Parallel Programming –Mandelbrot Set

106062530 張原嘉

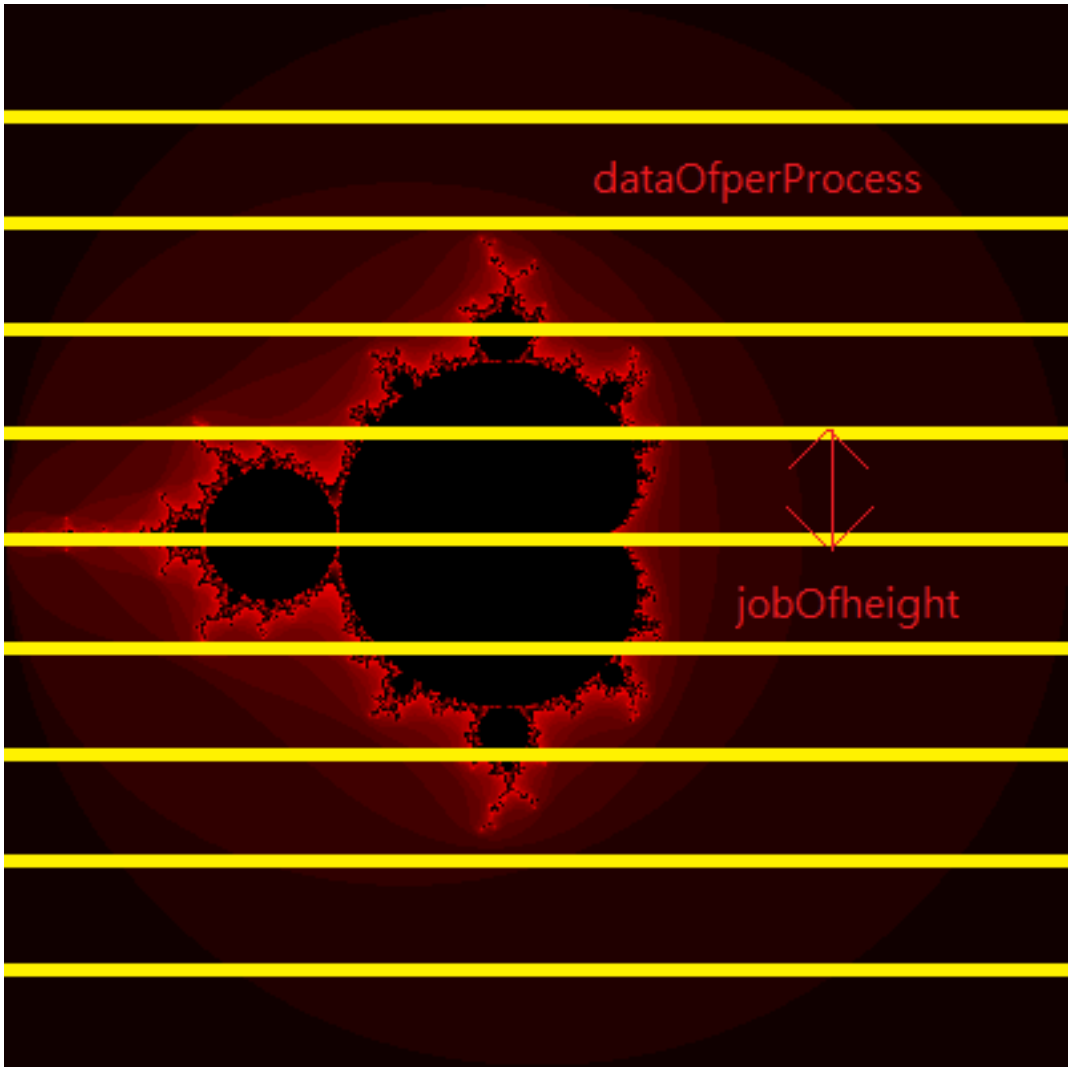
1. Implementation.

Implement、Partition、Reduce exec. Time 、Other efforts

MPI_Static：

在這個版本，主要採用 Row-major 方式切割整張圖，首先先把輸入參數中的 height 除以 numOfProcess，得出每個 Job 的高度，也就是 jobOfheight（當然，有時會有餘數，此時只要把每個 Job 的高度+1）接著乘以輸入參數的 width，即得每個 Process 的工作量（dataOfperProcess），利用此變數當作 Process-Buffer 的大小，接著進入 mandelbrotSet（ ）。此函數用其接收參數 jobOfheight 來當作每個 Process 開始計算的位置指標，因此函數中 y0 所用到的 jobOfheight，即是用來計算 Offset

((j+(jobOfheight * rankID))* ((upper - lower) / height) + lower;)



待每個 Process 做完計算並存入自己的 Buffer 後，使用 MPI_Gather（ ）來收集所有資料到 Master 的 Buffer 以進行繪圖。此函數最大的優點即是方便易用。

MPI_Dynamic：

此版本採 Master-Slave 的 Centralized work pool 架構來實現工作動態分配，利用 RankID 為 0 的 Master 處理工作分配，其餘 Slave Process 進行計算。因此，與 Static 最大不同在於 Master Process 並不會參與運算。程式首先檢查 Process 數量是否為 1，若為 1 當然沒有 Master/Slave 之分，因此即進行 Sequential 版的計算。若大於，則用 MPI_Send（ ）& MPI_Recv（ ）進行溝通。至於 Master 該如何識別當下的 Slave 是閒置、完成計算、

不再工作？MPI_ANY_TAG（dataTag、resultTag、terminateTag）、MPI_ANY_SOURCE 可讓 Master 知道，只要有 Process 還在 Send，便可根據 Source 以及 Tag 來決定 Slave 的下一步。當然，寫圖部分還是Master 自行寫入。資料切割方面，Master 一次送一整個 Row 給 Slave 進行計算。

OpenMP_Dynamic：

此版本是將 Sequential code 加上 OpenMP 關鍵字來實作。資料方面採用 Shared memory 以及 Private memory 管理。值得一提的地方在於，計算 Mandelbrot-Set 使用的兩層迴圈，可用 collapse(2)達到＂點＂的平行化，此舉可讓程式執行時更加平行化，藉以增加其效能。

Hybrid_Dynamic：

此版本為，在 MPI_Dynamic 基礎上加入 OpenMPI 的 API，此舉可讓 Slave Process 用多個 Thread 執行。

2. Experiment & Analysis

Methodology、*Scalability & Load Balancing*、*Discussion*、*Others*

System Spec.

課程提供之設備：

Intel(R) Xeon(R) CPU X5670 @ 2.93GHz、96GB memory、 2 x 6-core

Performance Metrics

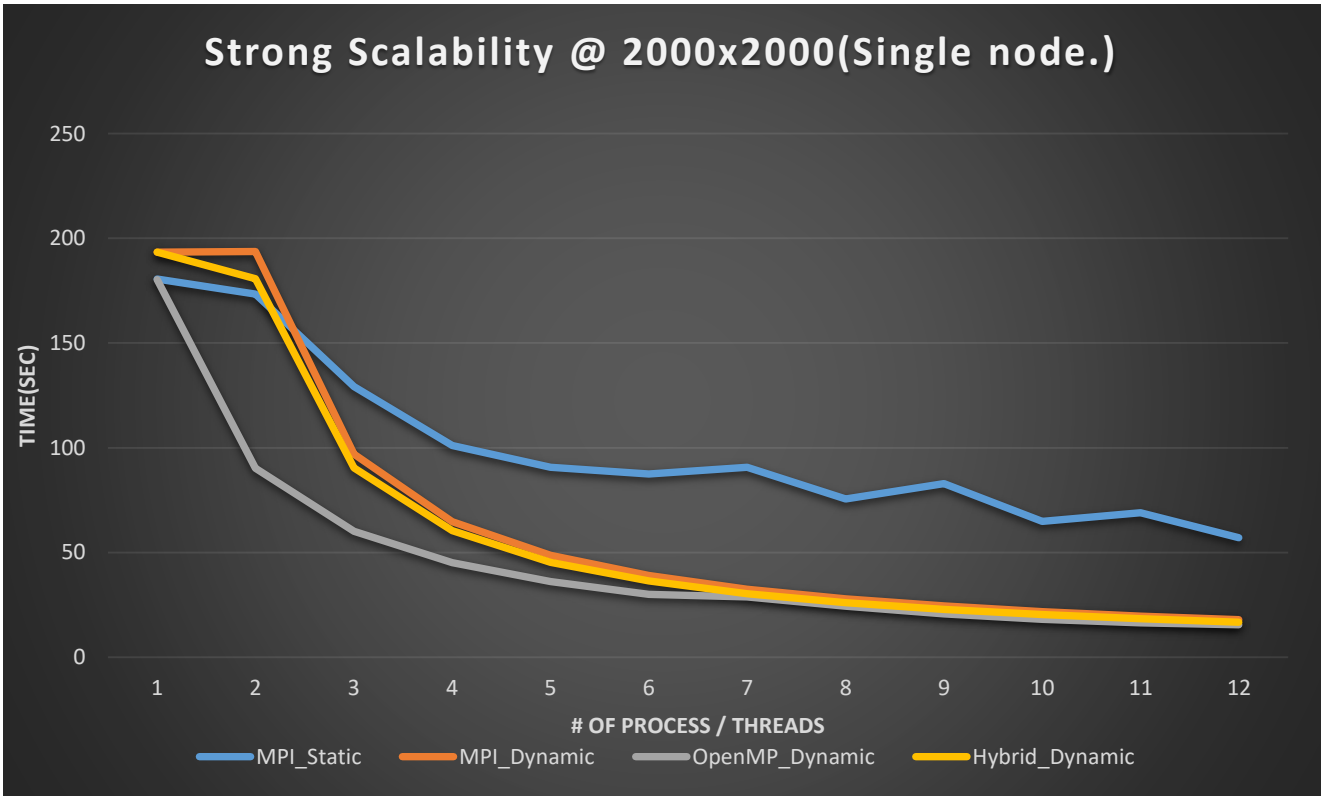
MPI 部分採用 MPI library 提供之 MPI_Wtime()進行時間紀錄，將指令放於 MPI_Init()後與 MPI_Finalize()前，兩者相減即為執行時間，單位為秒。Load Balance 針對計算資料的部分進行時間紀錄，方法為在 while 迴圈前後插入

指令，並用累加方式做紀錄，每個 RankID 有自己的值。

OMP 部分採用 OMP library 提供之 omp_get_wtime()進行時間紀錄，在主程式開始前及主程式結束後分別插入相應指令，即獲得執行時間，單位為秒。Load Balance 則用一個以 Thread 數為大小的陣列，累加紀錄平行化後每次的計算時間，而 omp_get_thread_num（ ）就是用來得知紀錄 Load Balance 陣列要開多大。

Hybrid 部分大致與 MPI、OMP 雷同，但輸出 thread 前，要多輸出 rankID 以分辨其來自哪個 Process。

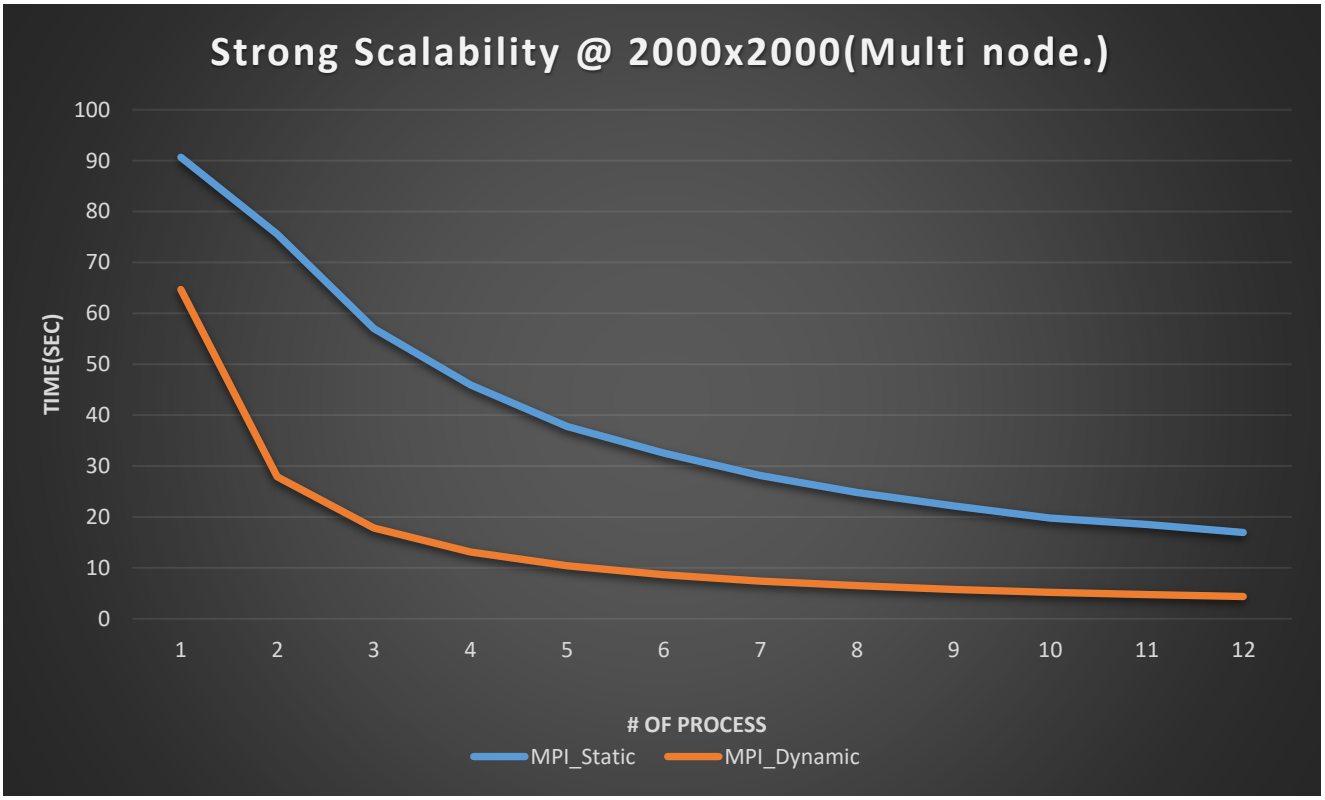
Strong Scalability（若無說明皆為 2000x2000）



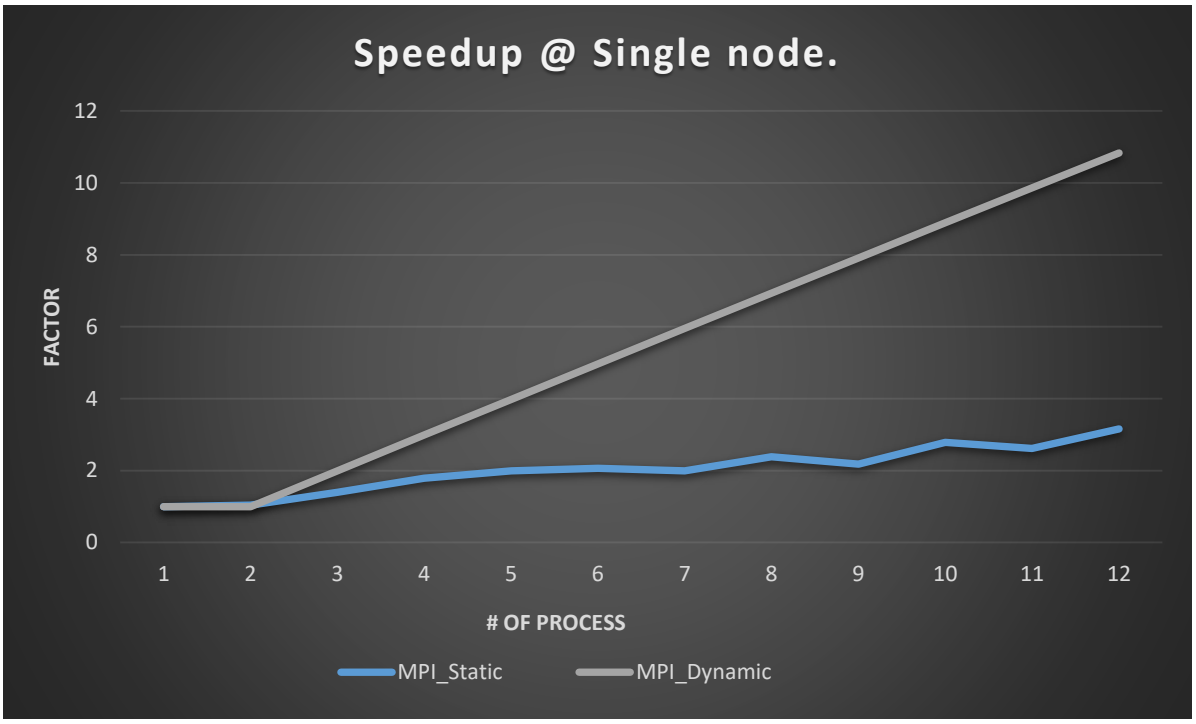
此圖中為在大 N 固定為 1，Process/Threads 從 1~12 的變化中，四種版本其執行時間的變化。不難發現 Dynamic 因本身有較好的 Loading Balance，因此其執行時間隨著 Process 數增加時，會比 Static 執行時間還要來得短許多。另個有趣的地方在，Dynamic 因採 Master-Slave，所以當 Process

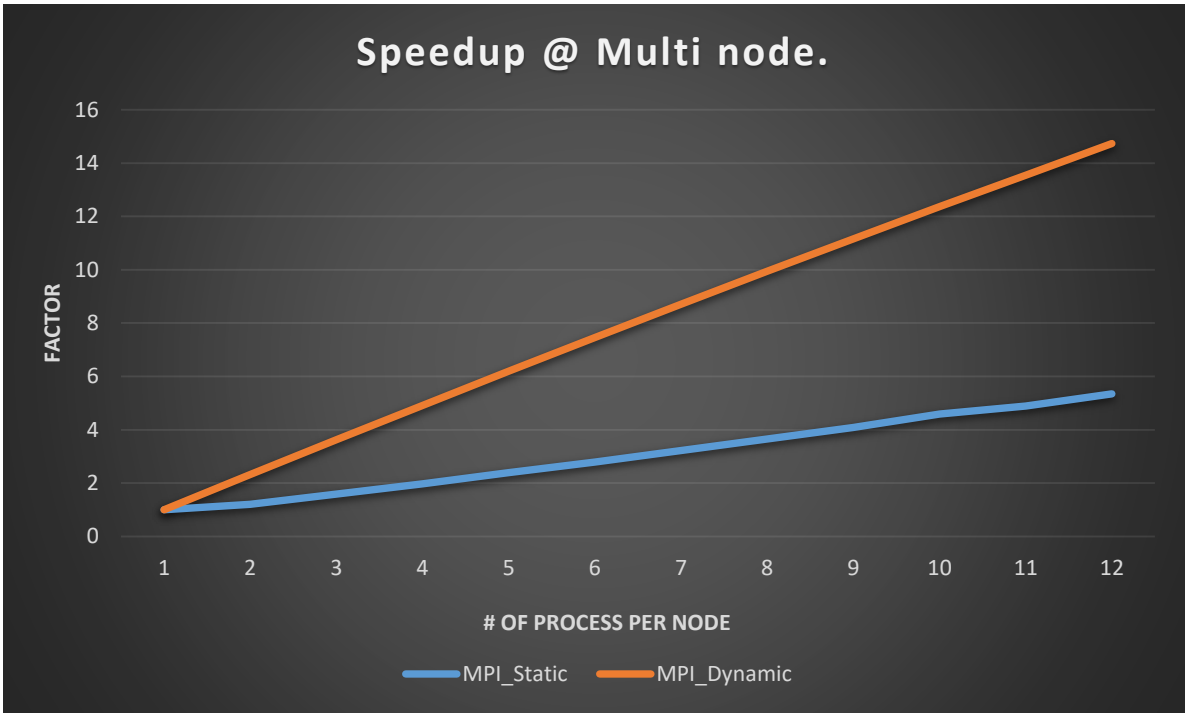
為 2 時，此兩個 Process 須相互溝通，而此動作頗花時間，因此說明了其執行時間和 Process 為 1 時極相同，也說明了此時還看不出 Centralized work pool 帶來的效益，待 Process 數大於 3 時才有顯而易見的改善。

以下 Speedup 圖亦能看出，不論是在 SingleNode 或 MultiNode，Dynamic 的 Speedup 成長幅度總是遠大於 Static 的成長幅度。此皆說明了做好 loading balance 對 Performance 有很大的正面效益。

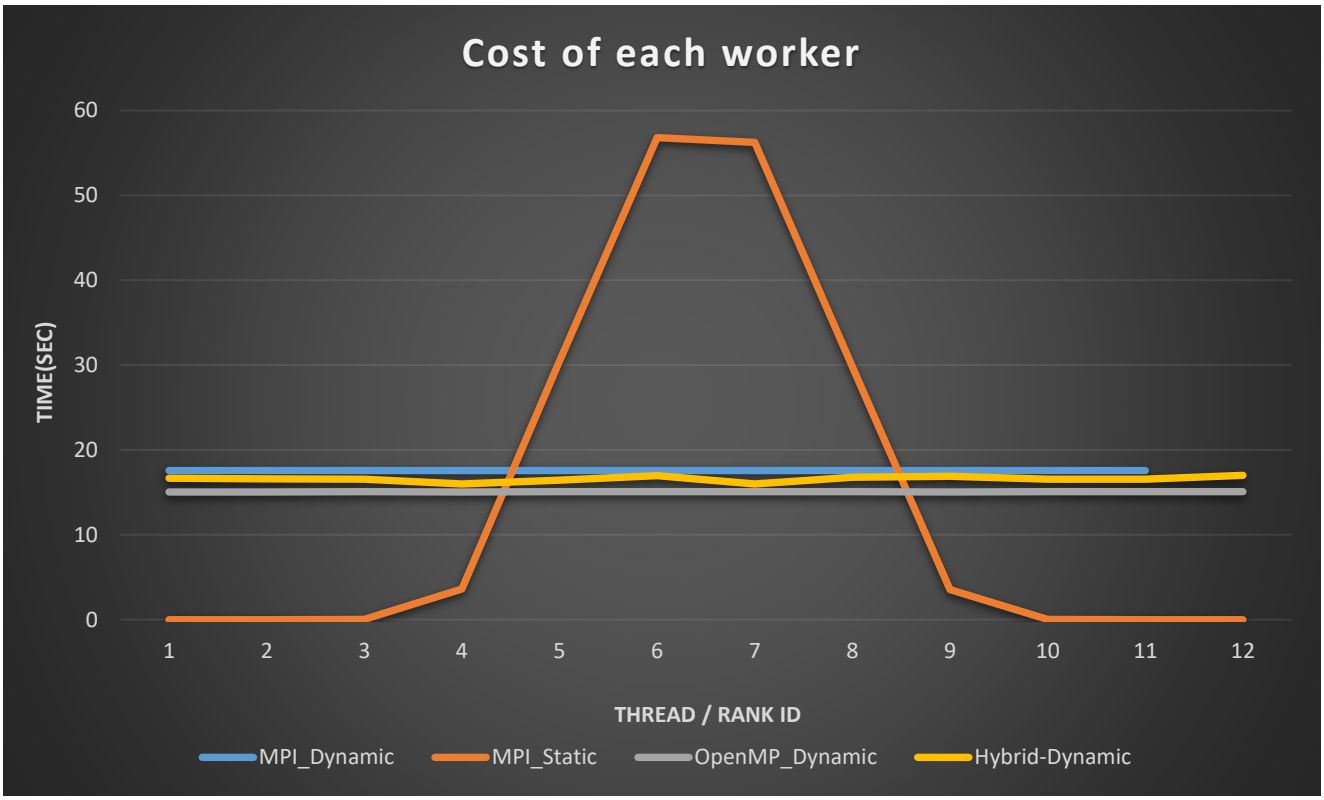


Speedup


Speedup @ Single node.
12
10
8
6
4
2
0
FACTOR
1
2
3
4
5
6
7
8
9
10
11
12
OF PROCESS
MPI_Static
MPI_Dynamic


Speedup @ Multi node.
16
14
12
10
8
6
4
2
0
FACTOR
1
2
3
4
5
6
7
8
9
10
11
12
OF PROCESS PER NODE
MPI_Static
MPI_Dynamic

Loading Balance



從圖中可看出，由於 Static 版本是採 Row-major 方式分配資料，實驗結果得知這種方式會讓有些 Process 有很多工作，而有很些很少，可見這不是一個好的切割方式。比起 Static，其他版本的 load balance 都還不錯，每個 worker 接收的資料都很平均，也就不會有某些 worker 執行時間過久，或有些 worker 有冗於等待的情況。當然，Load balance 是各版本的執行時間形成差異之關鍵。

3. Experience / Conclusion

在 N 等於 1 時，OpenMP 顯而易見是效能最好的。隨著 Node 數、Process 數增加，OpenMP 是否還是最好就需要與其他版本做比較，其好處是，對剛進入平行程式的新手來說，是一個易用的東西，但仍需要與 MPI 作結合以達到跨 Node 的溝通需求。另一方面，從 MPI_Static 與 MPI_Dynamic 可清楚瞭解到 Loading Balance 的重要性，不好的 Work load 會降低程式的 Performance，

如何讓工作量在各 Process 間盡量平衡，是需要多著墨的地方。總之，此次作業讓我對平行程式又有了更深且更廣的了解。