Image Denoising

IN3200/IN4200 Default Mandatory Project, Spring 2019

Note: Each student should independently program the required code and write her/his own short report. These are to be submitted at Devilry within the announced deadline. The details about the submission can be found at the end of this document.

1 Introduction

Image denoising refers to the removal (or decrease) of random noises in a "contaminated" image, such that the denoised image more closely resembles the original noise-free image. An example of image denoising is illustrated in Figure 1.

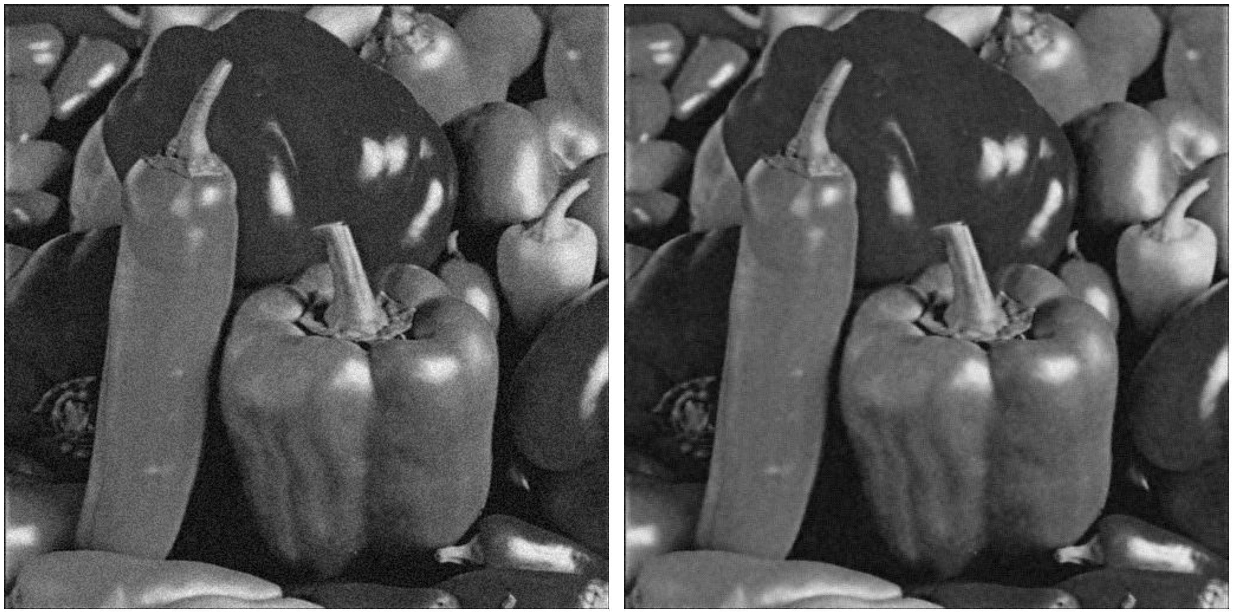

Figure 1: Left: a noisy image; Light: a denoised image.

The purpose of this project is to let the students demonstrate their abilities through the following tasks:

1. Translation of simple mathematical formulas to a working code, for the case of implementaing a simple denoising algorithm;
2. Parallelization of the denoising algorithm via MPI programming;
3. Writing a simple report.

2 A very simple denoising algorithm

An image can be (logically) arranged as a 2D array, containing $m \times n$ pixels:

$$\mathbf{u} = \begin{bmatrix} u_{m-1,0} & u_{m-1,1} & \cdots & u_{m-1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ u_{1,0} & u_{1,1} & \cdots & u_{1,n-1} \\ u_{0,0} & u_{0,1} & \cdots & u_{0,n-1} \end{bmatrix},$$

where each pixel has a scalar value for the case of a grey-scale image. (In this project, we will limit the implementations to grey-scale images only.)

Although there exists a wealth of image denoising algorithms (including methods that are based on deep learning), we will only consider a very simple algorithm named *isotropic diffusion*. This is an iterative procedure where each iteration computes $\bar{\mathbf{u}}$ as a "smoother" version of $\mathbf{u}$. More specifically, the following formula is used to compute $\bar{\mathbf{u}}$ based on $\mathbf{u}$:

$$\bar{u}_{i,j} = u_{i,j} + \kappa \left(u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j} \right).$$

Here, κ is a small scalar constant (such as 0.2 or below). Note that the above formula is used to compute the interior pixels of $\bar{\mathbf{u}}$, that is, $1 \le i \le m-2$ and $1 \le j \le n-2$. The boundary pixels of $\bar{\mathbf{u}}$ can, for simplicity, copy the corresponding boundary pixels of $\mathbf{u}$.

3 Using an external library for reading/writing JPEG images

The students are requested to implement the simple denoising algorithm to handle images in the JPEG format. There is a ready-made external C library package that can be used for, among other things, reading and writing JPEG images. The entire source code that is needed, together with a simple demo example (including simple-to-use Makefiles), can be downloaded from

`https://www.uio.no/studier/emner/matnat/ifi/IN3200/v19/teaching-material/one_folder.tar`

The students are strongly encouraged to download the tarball and try out the demo example under `one_folder/serial_example/`. The source code of the JPEG library is under `one_folder/simple-jpeg/`.

More specifically, the following two functions from the JPEG library will be needed for the project:

```
void import_JPEG_file (const char* filename, unsigned char** image_chars,
                       int* image_height, int* image_width,
                       int* num_components);
void export_JPEG_file (const char* filename, const unsigned char* image_chars,
                       int image_height, int image_width,
                       int num_components, int quality);
```

These two functions are for reading and writing a data file of the JPEG format. We remark that each pixel in a grey-scale JPEG image uses one byte, and a one-dimensional array of `unsigned char` (of total length $m \cdot n$) is used to contain all the pixel data of a grey-scale JPEG image. (In the case of a color JPEG image, a 1D array of `rgbrgbrgb`... values is read in.)

Moreover, the integer variable `num_components` will contain value 1 after the `import_JPEG_file` function finishes reading a grey-scale JPEG image. Value `1` should also be given to `num_components` before invoking `export_JPEG_file` to export a grey-scale JPEG image. (For a color JPEG image, the value of `num_components` is `3`.) We also remark that the last argument `quality` of the `export_JPEG_file` function is an integer indicating the compression level of the resulting JPEG image. A value of 75 is the typical choice of `quality`.

4 Data structure of an image related to denoising

It should be noted that a 1D array of type `unsigned char` is used by the JPEG library for reading and writing an image. A variable of type `unsigned char` only has an integer value between 0 and 255. This is not sufficient for doing accurate denoising computations. To this end, the following data structure should be used to store the $m \times n$ pixel values (of a grey-scale image) in connection with denoising:

```
typedef struct
{
  float** image_data;  /* a 2D array of floats */
  int m;               /* # pixels in vertical-direction */
  int n;               /* # pixels in horizontal-direction */
}
image;
```

5 Submission

Each student should submit, via Devilry, a tarball (named `project_xxx.tar`) or a zip file (named `project_xxx.zip`), with `xxx` being her/his candidate number, that contains at least the following files and sub-folders:

```
project_report_xxx.pdf
README.txt    (Info about compiling/running the serial/parallel codes)
serial_code/
parallel_code/
simple-jpeg/  (The JPEG library source code, excluding *.o *.a files)
```

5.1 Serial implementation

Each student is requested to write a serial program, named `serial_code/serial_main.c`, that has the following skeleton of the `main` function:

```
/* needed header files .... */
/* declarations of functions import_JPEG_file and export_JPEG_file */

int main(int argc, char *argv[])
{
  int m, n, c, iters;
  float kappa;
  image u, u_bar;
  unsigned char *image_chars;
  char *input_jpeg_filename, *output_jpeg_filename;

  /* read from command line: kappa, iters, input_jpeg_filename, output_jpeg_filename */
  /* ... */
  import_JPEG_file(input_jpeg_filename, &image_chars, &m, &n, &c);

  allocate_image (&u, m, n);
  allocate_image (&u_bar, m, n);
  convert_jpeg_to_image (image_chars, &u);

  iso_diffusion_denoising (&u, &u_bar, kappa, iters);

  convert_image_to_jpeg (&u_bar, image_chars);
  export_JPEG_file(output_jpeg_filename, image_chars, m, n, c, 75);

  deallocate_image (&u);
  deallocate_image (&u_bar);

  return 0;
}
```

As can be seen in the above code skeleton, five functions need to be implemented (and placed in a file named `serial_code/functions.c`):

```
void allocate_image(image *u, int m, int n);
void deallocate_image(image *u);
void convert_jpeg_to_image(const unsigned char* image_chars, image *u);
void convert_image_to_jpeg(const image *u, unsigned char* image_chars);
void iso_diffusion_denoising(image *u, image *u_bar, float kappa, int iters);
```

Function `allocate_image` is supposed to allocate the 2D array `image_data` inside `u`, when `m` and `n` are given as input. The purpose of function `deallocate_image` is to free the storage used by the 2D array `image_data` inside `u`.

Function `convert_jpeg_to_image` is supposed to convert a 1D array of `unsigned char` values into an `image` struct. Function `convert_image_to_jpeg` does the conversion in the opposite direction.

The most important function that needs to be implemented is `iso_diffusion_denoising`, which is supposed to carry out `iters` iterations of isotropic diffusion on a noisy image object `u`. The denoised image is to be stored and returned in the `u_bar` object. **Note:** After each iteration (except the last iteration), the two objects `u` and `u_bar` should be swapped.

5.2 Parallel implementation

The students are requested to write a parallel implementation, named `parallel_code/parallel_main.c`, that has the following skeleton of the `main` function:

```
/* needed header files .... */
/* declarations of functions import_JPEG_file and export_JPEG_file */

int main(int argc, char *argv[])
{
  int m, n, c, iters;
  int my_m, my_n, my_rank, num_procs;
  float kappa;
  image u, u_bar, whole_image;
  unsigned char *image_chars, *my_image_chars;
  char *input_jpeg_filename, *output_jpeg_filename;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num_procs);

  /* read from command line: kappa, iters, input_jpeg_filename, output_jpeg_file
name */
  /* ... */

  if (my_rank==0) {
    import_JPEG_file(input_jpeg_filename, &image_chars, &m, &n, &c);
    allocate_image (&whole_image, m, n);
  }

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

  /* 2D decomposition of the m x n pixels evenly among the MPI processes */
  my_m = ...;
  my_n = ...;

  allocate_image (&u, my_m, my_n);
  allocate_image (&u_bar, my_m, my_n);

  /* each process asks process 0 for a partitioned region */
  /* of image_chars and copy the values into u */
  /*  ...  */

  convert_jpeg_to_image (my_image_chars, &u);
  iso_diffusion_denoising_parallel (&u, &u_bar, kappa, iters);

  /* each process sends its resulting content of u_bar to process 0 */
  /* process 0 receives from each process incoming values and */
  /* copy them into the designated region of struct whole_image */
  /*  ...  */

  if (my_rank==0) {
    convert_image_to_jpeg(&whole_image, image_chars);
    export_JPEG_file(output_jpeg_filename, image_chars, m, n, c, 75);
    deallocate_image (&whole_image);
  }

  deallocate_image (&u);
  deallocate_image (&u_bar);

  MPI_Finalize ();
  return 0;
}
```

The functions `allocate_image`, `deallocate_image`, `convert_jpeg_to_image` andd `convert_image_to_jpeg` can be reused from the serial implementation. The new function `iso_diffusion_denoising_parallel` needs to enhance its serial counterpart with necessary MPI communication calls.

5.3 Short report

Each student should write a short report named `project_report_xxx.pdf` (with `xxx` being the candidate number of the student). The report should describe the most important programming info (especially related to the MPI parallelization). Time measurements of function `iso_diffusion_denoising_parallel` should also be included.

Additional requirement for the IN4200 students: Discussion about the code changes/extensions that will be needed for the case of denoising color images, that is, three values per image pixel. (Note: There is no need to write the actual implementation.)

5.4 Grading

The grade of the submission will constitute 20% of the final grade of IN3200/IN4200. Grading of the submission will be based on the correctness, readability and speed of the implementations, in addition to the quality of the short report.

6 Example of a noisy grey-scale image

`https://www.uio.no/studier/emner/matnat/ifi/IN3200/v19/teaching-material/mona_lisa_noisy.jpg`