HW2: Mandelbrot Set

107062223 歐川銘

I. Implementation

我把 implementation 分成兩個部分來講，一個是用 Pthread 實作的部分，另一個是用 OpenMP 跟 MPI hybrid 實作的 Hybrid 部分。

Pthread

Variable & Basic Configuration

這次實作的 Mandelbrot Set 中，在 pthread 因為只有一個 process ，最後每個 thread 要把計算的結果寫回到一個叫做 image 全域變數裡面，一開始先開好 width * height 大小讓每個 thread 可以存取，等到最後計算完之後再拿這個 image 的資料去畫圖。

```
image = (int*)malloc(width * height * sizeof(int));
assert(image);
```

另外在讓每個 thread 開始做計算之前，我開了一個 struct 參數當作 input 給每個 thread 來存取需要的資料，我一開始會先把一些資料先計算好，像是每個點之間的 x, y 差距，這樣可以減少運算，讓每次在算 x0, y0 的時候不需要再去計算一次差距，同時這個 struct 也會帶有 lock 的資料，讓每個 thread 在存取要做哪一個 task 的時候不會出現 access 同個 task 的狀況，讓程式正常執行。

```
// thread pool, 給每個 thread 參數跟現在 row 做到哪
struct threadpool_t {
  pthread_mutex_t lock;
  int thread_count;
  int row_now;

  // data
  int iters;
  int height;
  int width;
  double left;
  double right;
  double upper;
  double lower;
  double x0_add;
  double y0_add;
  double x0;
  double y0;
};
```

確定好要傳遞的參數跟資料後，就可以根據 cpu 數量呼叫 pthread_create 讓 thread 可以開始運算 Mandelbrot Set，最後執行結束之後再呼叫 pthread_join 確保每個 thread 都已經完成計算。

```
for (int i = 0; i < ncpus; i++) {
  pthread_create(&threads[i], NULL, threadpool_thread, &pool);
}

for (int i = 0; i < ncpus; i++) {
  pthread_join(threads[i], NULL);
}
```

Divide Data

每個 thread 分配 task 的方式我原本是使用 Batch Size 來做切割，就是每個 thread 會從 threadpool 拿到現在要做的某一段長度，然後再根據這個長度去計算 x0, y0 位置，但是這樣的做法是每個點都需要重複計算現在的 x0, y0 以及現在是不是到了最後一個點，寫法上也必須嚴謹判斷，不然就會產生 runtime error。

```
while (!pool->shutdown) {
  pthread_mutex_lock(&(pool->lock));
  arg->run_height = pool->work_now / pool->width;
  arg->run_width = pool->work_now % pool->width;
  arg->now = pool->work_now;
  pool->work_now++;

  if (pool->work_now == pool->work_size) pool->shutdown = true;
  pthread_mutex_unlock(&(pool->lock));
  mandelbrot_set(arg);
}
```

所以我改成用 row 來做去分，讓每個 thread 每次的 task 就是一排 row ，透過這樣簡單的想法去做維護，不僅可以減少計算 x0, y0 的時間，在 pthread_lock 的操作減少，load balance 的實驗有呈現不錯的結果(下面有實驗圖)，因此保留此做法。

```
pthread_mutex_lock(&(pool->lock));
row_now = pool->row_now;
pool->row_now += 1;
pthread_mutex_unlock(&(pool->lock));
```

Compute the Image

依據助教投影片提示，在計算的時候可以用 vectorization __m128d 來做優化，但是因為每個點前後 iteration 都有 dependency ，所以 vectorization 應該是用來同時計算不同點的結果，但因為要考慮到每個點的 iteration 次數不同，所以在每計算完一次就要查看其中是否有某一個點已經做完，要替換成下一個點進行計算。

舉例： 假設有 5 個 pixel 要進行計算
剛開始 pixel1 跟 pixel2 進去到 __m128d

[1, 2]
當 pixel 1 做完的時候, pixel2 還沒有做完，pixel1 換成 pixel3 繼續計算
[3, 2]
過一段時間後, pixel2 做完了, pixel3 還沒，pixel2 換成 pixel4 繼續計算
[3, 4]
最後 pixel4 先做完，pixel3 還沒，pixel4 換成 pixel5
[3, 5]
繼續計算直到全部算完

藉由以上做法可以大幅增加運算效率，因為同時在計算兩個點，以下是實作部分

Load Date to Vector

後來我發現 vector 可以想成就是 vector 操作，看你要把 double 的資料放到哪個地方可以直接用 assign 的方式給資料，因此在操作上就比較直觀，除了在運算的時候還是會 call 提供的 API，但是其他時候就會比較直接用 assign 做資料的搬移。

```
__m128d two_see;
__m128d length_square_see;
__m128d x_see;
__m128d y_see;
__m128d x0_see;
__m128d y0_see;
__m128d x_square_see;
__m128d y_square_see;
```

```
// update parameter
y0 = pool->lower + row_now * pool->y0_add;
x0 = pool->left;
x0_see[0] = x0;
x0_see[1] = x0 + pool->x0_add;
x_see[0] = x_see[1] = 0;
y0_see[0] = y0_see[1] = y0;
y_see[0] = y_see[1] = 0;
x_square_see[0] = x_square_see[1] = 0;
y_square_see[0] = y_square_see[1] = 0;
```

Computation

在計算每個點的地方就依照原本 sequence 的公式做點優化改變，然後轉換成 vector 的運算 API，原本公式會導致 x * x 跟 y * y 多做一次，所以我這邊前後變換順序讓，每次計算時就會把 x * x, y * y 算好，可以讓 computation 時間減少。

```
// see instructions
y_see = _mm_add_pd(_mm_mul_pd(two_see, _mm_mul_pd(x_see, y_see)), y0_see);
x_see = _mm_add_pd(_mm_sub_pd(x_square_see, y_square_see), x0_see);
x_square_see = _mm_mul_pd(x_see, x_see);
y_square_see = _mm_mul_pd(y_see, y_see);
length_square_see = _mm_add_pd(x_square_see, y_square_see);
++repeats[0];
++repeats[1];
```

每次上面計算完一次就會到這邊檢查有沒有點已經做完，需要替換成下一個點繼續運算，另外在 done 這個變數是用來判斷，結束迴圈之後有沒有點還沒有運算結束的，因為如果只剩下一個點的話，不要用 vectorization 會更快一點，因為 vectorization 的 clock 會再長一點。

```
if (length_square_see[0] >= constraint || repeats[0] >= iters) {
  image[run_now[0]] = repeats[0];
  repeats[0] = 0;
  run_now[0] = start + record_now;
  x_see[0] = 0;
  y_see[0] = 0;
  x_square_see[0] = 0;
  y_square_see[0] = 0;
  length_square_see[0] = 0;
  x0_see[0] = x0 + pool->x0_add * record_now;
  done[0] = (record_now >= pool->width) ? true : false;
  record_now += 1;
} if (length_square_see[1] >= constraint || repeats[1] >= iters) {
  image[run_now[1]] = repeats[1];
  repeats[1] = 0;
  run_now[1] = start + record_now;
  x_see[1] = 0;
  y_see[1] = 0;
  x_square_see[1] = 0;
  y_square_see[1] = 0;
  length_square_see[1] = 0;
  x0_see[1] = x0 + pool->x0_add * record_now;
  done[1] = (record_now >= pool->width) ? true : false;
  record_now += 1;
}
```

判斷最後這個點做完了沒，還沒做完的話就換成 double 型態繼續做運算。

```
if (!done[0]) {
  x = x_see[0];
  y = y_see[0];
  x_square = x_square_see[0];
  y_square = y_square_see[0];
  length_square = length_square_see[0];
  x0 = x0_see[0];

  while (repeats[0] < iters && length_square < constraint) {
    y = 2 * x * y + y0;
    x = x_square - y_square + x0;
    x_square = x * x;
    y_square = y * y;
    length_square = x_square + y_square;
    ++repeats[0];
  }
  image[run_now[0]] = repeats[0];
```

最後等到每個 thread 把 pool->row 的值加到操過 height，就代表全部結束了，最後 process 會根據全部寫在 image 的資料再去畫出 image。

Hybrid

在 hybrid 的實作，需要用到 OpenMP 跟 MPI，那因為計算 image 用 vectorization 的方式跟 Pthread 一樣，這邊就針對 Hybrid 怎麼切 data，process 怎麼溝通資料，以及 omp 的使用來做說明。

Data Split in Process

首先是每個 Process 要處理的資料，這邊處理上考慮到 load balance 的問題，沒有讓每個 process 都固定切一個區間去做計算，而是每個 process 只會處理自己 rank + size * n 的 row，因為如果每個 process 都固定切一個區間去做計算的話，根據觀察的圖，很有可能某段 row 都是黑色的，造成 load balance 不平均的問題。

```
for (int i = rank; i < height; i+=size) {
```

Divide Data in Threads

為了有效計算結果，每個 process 都確定自己要計算的 row 之後，再來就是 thread 要如何去做分工達到好的 load balance，在這邊我呼叫 for schedule(dynamic) ，讓每個 thread 針對 process 要跑的資料動態去做分工，哪個 thread 做完就再去拿下一個 row 運算，希望以此達到更好的 load balance 。

```
#pragma omp parallel for schedule(dynamic) num_threads(ncpus)
  for (int i = rank; i < height; i+=size) {
```

Process Communication

因為每個 process 的資料沒有共享，只有 process 下面的 thread 才可以共享資料，所以這邊採取的方式是每個 process 先開好自己的 image 都是 height * width 大小的 array ，然後都初始化為 0，再讓每個 process 底下的 thread 寫到各自的 image 陣列裡面，最後再透過 MPI_Reduce 去把每個 image 的都相加到 rank0，讓因為每個點都不會被重複計算到，所以可以保證這樣結果 image 陣列在 rank0 是正確的。

```
if (rank == 0) {
  MPI_Reduce(MPI_IN_PLACE, image, width * height, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  write_png(filename, iters, width, height, image);
} else {
  MPI_Reduce(image, NULL, width * height, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

II. Experiment & Analysis

Methodology

System Spec

這裡採用的是課程提供的機器 Apollo 來做測試。

Performance Metrics

計算時間的方式是使用 MPI_Wtime() 來紀錄所有的時間，為了更方便的使用，我另外寫了一個 calculate_time.h 檔案來紀錄時間，這樣能更清楚的區分現在是在計算什麼時間。

```
// calculate io time
calculate_time->record_io();
write_png(filename, iters, width, height, image);
calculate_time->update_io();
```

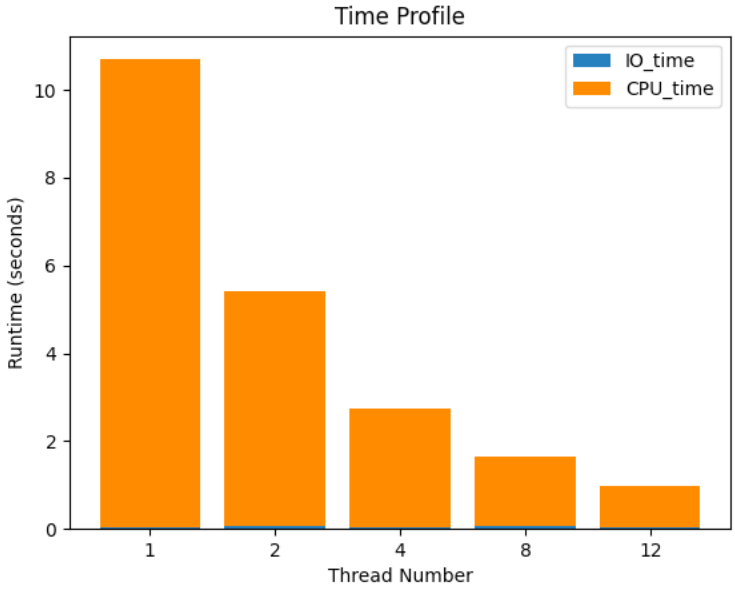
測資的部分是參考這次提供 testcase 裡面的 strict11.txt 裡面的 iteration 次數為 10000，在 1 thread 的時間為 10 秒左右，另外還有使用 strict33.txt 的測資最後來當作驗證，strict33 是屬於較大的測資，在 1 thread 的時間大概是 100 秒左右。

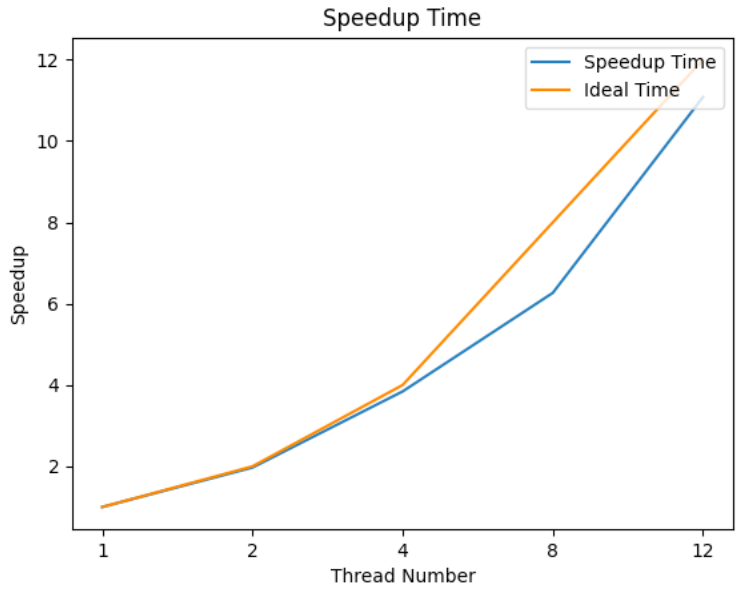
Scalability & Speedup

Pthread: Time Profile & Speedup Factor (Strong Scalability)

Strong Scalability

	1	2	4	8	12
CPU Time	10.6696	5.37377	2.71025	1.56306	0.927052
IO Time	0.0283912	0.0656345	0.028897	0.0606141	0.0490031





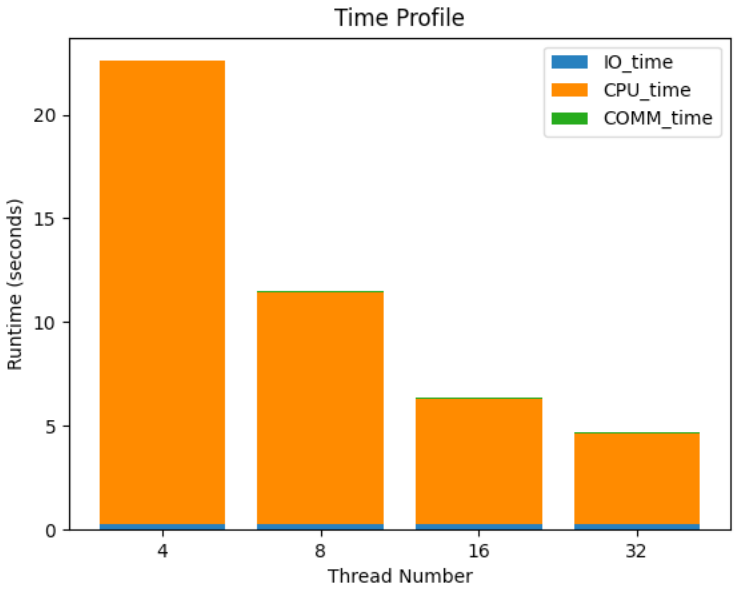
從 Pthread 結果數據來看，CPU time 幾乎佔據所有的時間，所以如果要優化的話，要從 CPU time 來做優化，從 Time Profile 這張圖可以得知，thread 數量越多，整體 CPU time 所需要的時間也明顯下降，並且從 Speedup 這張圖可以發現，實際上 Speedup 成長趨勢很接近完美的 Speedup 曲線，隨著 thread 數量增加，整體的 scalability 也有很好的進步，同時也說明 load balancing 也分配的不錯，每個 thread 的工作量很接近。

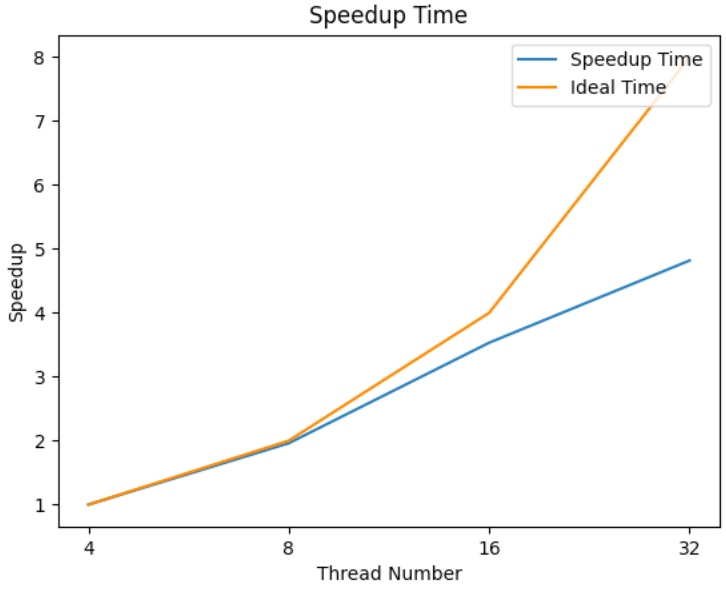
Hybrid: Time Profile & Speedup Factor (Strong Scalability)

預設都是開 4 個 process，並且設定每個 process 可以使用的 thread 數量，圖表上的 thread number 是全部 thread 的總數

Strong Scalability

Thread Number	4	8	16	32
CPU Time	22.3251	11.1769	6.05804	4.34553
COMM Time	0.0188683	0.0764282	0.0669639	0.0636431
IO Time	0.243986	0.268471	0.268891	0.27871





以上結果跟 pthread 結果其實差不多，CPU time 一樣是主要的優化項目，當整體 thread 數量越來越多的時候，speedup 也呈現很好的曲線上升，這代表 scalability 有隨著 thread 數量變多而變好的趨勢。

Load Balancing

在 Load Balancing 的實驗中，我去計算每個 thread 的個別花費時間，來驗證作法的分配效率，我總共計算了三種結果。

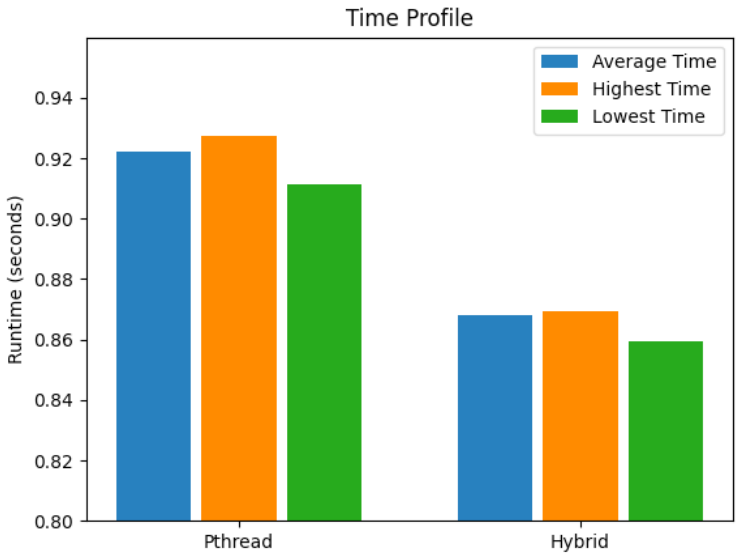
1. Average Time: 平均每個 thread 的計算時間。
2. Highest Time: 執行時間最長的 thread。
3. Lowest Time: 執行時間最短的 thread。

想要得到更好的 performance，應該要想辦法讓每個 thread 的 load balance 都差不多，這樣結束時間才會一樣，所以結果應該是要讓 Higest Time 以及 Lowest Time 跟 Average Time 越接近越好。

我是採用每次一個 thread 都直接拿一排 width 的點去做計算，這樣比較方便維護，因為只需要判斷現在 row 到哪個位置就可以當作終止條件，以及也減少許多計算，像是就不需要每個點都計算現在 y 的高度，用這個分配方式進行以下 Pthread 跟 Hybrid 的實驗， thread 數量都固定開 12，最後計算分配效率。

Load Balancing

	Pthread	Hybrid
Average Time	0.922019	0.868065
Highest Time	0.927443	0.869428
Lowest Time	0.91135	0.859344

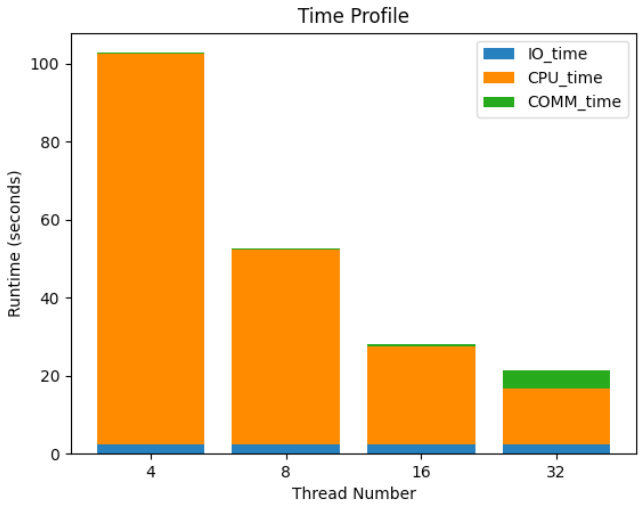


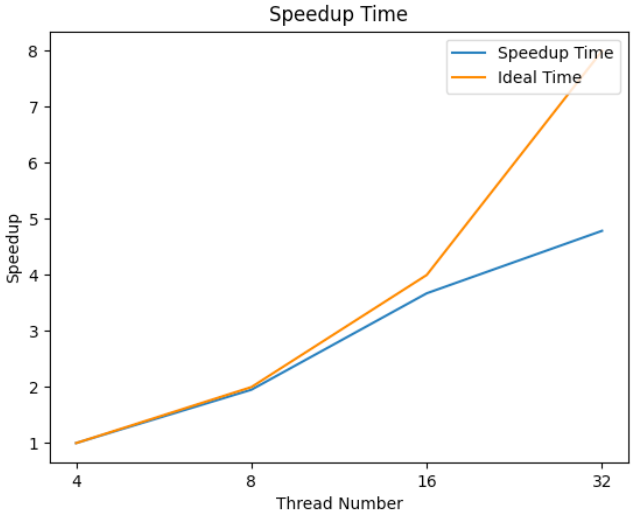
從實驗結果可以看到，不管是 Pthread 還是 Hybrid 的實作，highest time 跟 lowest time 差距都不會超過 0.02 秒, 雖然沒有達到每個 thread 的執行時間一樣，但是這個結果也顯示每個 thread 之間的工作量算是蠻平均的。

最後我有另外測試一筆在 strict33 的大測資執行在 hybrid 來檢查 speedup factor 跟 load balance 有沒有跟上面的實驗結果吻合。

Strong Scalability

Thread Number	4	8	16	32
CPU Time	100.251	50.1026	25.1338	14.4014
COMM Time	0.194065	0.181989	0.425768	4.70412
IO Time	2.361	2.36482	2.42848	2.36536





這個結果從圖表上面來看排除掉最後 32 個 thread 在 communication 時間之外，基本上 CPU_time 有隨著 thread number 數量變多而開始下降，這代表不管是在 scalability 或是 load balance 都有達到不錯的結果。

Other Experiment

除了 load balance 跟 speedup 之外，我在寫 code 的時候有針對 vectorization 做了一點小實驗，我發現直接對 __m128d 的陣列做 assign 比使用 _mm_set_pd 給值速度有些微差距，於是我寫了一個檔案就只針對賦值做測試

```
checkpoint = MPI_Wtime();
for (unsigned long long  i = 0; i < 100000000; i++) {
  a_sse = _mm_set_pd(2, 2);
}
x = MPI_Wtime() - checkpoint;

checkpoint = MPI_Wtime();
for (unsigned long long i = 0; i < 100000000; i++) {
  a_sse[0] = 2;
  a_sse[1] = 2;
}
y = MPI_Wtime() - checkpoint;

cout << endl << endl;
cout << "API cost time " << x << endl;
cout << "Assign cost time: " << y << endl;
cout << endl << endl;
```

結果如下

```
API cost time 2.73809e-07
Assign cost time: 1.67638e-08
```

時間差了超過 10 倍，這讓我之後的寫法都是直接 assign 來讓速度再變快些許。

III. Conclusion

在這次的實作中，一開始打完初始版本大概只有 700 秒的成績，當時看到跟助教的時間差距其實有點崩潰，因為不知道可能還要花多少時間再去做優化，後來看到投影片上面介紹的 vectorization 之後，才知道自己還有一大段路要走，做完 vectorization 之後為了更近一步的優化，前前後後嘗試了不少的 flag 以及把編譯器從 gcc 改成 clang 等等嘗試，最後終於從 700 秒的成績大概落在 374 附近，hw2b 成績最後也在 286 ，在 scoreboard 上面有著不錯的名次，雖然花費了不少的時間，卻也學到了不少東西，同時也知道優化其實還有更多的方向可以去做嘗試，讓我覺得以前對於這一方面的認知都太渺茫了，之後的作業也會繼續努力把優化跟時間做到更好。