Homework 2: Mandelbrot Set

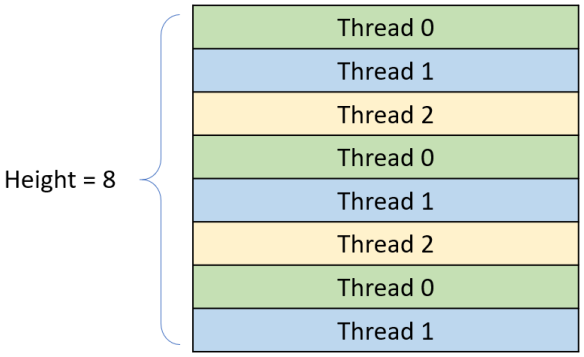
110062511 倪滙渝

■ Implementation

hw2a: Pthread parallelism

將 mandelbrot set 的計算寫成一個 pthread function，接著 iterative 做 height 的時候，每個 thread 會依據 index 的 remain 值分配，並計算其分配到的 height 中所有 width 區域。由於 mandelbrot set 可能在某個 height 區間 iterative 的次數會特別多，因此這樣分配能夠減少有 bottleneck 的情況，達到較好的效能。

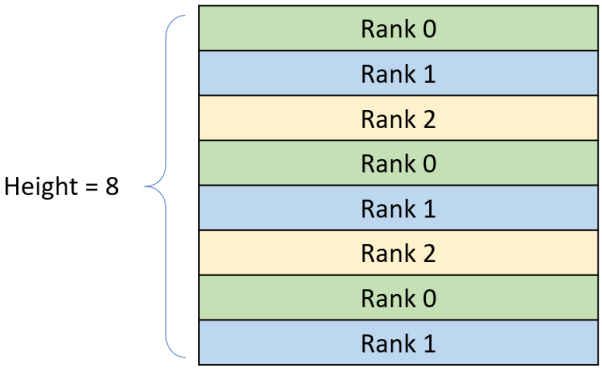
Ex: image height=8 and #thread=3



hw2b: Hybrid parallelism

切割不同區間的 height 給各個 MPI rank，同 hw2a 的分配方法。接著利用 omp parallel 以及 omp for schedule (dynamic) 來分配給 thread 計算，以此去平均分配每個 thread 的計算時間，達到較好的平行度。每個 rank 計算結束之後，會再傳給 rank 0 來整合各部分的計算結果。

Ex: image height=8 and #rank=3



■ Experiment & Analysis

✧ Methodology

使用 C 語言中的 clock_gettime()來計算時間。

✧ Performance Metrics

hw2a: Pthread parallelism

- Computing time
  包含範圍：計算該 pixel 是否符合 Mandelbrot Set
  計算方法：所有 thread 的平均執行時間
- IO time
  包含範圍：將計算結果轉換成 image
  計算方法：只有一個 process 輸出 image，直接計算

hw2b: Hybrid parallelism

- Computing time
  包含範圍：計算該 pixel 是否符合 Mandelbrot Set
  計算方法：有兩種實驗，分別計算所有 rank 或 thread 的平均執行時間
- Communication time
  包含範圍：MPI_Irecv、MPI_Wait
  計算方法：由於使用的是 master-slave 的架構，因此計算 bottleneck rank 0 的接收時間
- IO time
  包含範圍：rank 0 複製各個 MPI 結果 + 將計算結果轉換成 image
  計算方法：rank 0 的時間

✧ Plots: Scalability & Load Balancing

- 實驗參數

iters = 10000
x = [-2,2)
y = [-2,2)
width = 4000
height = 4000

hw2a: Pthread parallelism

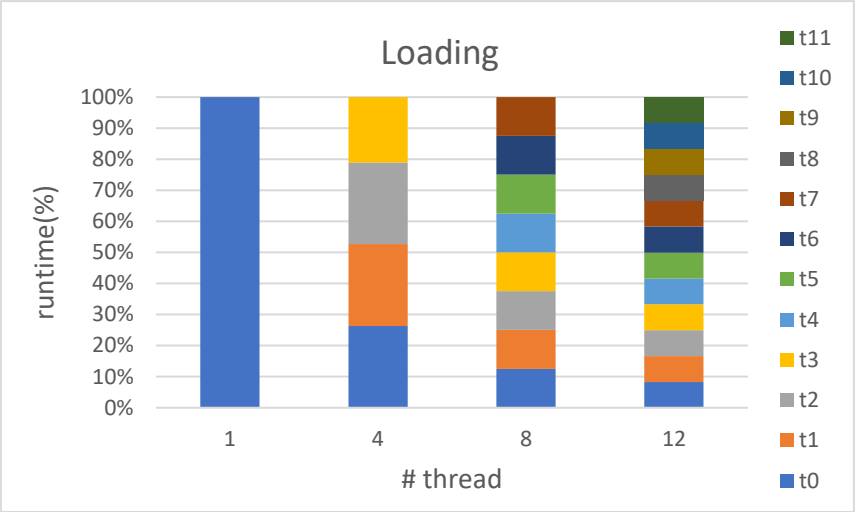


Figure 1: Loading of each thread (single node)

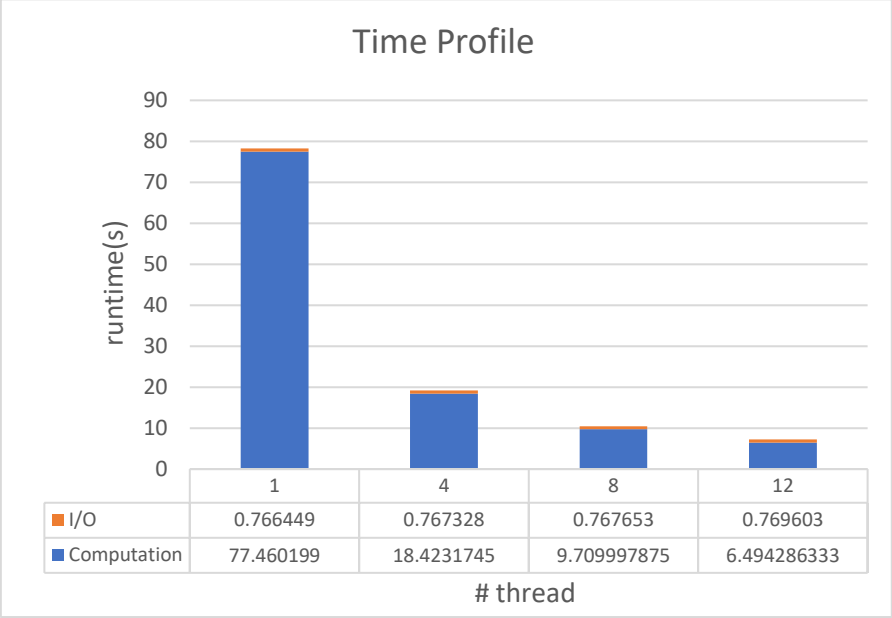


Figure 2: Time profile of different numbers of thread (single node)

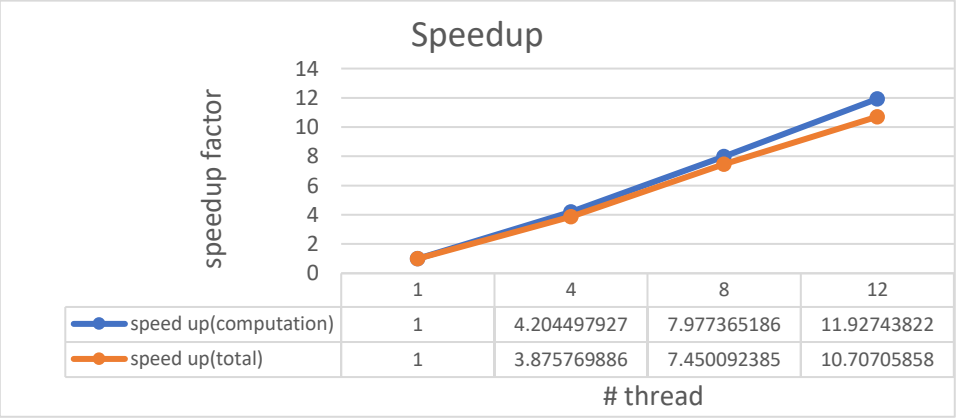
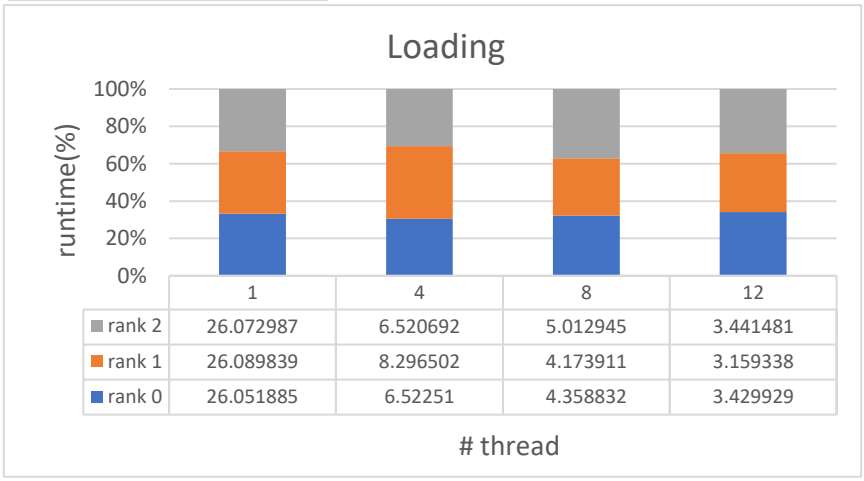


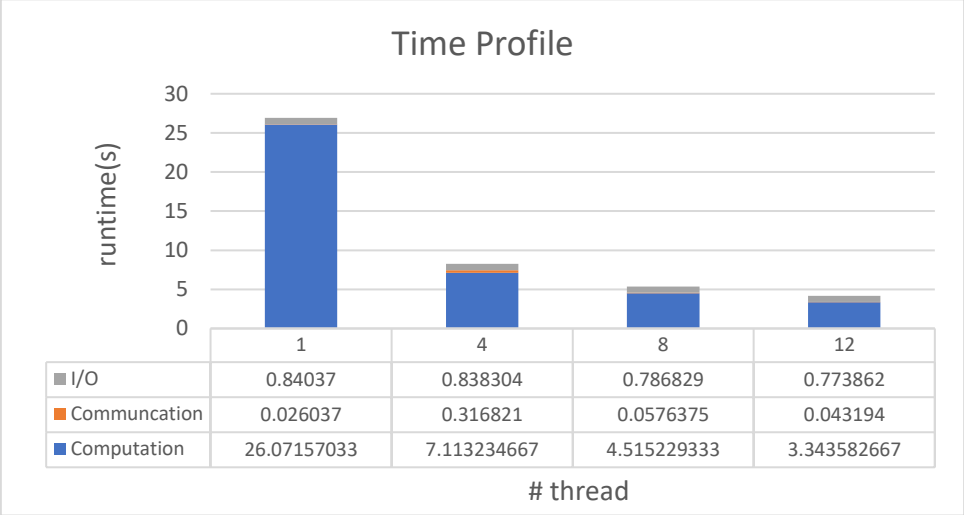
Figure 3: Speedup factor of different numbers of thread (single node)

hw2b: Hybrid parallelism



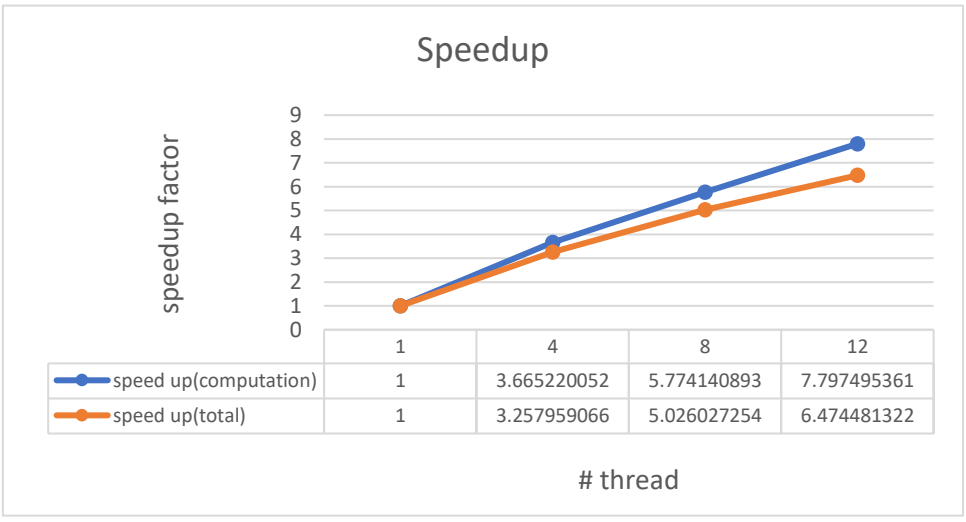
	1	4	8	12
rank 2	26.072987	6.520692	5.012945	3.441481
rank 1	26.089839	8.296502	4.173911	3.159338
rank 0	26.051885	6.52251	4.358832	3.429929

Figure 4: Loading of each node (all #node is set to 3)



	1	4	8	12
I/O	0.84037	0.838304	0.786829	0.773862
Communcation	0.026037	0.316821	0.0576375	0.043194
Computation	26.07157033	7.113234667	4.515229333	3.343582667

Figure 5: Time profile of different numbers of thread (all #node is set to 3)



	1	4	8	12
speed up(computation)	1	3.665220052	5.774140893	7.797495361
speed up(total)	1	3.257959066	5.026027254	6.474481322

Figure 6: Speedup factor of different numbers of thread (all #node is set to 3)

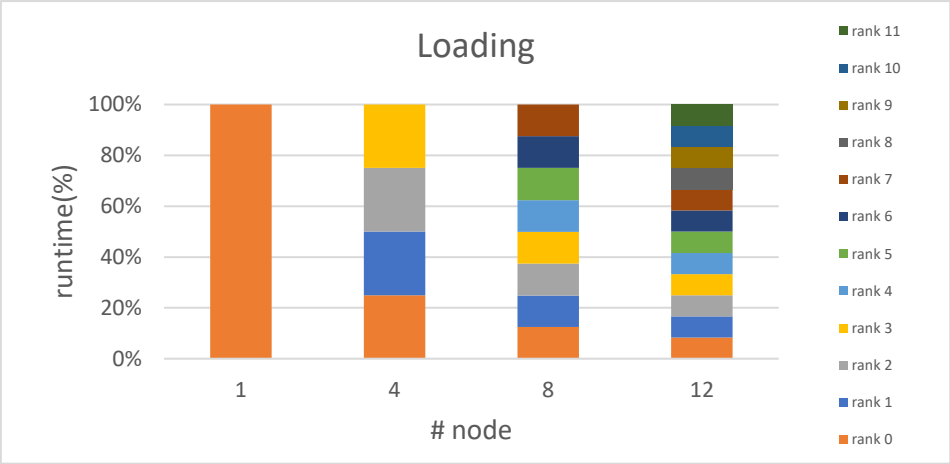


Figure 7: Loading of each node (all #thread is set to 3)

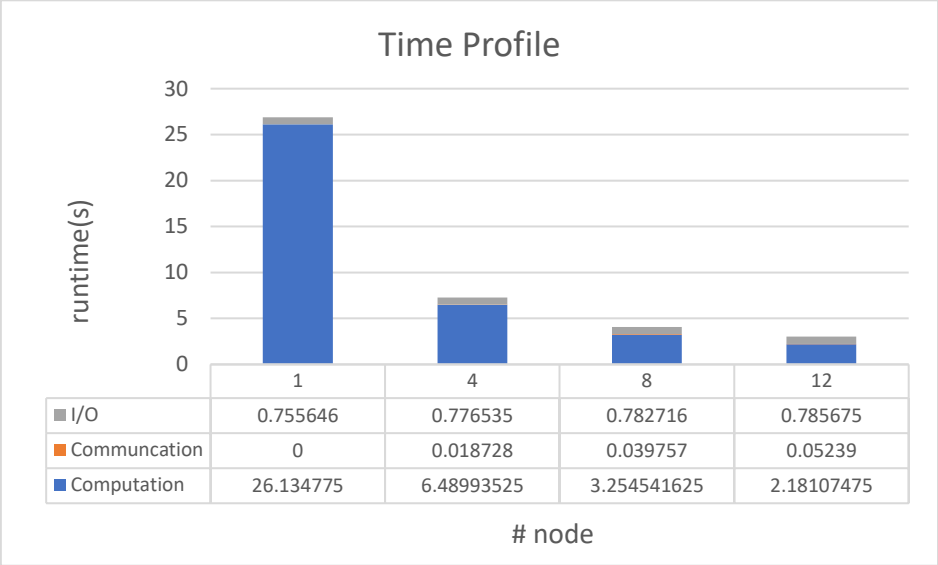


Figure 8: Time profile of different numbers of node (all #thread is set to 3)

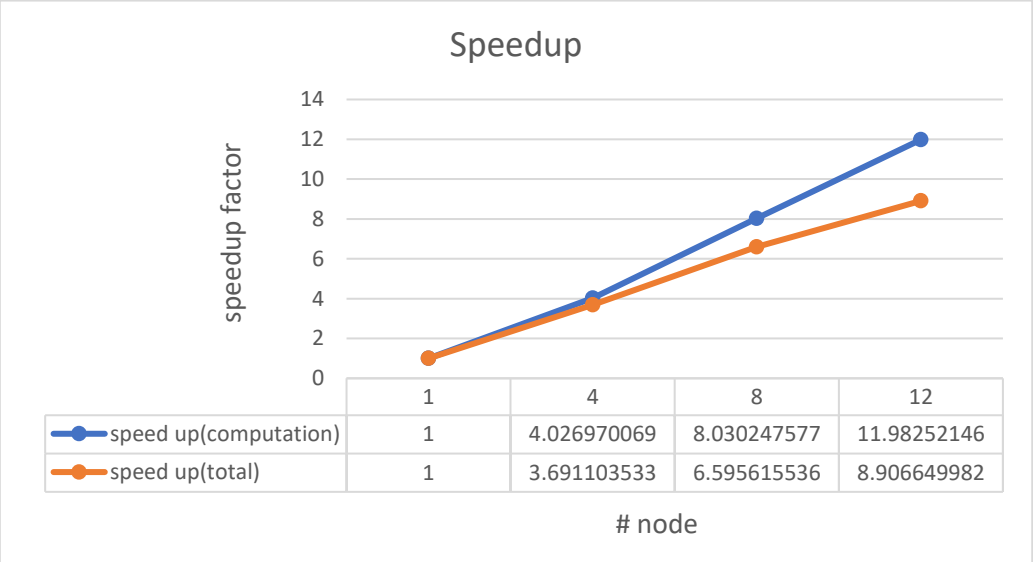


Figure 9: Speedup factor of different numbers of node (all #thread is set to 3)

✧ Discussion

Scalability

在 pthread 版本中，從 **Figure 3** 的 computation time 的 speedup 能夠隨著 thread 增加，達到理想的倍數成長，然而 **Figure 6** 的 computation time 無法因為 thread 增加而呈倍數成長。查了資料之後發現，因為 OpenMP 相較於 Pthread 還是比較 high level 的，並且 memory architecture 也會影響到 OpenMP 的 performance。

在 **Figure 3** 中，若考慮 pthread 的總花費時間，由於會受到 I/O 的影響，導致 scalability 稍差。

Reference: https://en.wikipedia.org/wiki/OpenMP#Pros_and_cons

Load balance

在兩個版本中，皆是利用 remain 的方式來分割計算工作，以此讓 Pthread 版本的 thread、hybrid 版本的 rank 的工作量均等，從 **Figure 1** 和 **Figure 7** 中能夠看出來。

■ Experience

這次的作業比起第一次來說，我認為 thread 的平行化比較好運用，因為 MPI 需要考慮 communication 的 overhead。然而 thread 的平行化，尤其是 OpenMP，可以讓 scheduler 自己去分配 (如 dynamic)，不須自己手動刻就能夠讓速度快很多，我覺得是很不錯的工具。