hw1_109062320

Homework 1: Odd-Even Sort
109062320 朱季葳

Implementation

Baseline

HANDLING ARBITRARY NUMBER OF INPUT ITEMS

在這個部分首先我先對communicator做shrinking來確保process的數量不會比data的數量多，所以就只剩下process數量小於等於data數量的情況，同時也可以避免浪費資源，因為我在這邊的做法是平均分散data到各個process去做處理來達到load balancing的目標。另外在這個部分因為要盡可能地達到load balancing，所以當資料的數量(n)大於process的數量(size)的時候，我會再把之後第size + 1 ~ n筆資料從rank0開始平均分散，直到該rank大於等於n%size。舉例來說，當我們有5筆資料還有3個process時，rank0~1就會分到2筆資料，而rank2就會分到1筆資料。而在這裡我將每個process所需要負責並讀寫的data數量存在一個稱為data_to_solve的變數，還有透過當前的rank來決定讀取file的offset，而根據前面的data分配方法，在這裡可以分為負責較多data的rank(for rank < n % size)，還有其餘的process。而在前者中，因為多的資料是從rank0開始分，所以前面已經有 `rank * (n % size) + rank` 筆資料有處理過了，在後者中就有 `rank * (n % size) + (n % size)` 筆資料已經處理過了。

P1: Handling the number of data to be processed and the file read/write offset

```
int data_to_solve, start_offset;
if(rank < n % size){
    data_to_solve = n / size + 1;
    if(rank == 0)start_offset = 0;
    else start_offset = rank * ( n / size ) + rank;
}
else{
    data_to_solve = n / size;
    start_offset = rank * ( n / size ) + ( n % size );
}

float* data = (float*) malloc((data_to_solve) * sizeof(float));
```

P2: open file and read data

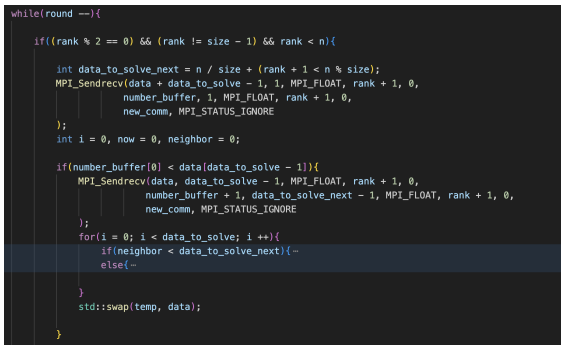
```
MPI_File_open(new_comm, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &input_file);
MPI_File_read_at(input_file, sizeof(float) * start_offset, data, data_to_solve, MPI_FLOAT, MPI_STATUS_IGNORE)
MPI_File_close(&input_file);
```

SORTING ALGORITHM

在sorting algorithm的部分，我會先針對在前面讀到的data做一次sorting，然後接下來再用一個迴圈跑odd even sort，並依據chap2講義最後一面所提供的演算法來實作odd even

sort，而在這裡因為在每次的iteration中，我都把odd phase和even phase跑完，所以我只需要 `n/2 + 1` 回合就可以把sorting完成(原本odd, even phase分開在不同iteration的做法會需要n + 1回合)，然後在實作的部分我會依據rank的單雙號還有是否位於邊界來作為判斷並依據講義上面給的指示傳data給鄰居(process with rank + 1 or process with rank - 1)然後再與鄰居的data做merge並把sort完的結果存到一個temp array(在這裡的做法是先記錄自己和鄰居的index然後互相比較當前index在array中的值哪個比較小就填入那個到temp，在把index加加直到temp被填滿)，最後再把temp和原本存放data的array做swap。

- P3:上圖擷取的部分為講義中parallel code的第一步，下列針對code其餘的部分是優化的範疇，將在**SendReceive-data**進行說明



```
while(round --){

    if((rank % 2 == 0) && (rank != size - 1) && rank < n){

        int data_to_solve_next = n / size + (rank + 1 < n % size);
        MPI_Sendrecv(data + data_to_solve - 1, 1, MPI_FLOAT, rank + 1, 0,
                     number_buffer, 1, MPI_FLOAT, rank + 1, 0,
                     new_comm, MPI_STATUS_IGNORE
        );
        int i = 0, now = 0, neighbor = 0;

        if(number_buffer[0] < data[data_to_solve - 1]){
            MPI_Sendrecv(data, data_to_solve - 1, MPI_FLOAT, rank + 1, 0,
                         number_buffer + 1, data_to_solve_next - 1, MPI_FLOAT, rank + 1, 0,
                         new_comm, MPI_STATUS_IGNORE
            );
            for(i = 0; i < data_to_solve; i ++){
                if(neighbor < data_to_solve_next){
                else{

            }
            std::swap(temp, data);

        }
```

Optimization

RADIX SORT

原本在baseline的部分，我使用STL的sort來排序local data，但是因為他的時間複雜度是O(nlogn)，而radix sort的時間複雜度是O(d(n+k)) for every element consists of `d digits` each of
which is an integer in the `range [0..k-1]` ，所以當data的range在 `[0..n^d]` 的時候，時間可以壓到linear time，所以在這裡選用radix sort進行sorting。此外，因為要對data的32個bit進行分組(分成四組)個別做counting sort，再加上sign bit會使得負數在比較的時候比正數大，所以要額外對每個data做shift還有masking之後變成unsigned int才能確保counting sort的答案正確，然後最後sort完之後再轉回floating point。

P4: converting floating point to unsigned int

```
static inline unsigned int Float2Int(float input){
    unsigned int ret = *(unsigned int*)&input;
    ret ^= -(ret >> 31) | 0x80000000;
    return ret;
}
```

P5: counting sort

```
/* Reset counters */
memset(b0, 0, sizeof(unsigned int) * (kHist * 4 + 5));
/* counting */
for(int i = 0; i < data_to_solve; i++){
    array[i] = Float2Int(data[i]);
    b0[_0(array[i])]++;
    b1[_1(array[i])]++;
    b2[_2(array[i])]++;
    b3[_3(array[i])]++;
}
/* prefix sums */
unsigned int sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
unsigned int tsum;
for(int i = 0; i < kHist; i++){
    tsum = b0[i] + sum0;
    b0[i] = sum0;
    sum0 = tsum;

    tsum = b1[i] + sum1;
    b1[i] = sum1;
    sum1 = tsum;

    tsum = b2[i] + sum2;
    b2[i] = sum2;
    sum2 = tsum;

    tsum = b3[i] + sum3;
    b3[i] = sum3;
```

SEND/RECEIVE DATA

在原始parallel code演算法的內容描述到把自己的local data傳給鄰居，但是有時候自己跟鄰居merge之後不用重新再進行sorting，所以再互相把自己的所有local data傳給鄰居會變得很耗時，然後又要花額外的時間去進行merge和swap，所以我把寫法改成先傳送自己的邊界值給對方，例如在process要傳給rank比他多1的右鄰時，因為彼此的data都已經先進行過sorting，所以只要比較右鄰最小的element(number_buffer[0])和自己最大的element(data[data_to_solve - 1])，並在自己最大的值比右鄰最小的值大的時候，再互相傳送其餘data給對方就好，並在進行merge和swap，否則就什麼都不做。

OTHERS

在其餘的部分我所進行的小優化主要有以下幾點

1. 減少分配記憶體的量，像是在odd-even sort的部分本來右鄰跟左鄰我是分開存在兩個不同的array，因為他們的size有可能不同，但是後來我直接都取最大的size，也就是在**handling-arbitrary-number-of-input-items**的部分所提到過的，當總data數量比process數量多時，rank數字較低的process就要分到多一筆data，而我在這裡預設大家的鄰居都會被分到多一筆data，然後重複使用同一個array去記錄鄰居的local data。並且因為有額外的變數去記錄鄰居的data數量，所以也不會hurt到correctness，同時也能減少空間的使用量。

2. 把所有new改成malloc，delete改成free，還有在initialize array的時候使用memset而不是跑一個迴圈去把值初始化，因為平均來說使用malloc, free, memset會跑得比較快，不過在這裡並沒有看到很明顯的效能差異。
    - reference:
        - **are-calloc-malloc-faster-than-operator-new-in-c** (https://stackoverflow.com/questions/23591196/are-calloc-malloc-faster-than-operator-new-in-c)
        - **are-zero-initializers-faster-than-memset** (https://stackoverflow.com/questions/40786375/are-zero-initializers-faster-than-memset)
        - **is-memset-more-efficient-than-for-loop-in-c** (https://stackoverflow.com/questions/7367677/is-memset-more-efficient-than-for-loop-in-c)
3. 把radix sort從function移出來，不透過call function而直接在讀完資料後實作radix sort，因為在call function的時候還要額外對caller-saved register和callee-saved register做load/store所以可能會hurt performance，還有把其餘的function都改為static inline function，不過這個優化帶來的效益其實並沒有很明顯，可能是因為compiler已經有把一些code優化掉了。

Experiment & Analysis

Methodology

SystemSpec

課程所提供的Apollo cluster

Performance Metrics

在測量時間上面我主要使用 `MPI_Wtime()` 還有 `Nvidia Nsight System` 去測量I/O time, communication time還有elapsed time。

- `MPI_Wtime()`
    - `I/O time` 分別在read和write的時候開檔案前標註一個時間戳記，到關檔案後再把新的時間戳記減去開檔案前的，最後再把兩個加起來

    - `communication time` 取MPI_Sendrecv前後的時間來做計算。
    - `elapsed time` 則是分別取最一開始 `MPI_Init()` 過後作為時間戳記還有 `MPI_Finalize()` 作為結尾的時間戳記
    - `computation time` 則為 `elapsed time` 減去 `I/O time` 和 `communication time`
- `Nvidia Nsight System`
    - I/O time:將 `MPI_File_open` , `MPI_File_read_at` , `MPI_File_close` , `MPI_File_write_at` 的時間加總
    - `communication time` 將 `MPI_Sendrecv` 的時間加總
    - `elapsed time` 則是將 `MPI_Finalize()` 的結束時間減掉 `MPI_Init()` 的開始時間
- 最後再將上面兩者互相比對確認實驗的正確性
  此外，單一種類的MPI library call則使用 `mpiP` profile去看各個mpi call所花費的時間，還有觀察有沒有做到load balancing，並在最後使用各個process所算出來的結果做平均來產生Single node, Multi node environment的實驗圖表，以及使用mpiP統計出來的結果來產生Load balancing部分的實驗圖表。
- P6: calculating I/O time, communication time, computing time, elapsed time

```
if(EXPR_MODE){
    // I/O start
    // [TODO] : final time - io time - communication time = computation time
    double final_time = MPI_Wtime();
    io_time_tmp = final_time - io_time_tmp;
    io_time += io_time_tmp;
    final_time -= start_time;
    double comp = final_time - io_time - communication_time;
    printf("I/O time = %f\nComputation time = %f\nCommunication time = %f\nElapsed time = %f\n", \
    io_time, comp, communication_time, final_time);
}
```

Plots: Speedup Factor & Profile

Experimental Method

Test Case Description:

在測資方面，我使用第35筆測資，因為在scoreboard上面，大家普遍都是這筆測資跑得最慢，所以想要透過對這筆測資進行實驗看有沒有能更顯著使performance成長的方式。

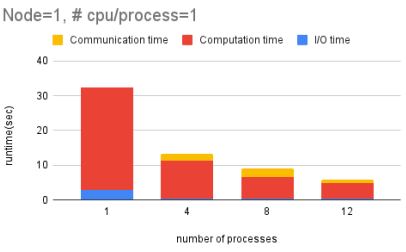
Performance Measurement && Analysis of Results

- data數量：536869888

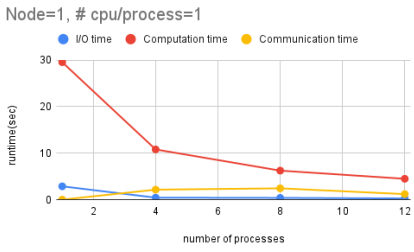
Single Node

- Time Profile:

P7-1



P7-2

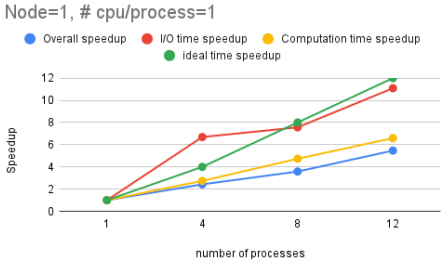


從上圖P7-1可以看出在1 node, 1cpu/process的環境下，隨著process的數量增加，runtime也隨之減少。接下來我將逐一對各個metric進行分析。

1. `computation time`：隨著process數量的增加有顯著的減少。由此可見將資料分散並平行處理他們的功效。
2. `communication time`：可以從P7-2看到communication time在一開始和process數量成正比，但是在process數量為12的時候可以看見他反而下降了，我推測這是因為每個process要傳給鄰居的data量都大幅減少了，再加上由於我使用的是blocking的MPI_Sendrecv，而從P11-1~4有關於testcase35的load balancing圖表可以看到在12個process的狀態下以及8個process的狀態下，load balancing的程度並沒有差很多，所以造成12個process的狀態下communication time減少的原因應該和Decreased Message Size比較有關係。
3. `I/O time`：從P7-2可以看到I/O time一樣是和process數量呈反比，而這其中的原因除了因為平均分散資料給每個process而減少的讀寫量，還有因為我在讀寫上是使用 `MPI_File_read()`, `MPI_File_write()` ，所以每個process在讀寫file都是獨立的，也因為不需要等待其他正在做I/O的process所以省下不少時間。

- Speedup:

P7-3



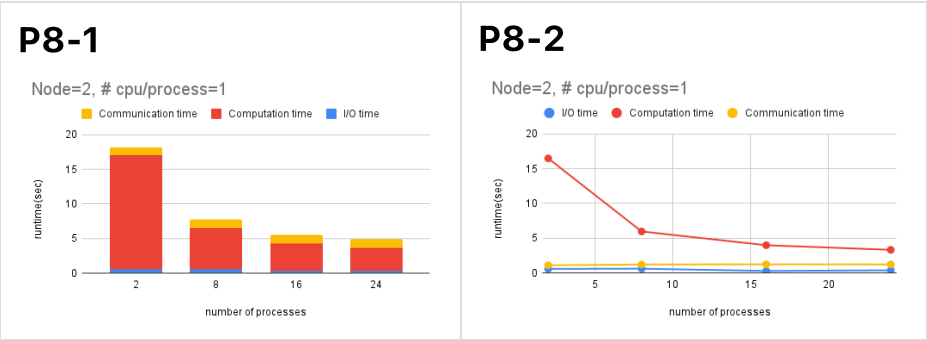
在speedup的部分，可以從P7-3看到在process數量為4的時候I/O time speedup大幅提升甚至超過ideal的值，而我

猜測這是因為只有一個process的情況下，該process要存取所有的資料，而底下的thread都要等待前面的thread做完I/O之後才能做I/O，否則就會hang在那裡。而相較之下，四個process的狀態下除了要讀取的資料減少以外，底下的process所需要等待的時間也是成倍減少的，所以才會出現speedup比ideal高的狀態。此外，我們也可以從圖中看到I/O time speedup在之後就回歸接近ideal time speedup的曲線了，我想這應該是因為process底下的thread等待做I/O的時間不是可以線性量化的，所以才會出現這樣的狀況。
至於在computation speedup的部分，我們可以看到他的speedup大約只有ideal的一半，而我猜測可能是因為這裡還是包含了其餘sequential code的部分，所以總體上的加速就會有限，不過可以肯定的是，在process數量增多之後，每個process需要負責的data也隨之減少，所以process的數量也與speedup倍數成正比。
而最後在overall的部分，基於communication time相較於只有一個process的情況下沒有speedup，還有computation speedup那邊只有ideal的一半，所以綜合下來的overall speedup大概只有ideal的一半而已。

Multi-node

2 Node

- Time profile



從上圖P8-1可以看出在2 node, 1cpu/process的環境下，隨著process的數量增加，runtime也隨之減少。接下來我將逐一對各個metric進行分析。

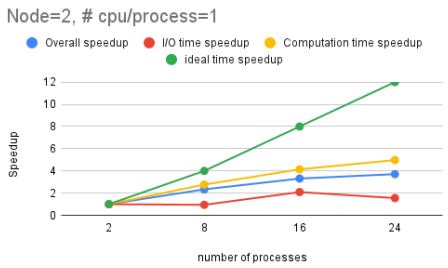
1. `computation time`：和1 node的環境下一樣，隨著process數量的增加有顯著的減少。
2. `communication time`：這裡和1 node最不一樣的點就是，他還額外多了node和node之間溝通的時間，而因為所有process被均分到兩個node上面，所以當process數量為

2的時候，那個時間即為兩個process在node之間的溝通時間。不過我們可以從P8-2看到communication time的增長幅度其實並不大，所以我猜測在分process的時候，以process數量8為例，可能是rank0~rank3的process在node1, 而剩下的在node2，所以並沒有過多node之間的溝通。如果細看同樣在8個process的情況，一個node和兩個node的環境之下，甚至可以發現兩個node的communication time並沒有比一個node的環境還多(詳細的數據在最後會附上記錄的表單)，而我推測這可能是因為有平均的分散資料，再加上只有兩個process需要進行node之間的溝通，所以就不會太過耗時。

3. `I/O time` ：在這裡的情況和single node所呈現出來的數據沒有相差很多，同樣是I/O time和process數量大約呈反比，所以就不再贅述了。

- Speedup

P8-3



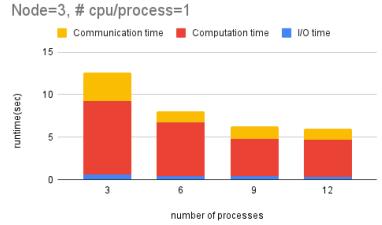
在這裡可以看到computation的speedup和1個node的時候差不多，都是和process數量成正比。但是在I/O time就沒有像之前一樣speedup的程度超過ideal，我們可以看到I/O time隨著process的增加，並不會一直不斷的加速，我想這大概是因為在讀取記憶體時，原本就需要的時間能被加速的空間有限，其中還可能包括cache miss, page fault等等的因素會因此而耗時，所以就沒有辦法呈現線性的speedup。而這樣綜合下來，整體上的加速大約不到ideal的一半而已。

3 Node

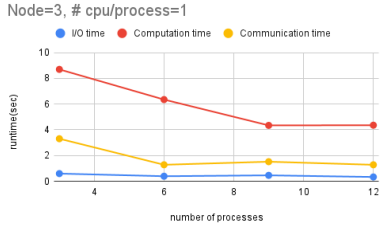
在這裡我透過更改cpu per process的數量來觀察當每個process獲得更多計算資源的時候，能夠對performance有多少影響。

- each process uses 1 CPU
    - Time profile

P9-1

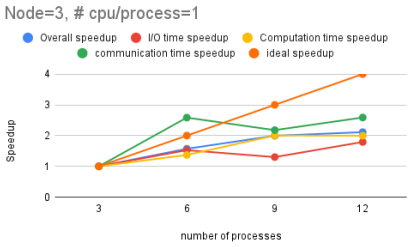


P9-2



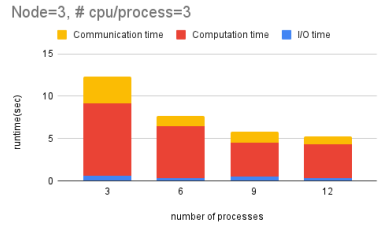
- Speedup

P9-3

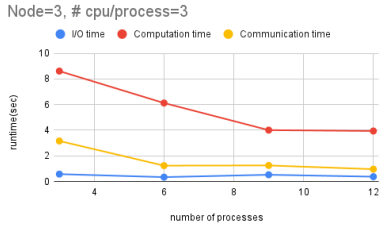


- each process uses 3 CPU
  - Time profile

P10-1

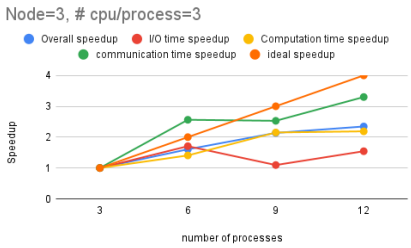


P10-2



- Speedup

P10-3



從上面兩組圖所呈現的結果來看，cpu per process的數量對於performance其實沒有太大的影響，我想這應該是因為這是一個I/O bound的程式，所以即使增加計算資源，對於performance所能improve的空間非常有限，所以可以看到這兩組圖的數據其實非常的相似。
而如果我們拿這兩組圖的數據去和前面single node,2 node的環境所跑出來的數據比對的話，會有一些額外的新發現，其中包括在3個node的環境之下，communication time隨著process的數量增多，時間卻沒有像1 node, 2 node的環境一樣變多或者持平，相反的，他加速了。而針對這點我推測是

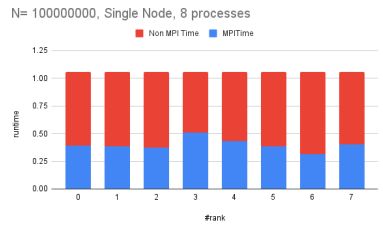
因為在單一node裡面，需要互相溝通的process數量減少了。舉例來說，同樣是12個process，在single node的環境之下，那12個process要互相溝通可能還要排隊等待IPC，但是在三個node的環境下，每個node裡面需要做IPC的process就只有4個了，即使多個node會使某些process會需要進行node之間的溝通，但是整體上來講，在這裡我們可以看到將同樣數量的process分散在多個node的效益是相當可觀的，而這也使原本的bottleneck, communication time的效能不再停滯不前。

Load Balancing

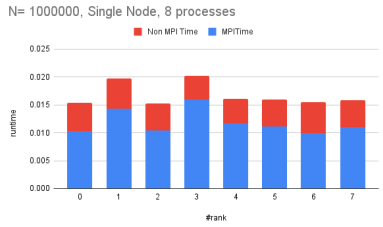
以下分別測試了第35筆以及第37筆testcase在1000000筆資料以及100000000筆資料在Single node, 12 processes的環境之下，各個process所花費的時間來觀察load balancing的程度。(Based on mpiP所給出的數據)

- Testcase : 35
  - 8 processes

P11-1

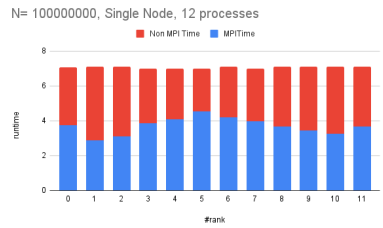


P11-2

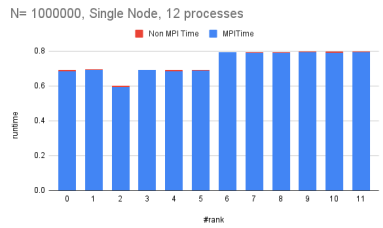


  - 12 processes

P11-3

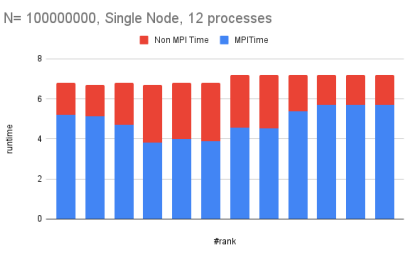


P11-4

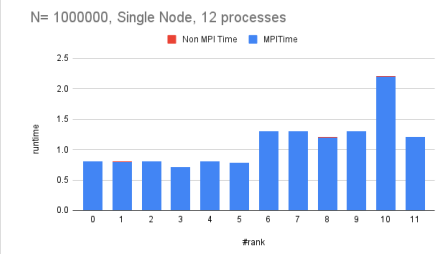


- Testcase : 37

P12-1



P12-2



從以上三組圖我們可以觀察到，data的數量不論是在哪一筆測資，都和Non MPI Time的比例成正比，而process數量和Non MPI Time的比例成反比，而這與前面所提到的data size per process降低所帶來的效益是一樣的。此外，透過圖表上的數據，我們也可以看到每個process的runtime無論是在第35筆還是第37筆測資，data和process數量為多少，時間差都差不到2秒，而這也代表我在**handling-arbitrary-number-of-input-items**所說明的方法是有效分散data並促成load balance的。

Optimization Strategies

Based on the analysis results, propose potential optimization strategies.

從實驗圖表來看，我覺得我的實作方法主要還有兩個部分可以再嘗試優化。

1. communication:在我的實作方法中，我使用的是Blocking的Sendrecv，如果在這裡換成Non-blocking的話，也許效能會有些微提升，但是因為每個process還是會有最多一筆的資料差，所以在這個部分可能還要額外處理，確定在鄰居和自己資料量有差的時候要互相等待，以免傷到correctness，但是可能也無法確定額外處理的部分加一加會不會又hurt performance，所以可能還是使用Blocking的作法可能會比較保險。
2. computation:不管在多少node, process, cpu數量的環境下，實驗數據都顯示computation time的speedup可能只有ideal的一半甚至不到一半，而在這裡我覺得針對local sort那邊的radix sort做優化，因為四次counting sort跑了好幾個迴圈在分別對資料做處理，如果可以針對這個部分再做平行化的話，也許效能可以再進一步的提升，但是這裡會有很多data dependency的問題，所以有可能要額外去互相等待其他process做完自己的部分，也會有發生deadlock的風險，所以在這個部分可能還是原本的作法比較保險。

Discussion

Compare I/O, CPU, Network performance. Which is/are the bottleneck(s)? Why? How could it be improved?

在**Performance Measurement && Analysis of Results**所呈現的圖表中可以發現以下bottlenecks

1. communication:在node數量為3以下的時候，communication time會隨著process的數量增加或者持平，但是並不會下降。而在這裡也許可以透過把Blocking Sendrecv改為non-blocking的send和recv來增進效能，但是他的效益並不大，而且正如我在optimization strategy所提到的問題，再加上在node數量是3的時候，communication time其實是會隨著process數量增加而減少的，所以改成non-blocking的效益可能不會很大。
2. computation:在這個部分，雖然speedup還是與process數量成正比，但是距離ideal還是有一段距離，所以我覺得可以再針對local sort的部分再做優化，而且這個部分的優化效益應該蠻大的，因為我在把local sort的方法換成radix sort之前是使用STL的sort，而在我改成radix sort之後用hw1-judge測試全部的測資，整體時間直接快了十幾秒，而我那個時候是將data分成兩組做counting sort，在之後我再換成四組counting sort的時候又快了6.7秒，所以可見在這邊的優化是效益很大的。

COMPARE SCALABILITY. DOES YOUR PROGRAM SCALE WELL? WHY OR WHY NOT? HOW CAN YOU ACHIEVE BETTER SCALABILITY?

在**Performance Measurement && Analysis of Results**所呈現的speedup圖表顯示，我的實作方式在不同node, process, cpu數量的環境之下，overall speedup與process數量皆呈現線性關係，但是因為距離ideal還是有一段距離，所以在scalability的方面還是有很大的進步空間，而improvement的部分則正如我在第一題discussion所提到的，針對communication以及computation的方面去進行優化，並使speedup能夠更接近ideal以達到更好的scalability。

Experiences / Conclusion

經過這次作業之後，我覺得我對於平行程式的認知不再僅僅是侷限在課本上的理論了，以前在學Amdahl's Law的時候甚至感覺扣掉sequential code之後只能增進這樣的performance怎麼感覺有點少，然後可能也只是照著題目算算performance的程度，但是在這次做完實驗之後，我覺得在parallel code的部分要達到ideal真的是非常困難的事情，在這整份作業當

中，我覺得最困難的部分大概就是針對實驗出來的數據然後去分析他了，因為有時候數據跟我想像中的結果其實落差蠻大的，像是我在3個Node的實驗數據看到communication time與process數量成反比的時候一度以為我實驗做錯了，然後前前後後用了好幾種方式去算performance metrics，其中包括ipm, mpiP, MPI_Wtime()還有Nvidia Nsight System，最後再統一比對然後才做成現在圖表的數據，還有一個讓我感到意外的點是我本來以為I/O會是我的bottleneck，因為從以前到現在被灌輸的觀念就是I/O很慢要等他才能繼續做其他事，然後還會有其他overhead，就整個把I/O妖魔化了，但是在這次的實驗結果看到communication time反而才是bottleneck，我想這可能也跟網路傳輸資料的速度不穩定有關係，然後我也有聽到有人在不同的時間做profile然後I/O的時間差了快七秒的案例，只能說實驗的數據真的蠻不可控的。而我在Optimization的部分也是收穫了蠻多的意外，其中包括減少記憶體的用量所帶來的效益，因為我原本在傳資料的時候是左鄰居右鄰居各一個array去放，後來覺得有點浪費空間就只留一個array然後統一放資料，畢竟他們沒有data dependency的問題，然後用hw1-judge跑出來的結果就快了將近五秒(當然也有可能是因為他很不穩的關係)。總而言之，經過這次的作業之後，我學到了蠻多的小細節，對於平行程式的運作還有MPI的API也更加熟悉了。

另外在這裡附上我的實驗數據表格

pp_hw1_expr#35

(https://docs.google.com/spreadsheets/d/1WZwVy6yINVC7pkh7M4VWVIui7_M9RHeBUwzve63IjYE/edit?usp=sharing)

pp_hw1_expr#37

(https://docs.google.com/spreadsheets/d/1zjcrxuivRlrbhj9M4KOuR9VGuiik8gPWGLfGZ-kJ0Ew/edit?usp=sharing)