hw2 Mandelbrot Set

ID: 112062520 / Name: 戴維恩

Implementation

1. hw2a - original

要將 sequential code 做 threads 的平行化，首先要判定程式可以拆解、沒有相依性的部分。在這個作業裡，每個 pixel 的計算都是獨立的，因此有兩個方式可以做平行化：

- 以 row 為單位分配給 threads
- 以 pixel 為單位分配給 threads

這邊的做法是以 row 為單位分配給 threads 計算，根據指定 threads 數量 `NUM_THREADS` 創造 threads 並呼叫 `job` 函數分配工作。

```
for (int i=0; i<NUM_THREADS; i++)
{
    rc = pthread_create(&threads[i], NULL, job, (void *)&t_data[i]);
    if (rc) printf("error from creating thread %d\n", i);
}
```

`job` 函數裡面會去取用一個全域變數 `next_row` ，讓每個 thread 知道自己應該計算哪一個 row ，並接著呼叫 `cal_row_j` 進行第 j 個 row 的計算，這種 dynamic scheduling 的方式會讓工作的分配較為平均。另外，為了讓每個 thread 都取用到正確的 `next_row` 數值，會需要用到 `pthread` 的 `lock` 。

```
void *job(void* arg)
{
    int j;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        j = next_row++;
        pthread_mutex_unlock(&mutex);
        if (j >= height) break;
        cal_row_j(j);
    }
    pthread_exit(NULL);
}
```

`cal_row_j` 函數裡面相對單純，就只是把原本 sequential code 的第二層迴圈移過來，計算第 j 個 row 每一個 pixel 的答案。

```
void cal_row_j(int j)
{
    double y0 = j * ((upper - lower) / height) + lower;
    for (int i=0; i<width; ++i)
    {
        double x0 = i * ((right - left) / width) + left;
        int repeats = 0;
        double x = 0;
        double y = 0;
        double length_squared = 0;
        while (repeats < iters && length_squared < 4)
        {
            double temp = x * x - y * y + x0;
            y = 2 * x * y + y0;
            x = temp;
            length_squared = x * x + y * y;
            ++repeats;
        }
        image[j * width + i] = repeats;
    }
}
```

2. hw2a - optimized *8

由於每個 pixel 答案的計算都是獨立的，不會有相依性，因此其實可以使用 intrinsic 函數去優化 `job` 函數，用特殊架構達成每次平行計算多個 pixel 的答案。這次使用 `AVX-512` 指令集，也就是一次可以計算 512 bits，每個 `double` 是 64 bits，因此 intrinsic 函數一次可以處理 8 個 `double`。

優化過後的迴圈一次會處理 8 個 `double` ，因此計數器 `i` 一次 `+=8` ； `_mm512_set1_pd` 、 `_mm512_setzero_pd` 、 `_mm512_set_pd` 可以將數值裝進 register 中， `__mmask8` 則是用來判斷 8 個數值中是否有些不需進行運算，例如 `_mm512_mask_add_pd` 就是根據 mask 來判斷哪些位置的數值需要做加法。最後使用 `_mm512_storeu_pd` 將運算完的數值存回 memory 當中，並寫入相對應的圖片 pixel 位置。

值得特別注意的是，由於一次計算 8 個數值有可能會出現總 pixel 數不能整除的狀況，因此最後一輪可能會計算到多的 pixel ，這時候有多用一個變數 `valid_pixels` 判斷是否有這種情況發生，若有則不會將多算的部分寫進圖片中。

```
void cal_row_j_vec(int j)
{
    double y0 = j * ((upper - lower) / height) + lower;
    double x_unit = (right - left) / width;
    __m512d y0_vec = _mm512_set1_pd(y0);
    __m512d four_vec = _mm512_set1_pd(4);

    for (int i = 0; i < width; i += 8)
    {
        int valid_pixels = (width - i) < 8 ? (width - i) : 8;
        double repeats_mem[8] = {0};
        __m512d repeats = _mm512_setzero_pd();
        __m512d x0_vec = _mm512_set_pd(
```

```
                (i+7) * x_unit + left,
                (i+6) * x_unit + left,
                (i+5) * x_unit + left,
                (i+4) * x_unit + left,
                (i+3) * x_unit + left,
                (i+2) * x_unit + left,
                (i+1) * x_unit + left,
                i * x_unit + left
            );
            __m512d x_vec = _mm512_setzero_pd();
            __m512d y_vec = _mm512_setzero_pd();
            __m512d xx_vec = _mm512_setzero_pd();
            __m512d yy_vec = _mm512_setzero_pd();
            __m512d xy_vec = _mm512_setzero_pd();
            __m512d ls_vec = _mm512_setzero_pd();
            __mmask8 diverge_mask = 0xFF;

            int count = 0;
            while(diverge_mask && count < iters)
            {
                repeats = _mm512_mask_add_pd(repeats, diverge_mask, repeats, _mm512_s
                x_vec = _mm512_add_pd(_mm512_sub_pd(xx_vec, yy_vec), x0_vec);
                y_vec = _mm512_add_pd(_mm512_add_pd(xy_vec, xy_vec), y0_vec);
                xx_vec = _mm512_mul_pd(x_vec, x_vec);
                yy_vec = _mm512_mul_pd(y_vec, y_vec);
                xy_vec = _mm512_mul_pd(x_vec, y_vec);
                ls_vec = _mm512_add_pd(xx_vec, yy_vec);
                diverge_mask = _mm512_cmp_pd_mask(ls_vec, four_vec, _CMP_LT_OQ);
                count++;
            }
            _mm512_storeu_pd(repeats_mem, repeats);
            for (int r=0; r<valid_pixels; r++) image[j * width + i + r] = (int)repeat
        }
    }
```

3. hw2a - optimized *16

原本以為 `AVX-512` 的極限就是每次運算 8 個 `double` ，不過在看文件的時候無意間發現有支援 `__mmask16` 這種 mask 型別，可以一次進行 16 個數值的判斷，因此改成以下的形式，讓迴圈每一輪進行 16 個 pixel 的運算，這樣的優化可以讓大筆測資速度快上許多。
不過因為計算本身並不支援 16 個 `double` ，因此等於把每個變數拆成前半與後半，只是需要 mask 判斷的環節會去進行 16 個數值的平行判斷。以下變數有加上 `_f` 後標的為 `front` （前半），而 `_b` 則代表 `back` （後半）。

```
void cal_row_j_vec_16(int j)
{
    double y0 = j * ((upper - lower) / height) + lower;
    double x_unit = (right - left) / width;
    __m512d y0_vec = _mm512_set1_pd(y0);
    __m512d four_vec = _mm512_set1_pd(4);

    for (int i = 0; i < width; i += 16)
    {
        int valid_pixels = (width - i) < 16 ? (width - i) : 16;
```

```
        int repeats_mem[16] = {0};
        __m512i repeats = _mm512_setzero_si512();
        __m512d x0_vec_f = _mm512_set_pd(
            (i+7) * x_unit + left,
            (i+6) * x_unit + left,
            (i+5) * x_unit + left,
            (i+4) * x_unit + left,
            (i+3) * x_unit + left,
            (i+2) * x_unit + left,
            (i+1) * x_unit + left,
            i * x_unit + left
        );
        __m512d x0_vec_b = _mm512_set_pd(
            (i+15) * x_unit + left,
            (i+14) * x_unit + left,
            (i+13) * x_unit + left,
            (i+12) * x_unit + left,
            (i+11) * x_unit + left,
            (i+10) * x_unit + left,
            (i+9) * x_unit + left,
            (i+8) * x_unit + left
        );
        __m512d x_vec_f = _mm512_setzero_pd();
        __m512d x_vec_b = _mm512_setzero_pd();
        __m512d y_vec_f = _mm512_setzero_pd();
        __m512d y_vec_b = _mm512_setzero_pd();
        __m512d xx_vec_f = _mm512_setzero_pd();
        __m512d xx_vec_b = _mm512_setzero_pd();
        __m512d yy_vec_f = _mm512_setzero_pd();
        __m512d yy_vec_b = _mm512_setzero_pd();
        __m512d xy_vec_f = _mm512_setzero_pd();
        __m512d xy_vec_b = _mm512_setzero_pd();
        __m512d ls_vec_f = _mm512_setzero_pd();
        __m512d ls_vec_b = _mm512_setzero_pd();
        __mmask16 diverge_mask = 0xFFFF;

        int count = 0;
        while(diverge_mask && count < iters)
        {
            repeats = _mm512_mask_add_epi32(repeats, diverge_mask, repeats, _mm5
            x_vec_f = _mm512_add_pd(_mm512_sub_pd(xx_vec_f, yy_vec_f), x0_vec_f)
            x_vec_b = _mm512_add_pd(_mm512_sub_pd(xx_vec_b, yy_vec_b), x0_vec_b)
            y_vec_f = _mm512_add_pd(_mm512_add_pd(xy_vec_f, xy_vec_f), y0_vec);
            y_vec_b = _mm512_add_pd(_mm512_add_pd(xy_vec_b, xy_vec_b), y0_vec);
            xx_vec_f = _mm512_mul_pd(x_vec_f, x_vec_f);
            xx_vec_b = _mm512_mul_pd(x_vec_b, x_vec_b);
            yy_vec_f = _mm512_mul_pd(y_vec_f, y_vec_f);
            yy_vec_b = _mm512_mul_pd(y_vec_b, y_vec_b);
            xy_vec_f = _mm512_mul_pd(x_vec_f, y_vec_f);
            xy_vec_b = _mm512_mul_pd(x_vec_b, y_vec_b);
            ls_vec_f = _mm512_add_pd(xx_vec_f, yy_vec_f);
            ls_vec_b = _mm512_add_pd(xx_vec_b, yy_vec_b);
            diverge_mask = _mm512_cmp_pd_mask(ls_vec_f, four_vec, _CMP_LT_OQ) |
            count++;
        }
        _mm512_storeu_si512(repeats_mem, repeats);
        for (int r=0; r<valid_pixels; r++) image[j * width + i + r] = (int)repea
```

```
    }
}
```

4. hw2b

作業 hybrid 的部分要將 `pthread` 改為 `OpenMP`，並使用 `MPI` 進行 process 的平行化。

```
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

程式架構上大致不變，`OpenMP` 可以使用 `#pragma omp parallel for` 自動排程；這邊安排每個 rank 不會算到連續的 row，例如若總共有 3 個 rank，rank 0 會被排到 row 0 / 3 / 6 ...，而 rank 1 會被排到 row 1 / 4 / 7 ...，以此類推。
這樣安排的原因是，根據觀察，通常要算比較久的那些 pixel （圖上顏色較深的部分）會聚集在同一塊，也就是說如果一個 row 的計算量很大，通常他的下一個 row 計算量也會很大。為了平均分配工作，希望每個 process 所計算的 row 可以錯開。

```
#pragma omp parallel for schedule(dynamic)
    for (int j=rank; j<height; j+=size) cal_row_j_vec_16(j);
```

其他的部分維持與 `pthread` 版本相同，最後使用 `MPI_Reduce` 收集每個 process 的運算結果。

```
MPI_Reduce(draw_image, image, height * width, MPI_INT, MPI_BOR, 0, MPI_COMM_WORLI
```

Experiment & Analysis

1. Methodology

System Spec: 實驗使用課程提供的 `qct server`。
Performance Metrics:

- Testcase: 使用助教提供測資中的 `strict06.txt` 、 `strict33.txt` 做實驗（圖表上註明）。
- Method: 僅計算從 `MPI_Init` 後至 `MPI_Finalize` 之間的執行時間，使用 `nsys / nvtx` 分割 `IO time / Calculation time / Communication time`，程式中使用到 `write_png` 處分類為 `IO time`，使用到 `MPI_Reduce` 處分類為 `Communication time`，剩餘時間為計算時間 `Calculation time`。

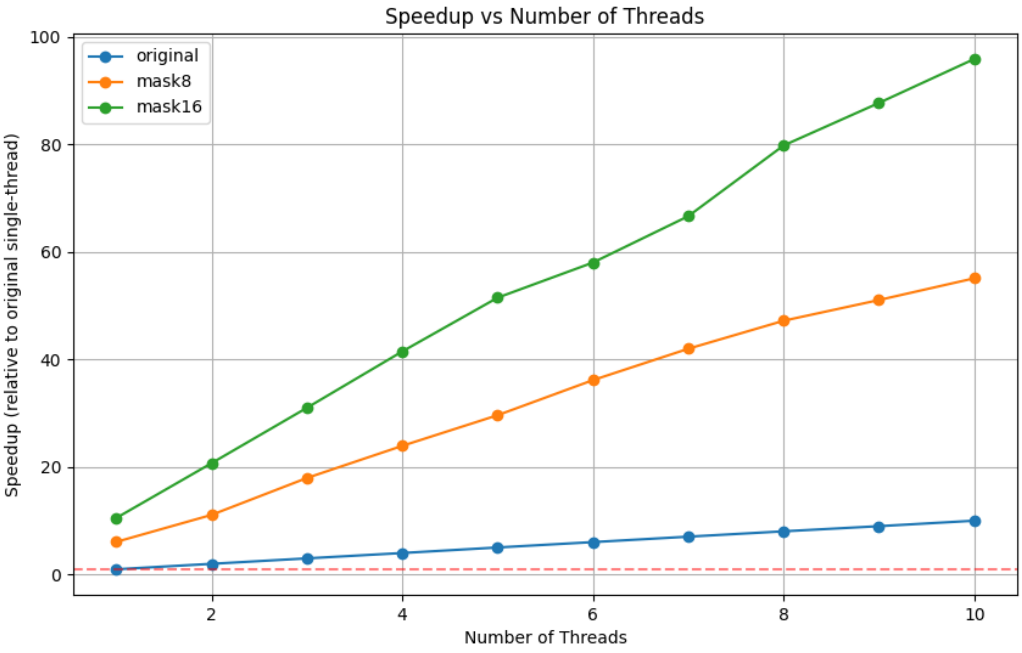
2. Plots: hw2a

Description

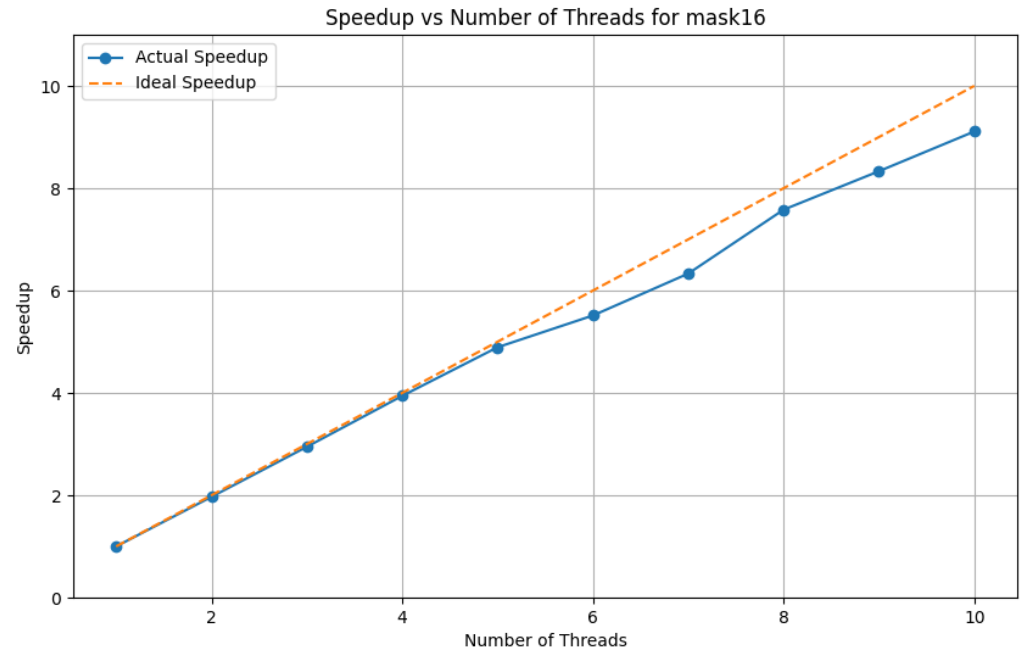
由於 `pthread` 總共寫了三個版本，我用第一版且 thread 數量為 1 的執行速度當作基準，測量三個版本的 speedup 比較圖；並用 performance 最佳的 mask16 版本計算 speedup，並分別用 thread 數量為 6 以及 12 測量每個 thread 的計算時間，用以衡量 load balance performance。

**以下實驗使用測資 `strict06.txt`*

3 Versions Comparison - Speedup

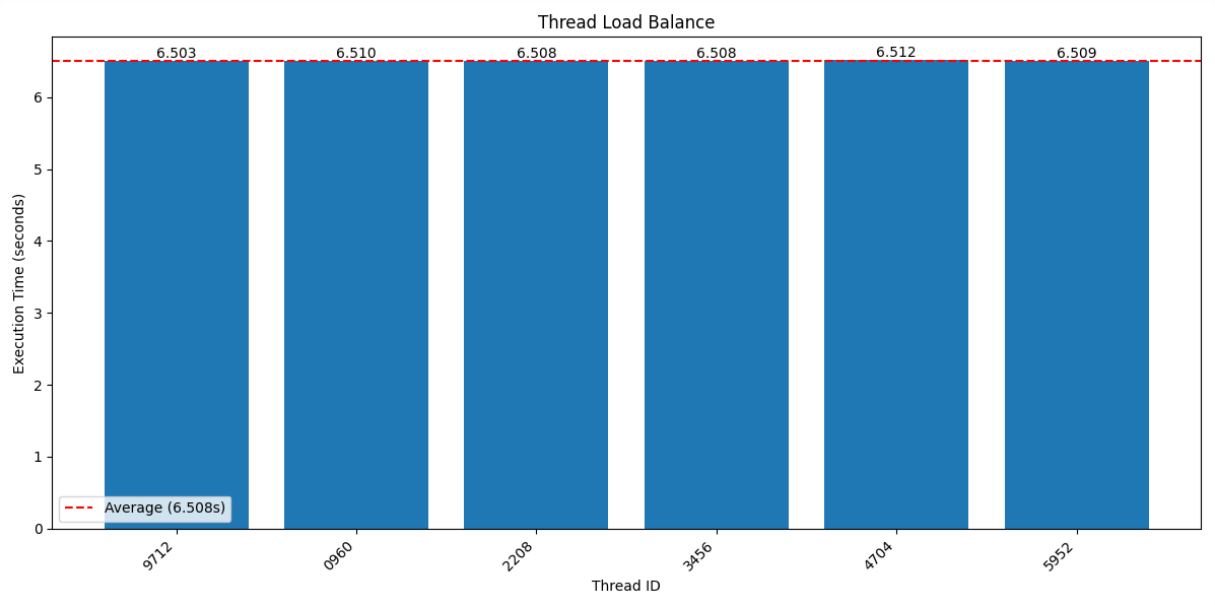


Mask16 Speedup

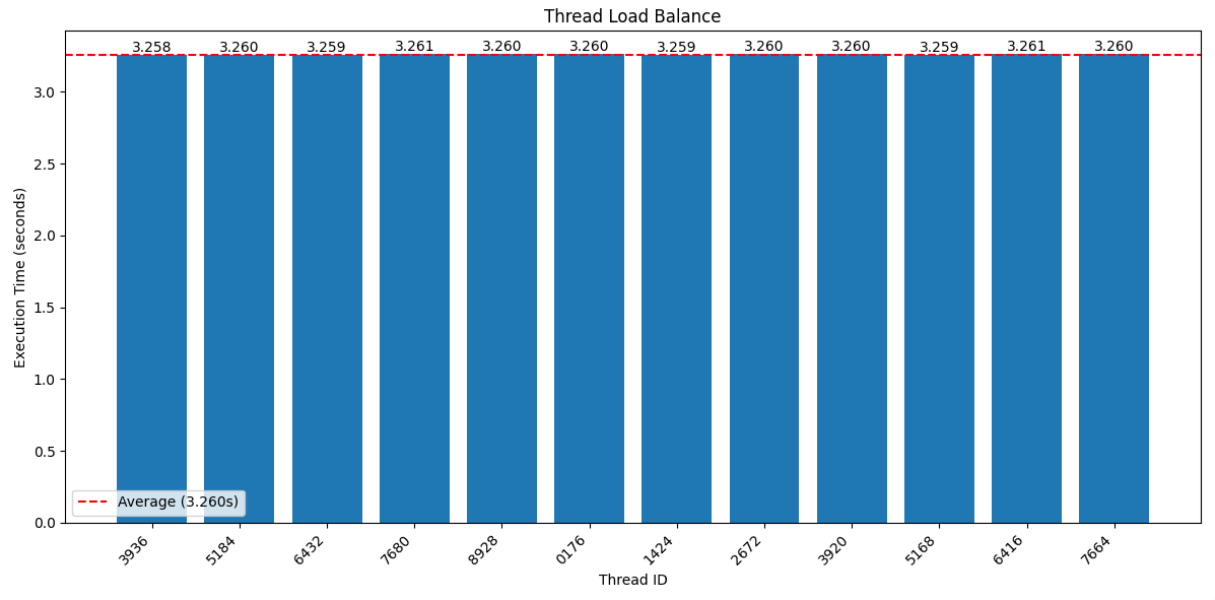


*以下實驗使用測資 `strict33.txt`

Mask16 Load Balance - 6 threads



Mask16 Load Balance - 12 threads



Discussion

首先關於三個不同版本比較的部分，可以發現以下幾個現象：

- mask8 和 mask16 相較於原始版本有顯著的效能提升
- mask16 在 12 個 threads 時達到近 3 倍的加速
- 隨著 threads 數量增加，mask8 和 mask16 的效能差異更為明顯，顯示更好的 vetorized 能力運用

單獨觀察 mask16 版本的 speedup 則可以發現：

- 加速曲線在 6-8 個 threads 時展現很好的的 scalability

- 超過 8 個 threads 後，加速增益趨緩，這裡列出幾個我認為可能的原因：
  - 記憶體頻寬限制
  - threads 管理的額外開銷
  - CPU 資源競爭

而 threads load balance 的部分，可以看到不管是 6 或是 12 個 threads 的實驗結果皆顯示相對平衡的工作分配，大多數 threads 有極為相近的執行時間。

4. Plots: hw2b

Description

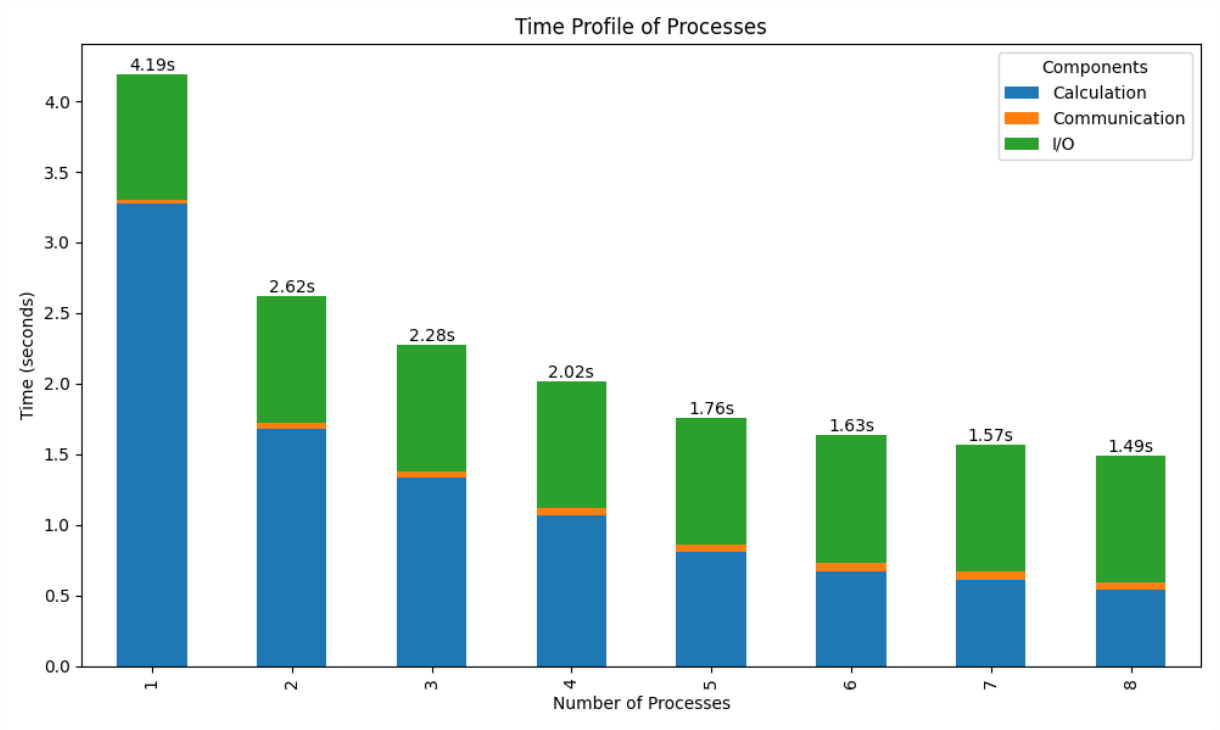
作業 hybrid 的部分，我首先測量同測資不同 process 數量的執行時間、計算 speedup，speedup 部分分為 calculation time speedup 以及 overall speedup，用來評估單看演算法優化的程式效能。接著，為了要評估每個 rank 所被分配到的工作量是否平均，分別以 rank size 為 3 / 5 / 8 測量每個 rank 的執行時間。

***以下實驗使用測資 `strict33.txt`**

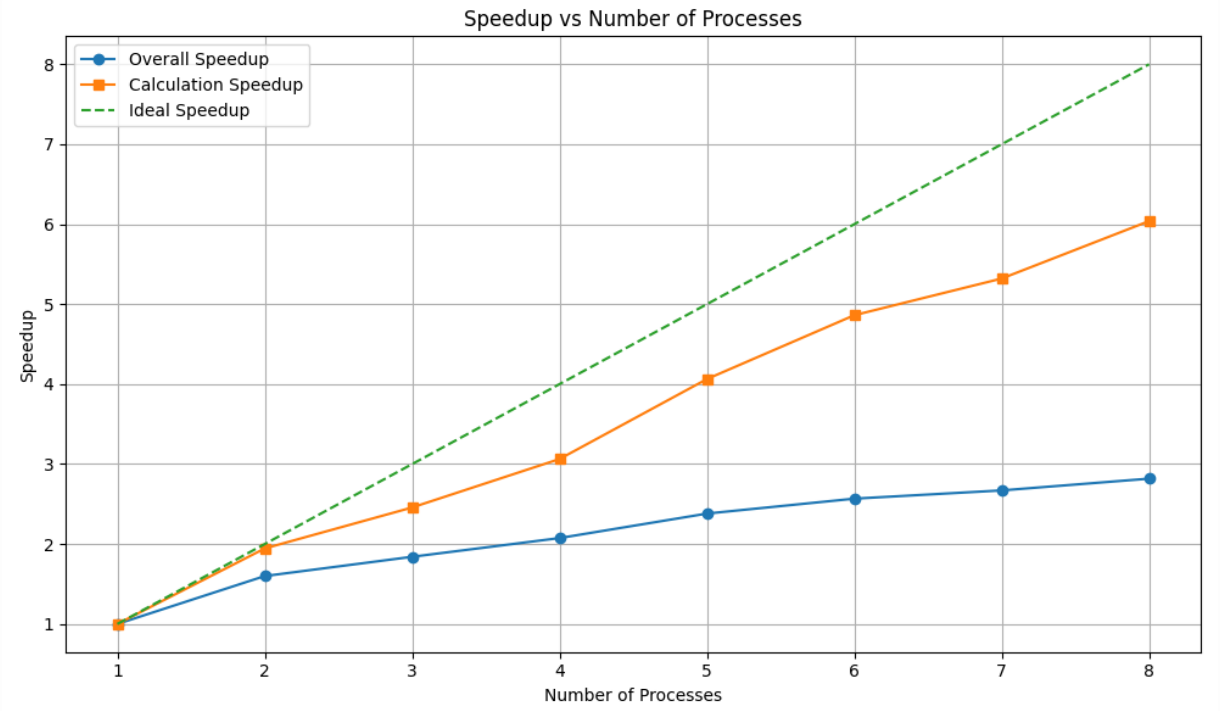
Time Profile Table

Number of Processes	Total Time (s)	Calculation (s)	Communication (s)	I/O (s)
1	4.19	3.30	0.02	0.87
2	2.62	1.70	0.05	0.88
3	2.28	1.35	0.06	0.86
4	2.02	1.05	0.10	0.89
5	1.76	0.80	0.09	0.85
6	1.63	0.65	0.11	0.88
7	1.57	0.60	0.10	0.86
8	1.49	0.55	0.07	0.87

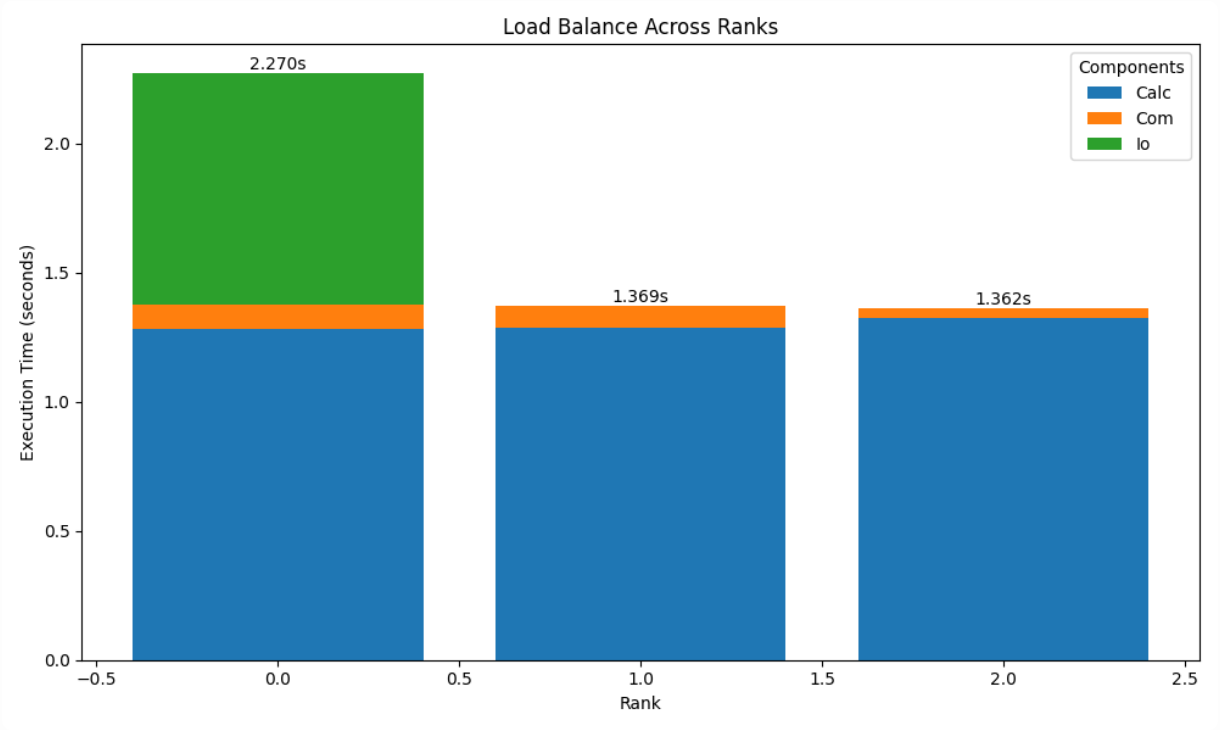
Time Profile Graph



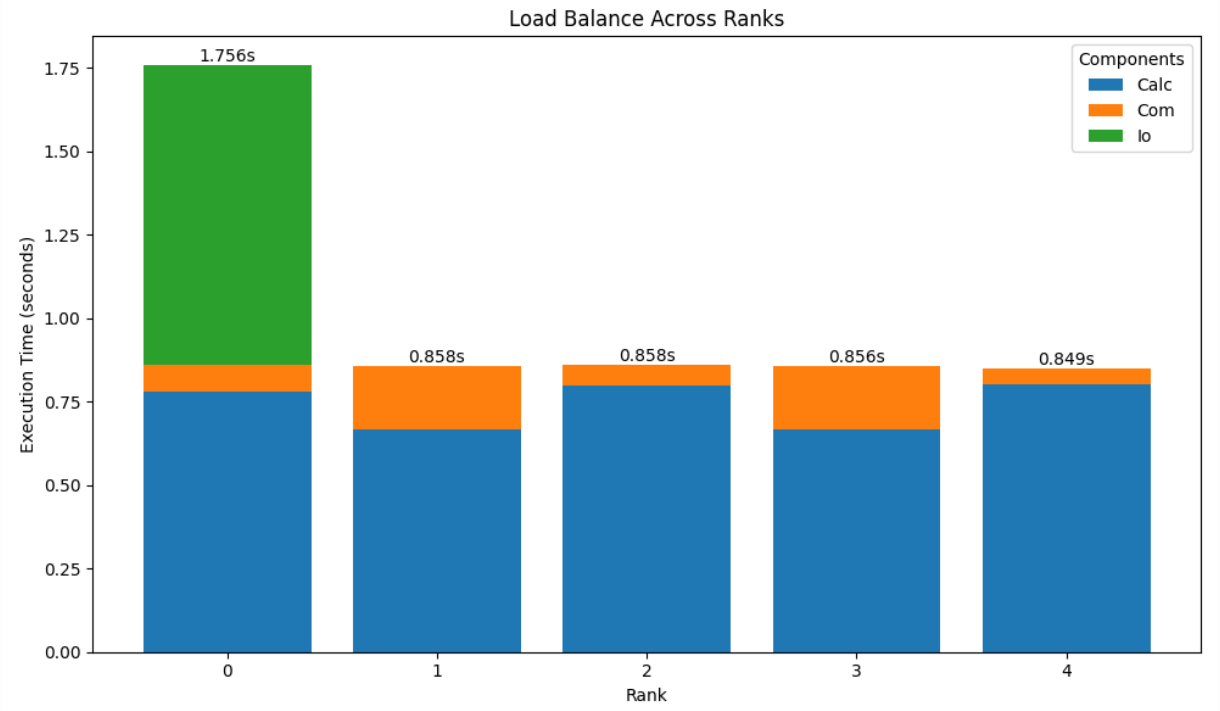
Speedup Graph



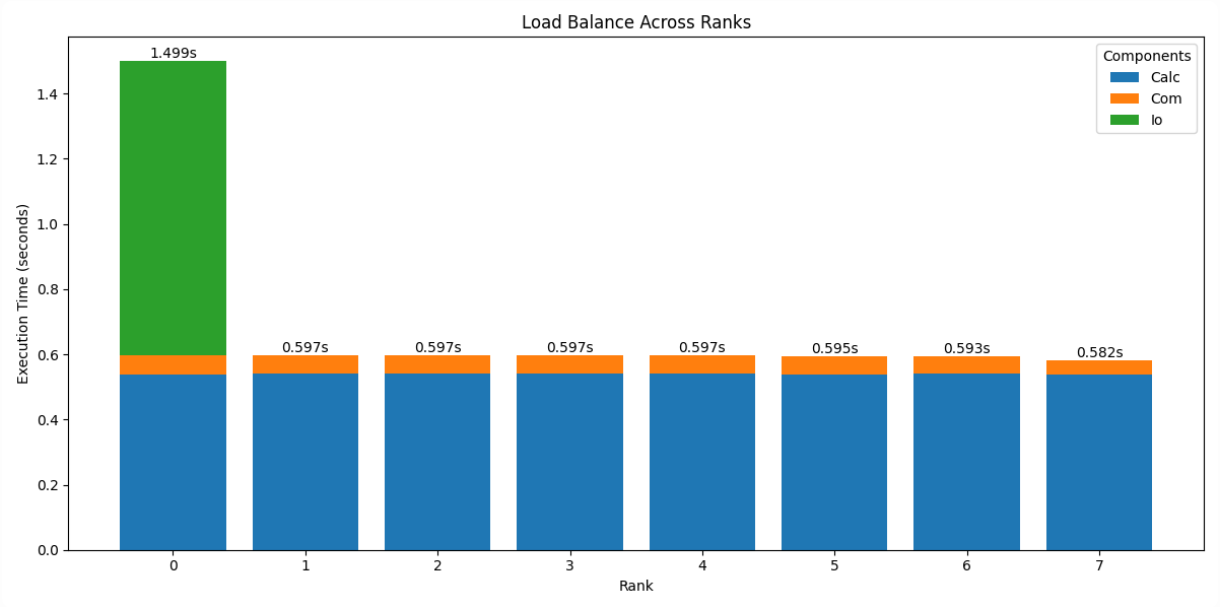
Rank Load Balance - 3 Ranks



Rank Load Balance - 5 Ranks



Rank Load Balance - 8 Ranks



Discussion

Time Profile 的部分可以看出以下幾點：

- calculation time 隨著 process 數增加而顯著減少
- communication time 維持相對穩定（0.02-0.11秒）
- I/O time 保持一致（約 0.87 秒），幾乎不受 process 數量影響
- 總執行時間曲線在 6-7 個 process 後趨於平緩

Speedup 的部分：

- Calculation speedup 在 4 個 process 前呈現近線性的擴展
- Overall speedup 受限於固定的 I/O 開銷
- 8 個 process 時達到最大約 2.8 倍的 overall speedup
- Calculation speedup 與 overall speedup 的差距隨 process 數增加而擴大

而在 load balance 的實驗中可以看見，普遍情況下每個 rank 的 calculation time 差異並不大，主要的瓶頸是 rank 0 需要在所有計算完成後進行圖片的 I/O。在特定情況下，以實驗結果來說就是 rank size 為 5 時，可以看到目前的工作分配方式還是有所限制，會讓每個 rank 的計算量稍顯不平均，其他先做完的 rank 會需要等候所有人做完，才可以進行最後的 communication，這邊是主要可以再更優化的地方。

Conclusion & Possible Improvements

這次作業實作了 Mandelbrot Set 的平行化計算，從單純的 `pthread` 到結合 `MPI` 與 `OpenMP` 的 hybrid 版本，過程中遇到許多挑戰與收穫。

最具挑戰性的部分在於優化 process 之間的溝通效率。實驗結果顯示，相同測資在 pthread 版本與 hybrid 版本之間存在顯著的執行時間差異，這凸顯了 process management 與工作量分配的重要

性。在實作過程中，曾嘗試使用 process pool 實現 process 的 dynamic scheduling，但發現這種方法需要頻繁地與 Master process 進行溝通，反而因溝通 overhead 造成整體效能下降。

經過多次嘗試與改進，最終採用了較為靜態的工作分配方式，讓每個 process 負責間隔的 row 進行計算。這種方法雖然減少了溝通開銷，但仍有改進空間。

目前想到可能的改進方向主要是 process dynamic scheduling 的優化：

- 實作 Non-blocking Communication
- 採用更有效的資料分割策略，減少 process 溝通次數

這次實作經驗顯示，在平行程式設計中，不僅要考慮計算效能的優化，更要注重 process 間的溝通效率。未來若要進一步提升效能，需要在工作分配的靈活性和溝通開銷之間取得更好的平衡。同時，也應該考慮根據不同規模的問題採用不同的平行化策略，以達到最佳效果。