**Technical Report: YOLO Object Detection Using Threading and ProcessPoolExecutor in Python**

**1. Introduction**

This project implements a real-time object detection system based on YOLO (using the yolov8n.pt model from the [ultralytics](https://github.com/ultralytics/ultralytics) library) on an input video stream, which can be either a webcam or a video file. The primary goal is to execute object detection in real time by leveraging both Python threading and multiprocessing, thereby improving performance over a sequential version.

Features:

* **Flexible Input Source:** The system supports both webcam streams and video files.
* **Selective Class Detection:** Users can choose to detect only specific classes (e.g., car, person) or run detection on all classes available in the model.
* **Concurrency via Threading and Multiprocessing:** The application uses two separate threads for reading and displaying frames as these are I/O-bound instructions, while the inference task is offloaded to a separate process, as it’s a CPU-bound task, using a ProcessPoolExecutor.

**2. Architecture and Code Structure**

The code is organized into several functions, each responsible for a specific partt of the video processing process:

**2.1. Model Initialization**

* **init\_inference\_worker()**  
  This function is used as an initializer in the ProcessPoolExecutor to load the YOLO model (yolov8n.pt). By initializing the model once per worker process we avoid repetitive and expensive reloads during inference.

**2.2. Inference**

* **inference\_worker(frame, selected\_classes, all\_classes)**  
  This function receives a frame and runs the YOLO model to perform predictions. For each result:
  + It iterates over the bounding boxes and verifies that the confidence score exceeds a threshold (0.4 in this case).
  + It extracts the bounding box coordinates and the corresponding class.
  + It filters detections: if all\_classes is set to False, only the detections corresponding to the classes in selected\_classes are retained.

The function returns a tuple containing the frame and a list of detections (each detection includes the coordinates, class name, and confidence score).

**2.3. Generating Class Colors**

* **generate\_class\_colors(class\_names)**  
  This function generates a dictionary that maps each provided class name to a random color. These colors are later used to draw bounding boxes for different detected classes in a visually distinct way.

**2.4. Reading Frames**

* **read\_frames(cap, frame\_queue, stop\_event)**  
  This function continuously reads frames from the video source (cap). It calculates the sleep time based on the video's FPS to pace the frame reading process. Each frame is then put into a thread-safe queue (frame\_queue) for further processing.

**2.5. Displaying Frames**

* **display\_frames(display\_queue, class\_colors, stop\_event)**  
  This function aims to display processed frames and retrieves frames (and their detections) from the display\_queue. For each detection, it draws a bounding box and overlays the class name along with its confidence score on the frame. The display is managed via OpenCV, and the window can be closed by pressing the q key.

**2.6. Main Function and Concurrency Management**

* **main()**  
  The main function orchestrates the entire workflow:
  + It initializes the video source (either a file or a webcam) and checks its accessibility.
  + Two queues are created: frame\_queue for storing raw frames and display\_queue for storing frames with detections.
  + A stop\_event is created to signal threads to stop (either when the video ends or when the user requests termination).
  + Two threads are launched:
    - **Reader Thread:** Executes read\_frames to continuously fetch frames.
    - **Display Thread:** Executes display\_frames to handle rendering of the frames.
  + In the main loop, frames are retrieved from the frame\_queue and passed to a worker process via ProcessPoolExecutor for inference. The resulting frame and detections are then placed in the display\_queue for visualization.

Using the ProcessPoolExecutor for inference allows the application to bypass Python’s Global Interpreter Lock (GIL), thereby effectively handling the CPU/GPU-bound inference workload in a separate process.

**3. Performance Analysis**

The application’s performance was evaluated by comparing the sequential (SEQ) and parallel (PAR) versions using various video inputs:

* **Input video 640x360 @ 24 FPS:**  
  SEQ: 11.68 FPS, PAR: 17.5 FPS (≈1.5× speedup)
* **Input video 640x360 @ 30 FPS:**  
  SEQ: 13.83 FPS, PAR: 19.81 FPS (≈1.43× speedup)
* **Input video 1280x720 @ 30 FPS:**  
  SEQ: 13.27 FPS, PAR: 15.26 FPS (≈1.14× speedup)
* **Input video 1920x1080 @ 24 FPS:**  
  SEQ: 7.2 FPS, PAR: 9.8 FPS (≈1.36× speedup)
* **Webcam:**  
  SEQ: 10.9 FPS, PAR: 15.41 FPS (≈1.41× speedup)

The parallel approach shows an improvement in throughput, especially for lower- to medium-resolution video inputs.

**4. Technical Considerations and Potential Improvements**

**4.1. Advantages of the Current Architecture**

* **Separation of I/O and CPU-bound Tasks:**  
  The use of separate threads for reading and displaying frames efficiently handles I/O operations, while the inference, being CPU-bound or GPU-bound, is managed in a separate process. This separation helps overcome limitations imposed by the Python GIL.

**4.2. Areas for Potential Improvement**

* **Configurability:**  
  Making parameters such as confidence thresholds, queue sizes, and class selection configurable (via a configuration file or command-line parameters) would enhance the system’s flexibility and adaptability to different use cases.
* **Inference Optimization:**  
  Leveraging hardware accelerators (like Tensor Processing Units) along with optimized inference libraries could reduce processing time per frame even further.

**5. Conclusion**

The provided code represents an effective implementation of a real-time object detection system using YOLO, which combines threading and multiprocessing to separate I/O operations (frame reading and displaying) from computationally intensive inference. The measured FPS improvements demonstrate benefits over a sequential approach.