Martina Danese

ASSIGNMENT 2

Foundations of High Performance Computing

Master in Data Science and Scientific Computing

2020/2021

For this Assignment, I developed an image-blurring algorithm, implementing both an OpenMP- and an MPI-based parallel code.

1. OpenMP code

When running on *nths* threads, the OpenMP code divides the overall work on the input image into *nths* sub-images of work according to a bi-dimensional grid, and assigns to each thread the blurring of one of these sub-images. The grid is obtained maximising at the same time the number of divisions in x (*nthsx*) and y direction (*nthsy*), in order to keep the splits on one axis as close as possible to the other. Each thread (*thid*) is then associated to a grid position (*xxth*, *yyth*) according to:

```
yyth[thid] ⟶ floor(thid/nthsx)
xxth[thid] ⟶ thid%nthsx
```

When possible, the number of pixels of the original image in x (or y) direction, namely *xsize* (*ysize*), is evenly divided among the *nthsx* columns (*nthsy* rows), leading threads to have same amounts of pixels *xpxl*[*thid*] (*ypxl*[*thid*]). When the number of threads does not allow for a perfect division of the number of pixels of the original image, the first *xsize*%*nthsx* (*ysize*%*nthsy*) sub-images are allocated with a extra pixel along that axis, namely adding 1 to *xpxl*[*thid*] (*ypxl*[*thid*]) as shown in Figure 1.

Thus, the thread numbered *thid* is mapped the corresponding starting index (*start_idx*) of the original image as:

$$\texttt{start\_idx[thid]} \longrightarrow \sum_{i=0}^{thid} \texttt{xpxl[i]} + \sum_{i=0}^{thid} \texttt{ypxl[i]} \texttt{ * xsize}$$

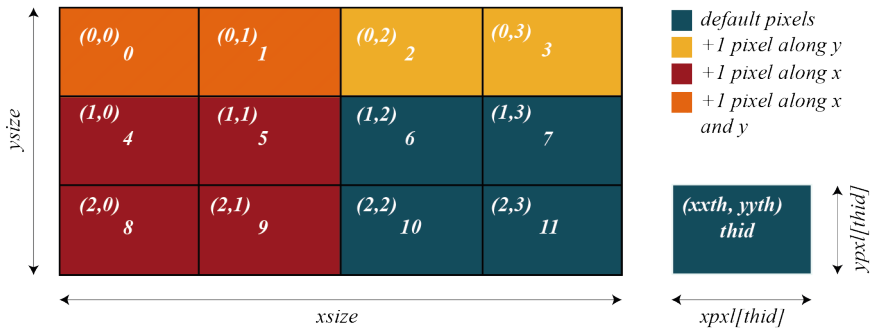


Figure 1: Policy for image division adopted both in MPI and OpenMP. With OpenMP notation: by default sub-images are initialised as `xpxl[thid] = floor(xsize/nthsx)` and analogously in y (blue cells), namely rounding them down. If this division originates a reminder, an extra pixel is added in that direction to the first threads on that column or row. For example, with a reminder of 1 in y direction an extra pixel to the default *ypxl* value is assigned to threads of the first row (yellow), while with a reminder of 2 in x, 1 is added to *xpxl* value for the sub-images of the first two columns (red). In MPI code a communicator is created for all the possible *xpxl* and *ypxl* combinations, so for the case in this picture four MPI communicators are required.

For the blurring, a parallel region is opened setting a close thread affinity policy. Every thread loops over all the pixels of its sub-image, iterating first in y and then x direction with two `for` loops. Inside, the blurring value of the single pixel is computed. This is achieved with two other nested loops running from $-khalfsize$ to $+khalfsize$ (the kernel integer half-size), and multiplying the pixel and its surrounding elements with the corresponding element of the kernel. Remaining inside the y and x loops, these results are collected and summed to find the final value of the blurred pixel.

Note that in the current version of the program the parallel region is opened merely for the blurring, and all the parameters of the sub-images are created beforehand and stored in shared arrays. Before this, I attempted several more complex versions, for example (i) I started the parallel region at the very beginning of the code and privately defined sub-images parameters herein, and/or (ii) I wrote threads results in a parallel region, always taking care properly of the synchronisation by means of single, critical or atomic regions in order to avoid thread races. Despite all the attempts done to optimise single and critical regions, all of these versions performed similarly or slightly worse than the simple one that I am presenting here. So I decided to rely on the KIS(S) principle.

I tested my algorithm by means of a subtraction of the resulting images with those provided. I estimated that this difference is always remarkably smaller than the required one, and I believe images are identical to those provided but for integer-double approximations.

2. MPI code

The idea behind MPI implementation is the same discussed for the OpenMP code. First, the formation of a bi-dimensional Cartesian splitting relies on MPI domain decomposition routines, which allowed also for the creation of a communicator, named *grid_communicator*. Then, *xpxl* and *ypxl* values are assigned as seen previously, also for cases in which threads do not allow for an exact division of $xsize$ or $ysize$. The image is opened and read once, letting each segment to be read exactly by the processor that will blur it. This approach presents the advantage of opening and reading the image just once, and avoiding communications among threads.

At this point, halo layers are sent among processors with the blocking functions `Send` and `Recv`. Even though halo layers are not modified, I chose blocking functions because (i) non-blocking functions were observed to wrongly exchange data, and (ii) as seen during classes and tested in Assignment 1, for small messages such as halo layers are, system buffers are used, allowing these small messages to perform as non-blocking. Vertical and horizontal nearest neighbours are directly identified with `MPI_Cart_shift`, while the diagonal next nearest neighbours are retrieved with `MPI_Cart_rank` function. Because the opening of numerous channels one after the other would lead to high latency, new `MPI_Datatypes` were used for sending the required data as a unique block. This is easily achieved assigning (i) the amount of pixels that must be exchanged in each row of the sending thread as the block value, (ii) *xpxl* of the sending thread as stride and (iii) the amount of lines that must be exchanged as counts.

Finally each process computes the blurring analogously to the previous case, but checking whether it is required to use any halo layer on the borders. Resulting data are gathered with `MPI_Gatherv` function and again creating MPI datatype so that the master processor can directly store then into the correct

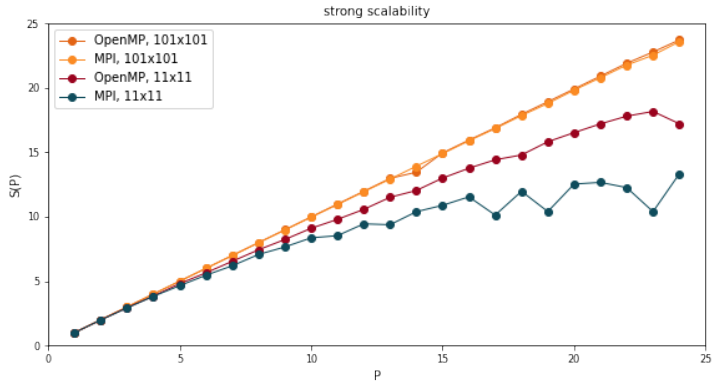


Figure 2: Resulting speed-up for the strong scalability. Tests were done by means of *earth-large.pgm* image with kernel sides of 11 and 101 pixels.

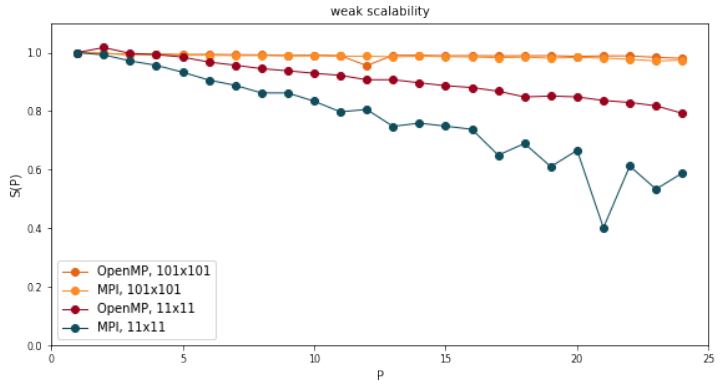


Figure 3: Resulting speed-up for the weak scalability. Tests were done by means of a 4000×4000 pixels image with kernel sides of 11 and 101 pixels.

position. Moreover, in case threads present different amounts of *xpxl* and/or *ypxl*, separate MPI communicators are created isolating threads with different dimensions (e.g. for the case presented in Figure 1, four Datatypes are required). This allows the master to correctly gather the data, and to avoid a line-by-line communication which would have required repeated openings and consequential increase in latency.

3. Scalability

In order to test the performances of my codes, I concluded a weak and a strong scalability with two kernel sizes.

The strong scalability is based on the blurring of the *earth-large.pgm* image, using a kernel with side value of $ksize = 11$ and 101 both for MPI and OpenMP codes. For $ksize = 11$ results rationalised in terms of speed-up (Figure 2) report a detachment from the ideal trend with both for OpenMP and MPI code, the latter having a more severe deterioration, but still remaining above the 50% of the ideal speed-up. I believe that this discrepancy is due to the extra communications that MPI requires. On the contrary, $ksize = 101$ results in almost perfect trends for both the codes.

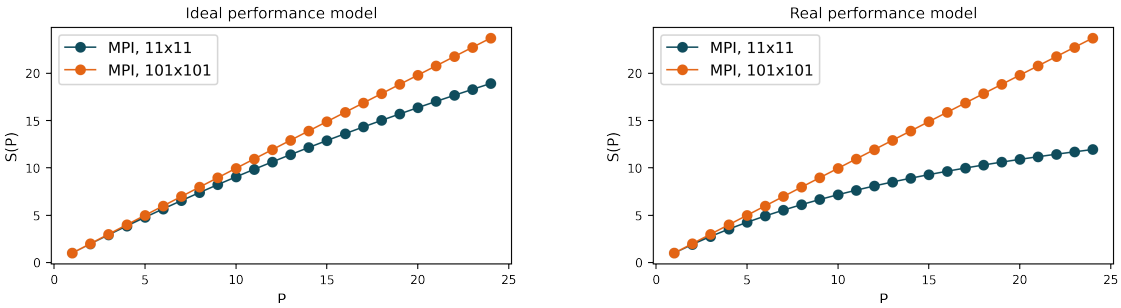
For the weak scalability (Figure 3), each processor was assigned to an equal amount of work, namely the blurring of a 4000×4000 pixels image created on-the-fly while running the scalability script. Results appear in agreement with those seen for the strong scalability, being the trends for $ksize = 101$ indiscernible from the ideals, while those of $ksize = 11$ worse, but still acceptable.

4. Comparison with a Performance Model

Scalability results were then compared to speed-ups obtained with a performance model, i.e. providing an estimation of the total computational time considering for the MPI code:

1. $T_{serial} = T_{read} + T_{writing}$. These values were manually identified for *earth-large.pgm* as $T_{read} \sim 0.3\,s$ and $T_{write} \sim 1\,s$.

2. $T_{communication} = T_{halo} + 3T_{allg} + T_{gatherv}$, where T_{halo} is the time required to communicate the halo layers, namely:
$$T_{halo} = \frac{(s + xpxl) \cdot k \cdot dir \cdot USI_{size}}{2 \cdot BW} + k \cdot T_{\lambda}\ ,$$
(or changing *xpxl* in *ypxl* according to the direction) where BW is the Orfeo within-node bandwidth, s is the kernel half-size, k is dimensionality of the problem, i.e. 2, dir the number of directions of message exchange, USI_{size} the size in bytes of an unsigned short integer, and T_{λ} Orfeo latency. Note that since the message is sent in two directions, the bandwidth is assumed to be double, so in the denominator BW is multiplied by a factor of 2. $T_{gatherv}$ is the final time required by the master to



(a) **(b)**

Figure 4

collect the slaves results. It can be expressed as[1]:

$$T_{gatherv} = \log(P) \cdot T_\lambda + \frac{P-1}{P} \frac{(xsize \cdot ysize) USI_{size}}{BW} \ ,$$

where P is the number of processors. Analogously, T_{allg} is the communication time required at the beginning by the master to collect the sub-domain dimensions ($xpxl$ and $ypxl$) and the starting index of the other processors:

$$T_{allg} = \log(P) \cdot T_\lambda + \frac{P-1}{P} \frac{USI_{size}}{BW} \ .$$

3. The computational time: $T_{comput} = T_{f.o.}(2 \cdot s^2 + 1)(xpxl \cdot ypxl) USI_{size}$, where $T_{f.o.}$ is the time of a floating point operation, estimated as $10^{-9}\ s$.

The model for the OpenMP code is analogous, but with the exclusion of the `MPI_Gatherv` and halo layers contributions.

The resulting speed-up (reported in Figure 4a for the MPI code for *earth-large.pgm*) is observed to correctly estimate an almost ideal trend for the 101×101 kernel, but presents optimistic results for a 11×11 kernel. I believe this is mainly due to (i) the overhead associated with the creation of MPI datatypes and communicators, which are not taken into account in the model, (ii) the assumption of doubling BW when messages are exchanged is ideal and probably is not true, and (iii) an overestimation of BW which was considered having its peak value, $12\,GB/s$. To take into account the latter effect, the BW was tuned until recovering results compatible with those in Figure 2 (resulting plots are reported in Figure 4b). This required a decreasing of an order of magnitude, which is probably unrealistic, but compensates also for the points (i) and (ii).

[1] Doerfler, Douglas, and Ron Brightwell. "Measuring MPI send and receive overhead and application availability in high performance network interfaces." European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. Springer, Berlin, Heidelberg, 2006.