多處理機平行程式設計 作業二 report

F74109032 陳均哲

Problem 1

1. 程式說明

圖片資訊傳送

最一開始Process 0讀進圖檔時，得到的圖片資訊(bmpInfo.biHeight, bmpInfo.biWeight)會利用MPI_Scatterv的方式傳送給其他process。然後就可以利用這個資訊計算每個process分別的local height是多少。

```
int low = total_height * my_rank / comm_sz;
int high = total_height * (my_rank+1) / comm_sz;
int local_height = high - low;
```

每個Process的local height則是用這個算法得到的。

記憶體分配

```
BMPSaveData = alloc_memory(local_height + 2, width);
BMPData = alloc_memory(local_height + 2, width);
MPI_Scatterv(*BMPSaveData, send_cnt, displs, MPI_CHAR, BMPData[1],
    local_height * width * 3, MPI_CHAR, 0, MPI_COMM_WORLD);
```

在alloc memory的時候我都有除了local height以外多挖兩列，目的是在計算的時候讓第一列跟最後一列儲存上下process的邊界暫存器。另外，讀圖檔的地方alloc memory給BMPSaveData的地方我也有多挖兩列(Process 0)，主要是處理只有一個process的情況。

然後利用MPI_Scatterv再把process 0讀進的圖檔資料分散給每個process。參數設定上是以MPI_CHAR為單位傳送，每一個process需要得到size是width * 3 * local_height。而我把資料都傳送到從BMPData[1]開始存入，預留第一列跟最後一列儲存上下process的邊界暫存器。

邊界暫存器處理

```
void edge_reg(RGBTRIPLE **data, int height, int width, int id, int size){
    int upper = (id + size - 1) % size;
    int lower = (id + 1) % size;
    if (size == 1){
        for (int i = 0; i < width; i++){
            data[0][i] = data[height][i];
            data[height + 1][i] = data[1][i];
        }
        return;
    }

    //將下界傳給下個process上界
    MPI_Send(data[height], width * 3, MPI_CHAR, lower, 0, MPI_COMM_WORLD);
    MPI_Recv(data[0], width * 3, MPI_CHAR, upper, 0,
```

```
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    //將下上界傳給上個process上界
    MPI_Send(data[1], width * 3, MPI_CHAR, upper, 0, MPI_COMM_WORLD);
    MPI_Recv(data[height + 1], width * 3, MPI_CHAR, lower, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    return;
}
```

前面先計算要得到邊界的其他process資料，然後把自己的上界和下界MPI_Send給上面和下面的process，MPI_Recv的部分則是接收上面process的下界以及下面process process的上界。另外如果只有一個process的情況就是單純把自己圖片的資訊複製到第一列以及最後一列。

運算及結果處理

每次處理運算前，我會讓BMPData設定成要拿去計算的資料，而BMPSaveData則設定成計算資料的儲存空間。每次計算完再利用swap()，讓兩者資料交換再迴圈執行下一輪運算。

```
MPI_Gatherv(BMPData[1], local_height * width * 3, MPI_CHAR,
    *BMPSaveData, send_cnt, displs, MPI_CHAR, 0, MPI_COMM_WORLD);
```

最後則是利用MPI_Gatherv把每一個Process的儲存資料傳送給Process 0的BMPData，最後再輸出圖片。因為最後一輪運算完swap過的關係，BMPData其實是運算完儲存過的資料，因此要切除上下邊界，從BMPData[1]開始傳送size一樣是local_height * width * 3，傳送給process 0 的 BMPSaveData，這樣後面可以直接輸出圖片。

2. 時間比較

平滑次數 = 100

comm_sz=	Serial	2	4	8	16	32	40
time(sec)	4.55149	2.3388	1.18417	0.640583	0.650687	1.44339	1.78289

平滑次數 = 1000

comm_sz=	Serial	2	4	8	16	32	40
time(sec)	46.0631	23.6162	12.0484	6.82877	6.75	10.3207	9.83671

3. 結果分析

處理這個圖片的計算因為次數較多，整體來說使用平行化時間應該會隨process數量增加而時間減少，也會隨平滑次數增加而執行時間增加。從comm_sz = 1~8的情況下可以看出來的確是如此，process數量與執行時間大致呈現成反比，至於執行時間也明顯要跟平滑次數成正比。但comm_sz = 16以後就開始持平，到comm_sz = 32以上時間還變長。

但我觀察到的是在cluster上測時間從remote看CPU Load常常都會到100%或快滿，因此測出來的時間容易不精準也很難精確地對時間結果做分析。當cpu load狀況比較好我自己執行程式的時候，使用comm_sz = 4以下cpu load 可以幾乎都在20%左右，因此可以執行時間較快。當comm_sz = 8時大蓋就會到快55%，此時時間還是很容易不準，但狀況好的情況我測到六秒多，而當我使用comm_sz = 16以上就都會到100%，因此comm_sz = 16以上幾乎都會變慢。除了執行自己的程式如果占用比較多核心跑，

每個核心同時在跑程式且核心間溝通次數多，造成耗cpu load高的問題以外，尤其是剛好也有別人在測搶資源又會特別慢。以上的時間結果已經是狀況比較好的情況下測出的數據了。

Problem 2

1. 程式說明

初始資料處理

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

一開始讓process 0得到n的資訊，接著利用MPI_Bcast廣播到所有的process。
接著每個process的local_size都是n/comm_sz。

array 記憶體處理

```
int *local_array = (int*)malloc(sizeof(int) * local_size);
int *buffer_array = (int*)malloc(sizeof(int) * local_size);
int *merged_array = (int*)malloc(sizeof(int) * local_size * 2);
int *global_array = (int*)malloc(sizeof(int) * local_size * comm_sz); //process 0
```

這邊我對每個process malloc出local_array, buffer_array以及merged_array，local_array是儲存自己local的keys。buffer_array是用來接收partner資料的array，merged_array則是拿local_array跟buffer_array合併後產生的結果array，因此malloc的size為其他兩者的兩倍。最後global_array則是只有在process 0有malloc出空間，目的是把其他process的local_array Gather到process 0儲存的位置。

local array 初始

```
srand(my_rank+1);
for (int i = 0; i < local_size; i++){
    local_array[i] = rand();
}
qsort(local_array, local_size, sizeof(int), cmp);
```

一開始設定亂數種子讓每個process的亂數產生不一樣。接著對每個local_array利用rand()得到local_size個keys。最後我對每個local_array進行local sort的方式是利用<stdlib.h>的qsort(), 複雜度O(n log(n))。

merge array

```
void merge(int *arr, int *arr1, int *arr2, int size){
    int i=0, j=0, k=0;
    while(i < size && j < size){ //當兩個陣列都還有剩餘元素
        //把小的數值先填進代表合併後的陣列
        if (arr1[i] < arr2[j]){
            arr[k++] = arr1[i++];
        }
        else {
            arr[k++] = arr2[j++];
        }
    }
    while (i < size){ //若第一個陣列還有元素且第二個陣列已經填完
        arr[k++] = arr1[i++];
    }
```

```
    while (j < size){ //若第二個陣列還有元素且第一個陣列已經填完
        arr[k++] = arr2[j++];
    }
    return;
}
```

我是用size來記錄array的空間有沒有填滿，然後設定i表示local_array的index，j表示buffer_array的index，k表示merged_array的index。而因為local_array跟merged_array都是已經排序過的陣列，就local_array[i]跟buffer_array[j]比較，比較小的key丟到merged_array，然後比較小的那個array index增加。一直持續做到有一方的keys已經全數存到merged_array，再把還沒全數存進去那個array剩餘的keys填滿。

phase 迴圈

```
partner = compute_partner(phase, my_rank, comm_sz);
if (partner == MPI_PROC_NULL) continue;

//把rank較大的process資料傳給rank小的prcoess計算，最後再把merge完比較大的keys取回
if (my_rank > partner){
    //rank較大的process
    MPI_Send(local_array, local_size, MPI_INT, partner, 0,
        MPI_COMM_WORLD);
    MPI_Recv(local_array, local_size, MPI_INT, partner, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else{
    //rank較小的process
    MPI_Recv(buffer_array, local_size, MPI_INT, partner, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    merge(merged_array, local_array, buffer_array, local_size);
    MPI_Send(&merged_array[local_size], local_size, MPI_INT,
        partner, 0, MPI_COMM_WORLD); //傳送較大的keys回去
    for (int i = 0; i < local_size; i++){
            local_array[i] = merged_array[i]; //複製較小的keys到自己的暫存器
    }
}
```

以上是每個phase的處理程式碼。一開始先利用講義上compute_partner的程式碼得到自己要交換資料partner，得到的會是自己的上一個process或下一個process。接著處理如果沒有partner的情況，就直接continue到迴圈下一次迭代才處理。

而主要交換資料的處理方式是我把process id 跟 partner比較。把rank較大的process資料傳給rank小的prcoess處理因此把rank大process的local_array利用MPI_Send傳給partner，partner則利用MPI_Recv把資料儲存在buffer_array。接著把local_array和buffer_array進行merge，把資料存在merged_array，最後把上半部分比較大的keys利用MPI_Send傳回rank大的process，原本process則直接local_array儲存資料取代原本local_array。rank較小的那個process則複製merged_array的前半部分到local_array。

資料輸出

```
MPI_Gather(local_array, local_size, MPI_INT, global_array,
    local_size, MPI_INT , 0, MPI_COMM_WORLD);
```

主要是利用MPI_Gather把每個process的local_array傳送到process 0 global_array裡面,
然後再印出陣列裡面的元素。

2. 時間比較

n=	Serial	2	4	8	16	32	40
10^5	0.013292	0.007814	0.004151	0.003036	0.005538	0.110144	0.113007
10^6	0.109854	0.059332	0.037479	0.031784	0.026022	0.931042	1.064249
10^7	1.191164	0.678763	0.391054	0.334367	0.266103	9.815960	10.828006

3. 結果分析

整個程式的運作流程大致上是先每個process操作local sort，然後跑phase迴圈proc_num次，每次merge兩個local array。操作local sort用qsort複雜度O((n/comm_sz) log(n/comm_sz))，而跑phase迴圈複雜度是O(n)，因為merge的複雜度O(n/comm_sz)，總共跑comm_sz次迴圈。因此總合起來整體的複雜度是O((n/comm_sz) log(n/comm_sz) + n)。從上面的數據，可以看出當n變成10倍，執行時間也大約也變成10倍，對照前面得到的結論還算是合理的。另外從這個結論，當comm_sz越高，執行時間理應越少。因此從以上這些結果，當comm_sz = 1~8，執行時間都有穩定變少，comm_sz = 16時則跟8差不多，但當comm_sz = 32以上時執行時間直接暴增。

我認為其中一個可能的原因是因為還要加上communication的時間。當comm_sz = 32以上時，qsort所花的時間已經沒有比16快很多了，取而代之的是在communication花的時間可能大約是兩倍。因此執行時間增加較多。另外一個我認為的原因就是cpu load的問題，當跑32個以上的process時，cpu已經承受不太住這麼多process同時執行並且互相溝通所耗的效能，因此執行時間才會大幅增加。