Parallel Programming HW1

114062640 謝秉宸

Implementation

Load balancing

為了load balancing，先計算data總數除以process數量，剩下的餘數r平均分配給前r個process，如此一來，data數量分配平衡，且可以直接用process rank來判斷data數。當遇到如testcase 3 process數量大於data數的情況時，也可以順利分配。

MPI_File_IO

使用MPI_File_read_at和MPI_File_write_at來進行資料的input、output。並且因為先前已經有計算好各process的data數量，所以可以確保各process不會重複讀入或有資料沒讀取到的情況發生。

```
int array_size=atoi(argv[1]);
    int process_size=array_size/size;
    int remain=array_size%size;
    int local_size=(rank<remain)?(process_size+1):process_size;

    MPI_File input_file, output_file;
    float *data = new float[process_size+1];
    MPI_File_open(MPI_COMM_WORLD, input_filename,
    MPI_MODE_RDONLY, MPI_INFO_NULL, &input_file);

    if(rank<remain)
        MPI_File_read_at(input_file, sizeof(float) * rank*(process_size+1), data,
        local_size, MPI_FLOAT, MPI_STATUS_IGNORE);
    else{
        int previous=(process_size+1)*remain;
        MPI_File_read_at(input_file, sizeof(float) *
        (previous+(rank-remain)*process_size), data, local_size, MPI_FLOAT,
         MPI_STATUS_IGNORE);
    }
```

```
MPI_File_close(&input_file);
```

Local sort

將data平均分給各processes後，各process先各自對現有的資料進行排序，我一開始使用標準函式庫的sort，但之後與同學討論發現，有另一個C++內建的boost::sort::spreadsort::spreadsort()，執行排序的時間比原先的sort快上不少。故最終選擇spreadsort作為我的local sort。

odd-even sort

若有n 個process，這一階段最多要進行n+1次兩兩互換，在even phase時，偶數process m會和process m+1互換資料，在even phase時，偶數process m會和process m+1互換資料，配對的process下稱partner process。我的實作是先檢查是否需要交換，再互相將自己的資料傳給partner process，各自進行merge後，rank小的process取得較小的那一半data，rank大的process則取得較大的那一半data

Optimization

1.原先merged_data這個array的宣告在for迴圈裏面，因為這個array其實是可以重複使用的，所以我將它移出for迴圈外，大幅下降memory allocation的時間。

2.原先我使用C++內建的std::merge，假設每個process有n個或n+1個data，那這樣std::merge就需要花費2n單位的時間，但若自己重新寫一個merge，可將時間降為n個單位。並在完成merge後，直接用swap交換記憶體位置，這樣也省下不少時間。

3.每次在資料交換前，都會先檢查是否需要交換，檢查方法為若rank低的最大元素小於等於rank大的最小元素，則不交換。值得一提的是，我原先判斷是小於，當我改為小於等於之後，judge總執行時間大幅下降了10秒。

額外嘗試

1.原先關於兩個processes之間資料交換的部分有另外一個想法，有沒有可能先找出兩個processes的中位數，再將rank小的process中大於中位數的數字傳送給rank較大的process，並將rank較大的process中小於中位數的數字傳送給rank較小的process，從兩個sorted array中找出中位數是leetcode中的一個經典hard題，這可以在O(log(min(m.n)))時間內完成，並可以大幅減少要傳送的總資料量。然而實際時做出來之後發現可能是因為每次建立連線都有MPI latency的原因，這樣為了找出中位數需要log(n)次的MPI latency，可能就是這個原因導致結果不合預期，故最後仍選擇使用傳送所有data給partner process的做法。

2.有時候不一定要跑n+1輪就排序完成了，所以我有試著加入檢查如果有連續的odd phase和even phase沒有交換就直接停止，但實際實作發現可能是每次MPI_Allreduce都要等到所有程式都同步才能執行，導致效率反而沒有原先那麼好。

Experiment& Analysis

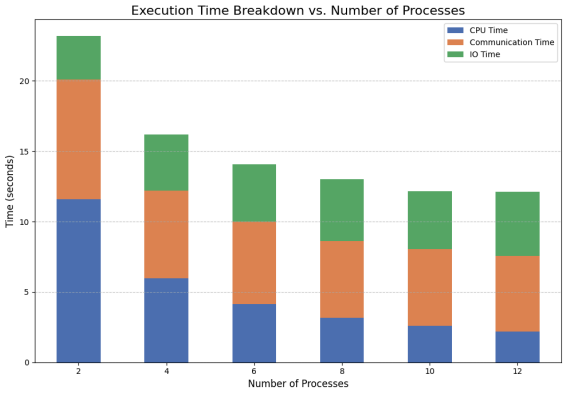
Methodology

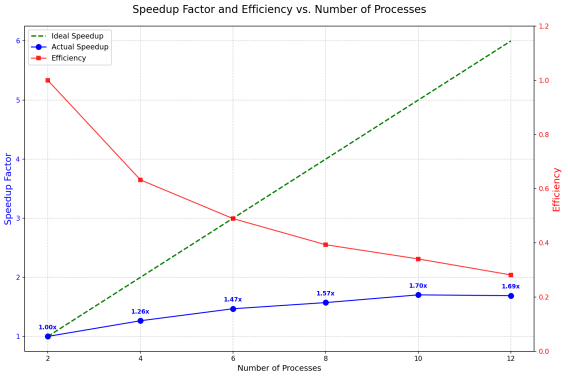
我使用課堂提供的cluster。並用MPI_Wtime()來記錄各階段的執行時間，我將程式執行階段分為io_time、comm_time、CPU_time，並記錄程式總執行時間，總時間減去io_time、comm_time即為CPU_time，最後將各process的time做加總。

Testcase

我選擇36.in作為我的test data，該筆資料大小為536869888，我認為這樣的資料量比較能看出平行化的優勢。又不會太大導致要執行很久。

Strong scalability

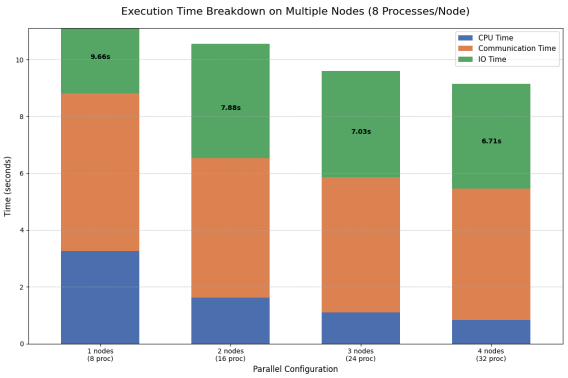




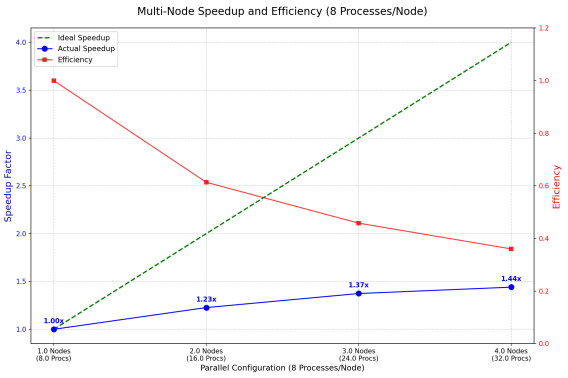
左圖是從 1 個 Process 到 12 個 Process 的各階段時間佔比圖。

右圖則為從 1 個 Process 到 12 個 Process 的speedup factor折線圖

從右圖可見，即使提升process數量也沒有依比例增加速度，所以strong scability效果並不好。會有這樣的原因我覺的是因為 Odd Even Sort 在改成 Parallel 時，隨著process增加，雖然每輪要通訊的資料量會降低，但同時也會執行更多輪，故可以看到左圖communication time佔比基本上變化不大，與之對應的是CPU time隨著process增加而顯著下降。



1 個 node 到 4個nodes (每個node8個processes)的各階段時間佔比圖



1 個 node 到 4個nodes (每個node8個processes)的speedup factor折線圖

由右圖可見加速最多只來到1.44倍，原因可從左圖看出，雖然CPU time有顯著地隨著時間下降，但IO time和 communication time變化不大，故隨著node增加，提升的速度沒那麼顯著。因此若想追求更高的效率，可能要從communication time著手，雖然我有想過其他減少communication time的做法，但結果不盡人意，以下為我嘗試過關於減少communication time的實驗。

MPI communication latency與單筆資料的傳送時間

動機:驗證上面先找出中位數再分配的方式為不可行。

使用以下程式來估算建立連線MPI連線的latency和傳送一筆資料的時間，發現MPI_latency事實上會隨傳送的資料大小而上升，這似乎有點違反直覺。但最終估算結果為latency time約為1~1000μs，傳送一筆資料則約為0.3ns

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size = 1; //size=1, 10000000, 50000000
    float *buf = (float*)malloc(size);
    MPI_Barrier(MPI_COMM_WORLD);

    double start = MPI_Wtime();
```

```
    if (rank == 0) MPI_Send(buf, size, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    if (rank == 1) MPI_Recv(buf, size, MPI_CHAR, 0, 0,
     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    double end = MPI_Wtime();

    if (rank == 0) printf("Latency: %f microseconds\n", (end - start) * 1e6);
    MPI_Finalize();
}
```

若使用先找出中位數再分配的方式，需先經過logn次的send(1)和recv(1)，之後再進行一次約n/2的資料量的訊息交換。若使用36.in的測資，並假設有24個process，每個process會被分配到約23000000筆資料，若要找出中位數需約log23000000=24次訊息交換，再加上後面的大量訊息交換，導致結果不佳，以此佐證此方法不可行。

Conclusion

做這次作業的過程可說是雲霄飛車，當我完成第一版程式時，時間約需要200出頭秒，但在花一些時間在優化 C++ 的架構後，時間大幅加快了許多，但與scoreboard上的前幾名仍有很大差距，我不禁好奇他們是如何優化他們的程式碼的，隨後我想到全新的交換資料的方法，雖然在實際實作後效果不合預期，但這個從思考到實作在到驗證的過程真的還蠻令人興奮的。另外，有聽說有修課的同學實作出radix sort來提升local sort的時間，也許與前段的差距就在這邊，又或許有更好的資料交換以減少communication time的方式，這應該才是真正拉開差距的地方。