CS542200 Parallel Programming
Homework 2: Mandelbrot Set

Bo-Wei Lee (李柏葳)

111065525

1 程式建置

1.1 概述

這一次的作業是 Mandelbrot Set，其目的爲在複數平面上畫出碎形的圖。在圖上的每一個點，代表者該複數座標在進行幾次複二次多項式後，絕對值會超過 2。由於題目規定不能對於演算法本身進行優化，每一個點都要計算到。所以這次的優化會主要注重在 load balancing 跟 SIMD 上。

1.2 Load balancing

根據題目，每一個點都需要做最多 iter 次的複二次多項式。一個點在計算新的複數值時，都必須要依賴上一步的值，所以在這部分無法做平行化。相反點與點之間的計算是互相獨立的，我們可以從這部分下手去做平行化，分配任務給不同的執行緒。
在實作分配機制前，我們必須要考慮幾件事情如下。

1. 是要採用 Static load balancing 還是 Dynamic load balancing？如果使用靜態的方式，在開執行緒前就先拿到需要執行的所有任務，可以省去執行緒之間溝通的時間，但是在分配工作方面會變的極爲困難，因爲我們無法預先知道每個點或區塊的總執行時間，造成使用率低落。所以本份程式是採用動態的方式，是在執行期間做工作的調度。

2. Work pool 的種類，在 hw2a 中，因爲執行緒都在同一台機器上，所以是採用 Centralized Work Pool 的方式。將需要執行的任務與結果放在全域變數中，執行緒將結果回傳至全域陣列，並拿取新的工作任務。

3. 分配的任務大小和形狀會是執行的一大關鍵，如果切的太大，容易會有執行緒執行時間不均的問題，切得太小則是拿取任務的溝通時間會太長。在任務中，我有對切成不同形狀的任務進行測試，總執行時間最短的是長爲圖片 width，寬爲一的長方形作爲 task 下去分配效果最好，可能是因爲切其他形狀要將拿到的任務編號轉爲二維座標的額外時間所致。

4. 如何實作 Work pool，爲了提升執行緒溝通的效率，我決定只使用一個 atomic_int 來代表下一個要執行的任務編號，而不是使用 pthread_mutex_lock。在初始化時其值爲零，每個執行緒在提取任務時會使用 fetch_add，拿取任務編號後將其值加一。如果拿到超過任務總數的編號，就結束這個執行緒。

1.3 初版，Only load balancing，HW2a 成績 748.88s

在這版，主要的計算架構跟提供的 sample code 差不多，有差異的是分配工作的方式。在這裡我將要計算的圖片分成許多高度一像素寬度不變的長方形，讓所有執行緒們拿取。這樣設計的好處是因爲 work pool 只需要管理一個變數，就是目前被執行的高度。當執行緒拿取任務時，只需要用 atomic 裡的 fetch_add，拿取的同時將高度加上一即可，非常節省共同溝通的效能。

1.4 SSE2 Vectorize 後版本，HW2a 成績 459.79s

根據以上的版本，我在這版主要用 vectorize 來做速度上的優化。使用的 extension 爲 SSE2，可惜的是工作站上沒有提供 AVX 的指令集。這版要優化的目的很簡單，就是將兩個 pixel 的浮點數綑綁成 __m128d 在一起計算，如此理論上可以省下一半的複二次多項式計算時間。實作核心部分程式碼如下。

在實作 SSE2 版中，我遇到比較大的麻煩是兩個 pixel 終止的 iteration 數量不一樣，如果強迫讓比較快就長度超過二的 pixel 跟還沒超過二的 pixel 繼續做運算，不僅會浪費掉效能，還有可能造成 overflow 的情況發生導致結果錯誤。解決方法就是將 __m128d 視爲兩套處理系統，各自擁有獨立檢查終止與替換機制，以及一個 counter 紀錄目前在處理哪個點位。當某一個 pixel 已達到終止條件，就把結果輸出，並替換成下一個未處理的 pixel。當其中一套系統嘗試撈取圖片外的 pixel 時，執行緒會停止目前兩套系統的計算，並重新計算可能沒算完的 pixel，然後再從 work pool 撈取下一個任務。

```
// Fetch new task (task shape is width * 1)
fetchedTask = curTask.fetch_add(1);
// Thread will start to terminate when all missions complete
while (fetchedTask < height) {
    // Set x, y, x0, x1, curPixel[2], repeats[2], first four var's is __m128d
    while (true) {
        if (length_squared[0] >= 4 || repeats[0] >= iters) {
            image[fetchedTask * width + curPixel[0]] = repeats[0];
            if (i >= width) break; // Break if all pixels are processed or
                processing
            // Load new pixel
        }
        if (length_squared[1] >= 4 || repeats[1] >= iters) {
            image[fetchedTask * width + curPixel[1]] = repeats[1];
            if (i >= width) break; // Break if all pixels are processed or
                processing
            // Load new pixel
        }
        // Calculate the next iteration and add the repeat value
    }
    if (curPixel[0] < width)
        // Re-calculate curPixel[0] left over from the break above
    if (curPixel[1] < width)
        // Re-calculate curPixel[0] left over from the break above
    // Fetch new task
    fetchedTask = curTask.fetch_add(1);
}
```

Listing 1: Sudo core code on threads

1.5 Hybrid 版本，HW2b 成績 317.03s

這個版本是改採用 OpenMP + MPI 的函式庫去實作的，OpenMP 的部分原理和 pthread 版一樣。比較不同的地方是 MPI 的部分，因爲要程式有跨到 Node，所以在工作的分配上難度會更加高。雖然 MPI 本身也有 mutex lock 的機制，但考量溝通的成本與實作的難度，後來決定是使用預先將要運算的區域先訂好，丟到不同的 Node 去做計算，最後再將處理好的區塊回傳至 master node 做組合。那在不同節點任務的分配上，我是採用穿插的方式，假設有三個節點要處理一張圖片，那第一個節點需要執行所有三的倍數列，第二個節點要執行三的倍數加一列，第三個節點要執行三的倍數加二列，以此類推。因爲 Mandelbrot Set 的點有 locality 的特性，每個點的數值通常相差不大，所以這樣的方式可以很有效的達到 load balancing 的效果。

2 實驗與分析

2.1 實驗用資料與裝置描述

1. 測試用的機器爲課程所提供的 Apollo 伺服器
2. 在以下實驗中，我選用的是公開資料集裡的 strict34.txt 做爲輸入
3. 測資的 attribute 爲 10000 -0.2931209325179713 -0.2741427339562606 -0.6337125743279389 -0.6429654881215695 7680 4320，會選用這筆資料是因爲其數據量夠大，可以凸顯平行程度帶來的優勢。
4. 計時方式使用 clock_gettime(CLOCK_MONOTONIC, &time);

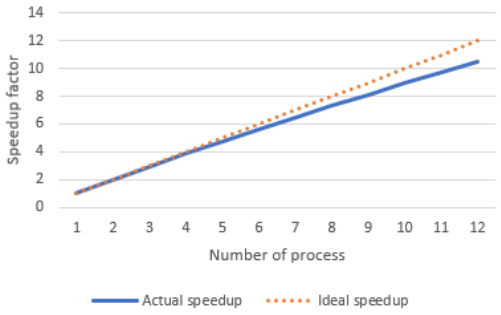

Figure 1: Process 數量對 Speedup factor

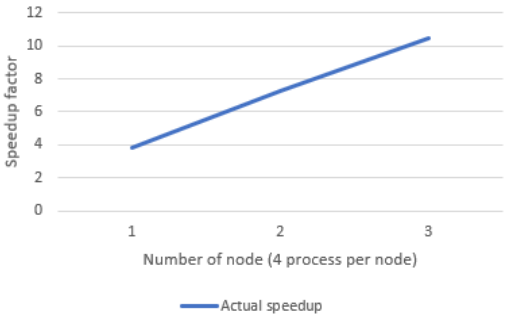

Figure 2: Node 數量對 Speedup factor

2.2 Node 及 Process 的數量對於執行時間的影響

圖 1 是以 Mandelbrot Set 的程式所使用的 process 數量對 Speedup factor 的圖。虛線代表程式要達到 Strong scaling 需要達到的 Speedup factor。這張圖上所有的 process 都是在同一個節點進行運算，所以 MPI_Gather() 的時間基本上可以忽略。在這張圖中可以看到，隨著 process 數量的提升，Speedup 的斜率雖然會越來越低，但是能有保持在一定的水準。
圖 2 則是使用 Node 數量的多寡對於 Speedup factor 的圖，每一個 Node 會有 4 個 process。可以看到即使使用多達 12 個 process，Speedup factor 也和圖一差距不大，代表在 load balancing 跟 MPI 回傳結果至 ranking 方面對於效能的影響很小。

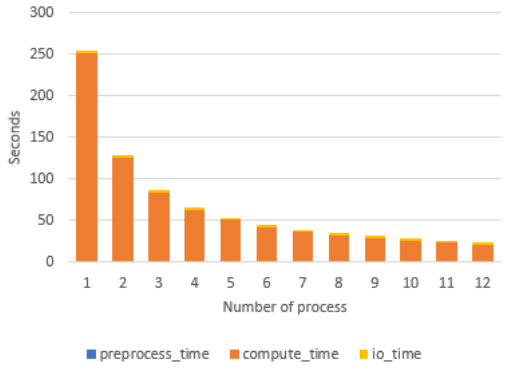

Figure 3: Process 數量對各項執行時間

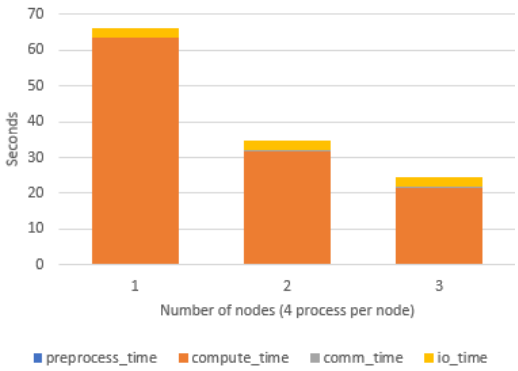

Figure 4: Node 數量對各項執行時間

2.3 Process 數量對各項執行時間

圖 3 爲 process 數量對各項執行時間的圖，其中 preprocess time 是從開始到產生最後一個執行緒的時長，compute time 爲在運算 Mandelbrot Set 的總時長，從最後一個執行緒被建立至最後一個執行緒完成運算爲止，io time 爲資料寫出至.png 檔的時間。圖 4 則是 Node 數量對各項執行時間的圖。在這兩張圖中，可以明顯的看到 Compute 的時間會與 process 的數量成反比，這是因爲任務執行的數量被平均分散至各執行緒所致，preprocess 跟 io time 時間變化不大，因爲這兩個部分我都沒有實作平行化，但這兩個時間對總時間的占比很小，所以影響不大。

2.4 Load balancing between MPI

	mixed task alloc. (s)	block task alloc. (s)
rank0 process0	19.340807	5.486131
rank0 process1	19.340807	5.486687
rank0 process2	19.340807	5.486960
rank0 process3	19.340807	5.487456
rank1	20.930407	26.184015
rank2	20.925207	30.800462

Table 1: 兩種不同的 MPI 任務分配方法 vs thread 執行時間

表一是兩種不同的 MPI 任務分配方法對上執行緒執行時間，rank 0 將所有執行緒列出，rank 1 及 2 是將其中四個執行緒做平均後列出。可以看到第一種任務分配方法，也就是將圖片切成許多長爲一寬爲 width 的長方形，然後輪流分配到不同的節點上，其效果會比將圖片長直接分成節點數量等分，再分配到不同節點的 load balance 效果會好很多。

3 結論

在這次的作業中，最讓我有成就感的地方莫過於在資料的分配上，MPI 的任務從一開始的長方形到 32 * 32 的正方形再到長爲一的長方形混用，在 scoreboard 上的排名也越來越前面，而且還是一次前進一二十名，眞的有被震撼到。