

Homework 2 report

112062524 詹博允

Implementation

- `pthread`
 - Task Partition (Pixel allocation)
 - 在 task request 的部分, 我使用了一個 mutex lock 來避免 critical condition 的發生
 - 一開始我選擇用 static allocation 來分配 Job, 我的做法是將圖視為 1D array, 然後將 pixel 平均分給每個 thread, 但是經過測試後發現 performance 並不好 (我的理解是因為 static allocation 很難處理 load balance), 這也跟老師上課時說的概念接近
 - 在第一個 solution 實作完後我便決定將其改成 dynamic allocation, 我的作法是把 image 視為一個 1D array, 每次 thread request task 時就會 allocate 連續的 20 個 pixel 給 thread. 當 thread 計算完他的 20 個 pixel 後便會再去拿 20 個 pixel 去做計算, 這樣的架構可以考慮到 load balance, 整體的 performance 也上升不少 (運算時間大約為 700 秒左右)
 - 以下是我的 Task Partition 架構

```
bool get_position(long int &my_start, long int &my_end) {
    if (end >= whole_len) {
        return false;
    } else {
        start = end;
        end += required_len;
        if (end > whole_len)
            end = whole_len;
        my_start = start;
        my_end = end;
        return true;
    }
}
...
```

```

pthread_mutex_lock(&mutex);
if (get_position(my_start, my_end) == false) {
    pthread_mutex_unlock(&mutex);
    break;
}
pthread_mutex_unlock(&mutex);
...

```

- Vectorization

- 為了減低 calculation time, 我實作了 vectorization, 概念上我把 pixel repeat calculation 的部分 vectorize, 整體的 performance 也有滿顯著的上升 (運算時間從 700 → 400), 以下是我的實作簡述

```

while (true) {
    // while (repeats < iters && length_squared < 4)

    // temp = (x*x) - (y*y) + x0;
    vec_x_squared = _mm_mul_pd(vec_x, vec_x);
    vec_y_squared = _mm_mul_pd(vec_y, vec_y);
    vec_temp = _mm_add_pd(_mm_sub_pd(vec_x_squared, vec_y_squared), vec_x0);

    // y = 2 * x * y + y0;
    vec_xy = _mm_mul_pd(vec_x, vec_y);
    vec_y = _mm_add_pd(_mm_mul_pd(vec_xy, _mm_set1_pd(2)), vec_y0);

    // x = temp;
    vec_x = vec_temp;

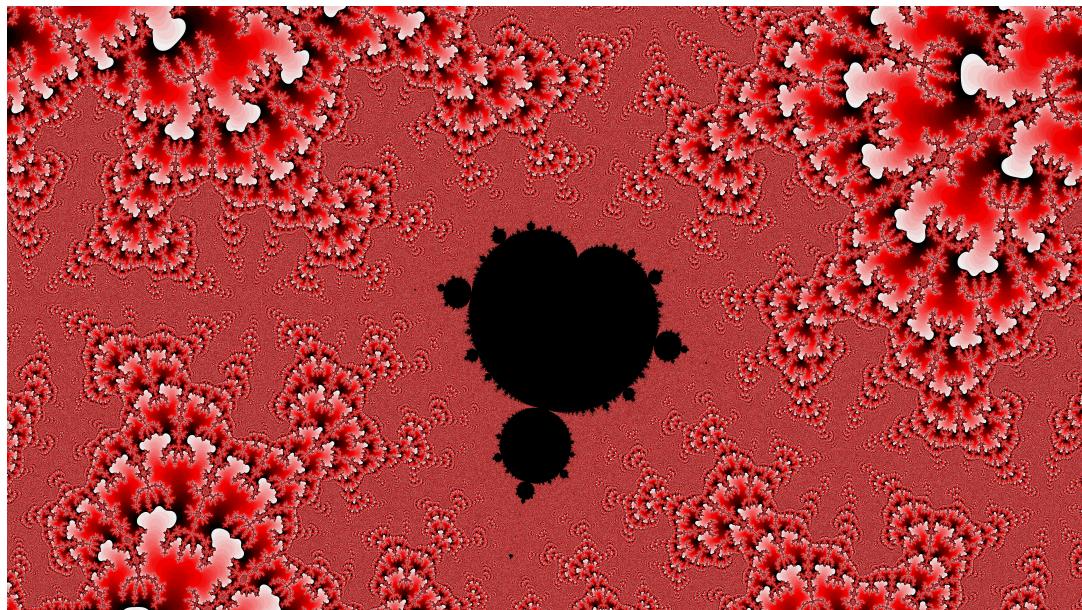
    // length_squared = x * x + y * y;
    vec_x_squared = _mm_mul_pd(vec_x, vec_x);
    vec_y_squared = _mm_mul_pd(vec_y, vec_y);
    vec_length_squared = _mm_add_pd(vec_x_squared, vec_y_squared);

    // ++repeats;
    vec_repeats = _mm_add_pd(vec_repeats, vec_ones);
    __m128d vec_tmp = vec_repeats;

    // while (repeats < iters && length_squared < 4)
    mask = _mm_and_pd(_mm_cmplt_pd(vec_length_squared, vec_four) \
                      , _mm_cmplt_pd(vec_tmp, _mm_set1_pd(iters)));
    maskResult = _mm_movemask_pd(mask);
    if (maskResult <= 2) break; // If any one value in mask is 0 -> break
}

```

- 在做完 vectorization 後，因為跟同學討論知道如果單純的 vector 運算，會遇到有一個 value 的數字已經算完後另一個數字還在計算導致其成為 head of block，所以這裡我做了 load balance，也就是當 vector 的一個 value 計算完後我會將其取出後放入下一個 value 進去計算
- Other optimization
 - 在跟朋友討論後知道可以優化寫入 image 的速度，整體的 performance 應該可以上升 20%，但是因為時間因素所以沒有實作



-
- **Hybrid**
 - Task Partition
 - 因為 MPI 的特性 (process 間的溝通成本很大)，這裡我選擇用了 static allocation，我的做法是將 image 裡的 pixel 平均分給所有的 process，以下是我的分法

```
假設有 13 個 pixel, 4 個 process :  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]  
  
rank0 = [0, 4, 8, 12]  
rank1 = [1, 5, 9]  
rank2 = [2, 6, 10]  
rank3 = [3, 7, 11]
```

- 依照 Mandelbrot set 的圖片來看(pixel 附近的 value 不會差太大)這樣的分法可以確定每個 node 的 loading 差不多, 算是一個 Naive 的分法, 但是整體 performance 其實不錯
- 但是這樣分配的問題是在 gather 時資料是雜亂的, 所以需要在花 $O(N)$ 的時間來把 data 重新排序好, 理論上這個 $O(N)$ 可以在優化成 $O(N/cpus)$, 但是一樣礙於時間壓力這裡就沒有再做優化了

理論上可以再優化的 for loop

```
/* draw and cleanup */
if (rank == 0) {
    int k = 0;
    for(int i = 0; i < whole_len/size; i++) { //3
        for(int j = 0; j < size; j++) { //4
            fullImage[k++] = tmp[displacements[j] + i];
        }
    }
    for (int i = 0; i < whole_len%size; i++){
        fullImage[k++] = tmp[displacements[i] + whole_len/size];
    }
    write_png(filename, iters, width, height, fullImage);
}
```

◦ Parallelization

- 我選擇使用 `#pragma omp parallel num_threads(num_cpus)` 來平行計算, 這樣做的原因是實作不需要再想要怎麼優化, 可以直接把 pthread 的 vectorization code 搬過來用, 所以我在 pixel calculation 的部分跟 pthread 幾乎一模一樣, 只有被每次被 assign 的 pixel 從連續的 20 個 pixel 變成跳動的 20 個 pixel, 而每個 thread 則是 dynamic 的在自己計算完 20 個 pixel 後再去拿另外 20 個 pixel 去計算

◦ Other optimization

- 像上面提到的一樣, sorting 的 for 迴圈應該可以再做優化, 但是礙於時間就沒有去實作了
- 在我的實作架構下,image 的寫入反而沒辦法做優化,主要是 coding 成本太大了.

Experiment & Analysis

i. Methodology

1. System Spec

跑在 apollo 上.

2. Performance Metrics

- `pthread` : 使用 `clock_gettime(CLOCK_MONOTONIC_RAW, start)` 與 `clock_gettime(CLOCK_MONOTONIC_RAW, end)` 相減作為 運算時間
- `hybrid` : 使用 `MPI_Wtime()` 來觀測各個 process 的運算時間, 以及用 profiler 檢視每個 process 的狀況

ii. Plots: Scalability && Load Balancing && Profile

▼ Experimental Method:

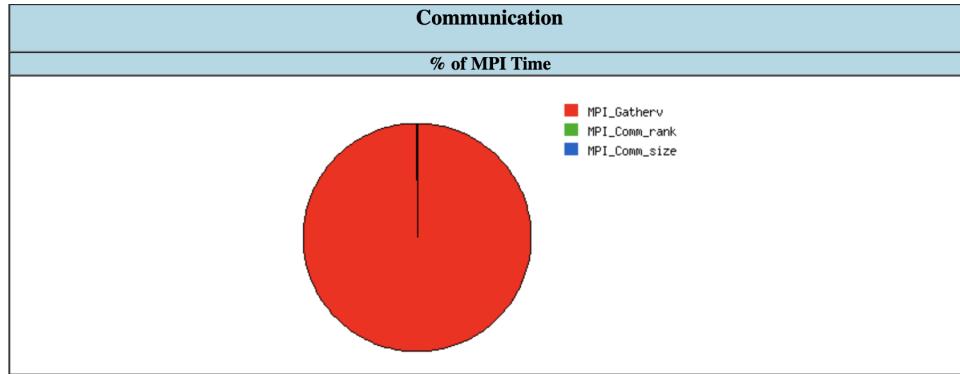
- 兩個 version 的實驗皆是用 `strict32.txt` 作為實驗對象
 - `pthread` : Node 與 process 固定為 1, thread 從 1 ~ 6
 - `hybrid` : Node 和 process 線性上升, 從 1 到 6 個 Node, 而 process 數也會相對應線性上升
- 在 profiler 的部分我則是跑了兩個 case
 1. hybrid, node = 6, process = 12, core = 4,
 2. hybrid, node = 1, process = 2, core = 4, 以下是我的指令

```
$ : IPM_REPORT=full IPM_REPORT_MEM=yes IPM_LOG=full \
LD_PRELOAD=/opt/ipm/lib/libipm.so \
srun -pjjudge -N6 -n12 -c4 ./hw2b out.png \
10000 -0.5506211524210792 -0.5506132348972915 \
0.6273469513234796 0.6273427528329604 7680 4320
-----
$ : IPM_REPORT=full IPM_REPORT_MEM=yes IPM_LOG=full \
LD_PRELOAD=/opt/ipm/lib/libipm.so \
srun -pjjudge -N1 -n2 -c4 ./hw2b out.png \
10000 -0.5506211524210792 -0.5506132348972915 \
0.6273469513234796 0.6273427528329604 7680 4320
```

▼ Performance Measurement

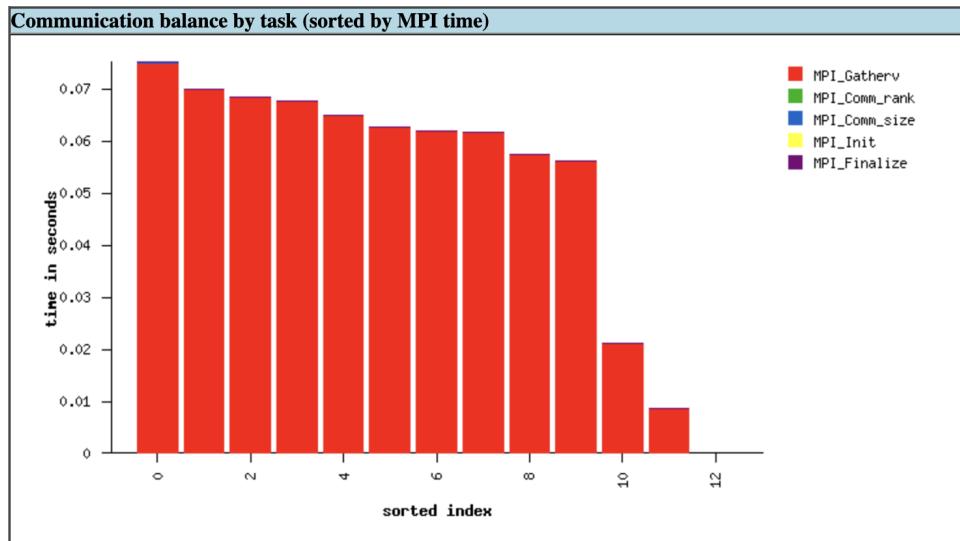
▼ Profiler outcome and analysis

- node = 6, process = 12
- communication



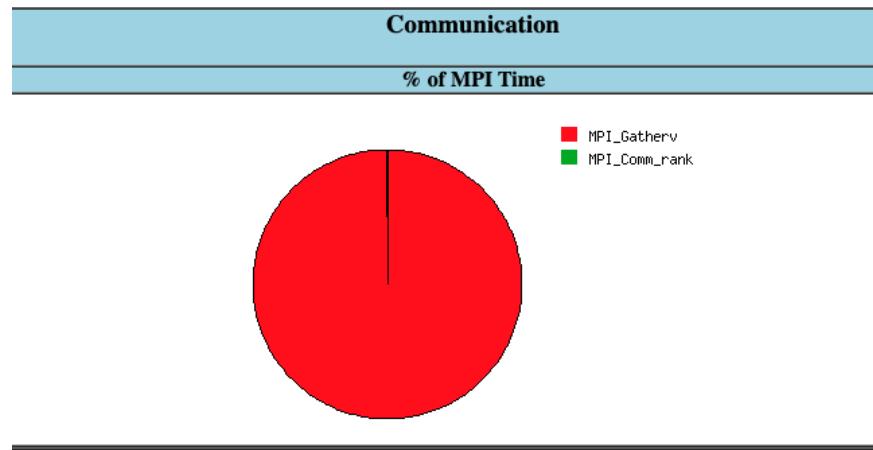
- 因為我在實作上只使用了 MPI_Gatherv 來做 data gathering, 所以這個 pie chart 也只有 MPI_Gatherv

- Communication balance by task

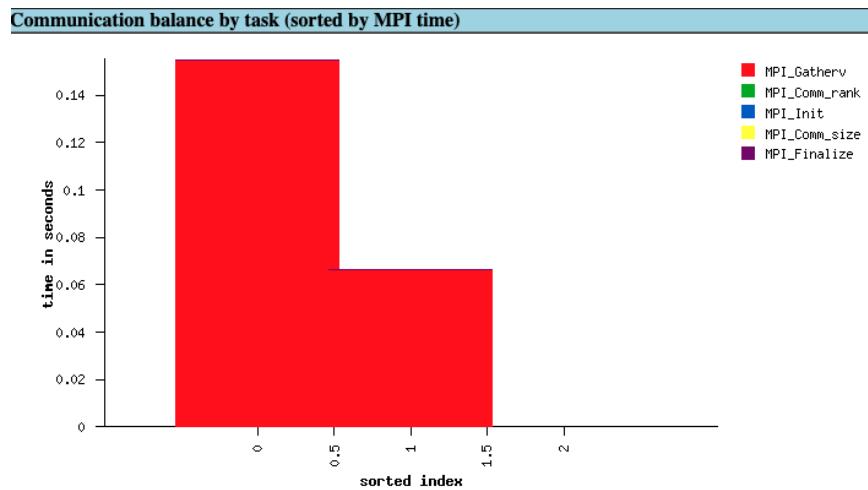


- 可以看到雖然在 data 上已經做了 load balance, 但是實務上 communication 仍然有 bias, 感覺在切割 pixel 上可以做得更好, 像是可以試試看連續的切, 或是依照 column 來切割, 也有可能是 MPI_Gatherv 所導致, 因為網路上有提到 MPI_Gatherv 的 performance好像會比 MPI_Gather 差

- node = 1, process = 2



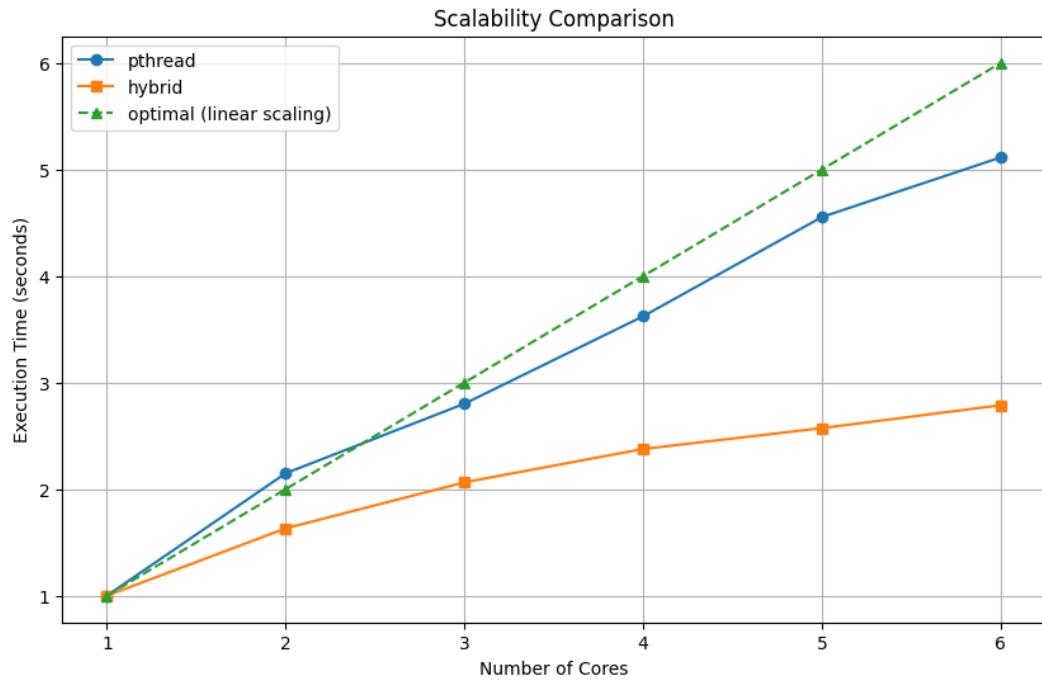
- 跟前一個實驗結過一樣, 此處就不贅述
- Communication balance by task



- 在兩個 node 情況下可以看到 load balance 的 bias 更明顯, 這是因為 data 需要 gather 到一個 process 做整合, 所以才會有這樣的情況發生, 如果要優化的話可能要選擇其他 MPI function ?

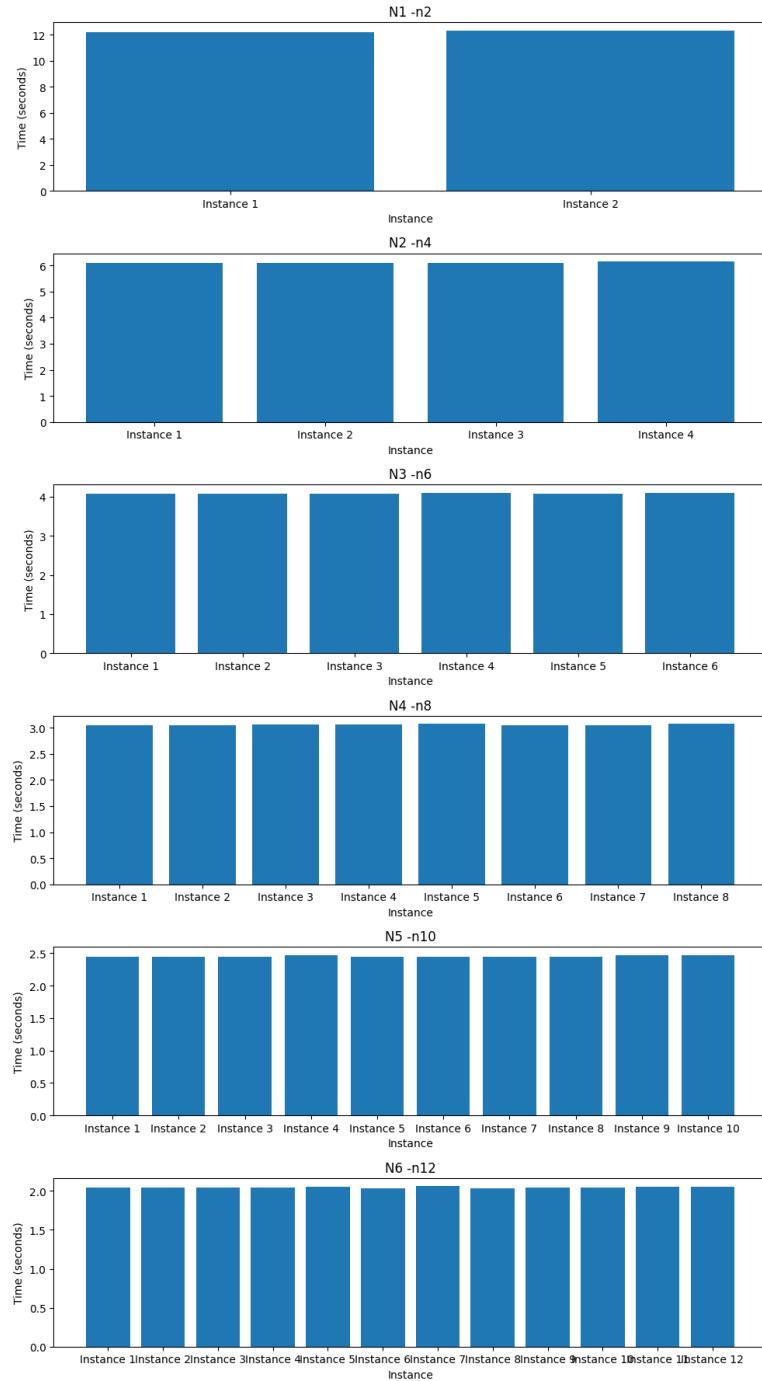
▼ Analysis of Results

- Scalability



- **Load Balance**

- 因為 pthread 跟 hybrid 用同樣的寫法, 所以 load-balance 的部分只做了 hybrid 的實驗



▼ Optimization Strategies

- 因為在 hybrid 的實作中我選擇用 `MPI_Gatherv` 來做資料整合, 但是從 profiler 看起來整體 communication 的 bias 蠻大的, 因為跟同學討論發現有人用 `MPI_Send`, `MPI_Recv` 來做資料整合, 說不定可以往這個方向做優化

2. 因為有跟同學討論到可以把 image write 的行為做平行化, 但是在衡量 coding 的複雜度後決定作罷
3. 我實作的 hybrid 在最後需要 $O(N)$ 來 sort 所有的 image pixel 回原位, 此處可以做優化, 但是因為時間因素便沒有再繼續優化

iii. Discussion

- **Scalability**
 - 可以看到在 scaling 的部分, pthread 有很好的 scalability, 但是 hybrid 就相對不好, 我分析的主要原因是為了配合 `MPI_Gatherv`, hybrid 在最後會花 $O(N)$ 做資料整理, 導致這裡成為了 bottleneck. 理論上這裡仍然可以做優化,
- **Load balance**
 - 可以從數據看到, 每個 thread 的 load-balance 非常地穩定, 不會有某個 thread 做太多事的情況發生, 這也讓我非常自豪

iv. Others

Experience & Conclusion

- **Your conclusion of this assignment**

這次的 project 可以體現 load balance 的重要性, 如果沒有做 load balance, 很容易就會有一個 thread/process 變成 head-of-block, 另外也可以從 vectorization 知道為什麼 GPU 加速可以這麼重要, 還有應該花更多時間在這個作業上, 雖然我已經預留五天給這個 homework, 但是整體時間還是不夠充足, 有些額外想做的 optimization 還是沒時間做

- **What have you learned from this assignment ?**

主要學到的內容有：

1. Load-balance : 在以往只知道 load-balance 的重要, 但是不知道真實影響有多大, 是到此次才知道會有多大的影響
2. vectorization : 理解如何做 vectorization, 以及知道 vectorization 的 code 要如何實作

- **What difficulties did you encounter in this assignment?**

主要的困難都是在將 code 改成 vectorization 時遇到的，因為 vectorization 的 coding style 與平時有蠻大的差異，所以花了不少時間做 coding 與 debug，以及在做 load-balance 的部分也是想了蠻久的，但是因為老師上課有提到一些 load balance 的 method 所以實作反而花的時間比想像的少