Parellel Programming HW2

109062113 葛奕宣

Implementation

Pthread:

我先透過 CPU_COUNT 取得可使用的 thread 數量，初始化我的 pthreads，接著我將 row 平均分配到這謝 threads 上，分配方法是 for (int j = thread_id; j < height; j += num_threads)，可以確保盡量平均的分配，而 col 的計算我有使用 vectorization 優化。

```
for (int j = id; j < height; j += num_threads) {
    double y0 = j * y_scale + lower;
    __m128d v_y0 = _mm_load1_pd(&y0);
    for (int i = 0; i < width; ++i) {
        if(i+1<width){
            double x0[2] = {i * x_scale + left, (i + 1) * x_scale + left};
            __m128d v_x0 = _mm_load_pd(x0);
            __m128d v_x = _mm_setzero_pd();
            __m128d v_y = _mm_setzero_pd();
            __m128d v_sq_x = _mm_setzero_pd();
            __m128d v_sq_y = _mm_setzero_pd();
            __m128i v_repeat = _mm_setzero_si128();
            __m128d v_length_squared = _mm_setzero_pd();
            int repeats = 0;
            while(repeats < iters){
                __m128d v_cmp = _mm_cmpgt_pd(v_4, v_length_squared);
                //if two > 4 break
                if (_mm_movemask_pd(v_cmp) == 0)
                    break;
                repeats++;
                __m128d temp = _mm_add_pd(_mm_sub_pd(v_sq_x, v_sq_y), v_x0);
                v_y = _mm_add_pd(_mm_mul_pd(_mm_mul_pd(v_x, v_y), v_2), v_y0);
                v_x = temp;
                v_sq_x = _mm_mul_pd(v_x,v_x);
                v_sq_y = _mm_mul_pd(v_y, v_y);
                v_length_squared = _mm_or_pd(_mm_andnot_pd(v_cmp, v_length_squared), _mm_and_pd(v_cmp, _mm_add_pd(v_sq_x, v_sq_y)));
                v_repeat = _mm_add_epi64(v_repeat, _mm_srli_epi64(_mm_castpd_si128(v_cmp), 63));
            }
            _mm_storel_epi64((__m128i*)(image + j*width+i), _mm_shuffle_epi32(v_repeat, 0b01000));
            i++;
        }
        else {
```

我依照 sequential code 將原本的變數都宣告成了__m128d，同時存了兩個 double，並將基本的四則運算用_mm operation 代替，而在 while 迴圈，我只判斷了 repeats 有沒有超過 iters，因為我在迴圈內用了 cmpgt 同時比較了兩個 length_squared，如果 compare 的結果都不符合則會 break，只要有一方符合就會繼續做下去，每一次迴圈都會計算新的 x 跟 y 並增加 repeats，這裡我觀察到平方被計算了兩次，所以我開了一個變數讓平方的值記下來。而在 v_repeats(存兩個單元個別的 repeats)的計算，我用 compare 的結果往右移 63 並 add_epi64，舉例來說原本是各 64’ 的 000…001111…11 會變成 000…000 0000…001，就可以正確增加。

而在 length_squared 的計算就比較複雜，我為了不去多做 if else 的判斷，所以直接用 compare 出來的結果配上 bitwise operation 去更新每一次的值，並讓停止更新的值能維持一定。

最後他們都會用_mm_storel_epi64 把值存入 share 的 image。這裡有個點是要注意 width 可能是奇數，要多一個判斷。

最後就是做 pthread_join 並 release memory。

Hybrid:

首先要去判斷 process 數量有沒有超過 height，如果有可以先將多餘的 finalize。

接著我用 process 分配 row，各別的 row 跟 col 都用 omp 的 parellel for 進行優化，而 width 的計算同樣使用 vectorization。

為了節省 memory 使用，我會先計算 height/world_size，得到每個 process 分配到的 row 數量，並開 image 為此大小用來存各個 process 運算的結果。最後再用 MPI_Gather 把 image 都匯集到 rank 0 的 final image，這裡會碰到一個問題是匯集回來的 final_image 內的資料順序不是照 row 的，因為假設 process = 2, height = 5，process 0 會分配到 row 4,2,0 ，process 1 會分到 row 3,1，最後匯集回來會變成 42013，而 write_png 的 buffer 讀取就要做調整，開一個新的變數每次增加 process hold 的 row 數量就好。

Experiment & Analysis

System Spec:

Apollo server

Performance Metrics:

For Pthread, I use clock_gettime to calculate the compute time, the time spend in computing mandlebrot set including pthread create and join. Using time command to get the total process time.

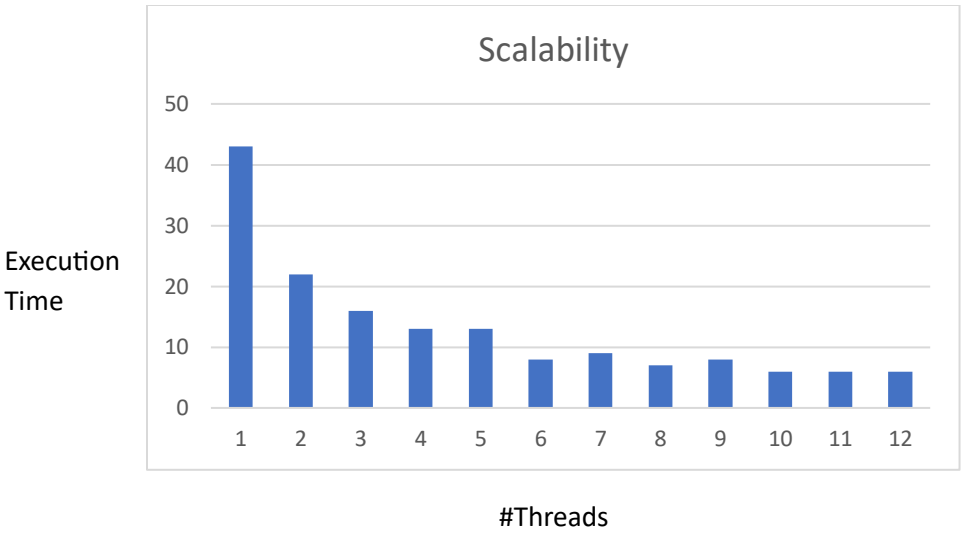
For Hybrid,

Pthread experiment & Analysis:

For testcase, I tune the slow1.txt to have a faster testcase. I use bash script to run 1 to 12 thread to test scalability.

```
#!/bin/bash

for c in {1..12}; do
    echo "$c thread"
    srun -pjudge -c$c time ./hw2a ./exp01.png 17417037 -0.789472 -0.7825277 0.145046 0.148953 2500 1200
done
```



We can see the time scales down according to the threads number, but when the threads exceeds 8 there is less significant reduction, it may cause by the deviation since the execution time is less than 7s.

```
[pp23s09@apollo31 hw2]
16.032455 second
21.337674 second
21.342745 second
21.366630 second
26.601344 second
31.884691 second
37.110627 second
37.147425 second
37.154370 second
42.420871 second
47.617206 second
58.175967 second
```

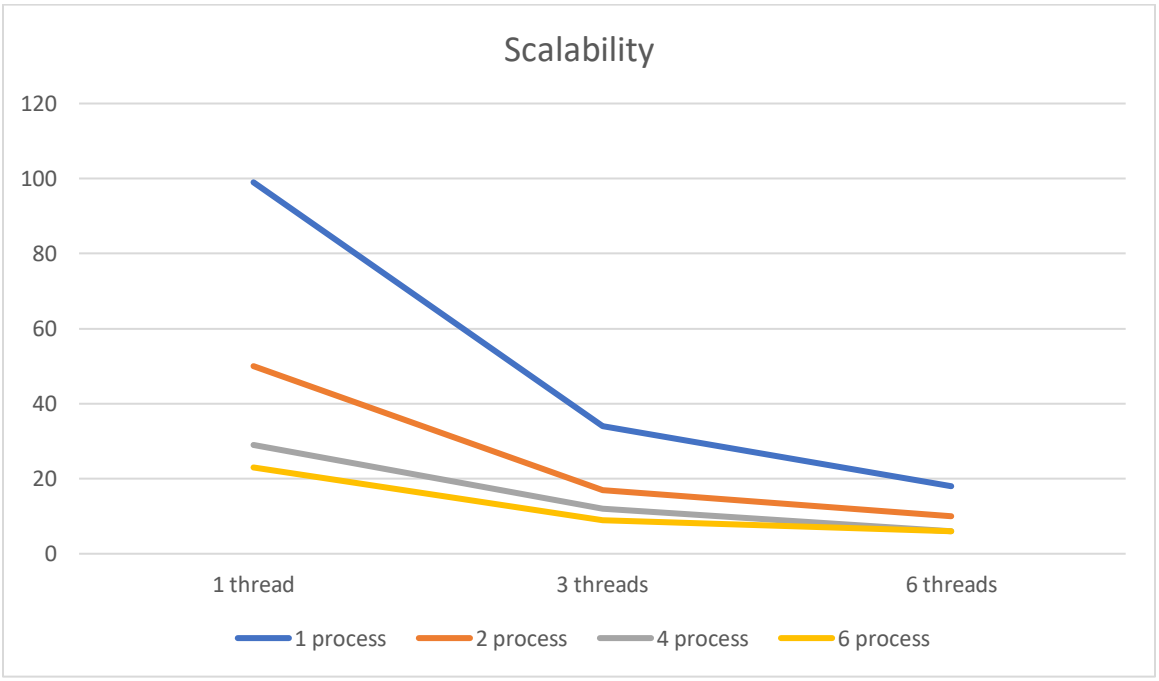
For the load-balancing, I use clock_gettime to calculate each thread's execution time. In above figure, the 12 number represents 12 thread's execution time. We can see they're not really balanced, the reason may comes from each row has different calculate complexity. So I will conclude that my pthread strategy is not load-balanced.

Hybrid experiment & Analysis:

I use slow1.txt as my testcase. And a test script iterate through 1,2,4,6 process with 1,3,6 threads. I use ipm to get the Total execution time.

```
#!/bin/bash

for p in {1,2,4,6}; do
    for c in {1,3,6}; do
        echo "$c thread, $p process"
        IPM_REPORT=full IPM_REPORT_MEM=yes IPM_LOG=full LD_PRELOAD=/opt/ipm/lib/libipm.so srun -pjudge -N3 -n$p -c$c ./hw2b ./exp.png 17417
    done
done
```



The result shows a great scalability from the view of extending process and extending threads. For the same thread number, the more process the better the performance. For the same process number, the result is also the same.

```
8.315058 second
15.334840 second
17.651174 second
22.322980 second
24.635508 second
25.794279 second
```

For load-balancing, I use 6 process with 10 threads to verify it. The result shows in above figure, each number is the time process executed. It's not really balanced because the row contains different complexity data.

Optimization Strategies:

As we can see in the above results, my program is not really balanced. For Both one, I think implement the master worker scheme may bring more balance.

Discussion:

My program yields great scalability especially in the hybrid one. Because the amount of data is quite large and my program really parallel them. And there's no much communication overhead. The whole program is bounded by computational time in mandlebrot set.

As for the load-balancing, I parallel each row. However, I didn't noticed that each row's execution time may widely different. So each process or threads will have significant differences in execution time. The master worker scheme may solve the problem. Or, since the data usually has longer width, maybe I can partition by cols.

Experience & Conclusion:

In this homework, I got a better view of using cluster to solve parallel problem. By combining MPI and OpenMP, I utilize not only the cpu in each process but also multi-process. Plus, I use vectorization. Using the lower-level intrinsic is such a joyful thing and also gives me a better performance. The whole homework process also let me think of many optimization strategies like partition row or col, how to vectorize the calculations…… I struggled a lot in doing the vectorization but finally I found out I should really take time reading the instructions in intel website. The whole process is valuable, Thanks TAs.