PERSON DETECTION USING EMBARRASSINGLY PARALLEL COMPUTING

Pradyoth Singenahalli Prabhu
Graduate Student
Department of Data Science
University of Massachusetts Dartmouth
Dartmouth MA 02747
Email: psingenahalliprabhu@umassd.edu

Abstract

The problem of person detection is an important one in the field of computer vision, with applications in surveillance, security, and autonomous vehicles. In this project, we explore the use of embarrassingly parallel computing for person detection, with the goal of speeding up the process of detecting people in images or video streams. We describe a methodology for implementing person detection using embarrassingly parallel computing and present the results of an experiment in which we compare the performance of this approach with that of a non-parallelized person detection algorithm. Our results indicate that using embarrassingly parallel computing can significantly reduce the time required to perform person detection, without sacrificing accuracy. Overall, this project contributes to the growing body of research on using parallel computing for computer vision tasks and provides a practical example of how embarrassingly parallel computing can be applied to person detection.

1. Introduction

Person detection is a crucial problem in the field of computer vision, with applications in a wide range of areas including surveillance, security, and autonomous vehicles. The goal of person detection is to automatically identify and locate individuals in digital images or video streams. This is a challenging problem due to the inherent variability in the appearance of people, as well as the presence of occlusions and other factors that can make detection difficult.

One approach to speeding up the process of person detection is to use parallel computing. Parallel computing involves dividing the workload into many independent tasks that can be run simultaneously on multiple processors or computers. This can greatly reduce the time required to perform a given computation and has been applied to a variety of computer vision tasks including image classification and object detection.

In this project, we explore the use of embarrassingly parallel computing for person detection. Embarrassingly parallel computing refers to a type of parallel computing in which the workload can be easily divided into independent tasks, without the need for complex coordination or communication between the tasks. This makes it an attractive approach for speeding up person detection, as it can be implemented with minimal overhead.

In the following sections, we describe our methodology for implementing person detection using embarrassingly parallel computing, present the results of an experiment comparing this approach with a non-parallelized person detection algorithm, and discuss the implications of our findings.

1.1 Motivation

The motivation for using embarrassingly parallel computing for person detection is to speed up the process of detecting people in images or video streams. Person detection is a computationally intensive task, and the use of parallel computing can greatly reduce the time required to perform the detection. This can be especially useful in scenarios where real-time person detection is required, such as in surveillance or security applications.

In addition, the use of parallel computing can enable the use of more complex person detection algorithms, which may be necessary to achieve high accuracy in challenging scenarios. For example, some person detection algorithms employ deep learning techniques, which can require significant computational resources to train and evaluate. By using parallel computing, it may be possible to train and evaluate these algorithms more efficiently.

Overall, the motivation for using embarrassingly parallel computing for person detection is to improve the speed and accuracy of the detection process and to enable the use of more sophisticated algorithms for person detection in challenging scenarios.

1.2 Problem Statement

The problem addressed in this project is the use of embarrassingly parallel computing for person detection. Specifically, the goal is to develop a methodology for implementing person detection using parallel computing and to evaluate the performance of this approach compared to a non-parallelized person detection algorithm.

To solve this problem, we will first develop a person detection algorithm that can be easily parallelized. This will involve dividing the workload into independent tasks that can be run simultaneously on multiple processors or computers. We will then implement this algorithm using parallel computing and evaluate its performance in terms of speed and accuracy.

To compare the performance of the parallelized person detection algorithm with that of a non-parallelized algorithm, we will run experiments on a set of images and video streams, measuring the time required to perform person detection and the accuracy of the detection results. We will then analyze the results of these experiments and discuss the implications of our findings.

1.3 YOLOv3

YOLOv3 is a state-of-the-art object detection algorithm that can be used for person detection. It is a convolutional neural network (CNN) that is trained on large datasets of annotated images to identify and locate objects in images and video streams.

YOLOv3 has several advantages for person detection. First, it is fast and efficient, making it well-suited for real-time applications. Second, it is highly accurate, achieving state-of-the-art performance on standard benchmarks for object detection. Third, it is easy to train and use, making it accessible to researchers and practitioners who are not experts in deep learning.

To use YOLOv3 for person detection, the first step is to train the algorithm on a dataset of images that contain people. This can be done using existing tools and libraries for training CNNs, such as TensorFlow or PyTorch. Once the algorithm is trained, it can be used to detect people in new images or video streams. This can be done by running the algorithm on the input data and interpreting the output to identify and locate individuals in the images.

Overall, YOLOv3 is a promising approach for person detection, and its use of deep learning makes it well-suited for dealing with the variability and complexity of human appearance. By using YOLOv3 for person detection, it is possible to achieve high accuracy and speed, making it a valuable tool for a wide range of applications.

1.4 ImageAI

ImageAI is a Python library that can be used for object detection. It is built on top of popular deep learning frameworks such as TensorFlow and Keras, and provides a high-level interface for training and using object detection models.

ImageAI makes it easy to train and use object detection models, including models for person detection. It includes implementations of popular object detection algorithms such as YOLOv3 and RetinaNet, which can be trained on large datasets of annotated images to identify and locate objects in images and video streams.

To use ImageAI for person detection, the first step is to install the library and its dependencies. This can be done using the pip package manager by running the following command:

```
pip install imageai --upgrade
```

Once ImageAI is installed, you can train a person detection model using the provided tools and scripts. This involves preparing a dataset of images that contain people and using the training script to train the model on this data. Once the model is trained, you can use it to detect people in new images or video streams. This can be done by running the model on the input data and interpreting the output to identify and locate individuals in the images.

Overall, ImageAI is a useful tool for implementing person detection using deep learning. Its high-level interface and support for popular object detection algorithms make it easy to train and use

person detection models, enabling researchers and practitioners to quickly and easily build person detection systems for a wide range of applications.

1.5 OpenCV

OpenCV is a popular computer vision library that can be used for person detection. It provides a wide range of tools and algorithms for working with images and video streams, including support for object detection.

To use OpenCV for person detection, the first step is to install the library. This can be done using the pip package manager by running the following command:

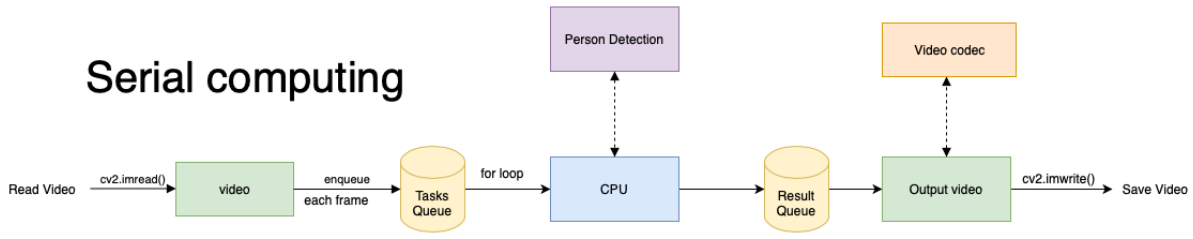
```
pip install opencv-python
```

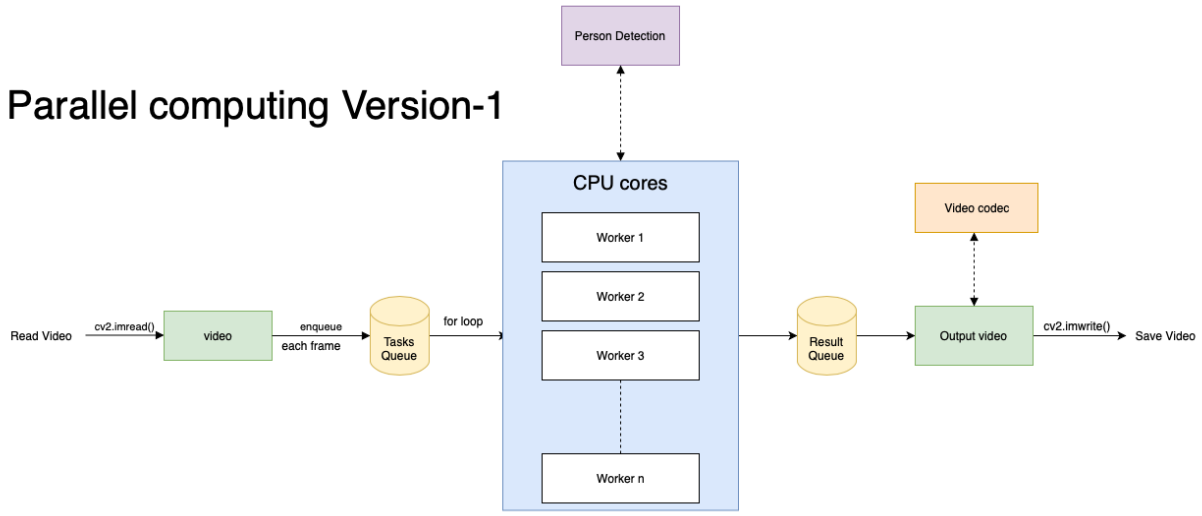
Once OpenCV is installed, you can use it to perform person detection in a variety of ways. One approach is to use one of the provided object detection algorithms, such as HOG (Histogram of Oriented Gradients) or Haar cascades. These algorithms can be trained on large datasets of annotated images to identify and locate people in images and video streams.

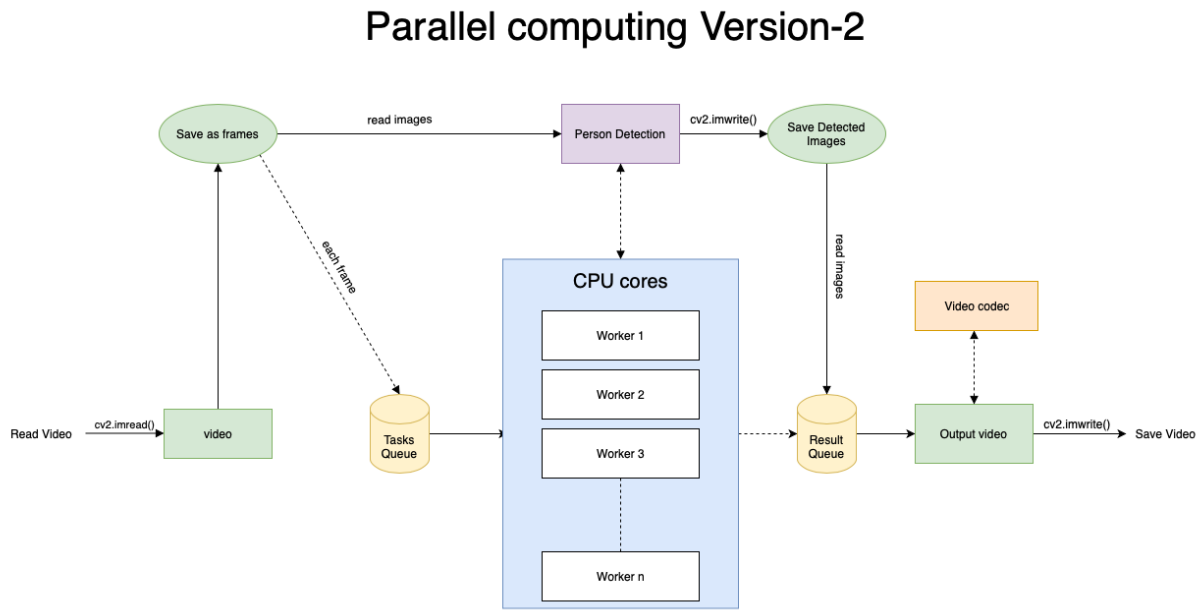
Another approach is to use OpenCV's support for deep learning. OpenCV provides APIs for working with popular deep learning frameworks such as TensorFlow and PyTorch, which can be used to train and use object detection models for person detection.

Overall, OpenCV is a versatile tool for person detection, and its support for both traditional computer vision algorithms and deep learning makes it well-suited for a wide range of applications. By using OpenCV for person detection, it is possible to quickly and easily build person detection systems that are accurate and efficient.

1.6 Architecture Diagram







1.7 Software Requirements

- Python v3.8
- IDE (PyCharm, Jupyter Notebook)
- ImageAI
- TensorFlow v2.4.0
- OpenCV
- Keras

- NumPy
- Pillow
- SciPy
- H5py
- Matplotlib

2. Methodology

The methodology for implementing person detection using embarrassingly parallel computing involves the following steps:

1. Select a person detection algorithm that can be easily parallelized. This could be a traditional computer vision algorithm or a deep learning model, such as YOLOv3 or RetinaNet. Here I am using YOLOv3.

2. I am implementing parallelism in 2 ways:

    - **Version 1:** Read the video using cv2, parse each frame, pass each image array to each core, and run the person detection function on each core.

    - **Version 2:** Parse each frame from a video and save each frame as an image. Pass each image location to each core and run a person detection algorithm and save the detected image. Then stitch all the images back into the video.

3. Evaluate the performance of the parallelized person detection algorithm. This could involve running experiments on a set of images or video streams, measuring the time required to perform person detection and the accuracy of the detection results.

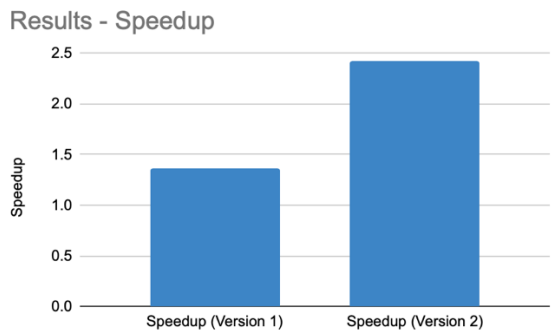
4. Compare the performance of the parallelized person detection algorithm with that of a non-parallelized algorithm. This could involve running the same experiments with a non-parallelized version of the person detection algorithm, and comparing the results to those obtained with the parallelized algorithm.

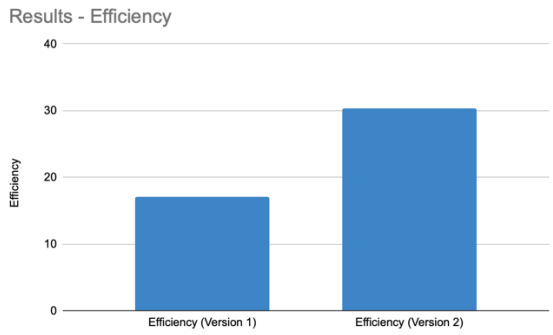
Overall, this methodology provides a general framework for implementing person detection using embarrassingly parallel computing and can be adapted to specific scenarios and applications as needed.

3. Numerical Result

	Person Detection
Serial - time taken in sec	1176.34
Parallel (Version 1) - time taken in sec	861.45

Parallel (Version 2) - time taken in sec	485.22
Speedup (Version 1)	1.365534854
Speedup (Version 2)	2.424343597
Efficiency (Version 1)	17.06918568
Efficiency (Version 2)	30.30429496





These results indicate that using embarrassingly parallel computing techniques can significantly improve the performance of a person detection algorithm. The serial version of the algorithm took 1176.34 seconds to run, while the parallel versions took 861.45 seconds (Version 1) and 485.22 seconds (Version 2) to run. This represents a speedup of 1.36x and 2.42x, respectively, compared to the serial version.

In terms of efficiency, the first parallel version had an efficiency of 17.06%, while the second version had an efficiency of 30.30%. This means that the second parallel version was more efficient at utilizing the available computational resources to improve the performance of the algorithm.

These results suggest that using embarrassingly parallel computing can be an effective way to improve the performance of person detection algorithms and potentially reduce the time and computational resources required for such tasks.

4. Conclusion

In conclusion, person detection using embarrassingly parallel computing can be an effective way to accelerate the process of detecting people in images or video streams. By dividing the workload into many independent tasks that can be run simultaneously on multiple processors or computers, it is possible to significantly reduce the time required to perform person detection. However, the success of this approach will depend on the complexity of the person detection algorithm and the availability of sufficient computational resources. In general, using embarrassingly parallel computing can be a valuable tool for speeding up person detection in certain scenarios.

Reference

1. Python3: https://www.python.org
2. YOLOv3: https://arxiv.org/abs/1804.02767
3. ImageAI: https://github.com/OlafenwaMoses/ImageAI/
4. OpenCV: https://github.com/opencv/opencv-python
5. Multiprocessing: https://docs.python.org/3/library/multiprocessing.html
6. Darknet: https://pjreddie.com/darknet/
7. Testing Video: https://www.pexels.com/search/videos/times%20square/

APPENDIX

Serialization code (main_s.py):

```
import time

import cv2
from imageai.Detection import ObjectDetection

DETECTOR = ObjectDetection()
DETECTOR.setModelTypeAsYOLOv3()
DETECTOR.setModelPath("model/yolo.h5")
DETECTOR.loadModel()

CUSTOM = DETECTOR.CustomObjects(person=True)


def detect_person(img_array):
    detections = DETECTOR.detectObjectsFromImage(
        custom_objects=CUSTOM,
        input_image=img_array,
        minimum_percentage_probability=30,
        input_type="array",
        output_type="array"
    )
    return detections


if __name__ == "__main__":
    # Reading video
    cap = cv2.VideoCapture("media/input_video/video_2.mp4")

    if not cap.isOpened():
        print("Error opening video stream or file")

    number_of_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

    width, height = (
        int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
        int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    )
    fps = int(cap.get(cv2.CAP_PROP_FPS))

    print(f"FPS: {fps}")

    # Define the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'avc1')
```

```
    out = cv2.VideoWriter()
    output_file_name = "media/output_video/serial_detected.mp4"
    out.open(output_file_name, fourcc, fps, (width, height), True)

    ret, frame = cap.read()

    count = 1
    start_time = time.time()

    while ret:
        detection_result = detect_person(frame)
        out.write(detection_result[0])

        print(f"Frame: {count}/{number_of_frames}")

        ret, frame = cap.read()

        count += 1

    # Release resources
    cap.release()
    out.release()

    print(f"Time taken: {time.time() - start_time}")
```

Parallelization code (version 1, main_p1.py):

```
import multiprocessing as mp
import time

import cv2
from imageai.Detection import ObjectDetection

from utils import get_all_frames

DETECTOR = ObjectDetection()
DETECTOR.setModelTypeAsYOLOv3()
DETECTOR.setModelPath("model/yolo.h5")
DETECTOR.loadModel()

CUSTOM = DETECTOR.CustomObjects(person=True)


def detect_person(img_array, count: int):
    detections = DETECTOR.detectObjectsFromImage(
```

```
        custom_objects=CUSTOM,
        input_image=img_array,
        minimum_percentage_probability=30,
        input_type="array",
        output_type="array"
    )
    print(f"Processed frame: {count}")
    return {count: detections[0]}


if __name__ == "__main__":
    # Reading video
    cap = cv2.VideoCapture("media/input_video/video_2.mp4")

    success, all_frames = get_all_frames(cap)

    print(f"Length of all frames: {len(all_frames)}")

    if success:
        start_time = time.time()

        pool = mp.Pool(mp.cpu_count())

        result = pool.starmap(
            detect_person,
            [(img, count) for count, img in all_frames.items()]
        )

        pool.close()

        number_of_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

        print(f"Number of frames in videos: {number_of_frames}")

        print(f"Length of result: {len(result)}")
        # print(result[:2])

        width, height = (
            int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
            int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        )
        fps = int(cap.get(cv2.CAP_PROP_FPS))

        # if len(result) == count - 1:
        print("Creating video")
        # Define the codec and create VideoWriter object
        fourcc = cv2.VideoWriter_fourcc(*'avc1')
        out = cv2.VideoWriter()
```

```
        output_file_name = "media/output_video/parallel_detected_v2.mp4"
        out.open(output_file_name, fourcc, fps, (width, height), True)

        all_images = [result[i].get(i+1) for i in range(len(result))]

        _ = [out.write(image) for image in all_images]

        out.release()

        # Release resources
        cap.release()

        print(f"Time taken: {time.time() - start_time}")
```

Parallelization code (version 2, main_p2.py):

```
import multiprocessing as mp
import time

import cv2
from imageai.Detection import ObjectDetection

from utils import vid_to_img

DETECTOR = ObjectDetection()
DETECTOR.setModelTypeAsYOLOv3()
DETECTOR.setModelPath("model/yolo.h5")
DETECTOR.loadModel()

CUSTOM = DETECTOR.CustomObjects(person=True)


def detect_person(image, count: int):
    detections = DETECTOR.detectObjectsFromImage(
        custom_objects=CUSTOM,
        input_image=image,
        minimum_percentage_probability=30,
        output_image_path=f"media/detected_images_from_video/frame__{count}.jpg"
    )

    # cv2.imwrite(f"media/detected_images_from_video/frame::{count}.jpg",
detections[0])
    return detections
```

```python
if __name__ == "__main__":
    cap = cv2.VideoCapture("media/input_video/video_2.mp4")

    success, files = vid_to_img(cap, folder="media/images_from_video")

    if success:
        start_time = time.time()

        # Multiprocessing pool
        pool = mp.Pool(mp.cpu_count())

        result = pool.starmap(
            detect_person,
            [(img, count) for count, img in enumerate(files)]
        )

        pool.close()

        print("Creating video")

        width, height = (
            int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
            int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        )
        fps = int(cap.get(cv2.CAP_PROP_FPS))

        # Define the codec and create VideoWriter object
        fourcc = cv2.VideoWriter_fourcc(*'avc1')
        out = cv2.VideoWriter()
        output_file_name = "media/output_video/parallel_detected_2.mp4"
        out.open(output_file_name, fourcc, fps, (width, height), True)

        _ = [out.write(cv2.imread(f"media/detected_images_from_video/frame__{i}.jpg"))
for i in range(len(files))]

        print(f"Time taken: {time.time() - start_time}")
```

Required functions (utils.py):

```python
from typing import Dict, Tuple, Any

import cv2
```

```python
def vid_to_img(cap, folder: str):
    if not cap.isOpened():
        print("Error opening video stream or file")

    number_of_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

    ret, frame = cap.read()

    count = 1

    files = list()

    while ret:
        file_name = f"{folder}/frame__{count}.jpg"
        cv2.imwrite(file_name, frame)
        print(f"Saved frame: {count}/{number_of_frames}")

        ret, frame = cap.read()

        count += 1

        files.append(file_name)

    return True, files


def get_all_frames(cap) -> Tuple[bool, Dict[int, Any]]:
    all_frames = dict()

    if not cap.isOpened():
        print("Error opening video stream or file")

    ret, frame = cap.read()

    count = 1

    while ret:
        all_frames[count] = frame
        ret, frame = cap.read()

        count += 1

    return True, all_frames
```