Performance comparison of an image composition program with sequential and parallel implementations

Giovanni Casini
7108181

Federico Chiesa
7108176

Abstract

The aim of this paper is to show how a parallel image composition program - an implementation of alpha composition based on the OpenCV library - can dramatically speed up image composition that can be used to generate datasets for object detecting neural networks.

Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

The algorithm we will be implementing is an alpha blending algorithm that works by blending the pixel values of two images based on their alpha (transparency) values.

In particular, our implementation will take a background image and a foreground image to be overlaid on the background image as input, then calculates a random point on the background image on which it will overlay the foreground image, combined with random scaling and rotation transformations.

The use of this algorithm in practice is to generate large datasets for object recognition neural networks automatically and without user intervention.

2. Implementation

Both sequential and parallel implementations share the same function called `add_transparent_image()`. This function performs the following steps:

- Overlay preprocessing: if needed this part of the code scales and/or rotates the overlay image

```
    if scalePercent != 100:
        width = int(foreground.shape[1] *
    scalePercent / 100)
        height = int(foreground.shape[0] *
    scalePercent / 100)
        dim = (width, height)
        foreground = cv2.resize(foreground,
    dim, interpolation = cv2.INTER_AREA)

    if rotationAngle != 0:
        foreground = imutils.rotate_bound(
    foreground, rotationAngle)
```

Listing 1. Overlay preprocessing

- Images metadata gathering: this code gets image metadata (height, width, channels) through cv2 library and checks that the number of channels is correct

```
    bg_h, bg_w, bg_channels = background.
    shape
    fg_h, fg_w, fg_channels = foreground.
    shape

    assert bg_channels == 3, f'background
    image should have exactly 3 channels (RGB
    ). found:{bg_channels}'
    assert fg_channels == 4, f'foreground
    image should have exactly 4 channels (
    RGBA). found:{fg_channels}'
```

Listing 2. Images metadata gathering

- Calculate sizes and crop images: this part of the code calculate the final image's sizes and crops background and foreground images

```
w = min(fg_w, bg_w, fg_w + x_offset, bg_w -
    x_offset)
    h = min(fg_h, bg_h, fg_h + y_offset, bg_h
     - y_offset)

    if w < 1 or h < 1: return

```

```
    # clip foreground and background images to the overlapping regions
    bg_x = max(0, x_offset)
    bg_y = max(0, y_offset)
    fg_x = max(0, x_offset * -1)
    fg_y = max(0, y_offset * -1)
    foreground = foreground[fg_y:fg_y + h, fg_x:fg_x + w]
    background_subsection = background[bg_y:bg_y + h, bg_x:bg_x + w]
```

Listing 3. Calculate sizes and crop images

- Image overlaying: overlays the two images creating the resulting image

```
# separate alpha and color channels from the foreground image
    foreground_colors = foreground[:, :, :3]
    alpha_channel = foreground[:, :, 3] / 255  # 0-255 => 0.0-1.0

    # construct an alpha_mask that matches the image shape
    alpha_mask = np.dstack((alpha_channel, alpha_channel, alpha_channel))

    # combine the background with the overlay image weighted by alpha
    composite = background_subsection * (1 - alpha_mask) + foreground_colors * alpha_mask

    # overwrite the section of the background image that has been updated
    background[bg_y:bg_y + h, bg_x:bg_x + w] = composite
```

Listing 4. Image overlaying

2.1. Sequential implementation

For the sequential implementation we used two `for` loops. The outer one iterates over `numberOfRuns` and the inner one iterates over `numberOfOverlays`. In the latter we run the `add_transparent_image()` function with random offsets, scaling and rotation. At the end of every run we store the time taken to complete the task in an array and then we calculate the mean value over the runs.

```
timesArray = []
for _ in range(numberOfRuns):
    start = time.time()
    for _ in range(numberOfOverlays):
        rotationAngle = random.randint(0, 359)
        scale = random.randint(25, 200)
        offsetX = random.randint(0, int(backgroundWidth - overlayWidth))
        offsetY = random.randint(0, int(backgroundHeight - overlayHeight))
        img = background.copy()
        add_transparent_image(img, overlay, offsetX, offsetY, scale, rotationAngle)
        #cv2.imshow("", img)
        #cv2.waitKey()
    end = time.time()
    timeTaken = end - start
    timesArray.append(timeTaken)
    print("Run ended in", timeTaken)
print("Average time taken for", numberOfRuns, " runs:", np.mean(timesArray))
```

Listing 5. Sequential main function

2.2. Parallel implementations

For the parallel implementations we used Joblib and Multiprocessing.

These two methods are similar in terms of how they run parallel computations, their main difference is in the syntax. We'll see in the following sections of this paper that the Joblib syntax is much more succint and cleaner than the Multiprocessing one.

2.2.1 Joblib

In the Joblib implementation we have a `for` loop that iterates over `numberOfRuns`, then we create the `Parallel` object that computes the `threadFunction` in parallel.

```
timesArray = []
    for _ in range(numberOfRuns):
        start = time.time()
        joblib.Parallel(n_jobs=numberOfThreads)(joblib.delayed(threadFunction)(background.copy(), overlay) for _ in range(numberOfOverlays))
        end = time.time()
        timeTaken = end - start
        timesArray.append(timeTaken)
        print("Run ended in", timeTaken)
        threadArray = []
    print("Average time taken for", numberOfRuns, "runs:", np.mean(timesArray))
```

Listing 6. Joblib main function

2.2.2 Python Multiprocessing

In the Python Multiprocessing implementation we have a `for` loop that iterates over `numberOfRuns`, then another `for` loop to create the threads and two more to start and join them.

```
timesArray = []
    for _ in range(numberOfRuns):
```

```
        for _ in range(numberOfProcesses):
            processArray.append(Process(target=
    threadFunction, args=[background.copy(),
    overlay, int(numberOfOverlays /
    numberOfProcesses), backgroundWidth,
    backgroundHeight, overlayWidth, overlayHeight
    ]))
        start = time.time()
        for process in processArray:
            process.start()
        for process in processArray:
            process.join()
        end = time.time()
        timeTaken = end - start
        timesArray.append(timeTaken)
        print("Run ended in", timeTaken)
        processArray = []
    print("Average time taken for", numberOfRuns,
     "runs:", np.mean(timesArray))
```

Listing 7. Multiprocessing main function

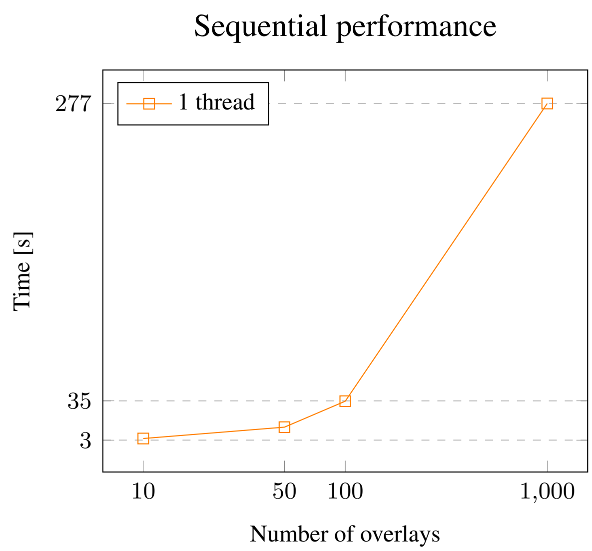
3. Tests

To test the speed of the different implementations of the algorithm, we ran them multiple times with a different number of overlay operations to do and then made an average of the times to make the tests as precise as possible.

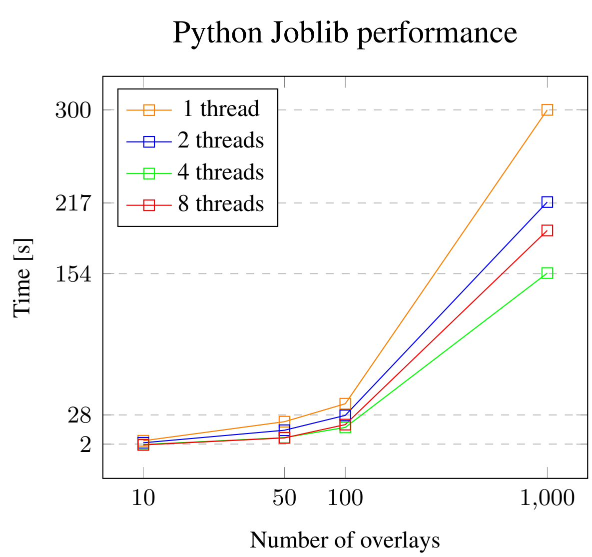
In the parallel implementations we kept the overall number of overlay operations the same as in the sequential implementation e.g. if we make a run with 100 overlay operations in the sequential implementation, in the parallel implementation we'd run 4 threads with 25 overlay operations each.

We ran all the tests on a Ubuntu 22.04 VM using VMWare Player with 16GB RAM and a maximum of 8 cores from the host machine, which has an Intel Core i7-1255U CPU.

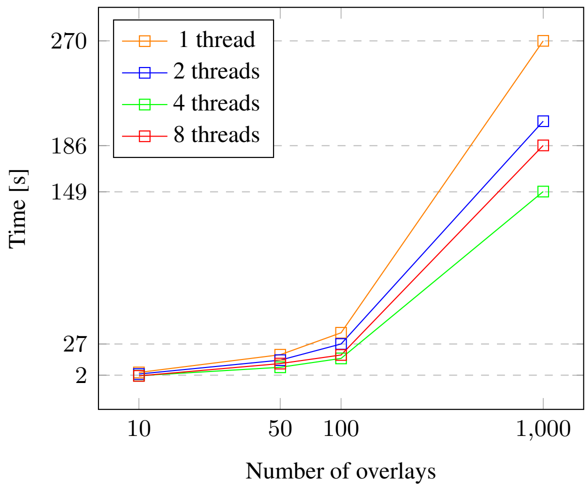
3.1. Sequential Implementation



3.2. Parallel Implementations



Python Multiprocessing performance

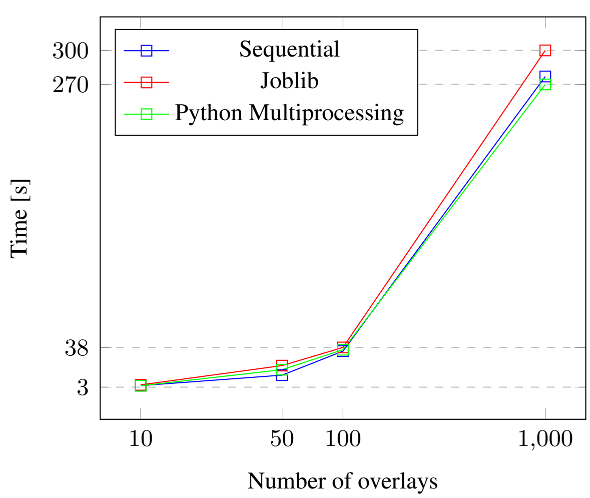


4. Conclusions

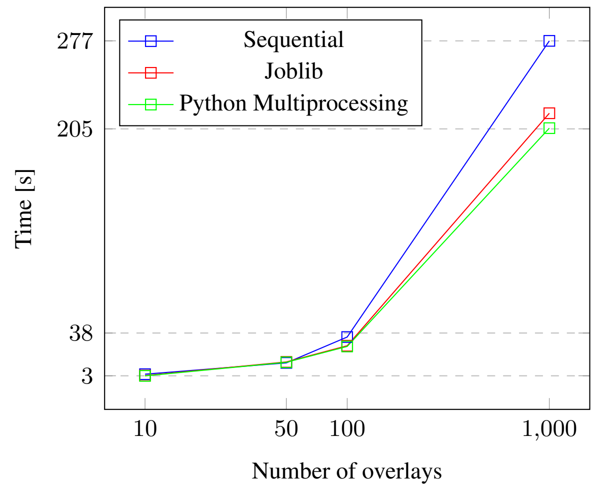
We can observe that all three implementations scale similarly, but the two parallel implementations have a significant time advantage compared to the sequential one.

With longer times, it looks like the Multiprocessing implementation is the best performing.

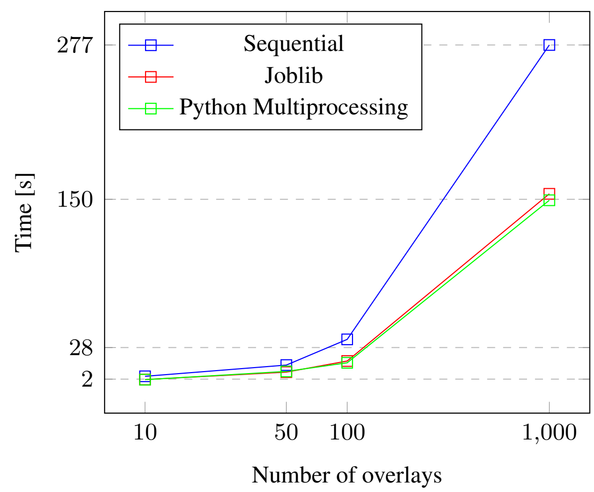
Comparison 1 thread



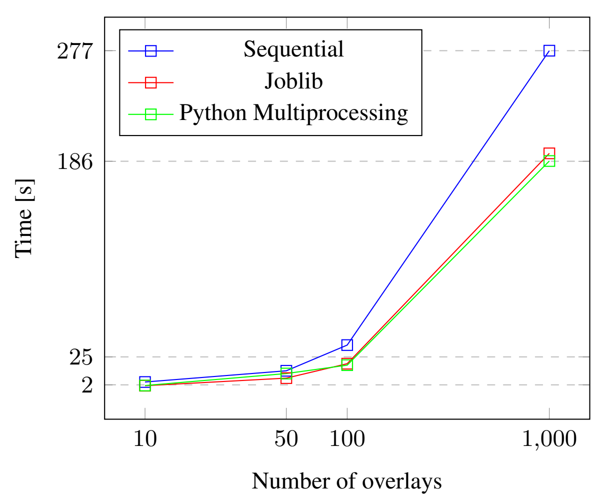
Comparison 2 threads

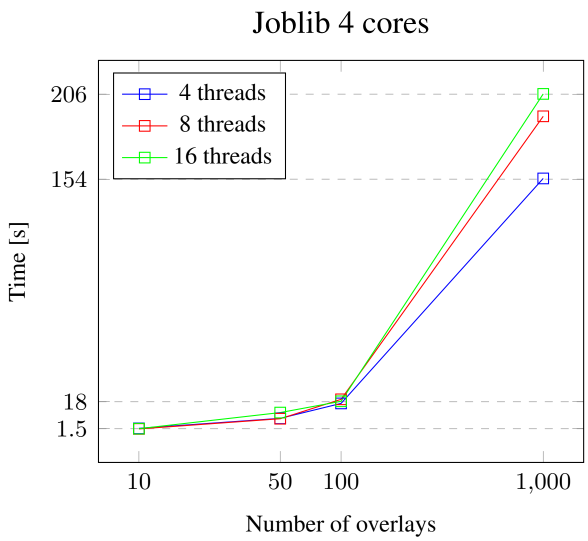


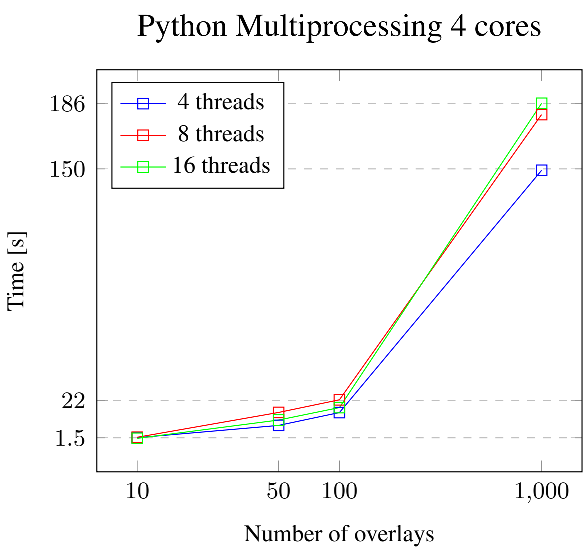
Comparison 4 threads



Comparison 8 threads




Joblib 4 cores
4 threads
8 threads
16 threads
Time [s]
206
154
18
1.5
10
50
100
1,000
Number of overlays


Python Multiprocessing 4 cores
4 threads
8 threads
16 threads
Time [s]
186
150
22
1.5
10
50
100
1,000
Number of overlays