Partial Exam: IN4200

Candidate number: 15301

May 10, 2019

1 Introduction

This project explores the process parallelization of greycale image denoising using the MPI message-passing standard. The project will primerly look at how and if the parallelization speed up the denoising. The algorithm used for denoising is the simple isotropic diffiuson algorithm. The smoothing procedure look as follows for this algorithm:

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

where $u^*_{i,j}$ is the new value of the pixel at position (i, j) after one iteration.

2 Implementation

Look at the header and program files for a more descreptive explanation of the program flow and the algorithm structure.

2.1 General implentation notes

Using the `simple_jpeg` library functions `import_jpeg_file` and `export_jpeg_file` it is possible to import and export a `.jpeg` pictures as array of the primitive type `char`. The struct `image` contains a 2D dimensional array pointer of type `float` (with 1D underlaying storage).

Casting and reforming 1 dimesional character array to the 2D floting point array in a `image` struct is nesscary to do the denoising. This is because doing artihmetric is better on `float` than `char` since it's can contain decimal numbers and not only whole number.

The serial implentation of the algorithm for denoising is the `iso_diffusion_denoising` function. Here the smoothing procedure given above is done for a given number of iterations in a while loop. The borders of the pictures are ignored for this case since the smoothing procedure would need the values of outside pixels outside the picture to calculated the new value (this is also true for the parallel denoising function).

2.2 MPI structure

In the parallel program the master rank process imports and exports the character array given from the `simple_jpeg` library functions. Using `MPI_Type_vector` we can access multiple columnns in the 2D image array or stride-wise for the 1D character array. The master rank can send out block-wise parts of the image using this so that the process can process the denoising in a cartesian block structure. It can also block-wise recieve and correctly place parts of the 2D image array using `MPI_Type_vector`.

In this project the `MPI_Comm` used is of a cartesian structure so to easier send to the neighbors the buffers needed to update the borders to each proccess. The blocks of the picture fro each procces are in the shape $(M/P_m) \times (N/P_n)$. Here the $M \times N$ are picture dimension and P_m and P_n number of process in each direction (total number of proccesses $P = P_m \cdot P_n$). The reminders are evenly divide out among the diffirent blocks.

In the `iso_diffusion_denoising_parallel` for each iteration in addition to doing the same as in the serial case, each process send it's borders as buffers to it's cartesian neighbors (possibly up, down, left, right), so that they can update their borders using this information.

3 Speed results

The program was test on a system with AMD Ryzen 1700 @ 3.8 GHz with 8 cores/16 threads as the CPU, and DDR4 RAM with speed set to 2666 GT/s. The programs are optimized with the -O2 flag in gcc/mpicc. For 100 iterations with kappa = 0.2 with the noisy Mona Lisa picture given as a example with the project, the serial and parallel denosing was timed to:

processes [number]	serial	1	2	3	4	6	8
time [sec]	1.17	1.10	0.61	0.51	0.49	0.49	0.49

From this we see that there is a significant speedup in the implementation uptill 4 cores afterwards the speed is constant. This maybe due the the overhead of the communication between the proccesses eating up all the would be speed increase, and the speed being then being primerally dependent on the interconnecting speed and delay between the proccesses.

4 RGB considerations

If we were to rewirte the program to denoising of RGB-pictures, we would need the array int the `image` struct to be a 3D array with 3 in height. To deliver and recieving the image to each proccess could be done the same way as done in this project just three times as is for each layer in height. For the 1D character array that means moving the start pointer for sending, along as it would a underlaying array for a 3D array.

In the `iso_diffusion_denoising` for each iteration one would need to do it layer by layers (assuming the denoising is color-independent). If not we could update the isotropic diffusion eqution to:

$$u^*_{i,j,k} = (1-6\kappa)u_{i,j,k}+\kappa\left(u_{i-1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i+1,j,k} + u_{i,j,k-1} + u_{i,j,k+1}\right). \tag{2}$$

The it would be nessecary to send slabs between each proccess (6, 1 for each direction) as buffers for updating the procedure between each iteration.