Homework 1: Odd-Even Sort Report

107062120 姚潔恩

Implementation

I. Handle an arbitrary number of input items and processes

為了可以用任意 numtasks (process 數量)處理任意大小的 n (data 數量)，需要考慮兩個情況。第一是 n < numtasks，這時我會使用 MPI_Comm_split 將 rank 大於等於 n 的 process 移出原本的 group 並呼叫 MPI_Finalize。這時就只剩下 rank 0~n-1 的 process，因此就可以順利完成 sorting。第二個情況就是 n >= numtasks。這時我們就必須處理 load balance 的問題。我採用的方法是先將 n / numtasks (整數除法)的量分到每一個 process，接著再取 n / numtasks 的餘數，將餘數從 rank 0 平均分配。也就是說 rank 小於餘數的 process 會多分到一個 data，負責 n / numtasks + 1 筆資料，而 rank 大於等於餘數的 process 就只會負責 n / numtasks 筆資料。

II. IO

為了 IO 的平行化，每個 process 在讀寫時都只會讀寫自己負責的區段。根據上面的規則可以算出每個 process 要讀寫的 offset，因此每個 process 就可以從各自的 offset 讀取自己負責的資料量，達到 IO 的平行。

III. Sorting

在開始 Odd-Even Sort 之前會先在 local 把自己的資料 sort 一次。我使用的方法是 8-bit 的 radix sort。原因是對 32-bit 的資料來說 8-bit radix sort 的複雜度為 O(4 * (N + 256))，其他 sorting 演算法大多是 O(NlogN)，因此在資料量很大的時候 radix sort 會比較有優勢。麻煩的點是 floating point 資料的 binary 形式不能直接使用 radix sort (因為 floating point 負數 sign bit 是 1)，因此需要將 floating point 做轉換。轉換的規則是如果 sign bit 是 1，那就 flip 所有 bit，反之就只 flip sign bit。flip sign bit 是為了讓正數能夠大於負數，至於如果 sign bit 為 1 那後面所有 bit 都要 flip 是因為負的數值越大在排序上應該是要越小，所以就再用 flip 把它變小。

IV. Odd-Even Sort

在 Odd-Even Sort 中會由很多輪的 rank n 和 rank n+1 的 process 互相傳遞 data 並排序，在 even phase 時 n 就是偶數，反之在 odd phase 時 n 就是奇數，我的實作是先 even phase 在 odd phase。溝通的方式我使用 MPI_Sendrecv，讓雙方同時把自己的 local data array 傳給對方。由於在進入 Odd-Even Sort 之前有先 sort 過 local data，所以可以用類似 merge sort 的方法來 merge 雙方的 array。假設 rank n 所負責的資料量是 k 那 rank n 就會從 rank n 和 rank n+1 的 array 中挑出前 k 小的數

字。相反的 rank n+1 就是挑出數字比較大的幾個。這個 merge 的步驟可以在 O(n) 的複雜度下完成。另外在 merge 之前我會先檢查 rank n 的最大的數字是不是大於 rank n+1 最小的數字，如果成立才有 merge 的需要。最後就是關於 sorting 結束的判定，我會在 odd phase 檢查 sorting 有沒有完成。方法是在每個 process 用一個 Boolean flag 表示有沒有排序交換的發生，在要進行下一次 even phase 之前用 MPI_Allreduce 將所有 flag 作 OR，檢查是不是所有 process 都沒有交換發生，如果都沒有交換發生就可以結束 Odd-Even Sort 了。

V. Optimization

A. Sorting 演算法優化：

我一開始是使用 std::sort，複雜度約為 O(NlogN)。換成 radix sort 後在資料量很大的情況下就會有一定程度的提升。

B. Odd-Even Sort 演算法優化：

我的第一個版本 Odd-Even Sort 是 rank n+1 將資料傳給 rank n 的 process，然後在 rank n 完成 merge 以後再將 rank n+1 的部分回傳給 rank n+1。這樣個缺點是 rank n+1 就會處於 idle 而且要有兩次溝通。使用 MPI_Sendrecv 只需要一次溝通而且兩邊各自 merge 自己需要的部分，不會造成 idle 浪費。

C. 其他優化：

減少使用 if/else 以及避免 variable declare。例如轉換 float 的 function 我就寫了三個版本，分別是從 if/else (下圖一)改成沒有 if/else 可是有使用一個 variable mask (下圖二)。接著才把 variable mask 拿掉，變成下圖三的樣子。

```
29  inline unsigned int FloatToRadix(float value)
30  {
31      unsigned int radix = *(unsigned int*)&value;
32      if ((radix >> 31) == 1) {   // if negative float, flips all bits
33          radix ^= 0xFFFFFFFF;
34      }
35      else{   // if positive float, flips the sign only
36          radix ^= 0x80000000;
37      }
38      return radix;
39  }
```

```
29  inline unsigned int FloatToRadix(float value)
30  {
31      unsigned int radix = *(unsigned int*)&value;
32      unsigned int mask = -(radix >> 31) | 0x80000000;
33      radix ^= mask;
34      return radix;
35  }
```

```
15  inline unsigned int FloatToRadix(float value)
16  {
17      unsigned int radix = *(unsigned int*)&value;
18      radix ^= -(radix >> 31) | 0x80000000;
19      return radix;
20  }
```

Experiment & Analysis

I. Methodology

A. System Spec

實驗環境使用課程提供的 server。實驗的測資使用第四十筆 testcase，資料數量為 536869888，足夠讓平行程式展現優勢。

B. Performance Metrics

實驗的計時方式是使用 MPI_Wtime()。在跑實驗的過程中我會紀錄三個時間，分別是 Total time、IO time 以及 Communication time。最後再將 Total time 減去 IO time 以及 Communication time 即可得到 CPU time。各項時間的定義如下：

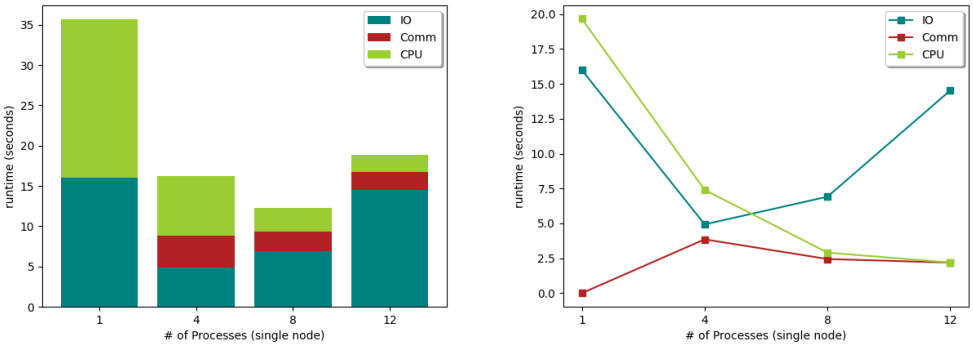
- IO：讀檔及寫檔的區段，也就是有使用 MPI IO routines 的地方。
- Communication：有使用 MPI 的資料 communication routines 的區塊。在我的 implement 中就是呼叫 MPI_Sendrecv 和 MPI_Allreduce 的時間會被計入。
- Total：從 MPI_Init 以後直到完成寫檔之後。CPU time 就是 Total time 減掉 IO time 以及 Communication time。這樣的做法雖然會讓 CPU time 增加一點額外的時間(計算時間所花的時間)，但對實驗結果的影響不大。

因為每個 process 都會各自計算時間，因此我 report 的各項時間就是全部 process 的平均值。最後為了取得更穩定的實驗數據，呈現於圖表上的時間都是跑三次實驗的平均。

II. Plots & Discussion

A. Time Profile

1. Single node



從上面的實驗結果可以觀察到幾個現象，首先 CPU time 隨著 process 數增加而減少。這部分就是平行程式發揮的地方，因為隨著 process 數目增加每個 process 所需處理的資料量就成反比的減少，因此可以減少 CPU time。

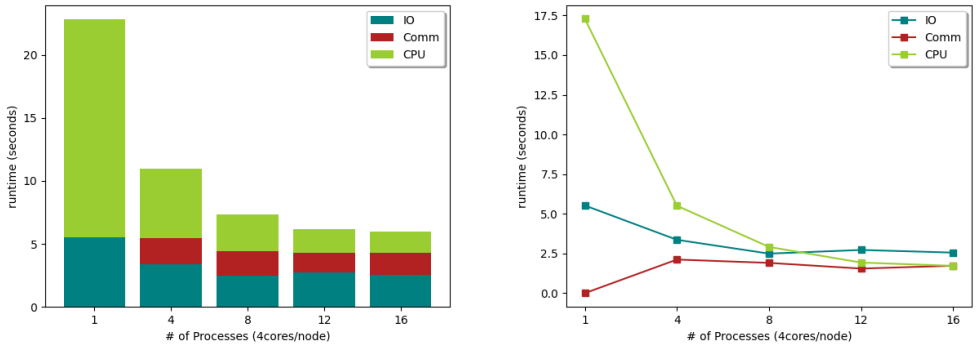
第二是 Communicate time 一開始會隨著 processes 數增加而成長。原因可能是當

process 越多，溝通時需要同步化的 overhead 就越高，例如在決定終止條件的 Allreduce 需要等所有 process 工作結束，就可能浪費大量時間。然而隨著 process 數繼續加大 Communicate time 就又減少了，推測可能的原因是因為要交換的 data 量變少，所以使得 Sendrecv 的時間減少。

第三是 IO time 一開始會隨著 process 數量變多而遞減，原因是 process 可以平行的做 IO 所以使得 IO time 減少。然而隨著 process 數變多 IO time 又會慢慢上升。推測是因為 IO port 有限，因此 process 上升到一定的數量對於 IO 的速度就沒有提升效果了，同時若有很多 process 競爭 IO 資源可能還會讓 IO 變慢。

2. Multiple node

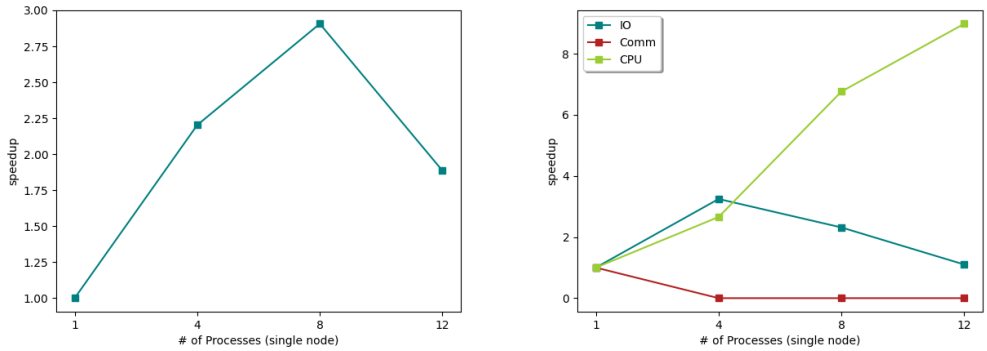
註：Multiple node 實驗我是使用--mincpus 在控制一個 node 上有幾的 core。例如如果設定 4cores/node，那在跑 8 個 process 的實驗時就會下-n8 -N2 --mincpus=4。



和 Single node 的實驗比較不同的地方是 IO time 並沒有因為 process 繼續增加而有明顯的反彈。推測是因為 process 平均分散在不同的 node，可能 IO 的資源比較不會那麼競爭。

B. Speedup Factor

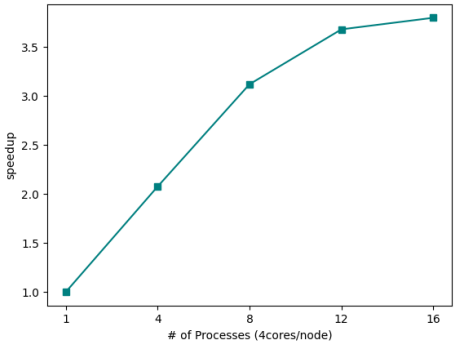
1. Single node

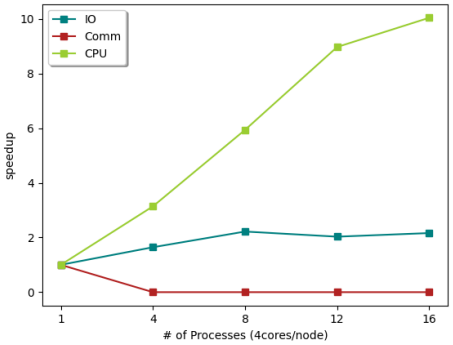


從實驗結果可以看到在 single node 上使用過多的 process 其實是有害的，speedup 會有倒退的情況。原因是同一個 node 上用太多 process 會有彼此競爭 IO 資源的

問題，也因此抵消了 IO 的平行效果。

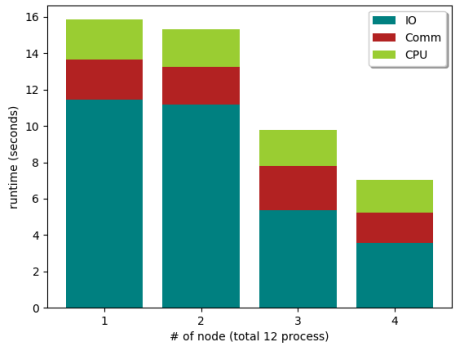
2. Multiple node

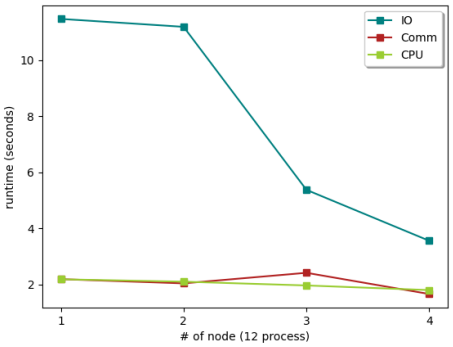


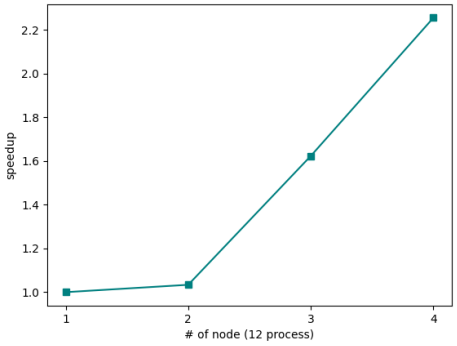


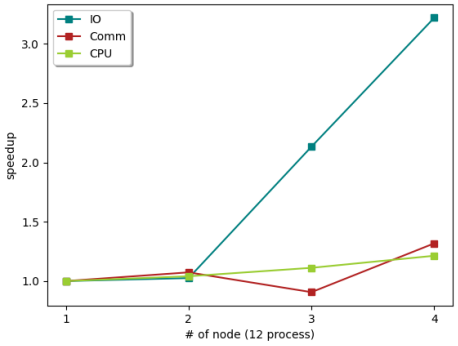
從實驗結果可以看到，即使使用 multiple node，這次實作平行化的 scalability 其實還是不太好，整個 speedup 的曲線低於理想值非常多。也可以注意到超過 8 個 process 以後 speedup 的效果就開始趨緩。從分項圖可以觀察到原因是在多 process 的情況下 IO time speedup 的程度達到極限，同時又有 parallel program 額外增加的 Communicate time。另外值得注意的是 CPU time 本身的 speedup 也是低於理想值，沒有增加幾倍的 process 速度就增加幾倍。

C. Further analysis









為了驗證前兩部分的實驗結果，我另外做了固定 process 數量測試不同 node 數的

實驗。我固定 process 數為 12 個，並將 process 平均分散在不同的 node 數上。根據實驗結果可以發現隨著 node 數增加時間會減少。其中原因也非常明顯，就是因為 IO time 隨著 node 數增加而大幅減少。這證實了前面的推測，當很多 process 塞在同一個 node 上時會讓 IO 很沒效率，反之若是能將 process 平均分散在不同的 node 上就可以使 IO time 大幅減少。

Conclusion

經過這次作業以後我對於平行化更有概念，也更熟悉 MPI 的各種 API。寫作業的過程中也是深深的體會了寫平行程式 debug 非常困難的感覺。平行程式的思考方式跟以往寫程式實在不同，一次又一次 debug 的過程都讓我更熟悉平行程式的思考模式。優化程式的過程也是花了相當多的時間，因為常常想到一個優化並寫完完程式後卻沒有實際反應在跑出來的結果，反而是改一些小地方或是 coding style 以後會有意想不到的效果。