Lab 2: MPI Collective Operations

EEL 6763 – Fall 2023

Sliding windows techniques for image processing

Commonly used for image processing, a sliding window algorithm uses a "window" (a rectangular region of fixed width and height) that "slides" across an image, as shown in Figure 1. For each instance of these windows, the value of the pixel of interest for that window is recalculated based on a filter mask applied to that pixel. In Figure 1, the pixel of interest is q2, q3, q4, and q5, respectively, for each of the windows. The mask is generally an equation(s) developed specifically for the filtering operation (e.g., a Sobel filter). The basic concept is to recalculate the value of each pixel of the entire picture based on the values of the current pixel and its adjacent pixels, as the window slides across each pixel.

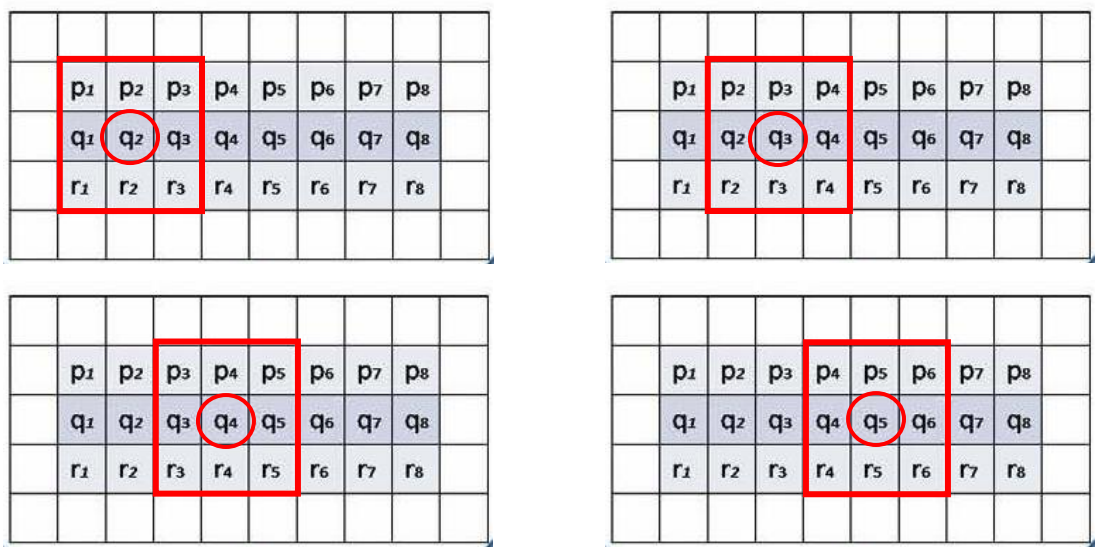

Figure 1. Illustration of a sliding window

For this lab, we will use a simple neighborhood weighted-averaging filter. The operator takes the weighted average of adjacent pixel values, as illustrated in Figure 2.

e' = (a+b+c+d+2e+f+g+h+i)/10

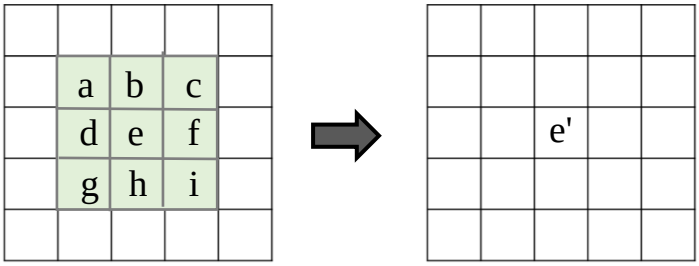

Figure 2. Weighted-averaging filter

Lab specifications:

- Implement the mask filtering operation on an image, represented by an N x N integer matrix, using MPI on R ranks (varying N and R on different runs).
- The mask operations must be distributed as evenly as possible for all ranks, including the master rank.
- Use MPI_Scatterv() to distribute the initial image matrix and use MPI_Gatherv() to collect the processed matrix. These are the only two MPI statements you can use for data transfer (e.g., no MPI_Send or MPI_Recv).
- To keep program simple, it is not required to process the first and last rows and columns.
- The matrix A should be initialized in one of the nodes with synthetic data using the rand() function. To simplify grading (i.e., everyone has the same input array), please initialize with the seed of 1 (the default value) and limit the random numbers to be between 0 - 255.
- You will provide code for four functions (initialize_data, distribute_data, mask_operation, and collect_results) with the following prototypes (you can change the prototypes to fit your coding style):

```
void initialize_data (int *A, int N);
int* distribute_data (int *A, int N);       // returns recv_buff for that rank
int* mask_operation (int *recv_buff, int N);     // returns updated_buff for that rank
void collect_results (int *updated_buff, int N, int *Ap);     // *Ap is processed matrix
```

- Use the following pseudocode for your main function (plus timing functions, etc.). N is size of the NXN Matrix. Depending on how you define function prototypes, you can change how you call them in the main() function.

```
int main(int argc, char **argv)
{  MPI_Init(&argc, &argv);
   int N = atof(argv[1]);
   initialize_data(A, N);
   temp1 = distribute_data (A, N);   // use scatterv
   temp2 = mask_operation(temp1, N);
   collect_results(temp2, N, Ap);   // use gatherv
   MPI_Finalize();
   return 0;
}
```

Hint: The N x N matrix is distributed as "chunks" of rows to each rank to process (say X number of rows of the matrix for this rank). In order to process the pixels in these X rows, this rank must receive X+2 rows (Why?). See Figure 3.

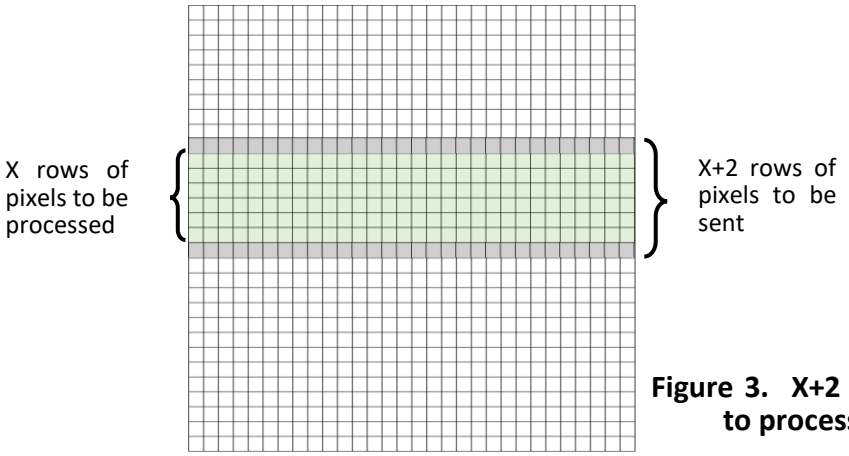

Figure 3. X+2 rows are required to process pixels in X rows.

SUBMISSION INSTRUCTIONS

Make sure your name is at the top of every file, including your report. You are to submit 2 files on Canvas:

(1) Create a directory named **Lab2**. Use the following structure and zip the entire directory and **submit the zip file** on Canvas using the following name: lastNamefirstNameLab2.zip

Lab2/

.c file
batch script
6 output files:

- Keep R = 4, vary N= 15, 100, 1000
- Keep N = 1000, vary R (number of ranks) = (4), 8, 16, 32
- Name each output file appropriately: e.g., OutputR4N15, OutputR32N1000
- Also, to maximize your credit (especially partial credit), use printf statements like they were used in the mat_mult code (e.g., number of elements per rank, partial results sent back by each rank, etc.)

(2) **A pdf file** using the following name: lastNamefirstNameLab2.pdf

- Any information or explanation that you want to give me (e.g., what worked and what did not, etc.)
- A "cleaned-up" complete **output file** just for **R=4, N=15** (copy/paste from your output file), containing ONLY the following (i.e., don't include any debugging outputs):
  - Values of the initial matrix
  - send_counts[0]-send_count[3]; displs[0]-displs[3] for the scatterv() function.
  - For each rank, part of the matrix that were sent to it
  - For each rank, updated values after computation
  - send_counts[0]-send_count[3]; displs[0]-displs[3] for the gatherv() function.
  - Updated values of the final (total) matrix
  - Total time taken

- Two tables – for the results of these two tables, remove all the print statements so they will not affect the total times
  - Table 1 (R = 4); Row labels: N = 15, 100, 1000; Column labels: total time taken

R = 4	
N	Time (sec)
15	
100	
1000	

  - Table 2 (for N = 1000); Row labels: R (# of ranks) = 4, 8, 16, 32; Column labels: total time taken

N = 1000	
R	Time (sec)
4	
8	
16	
32	