CS542200 Parallel Programming

Homework 2: Mandelbrot Set

108062373 蘇勇誠

Implementation

1. Pthread

在 Pthread 版本中，一開始 main function 會根據 CPU 的數量(n CPUs)，create n 個 thread，每個 thread 會各自運算各自的 task。在 main function 中，使用 pthread_join 作為 barrier，等待所有 thread 都完成各自該完成的 task 時，就會由 main function 將 image 運算完的結果 io 寫入檔案中。

接著會探討每個 thread 所要執行的 task 為何。由於圖片每個 row（height）所需執行的時間不同，因此若直接將 row 平均 partition 至各個 thread（如：第 k 個 thread 須執行 row k * height / # process ~ (k + 1)* height / # process – 1，會導致某些 thread 所需執行的時間特別長，因此 load balance 效果不佳。因此，我是使用 **dynamic allocate** 的方式分配所需運算的 row 給 thread，thread 每次只會運算一個 row。運算完一個 row 以後會根據一個 shared variable（cur_ height）檢視目前運算到哪個 row，並取得 cur_ height+1 作為下一次該 thread 所需運算的部分。而此 shared variable 必須被保護在 critical section 內，避免兩個 process 對 cur_height 執行 increment。用這種 **partition** 的方式可以達到不錯的 **load balance** 效果。詳細實作方式如下左圖所示，每個 thread 都會一直執行下左圖中的 while loop，每次執行 while loop 時會先在 critical section 取得目前要運算的 row，並對該 row 進行運算。

```
while(1) {

    // get row
    pthread_mutex_lock(&mutex);
    cur_height++;
    cur_row = cur_height;
    pthread_mutex_unlock(&mutex);

    if(cur_row >= height) {
        break;
    }

    cal_row(cur_row);
}
```

```
// temp = x * x - y * y + x0;
__m128d vec_temp = _mm_add_pd(_mm_sub_pd(x_mul_x, y_mul_y), vec_x0);

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

//x = temp;
vec_x = vec_temp;

// x*x and y*y
x_mul_x = _mm_mul_pd(vec_x, vec_x);
y_mul_y = _mm_mul_pd(vec_y, vec_y);

//length_squared = x * x + y * y;
vec_length_squared = _mm_add_pd(x_mul_x, y_mul_y);

repeats_tmp[0]++;
repeats_tmp[1]++;
```

接著，在計算每個 row 的時候，我使用了 SSE intrinsic 進行 vectorization。在 server 上的 vector 長度為 128 bits，因此使用 intrinsic 可以一次執行 2 筆 double 的 data，有效率地**降低 execution time 與增加 scalability**。詳細轉換至 intrinsic 的方式如上左圖所示，註解的部分為原本的 code，如此一來就能一次執行兩筆 data，大幅地降低執行時間。

2. **Hybrid (OpenMP + MPI)**

首先，會先利用 MPI create k 個 Process，接著就開始運算。若有 height 為 n，每個 Process 會分配到 n/k 筆 data 進行運算。實際 **Partition** 的方式為為 rank 0 會負責運算第 0, 0 + k, 0 + 2k … row 的 data 之運算，rank i 會負責運算第 i, i + k, i + 2k … row 的 data 之運算。而在每個 Process 當中，會利用 OpenMP Dynamic Schedule Thread。實際實作方式如下圖所示，size 為 process 的數量，rank k 的 Process 需負責 k, k + size, k + 2*size…等，並由 OpenMP Dynamic Schedule Thread。

```
/* omp parallel loop to solve mandelbrot set */
#pragma omp parallel for schedule(dynamic)
for(int x = rank; x < height; x+=size) {
  cal_row(x);
}
```

當所有 Thread 都計算完以後，每個 Process 會將算完的 data 傳送至 Rank id = 0 的 Process，傳輸的方式是透過 MPI 的 Communication。最後，再由 Rank 0 的 Process 將 Image 透過 I/O 寫入檔案中。

一開始我 Partition Task 的方式為 Rank k 的 Process 會負責 k * height / # process ~ (k + 1)* height / # process – 1。不過我後來發現這樣的 Partition 方式效果並不好，可能有些區段會相較難以計算（iteration 次數較多），而這樣的分配方式會將 iteration 較多的區段分在某些 Process 內，導致 load balance 不佳。因此，後來採用的方式為 rank k 的 Process 需負責 k, k + size, k + 2*size…區段的 data。如此一來，可以**減少 Execution Time，並增加 Scalability**。

3. **Optimization**
   a. Vectorization：如前面所述，我是以 SSE Intrinsic 進行 Vectorization 進行優化。
   b. Vectorization 時，會同時計算兩筆 data。我原本是兩筆 data 算完以後才會將結果寫回 image。不過用這種方式若一筆 data 先完成時，另一筆 data 就會 idle 等到兩筆 data 都完成才能寫回 image。因此，我後來改為當一筆 data 完成時，就將該筆 data 寫回 image，並將計算這筆 data 的 resource 交給下一個要執行的 task，以減少 idle time。
   c. Hybrid 版本 partition 方式採用 ank k 的 Process 需負責 k, k + size, k + 2*size…區段的 data，以達到 load balance。

Experiment

1. **System Spec**：使用課程所提供的 apollo server 進行實驗。

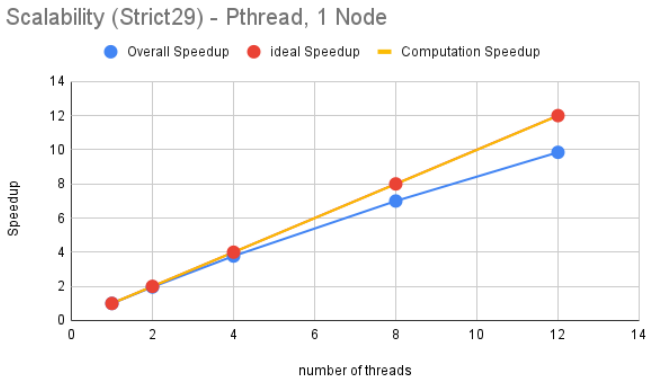
2. **Performance Metrics**：使用 clock_gettime 計算時間。會分別使用此方法算出 communication time、total time、i/o time。最後，再將 total time 扣掉 i/o time 與 communication time，即為 computation time。

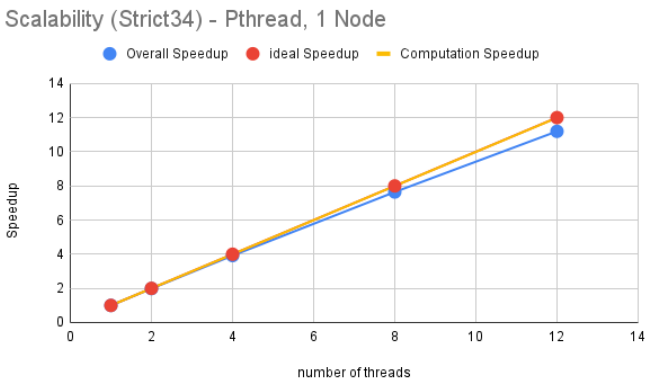
```
int main() {
    struct timespec start, end, temp;
    double time_used;
    clock_gettime(CLOCK_MONOTONIC, &start);

    .... stuff to be timed ...

    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);
}
```

3. 在本次實驗中有用到 Strict 26、Strict 29、Strict 34 作為 testcase。(iteration 均為 10000，width 均為 7680，height 均為 4320)

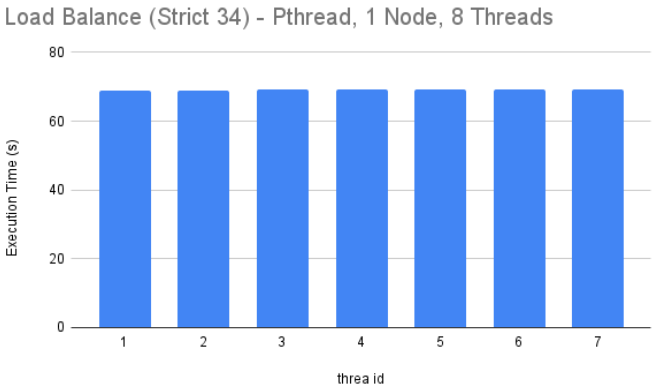
4. Pthread

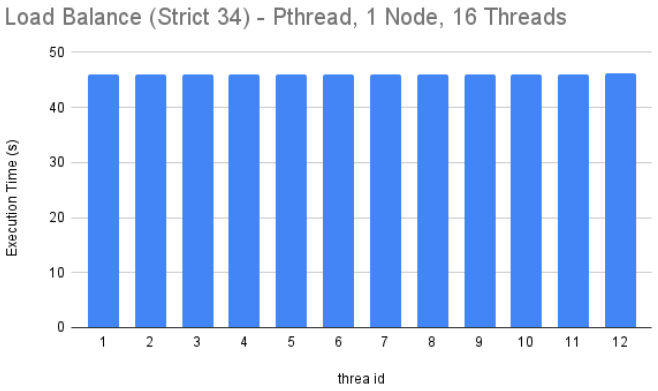
在 Pthread 版本中，我使用 Strict 29 與 Strict 34 作為 testcase。首先，我將 Node 數量固定為 1，並測不同 thread 數量的 **Speedup**，可以看出來 Computation Speedup 的曲線與 ideal Speedup 重疊。不過因為有 i/o time 的關係，i/o time 並不會因為 thread 數量增加而減少，因此 Overall Speedup 曲線會隨著 thread 數量增加而趨於平緩。





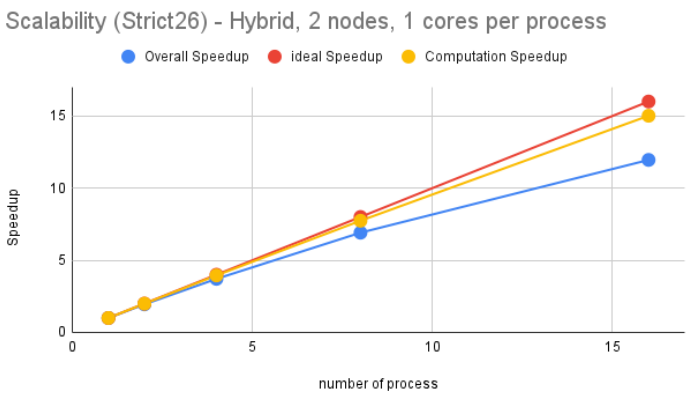
接著，探討 Pthread 之 **load balance** 的部分，我使用了 Strict 34 作為 testcase，分別測了 8 個 thread 與 16 個 thread 時，每個 thread 的 computation time，可以發現每個 thread 的 computation time 都差不多，非常 balance。

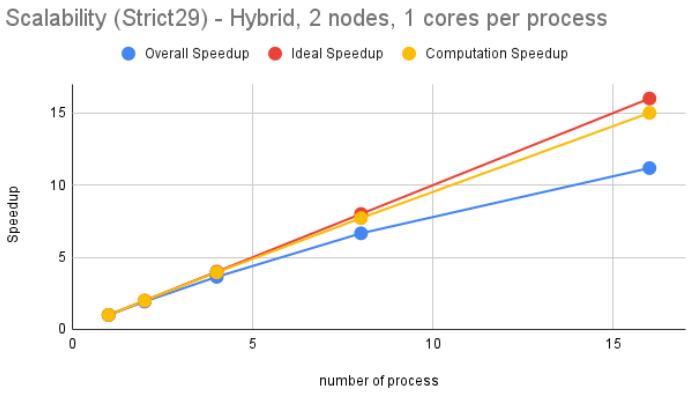




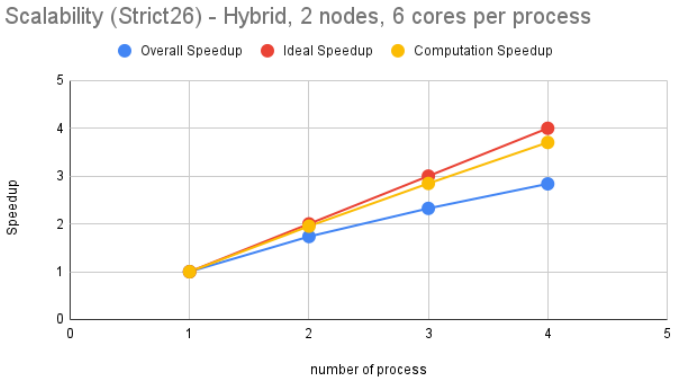
5. Hybrid

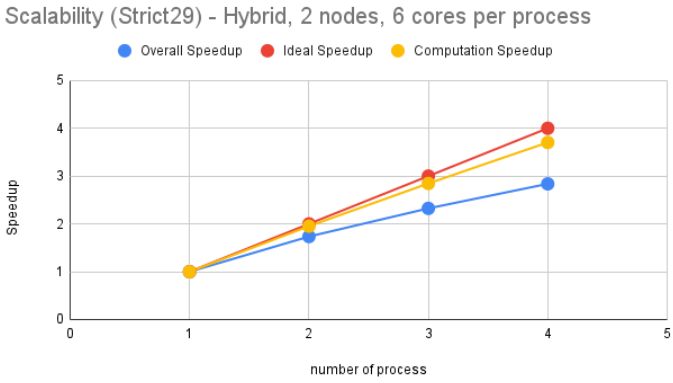
在 Hybrid 版本的 **Scalability** 計算中，我使用 Strict 26 與 Strict 29 作為 testcase。首先，我將 Node 的數量固定為 2，並分配給每個 Process 一個 Core。接著，我調整 Process 數量，以衡量 Scalability。可以看到 Computation Speedup 非常接近 ideal Speedup。不過 Overall Speedup 會因為 communication time 與 i/o time 影響，當 Process 數量增加時，Overall Speedup 曲線會越來越平緩。



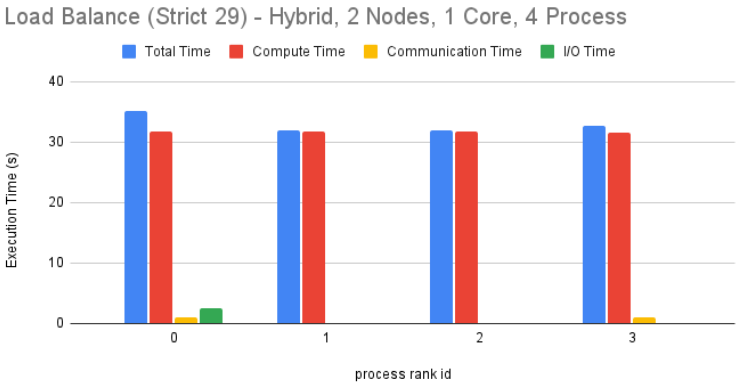


接著，我還是使用 Strict 26 與 Strict 29 作為 testcase 衡量 **Scalability**。將 Node 數量固定為 2，但分配給每個 Process 六個 Core。Scalability 結果跟 Core 為 1 的時候差不多。可以看到 Computation Speedup 非常接近 ideal Speedup。不過 Overall Speedup 會因為 communication time 與 i/o time 影響影響，當 Process 數量增加時，Speedup 會離 ideal Speedup 越來越遠。

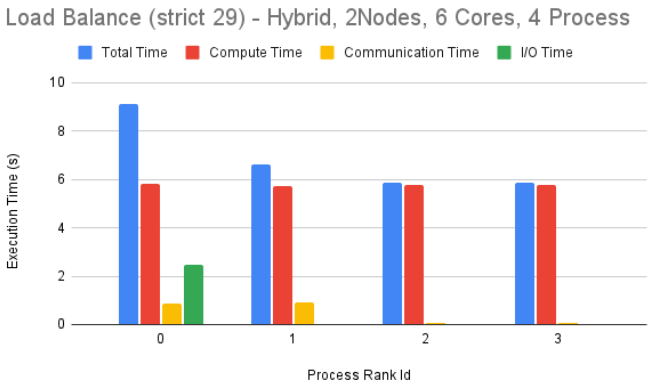




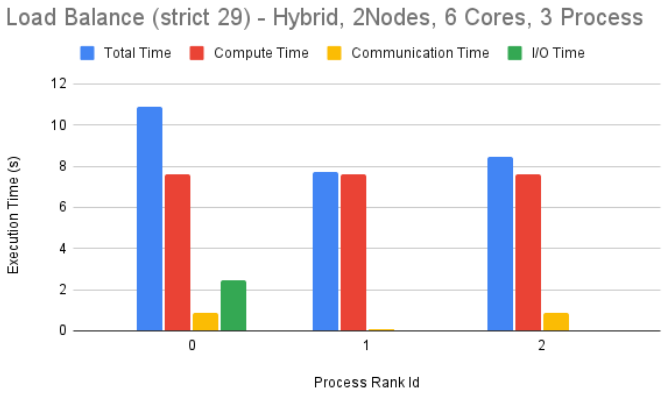
接著，探討 **Load Balance** 的部分，下圖包含三張圖表。圖一為 2 個 Nodes，4 個 Process，每個 Process 分別分配一個 Core。圖二為 2 個 Nodes，3 個 Process，每個 Process 分別分配六個 Core。圖三為 2 個 Nodes，4 個 Process，每個 Process 分別分配六個 Core。由下圖可見每個 Process 的 Computation Time 都非常接近。不過 Rank id 為 0 的 Process 會進行 i/o，因此 Rank id 為 0 的 Process Total Execution Time 會相較其他 Process 多。Rank id 為 0 以外的 Process 之 Total Execution Time 都很 Balance。



圖一



圖二



圖三

6. Discussion

a. Scalability

在 **Pthread** 版本中，可以看見 Computation Time 接近 ideal Speedup，主要是由於使用 Pthread 讀寫 image 時，不同 thread 處理的 row 不同，因此可以直接讀寫。而且使用 Pthread 易可以直接 Access Shared Space，不像 MPI 需與其他 Process 進行溝通。此外，由於可以進行 vectorization，也讓每個 thread 的 Execution Time 下降許多，因此 Computation Speedup 可以達到 ideal Speedup。不過 Overall Speedup 曲線會隨著 thread 數量增加而趨於平緩，主要是因為 i/o time 並不會隨著 thread 數量增加而減少

在 **Hybrid** 版本中，由於 MPI 程式不但有 I/O 的 Overhead，也有 Communication 的 Overhead。I/O 時間並不會隨著 Process 增加而減少，因為所有的 data 都是送到 Rank id = 0 的 Process 進行 I/O。而 Communication Time 很有可能隨著 Process 增加而增加。因此 Overall Speedup 曲線會相對於 Pthread 版本更於平緩。而 Computation Time 仍然很接近 ideal Speedup。

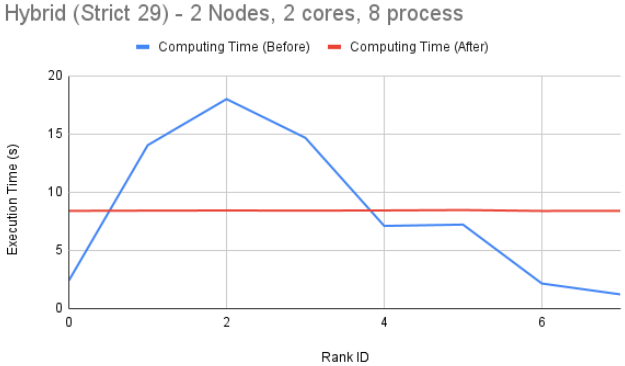
b. Load Balance

在 **Pthread** 版本中，由於是使用 dynamic allocate 的方式分配所需運算的 row 給 thread，thread 每次只會運算一個 row。運算完一個 row 以後會根據一個 shared variable（cur_ height）檢視目前運算到哪個 row，並取得 cur_ height+1 作為下一次該 thread 所需運算的部分。由此方式去分配 task 給 thread，因此 Pthread 版本中每個 thread 的執行時間都非常的 balance。

在 **Hybrid** 版本中，由於 Rank id 為 0 的 Process 會負責進行 I/O，因此 Rank 為 0 的 Process 會相較其他 Process 所需的執行時間還久。但撇除 Rank id 為 0 的 Process，其他 Process 的 Total Execution Time 仍算是 balance 的狀態。

7. Optimization

原先在 Hybrid Version，我是將 data 依序 partition 給各個 process 使用，也就是 Rank k 的 Process 會負責 k * height / # process ~ (k + 1)* height / # process – 1。不過經過 Profile 之後發現這樣的 Partition 方式效果並不好，可能有些區段會相較難以計算（iteration 次數較多），而這樣的分配方式會將 iteration 較多的區段分在某些 Process 內，導致 load balance 不佳。因此，我改變了 Partition Data 的方式，Rank 為 k 的 process 會負責 k, k + size, k + 2*size…等（size = # process），以達到更好的 load balance 效果。下圖紀錄了各個 Process 的 Computing Time，Computing Time(Before)為 Optimization 之前的結果，Computing Time(After)為 Optimization 之後的結果。



Experience & Conclusion

經過這次的學習到了如何進行 shared memory programming，也讓我對 critical section 更加熟悉。此外，也更加熟悉如何以 Intrinsic 進行 Vectorization，以達到更好的 Performance。我也透過這兩次作業學習到了 data partition 方法的重要性，好的 partition 方法會提高整體的 load balance。