IN4200- Mandatory Project, Spring 2019

Candidate number: 15350

April 2018

Image Denoising

This project consisted of two main files: paralell_main.c and serial_main.c. All of my functions is in "functions.c" with a header "functions.h". I'am also using a external library for reading/writing JPEG images which is icluded in serial-jpeg/. Serial and parallel uses the same functions except the denoising algorithm which is calculated different in each cases. Most of the explaining of how the code work is commented in the files.

Content

1. README.txt
   - which gives info about compiling/running the serial/parallel codes
2. serial_code/
   - MAKEFILE
     - Which compiles everything into a serial_main file
   - serial_main.c
   - functions.c
     - This is used by serial_main.c and parallel_main.c.
   - functions.h
     - Which is a header for functions and have the struct of the image object.
3. parallel_code/
   - parallel_main.c
   - MAKEFILE
     - Which compiles everything into a parallel_main file
4. simple-jpeg/
   - which includes the external library

Functions

The file functions.c have 6 functions:

I **Allocate_image**(image *u, int m, int n) ($\mathcal{O}(m/my\_m)$)

- Which allocates memory for the 2D array image_data inside u, when m and n are given as input

II **Deallocate_image**(image *u)($\mathcal{O}(m/my\_m)$)

- Which frees the storage used by the 2D array image_data inside u

III **convert_jpeg_to_image**(const unsigned char* image_chars, image *u) ($\mathcal{O}(m/my\_m * n/my\_n)$)

- Which converts a 1D array of unsigned char values into an image struct

IV **convert_image_to_jpeg**(const image *u, unsigned char* image_chars) ($\mathcal{O}(m/my\_m * n/my\_n)$)

- Which does the conversion in the opposite direction

Those functions are the same for serial and parallel. But when it comes to calculating the iterations of isotropic diffusion on a noisy image object u, we need to make separated functions for serial and parallel. But they have the same purpose.

I **iso_diffusion_denoising**(image *u, image *u_bar, float kappa, int iters) ($\mathcal{O}(m + n + iters * (m-2) * (n-2)$)

- Serial: Running for all inner points, setting horizontal points and updating $\bar{u}$ with u

II **iso_diffusion_denoising_parallel**(image *u, image *u_bar, float kappa, int iters) ($\mathcal{O}(iters * ((n-2) + (n-2) + (my\_m-2) * (n-2))$)

- Parallel: Running for all inner points, setting horizontal points and updating $\bar{u}$ with u.

Serial

The **serial_main.c** reads kappa, number of iterations , input-file and output-file from the command line. And after collection data and making space, it runs iso_diffusion_denoising() to do the denoising algorithm and find $\bar{u}$, and then it will convert it back, save to the output-file and deallocate memory.

Parallel

The **parallel_main.c** uses iso_diffusion_denoising_parallel(), which is more complicated then the serial one. The iso_diffusion_denoising_parallel() starts with getting the rank and number of processors. Then it all depends on which rank we are in, if we are in the

boundaries first and last rank, it will act a bit different with send and recv, while if we are in the middle ranks, when we are in the middle ranks it will recv ,send ,send, recv in that order.(Most of the explanations is in the program). In the parallel_main.c file i have separated the out prints of each processors in different colors, so we can see which process is running, my_m and n in each cases and so on. The program starts with broadcast which sends the values to each processors before starting. The build of the program is straight forward, it start with Scattering the array into different processors, and the size of the scattering depends on my_m and number of processors. After this the program will do the iso_diffusion_denoising_parallel() function, and after it is done it will start to gather all the information from each processors into my_image_chars (whole_image) and save the new image.

Denoising of color images

When we have a colored image we will get three values per image pixel, that means that we need to do calculations with a 3D matrix, allocated and deallocate functions needs to be changed such that it can allocate and deallocate a 3D matrix, and we need to change convert_jpeg_to_image() and convert_image_to_jpeg() such that it goes from 3D to 1D and the other way. The way to do this is to make three for loops that goes true the matrix and convert it to 1D with the functions: u→image_data[x, y, z] = image_chars[x + WIDTH * (y + DEPTH * z)] and the other way to go from image to jpeg. The denoising algorithm needs to be done in 3D, it means that we have u[i][j][k] instead, and the calculation needs to be done with this. The number of components for a color JPEG image will give 3 when running the import_JPEG_file() functions, and we need to add it to export_JPEG_file() function.

Time measurements and results

To run the main programs u need to be in the folder and run make in terminal, it does all the compiling for you. Then you need to run the serial_main and parallel_main:

1. SERIAL:
    - ./serial_main kappa n_iter input-file output-file
2. PARALLEL:
    - mpirun -np [n_proc] ./parallel_main kappa n_iter input-file output-file

I have used the Abel computers for all the runs. And since it has 4 processor it doesn't give any faster calculation when using high number of processors (as shown with 8 and 20 processors). As expected the parallel program runs much faster then the serial, as we see in the table. Two processors goes double so fast then serial, and it goes faster with more and more processors, but when we run with more then number of processors the PC have, it doesn't go any faster, since it only uses a processor repeatedly. I have tested for kappa = 0.2 in each case, used different number of iterations, measured time in each case and tested for varies numbers of processors, which gives the table and the noisy grey-scale image with different iterations:

SERIAL						
iterations	100	500	1000	2000	5000	10000
seconds	1.3291	6.6073	13.1941	26.3491	65.8231	132.1252

PARALLEL					
iters/parallel	2 processor	3 processors	4 processors	8 processors	20 processors
100	0.7083	0.6020	0.6135	0.6355	
500	3.5074	3.0787	2.9902	3.3602	
1000	6.9832	5.9596	6.0761	6.6533	
2000	13.9702	11.8982	12.1665	12.7488	
5000	35.3091	29.6207	29.8395	32.5373	33.1147
10000	69.8911	59.1054	59.1580	66.7454	65.5145

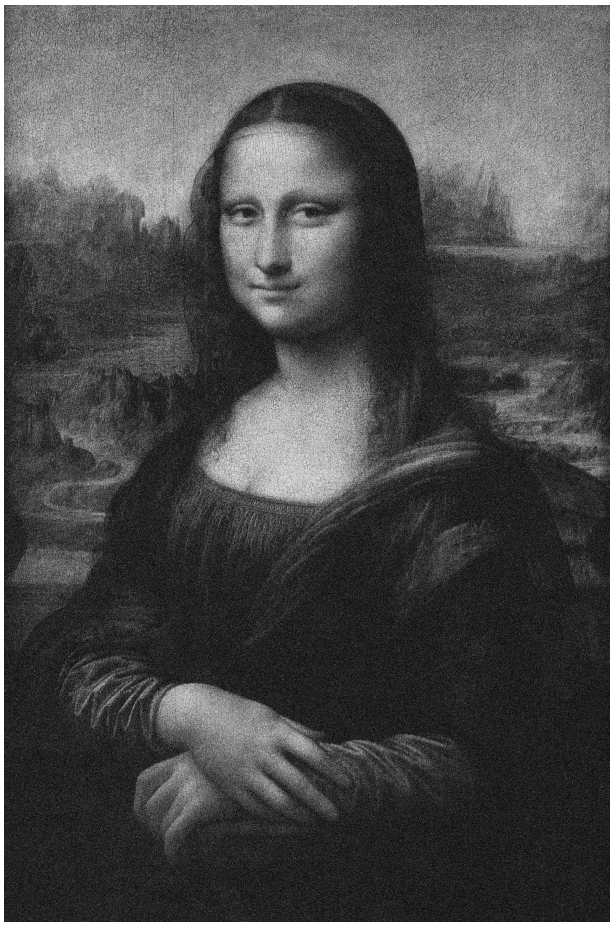


Figure 1: Original picture

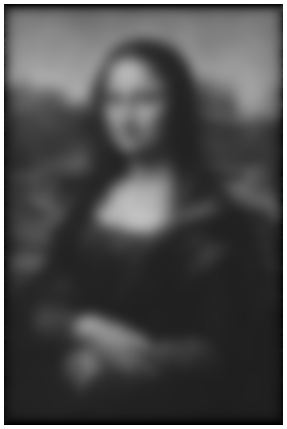
iters: 100

iters: 500

iters: 1000

iters: 2000

iters: 5000



iters: 10 000

Figure 2: After running denoising algorithm