Foundations of HPC

@ Data Science and Scientific Computing 2020-2021

ASSIGNMENT II, Dec 11

Due date: Jan, 8th, 2021 h23:59

In this assignment you are requested to implement the solution to the proposed problem both a MPI and a OpenMP code.* (so, we do expect 2 codes).

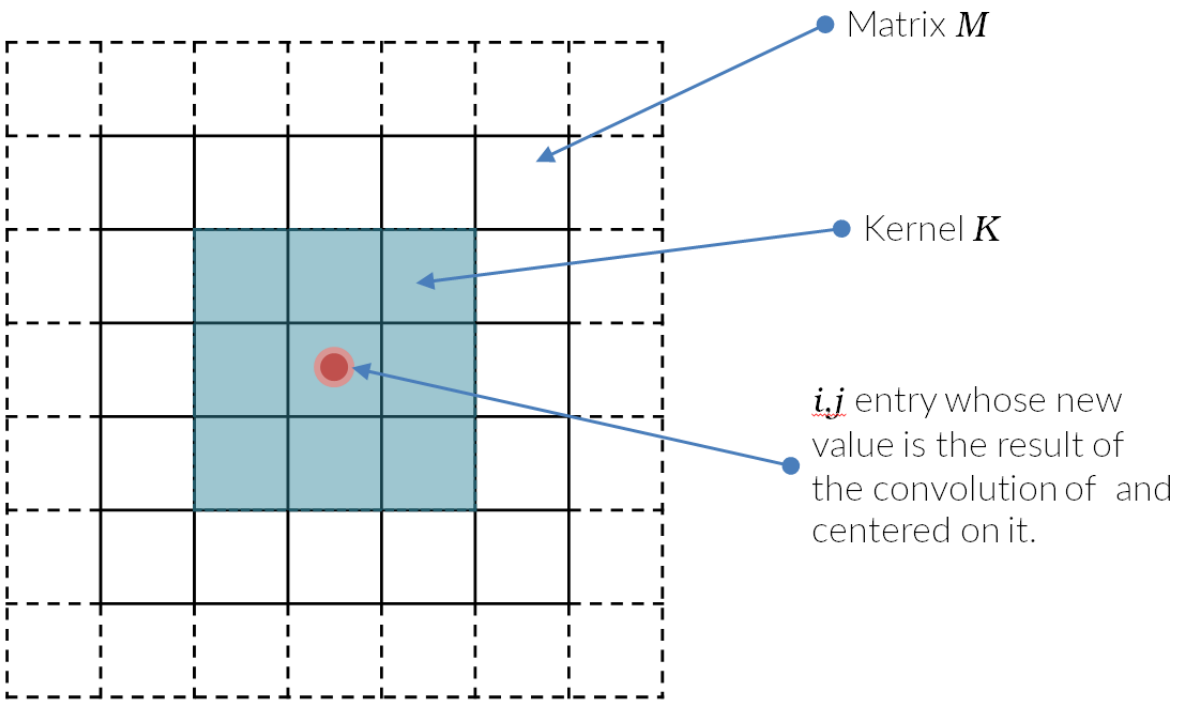
>> If you're willing, you may write only 1 hybrid MPI+OpenMP code. Some simple advices are offered in Appendix.

BLURRING of an IMAGE

In this exercise you are requested to write a code that can blur an "image", i.e. a 2D matrix

- Introduction to blurring -

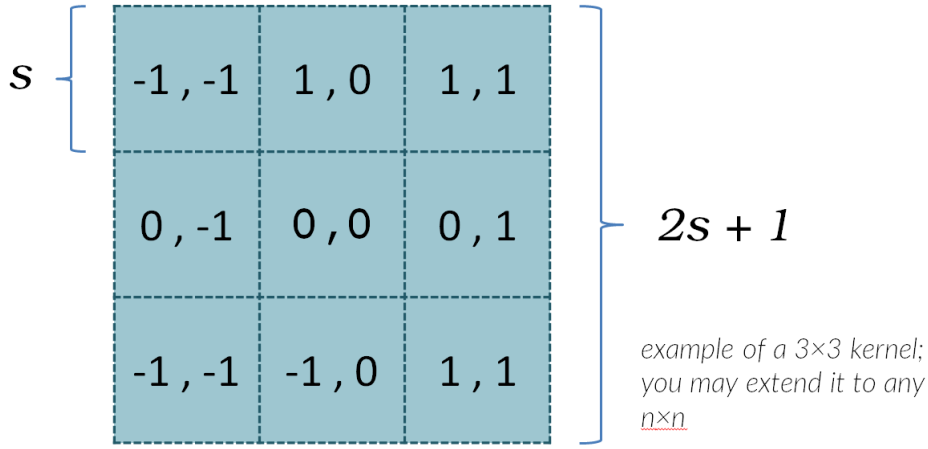
Blurring is a very common operation in many fields; although it is commonly associated to a post-processing step in digital photography the concept is more general and applies in whatever number of dimensions.
In non-math terms, we may define the *blurring* as the "smoothing" of a given quantity over a neighbourhood. More precisely, the *blurring* is the result of a running convolution of a matrix M with another smaller matrix K, often called "kernel", over each entry of the matrix M:



so that each entry $M_{i,j}$ (let's focus on 2D for the sake of simplicity, since the requirement of the exercise is in 2D) becomes

$$M_{i,j} = \sum_{u=-s,v=-s}^{u=s,v=s} M_{i+u,j+v} \times K_{u,v} \tag{1}$$

where s is the integer half-size of the Kernel K (consider kernel matrices that are always square and of odd order; if k is its size and $s = [k/2]$ is its integer half-size, in this text we index its entries as in the above figure from $-s$ to s)



The requirements for the Kernel K are then the following:

- K is a square matrix (*feel free to generalise to non-square kernels*)
- K's order is always odd;
- K is spherically symmetric respect to the entry $K_{0,0}$ (*feel free to generalise*)
- K is normalised to 1, i.e. when integrated over the volume of its support:

  $$\int_V K(\vec{x}) d\vec{x} = 1 \tag{2}$$

  which in our case translates to

  $$\sum_{u,v} K_{u,v} = 1 \tag{3}$$

  Having a normalised Kernel leads to the *conservation* of the quantity you're smoothing.

There are obviously different Kernel types:

Mean Kernel

Aka Average Filter or Box Filter
Using this filter amounts to equally average the pixels (the Matrix entries) around the central one and to replace it with the result. If the Kernel's size is k, the Kernel matrix K is then:

$$K = \frac{1}{k^2} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix} \tag{4}$$

I.e., in the smaller case $k = 3$

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{5}$$

Weight Kernel

That is a generalisation of the previous one, in that the weights assigned to each pixel are no more equal. A typical example is the common centrally-weighted Kernel in which the entry $(0, 0)$ holds a significant fraction f of the total value and the rest $1 - f$ is equally divided among the remaining entries of K. Using a centrally-weighted Kernel means that each pixel is dominating the new value assigned to it after the convolution.
For the sake of clarity, an example of centrally-weighted 3×3 Kernel is

$$K = \begin{bmatrix} w & w & w \\ w & f & w \\ w & w & w \end{bmatrix} \tag{6}$$

where $0 < f \leq 1$ and $w = (1 - f)/(k^2 - 1) = (1 - f)/8$ (for instance, $f = 0.52, w = 0.06$).
Obviously $f = 1$ means no blurring at all, which may be a good hint for a test of your code correctness.

Gaussian Kernel

That is a special case of the previous case, in which the weights of each entries are assigned by using a Gaussian function (in 2D in this case):

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{7}$$

where σ is the "half-size" equivalent of the half-size s of the Kernel K. In practice since the support of G is formally infinite, which would be unpractical for our purposes, you fill the K entries with approximated values, like in the following classical example:

$$K = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{8}$$

(remember to always normalise the Kernel to 1).

- Requirements -

Then, in this exercise:

1. Consider to have a "large" input matrix M of size $m \times n$ that we will call the "image". The origin of M is left to you. Take into account that for the purpose of code verification a `PGM` image will be given as input (see the Appendix for the details about this very simple format), so at least your code must be able to read in, and write out, a `PGM` file.
2. Consider to use integer values as entries of your image. You may also consider to use grey-scale images or, in other words, that to represent your pixels you need a single integer.
   The precision required is of 2 bytes, i.e. a `short int` type.
3. Consider to use a square Kernel K of general odd size k (and half-size $s = [k/2]$).
   The minimum size is obviously 3, and you may consider that $k \ll m$ and $k \ll n$.
4. Your code must be able to use different Kernels, i.e. not only the Kernel's dimension must be a free parameter but also the Kernel type must not be hard-coded. See here for a small discussion about Kernel types.
5. You can ignore the border effect, i.e. you are free not to process properly the border pixels for which your Kernel get out of the images' boundaries (which results in a small clipping in luminosity, i.e. on a "vignette" effect).

Although the requirements are as stated above, feel free to generalise: you may (i) represent M as a floating-point matrix and translate it to the integer representation, (ii) use non-square and non-symmetric Kernels, (iii) treat properly the borders, (iv) use coloured images (i.e. you need a triplet of values for each pixels in case of `RGB` representation or different ones in different colour-spaces).

Stefano: diciamo che se generalizzano potremmo riconoscere +1 punto sul voto finale o no?

- TO-DO lists -

From this assignment you are request to produce:

1. 2 source codes, one using MPI and the other using OpenMP, that perform the blurring of an image as detailed above. Before the Xmas break you'll be given a test case that will validate your result. It will consist in an `PGM` image that will be our input and in the same `PGM` image blurred used 2 different kernels (the details will come along with the images).
2. A scalability study of your code, both weak and strong.
3. A short report (possibly max. ~ 4 pages) about your code (basically how you implemented the blurring) and its scalability.
4. A performance model of your code.

Appendix I

Reading/Writing a `PGM` image

The `PGM` image format, companion of the `PBM` and `PPM` formats, is a quite simple and portable one. It consists in a small header, written in ASCII, and in the pixels that compose the image written all one after the others as integer values. A pixel's value in `PGM` corresponds to the grey level of the pixel, in `PPM` it is a triplet of integers for `RGB` channels.
Even if also the pixels can be written in ASCII format, we encourage the usage of a binary format.

The header is a string that can be formatted like the following:

```
printf( "%2s\n%d %d\n%d\n", magic, width, height, maximum_value );
```

where `magic` is a magic number that is "`P4`" and "`P5`" for `PGM` and `PPM` respectively, `width` and `heigth` are the dimensions of the image in pixels, and `maximum_value` is a value smaller than `65535`.
If `maximum_value < 256`, then 1 byte is sufficient to represent all the possible values and each pixel will be stored as 1 byte. Instead, if `256 <= maximum_value < 65535`, 2 bytes are needed to represent each pixel.

In the sample file `write_pgm_image.c` that you find the `Assignment2` folder, there is the function `write_pgm_image()` that you can use to write such a file once you have the matrix `M`.

In the same file, there is a sample code that generate a square image and write it using the `write_pgm_image()` function.
It generates a vertical gradient of N_x x N_y pixels, where N_x and N_y are parameters. Whether the image is made by single-byte or 2-bytes pixels is decided by the maximum colour, which is also a parameter.
The usage of the code is as follows

```
cc -o write_pgm_image write_pgm_image.c
./write_pgm_image [ max_val] [ width height]
```

as output you will find the image `image.pgm` which should be easily rendered by any decent visualizer .

Once you have calculated the matrix M, to give it as an input to the function `write_pgm_image()` should definitely be straightforward.

Appendix II

A note about hybrid MPI+OpenMP

Although we did not yet discuss this topic in the class, at the level you may use it here that is quite straightforward. As you have seen, it is obviously not a requirement but just an opportunity for those among you that like to be challenged.

As long as you use OpenMP regions in a MPI process for computation *only* and *not* to execute MPI calls, everything is basically safe and you can proceed as usual with both MPI and OpenMP calls and constructs.

It may be safer, however, to initialize the MPI library with a call slightly different than `MPI_Init()`:

```

int mpi_provided_threaD_level;
MPI_Init_threads( &argc, &argv, MPI_THREAD_FUNNELED,
&mpi_provided_thread_level);
if ( mpi_provided_thread_level < MPI_THREAD_FUNNELED ) {
    printf("a problem arise when asking for MPI_THREAD_FUNNELED level\n");
    MPI_Finalize();
    exit( 1 );
}

...;   // here you go on with BaU

MPI_Finalize();
return 0;
```