Local Binary Pattern: comparison between a sequential and parallel implementation

Beatrice Paoli
beatrice.paoli@stud.unifi.it

Abstract

The purpose of this paper is to implement the algorithm to compute the Local Binary Pattern for an image. There are many variations of this algorithm: the chosen implementation allows to choose any radius and number of pixels in the neighborhood and considers "uniform patterns" for labelling each pixel of the image. Both a sequential and a parallel implementation are presented and the two versions are compared in terms of execution times and speedup. The implementations are done in Python and the parallel version uses the Joblib package.

1. Introduction

Local Binary Pattern (LBP) is a simple and efficient texture descriptor that labels each pixel of an image with a binary number based on the neighboring pixels in order to create a local representation of the texture.

Used for texture analysis and classification, thanks to its computational simplicity it is widely used for many applications such as face analysis and motion analysis.

Many variants have been developed over time, but the basic idea of the LBP operator is to assign to each pixel of the greyscaled image a value by considering the difference of each of the 3x3 neighboring pixels with the center as binary values and then combine them in a clockwise or counter-clockwise to get the decimal label to assign to the pixel.

The histogram obtained from the occurrencies of these 2^8 labels can be used as a texture descriptor.

An extension of this basic concept allows to use circular neighborhoods of different sizes and different distances from the center. The operator is denoted as $LBP_{P,R}$ where P is the number of sampled neighbors and R is the radius of the circle where the sampled pixels stand.

If c_x and c_y are the coordinates of the center pixel, the coordinates of each neighbour pixel p can be computed as:

$$p_x = c_x - R\sin(2\pi p/P)$$
$$p_y = c_y + R\cos(2\pi p/P)$$

If these coordinates don't end into the center of a pixel, the grayscale value is bilinearly interpolated.

At this point, the LBP operator is computed as:

$$LBP_{P,R} = \sum_{p=0}^{P-1} s(g_p - g_c)2^p$$

where g_c and g_p are respectively the grayscale values of the center pixel and neighbouring pixel and:

$$s(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwhise} \end{cases}$$

This version of the LBP operator is not rotation-invariant as it depends on the order of consideration of the $0...P-1$ neighbours. So rotating the image will lead to different results.

To solve this problem, another variant of the LBP operator considers *uniform patterns* to achieve rotation-invariance. A circular neighbourhood of pixels is said to have a uniform pattern if there are few spatial transitions.

More formally, we define the function U that calculates the number of spatial transitions (bitwise 0/1 changes) in the neighbourhood pattern.

Only patterns with a value of U of at most 2 are considered uniform. With this, a new definition of the LBP operator is given.

$$LBP^{riu2}_{P,R} = \begin{cases} \sum_{p=0}^{P-1} s(g_p - g_c), & \text{if } U(LBP_{P,R}) \leq 2 \\ P+1, & \text{otherwhise} \end{cases}$$

where

$$U(LBP_{P,R}) = |s(g_{p-1} - g_c) - s(g_0 - g_c)| + \sum_{p=1}^{P-1} |s(g_p - g_c) - s(g_{p-1} - g_c)|$$

This reduces the number of possible labels to $P + 2$ values.

After obtaining the labeled image $f_l(x, y)$ the LBP histogram can be defined as:

$$H_i = \sum_{x,y} I\{f_l(x, y) = i\}, i = 0...n - 1$$

where n is the number of different labels assigned and:

$$I(A) = \begin{cases} 1, & \text{if } A \text{ is true} \\ 0, & \text{otherwhise} \end{cases}$$

If we need to compare histograms of images of different sizes, they must be normalized first:

$$N_i = \frac{H_i}{\sum_{j=0}^{n-1} H_j}$$

2. Implementation

The implementation is done in Python and uses the *Pillow* package to load and save images in RGB format, and the *numpy* package for handling arrays.

The application first loads the image in grayscale and converts it into a numpy array.

In the LBP function, the image is padded by adding a frame of pixels with the constant value of 0 and is done by using the numpy function `pad`. This is necessary for the computation of the LBP labels of the edges and to allow any radius to be used as a parameter. The size of the padding on each side is R rounded up to the nearest integer.

The LBP labels are then computed with the use of uniform patterns. To bilinearly interpolate the grayscale value g_p at (p_x, p_y), the following formula is used:

$$g_p = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \cdot \begin{bmatrix} x_2 - p_x & p_x - x_1 \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} y_2 - p_y \\ p_y - y_1 \end{bmatrix}$$

where $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$, $Q_{22} = (x_2, y_2)$ are respectively the bottom-left, top-left, bottom-right and top-right closest valid pixel values to (p_x, p_y).

The pseudo-code of the LBP function is the following:

Algorithm 1 Local Binary Pattern

Require: input image, P, R

```
Pad input image
for cy ∈ [0...height) do
    for cx ∈ [0...width) do
        lbp_val ← 0
        for p ∈ [0...P) do
            p_x ← c_x − R sin(2πp/P)
            p_y ← c_y + R cos(2πp/P)
            if p_x, p_y are integers then
                g_p ← input(p_x, p_y)
            else
                g_p ← bilinear_interpolation(p_x, p_y)
            end if
            if g_p ≥ g_c then
                lbp_val ← lbp_val + 1
            end if
        end for
        u ← U(LBP_{P,R})
        if u > 2 then
            lbp_val ← P + 1
        end if
        output(c_x, c_y) ← lbp_val
    end for
end for
return output image
```

Finally, the output array is used to compute the histogram and then both histogram and image are saved on a file.

3. Parallelization

The computation of the output label for each pixel only requires to read some pixels from the input image. The only write operation necessary is the assignment of the label to the output pixel and they are all independent from each other.

Therefore, this is an embarassingly parallel problem: each output pixel can be computed by a thread without the need of any synchronization.

The Python interpreter is based on the *Global Interpreter Lock* (GIL), which prevents the execution of multiple threads. To bypass this problem and achieve parallelization, we need to use processes instead of threads. Each process will have its own interpreter (including its own GIL) and memory, so their instantiation has a greater overhead compared to threads.

For this implementation, the *Joblib* library was used. It is a high level library that allows to parallelize computation through the use of a backend library (*multiprocessing* or *loki*) in a transparent way.

While tecnically each output pixel could be computed independently, it isn't convenient to spawn too many processess, nor to fraction a process job in too many parts.

Therefore the image is split in a number of sub-images equal to the number of processes to use. So each process computes only an output slice of the total output. The split and join is performed by the main process.

Each processes needs in input the image, the parameters P and R and the indices of the output sub-image. Each subprocess first performes some checks to avoid going out of bound and then computes the padded array (numpy's `pad` is extremely fast, no slowdown is caused by having this computation repeated for each process).

However, the only problem is represented by the input array which can be very big and can result to a very slow instantiation of the processess due to the array copy. This problem can be solved by using the `joblib.dump()` and `joblib.load()` functions to save and reload the numpy array efficiently.

The main process saves the array on disk and deletes it at the end of the function, while each process loads the file to perform the lbp operator on its slice of the image.

To recap, the main process:

1. Saves the input array on disk in a compressed format.
2. Splits the original image in sub-images.
3. Instantiates the sub-processes with the `Parallel` class, each with their own parameters.
4. After all sub-processess terminate, the output image is built from the returned slices.

The work of each subprocess is similar to the one described in Algorithm 1, the only things that changes are the for loops that only iterate on the assigned slice and the necessary shifting of some pixel coordinates.

4. Performance Analysis

In order to compare the performance of a sequential algorithm with its parallel version we can use the concept of *speedup*.

Speedup is measured as the ratio between the execution time of the sequential algorithm and the parallel one.

$$S = \frac{t_s}{t_p}$$

Ideally, we should look for *perfect speedup* or even *linear speedup*, meaning S should be equal or similar to the number of processors used to perform the parallel algorithm.

The tests were performed on a Intel Core i7-1165G7 with 4 physical cores and 8 logical cores. To evaluate the application performance, images of different sizes and different values for the parameter P were used.

The parameter R is not considered since its variation doesn't cause any change in execution times and speedup.

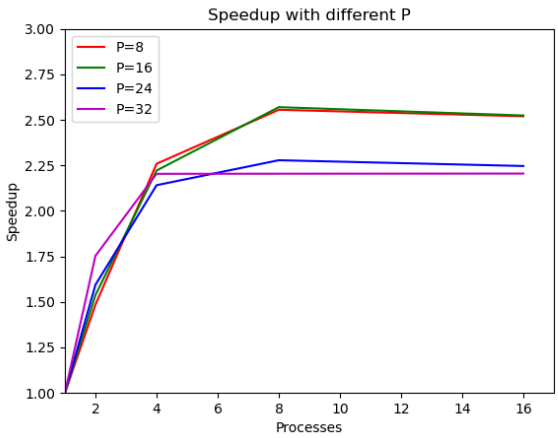

Figure 1. Speedup of the LBP operator on a 734x694 image and $R = 1$ with different values for P.

Processes	Execution Time (s)	Speedup
1	48.630	1.000
2	32.759	1.484
4	21.531	2.259
8	19.032	2.555
16	19.302	2.519

Table 1. Execution Times for P=8.

Processes	Execution Time (s)	Speedup
1	97.452	1.000
2	63.378	1.538
4	43.866	2.222
8	37.927	2.569
16	38.614	2.524

Table 2. Execution Times for P=16.

Processes	Execution Time (s)	Speedup
1	130.883	1.000
2	82.116	1.594
4	61.132	2.141
8	57.453	2.278
16	58.265	2.246

Table 3. Execution Times for P=24.

Processes	Execution Time (s)	Speedup
1	169.365	1.000
2	96.626	1.753
4	76.865	2.203
8	76.845	2.204
16	76.816	2.205

Table 4. Execution Times for P=36.

In Figure 1 we have the different speedups for different numbers of preocesses used and differente numbers of P, the neighbour size. The speedup increases when using between 2 to 8 processes and slightly decrease for higher numbers of processes, but still stay above the value of 2.

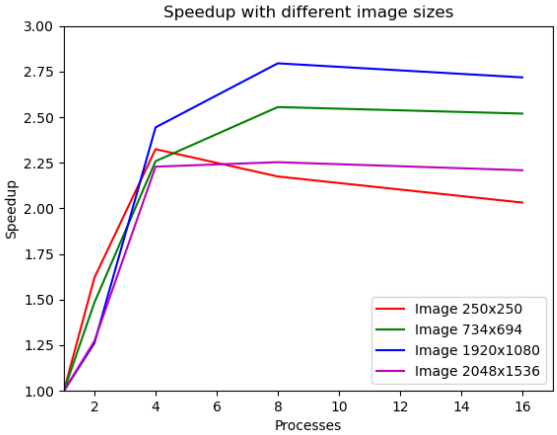

Figure 2. Speedup of the LBP operator with $P = 8$, $R = 1$ with different image sizes.

Processes	Execution Time (s)	Speedup
1	5.218	1.000
2	3.220	1.620
4	2.245	2.324
8	2.399	2.175
16	2.568	2.032

Table 5. Execution Times for the 250x250 image.

Processes	Execution Time (s)	Speedup
1	48.630	1.000
2	32.759	1.484
4	21.531	2.259
8	19.032	2.555
16	19.302	2.519

Table 6. Execution Times for the 734x694 image.

Processes	Execution Time (s)	Speedup
1	203.600	1.000
2	161.339	1.262
4	83.312	2.444
8	72.849	2.795
16	74.913	2.718

Table 7. Execution Times for the 1920x1080 image.

Processes	Execution Time (s)	Speedup
1	272.129	1.000
2	213.772	1.273
4	122.107	2.229
8	120.782	2.253
16	123.202	2.209

Table 8. Execution Times for the 2048x1536 image.

In Figure 2 we have the speedup trends for different input images. As before, the values increase between 2 to 8 processes and decrease afterwards.

For small images, like the 250x250 one, the execution times are very small even for the sequential version. As a consequence, smaller speedups are obtained for higher numbers of processes, since the overhead of process management starts to weight more than the computations to perform.