PP_HW2.md

Parallel Programming HW2

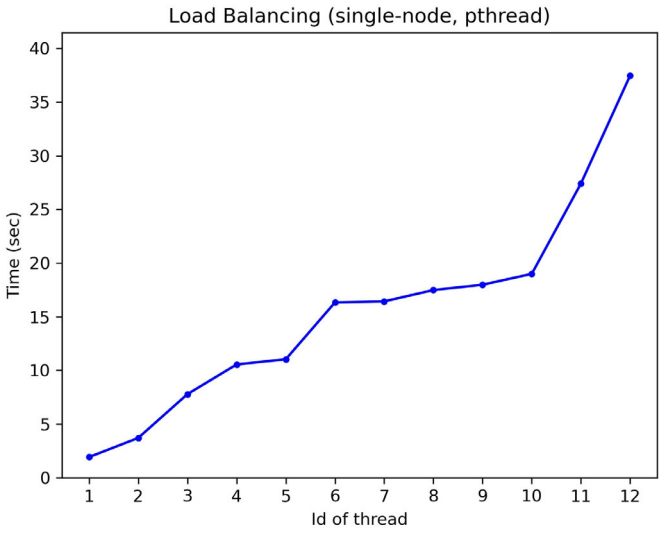
112164513 陳彥凱

Implementation

分為以下兩部分

- Pthread - Load Balancing

  一開始只是單純的把整個圖片範圍以高度做平均，分配給各個 threads，後來發現這麼做非常不 balance。



  故第二版改為建立一個 task queue，每個 thread 做完一份後就會再去拿一個 task，如此一來就不會像第一個版本，有些 thread 因為剛好被分配到計算量比較小的範圍，所以很早就做完了在 idle。而這個版本就 balance 了很多，可以參考 **Experiments & Analysis** 部分的 **plots - Load Balancing (single-node, pthread)**。 也因為多個 threads 都會去讀寫 task queue，所以需要使用簡單的 mutex lock 來保護它，確保計算範圍不會出問題；另外，我嘗試了很多種 task 的定義大小，最後發現直接把圖片的每一行當作是一個 task，整體執行速度會是最快的。

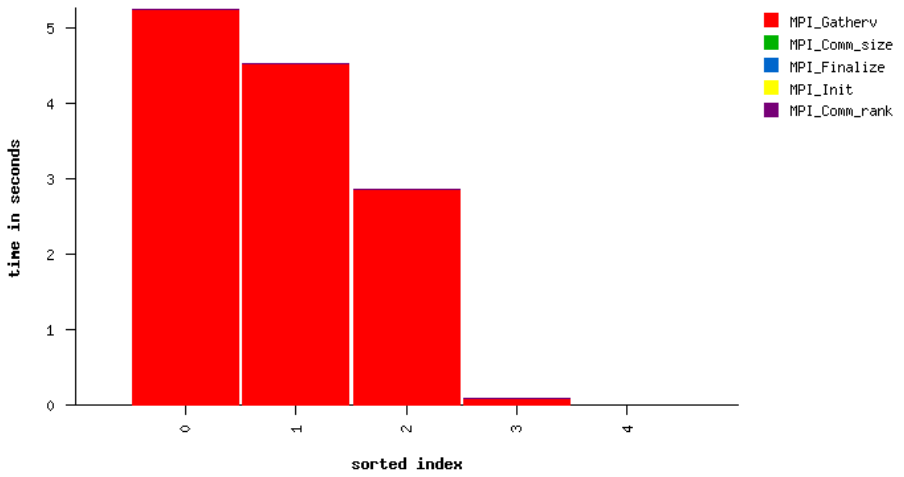
```
bool work_queue_get_task(work_queue_t* queue, task_t* task) {
    pthread_mutex_lock(&queue->mutex);
    if (queue->next_task >= queue->total_tasks) {
        pthread_mutex_unlock(&queue->mutex);
        return false;
    }
    *task = queue->tasks[queue->next_task++];
    pthread_mutex_unlock(&queue->mutex);
    return true;
}

int main(int argc, char** argv){

    // ... [Other section]

    int num_tasks = height;
    // Initialize work queue
    work_queue_t queue;
    work_queue_init(&queue, num_tasks);

    // Populate tasks in the queue
    int lines_per_task = height / num_tasks;
    for (int i = 0; i < num_tasks; ++i) {
        queue.tasks[i].start_line = i * lines_per_task;
        queue.tasks[i].end_line = (i == num_tasks - 1) ? height : (i + 1) *
lines_per_task;
        queue.tasks[i].left = left;
        queue.tasks[i].right = right;
        queue.tasks[i].lower = lower;
        queue.tasks[i].upper = upper;
        queue.tasks[i].width = width;
        queue.tasks[i].height = height;
        queue.tasks[i].iters = iters;
        queue.tasks[i].image = image;
    }

    // Create threads
    pthread_t threads[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL, compute_mandelbrot, &queue);
    }

    // ... [Other section]
}
```

- Hybrid - Load Balancing

  在 hybrid 版本中有兩個需要注意 load balance 的地方，第一個是每個 process 被分配到的工作量，第二個是每個 process 底下，使用 openmp 分配給每個 thread 的工作量。一開始也是先嘗試簡單的平均分配：

```
int rows_per_process = height / world_size;

// Calculate the remainder to distribute the extra rows
int remainder = height % world_size;

// Calculate the starting row for this process
int start_row = world_rank * rows_per_process + min(world_rank,remainder);

// Calculate the ending row for this process,
// the last process might have more rows if the height is not divisible
int end_row = start_row + rows_per_process + (world_rank < remainder);
```

從 IPM profiler 可以看到每個 process 的工作量很不平均，甚至有一個幾乎是 idle 的。

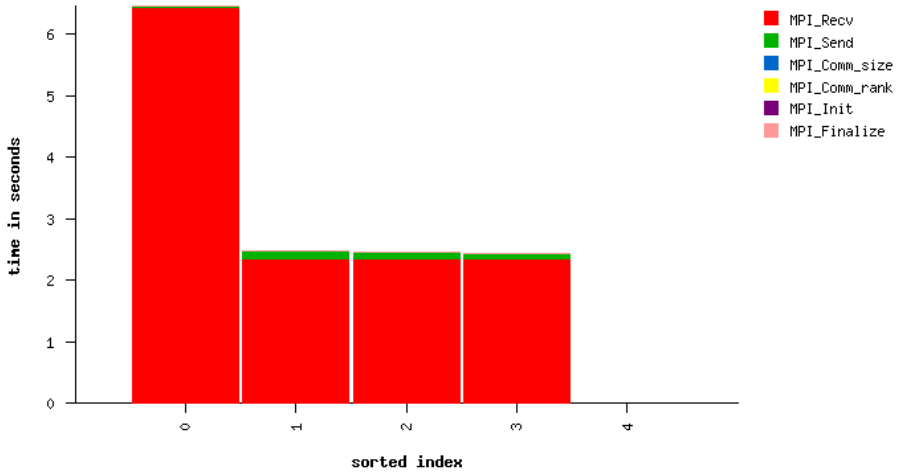


後來的策略跟 pthread 第二版一樣採取 task queue，以 process 0 作為 master 管理 task queue，其餘的 process 去執行計算。

```
if (world_rank == 0) {
    // Master process logic
    const int num_tasks = (height + rows_per_task - 1) / rows_per_task;
    int next_task = 0;

    while (next_task < num_tasks) {
        // Receive a task request from any worker
        int worker_rank;
        MPI_Recv(&worker_rank, 1, MPI_INT, MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Send the next task to the requesting worker
        Task task;
        task.start_row = next_task * rows_per_task;
        task.end_row = (next_task + 1) * rows_per_task;
        if (task.end_row > height) {
            task.end_row = height;
        }
```

```
            MPI_Send(&task, 2, MPI_INT, worker_rank, 0, MPI_COMM_WORLD);

            next_task++;
        }
    }
```

從 IPM profiler 可以看到除了 process 0 之外，其他的 process 執行時間都很平均，不過也相應的需要付出一些 communication time。



不過第二個 task queue 版本的效能提升並沒有我想像中的好，總共只快了10幾秒而已，這可能是因為第 process 0 需要做的事情太多了，需要管理 queue、畫圖，其他 process 雖然確實更平均的做完工作了卻沒辦法幫忙，導致總執行時間還是被拉長。

Experiment and Analysis

- Methodology
  - System Spec

    使用課程提供的 apollo cluster 做實驗，pthread 以及 hybrid 版本都在 judge partition 上執行。以 **strict36.txt** 作為測資：

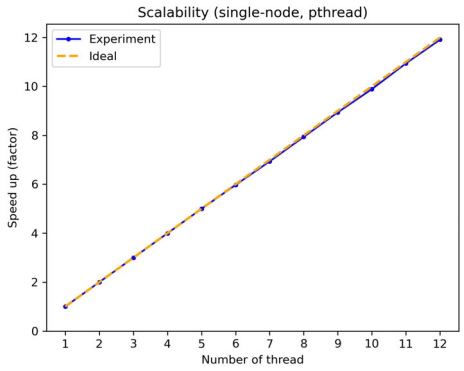
    srun ./exe $out 10000 -0.28727240825213607 -0.2791226112721823 -0.6345413372717312 -0.6385148107626897 7680 4320

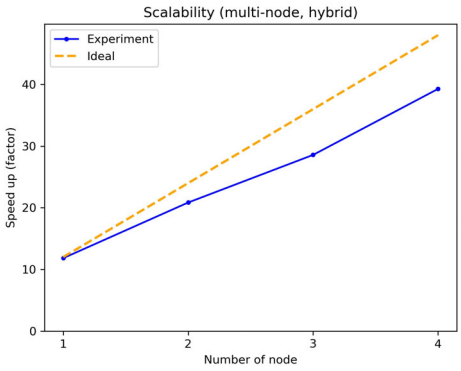
  - Performance Metrics

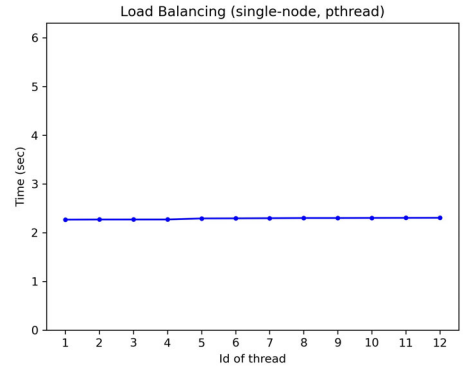
    在 pthread 版本中，使用 **<sys/time.h>** 裡的 **gettimeofday()** 來計算時間，可以算出每個 thread 單獨的執行時間和程式的總時間；執行總時間則去掉較不重要的 **write_png** 。

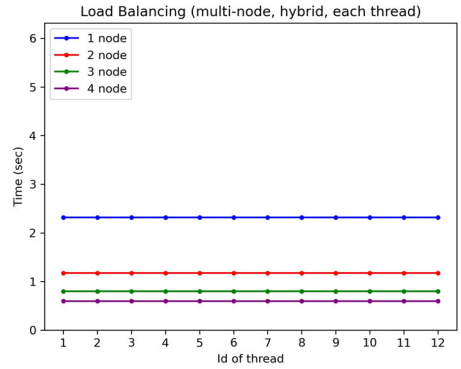
hybrid 的部分，使用 **MPI_Wtime()** 來取得 MPI 的總執行時間，同樣是去掉 **write_png** ；而在每個 process 底下的thread，則使用 **omp_get_thread_num** 先取得 **thread id**，再用 **omp_get_wtime** 計算時間。

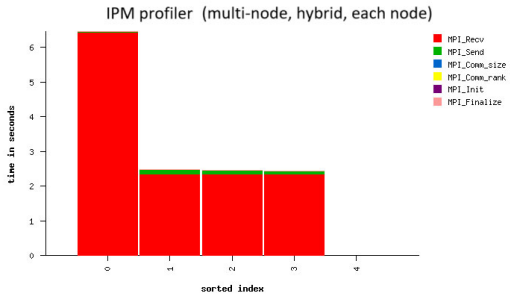
- Plots: Scalability & Load Balancing

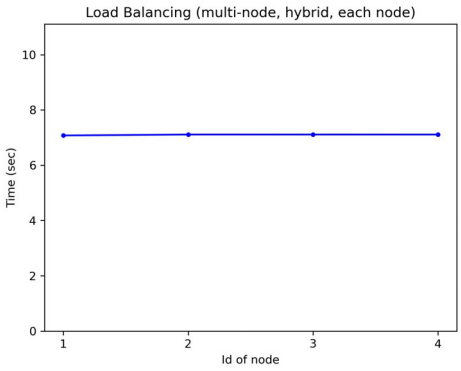












- Discussion
  - Scalability

    從圖中可以看到，在 **single-node, pthread** 平行的比較成功，scalability 幾乎完成了線性上升，這是因為 **Mandelbrot Set** 的計算量非常重，整支程式幾乎從頭到尾都在做計算，沒有 communication 也沒有 IO，只有在存取 task queue 時的 mutex 可能稍微減少了一些平行的部分。

    而 **multi-node, hybrid** 的部分就不是這樣了，離理想的 linear speedup 還有段距離，這是因為增加了 node 間的 communication，以及 master 為了調度工作所造成的額外工作量。

- Load Balancing

  在 **single-node, pthread** 能夠看出工作量分配得相當平均，這歸功於採用了 task queue 動態的分配工作量給每個 thread，減少 idle 的情況。

  而 **multi-node, hybrid** 的 MPI 執行時間，在最右下角的圖可看到儘管在各 node 之間非常的平均，但其實那是錯誤的，因為我是在 process 0 全部收到各 node 回傳結果之後才送出結束的訊號，使得所有 process 印出執行時間；正確應該要從 MPI profiler 來看 process 1, 2, 3 的計算時間就非常平均，但因為 process 0 必須作為 master 來調度工作，所以其實還是很不 balance；不過各 process 之中的每個 thread 因為採用了 omp dynamic schedule，所以執行時間就不意外的很平均了。

- 進一步的優化方向

  在這次的作業裡我沒使用到 vectorization 實作，因為在過程中不熟悉它的操作，導致出現很多 bug 最後只好放棄，不過這也提供了我未來進一步優化的方向。而在 hybrid 版本中，因為只使用了 MPI 來做 node 之間的溝通與工作調度，導致 MPI 的部分還是很不 balance。應該像 spec 提到的，使用 pthread 來處理溝通的部分，MPI 只負責 node 之間的工作調度就好，如此一來 scalability 應該會更好。

Conclusion

在這次的作業裡，我實作了更多上課提過的平行化技巧，也因此體驗到了一些要注意的事情，例如在 pthread 版本中使用的 mutex lock。而在這兩個版本中，hybrid 還是比較難正確實作出來的，使用 MPI 在做工作調度的時候，因為需要來回的傳送要計算的範圍、回傳計算好的結果給 master，所以只要一個小環節不小心沒對上就會卡住很久不知該怎麼 debug，不過也讓我更熟悉了 MPI 的各種 api 用法，累積了寶貴的經驗讓我下次再遇到類似的狀況能更快反應出來修正。

另外在 scalability 的部分我也覺得很有趣，因為這次的題目主要只有 load balance 能夠優化，沒有IO、communication也很少、主要計算部分的算法也不能更改，所以光是在 load balance 做好的情況之下，scalability 就能達到 ideal linear speedup。而我也觀察到 scalability 是跟整支程式的執行時間無關的，即使我的 hybrid 版本跑的比 pthread 版本快60幾秒，但因為不夠 balance 的關係，scalability 反而會是比較差的。相對的在 pthread 因為做好了 load balance，又因為這次的題目屬於 embarrassingly parallel problem，所以平行化效果就很不錯，執行時間也因此加快很多。

然而可惜的部分是沒有實作出 vectorization，如果再把這個技巧也用上的話，performance 就可以再繼續提升，而這次在 score board 上還是只能排在中後段的位置。在 deadline 前幾天開始，我也深刻體會到了老師跟助教們提醒過的 long queueing time 問題，這次作業的計算時間本來就比較長，有些同學在測試做實驗的話，就會占用更久的時間，所以下次應該要更早開始做作業，也保留更多進一步優化的時間。