# CS542200 Parallel Programming Homework 2:

# Mandelbrot Set

112062673 吳文雯

## 1.Implementation

### Hybrid version(hw2b):

- version 1(分段切割圖片):

一開始透過觀察圖片，看起來圖案大多蠻對稱的，因此決定用 height 來根據 process 數量分割區塊，除了最後一個 process 外，每個 process 處理 ceil(height/size)個 row，剩下的再交由最後一個 process 處理。

Process 內部透過 OpenMP 產生相當於 core 數量的 threads，分配工作的方式是利用 schedule(dynamic,1)，讓完成工作的 thread 再繼續計算後面的 pixel。每個 process 各有一個 buffer 能夠儲存 thread 計算完的結果，之後再透過 MPI_Gather，把各自的 buffer 內資料都交給 rank 0 來寫入圖片。

由於 MPI_Gather 會自動根據 rank id 的順序擺放資料，因此在寫入圖片時不需做額外處理，寫法跟助教提供的版本相同。

然而這個版本的效能並不好，後來發現 Mandelbrot Set 的圖樣其實非常多變，也未必都是上下對稱的，像是後面實驗選擇的 strict31，右下方就有一大部分的黑色區塊，因此這樣的分割方式對於 process 間的 load balancing 來說非常不適合，就改寫了第二個版本，效能比較會在實驗部分呈現。

- version 2(跳著切分圖片):

在這個版本中一次以一個 row 為單位，每個 process 取完就換下一個，因此負責的 row 編號會間隔 process 數量，這樣的方式會讓每個 process 都能處理到圖片中的每個大區塊，相對第一版來說所需的計算時間更為平均。

這裡一樣要計算 p=ceil(height/size)，除了用來配置每個 process 內部的 buffer 大小，更重要的是在 write png 需要透過這個數字來重組 pixel 計算結果，取完第一個 row 後必須加上 p，才是下一個 row 資料在 image 這個陣列中的起始位置，透過這種取資料的順序才能產出正確的圖片。

## Pthread version(hw2a):

- version 1(without vectorization):

使用的 thread 數量設定為跟 cpu 數量相同,因為如果一個 core 上有超過一個 thread,就會需要耗費額外的 context switch 時間，對於效能較為不利。

由於我是先寫 hybrid 再寫 pthread 版本，因此有了前面的經驗，在實作 pthread 時就直接讓每個 thread 如同原本的 process，以一個 row 為單位，間隔 thread 數量取負責的 row 來計算。

因為 pthread 在函數的參數傳遞上不如 MPI 來的便利,因此我將 cal 函數內所需的大部分參數都宣告為 global，包含用來儲存計算結果的 image 陣列。由於每個 thread 裡的 stack 是私有的，repeat 這個變數並不會被其他 thread 更改，且存入 image 陣列的 index，也會因為每個 thread 負責的 row 編號不同而異，因此即使不用 lock 也不會產生錯誤，能夠避免程式執行時間因同步化而增加。

在寫入圖片方面，由於 pthread 可以直接寫到 image 陣列適當位置，不需要像 MPI 一樣要先傳遞資料再重組，因此並不需要特別更改 write png 函數，就可以產出正確的圖片。

這個版本雖然可以順利通過 judge，但是在排名部分算是蠻後面的，考量 pthread 版本的程式碼相對 hybrid 來說較為簡潔，因此決定嘗試加入 vectorization，就有了第二個版本。

- version 2(with vectorization):

這個版本跟前面的差異只有在 cal 函數中加入了 vectorization 來加速計算,寫入圖片部分依然維持原本的 sequential 方式。

採用的工具是 SSE，由於我們的 data type 是 double，所以最多只能把兩個 data 打包在一起運算。我選擇的方式是以水平方向，從 0 開始每兩個 pixel 包成一組，這種方式對於我這樣以 row 為單位分配給 thread 的程式行為較為合適,只有當 width 為奇數時,才需要另外計算每個 row 剩下來的那一個 pixel。

另外為了加速運算，我把 length_square 跟 4 的比較也改在 register 內進行，可以減少透過記憶體的資料搬移，只需要透過迴圈進行的次數來遞增 repeat，而不必把運算結果再 load 回記憶體，對於提升速度來說很有幫助。最後只需要把第一筆資料、第二筆資料及當 width 為奇數另外計算的資料依據正確的 index 存入 image 陣列就完成了,不需要特別處理 write png 的部分，就可以產出正確圖片。

## 2. Experiment & Analysis

### i、Methodology

**(a). System Spec:** 使用的是課堂提供的 cluster。

**(b). Performance Metrics:**

比較程式整體執行時間:

- Pthread: 利用 clockgettime(CLOCK_MONOTONIC,…)測量從取得 cpu 數量到 write_png(…)後的時間。
- Hybrid: 利用 MPI_Wtime(…)測量從 MPI_INIT 到 write_png(…)後的時間。

比較 load balancing:

- Pthread: 利用 clockgettime(CLOCK_MONOTONIC,…)測量從 cal 函數中取得 thread id 到 return NULL 前的時間，計算結果為各 thread 的 computation time。
- Hybrid:
  利用 MPI_Wtime(…)測量每個 process 執行 mandelbrot set 計算的 computation time，以及在 MPI_Gather 前後放置 MPI_Wtime(…)來計算 communication time。
  利用 omp_get_wtime(…)蒐集每一個 thread 計算 mandelbrot set 的總時間，每次計算完後會加進陣列中那個 id 的 index 位置，最後再一起印出加總後的時間。

由於 write_png 都是 sequential 寫入，不特別計算 I/O time。

### ii、Plots: Speedup Factor & Profile (including discussion)

### Experimental Method:

- **Test Case Description:** 使用的 testcase 是 strict31，iteration = 10000,
  $x0 = -0.3070888275226523, $x1 = -0.2476068190345367,
  $y0 = -0.6245844142332467, $y1 = -0.6535851592208638,
  Height= 7680, width = 4320。

Pthread version:

- 使用單個 node，單個 process，利用 1、2、4、8、12 個 thread 的設置，觀察程式整體執行時間。

Computation time:

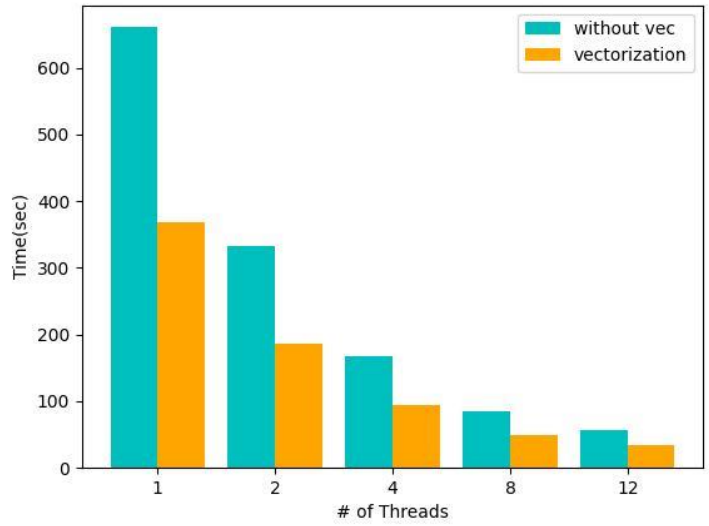


圖 1: pthread vectorization 版本及沒用 vectorization 版本時間比較

由這張圖可以明顯看出，使用 vectorization 的版本可以明顯減少計算時間。

Strong scalability:

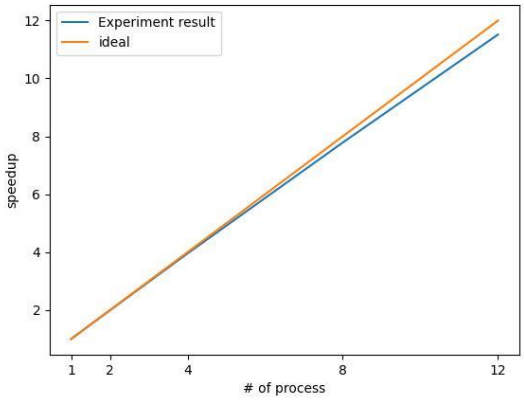


圖 2: without vectorization 版本

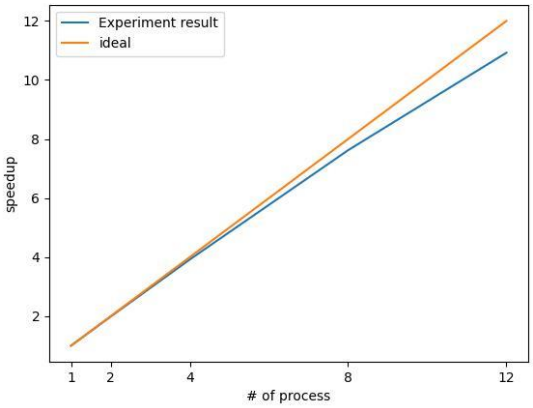


圖 3: vectorization 版本

透過這兩張圖可以看出，這兩個版本的 scalability 都還不錯，即使沒有使用 vectorization，可能因為工作分配的還算平均，因此在加速上的結果還不錯。不過讓我感到驚訝的是，沒有 vectorization 的版本的 scalability 在 thread 數量多時居然會比較好，我猜測原因可能是因為 vectorization 在計算部分加速非常多，最後 bottleneck 可能都在無法平行化的 i/o 部分，使得 scalability 程度趨緩，無法再有更明顯的加速。

Load Balancing among threads:

使用單 node，12 個 thread 來看各個 thread 的 computation time。

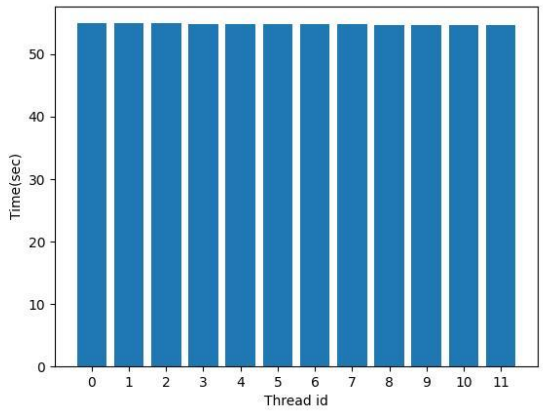


圖 4: without vectorization 版本

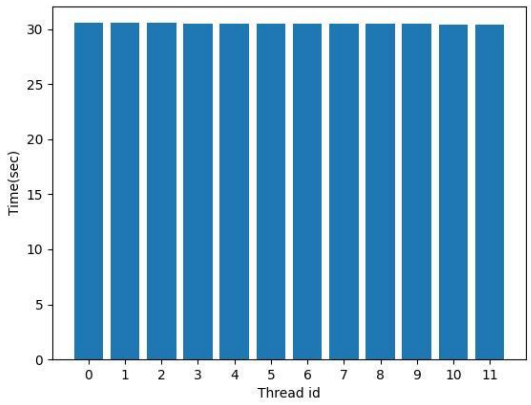


圖 5: vectorization 版本

從這裡可以看見跳著分切 row 的方式使得 thread 之間的 load balancing 算是蠻平均的，這應該也是 scalability 的結果不錯的原因。

Hybrid version:

- 使用 1、2、3、4 個 node，每個 node 一個 process，每個 process 12 個 thread 的設置，觀察程式整體執行時間。

Computation time:

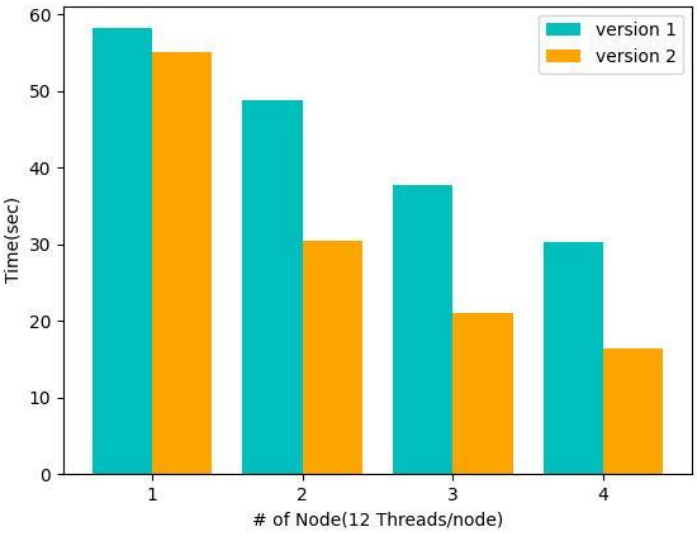


圖 6: version 1 及 version 2 整體執行時間比較

由這張表可以明顯看出，改用 version 2 的工作分配方式可以明顯減少計算時間。

**Strong scalability:**

Single node: 使用 1 個 process，1、2、4、8、12 threads

Multi-node: 使用 1、2、3、4 nodes，每個 node 各 1 個 process，每個 process 各 12threads

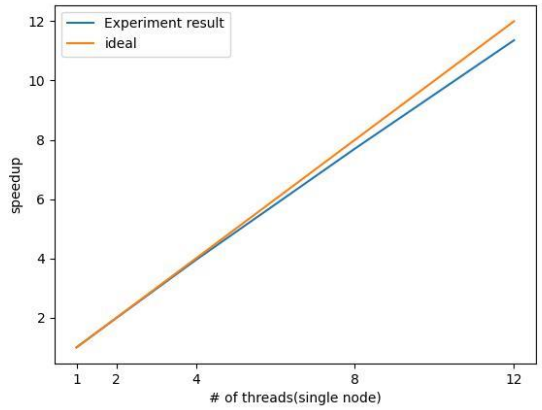


圖 7: version 1 (single node)

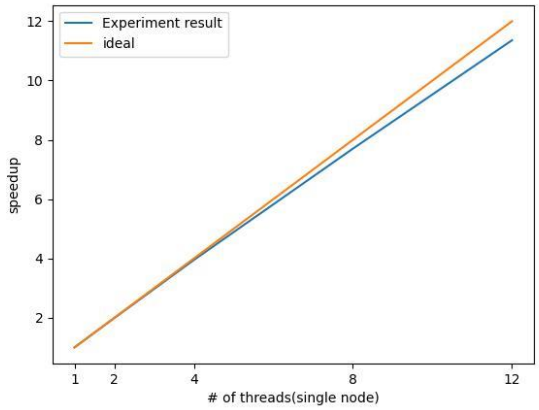


圖 8: version 2 (single node)

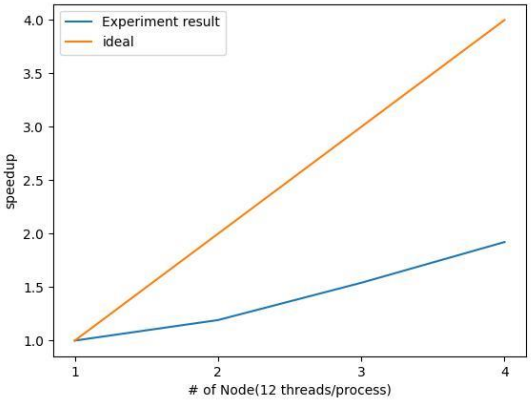


圖 9: version 1 multi-node (12threads/node)

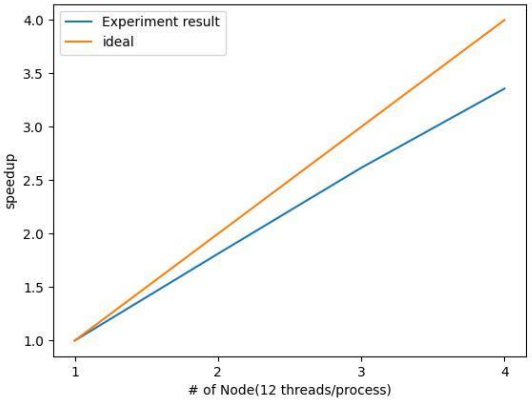


圖 10: version 2 multi-node (12threads/node)

在 single node 的實驗中，version 1 跟 version 2 的 scalability 幾乎相同，這可能是因為 load balancing 的差異只存在在 process 之間，而只使用一個 node 一個 process 的情況下，就看不出 load balancing 的影響。然而當使用了超過一個 node，也就是超過一個 process 時，process 間 load balancing 的影響就變得很明顯，version 1 會因為有 process 特別晚結束，而拖慢程式整體執行時間，當改用 version 2 時因為各 process 的執行時間較為平均，可以同心協力讓程式早點結束，因此有了比較好的 scalability。

## Load Balancing among processes:

使用 4 個 node，4 個 process，每個 process 12 個 threads 來看各個 process 的 load balancing (computation time)。

由於 communication time 跟 cpu time 比例太懸殊，化成圖表幾乎看不見 communication time，因此改用表格呈現數據。

**Version 1 (分段切割):**

| | Cpu time | Communication time |
|---|---|---|
| Rank 0 | 0.946120 | 0.092010 |
| Rank 1 | 7.910275 | 0.091764 |
| Rank 2 | 18.342785 | 0.091747 |
| Rank 3 | 28.441180 | 0.092024 |

**Version 2 (跳著切分):**

| | Cpu time | Communication time |
|---|---|---|
| Rank 0 | 13.898101 | 0.095509 |
| Rank 1 | 13.899186 | 0.134886 |
| Rank 2 | 13.909123 | 0.121769 |
| Rank 3 | 13.919963 | 0.108924 |

從這兩個表格可以看見，一開始按照高度平均分割份量的方式使得 process 間的 computation time 差距非常大，尤其在 strict31 這個 case 中，他的圖案分布在圖片中很不均勻，造成的影響就更大。而當改由 version 2 的分割方式後，每個 process 都可以分到一些較輕鬆的部份以及計算時間較長的部分，加減之後每個 process 的 load 就平衡很多了。

## Load Balancing among threads in the same process:

Single node: 使用 1 個 process， 12 threads

Multi-node: 使用 4 nodes，每個 node 各 1 個 process，每個 process 各 12threads

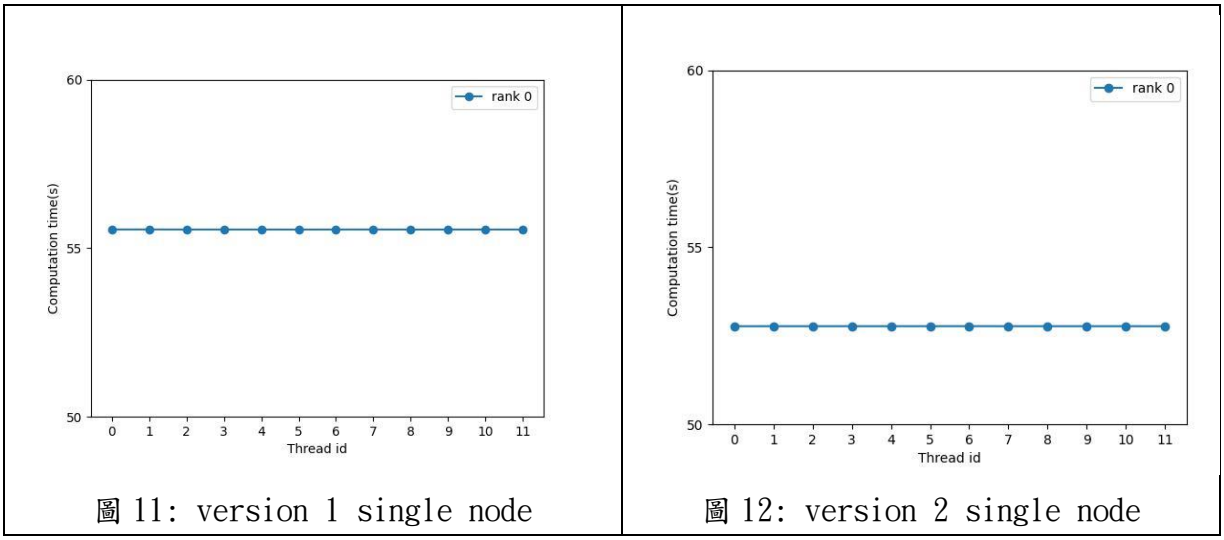


圖 11: version 1 single node

圖 12: version 2 single node

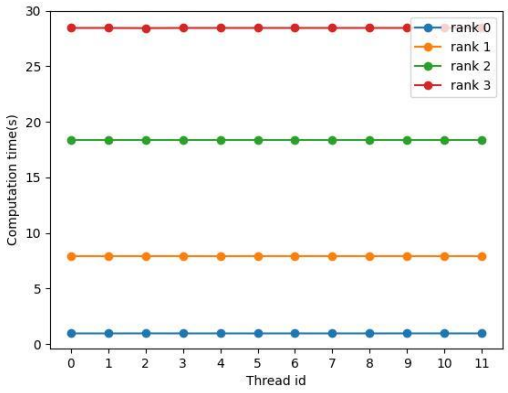


圖 13: version 1 multi-node (12threads/node)

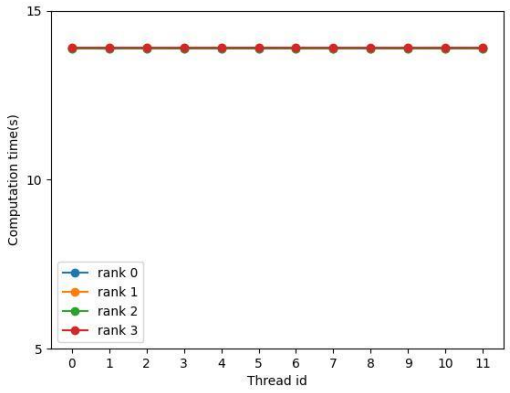


圖 14: version 2 multi-node (12threads/node)

從這裡可以看出不論是 version 1 或是 version 2，在 process 內部的 threads 計算時間都很平均，這可能是因為用 dynamic 的方式，讓完成計算工作的 threads 繼續拿取下一個 pixel，且因為 chunk size 設成 1，可以避免單一 thread 拿到連續兩個要計算很久的 pixel 的情況，可以讓 thread 之間的工作量更為公平。

## 4. Experiences / Conclusion

這次的作業中主要遇到三個比較大的關卡

- 因為使用跳著切分 height 的方式，在 hybrid 版本中必須重組資料，為了能繪製正確的圖片，必須非常了解 write_png 函數內部讀取資料的順序，過程中我發生過好幾次 segmentation fault，也一度產出上下顛倒的圖片，花了許多時間才終於調整出最終正確的版本。
- 在實作 pthread 版本時，其實一開始是想要實作 Thread pool，以一個 row 為單位分配工作，讓 thread 完成計算後再去拿取下一個 row 來計算。但是花了一些時間寫好 thread pool 所需的 struct 及函數後，卻一直發生 deadlock 的情況，後來迫於時間關係只好先改用 static 分配工作，幸好效能也不算太差。
- SSE 算是這次花最多時間的部分，由於我第一次使用 vectorization 這項技巧，對於函數要寫在哪、怎麼寫、要用甚麼函數都非常陌生，因此花了很多時間上網查看各種範例或是官方文件，好不容易把程式碼都「翻譯」好，卻發現必須把 while loop 條件判斷式拆成兩部分判斷，才能分別算出一組 pack 內的兩筆資料分別要走幾個 iteration。其中為了讓第二筆資料能夠正確被拿來跟 4 比較，必須使用 shuffle 這個函式，關於這點我也卡了好久才發現問題所在。最後看見 SSE 帶來接近 2 倍的加速真的覺得很值得，希望未來有機會運用到如 integer 等 bit 數比較少的 data type，能夠帶來更多倍的加速。

整體而言，這次作業嘗試了一些新的方法與技巧，雖然過程充滿挫折也耗費許多心力，但是看到結果真的蠻有成就感的!