HW2 Report

111065541 張騰午

Implement

Hw2a pthread

1. Static

一開始以 sched_getaffinity 偵測可用的 CPU 資源並接收參數，依照參數 allocate memory for image，此外預先計算每個 thread 的運算範圍，

```
unsigned long long step = height / num_threads;
unsigned long long rest = height % num_threads;
/* assign argument */
for(int i=0;i<num_threads;i++){
    if(i < rest){
        thr[i].start = i * step + i;
        thr[i].end = i * step + i + step+1;
    }else{
        thr[i].start = i * step + rest;
        thr[i].end = i * step + rest + step - 1+1;
    }
    thr[i].thread_id = i;

}
 for(int i=0;i<num_threads;i++){
     pthread_create(&threads[i], NULL, calculatePixels, &thr[i]);
 }
```

接著在迴圈中 pthread_create，create pthread 後進入 calculatePixels 函數。calculatePixels 函數中使用靜態分配的方式分配他們的起始與結束的 row。

```
for (int j = temp_args->start; j < temp_args->end; ++j) {
    double y0 = j * ((upper - lower) / height) + lower;
    for (int i = 0; i < width; ++i) {
        double x0 = i * ((right - left) / width) + left;
```

之後就在該 row 中逐一計算每個值，更新在 image[start_j*width+i]中，直到 pthread_exit，最後 pthread_join 並且 write_png 完成程式。

2. Dynammic(no vectorization)

一開始以 sched_getaffinity 偵測可用的 CPU 資源並接收參數，依照參數 allocate memory for image，接著在迴圈中 pthread_create，create pthread 後進入 calculatePixels 函數。

```
for(int i=0;i<num_threads;++i){
    pthread_create(&threads[i], NULL, calculatePixels,NULL);
}
```

calculatePixels 函數中使用動態分配的方式以 mutex_lock 的方式分配要運算的 row(將 cur_r 分給 thread 各自的 start_j)給每個 thread，得到的 start_j 就是接下來要運算的 row，若已經分完所有的 row 則將 start_j 設為 height，之後便能跳出 while 迴圈。

```
while(start_j < height){
    pthread_mutex_lock(&mutex);
    if(cur_r < height){
        start_j = cur_r;
        cur_r++;
    }else{ start_j = height;}//go break
    pthread_mutex_unlock(&mutex);
```

之後就在該 row 中逐一計算每個值，更新在 image[start_j*width+i]中，直到 pthread_exit

```
            for(i=0; i < width; ++i){
                x0 = i * drow + left;
                repeats = 0;
                x = 0;
                y = 0;
                length_squared = 0;

                while (repeats < iters && length_squared < 4) {
                    temp = x * x - y * y + x0;
                    y = 2 * x * y + y0;
                    x = temp;
                    length_squared = x * x + y * y;
                    ++repeats;
                }
                image[start_j * width + i] = repeats;
            }
        }
    }

    pthread_exit(NULL);
}
```

最後 pthread_join 並且 write_png 完成程式。

```
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);
```

3. Dynammic (vectorization)

先#include <emmintrin.h>

在 calculatePixels 加入 SSE register 用 vectorization 將 thread 要計算的 pixel 同一個 row 內兩兩包成一組，一樣計算 x,y,x0 等但現在改用 SSE register

```
__m128d x_vec = _mm_set_pd(0, 0);
__m128d x0_vec = _mm_load_pd(x0);
__m128d y_vec = _mm_set_pd(0, 0);
__m128d length_squared_vec = _mm_set_pd(0, 0);
```

每次進行 repeat 兩個值計算前，須先判斷是否 repeats2[0] < iters 以及 length_squared_vec 值小於 4，因為 length_squared_vec 現在有兩個值要判斷兩次，第二次須_mm_shuffle_pd 將第二個值 shuffle 出來才能進行判斷，若上述條件違反了則 lock 上鎖，repeat 值不能再更新。

```
while (!lock2[0] || !lock2[1]){
    if (!lock2[0]){
        if (repeats2[0] < iters && _mm_comilt_sd(length_squared_vec, vec_four)) { ++repeats2[0];}
        else {lock2[0] =true;}
    }
    if (!lock2[1]){
        if (repeats2[1] < iters && _mm_comilt_sd(_mm_shuffle_pd(length_squared_vec, length_squared_vec, 1), vec_four)) { ++repeats2[1];}
        else { lock2[1] =true;}
    }
```

之後 repeat 計算完成後再繼續更新各值

```
__m128d tmp_vec = _mm_add_pd(_mm_sub_pd(_mm_mul_pd(x_vec, x_vec), _mm_mul_pd(y_vec, y_vec)), x0_vec);
y_vec = _mm_add_pd(_mm_mul_pd(_mm_mul_pd(x_vec, y_vec), vec_two), y0_vec);
x_vec = tmp_vec;
length_squared_vec = _mm_add_pd(_mm_mul_pd(x_vec, x_vec), _mm_mul_pd(y_vec, y_vec));

image[start_j * width + i] = repeats2[0];
++i;
image[start_j * width + i] = repeats2[1];
```

Hw2b hybrid(MPI IO+openMP)

1. no vectorization

先 MPI_Init，判斷 height 是否小於 process 數量，若是小於代表有多餘的 process 則用 MPI_Comm_create 新的 group，只留下和 height 一樣多的 process

```
if (height < size)
{
    MPI_Comm_group(MPI_COMM_WORLD, &WORLD_GROUP);
    int range[1][3] = {{0, height - 1, 1}};
    MPI_Group_range_incl(WORLD_GROUP, 1, range, &USED_GROUP);
    MPI_Comm_create(MPI_COMM_WORLD, USED_GROUP, &USED_COMM);
    if (USED_COMM == MPI_COMM_NULL)
    {
        MPI_Finalize();
        return 0;
    }
    size = height;
}
```

每個 process 定義自己計算的 img_範圍

```
int step = ceil((double)height / size);

int *img_ = (int *)malloc(step * width * sizeof(int));
```

每個 process 接著進入 calculatePixels 函數計算自己的 img_數值，row 的部分用靜態方式分配每隔一個 size 數量分給一個 process(round robin 方式)，而 column 部分則用 openMP 動態方式分配給 process 中不同 thread 執行

```
for (int j = rank; j <height; j += size){
    y0 = j * d_y + lower;
    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < width; ++i){
```

最後 Gather 回來所有 process 的 img_，彙整成 image，由 rank=0 的 process write_png

```
image = (int *)malloc(size * step * width * sizeof(int));
MPI_Gather(img_, step * width, MPI_INT, image, step * width, MPI_INT, 0, USED_COMM);

/* draw and cleanup */
if (rank == 0){
    write_png(filename, iters, width, height, image, step);
}
```

Write_png 部分要注意，因為現在 process 計算 row 的順序採 round robin，寫入 png 的順序也必須修改

```
for (int y = 0; y < height; ++y) {
    memset(row, 0, row_size);

    for (int x = 0; x < width; ++x){
        int p = buffer[(y % size * step + y / size) * width + x];
        png_bytep color = row + x * 3;
```

2. **vectorization**

一樣先#include <emmintrin.h>

之後在 CalculatePixels 函數裡面進行向量化，同樣在分配好 row 後採兩兩一起計算，加入 SSE register 用 vectorization 將 thread 要計算的 pixel 同一個 row 內兩兩包成一組進行計算。不過由於分配 column 是由 openMP 動態分配給 threads，這裡讓迴圈一次前進兩步(i+=2)，確保相鄰的 column 都給同一個 thread，而不會發生重複計算。

```
#pragma omp parallel for schedule(dynamic, 1)
for (int i = 0; i < width-1; i+=2){
```

Experiment & Analysis

i、 Methodology

a. System Spec

所有程式都在課程提供的apollo.cs.nthu.edu.tw平台上進行測試

b. Performance Metrics

pthread

使用 strict 30 測資測試

iter=10000 x0=-0.3421054598064634 x1=-0.2373971478909443 y0=-0.6373595233099365 y1=-0.6884105743609876 w=7680 h=4320

使用 clock_gettime(CLOCK_MONOTONIC, ...)在 pthread 程式部分取得時間之後計算時間。用這個測量程式的 runtime 及每個 thread 的 time，再將所有 thread 平均，觀察 load balance 程度。

```
struct timespec start, end, temp;
double time_used;
clock_gettime(CLOCK_MONOTONIC, &start);
```

```
    clock_gettime(CLOCK_MONOTONIC, &end);
    if ((end.tv_nsec - start.tv_nsec) < 0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
    temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
    } else {
    temp.tv_sec = end.tv_sec - start.tv_sec;
    temp.tv_nsec = end.tv_nsec - start.tv_nsec;
    }
    time_used = temp.tv_sec + (double) temp.tv_nsec / 1000000000.0;
    printf("%f second\n", time_used);
```

Hybrid

使用 strict 27 測資測試

iter=10000 x0=0.27483841838734274 x1=0.4216774226409377 y0=0.5755165572756626 y1=0.5039244805312306 w=7680 h=4320

使用 MPI_Wtime();來測量時間，在 MPI_Init(...)後與 MPI_Finalize()前加上 MPI_Wtime()後將兩個值 相減可以得到整的程式的時間。另外在 omp 動態分配的部分用 omp_get_wtime()加在 thread 要計算的程式的前後再相減即可得到 thread 所花的時間並加入預先定義好存每個 thread 時間的 array。

```
#pragma omp for schedule(dynamic, 1)
for (int i = 0; i < width; ++i) {
    int thread_id = omp_get_thread_num();
    double start_times = omp_get_wtime();
```

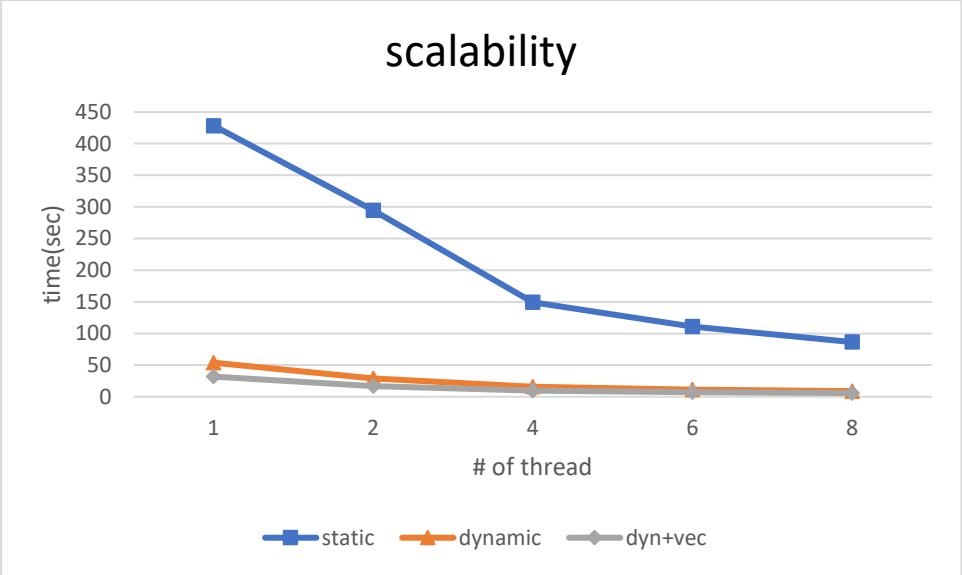
```
        double end_times = omp_get_wtime();
        double tmpt = (end_times - start_times);
        thr_times[thread_id] += tmpt;
    }
}
```

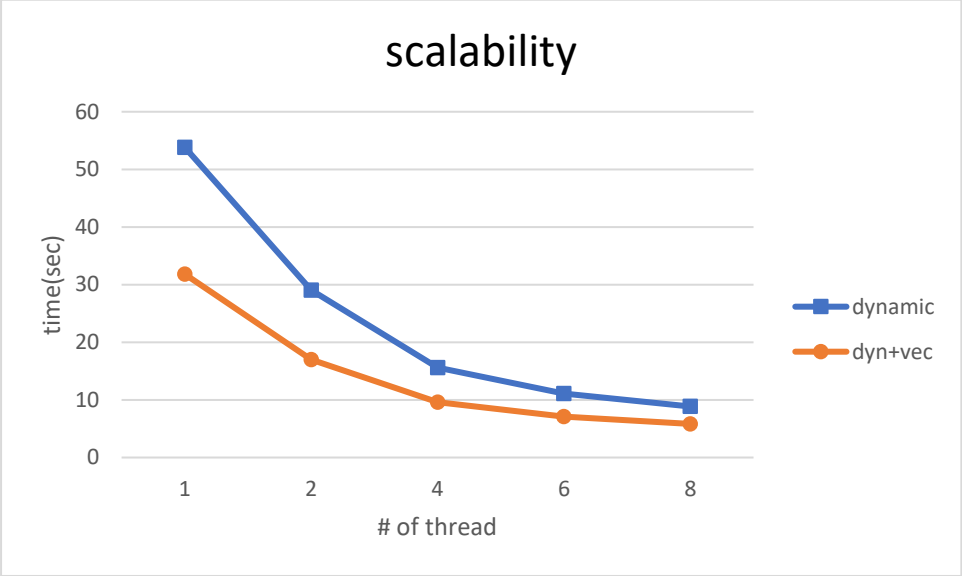
ii、 Plots: Scalability & Load Balancing & Profile

1. Strong scalability

pthread

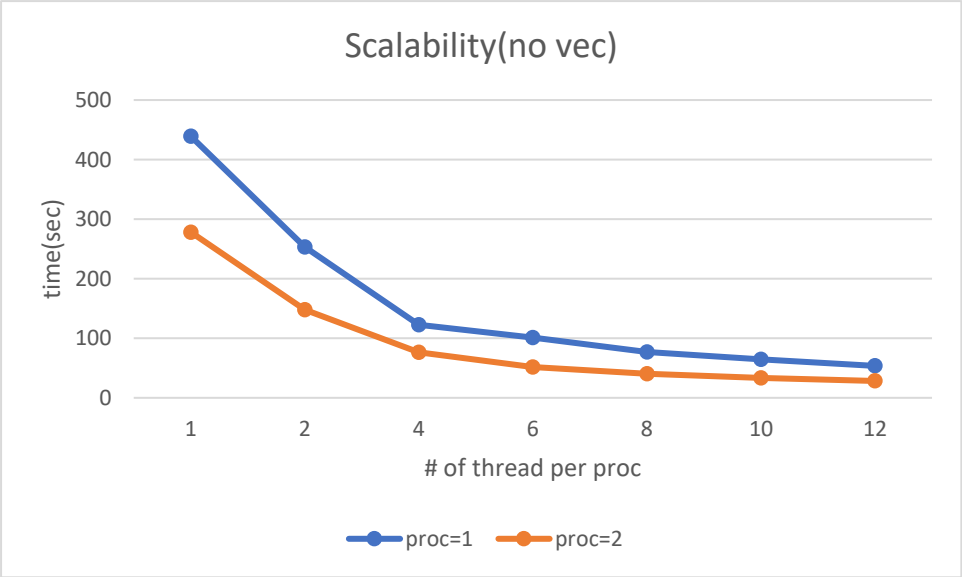
第一張圖比較 static, dynamic, dynamic + vectorization 三者，使用 static 方法在任何 thread 數量下時間都會遠高於其他兩個，所以再將其他兩個單獨拉出來比較。二者 strong scalability 狀況都不錯，向量化兩兩一起計算後幾乎提升了兩倍的速度。效果很不錯。

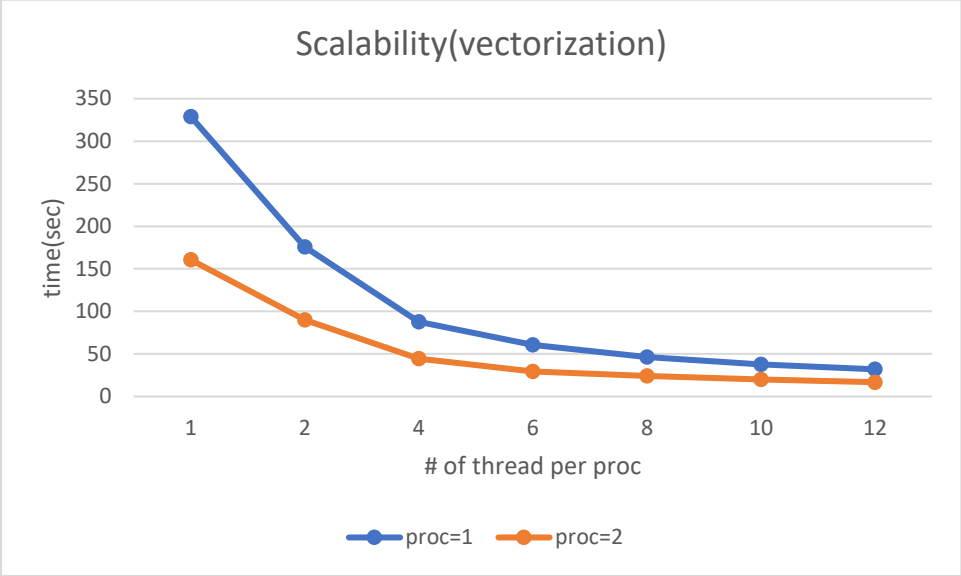




Hybrid

接下來比較 hybrid 部分，第一種是並未做 vectorization 的版本與做了的版本，兩者在 strong scalability 方面皆展現了不錯的表現。

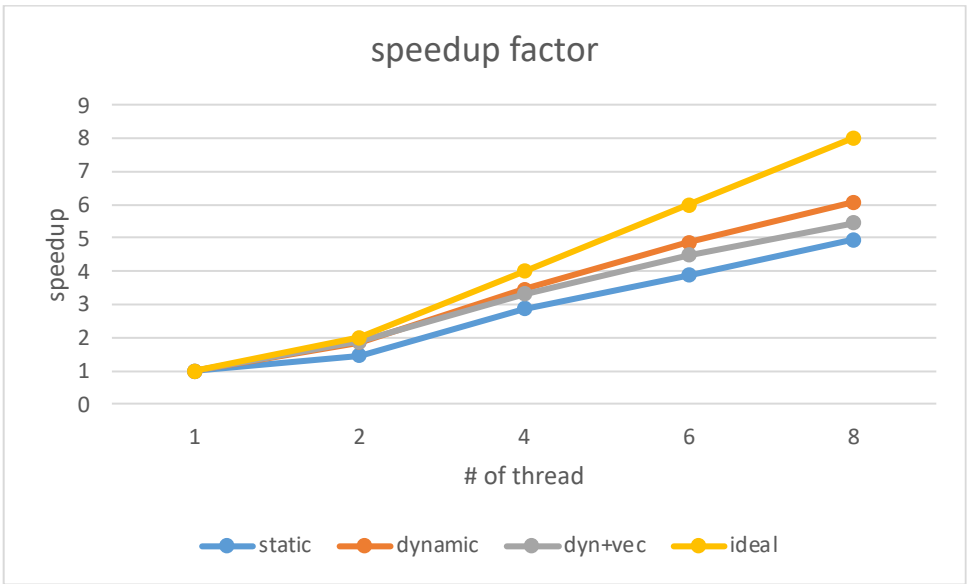




2. Speedup Factor

pthread

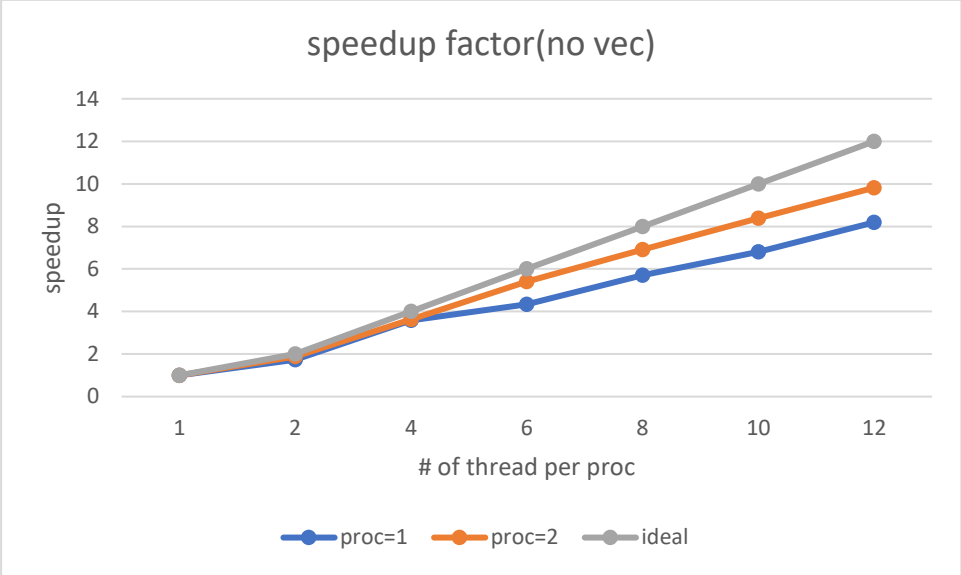
第一張圖比較 static, dynamic, dynamic + vectorization 三者，其中 static 的 speedup 是最差的，dynamic+向量化雖然速度較快但 speedup 方面在 thread 數量多於 4 之後略低於 dynamic。另外超過 4 個 thread 後和 ideal 的差異也逐漸顯現。

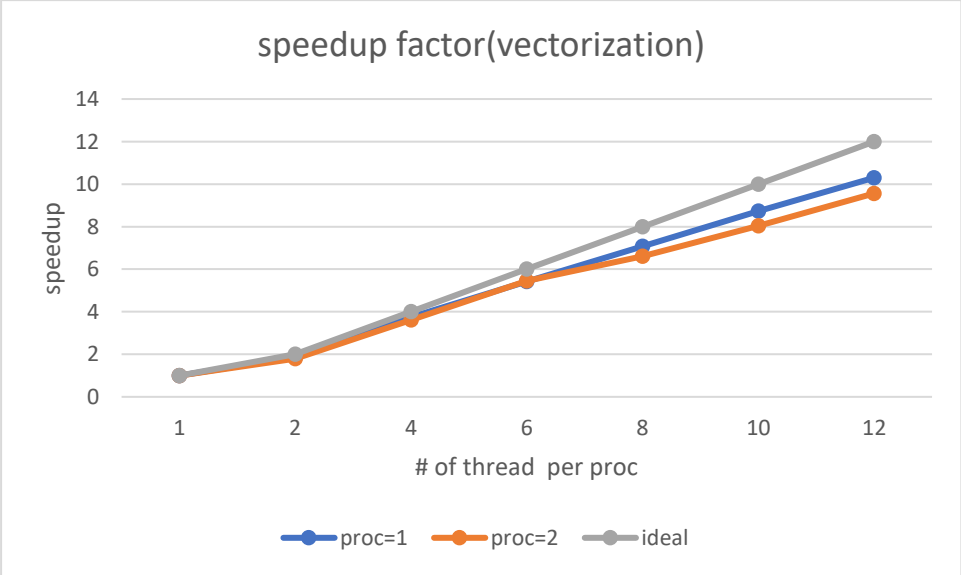


Hybrid

接下來比較 hybrid 部分，未做 vectorization 的版本與做了向量化的版本，兩者在 proc 數量等於 1 或 2 下的 speedup 表現。兩者都在 thread 數量超過 6 時和 ideal 漸漸出現較大差異。整體講 vectorization 版本在 speedup 上表現更好更貼近 ideal 值。

理論上 proc 數量多時因為分配 row 時還是使用靜態的方式做分配，分配完後 row 內的 column 才是 omp 進行動態分配，所以可能還是會有因分配 row 計算難度不同，導致先計算完的 proc 要等待尚未完成的 proc，拖慢整體速度的問題，因此 proc 增加時可能 speedup 不會那麼好。不過做實驗之後只有 vectorization 的版本有這個現象，no vectorization 的版本似乎並不明顯，可能要進一步更多實驗才能驗證。



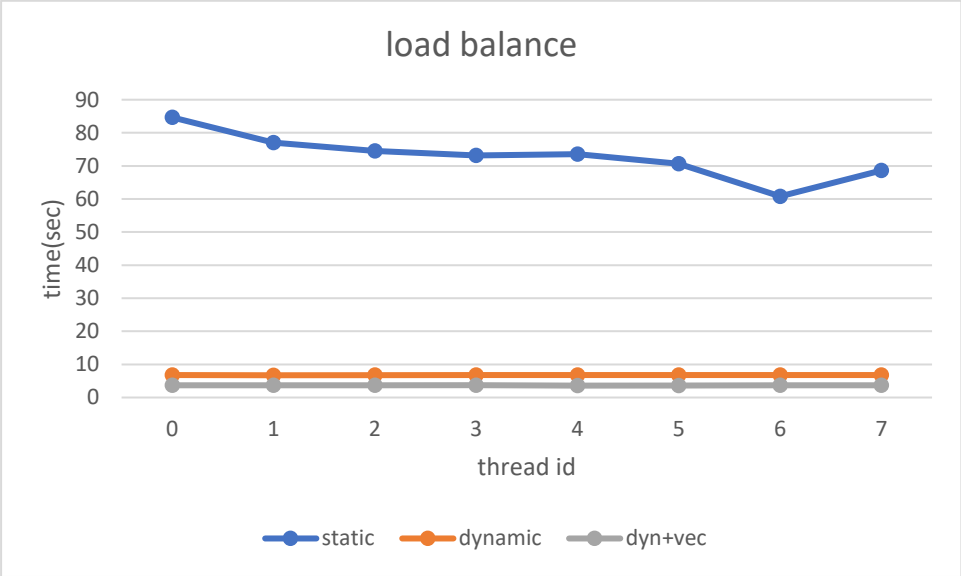


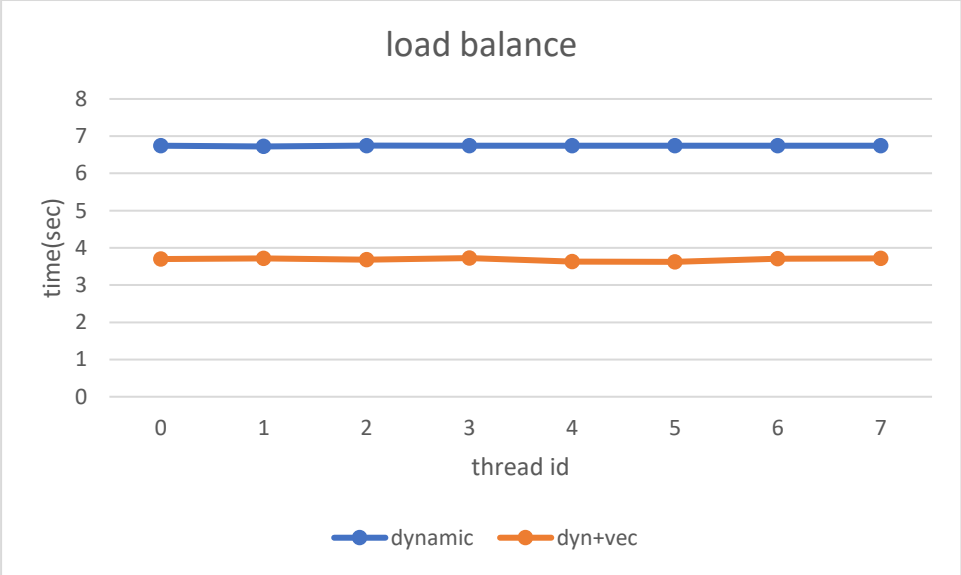
3. Load Balance

Pthread

Static 的方法明顯 load balance 不太好，最久和最快的 thread 差異高達 20 秒以上，把使用 dynamic 的兩個版本結果拉出來看，可以發現他們的 load balance 狀況不錯，各個 thread 的執行時間近乎相同，表示每個 thread 的

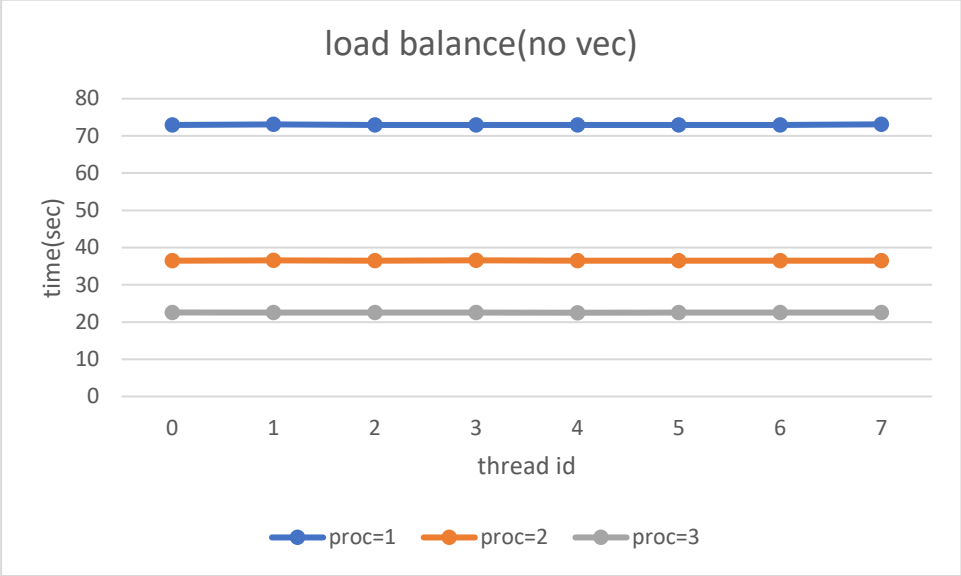
loading 很平均。

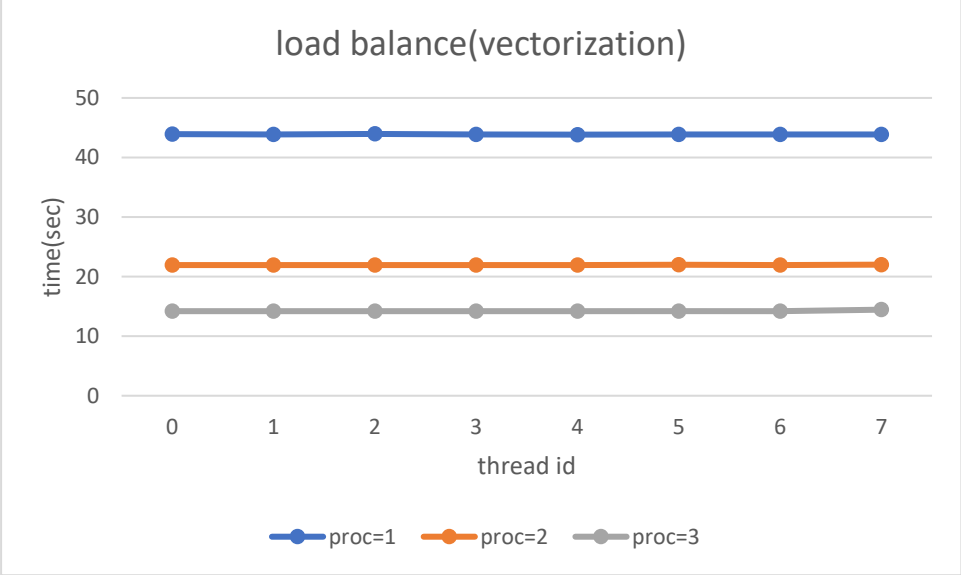




Hybrid

下圖是 Hybrid 有無向量化兩個版本在不同 procs 數量下每個 thread 的 loading 狀況。因為都使用動態分配，load balance 狀況很好，各個 thread 的執行時間近乎相同，在 1 到 3 個 procs 數量狀況下每個 thread 的 loading 都很平均。

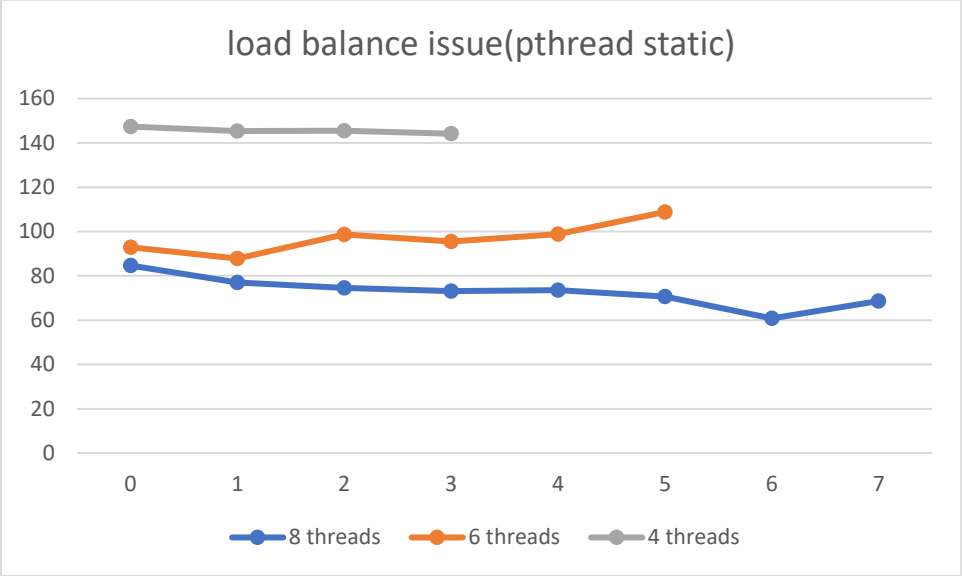




iii、 Others

Pthread

由於 pthread static 版本有比較嚴重的 load balance 問題，這邊測試增加 thread 數量能否減緩 load balance 問題，結果如下圖 6,8 threads 時 load balance 並沒有減緩反而有加劇跡象，增加 threads 數量並不會減緩 load balance 問題。



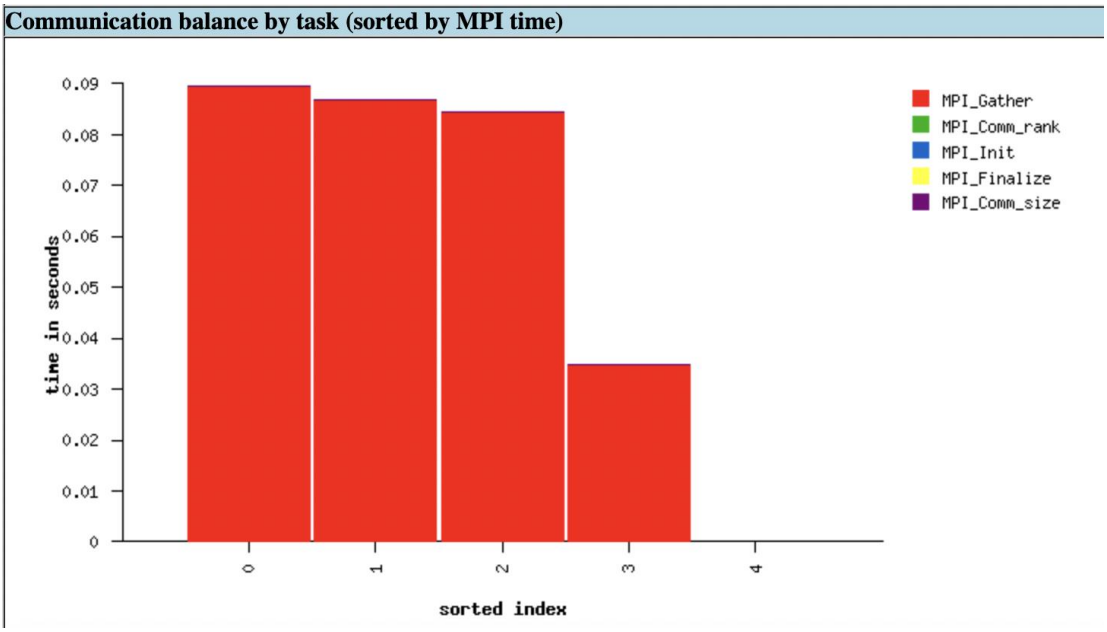
Hybrid

Hybrid 方面另外使用了 ipm profiler 進行分析 strict27 testcase，基本上這次程式花的 MPI 指令時間幾乎都在 MPI_Gather 上面，而最後一個 process 明顯花的時間又少於前面三個 process。

推測是因為使用 round robin 的方式靜態分配 row(如下圖)，

```
for (int j = rank; j <height; j += size){
    y0 = j * d_y + lower;
    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < width; ++i){
```

，若 row 數量無法被 process 數量(size)整除時，後面的 process 可能會分配到比較少的 row 的緣故。可見這種分配方式仍有一定的 load balance 問題存在。



iv、 Discussion

綜合以上測試結果與圖表顯示，scalability 方面隨著 threads 增加，pthread 大約到 threads 數量 4 以上 speedup 就會開始下降，hybrid 版本則是約 4~6 threads per proc 以上 speedup 趨勢就會下降，應該與一些未分配出去的操作時間有關(例如 write_png)，而 hybrid 版本則是來自於分 row 後才用 omp 動態分配 row 上每個 column。另外做 vectorization 也能在 speedup 上表現更好更貼近 ideal 趨勢。

而 load balance 部分可以發現只要使用了動態分配，整體就會達到不錯的 load balance，和使用靜態分配形成很大的對比。 而靜態分配時增加 threads 數量並沒有辦法解決 load balance 問題，但是例如 hybrid 時可以透過例如每隔一個 size 分配一個 row 給一個 thread 這種類似 round robin 的方式分配只在 column 分配時使用 omp 動態分配，這樣依然會有不差的 load balance。

Experience & Conclusion

如果只是要有個可行的方法這次作業以 sequential 版本改寫起來並不會太久，只是與第一次作業不同的地方在這回有不少不同的策略，使用靜態、動態分配方式，還有向量化，由於這次問題明顯存在不少 load balance 議題，有的點要 repeat 很多次有得很快能結束，所以動態分配格外重要，但我還是在 pthread 部分寫了一個靜態分配版本程式對照，和動態分配的差異也非常顯著，是只使用動態分配版本的數倍以上。
因為使用了動態分配解決 load balance 問題，scalability, speedup 效果都算不錯，不像 hw1 時沒有考慮這些因此 speedup 效果並未隨著 process 增多而顯著增加。另外這次作業也還是有些困難的地方，主要在 debug 不易，有試過一些方式發生 judge 時大部分通過只有少數幾個 case 以 9X%接近合格通過的比率被判定 wrong answer，但對於這種狀況很難 debug 到底圖片錯在哪裡，如果無法直接從程式邏輯上糾錯，很難以數百萬 pixels 的 png 圖片觀察錯誤在哪。另外在 vectorization 上面套用的方式沒有很多但也有嘗試使用 union 的概念但和同學討論並嘗試後也沒有比較好，最後還是用了現行方法，這次作業好處是有很多的討論空間也更了解了 load balance 的概念。