universität innsbruck

Leopold–Franzens–University
Austria, Innsbruck

Institute of Computer Science
Distributed and Parallel Systems Group

Distributed Systems Proseminar 2020W

Project 1 - Object recognition

Project Report

Supervisor: Fedor Smirnov, Sashko Ristov

Mathias Thöni, David Freina

Innsbruck
31 January 2021

Abstract

In the Distributed Systems proseminar at the University of Innsbruck our team was tasked with implementing a tool for automatic detection of children and dogs in given input video data using Amazons Web Services and the xAFCL tool. In order to reduce the computation time, the tool should not analyze the entire video, but instead extract frames every half second and then check for differences between the sequential pictures using OpenCV. If the difference was above a chosen threshold, we used AWS Rekognition to identify children and dogs. We decided to use `Node.js` for the implementation, but because there is no official OpenCV package for Node.js we used Python for frame extraction and picture delta detection. The workflow is controlled via xAFCL, which also distributes the workflow over multiple threads for parallelization. All Node.js functions were deployed on AWS Lambda. Because of the dependencies to Python packages - like OpenCV - we used AWS Elastic Container Registries to deploy our Docker images with the Python functions. Input videos and extracted frames are stored on a S3 Bucket. We had to restrain our parallel calls to AWS Rekognition to 50 due to API limitations of AWS. By distributing the workflow on multiple AWS Lambda calls, our tool was able to process multiple video files of different sizes (302.0 KB - 11.9 MB), successfully detect all dogs and children and return a human readable string containing type, time and number of detections in the given videos in a reasonable amount of time (85 seconds).

Contents

1 Introduction

As the use of cameras for automatic home surveillance advances, storing and analysing massive amounts of video data is expensive and time consuming. Therefor as part of our project for the proseminar Distributed Systems of our bachelor program at the University of Innsbruck, we created an automatic video analysing tool using Amazon Web Services. The tool uses AWS Rekognition to find dogs and children in raw video data and gives feedback at which time and how often the recognition yielded a positive result. The workflow is controlled via xAFCL and a corresponding YAML file. xAFCL in return calls Node.js functions deployed on AWS Lambda and Python functions deployed with AWS Elastic Container Registries.

By extracting frames every 0.5 seconds and then analysing them respectively, instead of analysing the entire video, the run time of our workflow is minimized. Furthermore the frame extraction is optimized for multiple input videos using data parallelism. After the completion of the frame extraction, the frame images will be distributed over multiple split folders. These are again used for data parallelism in the analyzing phase, where two images are held against each other and differences are investigated. If the difference is below a given threshold, that means nothing has changed and the pictures are deleted. This method reduces unnecessary recognition calls and improves the efficiency of the Function Choreography (FC). At last for each split folder a object recognition call is instantiated, again using data parallelism.

Because of the high data parallelism of our FC and the delimitation between simultaneous steps, one may improve the computational time needed by deploying the FC on a distributed system in contrast to a single autonomous system.

2 Solution

2.1 Compound functions

In our FC we have used three compound functions, all of type `ParallelFor`. By using them we aim to maximize the parallelization of our program by splitting the data of every step into small pieces.

The first `ParallelFor` section loops over all the given input videos and for each video it calls one `extractFrames` function. The second and third `ParallelFor` use a loop counter given by the first `ParallelFor` which corresponds to the number of folders which were generated by the `extractFrames` functions. For every folder which was generated one `analyzeFrames` and `awsRekognition` is executed.

2.2 FC functions

This section should describe the different dataIns/dataOuts of the FC respectively its functions and their usage. Generally we can note that the parallel compound functions have the same dataIns and dataOuts as their sequential counterparts. The only differences is that the dataIns are often split up and the dataOuts are aggregated by the parallel functions. Figure 2.1 shows how the FC is structured and how the functios are dependent on each other.

2.2.1 `getVideoLinks`

The `getVideoLinks` function takes two parameters as input. Both of them are given from a input json file. The first one is the `videoBucketId` which is the name of an S3 bucket where the video files are stored. The second parameter which is only passed through is called `numberOfFramesToAnalyzePerInstance`. We return two parameters called `videoLinks` and `numberOfVideos` from the `getVideoLinks` function. The former is a collection consisting of URL's of the different videos and the latter is the total number of videos in the S3 bucket.

2.2.2 `extractFramesParallel` and `extractFrames`

The dataIns for the `extractFramesParallel` are the `videoLinks` and `numberOfFrames-ToAnalyzePerInstance`. We distribute the `videoLinks` collection so that we call one `extractFrames` per video. The second parameter is replicated over all the executed functions. Every `extractFrames` function extracts a frame every 0.5 seconds and stores

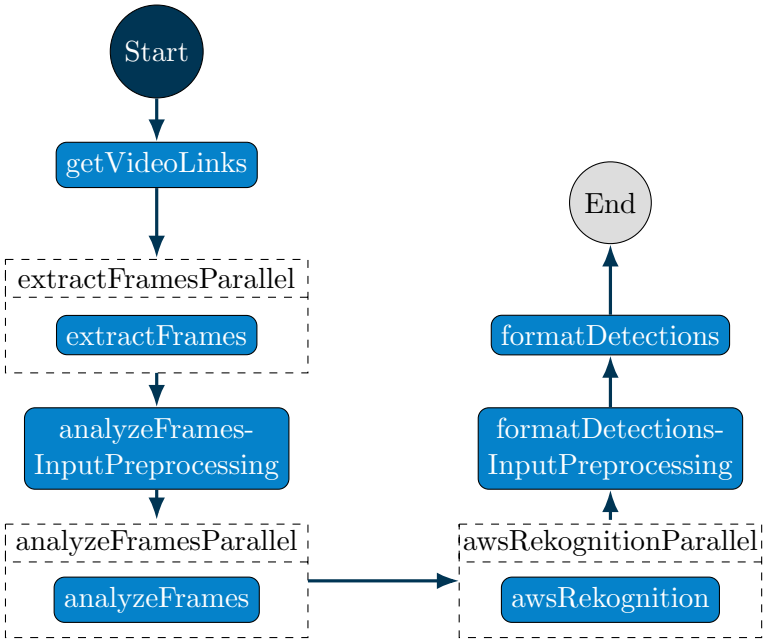

Figure 2.1: A figure of the FC inspired by AFCLToolkit

it in a folder. Everytime the function extracts more than `numberOfFramesToAnalyzePerInstance` it creates a new folder and stores the corresponding frames there. The parameter which is returned is a collection called `extractedFramesSplitFolders` which is then aggregated by the parallel function to a collection of collections.

2.2.3 `analyzeFramesInputPreprocessing`, `analyzeFramesParallel` and `analyzeFrames`

The auxiliary function `analyzeFramesInputPreprocessing` takes the dataOut from the `extractedFramesParallel` function and flattens it to a normal collection. Furthermore it counts the number of folders which is return as `extractedFramesSplitFoldersNumber`. The output is then used by the `analyzeFramesParallel` to distribute one folder per `analyzeFrames` instance. Every `analyzeFrames` compares the frames of the given folder by using a scikit function which returns a number between $0 <$ index < 1 which is used to decide whether a image is different enough to keep or too similar and therefor deleted. The function then returns the `analyzeFramesSplitFolder` which is aggregated by the parallel function to a collection. Additionally the `extractedFramesSplitFoldersNumber` is passed through.

2.2.4 `awsRekognitionParallel` and `awsRekognition`

The function `awsRekognitionParallel` takes to aggregated collection as input and distributes it to `awsRekognition` instances. Every instance uses AWS Rekognition to detect

labels for the given frames and then returns a `detections` object containing wheter a dog or a human was found. These objects are aggregated by the parallel compound function to a collection.

2.2.5 `formatDetectionsInputPreprocessing` and `formatDetections`

The aggregated collection from `awsRekognitionParallel` is then preprocessed by the auxiliary function `formatDetectionsInputPreprocessing` which takes the collection as input and returns a single object containing all objects from the collection. The object is then used as input for the `formatDetections` which returns a formatted string as output.

2.3 Complexity and parallelization opportunities

The FC is pretty linear structured which should really lower the overall complexity. Apart from the `ParallelFor` sections there are no simultaneous executed functions which should lead to a easily comprehensible FC. We think that we used most of the parallelization opportunities by parallelizing every compute intensive part of the FC while keeping functions which would not profit from parallelization or would simply not be parallelizable sequential. For example the preprocessing functions need to take an aggregated input which is then transformed in some way. There would be absolutely no benefit from parallelizing such an operation because it would just overcomplicate the overall FC.

3 Challenges

This section should inform about various challenges and problems we faced during the development of the FC and its functions.

3.1 Amazon Rekognition API limit

The main challenge of our implementation was the limitations of the maximum function calls to AWS Rekognition. Because of the size of a few video files, a lot of pictures had to be analyzed by Amazons object recognition software. But if too many calls are made in a small period of time, AWS refuses future calls and needs a cool down. This was reinforced by the fact that multiple `awsRekognition` instances are executed in parallel. In order to circumvent this, we limited the sequential instances to 50 using the `concurrency` constraint.

3.2 xAFCL limitations and problems

3.2.1 Passing through variables

Another problem was that we needed specific variables multiple times in different functions of our workflow. Because there is no such thing as global variables, we had to pass arguments through multiple functions in order to reuse them. The definition of global variables would be a nice addition to the xAFCL tool.

3.2.2 Missing documentation and features

We found it difficult to find all possible attributes of xAFCL functions (e.g. `passing`, `replicate`, `concurrency`) in the documentation. It would be good to extend it with these entries and some examples.

The reason we had to use preprocessing functions was that we could not find out if the xAFCL tool can flatten multiple Arrays to one Array when aggregating them after using `parallelFor`. That would be a nice addition to the xAFCL tool if it is really missing currently.

Furthermore the need of a loop counter for every `parallelFor` section made the workflow more complicated. In our opinion an additional `parallelForEach` would simplify the FC dramatically.

3.2.3 Bug?

Additionally we came across a possible bug in the xAFCL tool. After taking a break from developing the project we hadn't run our function for three weeks which lead to a Exception in the xAFCL tool because AWS Lambda first had to initialize the functions before they could be executed. Sadly we were not able to save an output containing said exception.

3.3 Programming language

While we are confident with basic JavaScript, we did not have any experience with Node.js. It took us quiet some time to read the documentations and get to know the framework. Furthermore there is no official OpenCV implementation in Node.js. That is why we used Python for splitting the videos and finding deltas between pictures.

4 Evaluation

This part of the report should provide evidence that the previously formulated project objectives are achieved with your implementation of the FC.

4.1 Methodology

Our main performance comparison metric was the execution time reported by the xAFCL tool. As mentioned before we had a variable called `numberOfFramesToAnalyzePerInstance` which was responsible for certain parallelization decisions made by the functions. Initially we started testing with a value of 10 which meant that the `extractFrames` function would place every 10 extracted frames in separated folders. Therefor every `analyzeFrames` and `awsRekognition` instance worked on one of those split folders giving us a huge amount of parallel executed instances. We then proceeded to try different sizes of `numberOfFramesToAnalyzePerInstance`. The observed execution times can be found in table 4.1. For every subsequent test run we emptied the S3 bucket leaving only the videos and did one warmup run for the next executions to prevent a cold start. Every instance of `analyzeFrames` had 4GB RAM and 5 minutes available, every instance of `extractFrames` had 2GB RAM and 5 minutes available and all other functions had 128MB RAM and 30 seconds available before they were terminated by AWS.

4.2 Results

Quickly it became apparent that increasing the number of frames analyzed per instance gave us much better execution time. We were able to reduce the execution time to about two-thirds by using 100 frames per instance instead of only 5. But we have to consider that increasing the frames per instance reduces our overall parallelism which we can see if we increase it further to 150 where we become much slower. The main takeaway from this is that the sweet spot for our frames per instance probably lies somewhere between 100 and 150. Using smaller amounts than 100 probably introduces to much overhead in terms of calling the functions on AWS Lambda and getting the return values that we are even slower.

Unfortunately we were not able to execute the workflow sequentially. However table 4.1 shows that the time needed increases when assigning to many frames to one instance. Therefor the sequential time would most likely worsen compared to the parallel execution on multiple instances.

frames per instance	execution time (ms)
5	127,150
10	102,288
50	86,985
100	84,404
150	106,326

Table 4.1: Observed execution times by varying the `numberOfFramesToAnalyzePerInstance` variable. All times were measured three times and averaged.

5 Conclusion (and future work)

Our implementation offers high efficiency due to the highly parallelized workflow of our FC and is able to simultaneously process multiple videos. Using the `numberOfFramesToAnalyzePerInstance` variable one would be able to further adjust the number of frames each instance has to process according to the underlying hardware infrastructure. As seen in the previous chapter doing so will positively affect the execution time. Because of the dependencies between the extraction, analyzation and recognition phase, we decided to synchronize all instances after each step in order to guarantee a correct result. As detection is done by Amazons service we do not have much room for improvements of correctness, except changing the chosen confidence and age thresholds.

5.1 Future work

It may be more efficient to use a producer-consumer pattern to instantly feed the extracted frames to a analyzing instance instead of synchronizing after each main step. Vice versa with the analysing and object recognition phase.

5.2 Recommendations for the PS in the future

The topic of our project is really interesting and has many practical use cases. It was mind-blowing what you could do with AWS and how light weighted such a otherwise massive project is when using xAFCL and AWS. We think that giving small insights in competing standards like IBM Watson and Google AI would be a good contrast to only using AWS. Of course the time schedule of the PS is tight, but just showing some of the basic features and differences between these tools would be amazing.

In addition we think that using AWS data base systems would have been interesting, but of course that is project dependent.

6 Appendix A

6.1 Configuration

6.1.1 credentials.properties

This file is needed by the xAFCL tool to authenticate against AWS. It should be provided in the same directory as the xAFCL.jar. Filling in the required values is mandatory.

6.1.2 objectRecognitionInput.json

The objectRecognitionInput.json file has two parameters which are configurable. First is the `videoBucketId` which should be the name of your S3 bucket which contains the given videos. The second parameter is the `numberOfFramesToAnalyzePerInstance` which configures how many frames one instance receives for analysis or recognition.

6.2 Deployment

6.2.1 Node.js functions

The Node.js functions should simply be uploaded as seperated AWS Lambda functions. All of them should have 128MB memory and 30 seconds execution time.

6.2.2 Python functions

The Python functions are deployed as a docker image due to their dependencies to other packages like OpenCV. We decided to use AWS ECR as our container registry because we could then specify those images directly when creating the AWS Lambda functions. To build the images using the provided Dockerfile's and push them to the AWS ECR we adhered to the push commands which can be viewed directly from the AWS ECR console. For a write up of those commands see figure 6.1.

Because those two functions do much more computation than the rest of the functions we also had to increase their RAM and time. `analyzeFrames` needs 4GB RAM for 100+ frames per instance and `extractFrames` is satisfied with 2GB RAM. For both of the functions we increased to time limit to 5 minutes.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS
    --password-stdin $ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com

docker build -t $IMAGE_NAME .

docker tag $IMAGE_NAME:latest $ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/
    $IMAGE_NAME:latest

docker push $ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/$IMAGE_NAME:latest
```

ecr_push_commands.sh

Figure 6.1: AWS ECR push commands