Parallel Programming Homework Report

Title: Homework 2: Mandelbrot Set (Pthread + OpenMP)
Name: 莊岱倫
Student ID: 114062616

Implementation Details

hw2a 與 hw2b 在整體結構上採用相同的設計理念，差異主要在於資料分割與工作分配策略。 以下分別說明其運行流程與資料切分方式。

運行流程：

1. 將影像切成獨立的子任務放入task queue
2. task queue動態分配工作「領取下一個子任務 → 計算 → 寫回」
3. 在子任務內用SIMD加速子任務的處理速度
4. 結果圖片按照row major寫出

資料分割方法：

hw2a

Task queue 創建一個共享的task queue並且用atomic操作保護使Process不能同時更改操作，分割時一列就是一個任務

```
typedef struct {
    RowTask* tasks;
    int ntasks;
    std::atomic<int> next;  // 改用原子變數
} TaskQueue;
typedef struct {
    int j_global; // 全域列索引
} RowTask;
```

每個執行緒一次領取一個 batch 的任務執行完後再回到 queue 搶下一批。

```
static void* worker_main(void* arg) {
    ThreadArg* ta = (ThreadArg*)arg;
    Context* ctx = ta->ctx;
    maybe_pin_this_thread(ta->tid, ctx->ncpus, ctx->pin_threads);

    int batch_size = ctx->batch_size;
    for (;;) {
        int count = 0;
        int start_idx = claim_next_task_batch(&ctx->q, batch_size,
&count);
```

```
            if (count == 0) break;

            // 處理這一批任務
            for (int i = 0; i < count; ++i) {
                int idx = start_idx + i;
                int j_global = ctx->q.tasks[idx].j_global;
                compute_row(ctx, j_global);
            }
        }
        return NULL;
    }
```

hw2b

資料分割部分分為2層

1. MPI
2. OpenMP

MPI層– Cyclic Row Distribution **切分資料** 透過 MPI 的 cyclic列分配把資料「打散」到各節點，每個 rank 處理 `j = rank, rank+size, rank+2*size, ...` 假設你有 12 筆資料（index 0～11），要分給 3 個 process（rank 0、1、2）：

ProcessID	DataID
P0	0,3,6,9
P1	1,4,7,10
P2	2,5,8,11

```
j_global = rank + j_local * size;
```

回收資料 因為 cyclic 讓各 rank 的列是「交錯」的MPI_Type_create_hvector直接把接收到的資料放回最終影像的正確行列

```
MPI_Datatype row_t;                     // 一整列 = width 個 int
MPI_Type_contiguous(width, MPI_INT, &row_t);
MPI_Type_commit(&row_t);

// 對於來源 rank r，交錯列之間的跨距（bytes）
MPI_Aint stride_bytes = (MPI_Aint)size * (MPI_Aint)width * sizeof(int);

// rows_r = 該 rank 實際擁有的列數（按 r, r+size, ... 數出來）
MPI_Datatype vec_t;
MPI_Type_create_hvector(rows_r, 1, stride_bytes, row_t, &vec_t);
MPI_Type_commit(&vec_t);

// 將 rank r 的交錯列直接 Irecv 到 final_image 正確起點 base_ptr = final_image
```

```
+ r*width
MPI_Irecv(base_ptr, 1, vec_t, r, 0, MPI_COMM_WORLD, &req);
```

OpenMP層– Dynamic Scheduling

```
#pragma omp parallel for schedule(dynamic, 32)
    for (int j_local = 0; j_local < my_rows; ++j_local) {
        int j_global = (!use_cyclic) ? (start_row + j_local) : (rank +
j_local * size);
        double y0 = (double)j_global * dyj + lower;
        __m128d ys = _mm_set1_pd(y0);

        int* rowp = image_part + (size_t)j_local * (size_t)width;

        int i = 0;
        for (; i + 5 < width; i += 6) {
```

每個thread會拿到myrow裡的32 rows，一但該thread做完就動態分配另外32 rows給他

technique to reduce execution time and increase scalability

1. reduce execution time（SIMD） 以 __m128d 同時用同一條指令計算兩個像素，這樣可以使計算的指令數少將近一半

   ```
   static inline __m128d iter_cal(__m128d &x, __m128d &y, __m128d &xx,
   __m128d         &yy, const __m128d x0, const __m128d y0) {
       __m128d xy = x * y;
       x = xx - yy + x0;
       y = C2 * xy + y0;
       xx = x * x;
       yy = y * y;
       return xx + yy;
   }
   ```

   如果要處理的像素量是奇數就用scalr去計算完

   ```
   static inline double iter_cal_scalar(double *x, double *y, double *xx,
   double *yy,
                                        const double x0, const double y0) {
       double xy = (*x) * (*y);
       *x = (*xx) - (*yy) + x0;
       *y = 2.0 * xy + y0;
       *xx = (*x) * (*x);
       *yy = (*y) * (*y);
       return (*xx) + (*yy);
   }
   ```

2. increase scalability（切分策略） 本次的任務不同資料之間的運算複雜度是不同的，所以直接把連續資料均分給不同process 或是 thread很顯然是會有load balance的問題，所以我在兩個版本得程式都實作了打亂資料分布的策略，去盡量讓每個子任務複雜度相同。

   hw2a 因為沒有process之間load balance的問題，直接讓thread到task queue去搶資料，這樣小批次動態的隨機性也讓整體的scalability提升，因為幾乎沒有thread會idle很久沒在做事

   hw2b 在這個版本裡因為要先將資料分給每個rank所以我打算MPI OpenMP兩層策略不同，MPI層有可能跨節點通訊所以最好一開始過發好資料，而不是動態搶資料，所以用cyclic的方式打亂資料，然後每個rank再透過local的記憶體去實作動態task queue，讓整體從上到下的load balance都有達到平衡以達到不錯的scalability

3. Experiment & Analysis

i. Methodology

Measure matrix

利用nsight中的nvtxRangePush(""); nvtxRangePop();

- **computing time**

  除了 **communication time, IO time之外都算作computing time**

- **communication time**

```
nvtxRangePop(); nvtxRangePush("Comm");
MPI_Gatherv(
        image_part, local_n, MPI_INT,
        image_full, recvcounts, displs, MPI_INT,
        0, MPI_COMM_WORLD
);
nvtxRangePop(); nvtxRangePush("CPU");
```

- **IO time**

```
nvtxRangePop(); nvtxRangePush("IO");
write_png(filename, iters, width, height, image_full);
nvtxRangePop(); nvtxRangePush("CPU");
```

Execution time

利用程式碼整體運行時間去當標準 hw2a-judge 247.78 hw2b-judge 184.96

Test Case Description

testcase/slow01

項目	內容
Application	Mandelbrot image generator
x 範圍 [x_0 , x_1]	[0.7894722222222222 , 0.7825277777777778]
y 範圍 [y_0 , y_1]	[0.145046875 , 0.148953125]
Max Iteration	174170376
Width	2549
Height	1439

testcase/strict34

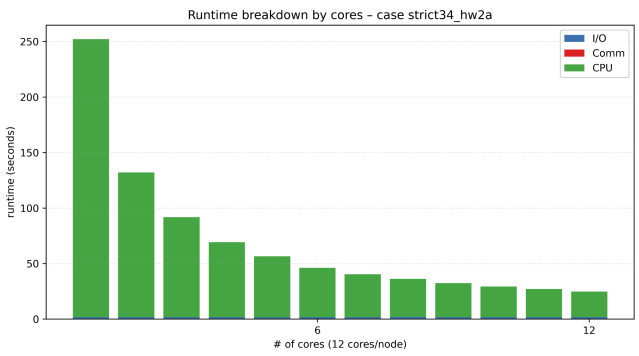
項目	內容
Application	Mandelbrot image generator
x 範圍 [x_0 , x_1]	[-0.5506164691618783,-0.5506164628264113]
y 範圍 [y_0 , y_1]	[0.6273445437118131,0.6273445403522527]
Max Iteration	10000
Width	7680
Height	4320

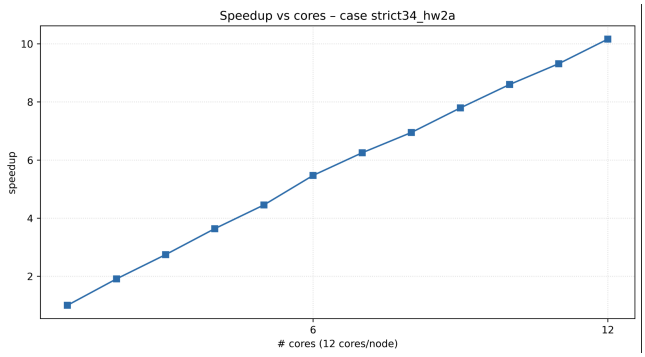
Parallel config

- hw2a - Node : 1 - Process num : 1 , 2 , 3 , … , 12
- hw2b - Node : 每12個多一個nade - Process num : 1 , 2 , 3 ,… . , 48

Analysis of result

hw2a-judge 184.96





- Speedup vs. Cores

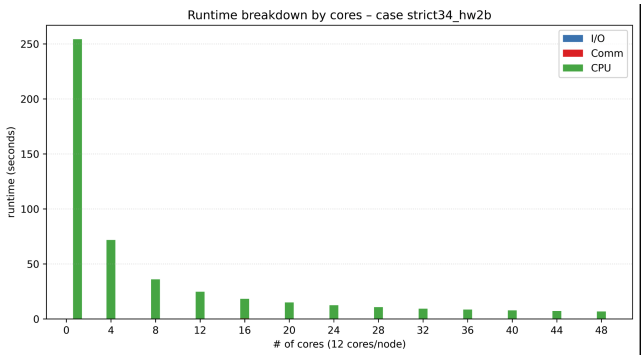
  此圖顯示隨著核心數增加，程式的加速比幾乎呈現線性成長，在 12 核時達到約 10 倍的效能提升，展現出極高的平行化效率。這代表程式中可平行化的部分占比極大，序列化區段與同步開銷極低，說明工作分

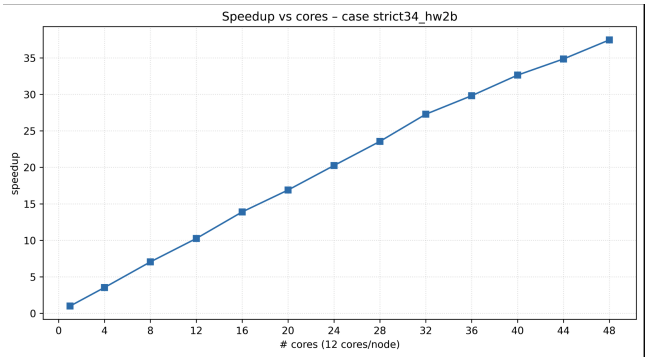
配均勻且執行緒間幾乎沒有等待或競爭，整體 scalability 表現優異。

- Runtime Breakdown by Cores

  從此圖可觀察到執行時間主要由 CPU 計算主導，而 I/O 與通訊（Comm）所佔比例極小，顯示系統資源集中於運算工作，通訊與輸出成本可忽略不計。隨著核心數增加，CPU 執行時間顯著下降，反映平行化能有效分攤計算負載。整體而言，程式運行效率高、開銷極低，是一個計算密集型且平衡良好的多執行緒實作。

hw2b-judge 184.96





- Runtime Breakdown by Cores

  此圖顯示整體執行時間隨著核心數的增加大幅下降，主要運算時間集中在 CPU 部分，而 I/O 與通訊（Comm）幾乎可忽略。這意味著程式屬於明顯的 計算密集型工作（compute-bound），平行化後幾乎沒有顯著的通訊瓶頸。整體趨勢呈現穩定下降，顯示任務分配均勻、排程開銷極低，隨著核心數提升可有效降低每個執行緒的計算負載。

- Speedup vs Cores

  從加速比曲線可見，speedup 隨核心數呈現近乎線性成長，在 48 核時達到約 37 倍的加速，效率約為 77%。此結果展現出優異的 scalability，代表平行化設計充分發揮硬體效能，且幾乎未受到同步、通訊或工作不平衡的限制。這樣的結果通常歸功於良好的 動態排程策略 與 資料切分（cyclic 分配），使所有核心能長時間保持高效運作。

Load balance 這個結果是由不同時間系統在不同附載度下的平均結果，可以看到資料切分的是非常平均所以不論是運行速度還是擴展性都相當不錯尤其在切分策略的選擇真的大大影響load balance的程度

Optimization Strategies

當前speedup還是略低於理想值，代表還是有進步的空間： **Memory Bandwidth / Cache locality**

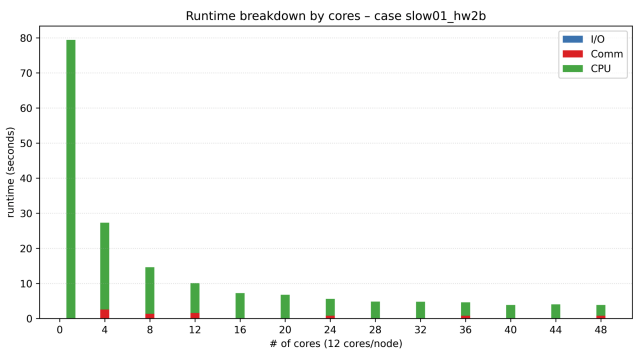
- 將每 thread 的工作區塊（row block）連續存取。
- 減少 cache line miss：確保影像 buffer 對齊、使用 restrict 指標。
- 若平台支援，可使用 NUMA-aware placement (numactl 或 OMP_PLACES=cores 配置)。

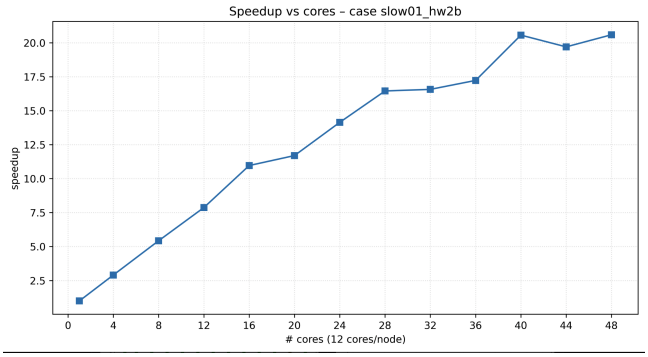
輸出與後處理 I/O 若要支援更大圖（如 8K 以上）採用 parallel I/O (MPI-IO) 或 分區寫出再合併。 可避免最終寫檔階段成為瓶頸。

Experiments

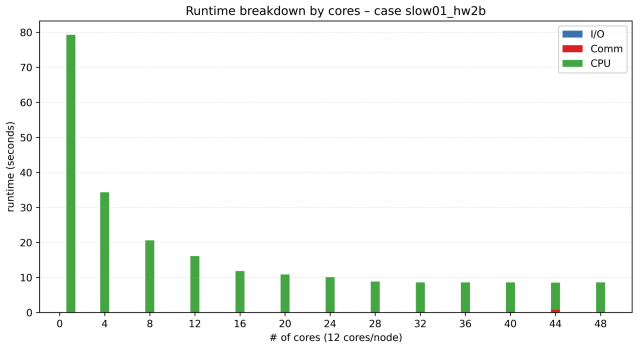
實驗1

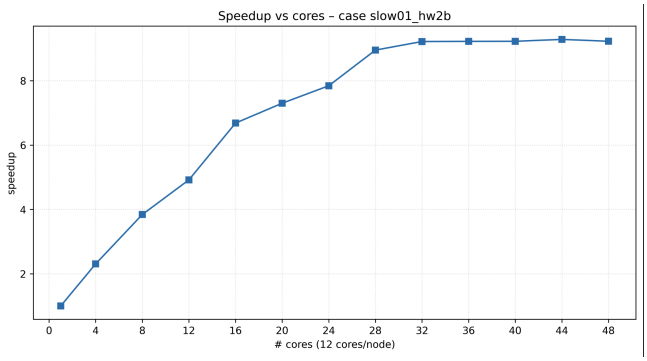
實驗目的：資料以連續的row分配還是循環分配比較好 資料循環分配





資料連續分配



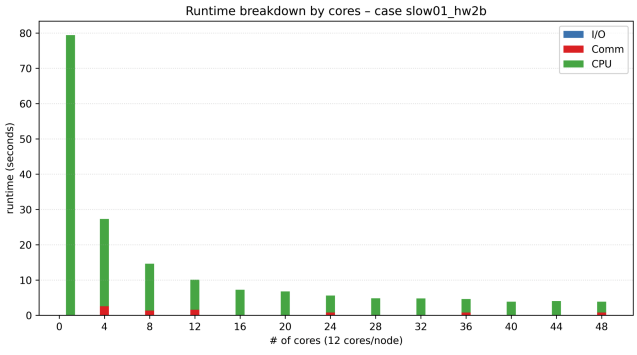


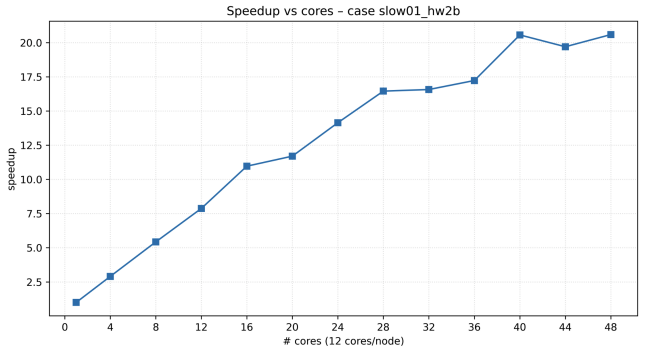
實驗結果： 經由循環跨列分配的結果可以讓每個子任務更平均，讓整體Scalability大幅提升 Speedup提升： 9 -> 21 judge的總秒數也大幅提升

```
Scoreboard: updated {68 371.88} --> {68 265.56}
```

實驗2

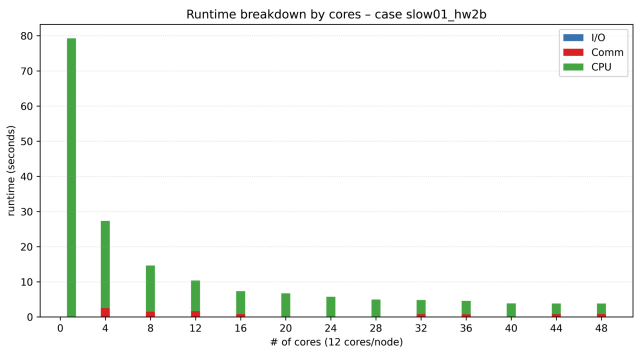
實驗目的：使用gatherv去收集結果還是用sendrecv比較快 gatherv

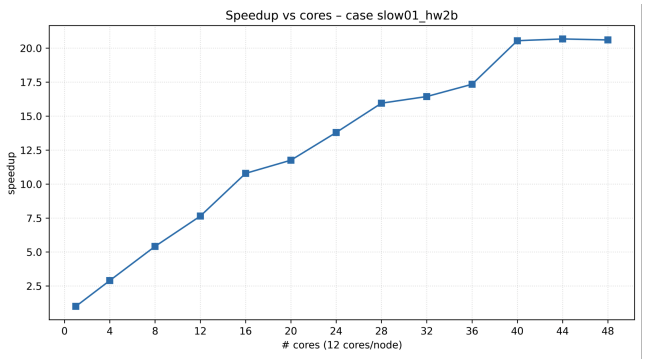




---

sendrecv





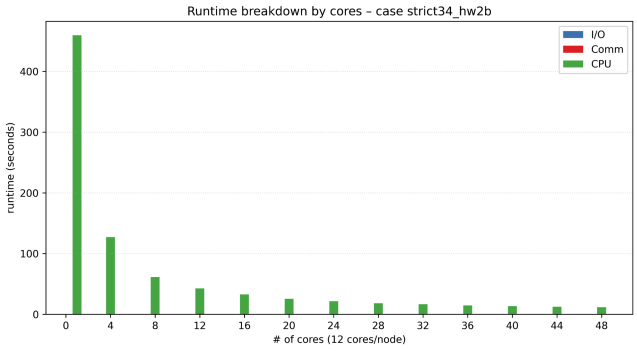
實驗結果： 在sendrecv的方法中最高speedup停在了20，但gatherv似乎還有機會在往上增長 就運行速度來看這兩種方法差不多

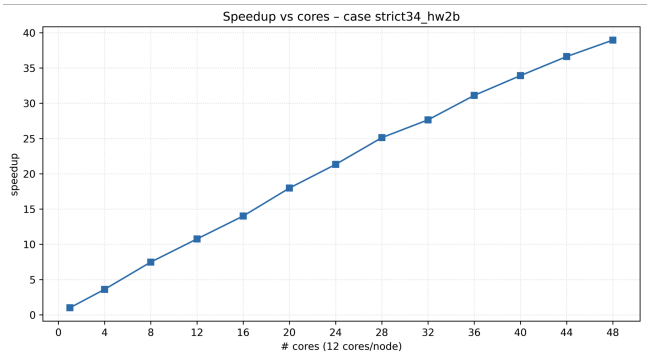
最後選擇使用gather作為結果收集的方法

實驗3

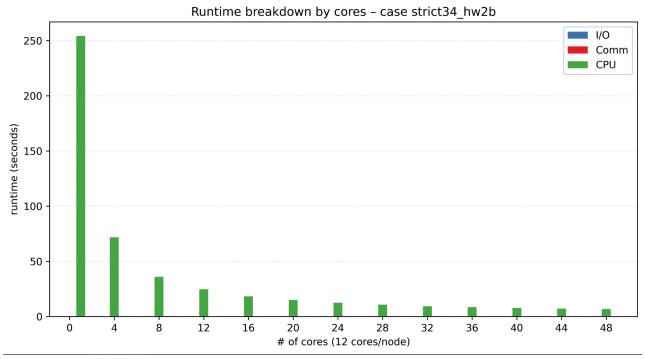
本實驗跑在strict34因為資料集才夠大顯示出效能提升

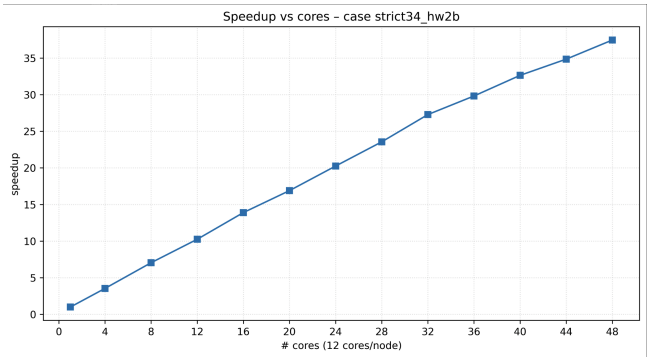
實驗目的：如果一次讓SIMD指令一次算三組資料 SIMD算一組





SIMD算三個





採用「三組 SIMD 向量併行」策略，同時計算 6 個像素。 這樣能增加指令平行度、降低 mask 分歧帶來的浪費、提升 FPU 管線利用率， 我處理的data 都沒有dependency所以可以同時操作

judge的總秒數也大幅提升

```
Scoreboard: updated {68 371.88} --> {68 186.27}
```

Conclusion and learn

本次實驗讓我體悟到load balance對Scalability的影響，有時候並不是單單是運算處理的優化，多個工作必須同時被協作也是很重要的，如果沒辦法平均的分配資料，就會出現某些工作節點運算過久成為瓶頸

Difficulties

1. 起初在實作task queue時無法很好的找到一個方法讓process不要重複取到相同任務，原本實作的lock又太耗時，變成花了很多時間在處理queue的鎖，最後找到一個atomic的操作方法才很好解決這個問題
2. SIMD early stop 實作時遇到一個問題，同個指令運算時遇到的兩個pixel可能發散的時間不同，如果兩個都發散了就不用繼續做了可以提早終止，所以我利用一個mask看兩邊的pixel是否都發散了，如果都發散了就提早離開while

```
while (k < iters) {
            int state = _mm_movemask_pd(mask);
            if(state == 0){
                break;
            };
            for (int t = 0; t < 64; ++t) {
                length_squared = iter_cal(x,y,xx,yy,xs,ys);
                repeats += _mm_and_pd(mask, c1);
                mask = (length_squared < c4);
            }
            k+=64;
}
```

並且經過實驗後我發現一個iter做三組SIMD運算可以更近一步提升效能

```
while (k < iters) {
            int st_any = (_mm_movemask_pd(m0) |
                          _mm_movemask_pd(m1) |
                          _mm_movemask_pd(m2));
            if (st_any == 0) break;

            for (int t = 0; t < 64 ; ++t) {
                // 組0
                len0 = iter_cal(&x0v, &y0v, &xx0, &yy0, xs0, ys);
                rep0 = _mm_add_pd(rep0, _mm_and_pd(m0, c1));
                m0   = _mm_cmplt_pd(len0, c4);
                // 組1
                len1 = iter_cal(&x1v, &y1v, &xx1, &yy1, xs1, ys);
                rep1 = _mm_add_pd(rep1, _mm_and_pd(m1, c1));
                m1   = _mm_cmplt_pd(len1, c4);
                // 組2
                len2 = iter_cal(&x2v, &y2v, &xx2, &yy2, xs2, ys);
                rep2 = _mm_add_pd(rep2, _mm_and_pd(m2, c1));
```

```
                m2   = _mm_cmplt_pd(len2, c4);
            }
        }
```