HW 2: Mandelbrot Set

105062600 Yi-cheng,Chao 04:49, November 22, 2016

Design Concepts – MPI_static

1. 雖然這次的總 size 數幾乎不會大於要處理的程式數目(也就是 hw1 的 N 值，在這次 hw2 則是#points of X)，但是為了防止特別的例子(例如#points of {X, Y} = {10, 1000}的情況)，因此我還是用了 MPI_Group_range_excl 來防止特殊例子產生，以便保證我可以在特殊例子發生時仍然能保持正常的分配。
2. 資源分配的話因為 Mandelbrot Set 正常情況輸出(X-axis range == Y-axis range && #points of X == #points of Y)在 X 軸超過一定數字以後幾乎都是很少的 iteration 數就會發散，因此我#points X 不能整除 size 數的部分通通給予最後一個 rank 做處理，以每條 column 為單位做迴圈的計算每個 pixel 的 iteration 數。
3. 如同我 hw1 的 tail 和 head 的方法，我使用 x_start 與 x_end 去紀錄每個 rank 所分配到的 column 起始和結束位置，每次運算完後把 iteration 數放入以 row-major 計算陣列位置的 local_buf 中，再用 MPI_Reduce 到 root_buf，如此便可以在最後一個 rank 利用陣列位置 decode 取得該著色的 pixel 位置資訊。
4. 此外我之後有寫了一個對記憶體的使用比較節省的版本，主要做法為使用最後一個 rank 做接收 x-axis location, y-axis location 與 iteration 數三者的接收，每個 MPI 程式的 buffer 使用上從兩個 total pixel 的大小降到一個 3 integers 大小的 buffer 而已，犧牲一個 rank 的 performance 也較佳，因此最後上傳版本為此版本。

Design Concepts – MPI_dynamic

1. 概念類似 MPI_static 版本，我也使用了 MPI_Group_range_excl 來防止特殊例子的產生，以便保證我可以在特殊例子發生時能保持正常的分配，但對動態分配而言的重點則比較偏向可以提早釋放我用不到的資源而不會讓處理器空轉。
2. 因為是動態分配，因此我使用 master-slave 的 centralized work pool model，使用 rank0 當作 master processor 處理工作的分配，而其餘 processor 做 slave procsessor 做 iteration 的計算。
3. Master processor 主要工作為只要還有 slave processor 在執行就不斷使用 MPI_Recv()接收結果，我使用 MPI_ANY_SOURCE 以及 MPI_ANY_TAG，讓 Master processor 只要有任何 slave processor 做完工作便可以接收結果，再去解析 sourceID 以及 TAG 屬性來去判斷是否要使用內容還有再送出工作。
4. Slave processor 在一開始會送出 BEGIN_TAG 告知 Master processor 需要工作，如果有執行運算結果則是送出整條 column 的 iteration 數以及 column number 並且標記 DONE_TAG 讓 master processor 知道這是要著色的 column，每個 slave processor 終止條件為如果收到來自 master processor 的 TERMINATION_TAG 則跳出迴圈結束計算。

Design Concepts – OpenMP_static & OpenMP_dynamic

1. OpenMP 版本因為是採用 shared memory 以及 private memory 來處理資料，只要設定好 private memory 要管理的值，直接在運算的 for loop 做 for schedule(static)即可完成靜態分配，並且因為平行化的部分只在 pragma omp parallel 內，因此還可以做到平行的進行 pixel 的著色。
2. 但因為 XSetForeground 以及 XDrawPoint 在執行比較大數據的時候有時會產生錯誤，因此我使用了 LOCK OpenMP Routine 做 Critical Section Design 防止 Xwindow 的錯誤。
3. dynamic 版本僅與 static 版本差別在 for schedule(dynamic, 1)的使用，交給系統去做 work pool 的 loading balance。

Design Concepts – Hybrid_static & Hybrid_dynamic

1. 主要架構都是 based on MPI version，只是在執行計算時的 for loop 加上各自版本的 OpenMP API。

Experiment & Analysis

System Environment

執行程式使用課程提供的 Batch Cluster。

Time Measurement

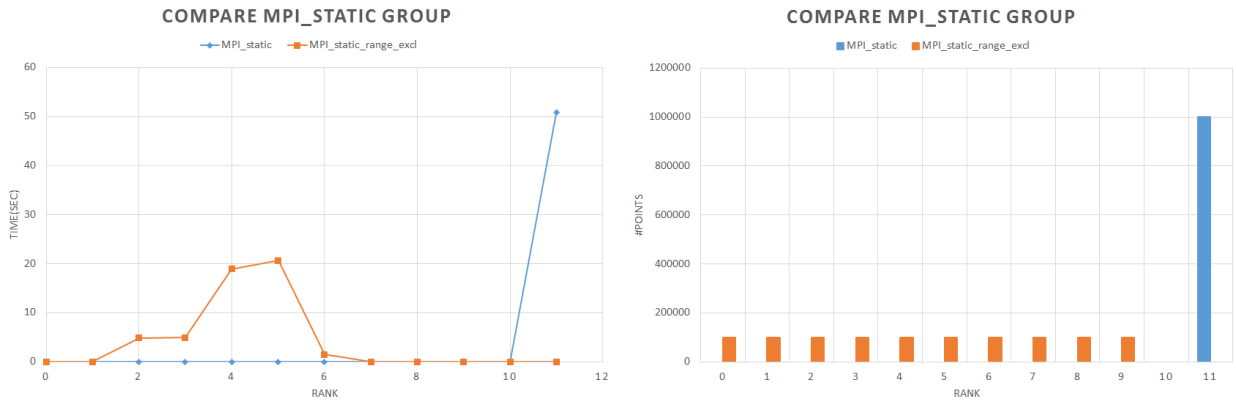
指令部分使用<sys/time.h>library 的 gettimeofday()函數來得到時間，精度可達微秒。

Performance Measurement

測資隨我的分析方法而改變，因此列在每個測資的下方。

MPI Group Range Excluding Analysis- Static

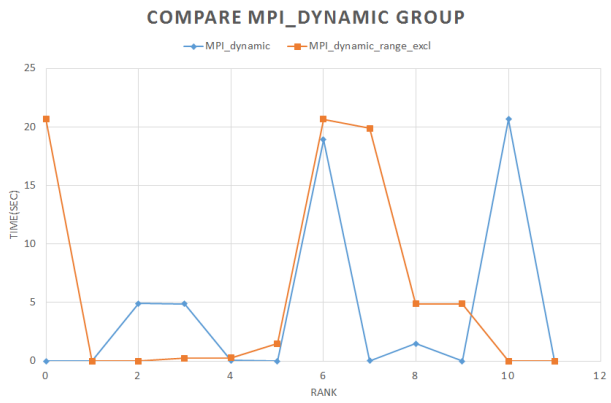
Input parameter: mpirun –n 12 ./MS_MPI_static 2 -2 2 -2 2 100 100000 disable

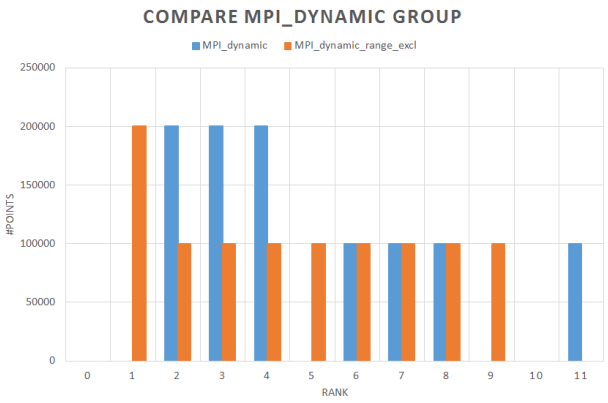


因為我分配 task 的方法是把工作都往最後一 Rank 送，因此在取 N/size 的 floor 時如果極端的例子(N < size)發生時會變成 sequential 版本而完全沒有平行化到，因此如果使用了 MPI_Group_range_excl()指令後可以使用適當大小的 processor 就好，不只大幅縮短 execution time，也可以提早 release 不使用的 processor，如上右圖所示，Rank10 與 Rank11 是不存在於執行 range_excl 後的版本中的。

MPI Group Range Excluding Analysis- Dyanmic

Input parameter: mpirun –n 12 ./MS_MPI_dynamic 2 -2 2 -2 2 100 100000 disable





然而 dynamic 版本對因為動態分配本身就有較好的 loading balance，performance 從圖上看起來反而 range_excl 版本還沒有更好，但主要好處還是在於可以釋放不執行的 processor 以便讓其他使用者使用而不霸佔資源。

Strong Scalability Analysis

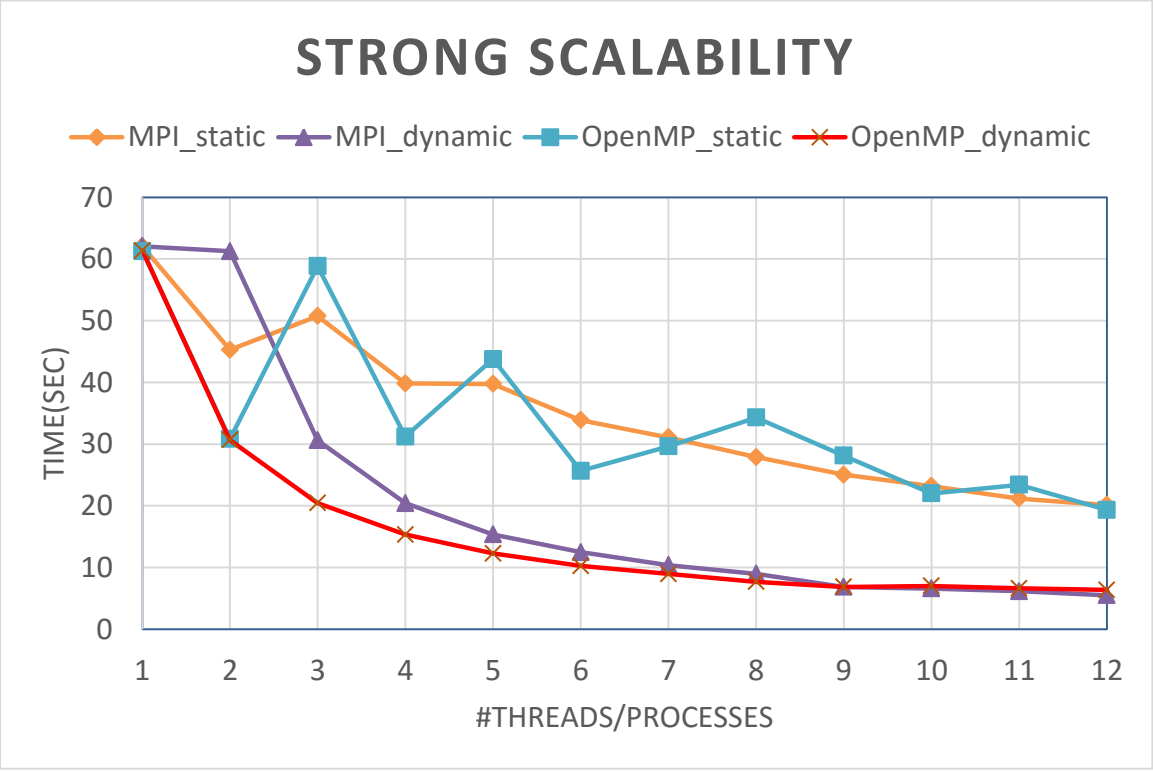
(x-axis: # of Threads/Processes; y-axis: Execution time (s))

Input parameter:

mpirun –n (1~12) ./executable 12 -2 2 -2 2 1000 1000 disable (for MPI)

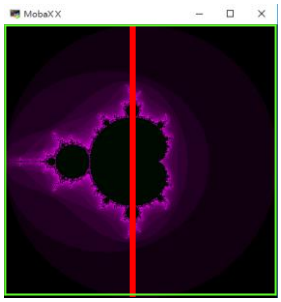
./executable (1~12) -2 2 -2 2 1000 1000 disable (for OpenMP)

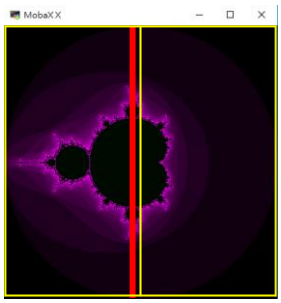
Scalability to number of cores(Problem size is fixed)，{MPI, OpenMP}x{static, dynamic}

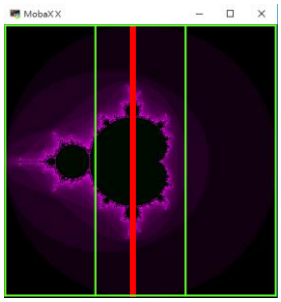


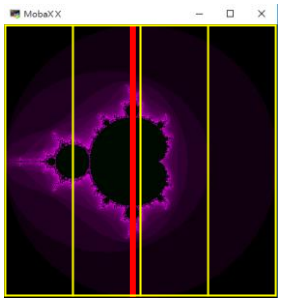
在兩者的 dynamic 版本因為有較好的 loading balance，所以可以看到比較符合理想中的 strong

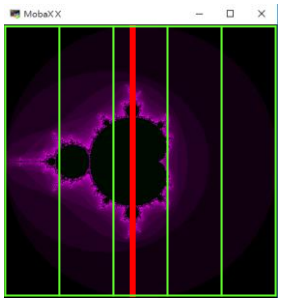
scalability，比較特別的地方則是在因為 MPI_dynamic 版本需要切割一個 processor 做工作指派的 master processor，因此在#processes=2 的時候 execution time 幾乎沒有減少，而且在#threads/processes 小於 9 以前都可以看到 MPI_dynamic 的#threads/processes=N 執行時間幾近等於 OpenMP_dynamic 的#threads/processes=N-1 的執行時間，而這點非常符合預期。
並且對 dynamic 版本而言，performance 方面是 OpenMP 優於 MPI 版本的。
而 static 版本則非常跳動，performance 在 OpenMP 版本和 MPI 版本也都互有優劣，整體儘管還是有隨著#threads/processes 越大執行時間而減少的趨勢，但可以發現會有#threads/processes=odd number 的 execution time 大於#threads/processes=even number 的情況，原因在於我的切割方式是以 column 為單位切割，而分配成奇數個 processor 時會會因為 loading 分配較偶數時差，故才會產生這樣的結果。
如果我們分開解析兩個 static 版本的 Loading balance，紅線為以 column 為單位分配的 bottleneck line，基本上可以理解離 bottleneck line 越近基本上所需要的計算越多，越遠則越少，切割後如下組圖所示，綠框為偶數為單位切割，黃框則為奇數為單位切割：

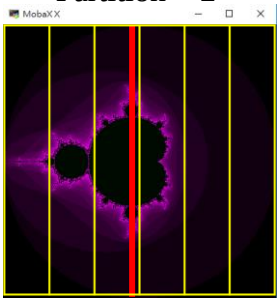

Partition = 1

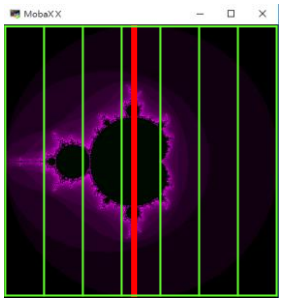

Partition = 2

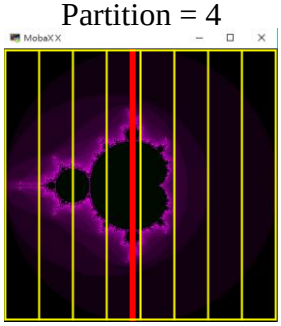

Partition = 3

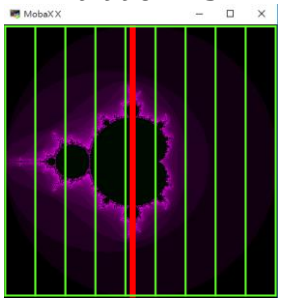

Partition = 4

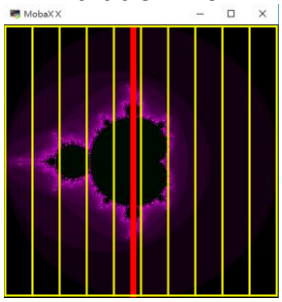

Partition = 5

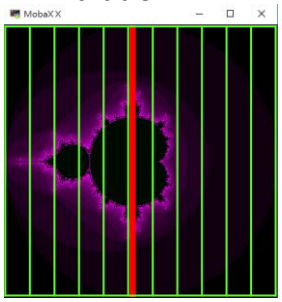

Partition = 6

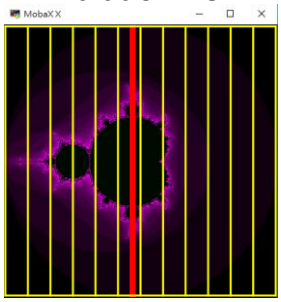

Partition = 7


Partition = 8


Partition = 9

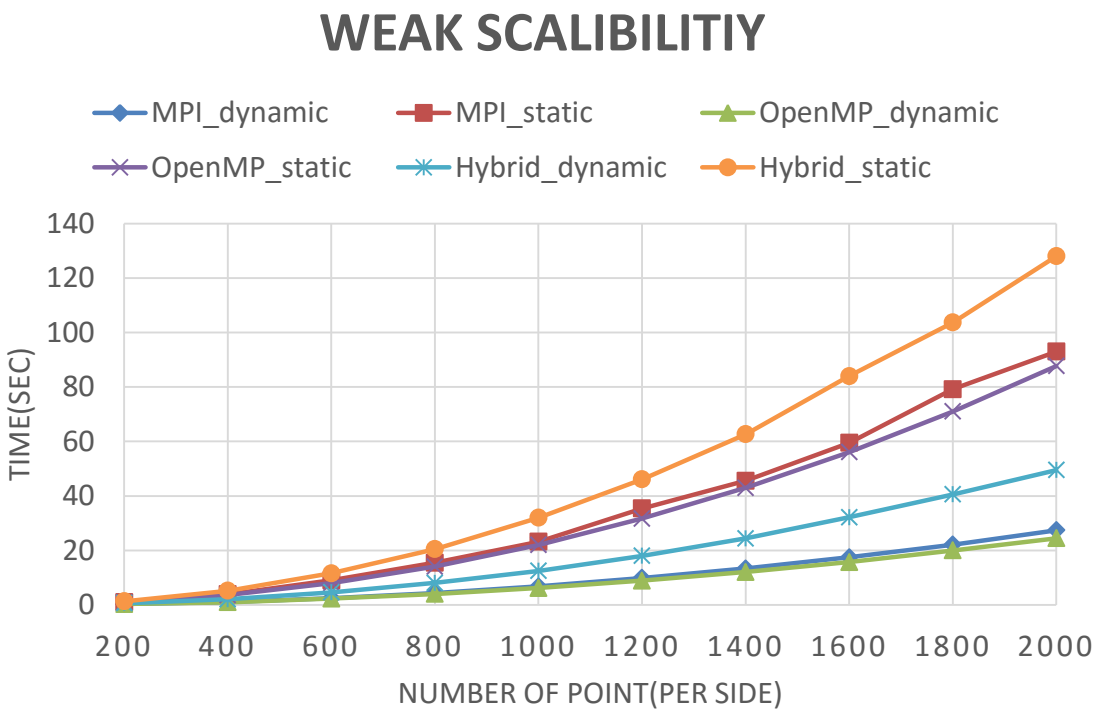

Partition = 10


Partition = 11


Partition = 12

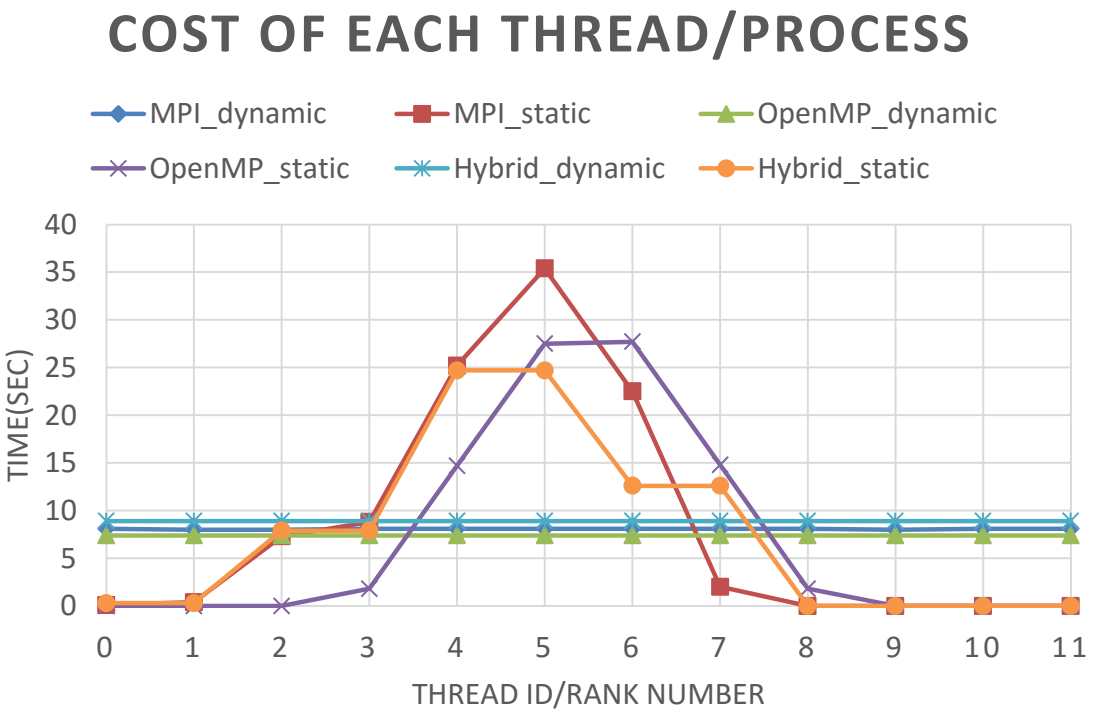
我們可以看到在#threads/processes < 9時bottleneck line都會切過奇數分配中間column的中間，導致中間 coloum 所分配到的 loading 過重，這點在 partition = 3 時尤其明顯，而偶數分配時，bottleneck line 恰好都會落在被偶數分配中間兩個 column 的分隔線，因此要成偶數 column 的 loading balaence 較奇數 column 的 loading balance 更好，才造成如此跳動的情況。

Weak Scalability Analysis
(x-axis: # of Point Per Side; y-axis: Execution time (s))
Input parameter:
mpirun –n 10 ./executable 10 -2 2 -2 2 (200~2000) (200~2000) disable (for MPI)
./executable 10 -2 2 -2 2 (200~2000) (200~2000) disable (for OpenMP)
mpirun –n 1 –ppn 2 ./executable 5 -2 2 -2 2 (200~2000) (200~2000) disable (for Hybrid)
Scalability to problem size (# cores is fixed)，{MPI, OpenMP, Hybrid}x{static, dynamic}



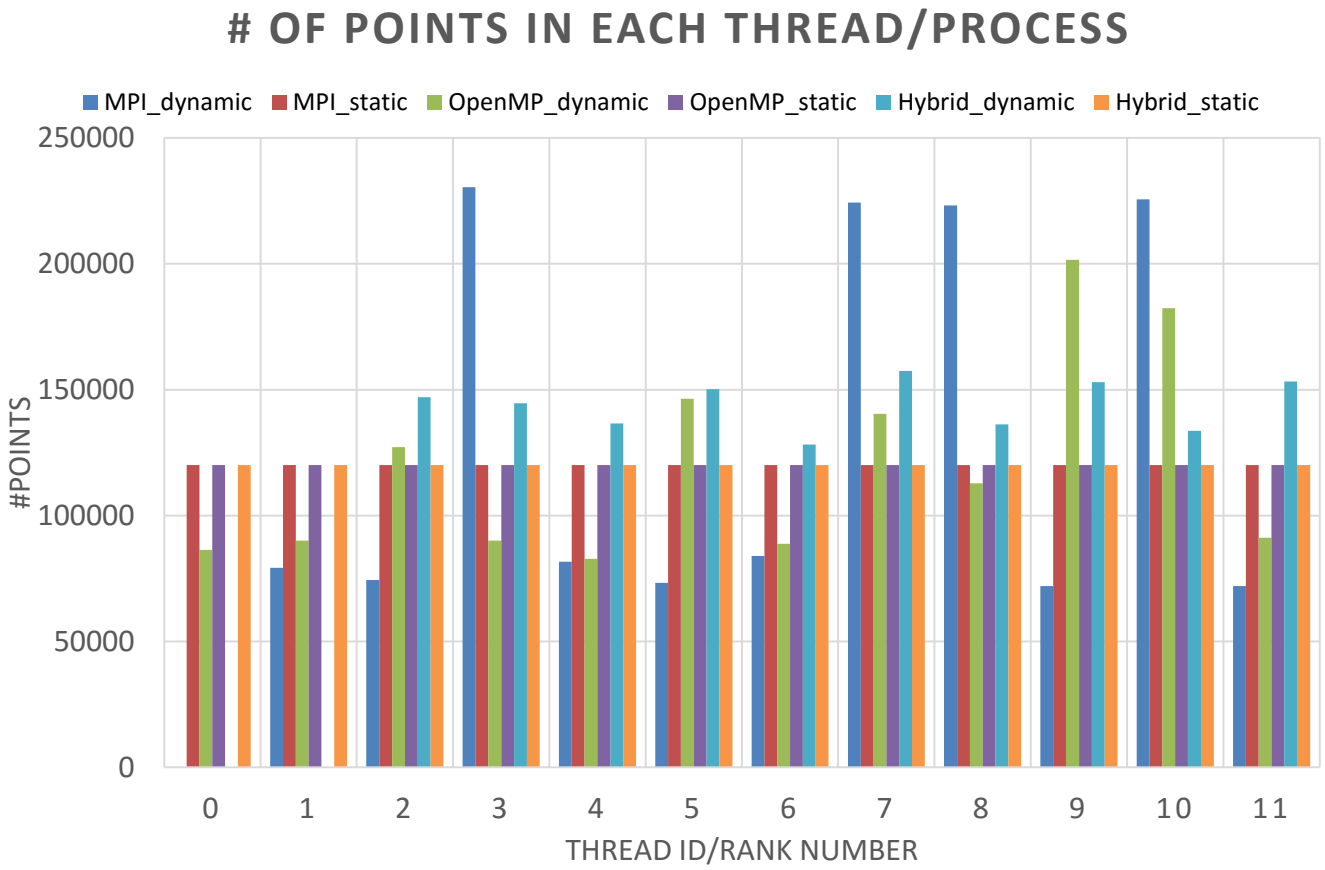
Weak scalibility 的部分我使用 problem size 的複雜度為 $O(N^2)$，因此期望會看到向上凹的指數成長曲線，六者執行時間大小為 Hybrid_static > MPI_static > OpenMP_static > Hybrid_dynamic > MPI_dynamic > OpenMP_dynamic，資源分配版本而言，因為 dynamic 版本 loading balance 較 static 版本好很多，因此 performance 較佳是很直觀的，結果也很符合預期，而 Hybrid 版本卻都比 MPI 以及 OpenMP 版本慢的原因我推測是來自我的寫法上使用單一 Rank 做 master processor 做資源的分配，因此對於每個版本儘管#cores(#threads * #MPI tasks)都一樣，但 OpenMP 因為資源分配交由程式 API 直接 schedule，因此實際可供平行的#cores = 10，而 MPI 版本會犧牲掉一個 Rank 做 master processor，實際可供平行的#cores = 9，Hybrid 版本則會犧牲一個 Rank 外加此 Rank 內的 thread，實際可供平行的#cores = 5，明顯少於其他兩者版本，dynamic 版本可以清楚看到 Hybrid 執行時間幾乎都是 OpenMP 版本及 MPI 版本的兩倍，也印證我的結論。

Loading Balaence Analysis
(x-axis : Thread ID/ MPI Task ID; y-axis : Execution time (s))
Input parameter:
mpirun –n 12 ./executable 12 -2 2 -2 2 1200 1200 disable (for MPI)
./executable 12 -2 2 -2 2 1200 1200 disable (for OpenMP)
mpirun –n 3 –ppn 2 ./executable 2 -2 2 -2 2 1200 1200 disable (for Hybrid)
Measure the computation time for each thread/process，{MPI, OpenMP, Hybrid}x{static, dynamic}



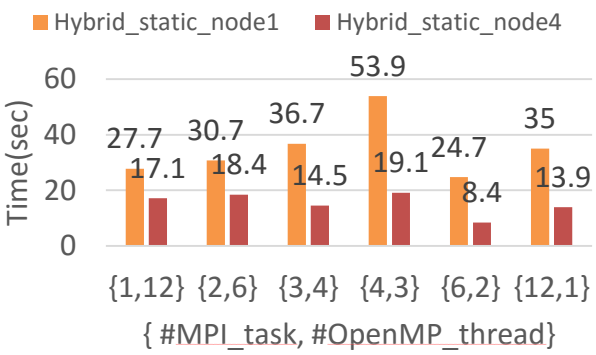
從 loading balance 的圖來看便明顯看到 dynamic 版本與 static 版本的差別，這也是造成兩者的執行時間差別這麼大的原因，因此好的 loading balance 會大大減少 bottleneck，對於 performance 有很大的正面效益。
此外因為我的 OpenMP 版本是從 Row 去進行 static 的 schedule，因此所呈現的山峰圖較其他兩者版本不同，是非常平均的分配，這也是因為 Mandelbrot Set 在虛數的部分在計算是否超過 escape radius 時正負號都會因為平方抵銷掉，因此是一個對 X 軸做線對稱的圖形，也造成 Cost of each thread 會呈現如此對稱的原因。

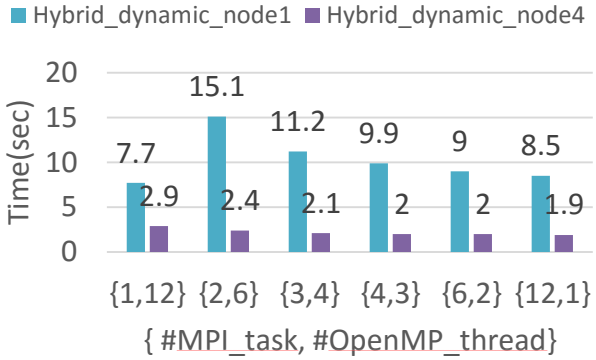
Loading Balaence Analysis
(x-axis : Thread ID/ MPI Task ID; y-axis : # of Point)
Input parameter:
mpirun –n 12 ./executable 12 -2 2 -2 2 1200 1200 disable (for MPI)
./executable 12 -2 2 -2 2 1200 1200 disable (for OpenMP)
mpirun –n 3 –ppn 2 ./executable 2 -2 2 -2 2 1200 1200 disable (for Hybrid)
Measure the computation points for each thread/process，{MPI, OpenMP, Hybrid}x{static, dynamic}



根據上圖的結論也可以看到六種不同版本的分配到的 pixel 數的差別，因為每個 pixel 所需要的計算量不一樣，因此 dynamic 版本的每個 core 的 pixel 總數會非常不平均，而 static 則都是平均分配的，值得注意提的地方是因為 MPI_dynamic 以及 Hybrid_dynamic 版本的 Rank0 都是作為 master processor 僅做 task 的分配，因此不會有任何的 #pixels。

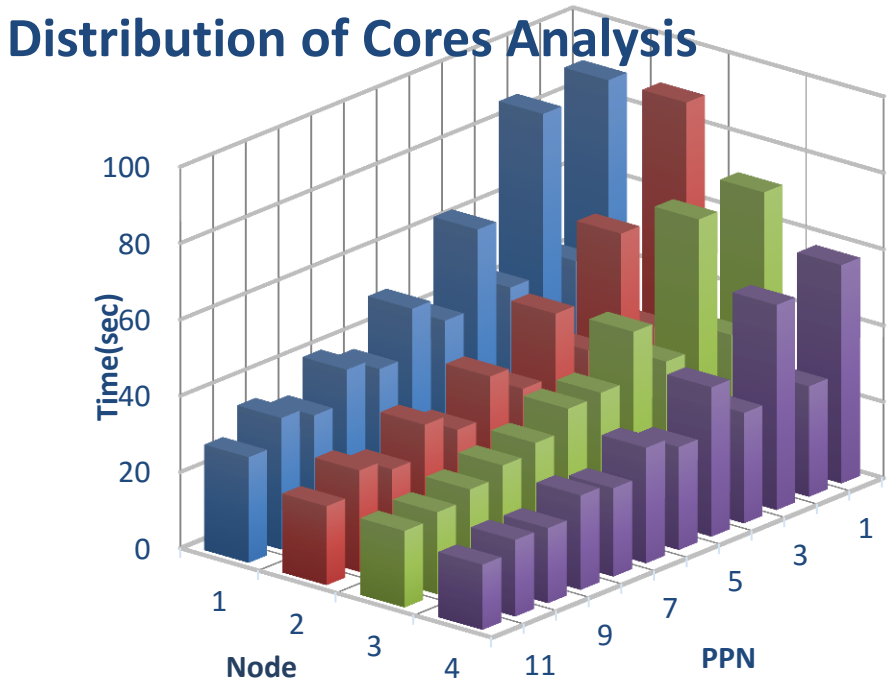
MPI Tasks & Threads Distribution Analysis – Hybrid_Static & Dynamic
(x-axis : {#MPI_task, #OpenMP_thread}; y-axis : Execution time)
Input parameter: #cores = 12; x/y-axis range=[-2, 2]; #point in x/y-axis= 1200;

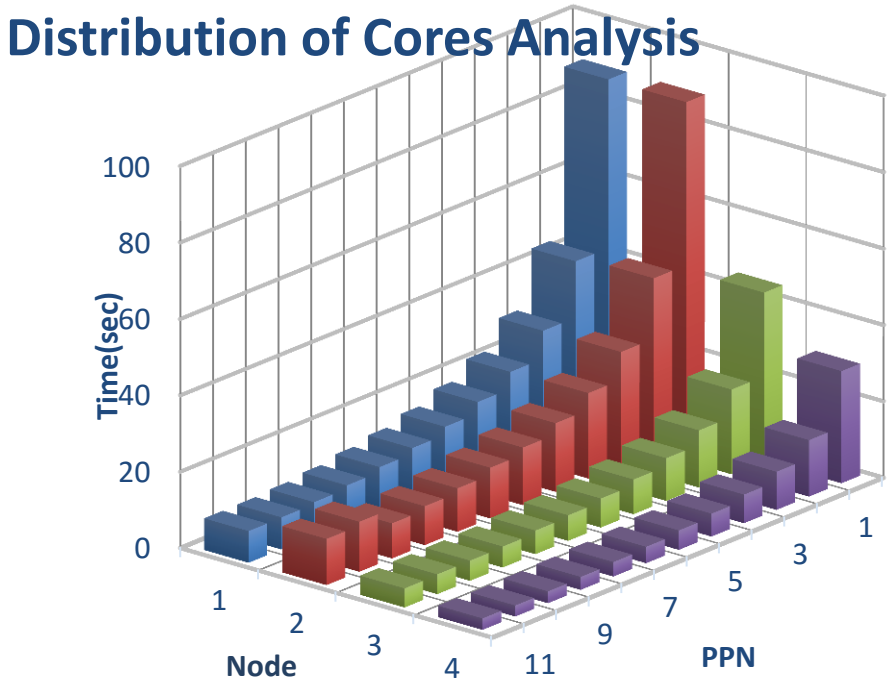




由於 Hybrid 版本也可以執行純 MPI 版本與純 OpenMP 版本，對於每組組合，根據圖形的結果大致上可以知道對於如果只有一個 node 而言，純 OpenMP 會有較高的 performance，而如果使用 4 個 node 的話，每個 node 內只有 Thread 則會表現的比每個 node 內只有 MPI_process 差，此外對於我的 Hybrid_static 版本，最佳配置為每個 node 內有 6 個 MPI_task 而每個 Task 擁有 2 條 thread，而 Hybrid_dynamic 版本則是 12 個 MPI_task 每個 task 擁有 1 條 thread 可以擁有最好的 performance。

Cores Distribution Analysis – Hybrid_Static & Dynamic
(x-axis : Threads per Nodes; y-axis : #Nodes; z-axis: Execution time)
Input parameter: #cores = 1 ~ 48; x/y-axis range=[-2, 2]; #point in x/y-axis= 1200;
因為 node 內的 process 數還可以分配成 process 或 thread，我統一使用 node 內都使用 thread。





Static 版本由於前面說到的奇偶數切割時的 Loading Balance 問題導致上下跳動幅度很大，但整體仍然可以看到因為總使用的#core 增加而增加 performance 的狀況，而 dynamic 版本的話則是無論 X 軸與 Y 軸都可以看到不錯的 strong scalibilty，只是也會隨著 ppn 的增加而會趨近平緩，因此我認為若在有限的資源下，Node 與 PPN 的平均分配應該是最好的配置方法了！

Conclusion & Experience

如果很直觀的想像，因為 OpenMP 是使用 shared/private memory 來溝通，因此會比使用 distributed memory 的架構還需要使用 MPI I/O 來進行溝通快很多，這點在 node=1 的時候是符合的，但在 node=4 的情況我們可以看出來把所有 ppn 都拿去做 thread 並不會比較好，我的認為可能是因為寫法上的問題產生不會比較好的結果，不過 OpenMP 真的是滿 powerful 的，程式也可以自動做 scheduling，最大的缺點大概就是不能跨平台吧！因此才需要 Hybrid 版本做平台與平台間的溝通！

此外在送 Job 的時候我發現如果我的指令使用如下

```
time mpiexec -ppn $NUM_MPI_PROCESS_PER_NODE ./MS_Hybrid_static $OMP_NUM_THREADS -2 2 -2 2 600 600 disable
```

在使用#PBS -l nodes={2, 3, 4}:ppn=1 的時候如果我維持 NUM_MPI_PROCESS_PER_NODE=1 export OMP_NUM_THREADS=1 的話程式都只會用到一個 processor(也就是變成 sequential 版本的處理時間)，不知道有沒有指令上的輸入錯誤，還請助教有空解惑！