Homework 1: Odd-Even Sort

108062313 黃允暘

Implementation

1. **Data distribution**:

   在本此作業中，我將每個 Node 視 odd even sort 的交換單位。因此要先將資料分配到每個 Node 當中。而為了使每個 Node 分配到的 data 數量相同，我的作法如下：假設有 n 筆資料、Node 的數量為 size，每個 Node 會先分配到 n/size 筆資料。考慮到會有 n/size 會有餘數的問題，假設餘數是 r，我會是讓前 r 個 Node 多分配到一筆 data。

   算出每個 Node 要取多少資料後，就可以計算出要讀取的資料位置(offset)，並透過 MPI_File_read_at ()，讀取資料。而為了等一下 Node 之間的資料傳輸，在這邊也會一並計算左右兩旁的 Node 大小。

```
int offset=0;
int remaining=n%size;
int local_size=n/size;
for(int i=0; i<rank; i++){
    if(i<remaining) offset+=(local_size+1);
    else offset+=local_size;
}
offset = offset*sizeof(float);
```

```
int right_size, left_size;
if(rank+1<remaining) right_size=local_size+1;
else right_size=local_size;
if(rank-1<remaining) left_size=local_size+1;
else left_size=local_size;
if(rank<remaining) local_size+=1;
```

2. **Sorting**:

   根據 odd even sort 的做法，Node 之間的資料交換會有兩個 phase:

   Odd phase: index 為奇數的 Node，會與右邊的 Node 進行資料換。

   Even phase: index 為偶數的 Node，會與右邊的 Node 進行資料換。

   在 sorting 實作的部分，我有使用到兩種 sorting 的辦法，分別應用在 Node 內部、Node 之間。

   (1) Node 內部：在與別的 Node 進行 odd even sort 之前，每個 Node 會先對內部的資料進行 sorting。而我在這邊用的方法是 spreadsort。

   (2) Node 之間：Node 之間資料的交換效率，會大大影響程式的 performance。我在這邊採用的是類似 merge sort 的方式。兩個 Node 會互相傳輸自己的資料給對方，而傳輸的方式是採用 MPI_Sendrecv()。而因為資料在 Node 內部就已經排序好了，所以在合併兩個排序好的陣列時，我們只需要不斷比較兩邊的第一個元素，把比較小的丟到新的大陣列，就能完成排序。

   只要不斷執行 odd phase, even phase，就可以完成全部資料的 sorting。另外，假設有 K 個 Node，最多只要交換 k+1 次就可以完成 sorting。

3. **Optimization**:

(1) Merge sort:

其實我最一開始的做法不是 merge sort，而是將左邊 Node 的資料傳給右邊的 Node，直接要右邊的 Node 在內部執行 quicksort。這樣的作法完全浪費了一個 Node 的計算資源。用 merge sort 的話，就可以利用到雙方的計算資源。

另外，在 merge sort 的過程，我還有做調整，使 sorting 速度加快。如同之前所提到的，兩邊的 Node 都會將資料傳給對方，但兩邊的 Node 都不用對全部的資料進行 sorting，只需要找滿自己 size 的資料數量就好。左邊的 Node 之需要找滿前半部的資料就好，而右邊的 Node 只需要找滿後半部的資料就好。

```
int neighbor_id=0, self_id=0;
for(int i=0; i<local_size; i++){ //找左半邊的data
    if(self_id<local_size && neighbor_id<neighbor_size){
        if((*data)[self_id]<neighbor_data[neighbor_id]){
            new_data[i]=(*data)[self_id];
            self_id++;
        }
        else{
            new_data[i]=neighbor_data[neighbor_id];
            neighbor_id++;
        }
    }
    else if (self_id<local_size){
        new_data[i]=(*data)[self_id];
        self_id++;
    }
    else{
        new_data[i]=neighbor_data[neighbor_id];
        neighbor_id++;
    }
}
```

(2) Early stop:

根據 odd even sort 的演算法，只要在一個 phase 中，都沒有發生過 sorting，就可以結束 odd even sort。因為當兩的 Node 進行資料交會時，個別的資料都是已經經過 sorting，因此只要「左邊 Node 的最後一筆資料」大於「右邊 Node 的第一筆資料」，這兩個 Node 就不用進行 sorting。

根據此特性，我的 sorting function 會的回傳值代表該次 Node 的交流，是否有發生 sorting，1 代表有，0 代表沒有。而我在每個 phase 結束的時候，都會利用 MPI_Allreduce()，去檢查倆倆 Node 是否有發生 sorting。若都沒發生的話，就會直接跳離迴圈，結束 odd even sort。

(3) 善用指標：

在執行 merge sort 的時候，有些步驟會需要將一個陣列指定給另一個陣列，如果使用 for 迴圈，複製陣列的內容，會花非常多的時間。因此用指標代替陣列的複製，會節省蠻多時間的。

Experiment & Analysis

1. Methodology

(a) System spec: 學校的系統環境

- 1 login node (apollo31) (200%CPU max)
- 18 compute nodes (1200% CPU max)
- Use `squeue` to view SLURM usage
- Cluster monitor: http://apollo.cs.nthu.edu.tw/monitor
- 48GB disk space per user

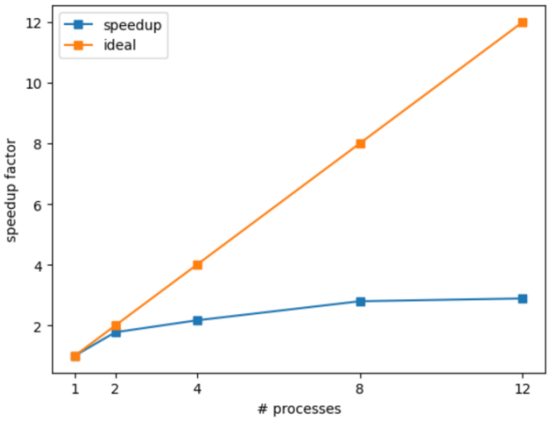
(b) Performance Metrics:

我利用 MPI_Wtime()去計算程式執行的時間，計算的方式如下圖。

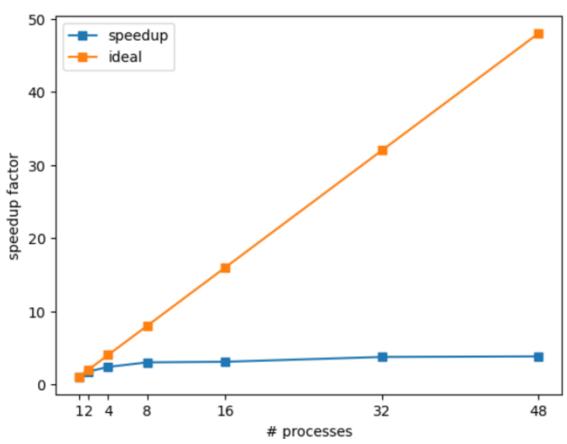
```
double starttime, endtime;
starttime = MPI_Wtime();
....  stuff to be timed  ...
endtime   = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```

2. Speedup Factor & Time Profile

(a) Speedup Factor



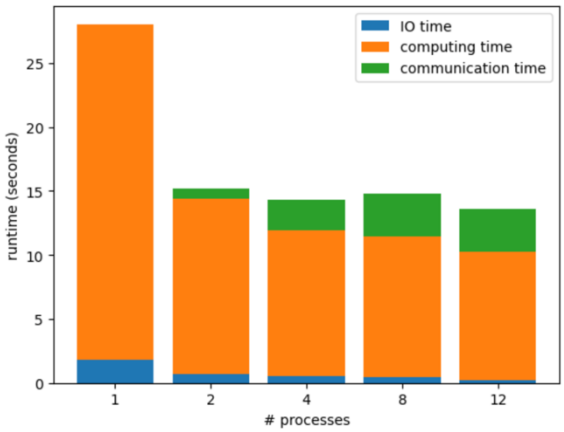
(i)1 Node



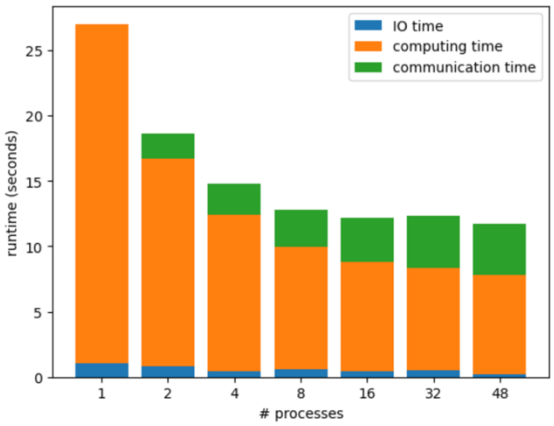
(ii)4 Nodes (12 processes/Node)

從圖表上可以看到，當 processes 的數量增多的時候，speedup factor 確實有上升。但隨著 processes 的增加，speedup 的程度離 ideal 越來越遠了。這是因為雖然 processes 的數量增加了，但 communication 的 tradeoff 也增加了，這一點可以從下一個實驗看出來。所以可以看到，當 processes 的數量從 1 增加到 8 時，speedup 有明顯的上升; 但當 processes 從 32 升到 48 時，speedup 卻幾乎沒有上升，這就代表 communication time 也上升的非常多。

(b) Time Profile



(i) 1 Node



(ii)4 Nodes (12 processes/Node)

可以看到 computing time 隨著 processes 的增加，有了明顯的下降，這是因為有多個 processes 可以同時處理 data。而隨著 processes 繼續增加，communication time 也有明顯的增加，因為隨著 processes 的增加，需要溝通的次數也增加了。這個現象也符合上一個 speedup factor 實驗的結論。

3. Discussion

透過實驗，可以發現雖然多個 processes 可以有效幫助 computing time 下降，但也會使 communication time 上升。因此 processes 並不是越多越好。另外，透過圖表也可以發現：computing time 並不會隨著 processes 的數量上升，而成比例下降。我覺得這是因為程式的 scalability 仍有上升的空間，目前的程式沒有辦法完全利用多個 processes 的優勢，這是未來可以改進的地方。

Experiences / Conclusion

這次的實作使我更了解 computing time、IO time 還有 communication time 的關係，也更了解 MPI 的操作。整體來說，實作 odd even sort 並不困難，困難的事之後的優化，從 sorting 的設計、for loop 的設計到變數的宣告等等，有很多要注意的地方。另外，cluster 的使用人數也會大大的影響 execution time，同一份 code，在人多或人少的時候跑，會有非常大的差異。