Parallel Programing HW1: Odd Even Sort

110062511 倪滙渝

1. Implementation

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

Case 1 process 數量=1

用 C++內建的 sort function 排列整個 sequence。

Case 2 process 數量> input items 數量

重新建立一個 communicator，其中 communicator 中 rank 數量和 input items 數量一樣多，多餘的 rank finalize 掉。

Case 3 process 數量< input items 數量

將 input items 平均分配給每個 process，若是仍有剩餘的 items，再將其分配給部分 process。 Ex: input items=18, process=5 → 依序分配為 4, 4, 4, 3, 3

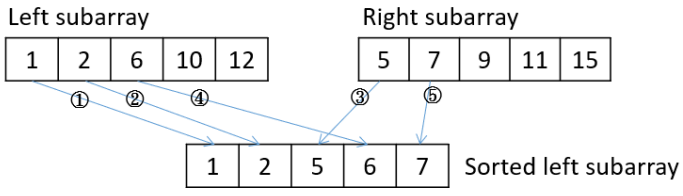
How do you sort in your program?

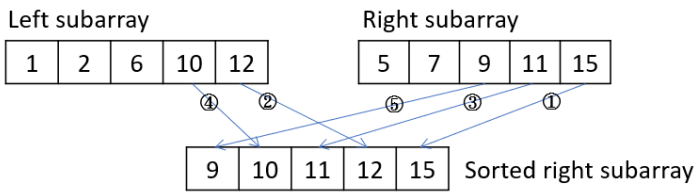
Initialize

首先 initialize 每個 process 中所包含的 list，使用 C++內建的 sort function 來排列。

Sort

接著兩個 rank 在溝通時，從頭開始依序比對，較小的放在新的 array，得到一個排序好的 array。在實作時，左邊的 rank 只會排前半段，右邊的 rank 只會排後半段，以減少耗費的時間。





Converge

每一次要 sort 之前，都會判斷是否左邊 rank 最大值小於右邊 rank 最小值，若是有，則會進行 sort，若是所有的 rank 都沒有進入判斷式，即為排序完成。

2. Experiment & Analysis

◆ Methodology

Performance Metrics: How do you measure the computing time, communication time and IO time? How do you compute the values in the plots?
使用 C 內建的 clock 函數來測試，每個 performance 所包含的範圍如下：

IO time

MPI_File_read_at、MPI_File_write_at、MPI_File_open、memcpy

Computing time

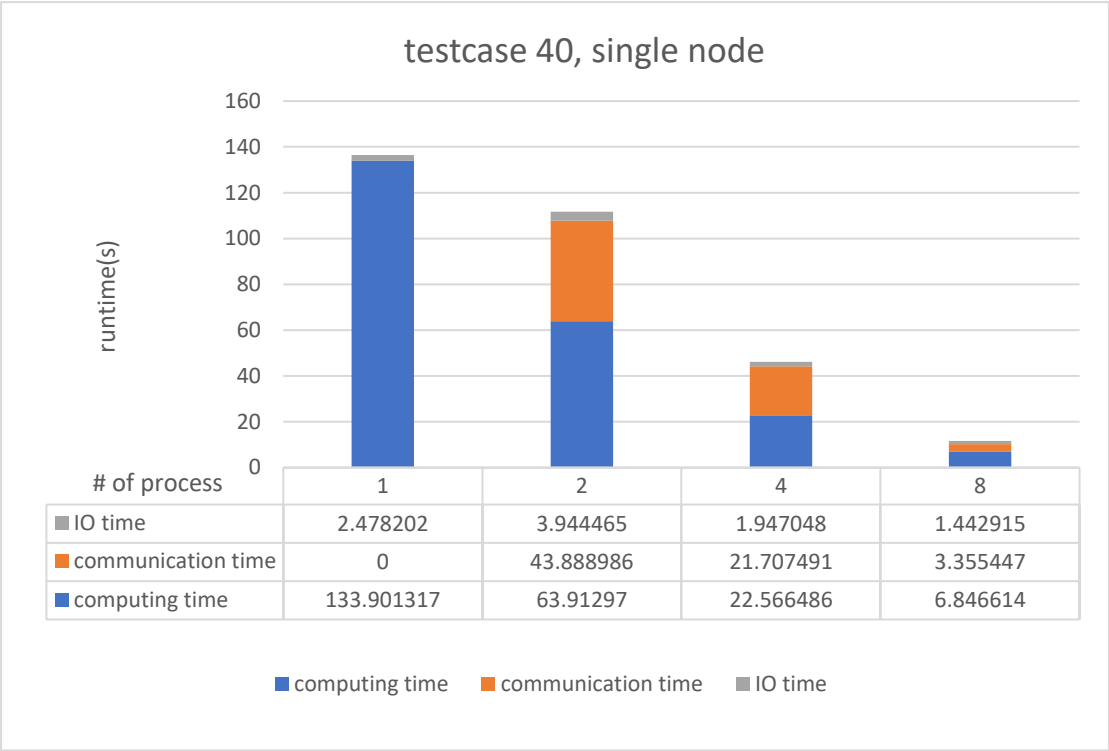
sort、merge_and_sort

Communication time

MPI_Send、MPI_Recv

Plots: Speedup Factor & Time Profile

以下使用 testcase 40 (n=536869888) 來執行實驗：



# of process	1	2	4	8
IO time	2.478202	3.944465	1.947048	1.442915
communication time	0	43.888986	21.707491	3.355447
computing time	133.901317	63.91297	22.566486	6.846614

Figure 1: Time Profile of Single Node Environment

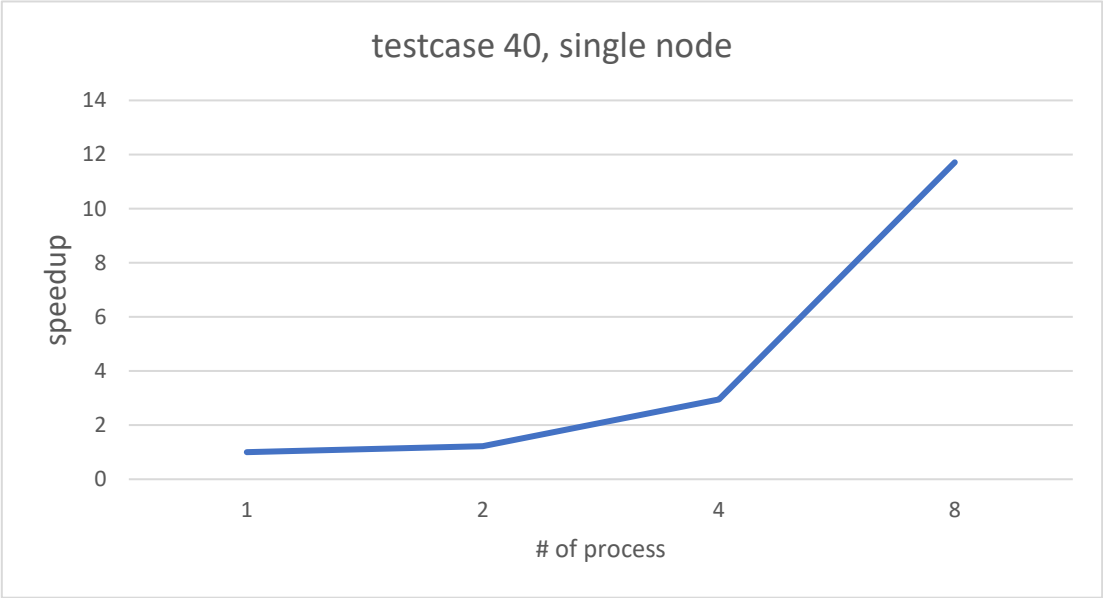
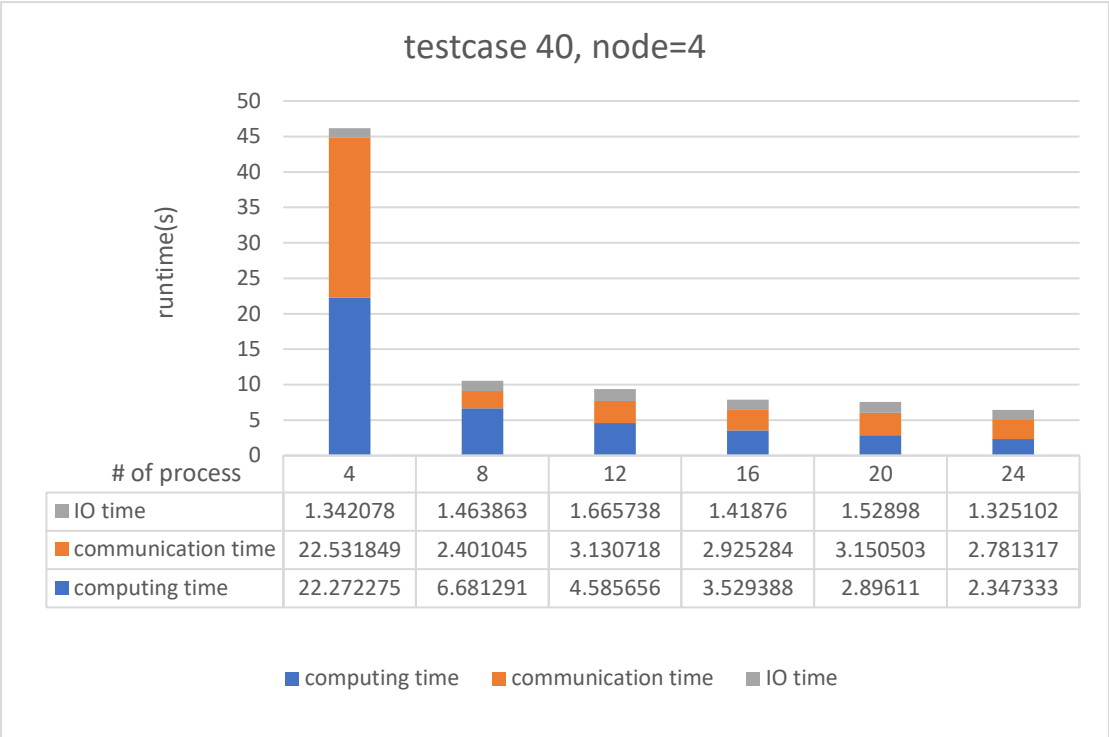


Figure 2: Speedup Factor of Single Node Environment



# of process	4	8	12	16	20	24
IO time	1.342078	1.463863	1.665738	1.41876	1.52898	1.325102
communication time	22.531849	2.401045	3.130718	2.925284	3.150503	2.781317
computing time	22.272275	6.681291	4.585656	3.529388	2.89611	2.347333

Figure 3: Time Profile of Multi Node Environment

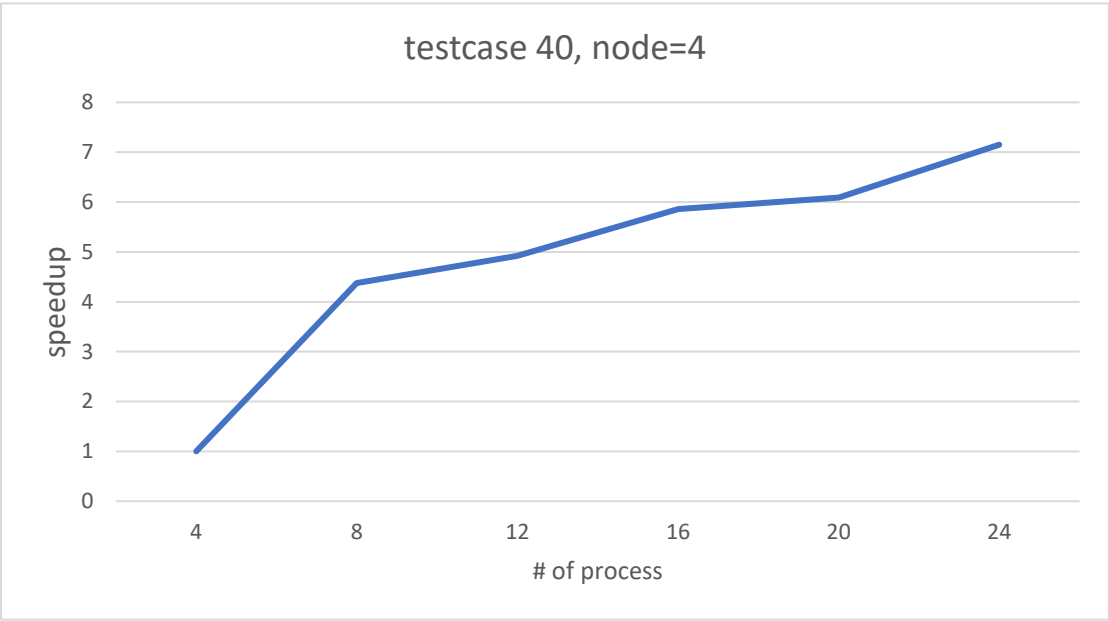


Figure 4: Speedup Factor of Multi Node Environment

◆ **Discussion (Must base on the results in your plots)**

Compare I/O, CPU, Network performance. Which is/are the bottleneck(s)? Why? How could it be improved?

在 single node 的情況下，可以看到在 process 數量為 1 的時候，computing time 佔據了大部分時間。而隨著 process 的數量增加，computing time 隨之會變少，到了 process 為 4 的時候，communication time 已經和 computing time 相差不遠了。

在 multi-node 的情況下，communication time 和 computing time 不會相差太多。若是詳細觀察各個 case，當 process 數量為 4、8、12、16 的時候，computing time 會比 communication time 來的多。當 process 增加為 20、24 的時候，computing time 可以減少許多，然而 communication time 反而變成 bottleneck。這表示若是資料量非常大的時候，不停地增加 process 數量並不是唯一的解方，也要處理 communication 時間耗費的問題。

Compare scalability. Does your program scale well? Why or why not? How can you achieve better scalability?

我認為我的方法沒有達到很好的 scalability，除了在分數板上排名並不是很亮眼之外，查了 MPI 的 library 發現，還有很多好用的工具可以來實作這個作業，只是沒有好好挖掘。由於在 process 數量夠大時 communication time 還是會阻礙效率的提升，也許用 nonblocking 的方法來進行 communication 會是一個不錯的選擇。

3. Conclusion

從這次作業中，我熟悉 process 之間是如何去實現 message passing，也了解 MPI 一些 function 的實作。這個作業的邏輯不會很難，困難的地方在於，由於我對於 C 語言比較生疏，對於分配 memory 以及找到對應的 index 等等的細節會很容易忽略，導致會有 runtime error，在這個情況下 debug 花了不少時間。不過當寫出來速度提升時，的確很有成就感，也許這是一個接觸平行程式一個好的開始。