Homework 2: Mandelbrot Set Report

- **Name**: 廖思愷
- **Student ID**: 112062519

Implementation

Pthread: Single node shared memory programming using Pthread

程式碼

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#define PNG_NO_SETJMP
#include <sched.h>
#include <assert.h>
#include <png.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <immintrin.h>

typedef struct {
    int id;
    int thread_count;
    int width;
    int height;
    int iters;
    double left;
    double right;
    double lower;
    double upper;
    int* image;
} thread_arg_t;

void write_png(const char* filename, int iters, int width, int height, const int*
buffer) {
    FILE* fp = fopen(filename, "wb");
    assert(fp);
    png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL,
NULL, NULL);
    assert(png_ptr);
    png_infop info_ptr = png_create_info_struct(png_ptr);
    assert(info_ptr);
    png_init_io(png_ptr, fp);
    png_set_IHDR(png_ptr, info_ptr, width, height, 8, PNG_COLOR_TYPE_RGB,
PNG_INTERLACE_NONE,
```

```
                    PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
    png_set_filter(png_ptr, 0, PNG_FILTER_NONE);
    png_write_info(png_ptr, info_ptr);
    png_set_compression_level(png_ptr, 0);
    size_t row_size = 3 * width * sizeof(png_byte);
    png_bytep row = (png_bytep)malloc(row_size);
    for (int y = 0; y < height; ++y) {
        memset(row, 0, row_size);
        for (int x = 0; x < width; ++x) {
            int p = buffer[(height - 1 - y) * width + x];
            png_bytep color = row + x * 3;
            if (p != iters) {
                if (p & 16) {
                    color[0] = 240;
                    color[1] = color[2] = p % 16 * 16;
                } else {
                    color[0] = p % 16 * 16;
                }
            }
        }
        png_write_row(png_ptr, row);
    }
    free(row);
    png_write_end(png_ptr, NULL);
    png_destroy_write_struct(&png_ptr, &info_ptr);
    fclose(fp);
}

void mandelbrot_calculate_point(double x0, double y0, int max_iters, int* result)
{
    double x = 0, y = 0;
    int repeats = 0;
    while (repeats < max_iters && x*x + y*y < 4) {
        double temp = x*x - y*y + x0;
        y = 2*x*y + y0;
        x = temp;
        repeats++;
    }
    *result = repeats;
}

void* mandelbrot_thread(void* arg) {
    thread_arg_t* targ = (thread_arg_t*)arg;

    __m512d v_four = _mm512_set1_pd(4.0);
    __m512d v_two = _mm512_set1_pd(2.0);
    __m512d v_x0, v_y0, v_x, v_y, v_xx, v_yy, v_xy, v_length_squared;
    __m512i v_repeats, v_one, v_iters;
    __mmask8 mask;

    v_one = _mm512_set1_epi64(1);
    v_iters = _mm512_set1_epi64(targ->iters);

    double x0_step = (targ->right - targ->left) / targ->width;
```

```
    for (int j = targ->id; j < targ->height; j += targ->thread_count) {
        double y0 = j * ((targ->upper - targ->lower) / targ->height) + targ-
>lower;
        v_y0 = _mm512_set1_pd(y0);

        int i;
        for (i = 0; i + 7 < targ->width; i += 8) {
            v_x0 = _mm512_set_pd(
                (i + 7) * x0_step + targ->left,
                (i + 6) * x0_step + targ->left,
                (i + 5) * x0_step + targ->left,
                (i + 4) * x0_step + targ->left,
                (i + 3) * x0_step + targ->left,
                (i + 2) * x0_step + targ->left,
                (i + 1) * x0_step + targ->left,
                i * x0_step + targ->left
            );

            v_x = _mm512_setzero_pd();
            v_y = _mm512_setzero_pd();
            v_repeats = _mm512_setzero_si512();

            mask = 0xFF;  // All 8 bits set
            for (int k = 0; k < targ->iters && mask; ++k) {
                // Compute x and y squares
                v_xx = _mm512_mul_pd(v_x, v_x);
                v_yy = _mm512_mul_pd(v_y, v_y);

                // Calculate lengths and update mask
                v_length_squared = _mm512_add_pd(v_xx, v_yy);
                __mmask8 new_mask = _mm512_cmp_pd_mask(v_length_squared, v_four,
_CMP_LT_OQ);

                // Break if all points have escaped
                if (!new_mask) break;

                // Update x and y
                v_xy = _mm512_mul_pd(v_x, v_y);
                v_x = _mm512_add_pd(_mm512_sub_pd(v_xx, v_yy), v_x0);
                v_y = _mm512_fmadd_pd(v_two, v_xy, v_y0);

                // Increment repeats where mask is true
                v_repeats = _mm512_mask_add_epi64(v_repeats, new_mask, v_repeats,
v_one);

                mask = new_mask;
            }

            int64_t repeats[8];
            _mm512_storeu_si512((__m512i*)repeats, v_repeats);

            for (int k = 0; k < 8; ++k) {
                targ->image[j * targ->width + i + k] = (int)repeats[k];
```

```
            }
        }

        // Handle remaining pixels sequentially
        for (; i < targ->width; ++i) {
            double x0 = i * x0_step + targ->left;
            mandelbrot_calculate_point(x0, y0, targ->iters, &targ->image[j * targ-
>width + i]);
        }
    }

    return NULL;
}

int main(int argc, char** argv) {
    /* detect how many CPUs are available */
    cpu_set_t cpu_set;
    sched_getaffinity(0, sizeof(cpu_set), &cpu_set);
    int num_cpus = CPU_COUNT(&cpu_set);
    // printf("%d cpus available\n", num_cpus);

    /* argument parsing */
    assert(argc == 9);
    const char* filename = argv[1];
    int iters = strtol(argv[2], 0, 10);
    double left = strtod(argv[3], 0);
    double right = strtod(argv[4], 0);
    double lower = strtod(argv[5], 0);
    double upper = strtod(argv[6], 0);
    int width = strtol(argv[7], 0, 10);
    int height = strtol(argv[8], 0, 10);

    /* allocate memory for image */
    int* image = (int*)malloc(width * height * sizeof(int));
    // assert(image);

    /* prepare thread arguments */
    int num_threads = num_cpus;  // Use number of CPUs as number of threads
    pthread_t threads[num_threads];
    thread_arg_t thread_args[num_threads];

    /* create threads */
    for (int i = 0; i < num_threads; ++i) {
        thread_args[i] = (thread_arg_t){
            .id = i,
            .thread_count = num_threads,
            .width = width,
            .height = height,
            .iters = iters,
            .left = left,
            .right = right,
            .lower = lower,
            .upper = upper,
            .image = image
```

```
        };
        pthread_create(&threads[i], NULL, mandelbrot_thread, &thread_args[i]);
    }

    /* wait for threads to finish */
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], NULL);
    }

    /* draw and cleanup */
    write_png(filename, iters, width, height, image);
    free(image);

    return 0;
}
```

實作說明 (以下的"行" = row)

1. 任務分割與交錯處理

每個 thread 透過交錯行的方式來分擔整張圖片的運算。具體來說，假設總共有`thread_count`個 thread ，則 thread `id`會負責計算所有行數為`id, id + thread_count, id + 2 * thread_count, ...`的影像行。這樣的分割方式比起讓一個 thread 處理連續的 rows，更能平衡整張圖片的運算量，避免單一 thread 過載。

2. 使用SIMD指令進行運算

採用AVX-512的SIMD指令來進行加速運算：

- **載入資料到SIMD暫存器**：每次處理8個像素點，並將需要使用到的常數、變數載入至SIMD暫存器中。
- **執行運算**：在每次迭代中，使用向量運算來同時計算8個點的x和y值的更新。
- **更新與檢查是否結束**：每次運算都會檢查當前8個點的是否都運算結束了，透過mask來篩選尚未結束的點並更新其迭代次數。
- **將結果搬回主記憶體**：當完成一行的計算後，會將暫存器中的計算結果存回主記憶體，填入到影像buffer中。

3. 處理SIMD不整除情況

如果影像的寬度無法被8整除，則無法一次處理8個像素點。在這種情況下，程式會先處理可以整除8的部分，然後在行的尾端使用一般的逐點計算方式，處理剩餘的像素點。

Hybrid: Multi-node hybrid parallelism programming using MPI + OpenMP

程式碼

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#define PNG_NO_SETJMP
```

```
#include <mpi.h>
#include <omp.h>
#include <sched.h>
#include <assert.h>
#include <png.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <immintrin.h>

void write_png(const char* filename, int iters, int width, int height, const int*
buffer) {
    FILE* fp = fopen(filename, "wb");
    assert(fp);
    png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL,
NULL, NULL);
    assert(png_ptr);
    png_infop info_ptr = png_create_info_struct(png_ptr);
    assert(info_ptr);
    png_init_io(png_ptr, fp);
    png_set_IHDR(png_ptr, info_ptr, width, height, 8, PNG_COLOR_TYPE_RGB,
PNG_INTERLACE_NONE,
                 PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
    png_set_filter(png_ptr, 0, PNG_FILTER_NONE);
    png_write_info(png_ptr, info_ptr);
    png_set_compression_level(png_ptr, 0);
    size_t row_size = 3 * width * sizeof(png_byte);
    png_bytep row = (png_bytep)malloc(row_size);
    for (int y = 0; y < height; ++y) {
        memset(row, 0, row_size);
        for (int x = 0; x < width; ++x) {
            int p = buffer[(height - 1 - y) * width + x];
            png_bytep color = row + x * 3;
            if (p != iters) {
                if (p & 16) {
                    color[0] = 240;
                    color[1] = color[2] = p % 16 * 16;
                } else {
                    color[0] = p % 16 * 16;
                }
            }
        }
        png_write_row(png_ptr, row);
    }
    free(row);
    png_write_end(png_ptr, NULL);
    png_destroy_write_struct(&png_ptr, &info_ptr);
    fclose(fp);
}

// Helper function for non-SIMD calculation of remaining points
void mandelbrot_calculate_point(double x0, double y0, int max_iters, int* result)
{
    double x = 0, y = 0;
```

```
    int repeats = 0;
    while (repeats < max_iters && x*x + y*y < 4) {
        double temp = x*x - y*y + x0;
        y = 2*x*y + y0;
        x = temp;
        repeats++;
    }
    *result = repeats;
}

// SIMD-accelerated function to compute the Mandelbrot set for multiple rows
void compute_mandelbrot_rows(int* rows_data, int* row_indices, int num_rows,
                             int width, int height, int iters,
                             double left, double right, double lower, double upper)
{
    double x0_step = (right - left) / width;

    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < num_rows; i++) {
        int row = row_indices[i];
        double y0 = row * ((upper - lower) / height) + lower;

        // SIMD variables
        __m512d v_four = _mm512_set1_pd(4.0);
        __m512d v_two = _mm512_set1_pd(2.0);
        __m512d v_x0, v_y0, v_x, v_y, v_xx, v_yy, v_xy, v_length_squared;
        __m512i v_repeats, v_one, v_iters;
        __mmask8 mask;

        v_one = _mm512_set1_epi64(1);
        v_iters = _mm512_set1_epi64(iters);
        v_y0 = _mm512_set1_pd(y0);

        // Process 8 pixels at a time using AVX-512
        int x;
        for (x = 0; x + 7 < width; x += 8) {
            v_x0 = _mm512_set_pd(
                (x + 7) * x0_step + left,
                (x + 6) * x0_step + left,
                (x + 5) * x0_step + left,
                (x + 4) * x0_step + left,
                (x + 3) * x0_step + left,
                (x + 2) * x0_step + left,
                (x + 1) * x0_step + left,
                x * x0_step + left
            );

            v_x = _mm512_setzero_pd();
            v_y = _mm512_setzero_pd();
            v_repeats = _mm512_setzero_si512();

            mask = 0xFF;  // All 8 bits set
            for (int k = 0; k < iters && mask; ++k) {
                // Compute x and y squares
```

```
                v_xx = _mm512_mul_pd(v_x, v_x);
                v_yy = _mm512_mul_pd(v_y, v_y);

                v_length_squared = _mm512_add_pd(v_xx, v_yy);
                mask = _mm512_cmp_pd_mask(v_length_squared, v_four, _CMP_LT_OQ);

                // Update x and y
                v_xy = _mm512_mul_pd(v_x, v_y);
                v_x = _mm512_add_pd(_mm512_sub_pd(v_xx, v_yy), v_x0);
                v_y = _mm512_fmadd_pd(v_two, v_xy, v_y0);

                v_repeats = _mm512_mask_add_epi64(v_repeats, mask, v_repeats,
v_one);
            }

            int64_t repeats[8];
            _mm512_storeu_si512((__m512i*)repeats, v_repeats);

            for (int k = 0; k < 8; ++k) {
                rows_data[i * width + x + k] = (int)repeats[k];
            }
        }

        // Handle remaining pixels sequentially
        for (; x < width; ++x) {
            double x0 = x * x0_step + left;
            mandelbrot_calculate_point(x0, y0, iters, &rows_data[i * width + x]);
        }
    }
}

int main(int argc, char** argv) {
    int rank, num_procs;

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    // Parse arguments
    // assert(argc == 9);
    const char* filename = argv[1];
    int iters = strtol(argv[2], 0, 10);
    double left = strtod(argv[3], 0);
    double right = strtod(argv[4], 0);
    double lower = strtod(argv[5], 0);
    double upper = strtod(argv[6], 0);
    int width = strtol(argv[7], 0, 10);
    int height = strtol(argv[8], 0, 10);

    // Set number of OpenMP threads based on available CPUs
    cpu_set_t cpu_set;
    sched_getaffinity(0, sizeof(cpu_set), &cpu_set);
    int num_cpus = CPU_COUNT(&cpu_set);
```

```
    omp_set_num_threads(num_cpus);

    // Calculate exact number of rows for this process
    int my_row_count = 0;
    for (int row = rank; row < height; row += num_procs) {
        my_row_count++;
    }

    // Allocate arrays for row indices and data
    int* row_indices = (int*)malloc(my_row_count * sizeof(int));
    int* local_rows_data = (int*)malloc(my_row_count * width * sizeof(int));
    assert(row_indices && local_rows_data);

    // Calculate row indices for this process
    int idx = 0;
    for (int row = rank; row < height; row += num_procs) {
        row_indices[idx++] = row;
    }

    // Compute assigned rows using SIMD-accelerated function
    compute_mandelbrot_rows(local_rows_data, row_indices, my_row_count,
                            width, height, iters, left, right, lower, upper);

    // Gather the row counts from all processes
    int* all_row_counts = NULL;
    if (rank == 0) {
        all_row_counts = (int*)malloc(num_procs * sizeof(int));
        assert(all_row_counts);
    }

    // Gather number of rows from each process
    MPI_Gather(&my_row_count, 1, MPI_INT, all_row_counts, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    // Prepare arrays for gathering data
    int* displs = NULL;
    int* row_displs = NULL;
    int* all_row_indices = NULL;
    int* full_image = NULL;

    if (rank == 0) {
        displs = (int*)malloc(num_procs * sizeof(int));
        row_displs = (int*)malloc(num_procs * sizeof(int));
        all_row_indices = (int*)malloc(height * sizeof(int));
        full_image = (int*)malloc(width * height * sizeof(int));
        assert(displs && row_displs && all_row_indices && full_image);

        // Calculate displacements
        displs[0] = 0;
        row_displs[0] = 0;
        for (int i = 1; i < num_procs; i++) {
            displs[i] = displs[i-1] + all_row_counts[i-1];
            row_displs[i] = displs[i];
        }
```

```
    }

    // Gather row indices
    MPI_Gatherv(row_indices, my_row_count, MPI_INT,
                all_row_indices, all_row_counts, row_displs,
                MPI_INT, 0, MPI_COMM_WORLD);

    // Prepare counts for data gathering (multiply by width)
    int my_data_count = my_row_count * width;
    int* all_data_counts = NULL;
    int* data_displs = NULL;

    if (rank == 0) {
        all_data_counts = (int*)malloc(num_procs * sizeof(int));
        data_displs = (int*)malloc(num_procs * sizeof(int));
        assert(all_data_counts && data_displs);

        for (int i = 0; i < num_procs; i++) {
            all_data_counts[i] = all_row_counts[i] * width;
            data_displs[i] = displs[i] * width;
        }
    }

    // Gather computed data
    MPI_Gatherv(local_rows_data, my_data_count, MPI_INT,
                full_image, all_data_counts, data_displs,
                MPI_INT, 0, MPI_COMM_WORLD);

    // Rank 0 reorganizes the data into the final image
    if (rank == 0) {
        int* temp_image = (int*)malloc(width * height * sizeof(int));
        assert(temp_image);

        // Copy data to correct positions
        for (int i = 0; i < num_procs; i++) {
            for (int j = 0; j < all_row_counts[i]; j++) {
                int src_row = j;
                int dst_row = all_row_indices[displs[i] + j];
                memcpy(temp_image + dst_row * width,
                       full_image + (data_displs[i] + j * width),
                       width * sizeof(int));
            }
        }

        // Write the final image
        write_png(filename, iters, width, height, temp_image);

        // Cleanup
        free(temp_image);
        free(all_row_counts);
        free(all_data_counts);
        free(data_displs);
        free(displs);
        free(row_displs);
```

```
        free(all_row_indices);
        free(full_image);
    }

    // Cleanup
    free(row_indices);
    free(local_rows_data);

    MPI_Finalize();
    return 0;
}
```

實作說明

Hybrid版本的程式以MPI為基礎進行**多節點分工**，每個節點負責計算整張影像的部分行數，盡量讓計算負擔在多個節點間平均分配。在每個節點內，則利用OpenMP來進行**多執行緒運算**，並配合SIMD指令（AVX-512）來實現對多個像素點的並行處理，提高單節點的計算效率。

實現細節

1. **Load Balance (MPI)**：
    - 類似 pthread 版本的作法，MPI負責在多個節點之間分配影像的行數。假設共有`num_procs`個節點，則節點`rank`負責計算影像中的行數為`rank, rank + num_procs, rank + 2 * num_procs, ...`。
    - 每個節點會根據分配到的行數，將要處理的 rows 存入一個 local array 中，以便後續OpenMP的並行處理。並且也要記錄 row indices，也就是每個 process 分別處理哪些 rows，這樣最後計算完才能將數據放入正確位置。
    - 這樣的分割方式比起讓一個 process 處理連續的 rows，更能平衡整張圖片的運算量，避免單一 process 過載。
2. **每節點內的 Thread 分工 (OpenMP)**：
    - 在每個節點內，透過OpenMP來平行處理分配到的 row 數量。為了提高負載均衡性，使用動態排程（`#pragma omp parallel for schedule(dynamic, 1)`）讓OpenMP動態分配 rows 給執行緒，每次分配 1 行。這樣的設計可以讓 thread 根據可用資源動態負責不同 rows，提升內部的負載均衡性。
3. **AVX-512 SIMD指令加速**：
    - 與pthread版本相似，使用AVX-512指令進行每行的像素計算。每次處理8個像素點，進行並行運算。
    - 計算過程包括載入SIMD暫存器、計算x和y的平方、更新座標、用 mask 檢查是否結束等。
    - 如果影像的寬度無法被8整除，則剩餘像素點使用逐點計算方式處理，避免資料遺漏。
4. **資料收集與影像合併 (MPI Gather)**：
    - 計算完成後，每個節點將計算結果回傳給主節點。主節點負責根據 row indices 將所有節點的計算結果組合成完整的影像。

Experiment & Analysis

Methodology

System Spec

使用課程提供的 QCT server

Performance Metrics

- Pthread version

    1. **計算程式執行時間與效能指標：**

        - 在程式中用 `clock_gettime(CLOCK_MONOTONIC, ...)` 測量各線程的執行時間以及程式執行的總時間，並在程式的最後計算以下指標：
            - **最小執行時間（Minimum Execution Time）**：取每次實驗中執行時間最短的 thread。
            - **最大執行時間（Maximum Execution Time）**：記錄執行時間最長的 thread。
            - **平均執行時間（Average Execution Time）**：取所有 thread 執行時間的平均值。
            - **標準差（Standard Deviation）**：計算各個 thread 執行時間的標準差，以評估執行時間的波動性。
            - **負載不平衡因子（Load Imbalance Factor）**：通過最大執行時間和最小執行時間的比率，來量化負載不均程度（越接近 1 越均衡）。

    2. **多線程配置測試：**

        - 通過不同的線程數進行多次測試，每組配置執行三次，記錄每次的執行時間及負載平衡數據。
        - 使用腳本自動執行程式，並將結果存入 `execution_times.txt` 和 `load_balance_data.txt` 文件，以便後續用 python 繪圖分析。

    3. **用 python 進行數據視覺化：**

        - **總執行時間圖**：繪製線程數對應的總執行時間折線圖，以直觀展示不同線程數下的總執行效率。
        - **加速比圖（Speedup）**：基於單線程執行時間計算加速比，並與理想線性加速進行比較，展示程式的並行可擴展性。
        - **負載不平衡圖**：繪製負載不平衡因子隨線程數變化的圖表，評估各配置下的負載均衡情況。
        - **標準差圖**：展示每種線程數下各線程執行時間的標準差，以觀察負載分配的一致性。

- Hybrid Version

    1. **計算程式執行時間與負載平衡指標：**

        - 使用 `clock_gettime(CLOCK_MONOTONIC, ...)` 測量程式執行的總時間

    2. **三種測試策略：**

        a. 固定總核心數(96)測試：

- 測試不同 process/thread 組合下的效能，但保持總計算資源不變。
- 觀察 MPI/OpenMP 比例對效能的影響。

b. Process擴展性測試：

- 固定每個 process 的 thread 數（2個），觀察增加 process 數量的效果。
- 評估 MPI 層級的擴展性。

c. Thread擴展性測試：

- 固定 process 數（8個），觀察增加 thread 數量的效果。
- 評估 OpenMP 層級的擴展性。

3. **數據視覺化：**

- 使用 Python 產生分析圖表：
  - 執行時間折線圖
  - 不同配置下的 Speedup 圖

Plots: Scalability & Load Balancing & Profile

Experimental Method

- Test Case Description
  - 本次實驗不管是 pthread version 還是 hybrid version 都選擇使用作業提供的 strict34.txt 測資進行測試，因為該測資是我在所有測資中執行時間最久的 (pthread version 與 hybrid version 都是), 所以選用這筆測資來實驗看看效能瓶頸可能在哪
- Parallel Configurations
  - Pthread Version
    - 在這個版本的實驗中，process 數都固定為 1 (-n1)，實驗不同 thread 數 (-c) {1, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96} 下的情況
  - Hybrid Version
    - 這個實驗採用三種不同的測試配置：

      1. **固定總核心數(96)的配置：**

```
srun -n96 -c1    # 96 processes, 1 thread each
srun -n48 -c2    # 48 processes, 2 threads each
srun -n32 -c3    # 32 processes, 3 threads each
srun -n24 -c4    # 24 processes, 4 threads each
srun -n16 -c6    # 16 processes, 6 threads each
srun -n12 -c8    # 12 processes, 8 threads each
srun -n8  -c12   # 8 processes, 12 threads each
srun -n6  -c16   # 6 processes, 16 threads each
srun -n4  -c24   # 4 processes, 24 threads each
```

```
srun -n2  -c48   # 2 processes, 48 threads each
srun -n1  -c96   # 1 process, 96 threads each
```

- 目的：測試在相同計算資源下，MPI processes 和 OpenMP threads 的最佳配比。

2. **Process擴展性測試配置**（固定thread數=2）：

```
srun -n1  -c2    # 基準點
srun -n2  -c2
srun -n4  -c2
srun -n8  -c2
srun -n16 -c2
srun -n24 -c2
srun -n32 -c2
srun -n48 -c2
```

- 目的：評估程式在多節點下的擴展性，觀察 MPI 通信開銷的影響。

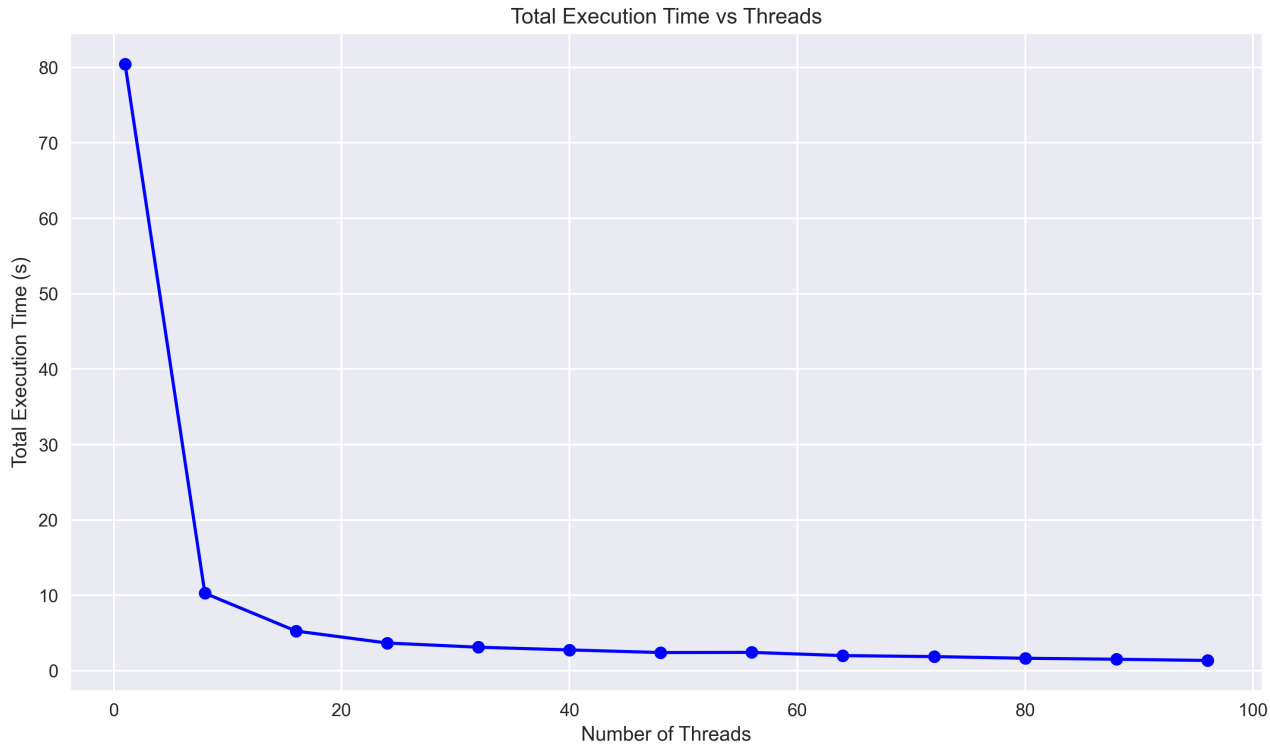
3. **Thread擴展性測試配置**（固定process數=8）：

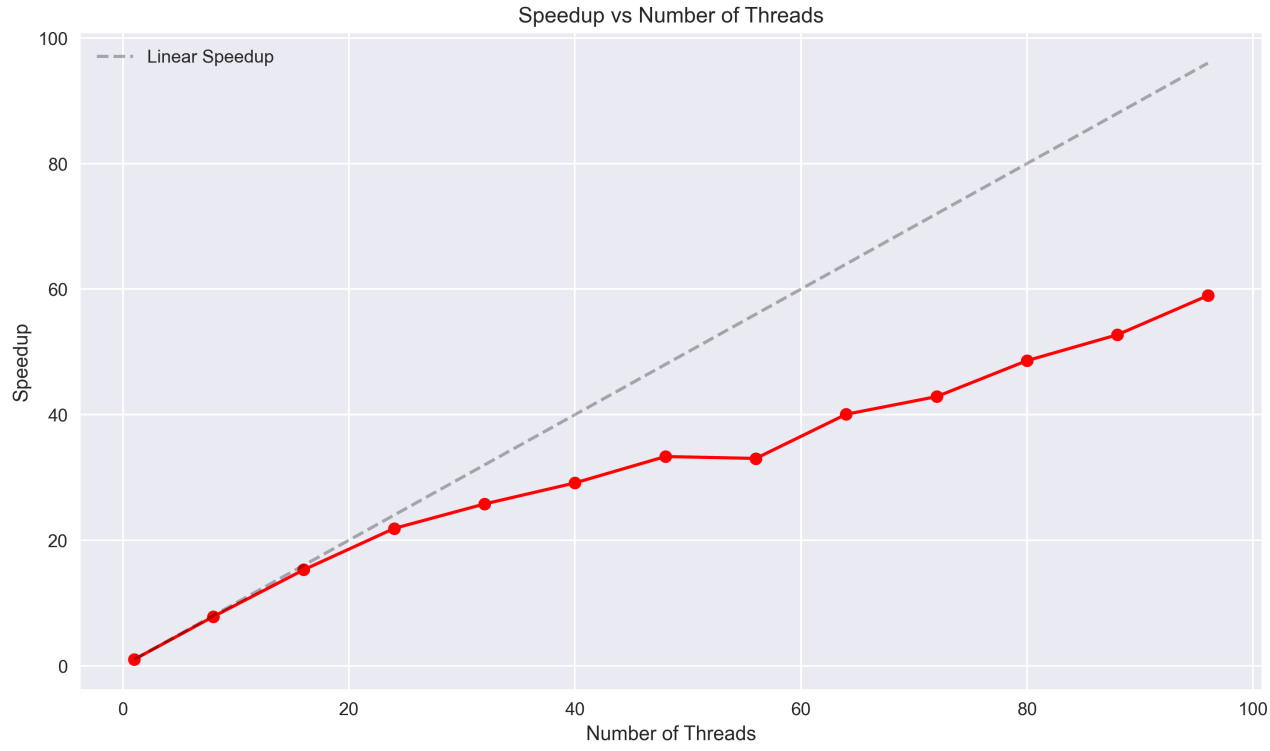
```
srun -n8 -c1
srun -n8 -c2
srun -n8 -c3
srun -n8 -c4
srun -n8 -c6
srun -n8 -c8
srun -n8 -c12
```

- 目的：評估程式在單節點內的擴展性，觀察 OpenMP 並行效率。

Analysis of Results

- 實驗一：Pthread Version Scalability





Total Execution Time vs Threads

根據總執行時間圖（Total Execution Time vs Threads）的結果，隨著線程數的增加，總執行時間明顯下降。單線程運行時，執行時間約為 80 秒，而增加線程數後，執行時間急劇縮短，尤其是在 8 到 16 線程之間。從 16 線程以後，總執行時間的減少幅度逐漸變小，這顯示出多線程計算在一定線程數下能顯著提升效能，但超過某個臨界值後，效能提升趨於緩慢。

分析此現象的原因如下：

1. **任務分割和負載均衡**：在少數線程時，任務分割較為集中，每個線程的負擔較重，因而隨著線程增加，任務能有效地被更多線程分擔，總執行時間快速下降。然而當線程數進一步增加至某個值後，每個線程所需處理的工作量變小，負載均衡的效果逐漸顯現出極限。
2. **資源競爭**：當線程數接近或超過 CPU 核心數時，各線程之間可能出現資源競爭，包括記憶體頻寬、快取等資源的爭奪。這會導致效能提升幅度減少。

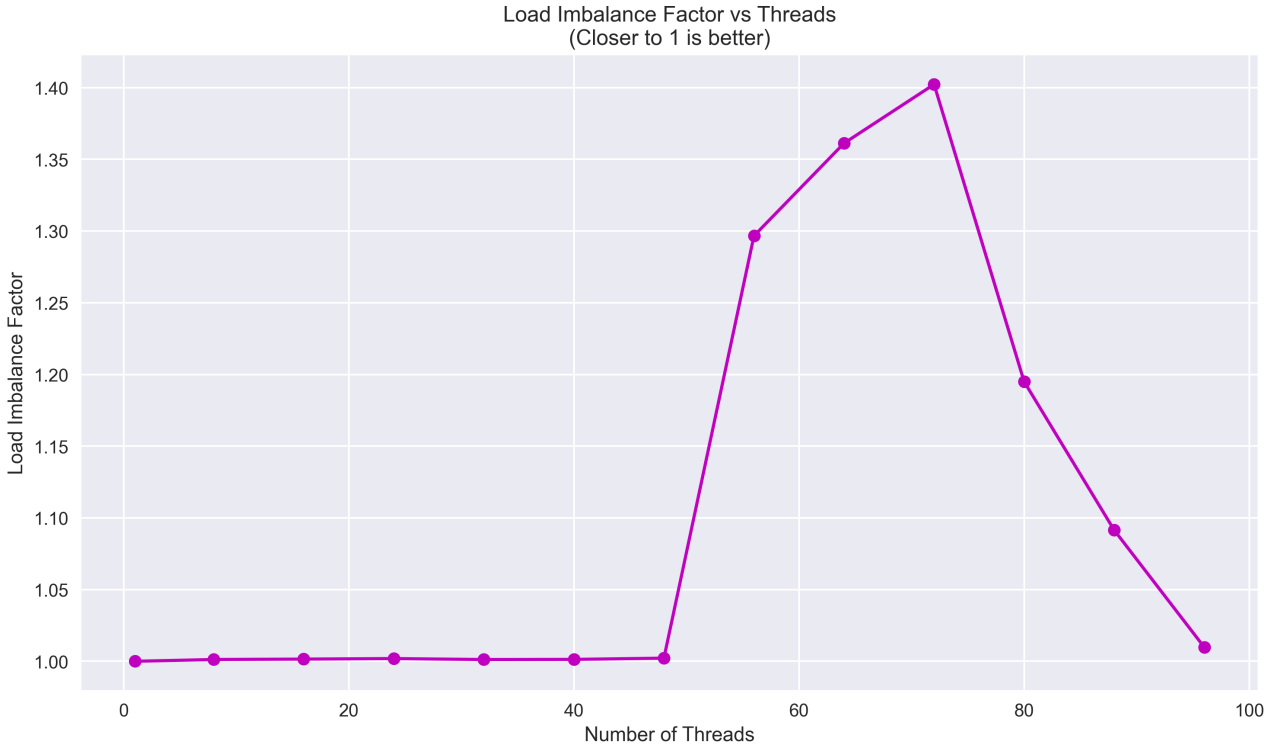
Speedup vs Number of Threads

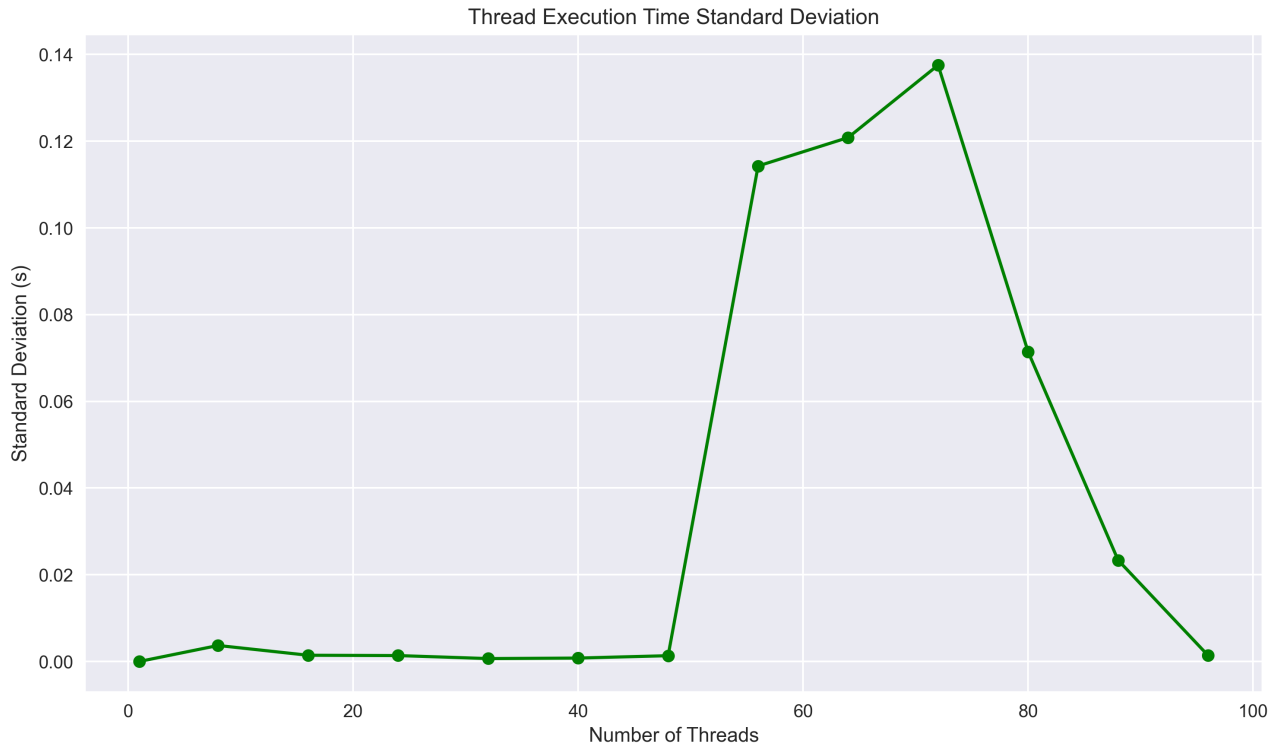
在加速比圖（Speedup vs Number of Threads）中，可以觀察到隨著線程數增加，加速比相應提升，但其提升的趨勢並未完全接近理想線性加速。雖然在前幾個線程配置中，加速比有接近線性增長的趨勢，但在 32 線程以後，加速比增長逐漸趨於緩慢，且與理想線性加速的差距逐漸拉大。

主要原因分析如下：

1. **Amdahl's Law**：該現象符合 Amdahl 定律，即當並行度受到限制造成某部分的計算無法被完全並行化時，系統的加速比會受到瓶頸限制。在此實驗中，隨著線程數增加，程式的執行速度可能被如線程間同步、資料搬移等開銷所限制，導致加速比無法達到理想值。
2. **同步開銷**：隨著線程數增加，線程之間的同步和資源競爭開銷也會增加。即使每個線程的計算負載較小，但同步機制和資源分配的耗時增加，影響了整體的加速效果。
3. **快取一致性問題**：在多線程環境下，若各線程之間的快取資料頻繁更新，則會引起快取一致性機制的開銷，進一步影響加速比。

- 實驗二：Pthread Version Load Balance





在負載不平衡因子（Load Imbalance Factor）與標準差（Standard Deviation）兩張圖中，皆呈現出隨著線程數的增加，指標先上升後下降的現象。當線程數小於 48 時，負載不平衡因子接近 1，標準差也接近 0，表明各線程的執行時間接近，負載分布較為均衡。然而，當線程數增加到 64 或更多時，負載不平衡因子急劇上升，標準差也顯著增加，顯示各線程間的負載分配開始不均勻，並在 64 至 80 線程之間達到峰值。隨後，當線程數接近或超過 96 時，兩者都逐漸回落，重新接近均衡狀態。

這種現象推測與以下因素有關：

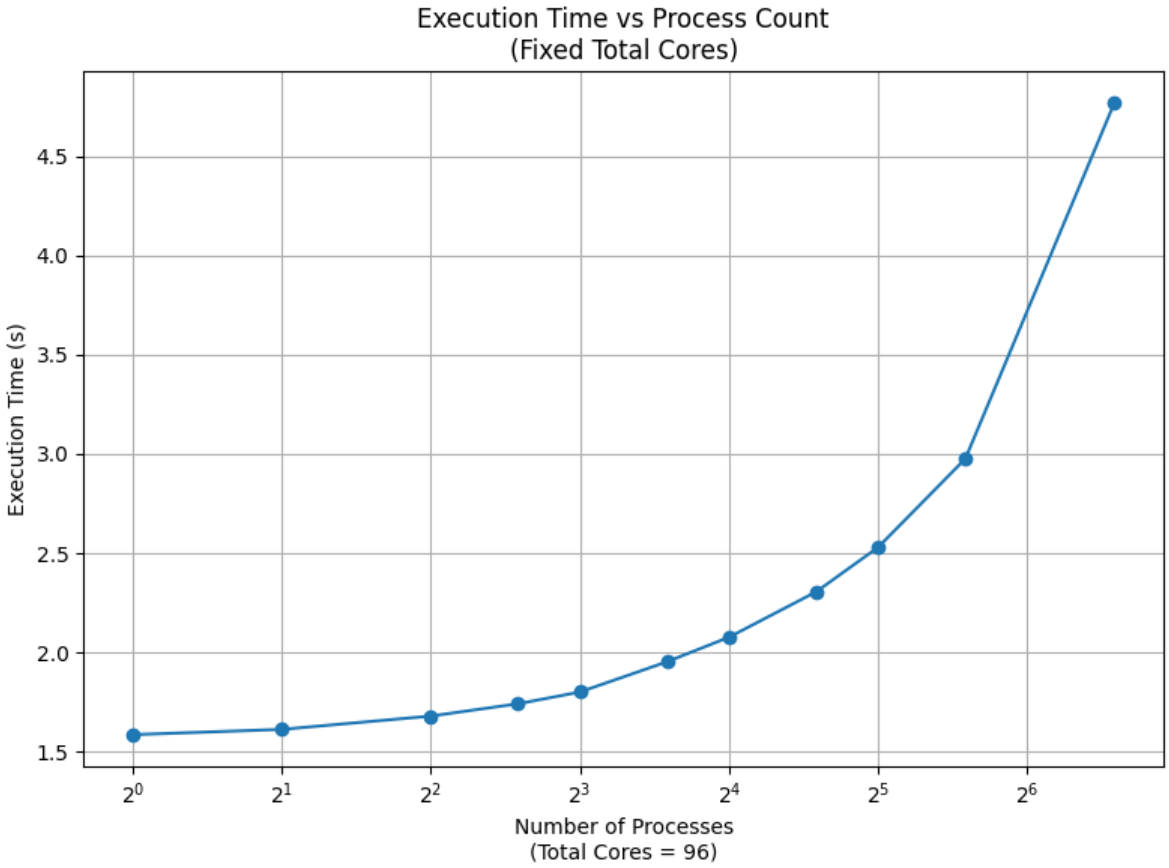
1. **任務分割方式**：在本實驗中，Mandelbrot Set 的計算採用交錯行的方式分配給各線程。推測是當線程數較少時（如 1 到 48），每個線程負責的行數較多，各行的計算負載差異被平攤，因此負載不平衡因子和標準差較低。然而，當線程數達到 64 或更多時，分配的行數變少，任務分割變得較細粒度化，這時某些線程可能集中在計算負載較重的區域上，導致負載分配不均衡。

2. **測資特性**：Mandelbrot Set 的計算負載高度依賴於各像素點的迭代次數。當線程數增加到 64 到 80 之間時，由於每個線程分配的行數減少，負載差異被放大，導致負載不平衡因子和標準差上升。隨著線程數進一步增加到 96 或更多，每個線程的分配負載變得極少，負載差異再次被平攤，負載不平衡因子和標準差因此下降。

3. **標準差排除極端現象的證據**：標準差圖的變化顯示出線程執行時間的分佈均勻性。在負載不平衡因子急劇上升的同時，標準差的顯著增長進一步證明了這並非單一線程執行特別快或特別久的極端現象，而是多數線程的執行時間分布出現了更大的波動，反映出多線程間負載分配不均的普遍性。

- 實驗三：Hybrid Version Scalability

- **固定總核心數(96)的配置**



在固定總計算資源（96個核心）的情況下，隨著MPI process數量的增加（同時thread數相應減少以維持總核心數不變），執行時間呈現出明顯的上升趨勢，特別是在process數量較大時，這種趨勢更為顯著。具體分析如下：

1. **低Process數區域（1-4 processes）：**
   - 在process數較少時（1-4個processes），執行時間維持在較低且穩定的水平（約1.5-1.7秒）
   - 這表明在少量process配置下，程式能夠有效利用OpenMP thread的並行性，並且MPI通信開銷較小
   - 此區域的效能表現最佳，特別是當process=1（純OpenMP）時，達到最低執行時間

2. **中等Process數區域（8-16 processes）：**
   - 當process數量增加到8-16個processes時，執行時間開始呈現緩慢但明顯的上升趨勢
   - 執行時間從約1.8秒上升到2.0秒左右
   - 這個階段的性能下降可能是由於：
     - MPI通信開銷開始變得顯著
     - 每個process內的thread數減少，影響了OpenMP的並行效率

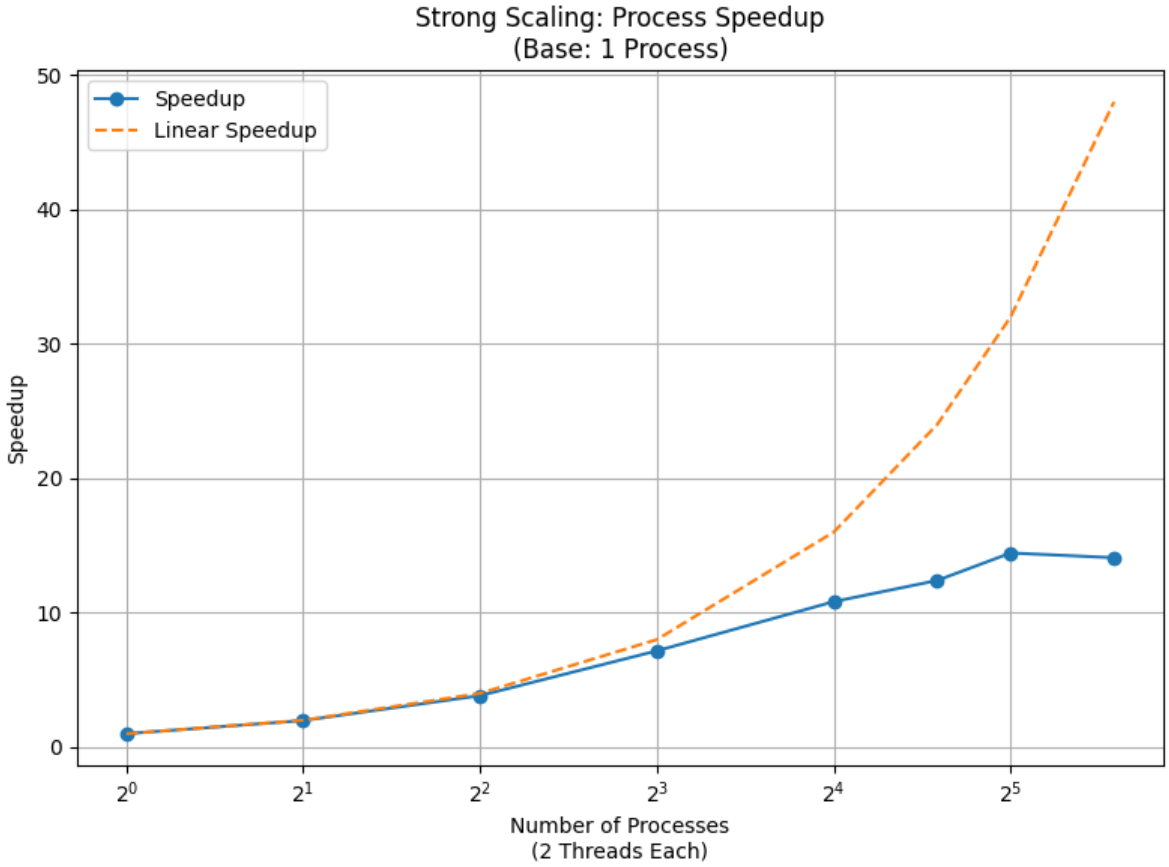
3. **高Process數區域（32-96 processes）：**
   - 在process數量達到32-96個processes時，執行時間急劇上升
   - 最大process數（96）時，執行時間達到約4.7秒，是最低執行時間的近3倍
   - 這種顯著的性能下降的原因應該是：
     - 大量的MPI通信開銷

        - 每個process只有很少的thread（甚至只有1個），無法充分利用多線程並行性
        - 數據分散和收集的開銷增加

    4. **整體趨勢分析：**

        - 在此hybrid並行實現中，MPI通信開銷對性能的影響遠大於OpenMP的thread管理開銷
        - 最佳配置應該是保持較少的process數量，讓每個process有足夠的thread來進行計算

- **Process擴展性測試配置**（固定thread數=2）



這張圖展示了在固定每個process使用2個threads的情況下，隨著process數量增加時的speedup表現。通過與理想的線性speedup（虛線）比較，我們可以詳細分析系統的擴展性：

1. **初始階段（1-4 processes）：**

    - 在較少process數量時（即1-4個processes），系統表現出接近理想的線性加速
    - Speedup幾乎與理想線重合，表明此階段的並行效率很高
    - 這個現象說明：
        - MPI通信開銷相對較小
        - 系統資源（如記憶體頻寬、快取等）足夠支援這種規模的並行計算
        - 負載平衡做得很好，各個process的工作量分配均勻

2. **中期階段（4-8 processes）：**

    - 在4到8個processes區間，speedup曲線開始與理想線性曲線產生偏離
    - 雖然仍在持續提升，但提升幅度開始減緩
    - 主要原因可能是：

        - MPI通信開銷開始變得明顯
        - 資源競爭開始影響性能
        - 任務分配的粒度變小，額外開銷的影響變得更顯著

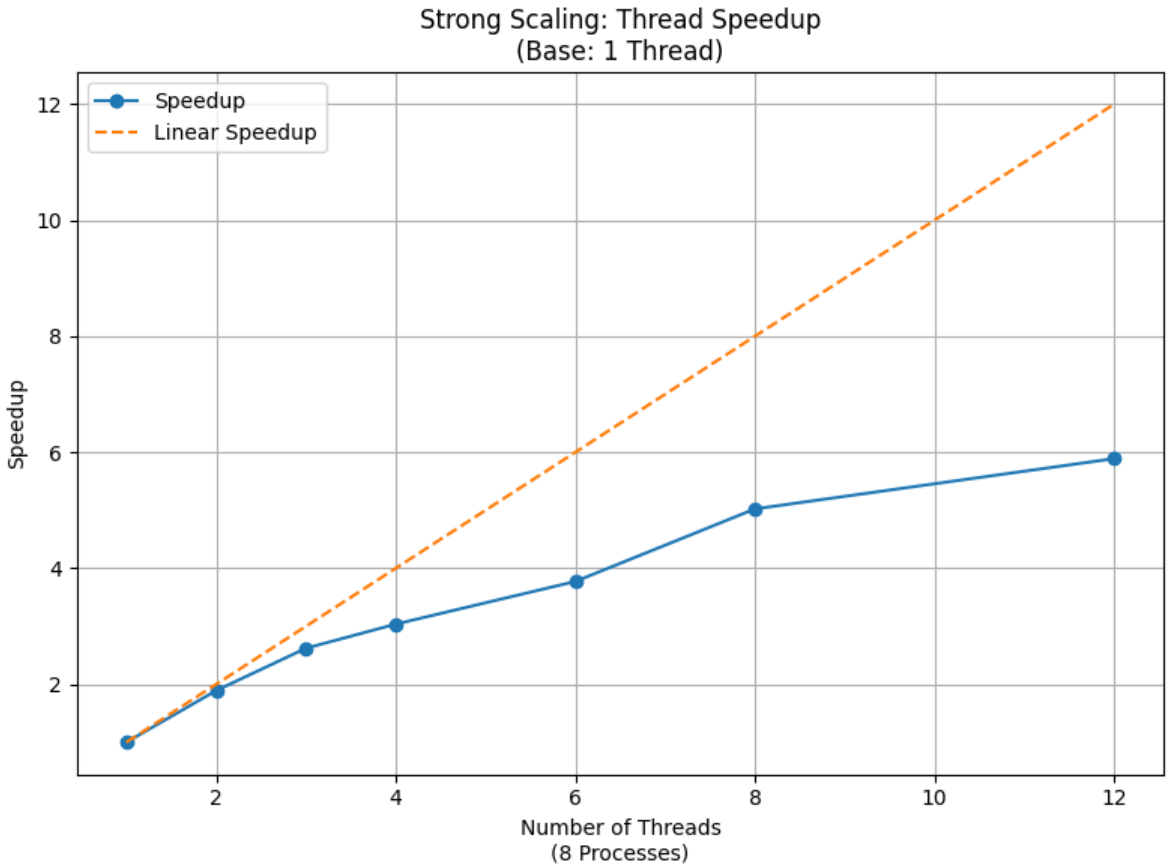
    3. **後期階段（8-32 processes）：**

        - 當process數量超過8個後，speedup曲線明顯偏離理想線性擴展
        - 在32個processes時，實際speedup約為15倍，遠低於理想的32倍
        - 到達32個processes後，speedup趨於平緩，甚至略有下降
        - 這種現象的可能原因：
            - MPI通信開銷成為主要瓶頸
            - 數據分散與收集的開銷顯著增加
            - 負載不平衡問題可能更加明顯
            - 系統資源達到飽和

    4. **擴展性分析：**

        - 系統展現出典型的strong scaling特徵：
            - 在小規模並行時效率高
            - 隨著並行度增加，效率逐漸下降
            - 最終達到效能瓶頸
        - 主要限制因素推測：
            - MPI進程間的通信開銷
            - 系統資源競爭
            - 任務分割的額外開銷

- **Thread擴展性測試配置**（固定process數=8）



這張圖展示了在固定8個MPI processes的情況下，隨著每個process中的OpenMP thread數量增加時的speedup表現。通過與理想的線性speedup（虛線）比較，我們可以詳細分析OpenMP層級的擴展性：

1. **初始階段（1-2 threads）：**
   - 在thread數較少時（1-2個threads），實際speedup幾乎完全符合理想線性擴展
   - 這表明：
     - OpenMP的並行開銷很小
     - 負載均衡效果良好
     - 記憶體頻寬和快取資源充足，能夠支撐有效的並行處理
2. **中期階段（2-6 threads）：**
   - 在2到6個threads區間，speedup曲線開始逐漸偏離理想線性曲線
   - 實際獲得的speedup雖然持續增加，但增長速率明顯放緩
   - 這種效能降低的可能原因：
     - 共享記憶體資源競爭開始顯現
     - OpenMP的thread管理開銷增加
     - 負載不平衡的影響開始加大
3. **後期階段（6-12 threads）：**
   - 當thread數超過6個後，speedup曲線與理想線性擴展的差距更加明顯
   - 在12個threads時，實際speedup約為6倍，遠低於理想的12倍
   - 曲線趨於平緩，表明增加更多threads的邊際效益很小
   - 效能瓶頸的主要原因可能是：
     - 記憶體頻寬飽和
     - 快取競爭加劇
     - 線程同步開銷增加

Optimization Strategies

基於前述的實驗結果與分析，我針對 Pthread 版本與 Hybrid 版本的程式，分別提出可能的優化策略如下：

Pthread 版本的優化策略

1. 動態任務排程（Dynamic Scheduling）

   問題分析：在 Pthread 版本的實驗中，當線程數增加到一定程度時，負載不平衡因子與標準差顯著增加，表示各線程的工作量開始不均勻。這可能是因為 Mandelbrot 集合的計算負載與像素點的位置高度相關，不同行（row）的計算複雜度可能差異很大。

   優化策略：將任務分配方式由原本的靜態交錯行（每個線程負責固定的行數）改為動態排程。可以在程式中實作一個任務隊列，線程在完成當前任務後，動態地從隊列中取得新的行進行計算。這樣可以確保當某些線程處理較複雜的區域而耗時較長時，其他線程不會閒置，從而平衡各線程的負載。

2. 重疊計算與 I/O（Overlapping Computation and I/O）

問題分析：在現有的實作中，計算完所有像素點後，才開始將結果寫入 PNG 檔案，導致計算與 I/O 操作串行化，未能充分利用系統資源。

優化策略：在 Mandelbrot 集合計算時，就開始同步地將計算結果寫入到影像檔案中，實現計算與 I/O 的重疊。這樣可以在計算的同時進行 I/O 操作，可能可以降低整體執行時間。

Hybrid 版本的優化策略

1. 優化 MPI 通信

   問題分析：在 Hybrid 版本的實驗中，隨著 MPI 進程數量的增加，執行時間顯著上升，這可能是由於 MPI 通信開銷的增加，特別是在數據收集與分發階段。

   優化策略：

   - 將多次小的通信操作合併為一次大的操作，減少 MPI 通信的頻率。
   - 使用 MPI 的非阻塞通信，讓計算與通信可以重疊進行，降低通信對計算的阻塞影響。
   - 根據網路拓撲結構，優化通信模式，減少通信延遲。

2. 調整 MPI 與 OpenMP 的並行比例

   問題分析：實驗結果顯示，在固定總核心數的情況下，增加 MPI 進程數反而導致執行時間增加，這是由於 MPI 通信開銷增加以及每個進程內的線程數減少，無法充分利用多線程並行性。

   優化策略：

   - 在可能的情況下，優先增加 OpenMP 線程數，減少 MPI 進程數，以降低通信開銷。
   - 通過實驗尋找最佳的 MPI 進程數與 OpenMP 線程數的組合，使得計算資源得到最佳利用。

3. 重疊計算與 I/O

   問題分析：在 Hybrid 版本中，計算完所有資料後才進行資料的收集與寫入，可能導致 I/O 操作與計算的串行化，未能充分利用資源。

   優化策略：實現計算與 I/O 的重疊，在計算過程中逐步將已完成的部分結果傳送給主進程，並開始寫入檔案。

4. 優化記憶體訪問與快取利用

   問題分析：記憶體訪問模式對性能有顯著影響。隨機訪問或跨 NUMA 節點的訪問會導致性能下降。

   優化策略：

   - 確保線程訪問的資料位於本地的記憶體節點，減少跨節點的訪問延遲。
   - 調整資料結構和訪問模式，提升快取命中率。

Experience & Conclusion

這次作業讓我學到了許多不同平行程式的實作方法，有使用不同 process 做平行的 MPI，有自己開 thread 做平行的方法，還有可以自動開 thread 做平行的 OpenMP，以及進一步用 SIMD 指令做平行化，讓我很驚訝的是，原來 MPI、OpenMP、SIMD 這些平行化的方法可以用在同一個問題上，寫成 hybrid 的程式碼，可以從原本效能

最差的版本 (大約 700 多秒)，提升到七八十秒左右，可見一個好的 implementation 真的對效能可以有很大幅度的改善，我覺得思考要如何優化程式並把想法實際實作出來的這個過程蠻有趣也蠻有成就感的，感謝老師與助教精心設計這份作業！