Assignment 2

Andrea Zasa

01/11/2021

ASSIGNMENT 2

During this assignment I will code a mixed MPI and openmp code for blurring an image. The idea is to divide the big image in sub-images and distribute them over the nodes using MPI routines. Once distributed the sub-images will be processed and the workload divided using an openmp for cycle. In the end the blurred sub-images will be recollected by the master node and the final result will be saved.

Section 1: The code implementation

The program start by initializing the MPI communicator and create a virtual two dimensional cartesian topology that will later be used for dividing the image. Then there is a rapid input check that look for the basic parameters that will be used in the code. The master node then load the image in and scatter the image informations, such as height and width, over the MPI processes. At this point dimensions for the sub-images are calculated and memory for them is allocated on each node; the main image can now be scattered. Now we get to the actual blurring.

Blurring is done by convoluting a matrix M, in this case the image, with a smaller one K called the kernel. The kernel can be chosen arbitrarily to obtain different results. In my code kernel can be selected as an input and chose among 3 common ones:

- Mean kernel: as every entry $K_{i,j} = \frac{1}{K_w}$ with K_w the number of entries of the kernel. This result in pixel the average of what surrounds them.
- Weight kernel: similar to the mean one but the central point has a fixed weight defined as an input and the other entries are $K_{i,j} = \frac{1-C_w}{K_w-1}$ (C_w is the central weight).
- Radial kernel: Is an approximation of a normal distribution in 2 dimension entries are distributed such as $K_{i,j} \sim e^{\frac{i^2+j^2}{2K_s}}$. In this case i and j represent the position relative to the center of the kernel and K_s is half size of the kernel.

The first step before the blurring is then to initialize the right kernel. To avoid latency in the calculation is for the best every MPI process has its own kernel. The sub-images are then blurred dividing the pixels calculations over the available openmp threads. Note there are two functions for performing the blur; one for the "internal" pixels and one for the borders that deals with this special cases and ensure there is no "border effects" that would make the image darker. The result of the blur is saved in a separate memory region as an in place algorithm would be way more difficult to implement and memory is not the primary concern in this project. In the end the master process recover the sub-images via a MPI_Gatherv instruction; saving the result in the local folder with the appropriate name.

I also tried to implement an alternative idea that, instead of scattering the totality of the image over the processes, would had sent only the relevant pixels, but I wasn't able to make this idea work due to problems with halo exchanges. The way the command MPI_scatterv divide the buffer is in fact dependent on the rank of the process and not on his position and this create problems when exchanging halo layers between neighbors. Sending each sub-image individually could solve this problem but I suspect it would greatly intensify the time consumption.

Section 2: Strong scalability

To have an idea of the performance of this code a strong scalability study was performed. The image used for the test was a large, 21600 x 21600, pgm image. The test was conducted on the orfeo cluster and using only one node up to 48 threads (N.B hiperthreading enabled). Due to the code using both MPI and openmp and the fact the test was conducted on a single node, for performing the test I had to cap the max number of openmp threads (2 for this scalability test) or else the program would always use all the cores available. Different runs were performed, changing the kernel type and its dimension, here is a summary of the results:

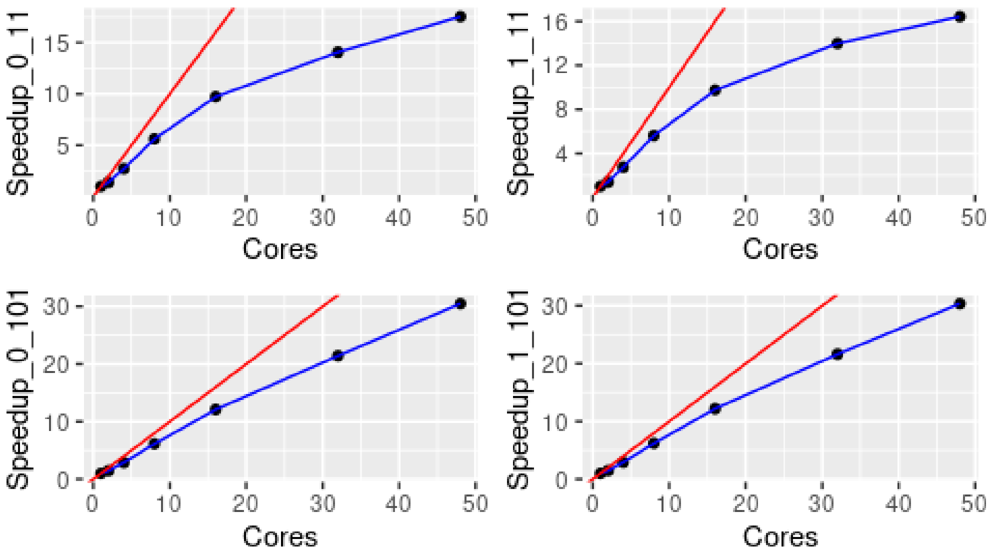

Figure 1: Strong scalabilty

For reference this are the times in seconds of the individual runs respectively (236, 235, 19030, 19183). The first number after speedup is the kernel type (0 for mean and 1 for weighted) the one following is the kernel size. As we can see program scales good until the 24 core mark, after which the hiperthreading may be the cause of the poor scalability. As one expects the scaling is better for the more intense tasks (ker_size = 101). Unfortunately this test is not really representative of the performance one may encounter using more than one node; in such cases communications speed would be significantly reduced and the code perform worse.

Section 2: Weak scalability

For the weak scalability, squared test-images were created such that the number of pixels in each one was proportional to the number of cores used in the computation: $\#pixels \sim cores$. This test was really interesting, before discussing it here is a graph with the results:

The strange result is due to the structure of the cluster and the relative small dimension of the images used. In fact the node used for the test has 2 different cpus and the first MPI instruction probably divide the workload above both of them with relative high latency in communication; later splits are instead done in the cpu itself those requiring less time in communication. The small workload is such that communications are not negligible.

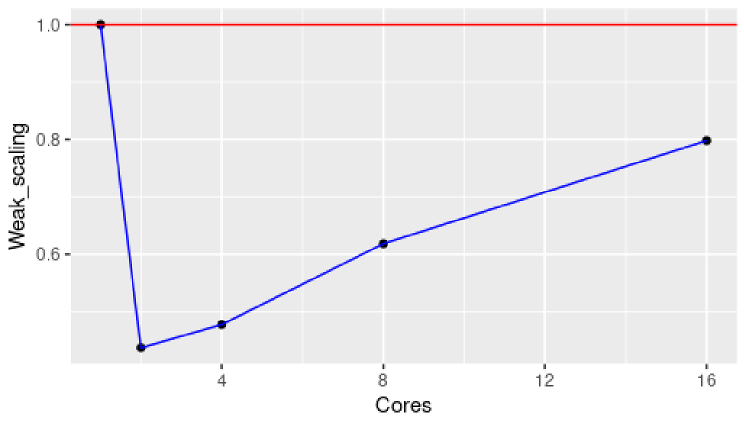

Figure 2: Weak scalability

Conclusions

The code written show good scalability for small number of cores and among one single node; unfortunately scalability over multiple nodes could not be performed due to the small walltime at disposal. Surly a good way to obtain better permormance rest in solving the problem in the division of the image described in the end of section 1. This ma be crucial for the program to scale on large cluster or when MPI communications are slower.