hw2_112062532

姓名：溫佩旻　學號：112062532

Implementation

HW2a - Pthread

這邊我主要實作的方法有以下兩個:

- **如何決定每個thread的工作量?**
  因為Mandelbrot set每個height的計算量都不一樣，因此本來是平均分配height給每個thread，後來改用類似thread pool的方式，讓每個thread共同存取`next_row`這個變數，也就是一個thread一次負責一個row的計算，算完一個task再去取的下一個task，盡量讓每個cpu工作量一致，來達到比較好的load balancing。這邊因為每個thread都需要共同存取`next_row`來計算下一個要運算的row位置，所以使用`pthread_mutex_t`來保護住避免race condition。

```
126     while(1) {
127         int row;
128 
129         pthread_mutex_lock(&task_mutex);
130         if (next_row >= height) {
131             pthread_mutex_unlock(&task_mutex);
132             break;
133         }
134         row = next_row++;
135         pthread_mutex_unlock(&task_mutex);
136 
137         computeMandelbrot_SIMD(row);
138     }
```

- **如何計算mandelbrot set?**
  在QCT Server中可以使用lscpu來看到Flags中有包含avx512，所以這邊我用AVX-512 的`__m512d`型別來加速 Mandelbrot set的計算，同時做8個雙精度(double)浮點數的計算，能大幅加快計算時間及增加平行度。
  當thread取得他目前要計算的row時，他會用上面所說的SIMD方式做計算，下面是程式碼的部分，主要是先將一些不變的常數值或是將之後會做到的常數值計算先存進AVX-512的register中，再用AVX-512提供的函式做計算(如`_mm512_fmadd_pd`)，計算完成後將結果存回主記憶體才能做存取，最後處理那些沒辦法被8整除的部分，剩餘部分就需要直接逐一計算而非使用SIMD的方式，以免造成錯誤的結果。

```
void computeMandelbrot_SIMD(int row) {
    const int simd_width = 8;  // 512 / 64bit
    int simd_end = width - (width % simd_width);
    __m512d vec_left = _mm512_set1_pd(left);
    __m512d vec_right_left_diff = _mm512_set1_pd((right - left) / width);
    __m512d vec_y0 = _mm512_set1_pd(row * ((upper - lower) / height) + lower);
    __m512d vec_length_squared_limit = _mm512_set1_pd(4.0);
    __m512d vec_two = _mm512_set1_pd(2.0);

    for (int i = 0; i < simd_end; i += simd_width) {
        __m512d vec_i = _mm512_set_pd(i + 7, i + 6, i + 5, i + 4, i + 3, i + 2, i + 1, i);
        __m512d vec_x0 = _mm512_fmadd_pd(vec_i, vec_right_left_diff, vec_left);
        __m512d vec_x = _mm512_setzero_pd();
        __m512d vec_y = _mm512_setzero_pd();
        __m512d vec_length_squared = _mm512_setzero_pd();
        __m512d vec_repeats = _mm512_setzero_pd();

        for (int rep = 0; rep < iters; ++rep) {
            __mmask8 mask = _mm512_cmp_pd_mask(vec_length_squared, vec_length_squared_limit, _CMP_LT_OQ);
            if (mask == 0)  break;

            __m512d vec_temp = _mm512_add_pd(_mm512_sub_pd(_mm512_mul_pd(vec_x, vec_x), _mm512_mul_pd(vec_y, vec_y)), vec_x0);
            vec_y = _mm512_fmadd_pd(vec_two, _mm512_mul_pd(vec_x, vec_y), vec_y0);
            vec_x = vec_temp;

            vec_length_squared = _mm512_add_pd(_mm512_mul_pd(vec_x, vec_x), _mm512_mul_pd(vec_y, vec_y));
            vec_repeats = _mm512_mask_add_pd(vec_repeats, mask, vec_repeats, _mm512_set1_pd(1.0));
        }

        double result[simd_width];
        _mm512_storeu_pd(result, vec_repeats);
        for (int j = 0; j < simd_width; ++j) {
            image[row * width + i + j] = (int)result[j];
        }
    }

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

HW2b - Hybrid(OpenMP+MPI)

這邊主要實作架構是採master-slave，並採用work pool的方式來分配任務給slave，下面說明比較重要的三大部分。

- **如何決定每個rank的工作量?**
  我一開始的想法是讓height平均分配給rank，但因為每個height計算量不一樣，會導致load balancing很差，後來是用主從式分法，把rank 0拿來當作是master，負責分派任務給其餘rank做運算，最後再把結果統整寫進png檔，想法上有點類似Pthread的實作。

  - master 實作
    分配任務的方式是以row為單位，每次會發送`next_row`給其餘rank，跟他們說現在需要負責的是哪一row，接著在MPI_Recv這邊會被block住，等待有process將工作做完，會先收到process傳送他負責的是第幾個row，接著再接收同一個process所傳送的訊息，來知道該process負責的row的結果是甚麼，透過memeory copy的方式把結果複製回image中，重複上述步驟直到做完所有task，再依序發送終止的訊息給其餘process。

```
142     if (mpi_rank == 0) {
143         int* image = (int*)malloc(width * height * sizeof(int));
144         assert(image);
145 
146         int next_row = 0;
147         int num_workers = mpi_size - 1;
148         MPI_Status status;
149 
150         for (int i = 1; i <= num_workers; ++i) {
151             if (next_row < height) {
152                 MPI_Send(&next_row, 1, MPI_INT, i, TASK_TAG, MPI_COMM_WORLD);
153                 ++next_row;
154             } else {
155                 int terminate = -1;
156                 MPI_Send(&terminate, 1, MPI_INT, i, TERMINATE_TAG, MPI_COMM_WORLD);
157             }
158         }
159 
160         while (num_workers > 0) {
161             int row;
162             MPI_Recv(&row, 1, MPI_INT, MPI_ANY_SOURCE, DATA_TAG, MPI_COMM_WORLD, &status);
163 
164             int* row_data = (int*)malloc(width * sizeof(int));
165             MPI_Recv(row_data, width, MPI_INT, status.MPI_SOURCE, DATA_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
166             memcpy(&image[row * width], row_data, width * sizeof(int));
167             free(row_data);
168 
169             if (next_row < height) {
170                 MPI_Send(&next_row, 1, MPI_INT, status.MPI_SOURCE, TASK_TAG, MPI_COMM_WORLD);
171                 ++next_row;
172             } else {
173                 int terminate = -1;
174                 MPI_Send(&terminate, 1, MPI_INT, status.MPI_SOURCE, TERMINATE_TAG, MPI_COMM_WORLD);
175                 --num_workers;
176             }
177         }
178 
179         write_png(filename, iters, width, height, image);
180         free(image);
181     }
```

- slave 實作
  除了rank 0以外的process都稱為slave，負責接收master送來的任務並執行計算，計算完畢後，會先MPI_Send告訴master現在負責的是哪一個row，接著馬上把該row計算完的結果傳給master，實現上述步驟直到收到 `TERMINAL_TAG` 的TAG。

```
182     else {
183         int row;
184         MPI_Status status;
185         int* row_data = (int*)malloc(width * sizeof(int));
186         assert(row_data);
187 
188         while (1) {
189             MPI_Recv(&row, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
190 
191             if (status.MPI_TAG == TERMINATE_TAG) {
192                 break;
193             }
194 
195             computeMandelbrot_SIMD(row, row_data);
196 
197             MPI_Send(&row, 1, MPI_INT, 0, DATA_TAG, MPI_COMM_WORLD);
198             MPI_Send(row_data, width, MPI_INT, 0, DATA_TAG, MPI_COMM_WORLD);
199 
200         }
201         free(row_data);
202     }
```

- **如何決定每個rank底下每個thread的工作量?**
  rank = 0 的process這邊主要負責的工作是分派任務及最後將結果統整畫入PNG檔中，基本上沒有什麼太複雜的計算部分，因此這邊就沒有特別使用OpenMP去做平行的動作。

那在除了rank = 0的process之外的process，也就是slave，會需要計算Mandebrot Set，雖然已經使用SIMD來做加速運算，但這邊還是可以使用OpenMP來將for迴圈平行計算來增加平行度，這邊schedule的部分我是使用dynamic的方式來進行運算。

- **如何計算mandelbrot set?**
  這邊和pthread一樣，使用AVX-512來做SIMD計算。

Optimization

- **load balancing**
  - Pthread
    最一開始是用最簡單的平行方式，就是把row平均平分給每個cpu，讓每個cpu計算到row數量盡量是一樣多的(假設平分後有餘數為r，那前r個cpu每個會多負責一個row的計算)，這樣看似平均的分法，實際上每個cpu被分配到所需的計算量是差距很大的，有些cpu被分配到的很快就算完，有些需要花很多時間，導致效率超差，雖然有過測資但需要713秒。
    因此後來改用類似thread pool的方式，讓每個thread一次先負責一個row的計算，算完一個task再去取的下一個task，並用`next_row`來記錄目前global計算到哪一個值，盡量讓每個cpu工作量一致，這樣時間可以減少一倍，變成約330秒。
  - Hybrid
    這邊最一開始的想法也是將每一個height平均分配給每個process，但這樣做load balancing效果很差，所以改用在process端會使用master-slave的方法去分配任務，每次分配一個row的計算量，slave做完一個row，再將結果傳給master，並取的下一個要計算的row資訊，其中每個process中的thread再透過OpenMP去做dynamic的schedule，讓程式自動去分配任務給thread。

- **vectorization**
  這邊因為pthread和hybrid的優化部分大致差不多，所以就合在一起寫。
  如上述Implementation的第二點，不是使用原本的逐一計算，而是用AVX-512來做SIMD計算，在上面load balancing已經加快一倍的情況下，使用SIMD可以再提升四倍速度左右，效果很好。除此之外，我在使用SIMD時是同時處理16個雙經度浮點數的運算，而不是8個，經過實測同時處理8個、16個、24個計算，16個可以最大化的利用CPU資源，獲得更好的結果。

Experiment & Analysis

Methodology

- **System Spec:** 一樣是在課堂提供的QCT Server上跑
- **Performance Metrics:**
  採用 `Nsight System` ,並用NVTX來計算時間。

  Pthread

  - **IO time:** 在`write_png`前後加上`nvtx`的push、pop，得到IO time。

- **computation time:** 在 `pthread_create` 前面和 `pthread_join` 後面加上 `nvtx` 的 push、pop，得到computation time。

Hybrid

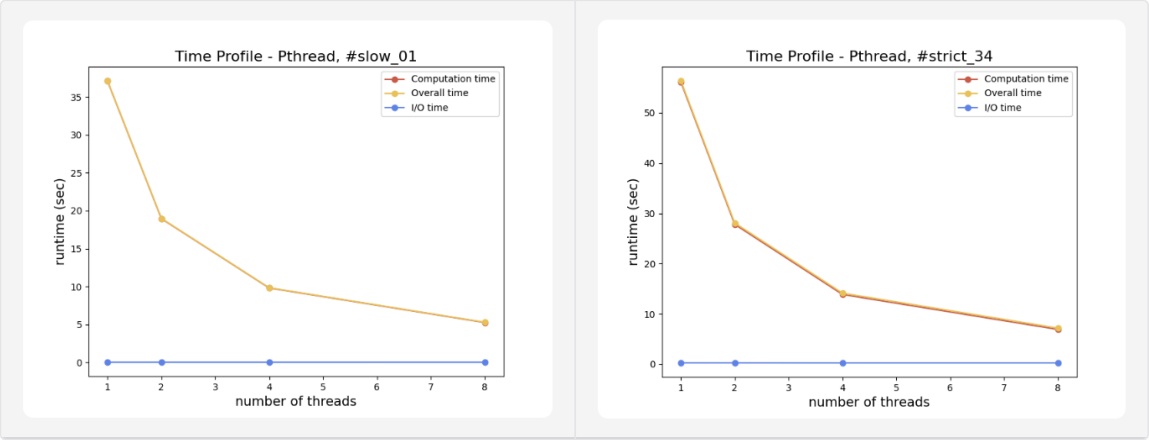
- **IO time:** 在讀取和寫入檔案的 `MPI_File_open` 前和 `MPI_File_close` 後都加上`nvtx`的 push、pop，以及`write_png`前後也加上`nvtx`的push、pop，得到IO time。
- **communication time:** 在 `MPI_Sendrecv` 的前後加上`nvtx`的push、pop，得到 communication time。
- **computation time:** 在 `MPI_Init` 後面和 `MPI_Finalize` 前面加上 `nvtx` 的push、pop紀錄程式總共執行時間，在扣除 IO time 和 communication time，剩下的就是 computation time。

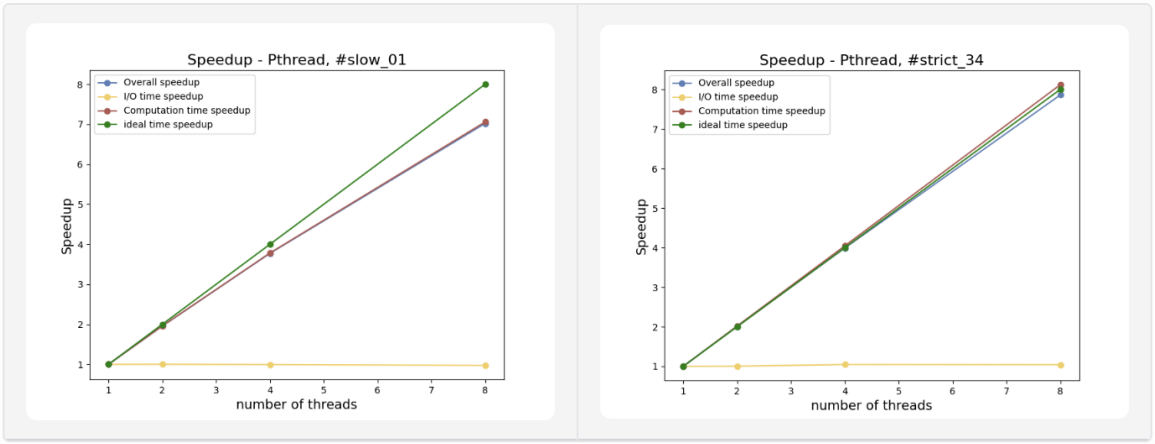
Plot

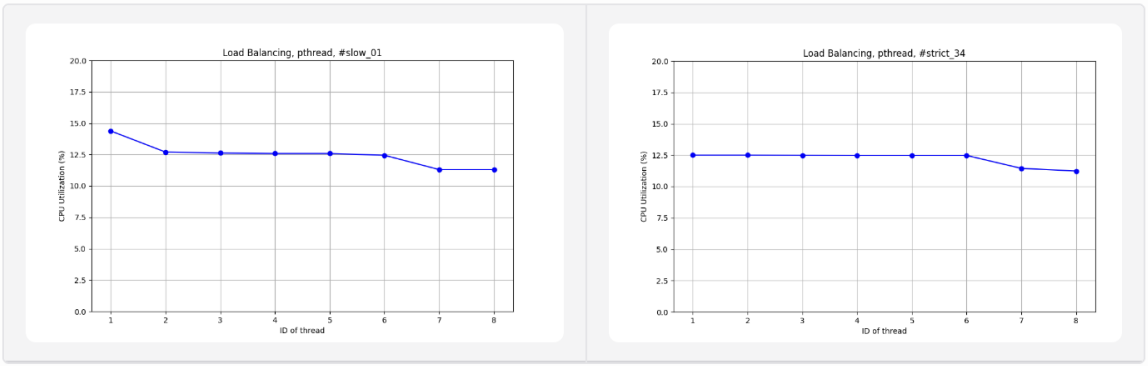
Pthread

- Experimental Method
  這邊我是拿slow_01和strict_34來去做測驗，因為slow_01是我表現相較比較差的測資，而strict_34是大家普遍來講所花費時間最高的測資。
  那在pthread的實驗中我是設定單一node單一process去開不同數量的thread(分別是1、2、4、8個)來去做speedup和Time Profile的實驗，在load balancing的實驗中則是看單一node單一process開8個thread的情況下，每個thread的cpu使用率如何。

- Analysis of Results





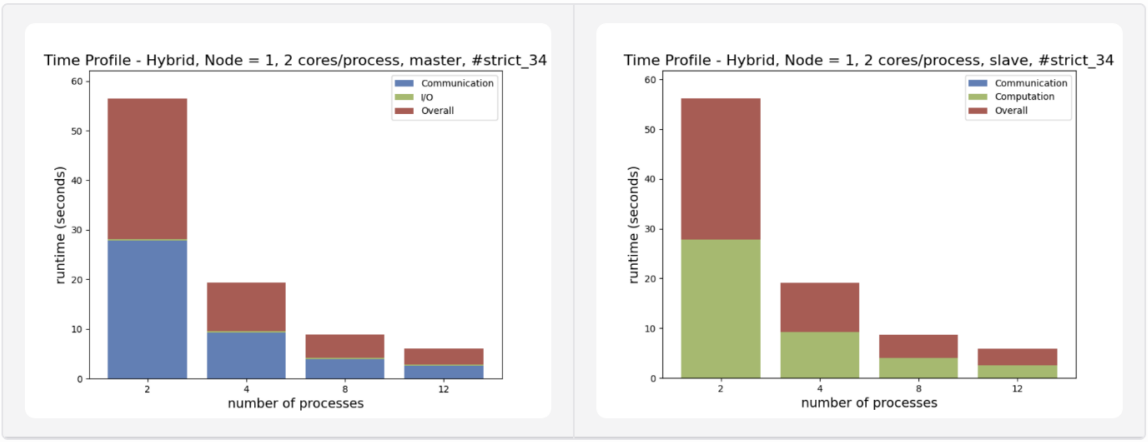


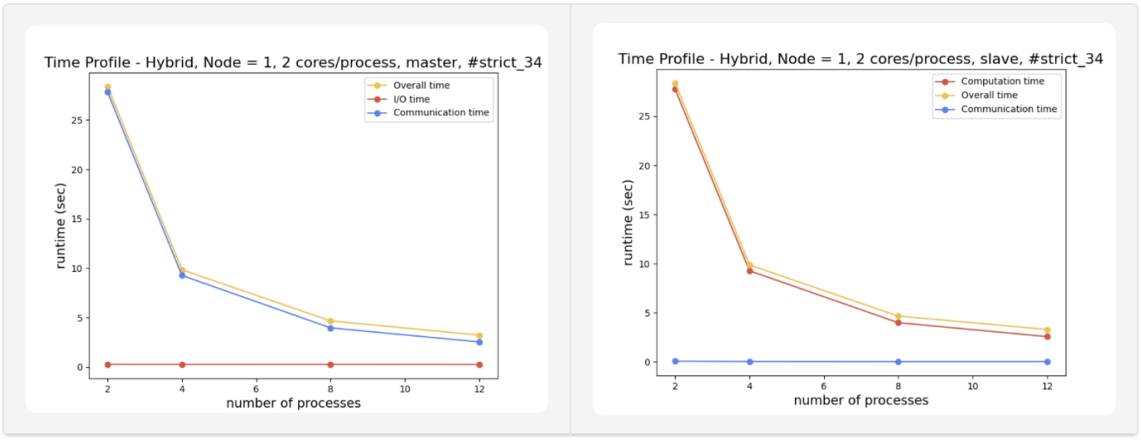
在pthread的time profile中可以看到computation time基本上已經和overall time完全一致，I/O時間非常少，這是因為我在寫入png檔時把壓縮選項拿掉，讓I/O time可以減少非常多。此外，在sppedup的圖中也可以觀察到，不論是computation speedup還是overall speedup，都已經趨近於idael，這代表基本上這隻程式在計算部分已經優化得差不多了，能夠達到線性成長。
在load balance的圖也可以看到每個thread的loading是差不多的，可以推測是因為dynamic去分配工作，讓每個thread工作量差不多。

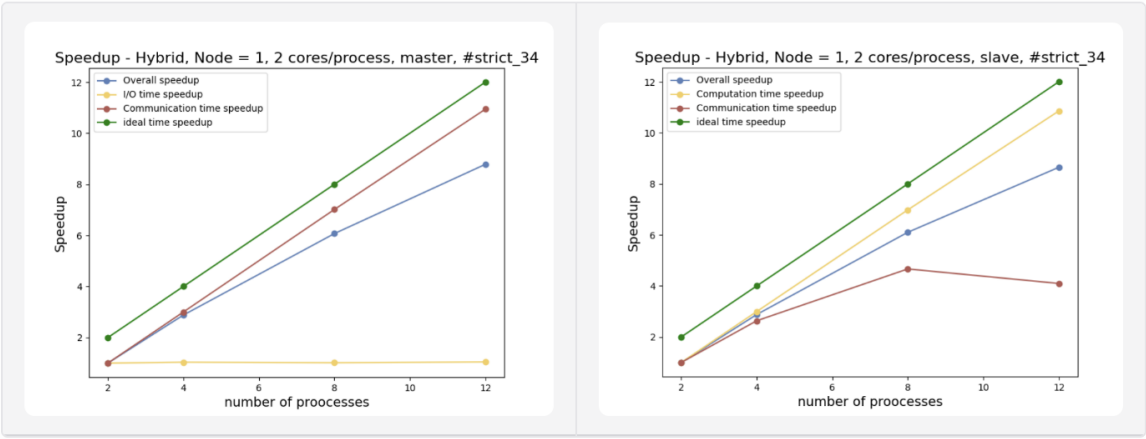
Hybrid

- Experimental Method
  這邊我是拿strict_34來去做測驗，因為strict_34是大家普遍來講所花費時間最高的測資。
  那在hybrid的實驗中我是設定單一node不同process數量(分別是2、4、8、12個)，2 cpu/process，來去做speedup和Time Profile的實驗，那因為我的程式是master-slave架構，所以在這邊的實驗數據會將slave和master分開看，那在slave的部分，我都是採用rank=1的process數據去畫圖呈現結果。

- Analysis of Results







我們可以從上面的圖中發現master的communication time隨著process數量增加，有逐漸下降的趨勢，這我猜測應該是因為在process數量較少時，master需花比較多時間去等待slave的計算，要等有slave計算完成後master才能接收到data在傳遞新的task給slave，同時也可以觀察到slave的計算時間是隨著process上升而遞增的，這呼應了我上面的猜測，當slave的計算時間減少，master就比較不會有閒置在等待的情況發生，那master的communication time也會隨著process增加而減少。

除此之外，也可以透過speedup的圖來驗證上述猜測，master的communication time的speedup曲線是和slave的computation time非常相似的。

Optimization Strategies

可以從上面的圖發現使用master-slave的架構來撰寫的話，在process數量沒那麼多時，會讓效能表現不太好，因為master可能會浪費很多時間在等待而被block住，因此我覺得可以優化的方向會是該怎麼有效利用master的閒置時間，像是在等待時可以去寫入PNG檔案，來增加平行度。更好的方法可能會是不要犧牲一個process去當master，而是另外開一個額外的thread去當master及寫入PNG檔，這樣可能會更好一些。

Experience & Conclusion

在這次作業中我覺得最重要的就是練習如何判斷哪些地方可以用SIMD加速，以及該如何加速，當然這也是整個作業最困難的部分，主要是之前沒有寫過相關vectorization的程式，加上網路上沒有太多教學可以參考，所以在查要用哪些函式的時候會比較困難麻煩，不過也因此學習到一個很酷的加速方法！