HW1 Odd-Even Sort Report

109062101 許佳綺

Implementation

Basic

這次作業中以process作為Odd-Even Sort的交換單位，首先要分資料到每個process上。我將長度為`n`的data array分為`n/numOfPorcess`大小分給每個process，若有未整除的狀況，則前`n%numOfPorcess`個process會多分到一個，如此一來，每個process須處理的data size差異不會超過1。這裡紀錄一個變數`localSize`表示每個process所維護的data長度。

接著計算`offset`，也就是每個process需要從哪裡讀出所分到的data段，讀取完畢後首先讓每個process sort 自己的local data，並進入Odd-Even Sort的交換。

由於Odd-Even Sort最多只需花`numOfProcess+1`次就能完成sorting，故我以for loop的方式，首先判斷目前是在odd-stage還是even-stage，接著在個別的stage中去判斷，奇數rank和偶數rank的process應與左邊或右邊的process交換data，並各自merge data。Merge data部份我使用`O(n)`算法，將兩個array的head抓住比較大小，取出所需要的`localSize`個data為止。

如在**odd-stage**中，奇數rank的process會與右邊的process交換資料，並取前`localSize`**小**的data到自己的`localDataBuf`中; 在偶數rank的process會與左邊的process交換資料，並取前`localSize`**大**的data到自己的`localDataBuf`中。

另外考慮到`numOfProcess`大於`n`的情況，則有些Process不會分到data，則直接將該process的rank設為-1，並且只要該stage中該process的rank=-1或是要交換的對象rank=-1，也會跳過該次Odd-Even Sort。

```
int maxSortNum = numOfProcess+1;
for(int i = 0; i < maxSortNum; i++){
  if(rank == -1) break;
  if(i % 2){
    // odd-stage
    if(rank % 2){
      // rank is odd, get a smaller half ...
    }else{
      // rank is even, get a larger half ...
    }
  }else{
    // even-stage
```

```
    if(rank % 2){
      // rank is odd, get a larger half ...
    }else{
      // rank is even, get a smaller half ...
    }
  }
}
```

最後將Odd-Even Sort結束後的data寫回file中。

Improvement

1. 減少if-else的判斷式計算以及memory allocation

   首先，上述的實作每次都需要判斷現在是奇數rank還是偶數rank，這部分可以換個想法實作，只看每次是需要跟左邊還是右邊的process交換，就可以避免掉太多if-else判斷。此外，上述的for-loop寫法中每次都需要計算`i%2`，當`i`很大則每次都須經過`%`計算會耗時，故改成while loop寫法，並利用一個布林變數`bool stage = rank & 1`紀錄現在該process是處於要與左邊還是右邊的process交換，再每次利用`stage ^= 1`切換要odd-even switch的對象。

   ```
   bool stage = rank & 1;
   while(maxSortNum--){
     if(rank == -1) break;
     if(stage && rightNeighbor != -1){
       ...
     }
     if(!stage && leftNeighbor != -1){
       ...
     }
     stage ^= 1;
   }
   ```

   接者，原本我為每個process創建了`leftRecvBuf`跟`rightRecvBuf`分別儲存從左邊及右邊交換到的data，但後來發現其實只需要創建一個`RecvDataBuf`即可，因為每次只會跟其中一邊交換data而已。並且merge 兩邊data的部分我一開始是將merge function寫好放在main function外面，需要的時候以function call的方式呼叫merge function並將data傳入計算，但後來發現也是可以將merge function裡的功能直接寫在main function中，避免額外的function call及data copy時間。

2. 使用不同的Local Sort方式

   原本使用STL內建的`sort()`，後來嘗試了`qsort()`及`boost::sort::spreadsort::float_sort()`，測試幾次後發現`boost::sort::spreadsort::float_sort()`會最快。

3. 減少MPI傳輸資料的次數

   在資料量很龐大的時候，若每次傳輸所有資料前可以先確認是否有需要傳輸，也就是兩個相鄰的process所擁有的data已經排序完畢，沒必要進行排序的話，即不用花時間傳輸資料了。故可以先比較若左邊的process中最後一個data已經小於等於右邊的process中第一個data時，即符合以排序好的情況，則不用傳輸。

4. 使用Non-Blocking Send Recv

   原本的實作我採用`MPI_SendRecv()`也就是Blocking的方式傳輸資料，後來嘗試使用`MPI_ISend()`, `MPI_IRecv()`, `MPI_Wait()`的Non-Blocking方式傳輸能在hw1-judge上跑出最快的成績，但相對來說比起Blocking方式也更不穩定一些，有時候會比較慢。

Experiment & Analysis

Methodology

1. System Spec

   使用課程所提供的 apollo server。

2. Performance Metrics

   使用 `MPI_Wtime()` 來計算時間，總時間是在 main function 裡的 MPI_Init() 之後直接加入 MPI_Wtime()，並在 MPI_Finalize() 之前接上 MPI_Wtime() 以計算整體時間。而 Communication 和 IO 亦同，在傳輸及檔案讀寫前後各加入start跟end的時間紀錄，利用`end-start`計算所花費的時間，而Calculation Time則是計算總程式的執行時間-Communication-IO的時間。

Plots: Speedup Factor & Profile & Discussion

以下將會直接在實驗圖表底下分析bottleneck、scalability及優化等狀況。

1. Experimental Method

   a. Test Case Description

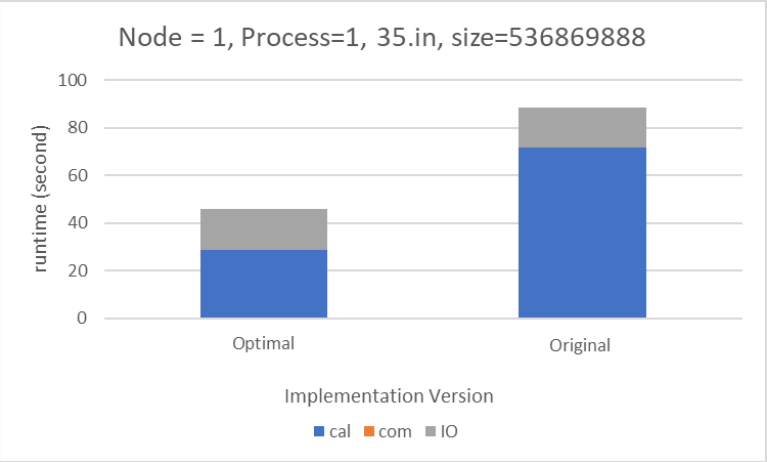
      我選擇testcase/35.in，data 長度為 536869888做為測試資料，在資料量大的情況下可以觀察出平行與否的效能差異。

   b. Parallel Configurations

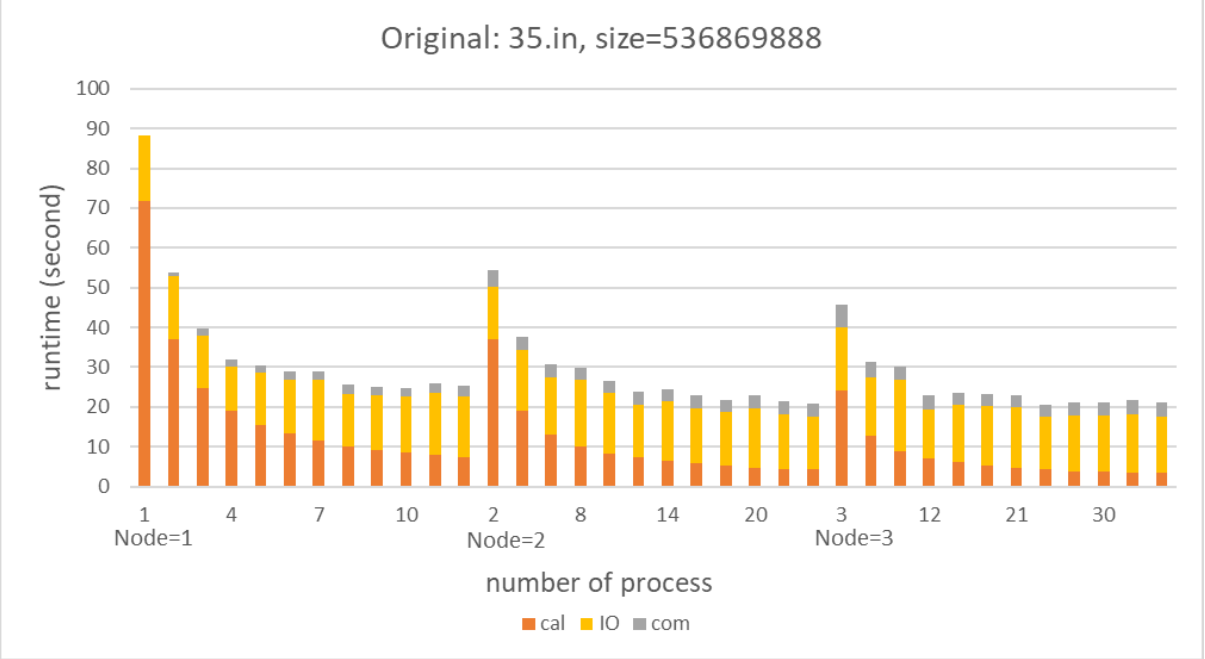
      以下的實驗將由Node數量從1~3(-N=1~3)，total process數量從1~36，core_per_process數量為1(defualt)的狀況去觀察。單個Node上不access超過12個process。

2. Performance Measurement and Analysis of Results

首先可以看到Node=1及Process=1的狀況下，不論是Original Version還是Optimal Version的**Bottleneck都會是calculation time(CPU time)**。此外，可以看到，在未平行的版本下改動像是local sorting演算法及其他減少if-else判斷、memory allocation等的撰寫方式是可以降低計算時間的。

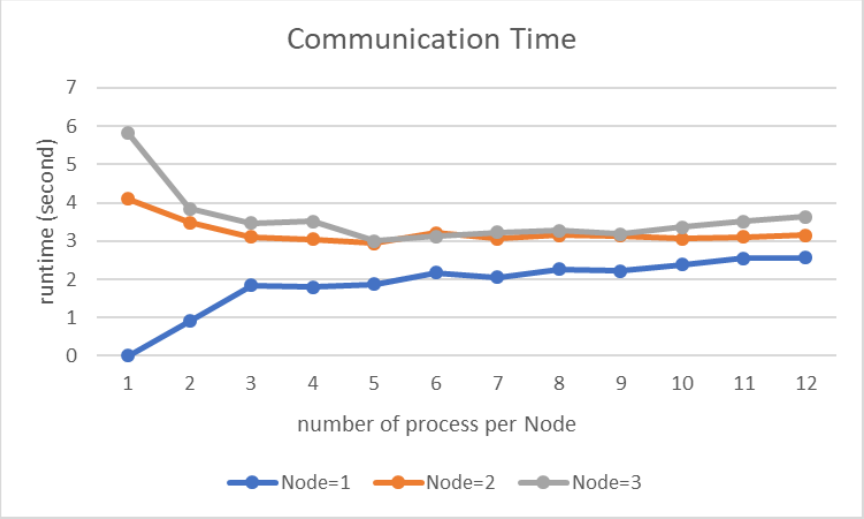


Original Version (Basic):



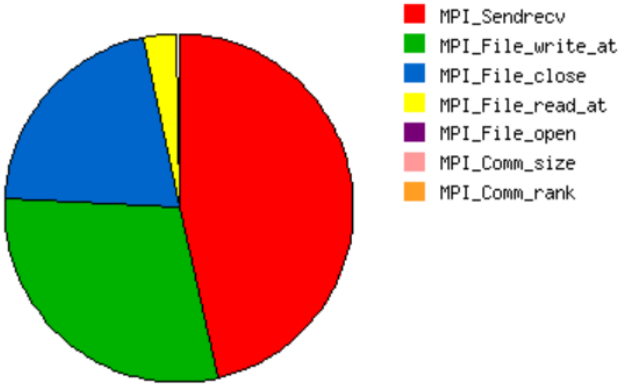
在上圖中可以看到，在固定Node數量下，增加process數量能讓總運行時間有顯著的下降，但是隨著每個Node上process越來越多(>6之後)，加速的效果就開始趨於平緩。接著仔細觀察運算的時間組成:

- Calculation time (橘) 隨著總process的增加而有顯著的下降，因為總process數量增加，單個process所處理的data數量就會變少。
- IO time (黃) 沒有因為process數量或是Node數量而有甚麼太大的改變。
- Communication time (灰) 可以觀察更詳細的圖:



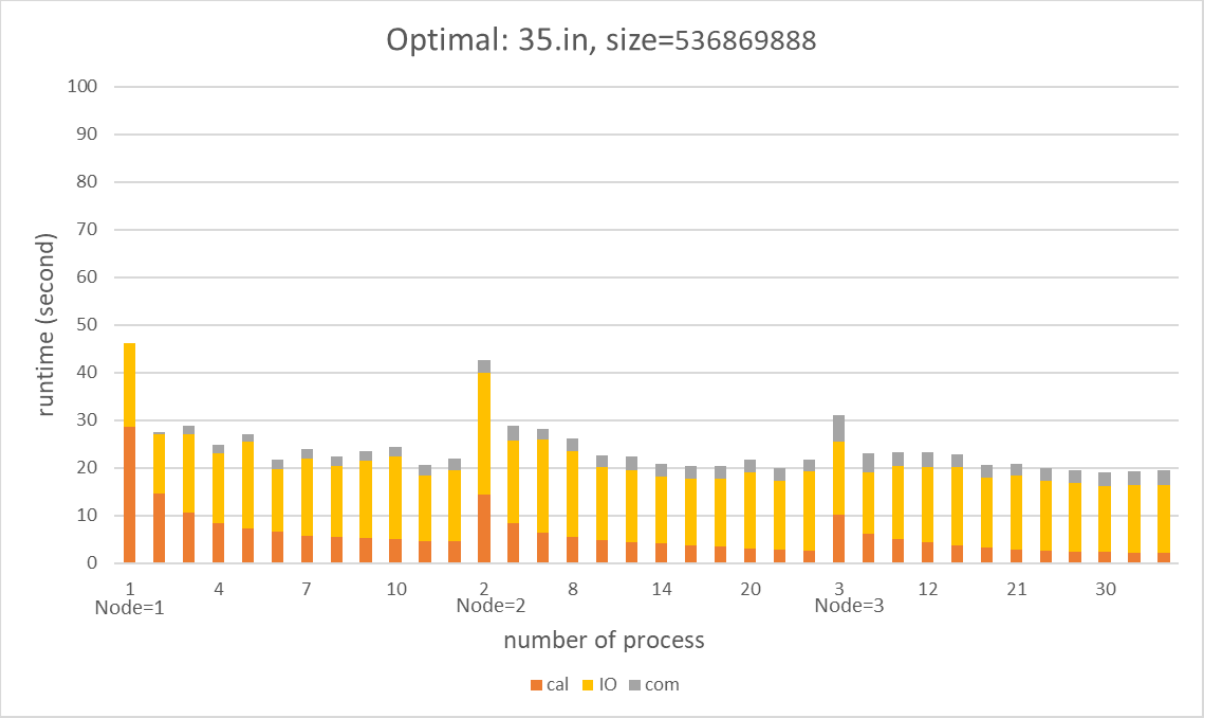
在Node=1時Communication time會比Node=2,3的時候還要少，是因為同個Node中的process傳輸資料會比起跨Node傳輸資料還要快，並且在Node=1時隨著Process數量上升Communication time 會上升，是因為需要耗費傳輸成本將data分給各個不同的process。另外，在Node=2,3時可以看到當每個Node的process數量上升時，Communication time會下降較快，這是因為要傳輸的資料變少了，但隨著process增多(number of process per Node > 3)，傳輸資料的時間成本會趨緩，推測可能是因為傳輸資料減少幅度不夠大導致Communication time下降幅度不高。

此外，使用Profiler觀察也會發現在MPI相關指令中，IO時間甚至超過一半，如下圖所示



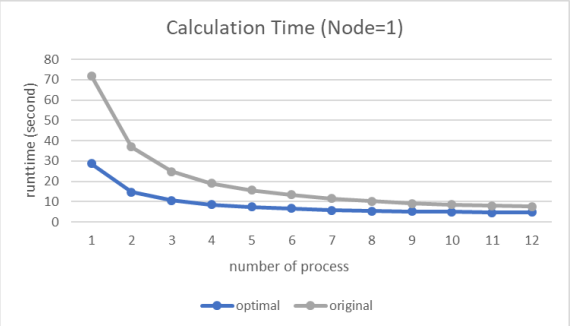
綜合上述分析，**Bottleneck為IO time**，因為當process數量上升，calculation time跟 communication time都能有顯著的改善，即使process數量上升，先做完的人可以先write，但是IO bandwidth有限，到最後就會是IO無法加速。

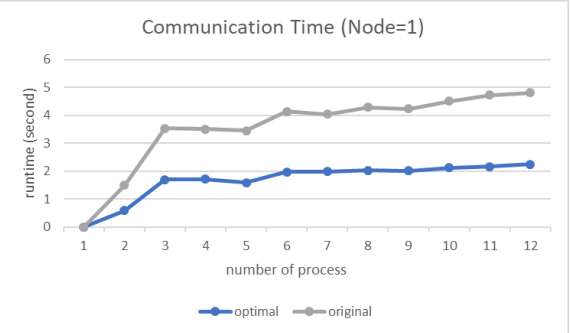
Optimal Version (Advance):

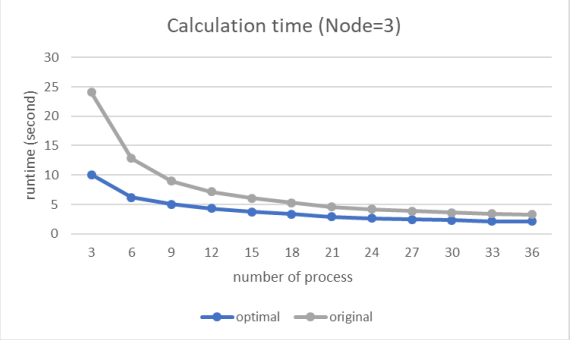


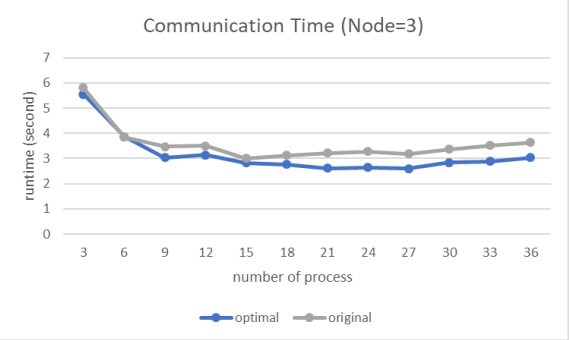
總執行時間看起來和Original版本的相較之下降低非常多，不過以整體趨勢(也就是隨著process增加)來看，仍有Calculation time降低漸緩及IO time沒什麼改變的現象，而觀察該圖及 Profiler 也可以發現**Bottleneck一樣是IO time**。我們接著可以仔細觀察運算的時間組成的改變:

可以看到Original Version的calculation和communication time都比Optimal Version還高。而這裡在Node=1時Optimal能比Original的communication time少更多，推測是因為我在Optimal Version中加入了檢查是否需要交換大量資料的code，因此在Node=1時的process數量比Node=3時process數量少，故每個process分到的data更多，有了先行檢查則可以避免傳輸大量資料，進而減少更多communication time。

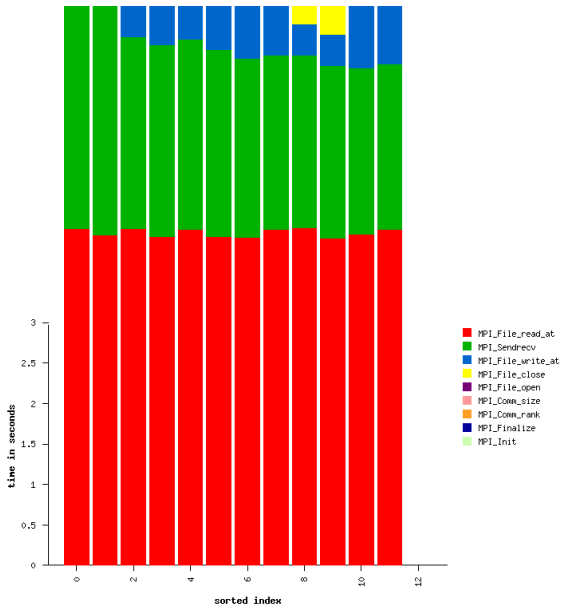


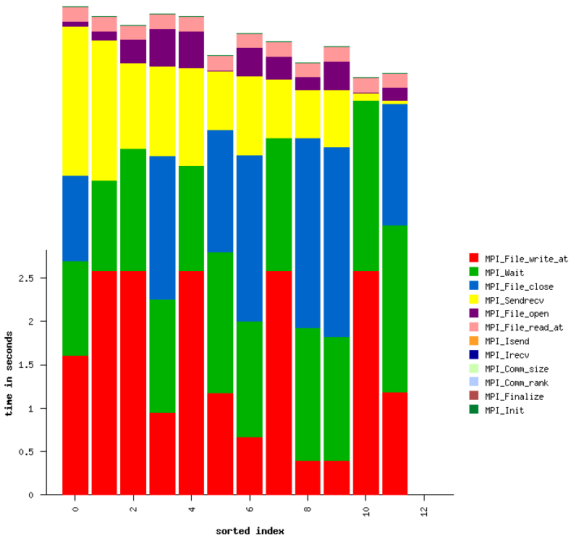






而在Optimal Version中我測試了使用Blocking的方式`MPI_Sendrecv()`和Non-blocking的方式`MPI_Isend()`，`MPI_Irecv()`傳輸資料，使用IPM profiler觀察MPI相關指令的耗時及Load Balance，發現Blocking的方式會Load Balance會相較Non-blocking更balance、更穩定一些(幅度不會這麼大)。

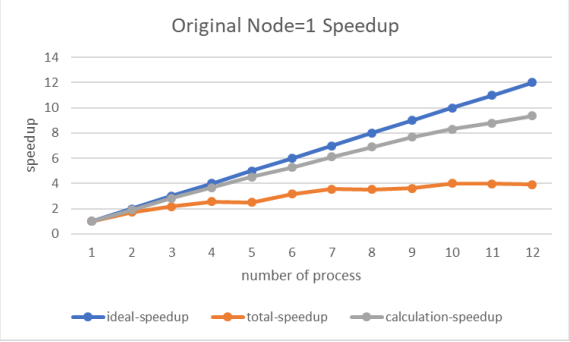


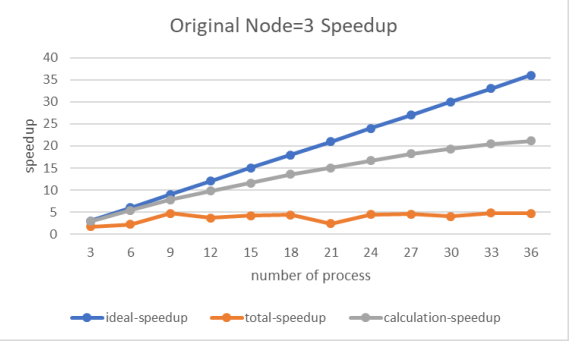


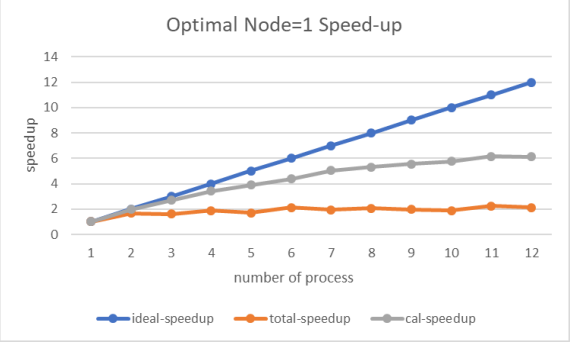
在hw1-judge上測試，雖然`MPI_ISend()`能跑出最佳的成績，但平均起來是`MPI_Sendrecv()`更穩定，最後還是選擇Blocking版本繳交。

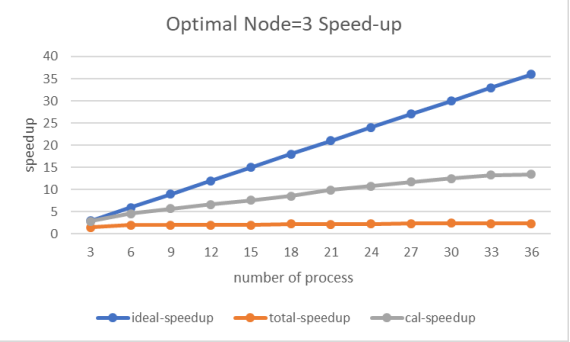
hw1-optimal	SendRecv	IsendIrecv
1	115.93	117.32
2	118.75	126.87
3	123.92	161.41
4	117.1	143.57
5	119.1	113.4
Average	118.96	132.51

Speedup Factor:









Scalability部分可以從上圖中看到Original Version跟Optimal Version的整體speedup會隨著process數量增加而上升，其中total-speedup效果雖然不及ideal-speedup，卻也有隨著process數量增加而上升，而這裡將calculation-speedup分開看的話會發現，其實隨著Scalability變大，加速的效果會比total-speedup更顯著，推測total-speedup的的上升沒有這麼明顯是因為IO time不太穩定，`MPI_write()`時常忽快忽慢，以及網路問題導致`MPI_Isend()`、`MPI_IRecv()`和`MPI_Wait()`的時間不穩定。此外，可以看到Original Version的calculation-speedup會更趨近於ideal-speedup，推測是因為最原本的sequential case做的越爛，計算speedup的分子就越大，後續因為平行而帶來的加速

效益就會比較明顯。像是在36個process時，Original Version需要3.33 sec的calculation時間，而Optimal Version只需要2.12 sec的calculation時間，由此可以觀察Optimal Version可以發現在process數量越加上升的狀況下還是做得比Original Version更好的，在這個狀況下也加速到14倍。然而隨著process數量上升到更大，不論是calculation-speedup還是total-speedup的效果都會趨緩。

Experiences / Conclusion

這次的平程作業真的學到超多，因為我不太熟悉MPI，所以花了一些時間思考該如何切割data、傳送資料等等的，等到第一版Odd-Even Sort寫出來後開始想優化的方式，才意識到除了思考那裡的工作可以平行加速之外，code的架構跟撰寫方式也會影響整體效能的，以及剪枝的一些優化操作等等，避免大量和不穩定Communication Time。此外，除了對程式邏輯的撰寫，我也因為這次的作業才了解到許多Profiler工具該如何使用，並透過這些工具來思考如何做實驗來檢查bolttleneck及驗證scalability，而對於每個實驗結果也需要去分析原因及可能優化的空間，這次的作業讓我練習許多分析的技能，並釐清自己對於平行程式的問題。在做作業時我遇到蠻多問題，像是有時候做了新的改變，但是在有些case的情況下會error，這種時候就非常難debug，目前我只會使用print觀察程式的運算，但是因為有平行所以output出來的東西也不見得有照著順序，就只能在下次實作之前先想清楚每個細節再下手，訓練自己從implement架構的劃分到實作的整個流程。

原本在跑實驗之前我想了一些方法要優化，像是除了拿左右邊的頭尾比較大小之外，可以直接將拿到的數字在自己的 `localDataBuf` 中做Binary Search，往交換Process傳那些需要的數字就好，這樣可以大降低Communication Time，但後來做實驗後才發現Communication Time好像也沒有糟到成為bottleneck的狀況，所以這個加速的idea也就暫時被擱置在一旁了(加上有bug沒有找到QQ所以沒有成功寫出來)。而實驗部分，對於IO time我真的蠻困惑的，server的IO似乎不是很穩定，有時候測量同樣的testcase跟code跑出來的數字可以差10幾秒，而跑實驗出來才發現ideal speedup真的不是一件容易的事，甚至會差距非常遠。希望作業結束後，能有機會聽到那些排名前三的同學分享優化的方法(老師或助教能鼓勵他們分享之類的)，我想多了解是否有甚麼優化手段，也希望之後能繼續進步。

非常感謝助教及教授在這停電的幾個星期中協助處理server及作業問題並延期作業deadline。