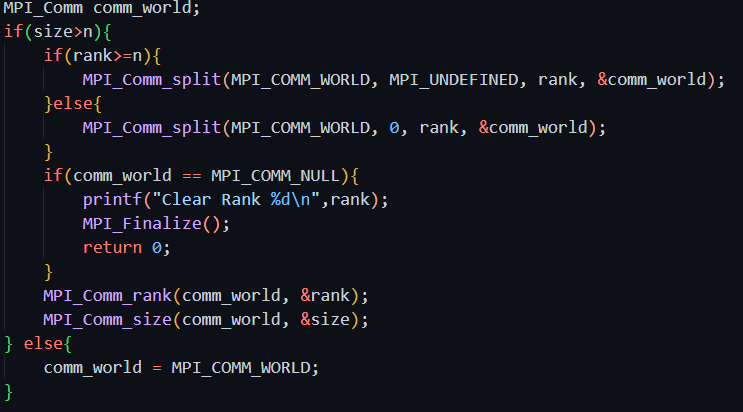
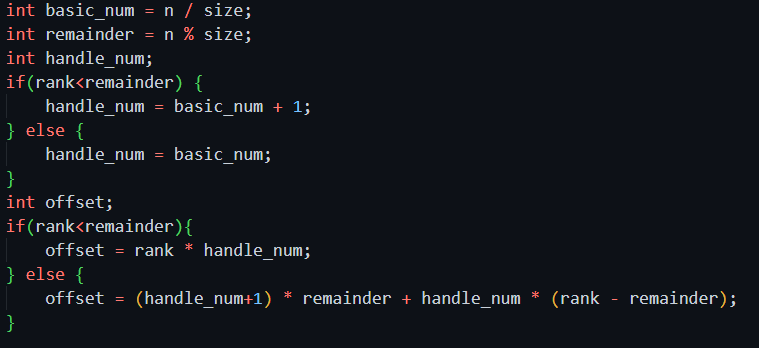
**Homework 1 : Odd-Even Sort Report**

**110164508 黃柏惟**

**Implementation**

1. **Handle an arbitrary number of input and processes**由於input的數量以及processes的數量皆為不固定的因此需要一些處理才能夠應用到任何情況下。 這邊共分為兩種情況，情況一為輸入的input number數量比processes少，這時候我會將多餘的processes透過MPI\_Comm\_Split()的方式將其關閉。 情況二為較常見的情況，當今天輸入的input比processes數量還要多的時候我會採用平均分配的方式，並且讓剩下除不盡的分配給前方的Processes，這樣子可以讓每個processes在處理資料的數量上可以盡量相同避免整份code在等其中一個processes完成communication。  
     
   ****  
   情況一的處理  
   ****  
   情況二的處理
2. **Sorting**在一開始load data時由於當前的data都是亂數排列的，因此需要先進行第一次的sorting，這邊直接使用boost library底下的spreadsort()，相較於qsort()整體執行時間上會快上不少。  
   在完成第一次sorting後後續在processes溝通時就不需要再採用sorting，因為此時的data皆為排列的狀態，因此直接將兩個array當前最小的值相比再merge會比較快。
3. **Odd-Even Sort**我的作法為每輪都會執行一次odd phase sort跟一次even phase sort。以odd phase sort為例，我會將所有奇數的rank的data往rank-1送。 Rank-1在收到data後就會比較送進來最小以及當前自己的最大值相比，由於此時雙方都是sort的狀態，因此只要自己的最大值比對方還要小的話我們就可以不用執行sort；反之，如果自己的最大比接收到的data最小值還要小的話那麼就代表需要執行sorting。 Sort的方式由於雙方都是排序過的狀態因此只需要透過pointer去進行最小的比較就可以比較快的把雙方的array merge and sort。在完成後我們會透過一個sort flag去記錄說該輪的這個rank有執行到data的變動。 並且我們會將sort過後的data後半段的部分傳回給rank並完成該階段的phase sorting。  
   在Even phase 時與Odd phase 相同故不再贅述。  
   當完成一次odd phase sort與even phase sort後我們會呼叫MPI\_Allreduce()去把sort的flag收回來進行檢查，如果有其中一個flag是false的話代表該輪有更動的當前的值因此需要全部再重新檢測一遍直到所有收回來的flag都是sort過的狀態。  
   在傳遞data的時候要特別注意邊界的處理(rank=0, rank = max\_rank),因為邊界的rank不像一般中間的rank可以朝左或朝右send 跟 receive。 以rank=0為例，今天在even phase的時候雖然他是even的但是他就不會往左send data，並且他也不會從左邊接收到計算完的data。

**Experiment & Analysis**

1. **System Spec**使用appollo.cs.nthu.edu.tw server進行實驗
2. **Performance Metrics**使用MPI\_Wtime()將要紀錄的區段進行包裹。 此次共進行三種行為分別是IO、Processes間的communication time、以及CPU執行運算的時間在不同的processes以及nodes的執行時間進行分析。 本次分析使用第40筆open test case去進行分析，該資料有536869888筆浮點數資料需要進行排序運算。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 8 |
| IO | 11.104 | 16.329 | 10.971 | 2.95 |
| Comm | 0 | 1.17 | 2.224 | 3.271 |
| CPU | 20.582 | 12.026 | 5.929 | 3.05 |
| Total sum | 31.686 | 29.525 | 19.124 | 9.271 |
| Speedup | 0 | 2.161 | 12.562 | 22.415 |

**實驗一、 Single node在不同Processes之分析**

**分析:** 可以觀察到當processes數量增加的時候，透過彼此間平行運算的特性，雖然花費了一些時間在進行communication上，但是卻可以大幅的縮減IO time以及 CPU time。 這邊在single node下加速有符合預期以線性的方式提高。

**實驗二 、Multi node在不同Processes之分析**

我們採用 4 processes可以分配到一個node的方式進行分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 8 |
| IO | 11.104 | 3.18455 | 2.76 | 3.256 |
| Comm | 0 | 3.066 | 3.546 | 4.145 |
| CPU | 20.582 | 6.169 | 2.937 | 2.348 |
| Total sum | 31.686 | 12.419 | 9.243 | 9.749 |
| Speedup | 0 | 19.266 | 22.443 | 21.937 |

**分析:** 可以看到在processes數量提升的同時如果給予對應的node數資源的話就可以改善在single node下提升processes數卻無法有更快執行速度的問題。 這邊的bottle neck是在當我把core數量提高到8時，整體的執行時間反而因為communication time過長而導致speedup反而沒有 4 cores時來的好。這邊推測是因為在core = 4的時候其實CPU time已經相對很短了，因此雖然我們在core = 8 時候有更多的資源去計算但是卻不足以彌補communication time的增加。因此如果想要再增加processor的話很有可能就發生communication time dominate，改進的方向可能就要著手於減少communication time，讓已經排序過的資料不要再重複的傳遞。

**實驗三 、Sorting Algorithm 分析**

由於在測試的時候有發現Sorting的演算法對於這次的作業其實佔了蠻大部分的CPU time，因此就想比較不同sorting演算法對於processes改變時的變化差異。

使用Qsort:



使用Spreadsort:



**分析:**當我們的processor數量越少時代表一個processor需要handle越多的data。因此可以從表中發現當今天發現到如果processor數量增加的同時speedup有明顯的優化的話，那麼代表code的bottle neck可能是在CPU的計算上，因此可以先著手於cpu time上的優化；如果今天發現processor數量增加的同時speedup只能有些微的改善時，那麼就代表code的bottle neck可能不是在CPU的計算上而可能是communication上。 根據實驗三了解到不同的speedup值可能代表著當前code不同的bottle neck。

**Conclusions**

算是第一次寫平行程式，一開始花了不少時間在理解平行化的概念。開始實作時也遇到了很多次Send/Receive沒有對起來使整份code卡死的情況發生。不過在完成後可以發現整體的執行時間有因為平行化的處理而降低不少，蠻有成就感的。