由於用此 function 可達成 nanosecond 級的採樣，得以更細緻的進行表現差異的分析。

一個標準的 `clock_gettime` 範例如下：

```
struct timespec start, end, temp;
double time_used;
clock_gettime(CLOCK_MONOTONIC, &start);

.... something to be measured ...

clock_gettime(CLOCK_MONOTONIC, &end);
if(end.tv_nsec < start.tv_nsec){
    output = ((end.tv_sec - start.tv_sec - 1)+(nano+end.tv_nsec-start.tv_nsec)/nan
o);
} else {
```

```

    output = ((end.tv_sec - start.tv_sec)+(end.tv_nsec-start.tv_nsec)/nano);
}
time_used = temp.tv_sec + (double) temp.tv_nsec / 1000000000.0;

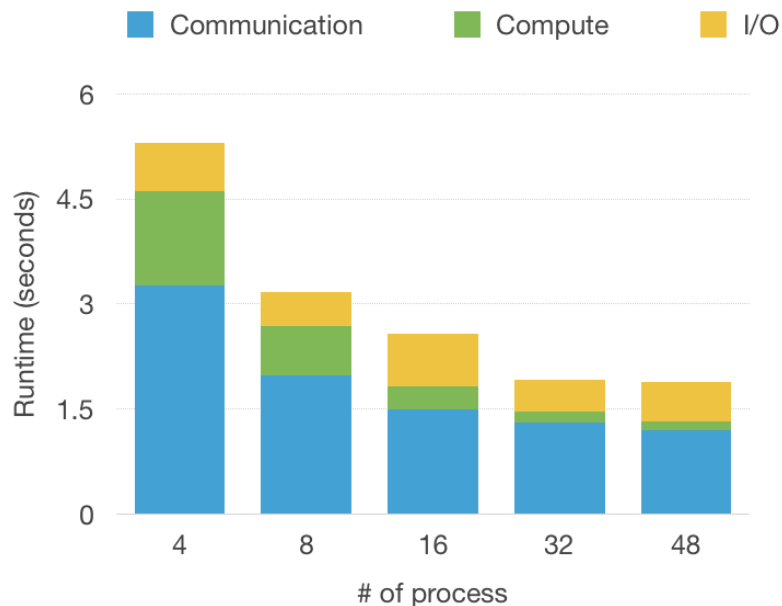
// Get what we want to evaluate
printf("%f second\n", time_used);

```

## Strong Scalability

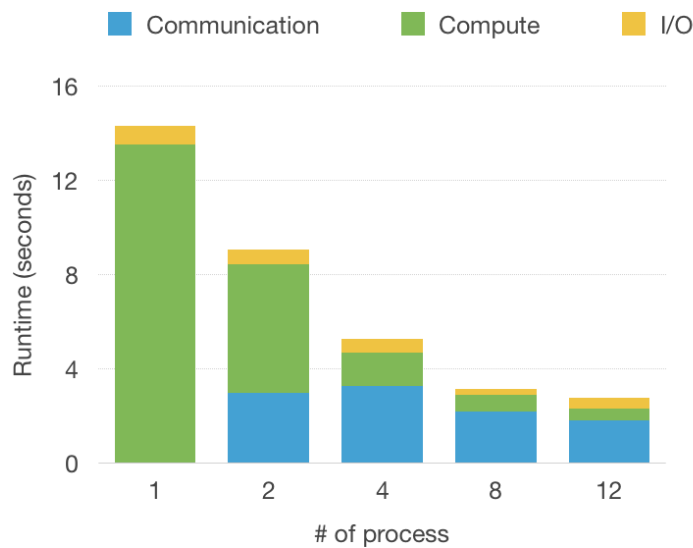
### Time Profile(Multiple Nodes = 4)

	Communication	Compute	I/O	Overall
4	3.271437	1.337375	0.694561	5.303373
8	1.985527	0.705102	0.473402	3.164031
16	1.489519	0.333748	0.753885	2.577152
32	1.304226	0.153394	0.460394	1.918014
48	1.204163	0.126012	0.556855	1.88703



### Time Profile(Single Node)

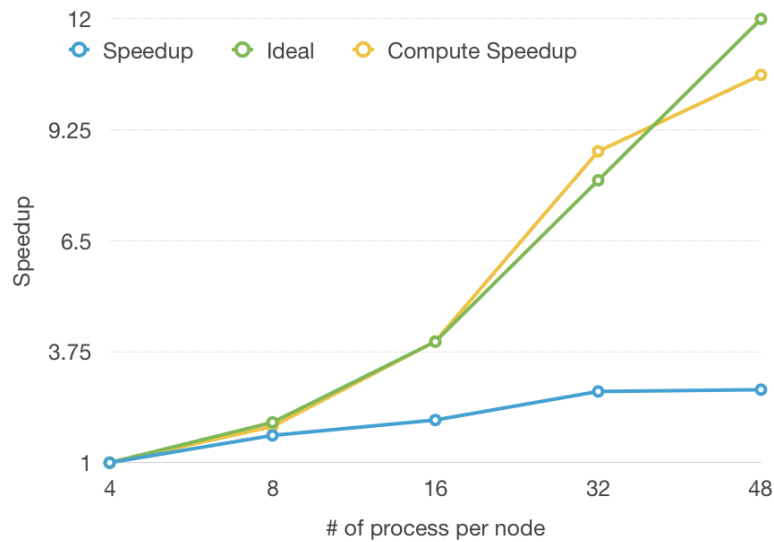
	Communication	Compute	I/O	Overall
1	0	13.527215	0.788679	14.315894
2	2.952286	5.487376	0.630761	9.070423
4	3.263282	1.406967	0.588521	5.25877
8	2.181296	0.716368	0.246197	3.143861
12	1.797199	0.492421	0.475472	2.765092



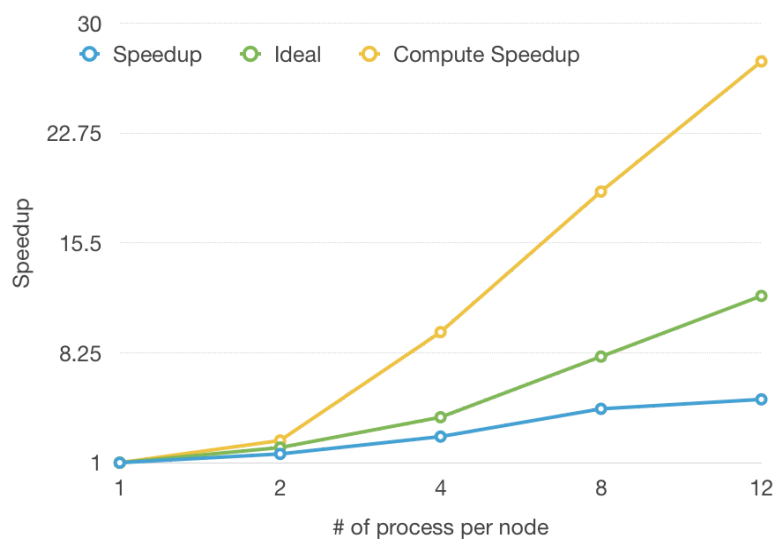
從實驗數據推測以下：

1. **Computation Time 與 Process 數量成反比**：當 Process 數量增多，單一 Process 所要處理的數據量便減少。也因此，computation time 也將逐步降低。
2. **Multi-node 時，Communication Time 緩慢下降**：當 Process 數量變多，溝通的時間也有稍微的下降，推測可能的原因，應該是單次的傳遞資料可較少，導致其現象。下方也透過實驗證明了這個假說。

**Speedup Factor(Multiple Nodes = 4)**



### Speedup Factor(Single Node)



從實驗數據推測以下：

1. **整體 Speedup (藍線) 無法達到 Ideal 情況(綠線)：**雖然隨著 Process 數量增加，Computation time 也有逐步降低，但 I/O Time 與 Communication Time 並無法有效提升，導致 Speedup 並無法完全如預期。
2. **Computation 的 speedup 高於預期：**在 Multi-Nodes 時，computation 的加速大致與 ideal 的情況貼近，但在 single node 的情況下，Computation 的加速

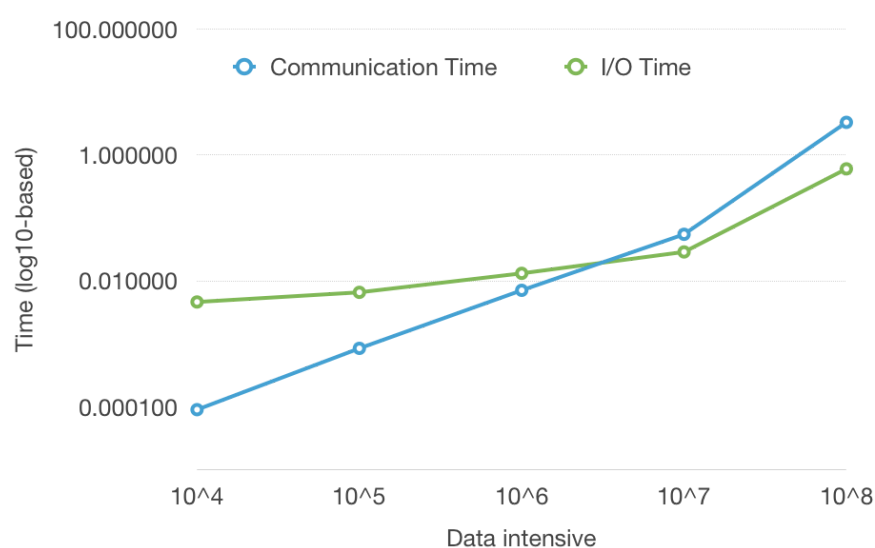
遠大於 ideal 的情況。個人推測可能的原因在於，相同資料在記憶體容易存取，讓計算時更快。

## Overall - Bottleneck

整合以上數據，由於 I/O 與 communication time 皆無法成比例的有效提升效率，此兩者為平行計算中最大的 bottleneck。若想有效降低 communication time，也許可透過 Non-blocking 的方式傳遞訊息，讓 computing time 與 communication time 得以在某些時候 overlap。然則，為求測資的穩定性，本次實作仍選擇以較安全的 Blocking communication (`MPI_Sendrecv`) 進行溝通。

## Relationship between data intensive & communication time

	Communication Time	I/O Time
$10^4$	0.000090	0.004608
$10^5$	0.000846	0.006554
$10^6$	0.007058	0.013125
$10^7$	0.054665	0.028684
$10^8$	3.271073	0.597786





在不同的資料密集度下，除了直觀的 I/O time 會與 data 量呈現正比，MPI Communication Time 與 data 量也會有密切正比關係。也因此，傳輸的量多寡是決定 Communication time 的關鍵。

## Conclusion

本次實作對於 MPI 系列有較多的了解。同時在整理數據時，才了解到如何展現實驗成果，用數據化的方式展現優化表現，甚至稍微整理實驗數據後，可以獲得新的認知及改進方向。同時，整體優化的過程也曠日費時，有時直覺想像中可優化的實作，數據上不一定也能呈現優化，不過過程中對 C 也有更深度的了解。同時也感謝 judge 團隊，輸入完 command 就能自動跑完測資也蠻神的。