Image Denoising

IN3200/IN4200 Obligatory assignment 2, Spring 2022

Note: This is the second of the two obligatory assignments that both need to be approved so that a student is allowed to take the final exam. (The grade of the final exam is otherwise *not* related to these two assignments.) Same requirements apply to IN3200 and IN4200 students for this assignment.

Note: Discussions between the students are allowed, but each student should write her/his own implementations. The details about the submission can be found in Section 5 of this document.

1 Motivation

Through this obligatory assignment, each student will get hands-on experience with the following topics:

1. Compilation of existing implementations of C functions (other people's code) as a stand-alone external library.
2. Parallelization and MPI implementation of a simple, real-world numerical algorithm.
3. Use of a parallel computing system (UiO's *Fox* cluster) that has a standard queuing system.

Note: It may take some time to apply for access to the Fox cluster and learn its basic usage. Each student is thus encouraged to start as soon as possible. Information about the Fox cluster and its basic usage can be found in Section 7.

2 A very simple denoising algorithm

Image denoising refers to the removal (or decrease) of random noises in a "contaminated" image. An example of image denoising is illustrated in Figure 1.

Numerically, 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, $1 \leq i \leq m-2$ and $1 \leq j \leq n-2$. The boundary pixels of $\bar{\mathbf{u}}$ can, for simplicity, copy the corresponding boundary pixels of $\mathbf{u}$.

For better results, $\bar{\mathbf{u}}$ can be subject for another denoising step, and so on (many iterations).

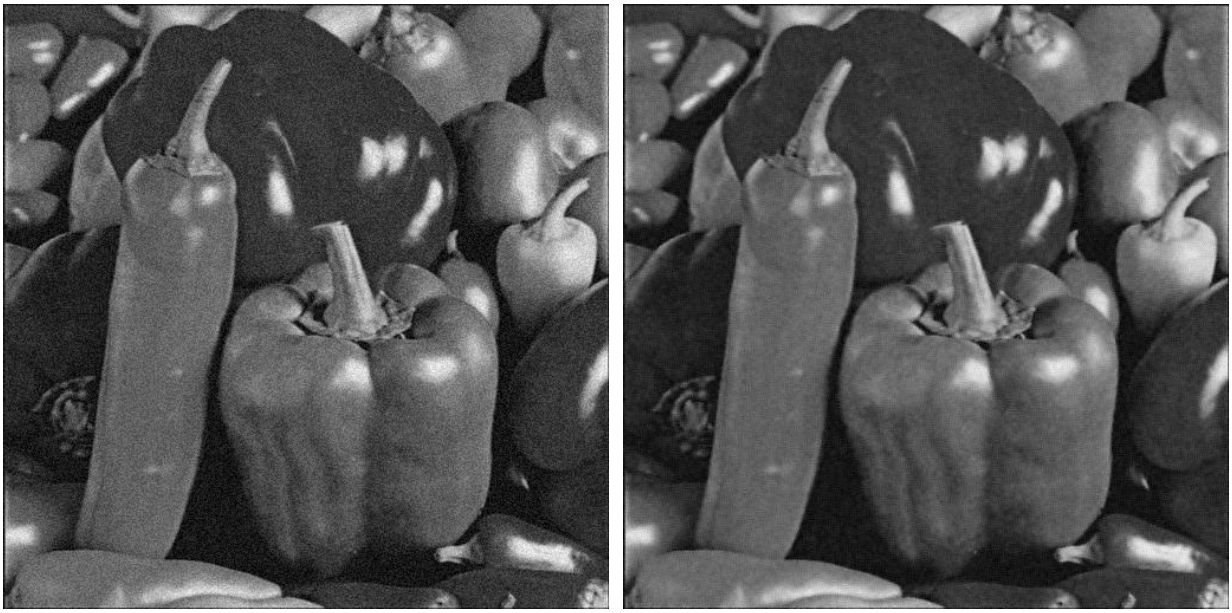

Figure 1: Left: a noisy image; Light: a denoised image.

3 Using an external library for reading/writing JPEG images

The students are requested to implement the simple denoising algorithm to handle grey-scale 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/v22/teaching-material/one_folder.zip`

Each student is strongly encouraged to download the zip file and try out the demo example under `one_folder/serial_example/`. The source code of the JPEG library can be found 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 (`.tar`) or a zip file (`.zip`). Upon unpacking/unzipping it should produce a folder named `IN3200_Oblig2_xxx` or `IN4200_Oblig2_xxx`, where `xxx` should be the candidate number of the student (can be found in StudentWeb). The folder should contain at least the following file and sub-folders:

```
README.txt    (Info about compiling/running the serial/parallel codes)
serial_code/
parallel_code/
```

There is no need to include the source code of the `simple-jpeg` external library.

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` and `convert_image_to_jpeg` can be reused from the serial implementation. The new function `iso_diffusion_denoising_parallel` needs to extend its serial counterpart with necessary MPI communication calls. Note: 1D partitioning is acceptable, and you can assume that all images are grey-scale.

6 Example of a noisy grey-scale image

`https://www.uio.no/studier/emner/matnat/ifi/IN3200/v22/teaching-material/mona_lisa_noisy.jpg`

7 The Fox cluster and its basic usage

Note: In case you already have access to a parallel computer with a working MPI installation, use of the Fox cluster is not required.

7.1 Apply for access to the Fox cluster

You need to first fill out an online application form (through "**Apply for access to a project**") on the following webpage:

`https://research.educloud.no/register`

Please choose"**ec54**" as the project, which has already been created for IN3200 and IN4200 students. In case you don't have a Norwegian electronic ID, please contact the lecturer as soon as possible.

7.2 Log in to the Fox cluster

After you are granted access (see the above step), please follow the instructions given on the following webpage:

`https://www.uio.no/english/services/it/research/platforms/edu-research/help/login-fox.html`

Please note that two-factor authentication is used by the Fox cluster, which requires some practice for novice users.

7.3 Compilation of MPI code

On the Fox cluster, it is recommened to use the OpenMPI installation and its corresponding `mpicc` compiler. To access the compiler, please remember to issue the following command as soon as you're logged in:

```
module load OpenMPI/3.1.4-GCC-8.3.0
```

7.4 Execution of a compiled MPI code

The Fox cluster uses the standard **slurm** queuing system, which requires that a job script for running a code. The following is a simple example of such a job script:

```
#!/bin/bash

#SBATCH --account=ec54
#SBATCH --job-name=simple
#SBATCH --nodes=1 --ntasks-per-node=8
#SBATCH --mem-per-cpu=2G
#SBATCH --time=0-00:05:00

set -o errexit  # Exit the script on any error
set -o nounset  # Treat any unset variables as an error
module --quiet purge  # Reset the modules to the system default

srun ./a.out
```

The above script requires that 8 MPI processes be started on one node of the Fox cluster. Each MPI process needs at most 2 GB memory and the required walltime usage is at most 5 minutes. The name of the compiled MPI program is assumed as `a.out`.

To submit the computing job that is described by the above job script, named for instance as `job.script`, the following command should be issue:

```
sbatch job.script
```

7.5 User guide for the Fox cluster

Please visit the following webpage

`https://www.uio.no/english/services/it/research/platforms/edu-research/help/hpc/docs/fox/index.md`

For example, you can find a dedicted subject named "Running jobs on Fox".