PP23_2023_HW1_ODD_EVEN_SORT_IN_MPI

by 吳泊諭, 112062585
hackmd link (https://hackmd.io/@sBeNJ4fqRNqa67PhyWWV4A/BJr_R7KWp)

Implementation:

名詞解釋
size: total process, n: 要sort的總長, rank: 第幾號process, data: local process array, tmp: communication buffer, buf: buffer for merge

我將實作區分為odd-even sort的over view、I/O、計算三個部分討論

odd-even sort的over view

我的odd-even sort主要想法如下:

- 先分配資料並且用 `MPI_File_read_at` 分配資料至每一個process的data
  - 若資料過小，例如size>n的情況，則直接將*所有資料*放到rank0的process，這樣可以避免每個process只能分到0-1個float的窘境。然後直接sort並存起來，接著early return。
- 對於每一個process的data都做initial sort
- 生成一個for loop，其iterate的次數是ceil(size/2)
  - 每次iteration都做下列的事情
  - 先來even phase，依照當前rank，分左右兩邊
    - 使用 `#define EVEN_PHASE 123` 作為溝通(`Sendrecv`)的tag，使其不會被odd phase傳送來的東西干擾到。
    - 必定會有小溝通。 `MPI_Sendrecv` 做抽樣，傳一個東西過去，將樣本放在tmp中
    - 對比tmp的樣本以及自己的data，條件性，依照需求作較大的溝通 `MPI_Sendrecv` ，傳送全部的data到隔壁process並且將其傳送來的東西放在tmp

        - 跑merge two sorted list
    - 再來odd phase
        - 使用 `#define ODD_PPASE 456` 作為溝通(`Sendrecv`)的tag，使其不會被even phase傳送來的東西干擾到。
        - 必定會有小溝通。 `MPI_Sendrecv` 做抽樣，傳一個東西過去，將樣本放在tmp中
        - 對比tmp的樣本以及自己的data，條件性，依照需求作較大的溝通 `MPI_Sendrecv` ，傳送全部的data到隔壁process並且將其傳送來的東西放在tmp
        - 跑merge two sorted list

> iterate的次數是ceil(size/2)的理由是我將odd-even sort的odd phase以及even phase全部展開為一次iteration。每一次iteration我都會跑even-phase以及odd-phase，是故我可以用ceil(size/2)次的iteration將最左邊的process資料移動到最右邊的資料。

- 將每個process的data用 `MPI_File_write_at` 寫入

I/O

I/O有兩個部分:資料分配以及溝通協定

資料分配

- 資料分配決定一開始每個process會拿到多少/自哪邊讀，以及最後在寫入時，分別要在寫多少/寫入哪邊。我的想法是要盡量的load balance，所以期望每個node可以均分要sort的東西，我要用n/size(我用n/size而不是n/(size-1)是因為我沒有要讓第0號process作為communication head)。那這就引出了一個問題，我該用ceiling還是floor?
    - `ceiling(n/size)` :overhead是每個process有可能會多處理一個float，而最後一個process可能拿到的東西會很少，會有很多空位(n<=size*ceil(n/size))。我認為這些多的空位可以直接用 `std::numeric_limits<float>::max()` 做填補，這樣我在做sorting的時候可以把這些人為加入的東西停留在最後一個process，使其不影響到其他process。這樣的作法會使每一個process都allocate依樣大的array，使整體code的整潔度上升。

    - `floor(n/size)` :好處是每個process有可能會少處理一個float，而Overhead是我要多handle最後剩下的 `(n-size*floor(n/size))` 。比較致命的是最後多出來的這個剩餘沒有allocate到process，我是必要多花一個Node來處理，這會讓整份code寫起來很雜這會讓整份code寫起來很雜亂。
    - 我經過以上考量，決定用ceiling(n/size)。
- 是故就從disk讀資料入Memory而言，如下所示。其中， `each_hold` 是每一個process均要allocate於記憶體的float數量。 `last_hold` 是最後一個process所持，真正有意義之資料個數，所以只讀入 `last_hold` 的數量，並在後面補滿 `std::numeric_limits<float>::max()`

```
int each_hold, last_hold;
int each_hold = std::ceil((n+size-1)/size);
float* data = new float[each_hold];
float* tmp = new float[each_hold];
float* buffer = new float[each_hold];
...
if((n%each_hold)==0) last_hold = each_hold;
else last_hold = n % each_hold;
if(rank!=0)
    MPI_File_read_at(input_file, sizeof(float) * rank * each_hold, data, eac
else{
    MPI_File_read_at(input_file, sizeof(float) * rank * each_hold, data, las
    // the rest of the data should be float max
    for(int i=last_hold;i<each_hold;i++) data[i] = std::numeric_limits<float
}
...
```

- 至於寫回則是根據目前rank，若是最後一個process，只寫 `last_hold` 的大小，其餘則寫 `each_hold` 的大小。

```
if(rank!=size-1)
    MPI_File_write_at(output_file, sizeof(float) * rank * each_hold, data, e
else
    MPI_File_write_at(output_file, sizeof(float) * rank * each_hold, data,
```

通訊協定

- 我在時做odd even sort的方式會採樣相鄰兩個process的極值，我會去比較左邊那個process的max以及右邊process的min誰比較大?
- 假如是左邊的最大比右邊的最小還小，考量到local process array裡面的東西都已經是monotonic increasing的狀態(每一輪iteration都會有一個**Merge**、一開始分配資料到local process array後就馬上會有初始的**sort**)，這代表左邊local process array的全部東西＜右邊local

process array的東西。是故，此時不需要額外做任何通訊協定。

- 倘若上述提問不成立，代表某些左邊local process array的東西應該要被移動到右邊去。因為不知道要移動多少個，是故每次船都是傳固定量的東西，而那個固定量即為 `each_hold` 。

以even phase為例，[`p_even` , `p_odd`]是相鄰的兩個process。
先做一個MPI_Sendrecv到相鄰的process

```
MPI_Sendrecv(   data+each_hold-1, SINGLE_ELEMENT, MPI_FLOAT,
                                  rank+1, EVEN_PHASE,
                                  tmp, SINGLE_ELEMENT, MPI_FLOAT,
                                  rank+1, EVEN_PHASE, MPI_COMM_WORLD, MPI_STA
```

詢問左邊的max和右邊的min誰大?即為data[each_hold-1]和tmp[0]比大小。data[each_hold-1]則傳輸全部的data到隔壁process，並開始做merge，否則不做傳遞。以此免去過度的資料傳輸。

```
if(data[each_hold-1] > tmp[0]){
                    // start large communication
                    MPI_Sendrecv(   data, each_hold, MPI_FLOAT,
                                    rank+1, EVEN_PHASE,
                                    tmp, each_hold, MPI_FLOAT,
                                    rank+1, EVEN_PHASE, MPI_COMM_WORLD, MPI_

                    distribute_portion2(data, tmp, buf, each_hold, true);
                }
```

- 這邊有一個觀察:我用ipm，其中他在 `MPI_Sendrecv` 那邊的時間量測，我觀察到large和small communication其實總耗費時間相去頂多一成，且兩者被呼叫的次數其實差不多。那這代表apollo在執行MPI的情況下，其實坐大通訊還是做小通訊都"差不多貴"。這代表我的是著只拿左邊max和右邊min的行為，其實很沒有省道多少東西，反倒是多浪費一次建立溝通的時間。所以，我便直接做了一個大通訊，然後條件性執行distribute_portion2(data, tmp, but, each_hold, true)，這個改動讓我的跑分到達122秒。改動如下:

```
// start large communication
MPI_Sendrecv(   data, each_hold, MPI_FLOAT,
                rank+1, EVEN_PHASE,
                tmp, each_hold, MPI_FLOAT,
                rank+1, EVEN_PHASE, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if(data[each_hold-1] > tmp[0])
    distribute_portion2(data, tmp, buf, each_hold, true);
```

計算

在前面的章節我有談論到，我會在一開始做sort以及在每一次iteration都會執行的merge，這兩個東西主要吃CPU的效能。

- 關於initial sort的部分，我原本是用 `qsort` ，但是後來我去參考網路上的資料，我發現他們
  用 `boost::sort::speardsort` ，我嘗試了一下，果然效能馬上提升。

```
// odd sorting approach
// qsort(data, each_hold, sizeof(float), compare);
// new sorting approach
std::boost::sort::speardsort(data, data+each_hold)
```

- 至於merge的部分，我實做了 `distribute_portion2(float* data, float* tmp,float* buf, int each_hold, bool take_small_portion)` 我是參考leetcode的**merge-two-sorted-lists** (https://leetcode.com/problems/merge-two-sorted-lists/)，其複雜度為O(n)。Overhead是多額外生成了一條長度為 `each_hold` 的array。我對於這個overhead的處理方式是盡量減少這條array的影響。由於他主要的影響是new的次數，所以我的目的是要減少new的次數。因此我做了一個變通，我不在function裡面生成array，取而代之的是直接在main中生成，並是對這個function額外多接了一個parameter `buf` 來傳入 `buf` 的指標，以重複利用 `buffer` ，便可以減少 `new` 的次數。
  - 創造一個 `buf` (長度為 `each_hold` 的float array)來當作暫存
  - 創造兩個curser，一個對當下該看 `tmp` 的哪邊負責，另一個對當下該看 `data` 的哪邊負責，依照當下process是在哪個半邊做選取。並在選完之後更新curser位置。
    - 左半邊:每次都拿 `tmp` 和 `data` 中較小的那一個
    - 右半邊:每次都拿 `tmp` 和 `data` 中較大的那一個
  - 當現在 `buf` 已經裝滿新的東西，便將 `buffer` 中的資料複寫到 `data` 上

Experime & Analysis

Methodology

- System spec: 我是使用助教提供的apollo作為環境
- Performance metrics: 至於I/O等測量則是用dynamic linking IPM處理。各function的時間量測我是用 `MPI_Wait()` ，將我要測量的東西夾起來，然後算兩次呼叫 `MPI_Wtime()` 的數值差。Speed則是直接用 `1/t` 計算。

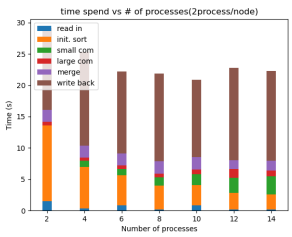
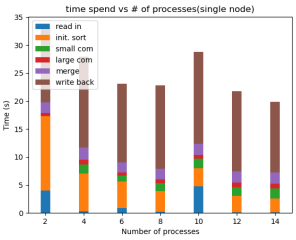
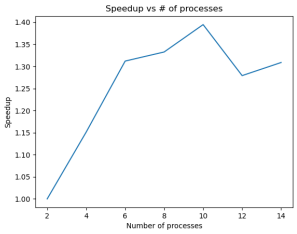
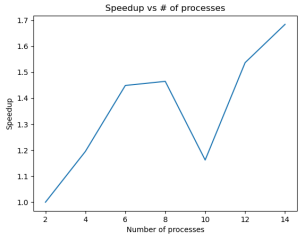
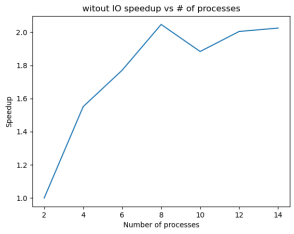
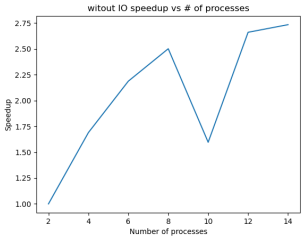
```
double start_time, end_time;
...
start_time = MPI_Wtime();
distribute_portion( data, tmp, but, each_hold, true);
end_time = MPI_Wtime();
printf("%lf", end_time - start_time);
```

- 然後我有寫一個 `run.sh` 來跑資料

```
mode='single'
for i in {1..7}
do
    j=`expr $i \* 2`
    if[mode!='single']do
        srun -pjudge -N$i -n$j ./hw1 536831999 /home/pp23/share/hw1/testcase
    done else do
        srun -pjudge -N1 -n$j ./hw1 536831999 /home/pp23/share/hw1/testcases
    done
done
```

Plots: Speedup Factor & Profile & analysis result

- Experimental method: 我適用case38作為分析主角，因為他的n很大(n=536831999)。

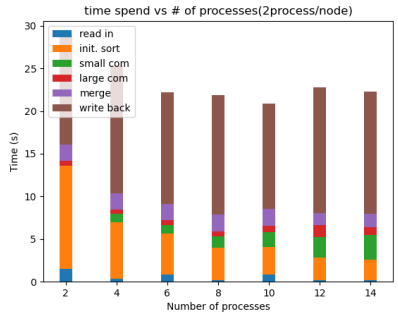
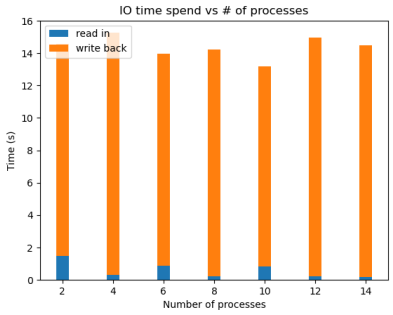
node usage	multi node	single node
total time		
total speed up		
without IO speed up		

- multinode:我會從1個node一路測到7個node(每一個node會領到兩個process，共使用使用2-14個process)。他花費的時間先是下降，然後再慢慢上升。然後我在依照process=2(只有一個node)作為基準，做出speed up圖。可以觀察到速度逐漸隨著process的scale上升而上升，當process=10時，達到最快速度，之後便開始往下減速。他的scalibility會先上升，接著慢慢勘始下修。

- single node:只使用一個node。花費時間長度如下所示，他花費的時間在>=6個process之後就基本上持平了。主要的時間花費都在write back部分，其原因請詳見multinode的buttleneck解釋。他的scalibility會有一個山谷，我對於這個奇怪的山谷，我的理解是我在跑n=10的時候，我分配到的那個node在memory中找不到 `38.txt` 、 `38.in` 。這始得我需要去disk裡面抓，令我正在memory的導致我在做I/O時，花了比平常更多的時間。

BUTTLENECK討論-MULTINODE

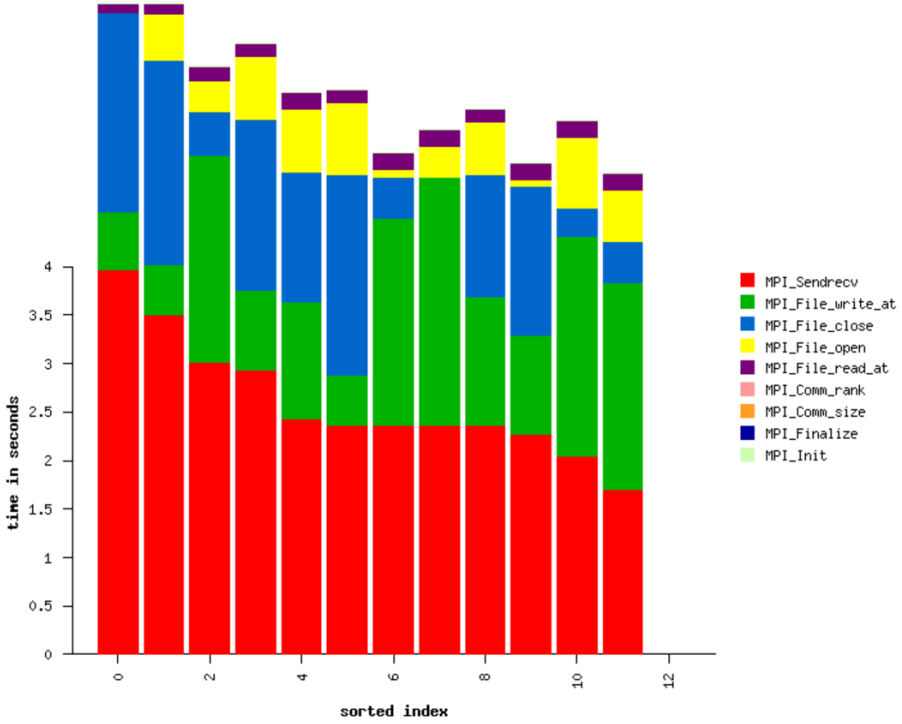
- 要處理buttleneck的問題，我先觀察了一下我的柱狀圖。我發現咖啡色的write都異常粗大(詳見下圖1)且基本沒有隨著我的node上升而上升(詳見下圖2)。

total time	only I/O
 1	 2

- buttleneck在write back的階段。我無論將node的數量提升多少，我都沒有辦法降低這個數值。考量到我在測量數字的時候，使用的command如下所示。 `srun -pjudge -N$i -n$j ./hw1 536831999 /home/pp23/share/hw1/testcases/38.in 38.out done` 。
- 在這種呼叫之下，我用judge partittion和home目錄下的那個disk是不同的disk。因為不在同一張disk之上，要去別的disk讀取大量資料會花費大量時間，所以每一個process才會拿出這麼大量的時間跑write。所以解決辦法便是使用助教提供的judge unit test + static mode處理。

  Home 目錄和 judge 時所用的 disk 不同因此速度有很大的差異，如果想要觀察 MPI IO 的 performance 可以使用前面提到的 judge unit test + IPM static mode 來達成效果。

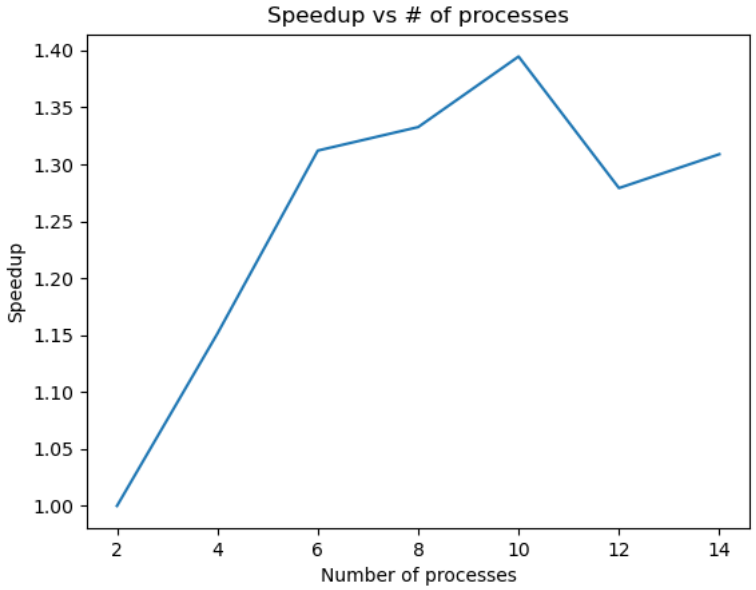
  。
- 它的效果立竿見影，經過改善之後 `MPI_File_write_at` 花費的時間減少為2秒左右。



- 此外，我還有思考過改動呼叫的MPI，例如改為 `MPI_Write_at_all` 等，但都效果不彰。

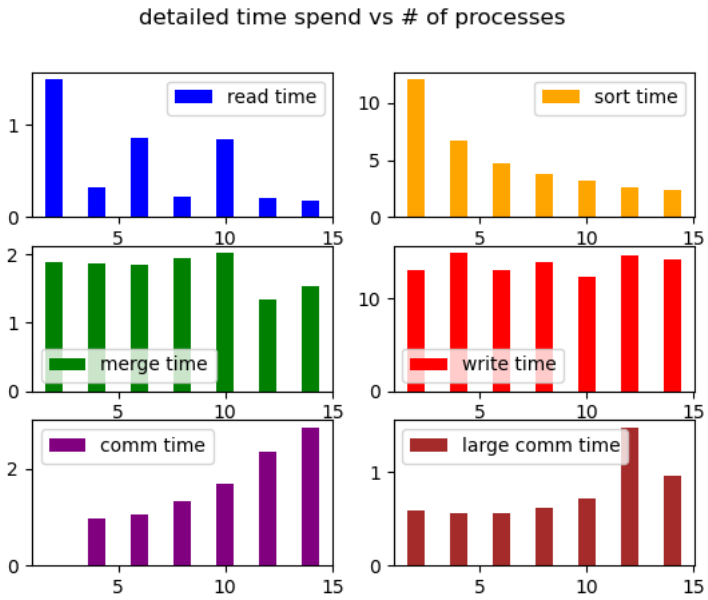
SCALIBILITY 討論-MULTINODE

- 從speedup圖可以看到在node=10的時候有明顯下垂。

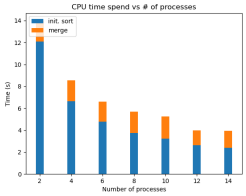
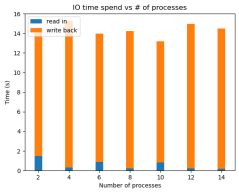
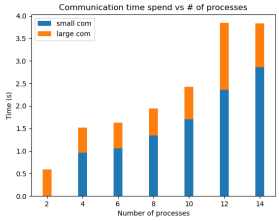


- 我要探討的是為什麼會有這個scalibility曲線彎折?為了解決這個問題，我將每種花費都詳細列出來。其中有一開始讀入的時間、初始sort的時間、merge花費的時間、小communication花費的時間、大communication花費的時

間、write花費的時間。

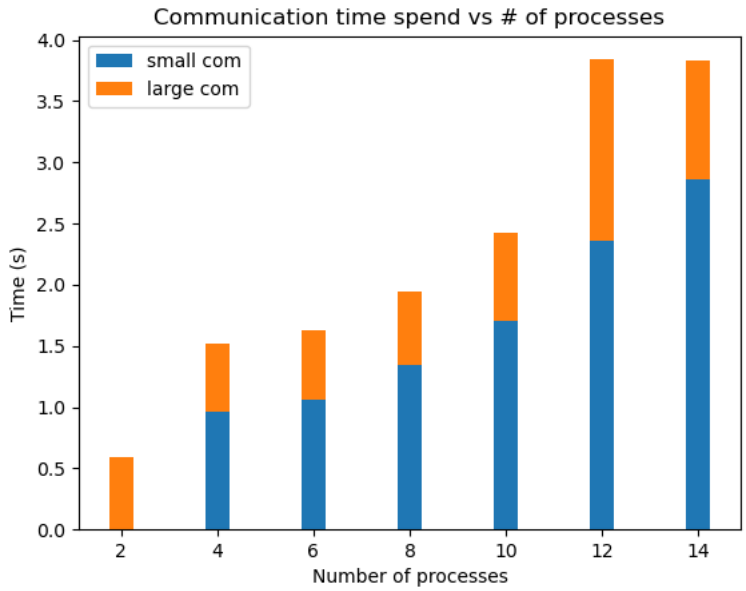


- 他們用 CPU, I/O, Network分類的話，可以分為以下幾類:
  - CPU: 初始sort+merge
  - I/O: read+write
  - Network: large communication+small communication

CPU: intital sort+merge	I/O: read+write	Network: large+small communication
		

- 去除已經討論過得I/O，現在便是CPU和Network會影響整體跑分。
  - 首先討論Network的部分，先說結論:MPI在通訊上有很大的overhead，Network的時間成本會因為node的增加而大幅上升。以這次要實作的odd-even sort為例，下圖顯示通訊成本隨著#process的上升而上升，

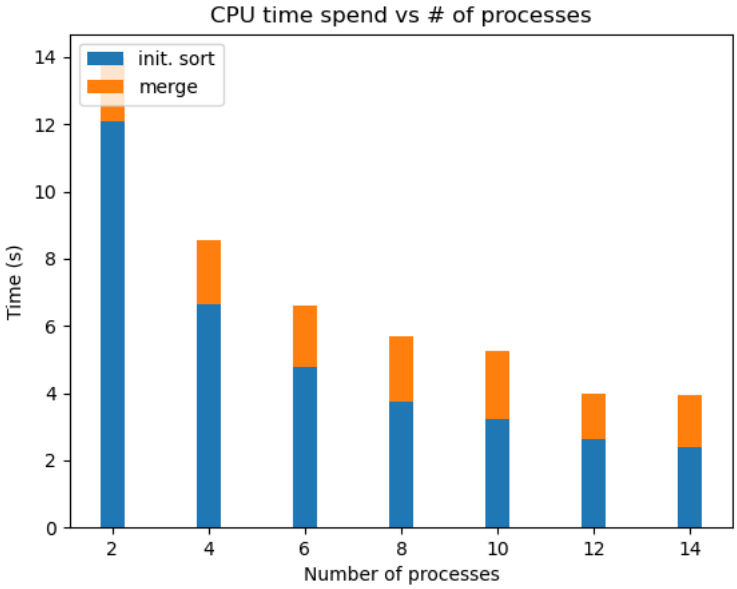
逐漸地從0.5秒上升到4秒。



  - 通訊主要做兩件事:普通的ACK/NACK訊號接收以及local process array的傳輸。前者像是低消，每一次通訊都要花。後者像是加點內容，價格依據要傳輸的local process array長度變動。
  - 考量到#process上升的情況下，每個process需要扛到的array長度理應下降，所以每一個process在傳輸自己的array時，需要耗費的時間理論上應該是變短了。表示每個process與其他process在溝通得時候，傳輸量的影響逐漸的下降了。那位什麼通訊時間花費還是越來越高呢?
  - 我認為，因為我是實作odd even sort，隨著#process線性上升，原先長度為n的資料會被越切越細碎，使的worst case情況下，最左邊的process會合最右邊的process越離越遠，是故要完整從最左邊到最右邊要走的path會越來越長。因為我沒有將root的process作為總管，是故我沒有辦法用early stop(總管蒐集其餘process看有沒有改動local process array，若大家都沒有，則直接停止for loop)的形式把for loop中止。此即代表，for loop的iterate次數會線性上升。

- 考量到總通訊時間是(普通的ACK/NACK訊號接收+local process array的傳輸)*總通訊次數，即便local process array的傳輸下降了，但是總通訊次數上升，反倒使通訊花費的時間變大了。

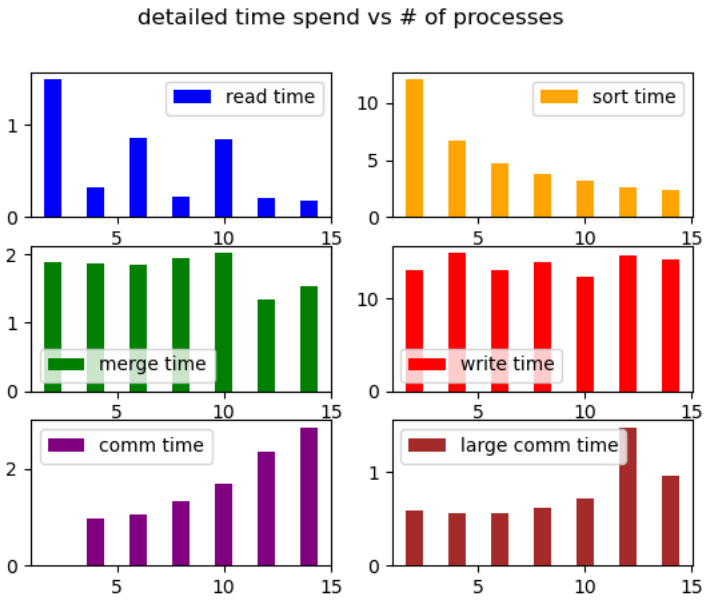
- 接著再去衡量CPU的部分，同樣先說結論:CPU的花費越來越低，node的增加會使CPU的時間花費越來越少。CPU主要是負責計算的部分，其中有initial sort以及merge。



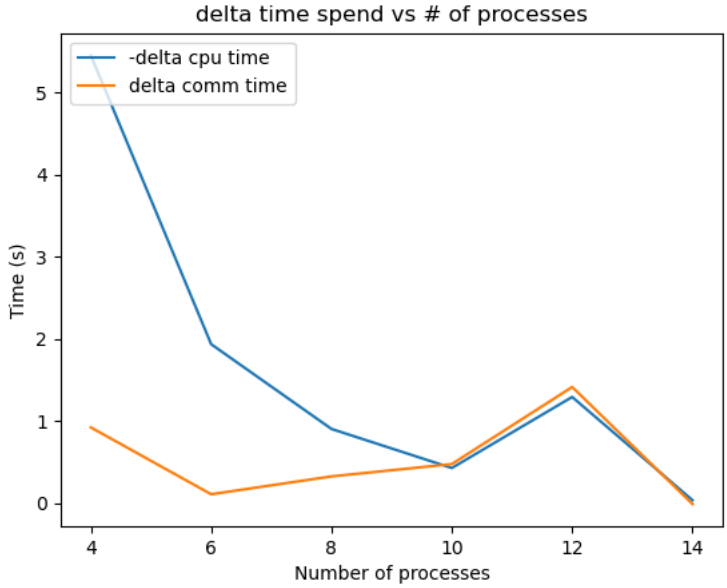
- 我是用 `boost::sort::speardsort` 作為initial的sort處理。考量到我在node上升的情況下，每個local process分配到的data array會越來越小，是故他要花費sort的時間會越來越短。

- 至於我每次做完通訊之後都要做merge的行為，如果將node的數量拉到和n依樣大，變可以視作bubble sort(1個node只有一格float)，此時initial sort的影響會會降到最低，整體data的移動只有靠merge一步步的搬移資料。
  - 如果將node的數量逐漸下降，initial sort的影響會漸大，但也只有限縮在初始那個node裡面會享受到sort的加速，其餘時間在別的node傳遞得時候都始要靠merge一格一格移動，所以可以說我在做odd even sort的時候，float的移動大部分還是靠merge。detailed time spend的merge time很明顯的表現出了

merge的時間花費相當的固定，約在1.5-2秒之間。



- 是故，考量到CPU=initial sort+merge，merge的時間花費基本不變動，而initia sort的時間花費則是隨著node的提升而下降。我可以宣稱CPU的時間花費隨著node的提升而降低。我衡量兩者的時間花費，我可以觀察到在process=10,node=5前後(process=8,node=4&process=12,node=6)，CPU time的時間花費大概都是省1秒，但是Network的時間花費大概都是貴2秒。**這表示Network造成的communication額外負擔大過增加#node的紅利**。而我想這也是為什麼在 `38.txt` ，助教是用 `nProc=12` 撰寫，因為此時增加一個node的communication overhead吃掉了其所的CPU紅利。



BUTTLENECK討論-SINGLE NODE

我認為single node和multipule node的overhead，都是在write的部分。其原因我想和multinode差不多，於此不多做贅述。

SCALIBILITY討論-SINGLE NODE

我在single node的效能大概提升到1.5倍之後，就沒有明顯上升。我覺得這是因為我拿來跑process的那個node已經滿了，故速度上不去。

Optimization strategies

我有使用到 `boost::sort::speardsort` 加速，替換原本的 `std::algorithm::qsort` 以及 `std::algorithm::sort` 。
就以溝通而言，我有嘗試使用 `MPI_Send` 配合 `MPMI_Recv` ，但是我後來發現 `MPI_Sendrecv` 最快。
原先我是將MP_Sendrecv統一做我有將MP_Sendrecv切為large communication以及short communication。我這樣做是為了減少在communication的overhead，避免過度的資料傳遞。但經過我的測試之後，就發現單純做communication的overhead很大，便改成只做large communication，並條件性的作merge。
我有嘗試過 `MPI_Write_at_all` 以及 `MPI_Write_at` 測試並嘗試改善寫入速度。由於這次要寫入的位置沒有重疊，所以我選用 `MPI_Write_at` 來做I/O。

Experiences/Conclusion

我在這次實驗中學到很多profiling的方法，我覺得很新奇。此外，我發現在執行 `hw1-judge` 的時候會友跑分漂移的情況，大概會有18秒左右的漂移。

參考

我這次MPI的實作是有參考幾個網站，他們給我的幫助，我十分感激!

- **使用pthread** (https://www.geeksforgeeks.org/odd-even-transposition-sort-brick-sort-using-pthreads/)
- **沒有使用到node 0，在load balence時效果不好** (https://github.com/ashantanu/Odd-Even-Sort-using-MPI)

- **有優化通訊協定，沒有把最後一個node寫得更簡潔**

  (https://github.com/Elven9/NTHU-2020PP-Odd-Even-Sort)

- **一次只有溝通一個東西，communication overhead太大**

  (https://stackoverflow.com/questions/47747287/odd-even-transposition-sort-with-strings-mpi-c)