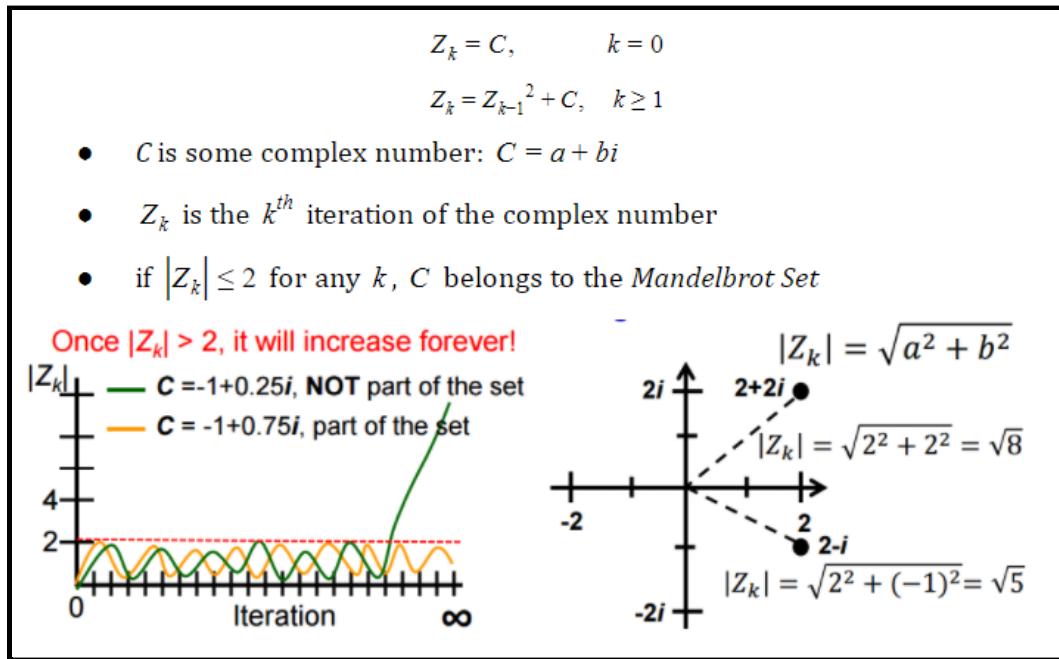
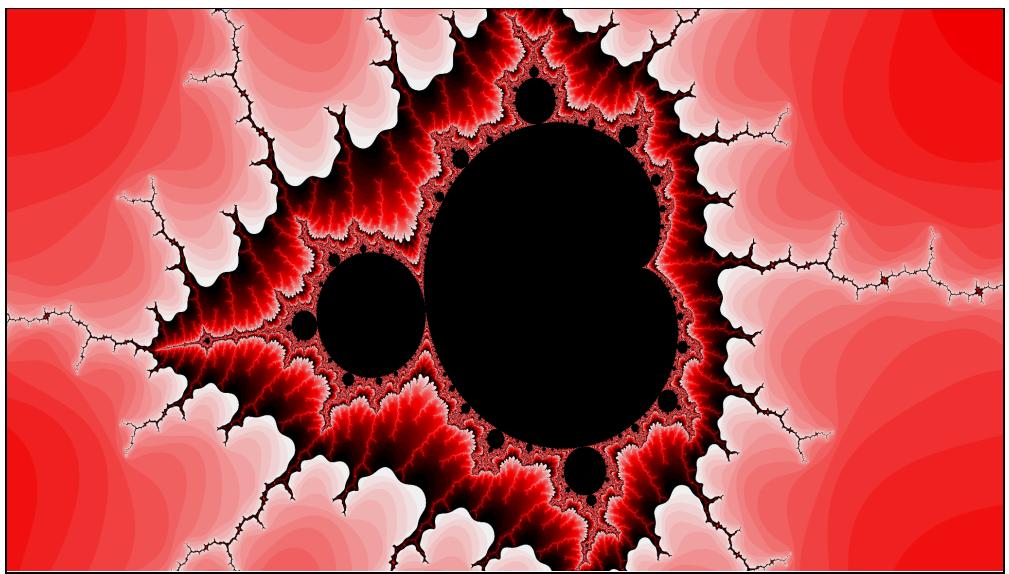


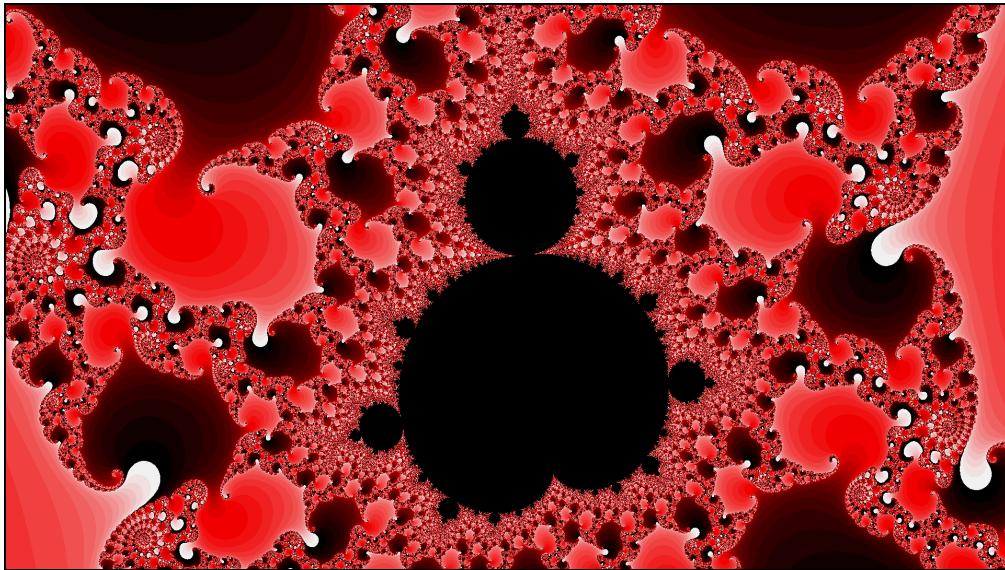
平行程式作業二報告

李浩榮 110062401

1. Implementation 實作







這次的作業目的是以**Pthread** 和 **MPI+OpenMP** (Hybird)兩個版本來實作**Mandelbrot Set**。如上圖所示，我們需要考慮的是x和y的座標值，以確保z向量的距離不會超過2。所以這裡我固定以**height**去切分，以達到load-balancing的目的，加速運算。

1. Pthread

- pthread這裡實作的目的是要看單一process內多個thread的處理速度。
- 單一process(或node)內的thread互相溝通的成本比跨越到其他process(亦multi-node)的成本來得低，理由是跨越會受到bandwidth的影響，導致latency, delay。
- 使用struct攜帶各種thread需要用到的資訊去處理 (如: left,right,upper,lower,thread_id, num_threads, image)。
- 以height為指標，每個thread拿一個height(從lower到upper為止)，去處理對應的width，直到全部height都結束。
- 這裡的height以一個current_row標記，以顯示目前大家thread處理的height到什麼進度了。每個thread若已經先完成手上的計算，那就繼續拿新的height去做，不讓thread有閒暇的時間。以達到時間上的utilize。

- 這裡lock和unlock是架在每個thread去要新的height的時候，以確保不會有人還沒有update新的current_row的時候，出現搶到同樣的height的情況。
- pthead這裡大家都是access同一個image，但是由於寫進去的位置不一樣所以不用做lock來保護critical section，當大家都跑完後，image就可以直接寫出到png。

2. MPI + OpenMP

- MPI負責不同process間的溝通。
- OpenMP負責一個process內的thread們溝通。
- 這裡是multi-node，所以如果以大家都去拿同一個height來做，其中的critical section會有卡著的情況，latency很容易上去，所以不建議這麼實作。
- 我是把height平分給每個process（這裡好像最多只能有四個process，每個process內有最多12個thread亦CPU），若不能整除則表示前height_remain個process會有多一筆資料的情況。
- 每個process內再以OMP的dynamic自己去分配12個num_threads要一次拿幾個資料去處理。
- hw2b的實作難度是每個process拿到的height區間不一樣，所以計算的時候的lower要確保有在對的值，且前height_remain個process需要處理的資料量是height_per_process+1個，其餘的只要處理height_per_process個height。
- 由於可能會出現四個process，個別拿了n+1, n+1, n, n的資料，所以local thread處理好的image要使用MPI_GatherV來實作個別thread回收的stride區間位移的功能，確保每個thread的image能順序且memory位置緊鄰（否則accurate精準度不會是100%）地寫到root_image上。（關於MPI_GatherV的運作圖我有在3.4的others附上）
- 使用rcounts來記錄每個process處理幾筆資料的大小，亦 $(height_per_process+1)*width$ 。
- 使用displs每個process寫到root_image的memory上的起始位置 $(displs+rcount)$ 。

2. Optimization & Effort Enhance 優化與改進

2.1. Pthread

- 先說原本我使用的方法是每個thread平均分配height，並且最後再使用MPI_Reduce回收各種。這裡會出現問題。
- 問題是每個thread分配的平均個height去處理，你以為會很公平，且可以達到最好的load-balance，然而不是，因為有些thread處理的數值比較大，或者小數沒有多，會比較早結束；相對的有一些thread會比較慢，以此為基礎，越來越多case，越來越多height處理下，每個thread的進度就會越拉越大，最後要收回的時候就會出現花很多時間在等其他thread的情況。
- 優化的解法是，height以current_row的方式放上去global variable。只要有thread已經做完目前計算時間比較短的case，就拿新的height繼續做，確保每個worker都被utilized。
- 再來是struct args攜帶資料的方式，我也有使用了把各種參數以global的方式去存寫，這種情況很容易出現導致不同thread同時access同一塊memory，雖然只是讀取，但還是不太好的寫法。所以個人就以struct傳入各個thread，需要的變數再自己local算出來（算一次放全域和每個thread自己算一次放local，這部分差別的延遲其實對於整個程式來說很小，畢竟不是到跨process的情況，所以以struct的寫法比較安全）。

2.2. MPI + OpenMP

- OpenMP本身就是已經被寫得很完善了，所以基本上沒有什麼可以優化的。唯一的差別是static和dynamic，但是我兩種方式都試過了，看是要自己決定每次thread一次拿多少筆去處理，還是讓它自己動態處理都沒有太多的效果，所以就擇一既可。

- 再來是即使你已經有成功讓每個thread從對的lower(height)開始去計算，但是你最後如果使用Reduce回來的數值也會有差異。因為MPI_Reduce或者MPI_Gather的buffer都是固定的，所以會出現有一些local_image的memory的值沒有被寫到root_image上，準確率就不會是100%。這裡要使用MPI_GatherV可以自由控制回收的每一筆資料的區間(我稱為stride)確保每個process回收的image能夠寫在對的位置上。

3. Experiment & Analysis 實驗與分析

3.1. Methodology 方法

3.1.1. System Spec 設備規格

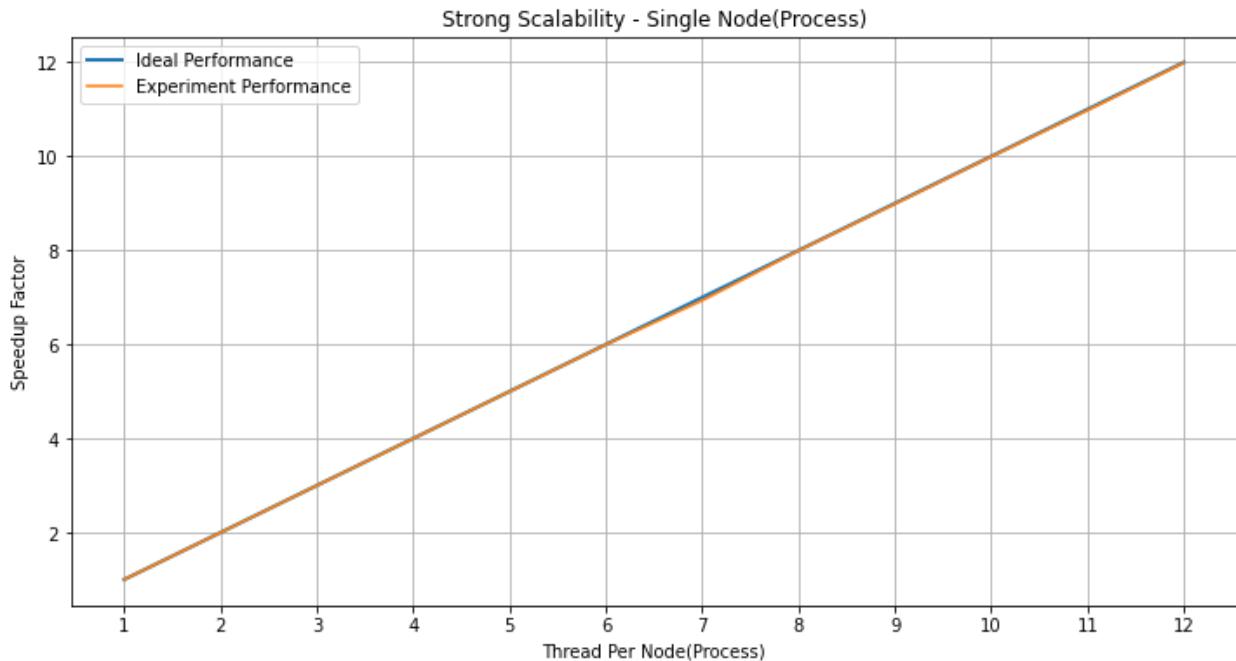
- 每個人分配到的配額是最多4個Nodes(Process)，每個Node(Process)有12顆CPU(thread)，如果一個CPU跑一個thread，最多可以跑48個thread。
- 48GB disk space per user
- OS: Arch Linux
- Compilers: GCC 10.2.0, Clang 11.0.1
- MPI: Intel MPI Library, Version 2019 Update 8
- Scheduler: Slurm 20.02.5
- Network: Infiniband

3.1.2. Performance Metrics 衡量標準

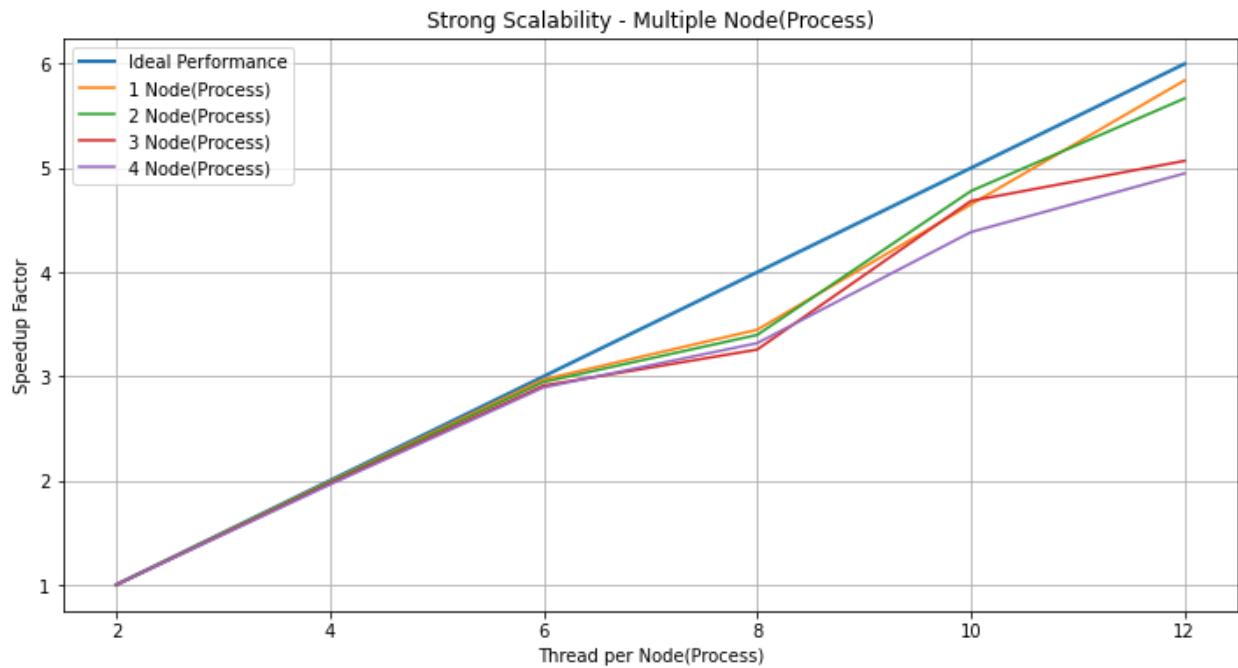
- 這次的作業由於要看load-balancing，所以時間上主要focus在程式內對於資料的運算時間，及MPI在各個process間的溝通時間兩者的總和，亦
 - TOTAL_TIME = CPU_TIME (for hw2a pthread)
 - TOTAL_TIME = CPU_TIME + COMM_TIME (for hw2b MPI+OMP)
- 我是使用clock_gettime(CLOCK_MONOTONIC)來取程式開始以及結束的時間差之和。

3.2. Scalability & Load Balancing

- Scalability的實驗是以助教給的testcases19為例。參數為 (10000
0.3380234375 0.3399765625 -0.0519765625 -0.0500234375 6443 6443)
- Testcase19 相對其他測資來說是比較不會跑太快而看不出程式 performance且也不會花太久時間。且大家在這個 testcase19 在 scoreboard上的表現基本上可以分為三個部分，分別是(13秒,25秒,34秒)。由此可見該測資對於程式的寫法很敏感，可以很明顯地反映出每個人的程式差異。
- 下圖是 hw2a 在 single node(process)下跑不同 thread 數的 Strong Scalability表現。



- 下圖是 hw2b 在multinode(process) 下跑1~12個 thread 的 Strong Scalability表現。



3.3. Discussion 討論

- **Discussion on Strong Scalability of Single Node**
 - 在hw2a的singlenode下，我們可以看到我們的表現非常貼近Ideal Performance，甚至大部分的地方都是重疊的，除了當thread數是7的時候會有一點小卡，除此之外都是完美的。
 - 會有這種表現其實不意外，首先這個hw2a是只使用single process，全部thread在一個process內部溝通的成本是比較低的，且全部thread寫出image的位置也不同，所以不會用critical section包著，也節省較多餘的時間。
- **Discussion on Strong Scalability of Multi Node**
 - hw2b為例的multinode的圖示顯示各個不同node(process)數量去處理不同thread數的strong scalability。請注意這張圖四條線是沒有

直接互相關係，這只是為了表示不同node數量的成長幅度而畫在一起。

- 從圖中可以合理得出：
 - 隨著node(process)的數量上升，程式執行所需時間越少。
 - 隨著thread的數量上升，程式執行所需的時間也越少。
 - 四個process所需的時間比一個process所需的時間少。
 - 但是一個process隨著thread數的程式執行的速度增加的幅度比四個process增加的幅度來得大很多。
 - 明顯可以驗證平行化的避免跨node(process)的溝通，內部thread溝通的成本比較低的事實。

- **Discussion on Load Balancing**

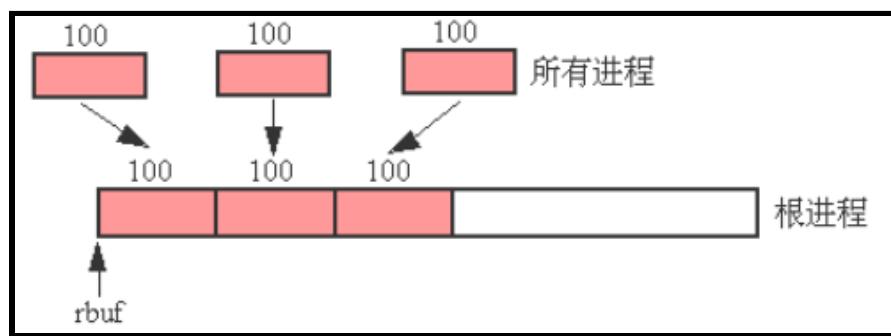
HW2a Pthread (1 process, 12 thread)		HW2b Hybird (4 process, 6thread)	
Thread_ID	Time(s)	Rank	Time(s)
0	23.407630	0	12.441299
1	23.407668	1	12.330357
2	23.406855	2	12.297908
3	23.407918	3	12.328760
4	23.407292		
5	23.407473	HW2b Hybird (4 process, 12 thread)	
6	23.388191	Rank	Time(s)
7	23.388852	0	7.236683
8	23.388715	1	7.154310
9	23.397880	2	7.132218
10	23.398436	3	7.128882
11	23.388262		
		HW2b Hybird (48 process, 1 thread)	
		Rank	Time(s)
		0	8.008895
		1	7.826687
		2	7.822792
	
		47	7.886810

- 上圖我做了hw2a的pthread以及hw2b的hybird版本的load-balancing的執行時間表現。

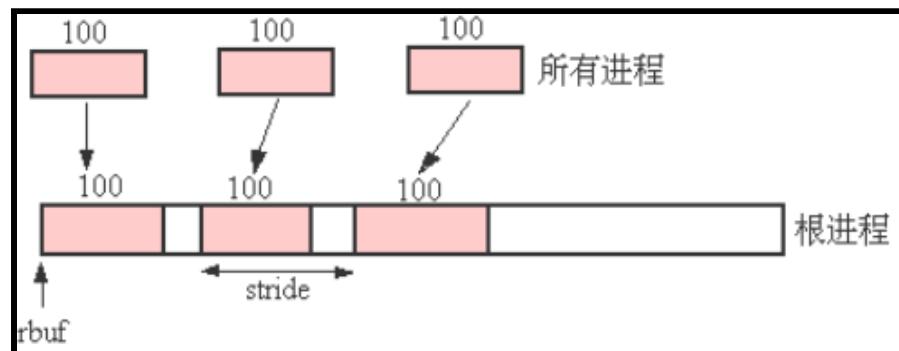
- 可以看到hw2a下幾乎全部thread的時間都一樣，合乎理想情況，因為大家是一起寫上去image對應的位置。
- Hybird版本的部分雖然我是使用只有rank==0的root process去做 write_png寫圖片的動作，自然會有一個rank也就是rank0的會比較慢是正常的。但是和其他的時間相比，還是有一點久，如果MPI寫入的方式再改成大家一起寫應該會更快。

3.4. Others 其他

- 下圖是上面提到的一般MPI_Reduce, 或者MPI_Gather的回收方式。



- 下圖是MPI_GatherV具有接收不同進程不同數量的數據的特性，提高了MPI的靈活性的架構示意圖。



4. Experience & Conclusion 總結

最後這份作業的實作目的不外乎讓我更加了解在Process間溝通的MPI,在thread間溝通的Pthread或OpenMP的運作方式，比起作業一對於平行化有更深的認知。接下來如果要做優化可以從Load-balancing下手。我們可以看到目前來說程式的Bottleneck不外乎是希望降低Node(process)間的網路BandWidth Latency, 如果每個都可以縮短一點時間，那大量的Node(process)回收且縮短的時間就應該會相當可觀！