Object Tracker

EC 526, Spring 2020, Boston University

Shubham Arora (aroras@bu.edu)
Krishna Palle (pallekc@bu.edu)
GitHub

Abstract

Object tracking is a well-studied problem within the area of image processing. The ability to track objects has improved drastically during the last decades, however, it is still considered a complex problem to solve. The importance of object tracking is reflected by the broad area of applications such as video surveillance, human-computer interaction, and robot navigation. We propose an application called Object tracker, which, upon receiving an input video and the coordinates of the object in the first frame of the video, will output the video with a bounding box around the object tracking it throughout the video. This application was built using the OpenCV library. The algorithm used in this application is then parallelised using OpenACC and OpenMPI to improve the performance of the application.

1 Introduction

This project proposes an application called Object Tracker which is an object tracking application that traces an object throughout the video. The crux of the project depends on an edge detection algorithm that we are using. The next section discusses in brief about Edge Detection and the algorithm used in the Project. The following section dives deeper discussing the algorithm used in the project and the flow of data. Later we discuss about the parallelisation techniques used and their impact on the performance with an example test case. Finally we discuss the results followed by conclusion and future work of the project.

2 Edge Detection

Edge detection can be classified as a bunch of mathematical methods that are applied on a grey scale digital image to identify set of curved lines where the image brightness changes sharply calling them edges. This can also be said as finding discontinuities in the image with respect to brightness. Figure 1 shows an example image before and after applying edge detection on it.

There are multiple edge detection algorithms in use currently and this project uses the Canny Edge Detection algorithm. The Canny Edge detector is an operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986[1].

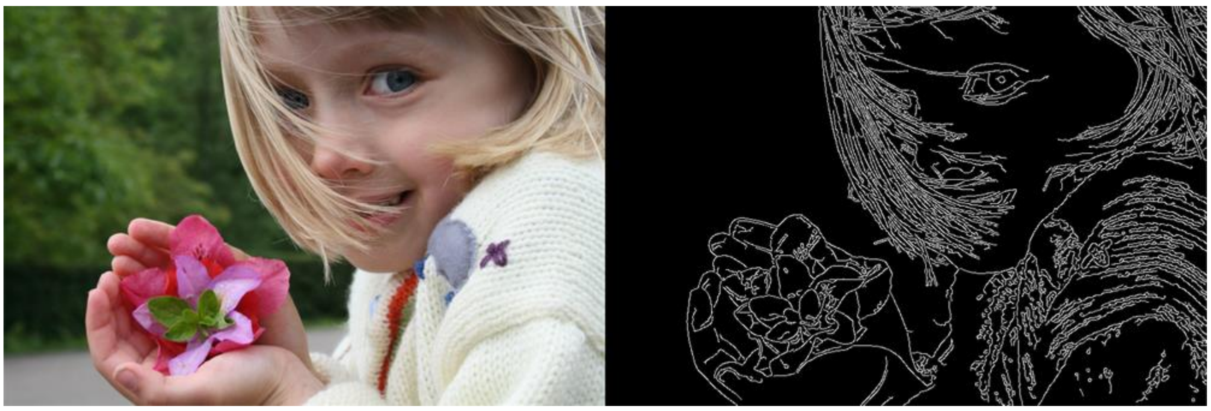


Figure 1: An example image before and after applying edge detection algorithm [1]

We can breakdown the Canny Edge Detection algorithm into 5 steps.

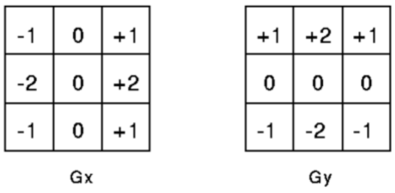
- Noise Reduction
- Gradient Calculation
- Non-maximum suppression
- Double Thresholding
- Edge tracking by Hysteresis

The noise reduction phase is when we take a grey scale image and apply a gaussian blur to it. We have a gaussian kernel and we convolve that kernel onto the image and reduce the noise. This will enable us to calculate the gradients of the pixels in the image more accurately. The formula for the gaussian blur kernel can be found below.

$$H_{ij} = \frac{1}{2\pi\sigma^2} exp\left(-\frac{\left(i-(k+1)\right)^2 + \left(j-(k+1)\right)^2}{2\sigma^2}\right); 1 \leq i,j \leq (2k+1)$$

Once we apply the gaussian blur to the image, we then calculate the gradients to detect the edge direction and intensity at every pixel. To do so, we have a lot of operators or kernels available. In this project we used Sobel operator to perform the gradient intensity and direction calculation. Image 2 shows how a Sobel kernel would look like. *Gx* is the output of a pixel when x-axis Sobel kernel is convolved with the image and respectively *Gy* is the output of a pixel by convolving the image with the y-axis Sobel Kernel. Below are the equations to calculate the intensity *G* and the direction θof the gradient

$$|G| = \sqrt{Gx^2 + Gy^2} \qquad \theta = arctan\left(\frac{Gy}{Gx}\right)$$



-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 2: Sobel Kernels in x-axis and y-axis direction

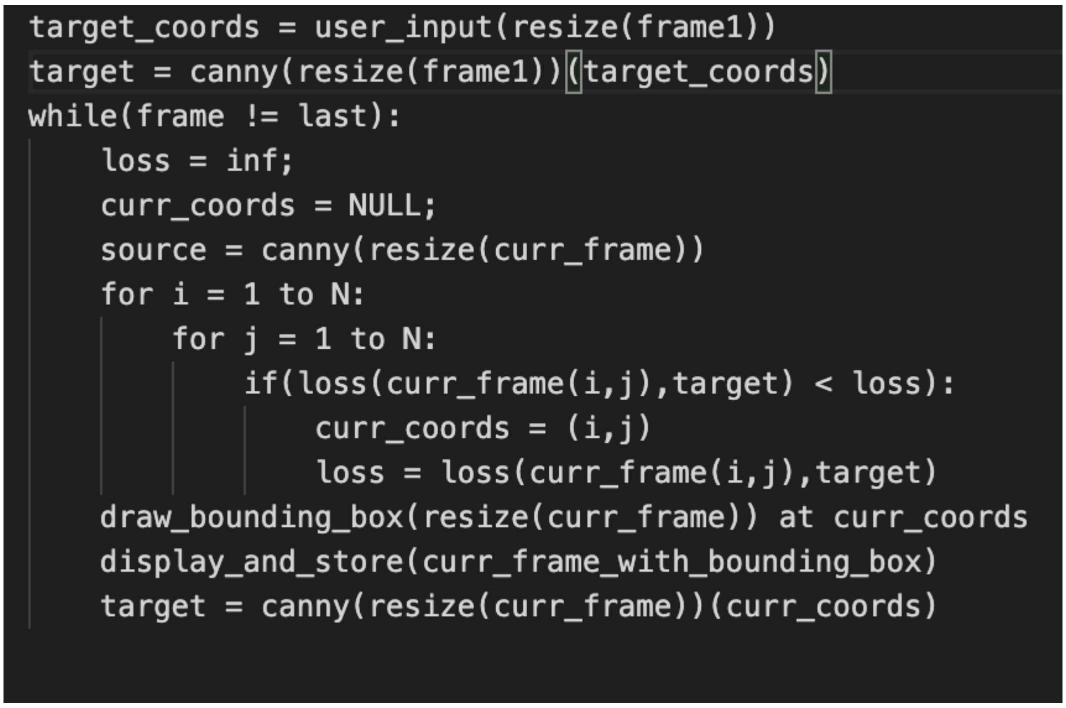
Once we have calculated the gradient intensities and direction, we then proceed to non-maximisation step. In this step we walk through all pixels and in the direction of its gradient, we find the pixel with highest intensity among its neighbours and neutralise all other pixels by setting their value to 0. This will remove all the thin edges that are irrelevant keeping only those high intensity edges for further processing.

Then we move to the double thresholding step. In this step, we neutralise all pixels below the low threshold value. All the pixels with intensity above the higher threshold are considered strong ones and contribute to edges. All the other pixels that fall between upper and lower threshold are considered weak pixels and the decision of keeping them or getting rid of them is decided in the next step.

The final step of the edge detection algorithm is edge tracing. All the weak pixels that are neighbours of strong pixels are considered to be contributing to the edges and hence those are preserved and all the other pixels are neutralised. The output coming out of this is the edge detection output of the algorithm.

3 Object tracker

Since we now understand how edge detection algorithm works, we use this edge detection algorithm in tracking object. Initially we pick the first frame and ask the user to draw a bounding rectangular box around the object he wants to track. We then run frame by frame in the video and use that information to track the object in each frame by running them through canny edge detection algorithm.



```
target_coords = user_input(resize(frame1))
target = canny(resize(frame1))(target_coords)
while(frame != last):
    loss = inf;
    curr_coords = NULL;
    source = canny(resize(curr_frame))
    for i = 1 to N:
        for j = 1 to N:
            if(loss(curr_frame(i,j),target) < loss):
                curr_coords = (i,j)
                loss = loss(curr_frame(i,j),target)
    draw_bounding_box(resize(curr_frame)) at curr_coords
    display_and_store(curr_frame_with_bounding_box)
    target = canny(resize(curr_frame))(curr_coords)
```

Figure 3: Object tracker algorithm

We run the first frame through canny edge detector and crop the output with the coordinates recorded as input from the user and call that `target`. In other words, target is the edge detection output of the object that user wanted to track. We then loop over all the frames to trace the object. For a single frame, we run it through canny edge detector and call the output as `source`. We then convolve the target over the source and calculate the loss of that target at every pixel location. We identify the object's new location is at the location with the lowest loss. The loss function is shown below.

$$loss_{ij} = \sum_{x}\sum_{y}\left|source_{ij} - target_{ij}\right|^2$$

Once we identify the pixel location with the lowest loss, we store the current frame after drawing a bounding box from the identified pixel location with the same height and width of the target. We also crop the output of canny edge detector of the current frame with the coordinates of the identified pixel location and update the value of target for the next frame. We perform these operations in a loop until the last frame. We then end up at the output of the algorithm.

Figure 4, shows the first frame of the video and the input the user has given us by drawing a rectangular bounding box around the object he wants to track. Figure 5 and 6 show an the canny output which is the source and the target value which would be convolved against the source to identify the location in the current frame.

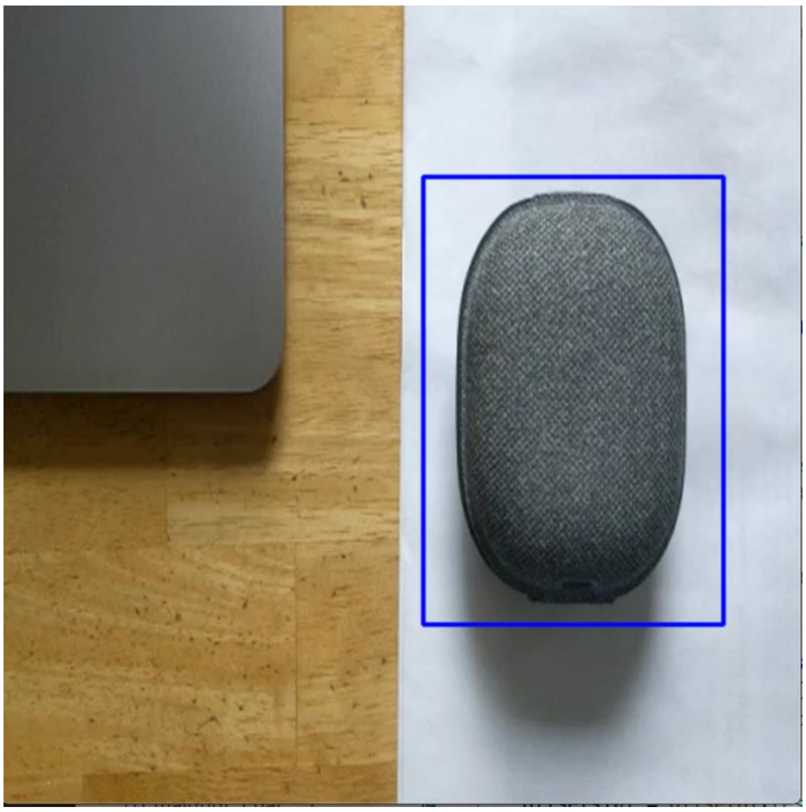


Figure 4: Frame 1 of the video with the user input which is a bounding box on the object the user wants to track

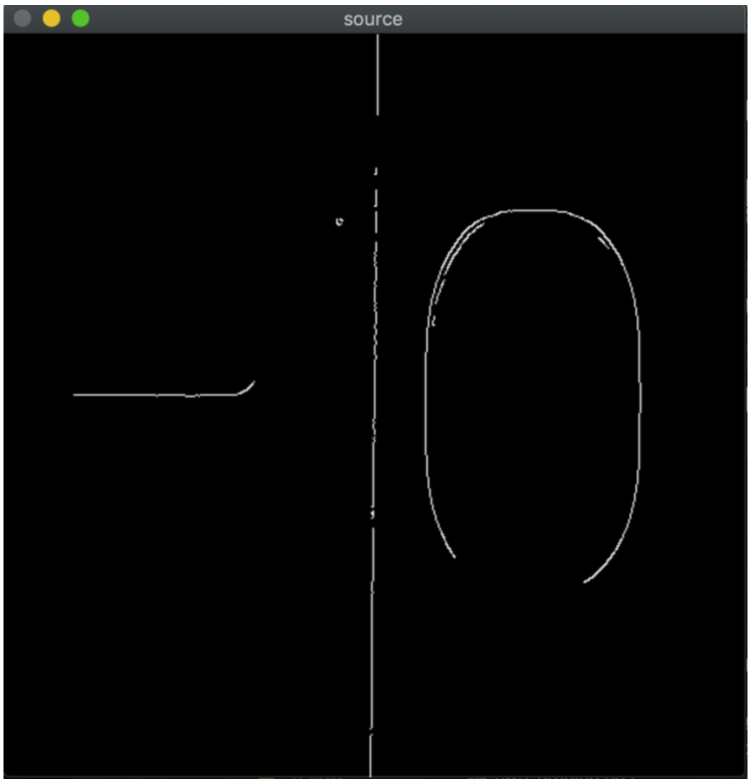


Figure 5: `source` variable in a loop in the application

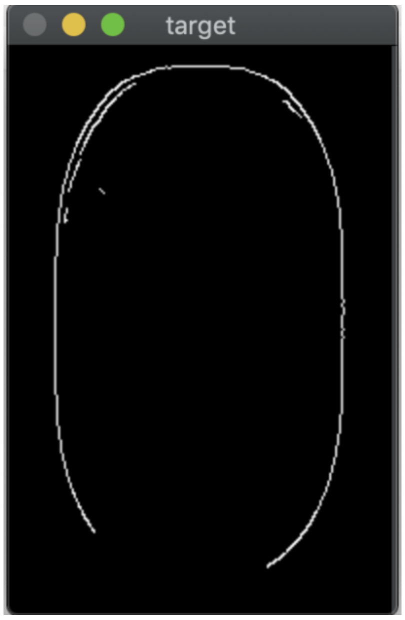


Figure 6: The `target` variable, the edge detection output of the object that the application is tracking

4 Parallelisation

If we carefully observe the algorithm, the loss function iterates over all locations and calculates the loss function. We planned to parallelise this part of the code.

We first started off with OpenACC, a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. By supplying some pragmas over the code that we want to parallelise to the `pgc++` compiler, it will generate intermediate code which would run on a GPU or multiple CPUs for faster performance. In our case, we wanted to parallelise the double `for` loop for calculating the loss at each pixel and storing the coordinates of the pixel with the lowest loss. The reduction problems are well discussed in the OpenACC documentation but our task was to also store the location at the min reduction. After searching through web, we could not find proper solution to this problem hence we are stuck there.

We parallelly also started to write OpenMPI code. OpenMPI is a message passing interface which is helpful in breaking down the serial problem into multiple parallel problems and allowing them to communicate. In our problem, we adopted the master slave architecture. Figure 7 talks about the architecture of our application in OpenMPI setting.

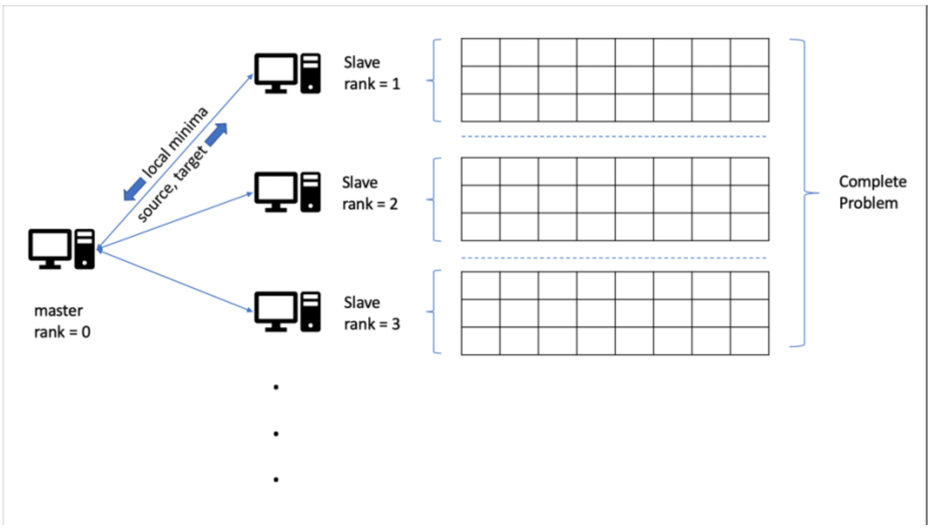


Figure 7: The Master Slave architecture of our application in OpenMPI setting

As discussed, our first process is always a master with rank 0. Every other process are identified as slaves. The communication among them happens as following. The master runs the code to take the inputs from the user and send them to all slaves. The problem of finding pixel with minimum loss is broken down into n sub problems where n is the number of slaves. Each slave then performs the same operation for calculating the loss at each pixel in their sub problem and sends the local minimum position and the loss to the master. Upon receiving all the local minimums from the slaves, the master picks the lowest of them which is the global minimum and identifies the position of the object in that frame. He then communicates this information to the slaves who use that information in preparing the target for their next process. This process in iterated over all the frames and the output video is populated frame by frame.

5 Results

We ran our serial code and the MPI code with multiple workers on a sample video file which contains 107 frames. According to the video dynamics, we choose the upper bound threshold for canny to be 200 and lower bound threshold to be 150. We resized each frame of the image to a 512x512 to evaluate our results faster. Below are the timings for different settings of run environment.

Type	Time per frame in seconds
Serial Code	4.04
MPI 1 - Slave	0.35
MPI - 3 Slaves	0.25
MPI - 5 Slaves	0.26

As we observe, we were able to bring down the execution time of the serial code by over 90%. However, if you observe, after the threshold of 3 slaves, further addition of slaves did not improve the runtime. We believe it would be because of the number of communications that happen between the master and slaves are directly proportional to the number of slaves. The number of communications that happen per frame is $2s - 1$ where s is the number of slaves.

6 Conclusion

In conclusion, we addressed the object tracking problem and introduced a tool called Object tracker which was built in C++ programming language using OpenCV library. The serial code was then parallelised using OpenMPI constructs and above 90% optimisation was achieved.

7 Future Work

There are multiple optimisation we could do to improve the performance and accuracy of the application. To begin with, every slave in the master-slave architecture performs a bunch of similar tasks. Master can take over that responsibility and perform the task and send the output of the task to the slaves. In or case, each slave performs canny edge detection on every frame separately. Master can do it once per frame and send the output information to the slaves. However there has to be a way to pass complex information enclosed in custom data structures which has no pre-built support in MPI library. Hence we have to create an custom data type to pass such information.

Instead of searching through all possible locations in the following frame, we start from the object location in the previous frame and cover a certain parameterised part of the image with the assumption that the image would not have travelled a lot of distance from its previous position in a single frame. This could bring down the computation exponentially.

Our current test was a simple use case, we would like to test multiple videos with variety of objects to see how the algorithm performs and improve the algorithm as required.

Our application is limited to tracking objects that do not change their shape or orientation throughout the video. We would like to work on it and incorporate change in shape and orientation by considering all possible combinations of the same. This of course would explode the computation time exponentially, however this is also another possibility to introduce more parallelisation and optimisation.

In addition to OpenMPI, visit back the OpenACC code and figure out the reduction problem that we were facing and implement the same program with OpenACC directives and compare the results from all environments.

References

https://en.wikipedia.org/wiki/Edge_detection

https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123