Project report

10. mai 2019

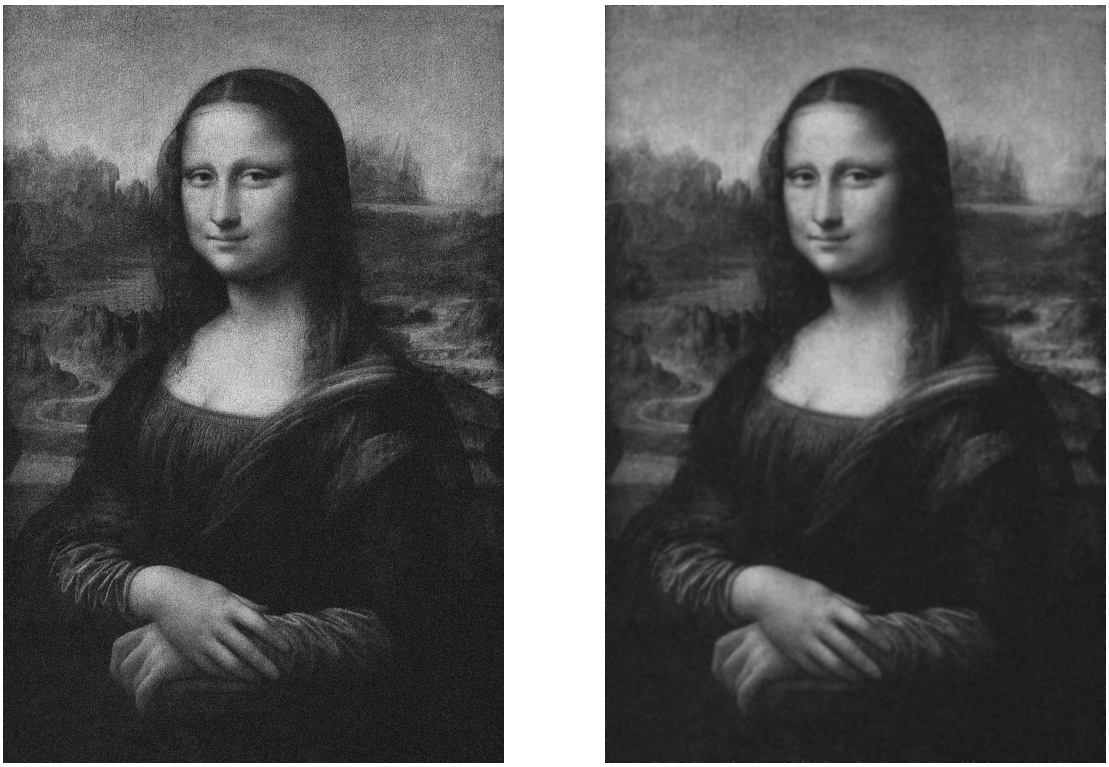
The parallel program

The parallel code is written exclusively for 4 processors. The program initializes the MPI environment, and then creates a cartesian grid of the processors, with dimensions 2×2. The program the assigns processor 0 to import the whole image, and subsequently allocate the whole_image structures. Next is the domain decomposition of the image. In case the imported photo has an odd number of pixels in any direction, the domains had to be divided in such a way that some processors receive a larger part of the image than others. This also means that the ones who have the same amount of vertical pixels have to be on the same row, and the ones who have the same amount of horizontal pixels have to be on the same column, in order to avoid future problems when sending/receiving data from grid neighbours. After the image has been divided, the u and $\overline{u}$-structs are allocated. The next step in the program is to partition and send the pixel data of the image to each processor. This task is given to processor 0. The other processors send processor 0 their domain size (my_m and my_n), allocate the appropriate size of the data to be received, and call the blocking MPI_Recv-function such that no processor can continue before all have received the necessary data. Processor 0 partions first to itself, before looping through all the processor ranks, where it receives their domain sizes, gets their cartesian coordinates, and then calculates their *start*- and *stop*-values in both vertical and horizontal direction using each processors coordinates, such that the appropriate part of the whole image can be partitioned to the correct processor. The pixel data is packed into a buffer, which is then sent to each processor. Once the data has been distributed, the one-dimensional data is converted to a matrix of size $my_m \times my_n$ using the *convert_jpeg_to_image*-function. Next, each processor enters the parallel denoising function. Here, the function *MPI_Cart_shift* is called twice, once for each dimension. This provides each processor with its corresponding upper and lower neighbours and their ranks. The processors then check what neighbours they have, and allocates arrays for receiving data from them. A loop is then entered, where the processes first share the necessary information with their neighbours. When sending data to a left or right neighbour, the data has to be packed. This is done by the function *packsendbuffer*, which takes the *image*-structure u and a buffer as input. Depending on whether the processor is sending to the right or to the left, the first or last column of their image data is packed to a buffer and then sent to their right or left neighbour. When sending rows to upper or lower neighbours, packing is not necessary, as the data as is already stored row-major. Next is calculating the denoised pixels using the isotropic diffusion algorithm. Since each processor has two neighbours, but at different locations, each processor has three loops: one for calculating the inner points of their data set (from $[1, 1]$ to $[my_m - 2, my_n - 2]$), one for calculating their rightmost/leftmost column, which requires the received data from a neighbour, and one for calculating their first or last row, which also requires received data. The last pixel, which requires neighbour data from both neighbour, is also calculated separately. The

final step inside the loop is to swap the u and $\overline{u}$-pointers. After the loop has run for a total of *iters* times, the neighbour data arrays are freed, and the function is over. The next step is to combine all the denoised pixel data into one whole image. Again, this task is given to process 0. All the other processes then have to pack their data into a buffer, such that the data can be more easily placed in the appropriate locations on the image. Process 0 first unpacks its own denoised pixel data, and then proceeds to do the same for all the other processes. The process is much the same as when partitioning the data. When the whole image has been constructed, the resulting data is converted to a one-dimensional array of type *unsigned char*, which can then be sent to the *export_JPEG_file*-function to produce a denoised JPEG output file.

Result

Examples of denoised images with the parallel code compared to the original photo are shown in figure (1). This was produced by running the program with 100 iterations, $\kappa = 0.2$, and export quality set to 75.



(a) (b)

Figur 1: Noisy image vs image denoised in parallel

Timing

For the parallel code, the wall clock time taken to run the *iso-denoising*-function will be the maximum time taken by a process. When running the program multiple times with 100 iterations of the denoising-function on the same picture, the maximum time varied between approx. 1500 and 2500 milliseconds.

Parallel example output

```
Import successful! Image dimensions: 4289 x 2835
My rank: 1 | Iso parallel time: 1482.240438ms
My rank: 2 | Iso parallel time: 1437.499285ms
My rank: 3 | Iso parallel time: 1396.185875ms
My rank: 0 | Iso parallel time: 1396.769524ms
-------------------------------------
Global time taken by iso_denoising: 1482.240438s
```