Report on Assignment 2
Foundations of HPC

Giulio Dondi

January 20, 2021

Contents

Section 1 - Blurring Algorithm

General consideratons

This assignment involved the creation of a parallel algorithm implementing a convolution blurring of a PGM image with a certain kernel. A built-in functionality allows the automatic creation of a symmetric kernel of specific size $M \times N$ (non-square kernels are supported) of Average (type 0), Weighted(type 1) and Gaussian(type 3) type. Alternatively, a kernel may be supplied from a file with the command line option *-kernel-file*, the file should specify the two dimensions NM followed by the individual entries line-by-line. The kernel is normalised within the program.

The convolution algorithm scans the image pixel-by-pixel and, for every pixel, does a scan of the kernel matrix and the corresponding image pixels, shiftend from the current working pixel. To ensure that the algorithm never steps outside the boundaries of the image, for every pixel four variables `offs_l, offs_r, offs_u, offs_d` are calculated, relative to the four directions. Each indicates the number of pixels between the current pixel and either the edge of the kernel or the edge of the image, whichever is closer. `offs_l, offs_u` are defined negative.
These variables determine the boundaries of the convolution scan per pixel: if we use kernel coordinates, where the current pixel is at (0,0) the scan will encompass columns between `offs_l` and `offs_r` and rows between `offs_u` and `offs_d`. Naturally the loop counters are translated in global image and kernel indices respectively to access the right elements.
Various versions of the blurring algorithm have been implemented for preliminary performance evaluation: different methods for buffering the blurred pixels (explained below) and ×2, ×4, ×8 unrolling of the convolution loops. They may be selected using compilation flags, no flag supplied selects ×4 unrolling.

The vignette effect has been addressed in this implementation. The effect arises close to the edges, when less than 100% of the kernel is used for the convolution. As the kernel is normalised as a whole, using fewer elements entails that the overall brightness of the blurred pixel is not preserved. The result is a dark halo around the outer edge of the image. This is easily addressed by re-computing the kernel normalisation constant whenever needed and normalising the blurred pixels, but this adds a great deal of computations and memory accesses.
Since the only object needed to compute these constants is really the kernel itself, it is possible to pre-compute them at kernel initialisation and de-code the proper value tduring the convolution process. Take for example a $3times$ kernel overlaid on the top left pixel of the image: the entire top and left edges of the kernel lie outside of the image by 1 pixel. Using the above definition of the `offs_` variables, their values are in this case: `offs_l=0, offs_u=0, offs_r=1,offs_d=1` . Summing the left, right and the up,down variables defines the position (+1,+1) in our kernel coordinates, i.e. below and to the right of the central pixel. Thus, we define a normalisation

matrix of the same size as the kernel itself and, for every possible combination of offset variables, combine their values to decode the position of the right value within.
With this method, normalising a pixel requires one memory access and one float operation, of course we store the constant so that this operation is a multiplication. This procedure is only needed close to the edges, on a relatively small subset of poxels. Testing determined that boundary checks are preferable to performing un-necessary renormalisations of the central pixels.

The blurred pixels need to be stored in a buffer, as the original corresponding pixels need to be kept around for the blurring of subsequent pixels. This implementation reduces the memory footprint by using a buffer smaller than the whole image. In order for a pixel to be definitely not required anymore, it needs to lie a number of rows and columns away from the current working pixel equal to half th kernel size (including the central kernel pixel). So for a square kernel of size $(2h - 1)$ and a $M \times N$ the buffer is $h \times N$. Each row of pixels is stored in the buffer as it's computed, when the last buffer line is filled in the first one can be safely written in the image and so on. At the end of the image scan the buffer is full of lines still left to be written: an additional scan of the buffer is performed, starting from the line following the one last computed, wrapping around the end until every line has been written.

Parallel implementations

The above algorithm has been parallelised separately in MPI and OpenMP. Both verisons follow a similar principle of operation: all the processes/threads are arranged on a grid and each is assigned a pair of grid coordinates, the image is sub-divided accordingly among the processes who then execute a modified version of the above algorithm on their own subimage. The blurred image is then re-assembled at the end of the procedure. We will first discuss features common to both versions.

A struct called `cell` contains all the data defining the subimage: its dimensions, the global image coordinates of its first pixel, the coordinates on the process/thread grid, its overall size in memory. The sub-images are defined wider and taller in order to include halo layers: the kernel will in fact move outside of the subimage when close to its edges, but still lie within the overall image. It is therefore necessary to include this strip of extra pixels which surround the subimage in order to achieve the correct result.
Information about the width of the halos in each direction is calculated together with the subimage cells and stored within the `cell` struct. Halos are usually $(h - 1)$ wide, where h is the half-width of the kernel. Extra checks are performed, since if an edge cell is narrower than this value then the halo of the next cell over will overlap it completely and step beyond the edge of the image itself.
Handling halos requires minor modifications to the blurring algorithm described above since the halo pixels should not be processed. Firstly, the boundaries of the pixel scan are modified based on the presence or absence of halo layers, and similar modifications are made to the buffer updating/writing code. However, a special situation may possibly arise that requires special handling: when cells are shorter in height than the buffer itself then the pixel scan will never fill it up completely, thus the buffer lines are only written at the end of the loop in this case.

The MPI implementation first creates the grid and initialises a Cartesian communicator. The master process is responsible for reading the image header and broadcasting the metadata to all the processes. Every process possesses the command line arguments and is therefore capable of initialising its own copy of the kernel. Each process then claculates the cell parameters and prepares its own local subimage. To import the data, the collective `MPI_File_read` routine is used, each process creates a subarray datatype to automatically fit the right data in the local image, including halos of course. The edian-swap process is done in parallel on the local image portions. After blurring, the same is done in reverse to write the image back to file using `MPI_File_write`. Since only the data without halos should be written, two sub-array datatypes are used: one that describes the internal pixels within the local halo buffer and one that describes the same set of pixels but in the context of the global image. Finally, the timing data collected is reduced on the master process to compute the per-process average of time measurements.

In the OpenMP implementation, the image data is allocated by the master process once the header has been read, but the memory is not touched until later. The parallel region is spawned and each thread initialises its own cell and creates a local copy of the kernel. Once the areas of responsibility are known, every thread touches its portion of image by writing zeros at the right indices: when the image file is read, each memory segment lies in the memory location closest to the thread that touched it. Endian swap is once again done by each thread on their own data. These image portions do not, however, include halos at this stage: a second sub-image buffer is initialised per thread, and the right data including halos is copied at the right places within. The halo data needs to be fetched elsewhere in the system but this is only done in this one occasion.

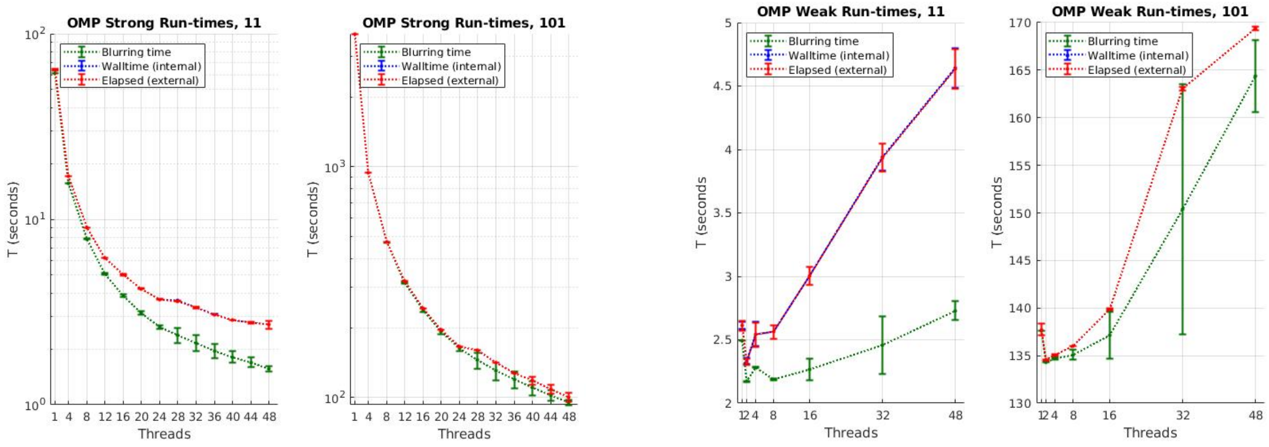
Once the blurring is complete, the internal subimage pixels need to be copied from the local buffer to the complete image buffer, and the right location is exactly the one that lies close by. This data is, however, interleaved with halo data in the local buffer and not contiguous: the data is then copied line-by-line so that the halo segments can be skipped. This is an explicit implementation of what the MPI subrray types are capable of accomplishing automatically. Timing data is collected and gathered using `omp reduce`, but this directive is enabled only if overall timing functionalities are enabled, otherwise a simple parallel region is spawned.

Section 2 - Benchmarks

Scalability tests were performed for both the MPi and OMP implementation. Strong scalability tests used a 21600 × 21600 reference image (with 2 bytes per pixel) and number of threads/processors varying between 1 and 48 (corresponding to 2 sockets, 16 cores per socket with x2 hyperthreading CPUs on a *gpu* node on the Orfeo cluster) with increment of 4.

Weak scalability tests instead used images generated from scratch with randon moise (again 2 bytes per pixel), starting from a baseline size of 4096 × 4096 and increasing both dimensions by $\sqrt{p}$ to produce an image p times the baseline size. Threads and processors still varied between 1 and 48 and only the powers of 2 values were considered.

Regarding affinity, for all OMP tests the affinity was set as `OMP_PLACES=cores, OMP_PROC_BIND=close`, while for the MPI runs the setting `--map-by core` was used.



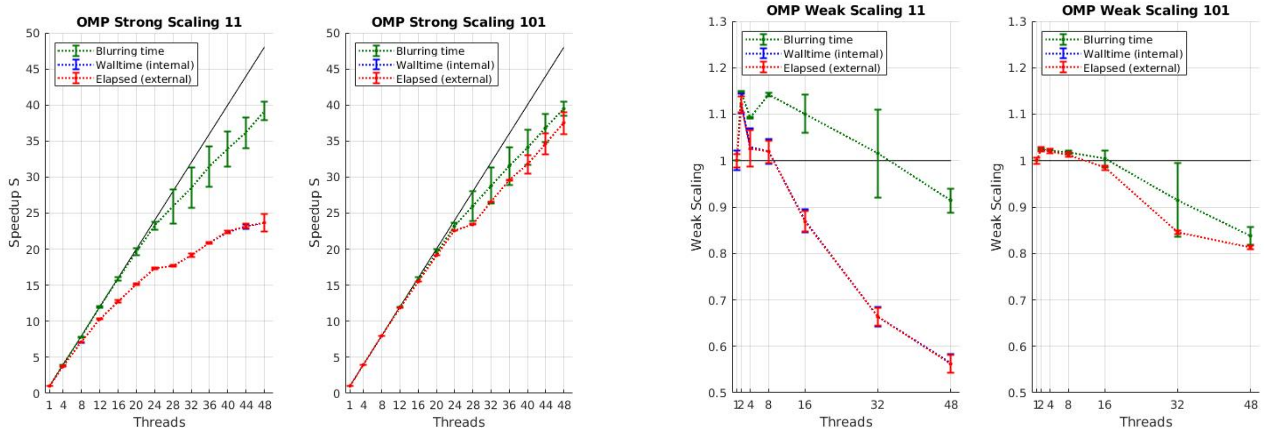
(a) OMP Strong scaling run-times, on the left for a 11 × 11 kernel, on the right for a 101 × 101 kernel.

(b) OMP Weak scaling scaling run-times, on the left for a 11×11 kernel, on the right for a 101 × 101 kernel.

Figure 1

Figure 6 reports the absolute run-times for the OMP algorithm in the strong and weak scalability tests, while fig. 2 reporta the scalability curves computed against the $p = 1$ run. The scaling of the blurring portion alone is compared against the total time scaling, measured both internally and externally. For strong scaling, the blurring time scales similarly for the two kernel sizes as the independent ariable is the image cell size without halos, which itself depends on the number of threads and not on the kernel size. The total time scales differently between the two kernel sizes due to the importance of the non-blurring operations performed: the blurring workload is roughly $w_{img} \times h_{img} \times w_{ker} \times h_{ker}$[1] which entails a workload $(101/11)^2 \approx 84$ times larger, and thus the non-parallelised parts of the OMP algorithm such as I/O impact much more. Requesting more than 24 OMP threads is expected to impact performance as hyperthreading

[1] The accurate estimate of the operations at the edges of the images reduces this estimate to ∼ 99.7% of this value.



(a) OMP Strong scalability plots, on the left for a 11 × 11 kernel, on the right for a 101 × 101 kernel.

(b) OMP Weak scaling scalability plots, on the left for a 11 × 11 kernel, on the right for a 101 × 101 kernel.

Figure 2

engages, as seen by a dip in the strong scaling plots.
The run-time of the OMP algorithm can be modelled as a function of the image dimensions w, h, kernel dimensions kw, kh and the number of workers p, the run-time is composed of the following pieces:

- time to read and write the image header and initialise the kernel: $2header\_t + ker\_init\_t$
- time to read and write the image to disk: $2\left(disk\_lat + (2h \times w)/disk\_bw\right)$
- time for each thread to touch its memory segment with zeros: $(mem\_lat + (2w \times h/p) \times mem\_op)$
- time to copy the sub-image with halos into each thread's local buffer: $2\left(mem\_lat + (2w_{blockhalo} \times h_{blockhalo}) \times mem\_op\right)$ where $w_{blockhalo} = (w + k\,(g_w - 1))\,/g_w$ is the average width of a haloed sub-image block[2], and $h_{blockhalo}$ is analogously defined.
- time to copy the sub-image without halos back to the global image buffer after blurring: $2\left(mem\_lat + (2w \times h/p) \times mem\_op\right)$
- time to perform the endian swap: $(w \times h/p) \times endian\_swap\_op \times ht\_fac$
- time to perform the blurring: $(w \times h \times kw \times kh/p) \times blur\_op \times ht\_fac$

The dominant factors in this model are the blurring time and the disk I/O time, in the $k = 11$ case the ratio between these varies between $[55.9; 1.44]$ as p increases, while in the $k = 101$ the ratio is $[2084.8; 72.5]$. The parameters of this model are reported in table 1 , they were either extracted from runs of the serialised version of this algorithm or from dedicated benchmarking. For the endian swap and the blurring portions, average times per operations were extracted from serial data, two different operation times are used for two different kernel sizes to reflect the different impact of the buffer allocations, conditional branchings and loop overheads on top of the convolution operations.

header_t	0.044 s	mem_lat	0.035 s
ker_init_t (11)	0 s	mem_op	0.5e-09 s/byte
ker_init_t (101)	0.03 s	disk_lat	1e-4 s
endian_swap_op	2.325e-10 s	disk_bw	2.5e+09 bytes/s
blur_op (11)	7.4e-10 s	ht_fac	0.005
blur_op (101)	4.4e-10 s		

Table 1: Empirical model parameters

[2] g_w is the horizontal size of the processors grid, while g_h would be the vertical size.

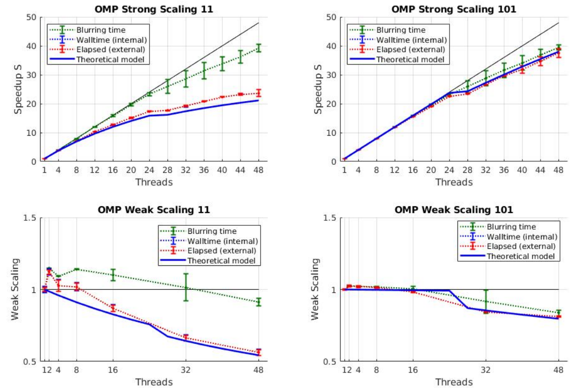
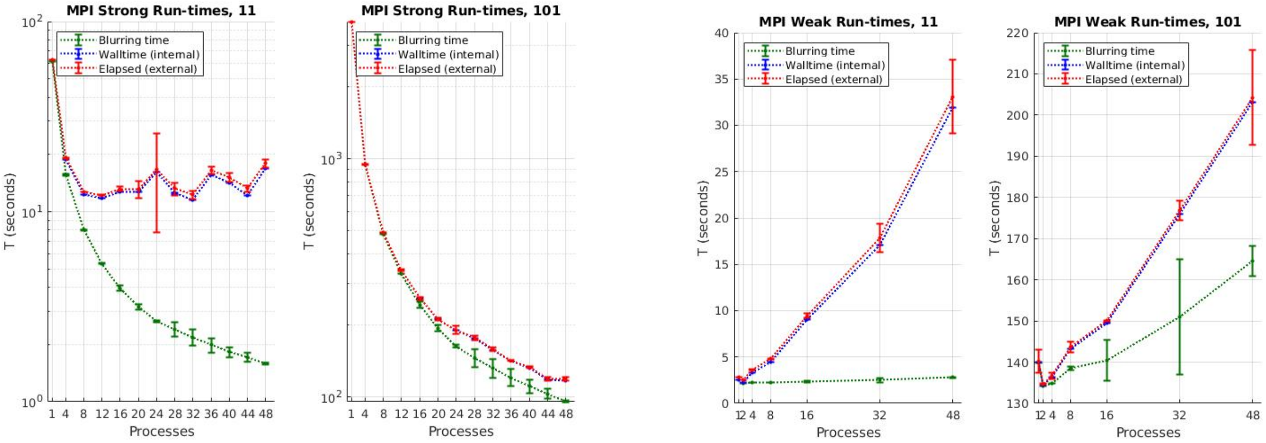


Figure 3: OMP comparison between theoretical model and scaling plots.

Figure 3 displays the results of this simple model against the scaling plots, showing that the main trends are captured, although the $k = 11$ case overperforms our model while the $k = 101$ underperforms it , especially in the weak case.

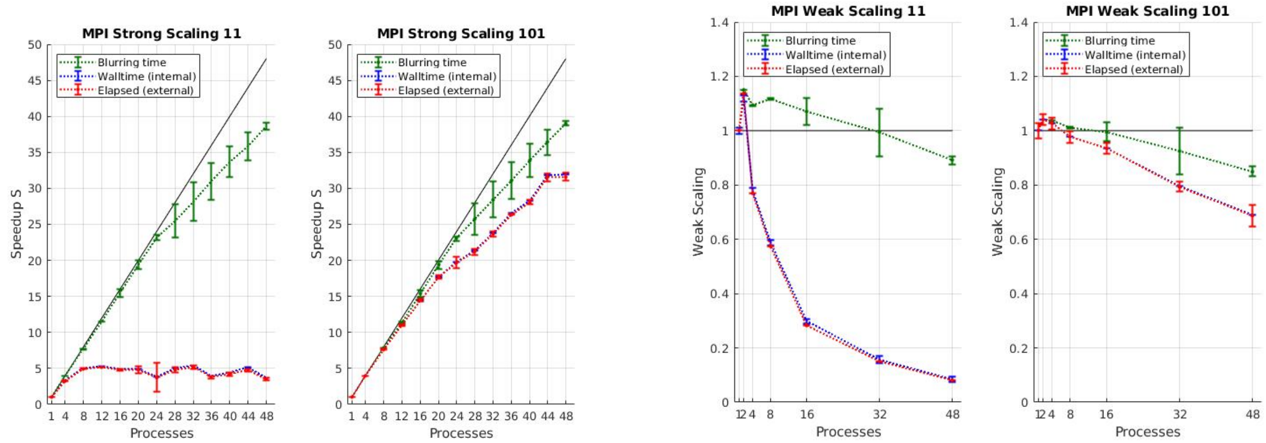
Turning now to the MPI algorithm results, figure 4 reports the absolute run-times, while scalability curves computed similarly to the previous case are reported in figure 5.



(a) MPI Strong scaling run-times, on the left for a 11×11 kernel, on the right for a 101×101 kernel.

(b) MPI Weak scaling run-times, on the left for a 11×11 kernel, on the right for a 101×101 kernel.

Figure 4



(a) MPI Strong scalability plots, on the left for a 11×11 kernel, on the right for a 101×101 kernel.

(b) MPI Weak scalability plots, on the left for a 11×11 kernel, on the right for a 101×101 kernel.

Figure 5

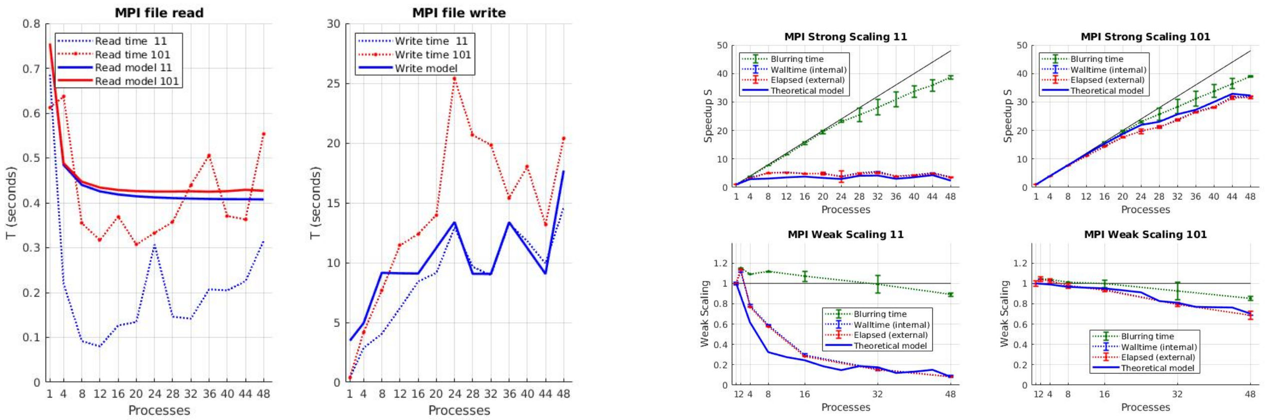
The scaling results are overall worse compared to the OMP algorithm: while OMP in the $k = 101$ achieved a ×36

speedup, MPI achieves about ×32 For the $k = 11$ runs, OMP achieved a ×23 speedup while MPI does not deliver more than ×5 improvement. The scaling of the blurring phase alone is very similar for the two algorithms, which is expected as each process operates an identical algorithm on identically-sized subimages. The reason for the worse performance must then be traced to the use of MPI-IO operations to read and write the image.
Figure **??** shows the average run-times per process of the `MPI_file_read` and `MPI_file_write`, showing that the reading part does scale with increasing processes, while the writing is negatively impacted by adding processors and adds an overhead ∼ 10 times higher compared to read.

A similar theoretical model was built for the MPI algorithm, using the same parameters of table 1 and the following building blocks:

- time to read and write the image header and initialise the kernel: $2header\_t + ker\_init\_t$
- time ro read the image using `MPI_file_read`, modelled as every process reading their entire subarray data block one after another sequentially: $(disk\_lat + 2w_{blockhalo} \times h_{blockhalo}/disk\_bw) \times p$
- time ro write the image using `MPI_file_write`, modelled as every process writing one single line of their own data buffer sequentially, until all lines have been written [3]: $(disk\_lat + 2\,(w\ /g_w)\,/disk\_bw) \times g_w \times h$
- time to perform the endian swap: $w \times h \times endian\_swap\_op/p$
- time to perform the blurring: $w \times h \times kw \times kh \times blur\_op/p$

The IO operations were modelled under the assumption that the filesystem on the Orfeo cluster does not support parallel operations. The IO model is shown against the data in figure 6a; the read model shows predictions for both the $k = 11$ and $k = 101$ cases as the quantities $w_{blockhalo}, h_{blockhalo}$ depend on the halo size, the general trend of the curves is reflected in the model but the predicted difference between the two buffer sizes is much smaller than the actual data.
The write model only shows one curve and agreement is somewhat good: the assumption of sequential block-line writing means an overhead given by the disk latency parameter for every block-line written which, itself, depends on how many distinct blocks there are on the horizontal dimension. This is the reason for the peaks and throughs in the curve, corresponding to particular arrangements of the processor grid as p varies. The difference between write performance for the two kernel sizes is puzzling, as the subimages written are deprived of halos and thus their size does not depend in the halo size. There may be an overhead in fetching the subimage data within the haloed buffer, which could explain part of this difference.



(a) Average times per process to read the image sub-array (left) and write it back(right).
Theoretical models have been overlaid.

(b) Weak scaling scaling run-times, omn the left for a 11 × 11 kernel, on the right for a 101 × 101 kernel.

Figure 6

[3] Once again, g_w is the horizontal size of the processors grid.

The dominant factors in the overall model are once again the blurring time and I/O time, although this time for the $k = 101$ case the ration between these two varies between $[1211.7; 5]$ as p increases while in the $k = 11$ case the I/O time the two factors lie within abut one order of magnitude of each other, and for high values of p the I/O time actually dominates.
Image 6b comares the overall model with the scaling data and shows good agreement., although the $k = 11$ predictions are outperformed by the data for $p < 20$