Framework for Real-Time Parallel and Distributed Natural Language Processing

D. Mileski*, V. Zdraveski*, M. Kostoska* and M. Gusev*
* Sts Cyril and Methodius University in Skopje, Faculty of Computer Science and Engineering, Skopje, North Macedonia
E-mail: dimitar.mileski@ieee.org, { vladimir.zdraveski, magdalena.kostoska, marjan.gushev }@finki.ukim.mk

***Abstract*—In this paper, we present a new framework for parallel and distributed processing of real-time text streams capable for executing NLP-Natural Language Processing algorithms. The focus is set on acceleration based on attention for building the topology, and not on the individual NLP algorithms. We elaborate the configuration of our specific use case, and discuss the reduction of the time required for system configuration in order to use the benefits of virtualization and containers.**

Research hypothesis: We can process more text tuples per unit time using the new developed framework for an algorithm that divides the sequential algorithm into smaller jobs and tasks including tokenisation, part of speech tagging, stopwords, sentiment analysis, where each of these individual jobs are specific nodes in the Apache Storm-based topology.

We have conducted an experimental proof-of-concept and found the optimal configuration confirming the validity of the hypothesis.

***Index Terms*—framework, real-time processing, natural language processing, parallel processing, distributed processing**

I. Introduction

The huge amount of textual data that is generated and the great development of natural language processing algorithms provide a wide range of possibilities. A problem would arise if we want to apply Natural Language Processing algorithm over real time text stream, and process a large amount of text tuples per unit time, in order to satisfy an application requirement. Time constraint will depend on the application requirement. To cope with the high speed of incoming texts real-time processing needs to implement a certain degree of parallelism.

One approach is to make fine-grained parallelism that could yield acceptable speedup, but in some cases this requires understanding of complex linguistic phenomena or how the algorithm is working in the fine-grained level [1]. But this requires specific knowledge of the algorithm and knowledge of NLP field.

All existing systems built to use a parallel and distributed infrastructure provide a lot of processing speed but have an additional learning curve as some require a change of platform or language, and do not allow direct implementation of the Python programming language as the most widely used development environment for NLP algorithms. This would mean either adapting existing algorithms or completely re-implementing existing algorithms written in Python, which is our initial motivation for this research.

The developed framework specified in this paper uses the Apache Storm environment to provide low latency, message delivery guarantee, and fault tolerance [2]. However, the main languages in which Apache Storm-based programs are developed are Java and Scala, which does not directly provide an environment for developing NLP-based algorithms with Python.

In this paper we present a new framework with specification of an initial system configuration in a container, which will provide an environment to write NLP algorithms in Python to be executed on the Apache Storm-based infrastructure and cluster.

We aim at testing the validity of the following research hypothesis: *We can process more text tuples per unit time with the new developed framework, just by dividing the sequential algorithm into jobs that include tokenisation, part of speech tagging, stopwords, sentiment analysis, if these individual jobs are nodes in the Apache Storm-based topology.*

To verify the validity of this hypothesis, we will realize a prototype implementation, test the performances and compare it to the conventional methods. We will first measure the time required for the different loads, different number of text tuples on the sequential implementation. Then the same sequential algorithm divided into jobs as tokenization, part of speech tagging, and placement of each job in a different node from the Apache Storm topology and we will measure the execution time for the corresponding loads. The hypothesis can be confirmed if the number of processed text tuples per unit time is increased.

The rest of the paper is organized according to the following structure. Section II presents the related work and findings in the specific field. System architecture in Section III should illustrate the significance of the Apache Storm topology and the way we should built topologies. Because the way Apache Storm topology is constructed is crucial for the acceleration obtained with this framework. Experimental methods used and the environment in which the experiments are executed, in detail are described in Section IV. Results are presented and evaluated in Section V. Finally, Section VI addresses the conclusions and gives directions for future work.

II. Related Work

There are various researches in the field of real-time NLP applications. Some of them are aimed at speeding up NLP algorithms but there is also a lot of research potential to parallelize or distribute the processing at different processing instance for large amounts of textual data [3]. This acceleration

approach where the processing takes place on several different processing instances gives much better results [4].

Although parallelization at the level of NLP algorithms can give quite good results, for real-time systems it is necessary to perform the same operation over multiple texts that arrive in parallel. One such architecture was developed with the help of Apache Storm [2], where each text operations such as tokenization, part of speech tagging, named-entity recognition, stemming, lematisation is defined as a node in the Apache Storm topology, so that multiple documents can be concurrently processed in different parts of the topology. Some nodes have multiple instances [4].

Two architectures have been proposed to handle large amounts of text, one of which uses the Apache Storm topology and the other uses batch processing, but batch processing is not in line with what we need, and that is text data flow analysis in real time [5]. A system called STREAMIT has been developed which is used for visualization of streams of textual data in real time, i.e. for monitoring streams of textual data. This system uses graphics cards with the help of CUDA implementation. The algorithm is executed on individual distributed threads on a grid of CUDA blocks. Each thread accesses shared memory that can write the results needed to visualize the text streams [6].

There are Real-Time event detection systems on social data streams. One such system is the Real-Time Detection of Traffic From Twitter Stream Analysis which uses the Twitter API and Weka (Waikato Environment for Knowledge Analysis) [7]. A scalable and distributed event detection system has been developed using Storm topology and Twitter text streams [8].

Trendminer: An Architecture for Real Time Analysis of Social Media Text also implements the concept of graph and nodes as processing units that can run in parallel but use the Java API and Hadoop. This system manages with a cluster of 6 machines, a total of 84 virtual cores, through 42 physical cores, to process data from the order of all daily tweets [9].

Another approach to speeding up text processing would be to speed up the implementation of individual NLP algorithms or fine-grained parallelisam. In [10] they show acceleration of text classification using SVM enhanced by multithreading and CUDA [10]. Implementation and comparison of parallel algorithms for pre-processing of social networking data based on GPGPU (NVidia CUDA) and Hadoop Map Reduce Architectures is given in [11]. This work compare and contrast the benefits and drawbacks of GPGPU and Hadoop Map-Reduce parallel algorithms for pre-processing of text data.

III. System Architecture

The sequential approach is mostly used for applications that process small amount of textual data or are not real-time processing systems for textual data. However, this is not recommended for applications that have a stream of text data as input. These applications need to perform NLP-based algorithms on an incoming textual data stream and output the results to a file, database, or visualize them. For these systems we need a completely different approach.

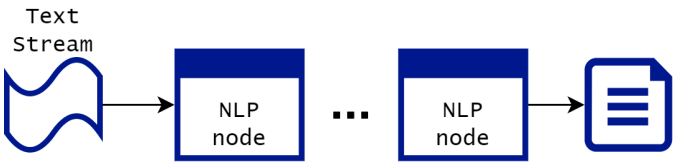

Fig. 1. Sequential NLP processing.

Apache Storm meets such requirements as a distributed real-time computing system. The main component of the Apache Storm is the topology. One topology is in the form of a direct acyclic graph (DAG). The nodes in the graph can be spouts, or bolts. Spouts are nodes from which the data stream comes, and Bolts are the processing nodes. The storm cluster consists of a master node (Nimbus), and multiple worker nodes. Each worker node has a supervisor who is in charge of managing the processes in the node. Another important part of an Apache Storm cluster is the Zookeeper cluster, which coordinates work between the master node (Nimbus) and the worker nodes. Apache Storm manages most of the things for us but still needs some configuration and has learning curve.

One of the basic issues is the problem that the whole Apache Storm-based system is primarily intended for Java environment. However, most ML-based algorithms, especially natural language processing, are written in Python. We will use the Streamparse Python library. Streamparse library lets you run Python code against real-time streams of data via Apache Storm. Primarily Apache Storm supports many programming languages but as proof of concept, Streamparse solves many of the problems.

Sequential processing is suitable for small textual data or data that does not have real-time requirements. Fig. 1 shows the sequential way of processing textual data, which we will use in the experimental part to compare and obtain the final results. NLP node further in the text is a certain processing node, which can be pre-processing such as tokenisation, stopwords, lemmatiosation, stemming and others, or a certain algorithm of natural language processing such as named-entity recognition (NER), part of speech tagging (POS), classification, sentiment analysis, text to speech.

Apache Storm-based topology for real-time stream NLP processing is presented in Fig. 2. The topology starts with a spout which is a stream of textual data. This is followed by the pre-processing section. tokenisation, stopwords, and lemmatisation /stemming. These three bolts, processing units can have multiple instances, so here we have the parallelization of pre-processing. Each tuple, independently of the others, flows through that part. This is followed by a bolt for sentiment analysis. So the tuples with positive sentiment continue to one branch and the bags with negative sentiment continue to another branch. And here we have parallel and distributed processing of text tuples. Finally the results are written in separate files. We see each of the nodes in the topology as an NLP job. The parallelism in our framework is at the job level. The specific implementation of each individual NLP

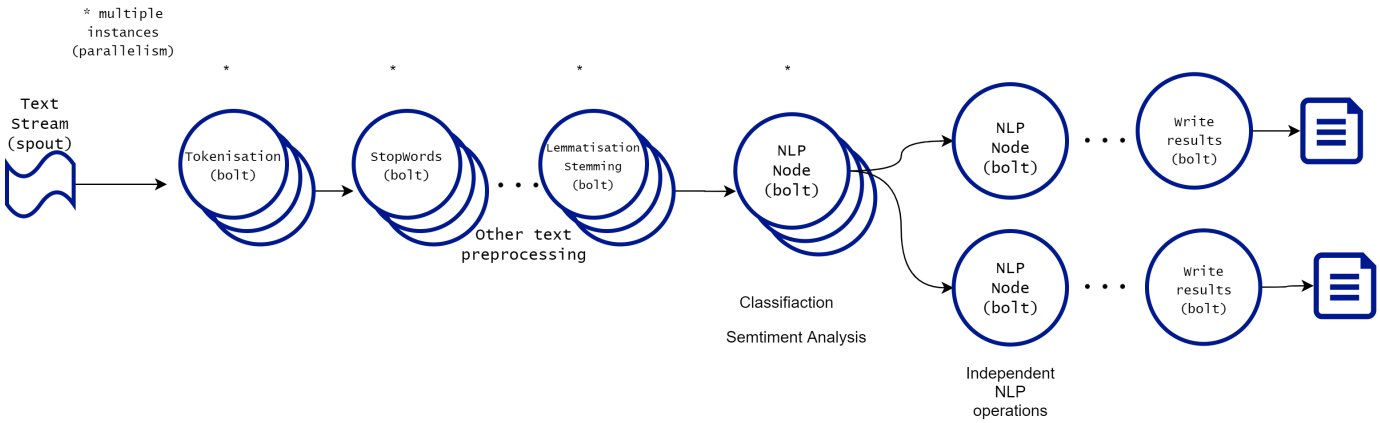

Fig. 2. Real time NLP topology

job affects the overall processing performance. For example if we have algorithm 1 and 2 both for tokenization, we declare tokenisation as NLP job. If algorithm 1 processes sentences faster than algorithm 2, then we will use algorithm 1 because it will affect the overall performance of the topology. In this case the parallelization of tokenization and implementation details of tokenisation are beyond the main focus of this research. Purpose of this paper is to explore job level paralelism.

Data independency of particular tasks is analyzed when building a topology, and for efficient parallelization it is important that a particular node is independent of previous processing nodes. Nodes that execute certain work and are independent of previous nodes can be efficiently parallelised in various ways. Therefore, prior to approaching the addressed problem, it is good to be able to define the processing units (nodes) including units for tokenisation, stopwords, classification in order to analyze the data dependency and to determine whether they are dependent to the other nodes.

The data dependence analysis for this specific system architecture showed the the following nodes: tokenisation, stopwords, Lemmatisation / Stemming are sequentially dependent on exchanged data. However, we can exploit parallelism for execution of multiple instances on the same node. We can arrange this type of processing due to independent tuples that move through the nodes. Parallelism after the sentiment node can be exploited for those text tuples in such a way that those the positive items will go to one branch of the topology and the negative items to the other branch. Note that the whole topology is executed over an Apache Storm-based cluster which can have more than one machine (physical or virtual) and each machine will execute multiple processes requested by the operating system.

The whole process of creating an NLP pipeline is explained with a certain level of abstraction. To create the topology, we use the streamparse library. The streamparse library provides tools to configure the number of bolts (parallel processing units), so that we can parallelize the same job, i.e. write how many instances of bolt node will be instantiated.

Then we create an Apache Storm cluster by creating a master node (Nimbus), worker nodes and a zookeeper cluster. This means that on each machine in the cluster we need to install and configure the necessary requirements as well as network connections. These requirements include Apache Storm requirements as well as NLP requirements.

Finally, the topology is deployed to the cluster. This type of processing of a large amount of text data implies that the processing of individual nodes should be as fast as possible. Each node will work on one NLP job, and if it is about certain ML-based algorithms, it is best for them to be pre-trained, that is only to be used for prediction.

The node with sentiment analysis from Fig. 2 will use a pre-trained model from the NLTK library [12].

The intention of this Framework is to speed up the processing of text tuples, not to develop a new methods in the field of Natural Language Processing. One of the recommendations is to use the principles of Transfer Learning [13], so that in each of the nodes in Strom Topology that do a certain NLP job for which a model is needed, to load the pre-trained model for each node separately. Using pre-trained models, libraries, and NLP toolkits will make the code cleaner and easier to understand. Because the cluster management process is more complex and can create problems with different virtual environments on different worker nodes. The Apache Storm Framework itself balances the work in the cluster nodes but still a careful approach is required to use of certain NLP algorithms in individual nodes.

IV. Experimental Method

The system specification to conduct experiments in this research is specified in Fig. 4. A virtual machine with a docker is installed on the system. The number of virtual CPUs assigned to this virtual machine is 4. Docker container contains the Apache Storm system, and streamparse library that lets you run Python code against real-time streams of data via Apache Storm. This type of virtualisation with containers allows an environment to use Apache Storm and write NLP algorithms.

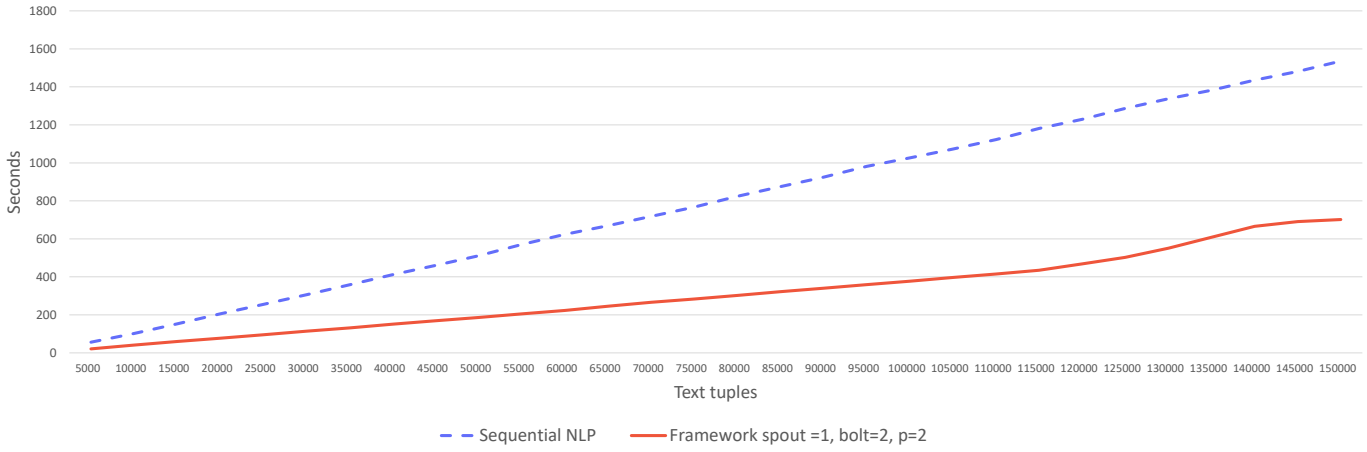

Fig. 3. Execution time for the Sequential NLP and Parallel Apache Storm-based experiments.

So we focus on how to build topologies as illustrated in the example Fig. 2.

The experiment proceeded as follows. Input in both approaches are text tuples with an average length of 280 characters, as a standard length used in social networks, so the posts have the maximum size of 280 characters. Execution time is measured for the two different approaches:

- *Sequential NLP* is a Python script in which the text tuples are processed sequentially and each of the text tuples moves through the pipeline - tokenization, stopwords, stemming, sentiment analysis. The sequential NLP experiment uses a Python-based algorithm, pre-processing and sentiment analysis are performed sequentially.
- *Parallel NLP model* specified by the Apache Storm-based topology with spouts, bolts and worker processes.
  - First a comparison of the sequential and parallel approach is made. The initial configuration of Apache Storm was bolt=1, spout=2, worker process=2.
  - Second, a comparison of different configurations for spout, bolt and worker process was made, and their respective speedup.

In both experiments, the response time (latency) T_{total} to finish the whole task is measured for different execution workloads W of 5.000, 10.000, 15.000, 20.000, 25.000 ... 150.000 text tuples respectively as input. At the end, calculations are made for the achieved speed v reached for a specific load W by (1).

$$v(W) = \frac{W}{T_{total}(W)} \tag{1}$$

For each of the experiments we measure the response time $T_S(W)$ and $T_P(W)$ respectively for sequential and parallel version for a specific load W. We calculate the speedup $S(W)$ of the Storm topology with respect to the sequential NLP pipeline by (2) for each workload W.

$$S(W) = \frac{T_S(W)}{T_P(W)} \tag{2}$$

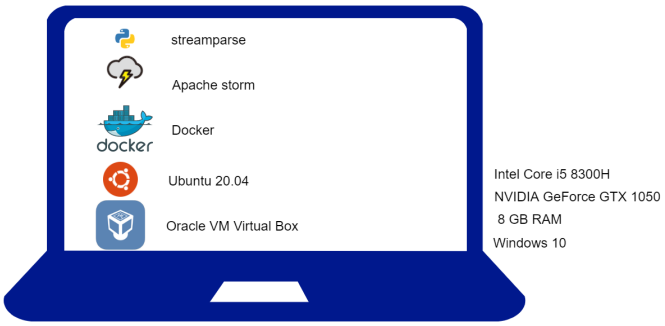

Fig. 4. System specification

V. Evaluation of Results

The results of both approaches are obtained and performed on the virtual machine illustrated in Fig. 4 with a goal to analyze the performance real-time natural language processing of text tuples.

Fig. 3 presents the results of execution time versus various input for both experiments. Increasing the number of text tuples results with a linear increase of the execution time for the Sequential NLP experiment.

It is important to note that the initial results in Fig. 3 are obtained with 1 spout (which emit tuples), 2 bolts (parallel processing units) where the sentiment analysis takes place, and 2 worker processes. With this configuration the speedup ranges from 2.2 to 2.8, as presented in Fig. 5.

Achieved performance for different configuration of spouts, bolts and worker processes is presented in Fig. 6. Fig. 7 presents the calculated speedup relative to sequential algorithm of different configurations.

We observe that the execution time increases linearly for all configurations as long as the workload W increases. The highest achieved speedup in the range 2.2 to 2.8 is achieved for the configuration specified by $spout = 1, bolt = 2$ and $p = 2$ (worker processes).

Note that the speedup slightly decreases after 115K text tuples, although its value is over 2.2.

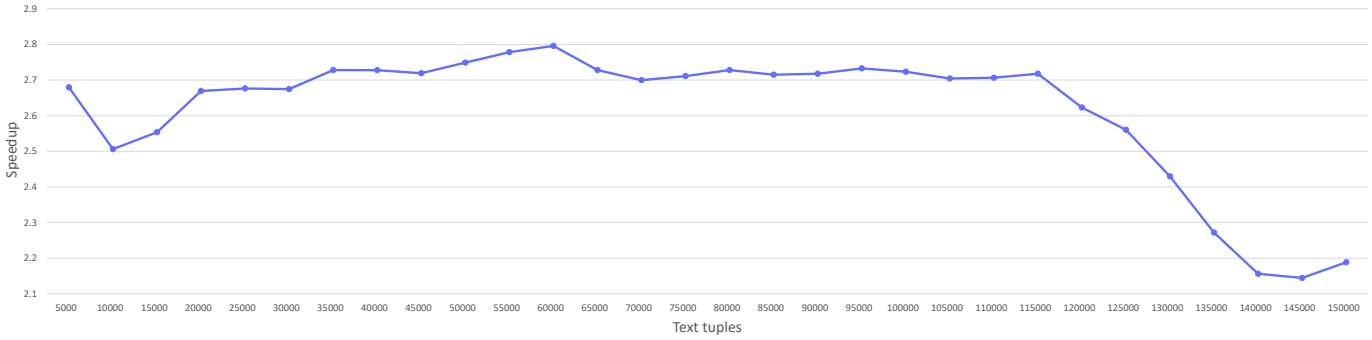

Fig. 5. Achieved speedup for the experiments

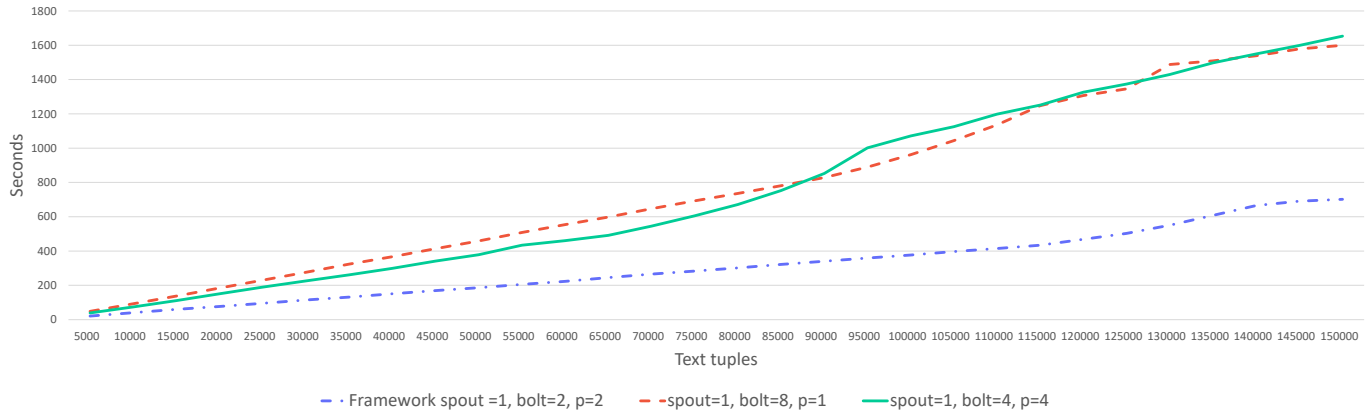

Fig. 6. Different configuration of spouts, bolts and worker processes

The main parameters to pay attention are the number of spout, bolt, and worker processes. A worker process executes a subset of a topology. An executor is a thread that is spawned by a worker process. Worker process belongs to a specific topology and may run one or more executors for one or more components (spouts or bolts) of this topology. So the number of worker processes, spouts and bolts affect the number of processes and threads that are created. Different number of spouts, bolts and worker processes will give different results. According to the Apache Storm documentation, the number of threads that will be created per worker process is calculated by dividing the number of spouts and bolts by the number of worker processes.

$$\frac{(spout + bolt)}{p} = threads \tag{3}$$

According to (3) we can calculate number of threads per worker process.

- spout = 1, bolt = 2, p = 2
  2 threads per worker process
  2 worker processes, 4 threads
- spout = 1, bolt = 8, p = 1
  9 threads per worker process
  1 worker process, 9 threads
- spout = 1, bolt = 4, p = 4
  2 threads per worker process
  4 worker processes, 8 threads

The experiment was conducted on a virtual machine that uses 4 cores, and this is why for this case the optimal configuration is $spout = 1, bolt = 2, p = 2$ which produces a total of 4 threads. Increasing the number of threads reduces performance and also allows the process to have unpredictable execution times, and occasional performance drops. Also a large number of worker processes and threads can result in out of memory exception, because we have limited working memory in each of the nodes in the Storm cluster.

A lot of experiments needs to be conducted to find the optimal number of $spouts, bolts$, and $worker$ processes such that for each implementation and system specification. In our case, we found that the best performance for this implementation and system specification of a 4-core virtual machine, is achieved by using 2 worker processes with 2 threads each, a total of 4 threads.

VI. Conclusion

A larger number of text tuples processed per unit time can be obtained by dividing natural language processing sequential algorithms, into smaller jobs, which can be tokenization, stopwords, part of speech tagging, named entity recognition,

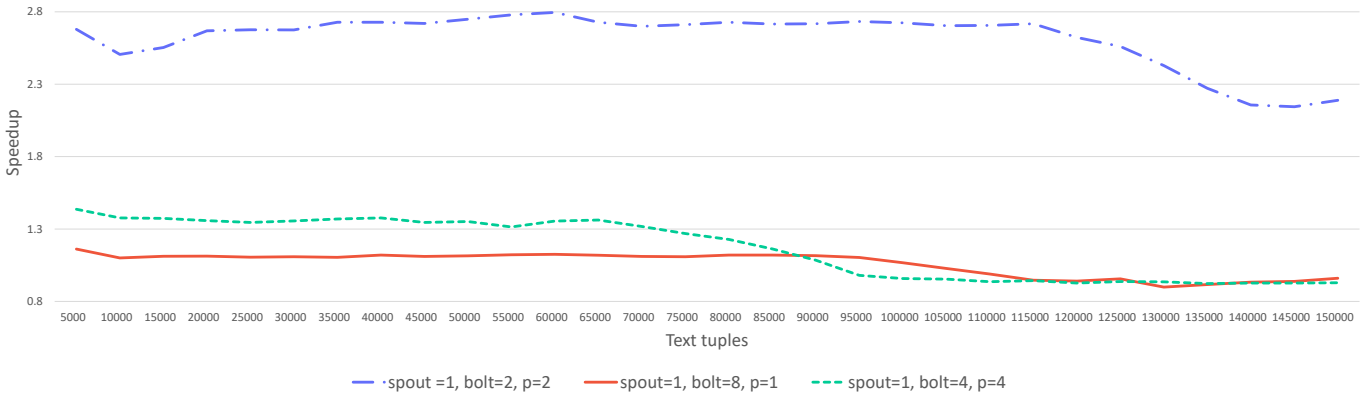

Fig. 7. Achieved speedup for different configuration of spouts, bolts and worker processes

sentiment analysis, each of those jobs can be a node, bolt-processing node, in the Apache Storm topology.

This can be achieved with the Python programming language and the proposed framework, leaving the same implementation of NLP algorithms. The change that needs to be made in order for the sequential algorithm written in Python script to be processed by the framework is to install a docker, instantiate a container with the initial configuration of Apache Strom and streamparse.

The customization of the code would be as follows:

- From the Python script where the sequential version of the code is written, take the part with the stream source and insert it into the spout node of the Apache Storm.
- For each of the defined NLP jobs make a Bolt-processing node in Apache Storm.
- Configure the Apache Storm for a specific cluster and configure the number of bolt-processing nodes instances for each NLP job.

Therefore the validity of the research hypothesis is confirmed and aan experimental proof-of-concept is provided we can get quite significant accelerations with this framework and with relatively little knowledge of implementation details about NLP algorithms, also working in the Python environment which is known for developing NLP algorithms. Final benefit is that we can take advantage of the libraries and large toolkits that already exist.

Future research will reveal answers about scaling the system and trying out different configurations for the Apache Storm cluster.

REFERENCES

[1] W. Jiang, Y. Zhang, P. Liu, J. Peng, L. T. Yang, G. Ye, and H. Jin, "Exploiting potential of deep neural networks by layer-wise fine-grained parallelism," *Future Generation Computer Systems*, vol. 102, pp. 210–221, 2020.

[2] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[3] R. Agerri, X. Artola, Z. Beloki, G. Rigau, and A. Soroa, "Big data for natural language processing: A streaming approach," *Knowledge-Based Systems*, vol. 79, pp. 36–42, 2015.

[4] X. Artola, Z. Beloki, and A. Soroa, "A stream computing approach towards scalable nlp." in *LREC*, 2014, pp. 8–13.

[5] M. Kattenberg, Z. Beloki, A. Soroa, X. Artola, A. Fokkens, P. Huygen, and K. Verstoep, "Two architectures for parallel processing of huge amounts of text," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, 2016, pp. 4513–4519.

[6] J. Alsakran, Y. Chen, Y. Zhao, J. Yang, and D. Luo, "Streamit: Dynamic visualization and interactive exploration of text streams," in *2011 IEEE Pacific Visualization Symposium*. IEEE, 2011, pp. 131–138.

[7] E. D'Andrea, P. Ducange, B. Lazzerini, and F. Marcelloni, "Real-time detection of traffic from twitter stream analysis," *IEEE transactions on intelligent transportation systems*, vol. 16, no. 4, pp. 2269–2283, 2015.

[8] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic, "Scalable distributed event detection for twitter," in *2013 IEEE international conference on big data*. IEEE, 2013, pp. 543–549.

[9] D. Preotiuc-Pietro, S. Samangooei, T. Cohn, N. Gibbins, and M. Niranjan, "Trendminer: An architecture for real time analysis of social media text," in *Sixth International AAAI Conference on Weblogs and Social Media*, 2012, pp. 38–42.

[10] S. Chatterjee, P. G. Jose, and D. Datta, "Text classification using svm enhanced by multithreading and cuda." *International Journal of Modern Education & Computer Science*, vol. 11, no. 1, 2019.

[11] V. J. Nirmal and D. G. Amalarethinam, "Parallel implementation of big data pre-processing algorithms for sentiment analysis of social networking data," *International journal of fuzzy mathematical archive*, vol. 6, no. 2, pp. 149–159, 2015.

[12] S. Bird, "Nltk: the natural language toolkit," in *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, 2006, pp. 69–72.

[13] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.