CS542200 Parallel Programming

Homework 2: Mandelbrot Set

107062103 王依婷

I. Implementation

a. 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。

b. 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。

```
301   #pragma omp parallel for shared(image) schedule(dynamic, 1)
302       // an iteration handle a row
303       for(int j = rank; j < height; j+=size){
304           __m128d y0, x0;
305           __m128d x, y, x2, y2;
306           __m128d length_squared, int2;
307           int pt_cnt = 0;
308           int pt_now[2];
309           int repeats[2];
310           bool fin[2];
311                     每個 process 都會跑同一份 code（MPI），
312           //Initiali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。

```
304         __m128d y0, x0;
305         __m128d x, y, x2, y2;
306         __m128d length_squared, int2;
307         int pt_cnt = 0;
308         int pt_now[2];
309         int repeats[2];
310         bool fin[2];
311
312         //Initialize
313 >       for(int k = 0; k < 2; k++){…
326         }
327
328         while(pt_cnt <= width){
329             y = _mm_mul_pd(x, y);
330             y = _mm_mul_pd(int2, y);
331             y = _mm_add_pd(y, y0);
332
```

(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。

```
105         while(pt_cnt <= width){
106             y = _mm_mul_pd(x, y);
107             y = _mm_mul_pd(int2, y);
108             y = _mm_add_pd(y, y0);
109 
110             x = _mm_sub_pd(x2, y2);
111             x = _mm_add_pd(x, x0);
112 
113             x2 = _mm_mul_pd(x, x);
114             y2 = _mm_mul_pd(y, y);
115 
116             length_squared = _mm_add_pd(x2, y2);
117 
118             ++repeats[0];
119             ++repeats[1];
120 
121             // 0的位置的點算到了
122 >           if((repeats[0] >= iters || length_squared[0] >= 4) && !fin[0]){…
138             }
139 
140             // 1的位置的點算到了
141 >           if((repeats[1] >= iters || length_squared[1] >= 4) && !fin[1]){…
157             }
158         }
159         // 0還沒算完
160 >       if(!fin[0]) {…
176         }
177         // 1還沒算完
178 >       if(!fin[1]) {…
194         }
```

II. Experiment & Analysis

i. Methodology

(a) System spec: 使用課程所提供的，並沒有使用其他額外系統。

(b) 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。

2. 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。

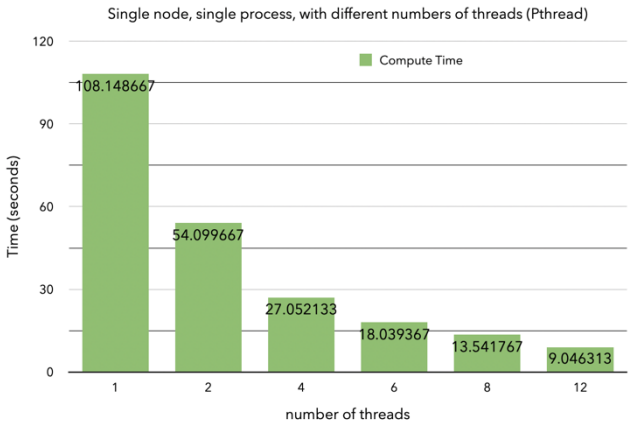
ii. Plots

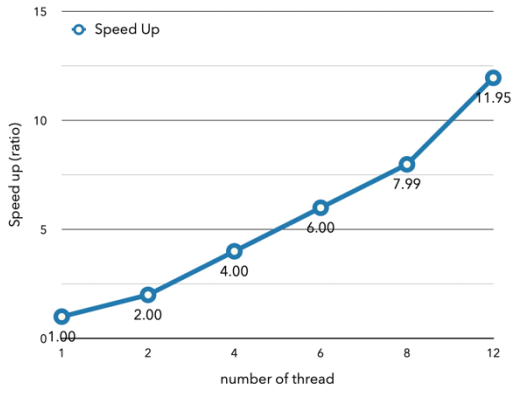
(a) scalability experiments

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

1. Pthread

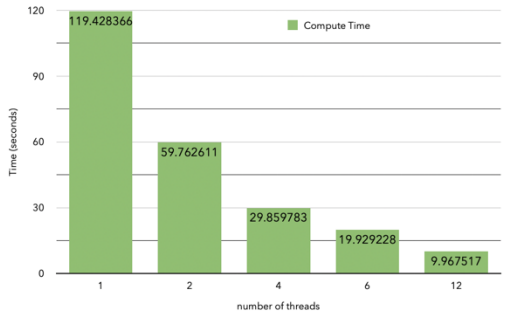
由於限定 Pthread 只有一個 process，因此實驗只有改變 thread 數量的部分。

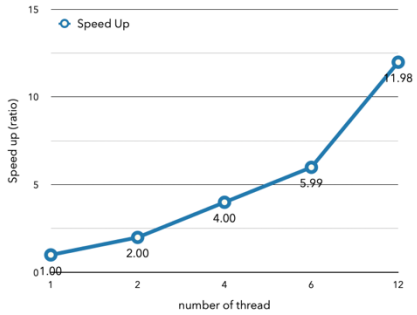




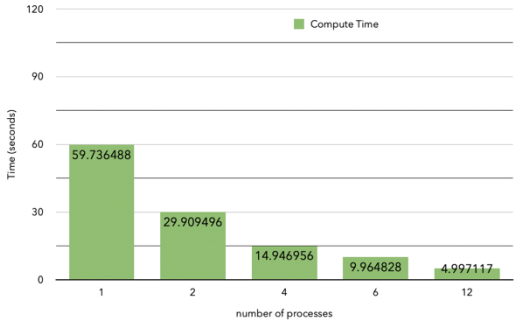
2. Hybrid

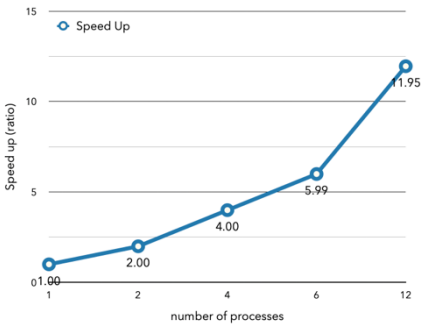
- 固定 node 中的 process 數量為 1，改變 thread 數量。從 single node 到 4 node 都分別做實驗。



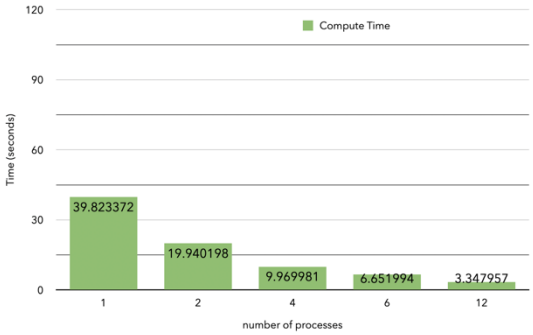


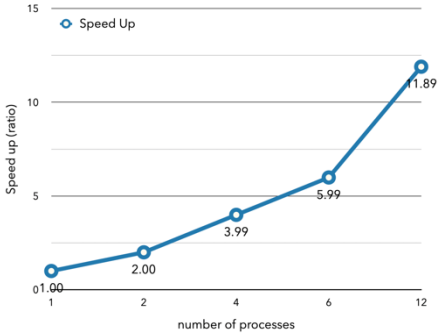
▲ single node, single process/node



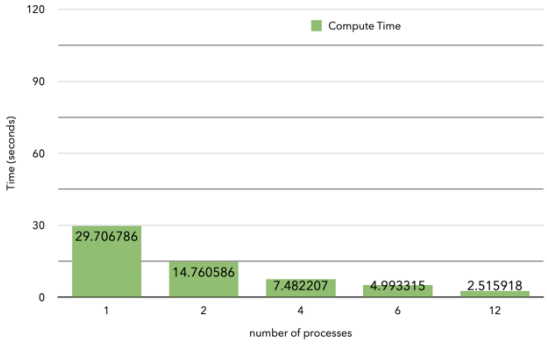


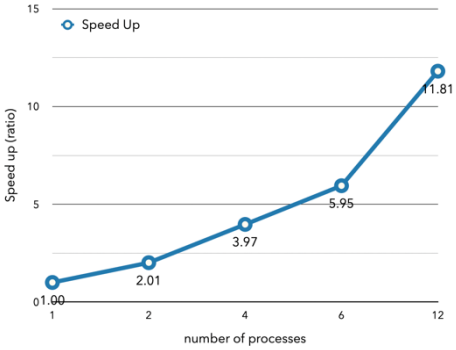
▲ 2 nodes, single process/node





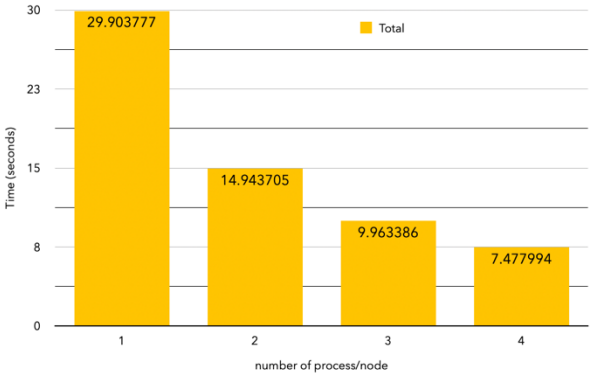
▲ 3 nodes, single process/node

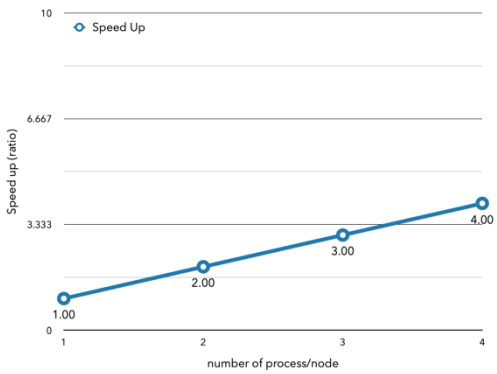




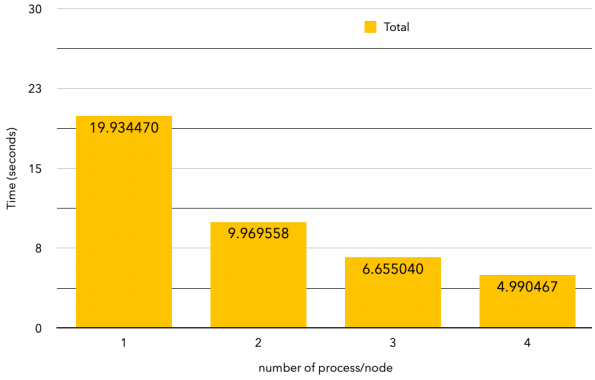
▲ 4 nodes, single process/node

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

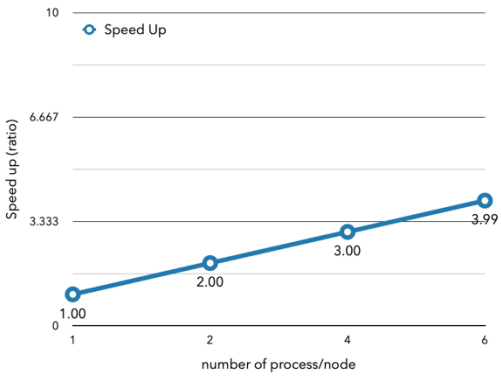




▲ 2 nodes



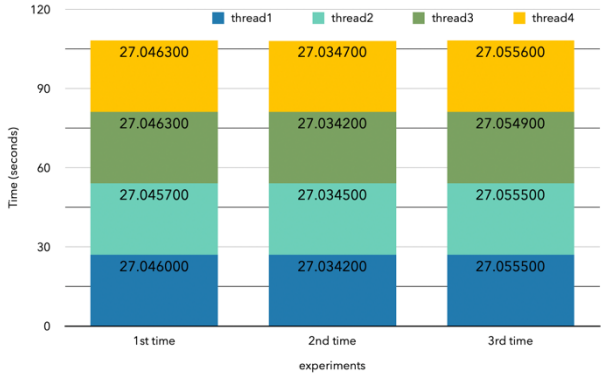
▲ 3 nodes



(b) 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

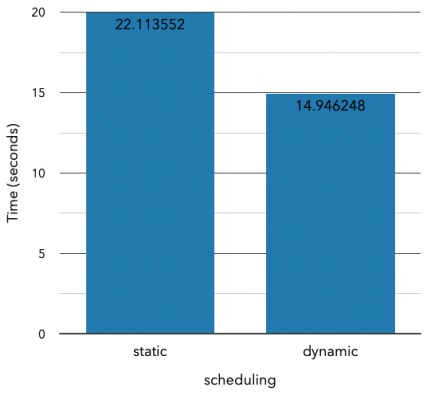


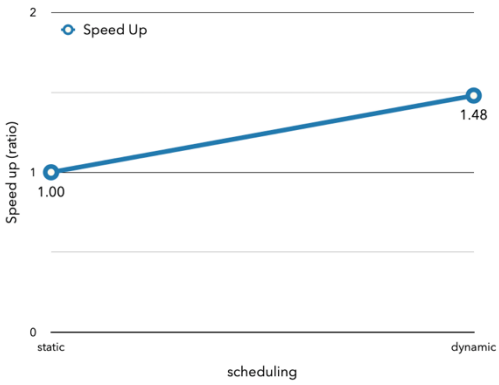
thread1	thread2	thread3	thread4	max difference
27.046000	27.045700	27.046300	27.046300	0.000600
27.034200	27.034500	27.034200	27.034700	0.000500
27.055500	27.055500	27.054900	27.055600	0.000700

2. Hybrid

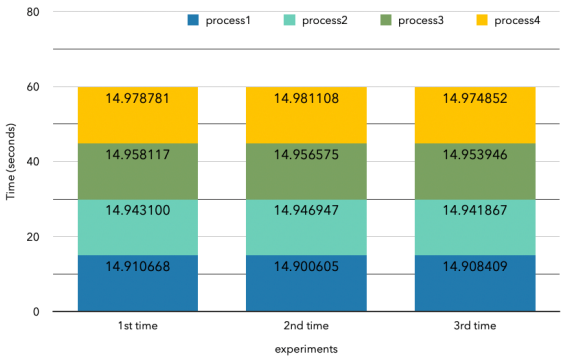
- 改變 OpenMP scheduling，比較兩者的差異

  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



process1	process2	process3	process4	max difference
14.910668	14.943100	14.958117	14.978781	0.068113
14.900605	14.946947	14.956575	14.981108	0.080503
14.908409	14.941867	14.953946	14.974852	0.066443

iii. Discussion

(a) 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。

(b) 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。

III. 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。