Parallel Programming
Homework 1 Report

110062208 林書辰

1. Implementation

i. 首先, size 記錄了 process 的數量, n 記錄了資料的大小。在分配每個process 的時候我分成兩個 case 來做判斷。

a. 如果 size > n 的話, rank < size 需要處理的資料大小是 1, 剩下的是0

b. 如果 size <= n, 每個rank會得到 floor($\frac{n}{size}$) 的資料, 但如果 n 沒辦法被整除的話, 會多出 n % size 個數字, 我會把它分給前面 n % size 個rank.

在計算完每個 rank 需要的資料量後計算 read in 的 offset 並讀取到 Data。

ii. 每個 process 讀取完自己要處理的資料後, 在 local sort 的時候我用了 boost::spreadsort::float_sort 來加速排序。

iii. Odd-Even Sort 的部分我先判斷 rank < n, 因為 rank >= n 沒有要處理的資料所以不進行 sort。接下來資料傳遞就分成 odd phase, even phase 說明。

a. Odd phase

如果 rank 是奇數的話, 傳資料給 rank - 1, 偶數且不是最後一個 rank則傳給 rank + 1。

b. Even phase

如果 rank 是奇數且不是最後一個 rank 的話, 傳資料給 rank + 1, 偶數且不是第一個 rank 則傳給 rank + 1。

利用 MPI_Sendrecv 同時傳遞資料並用 recv_Data 接收資料。排序的部分我利用變數 sorted 記錄這個phase 是不是 sorted 。接下來先判斷如果 rank - 1 傳過來的最後一個數字比 Data[0] 還小的話, 代表不需要排序, 反之, rank + 1 傳過來的第一個數字比 Data[task_size-1] 還要大的話也不需要排序。如果都不是這兩種情況的話, sorted = 0 並開始 merge。在 merge 的部分因為兩邊都是 sorted array, 所以可以直接用 O(n) 的方式 merge。這裡我用 temp_Data 暫時紀錄資料, 花的 iteration 是 2 * task_size。最後我用 MPI_Allreduce 把所有的 sorted AND 起來到 global_sorted, 當 global_sorted = 1 的時候結束排序。

iv. 因為我覺得 send recv 大量的資料很花時間且不一定所有資料都需要被傳過去 sort。所以我寫了一個能夠只傳部分資料給其他 rank 的 code。

```
//sent part of the data to decrease the send recv time
int numerator = 5, denominator = 10;
int task_size_send = max(1, (normal_size * numerator) / denominator);
int recv_size = max(1, (normal_size * numerator) / denominator);
```

傳給 rank + 1 的資料是 Data[task_size - task_size_send : task_size-1]，給 rank - 1 的則是 Data[0 : task_size_send-1]。這樣的方式可以減少傳輸的資料，也有機會減少傳輸不必 sort 的資料。

2. Experiment & Analysis

i. Methodology

a. System Spec

cluster provided in class。測試的時候用srun跑的

b. Performance metrics

我利用 mpiP 的 profile 來看I/O time, communication time, CPU time。最後再用 python把 mpiP 算出來的時間 plot 出來。

ii. Plots : Speedup Factor & Profile

- **Experimental Method**
    1. **Test case Description** : 我拿第 33 筆測資來測試與實驗，因為他是我所有測資裡面跑最慢的，我覺得這樣在測試的時候可能會有比較大的差別。資料大小是 536869888。
    2. **Parallel Configurations** : 我測試了 1-3 nodes, 1-12 processes 36種情況。我的主要測試的 code 在每次 Sendrecv 都只會傳 task_size 一半的資料量。
- **Performance Measurement**
    1. 我用 mpiP 的 profile 來做 performance 的分析
    2. mpiP 有提供 AppTime, MPITime, 還有所有的 MPI function的時間，再把每個task的時間取平均。我把MPI_Sendrecv以及

MPI_Allreduce都歸類在communication time。CPU time是Apptime - MPItime。

```
---------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------
---------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0       21.9       17.3    78.95
   1       21.9       15.7    71.82
```

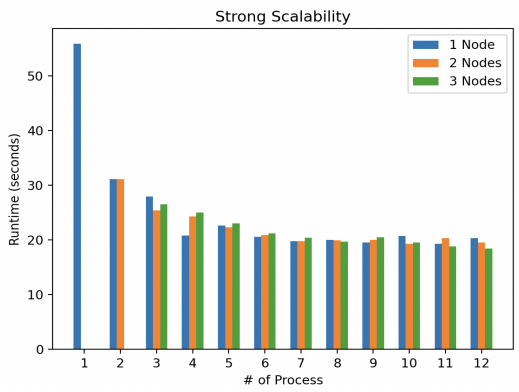
3. 在 mpiP 裡面有提供 Callsite Time statistics，可以在這裡計算所有MPI function的時間，我的做法是把每個 process 的 I/O time, communication time, CPU time都取平均。

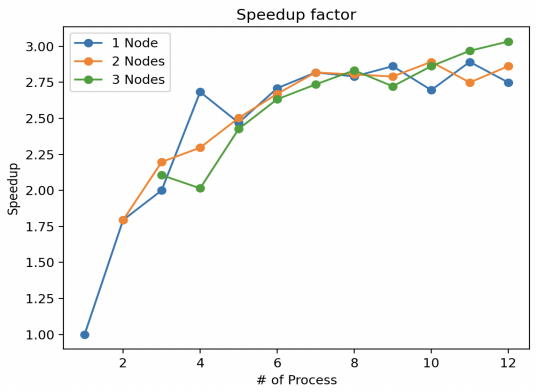
- **Analysis of Results**

我用 python 計算 mpiP 的結果並畫成下面所有的表格。

1. **Strong Scalability & Speedup**

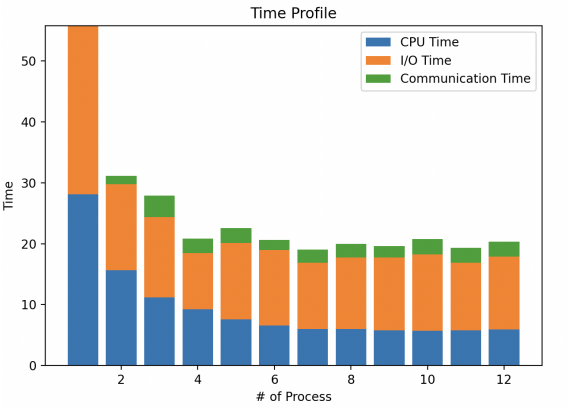
我紀錄了 1 到 12 個 processes 以及 1 到 3 nodes 的排列並且紀錄了 mpiP profile 裡面的 AppTime，可以看到用越多的 processes，執行的時間越短。但是可以發現 node 的數量似乎沒有什麼影響。在我看完 mpiP report 之後，我發現就算我用 3 個 nodes，這些 processes 也大多會集中在同一個 node，可能因為這樣所以 nodes 沒有太大的影響。也可以看到 Speedup 幾乎是越來越好的，不管多少nodes。

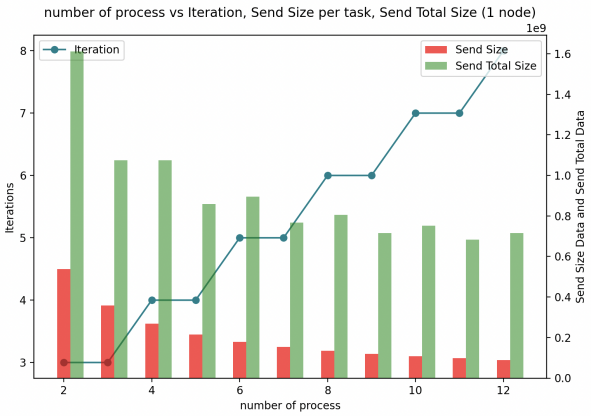


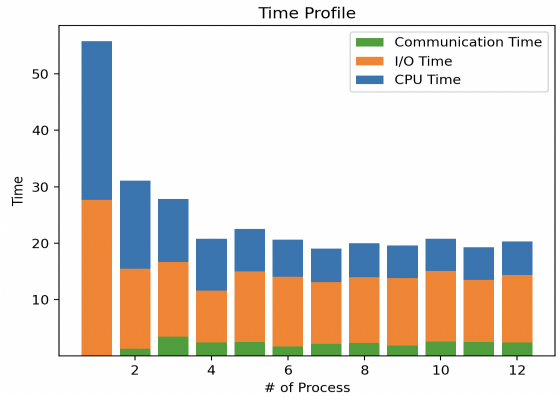


2. **Time Profile and others**

這裡我用 1 個 node，1 到 12 個 process 12種不同的方式做實驗，時間單位秒。可以看到用越多的 processes， CPU Time 越來越少，但在大約 7 個 processes 之後就沒什麼變化，I/O Time 似乎沒有明顯跟 # of process有關係，也有可能是因為我用 srun 跑導致 I/O 的變化比較大。但可以看到 communication time 在前面一兩個 case 有消耗相對多的時間，但不知道是不是誤差，跟 proccess 數量沒有太大的關係，所以我又畫了一張圖監視不同 process 數量的情況下每個 process 傳了多少資料，還有每個 processes 總共用了幾次的 send recv。可以看到總共傳的資料是越來越少的(右圖綠色條狀圖)，但也用了較多的 iteration。重新排序後可以看到 Communication Time 在這裡沒有加速的跡象。所以可以知道這樣只有對CPUTime的優化是相對明顯的。



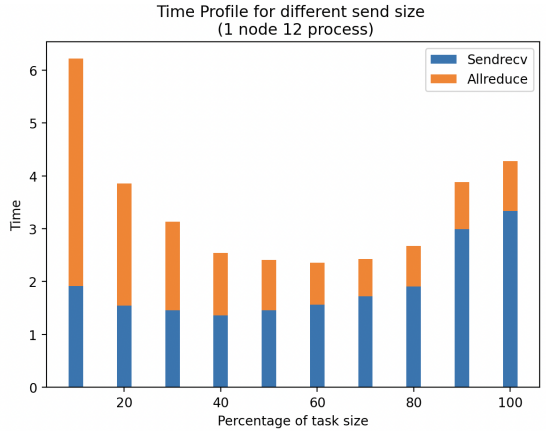


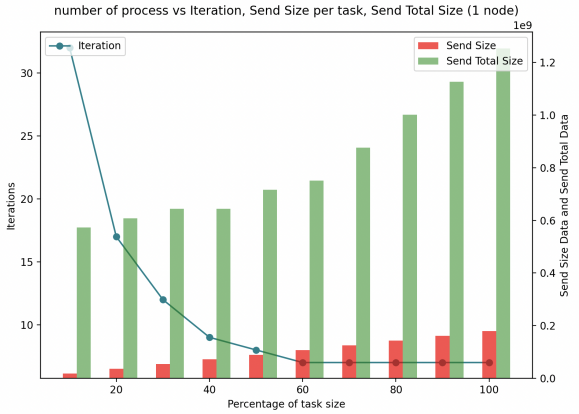


(重新排序後的結果)

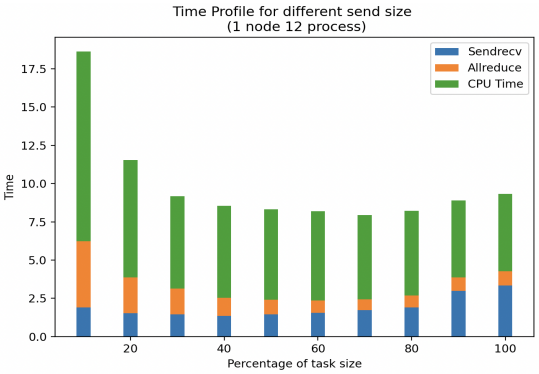
3. Optimization Strategies

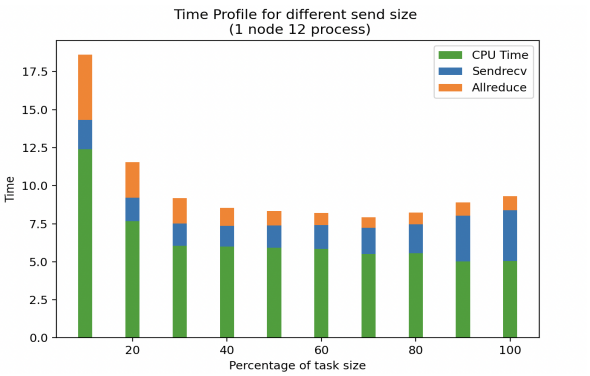
前面有提到關於 iteration 還有傳送資料的關係，為了進一步了解，我固定了 node, process 的數量在 1, 12。然後分別測試了每次只傳 10%, 20% … 100% task_size 的資料給隔壁的 process 進行 merge。因為固定了 process 還有 node，所以我這裡沒有討論 I/O 因為寫入資料大小一樣。對於 Senderecv，可以看到低點大約在 40%, 50%附近，相對的，Allreduce 的低點大約在 70%左右，可以看到他們加起來的低點大約在 50% - 60% 左右。





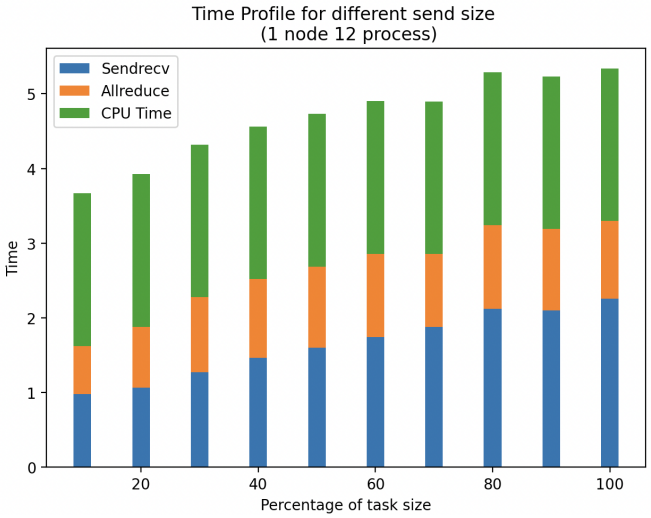
接著討論 CPU Time，因為它佔了比較多的時間，所以對整體的影響比較大，下圖綠色的部分是 CPU Time。可以看到 CPU Time 的時間是隨著傳送的資料量大小逐漸減少的，因為這個測試方法 iteration 相差得比較大，所以可以看到他對 communication Time 的影響。因為後面大概 60% 之後iteration就維持不變，所以多傳了不需要的資料導致 communication Time 開始上升。我們可以知道 CPU Time 是跟 merge 次數 (local 的操作) 比較相關的，所以考量到越後面 CPU Time 可能減少的機會小，communication Time 增加的可能性高，所以我覺得大概在每次傳輸 50% - 60%的時候會有最好的average 表現。(下面兩張圖只有調換上下順序)





iii. Discussion

1. 經過不同的測試之後，我推論出每次只傳 50% - 60% task size 的資料會在 average case 有相對好的成績。但我只有用一組資料進行測試，基於時間關係並沒有測試很多不同組合的資料，但我有拿最後一筆，也就是40筆的資料稍微測試，資料大小和第33筆一樣，同樣測 1 node 12 processes的情況下傳送不同資料大小的情況，這組資料剛好傳送10%到傳送100%資料都只需要五次的iteration就可以完成，所以可以發現時間是直線往上的，所以並沒有辦法確定一次傳多少%的資料會比較好，因為我們不知道怎麼樣可以讓他用最少的iteration完成。但相對的，這樣的情況增加傳送資料後需要的的時間也沒有增加太多，所以說我們可以捨棄一些這樣的極端狀況來優化像第33筆資料那樣可以進步更多的資料。



2. 我認為這個程式的bottleneck在I/O還有communication，如上面所說，我只想到一個相對好的方法去處理communication跟傳輸資料次數還有大小的，並沒有真的用到一個可以確實減少communication time的方法。關於I/O的話，如果有其他方法可以更快的讀取，那應該能夠提升許多的性能，因為照上面的圖表來看，I/O我並沒有方法能夠讓他加速，而且他也占了一大部分的時間，所以她是我最大的bottleneck。
3. 再回到上面的strong scalability，不管幾個nodes，執行時間很快就到達了低點沒有再下降，我認為沒有做到很好。我覺得如果可以隨著process增加減少I/O time的話或許可以表現得更好。

3. **Experiences / Conclusion**

這是我第一次學習有關平行運算的東西，雖然陌生但都還算好上手，刻出基本的程式還不算太困難，但我認為優化的部分還滿無力的，一開始想法很少，雖然後面想出來的也都不是太厲害的優化方法，倒是聽到別人說用boost:: spreadsort ::float_sort可以快很多讓我的程式進步了很多，到現在也還沒有太多能夠優化的想法，但是在寫完report、分析了這些結果之後，我知道其實我優化的很大一部分是CPU Time，原本以為花比較多時間的I/O，communicatino都沒有做到太多的優化。希望之後能聽到其他人的做法，也希望在之後的功課可以有更多的想法來優化。