Homework 1 : Odd-Even Sort Report

110164508 黃柏惟

Implementation

I. Handle an arbitrary number of input and processes

由於 input 的數量以及 processes 的數量皆為不固定的因此需要一些處理才能夠應用到任何情況下。 這邊共分為兩種情況，情況一為輸入的 input number 數量比 processes 少，這時候我會將多餘的 processes 透過 MPI_Comm_Split()的方式將其關閉。 情況二為較常見的情況，當今天輸入的 input 比 processes 數量還要多的時候我會採用平均分配的方式，並且讓剩下除不盡的分配給前方的 Processes，這樣子可以讓每個 processes 在處理資料的數量上可以盡量相同避免整份 code 在等其中一個 processes 完成 communication。

```
MPI_Comm comm_world;
if(size>n){
    if(rank>=n){
        MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, rank, &comm_world);
    }else{
        MPI_Comm_split(MPI_COMM_WORLD, 0, rank, &comm_world);
    }
    if(comm_world == MPI_COMM_NULL){
        printf("Clear Rank %d\n",rank);
        MPI_Finalize();
        return 0;
    }
    MPI_Comm_rank(comm_world, &rank);
    MPI_Comm_size(comm_world, &size);
} else{
    comm_world = MPI_COMM_WORLD;
}
```

情況一的處理

```
int basic_num = n / size;
int remainder = n % size;
int handle_num;
if(rank<remainder) {
    handle_num = basic_num + 1;
} else {
    handle_num = basic_num;
}
int offset;
if(rank<remainder){
    offset = rank * handle_num;
} else {
    offset = (handle_num+1) * remainder + handle_num * (rank - remainder);
}
```

情況二的處理

II. Sorting

在一開始 load data 時由於當前的 data 都是亂數排列的，因此需要先進行第一次的 sorting，這邊直接使用 boost library 底下的 spreadsort()，相

較於 qsort()整體執行時間上會快上不少。
在完成第一次 sorting 後後續在 processes 溝通時就不需要再採用 sorting，因為此時的 data 皆為排列的狀態，因此直接將兩個 array 當前最小的值相比再 merge 會比較快。

III. Odd-Even Sort

我的作法為每輪都會執行一次 odd phase sort 跟一次 even phase sort。
以 odd phase sort 為例，我會將所有奇數的 rank 的 data 往 rank-1 送。Rank-1 在收到 data 後就會比較送進來最小以及當前自己的最大值相比，由於此時雙方都是 sort 的狀態，因此只要自己的最大值比對方還要小的話我們就可以不用執行 sort；反之，如果自己的最大比接收到的 data 最小值還要小的話那麼就代表需要執行 sorting。 Sort 的方式由於雙方都是排序過的狀態因此只需要透過 pointer 去進行最小的比較就可以比較快的把雙方的 array merge and sort。在完成後我們會透過一個 sort flag 去記錄說該輪的這個 rank 有執行到 data 的變動。 並且我們會將 sort 過後的 data 後半段的部分傳回給 rank 並完成該階段的 phase sorting。
在 Even phase 時與 Odd phase 相同故不再贅述。
當完成一次 odd phase sort 與 even phase sort 後我們會呼叫 MPI_Allreduce()去把 sort 的 flag 收回來進行檢查，如果有其中一個 flag 是 false 的話代表該輪有更動的當前的值因此需要全部再重新檢測一遍直到所有收回來的 flag 都是 sort 過的狀態。
在傳遞 data 的時候要特別注意邊界的處理(rank=0, rank = max_rank),因為邊界的 rank 不像一般中間的 rank 可以朝左或朝右 send 跟 receive。
以 rank=0 為例，今天在 even phase 的時候雖然他是 even 的但是他就不會往左 send data，並且他也不會從左邊接收到計算完的 data。

Experiment & Analysis

I. System Spec

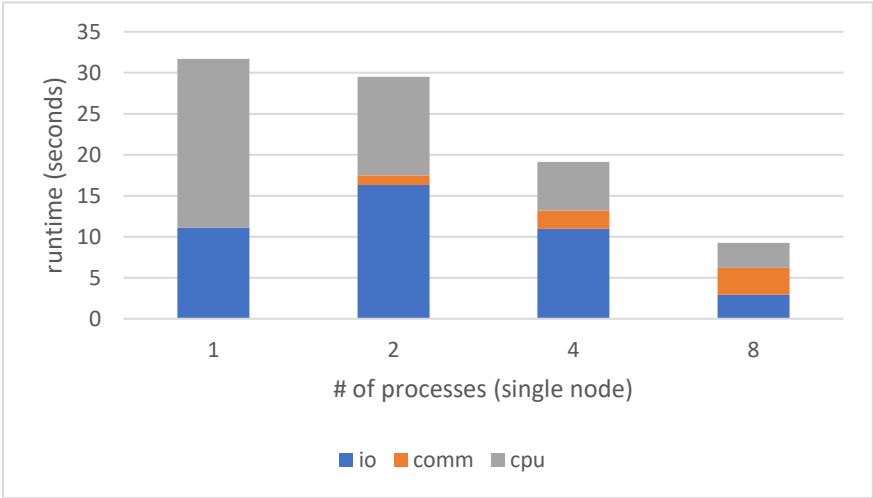
使用 appollo.cs.nthu.edu.tw server 進行實驗

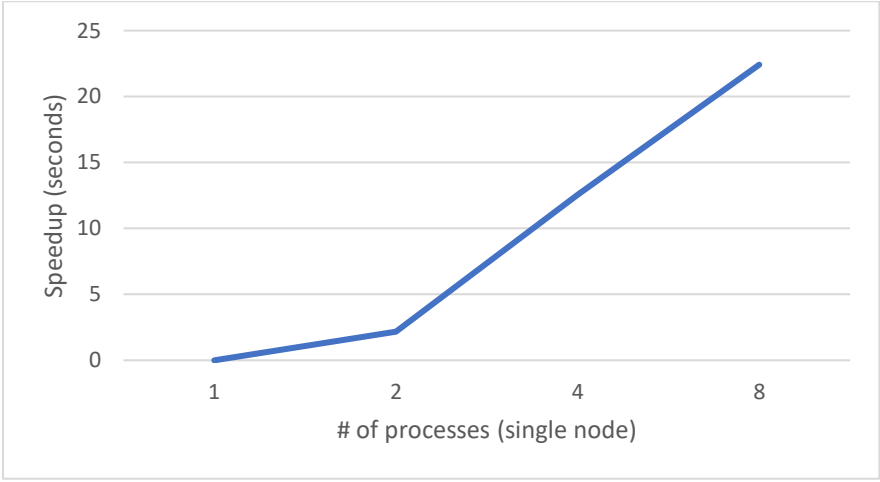
II. Performance Metrics

使用 MPI_Wtime()將要紀錄的區段進行包裹。 此次共進行三種行為分別是 IO、Processes 間的 communication time、以及 CPU 執行運算的時間在不同的 processes 以及 nodes 的執行時間進行分析。 本次分析使用第 40 筆 open test case 去進行分析，該資料有 536869888 筆浮點數資料需要進行排序運算。

實驗一、 Single node 在不同 Processes 之分析

	1	2	4	8
IO	11.104	16.329	10.971	2.95
Comm	0	1.17	2.224	3.271
CPU	20.582	12.026	5.929	3.05
Total sum	31.686	29.525	19.124	9.271
Speedup	0	2.161	12.562	22.415



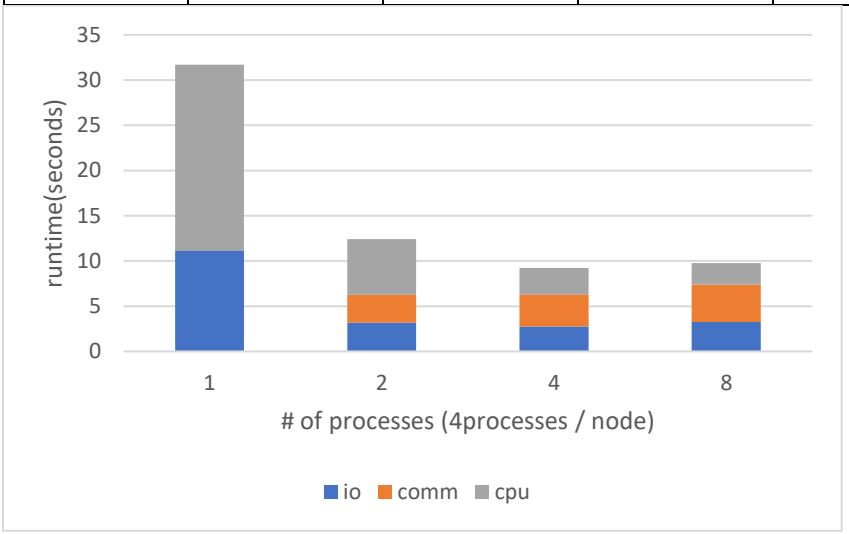


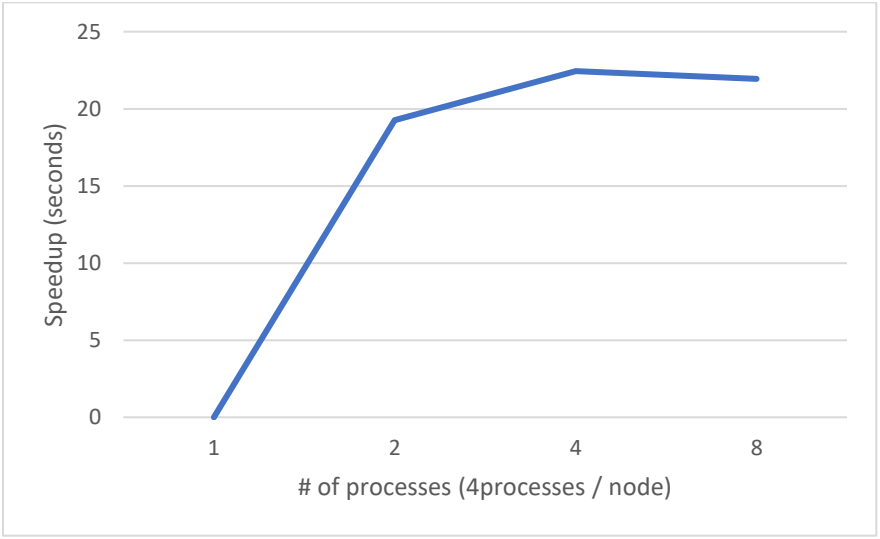
分析： 可以觀察到當 processes 數量增加的時候，透過彼此間平行運算的特性，雖然花費了一些時間在進行 communication 上，但是卻可以大幅的縮減 IO time 以及 CPU time。 這邊在 single node 下加速有符合預期以線性的方式提高。

實驗二 、Multi node 在不同 Processes 之分析

我們採用 4 processes 可以分配到一個 node 的方式進行分析

	1	2	4	8
IO	11.104	3.18455	2.76	3.256
Comm	0	3.066	3.546	4.145
CPU	20.582	6.169	2.937	2.348
Total sum	31.686	12.419	9.243	9.749
Speedup	0	19.266	22.443	21.937





分析： 可以看到在 processes 數量提升的同時如果給予對應的 node 數資源的話就可以改善在 single node 下提升 processes 數卻無法有更快執行速度的問題。 這邊的 bottle neck 是在當我把 core 數量提高到 8 時，整體的執行時間反而因為 communication time 過長而導

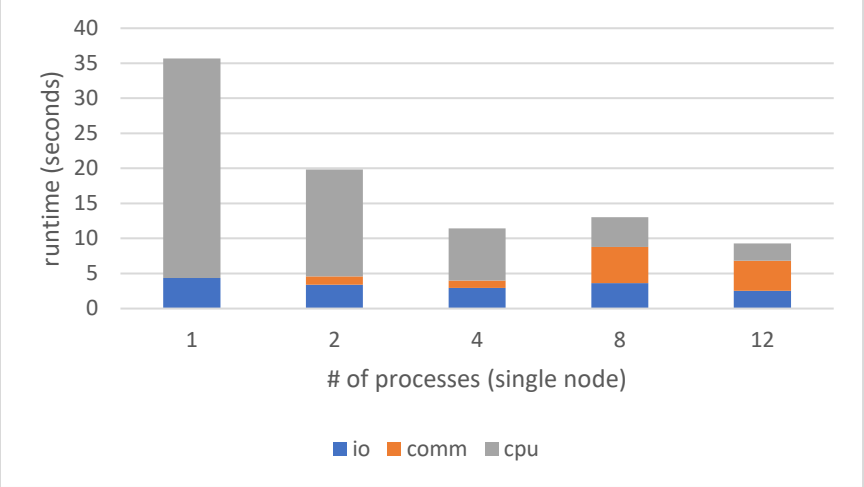
致 speedup 反而沒有 4 cores 時來的好。這邊推測是因為在 core = 4 的時候其實 CPU time 已經相對很短了，因此雖然我們在 core = 8 時候有更多的資源去計算但是卻不足以彌補 communication time 的增加。因此如果想要再增加 processor 的話很有可能就發生 communication time dominate，改進的方向可能就要著手於減少 communication time，讓已經排序過的資料不要再重複的傳遞。

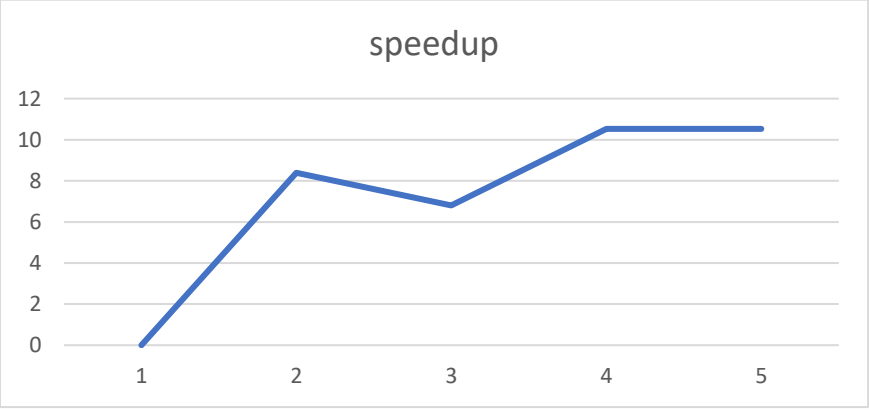
實驗三 、Sorting Algorithm 分析

由於在測試的時候有發現 Sorting 的演算法對於這次的作業其實佔了蠻大部分的 CPU time，因此就想比較不同 sorting 演算法對於 processes 改變時的變化差異。

使用 Qsort:

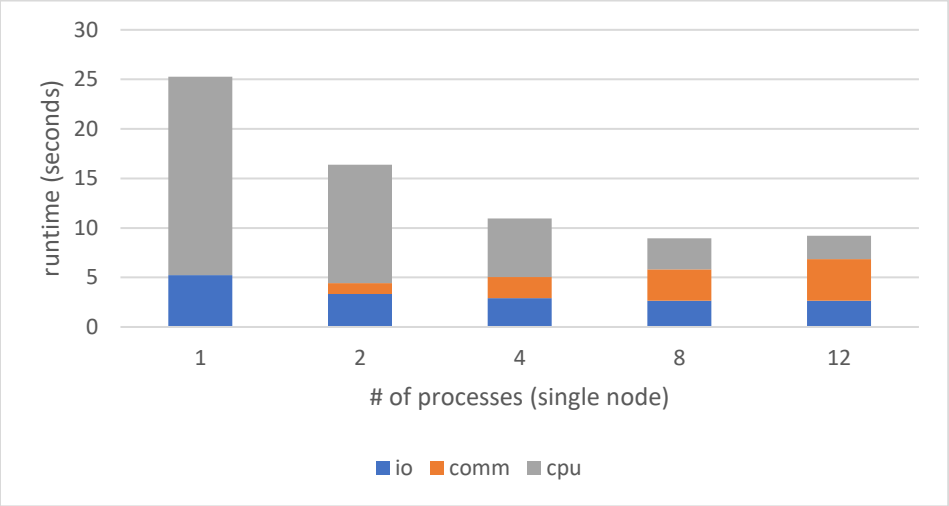
	1	2	4	8	12
io	4.38	3.391	2.942	3.614	2.5277
comm	0	1.178	1.05	5.16	4.2937
cpu	31.284	15.263	7.447	4.256	2.482
total sum	35.664	19.832	11.439	13.03	9.3034
speedup	0	8.393	6.802	10.5286	10.5286

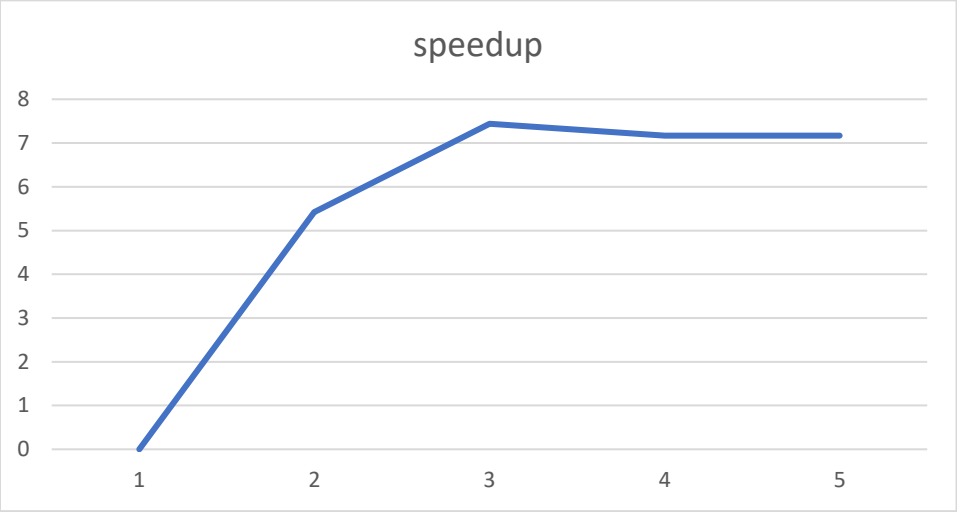




使用 Spreadsort:

	1	2	4	8	12
io	5.248	3.328	2.896	2.638	2.647
comm	0	1.105	2.15	3.173	4.195
cpu	20.01	11.956	5.924	3.138	2.382
total sum	25.258	16.389	10.97	8.949	9.224
speedup	0	5.419	7.44	7.165	7.165





分析:當我們的 processor 數量越少時代表一個 processor 需要 handle 越多的 data。因此可以從表中發現當今天發現到如果 processor 數量增加的同時 speedup 有明顯的優化的話，那麼代表 code 的 bottle neck 可能是在 CPU 的計算上，因此可以先著手於 cpu time 上的優化；如果今天發現 processor 數量增加的同時 speedup

只能有些微的改善時，那麼就代表 code 的 bottle neck 可能不是在 CPU 的計算上而可能是 communication 上。 根據實驗三了解到不同的 speedup 值可能代表著當前 code 不同的 bottle neck。

Conclusions

算是第一次寫平行程式，一開始花了不少時間在理解平行化的概念。開始實作時也遇到了很多次 Send/Receive 沒有對起來使整份 code 卡死的情況發生。不過在完成後可以發現整體的執行時間有因為平行化的處理而降低不少，蠻有成就感的。