Parallel Programming HW3 Report

102062209 邱政凱

1. Design

以下大致說明我對於六種演算法的實作方式：

MPI_Static:

使用跟講義上相同的架構，也就是由一個 Master Process 負責把 task Partition 好之後傳送給其他的 Slave Process。並且 master 不斷地 Recv slave 傳回來的結果(repeat 次數以及位置)，並且用 x window 把結果畫出來。每個 Slave Process 會在一開始接受到指定要計算的 row 的數目以及範圍，每次 Slave Process 算完一個位置的 Mandelbrot set 的 iteration 次數之後都會馬上把結果(repeats)、x 軸位置、y 軸位置回傳給 Master Process。

```
if(i<numY%slaveNum){
    rowCount = (numY/slaveNum) + 1;
}
else{
    rowCount =  (numY/slaveNum) ;
}
```

切給每個 Process 的 task 數的方法大致如上，是以 row 為單位來劃分每個 slave process 的 tasks。如果 rank<numY(row 的數目)%(slave process 數)的話就會多被分配到一個 row。也就是說每個 Process 會分到 row 的數目最多只會差 1。而 row 的分配是連續的，也就是說例如 rowCount=100，slave process1 會拿 0~100(row)，slave process2 會拿 100~200(row)、slave process 會拿 200~300(row)…。

MPI_Dynamic:

Dynamic 的部分同樣使用跟講義上相同的架構(centralized)，也就是由 rank 0 的 process 當作 master，然後同樣以 row 為單位劃分每個 slave process 的工作。整個動態分配工作的步驟大致如下：

1. 每當 slave 完成一個 row 的計算，就把當前的 row index 以及該 row 上的每一個點的 repeats 次數值一起回傳給 master。
2. Master 接收 slave 傳來的資料，並且確認目前是否還有未完成計算的 row，有的話就把下一個 row index 傳回去給該 slave process，沒有的話就傳-1 給 slave process。之後 master process 把 repeats 值用 x window 畫出來，並且重新回到 Recv 任何 slave process 回傳資料的等待狀態。

3. Slave Process 如果收到非負的 row index，則繼續計算該 row index 上每一個點的 repeats 值。如果收到-1 的話就結束所有工作離開迴圈。

OpenMP_Static:

由於 OpenMP 跟 MPI 不同，是屬於 shared memory model 的語言，因此我的做法就是直接開一個雙層的迴圈，並且用 OpenMP 去指名用靜待的方式分派以 row 為單位的工作給每個 thread。

```
#pragma omp parallel num_threads(threadNum) private(i, j)
{
    #pragma omp for schedule(static) nowait
    for(i=0;i<numY; i++) {
        for(j=0; j<numX; j++) {

            // DO MANDELBROT SET CALCULATION
            #pragma omp critical
                {
                    // Draw Points using X Window
                }

        }

    }
}
```

架構大致如上所示，用 OpenMP 的 for directive 來幫助平行的計算。可以看到由於我沒有指明 collapse clause，因此工作的平行化只限於外層迴圈，也就是 row 的迴圈。另外就是我的 schedule clause 沒有指名 chunkSize，因此每個 thread 要計算的 row 就是 continuous 地平均分配下去，也就是說大致上會跟我的 MPI_Static 分配 row 的方式相似。(chunk size 的影響我會用 extra 實驗來說明)

另外要注意的一點是 Mandelbrot set 本體的計算要使用到的變數都必須在 parallel directive 裡才宣告，因為這些變數是必須讓每個 thread 各自保有的 local variable，若每有特別指名的話會變成 shared variable，會導致錯誤的結果。

另外還有一點值得注意的就是由於 I/O 的資源必須要注意 synchronization 的問題，並能讓每個 thread 同時使用 xwindow 的函式，因此如果要畫點的話必須要把相關的程式碼包在 critical section 裡。

OpenMP_Dynamic:

```
#pragma omp parallel num_threads(threadNum) private(i, j)
{
    #pragma omp for schedule(dynamic,1) nowait
    for(i=0;i<numY; i++) {
        for(j=0; j<numX; j++) {

            // DO MANDELBROT SET CALCULATION
            #pragma omp critical
                {
                    // Draw Points using X Window
                }

        }

    }
}
```

我的 OpenMP_Dynamic 的架構大致上跟 Static 的版本一樣，差別主要是在於 for directive 的 schedule clause 我使用的是 dynamic 的參數而不是 static 的參數。我使用 chunkSize 為 1，雖然效率不一定是最好的但是這樣子能夠在實驗比較容易看出跟 static allocation 之間的差異。

另一點比較值得注意的就是雖然同樣是 dynamic allocation，但是 OpenMP 的實作方式跟 Massage Passing Model 不一樣的是我們不需要再去創建一個 master 來分配工作，因此所有的 thread 都可以參與到 Mandelbrot set 的計算本體的部分。

Hybrid_Static:

hybrid 的實作部分要混合 MPI 跟 OpenMP，同時使用 Massage Passing Model 以及 shared memory model 來達到最好的效果。實作關鍵的部分大致如下

```
    startX = rank;
    if(rank<numX%processNum){
        widthCount = (numX/processNum)+1;
    }
    else{
        widthCount = (numX/processNum);
    }


    store = (int*) malloc(sizeof(int)*numY*widthCount);

    #pragma omp parallel num_threads(threadNum) private(i, j)
    {
        #pragma omp for schedule(static) nowait
        for(i=0;i<numY; i++) {
            for(j=startX; j<numX; j+=processNum) {
                // Mandelbrot set calculation and use "store" array to store the result
            }
        }

    }

    if(rank==0){
        pixel = (int*)malloc(sizeof(int)*numX*numY);
    }
        int* displace = (int*) malloc(sizeof(int)*processNum);
        int* recvCount = (int*) malloc(sizeof(int)*processNum);
        displace[0] = 0;

    for(i=0;i<processNum;i++){
        if(i<numX%processNum){
            recvCount[i] = ((numX/processNum)+1) * numY;
        }
        else{
            recvCount[i] = (numX/processNum) * numY;
        }
        if(i!=processNum-1){
            displace[i+1] = displace[i] + recvCount[i];
        }
    }

MPI_Gatherv(store,widthCount*numY,MPI_INT,pixel,recvCount,displace,MPI_INT,0,MPI_COMM_WORLD);
```

要同時使用兩種模型，勢必要能夠清楚地分配每個 Process 負責的資料，並且再讓每個 Process 可以各自地創建許多 thread 來完成分配給這些 Process 的工作。

也就是說，data partition 分為兩個層次，row 的分配跟 column 的分配。Row 的部分使用 OpenMP 的 static schedule 來為我分配，而 column 的部分，我會讓每個 process 有一個起始的 column(startX) = 自己的 rank。接著該 process 要計算的部分就是 startX、startX+#process、startX+2*#process。也就是說例如我有 4 個 process，則 rank0 負責第 0、4、8…的 column，而 rank1 負責第 1、5、9….的 column，以此類推。
(這部分用跟前面不一樣的原因分配法是考慮到 Hybrid 的 Performance 會列為評分項目，因此就算是 static，如果可以把每個 process 負責的工作平均分散開來，也可以稍微減少 mandelbrot set 每個位置的點工作量不平均的問題。)

每個 Process 用 shared memory(OpenMP)的方式把自己份內的工作完成之後就把結果通通用 MPI_Gatherv 的方式由 rank=0 的 process 收集起來，並且統一用 X Window 畫出來。

Hybrid_Dynamic:

Hybrid_dynamic 的部分基本上跟 MPI_Dynamic 很像，同樣是使用 Master Slave 的架構，只是 slave 在計算從 Master 收到的 tasks 時，會使用 OpenMP，用 dynamic scheduling 的方式來加速每個 Slave Process 份內的 task 的計算速度。

另外還有一個以上所有版本都共通的實作細節，就是平均的劃分座標軸內的點的方式。先計算出 x 軸跟 y 軸的 stride，也就是 xstride = (right-left)/(#points in x) 以及 ystride = (top-bottom)/(#points in y)。而在 x 軸上每向右移動一個點，就是把座標加上 xstride，而在 y 軸上美向上移動一個點，就是把座標加上 ystride。利用這種方式我們就可以達到平均的在指定的座標內均勻地把指定的點給畫出來。

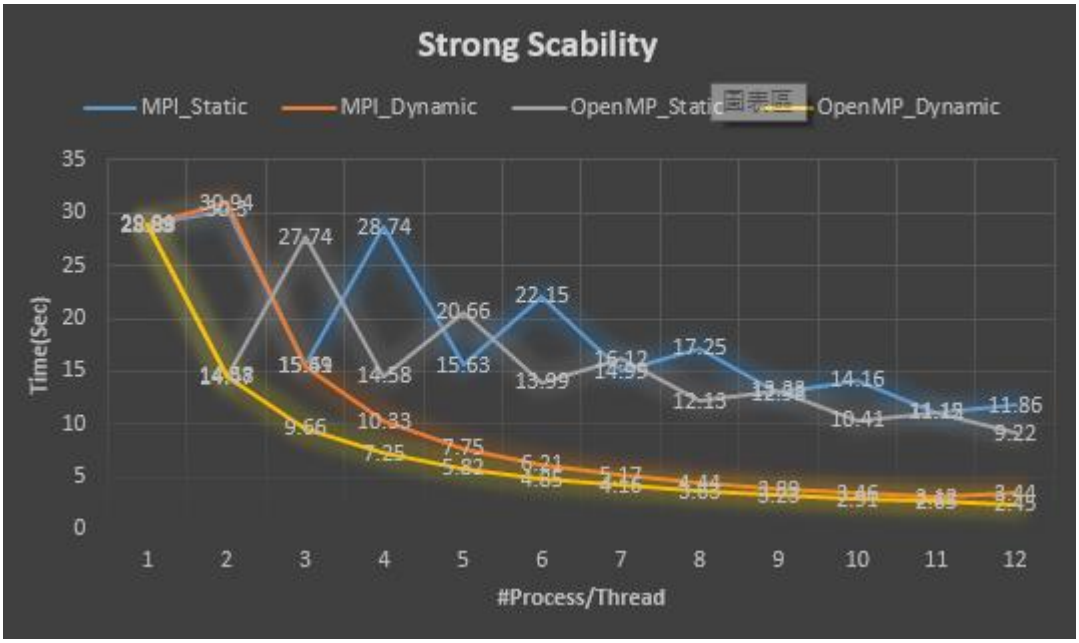
總結來說，不論是 MPI 還是 OpenMP 還是混合版本，我覺得在設計實作方式時最關鍵的還是如何把要計算的資料分給每個運算單元的方式。

---

2. Experiments & Analysis

以下所有實驗都是跑在課程提供的 Batch Server 上。

(a) Strong Scalability



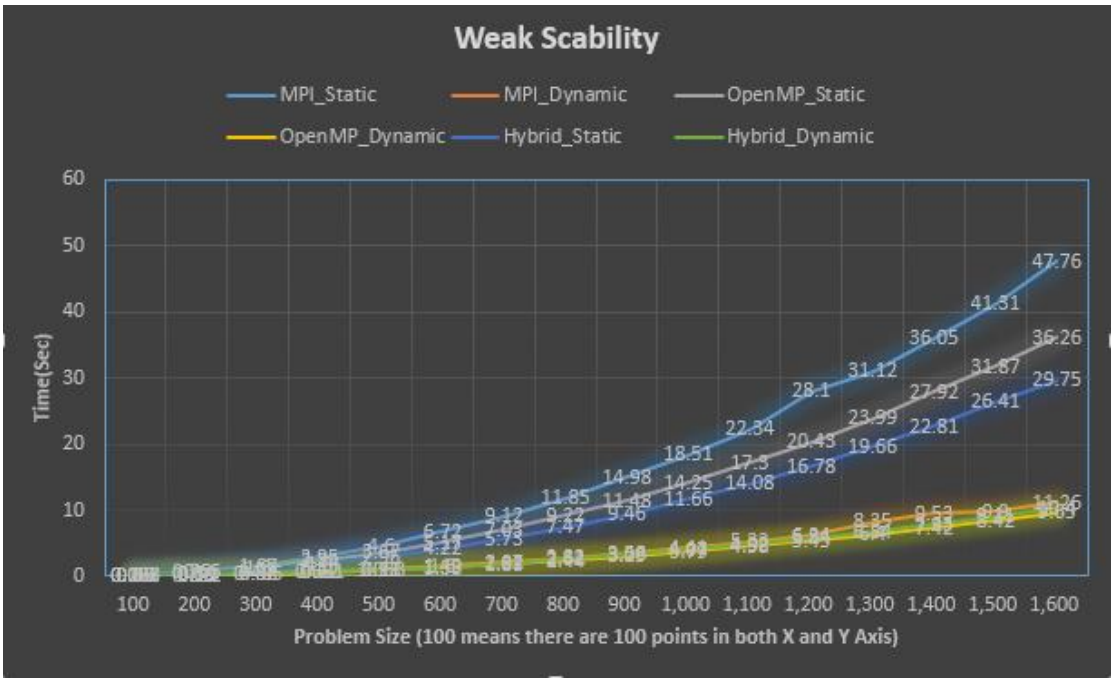
在 Strong Scalability 的部分，我們可以看到基本上 Static 的版本的效率整體而言是較差的，而且隨著 core 的增加時間是會起起伏伏的，反觀 Dynamic 版本的整體效率都是較佳的，而且隨著 core 的增加整體的時間是很穩定地減少的。

這部分的原因很簡單，因為 Mandelbrot set 是個隨著(x,y)位置的不同，運算量的差距非常的明顯。以實驗指定的坐標系(-2,-2)~(2,2)來說，幾乎最大的運算量需求都集中在靠近(0,0)，因此如果是 static 的版本，因為我 partition data 的方式是連續的，因此如果 slave process 或是 thread 的數量是奇數的時候，工作的分配會比較不均。(想像三個 core 的狀況，第一個 core 計算 y 軸-2~-0.7 的部分，第二個計算 y 軸-0.7~0.7 的部分，第三個計算 y 軸 0.7~2.0 的部分，很明顯工作量都會落在第二個 core 身上。(MPI 的部分因為第 0 個 process 是 master，所以實際上在運算 mandelbrot set 的部分只有 n-1 個 core，所以在上圖是 x 軸是偶數的值比較大)

至於 MPI 跟 OpenMP 之間的效率差距並沒有很明顯，這點我想是因為 Mandelbrot set 是 Embarrassing Parallel 的關係，不論用哪種 model 都可以很好的平行處理。

從 Strong scalability 的實驗可以很明顯的看出在 Mandelbrot set 這種運算量不平均的問題之中，動態地分配工作是非常的重要的。

(b) Weak Scalability



在 weak scability 的部分，我們固定使用的 core 數。我統一讓六個版本都是使用 12 個 core。

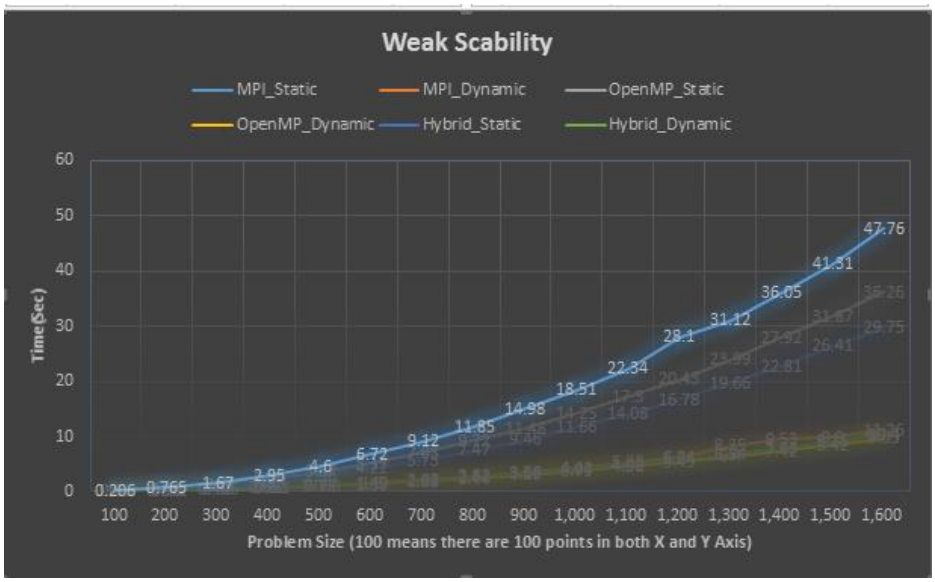
MPI: process = 12, 1 thread per process.

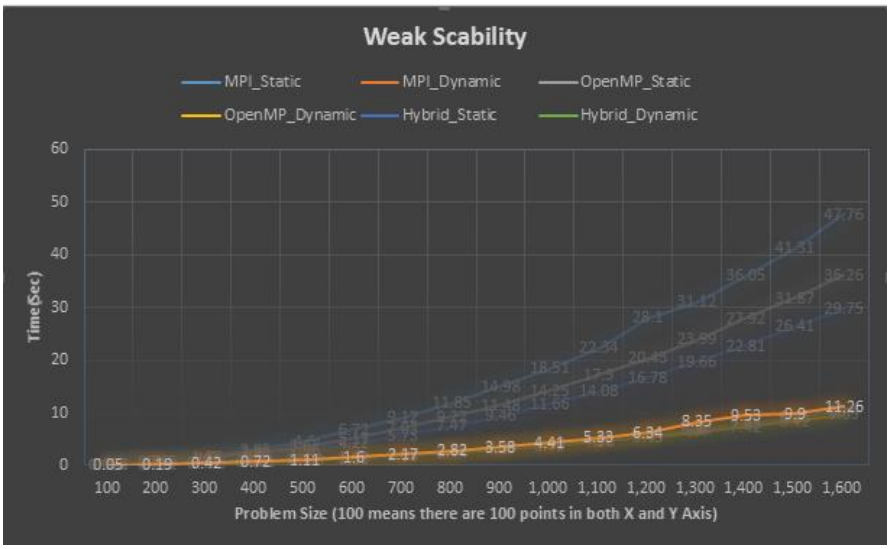
OpenMP: process = 1, 12 threads per process.

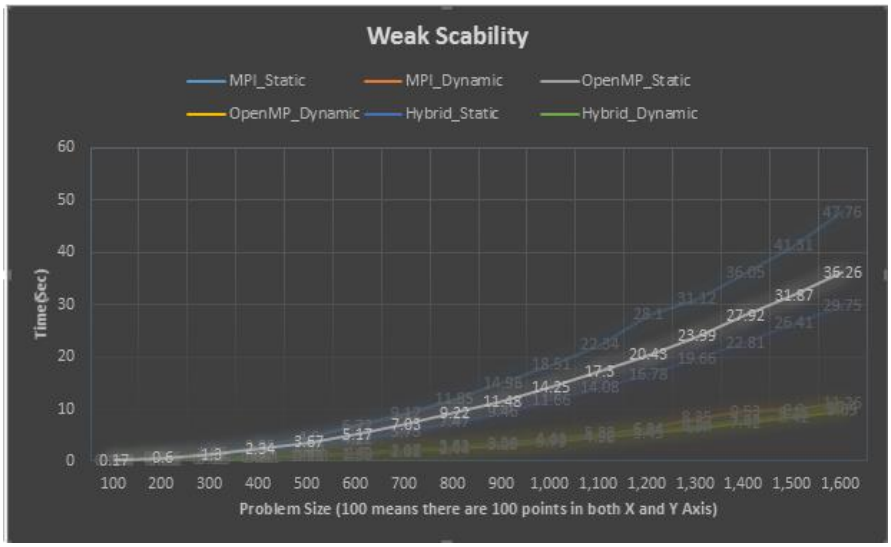
Hybrid: node=1, process = 4, 3 threads per process

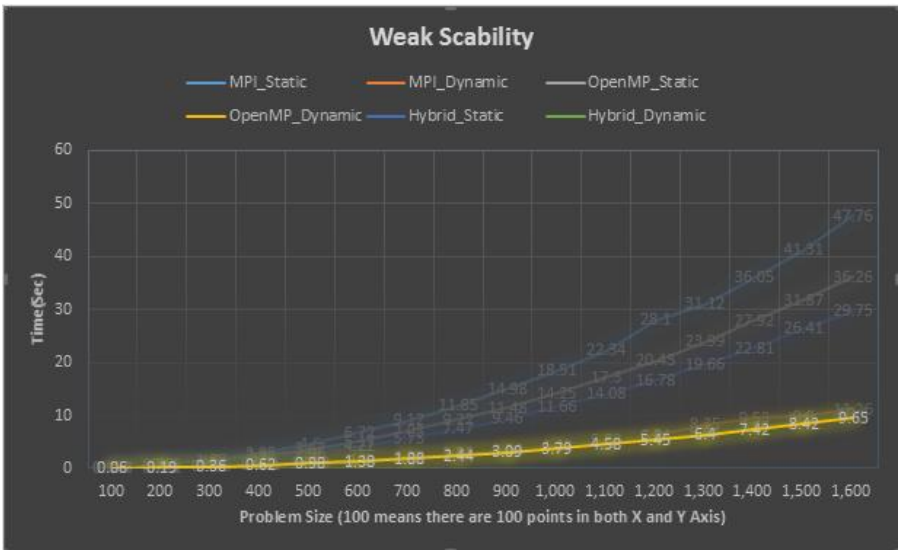
至於增加 Problem size 的方式，上圖的 x 軸，100 代表是 100*100 個點，200 代表 200*200 個點…以此類推。

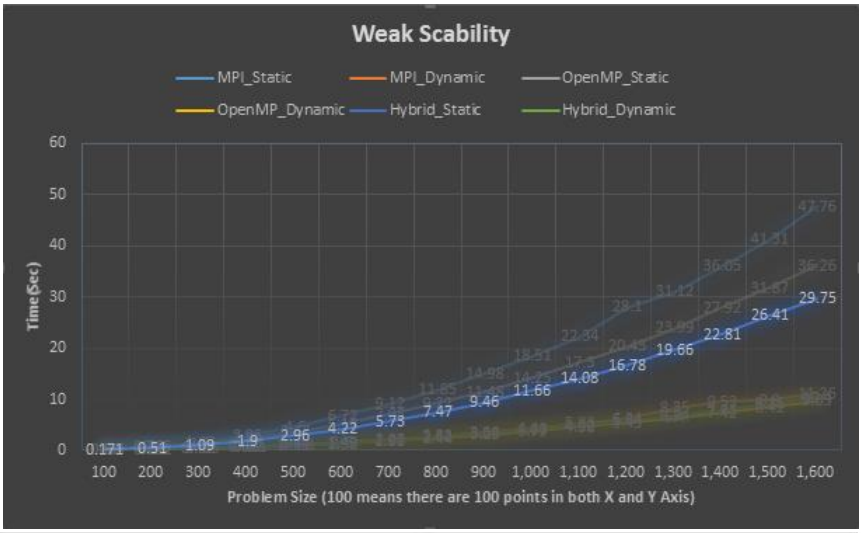
上圖的曲線有點混在一起，因此我又用了以下這些圖：

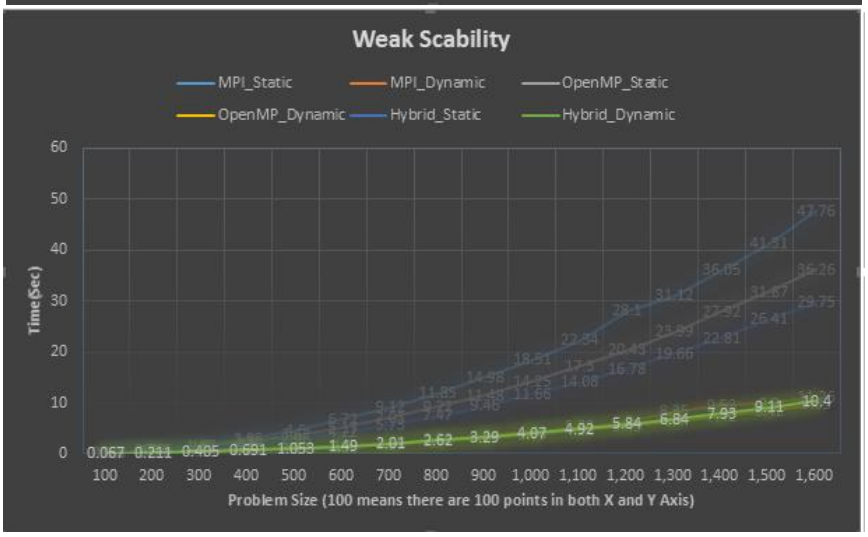



Weak Scability
MPI_Static
MPI_Dynamic
OpenMP_Static
OpenMP_Dynamic
Hybrid_Static
Hybrid_Dynamic
Time(Sec)
Problem Size (100 means there are 100 points in both X and Y Axis)


Weak Scability
MPI_Static
MPI_Dynamic
OpenMP_Static
OpenMP_Dynamic
Hybrid_Static
Hybrid_Dynamic
Time(Sec)
Problem Size (100 means there are 100 points in both X and Y Axis)


Weak Scability
MPI_Static
MPI_Dynamic
OpenMP_Static
OpenMP_Dynamic
Hybrid_Static
Hybrid_Dynamic
Time(Sec)
Problem Size (100 means there are 100 points in both X and Y Axis)





整體結果跟預期的大致一樣，不論哪種實作方式的曲線都是凹向上的，差別在於幅度的不同。

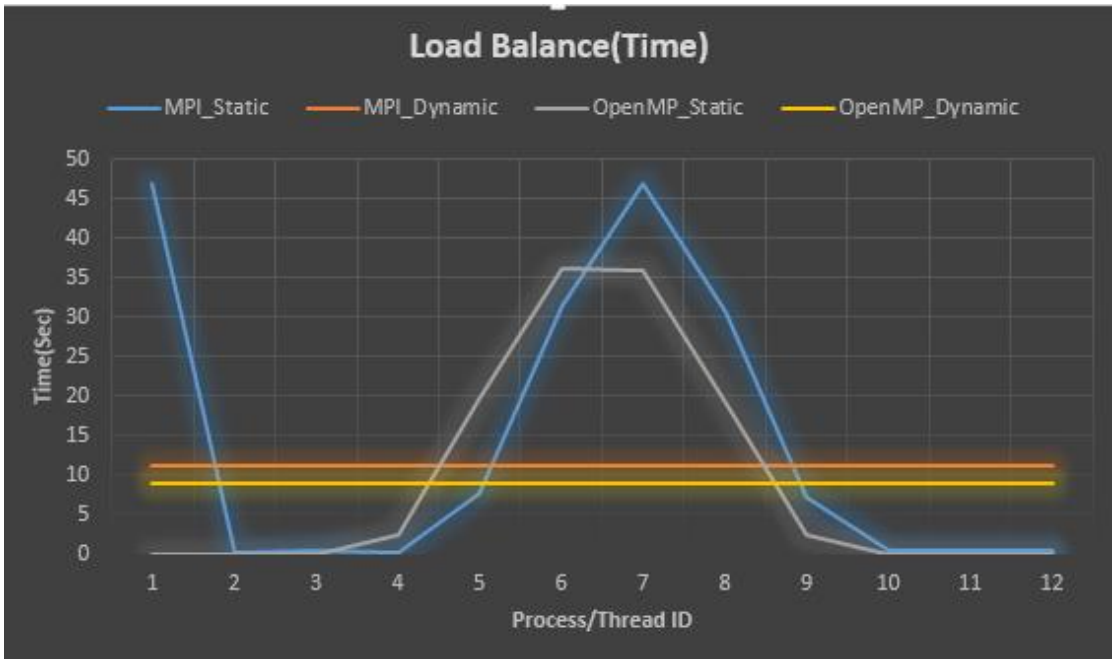
整體而言，因為 Mandelbrot set 是 embarrassing parallel 的問題，因此隨著 Problem size 的增加，整體的時間增加是穩定跟 Problem Size 成正比增加的。因為圖片的 x 軸是只單一軸上的點數量，因此圖片會呈現二次曲線的樣子。不過也許是因為 Core 的數量不夠多(12 而已)，我們還沒辦法太明顯的在 weak scalability 方面三種 programming model 的效率差別。

另外一個比較意外的地方是，同樣是 static，OpenMP 的時間增加幅度比 mpi 還要來的慢，而 Hybrid 又比 OpenMP 的增加幅度還要慢。(dynamic 則沒有這種現象)。這部分可能跟 API 的實作細節有些關聯。

雖然圖片看起來好像是 static 的 weak scability 比 dynamic 的要差，但是實際上是因為在 problem size 同樣時 dynamic 本來就比 static 要來的快，所以當兩個軸上的點數目變成兩倍(計算量變成四倍)時，其實不論是 static 還是 dynamic 增加的時間都是 4 倍左右的。

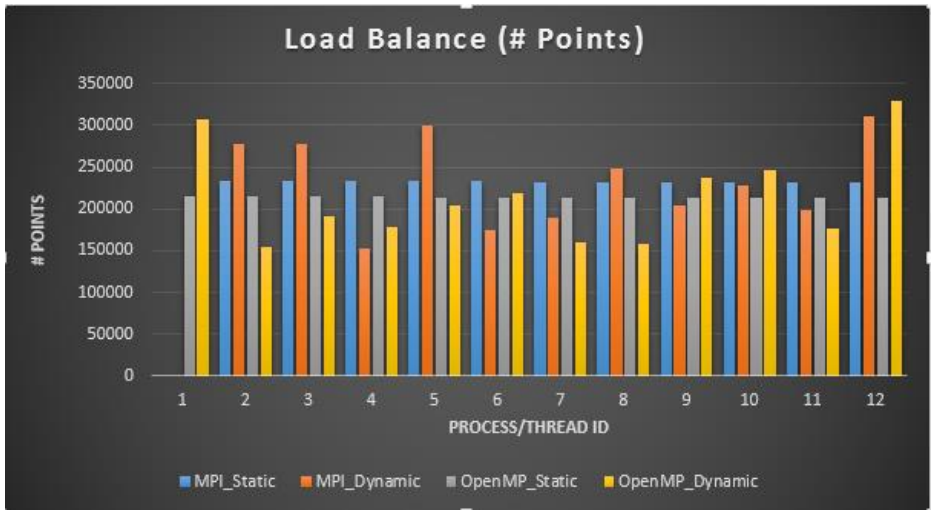
(c) Load Balance

在 MPI 跟 OpenMP 的部分，我都把 core 設定成 12(也就是跟上面的 weak scability 一樣)。



首先我們先看 MPI 跟 OpenMP 在時間上的 Load Balance。我們可以很明顯地看到 dynamic 的版本，每個 process/thread 的執行時間幾乎都是一樣的；而在 static 版本中，ID 介於中間的 process/thread 的執行時間遠遠大於其他的。這部分的原因前面也有提過，主要是因為切資料的方式是連續的，而 Mandelbrot set 的運算量幾乎集中於靠近(0,0)的中心部分，因此 ID 介於中間的 Process/Thread 的運算量就會非常非常的大。

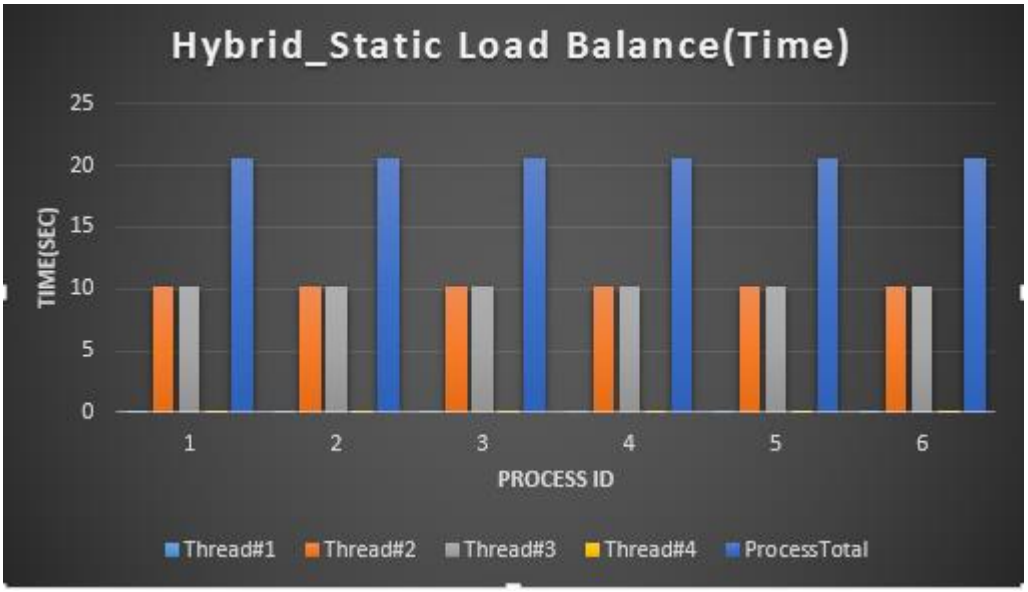
至於 MPI_static 的 process1 是因為他是 Master，執行時間一定比所有的 slave process 來的長。



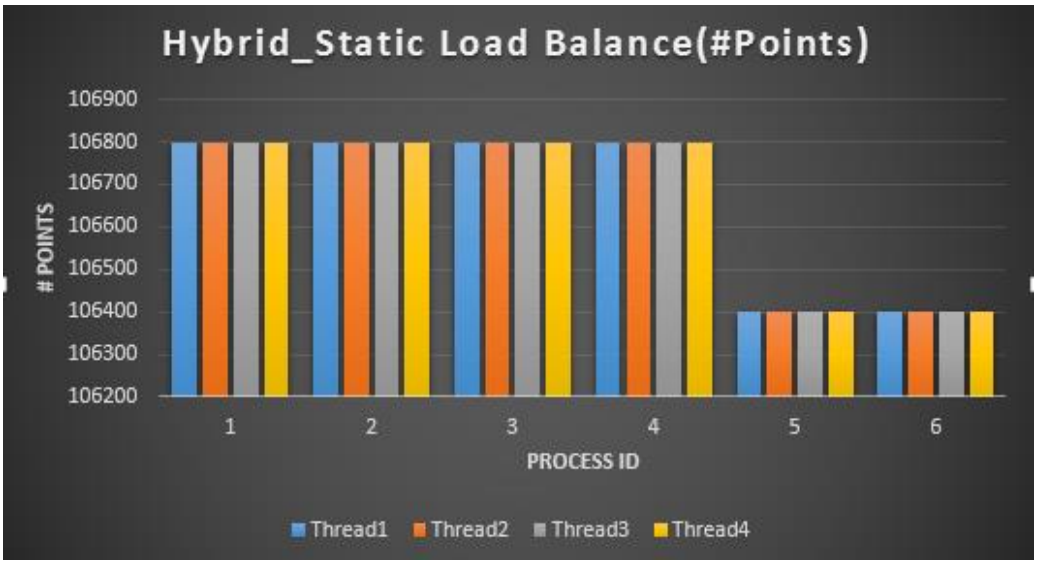
接著看到每個 Process/Thread 分配到的點的數量。可以看到跟上面的時間相反的是，在 static 的版本中每個 Process/Thread 拿到的點的數量是幾

乎一樣的，而在 Dynamic 的版本中每個 Process/Thread 拿到的點的數量是不固定的，也很難找出規則，因為整個計算過程是完全動態分配的關係。至於 Process1 的部分 MPI_Static 跟 MPI_Dynamic 會是 0 的原因是因為他們是 Master。

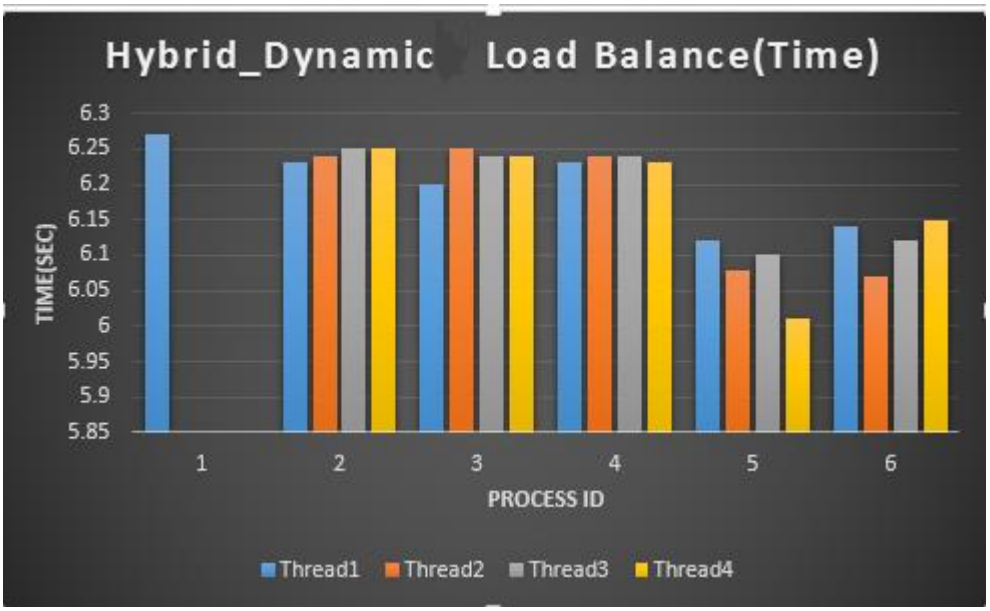
接著是 Hybrid Version 的 Load Balance。我使用的規格是 node =2，ppn=12, each node has 3 process & each process has 4 threads.



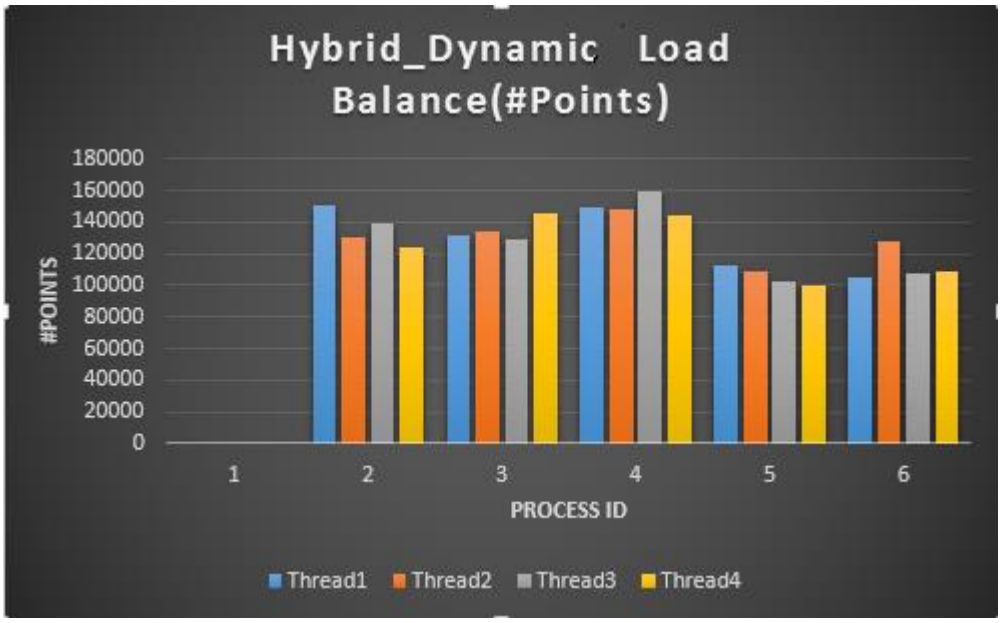
從上圖我們可以看出在 Hybrid_Static 的版本中，Process 間的工作是非常平均分配的，而每個 Process 內的 thread 之間的工作則如同其他的 static 版本一樣不平均。這部分是因為我的實作方式的關係。一開始分配工作給每個 Process 時，我不是像另外兩個版本一樣使用連續分配的方式，而是使用分散的分配方式，如同我在前一個 section 提到那樣。至於 thread 的部份的時間分配不均主要是因為我的 schedule 使用了 static 參數的關係。



在 Hybrid_static 的點分配中我們看到幾乎所有的 process 其中的 thread 分配到的點都是一樣的。畢竟是靜態的工作分配，最保險的做法就是給每個 core 都一樣的點的數目。(注意 y 軸的值，雖然 process 5,6 在 chart 上看起來比較小，但是其實分到的點的數量幾乎是沒差的)。



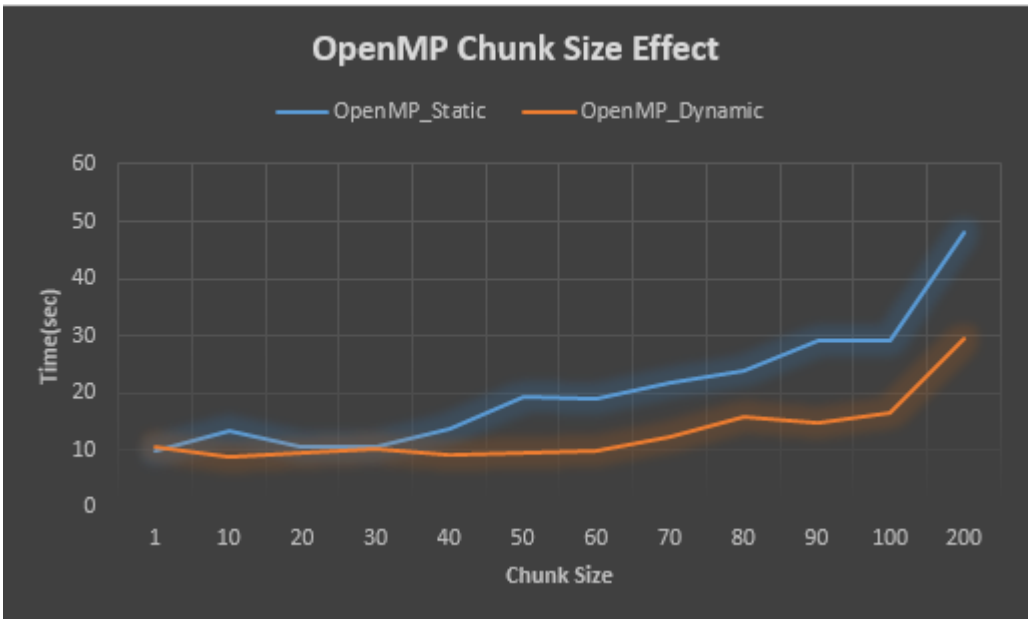
這邊則是 Hybrid_Dynamic 的時間分配。可以看到 Process1 因為是 Master 的關係，只有一個 thread 存在，而且執行時間也是所有 Process 裡面最長的。而其他 Process 以及生出來的 thread 的執行時間都是差不多的，這點因為是 dynamic scheduling 的關係所以不難猜到。



Hybrid_Dynamic 的點的分配則是可以看到 Process 1 因為是 Master 的關係完全沒有分配到點。Process 5 跟 6 拿到的 point 數則稍微少於 process2、3、4，不過因為是 dynamic scheduling 的關係，我覺得這部分應該只是湊巧是這樣的結果，而不是有什麼特別系統性的原因。

(d) Extra : Chunk Size Effect

OpenMP 的 for directive 在指定 static 或 dynamic 時，還可以另外傳入一個參數，也就是 chunk size。這個參數對於 static 的版本尤其重要。在上面的實驗中我為了能夠明顯的看出 static 跟 dynamic 的差異，因此不指定 chunk size 而讓 OpenMP 自動幫我連續分配 task，造成了很嚴重的 Load Balance 不均的狀況。在這個實驗中我會用 OpenMP 測量不同的 chunk size 對於整體效率的影響。(以 1600*1600，12 個 core 的設定進行)



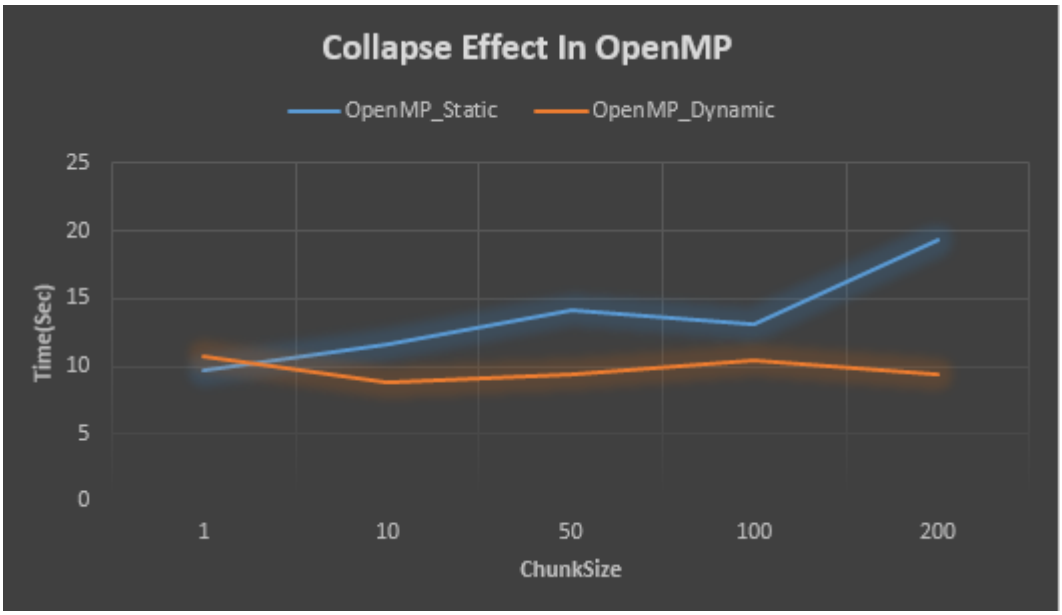
從上圖中我們可以看到，在 static 的版本中，隨著 chunk size 的增加，執行時間是會越來越長的。這部分主要是因為如果 chunk size = 1 則每個 thread 計算的 row index 會比較平均的分配，而如果 chunk size 增大，導致計算密集區域的 row index 都坐落在同一個 chunk 內，就會造成 load balancing 不佳的狀況發生，從而拖累整體執行效率。

在 dynamic 的版本中這個現象比較沒那麼明顯，但是我們還是可以看到一但 chunk size 大過某個特定的門檻後，儘管工作是動態分配的，卻因為 chunk size 的限制受限了整體動態分配的可能性。(雖然這樣說不定有時候可以減少動態分配工作的 overhead?)

(e) Extra: Collapse Effect

在 OpenMP 的 for directive 中，有一個稱為 collapse 的 clause 可以指名。若是有指名 collapse，則工作的分配就會不僅止於最外層的迴圈，而會包含到內層的迴圈，也就是工作的分配不在是以 row 為單位，而是以 point 為單位。在這個實驗中我會用 OpenMP_static 和 dynamic 測量不同的 collapse 版本，並以 chunk size 為變量測量整體效率的影響。(以 1600*1600，12 個 core

的設定進行)



首先看到 static 的曲線。跟(d)的曲線比較，我們可以發現在同樣的 chunk size 下，collapse 過後的 static version 的執行效率明顯的比沒有 collapse 過的要來的快。可以猜測 task 的分配從 row 為單位轉變成以 point 為單位某種程度可以減輕 Load balance 不均的狀況。

而 dynamic 的部分我們可以發現，跟(d)的曲線比較，儘管 chunk size 增加到 100 或 200，整體的時間卻沒有明顯的增加，這部分我想是因為工作的分配從 row 變成以 point 為單位，增加了 dynamic allocation 的機動性。

3. Experience

這次的作業讓我們接觸到了有別於 Massage Passing Model 的 shared programming model，以及混合兩種 model 的方式。

不同於 HW1，Scalability 主要是受限於問題可以同時平行的程度，這次的 Mandelbrot set 因為是 Embarrassing Parallel 的關係，運算的效率主要是受限於每個 core 的 load balancing。

因此，我覺得這次作業最主要學到的東西還是 Load balancing 的部分。經由這次的作業我見識到了在像是 Mandelbrot set 這種運算量在不同位置極度不平均的問題中，如果我們沒辦法有效率的發揮所有運算單元的運算效能，很有可能讓很多運算資源就這樣被閒置下來。因此動態的決定每個運算單元的任務是非常非常重要的。

要說這次作業遇到困難，我想應該還是如何分配要運算的點的位置給每個 Process/Thread 了吧。要想要能夠有效的分配工作，我覺得還有一個很重要的點就是要理解我們要解決的問題的特性。如果可以掌握問題的特性，我們就有辦法利用這些特性來提升 Load balancing 的程度。(像是 Mandelbrot set 運算量幾乎都是密集集中於(0,0)的附近。)