Report on Assignment 1 HPC

Sebastiano Zagatti

20 January 2021

1 Serial Blurring

In order to implement the blurring of a pgm image in both MPI and OpenMP, a serial version of the blurring code is needed; this will constitute the basis from which the two codes will be developed.
In this code the functions write_pgm_image, read_pgm_image and swap_pgm_image, which were provided for the assignment, have been used; moreover, three more functions have been added: Average_kernel, Weight_kernel and GaussianKernel, of course these three functions produce the three different kernels required by the assignment.
The first and the last functions take as inputs the sizes of the kernel (ksizex and ksizey) since they are uniquely defined by them, while the second one takes as an input also the central wight (w). Notice that ksizex and ksizey can have two different values, as long as they are both odd and greater than 1, meaning that a square kernel is not mandatory.
The three functions will all return a pointer referring to where the kernel matrix is stored (the matrix is actually stored as a vector whose values are those of the matrix read by rows).
Regarding the main of the code, the program will take the following 5 inputs (in order): name_of_the_image (without the extension .pgm), type_of_the_kernel (a number in $[0, 1, 2]$ according to the assignment's convention), x_size_of_the_kernel, y_size_of_the_kernel and, in the case of a weighted kernel, a sixth input: central_weight.
The first part of the code is dedicated to handling the input values and defining the output according to the requests of the assignment. Moreover, all the variables needed for the convolution are defined and the image is loaded in the part of the memory pointed by the pointer M by using the read_pgm_image function; another pointer N is also defined and the memory necessary for storing the output image is allocated and initialized (buffer image approach). At this point, according to the type of kernel requested, the corresponding function is called and the kernel is loaded.
After that, the computation of the convolution between the matrix and the kernel is implemented; this is achieved by four nested for loops: the first and the second loops (namely the ones on i and j) identify the element of the image matrix on which the kernel is centered (basically the element that we want to modify) and run on the two dimensions of the image matrix. The third and

fourth for loops are used to identify the elements of the image that will be needed for the computation of the convolution. The values of those elements are multiplied by the corresponding kernel elements (the kernel weights) and then summed up in order to define the new value of the element on which the kernel is centered:

$$N_{i,j} = \sum_{u=-s_x}^{u=s_x} \sum_{v=-s_y}^{v=s_y} M_{i+u,j+v} K_{u,v}$$

where s_x and s_y are the integer halves of ksizex and ksizey.
Notice that the an if statement is used to check whether the elements we are considering for the convolution are inside the image or not: in case of a negative response the faulty elements are not considered in the calculation and the kernel is re-normalized accordingly; in this way the border effect has been handled.
Finally, the image is written back using the write_pgm_image function.

2 OpenMP Blurring

The OpenMp implementation of the code presents some differences with the serial one: to allow the strong and weak scalability measures with a simple bash script, one more variable was introduced, namely the number of threads that need to be spawned, in order to avoid continuously changing the environmental variable $OMP_NUM_THREADS.
In order to satisfy the assignment's request, more handling of the input variables was needed, since four different ways to call the function are now possible:

- **4 inputs**: name of the image, mean or gaussian kernels (0 or 1), x and y sizes of the kernel; in this way the function is called using the syntax requested by the assignment and the maximum number of threads available is used.
- **5 inputs**: name of the image, weighted kernel, x and y sizes of the kernel, central weight; again the assignment's syntax is used.
- **5 inputs**: name of the image, mean or gaussian kernels (0 or 1), x and y sizes of the kernel and the number of threads used.
- **6am inputs**: name of the image, weighted kernel, x and y sizes of the kernel, central weight and number of threads.

Once this issue is solved, the OpenMP implementation is pretty straightforward: a parallel region is created using #pragma omp parallel with a number of threads set according to the previous casuistry; then the work of the first loop is divided among the threads using the work-sharing construct #pragma omp for.
Parallelizing only one loop is the best choice, since the spawning of threads introduces an overhead that would result in an overall decrease in the performance of the code in the case of nested parallelization.

3 MPI Blurring

The main idea behind the MPI implementation is to divide the image in a number of rectangular sub-matrices according to the number of processes available: each sub-matrix will be assigned to one of the processes, which will compute its convolution with the kernel. Of course, in order to have a correct convolution, some elements of the confining sub-matrices are needed, so this issue needs to be dealt with.
A possible issue could arise in case the dimensions of the matrix are not perfectly divisible by the number of processes available, the implemented solution is to let the master process take care of those elements.
The domain decomposition of the problem is achieved by creating a new communicator with a 2D cartesian grid topology; the master process will then rearrange the matrix in order to have an array where the elements of each sub-matrix are contiguous; moreover, it will also compute the final dimensions of the sub-matrices and the dimensions needed to compute the convolution; these values are sent to each process using MPI_Bcast. Furthermore, the master process will also send each sub-matrix (namely each sub vector obtained by re-arranging) to the other processes by means of an MPI_Scatter.
Each process will then check whether, depending on its position in the topology, there are any processes on its left, on its right, on top of it or under it: this is achieved by using MPI_Cart$_s$hift, and, depending on the result of this check, the process will send the parts of its sub-matrix needed for the convolution to the nearby processes. Each processor then proceeds to compute the values of the convoluted sub-matrix.
Finally, the master process gathers all the sub-matrices from the other processors by means of a MPI_Gather and joins them in order to form the resulting convoluted matrix.

4 Scalability Study

A scalability study of the two codes has been carried out on a thin node on the Orfeo cluster. For the strong scalability the image earth − notsolarge.pgm of size 10000×10000 pixels has been used for both codes, the results are displayed in the attached file strongscalability.png where both the 11×11 and 31×31 weighted kernels have been used, with a central weight value of 0.2.
The absolute values the plots are referred to, namely the values for the computation on 1 core/thread, are 72.37 seconds for the 11×11 kernel and 590.15 seconds for the 31×31 kernel using the OpenMP versions. The reference values for the MPI versions are 69.61 seconds for the 11×11 kernel and 555.23 for the 31×31 kernel.
The results with the 11×11 kernel are worst than the ones with the 31×31, this is somehow expected from Gustafson's and Amdahl's laws, since the size of the kernel contributes to the increase in the size of the problem.
To measure the weak scalability, the workload of the problem needs to be in-

creased in accordance with the number of processes/threads used, if the number of threads is doubled, the size of the image needs to be doubled, meaning that double the total number of pixels is needed. In order to do so a simple C program has been written using the functions provided in the materials for the assign, being i the number of processes/threads used, the size of the square image needed to compute the weak scalability is given by:

$$x_{size,new} = \sqrt{i}\, x_{size}$$

The starting image that has been considered is 6500×6500 pixels and the others are generated accordingly to be computed by the number of processors of a thin node, namely from 1 to 24, so the image processed by 24 cores will have a size of 31843×31843 pixels.
The results of the weak scalability are displayed in the attached file weakscalability.png, for both 11×11 and 31×31 weighted kernels with a central weight value of 0.2.
The absolute values the plots are referred to are 30.58 seconds for the 11×11 kernel and 249.54 for the 31×31 kernel using the OpenMP versios. For the MPI versions the values are 29.45 seconds for the 11×11 kernel and 234.48 seconds for the 31×31 kernel.
As in the previous case the results for the 31×31 kernel are better, since by increasing the size of the kernel we are increasing the size of the parallel fraction of the program.