

hw2_report

1. Title, name, student ID

110590007 白宸安

2. Implementation

hw2a : pthread版本

1. 首先是先定義好要用到的變數和資料結構，這邊把原本在main中的這些變數都搬到global來放，這樣比較方便其他function的取用。

PackTwoDouble是用來做vectorization的，使用union的話可以直接用.d[0]或.d[1]的方式去access裡面的兩個double。

```
std::queue<int> work_queue;
pthread_mutex_t work_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

union PackTwoDouble {
    alignas(16) double d[2];
    __m128d d2;
};

typedef struct {
    int thread_id;
} ThreadInfo;

int num_threads;
int* image;
int iters; // iteration幾次。int; [1, 2×108]
double left; // real軸的左邊邊界。double; [-10, 10]
double right; // real軸的右邊邊界。double; [-10, 10]
double lower; // image軸的下邊界。double; [-10, 10]
double upper; // image軸的上邊界。double; [-10, 10]
int width; // 圖片x軸有多少points。int; [1, 16000]
int height; // 圖片y軸有多少points。int; [1, 16000]
```

2. 讀取參數和設定image。

3. 把每個row都切成一個工作，放進work_queue裡。

```
// Initialize the work queue with rows to process
for (int j = 0; j < height; ++j) {
```

```
        work_queue.push(j);
    }
}
```

4. 建立多個threads，並讓他們執行render_mandelbrot這個function，等待算完就join thread，最後畫出image。

```
/* Pthread parallel */
pthread_t threads[num_threads];
ThreadInfo thread_info[num_threads];

for (int i = 0; i < num_threads; ++i) {
    thread_info[i].thread_id = i;
    pthread_create(&threads[i], NULL, render_mandelbrot, &thread_info[i]);
}

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);
```

5. render_mandelbrot

這個function由多個thread執行，在work_queue還沒空的情況下，只要有空閒的thread就會去work_queue裡面拿一個row出來算，呼叫cal_pixel_sse2(row); 來計算那個row。

算完的話就會回來，然後再要一個row來算，直到所有row都被算完。

為了避免衝突，去work_queue裡要row時要先拿lock才行。

```
void* render_mandelbrot(void* arg) {
    ThreadInfo* thread_info = (ThreadInfo*)arg;

    while (1) {
        int row;
        pthread_mutex_lock(&work_queue_mutex);

        if (work_queue.empty()) {
            pthread_mutex_unlock(&work_queue_mutex);
            break; // No more work to be done
        }

        row = work_queue.front();
        work_queue.pop();
        pthread_mutex_unlock(&work_queue_mutex);

        cal_pixel_sse2(row);
    }

    pthread_exit(NULL);
}
```

6. void cal_pixel_sse2(int row)

這個是主要計算的function。

這邊的計算是使用SSE2的vectorization，因此一次可以計算兩個pixel。

index用來指現在這個row中算到第幾個pixel（1個row有width個pixel）。

repeats用來記錄兩個pixel分別iterate多少次。

cursor用來記錄目前是算到哪兩個pixel。

下方mandelbrot公式基本上跟講義一樣，因此算式部分不多加說明。

主要流程如下：

只要還有pixel沒算完就一直跑。

可以想成算vectorization乘法的地方會一直跑，一次算兩個pixel，當有一個pixel算完時，就將該pixel的iteration數紀錄到image上，並把該pixel的值換成下一個要算的pixel的，這樣一來下一次計算時就會變成開始算新的pixel。

舉例：

一開始vectorization先算[pixel 0, pixel 1]，當pixel 0先算好時，就將pixel 0存進image。

接著更新pixel 0這格的值（vectorization有兩格，一次可存兩個double），換成pixel 2的。

也就是從下一次開始就是算[pixel 2, pixel 1]。

會一直算直到整個row的pixels都被算完就會結束。

```
void cal_pixel_sse2(int row){
    int index = 0;
    int repeats[2] = {0, 0};
    int cursor[2] = {0, 0};

    cursor[0] = index++;
    cursor[1] = index++;

    union PackTwoDouble length_squared, z_reals, z_imags, c_reals, c_imags;

    length_squared.d2 = _mm_set_pd(0, 0);
    z_reals.d2 = _mm_set_pd(0, 0);
    z_imags.d2 = _mm_set_pd(0, 0);

    double origin_imag = row * ((upper - lower) / height) + lower;

    c_reals.d[0] = cursor[0] * ((right - left) / width) + left;
    c_reals.d[1] = cursor[1] * ((right - left) / width) + left;

    c_imags.d[0] = origin_imag;
    c_imags.d[1] = origin_imag;

    while(cursor[0] < width || cursor[1] < width ){

        // if length_squared.d[0] < 4.0 break
```

```

// if length_squared.d[1] < 4.0 break

while(true){
    __m128d z_reals_squared = _mm_mul_pd(z_reals.d2, z_reals.d2);
    __m128d z_imags_squared = _mm_mul_pd(z_imags.d2, z_imags.d2);

    length_squared.d2 = _mm_add_pd(z_reals_squared, z_imags_squared);

    if(length_squared.d[0] > 4 || length_squared.d[1] > 4) break;

    __m128d z_reals_images_mul = _mm_mul_pd(z_reals.d2, z_imags.d2);

    z_reals.d2 = _mm_add_pd(_mm_sub_pd(z_reals_squared, z_imags_squared), c_reals.d2);
    z_imags.d2 = _mm_add_pd(_mm_add_pd(z_reals_images_mul, z_reals_images_mul), c_imags.d2);

    ++repeats[0];
    ++repeats[1];

    if (repeats[0] >= iters || repeats[1] >= iters) break;
}

if((length_squared.d[0] > 4) || repeats[0] >= iters) {
    // 儲存入本pixel的圖片
    image[row * width + cursor[0]] = repeats[0];

    repeats[0] = 0;
    cursor[0] = index++;

    z_reals.d[0] = 0;
    z_imags.d[0] = 0;
    length_squared.d[0] = 0;

    c_reals.d[0] = cursor[0] * ((right - left) / width) + left;
    c_imags.d[0] = origin_imag;

    if(cursor[0] >= width){
        c_reals.d[0] = 0;
        c_imags.d[0] = 0;
    }
}

if((length_squared.d[1] > 4) || repeats[1] >= iters) {
    // 儲存入本pixel的圖片
    image[row * width + cursor[1]] = repeats[1];

    repeats[1] = 0;
    cursor[1] = index++;

    z_reals.d[1] = 0;
    z_imags.d[1] = 0;
    length_squared.d[1] = 0;

    c_reals.d[1] = cursor[1] * ((right - left) / width) + left;
    c_imags.d[1] = origin_imag;

    if(cursor[1] >= width){
        c_reals.d[1] = 0;
        c_imags.d[1] = 0;
    }
}
}
}
}

```

hw2b : hybrid版本

hybird版本大架構跟pthread版的差不多，

相同的地方包括：

1. 初始化變數和讀取參數
2. 工作會切完放進work_queue
3. render_mandelbrot
4. void cal_pixel_sse2(int row)
5. vectorization

這些部分就先不重複說明。

比較不同的地方有：

1. 加上MPI的work pool來分row，rank 0當manager，其他的rank當worker，只要有空間的rank就分一個row給它計算，持續分配直到所有row都算完。

大致上參照了課程講義的方法來實作。

worker裡面不是直接access image，而是先將算好的東西存進一個buffer，這個buffer存有這個row上每個pixel的iteration數，在cal_pixel_sse2中算好的repeats也是先寫進buffer裡。等整個row算好後將buffer回傳給manager，接者manager會在將這個buffer的內容畫到對應的image row上。

```
if(rank == 0){ // master process
    /* allocate memory for image */
    image = (int*)malloc(width * height * sizeof(int));
    assert(image);

    /* MPI work pool */

    for (int j = 0; j < height; ++j) {
        row_queue.push(j);
    }

    // 初始化變數
    int* master_buffer = (int*) malloc(width * sizeof(int)); // 接收算完的那個row顏色分別要是多少，之後再由rank 0來畫
    int count = 0;
    int j;
    int termination_tag = -1; // Termination tag
    int completed_row = -1; // Variable to store the completed row
    int completed_worker;
    MPI_Status status;

    // 先送第一筆工作給workers
    for(int k=1; k<p; k++){
        j = row_queue.front();
        row_queue.pop();
        MPI_Send(&j, 1, MPI_INT, k, 0, MPI_COMM_WORLD);
        count++;
    }

    // 接著開始動態分工作
    do{
        MPI_Recv(master_buffer, width, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        completed_worker = status.MPI_SOURCE;
```

```

        completed_row = status.MPI_TAG; // Store the row that was processed
        count--;

        // draw the image
        //for(int i=0; i<width; ++i){
        //    image[completed_row * width + i] = master_buffer[i];
        //}
        memcpy(&image[completed_row * width], master_buffer, width * sizeof(int));

        // allocate next task
        if(!row_queue.empty()){
            j = row_queue.front();
            row_queue.pop();
            MPI_Send(&j, 1, MPI_INT, completed_worker, 0, MPI_COMM_WORLD);
            count++;
        }
        else{
            MPI_Send(&termination_tag, 1, MPI_INT, completed_worker, 0, MPI_COMM_WORLD);
        }
    }while(count > 0); // 在分發出去的工作都收回來之前不能停
}
else{ // worker process
    int received_row;
    Task_Struct T;

    // 創建一個buffer
    int* buffer = (int*) malloc(width * sizeof(int)); // 接收算完的那個row顏色分別要是多少，之後再由rank 0來畫
    while (1) {
        // Receive a row from the manager
        MPI_Recv(&received_row, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Check if there's no more work to be done
        if (received_row == -1) { // terminating tag
            break;
        }

        // Process the entire row
        int width_start = 0;
        int chunk_size = 10;

        while (width_start < width) {
            int width_end = width_start + chunk_size;
            if (width_end > width) {
                width_end = width; // Adjust the last chunk
            }

            // Create a task for the chunk
            T.row = received_row;
            T.width_start = width_start;
            T.width_end = width_end;

            // Push the task (chunk) into the work_queue
            work_queue.push(T);
            width_start = width_end;
        }

        // omp parallel
        #pragma omp parallel num_threads(num_threads)
        {
            render_mandelbrot(buffer, 0);
        }
        MPI_Send(buffer, width, MPI_INT, 0, received_row, MPI_COMM_WORLD);
    }
}
}

```

2. 一個process中的多個threads負責計算一個row，而pthread版本中是一個thread算一個row。

```

/* MPI work pool */

for (int j = 0; j < height; ++j) {
    row_queue.push(j);
}

```

3. 有將分配到的row在進行切分，切成多個size為10 pixels的chunk，每個thread一次拿一個chunk去計算。這邊的size為10是經過測試效果較好的設定，如果size太少會導致thread要一直拿工作，又有lock會擋可能造成變慢，而如果size太多會導致pixels load unbalance。

```

// Process the entire row
int width_start = 0;
int chunk_size = 10;

while (width_start < width) {
    int width_end = width_start + chunk_size;
    if (width_end > width) {
        width_end = width; // Adjust the last chunk
    }

    // Create a task for the chunk
    T.row = received_row;
    T.width_start = width_start;
    T.width_end = width_end;

    // Push the task (chunk) into the work_queue
    work_queue.push(T);
    width_start = width_end;
}

```

4. 這邊的thread是透過omp去呼叫，而不是使用pthread來實作。

```

// omp parallel
#pragma omp parallel num_threads(num_threads)
{
    render_mandelbrot(buffer, 0);
}

```

5. 當只有1個process的case下，就不使用MPI（只有1個process時就沒有人當worker）。

```

// 考慮只有一個process的情況
if(p == 1){
    printf("only 1 process\n");
    /* allocate memory for image */
    image = (int*)malloc(width * height * sizeof(int));
    assert(image);

    Task_Struct T;

    for (int j = 0; j < height; ++j) {
        int width_start = 0;
        int chunk_size = 20;
    }
}

```

```

        while (width_start < width) {
            int width_end = width_start + chunk_size;
            if (width_end > width) {
                width_end = width; // Adjust the last chunk
            }

            // Create a task for the chunk
            T.row = j;
            T.width_start = width_start;
            T.width_end = width_end;

            // Push the task (chunk) into the work_queue
            work_queue.push(T);

            width_start = width_end;
        }
    }

    int* buf = (int*) malloc(1 * sizeof(int));
    /* omp parallel */
    #pragma omp parallel num_threads(num_threads)
    {
        render_mandelbrot(buf, 1);
    }
}

```

Questions :

How you implement each of requested versions, especially for the hybrid parallelism.

如上方所述。

How do you partition the task?

MPI層面是用work pool。

thread層面是用一個task queue，持續去拿。

What technique do you use to reduce execution time and increase scalability?

1. MPI work pool
2. thread task queue
3. vectorization
4. 把row切成chunk已達到更好的load balance

Other efforts you made in your program

我覺得implement部分的主要effort在於有實作了這次作業建議的所有方法，如上方四點所述。

3. Experiment & Analysis

i. Methodology

(a). System Spec

使用apollo進行。

(b). Performance Metrics

使用MPI_Wtime()做主要測量。

使用time來測量整個程式執行的時間。

使用ipm來觀察MPI。

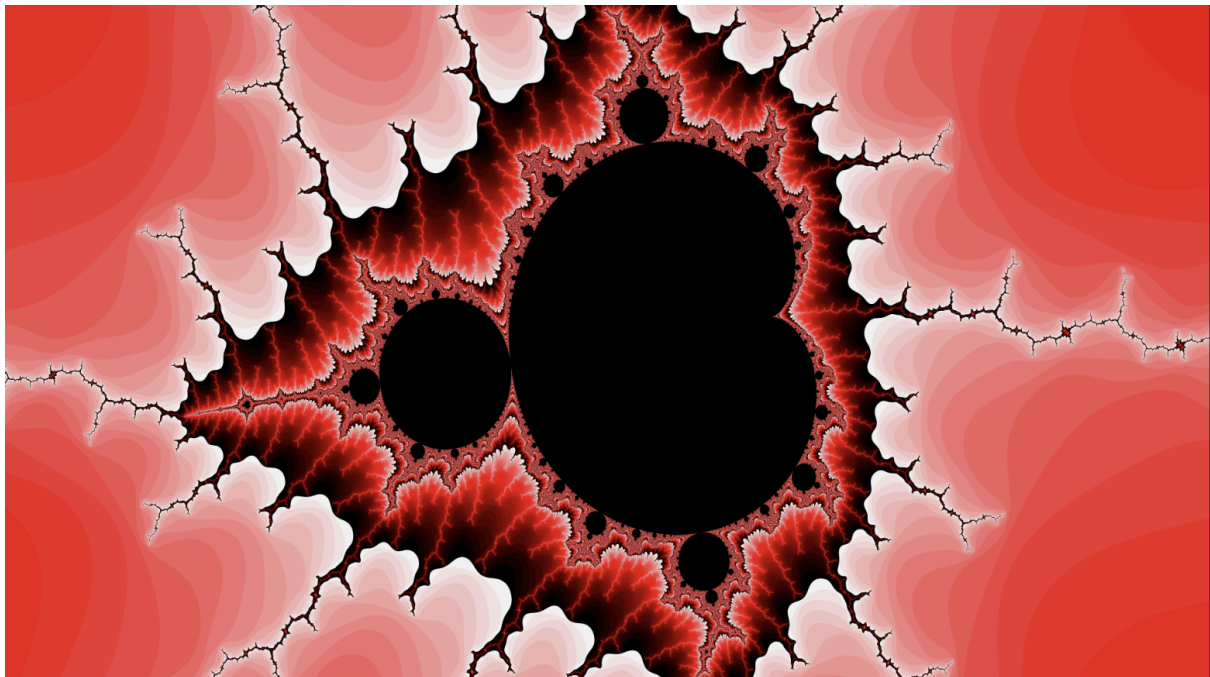
ii. Plots: Scalability & Load Balancing & Profile

Experimental Method:

Test Case Description :

選用strict 26作為test case。

因為感覺比較好看，而且整張圖比較紅。



參數為10000 -0.19985997516420825 -0.19673850548118335 -1.0994739550641088
-1.1010040371755099 7680 4320

Parallel Configurations :

分別使用以下進行實驗：

1. 多node環境(with 1 thread for each proc)：

1 node with 12 proc

2 node with 24 proc

3 node with 36 proc

4 node with 48 proc

2. 單node環境(with 1 thread for each proc)：

1 node with 1 proc

1 node with 4 proc

1 node with 8 proc

1 node with 12 proc

3. 單process環境：

1 proc with 1 thread

1 proc with 2 thread

1 proc with 3 thread

1 proc with 4 thread

問題：

sbatch 無法要12 proc with 4 threads for each proc的組合，似乎只能要3 proc with 4 threads或是12 proc with 1 thread。

原先實驗想測試1 node with 12 proc with 4 threads for each proc的但疑似無法這樣配。

所以改成用1 node with 12 proc with 1 threads for each proc的配法。

```
#SBATCH -n 12  
#SBATCH -N 1  
#SBATCH -c 1
```

Performance Measurement:

使用MPI_Wtime()去測量每個process所花費的時間。

使用time來測量整個程式執行的時間。

使用ipm來觀察communication time

共有四個metrics

1. computation time : 一個worker花費在計算的總時間
2. communication time : manager花費在recv其他worker花費的總時間
3. draw time : rank 0最後花在draw png花的時間，因為這個值會是一個constant（圖片長寬決定），跟scalability沒有關係，所以在呈現scalability上會先減去draw time以求精確。
4. rank 0 time : rank 0 執行整個程式所花的時間

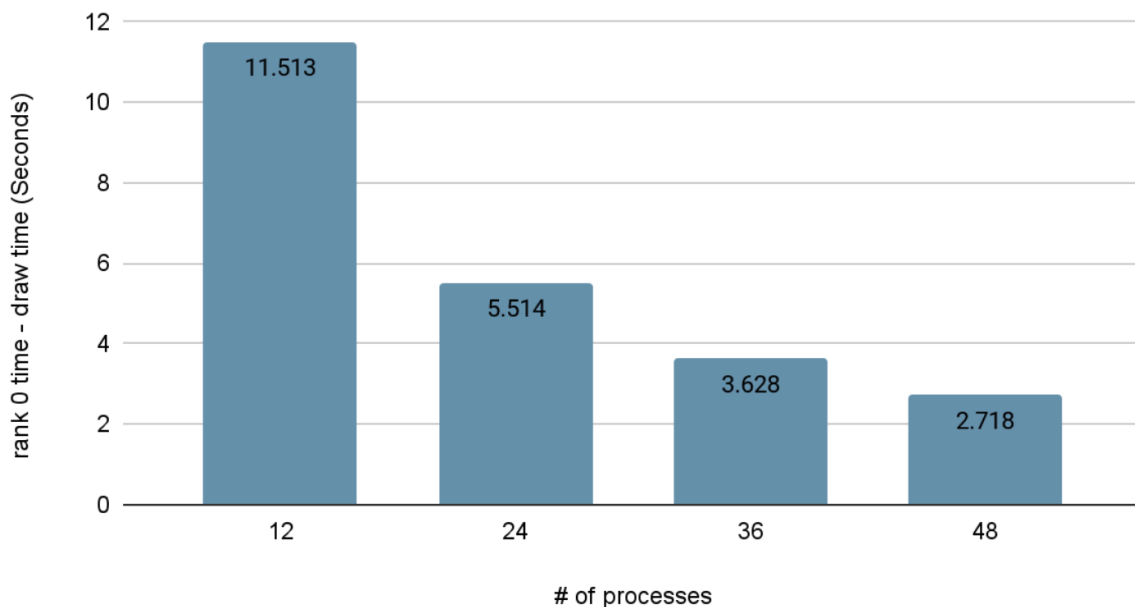
最後呈現的測量時間為rank 0 time - draw time，因為要讓程式結束一定要等所有的worker都算完並回傳好資料，所以會被bound在rank 0，看rank 0 time可以忽略其他影響time的因素，比如使用time指令來測量整個程式時會多出的constant時間，在scale小時會造成結果不精確。因此選擇以rank 0 time - draw time作為主要測量效能的metrics。

Analysis of Results:

1. 多node環境(with 1 thread for each proc)

Strong scalability :

Multi-node time profile



圖一：Multi-node time profile

Legend: speedup (blue line with markers), Ideal speedup (orange line).


# of processes	Actual speedup	Ideal speedup
12	10.988	12
24	22.944	24
36	34.871	36
48	46.546	48

從圖一中可以看出，大致上使用越多的process就能減少越多的執行時間。

MPI溝通帶來的overhead其實不高，可以從以下的ipm profile圖看出，MPI大部分時間都在做MPI_recv，而這個recv就是rank 0在等待其他worker回傳時的時間，只要worker做完就會回傳，所以基本上MPI溝通的時間會被bound在做最久的那個worker時間。但如果work非常balance的情況下，就會最小化MPI的溝通時間。

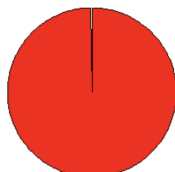
4972218

- Load Balance
- Communication Balance
- Message Buffer Sizes
- Communication Topology
- Switch Traffic
- Memory Usage
- Executable Info
- Host List
- Environment
- Developer Info



command: /home/pp23/pp23s46/homework2/.hw2b out.png 10000 -0.19985997516420825 -0.19673850548118335 -1.0994739550641088 -1.1010040371755099 7680 4320

codename:	unknown	state:	unknown
username:	pp23s46	group:	
host:	apollo32 (x86_64_Linux)	mpi_tasks:	12 on 1 hosts
start:	11/12/23/13:36:30	wallclock:	1.41933e+01 sec
stop:	11/12/23/13:36:42	%comm:	6.79138331701108
total memory:	3.331236 gbytes	total gflop/sec:	0
switch(send):	0 gbytes	switch(recv):	0 gbytes

Computation			Communication	
Event	Count	Pop	% of MPI Time	
			 <div><div>MPI_Recv</div><div>MPI_Send</div><div>MPI_Comm_size</div><div>MPI_Comm_rank</div></div>	

hw2 report

Load balance

以1 node 12 proc為例

Rank	Computation time
0	13.384884
1	11.512227
2	11.515172
3	11.512196
4	11.512648
5	11.514789
6	11.514406
7	11.512872
8	11.513720
9	11.512425
10	11.514410
11	11.512383

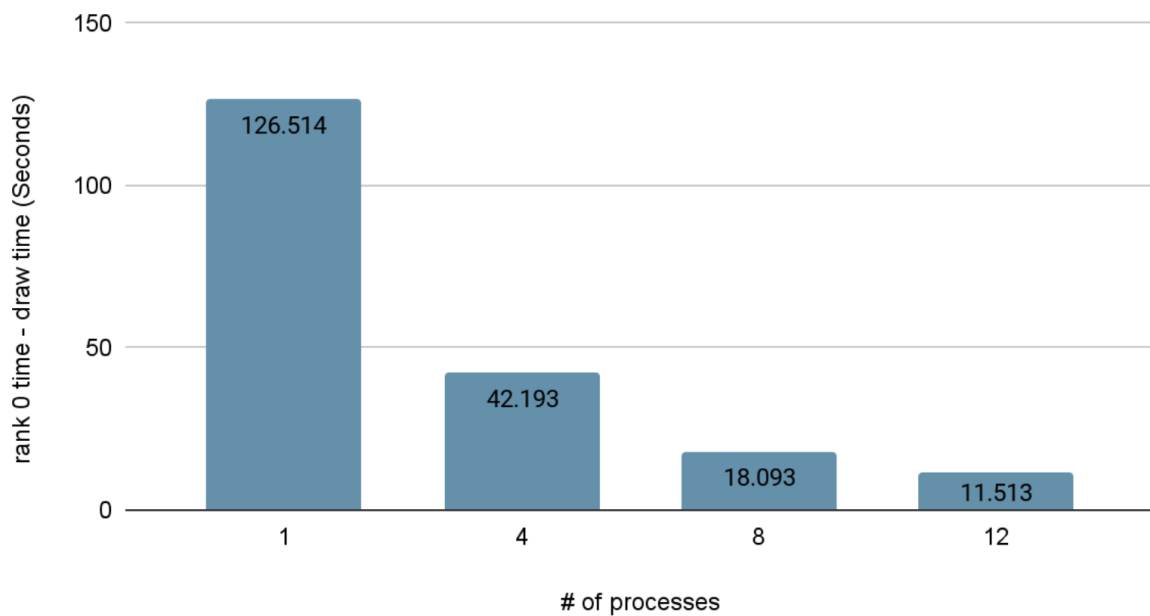
從表格中可以看到，程式的load balance效果還蠻不錯，大家都有接近的computation time，而rank 0因為負責當manager，rank 0的時間由等待其他worker回傳 + draw image組成，其中如果把draw image的時間減掉，可以得到和其他worker幾乎一樣的時間。

原因在於程式使用work pool將工作切分給每個worker，做完的人就回來繼續拿工作做，直到算完，這樣就能有效避免有某個worker要做特別久而導致大家都要等他的情況。

2. 單node環境(with 1 thread for each proc)：

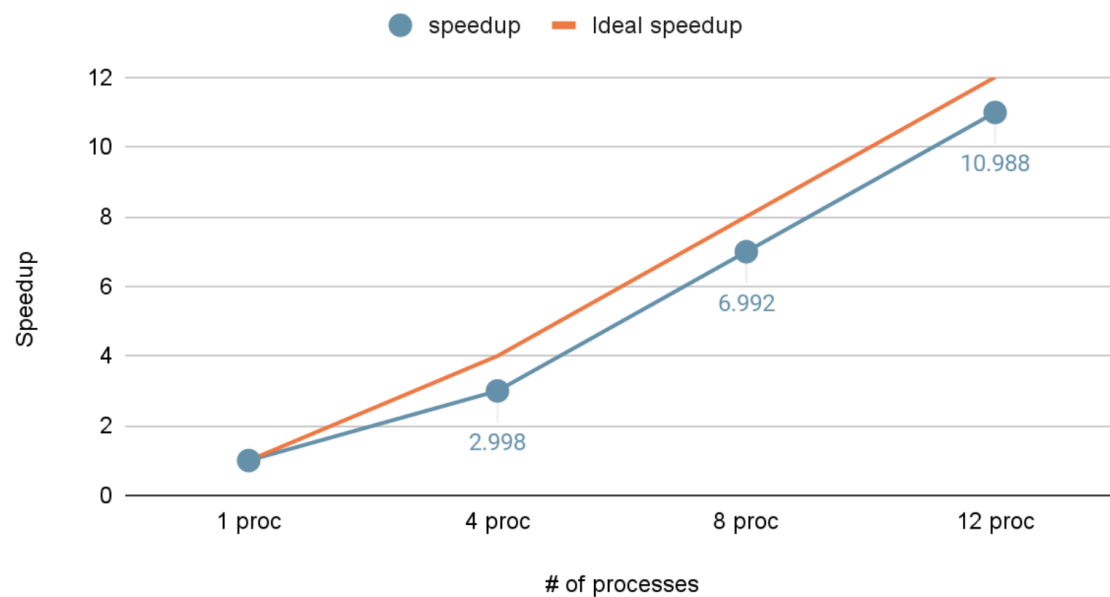
Strong scalability：

Single node time profile



圖四：single node time profile

Single node speedup chart



圖五：single node speed up chart

從上面兩張圖看出，在單node的情況大致都和在多node的情況一樣，都會隨著process數量而呈現接近線性的效能優化。

唯一較為特別的部分是，在1 proc的情況下沒有分manager和worker，但是只要proc大於1，就會切分成manager和worker，而這時rank 0會因為擔任manager而沒有辦法參與計算，所以會導致speed up其實只有增加(process數-1)。

比如用4 procs時，rank 0是manager，所以實際只有3個procs在工作，speed up也就只有2.998而已，少掉的那一個便是rank 0。

在Multi node的speed up圖中其實也有一樣的情況，只是因為scale較大比較不明顯。

Load balance

以1 node 4 proc為例

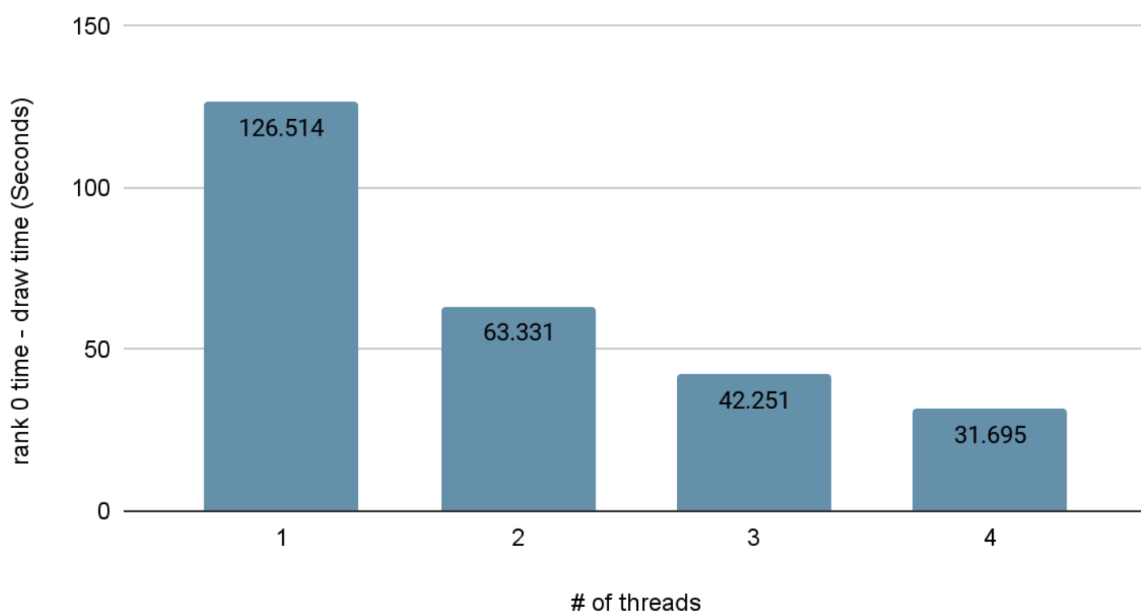
Rank	Computation time
0	45.126327
1	42.192596
2	42.192952
3	42.193372

Load相當balance，rank 0多出的部分是draw image所花的時間。

3. 單process環境：

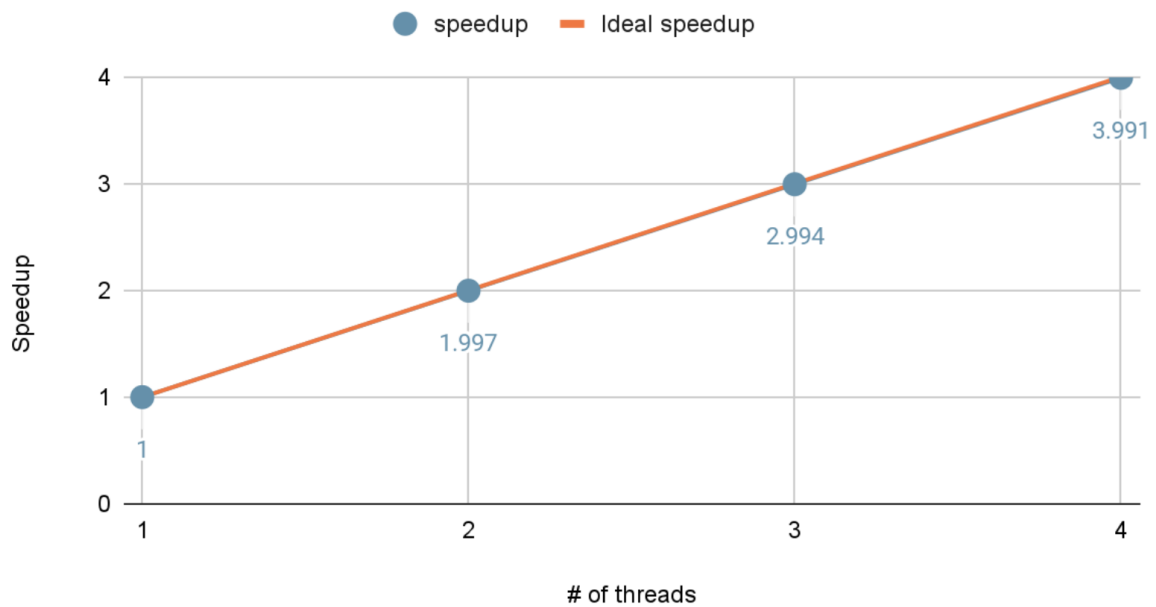
Strong scalability：

Single process time profile



圖六：single process time profile

Single process speedup chart



圖七：single process speed up chart

從圖六、圖七可以看出，thread層面的scalability效果也很好，speedup效果幾乎接近了Linear speedup。原因在於單一process使用thread時不會用到MPI work pool，不需要花一個人去當manager，所以能夠達成接近Linear speedup的笑過。

Load balance

以1 process 4 threads為例

thread	Computation time
0	33.569212
1	33.567678
2	33.567245
3	33.568394

Load相當balance。因為在thread層面使用了work_queue去記錄工作，每個thread去拿一個row來做，做完的就去要下一個row，如此便能動態分配每個人的工作，以達到load balance。

thread層面因為不用分manager，所有人都會來參與計算，所以單論計算效果時會優於使用4 processes的情況。

Optimization Strategies:

根據以上分析結果，我認為程式還能優化的部分有

1. 讓rank 0也參與部分計算工作，這樣在process scale較小的情況時才不會造成計算資源的浪費。
2. 或許能一次傳多一點row給一個worker來做，減少需要MPI溝通的次數，也許能再快一點。

下面的部分是實作這次作業時一路的優化結果：

{68 1429.96}

單純使用多個thread來跑mandelbrot set計算

{68 1429.96} --> {68 756.30}

改成用thread load balance的版本

{68 756.30} --> {68 751.28}

加上compile flag

-ftree-vectorize -march=native

{68 751.28} --> {68 564.05}

加上sse2的vectorization

{68 564.05} --> {68 514.59}

sse2中會去清空算完的channel的值

{68 514.59} --> {68 493.27}

把length_squared.d2 = _mm_add_pd(z_reals_squared, z_imags_squared);幫到上面算

減少重複算

```
__m128d new_z_reals_squared = _mm_mul_pd(z_reals.d2, z_reals.d2);
```

```
__m128d new_z_imags_squared = _mm_mul_pd(z_imags.d2, z_imags.d2);
```

{68 493.27} --> {68 403.75}

channel load balance

{68 380.94}

MPI work pool

{68 380.94} --> {68 375.68}

chunk size = 10

iii. Discussion (must base on the results in the plots)

(a). Compare and discuss the scalability of your implementations.

根據圖二、圖五、圖七，可以看出程式的scalability效果很不錯，幾乎接近Linear speedup，唯獨在多process中，也就是有使用到MPI work pool的case下，會因為要有一個manager而少掉1個process的計算資源。

而在thread的層面上，因為都會參與計算工作，所以speedup效果又再好了一些。

(b). Compare and discuss the load balance of your implementations.

根據上方三個load balance的表格，可以看出程式相當load balance，原因在於MPI層面使用了MPI work pool的機制，將圖片切成一個個row丟給每個worker去做，做完的就再回來要一個row，如此動態的分工作能讓process之間load balance。

而在thread層面使用work_queue，將收到的row切成一個個size為10的chunk，每個thread一次拿一個chunk來計算，算完就再拿下一個，動態的分配工作讓thread之間load balance。

iv. Others

另外做了多node多process多thread的混合實驗，想看這些全部加起來的平行效果如何。

多node多proc多thread環境：

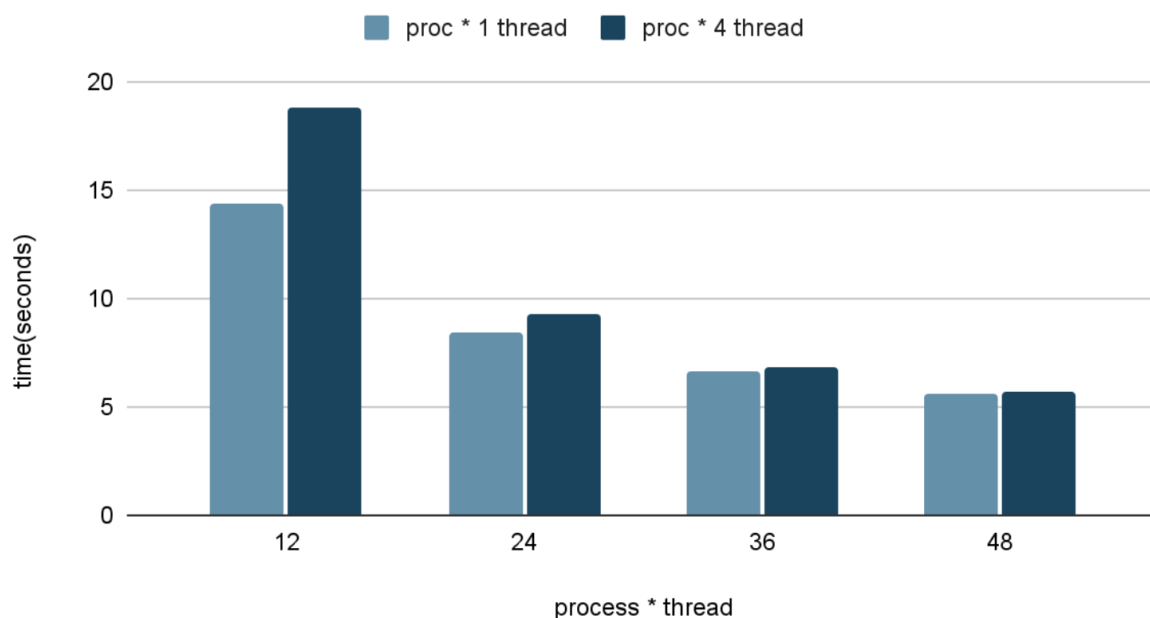
3 proc with 4 threads

6 proc with 4 threads

9 proc with 4 threads

12 proc with 4 threads

1 thread vs 4 thread



第一筆是12 proc * 1 thread 和 3 proc * 4 threads的對比，後面依此類推。

可以發現基本上process較多者會效能較好，是因為rank 0都會固定變成manager，所以在case 1，3 proc * 4 threads下，會直接少掉一個可以用的process，導致少了一些計算資源。

work pool有這個實務上的小缺點，當process scale較小時會浪費不少計算資源。

Experience & Conclusion

本次作業我覺得難度上比上次難蠻多的，這次要實作的功能有很多，而且還要做兩個版本，雖說基本上hybrid可以沿用pthread版本的大多東西，但實作細節上有蠻多小地方還是要修改。這次作業充滿了挑戰性。

這次實作最主要學到以下幾個東西：

1. MPI work pool的實作方式
2. thread的dynamic work allocation
3. vectorization
4. coding上的技巧

遇上最困難的部分我覺得是vectorization，它的資料型態變得非常難以閱讀，比如__m128d之類的，而且語法上也蠻難理解，後來是去網路上找了很多參考資料才大概了解怎麼用的，然後花費了幾乎一整天的時間才把cal_pixel改成cal_pixel_sse2。

最後，感謝幫忙改這篇report的助教，謝謝！