Parallel Programming
Homework 1: Odd-Even Sort
Report

106030009 葉蓁

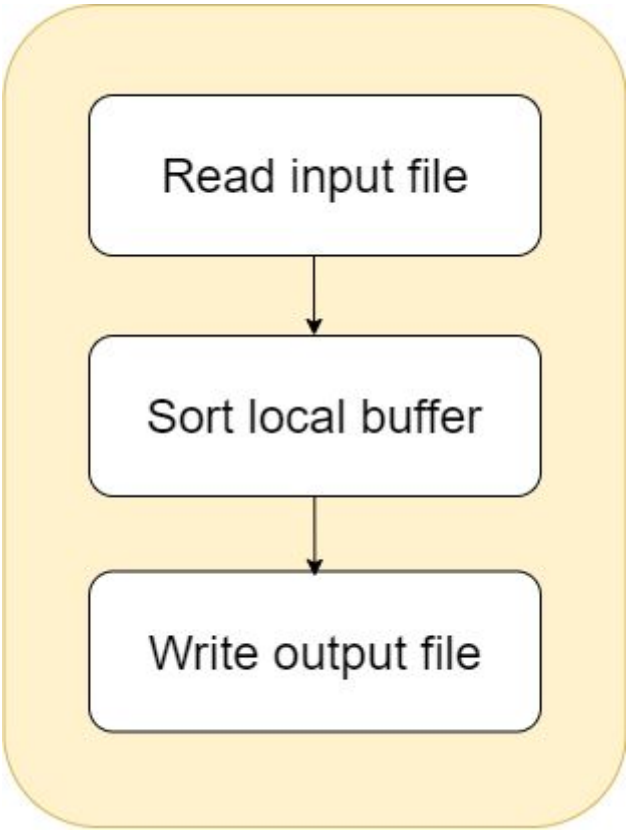
1. Implementation

針對每一個input, 分為三種case：

1. Only_One_Proc：只有一個 Process
2. Proc_Less_than_Num：Process數量比 input number 數量少
3. Proc_More_than_Num：Process數量比 input number 數量多

以下將分別針對三種Case說明程式邏輯：

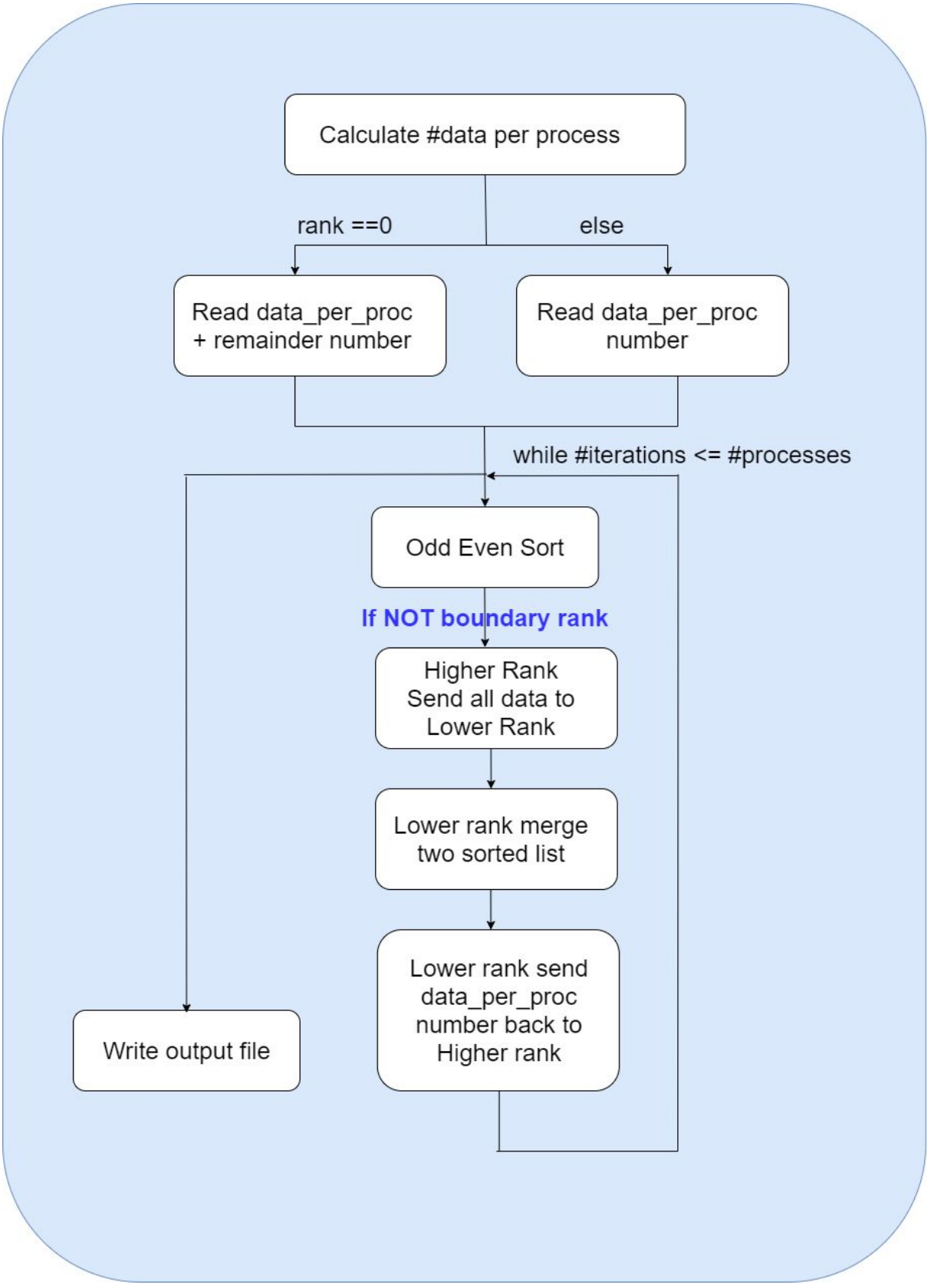
- Only_One_Proc：（相當於 sequential code）



因為只有一個 Process, 讀取全部的輸入資料，sorting一次後離開

時間複雜度：O(nlogn)

- Proc_Less_than_Num：



多數我們想像中的case都在此類。

首先為了load balancing，計算每個process平均要讀取的資料數：data_per_proc = n/size (總共幾筆數字/共有幾個process)

若除不盡，將餘數全部分給process 0 做。

接著進入 Odd-Even Sort環節：使用一個while 做 "size+1" 次數。(worst-case)

不論哪個phase，處理流程如下：

1. 兩個兩個task一組，rank較高者(以下簡稱R)傳自己所有的數字給rank較低者（以下簡稱L)。

2. L 使用 Two-way Merge將自己和 R 傳來的資料合併到一個Merge Buffer內

3. 再把Merge Buffer的**前半部份**留在自己(L內)

4. 後半部份傳給R。

前半部份：其實應該是前 LOCAL_DATA_SIZE 個數字留在自己的data buffer內，因為如果是rank 0，自己的local buffer會拿比其他rank多筆數字。

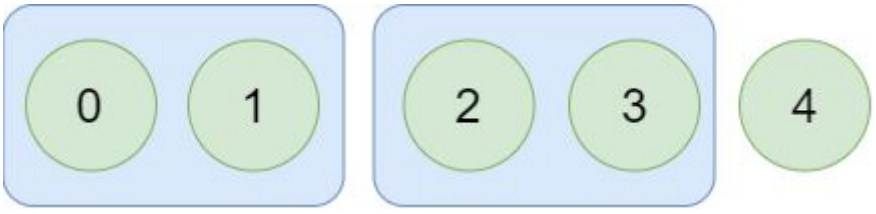
Odd-Even Sort完成後，每個process各自寫到 output file 正確的位置。

使用Two-way merge的好處：時間複雜度是O(N)，比直接call sort()的 O(NlogN) 快許多。

在Odd-Even sort階段，比較要注意的是**邊界的task**

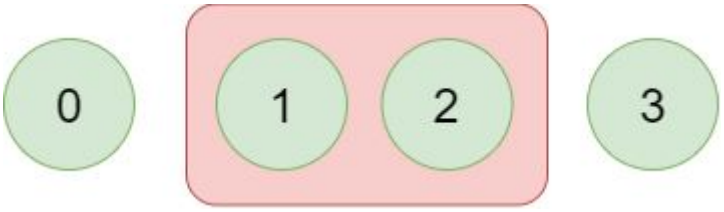
以下以兩個phase獨立說明：

Even-Phase:



如圖：若總共是奇數個task，最後一個task不能call send否則程式會hang住。

Odd-Phase:

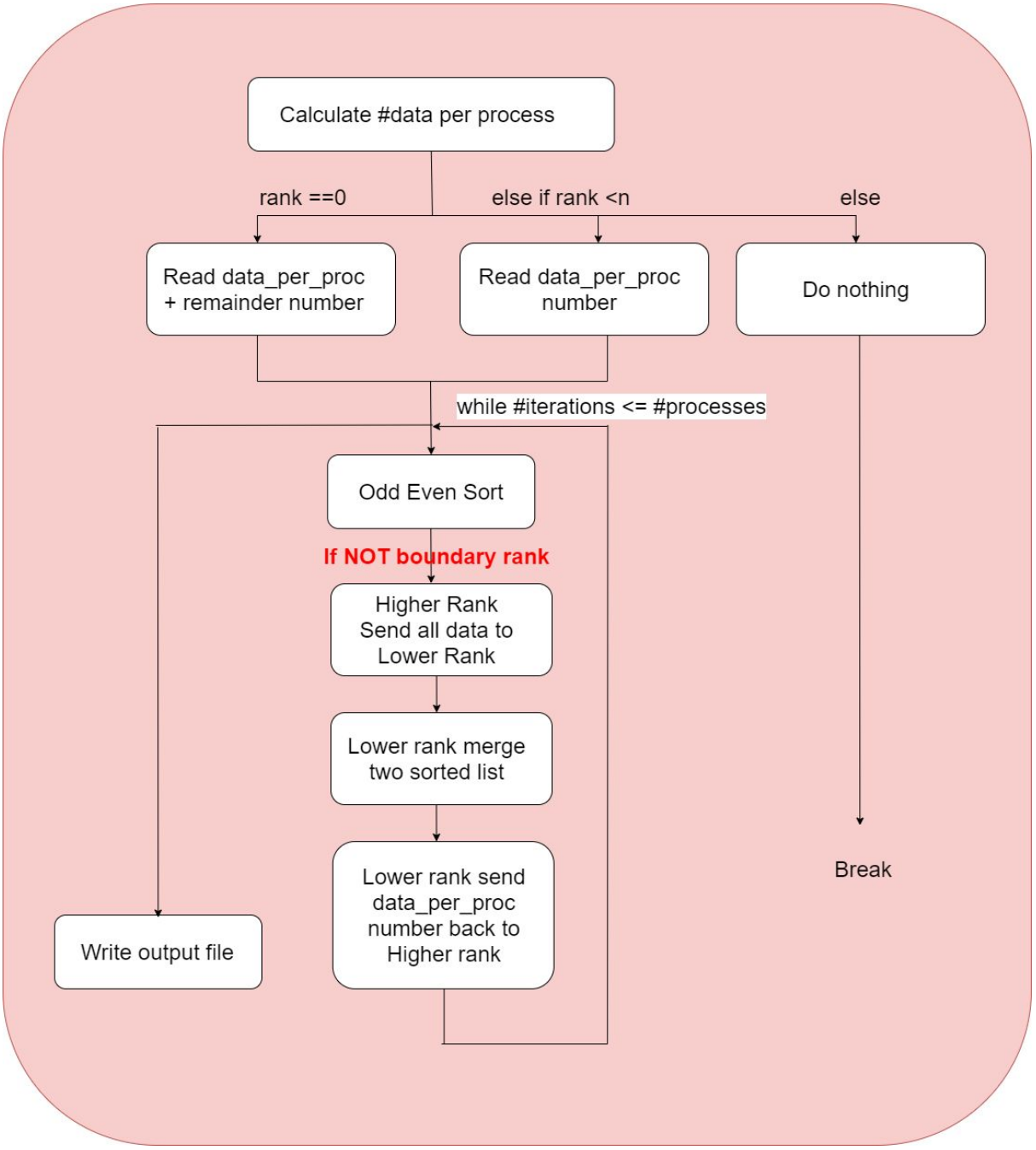


如圖：

1. rank 0 在 Even-phase一定不能做事

2. 若總數為偶數個task，最後一個task不能call send否則程式會hang住。

- Proc_More_than_Num：Process數量比 input number 數量多



在 Process 數比 Number 數多的情況，我直接讓 rank >= n 的 process 不做事，剩下的步驟跟 Proc_Less_than_Num 一樣。

2. Experiment & Analysis

嘗試優化的行為：

1. **加入Allreduce，每10iteration檢查一次是否要break**

   -> 效果不佳(增加comm時間)

2. **把 L proccess的local sort改成用Two-way merge**

   -> 降低時間複雜度(O(nlogn)->O(n))

如何收集圖表的資料：

首先：我寫了一個 gen_file.cc去生成 test_input.in，並且使用gen_ans.cc去生成正確的輸出結果作為正確答案的對照組(test_input.out)

如圖

```
[pp20s35@apollo31 test]$ ./gen_file 10000
10000 random float num
[pp20s35@apollo31 test]$ ./gen_ans 10000
0.001524 second
[pp20s35@apollo31 test]$
```

先設定好要跑那些測資，然後一個一個下去跑，每次Output出一個output file和執行時間。第一步要先確認答案是正確的：

```
[pp20s35@apollo31 my_ans]$ ./Verification_twoFiles test_500M_1.out ../test/test_input.out
Files are equal
```

然後再把執行時間填入excel表格

```
[pp20s35@apollo31 test]$ srun -N1 -n1 ../sample/hw1 500000000 ./test_input.in ../my_ans/test_500M_1.out
Took 73.71781 seconds.
CPU time: 65.20153 sec
IO time: 8.51628 sec
Communication time: 0.00000 sec
```

Methodology

(a) **System Spec** (If you run your experiments on your own cluster)
Please specify the system spec by providing the **CPU, RAM, disk and network (Ethernet / InfiniBand)** information of the system.

我使用apollo平台上之Cluster電腦計算。

Platform instruction - Apollo

- 20 nodes for this course (apollo31 - 50)
- Intel X5670 2x6 cores @ 2.93GHz
- 96GB RAM (each node)
- 5.5TB shared RAID5 disk
- QDR Infiniband

(b) Performance Metrics:

First, I define CAL_TIME_RANK be the rank number of the process that should measure time. For number of processes >1 : I use rank==1. For only one process: CAL_TIME_RANK = 0.

```
#define WTIME
#define CAL_TIME_RANK 1
```

For an MPI program, I use MPI_Wtime() function to record the start time and end time at the desired location, than the time duration between is (endtime - starttime).

```
//////// TIME MEASUREMENT //////////
#if defined(WTIME)
double starttime, endtime;
double IOstarttime, IOendtime;     // read, write file time.
// double CPUstarttime, CPUendtime;   // local_sort, merge time.
double COMMstarttime, COMMendtime; // Send, Recv time.
double COMM_total_time=0;
double IO_total_time = 0;
double CPU_total_time =0;
double Total_time = 0;
if (rank == CAL_TIME_RANK) {
    starttime = MPI_Wtime();
}
#endif
```

For IO time, I added the time for reading floating points from an input file, and the time to write results into an output file.

```
/////////// Read FLOAT from file //////////

if(rank==CAL_TIME_RANK) IOstarttime = MPI_Wtime();

MPI_File f; // file handler
rc =MPI_File_open(MPI_COMM_WORLD, argv[2], MPI_MODE_RDONLY, MPI_INFO_NULL, &f);
if (rc != MPI_SUCCESS) {
    printf ("Error opening file. Terminating.\n");
    MPI_Abort (MPI_COMM_WORLD, rc);
}

switch(state){…
MPI_File_close(&f);

if(rank==CAL_TIME_RANK){
    IOendtime = MPI_Wtime();
    IO_total_time += (IOendtime - IOstarttime);
}
```

For communication time, I added all the MPI_Send(), MPI_Recv() time . For example:

```
    if(rank==CAL_TIME_RANK) COMMstarttime = MPI_Wtime();
    MPI_Recv(buf , data_per_proc , MPI_FLOAT , rank+1 , 0, MPI_COMM_WORLD , &status);
    if(rank==CAL_TIME_RANK){
        COMMendtime= MPI_Wtime();
        COMM_total_time += COMMendtime-COMMstarttime;
    }
```

And for the CPU time, my metrics is : CPU time = Total time - IO time - Comm time

```
////////////// TIME STOP ///////////////
#if defined(WTIME)
if(rank==CAL_TIME_RANK){
    endtime = MPI_Wtime();
    Total_time = endtime - starttime;
    printf("Took %.5f seconds.\n", endtime - starttime);
    // printf("break at iteration %d\n",count);

    printf("CPU time: %.5f sec\n", Total_time - IO_total_time - COMM_total_time);
    printf("IO time: %.5f sec\n", IO_total_time);
    printf("Communication time: %.5f sec\n", COMM_total_time);
}
#endif
```

For Sequential time, I use clock_gettime() to output the time of the best sequential program.

```
struct timespec start, end, temp;
double time_used;
clock_gettime(CLOCK_MONOTONIC, &start);

//     ....  stuff to be timed  ...
```

```
clock_gettime(CLOCK_MONOTONIC, &end);
if ((end.tv_nsec - start.tv_nsec) < 0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
    temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
} else {
    temp.tv_sec = end.tv_sec - start.tv_sec;
    temp.tv_nsec = end.tv_nsec - start.tv_nsec;
}
time_used = temp.tv_sec + (double) temp.tv_nsec / 1000000000.0;

printf("%f second\n", time_used);
```

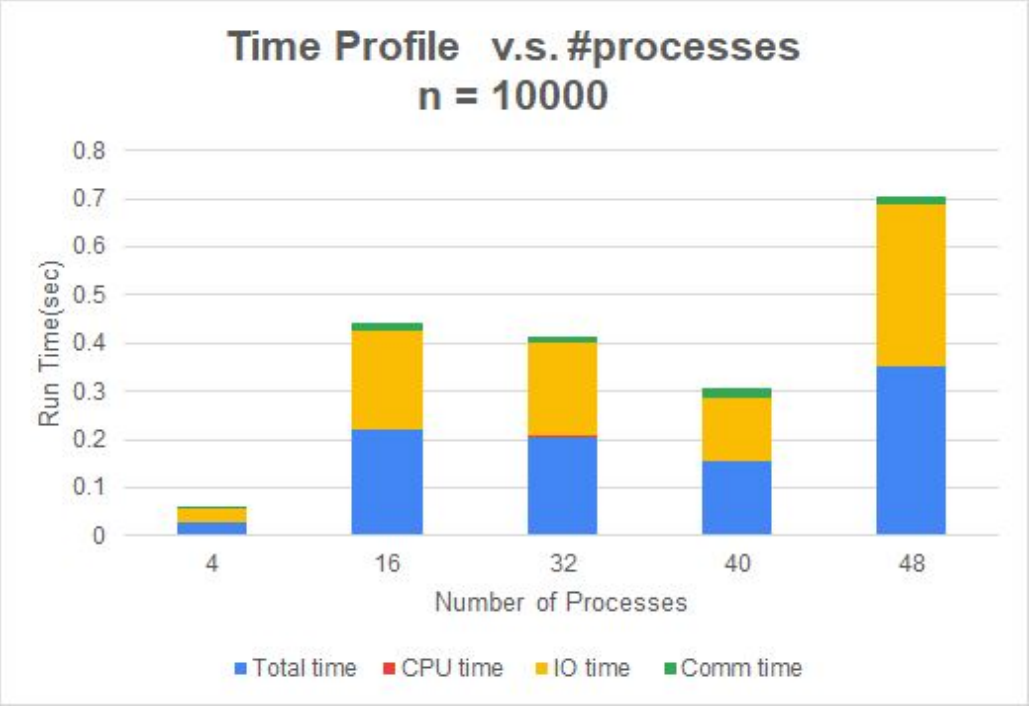
Plots: Speedup Factor & Time Profile

我嘗試三種 testcase size:

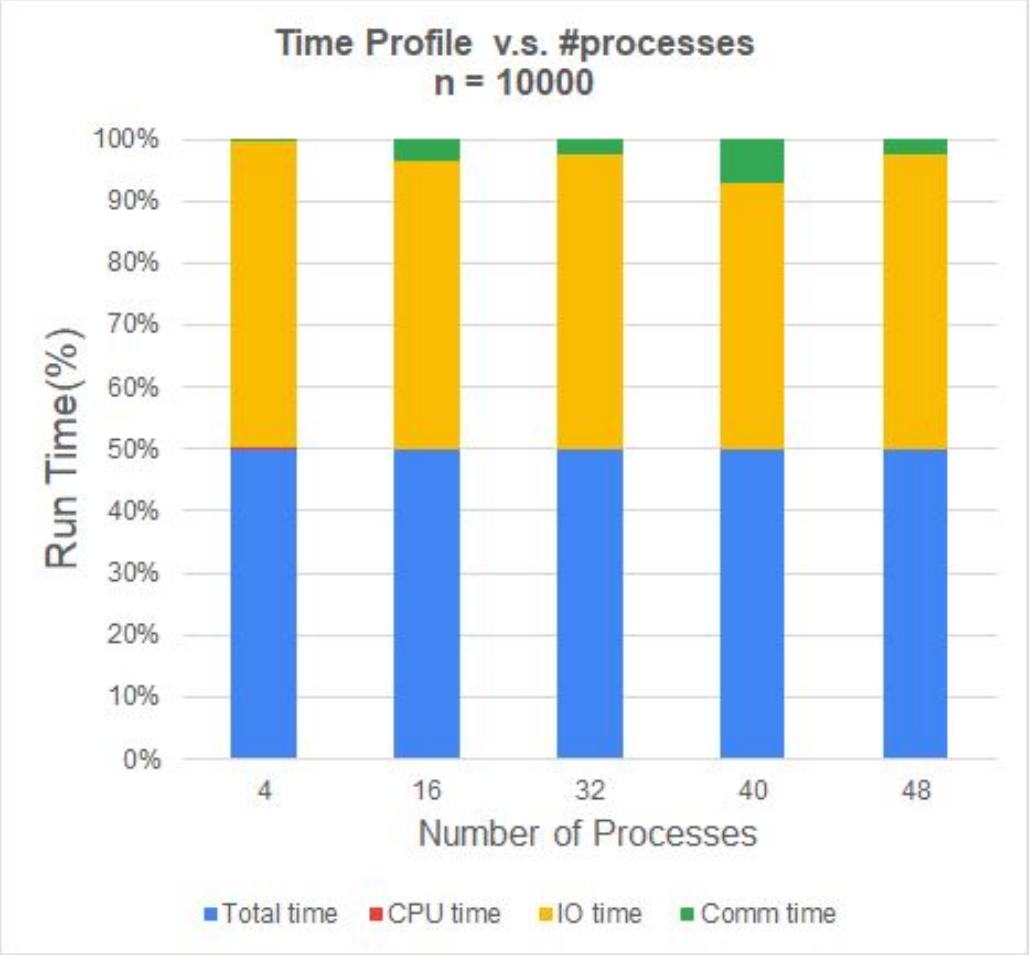
1. n=10000 (N=4)

Time Profile: (Unit: second)

Time Profile				
Testcase size	**# Computing Node**			
10000	4			
# processes	**Total time**	**CPU time**	**IO time**	**Comm time**
4	0.02758	0.00011	0.02732	0.00015
16	0.2203	0.00004	0.20533	0.01492
32	0.20625	0.00003	0.19627	0.00994
40	0.15352	0.00003	0.13211	0.02138
48	0.35234	0.00003	0.33558	0.01674


Time Profile v.s. #processes
n = 10000
Run Time(sec)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
4
16
32
40
48
Number of Processes
Total time
CPU time
IO time
Comm time

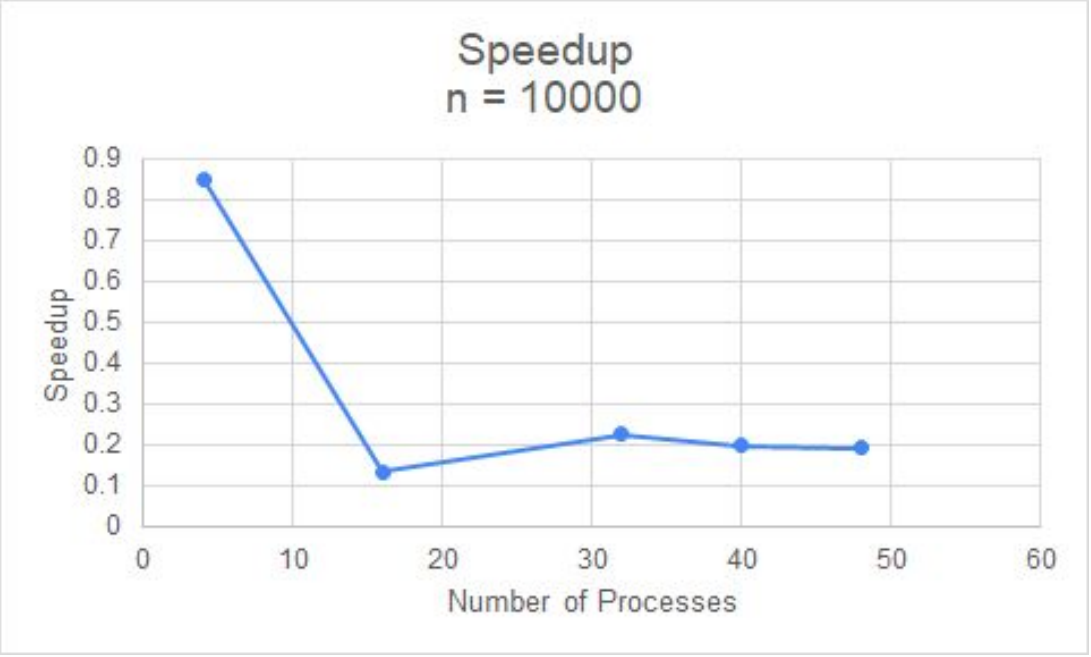
百分比圖表：


Time Profile v.s. #processes
n = 10000
Run Time(%)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
4
16
32
40
48
Number of Processes
Total time
CPU time
IO time
Comm time

Speedup:

Testcases size	N (computing node)		
10000	4		
#processes	**Sequential Time**	**Parallel Time**	**Speedup**
4	0.032766	0.03849	0.851286048
16	0.032766	0.24473	0.133886324
32	0.032766	0.14554	0.225133984
40	0.032766	0.16498	0.198605892
48	0.032766	0.17102	0.191591627

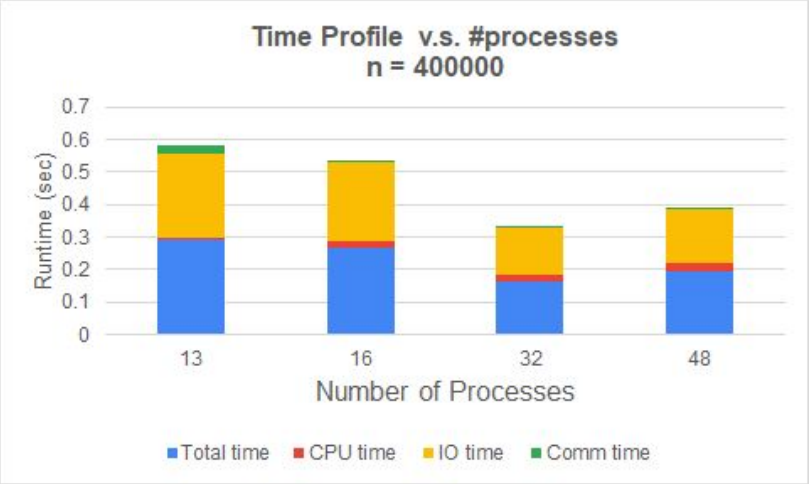
下圖為 N = 4 的 Speedup diagram :



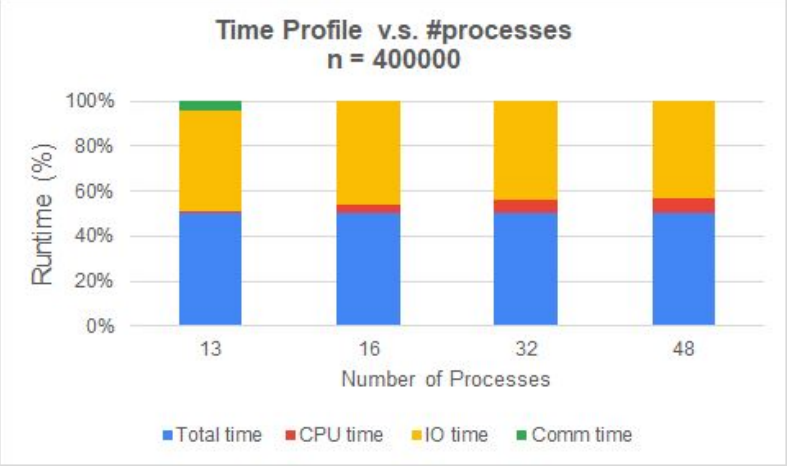
2. n=400000 (N = 4)

Time profile: (Unit: second)

Time Profile				
Testcase size	# Computing Node			
400000	4			
# Processes	Total time	CPU time	IO time	Comm time
13	0.29185	0.00411	0.26199	0.02575
16	0.26637	0.02022	0.24558	0.00057
32	0.16505	0.02118	0.14337	0.0005
48	0.1937	0.02539	0.16775	0.00056

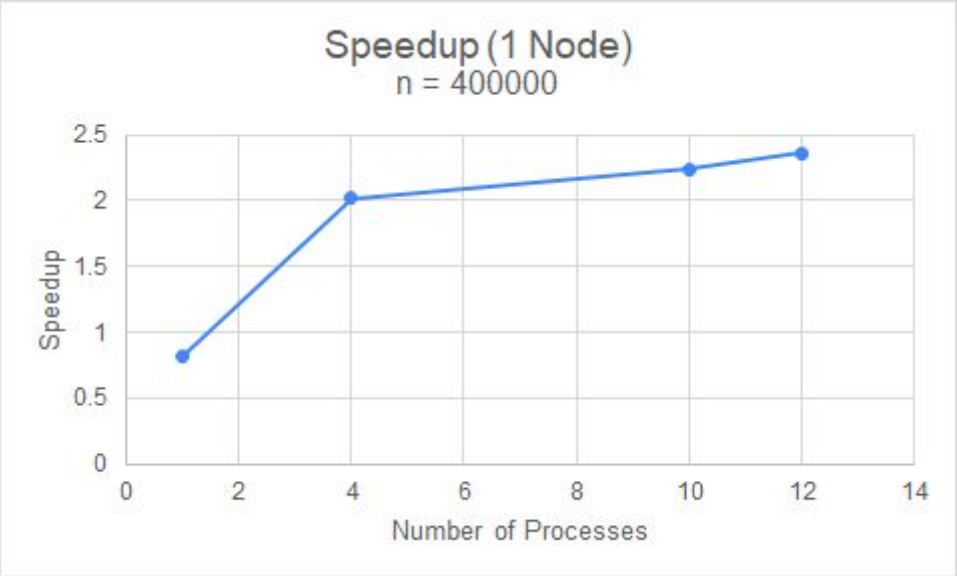


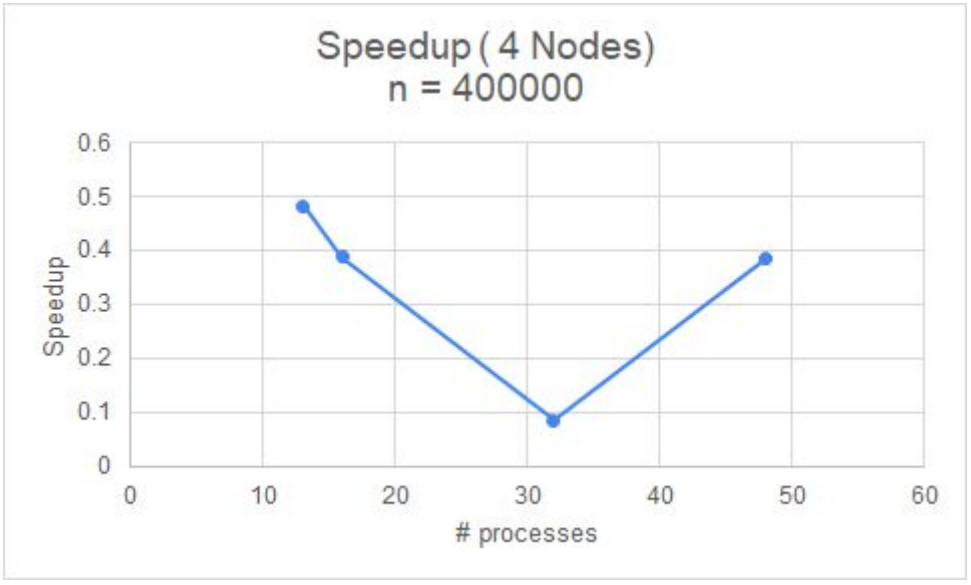
百分比：

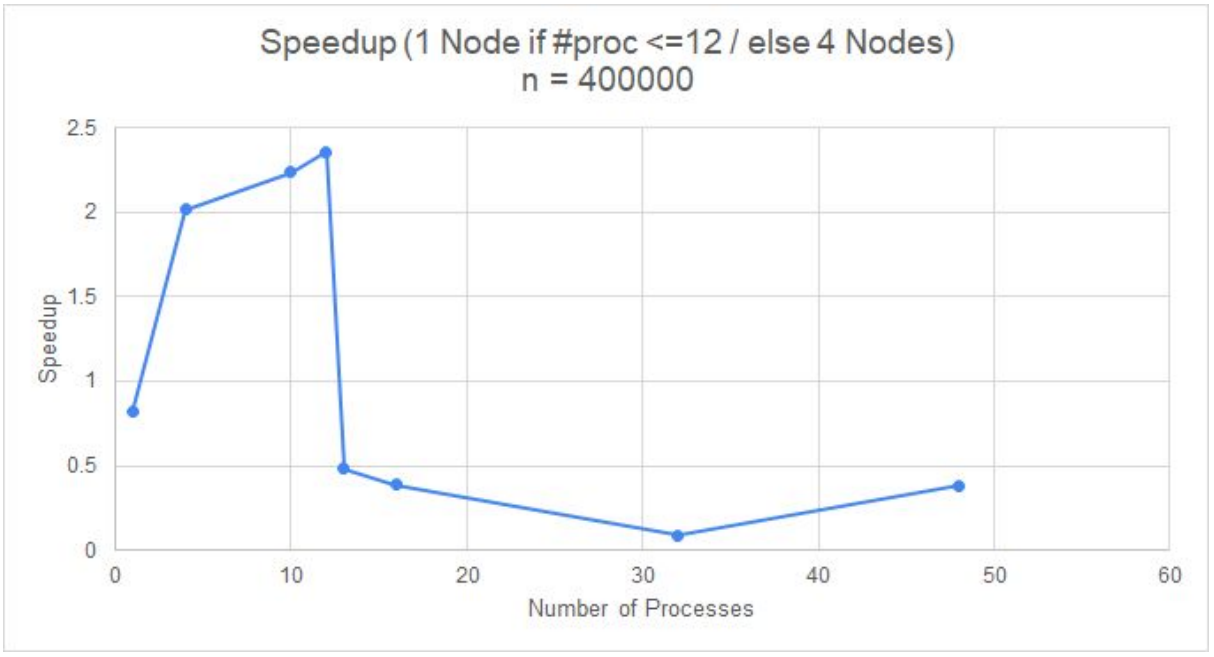


Speedup:

Testcase size	400000			
# processes	# Nodes	sequential	parallel code	speedup
1	1	0.068048	0.08312	0.8186718
4	1	0.068048	0.03378	2.014446418
10	1	0.068048	0.03043	2.236214262
12	1	0.068048	0.02884	2.359500693
13	4	0.068048	0.14082	0.483226814
16	4	0.068048	0.17502	0.38880128
32	4	0.068048	0.79396	0.085707089
48	4	0.068048	0.17731	0.383779821




Speedup (4 Nodes)
n = 400000
Speedup
0.6
0.5
0.4
0.3
0.2
0.1
0
0
10
20
30
40
50
60
processes


Speedup (1 Node if #proc <=12 / else 4 Nodes)
n = 400000
Speedup
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
Number of Processes

3. n=500000000 (N=4 ; 4 Computing Nodes)

Time profile: (Unit: second)

```
[pp20s35@apollo31 test]$ srun -N4 -n4 ../sample/hw1 500000000 ./test_input.in ../my_ans/test_500M_4.o
ut
Took 26.11765 seconds.
CPU time: 5.63451 sec
IO time: 19.49287 sec
Communication time: 0.99027 sec
```

Time Profile				
Testcase size	**# Computing Node**			
50000000	4			
# processes	**Total time**	**CPU time**	**IO time**	**Comm time**
4	26.11765	5.63451	19.49287	0.99027
8	17.97618	6.02469	11.05847	0.89303
12	15.05311	5.90884	8.11383	1.03043
16	15.86219	7.86286	6.44327	1.55606
32	14.56198	6.90761	5.43166	2.22271
48	14.2108	8.61139	3.99182	1.6076

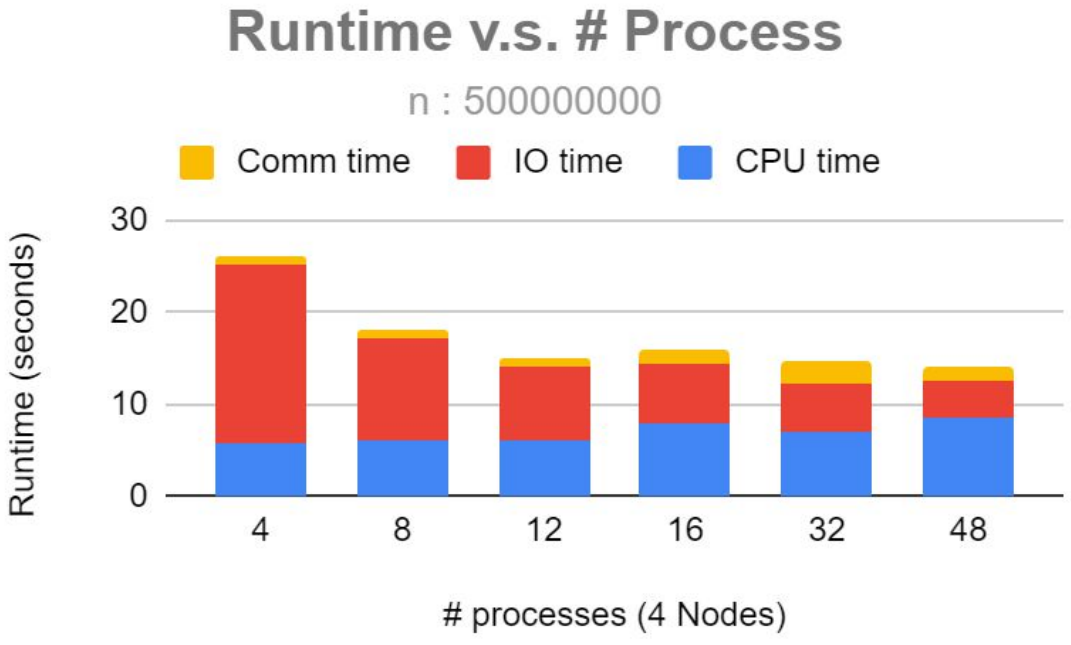
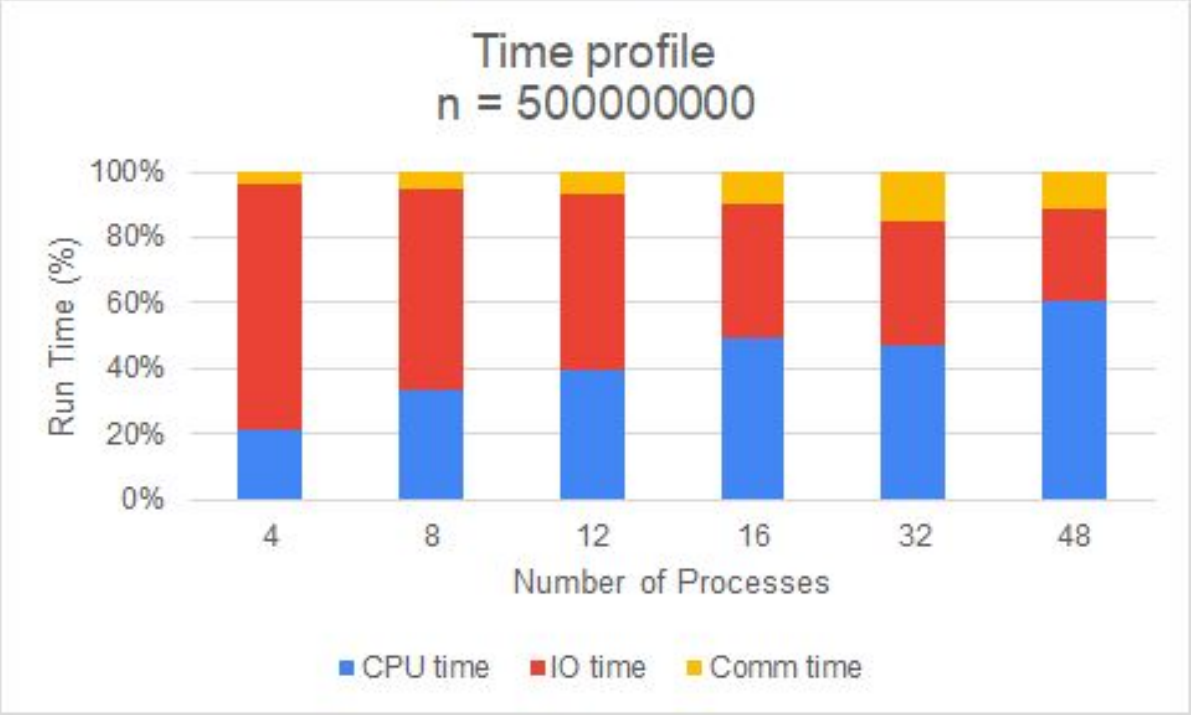


Figure 1: Time profile (Multi-Node)

百分比：



3. n=500000000 (N=1 ; 1 Computing Node)

Time Profile				
Testcase size	**# Computing Node**			
50000000	1			
# processes	**Total time**	**CPU time**	**IO time**	**Comm time**
1	73.71781	65.20153	8.51628	0
2	45.11096	4.18037	39.02375	1.90684
4	27.67104	3.45566	21.35266	2.86272
8	22.90968	3.41269	16.08448	3.41251
10	21.8015	3.12488	14.18552	4.4911
12	20.06522	3.73234	10.85819	5.47469

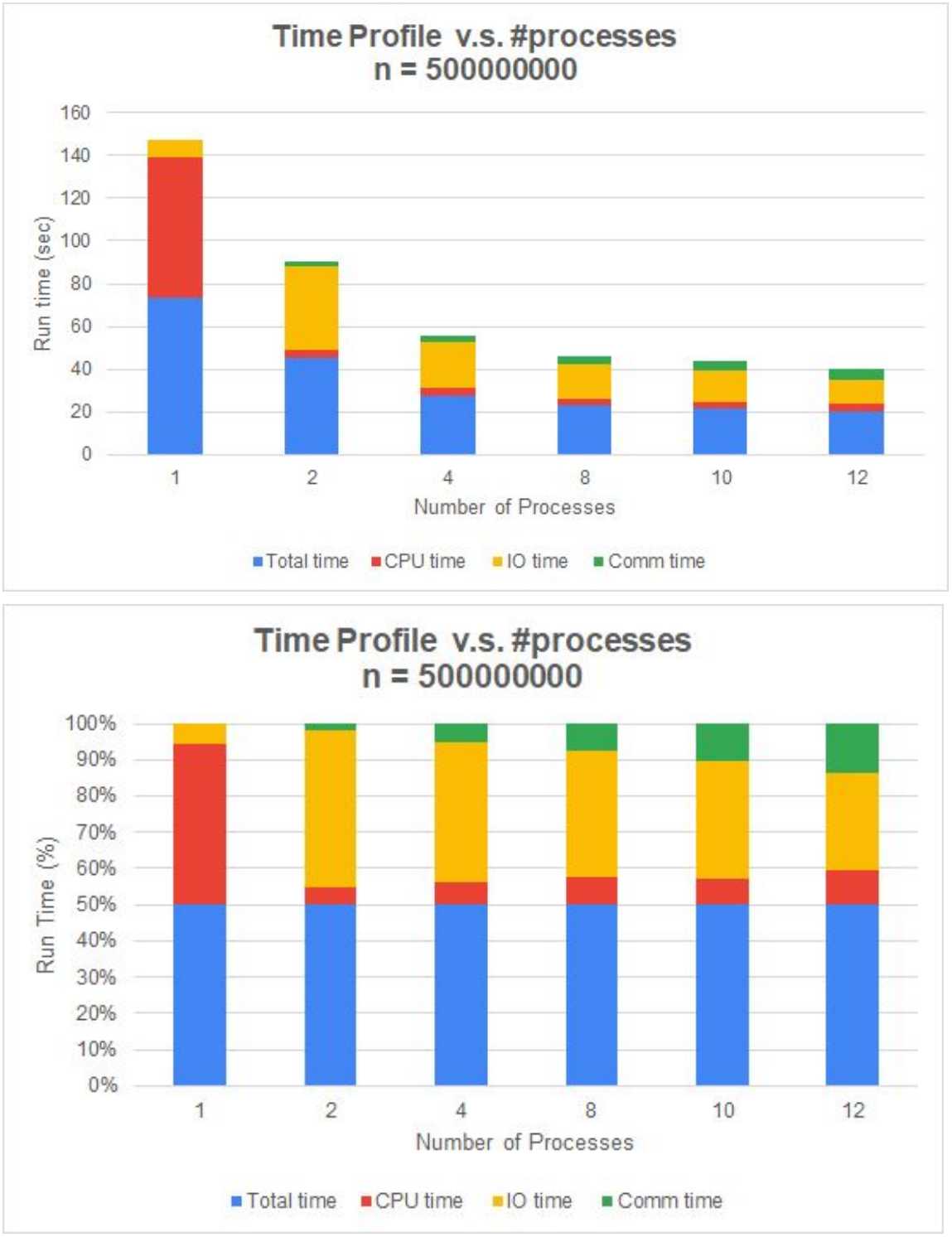
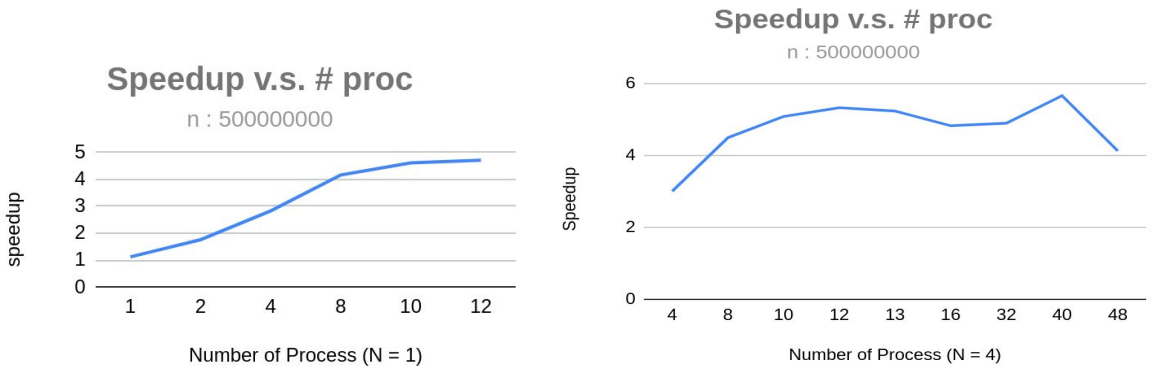


Figure 2: Time profile (Single Node)

Speedup : test case size (n) = 50000000

Left : Only one computing node (N=1) Right: 4 computing node (N = 4)



Overall Speedup: (for process size <=2: N=1; else, N=4)

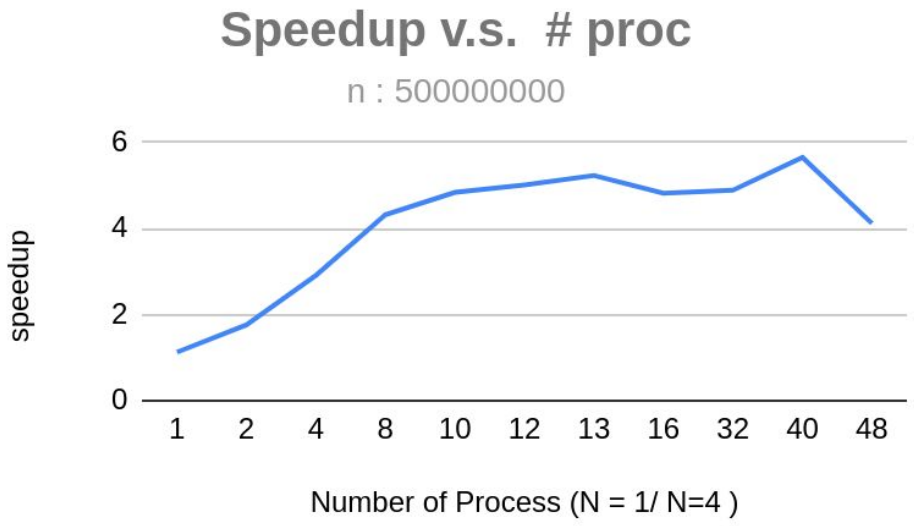
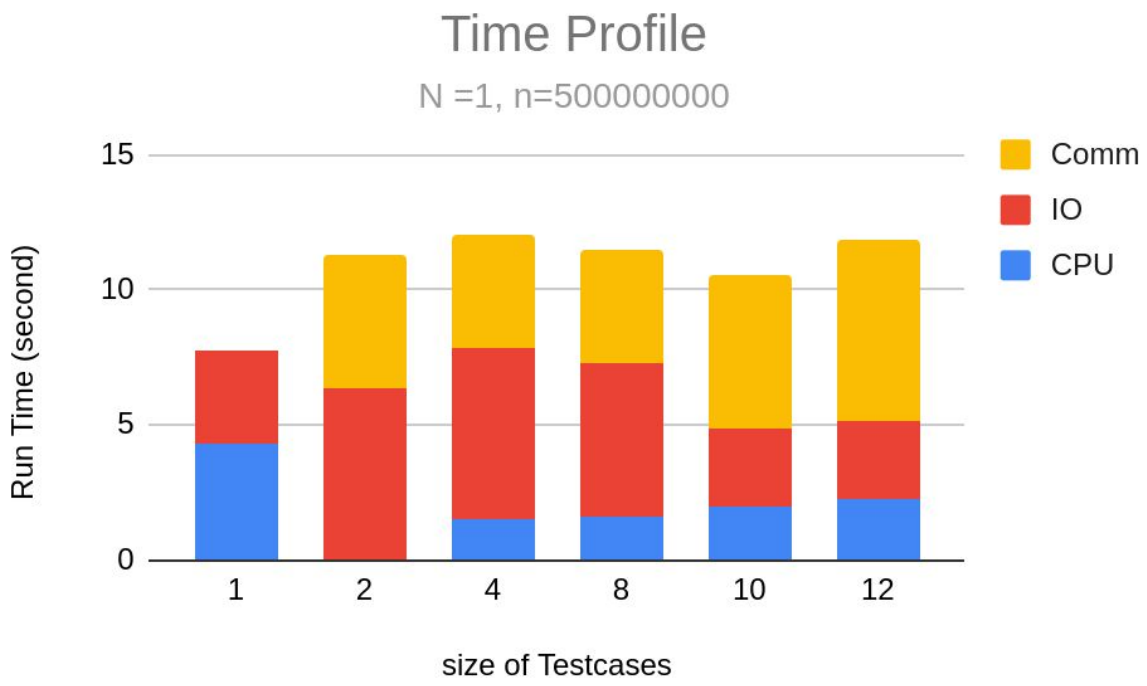
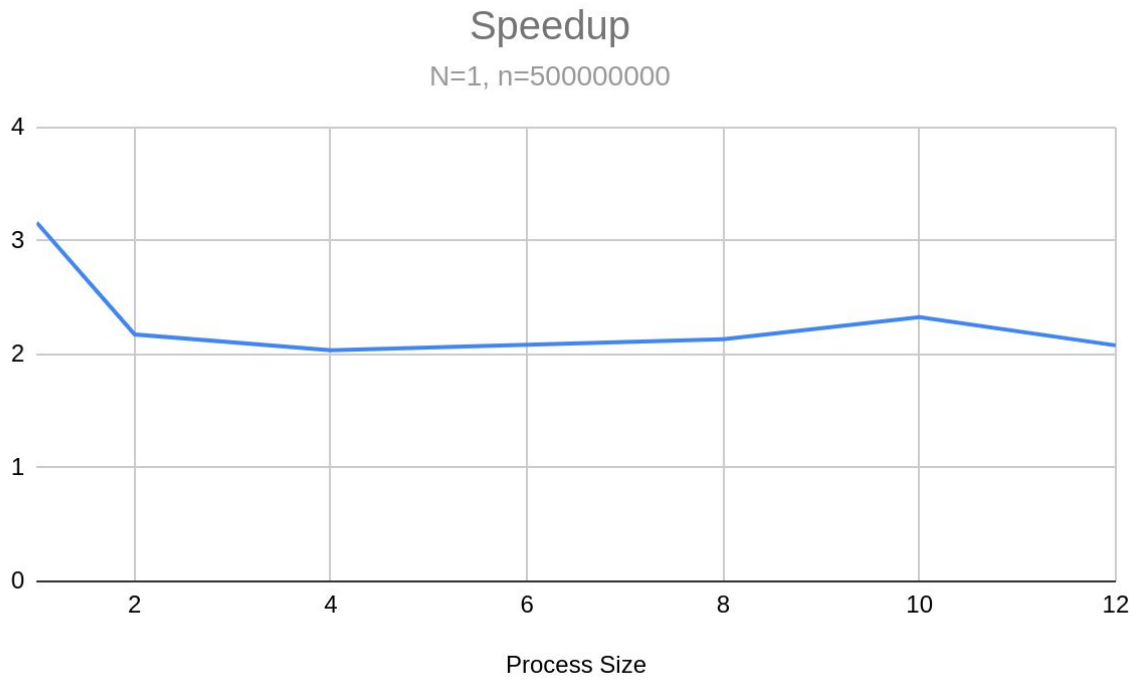


Figure 2: Speedup

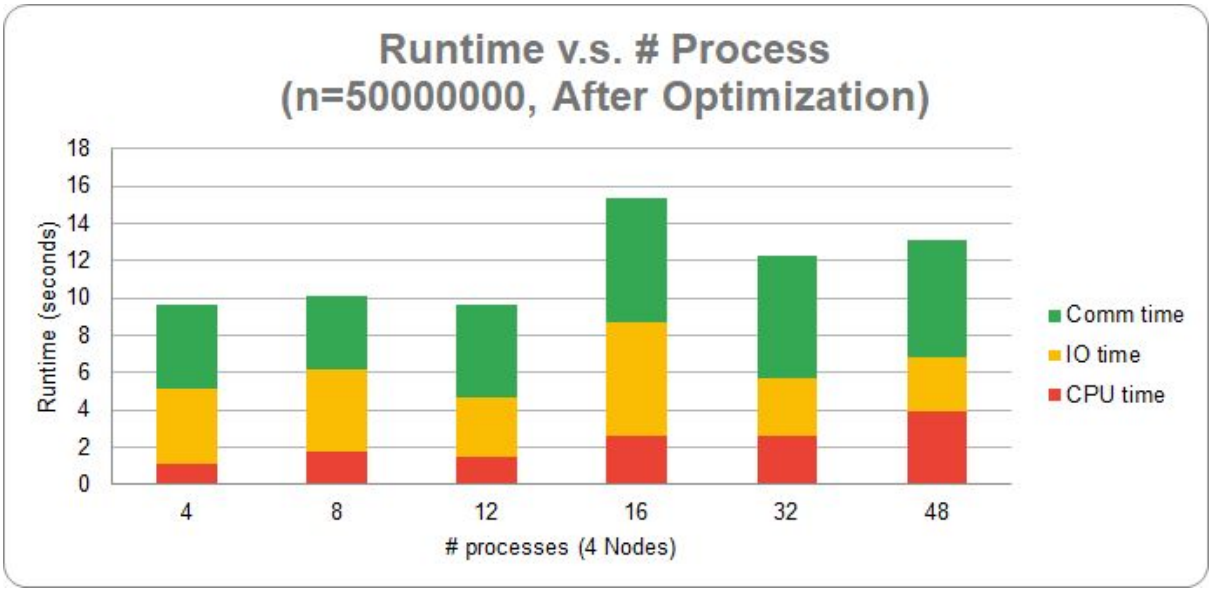
4. 優化後的 Testcase size : 500000000, N=1 ：



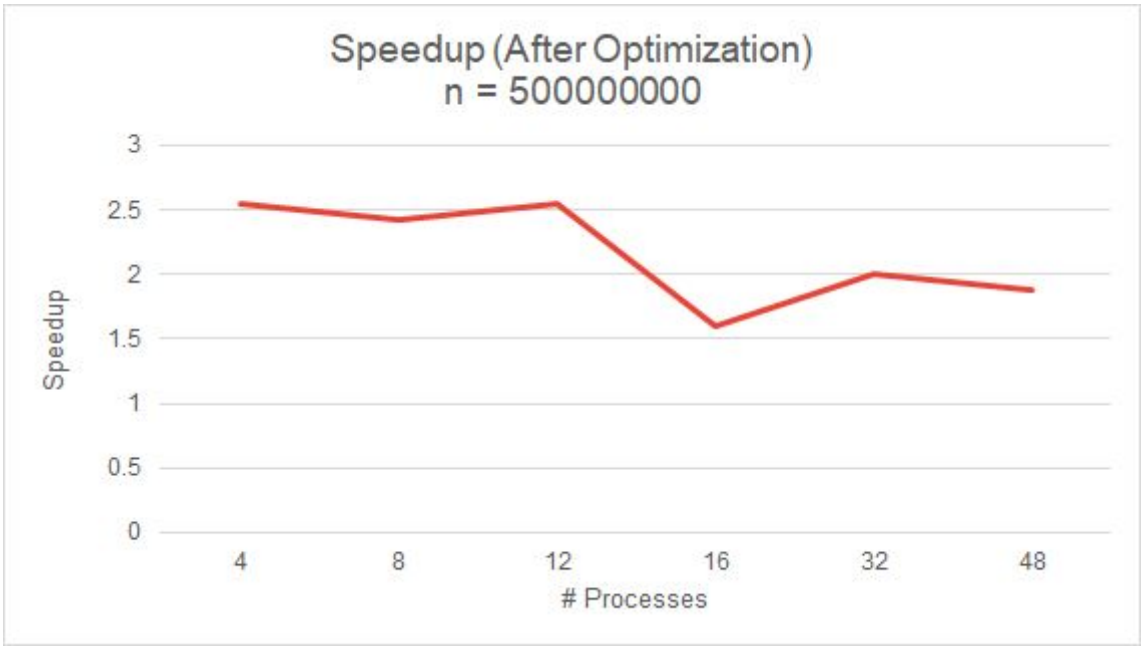
Time Profile				
size of Testcases	# Computing Node			
50000000	4			
Proc size	**Total time**	**CPU time**	**IO time**	**Comm time**
1	7.76341	4.34223	3.42118	0.0000
2	11.28829	0.00005	6.39737	4.89087
4	12.06050	1.53557	6.27040	4.25453
8	11.50580	1.57466	5.70063	4.23051
10	10.54757	1.94707	2.88138	5.71912
12	11.80637	2.24658	2.90628	6.65351



4. 優化後的 Testcase size : 500000000, N=4 ：



Time Profile				
size of Testcases	# Computing Node			
50000000	4			
proc size:	**Total time**	**CPU time**	**IO time**	**Comm time**
4	9.65435	1.06578	4.04273	4.54585
8	10.14791	1.7721	4.41604	3.95977
12	9.63565	1.51478	3.15059	4.97028
16	15.3429	2.56657	6.11075	6.66557
32	12.22478	2.61325	3.11043	6.5011
48	13.07596	3.91293	2.90155	6.26148



奇怪的是：當我後來優化，將sorting從原本 <algorithm> library的sort()換成 <boost/sort/spreadsort/float_sort.hpp> 的 float_sort()後，時間加速了不少!

但是 Speedup 竟然變成往下走... 好像process越多反而越慢，有點詭異阿...。

5. Discussion:

1. 從input size由10000 -> 400000 -> 500000000，可發現當輸入資料量大於一定程度時，平行程式的效能才變得顯著 (當單一程式sorting所花的CPU time時間遠大於程式之間溝通的Comm time時，Speedup的效果才出的來。
2. 討論三個不同 input size的bottleneck :

	# Computing Node	# Process			
	4	16			
Testcase size	**Total time**	**CPU time (%)**	**IO time (%)**	**Comm time (%)**	**Bottleneck**
10000	0.2203	0.018157059	93.20472084	6.772582842	**IO**
400000	0.26637	7.590944926	92.19506701	0.213988062	**IO**
50000000	9.65435	11.039376	41.87469	47.086	**Comm**

可見資料量少的時候，bottleneck傾向於 IO 的讀寫時間 ；但當資料量大於一定程度 (e.x. 百萬筆資料時)，瓶頸會在使用Communication，也就是Process間彼此溝通需花費的時間。

3. 討論 Testcase size = 500000000的 CPU, IO, Network (Comm)效能

以優化前結果而言，在資料量大時，因 Odd-Even sort的規則，只有相鄰的process可以交換資料，在worst case情況：若最後一個process拿到最小的數字，需一路傳給rank 0，而 process size又很大（e.g. 48個process）時，則需要至少47個 iteration才能把資料傳到rank 0者。如此因為溝通所需花的時間就變得相當可觀，整體overhead變大。

如何可以improve? 目前除了將最開始local sort的library不停嘗試更快的方法，在merging部份可能可以嘗試 Bitonic merge ，也許能將complexity再降低。

4. 討論 Testcase size = 500000000 的 Strong Scalability

優化前的Speedup curve, 不論N=1 或 N=4 看起來都算正常。雖然不是ideal的線性增加，但整體趨勢是往上增長。

優化後的Speedup 變得相當詭異：竟然是往下降的！可能原因是：更改後的library中的sorting用了神奇的演算法，（我應該要去trace

內部的code是如何實做。也許它優化的太快，造成溝通的overhead大於local sort所花的時間，因此process數越多時反而效能越差。

5. 程式 Time Complexity 估算：

設process size: m, number size: n。

O(const) +2*O(n/m)+ **m***O(const. +2*(n/m) + 4*2* (n/m))

= O(n/m)+m*O(n/m) = O((m+1)*(n/m)) **= O(n)** (和input size成正比)

算法：

初始化MPI,宣告變數等 + 讀/寫資料 +

while迴圈(count<=m)次 *(Send+Recv n/m筆數字 + merge(4次scan 2*n/m筆數字))

3. Experiences / Conclusion

1. Conclusion

平行程式的Speedup和Scalability除了和程式本身寫的好壞之外，和 Problem本身的平行度也有關係。若問題本身data之間彼此dependencies很高，很難發揮多個process同時進行的好處，complexity也可難達到O(sequential) / m (#processes) 的理想值。

但還是可以透過演算法設計，或資料處理方法，盡可能加快程式執行速度，提升程式效能。

2. Difficulties & Learn

過程中我遇到非常多問題，在此逐一列點，包含其解決方法：

a. 在 Write file 時，因為沒有加入：導致不能直接寫一個"新的檔案"(i.e.同時新建檔案+寫入)，導致我自己srun 01.in時明明output file跟sample answer一模一樣，卻在hw1-judge時全部都run time error。記得當時被這難關卡了非常久，後來好險助教協助指導，才解決。

b. 一開始對input file使用 od -tfF 指令後展開的樣子，讀取無法理解

```
[pp20s35@apollo31 testcases]$ hexdump 01.in
0000000 d37e c688 c6ee 46b9 954a c65b 192a c654
0000010
```

經查詢發現使用 因Big-endian，所以檔案讀取方式為：

第一個數字：0xc688d37e

第二個數字：0x46b9c6ee

以此類推

c. 初始我在 Odd-Even sort 時，每次只讓process傳一個數字（R傳最左邊者（最小值），L傳最右邊者（最大值））。然後兩個task 都要重新 sort後，重複以上動作。

我也不知道為什麼一開始沒有想到直接send, recv全部的data，最初只想著先求有，再求好。不過之後當然狂遇TLE。有一天突然想到：為何不一次傳多一點？

d. 在決定 while 迴圈要執行的次數時，我一直被卡在 n個數字worst-case 要做n次。其實是size次！因為總共有size個process, 假設最小的數字在最後一個rank, 至多size次就可以從rank LAST 傳到 rank 0了！改成while(cout <= size)後終於ac了！（喜極而泣）

e. 接著進入優化環節：因為我原本的作法是：每個iteration, R 傳給 L，L 把recv的data合併到自己的local data （使用vector.insert)。 sort 完後再把後半大的資料傳回給R。與同學討論後，得知每次都使用sort 的時間複雜度是 O(nlogn)。如果改成merge會變成只有O(n)，對整體效能會差非常多。因此我決定使用 Two-way merge。

但我做了一件非常不好的事：我宣告了以下的靜態陣列：

float buf [data_per_proc];

float MergeBuf [LOCAL_DATA_SIZE + data_per_proc];

與同學討論後，得知編譯器在compile的時候，針對靜態記憶體，會先allocate好 memory size，但不會到太大。如果run time時突然 input 一個非常大的資料，就很可能memory allocate不足，導致指到無效的address，導致segmentation fault!

後來改成使用 malloc() 動態宣告記憶體，就解決！

優化後的版本：從原本的798秒 -> 206 秒！ 整整快了快四倍！真驚人的差別。

f. 學到如何為自己的程式碼估算時間複雜度，並且透過 Time profile 找出程式執行時間的bottleneck，進而想辦法從瓶頸下手，提升程式效能。