Homework 2: Mandelbrot Set

107062612 熊祖玲

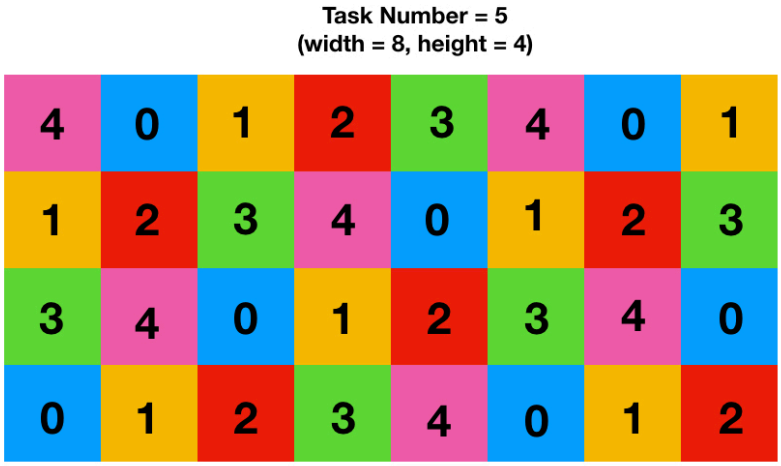
I. Implementation

A. OpenMP

在這個版本的實作上，我先寫好sequential的版本，而在squential版本中會出現兩個迴圈，計算所有座標軸上的複數經過幾次的運算，會得知這個複數是否屬於mandelbrot set，最後得到圖片中每個pixel的數值。然而，我的openmp指令加在這兩個for迴圈的外面，所以選擇使用#pragma omp parallel for schedule(dynamic, 64) copllase(2) ，將兩層迴圈展開達到平行的效果。

B. MPI_Static

在這個版本的實作中，有稍微複雜了一點，若沒有平均分配每個task的工作或被分配到的工作需要花比較多時間完成，會造成有些task很快完成工作，但仍需等待其它task(s)，如此一來會造成平行後的速度無法有效地提升。因此，我使用Round Robin的概念分配工作，每個task依序且輪流取得工作，如圖一所示，每個不同的顏色（編號）由不同的task計算，每個task須完成的工作量為6至7個，而且靠近中心點的工作不會由一個task全權負責。這不但可以確保每個task拿到的工作量是平均的，而且每個task做的每個工作需要的時間會參差不齊，不會讓其中幾個task的工作負擔特別大，成為整個程式的瓶頸。

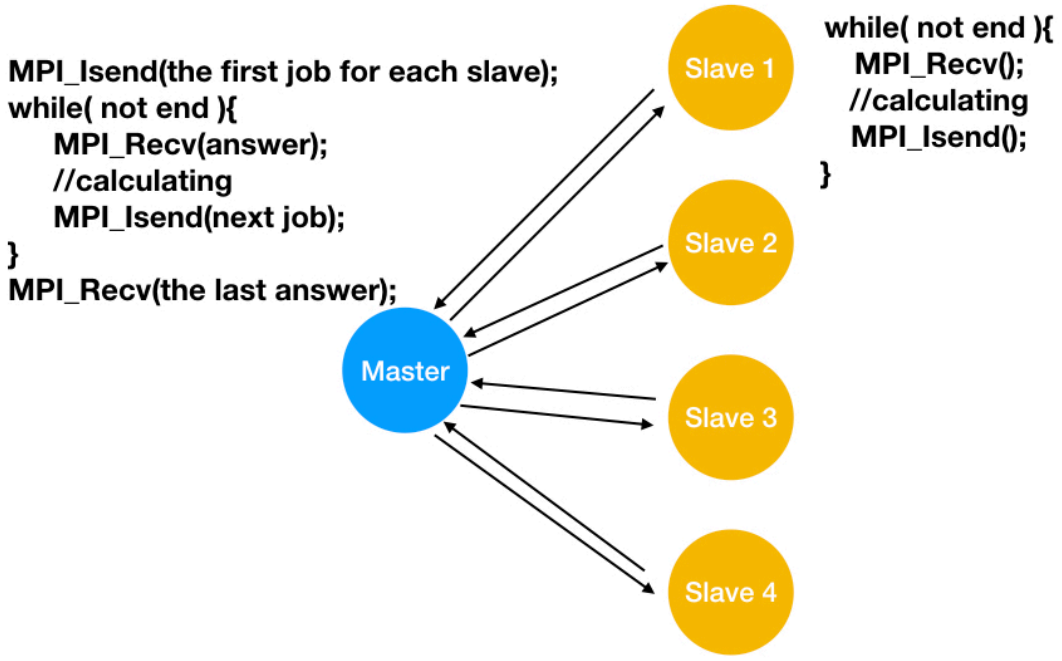


圖一、MPI_STATIC版本工作分配示意圖

C. MPI_Dynamic

在這個版本中需要動態分配工作，讓比較快完成工作的task幫忙比較慢的task，因此我使用master-slaves的模式，由master負責分配工作，每個slave一開始都會從master收到一個工作，此時master會進入迴圈等待slave傳遞訊息；在slave分別完成工作後，會將完成的答案傳給master，若master手上還有工作沒有被分配出去，則回傳新的工作給slave，若工作都分配完了，則傳訊息告訴所有slaves工作已全部完成，slave便結束所有運算，最後master再自行完成繪圖的工作。在傳遞訊息的部分，我都是使用MPI_Isend()，這可以避免task等待其它

task運算而被block太多時間，造成平行結果不佳（如圖二所示）；另外，我使用MPI_Isend()、MPI_Recv()中的MPI_TAG的欄位來判斷slave接收到的訊息及master傳出的訊息為下一個工作或是結束程式，並且利用MPI_Recv()中的MPI_Status的欄位得知master收到的訊息是來自哪一個slave，作為回傳訊息的目的端。



圖二、MPI_DYNAMIC傳遞訊息示意圖

D. Hybrid

在這個版本中，因為我的mpi_static效能比較好，因此我在mpi_static版本中的for迴圈前加上OpenMP的指令 #pragma omp parallel for schedule(dynamic, 64) ，進行更進一步地平行。

E. Others

1. 在網路上查資料的時候，發現當複數的實部 (x) 及虛部 (y) 滿足下列其中一個條件時，此複數必屬於Mandelbrot set，則直接將其repeat次數（經過幾次運算才發現不屬於Mandelbrot set）設為最大值，且可有效降低執行時間。

$$(x+1)^2 + y^2 < 0.0625 \qquad \text{(條件一)}$$

$$(x-0.25)^2 + y^2 < 0.5 \times \sqrt{(x-0.25)^2 + y^2} \qquad \text{(條件二)}$$

2. 在所有使用MPI的版本中，必須根據task數量決定各自需要處理的資料量。藉由MPI函式庫所提供的MPI_Comm_size(...)函式取得被建立的task數量量T，則每個 task必須處理 (width x height) /T個資料，此時將會遇到兩種問題:

   a) width x height小於T

   當此狀況發生時，表示有些task(s)會沒有被分到資料而空等，因此我選擇透過MPI_Group_range_incl(...)和MPI_Comm_create(...)函式將需要執行運算的 task(s)額外建立新的group和communicator，並且透過MPI_Finalize()將多餘的 task(s)終止。

b) N無法被T整除

在此狀況之下，表示資料無法被平均分配，我想到的作法有兩種。第一，將剩餘的資料全部分配給rank為T-1的task，但遇到T的值很大的時候，task T-1需要處理的資料量將會是其它task(s)的數倍，最後可能成為程式的瓶頸。第二種作法是將剩餘的資料再平均分配給前N mod T個task(s)，因此只有前幾個 task(s)會比其它的task(s)多一個資料，將會比第一種作法更加平均。因此我選擇使用第二種作法實作。

3.

II. Experiment & Analysis

A. Methodology

1. System Spec

使用課堂上提供之環境。

2. Performance Metrics

在實驗當中，我將測試資料中的圖片長寬皆設定為1000，因此我的問題大小為 1,000,000，而實部及虛部的範圍皆為 -2 ~ 2。

a) MPI_Static、MPI_Dynamic

利用MPI所提供的 MPI_Wtime()，在 MPI_Init()後面與 MPI_Finalize() 前記錄時間，並且扣除IO、溝通時間得到CPU time，做為後續分析 scalability 與 performance 的依據。 而在測量 Load balance 時，我只紀錄 mpi_static 或 mpi_dynamic 版本中計算複數是否為Mandelbrot set的部分，但有一個地方需要注意：計算的時間會隨著不同的 rank ID 而不同。因此，我將每個rank的計算時間做為 load balance 的分析依據。

b) OpenMP

在OpenMP中，我使用其所提供之API omp_get_wtime() 計算時間，在程式的開頭與結尾處加上這個指令，即可計算程式的執行時間，但我仍然有扣除處理IO的時間。而在做 Load Balance 的實驗時，我新增一個陣列，用來儲存每個thread的執行時間，OpenMP中有提供另一個API omp_get_thread_num() 來取得目前的 thread ID ，將每個thread的執行時間記錄在相對應index的位置中，以做後續分析。

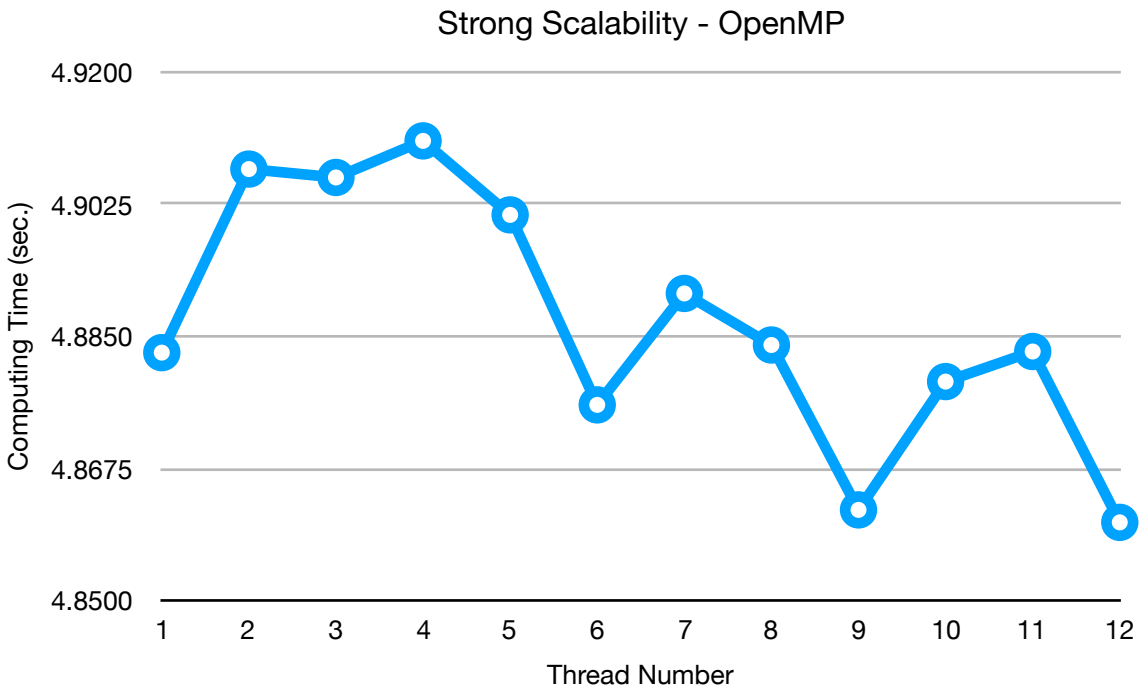
c) Hybrid

執行時間的計算方式與上面第a）項一致。在計算 load balance 的方式則與 OpenMP 相同，但不同的 rank ID 中可能會有相同的 thread ID，因此需要紀錄thread是來自於哪個task。

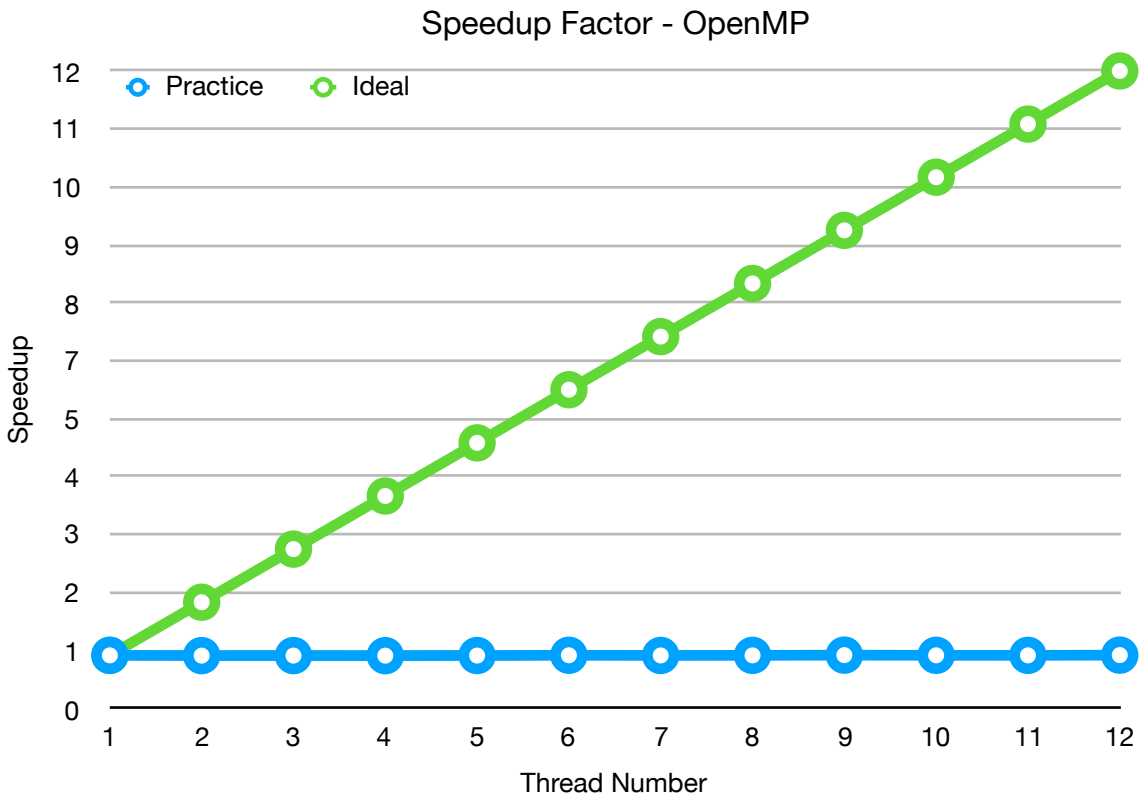
B. Scalability

1. OpenMP

在我的OpenMP版本中，Strong scalability 曲線跟預期中的很不一樣，computing time 沒有隨著 thread 的數量增加而增加，反而起伏很大。

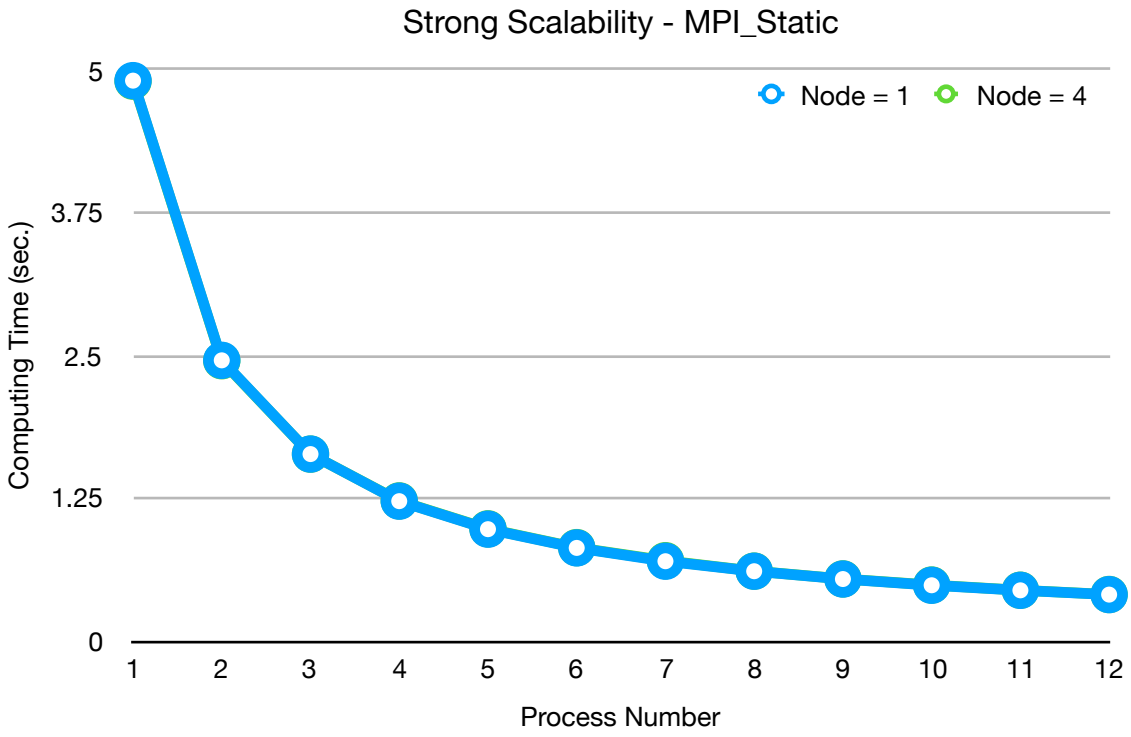


然而，在speedup factor曲線的表現也沒有很好，不僅如此，speedup可以說是沒有進步。

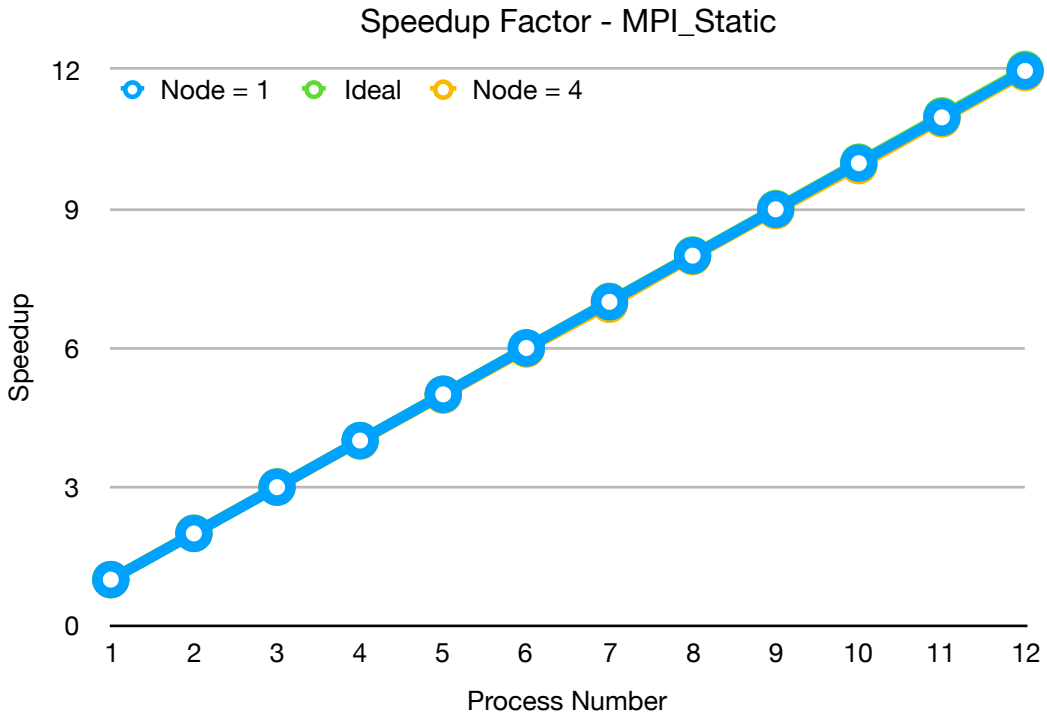


2. MPI_Static

由下圖可以得知：無論 Node 數量為何，當process數量增加，執行時間會隨之減少，由此可見，將程式平行化後可以帶來的加速效果很可觀，且兩條曲線幾乎一樣，我想Node不會影響加速效果。

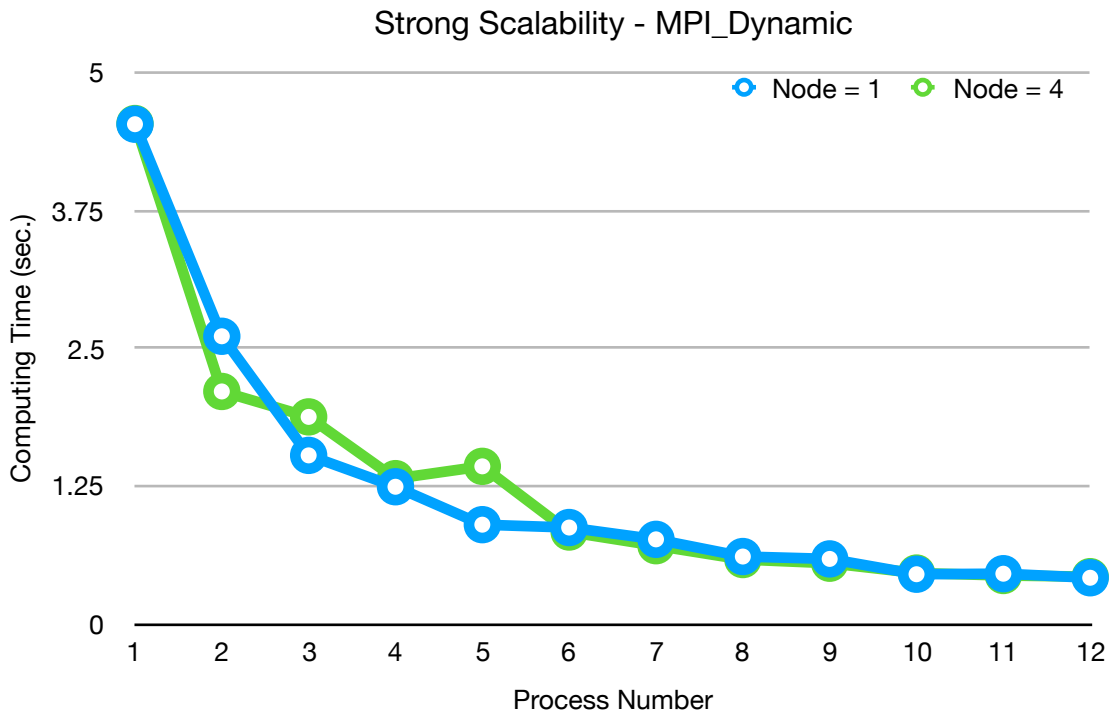


由下圖可以得知: 在 MPI_Static 版本中，無論node的數量為何，speedup factor 都會隨著process的數量上升而上升，實際曲線（藍色、黃色）與理理想曲線（綠色）幾乎完全相符。

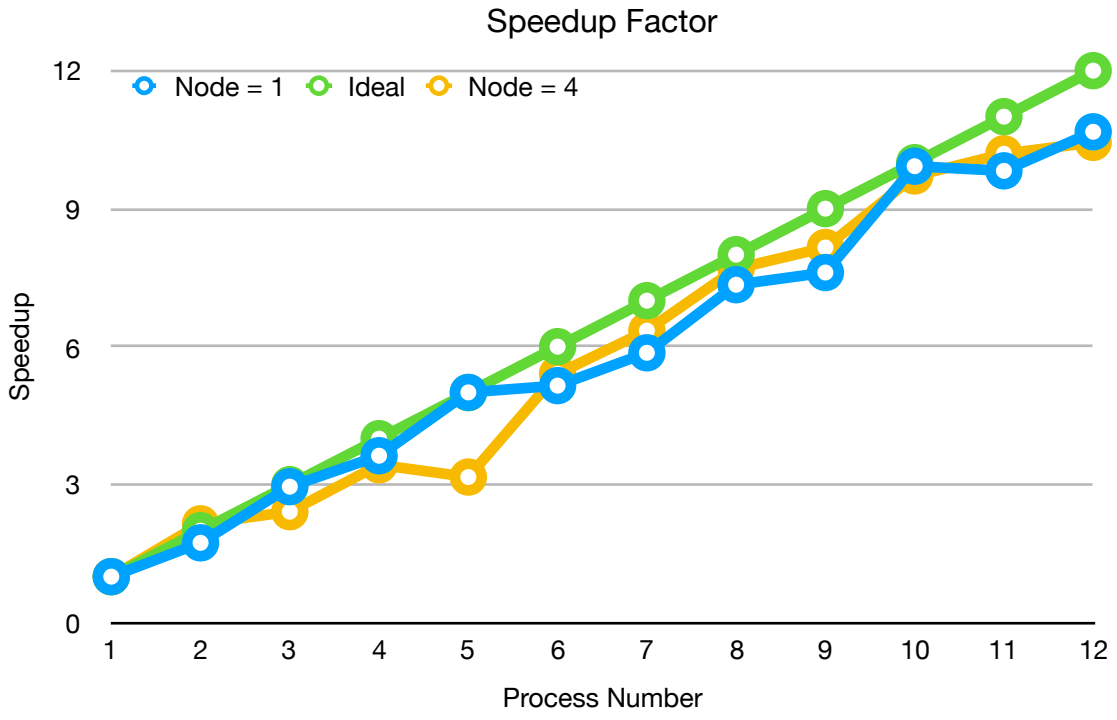


3. MPI_Dynamic

由下圖可以得知：當 Node 只有 1 個的時候（藍色），執行時間會隨著process數量增加穩定遞減，而 Node 增加到 4 個的時候（綠色），執行時間雖然會隨著process數量增加而減少，但在 process 數量為 3、4、5 的時候，執行時間會比藍色曲線還要大，在 process 數量為 2 的時候，執行時間會比藍色曲線小。

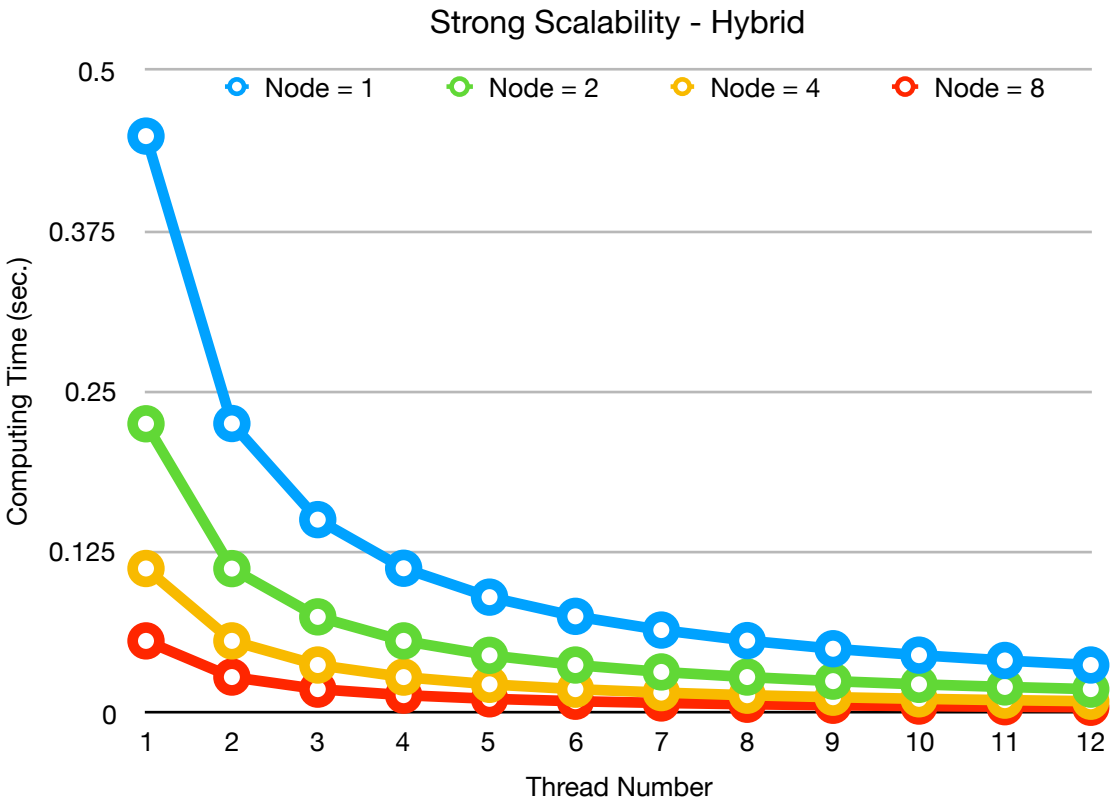


由下圖可以得知: 在 MPI_Dynamic 版本中，無論node的數量為何，speedup factor 都會隨著process的數量上升而上升，但實際曲線（藍色、黃色）與理理想曲線（綠色）還有一小段的差異，而 Node = 4 Process = 5的時候（黃色），加速效果卻不是很好。

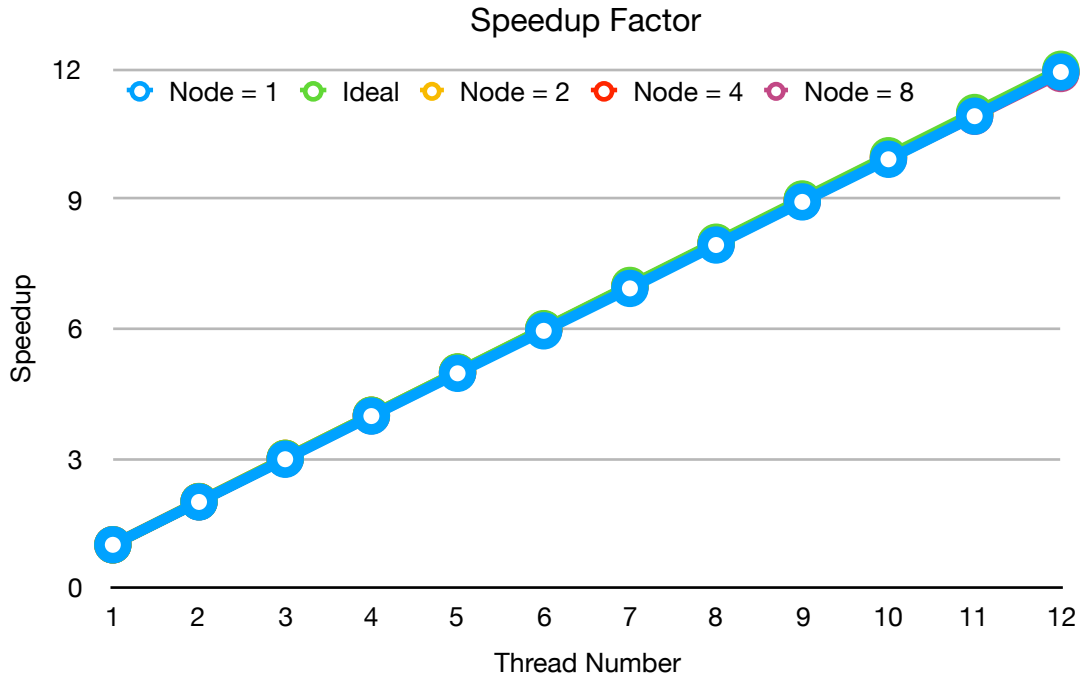


4. Hybrid

由下圖可以得知：無論 Node 數量為何，當 thread 的數量增加，執行時間會隨之減少，且同一個 thread 的數量下，執行時間也會隨著 Node 的數量增加而下降，由此可見，當 OpenMP 與 MPI_Static 並用時，會讓程式的加速效果更為顯著。

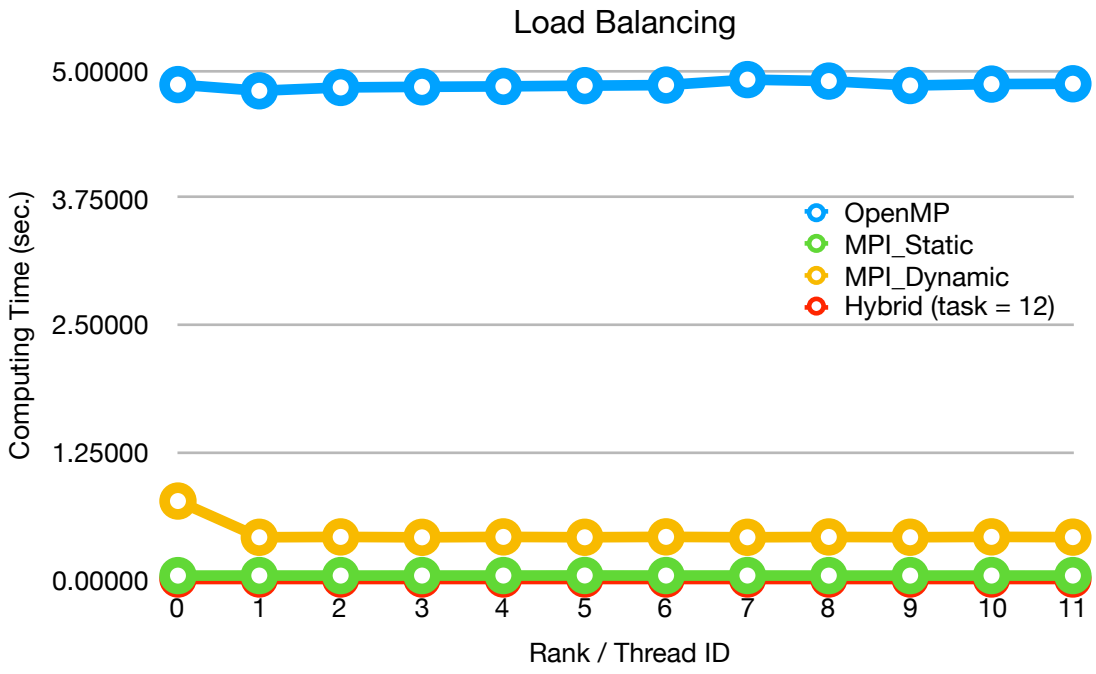


由下圖可以得知：在 Hybrid 版本中，無論node的數量為何，speedup factor 都會隨著process的數量上升而上升，且實際曲線（藍、黃、紅、紫色）與理理想曲線（綠色）幾乎完全相符。



C. Load Balancing

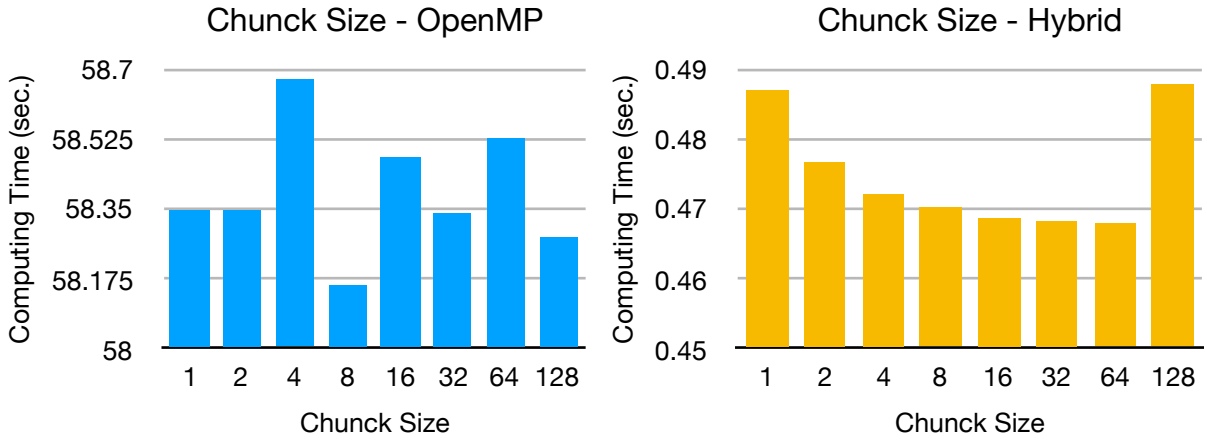
在我實作的四個版本當中， Load Balance 的分佈都很平均，因為我在分配工作的時候，有考慮到有些區塊的複數會有比較長的計算時間，所以在 OpenMP 版本中使用 dynamic scheduling 讓 load 比較輕的 thread 可以做更多的任務；而在有使用 MPI 的版本中，也平均分配好每個 task 的工作量，而由下圖可以清楚知道我的實作真的有達到 load balance 的效果。 但是在 MPI_Dynamic 的版本中，Rank 0 的 task的loading 相較其它 tasks 來說高了一些，因為在 Rank 0 的 task 部分負責收集答案並擺放到正確的 pixel 位置，所以需要處理的工作量會比其它的 taks 多一些，但整體來說是平均的。



D. Others

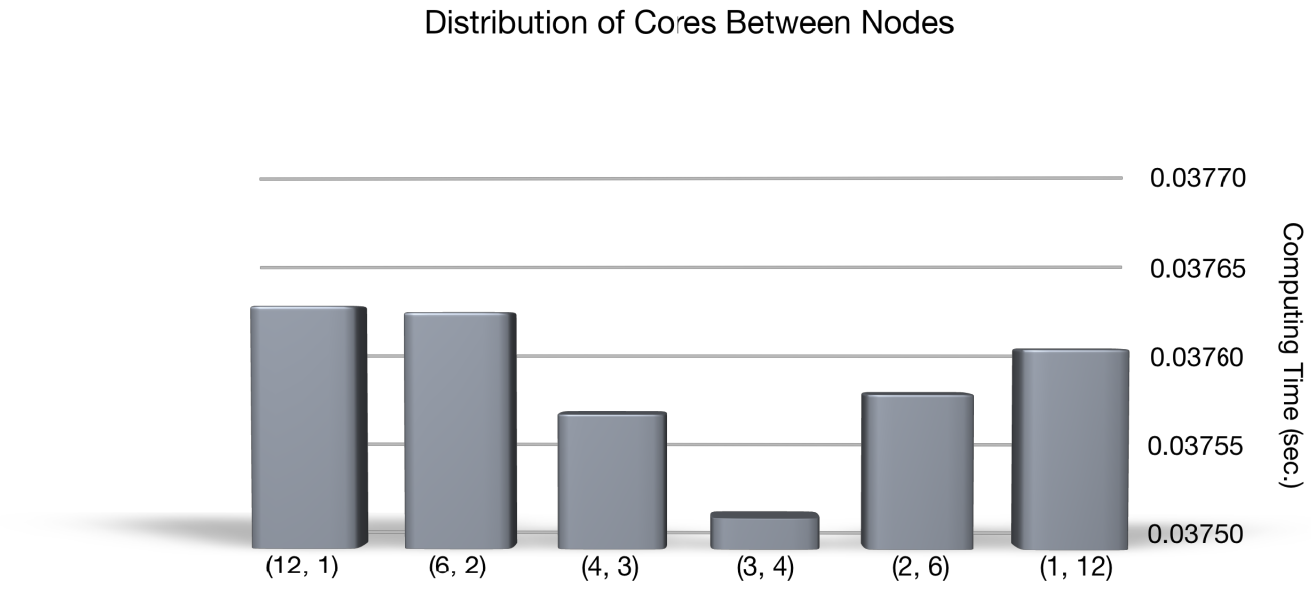
1. Chunck size analysis

我針對有使用 OpenMP 指令的兩個版本：OpenMP、Hybrid 進行 chunck size 的分析，希望可以找到一個 chunck size 可以達到最好的加速效果。在做這個實驗下，兩者皆是使用 12 個 threads；另外，Hybrid 也固定使用 12 個 nodes。實驗結果如下二圖所示，OpenMP 中使用chunck size 為 8 時會有最明顯的加速效果，Hybrid版本中則為 64。



2. Best Distribution of Cores Between Nodes

在這個實驗當中，固定 thread 的總數為 12，再分別執行得出 computing time 做比較，如下圖所示，當 core 和 node 的組合為 (3, 4) 時，加速的效果十分顯著，但仔細一看執行時間後發現：整體時間只有在小數點後三位有變化，可能是因為我的測試資料沒有非常龐大，所以沒有很大的差異。



Distribution of Cores Between Nodes

3. MPI_Static 兩種版本比較

一開始為了要讓 load balancing 有好一點的結果，我選擇將每個複數由左下往右上編號（如圖三所示），再將這些複數的編號做亂序排列（如圖四所示），接著將 Rank 0 的亂序結果傳給各個 task，若 task 共有 5 個，Rank 0 分到的複數編號為第0、10、12、6個，Rank 1 分到的複數編號為第4、9、13、8個，以此類推。各個 task 完成所有工作時，將結果回傳給 Rank 0，由其將答案還原至原始位置再製作圖片。

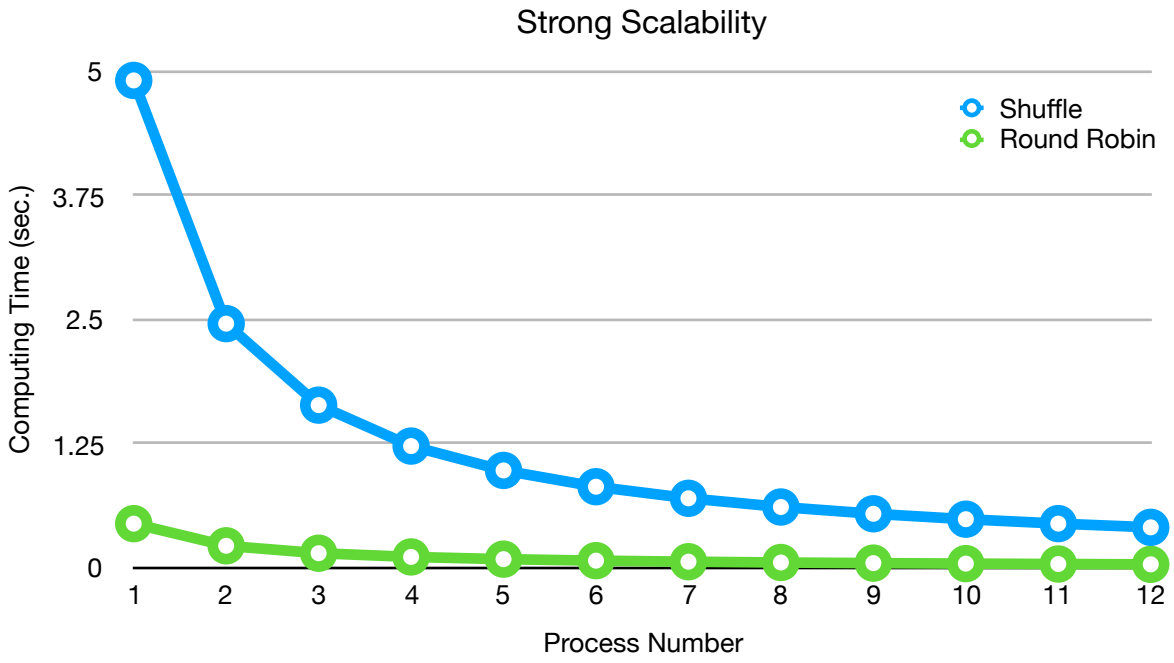
12	13	14	15	16	17
6	7	8	9	10	11
0	1	2	3	4	5

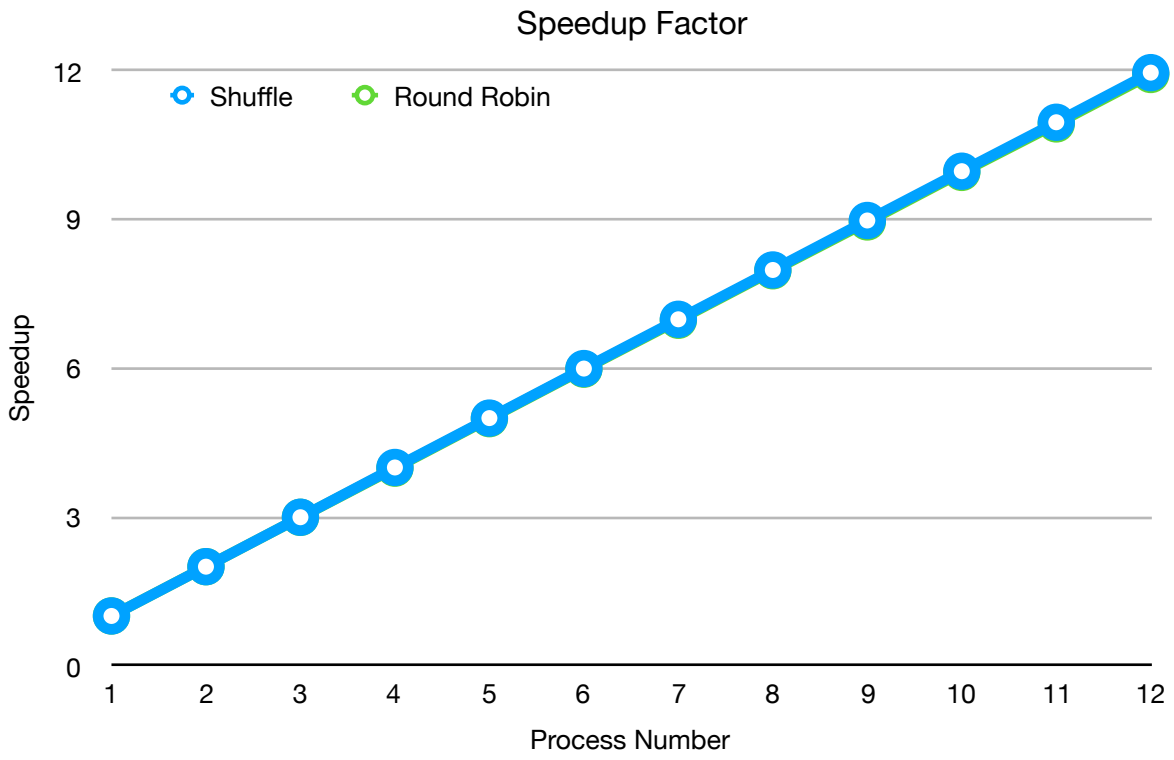
圖三、複數編號示意圖

15	2	11	7	1	14
13	8	5	16	3	17
0	10	12	6	4	9

圖四、複數亂序編排示意圖

但經過思考後覺得這個方法會比依序取資料還要慢，因為做完亂序後需要將結果傳給其它 tasks，會增加一次的資料交換，造成程式的瓶頸。為了證實自己的想法正確，我將兩者程式執行時間做比較，如下圖所示，使用亂序的方法真的會慢許多，strong scalability 及 speedup factor 的結果也不錯，但整體時間來說少了許多。


Strong Scalability
Shuffle
Round Robin
Computing Time (sec.)
5
3.75
2.5
1.25
0
1
2
3
4
5
6
7
8
9
10
11
12
Process Number


Speedup Factor
Shuffle
Round Robin
Speedup
12
9
6
3
0
1
2
3
4
5
6
7
8
9
10
11
12
Process Number

III. Experimence & Conclusion

在 OpenMP 的版本中，雖然不同的 chunck size 會影響整體的效能，但如果只是做這方面的調整，整體的執行時間並不會因此而改善許多，可能是還有可以平行的地方，因為我在處理平行程式的經驗尚有不足之處，不太明白該從何下手吧。

另外，在做 MPI_Static 版本時，原本是用 MPI_AllReduce 將答案收集起來，經過與實驗室的同學們討論後，發現用 MPI_Gatherv 就可以收集每個 task 的答案到一個指定的 task 中，如此一來，task 之間花費在溝通的時間就會少一些，而改善程式整體效能。

整體來說，這次的作業在實作上算是容易的，但同學們的實作能力都很厲害，讓我每天都會從計分板上降個 1~2 名，也因為有記分板即時顯示大概的名次，讓我這個求勝心很強的人有想要再努力改善程式的動力。