HW2 Report

資應所碩二 111065531 郭芳妤

Implementation

1. **hw2a - pthread**

   在 pthread 版本中，是參考老師上課 ppt 中實作的範例 code 建構一個 thread pool，此 pool 具有以下資料結構：考量到我們並不能事先知道哪一個 pixels 需要較久的計算量，所以透過動態分配 task 給 thread pool 中的 threads，可以避免事先分配、load 卻分得不平均的問題。

   ```
   struct threadpool_t {
     pthread_mutex_t lock;
     pthread_cond_t notify;
     pthread_t *threads;
     threadpool_task_t *queue;
     int thread_count;
     int queue_size;
     int head;
     int tail;
     int count;
     int shutdown;
     int started;
   };
   ```

   透過 thread pool handler function 中的這個 for loop，我們可以在一個 thread 解決手上任務後，直接檢查 task queue 中是否還有沒有做完的任務，並直接分配給它：

   ```
   for(;;) {
       /* Lock must be taken to wait on conditional variable */
       pthread_mutex_lock(&(pool->lock));

       /* Wait on condition variable, check for spurious wakeups.
          When returning from pthread_cond_wait(), we own the lock. */
       while((pool->count == 0) && (!pool->shutdown)) {
           pthread_cond_wait(&(pool->notify), &(pool->lock));
       }

       if(pool->count == 0){break;}
       /* Grab our task */
       task.function = pool->queue[pool->head].function;
       task.argument = pool->queue[pool->head].argument;
   ```

```
    pool->head = (pool->head + 1) % pool->queue_size;
    pool->count -= 1;

    /* Unlock */
    pthread_mutex_unlock(&(pool->lock));

    /* Get to work */
    (*(task.function))(task.argument);
}
```

如何定義一個 task？

- one pixel as 1 task

  一開始實作時，我採用的方式是一個 pixel 計算做為一個任務，但雖然 thread 完成一個 task 的速度很快，但要經常進到 handle function 去要下一個任務，所以導致時間會超級久（約1200秒）

- one row as 1 task

  後來改採用一個 row 作為一個 task，此時每個 thread 會一口氣計算完該 row 的所有 pixels 並寫回 image 以後，才去索取下一個 task（在這邊不用擔心 lock，因為寫回 image 的位置都不同，而且計算之間不會有 dependency），task function 如下：

```
/* by row assign task */
void MSet(void *arg_j){
    int *j;
    j = (int *) arg_j;
    for(int i = 0; i < width; i++){
        double y0 = *j * ((upper - lower) / height) + lower;
        double x0 = i * ((right - left) / width) + left;
        int repeats = 0;
        double x = 0;
        double y = 0;
        double length_squared = 0;
        while (repeats < iters && length_squared < 4) {
            double temp = x * x - y * y + x0;
            y = 2 * x * y + y0;
            x = temp;
            length_squared = x * x + y * y;
            ++repeats;
        }
        // pthread_mutex_lock(&lock);
        image[*j * width + i] = repeats;
    }
}
```

2. **hw2b - hybrid**

使用 hybrid version 時，需要先考量哪一個 process 負責的範疇、再根據 process 負責的範圍去動態分配任務給底下的 thread。在這邊為了避免直接按照順序切 row 給 process 會有分配不均的問題（比如計算量大的區塊都集中在下方，最後一個 process 的計算量就會過大），所以這邊的分配邏輯有兩層：

1. 取 row_id%(#process) 作為 process 要負責的 row（跳著分配 row 給 process）→ 這邊的做法是是先建構一個 mapping array 紀錄每一個 row 負責的 mpi rank id 為何。

```
int row_proc[height] = {0};
for(int j = 0; j < height; j++){
    row_proc[j] = j%mpi_ranks;
}
```

2. 對於每個 process 被分配到的特定 row，使用 omp for schedule (dynamic) 動態分配 pixel 的計算給 process 底下的 thread

整體架構如下：

```
#pragma omp parallel num_threads(thread) shared(image)
{
    #pragma omp for schedule(dynamic, 1) nowait
    for(int j = 0; j < height; j++){
        /* 先處理 j 和 mpi_rank 的對應關係 */
        int responsor = row_proc[j];
        if (mpi_rank == responsor){
            // printf("process %d working\n", responsor);
            // 每個 thread 開始 dynamic 地搶工作
            for(int i = 0; i < width; i++){
                double y0 = j * ((upper - lower) / height) + lower;
                double x0 = i * ((right - left) / width) + left;
                int repeats = 0;
                double x = 0;
                double y = 0;
                double length_squared = 0;
                while (repeats < iters && length_squared < 4) {
                    double temp = x * x - y * y + x0;
                    y = 2 * x * y + y0;
                    x = temp;
                    length_squared = x * x + y * y;
                    ++repeats;
                }
                image[j * width + i] = repeats;
            }
        }
    }
}
```

Experiments & Analysis

i. Methodology

- System Spec
  - 本次實作是利用課堂提供的 Apollo，下方的實驗是以系統的 3 個 test node 完成。
- Performance Metrics
  - 使用 clock_gettime() 以及 omp 中的 omp_get_wtime() 計算 thread、process 使用的時間（單位：秒）
  - 下方執行時間若為多個 process，均挑選所有 mpi_rank 中執行時間最大者。

ii. Experiments Content

💡 **Experiment Methods**

- testcases
  - fast: fast02.txt
  - slow: slow01.txt
  - strict: strict28.txt
- parallel configuration
  - hw2a - pthread
    - 1 process + 3、6、12 threads
  - hw2b - hybrid
    - single node (N = 1)
      - 1 process: 3、6、12 threads
      - 3 process: 1、2、4 threads
    - multiple node (N = 3)
      - 3 process：1、2、4、8 threads

Execution time under different setting

Pthread (n = 1)

Execution time	3 thread	6 thread	12 thread
fast02	1.664	0.839	0.421
slow01	70.093	37.359	21.77
strict28	34.914	17.483	8.742

Hybrid (1 node, n = 1)

Execution time	3 thread	6 thread	12 thread
fast02	2.194	1.531	0.43
slow01	72.79	38.79	22.60
strict28	36.23	18.13	9.07

Hybrid (1 node, n = 3)

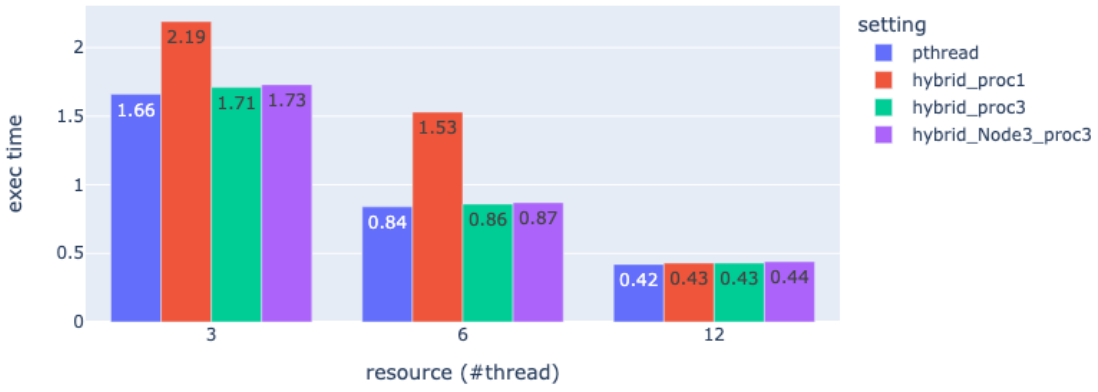
Execution time	1 thread	2 thread	4 thread
fast02	1.71	0.86	0.43
slow01	70.61	40.52	22.66
strict28	36.23	18.12	9.07

Hybrid (3 node, n = 3)

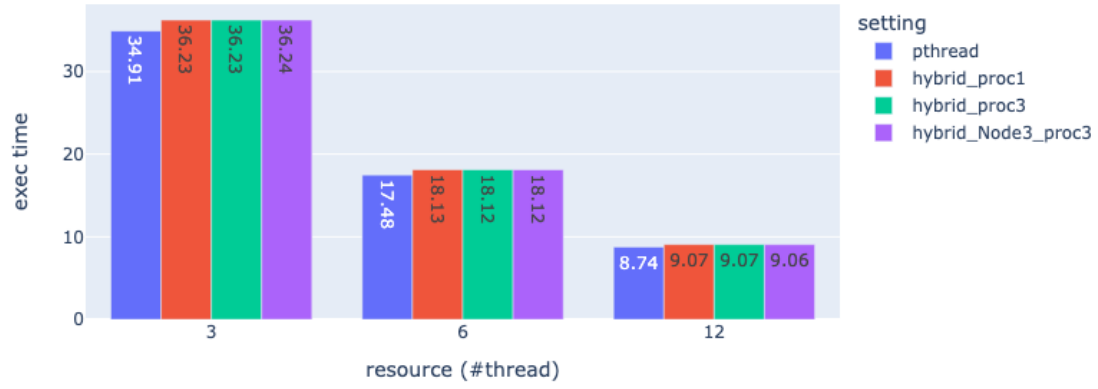
Execution time	1 thread	2 thread	4 thread	8 thread
fast02	1.73	0.8692	0.4394	0.23
slow01	76.09	40.36	20.94	13.68
strict28	36.24	18.12	9.06	4.55

▼ 由下三個圖表可以發現：在不同 data size 的情況下，整體都呈現 thread number 越多，總執行時間越小的趨勢，同時在 thread 數量越多的情況下，不同 setting 的差異越小。

Fast



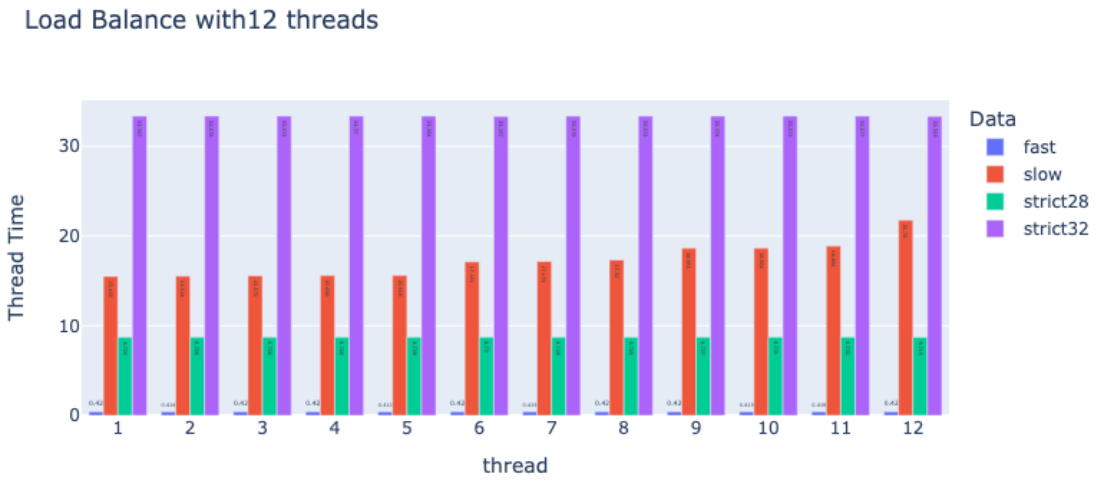
Strict



Slow

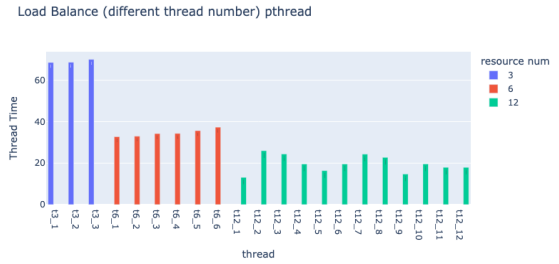
Load Balance under different settings

- 發現在所有 test case 中，隨著 data size 的增加，thread 之間的 load balance 表現越差：

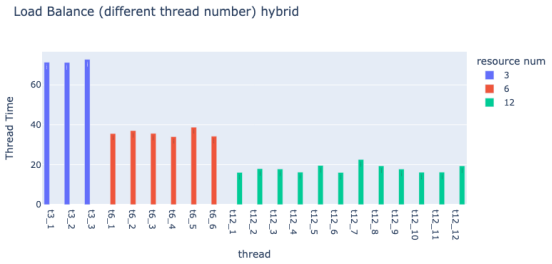


因此下方將以 slow01（本次使用的 3 個 test case 中時間最久的一筆）針對不同 setting 進行 load balance 的測試：

1. load balance between threads：比較不同 version 在相同 process 數量的表現，當 process 都2等於 1 時，可以發現兩個版本中都有隨著 thread 數量越多，thread 彼此之間的 load balance 變差的情形，但相較之下 hybrid version（右圖） 在 thread = 12 時，load balance 的狀況又更好一些。

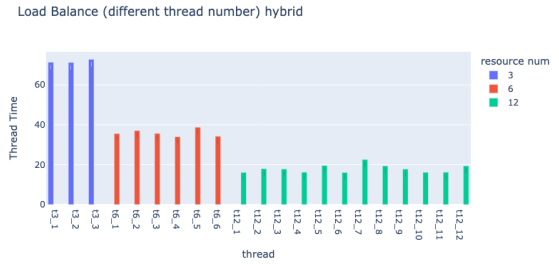


pthread (n = 1)

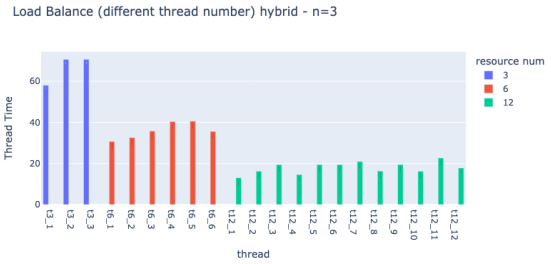


hybrid (n = 1)

2. 相同 version 底下，我們使用不同的 process 數量、但總資源數相同時 thread 的表現，可以發現 process = 3 時 load balance 的表現較差。在 process = 1 時，少量 thread 的情形下 thread 之間還可以看出 load balance，但 process = 3 時我們可以觀察到不論 thread 總數多少（3, 6, 12），執行時間都有較大落差，推論是 thread 之間分配到的工作量會因為 process 分配而受到影響。

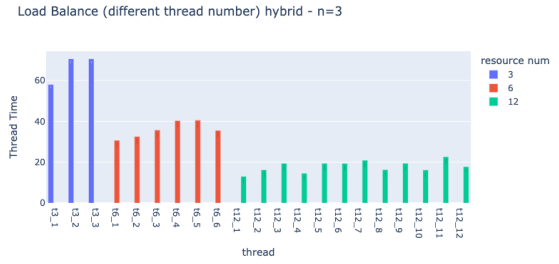


hybrid (n=1)

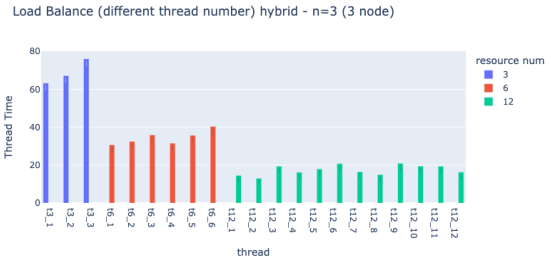


hybrid (n=3)

3. 比較相同 version、process、總資源數相同下，single node 和 multiple node 的 load balance 表現，可以發現兩者差異較小（但相較於 single node single process 的 hybrid 版本，兩者的表現都不算太好）



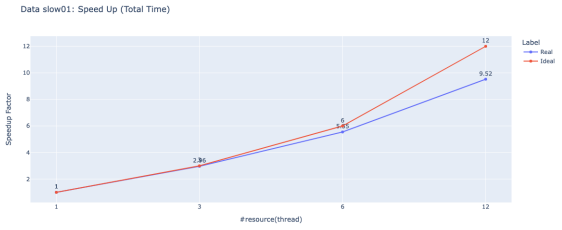
hybrid (n=3 under single node)



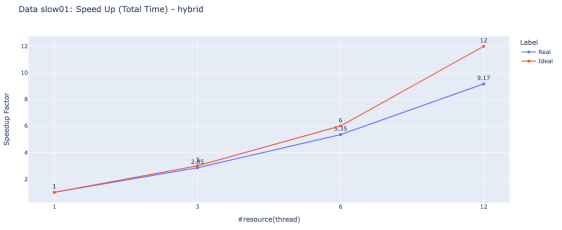
hybrid (n=3 under 3 node)

Speed up under different settings

1. 比較不同實作版本的 speed up：可以發現 pthread 的 speed up 較 hybrid 版本的 speed up 更好一些，推論可能跟我的實作方法中，hybrid version 在 schedule 後還需要多一個 process 的判斷有關（而 pthread 因為是直接使用 thread pool 實作所以更快速）

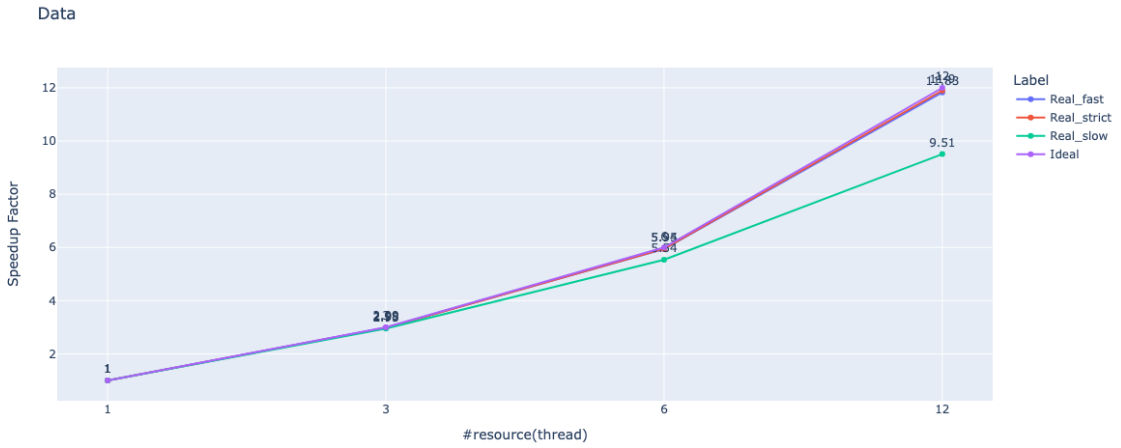


pthread version (n = 1)



hybrid version (n = 1)

2. 比較不同 data size 的 speed up（以 pthread 為例）：可以發現 fast 和 strict 測資和 Ideal speedup 幾乎重合，但 slow 測資的 speed up 有明顯掉下來的趨勢（這邊需要考量到我選擇的 strict 測資並不是執行時長最久的），可以推論我的實作上 scalability 可能是還需要加強的部分（目前還有辦法在所有 data size 上都達到相同的 speed up）



其他優化方法：

- SSE vectorization（但本次實作沒有嘗試成功）

Experience & Conclusion

- 粽上，本次實作起來的兩個版本都是在小測資時才有較好的 load balance 和 speed up，只要遇到計算量較大的測資、或者分配的資源增加，load balance 和 speedup 表現就會下降，整體而言 scalability 並不好。
- 在兩個版本中，雖然我自己的兩個版本實作起來的表現沒有差很多，但在和同學討論後有發現其他人的 hybrid 版本似乎都比 pthread 更好（而我在 speed up 上反而是 pthread 版本比 hybrid 更好，而 load balance 才是 hybrid 版本較佳）。
- 本次實驗時可以明顯感受到，因為有不同 version 以及不同 setting，可以比較的內容和維度又比 HW1 更多，但很可惜是在這次作業我沒有預留足夠的時間去一一實驗所以的 setting，希望自己可以在下次作業改進。