Second Assignment FHPC

Gianluca Guglielmo

January 12, 2021

1 Code Implementation

The c code developed for the elaboration of the image has two different versions: an MPI one and an OMP one. They share almost the same header file with some slight differences that account for the way the execution is carried. For the sake of simplicity, I will first describe the base of the program and then specify the differences in the two versions.

1.1 Generalities

The header file contains the implementations of the functions that will be used in the program.

The functions used to create the different kernels are three, they all take as input the height m, the width n and the pointer to the kernel, which is allocated previously in the main. m doesn't have to be equal to n as I have generalized for non-square matrices, but they both have to be odd.

The first is meankernel, which doesn't take other variables as input. The implementation is straightforward: the double variable det is the inverse of the area of the kernel. All the values of kernel are set equal to that.

The function weightkernel takes as additional inputs param, the value of the central point, and symm, a control parameter to eventually move param to another point in the kernel and create asymmetry through the use of the function center. The implementation of this function is trivial: using two nested for loops every index of the kernel is accessed and set to 1-param, then the central value is switched back to param.

The function gaussiankernel takes the same inputs as weightkernel, but param represents the standard deviation. Its implementation is divided in two parts. The first part elaborates a non-normalized gaussian centered in x_center, y_center. The determinant is calculated beforehand, then using two nested for loops we first elaborate the displacement from the center, then explicitly calculate the value of the gaussian in that `(x,y)` point. The second part simply accounts for the normalization of the kernel, since its borders do not extend to infinity, hence using it would cause a loss of brightness in the image.

The function `center` takes as input the pointers of `x_center` and `y_center` and asks the user to input their new values.

The function `read_pgm_image` takes as input a pointer to the void array that will contain the image `void **image` , a pointer to the file name `const char *image_name` and the pointers `int *maxval`, `int *xsize` and `int *ysize`, which will contain the values for the image dimensions and the size per pixel. The file is first opened, and the first line is scanned in order to get the relevant information about it. The comments are skipped by scanning for lines that start with `#`. Finally, the array dedicated to the image is allocated, but first we check if `maxval` is greater than 255. If so, the `(char *)malloc` allocation is double in size, in order to represent an `unsigned short int`. If there are memory problems the file is closed, while if there are I/O problems the pointer is freed and the function returns either `maxval` equal to -2 or -3.

The function `write_pgm_image` takes the same inputs as the previous function, with the difference that the name is the file that the image will be written back on. The image is written using `fwrite` from `stdio.h` and accounting for the dimension of every pixel given by `maxval`.

The function `elaborate` takes different inputs for different versions. In general it receives `void *ptr` to the image, its dimensions, `maxval` and two indexes `start` and `end` needed to select the area of the image to elaborate. Through four nested loops, each point is selected and elaborated through the convolution of the kernel and its borders. In order to block the border effect and its loss in luminosity, a check is carried on at each run of the inner loop: if the indexes are outside the borders of the image then `count` is accumulated with the values of the kernel that exceed and subsequently the pixel is normalized with `1-value`.

1.2 MPI Version

The MPI version needs some careful planning beforehand. First, we initialize the MPI execution environment with `MPI_Init`. Then, for each core we get the total number of cores `ranks` and the specific core's identity `i_rank`. Core `0` will be treated as root. First, based on the user's selection, each core creates the designed kernel. The image is then read on root on the array defined by the allocation of the pointer `void *ptr`, along with the values of `maxval`, `xsize` and `ysize`. These values are then passed via `MPI_Bcast` to the slave cores. The image will be split vertically and each core accounts for a portion, but in order to elaborate it they need a slightly bigger section of the one they will write on. This is due to the simple fact that each pixel is elaborated by looking at its neighbours, so the ones close to the border need information that would not be contained by a simple cut of the image. To solve this problem the following parameters are introduced: `start`, `end`, `first`, `last` and `flo`. `flo` stores the result of `floor(ysize/ranks)`, which is the same for all cores. This value represents the most homogeneous value to cut vertically an image when it's impossible to create equally sized chunks. Each core will have to elaborate a section whose dimensions are `xsize*flo`, while the last core will simply receive what's left of the image. `start` and `end` are local to each core, and represent the height boundaries of the portion each core will have to write on. `start` is equal to `i_rank * flo`, `end` is equal to `(i_rank + 1)*flo` for every core but for the last, where it's equal to `ysize`. `first` and `last` account represent the height boundaries of the portion each core will need values from. `first` is equal to `start - (m-1)/2` and `last` is equal to `end + (m-1)/2`, where `m` is the height of the kernel. In order not to exceed the borders of the image, `first`'s lower bound is set to `0` and `last`'s upper bound is set to `ysize`. The peculiarity of these values lies in the fact that they will be used multiple times during the execution of the program. They are also generated and stored in arrays located on the root node, whose names simply have `2` at the end. Three arrays `bigpic`, `sendcount` and `displs` are initialized. Using two nested for loops, `bigpic` is filled. Its values are taken directly from the image in ranges `(first2[i_rank]*xsize, end2[i_rank]*xsize)`. This way, through `MPI_Scatterv`, each core receives exactly the section it needs plus eventual out-of-border information without the need to scan the image each time. The received portion is then passed to the function `elaborate`, along with `first` and `last`, which are needed to check if the kernel is acting outside the image. The image is then reaggregated on root using `MPI_Gatherv`.

1.3 OMP Version

The OMP version is simpler than the MPI one due to not needing to previously subdivide the image. The number of threads is set by calling `set_num_threads`. The only difference between this and the serial version is in how the loops in `elaborate` are managed. The image is read, then sent to `elaborate`. Inside the function the directive `#pragma omp parallel` is called and before the first for loop the directive `#pragma omp for` is placed. `#pragma omp for private(val, count, j, ii, jj)` privatizes some variables needed by each thread.

2 Scalability & Performance Model

The performance model developed for the MPI implementation is the following:

$$\lambda = 1.35 \times 10^{-6} s$$
$$b = 12Gb/s$$
$$t_1 = \lambda + (2 \times xsize \times ysize) \times \frac{log(p-1)}{p}$$
$$t_2 = t_{comp} \times m \times n \times \frac{(2 \times xsize \times ysize)}{p} \sim \frac{29}{p} s$$
$$t_3 = 2 \times t_{read} + t_{MPIcall} \sim 0.7s$$

The following is the performance model for the OMP version:

$$t_1 = (2 \times xsize \times ysize) \times \frac{log(p-1)}{p}$$
$$t_2 = t_{comp} \times m \times n \times \frac{(2 \times xsize \times ysize)}{p} \sim \frac{29}{p} s$$
$$t_3 = 2_{read} \sim 0.7s$$

The image used for the analysis is "earth-notsolarge.pgm" with the average kernel of size 11×11. Since t_{comp}, t_{read} and $t_{MPIcall}$ weren't known they were measured empirically with some trials and approximate results were used. The following plot compares both the empirical and theoretical outputs of MPI and OMP in a log scale:

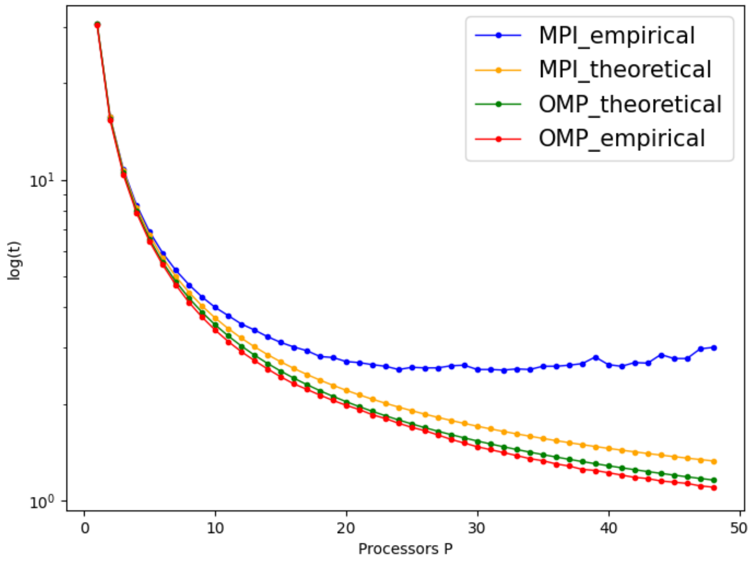

Figure 1: `N = `10^2`, max S(P) at P = 1`

The OMP empirical output sticks closely to the theory, while the MPI does so just for a small number of processors. The theoretical model didn't forecast that the times would start to grow as quickly as $p = 20$.