

# Parallel Programming

## Homework 2: Mandelbrot Set

- 111065524 李懷-

## Implementation:

### Pthread version:

#### 1. Create pthread:

將各個 thread 所需的 argument 的變數設成 global 或包進 struct 並定義 function 後

pthread\_create:

```
55     double start, end, left, right, lower, upper;
56     int width, height, iters, num_threads;
57     int* image;

58

59     struct ThreadArgs {
60         int tid;
61         // start, end, left, right, lower, upper;
62         // int width, height, iters, num_threads;
63         // int* image;
64     };
65 }

66
67     void *compute(void *arg) {
68         struct ThreadArgs *data = (struct ThreadArgs *)arg; // cast back
69         int tid = data->tid;
70
71         for (int i = 0; i < num_threads; i++) {
72             args[i].tid = i;
73
74             pthread_create(&threads[i], NULL, compute, (void *)&args[i]);
75         }
76     }
77 }
```

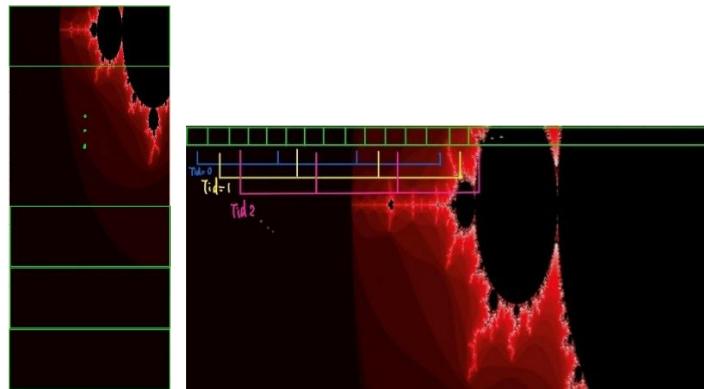
#### 2. Distributing data for each thread:

最一開始我是將 image 的 height 切分成# of threads 個去分配(如下圖 1)，每個 thread 處理一個區塊內的所有 pixels，但此方法會導致某些 thread 被分配到較多黑色區塊，代表需要算滿 iteration，導致較差的 load balance。

我後來試了以下兩種分配方法：

- 由於每個 pixel 間為 data independent，計算順序互不影響的，可將整個 image 的 resolution 攤開去平分，也就是從 image 左上到右下連續 pixels 依序去處理，每個 thread 則以# of thread 為 stride 逐一去處理 pixel(如下圖 2)，相較於最初的方法

thread 處理連續的 pixels，可能會導致某些 thread 處理到過多需要算滿 iteration(也就是在 set 裡的座標)的 pixels，導致 workload 不平均，因此此方法較每個 thread 處理連續區塊理應更能均勻的分配 workload。



舉例：假如 4 個 threads，第一個 thread 在整張 image 中負責處理的為第 0, 4, 8,..., 個的 pixels，再來便是得出 pixel 在複數平面的(x,y)座標，這裡 iteration 一次有兩個 pixels 是因為有做 vectorization：

```

58     int idx = tid;
59     for (; idx+1 < (long long)width * height; idx += num_threads*2) {
60
61         long long idx1 = idx;
62         int y1 = idx1 / width;
63         int x1 = idx1 % width;
64         double x0_1 = left + x1 * dx;
65         double y0_1 = lower + y1 * dy;
66
67         long long idx2 = idx + num_threads; // The next pixel for this thread
68         if (idx2 >= (long long)width * height) break; // Check bounds
69         int y2 = idx2 / width;
70         int x2 = idx2 % width;
71         double x0_2 = left + x2 * dx;
72         double y0_2 = lower + y2 * dy;
73
74     }
75
76 }
```

2. 即便第一種方法已經有不錯的 load balance，但仍會有某些 thread 先處理完的問題，所以另一種方法是將 pixels array 分成更小的 chunk，利用 locking 讓每個 thread 執行完一個小 chunk 後就進入到 critical section 去 grab 下一個 chunk 來處理(e.g. total pixels/(num\_of\_threads\*500)，不一次只處理一個 pixel 是因為 thread 會馬上計算完然後進入 critical section 去尋找下一個 pixel，但這會導致 thread 一直進入 critical section，導致整體平行化較差)，來達到更好的 thread 使用率及 load balance，我這邊是用 mutex 來實現：

```

58     int chunk_counter = 0;
59     int base_chunk, blocks;
60
61     pthread_mutex_t mutex;
62     int get_chunk(){
63         pthread_mutex_lock(&mutex);
64         int chunk = chunk_counter++;
65         pthread_mutex_unlock(&mutex);
66         return chunk;
67     }
68
69     while (true) {
70         int chunk_id = get_chunk();
71         int start = chunk_id * base_chunk;
72         int end = (start + base_chunk) < (width * height) ? (start + base_chunk) : (width * height);
73
74         if (chunk_id >= blocks) break;
75
76         // int idx = tid;
77         int idx = start;
78         for (; idx < end; idx += 2) {
79
80             // Process pixel at (x, y)
81
82         }
83
84     }
85
86 }
```

### 3. Vectorization:

由於我們有使用 sse2 的 instruction set 去做 vectorization，使用 `_m128d` data type 可以將兩個 double load 進 sse2 的 XMM register，一次處理兩個 pixels。

```
const __m128d two    = _mm_set1_pd(2.0);
const __m128d four   = _mm_set1_pd(4.0);
const __m128d one    = _mm_set1_pd(1.0);
```

並將原本 Mandelbrot set 的計算流程改為 intrinsic function:

```
__m128d y0v = _mm_set_pd(y0_2, y0_1); // reversed because first one is high 64 bits
__m128d x0v = _mm_set_pd(x0_2, x0_1);

__m128d x = _mm_setzero_pd();
__m128d y = _mm_setzero_pd();
__m128d repeats_pd = _mm_setzero_pd(); // doubles; convert back to int at the end
__m128d active_mask_pd = _mm_set1_pd(1.0);
for (int k = 0; k < iters; ++k) { // iterate up to iters since multiple lanes
    // x^2, y^2, xy
    __m128d xx = _mm_mul_pd(x, x);
    __m128d yy = _mm_mul_pd(y, y);
    __m128d xy = _mm_mul_pd(x, y);

    x = _mm_add_pd(_mm_sub_pd(xx, yy), x0v); // x^2 - y^2 + x0
    y = _mm_add_pd(_mm_add_pd(xy, xy), y0v); // 2*xy + y0

    __m128d len2 = _mm_add_pd(_mm_mul_pd(x, x), _mm_mul_pd(y, y));
    active_mask_pd = _mm_cmplt_pd(len2, four); // check if length_squared < 4

    // If both pixels diverge, break
    if (_mm_movemask_pd(active_mask_pd) == 0) {
        break;
    }

    // repeats += mask ? 1 : 0
    repeats_pd = _mm_add_pd(repeats_pd, _mm_and_pd(active_mask_pd, one)); // results
}
```

由於是一次處理兩個 pixels，這邊 `active_mask_pd` 會是兩個 pixels 的 mask，當兩個 pixel 都發散時 loop 會 break，`repeats_pd` 則是兩個 pixels 個別跑的 iteration 數，出 loop 後要再加一次 `active_mask_pd` 的 invert，因為若 pixel 的 mask 變 0 出 loop 後，會少加到 1 次 iteration，最後再將 double 轉回 int 並 load 回 memory，再寫到 global image 對應的 pixel 上：

```
repeats_pd = _mm_add_pd(repeats_pd, _mm_andnot_pd(active_mask_pd, _mm_set1_pd(1.0)));
__m128i repeats_i = _mm_cvtpd_epi32(repeats_pd); // convert back to int

// Store two pixels: note _mm_cvtpd_epi32 packs to low 2 ints of __m128i
int tmp[2];
_mm_storel_epi64((__m128i*)tmp, repeats_i);
image[idx1] = tmp[0];
image[idx2] = tmp[1];
```

## Omp+MPI version:

### 1. Distributing data

針對 hybrid 版，需要先將資料分配給 processes，每個 process 的資料再由 thread 去分配。

#### Data among MPI ranks:

我這邊試了兩種方法：

1. 單純將 image 分成連續 pixels 的 chunk 平均分配給 each MPI rank:

```
94 |     int rem = total_pixels % size;
95 |
96 |     int base_chunk = total_pixels / size, start = rank * base_chunk+(rank < rem ? rank : rem), end = start + base_chunk + (rank < rem ? 1 : 0), mpi_chunk_size = end - start;
97 |     // int thread_chunk = (base_chunk + (rank < rem ? 1 : 0))/NUM_THREADS, thread_rem = (base_chunk + (rank < rem ? 1 : 0))%NUM_THREADS;
98 |
99 |     int* rank_image = (int*)malloc(mpi_chunk_size * sizeof(int));
100 |     assert(rank_image);
```

2. 類似於上方 Pthread 版本 thread 間的分法，用 stride 去分，stride 設 rank 數，每個 rank 有個 indices array 去存有其負責處理的 image pixel 的 indices:

```
92 |     int* indices = (int*)malloc(mpi_chunk_size * sizeof(int));
93 |     // indices with stride: r, r+size, r+2*size, ...
94 |     for (size_t k = 0, g = (size_t)rank; k < mpi_chunk_size; ++k, g += (size_t)size)
95 |         indices[k] = g;
```

e.g. 總共 4 個 rank，rank0 處理(0,4,8,...)的 pixels

#### Data among threads:

Thread 之間資料的分配我試了以下：

1. 將每個 rank 分配到的 pixels 平均分配給 each thread，手動計算每個 thread 的在 global image 的 start 及 end index，再去計算座標：

```
101 | #pragma omp parallel for num_threads(NUM_THREADS) schedule(static)
102 |     for (int i = 0; i < NUM_THREADS; i++) {
103 |         // chunk each mpi rank with # threads
104 |         int thread_start = start+thread_chunk*i+(i < (int)thread_rem ? i : thread_rem),
105 |             thread_end = thread_start + thread_chunk + (i < (int)thread_rem ? 1 : 0);
106 |         for (int idx = thread_start; idx < thread_end; idx += 2) {
```

這分法是每個 thread 都會被平均分配到連續的區塊(process 分配不用 stride 的情況下)，然後一次 iteration 處理區塊內相連的 2 顆 pixels，在 process 沒有分散平分的情況下，此方法較 workload 較不平均，thread 會處理到連續的 pixels。

2. 用 static schedule 去分配 iterations 的 chunk，由於 static 是 threads 之間 round robin 的方式去分配 iterations，所以相較於 1 此方法較平均，但每個 thread 全部處理的 iterations 數平均，並不能很好地利用優先算完的 thread。
3. 利用 omp 的 dynamic schedule，並將 iterations 分得更細，我們這邊設 chunk\_size=total\_pixels/(num\_threads\*500)，處理完一個 chunk 的 iterations 後

`thread` 會接著繼續處理其他尚未被處理的 `chunk`(能著多勞的概念)，相較於(1)和(2)平均分配每個 `thread` 處理的 `iterations`，能將區塊分得更細，能更平均的分配 `thread` 的 `workload`:

```
int chunk_size = (mpi_chunk_size / NUM_THREADS)/500;
chunk_size = chunk_size < 10 ? 10 : chunk_size;
#pragma omp parallel for schedule(dynamic, chunk_size)
for (int i = 0; i < mpi_chunk_size; i += 2) {
```

\* 其他 `omp schedule` 的比較我放在下面 `discussion`。

最後上交的版本 `rank` 的分法是第二種跟 `thread` 的分法是第三種的，後續的 `implementation` 也是基於這分法。

再來從 `indices` 獲取 `pixel` 在 `image` 的 `index`，計算每個 `pixel` 的(x,y)座標並透過 `vectorization` 計算 `mandelbrot set`:

```
119     for (int i = 0, i < mpi_chunk_size, i += 2) {
120         int idx1 = indices[i];
121         // int idx1 = idx + start;
122         int y1 = idx1 / width;
123         int x1 = idx1 % width;
124         double x0_1 = left + x1 * dx;
125         double y0_1 = lower + y1 * dy;
126
127         // Pixel 2: (x2, y2)
128         int idx2;
129         if (i+1 >= mpi_chunk_size){
130             idx2 = 0; // Out of bounds
131         } else {
132             idx2 = indices[i+1]; // The next pixel for this thread
133         }
134         int y2 = idx2 / width;
135         int x2 = idx2 % width;
136         double x0_2 = left + x2 * dx;
137         double y0_2 = lower + y2 * dy;
```

## 2. Vectorization

Hybrid 的 `vectorization implementation` 跟 `pthread` 版的一樣，`code` 的部分請助教參照上方，但每個 `thread` 最後在寫入 `pixel` 值時是寫入在其 `MPI rank` 的 `rank_image array`:

```
191     int tmp[2];
192     _mm_storel_epi64((__m128i*)tmp, repeats_i);
193
194     rank_image[i] = tmp[0];
195     if (i+1 < mpi_chunk_size){
196         rank_image[i+1] = tmp[1];
197     }
```

由於 `rank_image` 是 `rank` 所有 `pixels` 的連續 `array`，最後在 `rank0` 合併起來時要考慮其對應到的 `image index` 及 `stride`，這裡使用 [MPI\\_Type\\_vector](#)，一種 `non-`

contiguous vector 的 datatype，可以設定 stride=rank 數，`MPI_Irecv` 會將 rank\_image 共 counts 個 ints 寫入到其在 image 對應的 pixel index 上，也就是寫在 image[r], image[r+stride], image[r+2\*stride],...，r 代表 receive 的資料來自哪個 rank，最後再由 rank0 輸出 png:

```
231 ˜     if (rank == 0) {  
232      image = (int*)malloc(width * height * sizeof(int));  
233      MPI_Request *reqs = (MPI_Request*) malloc(size * sizeof(*reqs));  
234      // receive each rank's data  
235      ˜     for (int r = 0; r < size; ++r) {  
236          int counts = base_chunk + (r < rem ? 1 : 0);  
237  
238          MPI_Datatype vec;  
239          MPI_Type_vector(counts, 1, size, MPI_INT, &vec); // stride=# of ranks  
240          MPI_Type_commit(&vec);  
241  
242          // Place pixels to corresponding indices in image  
243          MPI_Irecv(&image[r], 1, vec, r, 0, MPI_COMM_WORLD, &reqs[r]);  
244  
245          MPI_Type_free(&vec);  
246      }  
247  
248      ˜     if (mpi_chunk_size > 0)  
249          MPI_Send(rank_image, (int)mpi_chunk_size, MPI_INT, 0, 0, MPI_COMM_WORLD);  
250  
251          MPI_Waitall(size, reqs, MPI_STATUSES_IGNORE);  
252          free(reqs);  
253  
254      } else {  
255      ˜     if (mpi_chunk_size > 0)  
256          MPI_Send(rank_image, (int)mpi_chunk_size, MPI_INT, 0, 0, MPI_COMM_WORLD);  
257  }
```

### 3. Experiment & Analysis:

#### Methodology

##### (a) System Spec:

實驗皆使用課程提供的 apollo cluster，Compiler: GCC，MPI implementation:

OpenMPI

##### (b) Profiler & Metrics:

**Pthread** 程式我是直接用 `clock_gettime()` 去 log 每個 thread for loop 執行的時間來做 profiling (有試著使用 nsys 去，但只能 log 出 main thread 的 event，似乎沒辦法去看每個 worker thread 的細節)；

**Hybrid** 程式我是使用 nsys 去 profile mpi 及 omp 的 events，由於我用 complier 使用 gcc 時沒辦法抓到 omp event(似乎是跟 OMPT 有關)，所以 hybrid 版本實驗的部分 compiler 改為 icc。

#### Scalability 實驗:

我們計算了在不同 core 數下的 Speedup factor:

$$S_{\text{pthread}} = T_{\text{base}} / T_{\text{master\_thread}} ;$$

$$S_{\text{hybrid}} = T_{\text{base}} / T_{\text{rank0}} ;$$

$T_{\text{base}}$ : process 跟 core 數設 1 的 execution time；

$T_{\text{master\_thread}}$ : main thread 到 `pthread_join()` 執行完的 execution time；

$T_{\text{rank0}}$ : rank0 從 `MPI_Init()` 到 `MPI_Finalize()` 的 duration；

$CPU_{\text{rank0}}$ : 我們紀錄 rank0 的 cpu time，將總時間  $T_{\text{rank0}} - comm_{\text{rank0}} - *io_{\text{rank0}}$ ；

以上單位皆為 second(s)；

※ 由於 `write_png()` 幾乎沒有平行化的效益，可視為 constant across all scales，所以我們排除掉 `write_png()` 的執行時間，沒了 `fopen` 跟 `fwrite` 等後， $io_{\text{rank0}}$  可無視。

#### Load balance 實驗:

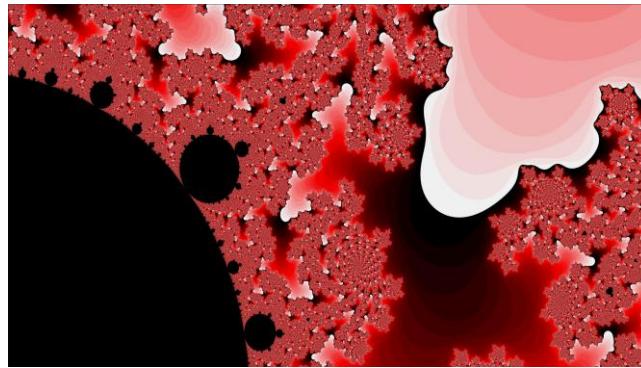
畫出每個 thread (or process) for loop 的 execution time 及透過 Nsight GUI 來看 parallel region 的 workload 是否有平均分配。

---

## Plots: Scalability & Load Balancing & Profile

- **Experimental Method:**

**Testcase:** 實驗我使用的 testcase 是 **strict35.txt**，從圖中可看到，此黑色區塊分佈較集於左下方，可以很好地看出我們 thread 間的 load balance 如何：



Args: 10000 -0.2931209325179713 -0.2741427339562606 -0.6337125743279389 -0.6429654881215695 7680 4320

### Parallel Configs:

- **Scalability Test:**

#### Pthread:

Node 跟 process 數固定為 1，core 數我們比較了 1~12 下的 speedup，設 core 數=thread 數，執行時間用 master thread。

#### Hybrid:

我們嘗試了幾種不同的  $n \times c$  組合來測試程式在總 core 使用數增加時的 speedup： 8、16、24、32、40、48(由於每個 user 在 cluster 上一次只能用 48 cores，所以我們 scalability 的 plot 最高就 cap 在  $n*c=48$ )。我們沒有 create 額外的 threads，因此每個 process 的 core 數即為其 thread 數。

- **Load balance Test:**

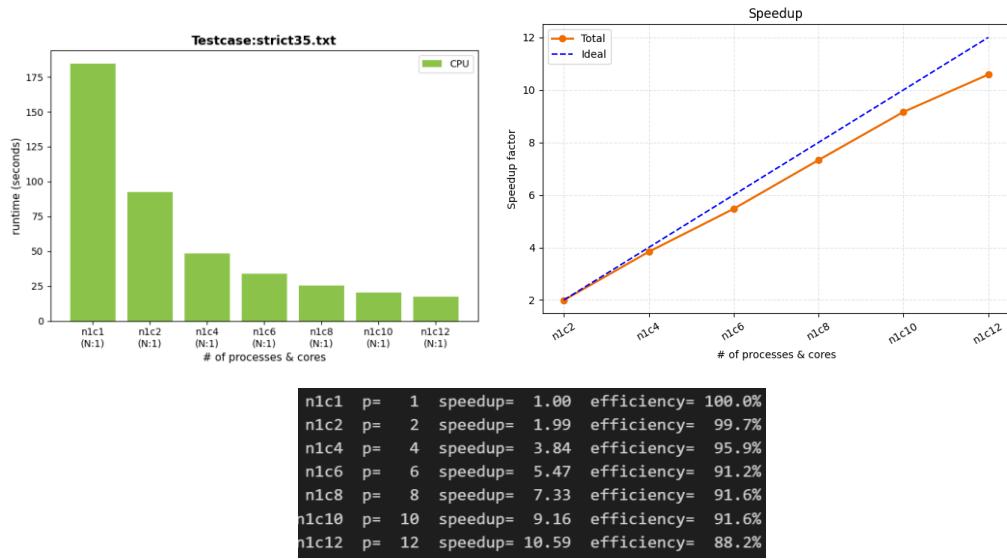
這部分則分開探討 process/thread 在不同的 workload 分配方法下的各 process/thread 的執行時間。

※Node 數我沒指定而是讓 slurm 去分配，因為每個 user 一次只能用 4 nodes，所以有些會用到>4 個 nodes 的 n 跟 c 的組合沒辦法跑，e.g. c6n8

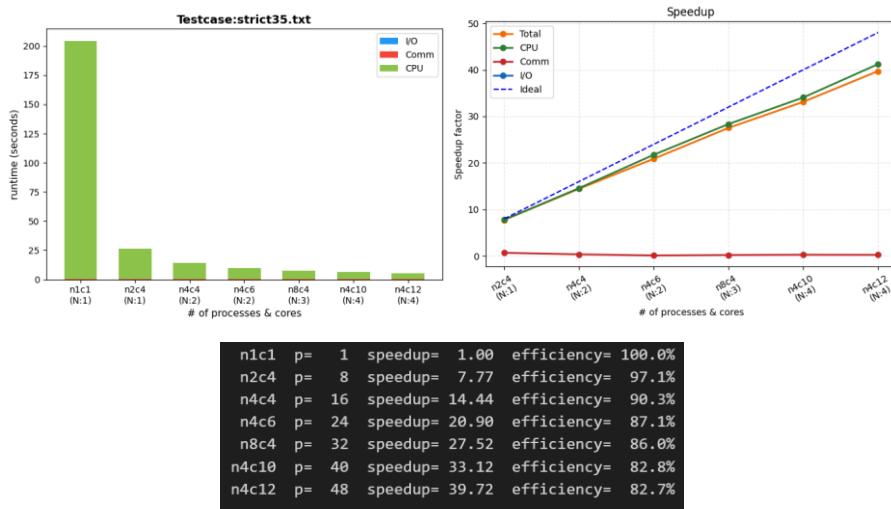
## ● Plots & Results:

### Scalability:

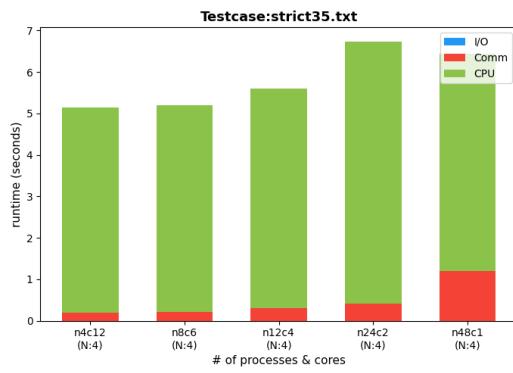
#### Pthread:



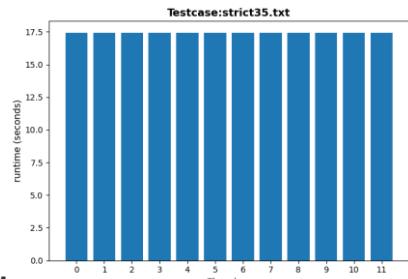
### Hybrid:



由上圖可以看到，兩個版本的 overall runtime 皆隨著整體使用的 processors 數上升而下降，speedup factor 幾乎是跟著  $n*c$  數 linear scale 的，代表無論是 process 還是 threads 間的 load balance 佳，Hybrid 版本雖有 MPI communication time 隨著 process 數上升，但不顯著，沒有像在 hw1 一樣成為 bottleneck，這也是因為在 Mandelbrot set 中，每個 rank 只有在最後算完 pixels 會把資料傳給 rank0，而沒有像 odd-even sort 一樣要在每個 phase 進行 rank pair 之間的溝通。



上圖為 hybrid 版本在同樣 threads 數的情況下不同的 process 跟 core 的組合，雖不明顯，但較小的 process 數會稍快一點，e.g. 同樣是使用 48 個 processors，n8c6 比 n12c4 稍快一些，主要是少掉一些 MPI 的 communication time 以及我們 thread 之間因為透過 dynamic schedule 的關係 workload 會分配地比 process 之間好。



**Load balance across threads(pthread):**

**Load balance across processes & across threads(hybrid):**

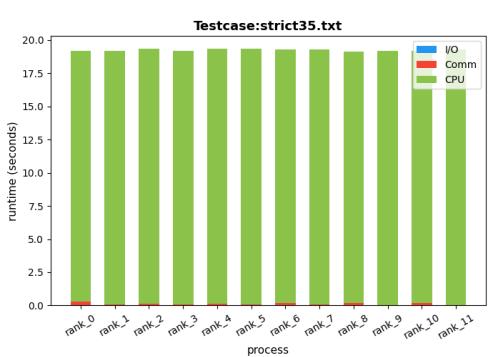


Figure 1: n12c1

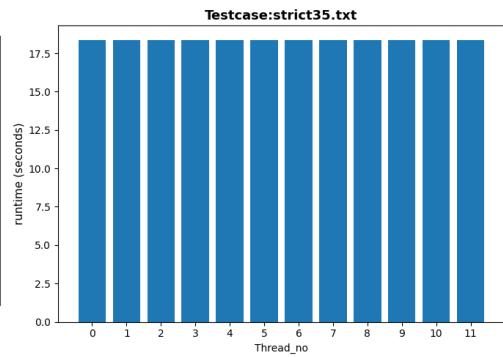


Figure 2: n1c12

---

- **Optimization Strategies:**

1. **Load balance** 的優化過程(performance 比較放在 discussion):

**Pthread :**

(1) 平均分配連續的 pixel chunk → (2) 考量到 Mandelbrot set 的特性，連續 pixels 的 chunk 可能導致 workload 分布不均，所以用 stride 去分配 → (3) 透過 locking 去更平均地分配每個 thread 的 workload；

**Hybrid:**

(1) processes 間平均分配連續的 pixels chunk → (2) 用 stride 去分配; threads 間用嘗試了不同的 omp schedule，將區塊分得更小，去更好利用每個 thread 以達到較好的 load balance。

2. **Other potential strategies:**

除了上述已實施的 rank 及 thread 間的 load balance 優化外，vectorization 的部分仍可以進一步優化，在 vectorization 計算兩個 pixels 時，目前的方式是採 2 個 pixels 同進同出，也就是說即便這對 pixels 中一個 pixel 已經發散算完了，但另一個 pixel 還未發散時，iteration loop 仍會持續進行，已算完的 pixel 會一直佔據著 simd register 一半的空間，直到二個 pixels 都發散 break 或 iteration 跑完，但理論上應該可以馬上把算完的 pixel 的空間釋出給下一顆 pixel 來達到更好的 SIMD 效率。

`write_png()` 目前是由 main thread 去執行，理論上這個 function 在 assign pixel 顏色的迴圈部分應該也是能由多個 thread 去分配執行，但考量到這邊只是 assign RGB 值而已，平行化的效益不大。

---

## Discussion

這裡我們用 implementation 提到的幾個分配 workload 的方法探討 scalability 及 process 跟 threads 間的 load balance，hybrid 的部分我透過 Nsight GUI 去觀察。

## Pthread:

### threads 間的 load balance:

我們每個 core 額外產出 4 個 pthreads 更好地看出差距:

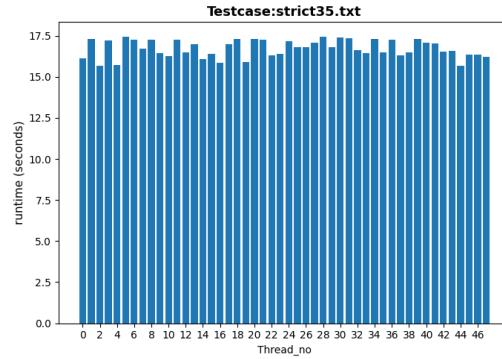


Figure 4: stride 分法

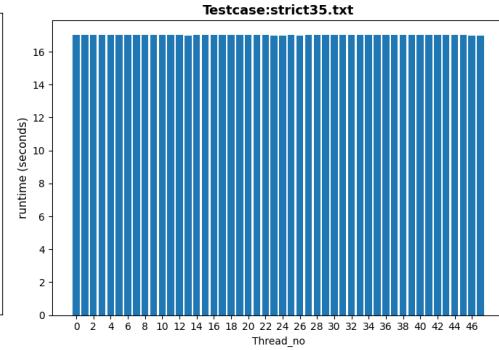


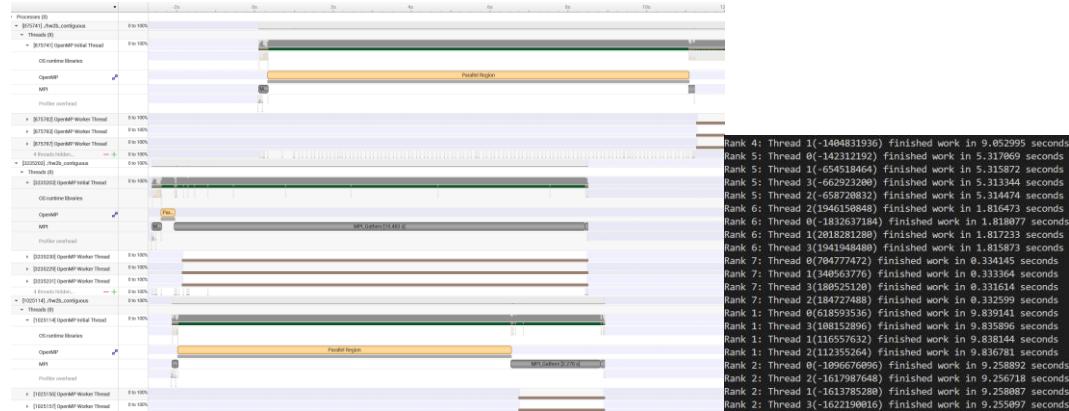
Figure 5: Mutex 配小 chunk

## Hybrid:

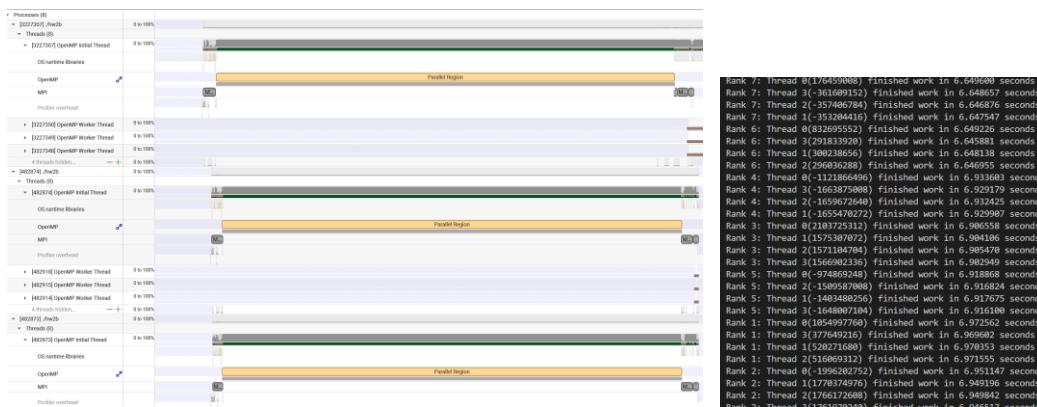
### processes 間的 load balance:

這裡我比較不同 mpi ranks 數下的 load balance , process 跟 core 數固定為-n8 -c4:

#### 1. Contiguous pixels 的 chunk:



#### 2. 每個 process 以 process 數為跨度去處理 pixels:

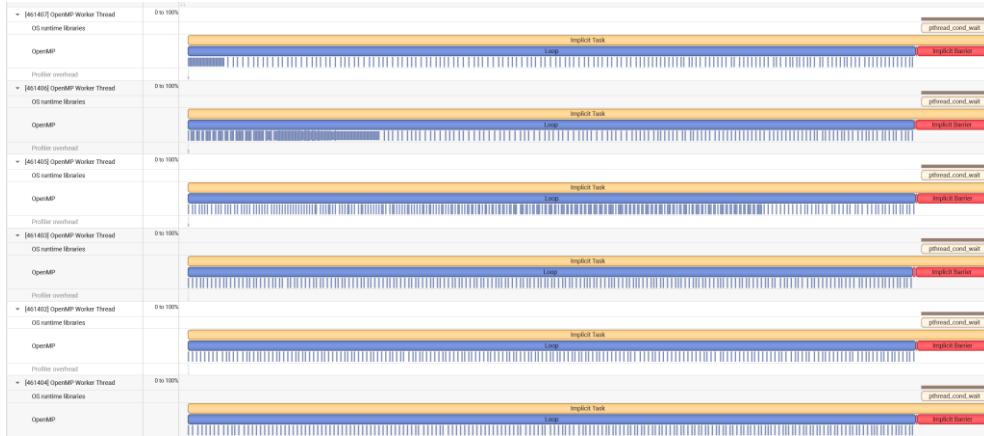


可看到(1)因為 Mandelbrot set 的特性，可能連續 pixels 都是在 set 內的，導致 processes 間的 load balance 很差；(2)每個 process 處理的 pixels 是散開的，比較不會有一次處理過多在 set 的 pixels，計算時間相近，但表充分利用每個 process，沒有 load balance 不好而造成某些 process idle 的情況。

### threads 間的 load balance:

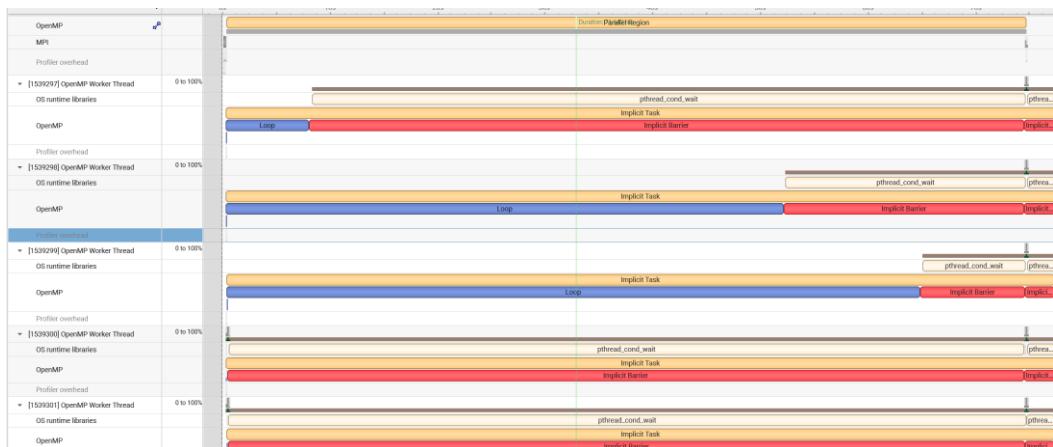
這裡我比較不同 omp schedule 下的 load balance，固定為-n1 -c8。

#### 1. dynamic schedule with chunk\_size=total\_pixels/(num\_threads\*500):



藍色 loop 為一個 thread 執行計算 set 的總時長，下面細小的藍色區塊代表一個 iterations chunk，chunks 間越緊密代表計算時間短；chunk 後的空格越大就代表那個 chunk 計算較長，可能含有較多黑色沒發散的 pixels。可以看到將區塊切小並用 dynamic schedule 後，各 thread 的計算總時長相當，代表沒有 thread 算完就 idle 在那邊造成 bottleneck 的情況，thread 間的 load balance 佳。

#### 2. dynamic schedule with chunk\_size=total\_pixels/(num\_threads) (等同於每個 thread 處理一個連續大 chunk，相當於 implementation thread 中的第一種分法):



可以看到有些 thread 的藍色 loop 條很短，代表很快就處理完了，但有些 thread 却執行很長的時間，此方法 load balance 差，導致整個 program scale 的不好。

此方法在 process 為 1 的情況下 load balance 特別差，但如果有多個 processes 搭配 stride 的分法，即便 thread 一次處理一個大 chunk，其處理的 pixels 已是分散非連續的，load balance 不會太糟。

### 3. static schedule with chunk\_size=total\_pixels/(num\_threads\*500):



由於是 round robin 的方式去處理 chunk，每個 thread 處理的 iteration 數平均，load balance 沒有像 dynamic 那樣好(這裡 GUI 顯示每個 thread 只有處理一個 chunk，是因為雖然我們把 loop 切成許多小塊，但跟 dynamic 不同的是 static 每個 thread 分配到的 iterations 是固定的且連續執行的，沒有像 dynamic 那樣動態分配)。

### 4. static schedule with chunk\_size = 1:



由於 chunk size 為 1 且為 round-robin，其實就是等同於 stride 的分法，load balance 好，但在 processes 已經採此分法的情況下，效益可能沒有 dynamic 搭配小 chunk\_size 好。

## 5. guided schedule:



由於 guided 類似 dynamic，只是一開始 chunk size 較大再來漸漸 decay，可透過 GUI 看到 chunks 越來越密集(表示越來越快，代表 size 越來越小)，load balance 好，但有可能因為某些 threads 在 chunk\_size 還很大時被分配到 workload 較大的 chunk，導致 load balance 仍有點不平均，略輸 dynamic。

**Conclusion:** 用 static schedule 的話 chunk\_size 對於 load balance 的影響不特別大； dynamic 雖然 load balance 較好，但可能會因為 chunk\_size 設太大導致 load balance 差 (如(2))或太小導致各 thread 一直進 critical section； guided 則是不用考慮 chunk\_size 就可以達到接近 dynamic 的效果。

Scalability of each schedule(CPU time only):

---

## 4. Experience & Conclusion

透過這次 Mandelbrot set 的作業，我深刻理解了在 multiple-processes 及 multi-threads 程式中 load balance 為影響 scalability 好壞的關鍵因素，這是在 HW1 的 Odd-even sort 沒體會到的，其中為了達到好的 load balance 而嘗試了不同的 workload 分配方式，像是 pthread 中實作了 locking 機制來增加 thread 的使用率、Hybrid 中嘗試了不同的 OMP 寫法及 schedule，並透過實驗驗證了不同方法下的 load balance 差異，由於 threads 間是 shared memory，導致常遇到 race condition 需要 debug，其他還有學到透過 sse2 的 instruction 及 SIMD register 來實作 vectorization，又因其 debugging 較為困難，花了一些時間，另外還有一些遺憾的是最後程式的效率還是跟 scoreboard 上前幾名的同學有顯著的差異，代表程式仍有很大的優化空間。

實作說明
pthread: 解釋如何分配資料
hybrid: 解釋如何分配資料
hybrid: 解釋如何選擇 omp schedule 的方式
說明 Manderolt set 計算量不均的問題
解釋如何處理計算量不均
解釋 vectorization 實作方法
是否有對程式進行優化
解釋具體的優化方式與過程

實驗方法
說明機器規格
說明使用的 MPI / Compiler
說明使用的工具 / Profiler
說明時間的測量方法
說明 CPU, IO, Comm time 的測量範圍

實驗數據與分析
有 CPU, IO, Comm time 的 breakdown
有做 scalability 的實驗
合理解釋 Scalability 結果背後的原因
有做 thread 和 process 分配方法的效能分析
有討論 Load balance
有獨立出 CPU time 的 scalability
根據實驗結果得出合理結論，並與實驗圖表中的結果相符
有活用 Profiler 的數據來討論
有正確透過數據推斷 bottleneck
根據分析結果，實作/提出潛在的優化方式
有優化前後效能的比較