Parallel Programming HW1

112164513 陳彥凱

Implementation

分為以下步驟

- Load Balancing

  儘量平均分配相同大小的data給每個process，我發現在向上取整的時候，分子分母都必須為double否則會有問題；再來記錄 **generalProcessCount** 以供最後一個process跟鄰居溝通使用，最後一個process的data數量就是總數N扣掉已分配好的；**offset** 是每個process所負責的data開始的位置。藉由以上來完成接收任意大小的input與process數量設定。

```
// scatter the data to each process
dataPerProcess  = (int)(ceil((double)N/(double)numOfProcess));
generalProcessCount = dataPerProcess;
lastProcessCount = N- (dataPerProcess * (numOfProcess-1));
offset   = rankID * dataPerProcess * sizeof(float);
if(rankID  numOfProcess -1) dataPerProcess = lastProcessCount;
```

- MPI IO

  讀取與寫入資料分別使用 **MPI_File_read_at** 和 **MPI_FILE_write_at**，根據 **offset** 位置平行的讀取與寫入。

```
// file read
MPI_File fileInput, fileOutput;
MPI_File_open(customWorld, argv[2], MPI_MODE_RDONLY, MPI_INFO_NULL, &fileInput);
MPI_File_read_at(fileInput, offset, localBuffer, dataPerProcess, MPI_FLOAT,  MPI_STATUS_IGNORE);
MPI_File_close(&fileInput);
//file write
MPI_File_open(customWorld, argv[3], MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fileOutput);
MPI_File_write_at(fileOutput, offset, localBuffer, dataPerProcess, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&fileOutput);
```

- Local Sort

  接著將每個process負責的data都先sort好，再以process為單位做odd-even sort，跟以每個element為單位相比，能夠減少sort的次數與時間。

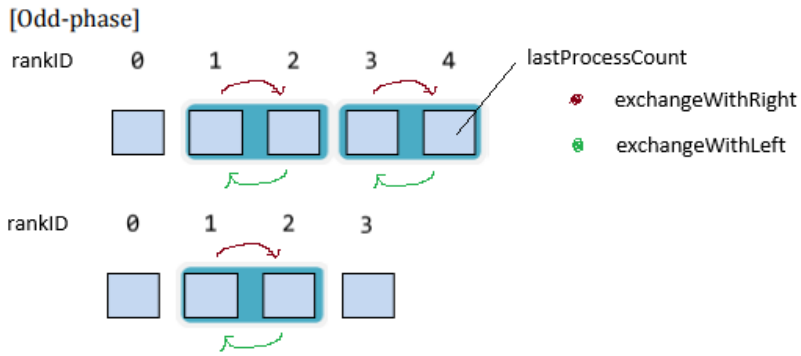
  process內的排序方法我選擇c++ boost library提供的spreadsort，它採用混合式的策略，在特定資料量以下為comaprison-based，反之為radix-based，經測試在資料量大時兩者的差距會>20%。

```
// in-process sort
boost::sort::spreadsort::spreadsort(localBuffer, localBuffer + dataPerProcess);
```

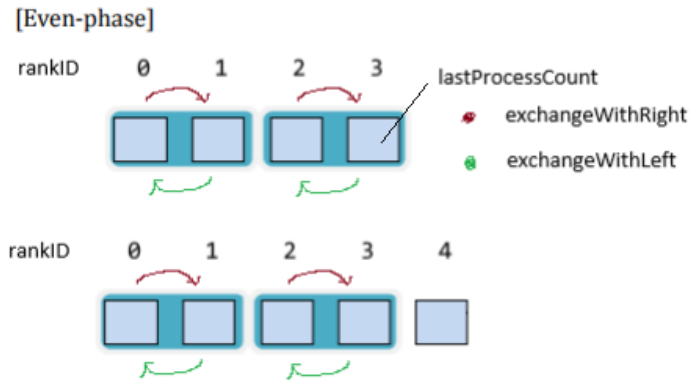
- Between Process Odd-Even Sort

  再來是process level odd-even sort，終止條件為process數量，將所有process檢驗過以避免意外結果。

  這邊要考慮以下情況：



在odd-phase時，如果process的rankID是奇數並且又是倒數第二個，則當它跟右邊的最後一個process溝通時，接收的數量需為 **lastProcessCount** ，其他的奇數process送跟收的數量都是 **generalProcessCount** ，但如果process總數為偶數時，最後一個奇數process就不能再跟右邊溝通了（因為它是最後一個）；而當process的rankID是偶數並且是最後一個時，此process跟左邊溝通時送的數量需要為 **lastProcessCount** ，其他的偶數process送跟收的數量都是 **generalProcessCount**，但第0個process就不能再跟左邊溝通。



even-phase的情況類似，就是上面的情況反過來。如果process的rankID是奇數並且又是最後一個，則當它跟左邊的process溝通時，送的數量需為 **lastProcessCount** ，其他的奇數process送跟收的數量都是 **generalProcessCount** ；而當process的rankID是偶數並且是倒數第二個時，此process跟右邊溝通時接收的數量需要為 **lastProcessCount** ，其他的偶數process送跟收的數量都是 **generalProcessCount**，但最後一個偶數process就不能再跟右邊溝通。

```
// between process odd-even sort
for (int i = 0; i < numOfProcess ; i++) {
    //odd-phase
    if (isOdd(i)) {
        if (isOdd(rankID)) {
            if (rankID  numOfProcess - 2)
                exchangeWithright(localBuffer, rankID, generalProcesscount, lastProcessCount);
            else if (rankID != numOfProcess - 1)
                exchangeWithright(localBuffer, rankID, generalProcesscount, generalProcesscount);
        }

        if (isEven(rankID)) {
            if (rankID  numOfProcess - 1)
                exchangeWithleft(localBuffer, rankID, lastProcessCount, generalProcesscount);
            else if (rankID != 0)
                exchangeWithleft(localBuffer, rankID, generalProcesscount, generalProcesscount);
        }
    }
```

```
    //even-phase
    if (isEven(i)) {
        if (isOdd(rankID)) {
            if (rankID  numOfProcess - 1)
                exchangeWithleft(localBuffer, rankID, lastProcessCount, generalProcesscount);
            else
                exchangeWithleft(localBuffer, rankID, generalProcesscount, generalProcesscount);
        }

        if (isEven(rankID)) {
            if (rankID  numOfProcess - 2)
                exchangeWithright(localBuffer, rankID, generalProcesscount, lastProcessCount);
            else if (rankID != numOfProcess - 1)
                exchangeWithright(localBuffer, rankID, generalProcesscount, generalProcesscount);
        }
    }
}
```

再來是跟左右溝通的部分，想法是左邊的process會把較小的半邊留下，右邊的process會把較大的半邊留下。以左邊的process執行 **exchangeWithRight** 為例：首先使用 **MPI_Sendrecv** 將自己的data傳給鄰居並接收鄰居的data，接著找出較小的那一半data並將 **localBuffer** 更新（只做到自己原本的data數量即停止）。**exchangeWithLeft** 就是倒過來，從後面大的開始拿，一樣拿到自己原本的data數量即停止。

```
void exchangeWithright(float* localBuffer, int rankID, int sendCount, int recvCount) {

    MPI_Sendrecv(localBuffer, sendCount, MPI_FLOAT, rankID + 1, 1,
        receiveBuffer, recvCount, MPI_FLOAT, rankID + 1, 0, customWorld, MPI_STATUS_IGNORE);

    //preserve samller half
    int x = 0, y = 0;
    for (int i = 0; i < sendCount; i++) {
        if (yrecvCount || (y < recvCount && x < sendCount && localBuffer[x] <= receiveBuffer[y]))
            temp[i] = localBuffer[x++];
        else
            temp[i] = receiveBuffer[y++];
    }
    for (int i = 0; i<sendCount; i++)
        localBuffer[i] = temp[i];
}

void exchangeWithleft(float* localBuffer, int rankID, int sendCount, int recvCount) {

    MPI_Sendrecv(localBuffer, sendCount, MPI_FLOAT, rankID - 1, 0,
        receiveBuffer, recvCount, MPI_FLOAT, rankID - 1, 1, customWorld, MPI_STATUS_IGNORE);

    //preserve larger half
    int x = sendCount - 1, y = recvCount - 1;
    for (int i = sendCount - 1; i >= 0; i--) {
        if (y<0 || (y>=0 && x>=0 && localBuffer[x] >= receiveBuffer[y]))
            temp[i] = localBuffer[x--];
        else
            temp[i] = receiveBuffer[y--];
    }
    for (int i = 0; i<sendCount; i++)
        localBuffer[i] = temp[i];
}
```

- Optimization
  - 以process為單位做odd-even sort而非以element為單位。
  - 用spreadsort替代std::sort。
  - 減少迴圈內memory allocation次數，一開始就分配好所需的array（**temp**, **localBuffer**, **receiveBuffer**）。

Experiment and Analysis

- Methodology

  使用課程提供的apollo cluster做實驗，singlenode跑在test partition，multinode跑在judge partition上，以第40筆testcase作為測資，資料量為536869888。

  - CPU time

    在 **MPI_Init** 和 **MPI_Finalize** 之間加上 **MPI_Wtime** 記錄整支程式執行的時間，再扣掉Comm time和IO time即為computing time。

  - Comm time

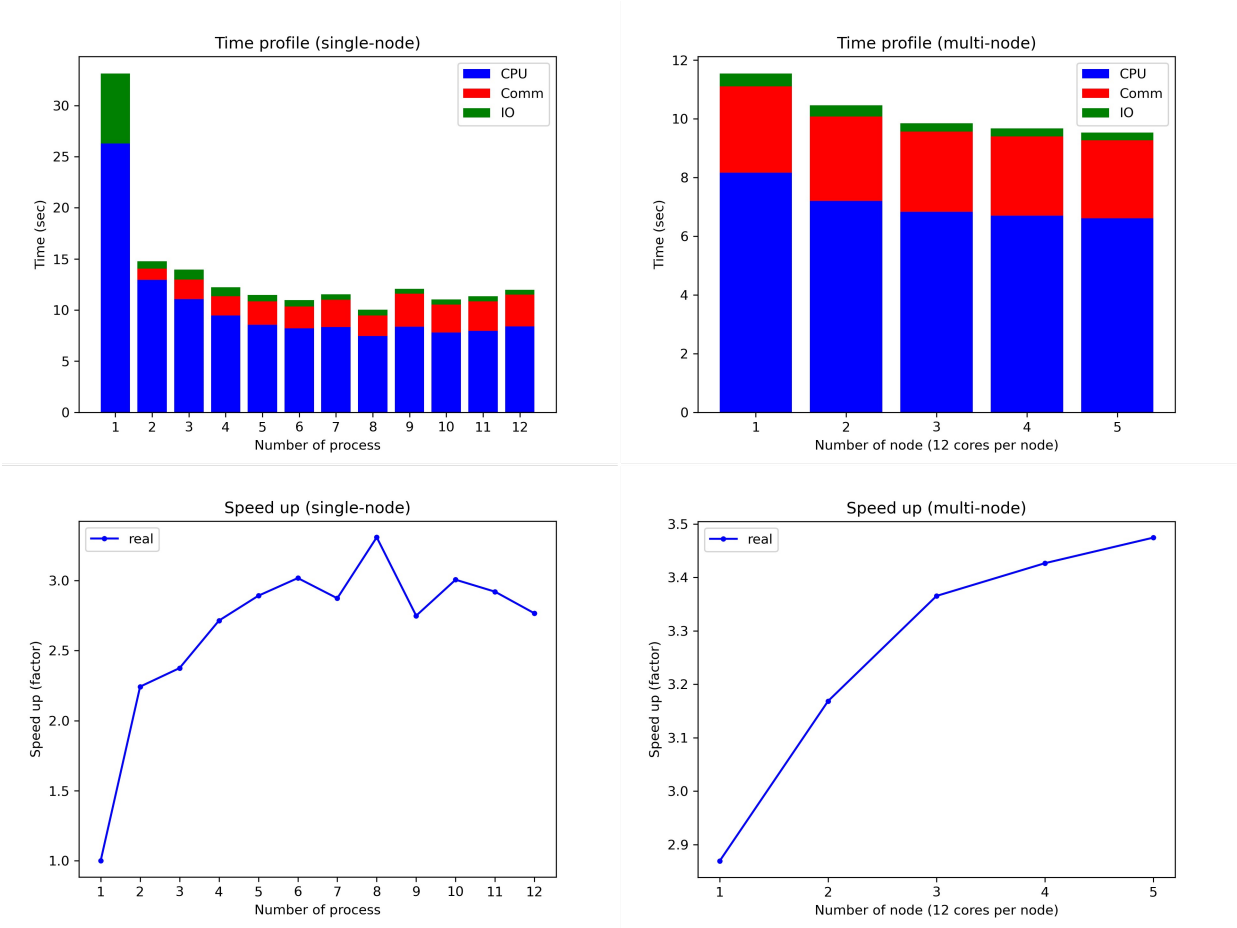
    在每個 **MPI_Sendrecv** 的前後加上 **MPI_Wtime** 記錄時間，兩者相減的加總即為communication time。

  - IO time

    在 **MPI_File_open** 前面和 **MPI_File_close** 後面加上 **MPI_Wtime** 記錄時間，兩者相減的加總即為IO time。

  - 將每個process的三種時間加起來取平均，以得到該筆實驗的三種時間。

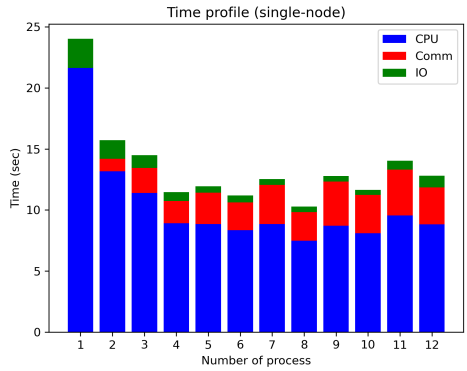
- Plots: Time Profile and Speed up



- Discussion

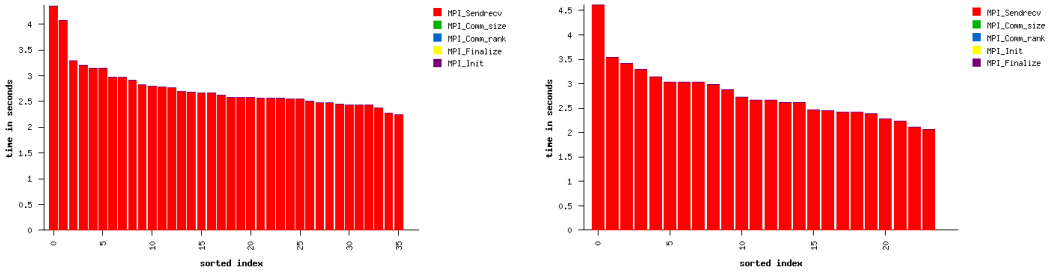
  - Time Profile

    在single node的部分，隨著process增加，總時間和CPU time皆呈現下降趨勢，當然因為process數增加Communication time也隨之增加，IO的部分除了單一個process時高了一點，其他則沒有太大的差異，值得一提的是做了好幾次實驗，每次的IO time都不太一樣，推測可能跟當下的使用者數量有關，造成IO time不穩定；這裡附上另一次實驗數據，可以看到IO time是很不均勻的。



上圖為single node另一次的實驗，從兩次實驗都可以看到，在8個process之後，總時間就呈現上升趨勢，這是因為可平行化的部分越來越小，逐漸被序列化的部分支配，因此再增加process也沒辦法讓運算時間下降，並且因為溝通時間而讓總時間增加，bottleneck為Communication time和IO time。

在multi-node的部分，隨著Node增加，總時間及CPU time呈現下降趨勢，但也同樣是因為可平行化的部分逐漸變小，故CPU time下降趨勢變得越來越緩；而IO time和Communication time並沒有太大的差異，不論如何增加node數都不會繼續下降，故此時的bottleneck也是Communication time和IO time。



由IPM profiler觀察第40筆與第39筆測資的執行狀況，各個process的load還算是平均，第0個process因為任務調度、通訊和同步、IO...等等，所以loading自然會是比較高的。

- Speedup Factor

  在single node的部分，加速比最大到了約3.3倍，Multi node則到了約3.45倍，可見平行化是有益的，但都離理想的線性加速很遠，應該還有辦法改寫使得scability程度提升。從IPM profiler的結果可以觀察到，communication time確實還是有能再改善的空間。

- 進一步的優化方向

  IO的優化可以藉由真正的平行file system來實現，上課時老師有提到目前仍然是一般non parallel的file system；而Communication的優化也許可以透過使用不同的通訊協定來實現。

Conclusion

在這次的作業裡，我熟悉了MPI的各種API，以及profiler工具，並且因為是自己一步步完成的，所以對整支程式平行化的流程與原理有了更深的理解。

我覺得最耗心力的是最後在做效能評估的部分，除了因為是第一次使用這些工具之外，還要確認實驗做出的結果是否合理，系統不穩定時給出的數據都會差異很大；在作圖的部分因為IPM給出的部分圖表怪怪的，MPIP的資料又都是純文字的，所以還是回過頭用熟悉的python matplotlib來處理，再用shell script一次寫好要做的各項實驗自動化執行，這次學到的經驗對以後不管是課堂上或實驗室的計畫要做實驗，都會是很大的幫助。

另外在優化程式的部分也讓我十分苦惱，能想到可改善的點修完之後，在score board上還是只能排在中後段的位置，平常自己在做實驗室計畫的時候通常都只要求動得起來就好，也藉此提醒我應該要改一下這個不好的習慣，在後續的課程學習更多相關知識來精進自己。