CS542200 Parallel Programming

Homework 2: Mandelbrot Set

107062103 王依婷

1. **Implementation**
2. Pthread

這個版本主要以Pthread實作thread層面的平行化，因為每個點所需要的計算量不一定相同，不能單純地將所有需要計算的點平均分配給所有thread，因為可能會造成load balancing問題。因此實作上需做dynamic scheduling，而我是使用了thread pool的概念去控制排程，只是所有要做的task是固定的量，並不是像它原先的使用情境。

thread pool本身實作上參考了一些現有的一些固定格式寫法，基本上也和上課講義的差不多。整個流程首先會先建立起要由所有thread共同使用的thread pool，用pthread\_create建立給定數量的thread。在pthread\_create中的function參數，放的是另一個用來讓thread拿取task的function，它會持續嘗試拿新的task來計算。

再一一將task，放入pool中的task queue中。這邊我設定一個task是計算一行（row）數字點的量，一行中總共有width個點，總共有height行/height個task。這邊的task結構包含Mandelbrot set計算及將值寫到image中的function，以及計算時會使用到的相關數字，包成一個陣列方便之後當作參數傳遞。

最後再將thread pool中所有thread用pthread\_join集合在一起，結束他們，並將thread pool destroy掉。

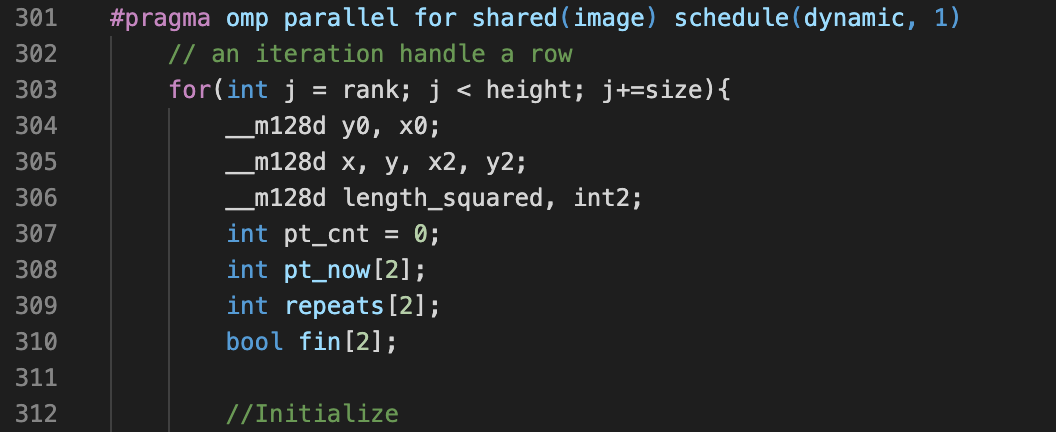
其中，因為thread pool是由所有thread共用，和它相關的操作都必須注意同步化的問題，使用了mutex lock搭配condition variable去控制。在確定整個流程正確之後，我將計算改為vectorization版本來加速，在下面的部分會進行說明。

1. Hybrid (MPI + OpenMP)

第二個版本結合了MPI和OpenMP，MPI用來把task分給不同的process，OpenMP再進一步分給不同的thread來做運算。

在process層級，因為覺得process之間的溝通會花掉不少時間，所以平均將所有的數字點分給所有的process，而沒有實作一個scheduler來控制各個process去取data。取的時候同樣以行（row）為單位來取，因此每個process平均處理height/size行。每個process中都會declare一個大小相同的image陣列，所有位置初始為0，用來存放計算完所得到的repeat值，最後再使用MPI\_Reduce將所有image陣列reduce到rank = 0的node上，並寫成png檔。

而在thread層級，將計算部分放在OpenMp的parallel region中，用for directive來跑過目前處理的那一行中所有的點，並且使用dynamic scheduling，chunk設為1，讓每個thread在一算完手上的點時，就會被給予一個新的點，繼續計算。image陣列在所有的thread之間是共用的，但因為寫入的位置都不相同，是根據當下計算的點決定，所以不同的thread並不會彼此影響到。

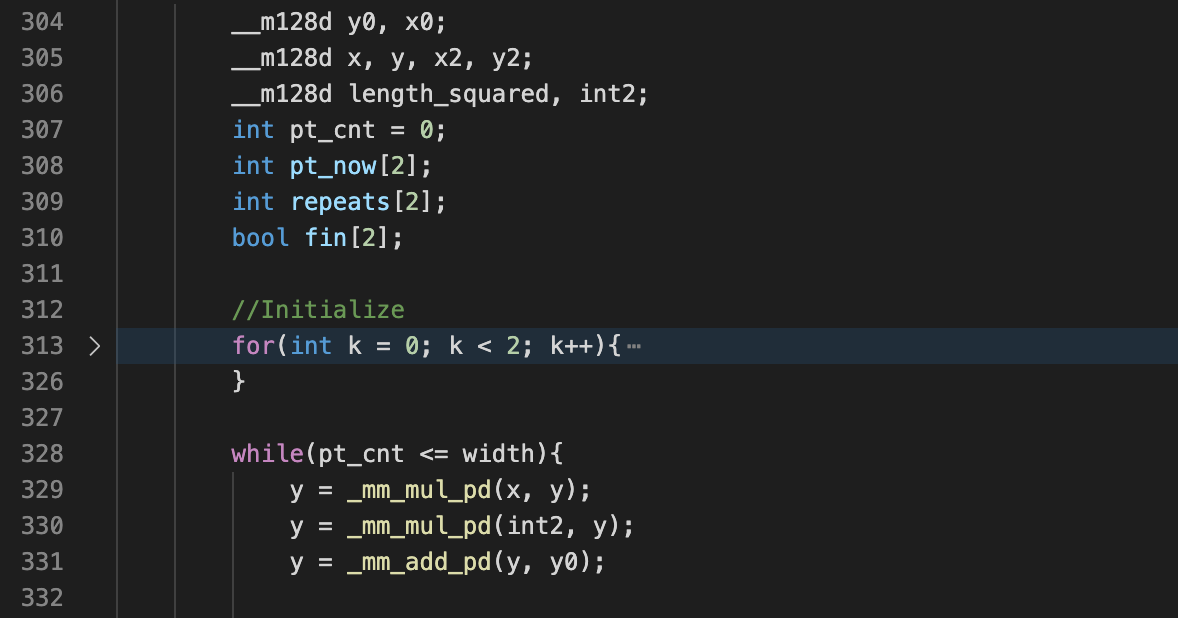
一開始我先確定計算部分使用原本sequential code中的會正確無誤，之後也把計算改為vectorization版本來加速。同樣地，在實作vectorization時一個array能夠存放兩個double數字，因此速度最快能提升到兩倍。實作的細節和2a是相同的，因此也一樣放到下面說明。

每個process都會跑同一份code（MPI），

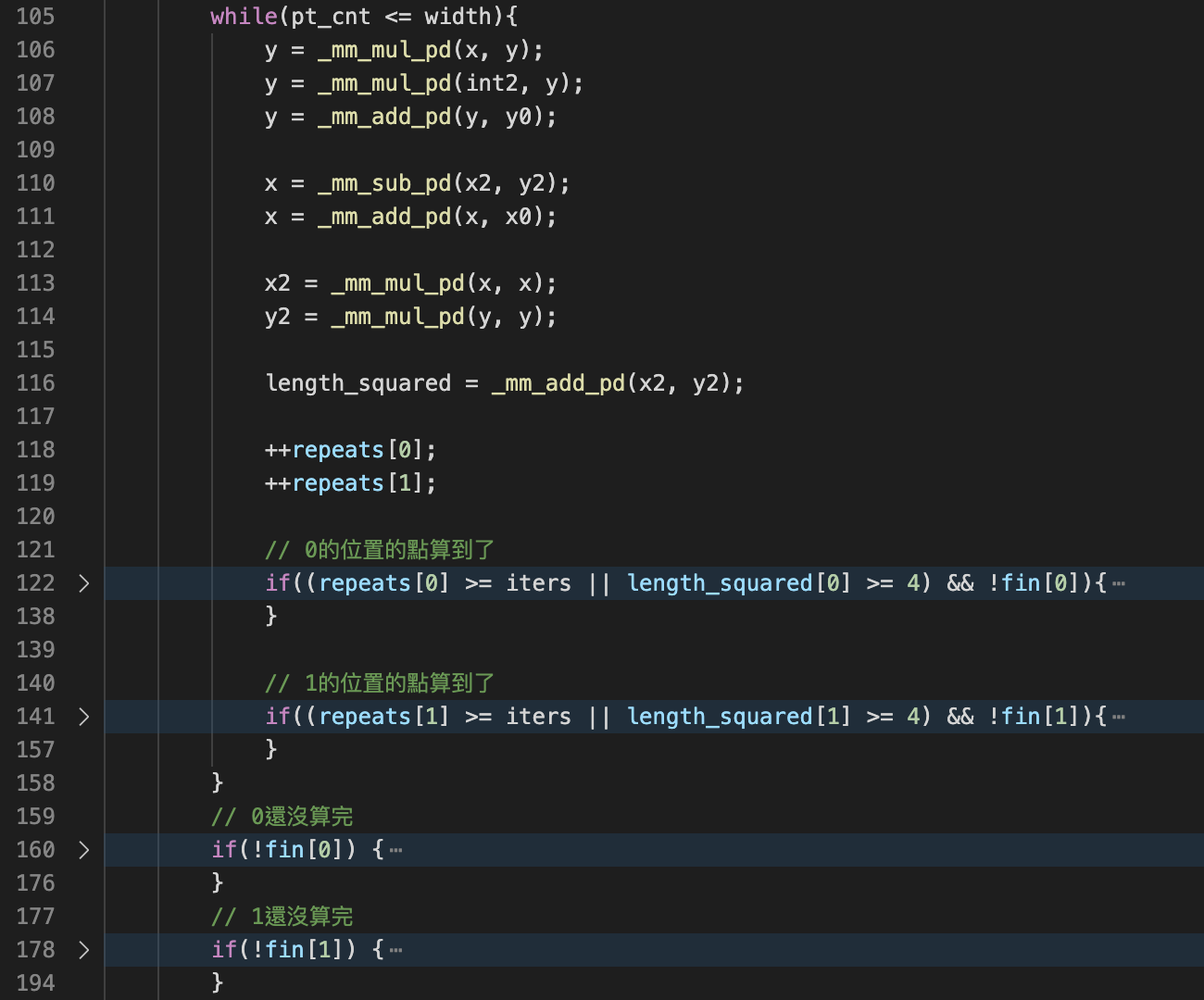
都有這個parallel region，讓thread能平行計算（OpenMP）

+ Vectorization

在thread做計算的部分使用了manual vectorization的方式來加速，可以同時做兩個double的計算，最快可以達到兩倍快的速度；不過因為在Mandelbrot set的計算中，每個點的計算量不同，因此並無法剛好以2的倍數進行計算的加速，所以實際上難以達到兩倍。

首先將計算會用到的數值以\_\_m128d宣告好，並將其中的兩項的數值都初始化成0或是需要的值。其中，定義了幾個新的變數，分別是pt\_cnt、pt\_now[2]和fin[2]，pt\_cnt代表目前在所有點（一個thread一次計算一行，共有width個點）中算完了幾個點，pt\_cnt是個陣列，代表著做vectorization後同時做的兩個計算目前在算哪個點，等結束後能算出在image中的位置，把值放入，fin則代表兩個計算是否已抵達最後的點。

（為了後面敘述方便，將做了vectorization後同時進行的兩個計算稱為a和b）計算的過程也有點像是dynamic的狀態，當a結束一個點時，他會再去取下一個新的點，reset陣列裡的值來繼續計算，並且更新pt\_cnt的值，反之b亦然。於是這邊相對於sequential的規律for迴圈，改為while迴圈來執行，判斷當所有點還沒被計算過時，a和b都會一直做運算、拿新的點、再運算。

到了最後，a和b也不一定會一起結束。假設a拿了最後一個點，會將pt\_cnt加一，變成width，接著不管是a還是b先算完，都會發現目前的pt\_cnt=width，代表所有點都已經被取了，因此會將它的fin設成true，之後它的計算都不會被用到。但因為整體機制的關係，他還是會把pt\_cnt加一，while迴圈也因此被結束，可能還有另一個計算還沒做完。這時可以用fin來判斷是a還是b尚未結束運算，再接著算完即可。

1. **Experiment & Analysis**
   1. Methodology
      1. System spec: 使用課程所提供的，並沒有使用其他額外系統。
      2. Performance metric: （for computing time）
         1. Pthread

試了幾個計算時間的方法，最後用chrono函式庫裡的steady\_clock::now()在每個thread計算開始前和後取得當下時間，相減得到duration後再加起來除以thread數量。

* + - 1. Hybrid

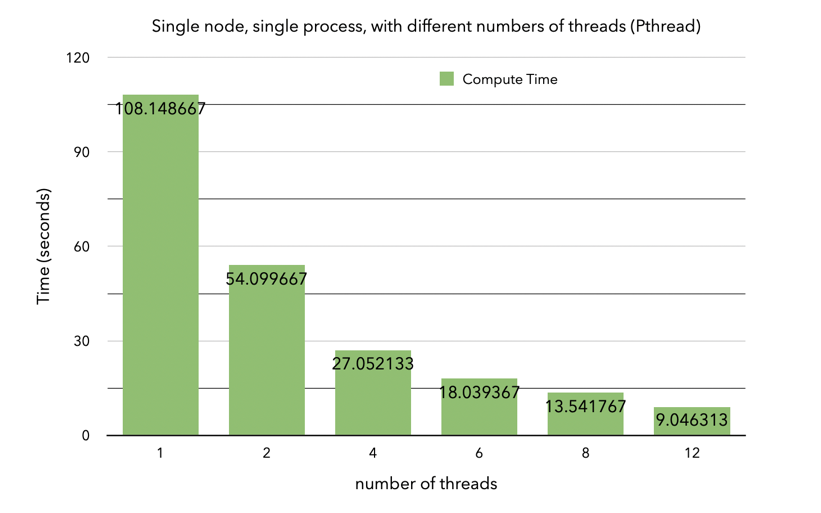
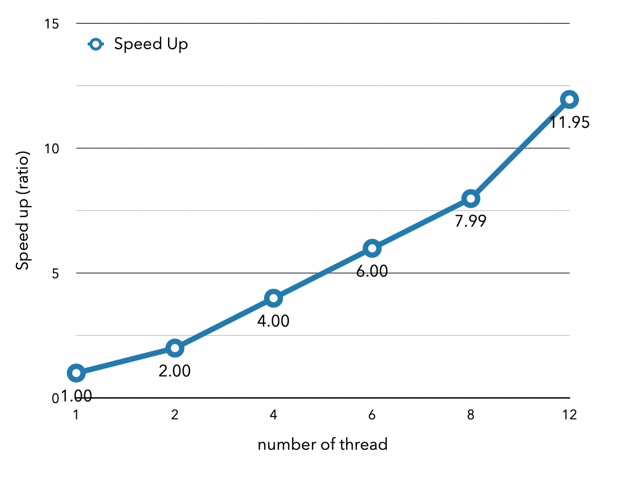
計算的部分都是放在每個process的parallel region中，我使用了OpenMP的omp\_get\_wtime()來取得進入region前和出了region時的時間，相減後用MPI\_Reduce 在rank=0的process上相加，最後除以process的數量。

每個實驗都跑三次取得平均，繪製成數據圖。Speed up的部分是將實驗中最基本設定的花費時間除以其他設定花費時間，所計算出的比值。

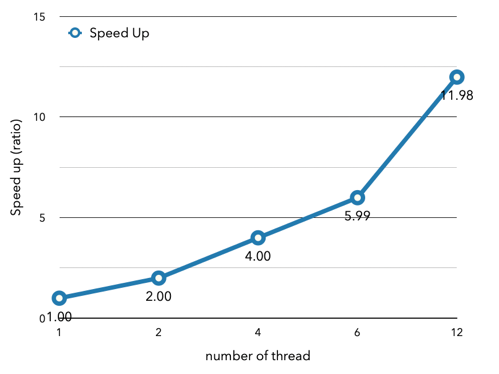
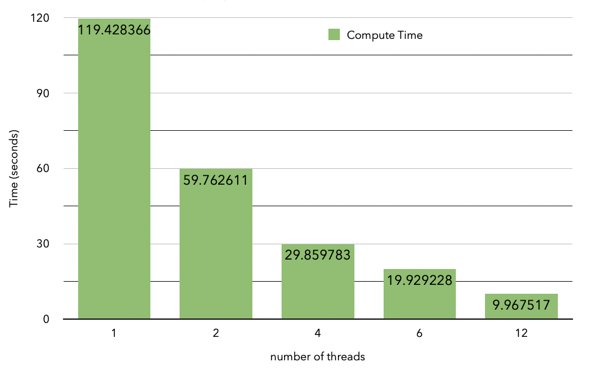
* 1. Plots
     1. scalability experiments

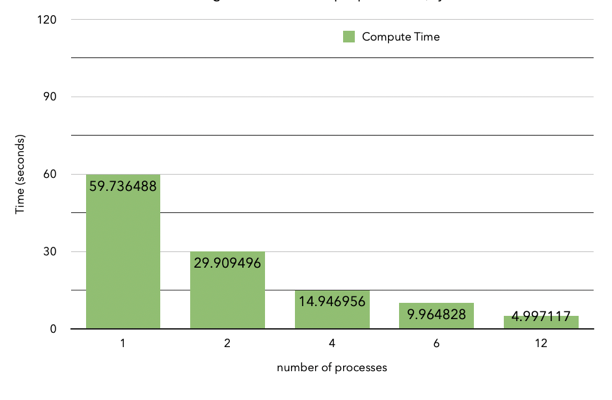
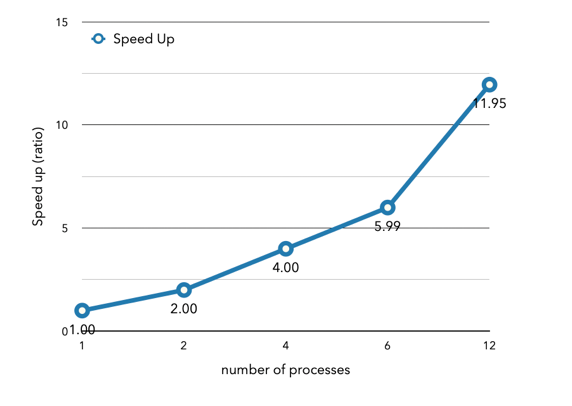
取strict29作為實驗的testcase。以下分為Pthread和Hybrid分別展示和探討：

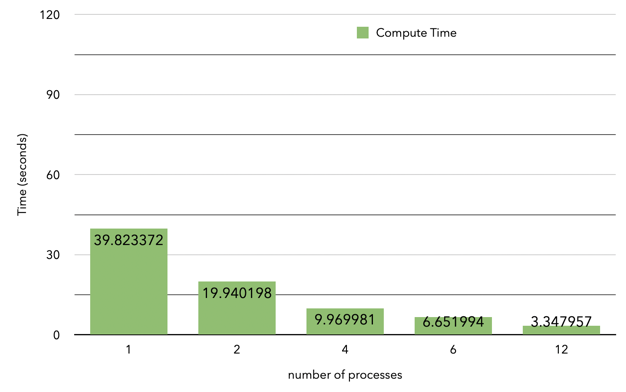
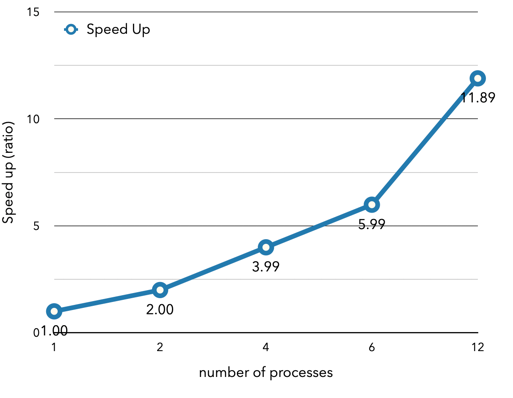
* + - 1. Pthread

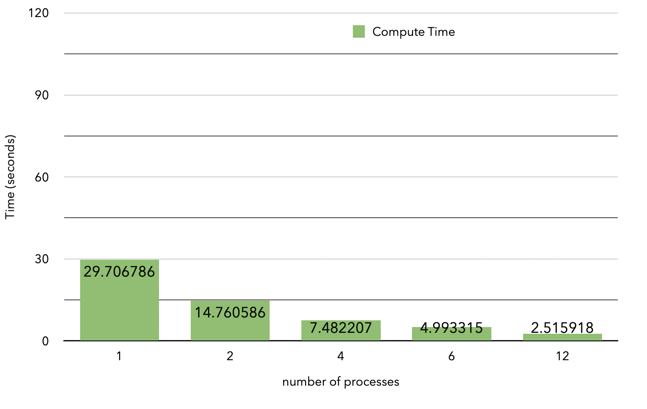
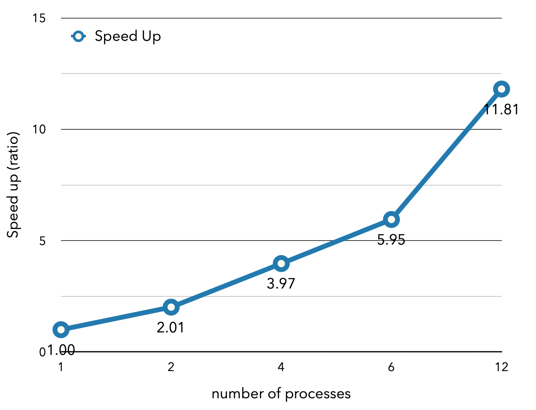
由於限定Pthread只有一個process，因此實驗只有改變thread數量的部分。

* + - 1. Hybrid
* 固定node中的process數量為1，改變thread數量。從single node到4 node都分別做實驗。

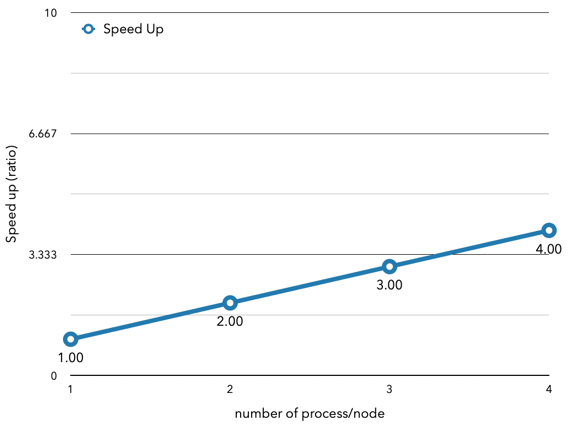
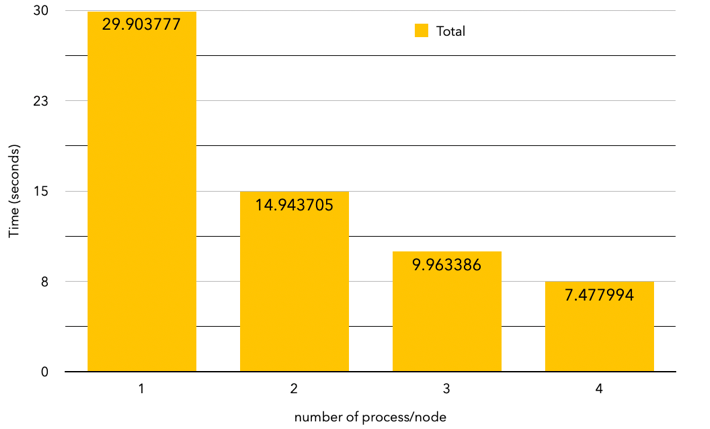


▲ single node, single process/node

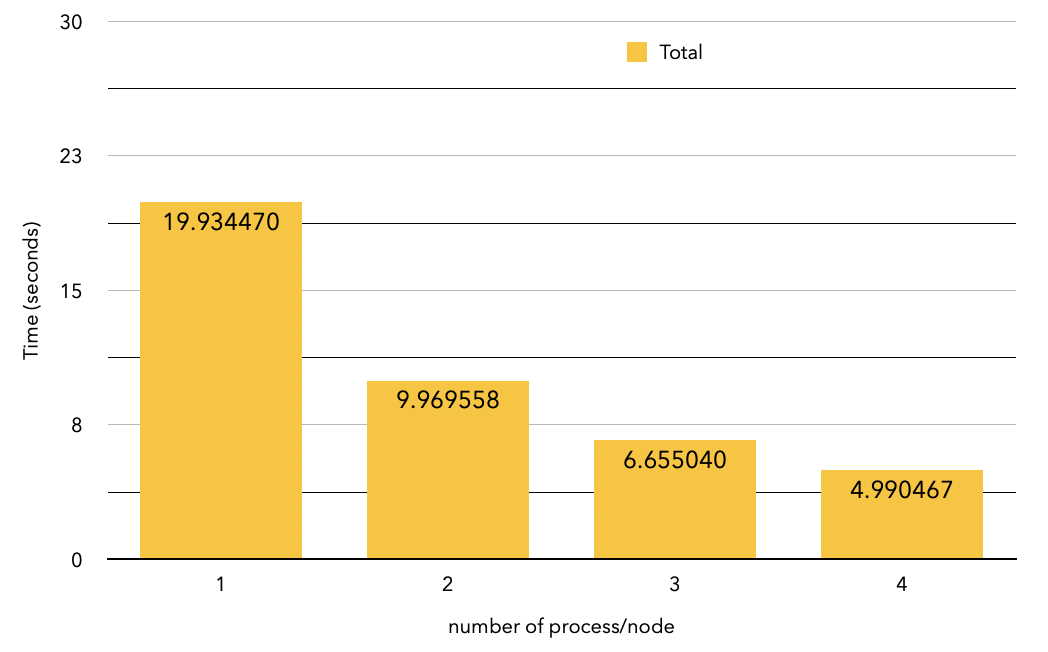
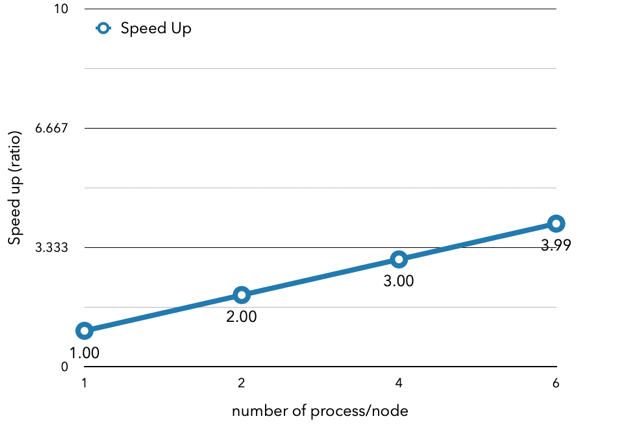
▲ 2 nodes, single process/node

▲ 3 nodes, single process/node

▲ 4 nodes, single process/node

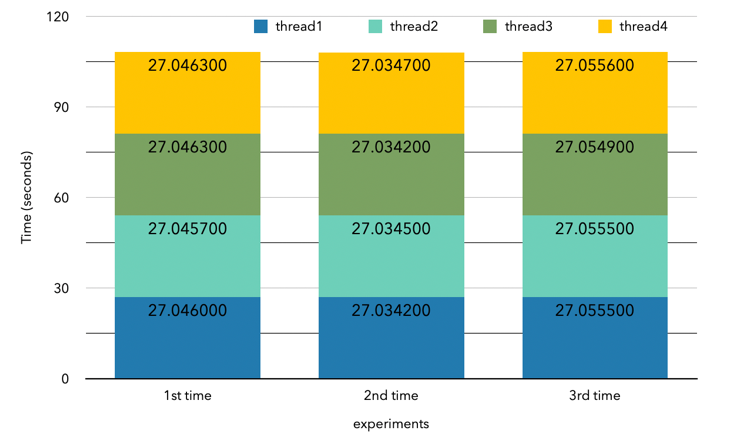
* 固定每個process所擁有的thread數量（4個），改變process數量

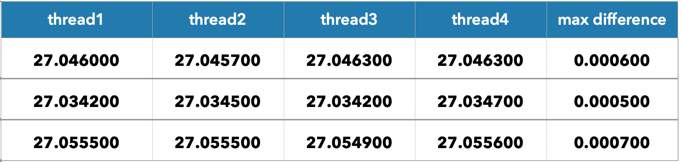
▲ 2 nodes

▲ 3 nodes

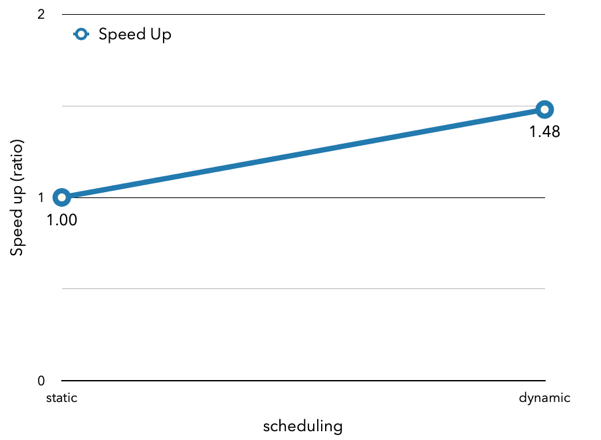
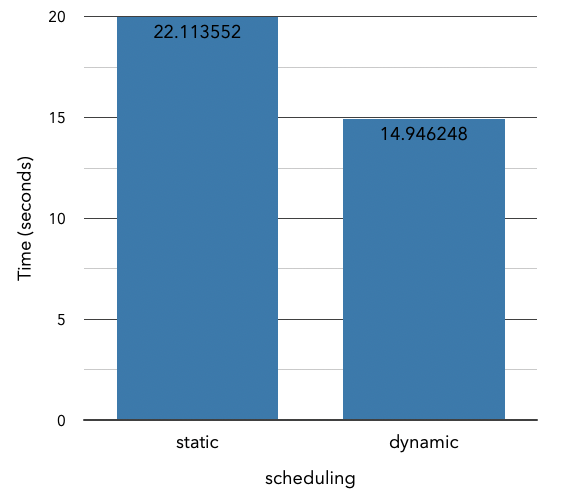
* + 1. balanced experiments
       1. Pthread
* 單純查看每個thread之間所花時間的差異，重複三次

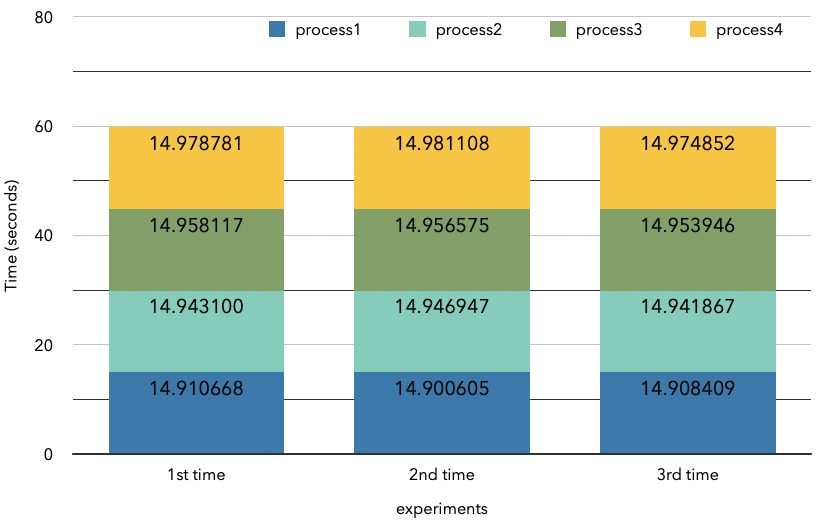
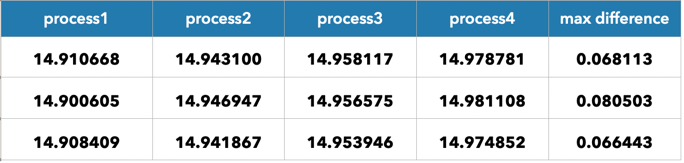
固定thread數量為4





* + - 1. Hybrid
* 改變OpenMP scheduling，比較兩者的差異

共進行三次取平均，設定固定為2 node, 2 process per node, 2 thread per thread

* 觀察每個process之間所花時間的差異，設定同上點
  1. Discussion
     1. 從上面的數據可以看出，隨著thread/process的增加，計算的速度也成比例地變快，因為能完整分配掉所有計算工作到所有的thread上，不會受到其他overhead影響。也可以發現，不同的process數量搭配上不同的thread數量，最後相乘起來的總量相同的話，Computing time其實也差不多，並不會單獨受到process或thread數量的影響，只和最後有多少人能做工作有關。
     2. 在Pthread版本中，每個thread個別的計算時間非常接近、平衡，最大的差異也在0.0005~0.0007秒之間而已。因為有使用像是dynamic scheduling的thread pool，讓各個thread算完一個點後就可以繼續接著算下一個點，充分利用到資源，是以計算量來分配的概念，而不是在初始時就給定固定點的數量，導致分配不平均的可能。

在Hybrid版本中，process的層級並沒有特別做處理，偏向平均分配數量，因此process間的Computing time差異稍微大了一些，約在0.06~0.08秒之間。而觀察thread的層級的scheduling方式，可以看到使用了dynamic scheduling能夠有約1.48的speed up，計算能更好的分給thread去執行，減少時間的浪費。

1. **Conclusion**

這次的作業讓我又稍微更熟悉了三個平行程式設計相關的interface，不過在Hybrid版本沒有嘗試再把Pthread加進去試試，如果有機會嘗試的話感覺也挺有趣的。也是第一次實作在作業系統時就學到的thread pool！而遇到比較困難的地方是在實作Vectorization時，對於它的概念一開始還不太清楚，不太會打而且還常常報一長串錯誤，慢慢打著才了解，最後看到時間大幅減少，也覺得它真是強大！最後是我覺得Mandelbrot set居然可以畫出那樣的圖形，有魔性又帶有規律，實在是太酷了。