HW2 Mandelbrot Set

student ID: 112062619 **name:** 陳航希

Implementation

hw2a – 基礎 pthread 版本(使用 mutex 動態分配 rows)

在最初的 sequential 版本中，整個程式的主要流程會像下面這樣:

1. 外層使用兩層迴圈，逐一掃過影像的每個 pixel
2. 對每個 pixel 的座標 (x,y)，計算其是否屬於 Mandelbrot Set
3. 將迭代結果存進影像image，最後輸出成 PNG 檔

由於 每個 pixel 的計算彼此完全獨立的關係，所以這個問題非常適合做平行化。我們可以把每一個row或每個pixel分配給不同的thread處理。

這裡採取 以 row 為單位的平行化，讓每個 thread 動態抓取下一個 row 來計算。

1. 創建多個執行緒

```
    for (int i=0; i<ncpus; ++i) {
        args[i].id = i;
        args[i].nthreads = ncpus;
        args[i].shared = &shared;
        pthread_create(&threads[i], NULL, worker, &args[i]);
    }
```

pthread_create 會建立多個 threads，每個 thread 會進入 worker() 工作。

2. 動態分配工作:共享變數+mutex 鎖

```
pthread_mutex_t row_mutex = PTHREAD_MUTEX_INITIALIZER;
int next_row = 0;
```

next_row 代表下一個要被計算的 row 編號，所有 threads 都會共用這個變數。

為了避免多個 threads 同時修改導致race condition，必須用mutex鎖 保護這段程式。

```
    while (1) {
        pthread_mutex_lock(&row_mutex);
        int j = next_row++;
        pthread_mutex_unlock(&row_mutex);
```

每個 thread 取到自己要計算的 row 後，就可以放心離開鎖區、進行獨立運算。

但用這樣的方式每次取 row 都要 lock/unlock，thread 數一多會造成同步瓶頸。所以有下面的優化。

剩下的計算工作跟sequantial版本的相同，最後用pthread_join,拿回資料。

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

hw2a – 優化 pthread 版本(atomic + chunk + SSE2向量化)

1. 使用 atomic 取代 mutex

這裡改用 C++ 的 std::atomic<int> 進行無鎖操作:

```
std::atomic<int> next_row(0);
while (1) {
        int j_start = next_row.fetch_add(CHUNK);
        if (j_start >= s->height) break;

        int j_end = j_start + CHUNK;
        if (j_end > s->height) j_end = s->height;
```

這樣 threads 之間可以安全地同時操作 next_row，不再需要mutex，降低同步開銷。

2. 加入CHUNK想法

一次分配多行(例如 2 行)給同一個 thread:

```
#define CHUNK 2
for (int j = j_start; j < j_start + CHUNK; ++j)
    compute_row_sse2(j, s);
```

這樣 threads 之間切換工作更少，也能在 workload 不均時自動達成動態負載平衡。

3. SSE2 向量化運算

因為每個pixel的結果互不相關且SSE2 指令集提供 128-bit 的向量暫存器，我們可以一次同時計算兩個pixels(2 doubles)。

在實作上，我將每一列的像素資料以兩個為一組的形式進行批次運算。

首先使用 _mm_setr_pd() 將兩個相鄰像素的初始座標組成向量，再以 _mm_add_pd()、_mm_mul_pd()、_mm_sub_pd() 等指令完成實部與虛部的平方、加法與乘法運算。

藉由維護一個 active_mask，可以讓每個 lane(對應到一個 pixel)在超出邊界時停止計算，而仍在集合內的 pixel 可以繼續進行迭代。最後得到結果。

```
void compute_row_sse2(int j, SharedData *s) {
    double y0 = j * s->dy + s->lower;
    __m128d y0_vec = _mm_set1_pd(y0);
    __m128d four_vec = _mm_set1_pd(4.0);

    int base = j*s->width;
    int* row_out = s->image+base;

    for (int i=0; i<s->width; i+=2) {
        int valid = (s->width - i < 2) ? (s->width - i) : 2;

        __m128d x0 = _mm_setr_pd(i * s->dx + s->left,
```

```
                                (i + 1) * s->dx + s->left);
        __m128d x = _mm_setzero_pd();
        __m128d y = _mm_setzero_pd();

        int cnt0 = 0, cnt1 = 0;
        int active_mask = (valid == 2) ? 3 : 1; // lane 0/1 是否仍需計算

        for (int k = 0; k < s->iters && active_mask; ++k) {
            __m128d x2 = _mm_mul_pd(x, x);
            __m128d y2 = _mm_mul_pd(y, y);
            __m128d xy = _mm_mul_pd(x, y);

            __m128d new_x = _mm_add_pd(_mm_sub_pd(x2, y2), x0);
            __m128d new_y = _mm_add_pd(_mm_add_pd(xy, xy), y0_vec);

            __m128d len2 = _mm_add_pd(_mm_mul_pd(new_x, new_x),
                                      _mm_mul_pd(new_y, new_y));
            cnt0 += (active_mask & 1) ? 1 : 0;
            cnt1 += (active_mask & 2) ? 1 : 0;
            int stay = _mm_movemask_pd(_mm_cmplt_pd(len2, four_vec));
            active_mask &= stay;
            x = new_x;
            y = new_y;
        }
        row_out[i] = cnt0;
        if (valid == 2) row_out[i + 1] = cnt1;
    }
}
```

hw2b – Openmp + MPI

1. MPI 層方面

因為mandable set在每一個row的計算負載會有很大的差距(如在邊界區域可能很快就算完,但越靠近中心需要的時間越久), 所以這裡使用了

```
rows[lr] = rank + lr * world;
```

讓每個rank不會被分配到連續的row,所以如果共有4個rank,就會是 rank0:分配到row: 0/4/8…, rank1:分配到row:1/5/9… 這樣能讓各節點分配到的區域負載更平均,
特別是影像中央部分計算較重的問題可被分攤。

2. OpenMP 層

在每個 rank 內, 再使用多執行緒處理寬度方向的像素:利用 omp_lock_t 動態控制每個 thread 取得的區塊。

```
omp_set_lock(&lk);
x0 = sched_x;
```

```
cnt = OMP_BLOCK_W;       // 每次取 64 pixels
sched_x += cnt;
omp_unset_lock(&lk);
compute_block_sse2(&S, j_global, x0, cnt, row_out);
```

這樣 threads 間可以動態平衡工作量，不會因為特定列區域較重而閒置。

3. SSE2 vectorization

這個部分跟pthread的版本相同，以兩個像素為單位運算。

4. MPI_Gatherv

所有 MPI ranks 完成後，透過MPI_Gatherv 將資料傳回 rank 0，依據原始行順序重新拼回完整影像並輸出 PNG。

Experiment

setup:

System Spec: 使用課堂提供的機器(apollo)。

Performance Metrics:

實驗數據**:**

針對speed up, scalability的實驗使用 strict06.txt

針對load balancing的實驗使用 strict32.txt

Metrics:

實驗透過 MPI_Wtime() 手動標記程式中各個區段，將總執行時間細分為以下三類：

computation time , **communication time, io time**:

Computation Time：用在計算mandable set的時間，hw2a, hw2b都有。

Communication Time：紀錄mpi通訊的時間，只會出現在hw2b中

IO time: 程式中使用到 write_png的部分, hw2a,hw2b都會有

1. hw2a - pthread

因為pthread版本共有兩種，以下圖表分別記錄了兩種版本的**Thread**數與執行時間比較跟 **Strong scalability speedup**比較

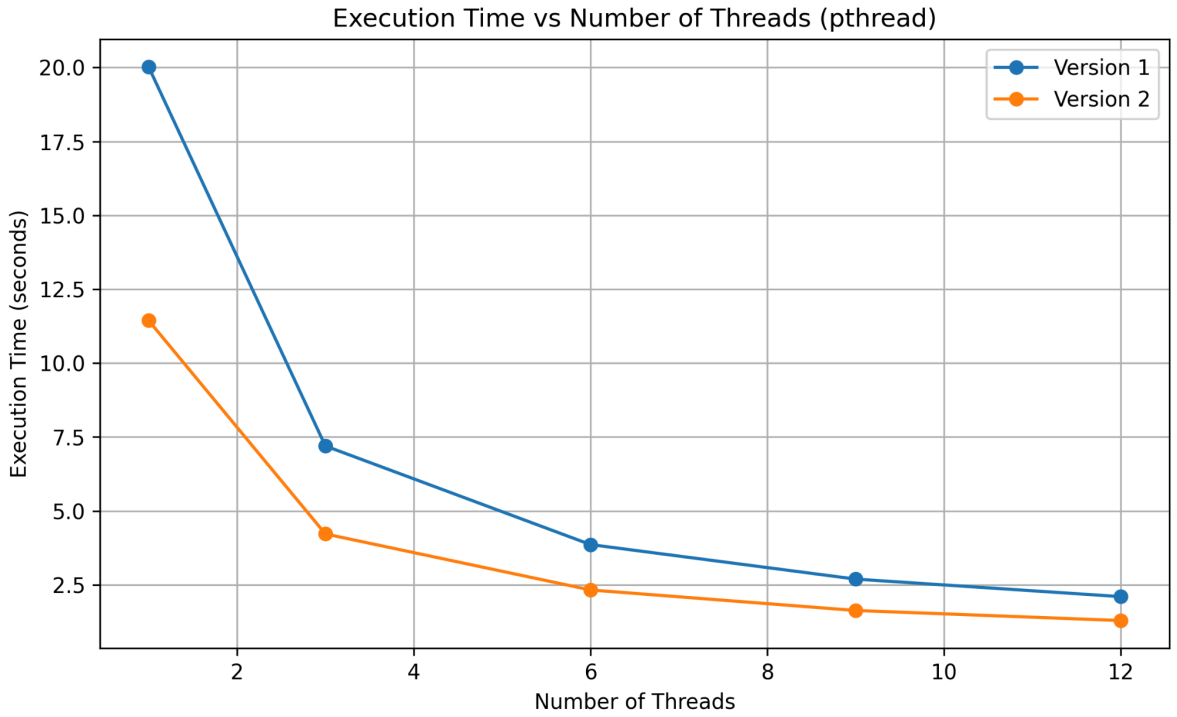
Thread數與執行時間 這部分記錄了同一支程式在不同的thread數下會有的execution time。

Strong scalability 以各自在thread數=1的情況下看thread數的增加會有的速度提升。

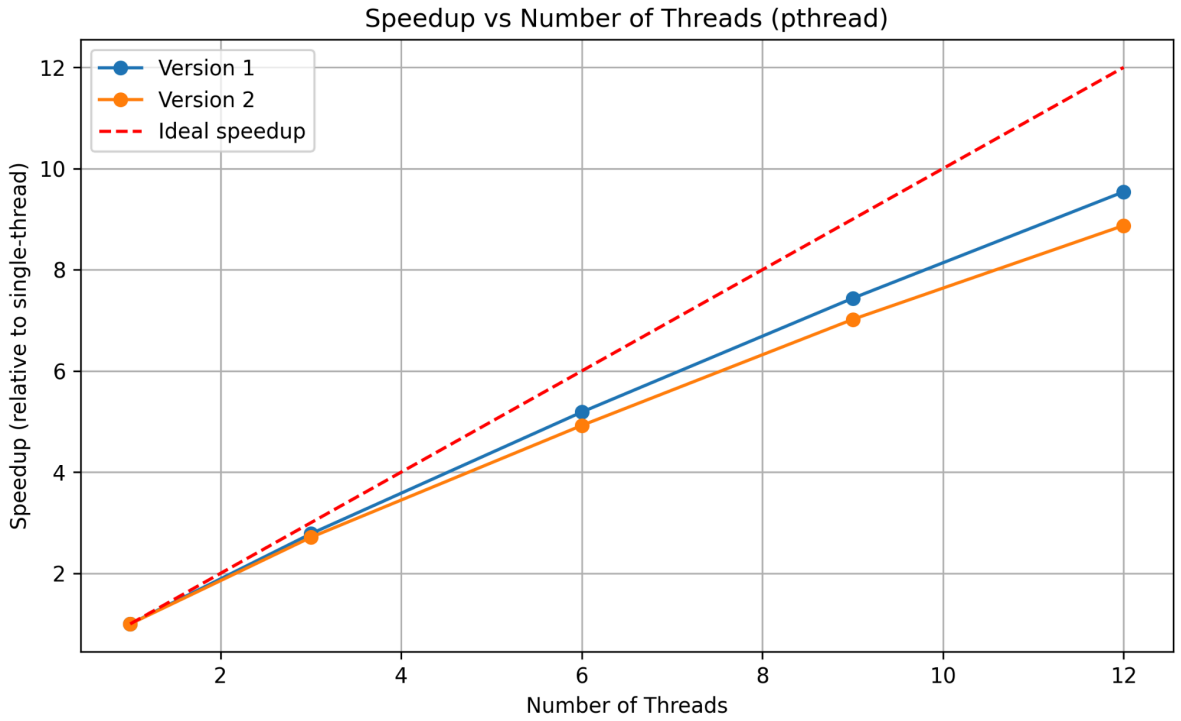
在load balancing實驗的部分，為了要評估每個 thread 所被分配到的工作量是否平均，分別以 thread大小 為 6, 12 測量每個 rank 的執行時間。

以下實驗使用測資 strict06.txt

Thread數與執行時間比較

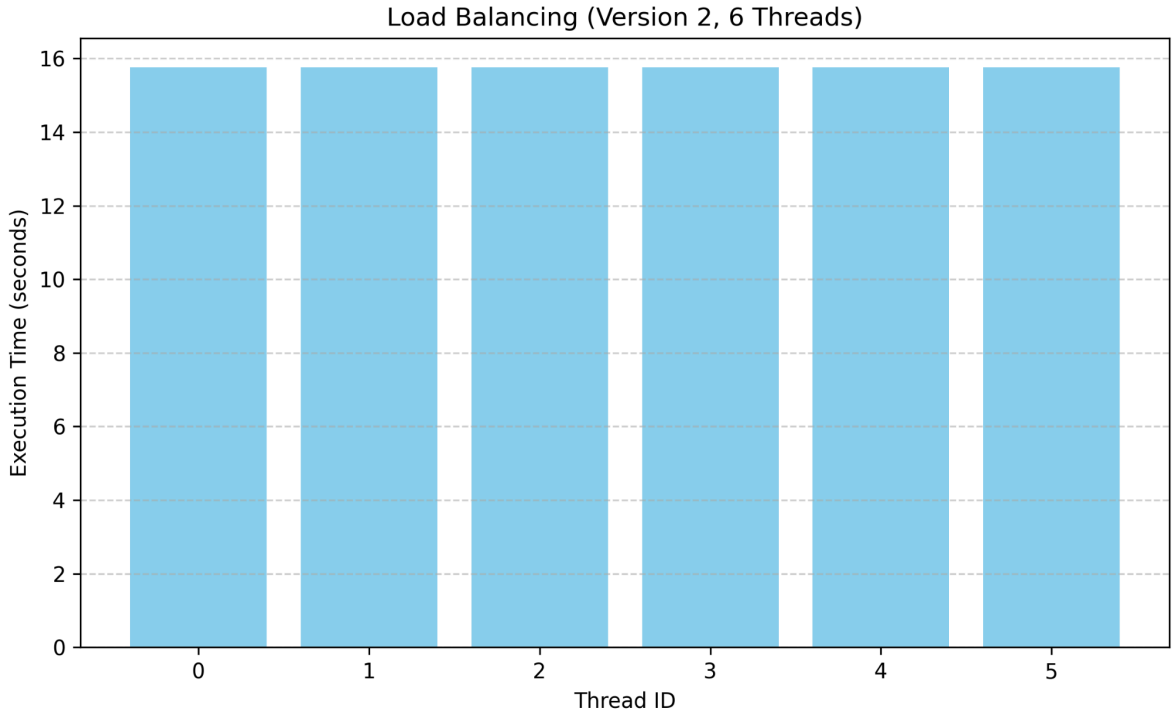


Strong scalability speedup比較

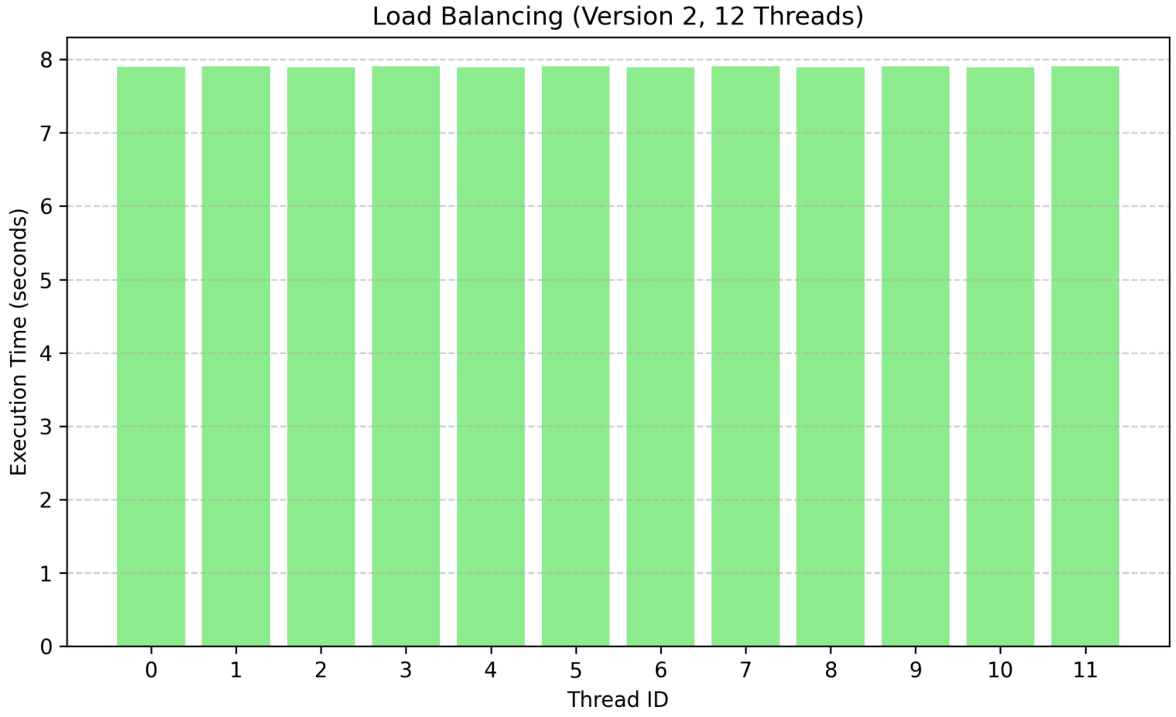


以下實驗使用測資 strict32.txt

版本二load balancing - 6 threads



版本二load balancing - 12 threads



Discussion:

這次實驗比較了兩個 pthread 版本:

版本一是基礎實作，用 mutex 控制 row 的動態分配

版本二則改成 atomic+chunk 分配，並加入 SSE2 向量化運算 來加速每個 pixel 的計算。

結果顯示，版本二的執行時間明顯較短，整體效能提升顯著。

原因包括:

- 使用 stdatomic 取代 mutex，消除了鎖競爭帶來的同步延遲
- 一次分配多行 (chunk) 給 thread，減少頻繁搶鎖與排程
- SSE2 向量化可同時計算兩個 pixel，單執行緒吞吐量接近兩倍

Load Balancing 的部分可以看到，不論在 6 或 12 threads 下，每個 thread 的執行時間都幾乎相同，代表動態分配的效果很好。

Strong Scalability的部分，雖然版本二的絕對速度更快，但版本一的 speedup 比例稍微更接近理想線。

這可能是因為版本二在單執行緒就已經經過 SSE2 優化，導致 baseline 較快，使得 speedup 比例看起來略低。另外 chunk 分配的粒度較大，可能讓最後幾個 thread 稍微閒置。

2. hw2b - openmp + mpi

在hybrid的部分，我分別記錄了不同**process**數下**,**花的時間與各個部分時間占比 跟**total**時間與**Strong scalability speedup**的比較

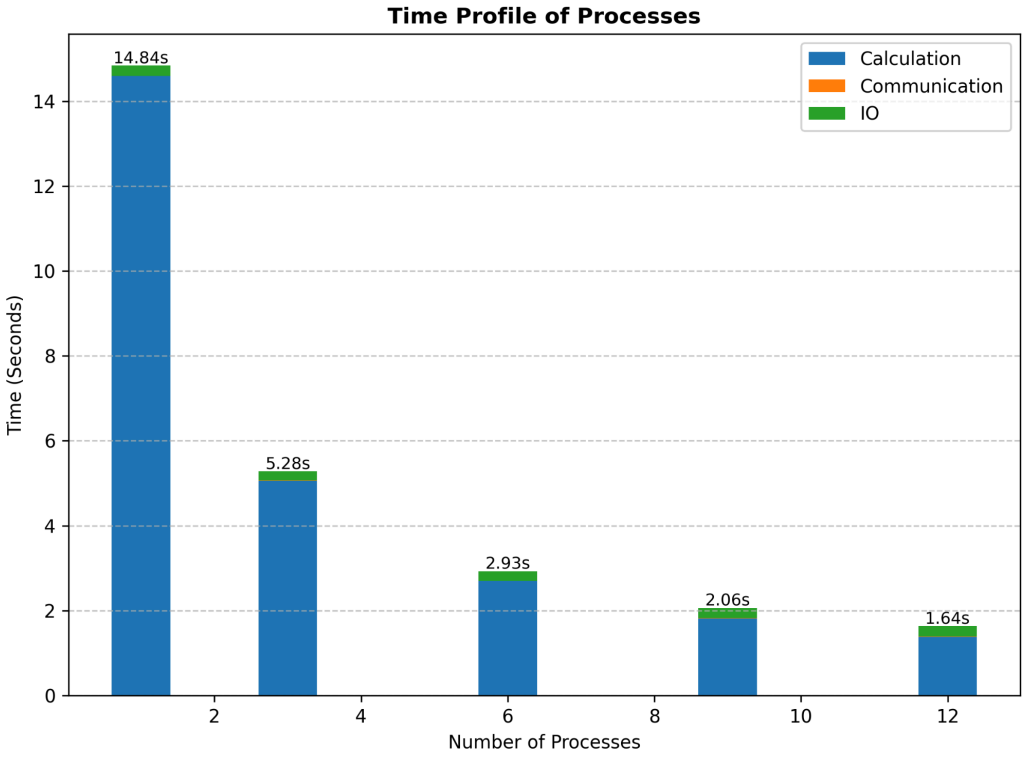
在第一個部分測量了同測資不同 process 數量的執行時間。

在第二個speedup的部分記錄了 calculation time speedup 以及 overall speedup，用來評估單看演算法優化的程式效能。

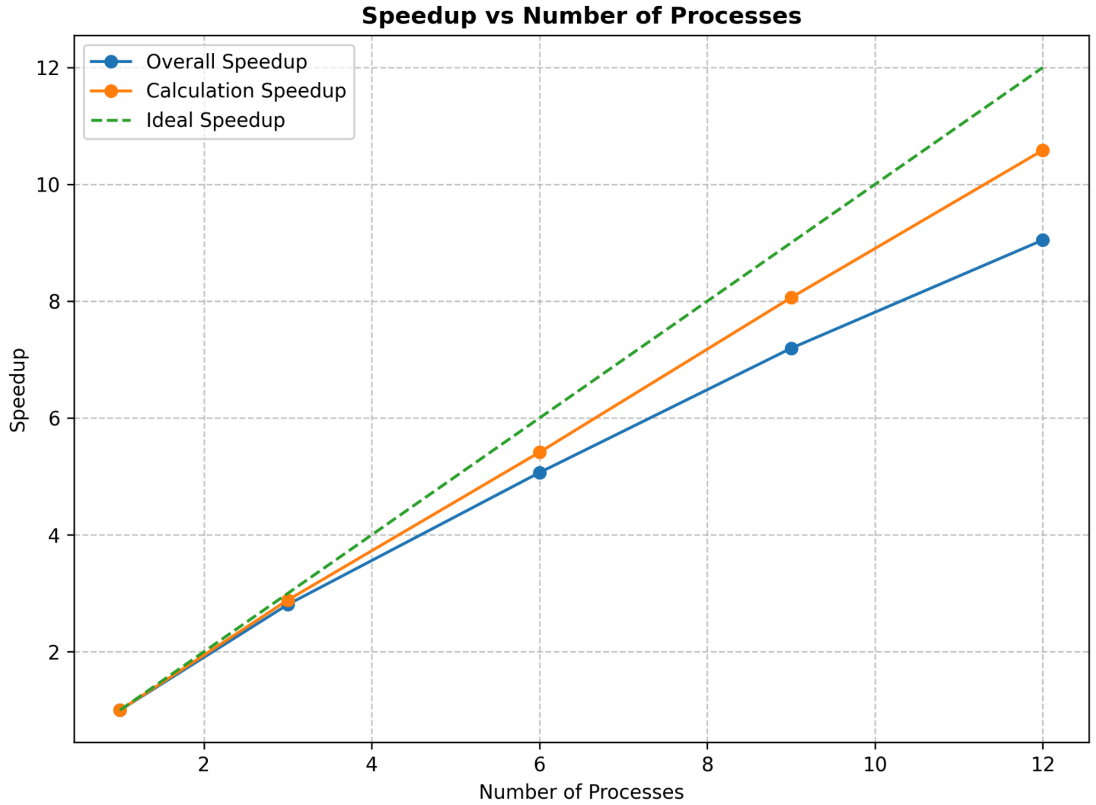
在load balancing實驗的部分，為了要評估每個 rank 所被分配到的工作量是否平均，分別以 rank size 為 6, 12 測量每個 rank 的執行時間。

以下實驗使用測資 strict06.txt

紀錄不同**process**數下,各個部分時間占比

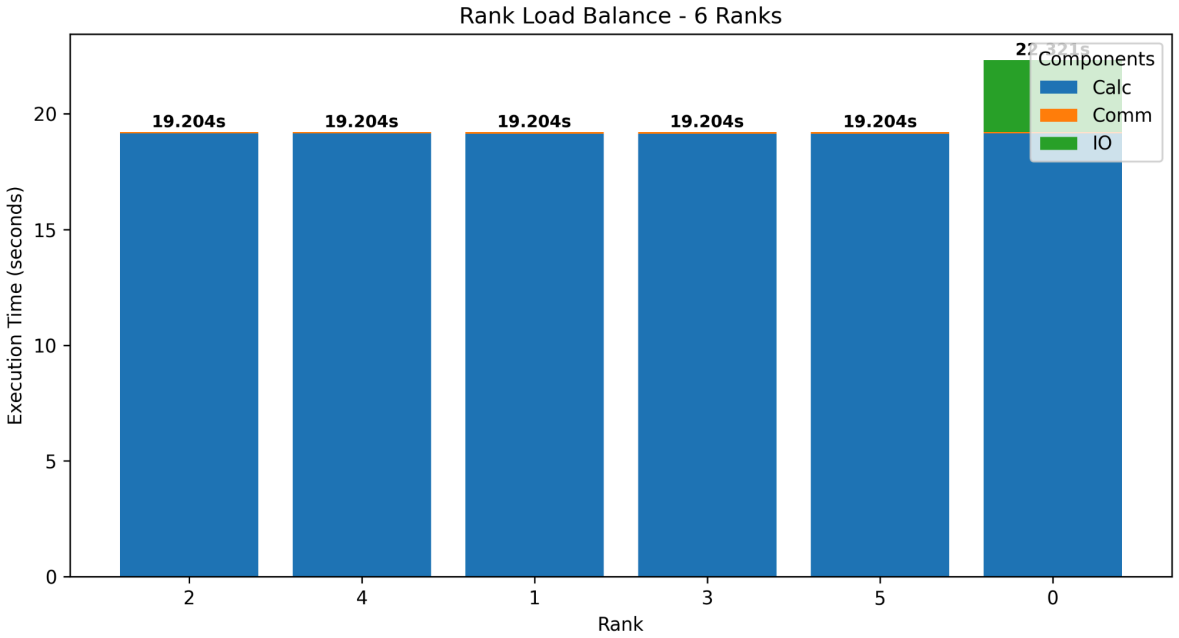


Strong scalability speedup - 紀錄overall speed, computation speedup 比例

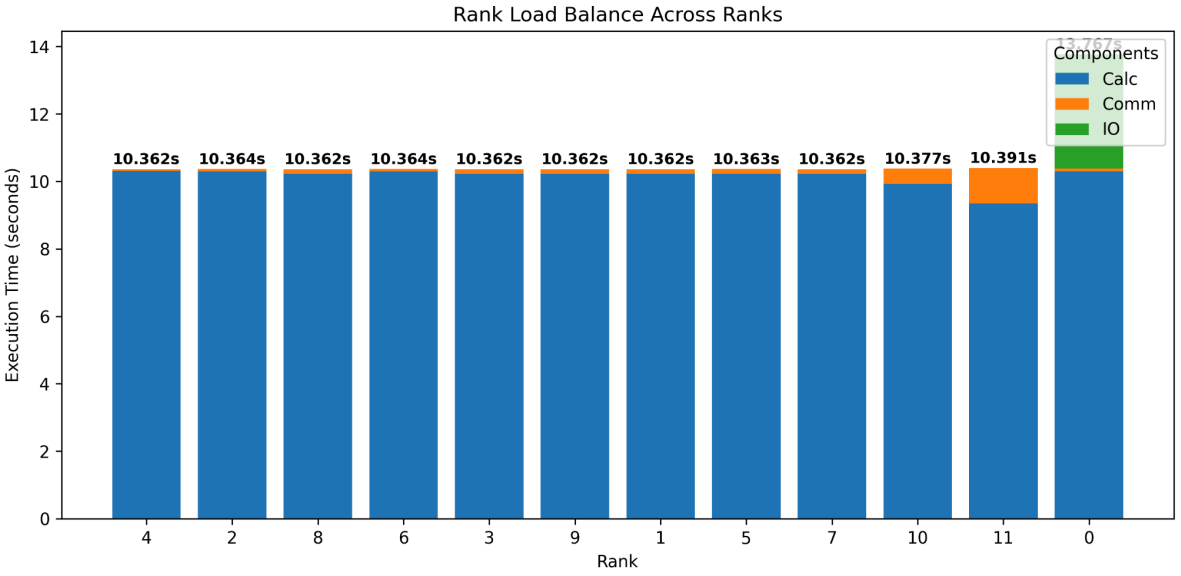


以下實驗使用測資 strict32.txt

hybrid load balancing - 6 processes



hybrid load balancing - 12 processes



Discussion:

從整體實驗結果來看，hybrid版本在效能和負載平衡上都有不錯的表現。

在 Strong scalability 部分，隨著 process 數從 2 增加到 12，整體執行時間明顯下降，顯示出良好的加速效果。從圖上可以看到 Computation speedup 幾乎貼近理想線，而 overall speedup 稍微低一點，主要是因為有額外的 communication 和 I/O 開銷。這代表計算部分的 scalability 很好，但隨著 process 增加，io佔據的比例也會增加。
在 Load balancing 的實驗中，可以看到每個 rank 的執行時間幾乎相同，代表任務分配是相當平均的。只有 rank 0 的執行時間稍長一點，原因是它負責使用 MPI_Gatherv 收集所有結果並輸出 PNG 檔，所以多了一些 I/O 時間。
MPI 這層的 interleaved row 分配方式 (rows[lr] = rank + lr * world) 效果明顯，讓每個 rank 都能分到靠近影像中心（計算較重）的區域，避免某些 rank 工作太多、某些太少的情況。

Conclusion & Possible Improvements

這次作業主要是把 Mandelbrot Set 做成平行化版本，從最初的 pthread 實作一路優化到用 atomic + chunk + SSE2。
過程中最大的挑戰是怎麼在多執行緒下保持負載平衡，同時減少同步開銷。
實驗結果顯示，atomic 版本的效能明顯比 mutex 好，SSE2 也讓單執行緒的速度快很多。整體來說，優化後的版本在執行時間和 load balancing 上都表現不錯。
不過在 scalability 上還是有些限制，例如記憶體頻寬和thread 管理的額外開銷，導致超過 8~12 threads 後 speedup 變得不明顯。
未來可以改進的方向包括：

- 用 動態 chunk size 讓分配更靈活
- 用更加聰明的方式做資料分割

整體來說，這次作業讓我學到很多關於平行化的細節，尤其是要讓多執行緒真的同時有效工作比想像中難得多。