parallel programming hw1

106062230 徐嘉欣

Implementation

```
MPI_Init(&argc,&argv);
int rank, size;
unsigned long long int data_length, read_pos, rest;
unsigned long long int num = (atoll(argv[1])); // the size of the array
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Request request, request2;

data_length = num/(size);
rest = num%(size);
```

首先先initialize MPI，接著去讀取下的command中，n的數值為多少，將其轉換為long long的格式，最後去計算說平均每個process需要處理多少的數字，分配完後還會剩下多少還沒分配到的數字。

```
unsigned long long int recv_data_length = data_length;

if(rank+1 < size && rank+1 < rest) ++recv_data_length;
if( rank < rest ) {
    ++data_length;
    read_pos = data_length*rank;
}else{
    read_pos = rest + rank*data_length;
}
```

接下來想要去計算自己的array與接收到的array的大小。
其中第三行的 `rank+1 < size` ，意思是若我下一個的process的rank比所有要sort的數字少，代表他不會分配到數字，所以就不需要再去管 `recv_data_length` 。
第四行至第九行則在計算是否有需要多負擔一個數字。

```
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, argv[2], MPI_MODE_RDONLY, MPI_INFO_NULL, &f);
float *data = (float*) malloc(sizeof(float) * data_length);

if(rank<num) {
    MPI_File_read_at(f, sizeof(float) * read_pos, data, data_length, MPI_FLOAT,
                     MPI_STATUS_IGNORE);
    boost::sort::spreadsort::float_sort(data, data+data_length);
}
```

首先先開啟input file，將自己所對應位置的數字吃進來，並且去sort他們。

接下來的code在rank是奇數還是偶數的時候其實差不多，挑奇數的情況解釋。

```
// rank是奇數時執行
char change;
MPI_Isend(data, data_length, MPI_FLOAT, rank_minus_1, 0, MPI_COMM_WORLD, &reque
MPI_Recv(&change, 1, MPI_CHAR, rank_minus_1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNO
if(change)
    MPI_Recv(data, data_length, MPI_FLOAT, rank_minus_1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```

一開始會先send資料給rank-1的process，接著等待rank-1的process回傳是否有改變順序，若有的話才需要接收改變的資料。

```
// rank是奇數時執行
MPI_Recv(data2, recv_data_length, MPI_FLOAT, rank_plus_1, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
float *data_iter=data, *recv_data_iter=data2, *temp_iter=temp;
float *data_end_iter = data+data_length, *recv_end_iter = data2+recv_data_lengt
if( data[data_length-1] <= *recv_data_iter )
    MPI_Isend(&flag, 1, MPI_CHAR, rank_plus_1, 0, MPI_COMM_WORLD, &request);
else if ( data2[recv_data_length-1] <= *data_iter
          && recv_data_length == data_length){
    flag = 1;
    MPI_Isend(&flag, 1, MPI_CHAR, rank_plus_1, 0, MPI_COMM_WORLD, &request);
    data = recv_data_iter;
    data2 = data_iter;
    MPI_Isend(data2, recv_data_length, MPI_FLOAT, rank_plus_1, 0, MPI_COMM_WORL
              &request);
}
else {
    while( data_iter != data_end_iter && recv_data_iter != recv_end_iter){
        if(*data_iter <= *recv_data_iter){
            *temp_iter = *data_iter;
            temp_iter++;
            data_iter++;
        } else{
            *temp_iter = *recv_data_iter;
            temp_iter++;
            recv_data_iter++;
        }
    }
    flag = 1;
    MPI_Isend(&flag, 1, MPI_CHAR, rank_plus_1, 0, MPI_COMM_WORLD, &request);
    while( data_iter != data_end_iter ) {
        *temp_iter = *data_iter;
        temp_iter++;
        data_iter++;
    }
    while( recv_data_iter != recv_end_iter ){
        *temp_iter = *recv_data_iter;
        temp_iter++;
        recv_data_iter++;
    }
    MPI_Isend(temp_data_length, recv_data_length, MPI_FLOAT, rank_plus_1, 0,
              MPI_COMM_WORLD, &request);
    for( data_iter = data, temp_iter = temp ; data_iter != data_end_iter ;
         data_iter++, temp_iter++) *data_iter = *temp_iter;
}
```

再來會接受rank+1的process傳送來的資料，此時會分三種情況：

1. 我的最大的數比別人最小的數字小 -> 不需要再比較
2. 別人最大的數字比我最小的數字小 -> 交換兩個人的數字
3. 先將兩個sorted array merge成一個sorted array，再將前面的部分留著，後面的部分傳回給rank+1。

```
char flag_;
MPI_Allreduce(&flag, &flag_, 1, MPI_CHAR, MPI_BOR, MPI_COMM_WORLD);
if(flag_ == 0) break;
```

接著去reduce全部process的flag值去做bitwise or，若有人為1的話，大家接收到的數值就會是1，代表還沒sort完。

```
MPI_File f2;
MPI_File_open(MPI_COMM_WORLD, argv[3], MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &f2);
MPI_File_write_at(f2, sizeof(float) * read_pos, data, data_length, MPI_FLOAT,
                  MPI_STATUS_IGNORE);
```

sort完後，創立一個output file，並且將數字寫入對應的位置，就大功告成了。

Experience & Analysis

System Spec

使用apollo進行實驗測量。

Methodology: Performance metrics

```
starttime_comm = MPI_Wtime();
MPI_Recv(data2, recv_data_length, MPI_FLOAT, rank_plus_1, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
totalTime_comm += MPI_Wtime() - starttime_comm;
```

在任何要溝通(e.g, MPI_Recv、MPI_Isend、MPI_Allreduce)的地方前後會先取當時的time，並在結束後計算溝通的時間並累加全有溝通的時間，I/O time也是一樣的做法。CPU time則是整個program使用 `srun time -N -n ./hw1` 這樣的方式去測每個process跑的時間，再去扣除I/O、comm的時間。

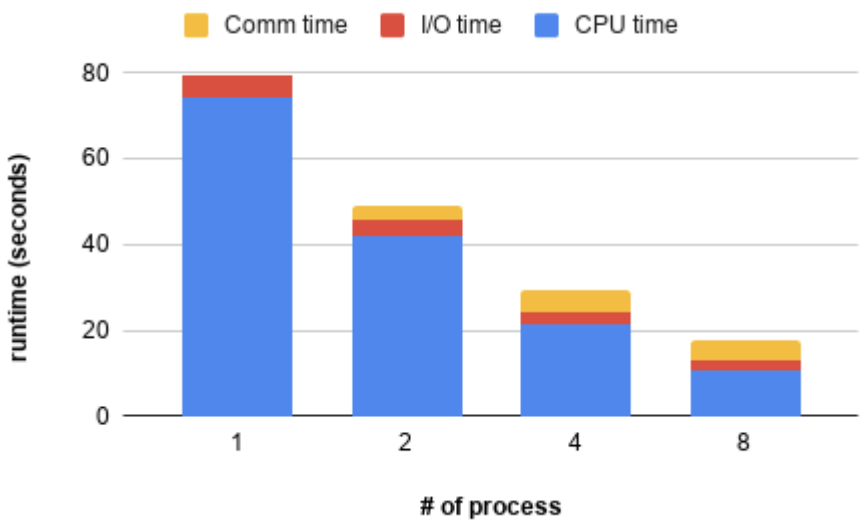
```
====================39====================
[Rank 7] Comm time = 6.332027, I/O time = 1.378913
[Rank 6] Comm time = 3.264407, I/O time = 1.555502
[Rank 2] Comm time = 4.309204, I/O time = 2.278138
[Rank 4] Comm time = 4.311326, I/O time = 2.282457
[Rank 5] Comm time = 5.146581, I/O time = 2.276927
[Rank 1] Comm time = 5.231023, I/O time = 2.199464
[Rank 3] Comm time = 5.487907, I/O time = 2.650469
[Rank 0] Comm time = 3.229001, I/O time = 2.770945
16.18user 0.46system 0:18.50elapsed 89%CPU (0avgtext+0avgdata 332696maxresident)k
0inputs+524296outputs (0major+72168minor)pagefaults 0swaps
15.14user 0.96system 0:18.50elapsed 87%CPU (0avgtext+0avgdata 1428072maxresident)k
0inputs+524296outputs (0major+329129minor)pagefaults 0swaps
14.67user 1.04system 0:18.53elapsed 84%CPU (0avgtext+0avgdata 1315020maxresident)k
2568inputs+524296outputs (0major+291229minor)pagefaults 0swaps
15.04user 1.08system 0:18.53elapsed 87%CPU (0avgtext+0avgdata 1315672maxresident)k
1880inputs+524296outputs (0major+303400minor)pagefaults 0swaps
14.43user 1.18system 0:18.54elapsed 84%CPU (0avgtext+0avgdata 1405036maxresident)k
2384inputs+524288outputs (0major+274444minor)pagefaults 0swaps
```

若是有多個process，則會先單獨計算各個process的CPU time、Comm time、I/O time，再取所有process的算術平均。

Plots: Speedup Factor & Time Profile

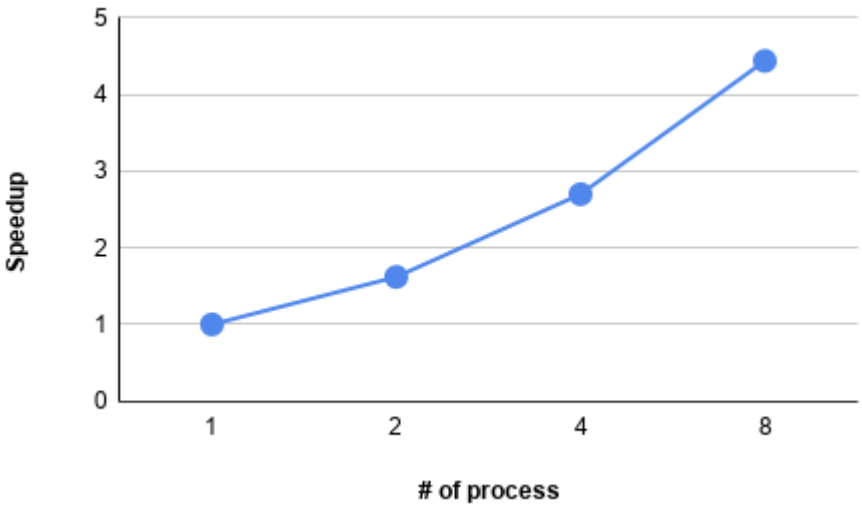
在一個node上跑不同數量的process (run testcase39)

# of process	CPU time	I/O time	Comm time	Total time	Speedup
1	74.04s	5.35s	0s	79.39s	1
2	42.03s	3.81s	3.13s	48.97s	1.62
4	21.63s	2.81s	4.92s	29.36s	2.70
8	10.89s	2.06s	4.94s	17.89s	4.44

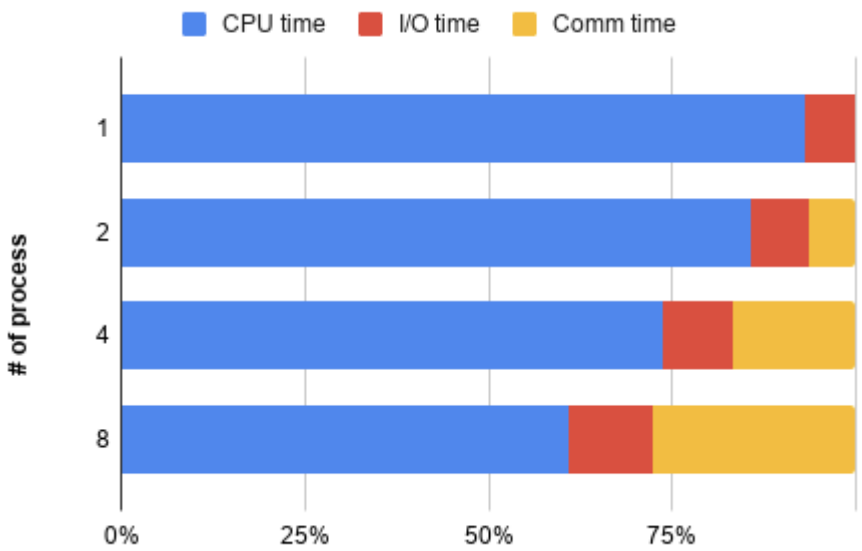


從上圖與表格可以看出以下幾點：

- CPU time與process數量大致上呈現反比，原因就是將工作拆封給各個process去做，理所當然地分給愈多的process，CPU time就愈少
- comm time隨著process數變多而逐漸增加，推測原因有二：
  1. 在worst case中，若最大的數在rank最小的process手中或是最小的數在rank最大的process手中，一共要進行 `#process -1` 次的交換，process愈多，則可能需要更多次的交換與溝通
  2. 在最後確認是否sort完而進行之MPI_Allreduce，因process的數量變多，所以需要等待各個process傳送資料並做operation，導致溝通時間上升
- I/O time雖然隨著process數上升而有所下降，但下降的幅度是愈來愈少，因為有些工作(如開啟檔案)是無法切割的，並且當read/write的size變小時，有時會發生buffer塞不滿的情況，這樣就會使得overhead增高



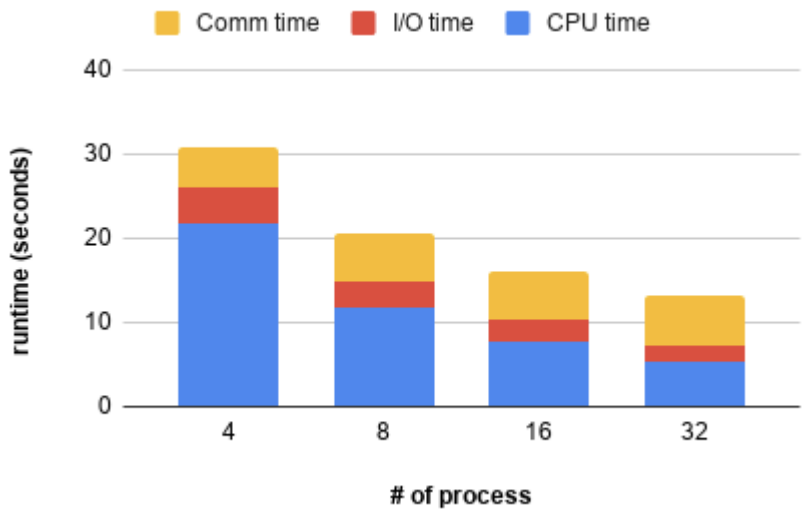
這樣的scalability應該算還不錯，因為每當process數增加兩倍時，speedup大約都是1.6左右。因為有溝通、I/O這些工作，再加上工作切割給各個process，有時還會增加一些多的CPU工作，以至於無法達到linear speedup。



將不同process數所耗費的CPU time、I/O time、Comm time的占比畫出來比較，可以發現在process數還沒有到很多時，CPU time會是bottleneck，但隨著process數增大，comm time的占比也愈來愈大，因此這邊可以推測當process到達一定數量時的bottleneck就會是comm time了。

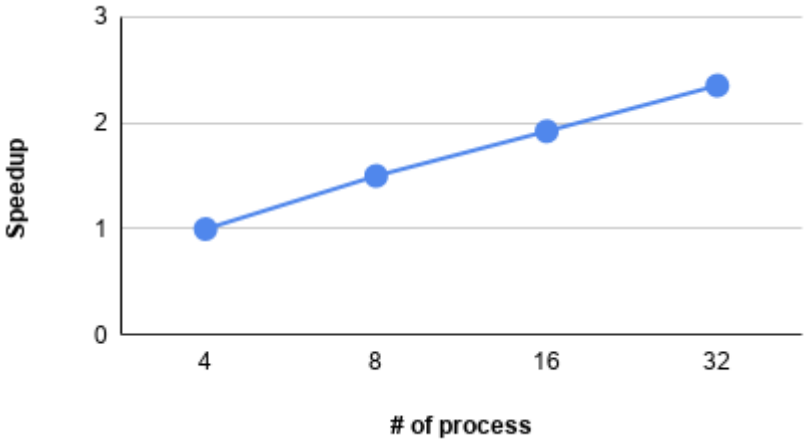
在四個node上跑不同數量的process (run testcase39)

# process	CPU time	I/O time	Comm time	Total time	Speedup
4	21.61s	4.33s	4.90s	30.84s	1
8	11.68s	3.05s	5.84s	20.57s	1.50
16	7.69s	2.61s	5.75s	16.05s	1.92
32	5.32s	1.93s	5.86s	13.11s	2.35

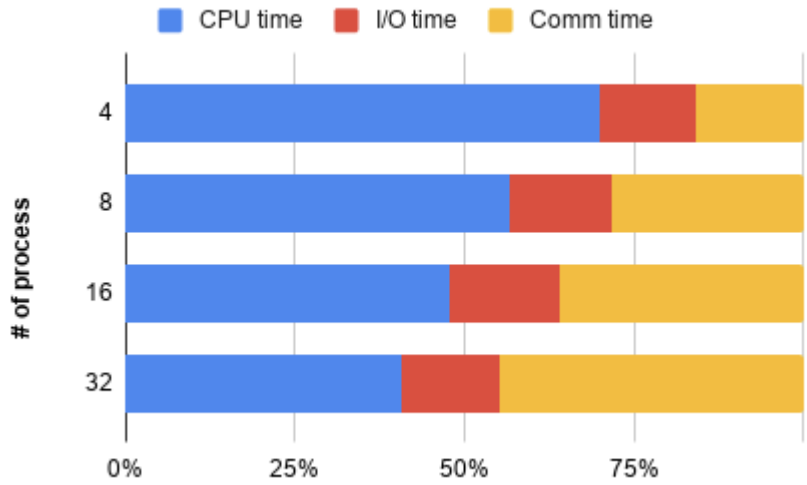


從上表與上圖可發現，CPU time在process數愈來愈多時，降低的幅度是愈來愈小，符合剛才推測的**工作切割給各個process，有時還會增加一些多的CPU工作**的情況，多做了如資料複製、搬移的動作。

而comm time在process數增多時，也成為影響整個runtime最多的buttleneck，連帶影響了speedup。



上圖與在一個node上跑不同數量的process的speedup的圖相比，可發現speedup在process數漸增的同時，上升的速度是愈加緩慢，comm time不會因process變多而減少，合理推測當process數再繼續上升下去的話，到達一定數量後，或許再增加process反而會使runtime變長(因為comm time)。



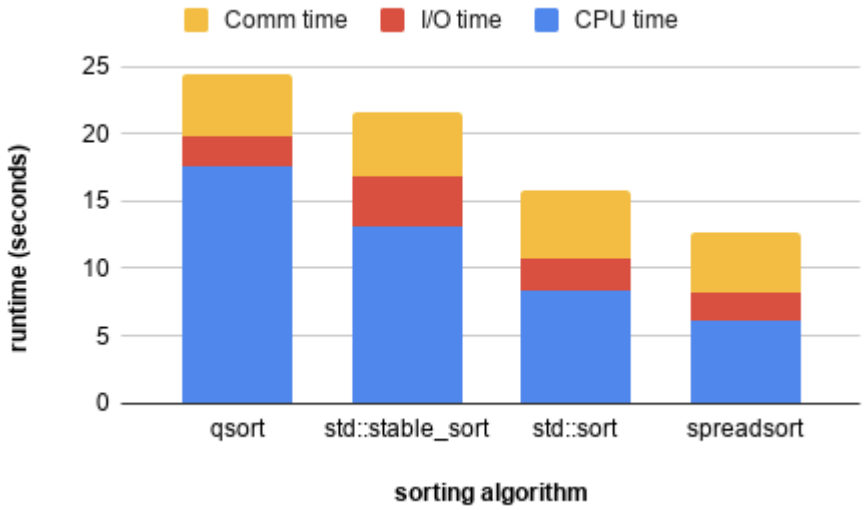
從不同process數所耗費的CPU time、I/O time、Comm time的占比來看，可發現在process數達到32時，comm time已成為bottleneck，若我們無法有效地降低comm的時間，盲目地增加process數反而會花費更多的時間在處理原先沒有做的事情，是否符合效益還不好說。

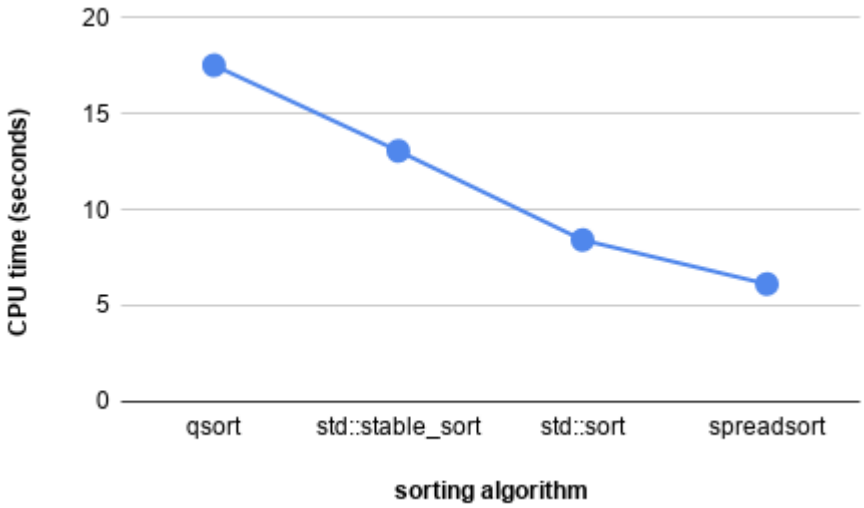
比較不同sorting algorithm對runtime的影響 (run testcase39)

我們都知道在測資數量愈來愈大時，演算法影響sort的時間也是更加的多，在實作hw1時原先是使用qsort去實作，但換成std::sort之後竟然可以使程式快上不少，因而想要討論不同的sorting algorithm對於此次作業runtime的影響。

以下的比較是跑testcase39，使用**2個nodes及8個processes**。

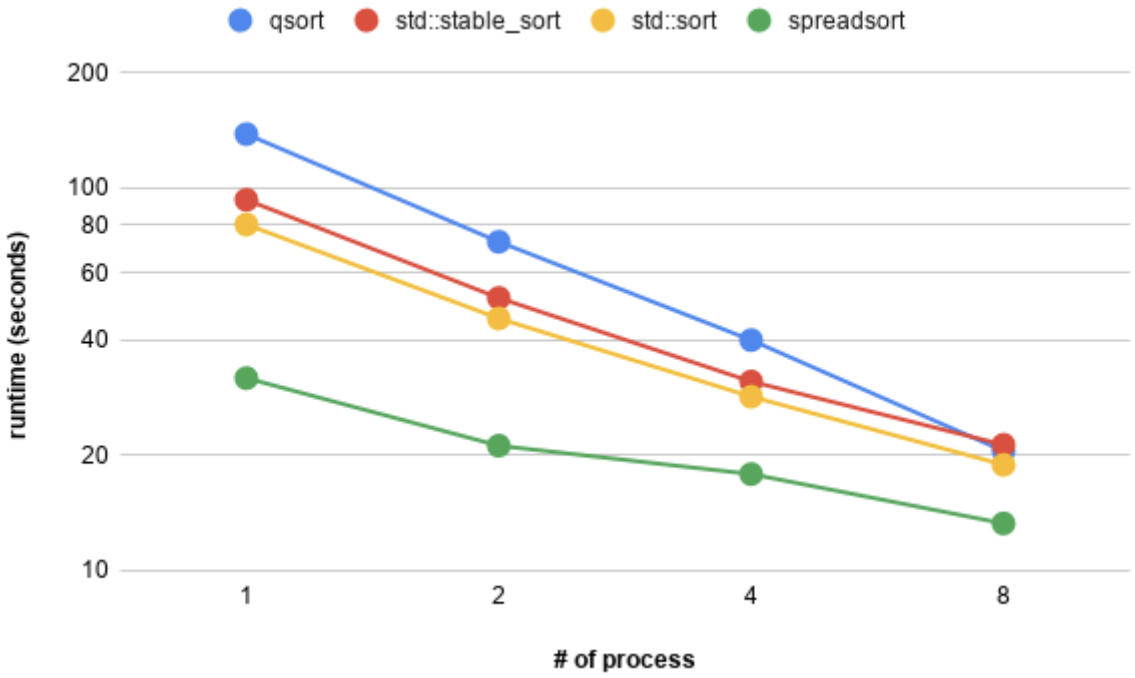
sorting algorithm	CPU time	I/O time	Comm time	Total time
qsort	17.53s	2.23s	4.71s	24.47s
std::stable_sort	13.06s	3.73s	4.85s	21.64s
std::sort	8.41s	2.29s	5.03s	15.73s
spreadsort	6.11s	2.06s	4.55s	12.73s





從上面兩圖與上表可發現不同的sorting algorithm甚至可影響到CPU time將近3倍的時間，甚至sort還有做一次，竟然可以影響到如此大，在I/O time、comm time將近不變的情況下，使用不同的sorting algorithm甚是能使時間降低至一半以下。

以下的比較是跑testcase39，使用**1個node跑不同數量的process**。



可以發現不同sorting algorithm在process數不同時，影響還是非常大，在只使用一個process時，spreadsort的時間(31.89s)大約只有qsort(138.53s)的1/4而已，然而sorting的速度愈快，scalability反而卻沒有那麼好，從藍線與綠線的斜率可看出，qsort的speedup相對於spreadsort，來得稍微高一些。猜測原因是因為快速的sorting algorithm已經減少非常多的runtime了，所以sorting time佔runtime的時間就沒有那麼多了，因而增多process能加速的效果也比較沒那麼好。

Discussion

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

  在process數少的時候，CPU time會是bottleneck，這可以透過增加process數去分散CPU計算的工作，繼而使CPU time降低。然而在process達到一定數量時，CPU time很難再有效地降低，comm time就會成為bottleneck，若要降低comm time，則只能減少process的數量。因此要在comm time與CPU time這兩者中做trade off，一味增加process數不一定是有效率的做法。

- Compare scalability. Does your program scale well? Why or why not? How can you achieve better scalability? You may discuss for the two implementations separately or together.

  我認為我作業的scalability還算可以，雖然無法達到linear speedup的程度，但大約每增加一倍的process，speedup也都有1.5~1.6左右的程度，單看CPU time的話，speedup的程度也算還不錯(1.5~接近2)。若要達到較好的scalability，就要盡量load balance，並且能減少溝通的次數的話是最好(因為溝通時間在process增加的時候會成為bottleneck)。

Experiences / Conclusion

在這次作業中，我學習到該如何透過多個process去有效加速整體程式，並且學到了process與process間溝通的方法與技巧。當初在上演算法時，對於不同的演算法是否有很大的影響這點，一直沒有深刻的理解，過往很多課程都是有寫出來就可以了，並沒有要求效率，這次很難得可以跟同學去比效能，才發現好的演算法有很大的影響。

開始寫這次作業的時候，其實一路上都還算順利，只有在測資大小變大的時候必須要用 `malloc` 這點讓我卡了一下，很快就能過了40筆測資，整份作業當中最難的其實是optimize自己的code，從最開始過40筆測資時一共是240多秒，到現在的149秒，耗費了比達到正確性還要多非常多的時間，達成正確性花了一多天，然而改良效能卻花了整整快兩個禮拜，目前也碰到了瓶頸不知道該如何繼續加強效能，因為自己還是比最前方的同學慢了10多秒，代表我還有很多空間能學習、進步，期許在之後的課堂中能更精進自己。