2017

Should We Place the License Plate Tracking System in the Cloud?

Razib Iqbal
Missouri State University

Matthew Kenney
MSU Undergraduate

Jamil Saquer
Missouri State University

Recommended Citation

Iqbal, Razib, Matthew Kenney, and Jamil Saquer. "Should We Place the License Plate Tracking System in the Cloud?." In Proceedings of the 14th International Joint Conference on e-Business and Telecommunications, pp. 77-80. 2017.

Should We Place the License Plate Tracking System in the Cloud?

Razib Iqbal, Matthew Kenney and Jamil Saquer
Department of Computer Science, Missouri State University, Springfield, Missouri, U.S.A.

Keywords: Video Surveillance, Parallel Processing, Apache Storm, Microsoft Azure, Google Cloud, HDInsight.

Abstract: We developed a software system to extract and track vehicle license plate numbers from real-time surveillance cameras and crowd sourced video streams. The system can also calculate the probable routes of a vehicle over a range of dates based on the geographical coordinates. In this paper, we present both of our linear and parallel processing implementation schemes and analyze the performance based on evaluation results. Our results show that while cloud based parallel processing can address the scalability needs, performance outweighs the cost only when the real-time streaming data becomes increasingly large.

1 INTRODUCTION

The ubiquitous nature of Internet of Things (IoT) and crowd sourcing provide a greater opportunity to city authorities and law enforcement agencies to monitor the state of the urban environment in real-time (Khan, 2015). However, increased data volume and real-time monitoring requirements also increases the data processing and visualization cost and complexity. In this respect, we present a license plate tracking system (LPTS) and compare its performance in various scenarios like linear and parallel processing in local and cloud platforms to analyze the rational of using cloud platforms for real-time processing of big data.

From our literature review, it is evident that license plate detection and recognition has attracted lots of research interests recently. In (Li, 2016), a combination of deep learning and recurrent neural networks are used to recognize the plate and plate characters. Support vector machines is another class of machine learning technique that has been applied (Parasuraman, 2010) to address the problem of real time plate detection by skipping the character segmentation stage. Other recent research initiatives include a detection scheme in low resolution images (Kumar, 2016), license plate localization with Marckov Chain Monte Carlo process (Cao, 2014), and a deep convolutional neural network based method (Jain, 2016). Irrespective of the schemes, improvements in accuracy and speed come with a trade-off in computation power. While lots of data are available to initially train the systems, computationally complex systems might be expensive to deploy and maintain. Our motivation for LPTS is based on the initiatives for human identification and tagging in a multi-camera video surveillance system (Moctezuma, 2015). LPTS not only detects the plate numbers but also can track them in non-overlapping cameras including crowd sourced videos. It extracts the plate numbers from static/crowd sourced video feeds and stores them in a database. LPTS can generate the routes of vehicles on demand and plot them on a visible map based on the previously acquired coordinates.

LPTS uses both classes of big data processing, static and streaming (Eskandari, 2016), where live video streams pass through license plate detection and number extraction module, and then it enables the users to query the license plates and generate their routes over a range of dates. In order to process high volume of static data offline, Hadoop (*hadoop.apache.org*), an open source implementation of MapReduce was introduced (Dean, 2004). However, Hadoop is not suitable for real-time streaming data where batch processing approaches cannot be used. To address this drawback, a number of new frameworks have been proposed (Heinze, 2014). Apache Spark (*spark.apache.org*) implements stream processing as a chain of batch processing at an interval of one second. Therefore, the processing of Spark is slower than pure stream processing. Apache Storm (*storm.apache.org*) was introduced for real-time, distributed, scalable and reliable framework for stream processing. An application in the Storm is called a topology, which consists of a number of

Iqbal, R., Kenney, M. and Saquer, J.
Should We Place the License Plate Tracking System in the Cloud?.
DOI: 10.5220/0006470000770080
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 5: SIGMAP*, pages 77-80
ISBN: 978-989-758-260-8

tasks. We initially implemented the LPTS system to be run linearly at a local server, however, the aforementioned features of LPTS along with heterogeneous video adaptation, data bursts, fluctuating data transfer rates make resource allocation critical. Therefore, we explored the cloud platforms and adapted LPTS to utilize the Apache Storm framework for on-the-fly resource scalability. We deployed our system in both Microsoft Azure and Google Cloud platforms. Based on our evaluation results, we conclude that cloud based parallel processing is only beneficial when the scalability needs and performance outweighs the cost of the deployment.

2 LICENSE PLATE TRACKING SYSTEM (LPTS)

Our LPTS consists of two subsystems - License plate recognition and License plate search. All the operations involved in these subsystems are performed in a pipelined synchronous manner. We utilized OpenCV (*opencv.org*) for frame preprocessing and Openalpr (*github.com/openalpr*) to detect the license plate characters. LPTS accepts video feeds from stationary cameras as well as crowd sourced videos, and adapts it for preprocessing in the license stream module. To reduce the overall computation load, we perform temporal homogeneous video adaptation to achieve a lower frame rate, e.g. 5fps. License plate detection module also processes the frames (resizing and gray scaling) and passes it to a function to find all of the contours in a given frame. Contours can be explained as a set of points bounding a region that has the same color or intensity. We eliminate the bad contours by drawing a rectangle bounding every contour and matching it against a range of expected width and height ratio. For character recognition, we rely on the Openalpr function. We finally save the character outputs (if any), in a database along with its timestamp, coordinates, and picture id of the frame.

In response to a search query, LPTS returns the waypoint objects sorted by the timestamps. The program is written in JavaScript and uses Google API to generate a map of all the routes that a vehicle has taken over a specified time period. The creation of the visual map is broken into four stages: Waypoint object creation, Route creation and addition to wrapper object, Map route creation, and Listener event creation. A waypoint object of a license plate number found in the database consists of a timestamp, latitude and longitude information and a video frame. An array of waypoint objects is then used to create the routes. Route end points are determined based on the difference between a waypoint's time stamp and the time stamp of the waypoint ahead of it. If the difference is greater than a specified amount, e.g. 30minutes, then the current waypoint becomes the last waypoint in that route's array. Therefore, each route is an array of JSON objects and is stored in a wrapper JSON object which holds all of the routes. The routes in the map are represented by colored lines connecting similar colored nodes showing the coordinates and the routes. Each route is given its own separate directions renderer object on the map so that it can be toggled on and off. The renderer object sets the line color and also puts the route on the map. For the markers, since a stop point is seen as the start point of the next route and shares a marker with that route, if an earlier route is toggled off then the next route keeps that marker visible on the map until it is toggled off as well. Stop points are shown in red while the rest of the markers match the route's polyline color. The node may be "clicked" by the user which creates a Bootstrap modal containing a snapshot of the vehicle at the time of video capture. A map is shown in Figure 1 for illustration purpose.

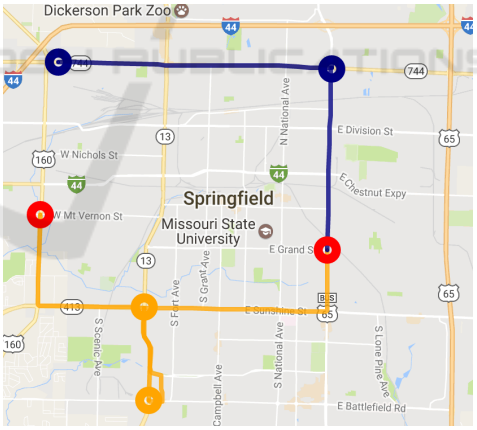

Figure 1: Map generated by LPTS for a vehicle.

3 TOPOLOGY FOR PARALLEL PROCESSING

In this section, we present our scheme for scheduling the executors across the nodes in a Storm cluster. We utilized HDInsight (2016), which is a cloud distribution on Microsoft Azure for big data analysis. HDInsight has provisions to create a storm

cluster in order to implement a topology and submit it to the cluster.

An Apache Storm cluster consists of three distinct types of nodes. The *Nimbus* node performs many responsibilities that are expected from a master node, e.g. distributing the code, assigning tasks, and monitoring failures. Usually, a Storm cluster will have two Nimbus nodes to provide fault tolerance. The *Zookeeper* nodes simply pass messages between the worker nodes and the Nimbus. Zookeeper nodes monitor the health of worker nodes and communicate failures to the Nimbus. The *worker* nodes run a daemon called, Supervisor, which starts and stops tasks while waiting for assignments from the Zookeepers.

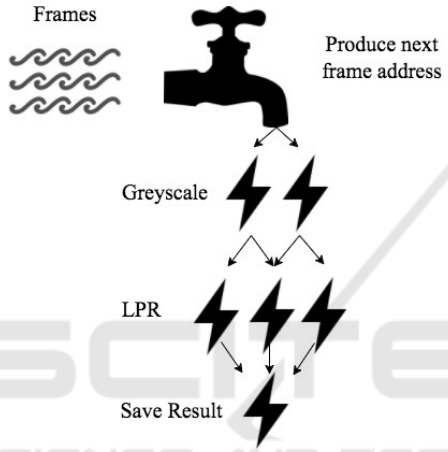

Figure 2: Storm topology for LPTS.

In order to utilize a Storm cluster, we implemented a Storm topology. We used Java for our implementation and applied the Maven build automation to help us with the packaging dependencies. A topology consists of three components - *stream*, *spout*, and *bolts*. Stream is an abstraction of the input to be processed by the cluster. Our stream consists of the video frames that are being stored on a public file server. Stream is then transformed using spout and bolts. Spout starts by performing the initial preparation of the stream by packaging it into separate tuples and emitting the tuples so that they can be processed by the bolts. The spout in our topology repeatedly produces and emits the next frame to be processed by the bolts. Each bolt has specific processes that are to be performed on the tuples they receive. The topology specifies in what order the bolt processes should be performed on the tuples. In our experimental setup, topology has three bolts. First, the greyscale bolt downloads the image using the frame address from the spout. The bolt uses a method provided by OpenCV to produce a greyscaled version of the frame and emits it as a byte array. Next, the byte array is received by the License Plate Recognition (LPR) bolt and the modified Openalpr recognize function is used to retrieve the license plate number from the frame. Finally, if a license plate is detected in the frame, then the results are emitted to the save bolt. The save bolt connects to a cloud Microsoft SQL Server and inserts a record containing the results it received from the LPR bolt. Figure 2 models our specific implementation of the topology.

4 EXPERIMENTAL RESULTS

For parallel processing, we used HDInsight 3.5 with Apache Storm 1.0.1 installed on Ubuntu 16.04. We performed our experiments on a Storm cluster with two Nimbus nodes, three ZooKeeper nodes and a varying number of Supervisor nodes, where each node has four cores (Intel Xeon E5 2.2 Ghz, 8GB ram). When a Supervisor node is created, a script is ran that installs OpenALPR and its dependencies. We compare our Storm cluster with a linear implementation of the LPTS running on two different systems: a local four core machine running Ubuntu 14.04 (Intel i7 3Ghz, 8GB ram), and a cloud four core virtual machine running Ubuntu 16.04 (Intel Xeon E5 2.6 Ghz, 8GB ram) created using the Google Compute Engine (GCE) platform. We analyze the performance based on the average frames processed per second and cost per hour of each cloud deployment.

Table 1: Different platform performance comparison.

Platforms	Time per frame	FPS
Local Linear	145.6 ms	6.88
GCE Linear	167.35 ms	5.97
Azure HDInsight	209.2 ms	19.16

In the test video inputs, around 30% frames had a license plate. In Table 1, we show the average processing time per frame to perform license plate recognition (using single core) along with the total frames per second (FPS) the system achieved (using all four cores for Azure) over a two-hour period. From the results, we can see that the local machine achieved the fastest processing time utilizing a single core but it had a slower overall FPS than the Storm cluster (i.e. Azure HDInsight). The GCE Linear deployment with a single core utilization had a slightly slower processing speed than the Local

Linear deployment which also resulted in a lower FPS. The Azure Storm cluster's single core had the slowest processing time yet had the highest FPS due to our parallel processing scheme. These results demonstrate that given a linear implementation, the processing speed will have a direct impact on the overall speed of the LPTS. However, we can get a higher FPS with parallel processing even though each core might exhibit a slower processing power.

In the previous experiment, a GCE virtual machine costs $0.19 per hour whereas the cost of the Storm cluster with a single Supervisor node is $2.22 per hour. In our next experiment, we looked at the cost of scaling the Storm cluster versus adding more Google cloud virtual machines. We used data points gathered from performing tests on a four node Storm cluster while scaling it by a single node at a time up to seven nodes in the cluster. Similarly, in GCE Linear implementation, we manually added new virtual machines to handle the increased loads. From Figure 3, we can see that at 55 FPS and above, the Storm cluster is a more cost-efficient option than the GCE linear implementation. Therefore, we can conclude that scalability and performance only outweighs the cost when the number of real-time streaming data becomes increasingly large. Otherwise, a local linear implementation and deployment might be sufficient.

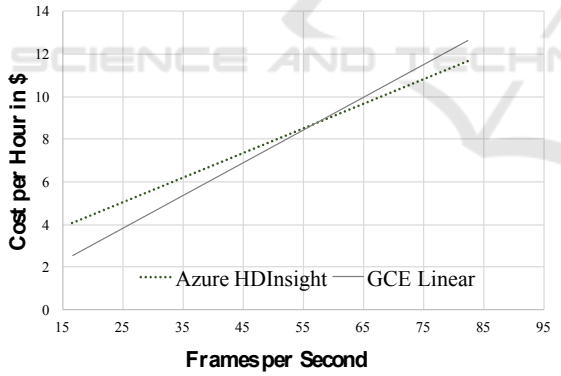

Figure 3: Cost per hour vs Performance comparison of Azure HDInsight and Google Compute Engine.

5 CONCLUSIONS

In this paper, we presented a linear and parallel implementation of our license plate tracking system, and the respective performance results. From the experimental results, we found that – a) Local linear deployment offers the fastest computation time however it is not dynamically scalable compared to the cloud deployment; b) Storm cluster running on the Microsoft Azure platform becomes more cost efficient than the linear implementation on the Google Compute Engine platform only when the work load exceeds a certain threshold. Therefore, parallel computing scheme in Azure makes it the best choice for scalability and higher workloads.

As future work, we plan to parallelize the license plate recognition steps as well as improve the routes prediction process.

REFERENCES

Khan, Z., Anjum, A., Soomro, K. and Tahir, M. A., 2015. Towards cloud based big data analytics for smart future cities. In *Journal of Cloud Computing*, 4(1).

Moctezuma, D., Conde, C., de Diego, I. M., and Cabello, E., 2015. Soft-biometrics evaluation for people re-identification in uncontrolled multi-camera environments. In *EURASIP Journal of Image and Video Processing*, 2015:28.

Eskandari, L., Huang, Z., and Eyers, D. 2016. P-Scheduler: adaptive hierarchical scheduling in apache storm. In *Proceedings of Australasian Computer Science Week Multiconference* (ACSW '16).

Dean, J. and Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. In *Proceedings of 6th conference on Symposium on Operating Systems Design & Implementation* (OSDI'04).

Heinze, T., Aniello, L., Querzoni, L., and Jerzak, Z., 2014. Cloud-based data stream processing. In *Proceedings of 8th ACM International Conference on Distributed Event-Based Systems* (DEBS '14).

Kumar, T., Gupta, S., and Kushwaha, D. S., 2016. An Efficient Approach for Automatic Number Plate Recognition for Low Resolution Images. In *Proceedings of 5th International Conference on Network, Communication & Computing* (ICNCC '16).

Cao, L., Zhang, X., Chen, W., and Huang, K., 2014. License Plate Localization with Efficient Markov Chain Monte Carlo. In *Proceedings of International Conference on Internet Multimedia Computing and Service* (ICIMCS '14).

Jain, V., et al., 2016. Deep automatic license plate recognition system. In *Proceedings of 10th Indian Conference on Computer Vision, Graphics and Image Processing* (ICVGIP '16).

Li, H. and Shen, C., 2016. Reading Car License Plates Using Deep Convolutional Neural Networks and LSTMs. *arXiv* preprint arXiv:1601.05610.

Parasuraman, K. and Subin, P.S., 2010. SVM based license plate recognition system. In *Proceedings of the IEEE International Conference on Computational Intelligence and Computing Research*.

2016, An introduction to the Hadoop ecosystem on Azure HDInsight, viewed 23 April 2017, < https://docs. microsoft.com/en-us/azure/hdinsight/hdinsight-hadoop -introduction >