

多機平行處理 HW2

1. 影像平滑

程式說明

```
MPI_Datatype mpi_row;  
MPI_Type_contiguous(3 * bmpInfo.biWidth, MPI_UNSIGNED_CHAR,  
&mpi_row);  
MPI_Type_commit(&mpi_row);
```

宣告 mpi derived data mpi_row 來傳送一行圖片資料

```
partner1 = id - 1 >= 0 ? id - 1 : id - 1 + numprocs;  
partner2 = id + 1 < numprocs ? id + 1 : 0;
```

紀錄與該 process 交換邊界資料的 process

```
int sendcounts[numprocs], displs[numprocs];  
int elements_pre_proc = bmpInfo.biHeight / numprocs;  
int rem = bmpInfo.biHeight % numprocs;  
for (int i = 0; i < numprocs; i++)  
{  
    sendcounts[i] = elements_pre_proc;  
    displs[i] = i * elements_pre_proc;  
}  
sendcounts[numprocs - 1] += rem;  
int recvcount = sendcounts[id];  
MPI_Scatterv(BMPptr, sendcounts, displs, mpi_row, &BMPSubData[0][0],  
recvcount, mpi_row, 0, MPI_COMM_WORLD);
```

計算 MPI_Scatterv 所需的 sendcounts 和 displs 並分配給各個 process

每個 process 計算 Height / numprocess 行資料，若 Height 不是 numprocs 的倍數則餘數會被分配到最後一個 process

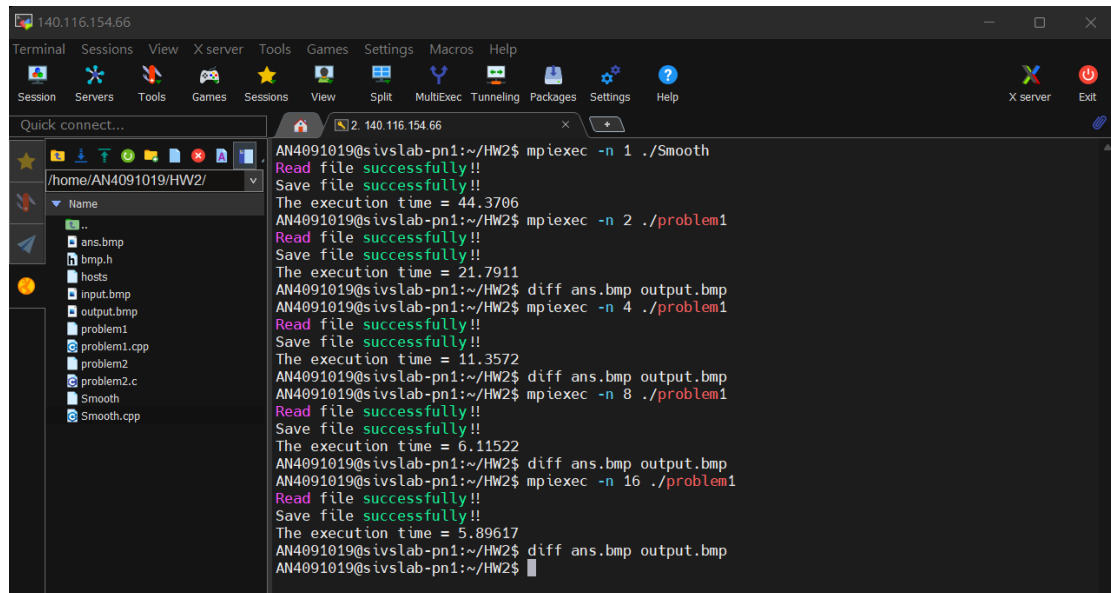
```
MPI_Irecv(&BMPSubData[sendcounts[id]][0], 1, mpi_row, partner2,  
1, MPI_COMM_WORLD, &request[0]);  
MPI_Irecv(&BMPSubData[sendcounts[id] + 1][0], 1, mpi_row,  
partner1, 0, MPI_COMM_WORLD, &request[1]);  
MPI_Isend(&BMPSubData[0][0], 1, mpi_row, partner1, 1,  
MPI_COMM_WORLD, &request[2]);  
MPI_Isend(&BMPSubData[sendcounts[id] - 1][0], 1, mpi_row,  
partner2, 0, MPI_COMM_WORLD, &request[3]);  
MPI_Waitall(4, request, status);
```

在進行每次平滑運算之前會將邊界資料用 non-blocking send 傳送給鄰近的兩個 process

```
MPI_Gatherv(&BMPSubData[0][0], recvcnt, mpi_row, BMPptr,  
sendcounts, displs, mpi_row, 0, MPI_COMM_WORLD);
```

將計算後的結果存回 process 0

執行結果:

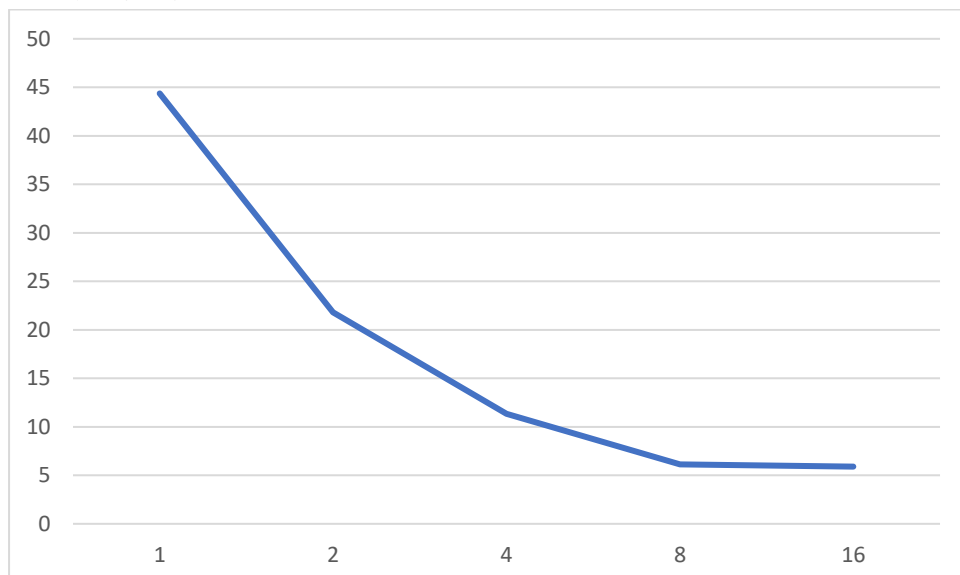


The screenshot shows a terminal window with the following commands and outputs:

```
AN4091019@sivslab-pn1:~/HW2$ mpiexec -n 1 ./Smooth  
Read file successfully!!  
Save file successfully!!  
The execution time = 44.3706  
AN4091019@sivslab-pn1:~/HW2$ mpiexec -n 2 ./problem1  
Read file successfully!!  
Save file successfully!!  
The execution time = 21.7911  
AN4091019@sivslab-pn1:~/HW2$ diff ans.bmp output.bmp  
AN4091019@sivslab-pn1:~/HW2$ mpiexec -n 4 ./problem1  
Read file successfully!!  
Save file successfully!!  
The execution time = 11.3572  
AN4091019@sivslab-pn1:~/HW2$ diff ans.bmp output.bmp  
AN4091019@sivslab-pn1:~/HW2$ mpiexec -n 8 ./problem1  
Read file successfully!!  
Save file successfully!!  
The execution time = 6.11522  
AN4091019@sivslab-pn1:~/HW2$ diff ans.bmp output.bmp  
AN4091019@sivslab-pn1:~/HW2$ mpiexec -n 16 ./problem1  
Read file successfully!!  
Save file successfully!!  
The execution time = 5.89617  
AN4091019@sivslab-pn1:~/HW2$ diff ans.bmp output.bmp  
AN4091019@sivslab-pn1:~/HW2$
```

在 process 數少的時候可以看到平行化是可以快兩倍的速度，在 process 數 = 8 之後效能就提升就非常少

更新邊界資料的傳送次數是 $O(N_{\text{Smooth}}(\text{常數定值}) * \text{numprocs})$ 驗證效能一開始可以提升兩倍



遇到困難:

不知道為什麼在使用 MPI_Scatter 的時候如果傳 BMPSaveData[0] 當第一個 parameter 他就會報錯，但如果我拿另一個 RGBTRIPLE * 指標指向同樣 BMPSaveData[0] 再傳進去就可以正常執行

後來也有拿二維 int 陣列測試也是一樣用 Pointer to Pointer 相關的都會報錯
正常:

```
RGBTRIPLE *BMPptr = BMPSaveData[0];  
MPI_Scatterv(BMPptr, sendcounts, displs, mpi_row, &BMPSubData[0][0],  
recvcount, mpi_row, 0, MPI_COMM_WORLD);
```

報錯:

```
MPI_Scatterv(BMPSaveData[0], sendcounts, displs, mpi_row,  
&BMPSubData[0][0], recvcount, mpi_row, 0, MPI_COMM_WORLD);
```

2. Odd Even Sort

程式說明:

```
#define MAX 10000
```

隨機分發數時的最大值，預設為 10000

```
void oddEvenSort(int *arr, int n)  
{  
    int phase, i, tmp;  
    for (phase = 0; phase < n; phase++)  
    {  
        if (phase % 2 == 0)  
        {  
            for (i = 1; i < n; i += 2)  
            {  
                if (arr[i - 1] > arr[i])  
                {  
                    tmp = arr[i - 1];  
                    arr[i - 1] = arr[i];  
                    arr[i] = tmp;  
                }  
            }  
        }  
        else  
        {  
            for (i = 1; i < n - 1; i += 2)  
            {  
                if (arr[i] > arr[i + 1])  
                {  
                    tmp = arr[i + 1];  
                    arr[i + 1] = arr[i];  
                    arr[i] = tmp;  
                }  
            }  
        }  
    }  
}
```

執行 oddEvenSort 的 function

```
int num_per_process = n / numprocs;
int rem = n % numprocs;
int sendcounts[numprocs], displs[numprocs];
for (int i = 0; i < numprocs; i++)
{
    sendcounts[i] = num_per_process;
    displs[i] = i * num_per_process;
}
sendcounts[numprocs - 1] += rem;
```

因為最後會用 `MPI_Scatterv` 將 `sort` 結果回傳所以計算 `sendcounts` 和 `displs`
每個 `process` 會處理 $n / \text{numprocs}$ 個數，若 n 不是 `numprocs` 的倍數則餘數
會加在最後一個 `process`

```
local = (int *)malloc(sizeof(int) * (2 * num_per_process + rem));
```

每個 **process** 各自用來儲存被分配到的數的陣列，任兩個 **process** 被分配到的數總和的最大

```
oddEvenSort(local, sendcounts[id]); // sort local array
```

將各 process 中的 local 先 sort

```
int oddrank, evenrank;

if (id % 2 == 0)
{
    oddrank = id - 1;
    evenrank = id + 1;
}
else
{
    oddrank = id + 1;
    evenrank = id - 1;
}

if (oddrank == -1 || oddrank == numprocs)
```

```

    oddrank = MPI_PROC_NULL;
    if (evenrank == -1 || evenrank == numprocs)
        evenrank = MPI_PROC_NULL;

```

計算在奇偶數 phase 時交換的對象

```

    if (phase % 2 == 0)
    {
        if (evenrank == MPI_PROC_NULL)
            continue;
        if (id % 2 == 0)
        {
            MPI_Recv(&local[sendcounts[id]], sendcounts[evenrank],
MPI_INT, evenrank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            oddEvenSort(local, sendcounts[id] +
sendcounts[evenrank]);
            MPI_Send(&local[sendcounts[id]], sendcounts[evenrank],
MPI_INT, evenrank, 0, MPI_COMM_WORLD);
        }
        else
        {
            MPI_Send(local, sendcounts[id], MPI_INT, evenrank, 0,
MPI_COMM_WORLD);
            MPI_Recv(local, sendcounts[id], MPI_INT, evenrank, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }

```

在偶數 phase 時先將奇數 process 的陣列加到偶數 process 中，兩個陣列的數一起執行 odd even sort 再將排後面的數回傳到奇數 process

*奇數 phase 大同小異

```

MPI_Gatherv(local, sendcounts[id], MPI_INT, arr, sendcounts, displs,
MPI_INT, 0, MPI_COMM_WORLD);

```

最後再將所有 local 回傳到 process 0

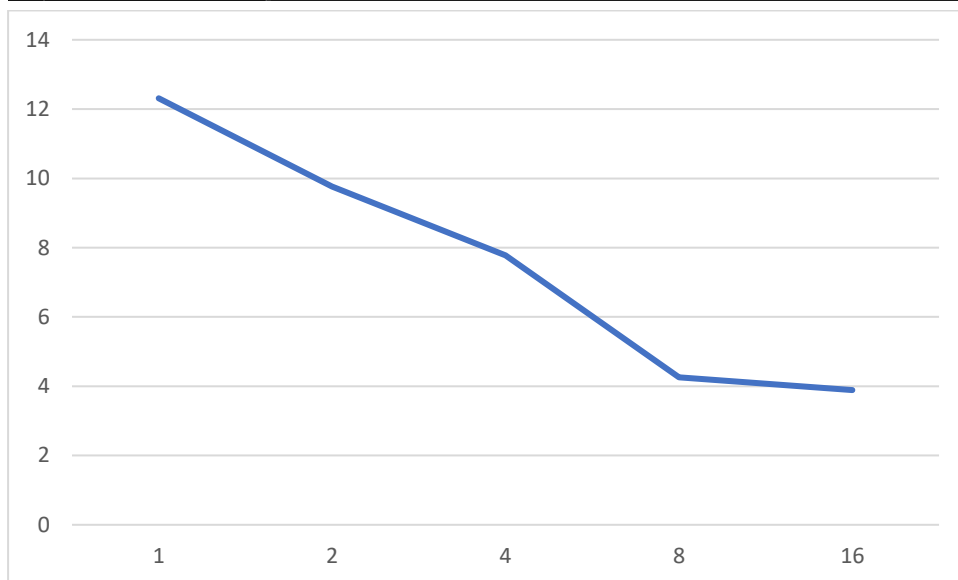
執行結果:

測試執行正確性 n = 30, MAX = 10000:

```
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 1 ./problem2
30
sorted array :
761 847 875 1015 1263 1396 1492 1510 2033 2211 2537 3037 3257 3561 5019 5378 5535 6037 7022 7072 7254 7296 7507 7556 8176 8233 8632 9031 9075 9365
process finish in 0.000057 time
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 2 ./problem2
30
sorted array :
0 567 726 771 814 856 921 1007 1508 1532 1545 2123 2470 2642 2752 2969 3667 3820 3865 4096 4353 4805 4920 5864 6109 6959 7036 7601 8804 9621
process finish in 0.000065 time
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 3 ./problem2
30
sorted array :
347 404 1271 1278 1858 2141 2339 2366 2392 2576 3110 3636 4375 4818 4928 5152 5638 5690 6527 6585 6663 6674 7210 7763 7832 8499 9047 9214 9510 9824
process finish in 0.000062 time
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 4 ./problem2
30
sorted array :
186 392 474 639 1930 1997 2400 2513 3097 3229 4206 4422 4608 4766 5104 5279 5564 5625 6051 6316 7549 7691 8056 8301 8476 8491 9282 9343 9404 9939
process finish in 0.000061 time
AN4091019@sisvslab-pn1:~/HW2$
```

測試平行化加速 n = 100000, MAX = 10000000(不輸出 sort 結果):

```
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 1 ./problem2
100000
process finish in 12.312292 time
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 2 ./problem2
100000
process finish in 9.763345 time
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 4 ./problem2
100000
process finish in 7.779359 time
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 8 ./problem2
100000
process finish in 4.258481 time
AN4091019@sisvslab-pn1:~/HW2$ mpiexec -n 16 ./problem2
100000
process finish in 3.889218 time
AN4091019@sisvslab-pn1:~/HW2$
```



可以看到這隻程式平行化的效能並沒有第一隻好，觀察程式可以發現程式中的 Send & Recv 包在一個執行 numprocs 次的 for 迴圈中，代表 numprocs 個 process 分別執行總共是 $O(\text{numprocs}^2)$ 所以在 numprocs 變大的同時傳輸資料的時間加大更多。