A Machine Learning-Based Elastic Strategy for Operator Parallelism in a Big Data Stream Computing System

Wei Li[1], Dawei Sun[1](✉), Shang Gao[2], and Rajkumar Buyya[3]

[1] School of Information Engineering, China University of Geosciences, Beijing 100083, People's Republic of China
{leeway, sundaweicn}@cugb.edu.cn

[2] School of Information Technology, Deakin University, Melbourne, VIC 3216, Australia
shang.gao@deakin.edu.au

[3] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia
rbuyya@unimelb.edu.au

Abstract. Elastic scaling in/out of operator parallelism degree is needed for processing real time dynamic data streams under low latency and high stability requirements. Usually the operator parallelism degree is set when a streaming application is submitted to a stream computing system and kept intact during runtime. This may substantially affect the performance of the system due to the fluctuation of input streams and availability of system resources. To address the problems brought by the static parallelism setting, we propose and implement a machine learning based elastic strategy for operator parallelism (named Me-Stream) in big data stream computing systems. The architecture of Me-Stream and its key models are introduced, including parallel bottleneck identification, parameter plan generation, parameter migration and conversion, and instances scheduling. Metrics of execution latency and process latency of the proposed scheduling strategy are evaluated on the widely used big data stream computing system Apache Storm. The experimental results demonstrate the efficiency and effectiveness of the proposed strategy.

Keywords: Operator parallelism · Runtime awareness · Resource allocation · Machine learning · Stream computing · Distributed system

1 Introduction

In recent years, big data has driven the rapid advances in distributed systems. There are generally two processing methods for big data: batch processing and stream processing [1]. Compared with batch processing, stream processing is more suitable for real-time applications. Distributed stream processing platforms enable big data applications to process continuous stream data and obtain near real-time feedback [2]. At present, the mainstream distributed stream processing platforms include Apache Storm [3], Apache

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2022
Published by Springer Nature Switzerland AG 2022. All Rights Reserved
W. Xiang et al. (Eds.): BROADNETS 2021, LNICST 413, pp. 3–19, 2022.
https://doi.org/10.1007/978-3-030-93479-8_1

Flink [4], Apache Spark (Spark Streaming) [5], Apache Samza [6], Apache Apex [7], and Google Cloud Dataflow [8]. Through an elastic execution engine, Flink can support batch processing tasks and stream processing tasks at the same time, as well as state management. It suits projects that require high throughput, low latency and demand state management or window statistics. Storm requires to design a topology first and then assign the topology to Execution nodes in a cluster, making it more suitable for small independent projects with low latency. Spark Streaming divides the input data stream into multiple batches through micro-batch processing, which is more suitable for projects in the Spark ecosystem. The work in this paper is optimized based on the widely used Storm platform, but the entire design, its strategy and model are not only limited to the Apache Storm platform. It can be applied to a variety of related streaming computing environments.

With the Storm default scheduling, if there are idle resources, uneven load and overload problems may occur [9, 10]. If no idle resources, there might be poor resource distribution caused by computing and communication bottlenecks in heterogeneous clusters [11]. The fundamental problem is that once the relevant parameter configuration is determined, the system cannot optimize parameter configuration during runtime. To support elastic adjustment, we face the following challenges: first, our solution must be compatible with the mainstream streaming computing platforms, such as Apache Flink, Apache Storm, and Apache Spark Streaming; the second is that the entire process must be monitored in real time to achieve true self-regulation; finally, the problem that needs to be solved is when using high-overhead pluggable scheduling, it is likely to introduce a new bottleneck affecting the whole performance [12].

1.1 Contributions

Motivated by the above discussion, we propose an elastic scaling strategy for operator parallelism (Me-Stream). It supports self-adjustment during runtime, can effectively optimize resource allocation and ensure the smooth operation of the system. In this paper, all the three aspects of Me-Stream are discussed, summarized as follows:

(1) We provide a formal definition of the elastic scaling strategy for operator parallelism, and realize the complete process of self-adjustment in operation.
(2) We design the architecture of the parallelism strategy for elastic scaling operations to solve new bottlenecks caused by pluggable scheduling.
(3) We evaluate the optimization performance of the strategy by metrics of execution latency and process latency on Storm to demonstrate the effectiveness of the proposal.

1.2 Paper Organization

The rest of the paper is organized as follows. In Sect. 2, Me-Stream, together with a model for intelligent tuning solution are introduced. Section 3 focuses on the detailed discussion of Me-Stream and the algorithm design, where a machine learning model is adopted to find the better parallel migration path and resource allocation without manual intervention. Section 4 introduces the experimental environment, parameter

settings and performance evaluation of Me-Stream. Section 5 reviews related work on runtime elastic optimization of parallelism in distributed systems. Finally, conclusions and future work are presented in Sect. 6.

2 Me-Stream Architecture

This section mainly focuses on the parallelism optimization of streaming application topology for dynamic data streams. An intelligent optimization solution to the parallelism of running instances without manual intervention is provided. The proposal is to solve the inability of self-adjustment during operation after the relevant parameter configuration is determined.

As shown in Fig. 1, first of all, at the runtime, a monitoring process needs to obtain bolts related data in real time [13]. The data set can be obtained through IO or crawlers. Then, based on the flow perception, the data set is cleaned in real time and output to the parallel degree bottleneck identification to obtain the bottleneck level. When the preset conditions are met, the monitoring process executes rebalance to redistribute slots. The whole process does not require manual intervention. Storm's default scheduling does not consider inter-process optimization or inter-node optimization, which will result in poor configuration of instance parameters with the same computing resource consumption [14, 15]. Through the parallelism bottleneck identification, the topology bottleneck level can be identified, then it is passed into the parameter plan generation together with all-slots. A topology parameter plan is created, and the resources are reallocated according to the default schedule.

At this time, if a scheduling with a large overhead is produced, it is likely to become a new bottleneck. Therefore, it is necessary to design a matching instance scheduling that has better performance than the default scheduling of Storm on the basis of generalization. Through the parameter migration conversion, the topology parameter plan is converted into a migration plan and stored in the routing table. By now, the resource reallocation is completed according to the migration plan.

An intelligent tuning solution model is designed to solve the problem of the parallelism of running instances without manual intervention. The whole process is as follows:

(1) Obtain relevant data of bolts;
(2) Identify parallel degree bottleneck;
(3) Generate parameter plan;
(4) Conduct parameter migration and transformation;
(5) Schedule instances;
(6) Execute the rebalance command;
(7) Complete resource redistribution.

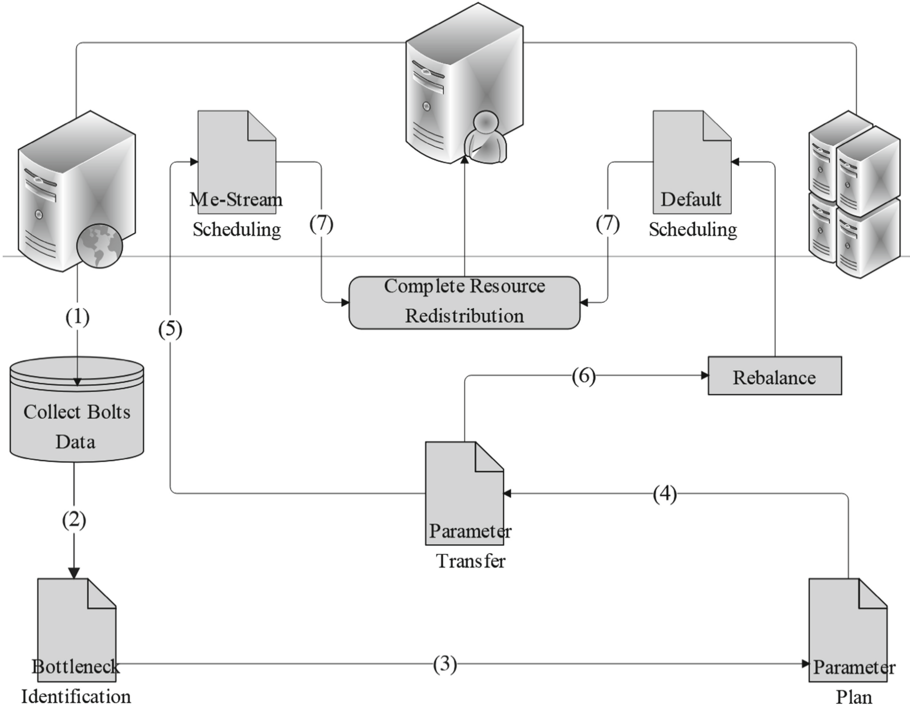

Fig. 1. Elastic scaling process of operator parallelism strategy (Me-Stream).

3 Me-Stream Framework

This section introduces in detail the processes of parallelism bottleneck identification, parameter plan generation, parameter migration, instances scheduling and how to complete the parallelism optimization for running instances using the elastic scaling strategy (Me-Stream).

3.1 Parallel Bottleneck Identification

First, the strategy traverses the nodes and the executed tasks in each topology in turn, quantifies the bottlenecks existing in the current topology through the execution latency, and calculates the maximum execution latency as the bottleneck. After all the topology traversal is completed, the bottleneck levels are sorted according to the execution latency, from the highest to the lowest. Among them, T_{calc} represents the calculation time for task c_j on node n_i, m function represents the required processing power under the complexity of current task, x function represents the complexity of the calculation task, and p function represents the ability of the assigned executor to process n_i. The preliminary deduction formula is defined by (1).

$$T_{calc}(n_i, c_j) = \frac{m(x(c_j))}{p(n_i)}. \tag{1}$$

Secondly, considering the communication bottleneck factors between nodes in different network environments, the strategy sequentially traverses the nodes and the executed tasks in each topology in different network environments, quantifies the bottlenecks in the current topology through the process latency, and calculates the maximum process latency as the bottleneck. After all the topology traversal is completed, the bottleneck levels are sorted according to the process latency. Among them, T_{comm} represents the communication time from n_{i-1} to n_i and n_i to n_{i+1}, m function represents the required processing capacity under the complexity of the calculation task, and l represents transmission link bandwidth. The preliminary deduction formula can be described by (2).

$$T_{comm}(n_i, c_j) = \frac{m(c_j)}{l_{i-1,i}} + \frac{m(c_j)}{l_{i,i+1}} = \frac{m(c_j)(l_{i-1,i} + l_{i,i+1})}{l_{i-1,i}l_{i,i+1}}. \tag{2}$$

In summary, the formula for calculating the sum of the parallel bottleneck time of tasks on all nodes is described by (3) (the maximum of calculation time and the communication time is the bottleneck time).

$$T = \max\begin{cases} Sum(T_{calc}(n_i, c_j)), \\ \frac{Sum(T_{comm}(n_i,c_j))}{2}. \end{cases} = \max\begin{cases} Sum(\frac{m(x(c_j))}{p(n_i)}), \\ \frac{Sum(m(c_j)(l_{i-1,i}+l_{i,i+1}))}{2l_{i-1,i}l_{i,i+1}}. \end{cases} \tag{3}$$

The above explains how to identify the main bottlenecks from communication bottlenecks and calculation bottlenecks in a cluster environment.

Next, we need to know when to perform the reallocation. In order to know this threshold accurately, we design a threshold identification function based on the linear regression. The specific steps are as follows:

(1) First, according to the above parallel bottleneck identification method, a first-order binomial linear regression equation is created. The data set (t, γ) is obtained by collecting, classifying, and labeling the original data (original data is obtained through crawlers and hooks), where t represents the original data timestamp, and γ represents the average delay at timestamp t.

$$\frac{1}{m}\sum\nolimits_{i=1}^{m} (f(t) - \gamma_i)^2 = e(f, \gamma). \tag{4}$$

Where $f(t)$ is the threshold identification function, γ_i is the actual value, $e(f, \gamma)$ is the mean value distribution, defined as the mean error. The smaller the mean error, the more accurate $f(t)$ is. It is the linear regression function produced on the training set.

(2) Then, according to the principle of linear regression:

$$\frac{1}{m}\sum\nolimits_{i=1}^{m}(wt_i + b - \gamma_i)^2 = e(f, \gamma). \tag{5}$$

(3) Next, the partial derivatives of w and b can be obtained by the linear regression function of the first-order binomial linear equation:

$$w = \frac{\sum_{i=1}^{m}\gamma_i(t_i - \bar{t})}{\sum_{i=0}^{m}t_i^2 - \frac{1}{m}\left(\sum_{i=1}^{m}t_i\right)^2}, b = \frac{1}{m}\sum\nolimits_{i=1}^{m}(\gamma_i - wt_i). \tag{6}$$

(4) Finally, using the least squares method to calculate the w and b. when the sum of the Euclidean distance between the training set and the fitted linear labeling function is the smallest, the labeling function is the threshold identification function.When the fitting function becomes stable, the non-monitoring period can be entered, which can effectively reduce training overhead and release resources. The threshold identification function is associated with topologies and can be cached. Therefore, each threshold identification function does not depend on the selection of the training set, and can be used in parallel with the operator of another system in the current cluster. However, each threshold identification result is generated in the current topology instance and destroyed at the end of the topology's life cycle.

(5) After obtaining threshold identification function, Me-Stream records the reference bottleneck by comparing the value of the threshold identification function f and the actual value γ in real time. If the mean error $e(f, \gamma)$ between the value of the function and the actual value is positive under the accuracy requirement, record the value as an effective bottleneck value. We take the maximum effective bottleneck value in the bottleneck interval as the reference bottleneck (the bottleneck time interval depends on the data set interval and automatic redistribution time setting. The default is 1 min).

(6) When the reference bottleneck occurs multiple times in an interval and the bottleneck time obtained by the threshold identification function is in the same order of magnitude, reallocation is performed.

Algorithm 1 Bottleneck Identification Algorithm

Input:
n_i, c_j: Node i,task j.
w, b: partial derivative w,b.
T: the maximum bottleneck time between n_{i-1}.and n_i.
l: Transmission link bandwidth.
$T_C(n_i, c_j)$: the average computing time for task j on node i.
$T_t(n_i, c_j)$: the sum of the average transmission time from the preceding node to the succeeding node.
$x(c_j)$: the complexity of computing task j.
$m(x(c_j))$: the function that calculates the amount of processing power required under the complexity of the task.
Tr,$e(f, y)$: Reference bottleneck linear regression function.
$T_{bottleneck}$: Bottleneck time list.
Output: $l_{bottleneck}$:Bottleneck level list.

1: **procedure** Bottleneck Identification Algorithm
//Training function
2: **Calculate** $\frac{1}{m}\sum_{i=1}^{m}(f(x) - y_i)^2 = e(f, y)$
3: **from** $\frac{1}{m}\sum_{i=1}^{m}(wx_i + b - yi)^2 = e(f, y)$
4: **Calculate** $w = \frac{\sum_{i=1}^{m} y_i(x_i - \bar{x})}{\sum_{i=0}^{m} x_i{}^2 - \frac{1}{m}(\sum_{i=1}^{m} x_i)^2}$, $b = \frac{1}{m}\sum_{i=1}^{m}(y_i - wx_i)$
//Cycle comparison
5: **while**(START FLAG)
6: **Calculate** $T_C(n_i, c_j) = \frac{m(x(c_j))}{p(n_i)}$
7: **Calculate** $T_t(n_i, c_j) = \frac{m(cj)}{l_{i-1,i}} + \frac{m(cj)}{l_{i,i+1}} = \frac{m(cj)(l_{i-1,i} + l_{i,i+1})}{l_{i-1,i} l_{i,i+1}}$
8: $T = \max(T_C, T_t)$
9: $Tr = e(f, y)$
//Compare T AND Tr
10: **if** T/Tr < 10 then
//Judge the magnitude of the same number
11: $T_{bottleneck}[i] = T$
12: i++
13: **end if**
14: **end while**
//After sorting,
//the sequence number is regarded as the bottleneck level in turn
15: SORT($T_{bottleneck}$) GET $l_{bottleneck}$
16: **return** $l_{bottleneck}$

3.2 Parameter Plan Generation

Bottleneck level priority: The task with the highest bottleneck level gets slots allocated first, then the remaining slots are allocated in turn to tasks with lower bottleneck levels.

This allocation strategy considers the weight of bottleneck level more, and is suitable for situations where the difference of bottleneck time between topologies is large. The bottleneck time is calculated for different topologies. Each topology calculates the bottleneck time and then sorts them globally.

$$\overline{N}_{Task} = \varphi \frac{N_{Executer} + N_{Bottleneck}}{N_{Executer}} N_{Task}. \tag{7}$$

Parameter planning priority: According to the bottleneck levels from high to low, the previous executor number is added to the bottleneck level multiplied by the coefficient (default 1). At the same time, the number of tasks is increased by the corresponding multiple times. This allocation strategy controls the weight of the bottleneck level by a coefficient φ, and is suitable for situations where the bottleneck time between topologies has little difference. The bottleneck level and parameter schedule on the example WordCount instance are as follows (Table 1):

Table 1. Bottleneck level and parameter schedule on the WordCount instance

Topology	Worker number	Executor number	Task number	Bottleneck level
T1	3	8	16	4
T2	5	10	10	2
T3	3	5	10	1
T4	6	10	20	3

Algorithm 2 Parameter Plan Algorithm

Input:
N_{Task}: Number of resources allocated to previous tasks.
$\overline{N}_{Task}$: Number of resources allocated to current tasks.
$l_{bottleneck}$: Bottleneck level list.
φ: Allocation coefficient
Output: $\overline{N}_{Task}$: Transmission link bandwidth.

1: **procedure** Parameter Plan Algorithm
//Add the previous number of executors to the number of bottleneck levels multiplied by a factor (default 1)
2: **Calculate** $\overline{N}_{Task} = \varphi \frac{N_{Executer} + l_{Bottleneck}}{N_{Executer}} N_{Task}$
3: **return** $\overline{N}_{Task}$

3.3 Parameter Migration and Conversion

Parameter migration conversion is conducted based on parameter planning. Its process is as follows:

(1) When Me-Stream program is started, the table columns N(k,v) and P(k,v) will be created automatically;
(2) At runtime, the current node and port are saved into the corresponding keys;
(3) Before redistribution, a new operator allocation is generated based on the parameter schedule;
(4) After completing the allocation, the node and port from the new allocation result are assigned to replace the corresponding value in the routing table;
(5) After N (k, v) and P (k, v) are updated, they are provided as a migration path on the example WordCount instance to the new scheduling (Table 2).

Table 2. Parameter plan and migration path on the WordCount instance

Topology	Executor number	Task number	Operator number	Slots		Migration path	
				Node	Port	N (k,v)	P (k,v)
T1	1	1	{ [1, 2]…}	S1	6700	(1, 3)	(1, 3)
	1	2		S1	6700	(1, 3)	(1, 3)
…							
T2	2	3	{…[3]…}	S2	6701	(2, 2)	(2, 1)
…							
T3	3	9	{…[9, 10]…}	S3	6702	(3, 1)	(3, 2)
	3	10		S3	6702	(3, 1)	(3, 2)

Algorithm 3 Migration Algorithm

Input:
k, v: Key,value.
n_i, p_j: Node i,port j.
$\bar{n}_i, \bar{p}_j$: Node i,port j.
Output: N(k,v): Node migration path.
P(k,v): Port migration path.

1: **procedure** Migration Algorithm
//During operation, the current node and port are stored in the corresponding keys.
2: **while**(!EMPTY)
3: N(k, v).put(n_i, $\bar{n}_i$)
4: P(k, v).put(p_j, $\bar{p}_j$)
5: i++
6: j++
7: **end while**
8: **return** N(k,v),P(k,v)

3.4 Instances Scheduling

Bottlenecks may be created during the optimization process because of the computing and communication bottlenecks on heterogeneous cluster nodes, the stateful and stateless instances at the instance layer [16, 17], and some complex pluggable scheduling. As such, an instance scheduling that can directly identify the migration table is designed, and the corresponding configuration is provided as the default setting. The specific instance scheduling steps are executed as follows:

(1) Call the cluster's *needsSchedualerTopologies* method to obtain the topology that needs to be assigned with tasks, and store all the topologies in the keys of N (k, v) and P (k, v) according to the bottleneck level.
(2) Call the cluster's *getAvailableSlots* method to obtain the resources available in the current cluster, return them in the form of a collection of <node, port>, and allocate them to available slots.
(3) Call the cluster's *compute-executors* method to convert the topological executor information into a collection of <start-t ask-id, end-task-id> and store it in all executors.
(4) Call the *getAliveAssignedNodeAndPort* method of *eventScheduler* to obtain the resources acquired by the current topology, and return the <node + port, executor> collection and store it in *alive-assigned*.
(5) Call the overriding *slot-can-ressign* method in Me-Stream to determine whether the Slots information is active, then select the slot that can be reassigned and store it in the *can-ressigned* variable.
(6) Call the overriding *bad-slot* method in Me-Stream to calculate the number of slots that can be released in the current topology. If it is greater than the number of slots currently allocated, call the cluster's *freeSlots* method to release them.
(7) Call the *migration-path* method in Me-Stream and allocate all execution programs based on the N (k, v) and P (k, v) records calculated by all topologies before scheduling.

4 Performance Evaluation

In this section, the experimental environment and parameter settings are first discussed, followed by the analysis of performance evaluation results.

4.1 Experimental Environment and Parameter Settings

The proposed Me-Stream system is implemented on Storm 2.1.0, and installed on top of Ubuntu 20.04.1. Real-life data experiments are conducted on the computing cluster at Alibaba Cloud Computing. The cluster consists of 28 machines, with 1 designated machine serving as the master node, running Storm Nimbus, 2 designated as Zookeeper nodes, and the rest 25 machines working as Supervisor nodes. The software configuration of Me-Stream platform is shown in Table 3.

Table 3. Software configuration of Me-Stream.

Software	Version
OS	Ubuntu 20.04.1 64bit
Storm	Apache-Storm-2.1.0
JDK	Jdk1.8 64bit
Zookeeper	Zookeeper-3.4.14
Kafka	Kafka-2.3.0
Redis	Redis-6.0.5

Moreover, one DAG with WordCount function is submitted to the computing cluster. The logic graph of WordCount is shown in Fig. 2.

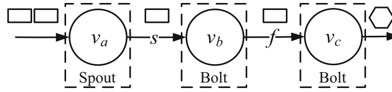

Fig. 2. Logical graph of WordCount in Me-Stream

In Storm, the WordCount instance is used to simulate random words input into Spout through Kafka, and messages from different partitions are evenly distributed to different executors for consumption. When Spout parallelism is set to 1, there is no need to adjust the parameters. Therefore, our focus is to test the system performance when the spout has multiple executors. Under normal circumstances, the Capacity value range is between 0.0x and 0.2. When the value is close to 1, it indicates that the load is severe and the degree of parallelism needs to be increased. At the same time, when the failure value is not 0, it means that the load is serious and there are tuples that experience failure or time out. At this time, the parallelism of Spout should be increased accordingly. We simulate a normal situation where the Capacity value is small and Failure value is 0. The following describes the experimental verification in detail, and the parameter table applied in the entire experimental process is shown in Table 4.

Table 4. Table of parameter settings in the experiments.

Parameter	Explain
Emitted	Number of tuples launched to date
Transferred	Number of tuples successfully transferred to the next bolt to date
Complete latency (ms)	The average time taken for each tuple to be fully processed in tuple tree to date
Acked	Number of tuples successfully processed to date
Failed	Number of tuples failed or timed out to date

4.2 Performance Results

We consider the average delay data set of topologies within 38min $\sim$ 50min under the default Storm scheduling strategy and Me-Stream optimization strategy for comparison. The experimental settings contain two evaluation parameters: execute latency EL and process latency PL.

(1) Execute latency.
Execute latency reflects the overall execution time for all running DAGs, and it is evaluated by the timestamp from the execution of the function to the end of per DAG. The smaller the execution latency, the stronger the data processing ability of the elastic stream computing system.

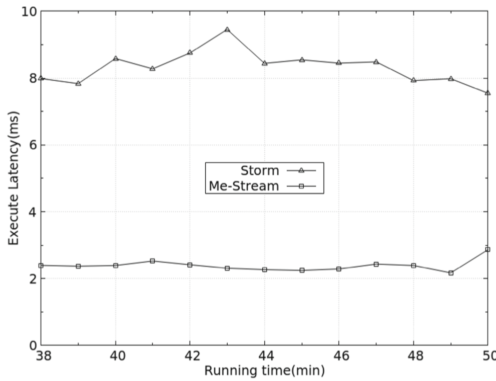

Fig. 3. Comparison of execute latency between the default scheduler and Me-Stream on the WordCount instance.

When the data input rate is stable, Me-Stream has a lower execution latency comparing to the *DefaultScheduler* on Storm platform. As shown in Fig. 3, with the capacity remains unchanged during the whole process, the average execute latency by Me-Stream and by the default scheduler at the stable stage are 2.3886 ms and 8.3267 ms, respectively. It demonstrates that the execution latency by Me-Stream is lower than that of the default scheduler on the given instance when the input rate is stable.

(2) Process latency.
Process latency reflects the overall processing time for all running DAGs, and it is evaluated by the timestamp of each DAG passed from the tuple arrival to the ack. The smaller the processing latency, the stronger the data processing ability of the elastic stream computing system.

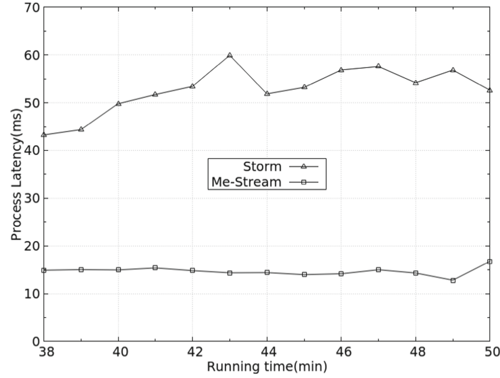

Fig. 4. Comparison of process latency between the default scheduler and Me-Stream on the WordCount instance.

When the data input rate is stable, Me-Stream has a lower process latency comparing to the *DefaultScheduler* on Storm platform. As shown in Fig. 4, with the capacity remains unchanged during the whole process, the average process latency by Me-Stream and by default Storm strategy at the stable stage are 14.6867 ms and 52.7333 ms, respectively. It demonstrates that the process latency by Me-Stream is lower than that of the default Storm strategy on the given instance when the input rate is stable.

We also respectively collect statistics on execute delay, process delay and total delay data sets of the *DefaultScheduler* and the Me-Stream optimization strategy, as shown in Fig. 5 and 6.

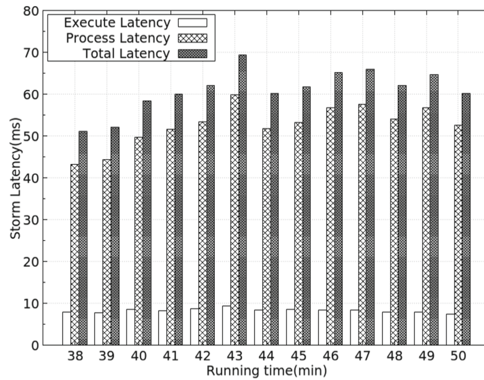

Fig. 5. Statistics of execute delay, process delay and total delay data sets of the default scheduler on the WordCount instance.

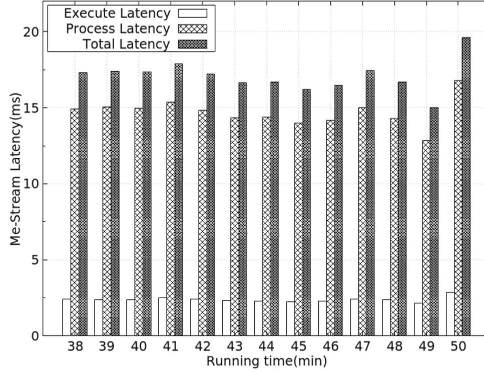

Fig. 6. Statistics of execute delay, process delay and total delay data sets of the Me-Stream optimization strategy on the WordCount instance.

5 Related Work

The application of machine learning models can produce better parallel migration paths and better resource allocation without manual intervention. However, the time-consuming training process greatly limits the efficiency of machine learning methods, and the inconsistency of state and data can also cause considerable overhead. Researchers have been trying to address these issues.

In [18], a double exponential smoothing method was proposed to predict abnormal events, which solves the shortcoming of the Markov model that requires a training process. By designing a seven-phase protocol for traffic-aware active migration, it handles the inconsistency of state and data in the load balancing partition.

In [19], a pipeline data processing model based on streaming applications was mentioned. When the ratio of input data to output data of upstream neighbor operations is known, the input data of downstream neighbor operations can be obtained in advance. The linear relationship is obtained through learning and analysis, and the average value of the probability distribution during the monitoring period is taken. The concept of the average value of the probability distribution during the monitoring period is also added to the original algorithm, which can effectively reduce the error of the data set and the function value, and improve the efficiency and accuracy of training. When the fitting function becomes stable, the non-monitoring period can be entered, which can effectively reduce training overhead and release resources.

In [20], the ideas of learning rate and discount factor were introduced on the basis of fitting. Data sets that have a greater impact on the data stream are stored in the evaluation table. When data with a large influence offset continuously appears, its weight can be added to influence according to the evaluation result, so as to achieve the purpose of better training the result function.

In [21], a cost-effective resource allocation model was proposed. Its purpose is to allow users to automatically and efficiently deploy applications in local or cloud clusters, and developed a profiler for Spark, which can analyze applications in actual

clusters according to different resource allocation schemes and input workloads. Based on the application profile received from the profiler, dSpark uses the proposed resource allocation model to select a cost-effective resource allocation plan based on the deadline in order to deploy the application to the cluster.

The above prior works provide valuable insights into the potential solutions to the static parallelization setting problems using elastic strategies of machine learning. However, for big data stream applications, innovative methods need to be developed, and the characteristics specific to the big data flow computing environments need to be considered when exploring elastic non-manual intervention. A summary of the comparison between our work and other closely related works is given in Table 5.

Table 5. Comparison of Me-Stream and related work

Parameter	Related work				Me-Stream
	[18]	[19]	[20]	[21]	
Versatility	✗	✓	✓	✗	✓
Parallelism	✓	✓	✓	✓	✓
Machine learning	✗	✓	✓	✗	✓
Cost saving	✗	✓	✗	✓	✓
Resource saving	✗	✗	✓	✓	✓

6 Conclusions and Future Work

In this paper, an elastic scaling strategy for operator parallelism Me-Stream is proposed. It can intelligently perform instance parallelism without manual intervention at runtime. Starting from the Storm Instance parameter level, we first initiate a monitoring process to obtain the bolts-related data in real time through traffic sensing, then analyze and use them, followed by self-optimizing the resource allocation from time to time. This paper mainly solves the following problems:

(1) It is not transparent for Storm users to use API to set parallelism for operators in a topology at runtime, that is, users need to run the API frequently to change the configuration of their applications.
(2) Storm users may not know how to optimally adjust the parallelism. We use a machine learning model to achieve a better parallel migration path. The model can achieve a better effect in terms of resource allocation without manual intervention, and has a certain learning ability.
(3) Storm distributes instances to work programs and work program nodes in a round-robin manner by default. The number of configured work programs is still evenly distributed. The instance scheduling we designed can achieve better compatibility with the intelligent tuning scheme under the premise of ensuring good generalization.

Future work will focus on the following aspects:

(1) Adapt Me-Stream to other big data stream computing environments.
(2) Deploy Me-Stream in a real big data stream computing environment.

Acknowledgements. This work is supported by the National Natural Science Foundation of China under Grant No. 61972364, the Fundamental Research Funds for the Central Universities under Grant No. 2652021001, and Melbourne-Chindia Cloud Computing (MC3) Research Network.

References

1. Cao, H., Wu, C.E.Q., Bao, L., Hou, A., Shen, W.: Throughput optimization for Storm-based processing of stream data on clouds. Future Gener. Comput. Syst. **112**, 567–579 (2020)
2. Paris, C., Stephan, E., Gyula, F., Seif, H., Stefan, R., Kostas, T.: State management in Apache Flink: consistent stateful distributed stream processing. Proc. VLDB Endow. **10**(12), 1718–1729 (2017)
3. Apache, Storm. http://storm.apache.org
4. Flink. https://flink.apache.org/
5. Spark Streaming. https://spark.apache.org/streaming/
6. Samza. http://samza.apache.org/
7. Apex. https://apex.apache.org/
8. Google Cloud Dataflow. https://cloud.google.com/dataflow/
9. Deng, S., Wang, B., Huang, S., Yue, C., Zhou, J., Wang, G.: Self-adaptive framework for efficient stream data classification on storm. IEEE Trans. Syst. Man Cybern. Syst. **50**(1), 123–136 (2020)
10. Li, C., Zhang, J., Luo, Y.: Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm. J. Netw. Comput. Appl. **87**, 100–115 (2017)
11. Muhammad, A., Aleem, M., Islam, M.A.: TOP-Storm: a topology-based resource-aware scheduler for Stream Processing Engine. Cluster Comput. **24**(1), 417–431 (2020). https://doi.org/10.1007/s10586-020-03117-y
12. Pathan, R., Voudouris, P., Stenstrom, P.: Scheduling parallel real-time recurrent tasks on multicore platforms. IEEE Trans. Parallel Distrib. Syst. **29**(4), 915–928 (2018)
13. Li, H., Wu, J., Jiang, Z., Li, X., Wei, X.: Task allocation for stream processing with recovery latency guarantee. In: Proceedings of the 2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, pp. 379–383. IEEE Press,September 2017
14. Zhang, J., Li, C., Zhu, L., Liu, Y.: The real-time scheduling strategy based on traffic and load balancing in storm. In: Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications, HPCC 2016, pp. 372–379. IEEE Press, January 2017
15. Muhammad, A., Aleem, M.: A3-Storm: topology-, traffic-, and resource-aware storm scheduler for heterogeneous clusters. J. Supercomput. **77**(2), 1059–1093 (2020). https://doi.org/10.1007/s11227-020-03289-9
16. You, Y., Demmel, J.: Runtime data layout scheduling for machine learning dataset. In: Proceedings of the 46th International Conference on Parallel Processing, ICPP 2017, pp. 452–461. IEEE Press,September 2017

17. Al-Sinayyid, A., Zhu, M.: Job scheduler for streaming applications in heterogeneous distributed processing systems. J. Supercomput. **76**(12), 9609–9628 (2020). https://doi.org/10.1007/s11227-020-03223-z
18. Cheng, D., Wang, Y.: Adaptive scheduling parallel jobs with dynamic batching in spark streaming. IEEE Trans. Parallel Distrib. Syst. **29**(12), 2672–2685 (2018)
19. Wei, X.: Pec: proactive elastic collaborative resource scheduling in data stream processing. IEEE Trans. Parallel Distrib. Syst. **30**(7), 1628–1642 (2019)
20. Wang, W., Zhang, C.:An on-the-fly scheduling strategy for distributed stream processing platform. In: IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (2018)
21. TawfiqulIslam, M., Karunasekera, S., Buyya, R.: dSpark: deadline-based resource allocation for big data applicationsin apache spark. In: IEEE 13th International Conference on e-Science, 24–27 October 2017