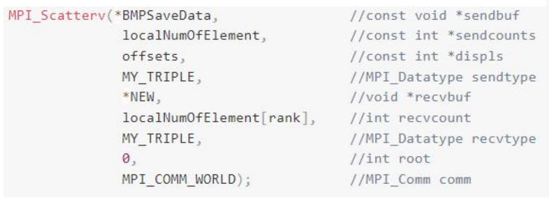
F74109016 **葉惟欣 資訊系大三甲**

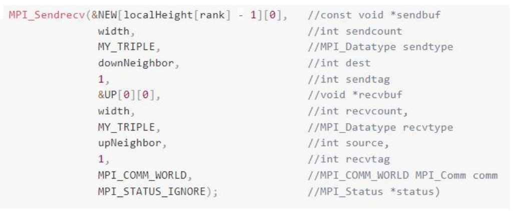
題目一. 影像平滑

1.What have you done。

在這個作業我一開始處理 MPI_Scatterv 的時候花了很多時間去思考，裡面的參數到底要填甚麼，還有 MPI_Gatherv 的函數使用方法。最後



```
MPI_Scatterv(*BMPSaveData,              //const void *sendbuf
             localNumOfElement,         //const int *sendcounts
             offsets,                   //const int *displs
             MY_TRIPLE,                 //MPI_Datatype sendtype
             *NEW,                      //void *recvbuf
             localNumOfElement[rank],   //int recvcount
             MY_TRIPLE,                 //MPI_Datatype recvtype
             0,                         //int root
             MPI_COMM_WORLD);           //MPI_Comm comm
```



```
MPI_Sendrecv(&NEW[localHeight[rank] - 1][0],  //const void *sendbuf
             width,                           //int sendcount
             MY_TRIPLE,                       //MPI_Datatype sendtype
             downNeighbor,                    //int dest
             1,                               //int sendtag
             &UP[0][0],                       //void *recvbuf
             width,                           //int recvcount,
             MY_TRIPLE,                       //MPI_Datatype recvtype
             upNeighbor,                      //int source,
             1,                               //int recvtag
             MPI_COMM_WORLD,                  //MPI_COMM_WORLD MPI_Comm comm
             MPI_STATUS_IGNORE);              //MPI_Status *status)
```

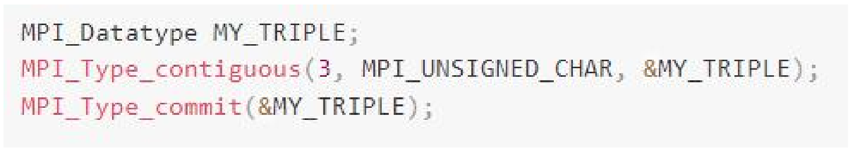
我在寫程式的時候，在後面把官網的參數都寫上去，方便自己去看要傳給誰。還有變數的命名很重要，讓我可以想得比較快，知道現在該用哪個變數放到參數中。

問題:

其中困惑我很久的是 offsets 與 localNumOfElement。因為原本查 RGBTRIPLE 是用 char，但我 offsets 的陣列存放的數量積累總和(相當於從起點到要傳遞的 block 的距離)，與 localNumOfElement(從 offsets 算起後幾個 MPI_TYPE 都是要傳的)用的單位都是 hight / number of processor。所以一直有記憶體碎裂的問題。

解法:

後來我 commit 新的 DataType 來讓一個單位放三個 MPI_CHAR，如此一來單位變得比較清楚後，就再來理解 offsets 和 localNumOfElement 的關係就很快



```
MPI_Datatype MY_TRIPLE;
MPI_Type_contiguous(3, MPI_UNSIGNED_CHAR, &MY_TRIPLE);
MPI_Type_commit(&MY_TRIPLE);
```

問題:

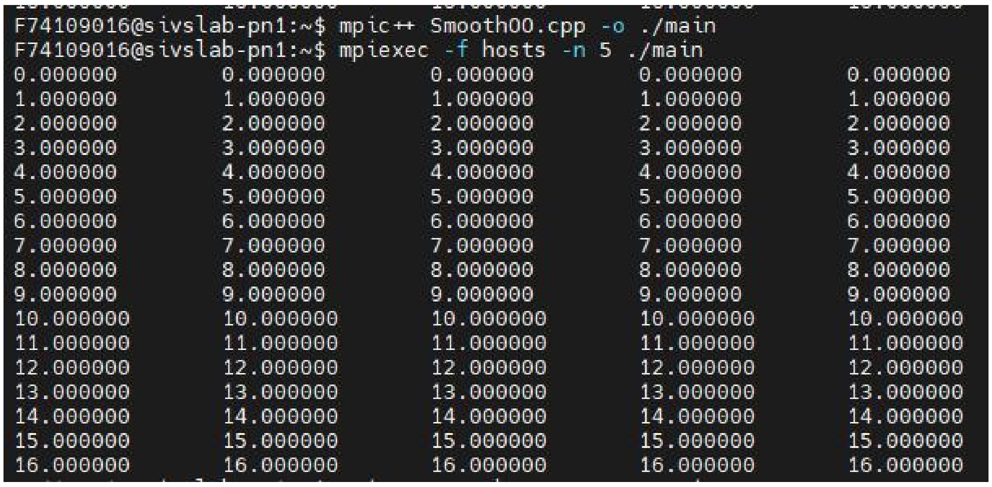
我是用 processor root 讀檔案，再將讀到的資訊傳遞給每個 processor，但在傳遞的時候先是用 Scatterv 來傳遞 block。後來再做平行處理每個 block 的 edge 在座 sendrecv 做資訊交換。然而每次 run 程式一直跑 terminate at signal 9 or 11。

解法:

後來我用 comment 掉每個部份來 debug。可以很清楚資料哪個程式區段有錯。

且我把整個資料型態轉成整數，來做處理。看看每個操作是否有符合我的預期，再將型態改成整數。如下圖，然後陣列元素(整數)也做平滑處理，來檢查是否符合預期的結果。

原先



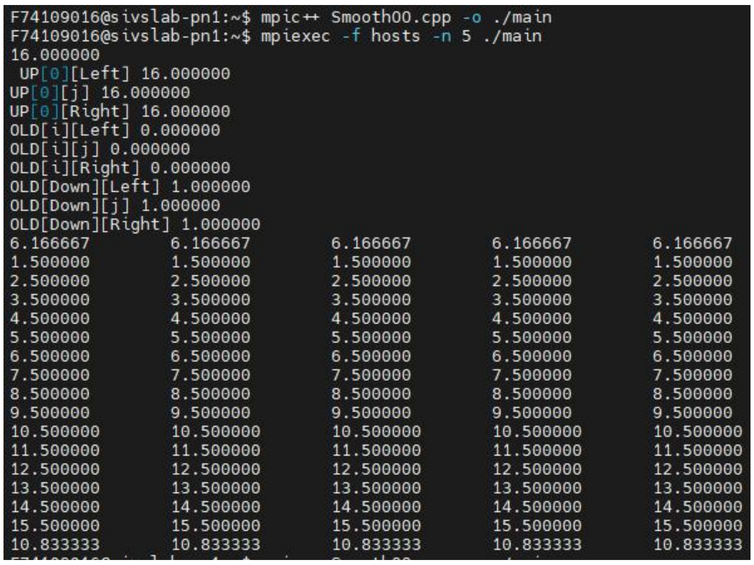
```
F74109016@sivslab-pn1:~$ mpic++ Smooth00.cpp -o ./main
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 5 ./main
0.000000        0.000000        0.000000        0.000000        0.000000
1.000000        1.000000        1.000000        1.000000        1.000000
2.000000        2.000000        2.000000        2.000000        2.000000
3.000000        3.000000        3.000000        3.000000        3.000000
4.000000        4.000000        4.000000        4.000000        4.000000
5.000000        5.000000        5.000000        5.000000        5.000000
6.000000        6.000000        6.000000        6.000000        6.000000
7.000000        7.000000        7.000000        7.000000        7.000000
8.000000        8.000000        8.000000        8.000000        8.000000
9.000000        9.000000        9.000000        9.000000        9.000000
10.000000       10.000000       10.000000       10.000000       10.000000
11.000000       11.000000       11.000000       11.000000       11.000000
12.000000       12.000000       12.000000       12.000000       12.000000
13.000000       13.000000       13.000000       13.000000       13.000000
14.000000       14.000000       14.000000       14.000000       14.000000
15.000000       15.000000       15.000000       15.000000       15.000000
16.000000       16.000000       16.000000       16.000000       16.000000
```

平滑處理一次

每個元素等於 i 值 也相當於原本等於右邊的圖。可以看到平滑處理兩次次結果正確

(16 + 16 + 16 + 0 + 0 + 0 + 1 + 1 + 1)/9 + 0.5 = 5.66666 + 0.5 = 6.1666667

符合預期



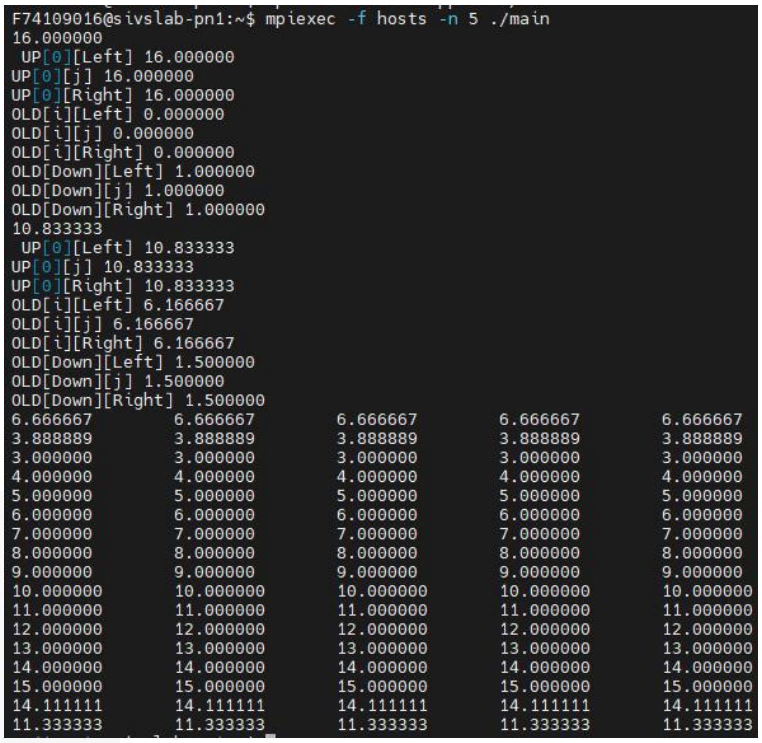
```
F74109016@sivslab-pn1:~$ mpic++ Smooth00.cpp -o ./main
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 5 ./main
16.000000
 UP[0][Left] 16.000000
UP[0][j] 16.000000
UP[0][Right] 16.000000
OLD[i][Left] 0.000000
OLD[i][j] 0.000000
OLD[i][Right] 0.000000
OLD[Down][Left] 1.000000
OLD[Down][j] 1.000000
OLD[Down][Right] 1.000000
6.166667        6.166667        6.166667        6.166667        6.166667
1.500000        1.500000        1.500000        1.500000        1.500000
2.500000        2.500000        2.500000        2.500000        2.500000
3.500000        3.500000        3.500000        3.500000        3.500000
4.500000        4.500000        4.500000        4.500000        4.500000
5.500000        5.500000        5.500000        5.500000        5.500000
6.500000        6.500000        6.500000        6.500000        6.500000
7.500000        7.500000        7.500000        7.500000        7.500000
8.500000        8.500000        8.500000        8.500000        8.500000
9.500000        9.500000        9.500000        9.500000        9.500000
10.500000       10.500000       10.500000       10.500000       10.500000
11.500000       11.500000       11.500000       11.500000       11.500000
12.500000       12.500000       12.500000       12.500000       12.500000
13.500000       13.500000       13.500000       13.500000       13.500000
14.500000       14.500000       14.500000       14.500000       14.500000
15.500000       15.500000       15.500000       15.500000       15.500000
10.833333       10.833333       10.833333       10.833333       10.833333
```

平滑處理兩次

每個元素等於 i 值 也相當於原本等於右邊的圖。可以看到平滑處理兩次次結果正確

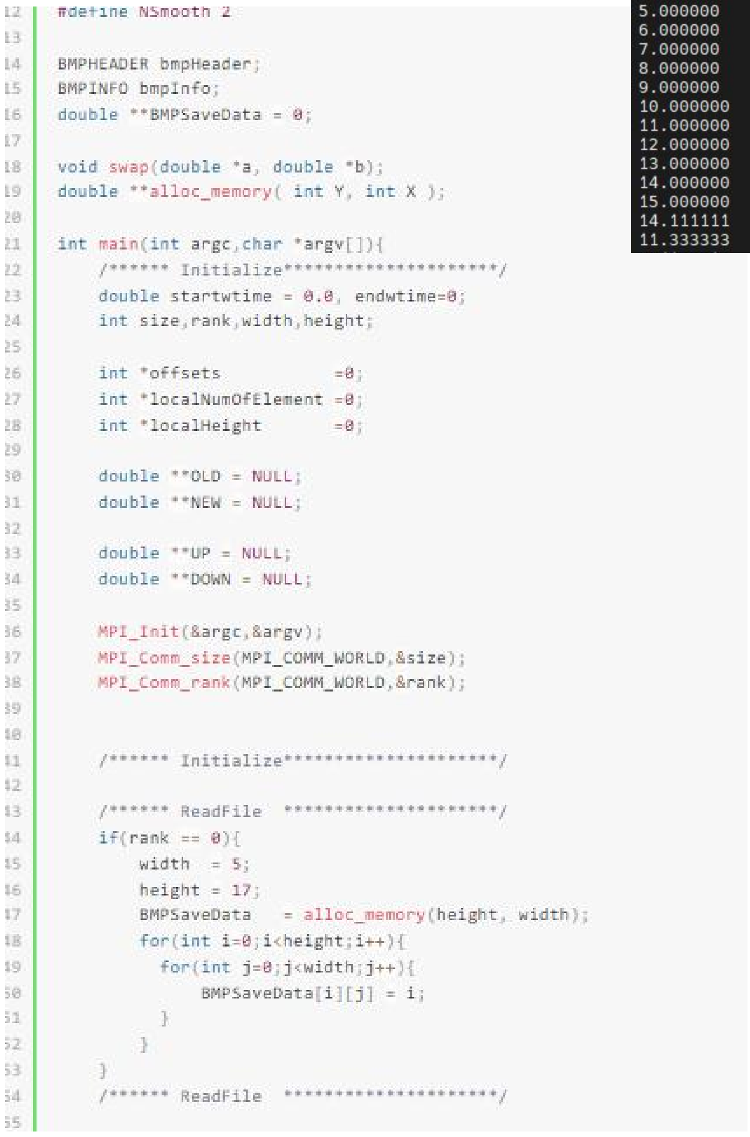
(10.83333+10.83333+10.83333+ 6.1666667+6.1666667+ 6.1666667+ 1.5 + 1.5 + 1.5)/9 + 0.5 = 6.1666667 + 0.5 = 6.6666667

符合預期



```
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 5 ./main
16.000000
 UP[0][Left] 16.000000
UP[0][j] 16.000000
UP[0][Right] 16.000000
OLD[i][Left] 0.000000
OLD[i][j] 0.000000
OLD[i][Right] 0.000000
OLD[Down][Left] 1.000000
OLD[Down][j] 1.000000
OLD[Down][Right] 1.000000
10.833333
 UP[0][Left] 10.833333
UP[0][j] 10.833333
UP[0][Right] 10.833333
OLD[i][Left] 6.166667
OLD[i][j] 6.166667
OLD[i][Right] 6.166667
OLD[Down][Left] 1.500000
OLD[Down][j] 1.500000
OLD[Down][Right] 1.500000
6.666667        6.666667        6.666667        6.666667        6.666667
3.888889        3.888889        3.888889        3.888889        3.888889
3.000000        3.000000        3.000000        3.000000        3.000000
4.000000        4.000000        4.000000        4.000000        4.000000
5.000000        5.000000        5.000000        5.000000        5.000000
6.000000        6.000000        6.000000        6.000000        6.000000
7.000000        7.000000        7.000000        7.000000        7.000000
8.000000        8.000000        8.000000        8.000000        8.000000
9.000000        9.000000        9.000000        9.000000        9.000000
10.000000       10.000000       10.000000       10.000000       10.000000
11.000000       11.000000       11.000000       11.000000       11.000000
12.000000       12.000000       12.000000       12.000000       12.000000
13.000000       13.000000       13.000000       13.000000       13.000000
14.000000       14.000000       14.000000       14.000000       14.000000
15.000000       15.000000       15.000000       15.000000       15.000000
14.111111       14.111111       14.111111       14.111111       14.111111
11.333333       11.333333       11.333333       11.333333       11.333333
```

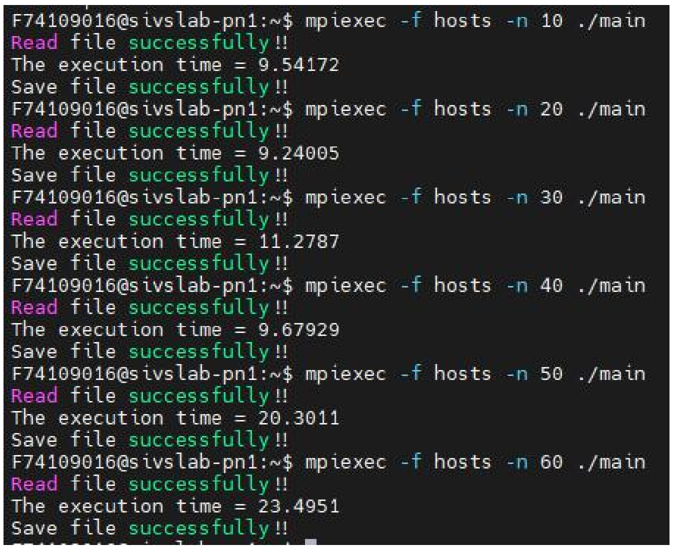
用整數來做測試的程式碼



```cpp
#define NSmooth 2

BMPHEADER bmpHeader;
BMPINFO bmpInfo;
double **BMPSaveData = 0;

void swap(double *a, double *b);
double **alloc_memory( int Y, int X );

int main(int argc,char *argv[]){
    /****** Initialize********************/
    double startwtime = 0.0, endwtime=0;
    int size,rank,width,height;

    int *offsets            =0;
    int *localNumOfElement =0;
    int *localHeight        =0;

    double **OLD = NULL;
    double **NEW = NULL;

    double **UP = NULL;
    double **DOWN = NULL;

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


    /****** Initialize********************/

    /****** ReadFile  ********************/
    if(rank == 0){
        width  = 5;
        height = 17;
        BMPSaveData   = alloc_memory(height, width);
        for(int i=0;i<height;i++){
          for(int j=0;j<width;j++){
              BMPSaveData[i][j] = i;
          }
        }
    }
    /****** ReadFile  ********************/
```

2.Analysis on your result

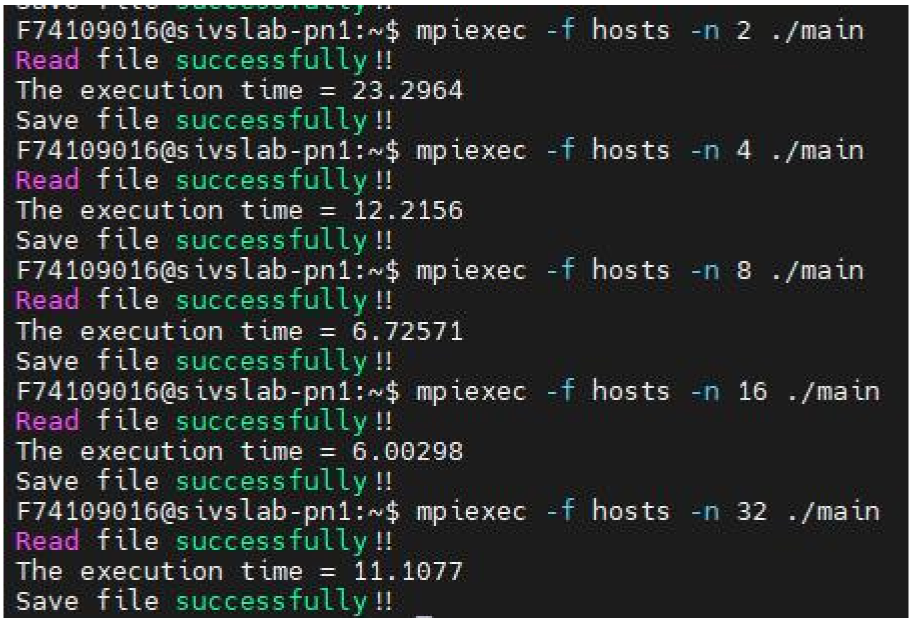
請觀察作業中，改變不同的平滑化的次數與不同數量的處理程序對執行時間有何影響。請寫出觀察結果，並回答為何會有這種結果，寫在結果分析中。



```
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 10 ./main
Read file successfully!!
The execution time = 9.54172
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 20 ./main
Read file successfully!!
The execution time = 9.24005
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 30 ./main
Read file successfully!!
The execution time = 11.2787
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 40 ./main
Read file successfully!!
The execution time = 9.67929
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 50 ./main
Read file successfully!!
The execution time = 20.3011
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 60 ./main
Read file successfully!!
The execution time = 23.4951
Save file successfully!!
```

觀察: processor 數量越多，執行的時間就越多。

分析觀察原因 : 可能是因為 processor 越多，就代表同樣一張圖片每次 smooth 就要做更多次的資料交換(sendrecv)，也因此就會耗更多時間來達到每個切分的 block 的邊緣處理。而 processor 少時，每次 smooth 的總資料交換較少，所以執行時間短

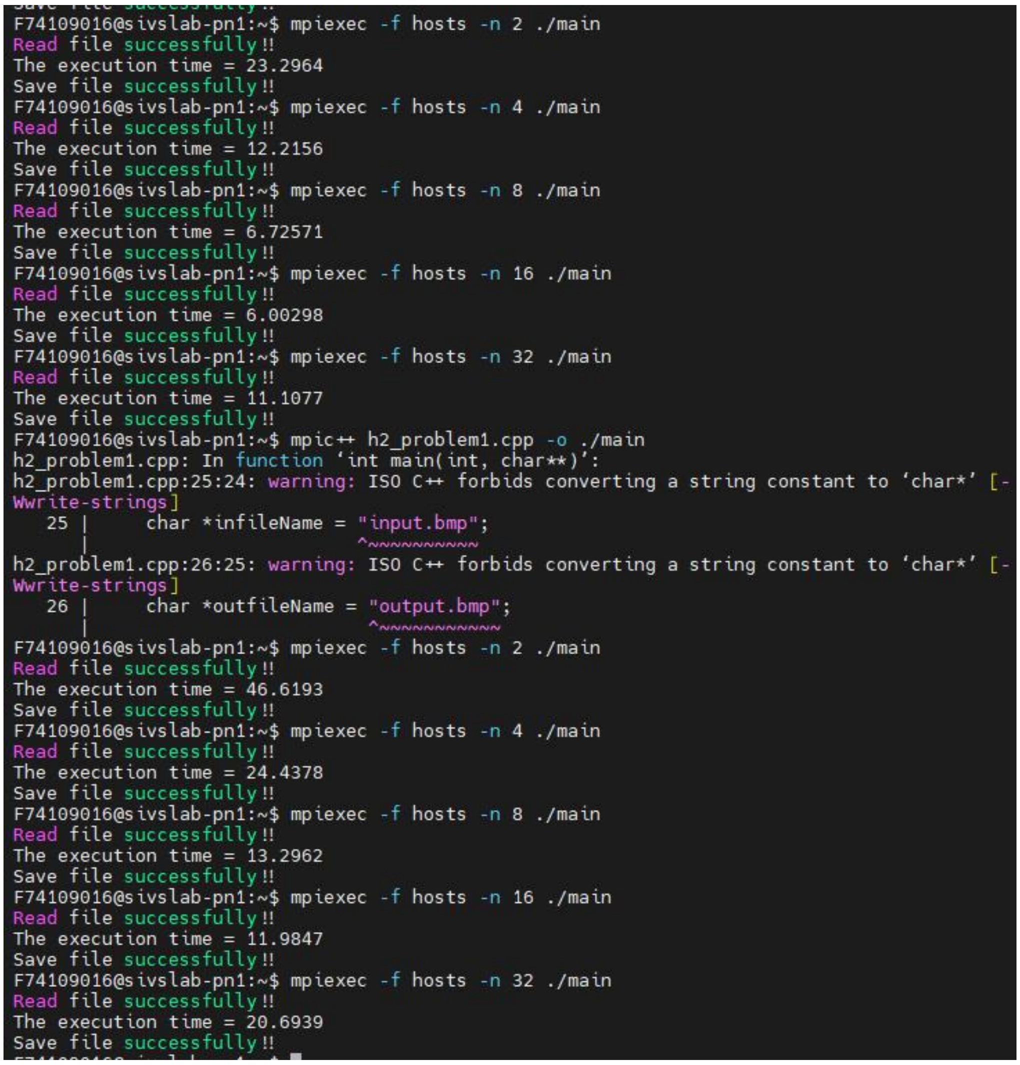


```
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 2 ./main
Read file successfully!!
The execution time = 23.2964
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 4 ./main
Read file successfully!!
The execution time = 12.2156
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 8 ./main
Read file successfully!!
The execution time = 6.72571
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 16 ./main
Read file successfully!!
The execution time = 6.00298
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 32 ./main
Read file successfully!!
The execution time = 11.1077
Save file successfully!!
```

觀察 : 發現較少顆的 processor 時，執行時間隨著 processor 數量變多，有變少的趨勢。

分析觀察原因:結合觀察一的可能分析原因，在 processor 數量還沒有多大，資料交換的 loading 很大時，平行處理是可以增加平滑處理的速度的。但一旦 processor 數量變多，這樣付出的代價遠比增加的效能多。

當平滑次數變成兩倍時 觀察:
smooth 次數從 1000 變成 2000 變成兩倍後執行時間也變為兩倍了，代表平滑處理是線性時間變化的。上面 smooth 1000 下面 smooth 2000



```
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 2 ./main
Read file successfully!!
The execution time = 23.2964
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 4 ./main
Read file successfully!!
The execution time = 12.2156
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 8 ./main
Read file successfully!!
The execution time = 6.72571
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 16 ./main
Read file successfully!!
The execution time = 6.00298
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 32 ./main
Read file successfully!!
The execution time = 11.1077
Save file successfully!!
F74109016@sivslab-pn1:~$ mpic++ h2_problem1.cpp -o ./main
h2_problem1.cpp: In function 'int main(int, char**)':
h2_problem1.cpp:25:24: warning: ISO C++ forbids converting a string constant to 'char*' [-
Wwrite-strings]
   25 |     char *infileName = "input.bmp";
      |                        ^~~~~~~~~~~
h2_problem1.cpp:26:25: warning: ISO C++ forbids converting a string constant to 'char*' [-
Wwrite-strings]
   26 |     char *outfileName = "output.bmp";
      |                         ^~~~~~~~~~~~
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 2 ./main
Read file successfully!!
The execution time = 46.6193
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 4 ./main
Read file successfully!!
The execution time = 24.4378
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 8 ./main
Read file successfully!!
The execution time = 13.2962
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 16 ./main
Read file successfully!!
The execution time = 11.9847
Save file successfully!!
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 32 ./main
Read file successfully!!
The execution time = 20.6939
Save file successfully!!
```

結論:平行程式化，的效能提升非線性，可能有代價。
但平滑影響處理的效能是線性的。

3.Any difficulties?

這次的作業我寫超久才寫出來，最後看了 stackflow 的一篇文章才真正理解傳遞的參數到底是在傳甚麼。網路上有很多人在討論，但每一個 case 要處理的都不一樣。所以我很難用網路上的例子來思考自己究竟錯哪，還是最後通盤了解後，debug 才變快。
參考網址:
https://stackoverflow.com/questions/9269399/sending-blocks-of-2d-array-in-c-using-mpi/9271753#9271753

4.(optional) Feedback to TAs

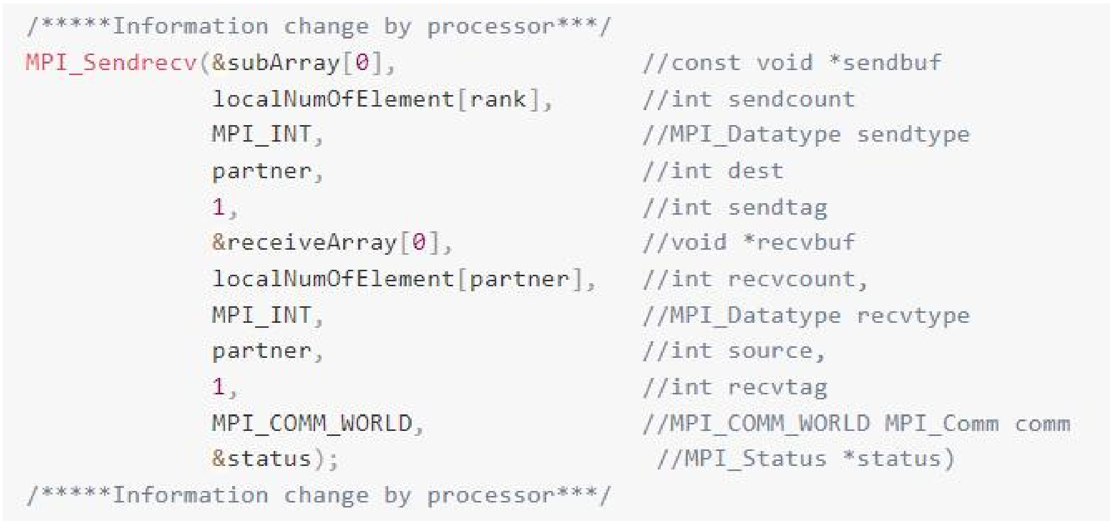
這次的作業我用整數去計算，平滑處理的結果都符合預期，但到 BMP 的時候卻 diff 有不同的結果，想問助教有甚麼可能的原因。

題目二. 平行化 odd-even sort

1.What have you done

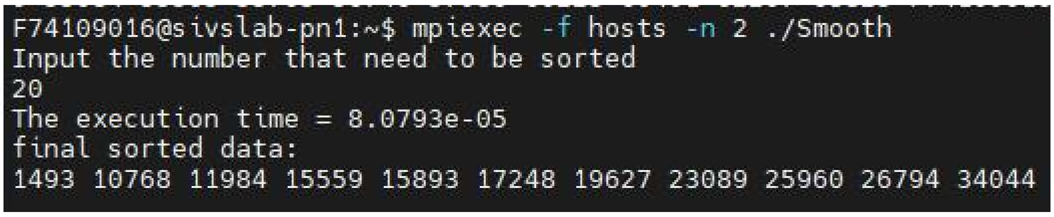
在這一題我又重新想 Sendrecv，
上面的 sendcount 是從要 send 的人的角度去看，他 send 了 sendcount 數量的 MPI_INT。而下面的 recvcount 是從要收的人去看，他要收 localNumOfElement[partner]也就是 partner 的元素數量。

這也就是說在當一個 processor 看到這段程式碼，
他同時是 send 的人，要送自己的數量給別人 localNumOfElement[rank]個元素。
同時也是收的人，他要從 partner 那裏收 localNumOfElement[partner]個元素

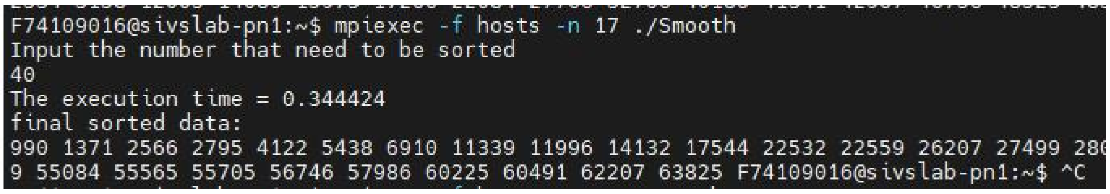


```
/*****Information change by processor***/
MPI_Sendrecv(&subArray[0],                  //const void *sendbuf
             localNumOfElement[rank],       //int sendcount
             MPI_INT,                       //MPI_Datatype sendtype
             partner,                       //int dest
             1,                             //int sendtag
             &receiveArray[0],              //void *recvbuf
             localNumOfElement[partner],    //int recvcount,
             MPI_INT,                       //MPI_Datatype recvtype
             partner,                       //int source,
             1,                             //int recvtag
             MPI_COMM_WORLD,                //MPI_COMM_WORLD MPI_Comm comm
             &status);                       //MPI_Status *status)
/*****Information change by processor***/
```

2.Analysis on your result

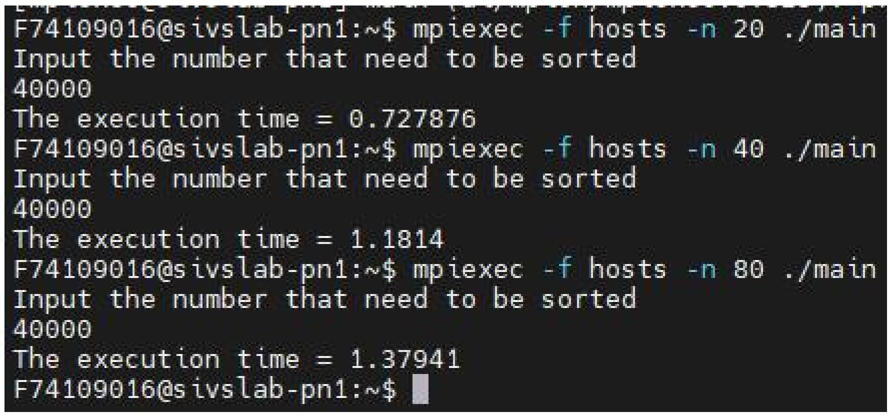


```
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 2 ./Smooth
Input the number that need to be sorted
20
The execution time = 8.0793e-05
final sorted data:
1493 10768 11984 15559 15893 17248 19627 23089 25960 26794 34044
```



```
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 17 ./Smooth
Input the number that need to be sorted
40
The execution time = 0.344424
final sorted data:
990 1371 2566 2795 4122 5438 6910 11339 11996 14132 17544 22532 22559 26207 27499 28
9 55084 55565 55705 56746 57986 60225 60491 62207 63825 F74109016@sivslab-pn1:~$ ^C
```

觀察 : 當要 sort 的數量很少時，其實沒有很顯著的影響，反而分越多 processor，資料溝通會耗時間，遠比用內建的 sort 演算法來處理每個 processor 預處理的時間快。



```
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 20 ./main
Input the number that need to be sorted
40000
The execution time = 0.727876
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 40 ./main
Input the number that need to be sorted
40000
The execution time = 1.1814
F74109016@sivslab-pn1:~$ mpiexec -f hosts -n 80 ./main
Input the number that need to be sorted
40000
The execution time = 1.37941
F74109016@sivslab-pn1:~$
```

觀察 : 當很多 processor 時，時間也沒有變比較快。

分析 : 我認為是 odd-even sort 演算法跟 processor 數量有關，當有多少顆 processor 時就要迴圈跑幾次。不像平時演算法的 merge sort 就是處理數量 log_2(number)。

3.Any difficulties?
要複習 merge sort 的兩個指標花一點小時間，還有重新認識 sendrecv，好像又更熟悉了。
4.(optional) Feedback to TAs
以前在學 merge sort 精隨就是 divide and conquer，現在真的是分給每個 processor 做，感覺很有實作的精神。