Parallel Computing - 2024/25
Final assignment
Image stitching

Matteo Spataro
matteo.spataro@edu.unifi.it

Abstract

For this assignment, we implemented a sequential and a parallel image stitching program to create panoramic images using Python. This problem consists of assembling multiple images of a larger panorama to offer a wider field of view.

To obtain natural-looking panoramas, this requires implementing techniques in order to reduce the differences between the input images. After describing the main implementation details of the sequential version and the methodologies applied to parallelize the code, we will analyze the program using CProfiling.

Finally, the results will be shown on some images and the performance of the programs will be analyzed, also by modifying the parameters involved.

1. Introduction

Panoramic image stitching aims to extend the field of view beyond the limits imposed by a single photograph. Starting from multiple shots taken from different directions, the process is based on identifying common points of interest, matching them and estimating geometric transformations that can combine them into a single large image called a *panorama*. This fusion may require knowledge of certain camera parameters and the adoption of projection models that minimize visual distortions in the shots.

After compensating for any discrepancies in exposure and color balance, blending techniques are used to generate a final composition that is homogeneous and free of obvious artifacts. The result should be a single, fluid and continuous panoramic image capable of representing wide scenes without blurring effects such as vignetting.

In this assignment, we chose to process the images in pairs and to perform a *cylindrical projection*, i.e. to map each image from a planar structure to a spherical structure, in order to compensate for perspective distortions caused by the camera lens.

After that, the image is converted to grayscale and the corners of the images are found using the algorithm developed by Chris Harris & Mike Stephens called "Harris Corner Detection". It calculates the difference in light intensity by moving a small window in all directions, using the formula

$$R = det(M) - k \cdot (trace(M))^2 \tag{1}$$

where

- $det(M) = \lambda 1 \cdot \lambda 2$ (product of eigenvalues);
- $trace(M) = \lambda 1 + \lambda 2$ (sum of eigenvalues);
- k is an empirical parameter (typically 0.04).

So the magnitudes of these eigenvalues decide whether a region is a corner, an edge, or flat:

- when $|R|$ is small, which happens when $\lambda 1$ and $\lambda 2$ are small, the region is flat;
- when $R < 0$, the region is edge;
- when R is large, which happens when $\lambda 1$ and $\lambda 2$ are large and $\lambda 1 \sim \lambda 2$, the region is a corner.

Once all the key points of the two images have been found, i.e. after the extraction of the image features, a numerical representation is computed for each feature detected: this value is called "descriptor." By comparing these descriptors, it is possible to identify the same feature in different images, so as to calculate the geometric transformation necessary to merge the images.

Next, features found too close to the edges of the image are set to zero to avoid those that are partially outside and less relevant to the match. Finally, the features are filtered, keeping only those that are local maxima within a fixed-size window. This increases the chances of extracting descriptors that are unique enough to identify all important features and ensure that the descriptor remains

similar even after slight variations in the image, for example due to different brightness or the presence of noise.

After comparing the features of the two images, the matrix of matched points is passed to the RANSAC (RANdom SAmple Consensus) algorithm. This is an iterative method for estimating the parameters of a mathematical model in the presence of data affected by outliers, with the aim to remove them. Starting from a set of observations $D \subset \mathbb{R}^n$, the goal is to identify a subset $I \subset D$ of inliers that are compatible with a model M.

To do this, at each iteration a subset $S \subset D$ of points is randomly selected to generate an estimate of the model: if the number of points in S classified as inliers exceeds the best subset obtained so far, then S will be the new best subset.

In our case, the number of iterations has been set a priori by a global variable `RANSAC_K` and a warning message is printed if the sign of the new estimate is opposite to the sign of the previous estimate: this implies that the new image will be merged on the opposite side to the one previously.

As a final step, a stitching algorithm is applied to merge the two images. After researching the stitching algorithms available in the literature, we implemented a version that uses *pyramid blending*. This algorithm calculates the padding needed to align the two images, introducing black pixels where no valid data exists. From here, the method continues by merging the images row by row through pyramid blending: a binary mask is constructed in which the pixels have a value of "1" up to the junction point of the image. Next to this point, the pixels has a value of "0" and the method apply a Gaussian filter along the horizontal axis to achieve a smooth transition.

The implementation of pyramid blending attempts to faithfully reproduce the original algorithm and the row-by-row approach reduces the complexity of the memory used, with the downside of being less effective in images with features that are more extensive on the vertical axis than on the horizontal axis.

Since this assignment assumes that panoramic images are taken with movements on the horizontal axis, this method does not negatively affect the effectiveness of the stitching algorithm.

2. Implementation

The language used for this program is **Python** and we decided to split the code into 5 files:

1. "`main.py`" is the file through which the sequential and parallel versions of the program are executed;

2. "`feature.py`" contains all the methods for collecting and processing image features, i.e., the methods for "Harris Corner Detection", for extracting descriptors and for processing them in pairs;

3. "`stitch.py`" contains the methods for image stitching, including the outliers removal RANSAC and the methods for implementing pyramid blending;

4. "`utils.py`" collects all the more generic methods, i.e., to load images and to apply cylindrical projection and padding;

5. "`params.py`" contains all the program's constant variables, which are necessary for faster management of the parameters of the entire code.

2.1. Sequential

Table 1 describes the pseudocode of the sequential version of the Panoramic Image Stitching algorithm.

Algorithm 1 Sequential Image Stitching

```
1:  procedure SEQUENTIALSTITCHING
2:      images ← Load all input images
3:      cylinderImages ← empty list
4:      for all image ∈ images do
5:          projected ← cylindrical projection of image
6:          append projected to cylinderImages
7:      stitchedImg ← copy of cylinderImages[0]
8:      shifts ← [[0,0]]
9:      cacheFeatures ← [[], []]
10:     for i = 1 to length(cylinderImages) − 1 do
11:         img1 ← cylinderImages[i − 1]
12:         img2 ← cylinderImages[i]
13:         [descriptors1, position1] ← cacheFeatures
14:         if descriptors1 is empty then
15:             corners1 ← Harris corners in img1
16:             [descriptors1, position1] ← extract features
17:         corners2 ← Harris corners in img2
18:         [descriptors2, position2] ← extract features
19:         cacheFeatures ← [descriptors2, position2]
20:         matchedPairs ← match descriptors
21:         shift ← estimate shift using RANSAC
22:         append shift to shifts
23:         stitchedImg ← stitch stitchedImg with img2 using shift
24:     aligned ← align panorama using shifts
25:     output ← crop aligned to remove black borders
26:     return output
```

The images are read using the **OpenCV** library, using the method `cv2.imread("image path")`. This allows us to perform operations on images using lists

of `np.ndarray`, according to the NumPy array format: Height x Width × 3.

The translation vectors between images are stored as lists of integers, since panoramic alignment occurs at the pixel level. For feature descriptors and their positions, NumPy arrays or lists of tuples are used to support distance-based matching and robust estimation of correspondences using RANSAC.

Analyzing the computational complexity, for N images of dimension (H x W x 3) and suppose F as the average number of features in each image, it emerges that image loading, cylindrical projection, Harris Corner algorithm and image stitching algorithm have linear complexity $\mathcal{O}(N \cdot H \cdot W)$ because they operate linearly on each input image. As for RANSAC, it has been shown to have complexity $\mathcal{O}(K \cdot M)$, where K is the number of iterations and M is the number of matches being examined, while pyramid blending constructs each level in $\mathcal{O}(W)$ for H rows, for a total of $\mathcal{O}(W \cdot H)$.

Therefore, we conclude that the sequential algorithm of panoramic image stitching has superlinear complexity.

2.2. Parallel

The Global Interpreter Lock (GIL) prevents the simultaneous execution of more than one piece of Python bytecode. This means that for anything not related to I/O, using multithreading will not provide any performance benefits.

To overcome the GIL limitation, we used Python's multiprocessing library, which implements the shared memory programming paradigm. Multiprocessing generates processes instead of creating threads, so that parallelization occurs thanks to subprocesses that divide the work, in contrast to the OpenMP approach, where there is a parent process that manages the workload.

The instruction that allows us to create a pool of processes is `mp.Pool(cores)`, creating the number specified by the `cores` variable. In this assignment, we used the `pool.starmap` method to project each image onto the cylindrical coordinates independently: starmap divides the tasks on the images between the processes in the pool.

Feature matching is also parallelized, dividing the set of descriptors of the first image into the number of processes in the pool and comparing them with the descriptor of the second image. To improve parallelism, at each process will be assigned a quantity of work defined by the `chunksize` variable, with the aim of reducing scheduling overhead.

A final method subjected to parallelization is the blending algorithm: after aligning the images under examination, mergers are performed in `chunksize` groups, creating processes that execute the pyramid blending algorithm on parts of the same image simultaneously.

3. Profiling

Python offers two tools for profiling code, i.e., measuring which parts of the program are executed most often and reporting the time taken by each. These tools are "**cProfile**" and "profile" and in this assignment we used the first.

The following instructions were added to the `main.py` file

```
with cProfile.Profile() as profile:
   sequentialStitching()
```

and the results are saved and analyzed using `pstats.Stats()` with the instructions

```
results = pstats.Stats(profile)
results.sort_stats(pstats.SortKey.TIME)
results.print_stats(20)
results.dump_stats("seqStitching.prof")
```

Through this profiling, we were able to focus on optimizing the slowest parts of the code that were causing bottlenecks. Specifically, thanks to the installation of the "*snakeviz*" package, it was possible to dynamically visualize the generated profiling, like the ones in Figure 1 and Figure 2. The first one shows the screenshots of the profiling obtained from the sequential version of the program, while the second one shows the profiling of the parallel version.

From the profiling, we can see that the parallel part is actually faster in terms of average execution time, but almost 77% of the time is spent on process overhead. Despite various attempts, we were unable to further reduce the overhead time, but we believe that it is (largely) intrinsic to the multiprocessing model in Python.

4. Experiments and results

The experiment conducted in this project aims to quantify the time advantage derived from using parallelization through Python's multiprocessing library.

We applied the image stitching method to two sets of photographs contained in the "input" folder:

1. the photos in the "libraryHD" folder are 8 images taken vertically outside the San Giorgio library in Pistoia: they have a size of 3472x4640, for a total of 16.110.080 pixels and an average file size of 6,125 MB;

2. the photos in the "library" folder are the reduced versions of the previous 8 images: with a size of

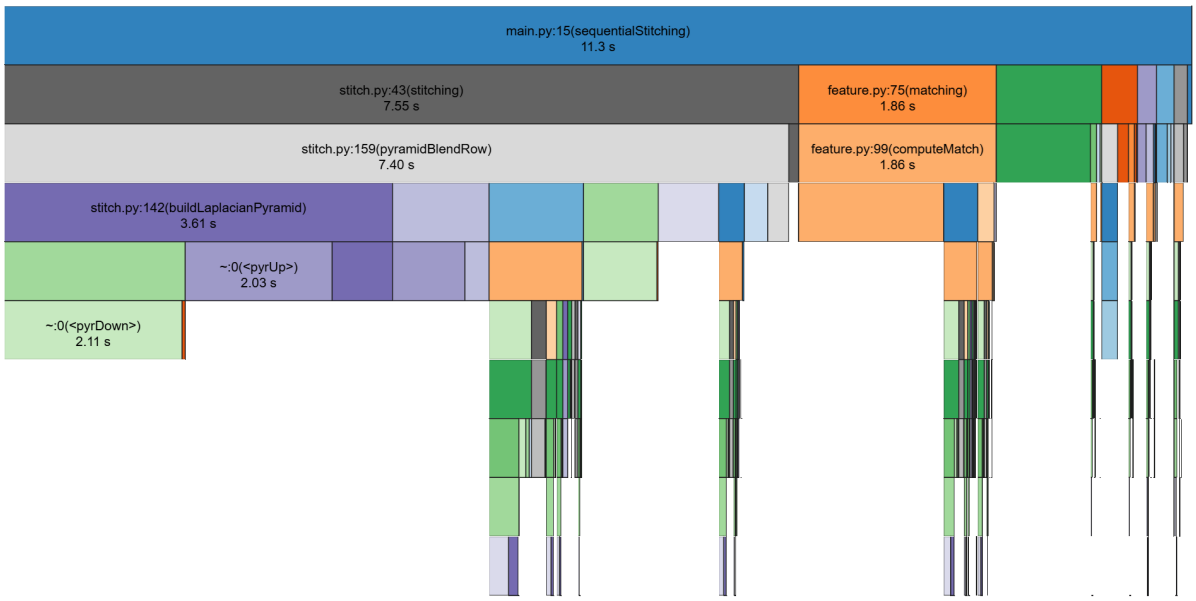

Figure 1: Snakeviz visualization of the profiling obtained from the sequential program.

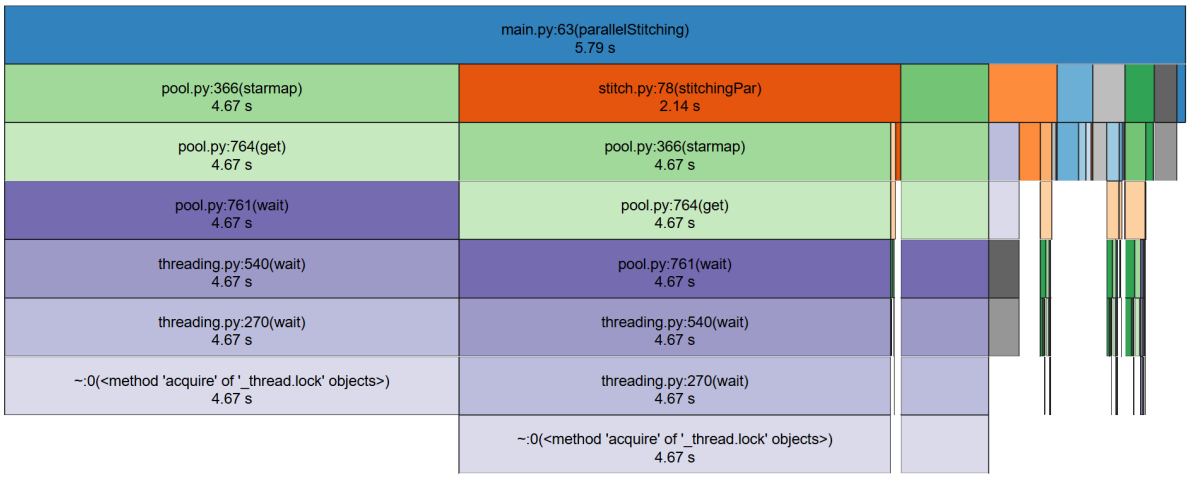

Figure 2: Snakeviz visualization of the profiling obtained from the parallel program.

1080x1443, they contain about 1/10 of pixels of the original version, with an average file size of 675 KB.

Having two sets of photos that differ in file size allows us to observe how the parallel program behaves as the size of the problem increases.

The first experiment we conducted aimed to vary the number of cores involved in parallelization, comparing each execution time with the results obtained from the sequential version of panoramic image stitching. The number of cores was varied from 2 to 16 with step size of two, while all the other parameters were left unchanged.

The results obtained are shown in Figure 3, where the blue curve is made from the execution times of the programs over the "low" resolution images and the orange curve utilized the HD version of them. These results show us how increasing the number of cores in Python multiprocessing parallelism brings benefits in reducing the completion time of the entire program, but the quality of these benefits diminishes more and more until reaching the optimum. In fact, it is possible that increasing the cores may move us away from the optimum, as can be seen from the times obtained on small images.

The speedup of a parallel program with p cores is given

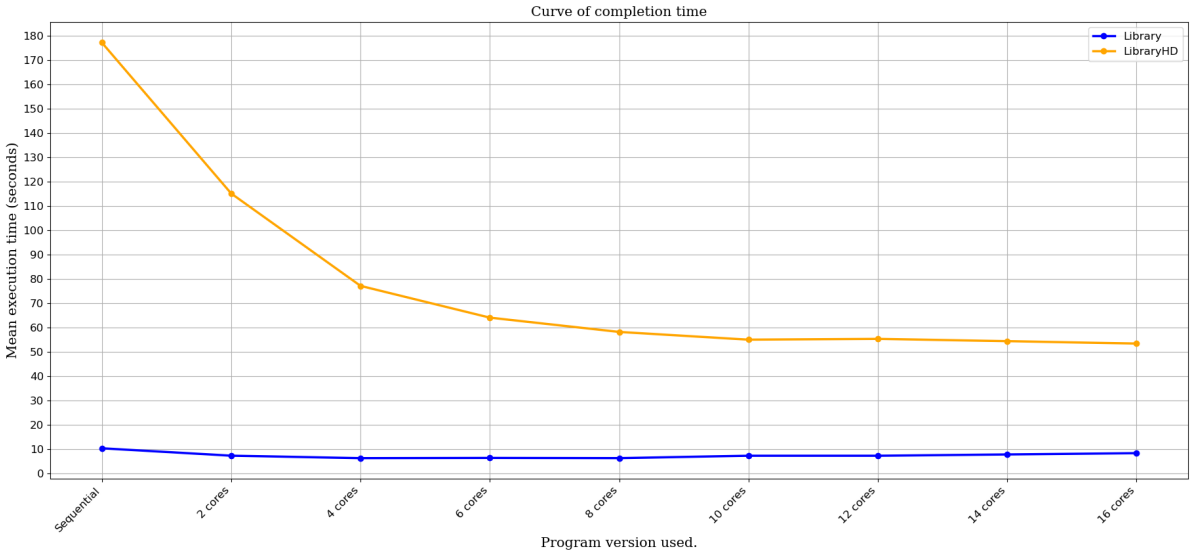

Figure 3: Curve of the mean execution time varying the number of cores.

by the formula

$$S_p = \frac{t_s}{t_p} \tag{2}$$

where t_s is the time in seconds obtained from the sequential version of the program, while t_p is the time in seconds obtained from the parallel program with p cores. Figure 4 shows the speedup curve of our parallel image stitching program as the number of available cores varies. On the other hand, when using HD images, the speedup curve grows steadily, reaching a value of 3,3 with 16 cores. The increase is significant and regular, indicating that the parallel version of the code needs a certain dimensionality of the problem in order to scale much better.

The last experiment aims to evaluate the quality of the resulting panoramic image. Unfortunately, there is no formal metric that we can apply in this case, so we based our assessment on the visual perception of the resulting image. We experimented with the values of the parameters involved, such as the size of the Gaussian window and the blending window, the size of the image's focal point, and collected all the results in the "comparisons" folder.

To visualize the results, Figure 5 shows all 8 input images and Figure 6 shows the panoramic image produced by our panoramic image stitching algorithm.

5. Conclusions

In conclusion, the implementation of panoramic image stitching in Python allowed us to explore both the theoretical and practical aspects of image processing using parallel programming.

After designing a sequential version of the algorithm, we created a parallel version using the multiprocessing library, with the aim of improving execution times by distributing the computational load across multiple processes.

The program's performance was profiled using cProfile and Snakeviz, which highlighted how certain phases of the algorithm were the main bottlenecks of the program. By parallelizing these phases, we achieved a reduction in average execution time, especially in the case of higher-resolution images.

Specifically, processing the "libraryHD" image set (8 photographs, each measuring 3472x4640 pixels) showed a performance improvement in the parallel version compared to the sequential version, reducing the total execution time from 177 seconds to 55 seconds. On the other hand, for the reduced "library" dataset, the execution speed increased only marginally, a sign that parallelism only becomes truly effective when the computational load justifies the overhead introduced by process management.

Overall, the results obtained confirm the effectiveness of parallelization for computationally intensive tasks such as high-definition panoramic image stitching.

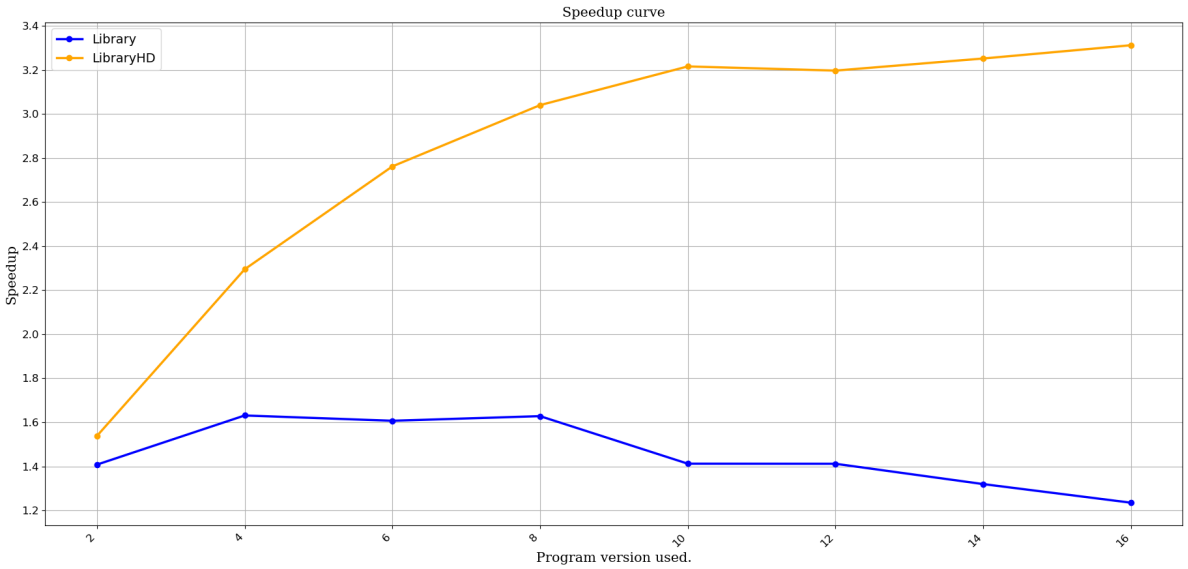

Figure 4: Speedup curve of the parallel image stitching varying the number of cores available.

(a) Image 1

(b) Image 2

(c) Image 3

(d) Image 4

(e) Image 5

(f) Image 6

(g) Image 7

(h) Image 8

Figure 5: Grid with "Library" images.

Figure 6: Panoramic image output.