110062118
資工 25 賴姿妘

HW2: Mandelbulb

學號：110062118
姓名：賴姿妘

1. 實作

這份作業中，以 row 為基本分配單位，使用 MPI 分給各 processor，對於每個 processor 獲得的 chunk，使用 OpenMP 將各 pixel 的運算分配給各個 thread。

✧ OpenMP

由於每次運算 Mandelbulb 所需時間不同，因此使用 dynamic 分配工作，又因為各 pixel 運算相互獨立，使用 collapse 使 task 的粒度更小，能更有效分配任務。下圖為程式實作。

```
#pragma omp parallel for num_threads(num_threads) collapse(2) schedule(dynamic)
for (int i = start_row; i < start_row + num_rows; ++i) {
    for (int j = 0; j < width; ++j) {
```

✧ MPI

本作業將 static 和 dynamic 工作分配混合使用，若 n>2 使用 dynamic，否則使用 static 方式。每個 process 使用下列方法進行初始化。

```
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

以下將說明 static 方式。每個 process 獲得 height/(processor 數量)列的運算工作。最後一個 process 需協助計算剩餘工作。最後各 process 使用 gather 將運算結果整合至 rank 0。

```
int chunk_size = height / size;
    int start = rank * chunk_size;
    int end = (rank == size - 1) ? height : (rank + 1) * chunk_size;
```

```
MPI_Gather(rank == 0 ? MPI_IN_PLACE : raw_image + start * width * 4, chunk_size * width * 4, MPI_UNSIGNED_CHAR,
        raw_image, chunk_size * width * 4, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
```

以下將說明 dynamic 方式。Dynamic 方式使用 master/ slave 架構，rank0 為 master，其餘為 slave。每個 slave process 一次會獲得 10 個 row 的運算工作。由 master 負責進行工作調度，計算任務將依列序分配給不同 slave。下圖為程式實作。Rank 0 接收從任意 slave 的工作需求，slave 會傳遞 jobRequest 訊息，該訊息包含 slave rank、是否有已完成計算任務、完成任務的列起點、任務長度。若發現有完成的任務，master 將接收 slave 傳遞之計算成果，接者傳遞新的任務訊息給 slave。

110062118
資工 25 賴姿妘

```
while (next_row < height) {

    MPI_Recv(jobRequest, 4, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    slave_rank = jobRequest[0];
    hasProduct =  jobRequest[1];
    start_row = jobRequest[2];
    num_of_row = jobRequest[3];

    if(hasProduct){
        MPI_Recv(raw_image + start_row * width * 4, num_of_row * width * 4, MPI_UNSIGNED_CHAR, slave_rank, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    int rows_to_send = std::min(rows_per_request, static_cast<int>(height - next_row));
    int row_range[2] = {next_row, rows_to_send};
    MPI_Send(row_range, 2, MPI_INT, slave_rank, 1, MPI_COMM_WORLD);
    //MPI_Send(&next_row, 1, MPI_INT, slave_rank, 1, MPI_COMM_WORLD);
    //MPI_Send(&rows_to_send, 1, MPI_INT, slave_rank, 2, MPI_COMM_WORLD);


    next_row += rows_to_send;
```

若已無任務須要發配，master 將等待各 slave 回傳成果，並通知各 process 進行終止。下圖為程式實作。

```
for (int i = 1; i < size; ++i) {

    MPI_Recv(jobRequest, 4, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    slave_rank = jobRequest[0];
    hasProduct =  jobRequest[1];
    start_row = jobRequest[2];
    num_of_row = jobRequest[3];

    if(hasProduct){
        MPI_Recv(raw_image + start_row * width * 4, num_of_row * width * 4, MPI_UNSIGNED_CHAR, slave_rank, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    int terminate = -1;
    int row_range[2] = {terminate, 0};
    MPI_Send(&terminate, 1, MPI_INT, slave_rank, 1, MPI_COMM_WORLD);
}
```

在 master 實作中，每次派發任務後，master 也會負擔少數運算任務(兩列)，等待運算完畢後，master 會等待其他 slave 的工作需求。這個設計使 master 也能參與運算任務，有效利用資源。

```
next_row += rows_to_send;

if(next_row>=height)
    break;
start_row =  next_row;

#pragma omp parallel for num_threads(num_threads) collapse(2) schedule(dynamic)
    for (int i = start_row; i < start_row + myWork; ++i) {
```

以下為 slave 的實作，slave 會向 master 發送要求工作的訊息，若有計算成果，將在下一次 send 傳輸計算成果。接著，等待 master 分派工作。

```
int  jobRequest[4] = {rank,hasProduct,start_row,num_rows};
MPI_Send(jobRequest, 4, MPI_INT, 0, 0, MPI_COMM_WORLD);
if(hasProduct){
    MPI_Send(raw_image + start_row * width * 4, num_rows * width * 4, MPI_UNSIGNED_CHAR, 0, 3, MPI_COMM_WORLD);
}
MPI_Recv(row_range, 2, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

110062118
資工 25 賴姿妘

2. 分析

a. Balance

✧ Balance between process

以下測試以-N 2 -n 4 -c 6 6 3.726 0.511 -0.096 0 0 0 1024 1024 參數測試，測量每個 process 用於計算 pixel 的時間。首先是 static 方法的結果，可以看見，各個 process 的 loading balance 非常的差。

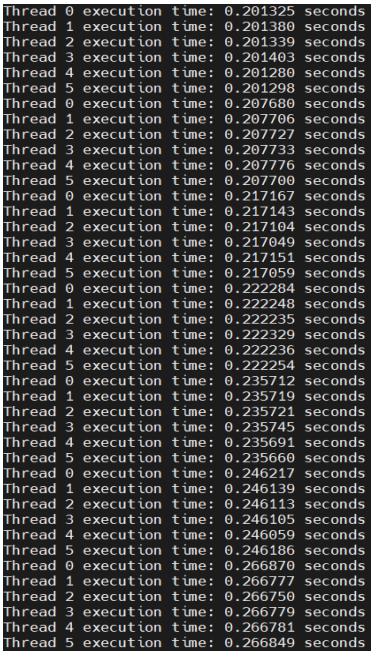
rank 0 time 6.304003s	rank 2 time 46.412061s
rank 1 time 45.109483s	rank 3 time 6.285111s

以下則是 dynamic 方法的結果，使用動態分配後，各 slave rank 的 work load 十分平衡。

rank 0 time	rank 2 time 28.964946s
rank 1 time 28.919402s	rank 3 time 29.084492s

✧ Balance between threads

以下測試以-N 2 -n 4 -c 6 6 3.726 0.511 -0.096 0 0 0 1024 1024 參數測試，在 process 之間為 dynamic working allocation 的條件下，測量每個 thread 在每個 chunk 的執行時間。由右圖可以看見，基本上每個 thread 執行時間相近，代表有充分運用各 thread 的運算資源。下表為其中一組 sample：



Thread0 0.247896s	Thread1 0.247605s	Thread 2 0.247923s
Thread 3 0.247916s	Thread 4 0.247943s	Thread 5 0.247884s

110062118
資工 25 賴姿妘

b. Scalability

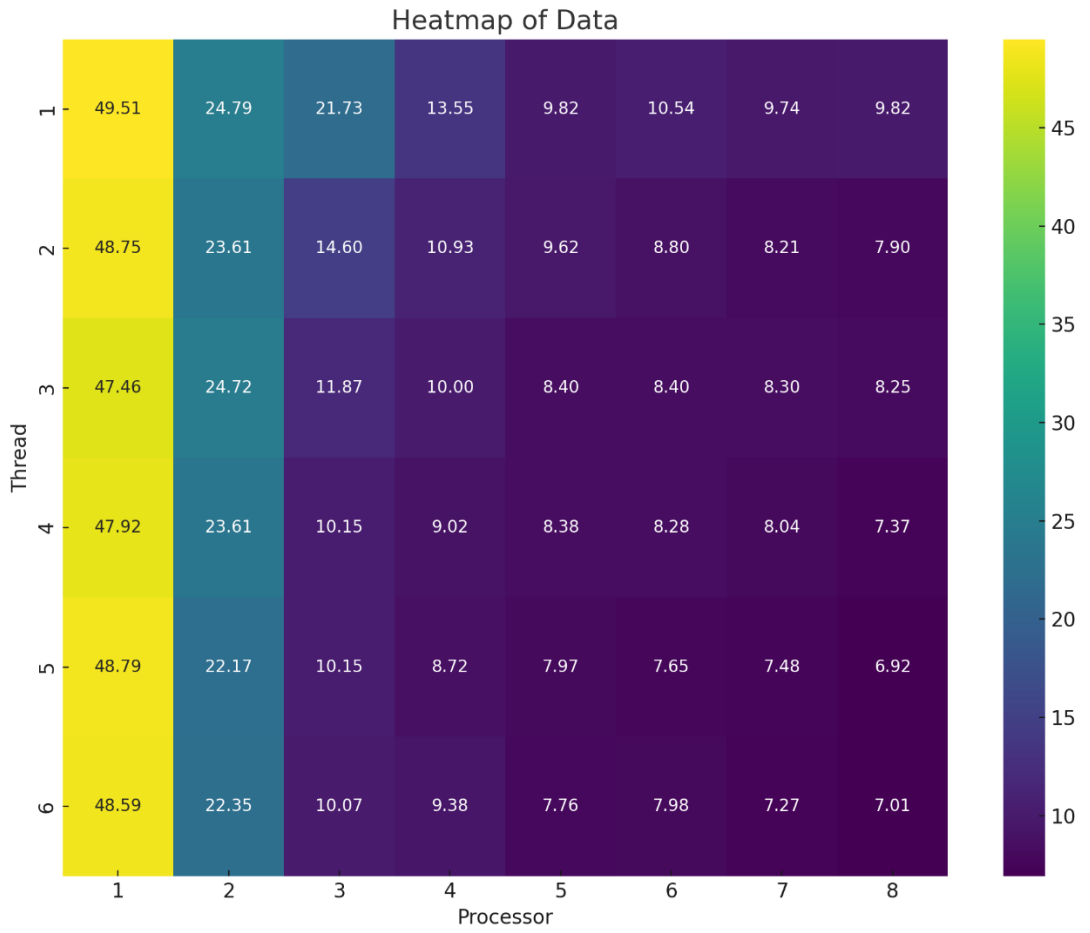
以下實驗基於下列參數進行，”??”代表變數：

srun -A ACD113026 -N ?? -n ?? -c ?? --mpi=pmix ./hw2 ?? 3.726 0.511 -0.096 0 0 0 256 256

✧ Threads and processes

當節點數為 1 時，繪製不同 thread 和 processor 對程式運行速度之影響，圖中數字代表執行時間，單位為秒。由圖可以觀察到我的程式在 thread 為 6 和 processor 為 8 時達到飽和，推測是因為每個 chunk 限制為 10 個 row，因此在使用 dynamic schedule 的狀況下，效能隨 threads 數下降幅度會逐漸趨緩。由於程式架構主要為 master/slave，因此整體 MPI 通訊次數跟時間應該不會隨者 processor 數量增多而有明顯增長，達到飽和應該主要是因為 amdahl’law。

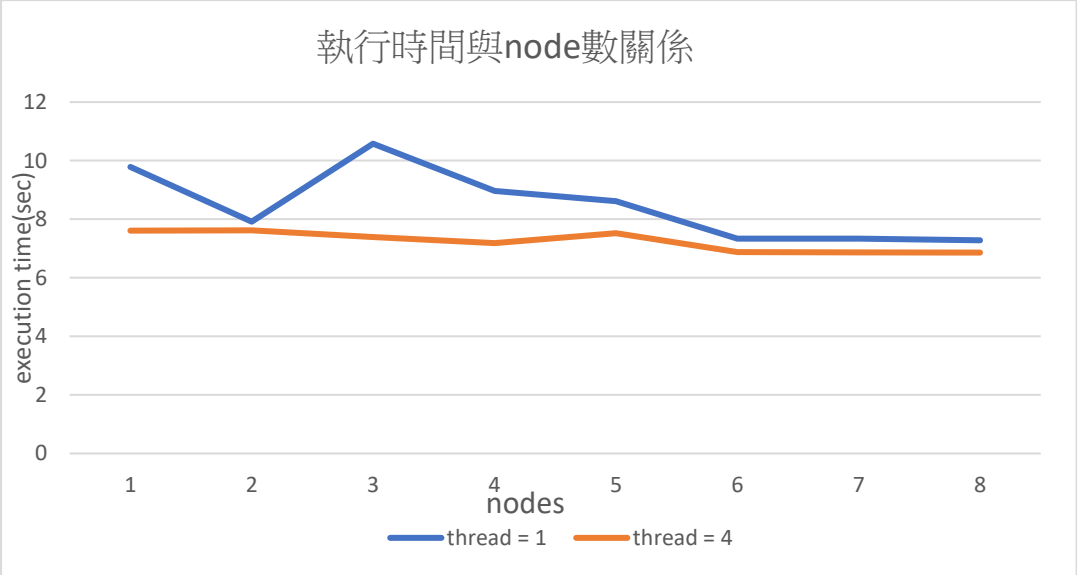
另外，當總執行緒相同時，processor 數越高速度越快，一開始會覺得不合理。有可能是因為 processor 的增加使得記憶體空間更大，減少 miss 的機會，使速度更快。



110062118
資工 25 賴姿妘

✧ Nodes

當 processor 數為 8 時，繪製不同 thread 和 processor 對程式運行速度之影響。當 node 數為 6 時其加速效益達到飽和。增加節點使記憶體增加，減少 miss 的機會，提高執行速度。



3. 結論

✧ 收穫與困難

在這次作業，使用 MPI 實作 processor 之間的平行化，並使用 message passing 的方式進行資料傳遞。這些都是之前作業系統有聽過的概念，能在作業中實際操作頗有收穫。除此之外，思考如何達到 MPI 的動態任務分配，從中也學到很多。

由於這次較晚開始寫作業，沒有太多時間可以嘗試比較漂亮的做法，前期在慢慢熟悉 MPI 的用法，還有 trace code，因此花了不少時間。我一直在思考如何有效利用 master/slave 的 master，或者是有沒有方法實作 process 之間的 task stealing。我覺得這次可惜的是沒有嘗試非同步的 send,recv 不知道會不會對運算速度有幫助。