

# Parallel Programming Hw2

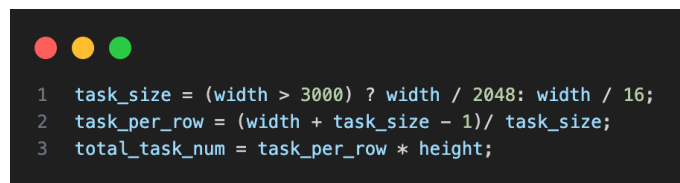
111062503 吳冠志

## A. Implementation Detail

本次作業預期透過 Pthread 以及 MPI、OpenMP 等工具實作 Mandelbort Set，我們無法對計算各 pixel 的數值做優化，因此本次實作將著重在如何將工作平均分配。

### 1. Pthread

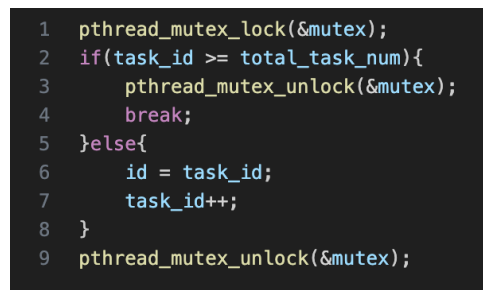
在讀取不同 task 的參數後，我們會計算其 task 大小 (task\_size)、task 在一個 row 的數量(task\_per\_row)以及全部 task 數量(total\_task\_num)供 thread 做計算



```
1 task_size = (width > 3000) ? width / 2048: width / 16;  
2 task_per_row = (width + task_size - 1) / task_size;  
3 total_task_num = task_per_row * height;
```

Figure 1. task 參數計算方式

接著我們會透過 mutex 去控管目前的 task，在一個 thread 完成自己的 task 後，他們可以去領取新的 task，且此過程被 mutex 控制住因此不會有兩個 thread 拿到相同 task 的情況發生，在 task 達到當初計算的 total\_task\_num 時即停止計算。



```
1 pthread_mutex_lock(&mutex);  
2 if(task_id >= total_task_num){  
3     pthread_mutex_unlock(&mutex);  
4     break;  
5 }else{  
6     id = task_id;  
7     task_id++;  
8 }  
9 pthread_mutex_unlock(&mutex);
```

Figure 2. 透過 mutex lock 索取下一個 task

本次實作透過切 row 來分配 task，width/task\_size 的 ceiling 為一個 row 的 task 數，除了最後一個 task 之外的 task 皆相等，而為了避免會有 task 跨不同的 row 造成計算上的困難，每 row 之最後一個 task 為此 row 剩餘的 pixel。下圖為 width = 8 且有 3 個 thread 之範例。

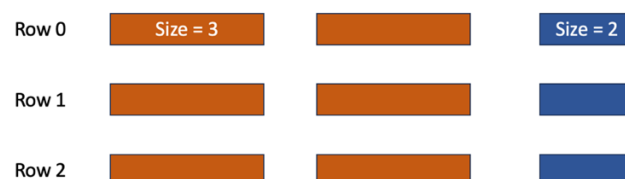


Figure 3. task 資料分法，每 row 最後一個 task 小於等於其他的 task

因此對每個 task 都可以計算此 task 位於哪一個 row、起始位置為何以及此 task 共需計算幾個 pixel。接著透過原 Mandelbort Set 計算各 pixel 並透過 SSE 加速。

## 2. MPI + OpenMP(Master-Slave)

Hybrid 版本透過 Master-Slave 進行實作，process 0 作為 Master Process 將 task 分配給其他 Slave Process，並在接收到計算結果後將其拷貝至 image 中，而 Slave Process 則在每次計算完此 task 後等 Master Process 傳送的新 task id 並繼續工作直到 Master Process 傳送完成訊號。本此切 task 的方式與 hw2a 相同，以不造成有 task 在不同 row 之前提去做任務的切分。

與 Pthread 版本不同的是，這次的 image 儲存全部有 Master Slave 處理，因此需要由 Master Thread 保持額外資訊，目的是在接收到 Slave Process 傳遞的資訊時可以盡快儲存，無需做額外的計算。(410)

## 3. MPI + OpenMP(Static)

如果工作可以分配的平均，讓各 processor 執行時間不會差太遠，那 static 表現效果應該比 Master-Slave 表現效果要好，因為發現 Mandelbort Set 鄰近 row 計算量相似的特性，本次實作透過 row 去切 task，row i 由 rank 為  $(i \% \text{size})$  負責執行，透過跳切的方式分配工作最後透過 MPI\_Gather 集合到 rank 0 並寫入 png。

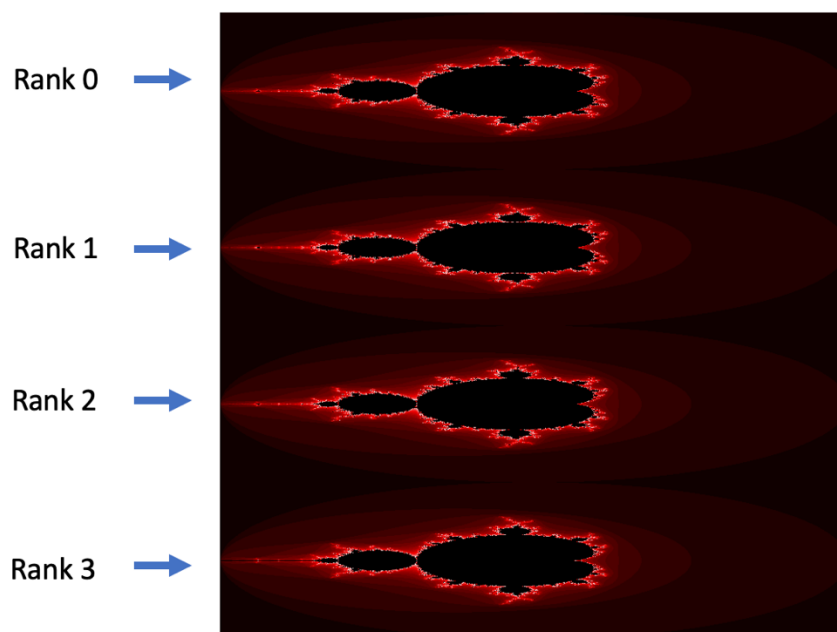


Figure4. 將各 process 計算的 pixel 依序輸出，可以發現不同 process 計算的圖形類似，推斷工作量差異不大

以結果來看 Static 效能比 Master-Slave 要好，因此實驗階段會以此版本做分析

## B. Experiment & Analysis

### 1. System Spec

本次作業使用課程提供的 **Apollo Server** 進行實驗，在不同實驗設置下進行實驗。本次欲進行比較之 **task** 為 **slow01.txt** 以及 **strict28.txt**，前者擁有大量的 **iteration** 次數，而後者擁有極高的 **width** 以及 **height**，透過在相對大的數據底下做實驗比較撰寫程式的 **Scalability** 以及 **Speedup** 等效能指標。

### 2. Performance Metrics

以下解釋各 **metrics** 的量測方法，本次透過 **clock\_gettime** with **CLOCK\_MONOTONIC** 計算各 **metrics** 時間。

#### a. CPU Time(Computing time)

計算 **Mandelbrot Set** 各 **Pixel** 之 **Repeats** 次數，**hw2a** 以及 **hw2b** 都需進行比較

#### b. Communication Time

**Hw2b** 中 **MPI\_Gather** 所花費的時間，因為與 **total time** 相比較小在本次作業忽略不計

這邊我並沒有計算寫入 **png** 的時間，因為本次作業主要針對有進行優化的程式進行效能比較，因為沒有對寫入 **png** 時間做優化因此不在此階段比較。

### 3. Plots: Speedup Factor & Profile

#### a. Speedup Factor

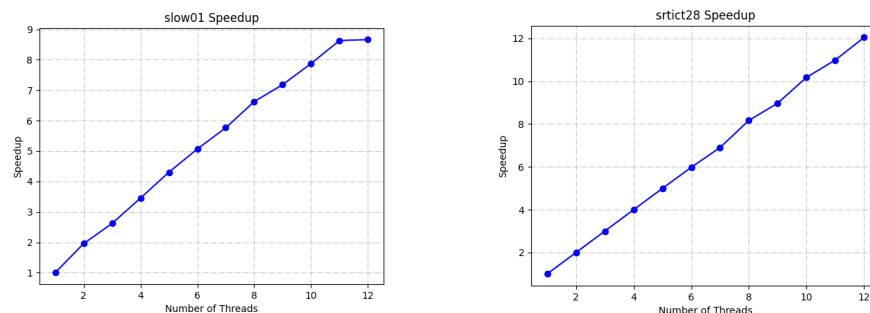


Figure 5. hw2a 對 slow01 以及 strict28 的 Speedup 表現

透過上圖可以觀察到，**hw2a** 透過 **pthread** 所得到的 **Speedup** 在不同 **case** 下都有幾乎接近 **ideal speedup** 的表現，因為 **pthread** 不需要 **Send/Recv**，因此造成 **Delay** 的原因可能是競爭 **mutex lock**，在本次作業中是提取下次要作業的 **task**，看起來在本次實驗中此種 **Delay** 影響不大，效能提升明顯。

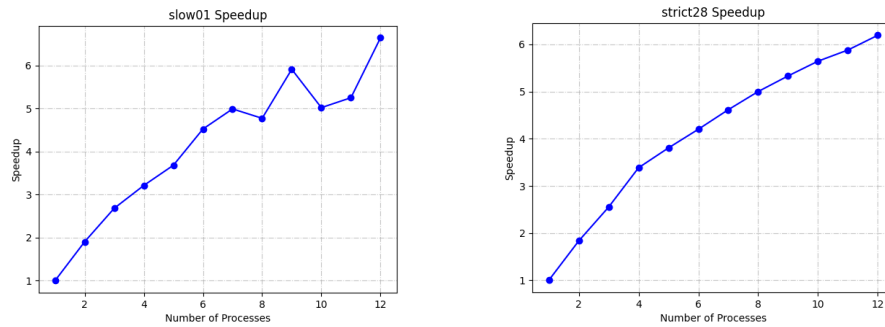


Figure 5. hw2a 對 slow01 以及 strict28 的 Speedup 表現

可看到與 hw2a 相比，雖然此 Speedup 仍接近 ideal Speedup，但其效能的提升相對不穩定，代表本次使用的工作分法可能在部分資料可能不能達到很好的平均工作分配，但確實還是帶來不錯的效能提升，且在工作期間 process 間不需溝通，省去許多 Master-Slave 所需要的額外負擔。

#### b. Load Balance

透過固定 process/thread 數量下去評估各 process/thread 工作時間是否相近

| Thread ID | Total Time |
|-----------|------------|
| 0         | 26.68      |
| 1         | 28.75      |
| 2         | 28.90      |
| 3         | 28.96      |

Hw2a應用於slow01各thread時間

| Thread ID | Total Time |
|-----------|------------|
| 0         | 9.93       |
| 1         | 9.93       |
| 2         | 9.93       |
| 3         | 9.94       |

Hw2a應用於strict28各thread時間

| Rank | Total Time    |
|------|---------------|
| 0    | 10.517(0.308) |
| 1    | 9.311         |
| 2    | 9.291         |
| 3    | 10.258        |

Hw2b應用於slow01各process時間

| Rank | Total Time   |
|------|--------------|
| 0    | 5.828(2.794) |
| 1    | 2.991        |
| 2    | 2.993        |
| 3    | 3.005        |

Hw2b應用於strict28各process時間

Figure 6. 不同 process/ rank 在不同資料下的總耗時時間

其中 2b 的括號為 Rank 0 將給定 array 寫入 png 所需時間，由上圖可看出 Hw2a 動態分配 task，做完的 thread 主動完成下一個 task，因此不會有 process 領先別人多個 task，Load Balance 表現良好。而 2b 因為需由 Rank 0 process 寫入 image，因此耗時較長，image 的 height 以及 width 較大者，Load Balance 效果較差，因為需花較多時間做寫入且此時間無法攤平到其他 process。

#### 4. Compare

本次可以調整的參數不多，hw2b 固定工作量讓不同 task 進行分工後 Gather，因此無法 tune 他的工作 size，hw2a 是 dynamic 的分工，完成一份工作後領取一份新的工作，因此工作量的大小是可以調整的參數。單份工作如果太大，可能無法達到 Load Balance，但如果太小，則可能造成多個 process 同時競爭 mutex 造成額外的效能損耗，因此合適的工作量相當重要。

#### a. Hw2a task size

本次實驗一樣針對 slow01 以及 strict28 進行實驗，task size 的設置分別實驗不同大小下不同 data 的計算效能

| Task Size | Total Time | Task Size | Total Time |
|-----------|------------|-----------|------------|
| 10        | 10.39      | 10        | 6.58       |
| 50        | 11.54      | 50        | 6.33       |
| 100       | 11.54      | 100       | 6.28       |
| 500       | 11.54      | 500       | 6.33       |
| 1000      | 11.54      | 1000      | 6.28       |
| 2000      | 11.54      | 2000      | 6.28       |

Slow 01

Strict 28

Figure 7. hw2a 中不同資料下不同 task size 的表現

我原本預期使用不同大小的 task，會很大的改變其工作分配的平均性，近一步影響程式效能，但看起來不同 task 大小對執行效能影響不大，且很難找到各 Data 適合的 task size，推測可能 task size 到一定大小就已經 load balance，再往下切效益不大。最後決定改回最初的設定以 Row 為單位分配 task，因為這樣可以不用花費額外成本計算 task 位於哪個 Row 以及此 task 的初始位置等資訊。

## 5. Discuss

#### a. Scalability

為了驗證本次方法是否有 scalability，我各找了四種類型 Data，分別為(少 pixel，小 iters)、(中 pixel、大 iters)、(大 pixel，小 iters)、(大 pixels、大 iters)，對應資料分別為 Fast06(483\*631, 5455)、Slow02(800 \* 800, 54564)、Strict19(6443\*6443, 10000)、Slow01(2549\*1439, 174170376)

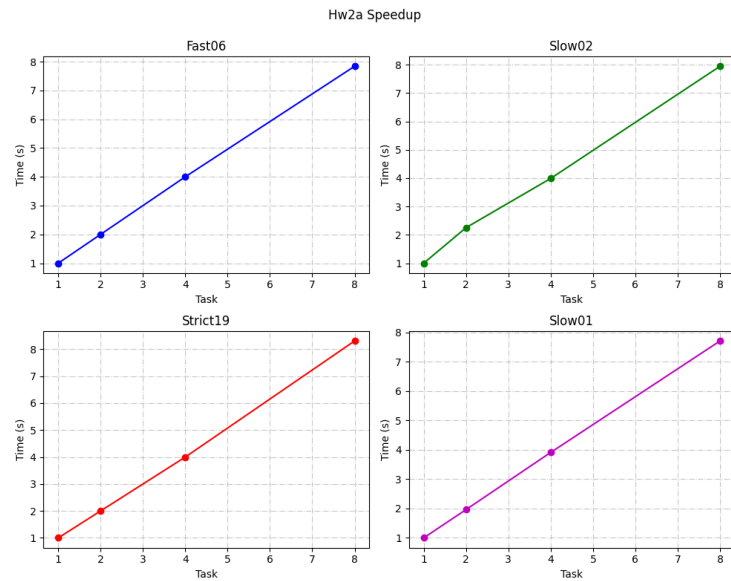


Figure 8. hw2a 下不同 scale 資料之 Speedup

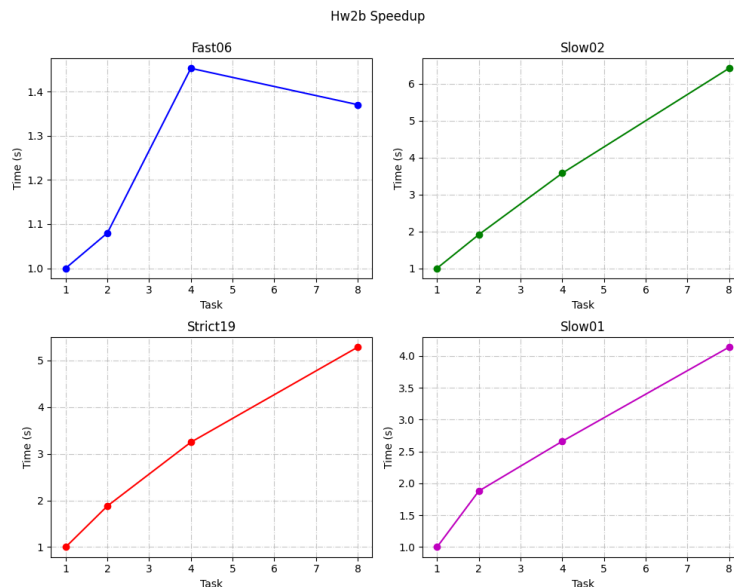


Figure 8. hw2b 下不同 scale 資料之 Speedup

從這兩張圖可以發現，hw2a 和 hw2b 在不同 scale 的資料下皆接近 ideal speedup(雖然 fast06 因資料太小不太穩定)，因此可以聲稱本次的實作具有好的 Scalability

#### b. Communication Load Balance

Total Time 的 Load Balance 有在前一個階段描述，在此階段想要比較 Communication time 的 Balance

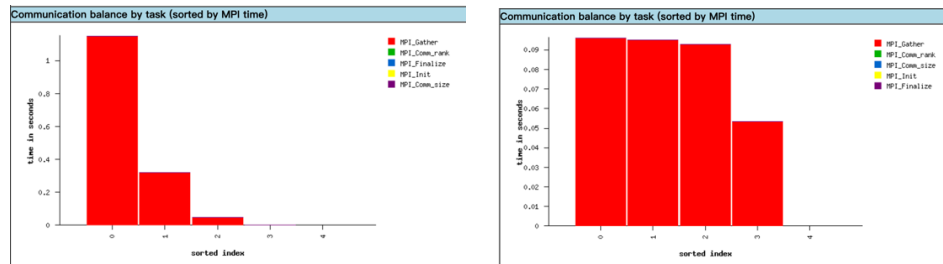


Figure 9. hw2b 下 slow01(左)以及 strict28(右)之 MPI 指令耗時分析  
左圖和右圖分別為 slow01(iter 多)以及 strict28(height & width 大)的 MPI 使用時間柱狀圖，我原先預期後者應該要花較多的時間在傳輸，因為其 pixel 數比較多，但結果與我預期相反，不但前者較耗時且不 Balance，後來推測因為 Gather 是 Blocking code，需要等大家都傳輸才能繼續往下做，因此原本工作不平均也可能影響到 Gather 時間，因為 strict28 iter 較小所以影響較少。因此在此方法下 rank 0 process 除了要做一般的計算，仍需要等其他 process 做 Gather 以及寫入 png，負擔較大，但在此問題下因為 Mandelbrot Set 計算量太大因此此部分造成的不 Balance 沒有對整體造成太大的影響，總體來說還是 Balance 的。

## 6. Experience & Conclusion

本次作業實作 Mandelbrot Set，我自己覺得能夠改進的地方沒有作業一來得多，比較著重在如何把工作分配的平均，雖然工具變多了

(Vectorization、Pthread)但實作起來或許大家都大同小異，實作了數個版本但效果都不是太好，有點小挫折。

但透過這次作業也學到了許多，SSE 程式的撰寫、正確的分配 memory 空間、透過 bit operation 讓程式加速等，還有一些想嘗試的方法(像是在 vectorization 階段做完一個 pixel 就立刻 load 下一個 pixel 不要等另一個 pixel 做完，以及在 hw2b rank0 可以透過部分 thread 做提早 gather 而不是讓大家都做完才 gather 增加 rank0 的負擔等等)，有一些想法不知道能不能成功，但迫於時間因素沒辦法在時間內完成，告訴自己下次要提早開始。