CS542200 Parallel Programming

Homework 1: Odd-Even Sort

107062103 王依婷

I. Implementation

這次的作業要使用 MPI 實作 Odd-Event sort，以下將整個程式部分分成 I/O、process 之間的溝通及 sorting 來簡單說明：

i. I/O

做 I/O 前，首先要確定每個 process 該取到的 data 數量。input 的檔案大小對於實作有兩種狀況，一種是數字的數量（n）大於等於 process 數量（size），另一種是相反。若是第一種情況，則會將所有數字平均分配給所有的 process，也就是說每個 process 都至少會拿到 n/size 個 data，剩下餘數的部分再分配下去。

```
int data_cnt = (rank < n%size) ? n/size+1 : n/size;
```

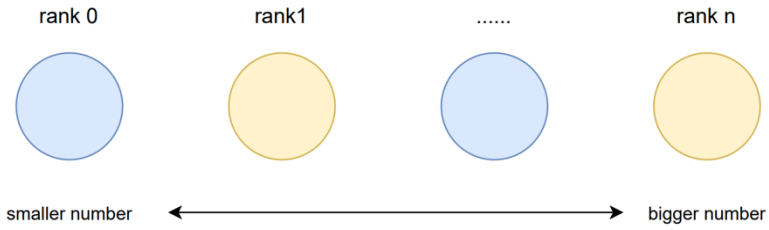
若是第二種情況，將會有一些多出來的 process 沒有事情做，因此將這些多的 process 結束掉，避免不必要的困擾。做法為建立一個新的 group 和 communicator，排除掉多餘的 process，呼叫 MPI_Finalize() 把他們結束，後續做的一切溝通都使用新的 communicator。第二種情況下每個 process 都剛好拿一個 data。

```
// deal with the case that size (# of process) > n (# of data)
if(size > n){
    int ranges[][3] = {{n, size-1, 1}};
    size = n;
    MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);
    MPI_Group_range_excl( worldGroup, 1, ranges, &newGroup);
    MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);
}
if (newComm == MPI_COMM_NULL){
    // printf("%d\n", rank);
    MPI_Finalize();
    exit(0);
}
```

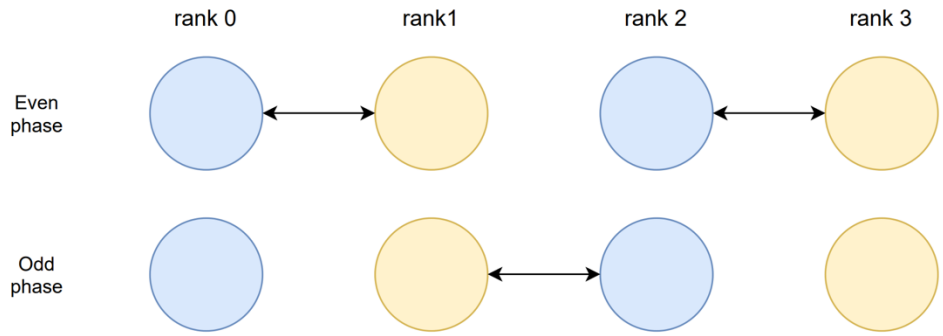
接著使用 MPI_File_open / MPI_File_close 開關檔案，再進行檔案的讀取。老師在上課時提到，MPI I/O 有 independent 和 collective 兩種，collective 在單一 process 的 data 並不大時使用會比較快，因為一次讀/寫較大量的 data 會比很多次的讀/寫少量 data 還來得快。因此我在這邊做個很粗略的判斷，如果 data 數量夠多時就使用 independent I/O（MPI_File_read_at/MPI_File_write_at），否則使用 collective I/O（MPI_File_read_at_all/MPI_File_write_at_all）。讀寫時的 offset 由 process 所取的 data 數量計算而來。

ii. process 間的溝通（Communicate）

每個 process 都有一串待排序的數字，內部排好之後會再和相鄰的 process 互相傳資料溝通比較，做進一步的排序。最後排成如圖所示之數字由小到大順序，寫到檔案中。



一開始我的做法是編號較小的 process 會將所有 data 傳給他右邊編號比較大的 process，排序好所有 data 之後將較小的那一半送回去。但這樣一來，所有的計算只集中在一半的 process 之上，非常不平衡，因此後來修改成左右兩個 process 都會將自己排序好的 data 傳送給對方，一起做排序的動作，並且留下正確的那半邊。



首先，編號較小 process 傳送最大的數字給編號較大的 process，且接收對方傳來的最小的數字，當編號較小 process 內最大的數字仍小於編號較大 process 內最小的數字，代表數字的順序已經排好，不需要再做排序，也不需要再傳接所有數字 data；反之，互傳整串 data 來做進一步的排序。要注意的是編號最小(頭)和編號最大(尾)的 process 不一定會參與傳接 data 的部分，需要用判斷式判斷。

在傳送及接收 data 的部分，我都是使用花費較少時間的 MPI_Sendrecv 來取代分開的 MPI_Send 和 MPI_Recv。這邊附上 Even phase 中 even node 的程式作為輔助說明。

```
// even phase
if(rank%2 == 0){ // even node (left)
    if(rank != size-1){ ──→ 編號最大的 process 右邊沒有別人了，不參與溝通
        MPI_Sendrecv( &data[data_cnt-1], 1, MPI_FLOAT, rank+1, 0, &tmp, 1, MPI_FLOAT, rank+1, 0, newComm,  MPI_STATUS_IGNORE);
        if(data[data_cnt-1] > tmp){ ──→ 先判斷是否需要進一步的比較、排序
            data_recv_cnt = (rank+1 < n%size) ? n/size+1 : n/size;
            float *data_recv = (float*) malloc(data_recv_cnt*sizeof(float));
            // MPI_Send(data , data_cnt , MPI_FLOAT, rank+1, 0 , newComm);
            // MPI_Recv(data_recv , data_recv_cnt , MPI_FLOAT, rank+1, 0 , newComm, MPI_STATUS_IGNORE);
            MPI_Sendrecv(data , data_cnt , MPI_FLOAT, rank+1, 0, data_recv , data_recv_cnt , MPI_FLOAT, rank+1, 0 , newComm, MPI_STATUS_IGNORE);
            // if(data[data_cnt-1] > data_recv[0]){  // need to sort
                Sort(Left, data, data_cnt, data_recv, data_recv_cnt);
                for(int i=0; i<data_cnt; i++) {
                    data[i] = new_data_own[i];
                }
                doSort = true;
            // }
            free(data_recv);
        }

    }
}else{ // odd node (right)
```

iii. Sorting (Compute)

process 內部的排序，使用 boost library 中的 spreadsort，選擇浮點數的排序 float_sort。兩個 process 進一步排序的部分，如上圖所示會呼叫自己寫的 Sort function，傳入 data 及自己的位置 (是編號較小的左邊還是較大的右邊)。

每次做進一步的排序時，都會將 doSort 設成 true，代表這次的 phase 中有做排序。當所有 process 經過一個 even phase 和一個 odd phase 後，doSort 都還是 false 的話，代表已經將所有 data 排序完成，這就是排序的終止條件。因此在兩個 phase 之後，會將每個 process

的 doSort 做 logical OR，再發送給所有 process 來判斷是否要繼續排序，這個部分我使用了 MPI_Allreduce 這個 API 來做。

而在 function 中會像是要合併兩串有序 data 一樣，一一比較兩串 data 中的數值大小，放到新的 array 中。若自己是左邊，代表要拿數值較小的那半邊，因此從兩串 data 最小的數開始比較起，直到已取得足夠數量的數字為止。若自己是右邊，則從最大的數開始比較。如此一來，不但 process 間的工作量較為平均，計算量的差異最多也就是一個數字，更不用將兩串 data 完整的合併排序完才能得到結果，每個 process 也只要比較、排序到 data 的數量足夠就好。以下附上部分程式。

```
void Sort(int isRight, float* data_own, int data_own_cnt, float* data_recv, int data_recv_cnt){
    if(!isRight){
        int need_cnt = 0, i = 0, j = 0;
        bool flag = false;
        while(i < data_own_cnt && j < data_recv_cnt){
            new_data_own[need_cnt++] = (data_own[i] < data_recv[j]) ? data_own[i++] : data_recv[j++];
            if(need_cnt == data_own_cnt){
                flag = true;
                break;
            }
        }
        if(!flag) {
            while(i < data_own_cnt){
                new_data_own[need_cnt++] = data_own[i++];
                if(need_cnt == data_own_cnt){
                    flag = true;
                    break;
                }
            }
        }
        if(!flag) {
            while(j < data_recv_cnt){
                new_data_own[need_cnt++] = data_recv[j++];
                if(need_cnt == data_own_cnt){
                    flag = true;
                    break;
                }
            }
        }
```

II. Experiment & Analysis

i. Methodology

(a) System spec: 使用課程所提供的，並沒有使用其他額外系統。

(b) Performance metric: 在每個 process 上使用 MPI_Wtime() 這個

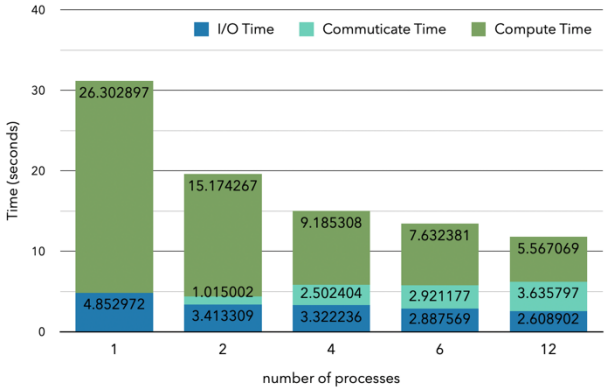
API 取得 I/O 前後、溝通前後的時間，相減後得到所花費的時長，而 Computing time 則是由全程花費時間減去 I/O 和溝通時間。最後將所有 process 的各種時間用 MPI_Reduce 加總取平均，reduce 到 rank=0 的 process 印出，每種實驗設定都跑五次取得平均，繪製成數據圖。Speed up 的部分是將實驗中最基本設定的花費時間除以其他設定花費時間，所計算出的比值。

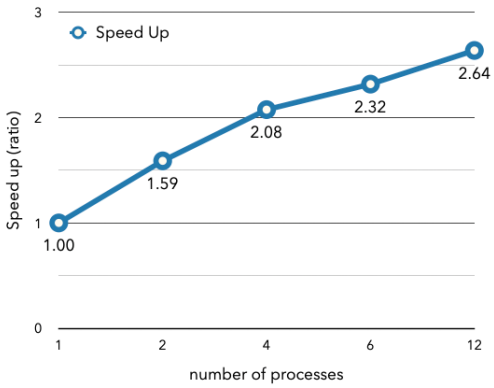
ii. Plots

取第 35 個 testcase 作為實驗的 case。

(a) Single node (N=1) with different numbers of processes (n):

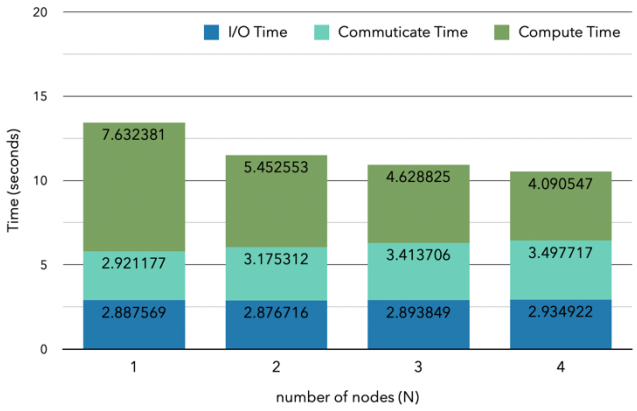
探討在固定 single node 的情況下，增加 process 數量對 performance 的改變。

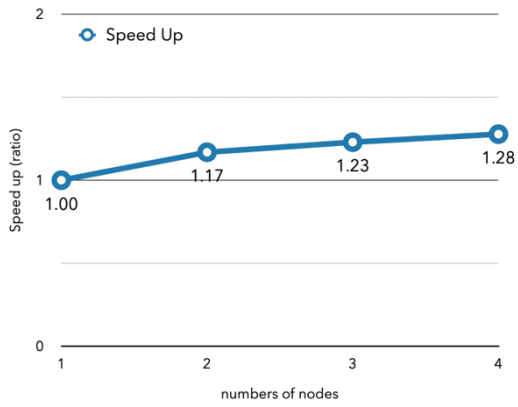




(b) Multiple nodes with fixed numbers of processes (n=6) per node:

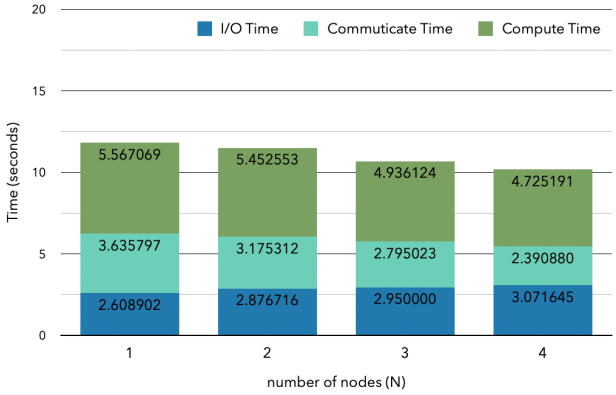
每個 node 所使用的 process 數是固定的，改變總共使用的 node 數量，看看會有什麼差異？

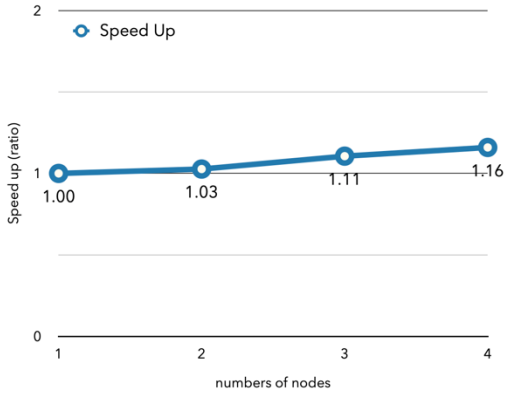




(c) Multiple nodes with fixed numbers of processes in total (n=12):

接著換成固定 process 的總量，試著分配給不同數量的 node。





iii. Discussion

(a) 除了同樣資料量的實驗之外，我也有試著以相同的配置跑不同大小的資料，發現當要處理、運算的資料量比較大時，花費最多時間的就是 Computing time (資料量較小時是 I/O)。在 ii(a)的實驗中，process = 1 時，代表所有的資料是全部一起一口氣做 sorting，而 process 數量增加代表在同一時刻，不需要排序那麼多的數字，因此 process 內部排序所花的時間會成比例減少。但是實際上的數據在 Computing time 的地方減幅是越來越小的，可知比較大的問題是出在 process 間進一步的排序，所以可以針對這部份優化，Computing time 就能夠減少。(當然，在 process 間的排序也能再使用更快的排序演算法來加速！)

不過在做實驗時，也發現如果有很多人同時在跑時，整體的速度會受到影響而變慢，其中 I/O 的速度慢了非常非常多，因此如果都是在多人使用的環境下，I/O 就會成為最大的 bottleneck。

```
[pp21s50@apollo31 sample]$ srun -n12 -N1 ./calTime 536869888 ../testcases/35.in ../testcases/test.out
Average I/O time: 51.749565
Average COMM time: 3.627085
Average COMP time: 5.577680
```

(b) 我覺得這個 program 的 scalability 並沒有到很好，從 ii(a)(b)都可以看到增加 process 數量後，speed up 的變化是趨緩的，尤其是

ii(b)，從 single node 共 6 個 process 增加到 4 nodes 共 24 個 process，speed up 都沒有超過 1.5。可以觀察到在 ii(b)中，I/O 的時間並沒有很明顯的改變，但 Computing time 減幅不大，再加上較無法避免的 Communication time 增加，造成速度差異不大。也許因為這次的 program 比較沒有做到 computing 和其他部分的 overlap，所以不能將時間吸收掉，若能試著在同樣的時間內做更多不同的事情，也許能夠增加 scalability。

不過從 ii(c)可以看到雖然 process 總量相同，將他們平均分配到不同 node（機器）上能夠稍稍提高 performance。

III. Conclusion

像老師、助教所說的，這次的作業要完成沒有到很困難，主要是要讓我們對沒使用過的 MPI 熟悉、上手，一開始覺得最困難的是有 bug 但是資訊沒有很多，找不太到問題出在哪邊，好難 debug，還有還沒有那麼習慣平行程式的觀點。雖然作業本身不難，不過如果要讓 performance 有較顯著的提升，就不是那麼容易了，除了計算本身之外，目前還有點不知該從何處下手的感覺。透過第一次的作業，我覺得對 parallel programming 實作更熟悉了一些，以前只是在講義、課本中的知識、理論，在自己實際做了之後才更能體會「平行」帶來的優化和要關注的點。

IV. Reference

i. https://www.open-mpi.org/doc/v4.0/

ii. https://stackoverflow.com/questions/13774968/mpi-kill-unwanted-processes

V. 實驗數據連結

https://drive.google.com/file/d/1FbAmsrxmqQvZs1uWnqIkZCqbc3oU93FW/view?usp=sharing