Parallel Computing Project 1 - Local Binary Pattern

Irene Campaioli
irene.campaioli@edu.unifi.it

Abstract

In this project we implement the Local Binary Pattern texture descriptor in C++, comparing the performances of a sequential approach and a parallel one using the OpenMP library.

1. Introduction

Local Binary Pattern (LBP) is a powerful texture operator that extracts a histogram as a descriptor of the texture itself. Due to its robustness to monotonic gray-scale changes and computational simplicity, LBP is a popular choice in Computer Vision applications, like image comparison or template matching. For each pixel in the image, the operator considers a neighborhood centered in said pixel, called pivot, and extracts a binary code by thresholding all the pixel values in the neighborhood.

Figure 1. LBP texture descriptor

For this exercise, we implemented LBP in its simplest form:

- grayscale-images
- 3x3 neighborhood
- central pixel as pivot

The result is a histogram of 256 bins, representing all possible 2^8 8-bit codes. This ensures robustness to illumination variations but it can still be sensitive to noise. The operator can therefore be extended to neighborhoods of different sizes, using circular neighborhoods and *uniform patterns* which assure invariance to rotation, but we will not be considering them for this exercise.

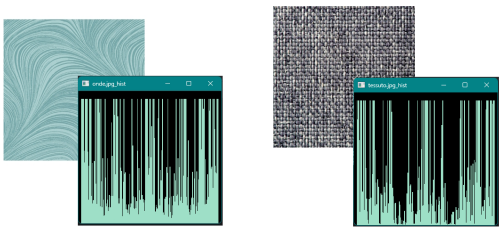

Figure 2. Result of LBP on two different textures.

2. Implementation

All code is available at `https://github.com/crashkeys`, divided in two folders for the sequential version and the OpenMP version respectively. It requires the OpenCV library to handle images inputs.

2.1. Sequential solution

The core of the program is in `LBP.cpp`, in which `createCode` generates the 8-bit code for each pixel and `drawHist` generates the histogram image. To handle pixels on the edges as pivots, a 1px black padding is added.

```
bitset<8> createCode(Mat& frame, int i, int j) {
    int neighbors[8];
    neighbors[0] = frame.at<uchar>(i - 1, j - 1);
    neighbors[1] = frame.at<uchar>(i - 1, j);
    neighbors[2] = frame.at<uchar>(i - 1, j + 1);
    neighbors[3] = frame.at<uchar>(i, j + 1);
    neighbors[4] = frame.at<uchar>(i + 1, j + 1);
    neighbors[5] = frame.at<uchar>(i + 1, j);
    neighbors[6] = frame.at<uchar>(i + 1, j - 1);
    neighbors[7] = frame.at<uchar>(i, j - 1);

    int threshold = frame.at<uchar>(i, j); //central
     pixel
    std::bitset<8> code;
    for (int ii = 0; ii < 8; ii++) {
        if (neighbors[ii] > threshold)
            code[ii] = true;
        else
            code[ii] = false;
    }
    return code;
}
```

The code is created clock-wise starting from the upper-left corner of the neighborhood; `frame.at<uchar>` reads the grayscale pixel value given the coordinates. If this value is greater than the central pixel, the correspond-

ing bit is set to `1`, otherwise to `0`. The function returns a `std::<bitset>` object, a class template of the standard library, representing specifically fixed-sequences of N bits.

```
Mat drawHist(Mat &frame) {
    int hist[256] = {}; //init

    for (int i = 1; i < frame.rows-1; i++) {
        for (int j = 1; j < frame.cols-1; j++) {
            bitset<8> code = createCode(frame, i, j);
            hist[code.to_ulong()]++;
        }
    }
    ...
```

The function `drawHist` scans the image on all pixel (two nested `for` loops) and generates the 8-bit code by calling the aforementioned `createCode` method. Thanks to the `<bitset>` class, we can easily transform it into an `int` using the `code.to_ulong()` method, and count all occurrences in the 256-bins histogram.

Refer to github for the second part of the code, that handles the graphics part and draws the histogram as a bar plot using the `openCV` library.

2.2. OpenMP solution

The multithread version doesn't change the structure of the code: a parallel directive is added in `drawHist` to handle the two nested for loops.

```
#pragma omp parallel reduction(+:hist) default(none)
     shared(frame)
    {
#pragma omp for
    for (int i = 1; i < frame.rows-1; i++) {
        for (int j = 1; j < frame.cols-1; j++) {
            bitset<8> code = createCode(frame, i, j);
            hist[code.to_ulong()]++;
            }
        }
    }
```

Using the reduction directive, we implicitly create a private `hist` variable for each thread, and then sum all the results into the public one.

This solution is possible only with OpenMP version 4.5 onward, which implemented the reduction directive. We also provide another solution that uses OpenMP 2.0. Thus, we have to use a critical section that handles the synchronization between all the threads

```
#pragma omp parallel shared(hist, frame, histImage)
     default(none)
{
    int priv_hist[256] = {};

#pragma omp for
    for (int i = 1; i < frame.rows-1; i++) {
        for (int j = 1; j < frame.cols-1; j++) {
            bitset<8> code = createCode(frame, i, j);
            priv_hist[code.to_ulong()]++;
        }
    }

#pragma omp critical
    for (int i = 0; i < 256; i++) {
        hist[i] += priv_hist[i];
    }
```

Image width	Sequential	OMP 2.0	OMP 4.5
		μs	
5	459	821	928
13	355	269	144
25	320	172	113
60	941	335	226
		ms	
900	143.1	26.7	18.6
1500	314.2	52.3	37.0
5000	3037.4	581.3	489.4
15000	27 765	5 039	3 302
30000	122 508	23 124	12 724

Table 1. Completion time of sequential version vs. parallel version of generated images of varying size.

We will see in the following section how this constraint affects the execution time.

The next section illustrates different tests and compares the computation time between all the solutions.

3. Experimental Results

All experiments are performed on a AMD Ryzen 5 7600 with 6 cores and 12 threads. All tests were repeated five times in order to validate the data measured.

First of all, the correctness of the method is tested on different textures, as shown in Fig.2. Different textures return different looking histograms. Using appropriate distance functions we could determine the difference or similarity between the two textures by calculating the distance between the two histograms.

3.1. Increasing the image size

We compare the completion time of the sequential version and the 12-threads parallel version on images of increasing sizes. We start from a 900x900px image and increase its size by adding a random grayscale value padding of $30 * i$ pixels, where i is the number of iteration, for a total of 30 iterations. We also performed the same experiment on very small images: starting from 5x5px and increasing its size by a 1px padding each iteration. Results are shown in Fig 3 and Table 1. We can see how the parallel version greatly decreases the completion time as the size of the image grows, and how the use of the critical section influences negatively the completion time of the parallel version using OMP v2.0.

3.2. Changing the number of threads

We performed the experiments up until now with a set number of 12 threads, as that's the number of logical cores our machines has. For this section of experiments, we test

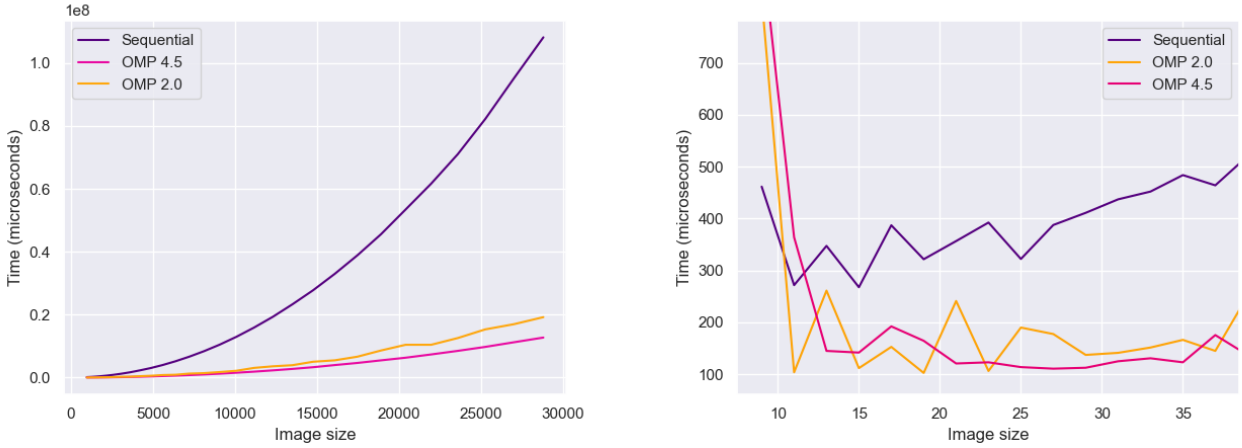

Figure 3. Completion time (in microseconds) for images with varying size. We can see how both parallel versions perform increasingly better as the width grows. The OMP v4.5 version is slightly better than the 2.0 one. On the right, we show a zoom of the performances for very small sizes. Here, the sequential version performs initially slightly better, as the parallel ones waste time in the creation of the threads. In this case, both the v4.5 and v2.0 are very similar.

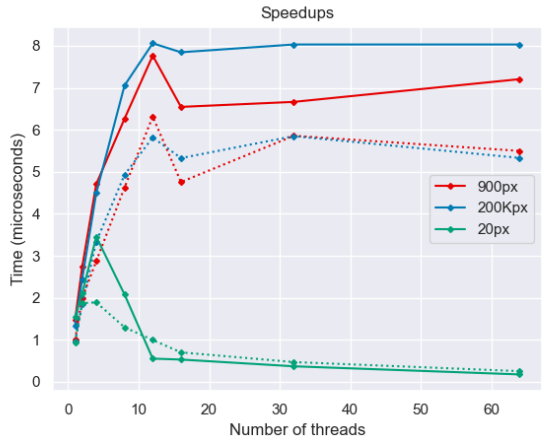

Figure 4. Speedup varying the number of threads, for different image sizes. Full lines refer to OMP v4.5 while dotted lines refer to OMP v2.0

the use of a different number of threads, and we will confirm that 12 threads was indeed the best choice.

Fig.4 shows the speedup of the parallel versions varying the number of threads for three images of different size. As expected, the performance has a peak at 12 threads, and it starts to slowly decrease after that. Despite that, the curve is always sub-linear. Comparing the two different OMP versions, the 4.5 one is consistently better, reaching a speedup of over 8x, and confirming the usefulness of the use of a reduction operation. It's interesting to notice the very different behavior of the small image: in this case the parallel version is better only for a fewer number of threads, and its performance decreases rapidly after that.

Threads	Image width		
	20Kx20K	900x900	20x20
1	1.35	1.47	1.55
2	2.44	2.73	2.09
4	4.49	4.71	**3.45**
8	7.05	6.26	2.08
12	**8.06**	**7.76**	0.55
16	7.84	6.54	0.53
32	8.03	6.66	0.37
64	8.03	7.20	0.18

Table 2. Speedups. For clarity sake, we only show the OMP 4.5 version

4. Conclusion

Despite it being a very easy problem to solve, even for the sequential version, the parallel approach that uses OMP 4.5 is almost always the preferable choice. The sequential version is to be chosen only for very small images, where it's not useful to waste time in creating threads.