CS542200 Parallel Programming Homework 1:

Odd-Even Sort

112062673 吳文雯

1.Implementation

◆ How do you handle an arbitrary number of input items and processes?

當資料量為 n 時，利用整數除法計算 n 除以 process 數量，算出的結果是除了最後一個 process 要分配的數量，剩下的交由最後一個 process 處理。雖然最後一個 process 分到的數量可能會比其他 process 多，但最多也只會多 process 數量減一個資料量，而因為 process 數量不會到非常大，且最後一個 process 參與交換的回合數較少，因此我認為這樣的資料量差異還算在可以接受的範圍內。

◆ How do you sort in your program?

在開始跟其他 process 交換資料以前，先利用 spreadsort 將自己分配到數字由小到大排序，會選擇 spreadsort 是因為當要處理的數字量很大時，他算是速度比較快的一種排序方法。

在進行 odd-even phase 的交換時，每個 phase 讓有人配對的 process 將自己所有的數字傳給鄰居，並且接收鄰居那邊的資料。接著 rank 較小的 process 就會進行 Merge_small，取得在兩人的所有資料間相對較小的數字，方法類似 mergesort 中的 merge 階段，因此取得的數字會是排序好的，而 rank 較大的 process 則會取得較大的數字，兩邊取得的數量都是原本分配到要處理的資料量。最多經過 process 數量個 phase，所有的數字就會完成排序。

◆ Other efforts you've made in your program.

在這次的程式中為了增進效能作了以下優化:

a. 避免反覆調整記憶體大小: 只在一開始使用 new 分配給一個 process 兩個 buffer(buf 和 tmp)，大小是他要處理的數字量，buf 用來放要讀取/寫入檔案的數字，tmp 是在 Merge_small 或 Merge_large 階段使用的暫存 buffer。在之後的過程中，process 就只會處理固定的數字量，不會再去動態調整記憶體的大小。
b. 使用指標對調取代 data copy: 首先，在 Merge_small 及 Merge_large 函數都是利用指標取得 buf 及 data 的位置，在進行資料比較，而不是傳整個陣列資料。再者，在 Merge_small 及 Merge_large 中會將新蒐集到的數字放在 tmp 陣列中，從函式返回後會讓原本指向自己 buf 陣列跟指向 tmp 陣列的指標交換，比起將 tmp 的資料 copy 到 buf 中，這樣的作法更為快速。

c. 避免重覆計算:由於每一輪 odd-phase 跟 even-phase 中，每個 process 配對的鄰居都是固定不變的，因此在進入 iteration 前就先算好鄰居的 rank 及數字量，並且存在變數中，如此在每一輪不用重新計算。
d. 在程式碼中盡量避免過多的 branch 敘述或邏輯運算，避免在程式中層層比較會浪費時間。
e. 所有 if-else 結構都是先將常遇到的情形放在前面,這樣可以減少不必要的條件檢視。
f. 將 MPI_Send 及 MPI_Recv 改成 MPI_Sendrecv()。
g. 一開始使用 qsort 後來改成 spreadsort。

2. Experiment & Analysis

i、Methodology

(a). System Spec: 使用的是課堂提供的 cluster。

(b). Performance Metrics:

在比較程式整體執行時間時會採用 ipm 產出的 wallclock。至於在了解各個部分執行時間會利用 MPI_Wtime()分別計算各個我認為較花時間的階段，詳細的說明如下:

- Cpu time = 一開始拿到資料後各自 sorting+中間 merge 資料的時間。
- I/O time =
  (MPI_File_open+ MPI_set_view +MPI_File_read_all+MPI_File_close) +
  (MPI_File_open+ MPI_set_view +MPI_File_write_all+MPI_File_close)
- Communication time = MPI_Send + MPI_Recv。(之後改成 MPI_Sendrecv)

ii、Plots: Speedup Factor & Profile (including discussion)

Experimental Method:

- Test Case Description: 使用的 testcase 是 35.in，資料量為 536869888。

- Parallel Configurations:

先使用單個 node，12 個 processes，這是因為想避免受到網路或者 node 之間工作分配的影響，希望看見程式設計本身可以改善的地方。

Ipm-profile 結果如下:

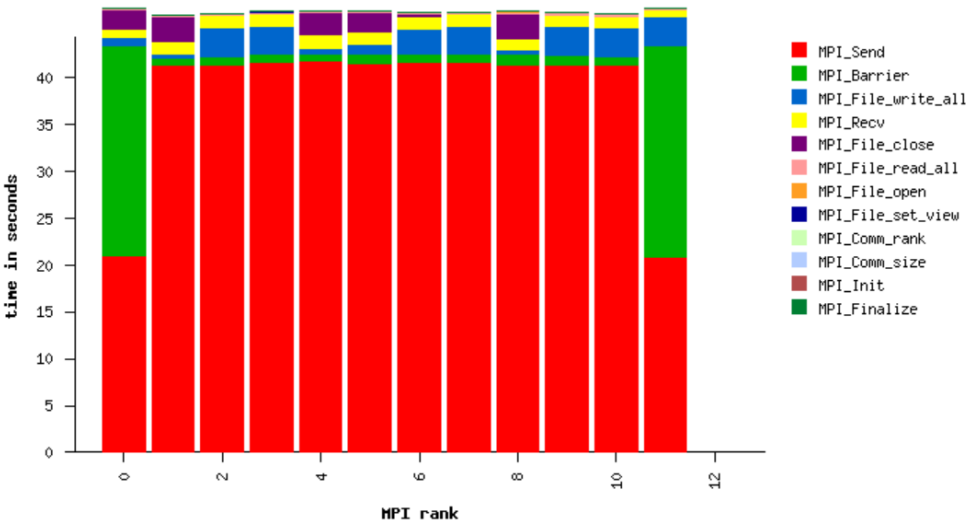

圖 1: Single node, 12 processes communication balance by rank

由這張圖中可以看出，各個 process 間的 loading 大致平均，頭尾兩個 process 的 send 比例特別短，而 barrier 比例特別高，這是因為他們在 odd-phase 時會因為沒人可以配對而 idle，我認為屬於正常現象。

因為在 ipm 的圖表中無法清楚看出各個 process 在 i/o、cpu 及 communication 的時間比例，因此我利用 MPI_Wtime()計算 process 在各個主要部分所花費的時間，結果如下圖。

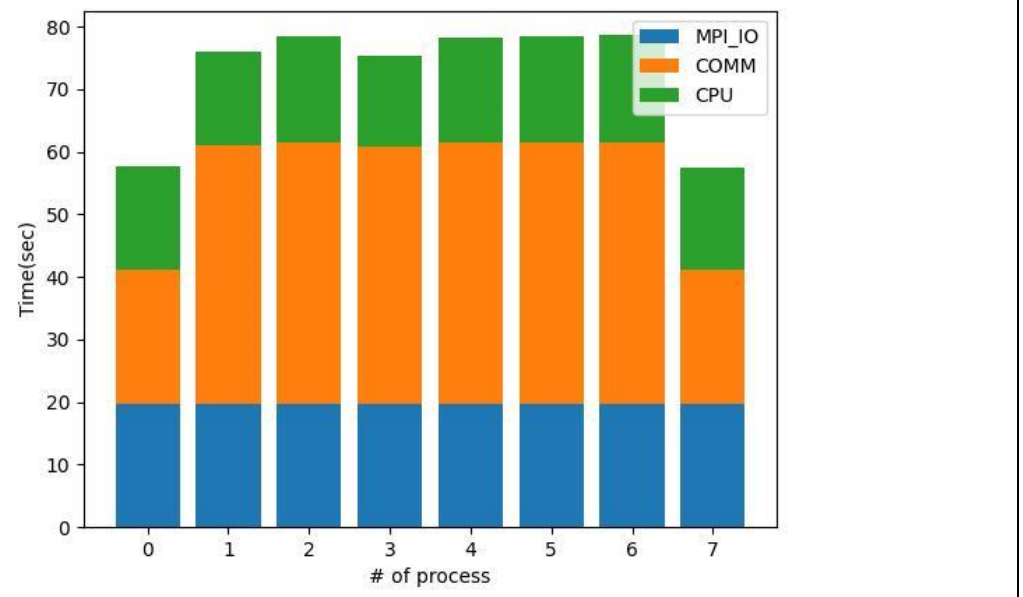

圖 2: Single node, 8 processes profile

從這裡可以看出 process 花在 communication 的時間是這三者當中最高的。
我也進行了實驗比較在單一 node 上使用不同 process 數量的結果。

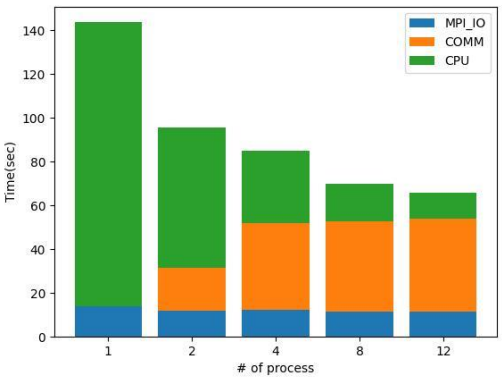

圖 3: Single node, 12 processes profile

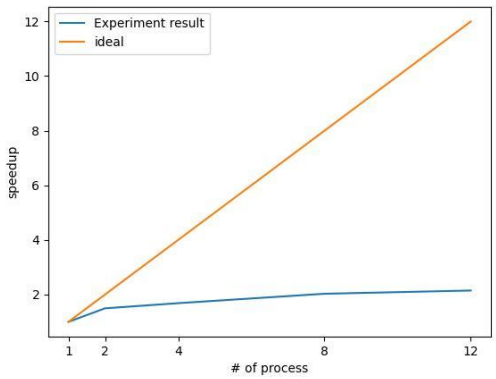

圖 4: Single node, 12 processes scalability

當 process 越多時，每個 process 分到的計算量就越少，因此 cpu time 就明顯減少。但同時也因為有更多的 process 需要互相溝通傳資料，communication time 就會上升，可以看到當數量為 2 及 4 時，溝通時間大約差了兩倍。

在 scalibility 的部分可以看到加速效果很差，可能是因為在 process 間的溝通 overhead 太重，導致加速效果並不好。

而若是使用多個 node，每個 node 8 個 processes 的實驗結果如下:

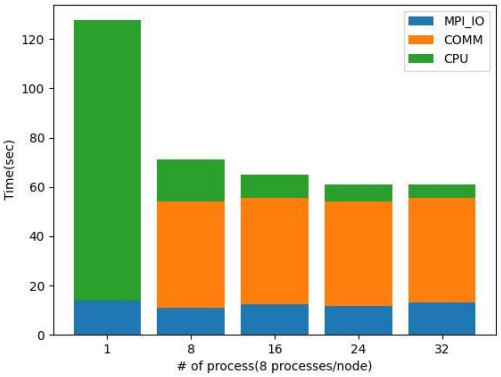

圖 5: Multi-nodes profile (8 processes/node)

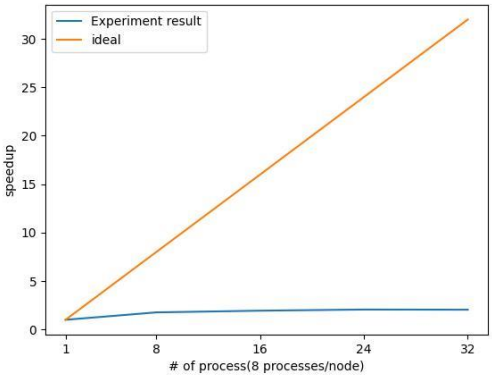

圖 6: Multi-nodes scalability

這裡可以觀察到 communication time 在 processes 數量從 8 上升到 32 之間並沒有太顯著的差異，這可能是因為在這個演算法中，process 只會跟自己前後的 process 交換資料，傳遞的對象並不會跟著上升。同時，雖然傳資料的次數跟 process 數量相當

(因為要進行 process 數量個回合)，但是每次要傳送的資料也會隨著 process 數量變多而下降，因此可能導致 process 數量在 8-32 之間，單一 rank 所需的 communication time 是差不多的。
另外一個觀察是當使用 32 個 processes 時，communcation time 幾乎佔據所有的時間，因此認定他就是整個程式的 overhead。從圖 6 也可以看出 scalabilty 跟 ideal speedup factor 的差距顯得更大了。

Optimization:

為了改善 communication 帶來的 overhead，且考量原本程式在 MPI_Send()之後立刻呼叫 MPI_Recv()，中間也沒有其他運算，因此決定改成使用 MPI_Sendrecv()，改善後的效能結果如下:

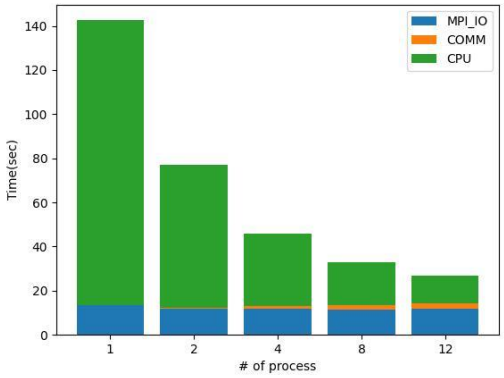

圖 7: Single node, 12 processes profile (使用 MPI_Sendrecv)

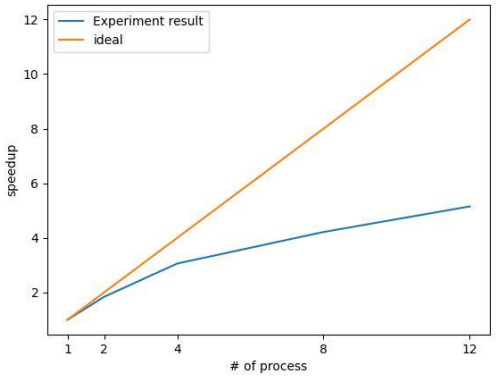

圖 8: Single node, 12 processes scalability (使用 MPI_Sendrecv)

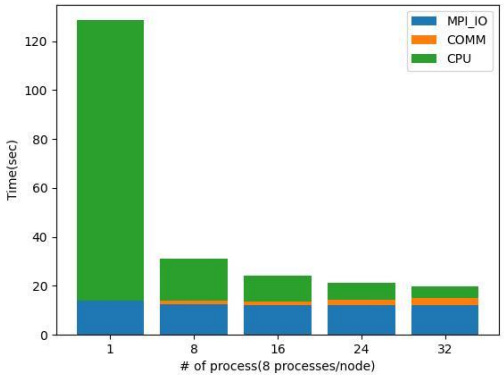

圖 9: Multi-nodes profile (8 processes/node) (使用 MPI_Sendrecv)

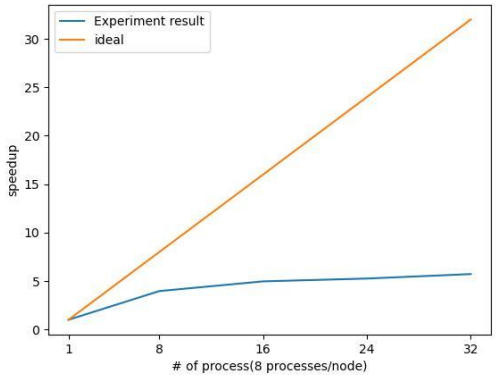

圖 10: Multi-nodes scalability (使用 MPI_Sendrecv)

從圖表中可以發現不論是使用單個 node 或是多個 nodes，communication time 都明顯減少許多，整個程式執行時間下降，scalabiltiy 的部分也提升了一些。

雖然經過優化以後的效能已經提升了許多，但是在 strong scalability 的實驗結果，

還是可以看到距離 ideal speedup factor 有很大一段差距。這可能是因為 odd-even sort 是一輪一輪進行的，在每一輪中的 MPI_Sendrecv 會讓 send 和 recv 兩邊的 process 同步化，因此要等到鄰居也完成上一輪的處理後，才會進入下一輪，如此才能確保最後的結果正確。同時，在 odd-phase 期間，頭尾的 process 可能會因為沒人配對而無法做任何事情，這可能都是導致程式整體無法達成線性加速的原因。

之後我也嘗試改善 cpu 所花的時間，將原本的 qsort 改為 spreadsort，調整後的結果如下：

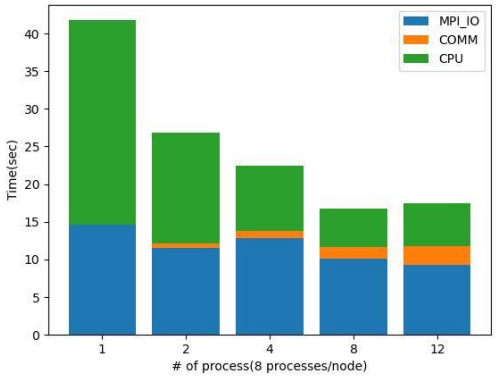

圖 11: Single node, 12 processes profile (使用 spreadsort)

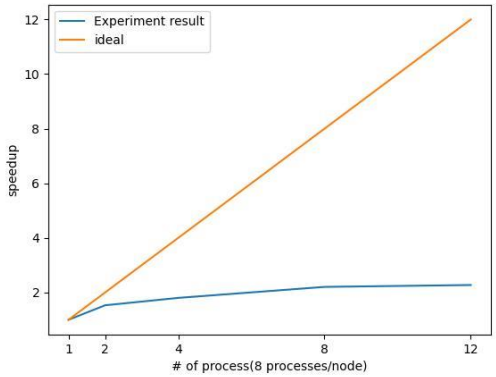

圖 12: Single node, 12 processes scalability (使用 spreadsort)

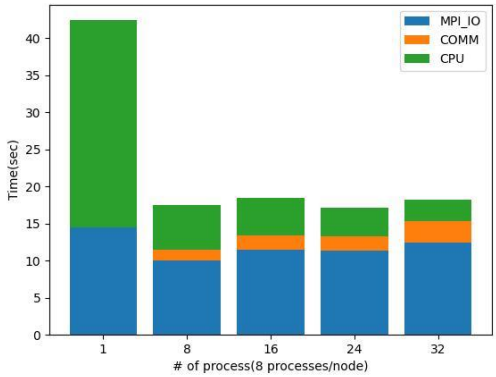

圖 13: Multi-nodes profile (8 processes/node) (使用 spreadsort)

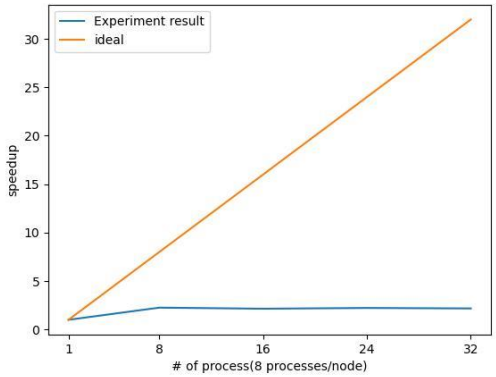

圖 14: Multi-nodes scalability (使用 spreadsort)

由圖表可以看出程式花在 cpu 的時間大幅縮減，使得程式可以在更快的時間內完成排序。不過在 scalability 的部分則有些微變差的情形，這可能是因為 spreadsort 只能減少在一開始排序自己資料的時間，進入 iteration 後還是需要等待其他 process 完成 merge 數字，才能一起進入下一輪。同時，這個改善方法也無法縮短 i/o time 及 communication time 的時間，最終 i/o time 成為整個程式的 bottleneck，必須等待最慢的 process 寫入檔案後才能一起結束，而當使用的 node 越多時，process 間寫入磁碟的速度可能越不平均，這可能也是導致 scalabilty 不是很好的原因之一。

4. Experiences / Conclusion

- 一開始我採用的方式是一次傳一個數字給鄰居，透過比較大小來決定下一輪要傳同一個數字或是下一個數字，結果跟目前版本一樣都是讓 rank 較小的取得兩個 process 中較小的數字，且是排序好的。然而這樣的做法在 judge 時就產生了好幾個 time exceeded 的結果，因此後來花時間思考如何用較少的時間一次傳送整個陣列的資料，才得到後來的版本。
- 為了減少程式較少的執行時間，這次基本上是採用"以空間換取時間的方式"，每一個 process 都多開了一個跟數字量一樣大的記憶體空間當暫存 buffer，這點跟之前盡量節省記憶體空間的程式邏輯較不一樣。此外，在撰寫過程中也會很斟酌每一行的寫法，希望能盡量提升程式的執行效能，相信這次的經驗對往後寫程式會很有幫助。