IN3200 - PROJECT
IMAGE DENOISING

CANDIDATE NO. 15229

MAY 10, 2019

Contents

1. Introduction

The goal of this project was to implement an algorithm for denoising images. The project consists of two parts, one with a serial implementation of the algorithm and one parallelized version. For the latter version I used MPI (Message Passing Interface) to parallelize the code.

2. Program structure

2.1. **Common functions.** Both the serial and parallel version of the code shares most of the functions:

- `import_JPEG_file()` and `export_JPEG_file()`, from the `simple-jpeg`-library, which imports and exports jpg-files to and from the program.
- `allocate_image()` and `deallocate_image()`, which allocates and deallocates memory to hold the image information in a `struct` of the type `image`.
- `convert_jpeg_to_image()` and `convert_image_to_jpeg()` handles the conversion from jpeg to arrays containing the image information.

2.2. **Serial implementation.** The serial version implements the denoising algorithm in the function called `iso_diffusion_denoising()`, and timing measurement is done in the `main()` function.

2.3. **Parallell implementation.** The parallellized version uses MPI, and the denoising algorithm is contained in the function `iso_diffusion_denoising_parallel()`. In the `main()` function the image is distributed on all processes, as evenly as possible. The image is split horisontally, as shown in figure 2.1. I chose this labour division to make the communication between processes less complex. If the number of elements in the picture is not divisible on the number of processes, the remainder is distributed on all processes except the last one, which means that

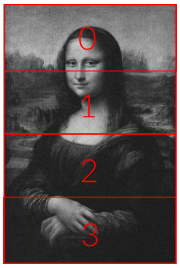

FIGURE 2.1. An example of the division of an image with 4 processes. The processes are numbered from 0 to 3.

FIGURE 3.1. Example of results of image denoising with $\kappa = 0.2$ and 200 iterations. Left: Original, noisy image. Right: Denoised image.

only one process has a smaller amount of image elements than the others, while the others have an equal amount of elements.

In the `iso_diffusion_denoising_parallel()` function, each process sends the boundaries (or boundary, in the case of the processes with lowest and highest rank), to the bordering processes, in order to compute the denoising for the complete interior of the picture. To avoid deadlock in this send/recieve-process, I have used a series of `if`-tests to make even-numbered processes start with sending, and odd-numbered processes start with recieving. After all boundaries/edges have been shared between neighbouring processes, the denoising algorithm is implented.

3. Example image

An example of a denoised image is presented in figure 3.1.

4. Time measurements

In this section I will present time measurements of the diffusion functions of both the serial and parallel code. All measurements are done with $\kappa = 0.2$ and 100 iterations. The hardware used is the following:

- Processor: Intel Core M-5Y71 CPU @ 1.20GHz × 4
- Memory: 8 GB DDR3L SDRAM @ 1600 MHz - PC3L-12800

The program was run on Ubuntu 18.04 with the C compiler `gcc`, version 7.3.0.

The time measurements for the parallelized version as a function of number of processes is shown in figure 4.1. The measurements are done only on the diffusion algorithm itself. I

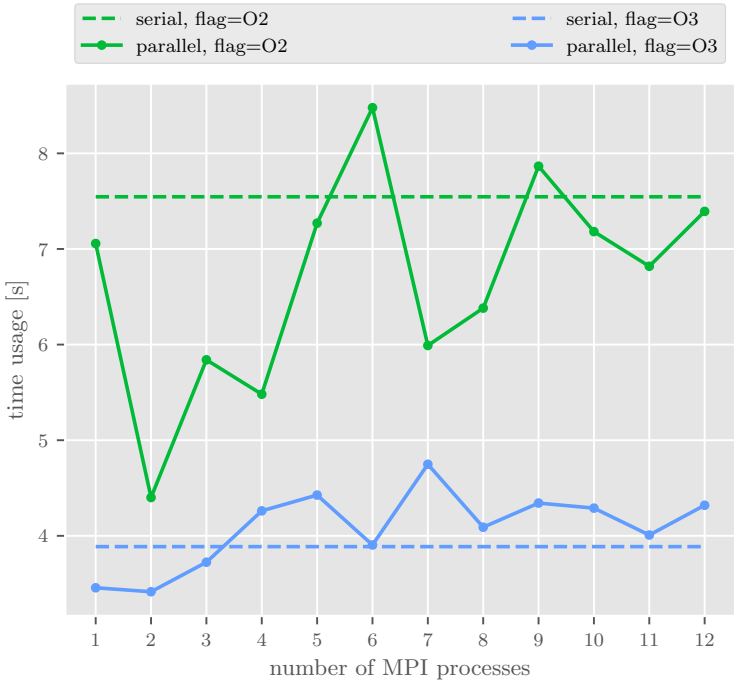

FIGURE 4.1. Time measurements of the diffusion algorithm, as a function of the numbers of processes. The measurements are averaged over 10 runs per each number of processes, with 200 iterations and $\kappa = 0.2$. Two different optimization flags were tested.

also tested the difference between the optimization flags `-O2` and `-O3`, which are used during the compilation of the program. The latter one contains even more optimization than the former, but increases the compilation time[1] As we can see, it is the `-O3` flag that gives the best performance. According to this analysis, the optimal number of MPI processes is 2. The computer I used has 2 cores with 2 threads on each, in total 4 threads. When using OpenMP, as we did in the partial exam of IN3200 earlier this spring, the best result happened when using 4 processes, one on each thread. In this project, where we use MPI, it seems not to be the case. The measurements have a fairly high degree of uncertainty, because the performance was easily affected by the heat generation of the processor.

Furthermore, we can see from figure 4.1 that the increase in performance from serial to parallel version of the code was quite small when running with the `-O3` flag. I expected a higher increase in performance, so there might be some problem with the implementation of the parallelized code, for example because of the much overhead. There may be a larger difference when dealing with larger images, when the diffusion of the images is even more demanding than the communication between processes.

[1]According to the `gcc` documentation: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.