Parallel Programming HW3 Report

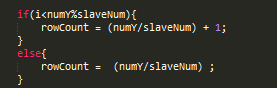
102062209 邱政凱

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。



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)…。

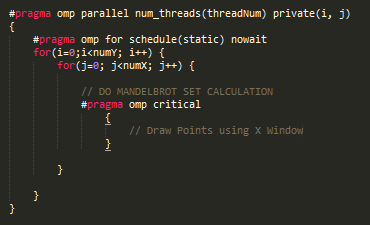
MPI\_Dynamic:

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。

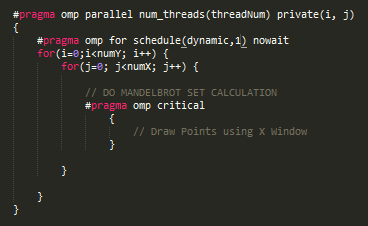


架構大致如上所示，用OpenMP的for directive來幫助平行的計算。可以看到由於我沒有指明collapse clause，因此工作的平行化只限於外層迴圈，也就是row的迴圈。另外就是我的schedule clause沒有指名chunkSize，因此每個thread要計算的row就是continuous地平均分配下去，也就是說大致上會跟我的MPI\_Static分配row的方式相似。(chunk size的影響我會用extra實驗來說明)

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:

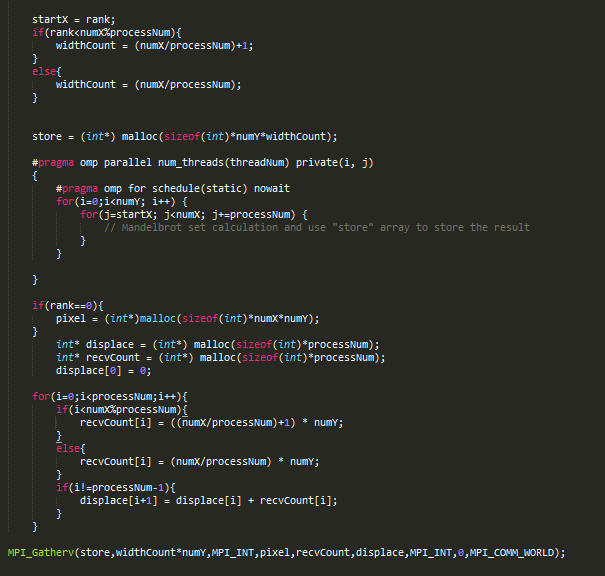


我的OpenMP\_Dynamic的架構大致上跟Static的版本一樣，差別主要是在於for directive的schedule clause我使用的是dynamic的參數而不是static的參數。我使用chunkSize為1，雖然效率不一定是最好的但是這樣子能夠在實驗比較容易看出跟static allocation之間的差異。

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



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。

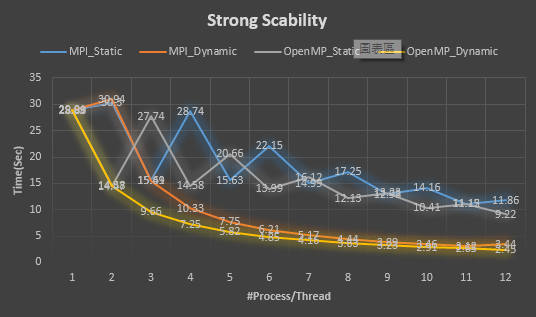
另外還有一個以上所有版本都共通的實作細節，就是平均的劃分座標軸內的點的方式。先計算出x軸跟y軸的stride，也就是xstride = (right-left)/(#points in x) 以及ystride = (top-bottom)/(#points in y)。而在x軸上每向右移動一個點，就是把座標加上xstride，而在y軸上美向上移動一個點，就是把座標加上ystride。利用這種方式我們就可以達到平均的在指定的座標內均勻地把指定的點給畫出來。

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。

1. Experiments & Analysis

以下所有實驗都是跑在課程提供的Batch Server上。

1. Strong Scalability



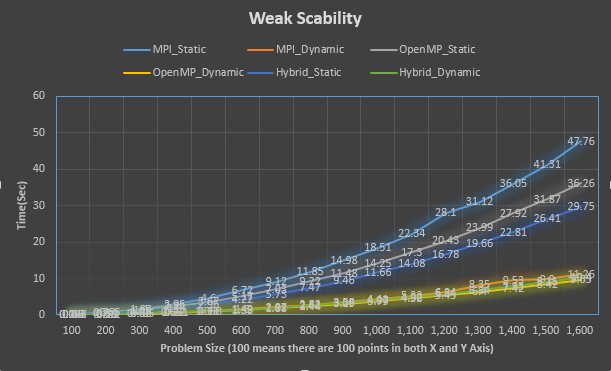
在Strong Scalability的部分，我們可以看到基本上Static的版本的效率整體而言是較差的，而且隨著core的增加時間是會起起伏伏的，反觀Dynamic版本的整體效率都是較佳的，而且隨著core的增加整體的時間是很穩定地減少的。

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。

1. 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。

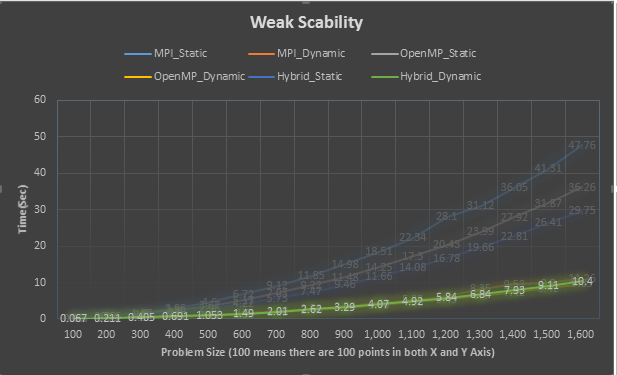
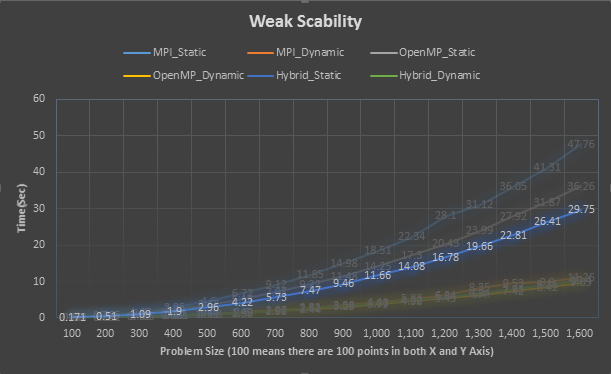
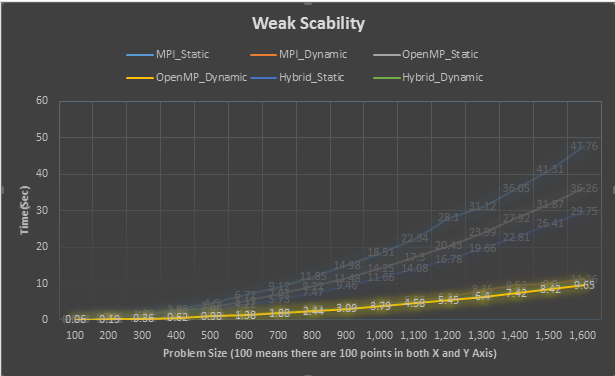
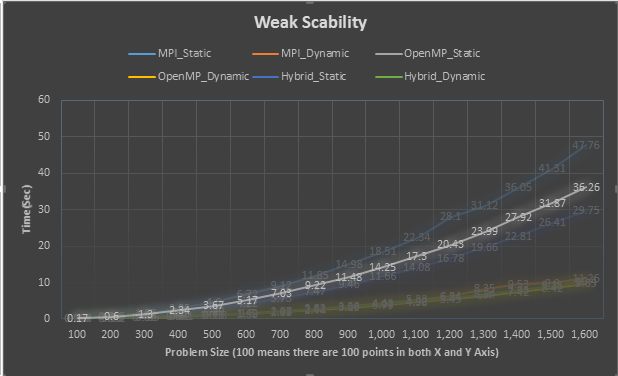
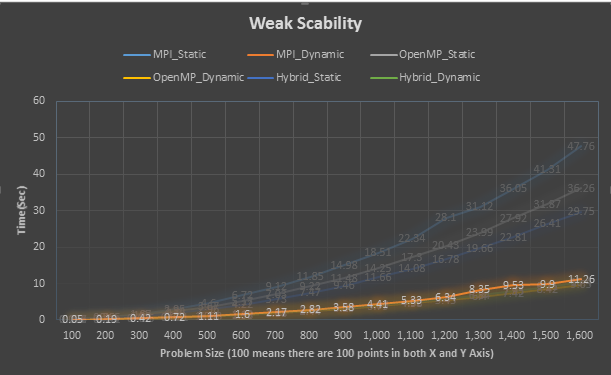
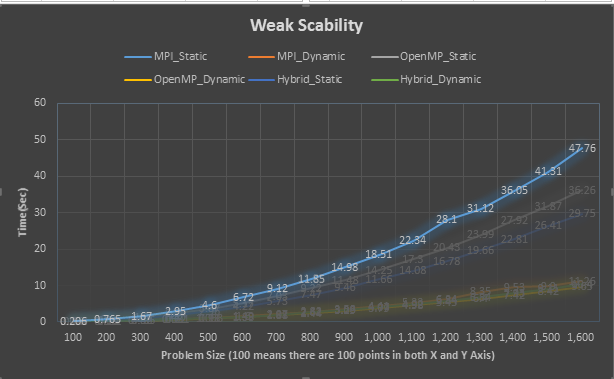
MPI: process = 12, 1 thread per process.

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：



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

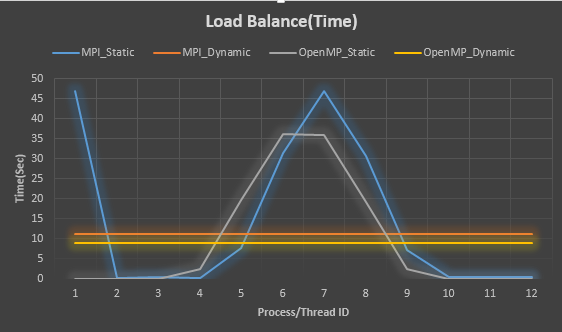
隨著Problem size的增加，整體的時間增加是穩定跟Problem Size成正比增加的。因為圖片的x軸是只單一軸上的點數量，因此圖片會呈現二次曲線的樣子。不過也許是因為Core的數量不夠多(12而已)，我們還沒辦法太明顯的在weak scalability方面三種programming model的效率差別。

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。

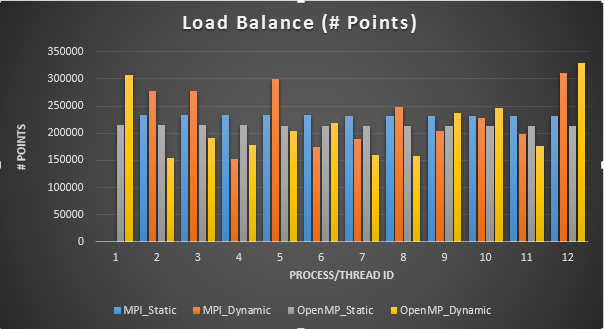
1. Load Balance

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)。



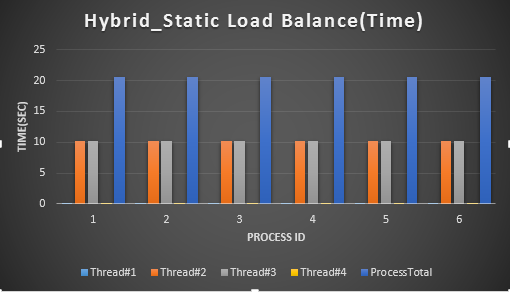
首先我們先看MPI跟OpenMP在時間上的Load Balance。我們可以很明顯地看到dynamic的版本，每個process/thread的執行時間幾乎都是一樣的；而在static版本中，ID介於中間的process/thread的執行時間遠遠大於其他的。這部分的原因前面也有提過，主要是因為切資料的方式是連續的，而Mandelbrot set的運算量幾乎集中於靠近(0,0)的中心部分，因此ID介於中間的Process/Thread的運算量就會非常非常的大。

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。

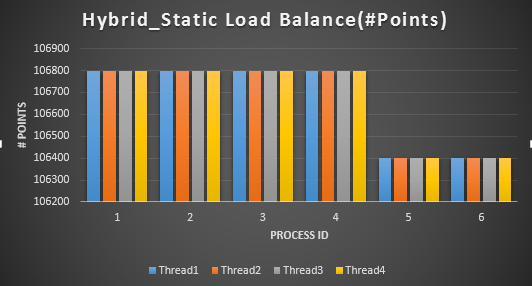


接著看到每個Process/Thread分配到的點的數量。可以看到跟上面的時間相反的是，在static的版本中每個Process/Thread拿到的點的數量是幾乎一樣的，而在Dynamic的版本中每個Process/Thread拿到的點的數量是不固定的，也很難找出規則，因為整個計算過程是完全動態分配的關係。至於Process1的部分MPI\_Static跟MPI\_Dynamic會是0的原因是因為他們是Master。

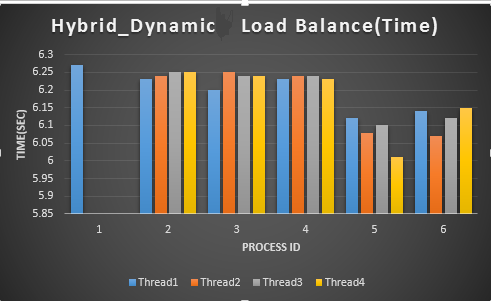
接著是Hybrid Version的Load Balance。我使用的規格是node =2，ppn=12, each node has 3 process & each process has 4 threads.



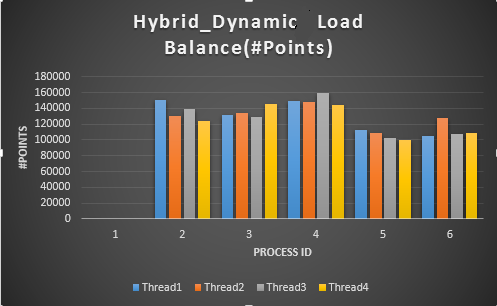
從上圖我們可以看出在Hybrid\_Static的版本中，Process間的工作是非常平均分配的，而每個Process內的thread之間的工作則如同其他的static版本一樣不平均。這部分是因為我的實作方式的關係。一開始分配工作給每個Process時，我不是像另外兩個版本一樣使用連續分配的方式，而是使用分散的分配方式，如同我在前一個section提到那樣。至於thread的部份的時間分配不均主要是因為我的schedule使用了static參數的關係。



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)。



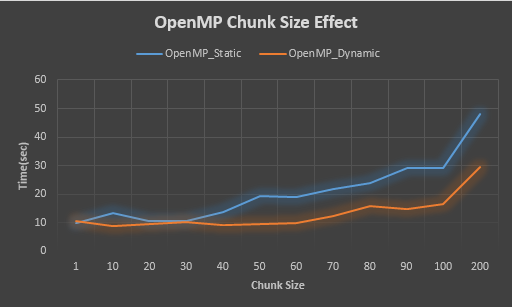
這邊則是Hybrid\_Dynamic的時間分配。可以看到Process1因為是Master的關係，只有一個thread存在，而且執行時間也是所有Process裡面最長的。而其他Process以及生出來的thread的執行時間都是差不多的，這點因為是dynamic scheduling的關係所以不難猜到。



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。

1. 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)

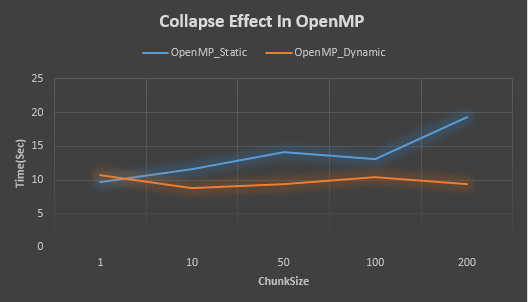


從上圖中我們可以看到，在static的版本中，隨著chunk size的增加，執行時間是會越來越長的。這部分主要是因為如果chunk size = 1則每個thread計算的row index會比較平均的分配，而如果chunk size增大，導致計算密集區域的row index都坐落在同一個chunk內，就會造成load balancing不佳的狀況發生，從而拖累整體執行效率。

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?)

1. 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。

1. 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。)