BGElasor: Elastic-Scaling Framework for Distributed Streaming Processing with Deep Neural Network

Weimin Mu, Zongze Jin, Weiping Wang, Weilin Zhu, Weiping Wang

To cite this version:

Weimin Mu, Zongze Jin, Weiping Wang, Weilin Zhu, Weiping Wang. BGElasor: Elastic-Scaling Framework for Distributed Streaming Processing with Deep Neural Network. 16th IFIP International Conference on Network and Parallel Computing (NPC), Aug 2019, Hohhot, China. pp.120-131, 10.1007/978-3-030-30709-7_10 . hal-03770524

HAL Id: hal-03770524

https://inria.hal.science/hal-03770524v1

Submitted on 6 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

BGElasor: Elastic-Scaling Framework for Distributed Streaming Processing with Deep Neural Network

Weimin Mu[1,2], Zongze Jin[1,2 ⋆], Junwei Wang[1,2], Weilin Zhu[2], and Weiping Wang[2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
{muweimin, jinzongze, wangjunwei, zhuweilin, wangweiping}@iie.ac.cn

Abstract. In face of constant fluctuations and sudden bursts of data stream, elasticity of distributed stream processing system has become increasingly important. The proactive policy offers a powerful means to realize the effective elastic scaling. The existing methods lack the latent features of data stream, it leads the poor prediction. Furthermore, the poor prediction results in the high cost of adaptation and the instability. To address these issues, we propose the framework named BGElasor, which is a proactive and low-cost elastic-scaling framework based on the accurate prediction using deep neural networks. It can capture the potentially-complicated pattern to enhance the accuracy of prediction, reduce the cost of adaptation and avoid adaptation bumps. The experimental results show that BGElasor not only improves the prediction accuracy with three kinds of typical loads, but also ensure the end-to-end latency on QoS with low cost.

Keywords: Data stream processing · Load prediction · Deep neural network · Gated recurrent units · Elasticity.

1 Introduction

With the rapid development of the Internet and the rise of the Internet of Things (IoT), various software and sensors continuously generate massive amounts of continuous data streams. Distributed stream processing systems (DSPSs) [1–5] offer a powerful means to carry out data stream processing applications (DSPAs). The Quality of Service (QoS) is important for the DSPAs, which is commonly measured through end-to-end latency and throughput. For example, when an intrusion occurs, determining and warning operations should be made in a certain time window. Nevertheless, data streams have the characteristics of load varying and sudden burst. In this case, to ensure the QoS requirements of the DSPAs, the elasticity of the DSPSs has become more and more important.

⋆ Corresponding author

Many researches have focused on improving the elasticity of the DSPSs. They can be divided into two classes, one is based on the reactive policy and the other one is based on the proactive policy. Although the reactive policy [6–8] is widely used by many DSPSs, it results in some severe issues, such as the QoS degradation and the frequent scaling actions. The elastic scaling happens when the performance of DSPAs does not match the work load.

To address these problems of the reative policy, many researchers propose the proactive policy. With the proactive policy, the scaling actions are executed in advance based on the prediction result of some performance metrics. The existing predicting methods are mainly based on time-series models and some machine learning methods. Traditional time series models, such as MA, ARIMA [9], and Holt-Winters [10] have been widely used. Meanwhile, some methods based on machine learning are often used to predict. Zacheilas et al. [11] provides an adaptive algorithm based on the prediction of the load and latency in upcoming time windows using Gaussian Processes. Repantis et al. [12] proposes a hot-spot prediction technique based on the linear regression for the purpose of alleviating application hot-spots in the DSPA. Hidalgo et al. [13] proposes a method to adjust the parallelism of the operators using Markov chain model. However, these methods lack the latent features of data stream, it leads the poor prediction. The reason is that they can not capture the nonlinear characteristics of drastic fluctuating data stream [14, 15] well. Furthermore, the poor prediction results in the high cost of the elastic scaling and the instability.

In order to statisfy the QoS of the DSPAs well, in this paper, we propose a proactive and low-cost elastic-scaling framework based on the accurate prediction using deep neural networks. In summary, our paper makes following contributions.

- We propose the framework named BGElasor, which is a proactive and low-cost elastic-scaling framework based on the accurate prediction using deep neural networks. It can capture the potentially-complicated pattern to improve the accuracy of prediction, reduce the cost and frequency of the elastic scaling.
- As far as we know, our work is the first to use the bidirection gated recurrent units neural networks (BiGRU) to catch the features of the fluctuations of the data stream and build the prediction module (Predictor) to predict the input rate of operators in the DSPSs.
- Besides, we propose a cost-based elastic-scaling algorithm, named CBA and build the elastic scaling module (ElasticityController). It invokes Predictor for multiple time windows ahead of the current state and finds the right point to increase (scale-out) or decrease (scale-in) the parallelism degree of each operator with low cost.
- Finally, our BGElasor runs on DataDock, which is our data stream processing system. The experimental results show that BGElasor not only improve the accuracy of prediction on three kinds of typical loads, but also ensure the end-to-end latency on the QoS.

The rest of our paper is organized as follows. In Section 2, we introduce the background, including model of the data stream processing, our distributed stream processing system, DataDock. Section 3 mainly describes the design of BGElasor. We show experimental results of our framework in Section 4. Finally, Section 5 concludes our paper.

2 Background

In this section, we first present the model of the data stream processing. Then we present our data stream processing system, DataDock.

2.1 DSP Model

A DSPA over data stream is usually organized as a directed acyclic graph G = (O, S), where O is the operator set and S is the stream set. An operator $o \in O$ represents a sort of computation logic, such as filter, join, aggregate or user-define function. ***Src*** and ***Sink*** are two special operators in G, which are responsible for spouting source streams and collecting final results, respectively. A stream $s \in S$ is a directed arc $(o_p$, $\mathrm{o}_c)$, o_p, $o_c \in O$, where o_p and o_p are the producer and consumer respectively of s. When a DSPA is submitted to the underlying cluster to execute, its logic operator graph will be transformed into the execution graph, in which each operator o is parallelized into multiple instances $I_o = \{o^1, ..., o^\alpha\}$, where $\alpha \in \mathbb{N}^+$ is the parallelism.

2.2 DataDock

DataDock is our distributed data stream processing system implemented in Java. It is mainly aimed at satisfying the heterogeneous data preprocessing requirements. In order to ensure that the system executes efficiently, DataDock only retains core functions of a DSPS, such as the DSPA definition, the job scheduling.

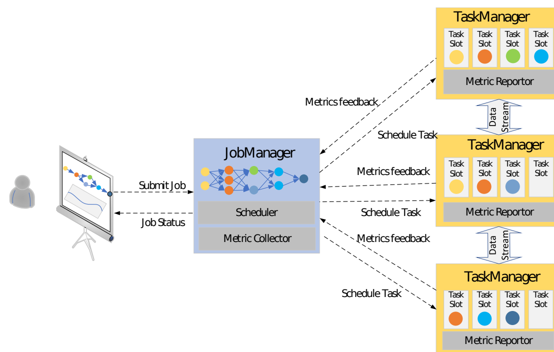


Fig. 1. Architecture of DataDock.

As shown in Fig. 1. DataDock offer users an user interface (UI) to define the DSPA. The JobManager receives a DSPA and turns it into an execution graph. Then the execution graph is scheduled to execute in a set of TaskManagers. The TaskManager runs on a node and is responsible for local resource allocation and instance management. During the execution, the TaskManager continuously collects the performance metrics of each operator instance and reports them to the JobManager. The MetricCollector is in charge of gathering these metrics and storing into MetricDatabase. At the runtime, DataDock allows the instances to quickly discard input data to make sure the latest data receives priority processing. For the fault tolerance, DataDock adopts the similar fault-tolerant policy as Storm to ensure that each event is processed at least once.

3 Framework

3.1 Overview

We show our framework BGElasor as shown in Fig. 2, it contains two important modules, the Predictor and the ElasticityController. We use the Predictor to predict the input rate of all operators. Then we refer the results of the prediction and use the ElasticityController to adjust the parallelism of operators.

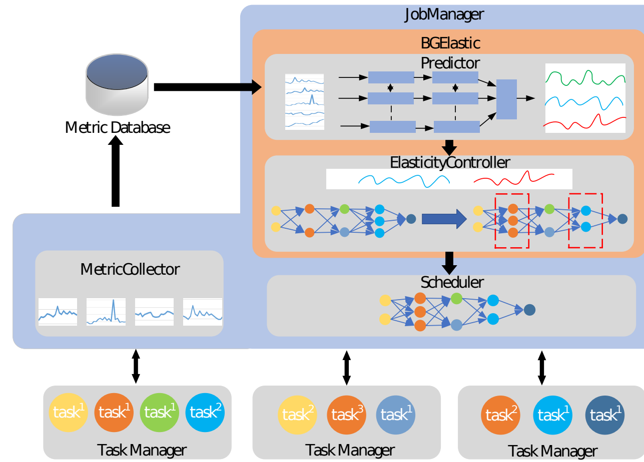


Fig. 2. The Architecture of BGElasor

At the runtime, Predictor runs periodically and reconstructs the input rate model for each operator using BiGRU. Then we use the ElasticityController to update the execution graph based on predictions given by Predictor. At last, we use the JobManager to schedule according to the lastest execution graph.

3.2 Predictor

In this section, we build the Predictor to predict the input rate of each operator based on BiGRU. The Predictor contains three parts, the model input, the prediction networks and the model output.

Model Input. At the runtime, our model is trained offline. The Predictor periodically reads input rate metrics over the past b time period of each operator from MetricDatabase and normalizes them to the range $[0, 1]$ with the Min-Max scaler.

For the prediction of input rates sequences, formally, we use t to denote the current time and use o to denote the operator whose input rates will be predicted. We use b and f to denote the length of historical time window and the length of future time window respectively. $v(o,\ t)$ denotes the input rates of o at time t, so, we use $(v(o,\ t\text{-}b),\ v(o,\ t\text{-}b{+}1),\ ...,\ v(o,\ t))$ to denote the input rates sequence of o over the past b time period, that is, the input of our predictor. Correspondingly, $(v(o,\ t{+}1),\ (v(o,\ t{+}2),\ ...,\ (v(o,\ t{+}f))$ denotes the input rates sequence of o in the future f time period, which is the output of the Predictor.

For example, assuming the current time is t, the model input is $X_t{=}(v(o,\ t\text{-}b),\ v(o,\ t\text{-}b{+}1),\ ...,\ v(o,\ t))$, the label at t is $y_t{=}(v(o,\ t{+}1),\ (v(o,\ t{+}2),\ ...,\ (v(o,\ t{+}f))$, and each element ranges between $[0, 1]$. Similarly, the model input at time $t{+}1$ is $X_{t+1}{=}(v(o,\ t\text{-}b{+}1),\ v(o,\ t\text{-}b{+}2),\ ...,\ v(o,\ t{+}1))$ and correspondingly the label is $y_{t+1}{=}(v(o,\ t{+}2),\ (v(o,\ t{+}3),\ ...,\ (v(o,\ t{+}f{+}1))$.

Prediction Networks. In face of drastic fluctuating characteristics of data stream, methods based on deep neural networks have shown better performance, compared with existing methods, for their powerful nonlinear generalization abilities. Long short-term memory neural networks (LSTM) and gated recurrent units neural networks (GRU) are two popular deep neural networks to predict the trends of the time-series. Compared with LSTM, the training process of GRU is more efficient, which is more suitable for the scenario of the data stream processing. But there is one shortcoming of GRU. Since it is only able to process the data in one direction ignoring the continuity of data changes, GRU can only capture the partial features of metrics. And its bidirection version, BiGRU, uses two separate hidden layers to process data in two directions to obtain more information in the time dimension during the training stage. To achieve the higher accuracy, we use BiGRU to predict the input rate.

The BiGRU formulation is as follows:

$$F_t = \sigma(W_F x_t + U_F F_{t-1} + b_F) \tag{1}$$

$$B_t = \sigma(W_B x_t + U_B B_{t+1} + b_B) \tag{2}$$

$$y_t = \sigma(V_F F_t + V_B B_t + b_o) \tag{3}$$

W_F and U_F denote the input-to-forward layer weight matrices. W_B and U_B are the weight matrices of the output-to-backward weight matrices layer. b_F, b_B and b_o denote biases of forward, backward and output layer, respectively. σ denotes the nonlinear activation function, such as Sigmoid function and Rectified Linear Unit.

Model Output. We first use BiGRU hidden layers to construct the BiGRU network. As mentioned in above sections, BiGRU can capture both forward and backward dependencies to make full use of the input data and learn the complex and comprehensive features. Then, we add a dense layer to transform high-dimensional data into the low-dimensional data to make predictions. During the training process, we use the mean square error (MSE) as the loss function which is computed using the following equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2 \tag{4}$$

3.3 ElasticityController

In this section, we build the ElasticityController. The ElasticityController can ensure the end-to-end latency with the minimum elastic-scaling cost. It contains two parts, the Cost Model and Cost-Balance-Algorithm (CBA).

Cost Model. We build a cost model to evaluate the total cost of all elastic-scaling actions for an operator from the current epoch C to the future epoch F. The cost is defined as:

$$W_o(\mathbf{n}) = p_o \sum_{t_C}^{t_F} n_{o\cdot t} + \sum_{t_C}^{t_{F-1}} (p_o^u C_{o\cdot t}^u(n) + p_o^d C_{o\cdot t}^d(n) + p_o^r C_{o\cdot t}^r(n)) \tag{5}$$

$$C_{o\cdot t}^u(n) = \max(0, n_{o\cdot(t+1)} - n_{o\cdot t}) \tag{6}$$

$$C_{o\cdot t}^d(n) = |\min(0, n_{o\cdot(t+1)} - n_{o\cdot t})| \tag{7}$$

$$C_{o\cdot t}^r(n) = \begin{cases} 0 & n_{o\cdot(t+1)} = n_{o\cdot t} \\ 1 & n_{o\cdot(t+1)} \neq n_{o\cdot t} \end{cases} \tag{8}$$

$W_o(\mathbf{n})$ is the total cost. p_o is the cost of system resources used by the single instance for each operator o. $n_{o\cdot t}$ is the instance number of operator o at time t. p_o^u is the startup-cost of a single o instance. p_o^d is the shutdown-cost of a single o instance. $C_{o\cdot t}^u(n)$ is the startup times of instances of o.$C_{o\cdot t}^d(n)$ is the stop times of instances of o. p_o^r is the cost of each re-routing. $C_{o\cdot t}^r(n)$ is o's re-routing times. To satisfy the end-to-end latency, we ensure that the processing capability of each operator is not less than the data input rate. In other word, $p_o \geq in_o$ at any time.

The processing capability of o is expressed as $p_o = \lambda_o n_o$, where λ_o is the capability of one instance of o. We know that $W_{o\cdot base}(\mathbf{n}) = \sum_{t_C}^{t_F} p_o n_{base}$, $\Delta W(\mathbf{n}) = W_o(\mathbf{n}) - W_{o\cdot base}(\mathbf{n})$ and $\Delta n_o = n_o - n_{base}$. The Cost Mode is defined as:

$$\begin{aligned} \min \quad & \Delta W_o(\mathbf{n}) = p_o \sum_{t_C}^{t_F} \Delta n_{o\cdot t} + \sum_{t_C}^{t_{F-1}} (p_o^u C_{o\cdot t}^u(n) + p_o^d C_{o\cdot t}^d(n) + p_o^r C_{o\cdot t}^r(n)) \\ \text{s.t.} \quad & \Delta n_o \geq 0 \end{aligned} \tag{9}$$

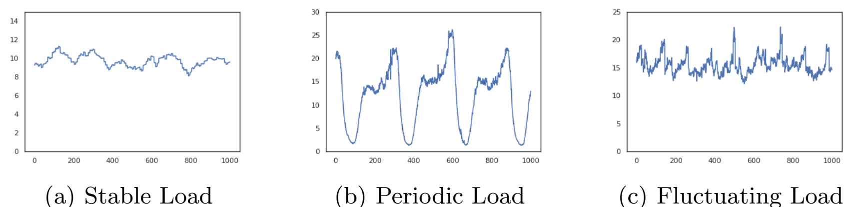
Cost-Balance-Algorithm. As mentioned above, we get the cost expression ΔW_o for operator o. The aim of the optimization should be $min(\Delta W_o)$ with the constraint: $\Delta n_o \geq 0(\forall o \in O, \forall t \in T)$. In order to address this issue, we propose the Cost-Balance-Algorithm (CBA). CBA improves the basic simple proactive elasticity algorithm by taking the cost of instance operations (e.g.re-routing and startup/shutdown) into account. CBA balances these three parts of the cost to guarantee lower system cost. CBA is divided into 3 steps. Firstly, we only consider computing capability to find the optimal scheduling timetable. Secondly, we refer the cost of re-routing to optimize timetable. At last, we refer startup and stop cost to optimize timetable. In CBA, the *act* denotes the result of the previous step and the input for the next step.

4 Experiments

4.1 Settings and Datasets

Settings. Our evaluations run on a cluster consisting of ten machines. These machines are all comprised of two eighteen-core Intel Xeon E5-2697 2.30 GHz CPUs, 256GB memory, and 500GB disks. One of the machines is used as JobManager and MetricDataBase, seven machines are used as TaskManagers, and the remaining machines with NVIDIA TESLA P4 GPUs are used to train and evaluate our prediction models.

Datasets. We collect the input rates records from our online DataDock system in 60 days as the dataset and you can download our dataset at `https://github.com/alexmu/DSP-R-BGElasor`. The dataset contains three type loads, which are stable load, periodic load and fluctuating load, as shown in Fig. 3. We divide them into three sets: a training set (from the beginning to the 40th day), a validation set (from the 40th day to the 50th day) and the test set (the 50th day to the last day). The training set is used to train prediction models, the validation set is used to optimize hyper-parameters and prevent overfitting, and the test set is used to evaluate the effectiveness of the prediction models.



(a) Stable Load (b) Periodic Load (c) Fluctuating Load

Fig. 3. Three types of data stream load.

Algorithm 1 CBA

Step 1

```
1: for cur : seq : prediction.non-reducings do
2:   if cur < act[cur-1].max then
3:     act[cur-1].up(roundUp((cur- act[cur-1].max)/p))
4:   end if
5: end for
6: for cur : seq : prediction.reducings do
7:   if cur < act[cur].min then
8:     act[cur].down(takeDown((act[cur].max - cur)/p))
9:   end if
10: end for
```

Step 2

```
1: for cur : seq.reverse : prediction.non-reducings do
2:   if act[cur].hasAct then
3:     cost0 = p^r, cost1 = act[cur].upNum * t
4:     act.mv(cur → cur-1) when cost0 > cost1
5:   end if
6: end for
7: for cur : seq : prediction.reducings do
8:   if act[cur].hasAct then
9:     cost0 = act[cur].downNum * t, cost1 = p^r
10:    act.mv(cur → cur+1) when cost0 > cost1
11:  end if
12: end for
```

Step 3

```
1: act.setHasChanged()
2: while act.hasChanged do
3:   for (down, up, ΔT) ∈ act.adjacency do
4:     min_num = min(down.num , up.num)
5:     cost0 = ( p^d +p^u ) * min_num + p^r
6:     cost1 = p * min_num * ΔT
7:     act[down, up].cancelAct(min_num) when cost0 > cost1
8:   end for
9: end while
10: return act
```

4.2 Predictor Evaluation

In this section, we compare the prediction performance of different algorithms, including ARIMA, SVR, LSTM, GRU, BiLSTM and BiGRU. We use the Root Mean Square Errors (RMSE) and Mean Absolute Errors (MAE) as the evaluation metrics. $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2}$ and $MAE = \frac{1}{n}\sum_{i=1}^{n}|x_i - \hat{x}_i|$

where x_i is the observed input rate, and $\hat{x}_i$ is the predicted input rate. All the compared models in this section are trained and tested multiple times to eliminate outliers, and the results of them presented are averaged to reduce random errors.

The experimental results are shown in Table 1. In our experiment, to balance the results of results, we repeat 10 times and get the results. For the stable load, the models all get low RMSE and MAE results. For periodic load, deep learning models significantly outperform other traditional models, such as ARIMA and SVR, because neural networks can learn more latent features from historical data. For the fluctuating load, the traditional linear function can not deal with the situation better, because the data load changes drastically. But the neural networks, which leverage nonlinear representations, can extract the hidden features from the time series data and detect the load change more accurately. And in our experiments, our model achieves the best performance.

Table 1. THE RMSE AND MAE OF EACH MODELS

Model	**Load Type**					
	Stable Load		**Periodic Load**		**Fluctuate Load**	
	RMSE	**MAE**	**RMSE**	**MAE**	**RMSE**	**MAE**
ARIMA	0.207	0.156	1.252	0.888	1.001	0.713
SVR	0.163	0.127	0.789	0.589	0.720	0.504
LSTM	0.141	0.113	0.541	0.397	0.639	0.433
GRU	0.124	0.098	0.531	0.381	0.637	0.434
BiLSTM	0.119	0.094	0.529	0.385	0.624	0.426
BiGRU	**0.101**	**0.081**	**0.490**	**0.341**	**0.581**	**0.401**

4.3 ElasticityController Evaluation

We evaluate the ElasticityController from three aspects, the total cost, the adaptation frequency and the latency guarantee. We compare CBA with 2 other algorithms, the Standard Reactive Elasticity Algorithm (SREA) and the Simple Proactive Elasticity Algorithm (SPEA). SREA considers only the current load and adjusts the instance number reactively. SPEA considers the load of the current epoch and the next epoch.

Total Cost. In this part, we evaluate the total cost of the three algorithms. Firstly, we get the instance number of operators o at the same epoch with three

algorithms. Then we use the data prediction result to generate the scheduling process. At last, we calculate the cost of each time and get the total cost of the experiment.

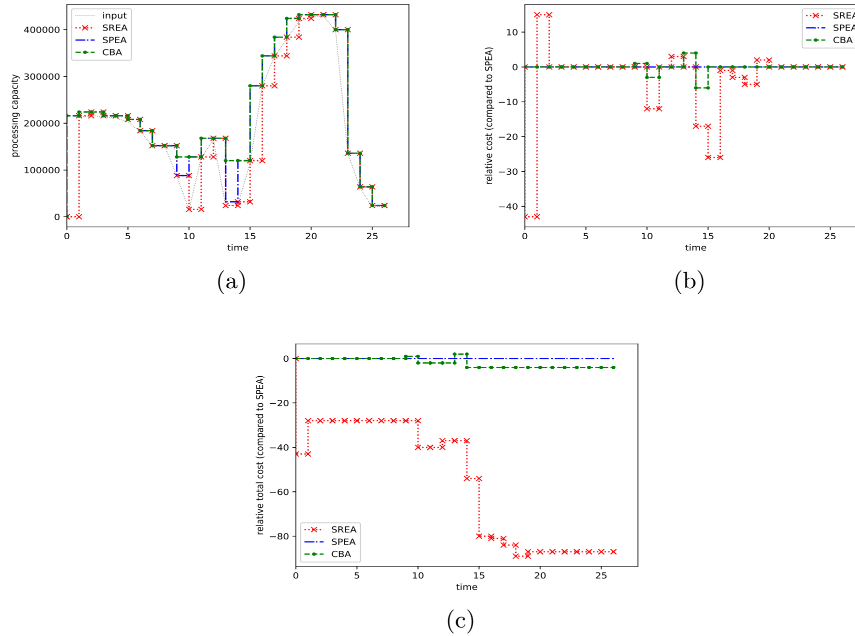


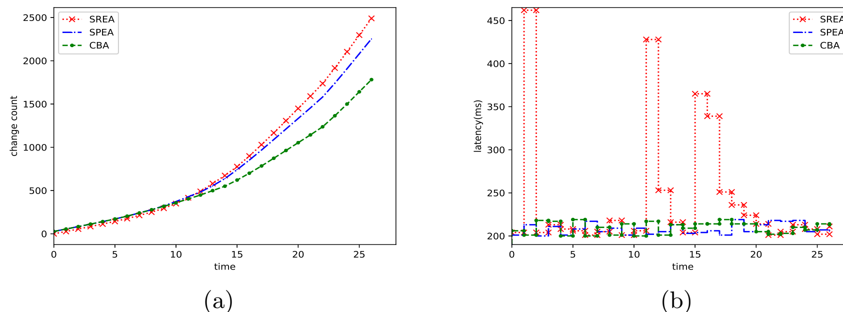
Fig. 4. Scheduling Result

In Fig.4, we set p_o=8000, p_o^u=5000, p_o^d=5000 and p_o^r=100. The Fig.4(a) represents the processing capacity. The Fig.4(b) represents the relative cost of each algorithm. The value is compared with the baseline of SPEA to get the relative cost. The Fig.4(c) represents the total relative cost. The baseline is the same as Fig.4(b).

When we consider both epoch 0 and 1, we find that the SPEA and CBA cost more than SREA because SPEA and CBA take the prediction into account. CBA considers the cost of the instance startup or shutdown and sometimes does not stop and start instances. Thus CBA costs less than SPEA.

Adaptation Frequency. In this part, we evaluate the adaptation frequency of the three algorithms. We collect the numbers of the instance number change at each epoch. The result is shown in Fig.5(a).

SREA considers only the current load, so it starts or stops instances later than the others from epoch 1 to epoch 10. However, after the epoch 10, SREA starts and stops more instances because of the sudden input rate fluctuation. Compared with SPEA, CBA takes the cost of instance startup and shutdown into account,



(a) (b)

Fig. 5. Adaptation Frequency and End-to-end Latency

so it is not necessary to deal with the instance startup and shutdown during the input rate fluctuation. So CBA changes less than SPEA.

End-to-end Latency Guarantee. In this part, we focus on the latency guarantee. We use the DataDock to compare three algorithms and record the end-to-end latency of each algorithm. The result is shown in Fig. 5(b).

CBA and SPEA take the prediction into account and start instances before input rate rises, so they deal with the data fast and stably. SREA only starts instances when the input rate is beyond the processing capacity. It results in the processing waiting everytime when input rate rises beyond processing capacity. So SREA is unstable. In our experiments, CBA outperforms the others.

5 Conclusion

In this paper, we propose a framework, BGElasor, contains two important modules, the Predictor and the ElasticityController. The Predictor based on BiGRU to get the precise prediction result of the input rate of each operator. Then we refer the results of prediction and use the ElasticityController to adjust the parallelism of operators. Experiments on the real load demonstrate our framework is better than the state-of-the-art methods, which not only improves the prediction accuracy with three kinds of the data loads, but also ensure the end-to-end latency on the QoS with the low cost.

References

1. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: the stanford stream data manager," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 19–26, 2003.
2. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine," in *CIDR 2005,*

Second Biennial Conference on Innovative Data Systems Research, pp. 277–289, 2005.
3. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: distributed stream computing platform," in *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops*, pp. 170–177, 2010.
4. "Storm." `http://storm.apache.org/`.
5. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink^{TM}: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.
6. R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013*, pp. 725–736, 2013.
7. V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, 2012.
8. B. Gedik, S. Schneider, M. Hirzel, and K. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, 2014.
9. N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," in *ACM/SPEC International Conference on Performance Engineering, ICPE'13*, pp. 187–198, 2013.
10. C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13*, pp. 15–26, 2013.
11. N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic complex event processing exploiting prediction," in *2015 IEEE International Conference on Big Data, Big Data 2015*, pp. 213–222, 2015.
12. T. Repantis and V. Kalogeraki, "Hot-spot prediction and alleviation in distributed stream processing applications," in *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008*, pp. 346–355, 2008.
13. N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *Journal of Systems and Software*, vol. 127, pp. 205–216, 2017.
14. Y. Xing, J. Hwang, U. Çetintemel, and S. B. Zdonik, "Providing resiliency to load variations in distributed stream processing," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, pp. 775–786, 2006.
15. Y. Xing, S. B. Zdonik, and J. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005*, pp. 791–802, 2005.