Journal of Network and Computer Applications 206 (2022) 103462

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

A state lossless scheduling strategy in distributed stream computing systems

Minghui Wu [a], Dawei Sun [a,*], Yijing Cui [a], Shang Gao [b], Xunyun Liu [c], Rajkumar Buyya [d]

[a] School of Information Engineering, China University of Geosciences, Beijing, 100083, PR China
[b] School of Information Technology, Deakin University, Waurn Ponds, Victoria 3216, Australia
[c] Artificial Intelligence Research Center, National Innovation Institute of Defense Technology, Beijing, 100071, PR China
[d] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Keywords:
Stream computing
Online scheduling
State management
Bipartite graph
Hierarchical migration

ABSTRACT

Stateful scheduling is of critical importance for the performance of a distributed stream computing system. In such a system, inappropriate task deployment lowers the resource utilization of cluster and introduces more communication between compute nodes. Also an online adjustment to task deployment scheme suffers slow state recovery during task restart. To address these issues, we propose a state lossless scheduling strategy (Sl-Stream) to optimize the task deployment and state recovery process. This paper discusses this strategy from the following aspects: (1) A stream application model and a resource model are constructed, together with the formalization of problems including subgraph partitioning, task deployment and stateful scheduling. (2) A multi-factor topology partitioning method is proposed using a quantum particle swarm algorithm. The assignment between tasks and nodes is optimized using a bipartite graph minimum matching algorithm. (3) A hierarchical local topology migration is performed when an online scheduling is triggered, which ensures the processing sustainability of data streams. (4) A fragment loss-tolerant jerasure tool is used to divide the state data into fragments and periodically save them in upstream vertex instances, which ensures the available fragments be able to reconstruct the whole state in parallel. (5) Metrics including latency, throughput and state recovery time are evaluated in a real distributed stream computing environment. With a comprehensive evaluation of variable-rate input scenarios, the proposed Sl-Stream system provides promising improvements on throughput, latency and state recovery time compared to the existing Storm's scheduling strategies.

1. Introduction

In the era of big data, emerging real-time applications are becoming increasingly complex and applied in various areas, such as industrial automation and robotics, real-time recommendations and business monitoring (Alghamdi et al., 2017). Most of the applications emphasize real-time and accuracy, e.g., a millisecond response must be provided given the input data. To better support applications in real-time streaming environments, a series of streaming computing frameworks have emerged, such as Spark streaming (spark, 2022), Twitter heron (Twitter, 2022), Apache Flink (Apache, 2022a), Samza (Apache, 2022b) and Apache Storm (Apache, 2022c). Among them, Storm is the one that is more suitable for real-time data processing scenarios and has advantages (Sainik and Khajuria, 2014) in fault tolerance, message handling, transaction management and development testing.

High system throughput and low system response time are two key performance metrics for stream computing systems (Gedik et al., 2014). Application scheduling (Li et al., 2017a, 2019a) plays an important role in achieving these goals. As usually requiring multi-processor systems to implement, the applications need to be modeled as Directed Acyclic Graphs (DAGs) (Marchal et al., 2018; Fu et al., 2021) to capture the task dependencies, before an event scheduler steps in to allocate topological tasks properly to each compute node. Different scheduling methods have been proposed by researchers. EvenScheduler (Apache, 2022c) is an example of event scheduler and has been built into the Storm platform. ResourceAwareScheduler (Apache, 2022c) allocates resources to each user. When a cluster has additional free resources, these resources can be allocated to users in a fair manner. P-Scheduler (Eskandari et al., 2016) assigns heavily communicating tasks to the same compute node to reduce the network latency. MT-Scheduler (Al-Sinayyid and Zhu, 2020) first maps a stream application to a DAG, then minimizes communication latency and computation latency by a dynamic programming algorithm for tasks on critical paths, and finally uses a greedy algorithm to place tasks on non-critical paths sequentially. GFP-Scheduler (Pathan et al., 2018) assigns priority to DAG tasks and their

* Corresponding author.
E-mail addresses: wuminghui@cugb.edu.cn (M. Wu), sundaweicn@cugb.edu.cn (D. Sun), cuiyijing@cugb.edu.cn (Y. Cui), shang.gao@deakin.edu.au (S. Gao), xunyunliu@gmail.com (X. Liu), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.jnca.2022.103462
Received 20 December 2021; Received in revised form 4 April 2022; Accepted 7 July 2022
Available online 13 July 2022

subtasks. In addition, a task-level scheduler and subtask-level scheduler are applied to determine the prioritized ready tasks and execution tasks, respectively.

However, when a rescheduling event is triggered, some schedulers choose to kill the entire topology and restart, which might not be the best solution as it causes the loss of topology state (Rho et al., 2017). The topology will no longer process data until all the topological instances are restarted and their states are recovered. Some other schedulers choose to adjust the instances of a topology online. The adjustment to the instances deployed on a node may result in state loss because of the influence of factors such as network packet loss or node failure. It may also trigger a slow state recovery. If there were a better state management method to locally adjust the topological instances during the rescheduling phase and perform fast recovery on these instances, the system performance might be improved. These thoughts motivate our research on efficient state lossless scheduling strategy.

A state lossless scheduling strategy is expected to determine when and how to reschedule the vertices of a running DAG based on the fluctuating data streams and resource consumption, and manage the state of vertices efficiently. To achieve these, we first obtain the communication load between tasks, the resource consumption of tasks and the resource consumption of nodes. Then Sl-Stream reschedules the local vertices of the DAG without killing the entire topology. It supports parallel state recovery of the vertices to reduce the impact of rescheduling on the system. Sl-Stream also ensures the reliability of state data by introducing erasure coding technology. The objectives of continuity of data stream processing, reliability of state data and fast recovery of task states can therefore be achieved by Sl-Stream to some extent.

1.1. Contributions

A state lossless scheduling strategy (Sl-Stream) is proposed to improve the throughput and latency of a distributed stream computing system. Our contributions are summarized as follows:

(1) A general stream application model and a resource model are provided, along with the formalization of problems including subgraph partitioning, task deployment and stateful scheduling.
(2) A multi-factor graph partitioning method is proposed, which uses a quantum particle swarm algorithm to divide a DAG into subgraphs. Also a one-to-one matching model between subgraphs and nodes is constructed using the bipartite graph minimum matching algorithm.
(3) A hierarchical local topology migration is implemented for rescheduling, which ensures the processing sustainability of data streams.
(4) A fragment loss-tolerant jerasure tool is used to divide the state data into fragments. The fragments are periodically saved in upstream vertex instances and the state can be reconstructed by the available fragments in parallel when needed.
(5) Metrics such as system latency, response time and state recovery time are evaluated to verify the effectiveness of the proposed state lossless scheduling strategy.

Experimental evaluations are conducted on real-world data and the results prove the effectiveness of the strategy.

1.2. Paper organization

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 introduces the system model, the stream application model and the resource model. Section 4 formalizes the problems of subgraph partitioning, task deployment and stateful scheduling. Section 5 introduces the optimization methods to address the problems identified in Section 4. Section 6 explains the framework and main algorithms of Sl-Stream. Section 7 evaluates the performance of Sl-Stream and Section 8 concludes the paper.

2. Related work

In this section, we review state-of-the-art work in two related areas: task scheduling and state management for stream processing. A comparison between our work and the relevant research is summarized in Table 1.

2.1. Task scheduling for stream processing

In stream computing systems, a better scheduling strategy is essential for improving throughput and reducing latency. Many researchers focus on the deployment of stream applications. However, the communication load between tasks and the resources on nodes can change due to fluctuations of data streams. It is challenging to find an optimal deployment.

A task scheduling algorithm (Li et al., 2019b) based on deadline constraints was proposed, where a classification scheduling strategy was used to ensure the priority of urgent tasks and four switching strategies were proposed for urgent tasks.

Focusing on the workflow scheduling problem in cloud environments, the authors proposed a multi-objective optimization algorithm (Ebadifard and Babamir, 2018) with diversity criterion based on an extension of the black hole heuristic. It improves the resource utilization efficiency of the system and eases the load imbalance problem.

A single-objective firefly algorithm (Ebadifard et al., 2018) was proposed that allows for appropriate task deployment based on the processing power of each virtual machine. The effectiveness of the algorithm is evaluated by modeling, and the results show that the algorithm can improve the resource utilization and reduce the task completion time of the system.

A dynamic container resource allocation mechanism (CRAM) (Runsewe and Samaan, 2019) was proposed. A game-theoretic approach is used to distribute the workload to the corresponding machines in a cluster to maximize the overall performance of the system.

A new dynamic scheduling technique (Barika et al., 2021) was proposed . It combines GA and two-level greedy algorithm to meet the dynamic nature of data stream and minimize execution costs.

A dynamic task scheduling scheme (Ebadifard et al., 2021) was proposed, which adopts the Hone: Mitigating Stragglers in Distributed StreamProcessing With Tuple Schedulingybee algorithm to optimize load balancing, reduce makespan, increase resource utilization and improve the reliability of the system.

A new list scheduling algorithm (Djigal et al., 2021) assigned task topologies to a heterogeneous network in order to minimize the scheduling length. There are two main processes: task prioritization phase and processor selection phase. However, this algorithm may cause resources idle in the data center.

A directed acyclic graph scheduling algorithm (Al-Maytami et al., 2019) was proposed based on predicting task computation time. The computation is simplified by applying Principle Components Analysis. However, the algorithm is suitable for static scheduling, i.e., assuming that the speed at which the task arrives at the processor is known.

An ant colony algorithm (SP-Ant) (Farrokh et al., 2022) was used to find the best operator assignment plan by considering the inter-node communication latencies of operators in a heterogeneous network. It can arrange highly communicative operators on the same worker node to reduce the system response time. However, it does not consider the fact that the resource consumption of nodes can also impact the processing time of the system. When a rescheduling is triggered, SP-Ant may cause state loss.

An application topology was divided into multiple parts by T3-Scheduler (Eskandari et al., 2018), where each subtopology includes highly communicative tasks, with a size determined by the capacity of compute node in the heterogeneous cluster. Although reducing communication latency is beneficial for system response latency, the resources of nodes are also a metric that cannot be ignored.

Table 1
Related work comparison.

Related works	Aspects			
	Graph partitioning	State lossless migration	State backup	Parallel recovery of state
Wu and Tan (2015)	✗	✔	✔	✔
Cardellini et al. (2016)	✗	✔	✗	✔
Ebadifard and Babamir (2018)	✔	✗	✗	✗
Ebadifard et al. (2018)	✔	✗	✗	✗
Li et al. (2019b)	✗	✗	✗	✗
Runsewe and Samaan (2019)	✗	✗	✗	✗
Zhang et al. (2020)	✗	✔	✗	✔
Zhao et al. (2021)	✗	✔	✗	✔
Our work	✔	✔	✔	✔

In summary, the above solutions provide valuable insights into the scheduling problem. The Sl-Stream system is able to consider multiple factors for graph partitioning and perform node selection for the partitioned subgraphs to improve the resource utilization of the data center.

2.2. State management for stream processing

Effective state management not only improves the system performance, but also improves its reliability. In recent years, many researchers have attempted to optimize the state management for stream processing systems.

Based on a multi-tenant scheduler (Wu and Tan, 2015), the state of each operation was sliced and these slices were backed up and stored to different compute nodes. However, when the original state node and the node storing the backup state slices fail at the same time, it results in missing state.

An automatic elasticity mechanism and a state migration mechanism (Cardellini et al., 2016) were introduced to support the migration of internal state of operators on different nodes. They maintain information integrity and allow concurrent migration of multiple nodes. However, state failure recovery is not considered.

A new stream processing system supporting state access, TStream (Zhang et al., 2020), was introduced by leveraging transactional semantics for concurrent state access. Improved scalability is achieved by adding two designs, i.e. dual-mode scheduling and dynamic restructuring execution.

A stream processing system (Zhao et al., 2021) that supports timestamped state sharing was proposed to make up for the fact that existing stream processing systems only supported state local access.

State management and fault recovery mechanisms (Liu et al., 2020) were introduced. The operators are organized into a ring to ensure that each node has a corresponding neighbor node. The state of each operator is divided into multiple fragments and stored in neighboring nodes. The state of neighboring nodes is checked periodically so that when the state is lost, it can be recovered by different state fragments.

Sl-Stream system can not only restore the state of migrated instances in parallel to improve the system performance, but also perform fault-tolerant backup of state data to improve the system reliability.

3. System model

Before introducing the Sl-Stream strategy and its related algorithms, we first explain the stream application model and the resource model in big data stream computing environment. For the sake of clarity, in Table 2, we summarize the main notations used throughout the paper.

3.1. Stream application model

The logic of each stream application can be represented as a DAG. It is composed of a set of vertices and a set of directed edges, defined as $G = (V(G), E(G))$. $V(G) = \{v_i | i \in 1, \ldots, n\}$ denotes a finite set of n vertices, and each v_i is an operation with a special function. $E(G) = \{e_{u,v} | 1 \leq u, v \leq n \text{ and } u \neq v\}$ is a finite set of directed edges, and the weights associated with the edges are denoted as communication costs. User submits a constructed DAG to a data center, which then creates multiple instances $v_{i,j}$ for each vertex v_i. The instances of the same vertex have the same function and each $v_{i,j}$ has a certain weight that represents the computational cost of $v_{i,j}$.

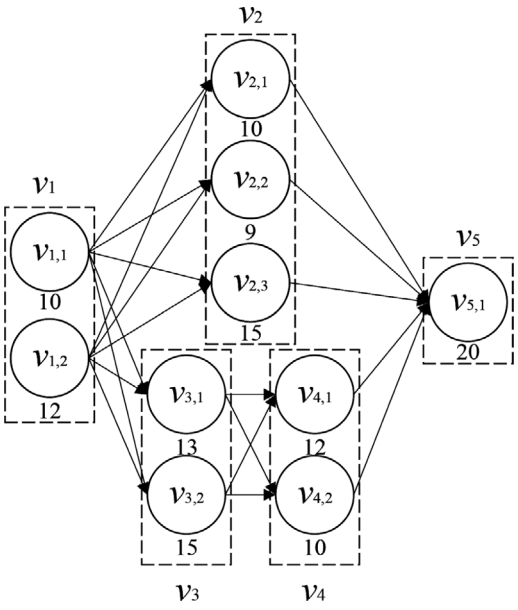


Fig. 1. An example DAG with different number of instances for each vertex.

As shown in Fig. 1, the example DAG has 5 vertices v_1, v_2, v_3, v_4 and v_5. The number of vertex instances of v_1 is 2, including $v_{1,1}$ and $v_{1,2}$. v_2 has 3 instances, i.e. $v_{2,1}, v_{2,2}$ and $v_{2,3}$. v_3 and v_4 have 2 instances each, and v_5 has 1 instance $v_{5,1}$. Each instance has an estimation value of resource consumption, e.g. instances $v_{1,1}$ and $v_{1,2}$ are expected to consume 10 and 12 units of resources.

3.2. Resource model

In a mapped DAG $G = (V(G), E(G))$, if we use edges $E(G)$ to denote the communication among tasks and vertex weights $V(G)$ to denote the computing resources required by tasks, the resource model can be constructed by considering the two factors.

(1) The communication among tasks. We define $r(v_{i,k}, v_{j,m})$ as the data tuple transmission rate per unit time between two instances $v_{i,k}, v_{j,m}$ and it satisfies (1)

$$r(v_{i,k}, v_{j,m}) = \begin{cases} 0, & v_{i,k} \text{ and } v_{j,m} \text{ on the same compute node,} \\ E_r, & \text{otherwise,} \end{cases} \quad (1)$$

Table 2
Description of main symbols used in the Sl-Stream models.

Symbol	Description	Symbol	Description
G	Topology of a streaming application	n_c	Compute node
G_{sub_k}	Sub-topology k of G	L_{n_c}	CPU utilization of n_c
v_i	Vertex i in topology	M_{n_c}	Memory utilization of n_c
$v_{i,j}$	Vertex instance j of vertex i	$R_{v_{i,j},n_c}$	Resource consumption of $v_{i,j}$ running on node n_c
$r(v_{i,k}, v_{j,m})$	Data tuple transmission rate between $v_{i,k}$ and $v_{j,m}$	$R^m_{v_{i,j},n_c}$	Memory utilization of $v_{i,j}$ running on node n_c
E_r	Average transmission rate	$R^c_{v_{i,j},n_c}$	CPU utilization of $v_{i,j}$ running on node n_c
$q_{v_{i,j}}$	Number of data tuples processed by instance $v_{i,j}$	$R^c_{G_{sub_i}}$	Estimated CPU resource of G_{sub_i}
C_{n_c}	Set of instances running on node n_c	$R^m_{G_{sub_i}}$	Estimated memory resource of G_{sub_i}
$size_{G_{sub_i}}$	Size of subgraph G_{sub_i}	p_{old}	Old partition scheme
$r(G_{sub_i}, G_{sub_j})$	Communication load between G_{sub_i} and G_{sub_j}	p_{new}	New partition scheme

where E_r denotes the average transmission rate during time $[t_s, t_e]$, t_s denotes the start time and t_e denotes the end time. There may be transient fluctuation in the arrival rate of data stream and E_r can effectively avoid its effects. E_r can be calculated by (2).

$$E_r = \frac{\int_{t_s}^{t_e} E_{r_t} dt - \max(E_{r_t}) - \min(E_{r_t})}{t_e - t_s}. \tag{2}$$

where E_{r_t} denotes the transmission rate at time t, and $t \in [t_s, t_e]$.

(2) The computing resources required by tasks. In a data center, the resources of a compute node n_c can be measured in different dimensions, such as CPU, memory and I/O (Li et al., 2017b). Our pressure experiments show that when the CPU and memory of a node reach their bottleneck, the I/O resources of the node are not affected much and still remain sufficient. Therefore, we only focus on the CPU and memory utilization of nodes in this paper.

At time t, compute node n_c may run multiple instances. If we denote the set of these instances as C_{n_c}, the CPU real-time utilization of node n_c as L_{n_c}, the number of data tuples processed by each vertex instance $v_{i,j}$ running on node n_c ($v_{i,j} \in C_{n_c}$) at time t as $q_{v_{i,j}}$, the CPU utilization of vertex instance $v_{i,j}$ running on node n_c can be calculated by (3).

$$R^c_{v_{i,j},n_c} = \frac{q_{v_{i,j}}}{\sum_{v_{k,m} \in C_{n_c}} q_{v_{k,m}}} \cdot L_{n_c}, \tag{3}$$

where $R^c_{v_{i,j},n_c}$ denotes the CPU utilization of instance $v_{i,j}$, and $v_{k,m}$ ($v_{k,m} \in C_{n_c}$) denotes one instance on node n_c.

Similarly, at time t, we can also obtain the memory utilization of node n_c, denoted as M_{n_c}. The memory utilization of instance $v_{i,j}$ running on node n_c can be calculated by (4).

$$R^m_{v_{i,j},n_c} = \frac{q_{v_{i,j}}}{\sum_{v_{k,m} \in C_{n_c}} q_{v_{k,m}}} \cdot M_{n_c}, \tag{4}$$

where $R^m_{v_{i,j},n_c}$ denotes the memory utilization of instance $v_{i,j}$ on node n_c.

Therefore, the resources consumed by vertex instance $v_{i,j}$ on node n_c at time t, denoted as $R_{v_{i,j},n_c}$, can be calculated by (5).

$$R_{v_{i,j},n_c} = \alpha \cdot R^c_{v_{i,j},n_c} + (1-\alpha) \cdot R^m_{v_{i,j},n_c}, 0 < \alpha < 1, \tag{5}$$

where α is a weighting factor of the CPU and memory utilization for instance $v_{i,j}$ on node n_c.

4. Problem formulation

In this section, we formalize the scheduling problems in stream computing systems, which mainly include subgraph partitioning, task deployment and stateful scheduling.

4.1. Subgraph partitioning problem

Based on the above models, the subgraph partitioning problem (Jiang et al., 2017; Fischer and Bernstein, 2015) can be described as follows. A stream application $G = \{V(G), E(G)\}$ is deployed to a data center, which consists of m usable compute nodes $\{n_1, n_2, \ldots, n_m\}$. Assume that the number of compute nodes to be used by the stream application G is k and $k <= m$, then the number of subgraphs into which G is partitioned is k. The mathematical model of the subgraph partitioning problem is represented by (6) and (7).

$$G \Leftrightarrow \{G_{sub_1}, G_{sub_2}, \ldots, G_{sub_k}\}, \tag{6}$$

Subject to

$$\begin{cases} \min \sum_{i=1}^{k} \sum_{j=1}^{k} r(G_{sub_i}, G_{sub_j}), \\ \frac{R^c_{G_{sub_i}}}{\sum_{j=1}^{k} R^c_{G_{sub_j}}} \approx \frac{size_{G_{sub_i}}}{\sum_{j=1}^{k} size_{G_{sub_j}}}, \\ \frac{R^m_{G_{sub_i}}}{\sum_{j=1}^{k} R^m_{G_{sub_j}}} \approx \frac{size_{G_{sub_i}}}{\sum_{j=1}^{k} size_{G_{sub_j}}}, \end{cases} \tag{7}$$

where $r(G_{sub_i}, G_{sub_j})$ denotes the communication load between subgraph G_{sub_i} and subgraph G_{sub_j}, which can be calculated by (8). $size_{G_{sub_i}}$ represents the size of subgraph G_{sub_i}. $R^c_{G_{sub_i}}$ is the estimated CPU resources to be used by subgraph G_{sub_i}, which can be calculated by (9). $R^m_{G_{sub_i}}$ represents the estimated memory resources to be used by subgraph G_{sub_i}, which can be calculated by (10).

$$r(G_{sub_i}, G_{sub_j}) = \sum_{v_{i,k} \in G_{sub_i}} \sum_{v_{j,m} \in G_{sub_j}} r(v_{i,k}, v_{j,m}). \tag{8}$$

$$R^c_{G_{sub_i}} = \sum_{v_{i,j} \in G_{sub_i}} R^c_{v_{i,j}}. \tag{9}$$

$$R^m_{G_{sub_i}} = \sum_{v_{i,j} \in G_{sub_i}} R^m_{v_{i,j}}. \tag{10}$$

where $R^c_{v_{i,j}}$ and $R^m_{v_{i,j}}$ represent the computed CPU and memory consumption by instance $v_{i,j}$, respectively.

As shown in Fig. 2(a), a streaming application G is partitioned into 3 subgraphs. The objectives of subgraph partitioning are to minimize the communication cost among subgraphs and make the resource consumption among the subgraphs relatively balanced. The deployment problem of subgraphs to compute nodes is to be discussed next.

4.2. Task deployment problem

Suppose we successfully partition the stream application G into k subgraphs $\{G_{sub_1}, G_{sub_2}, \ldots, G_{sub_k}\}$, then our task deployment problem is converted to: how to select k nodes from m available compute nodes and allocate k subgraphs to them? A subgraph can have multiple compute nodes as selection candidates, but a subgraph can only be allocated to one compute node, which means the subgraph and compute node are in a one-to-one mapping relationship. Suppose the decision variable is

$x_{i,j}$ (11), a subgraph G_{sub_j} can be allocated to a compute node n_i if (12) is satisfied.

$$x_{i,j} = \begin{cases} 1, \text{if } L_{n_i} > R^c_{G_{sub_j}} \text{ and } M_{n_i} > R^m_{G_{sub_j}}, \\ 0, \text{otherwise}, \end{cases} \tag{11}$$

where $x_{i,j}$ indicates whether the subgraph G_{sub_j} is assigned to node n_i, $i = \{1, 2, \dots, m\}$ and $j = \{1, 2, \dots, k\}$.

$$\min Z = \sum_{i=1}^{m} \sum_{j=1}^{k} w_{i,j} \cdot x_{i,j}, \tag{12}$$

subject to

$$\begin{cases} \sum_{j=1}^{k} x_{i,j} = 1, \quad i = 1, 2, \dots, m, \\ \sum_{i=1}^{m} x_{i,j} = 1, \quad j = 1, 2, \dots, k, \\ x_{i,j} = 0 \text{ or } x_{i,j} = 1, \end{cases} \tag{13}$$

where $w_{i,j}$ denotes the predicted cost of subgraph G_{sub_j} running on node n_i, and $w_{i,j}$ can be calculated by (14).

$$w_{i,j} = \mu \cdot (L_{n_i} - R^c_{G_{sub_j}}) + (1 - \mu) \cdot (M_{n_i} - R^m_{G_{sub_j}}), \tag{14}$$

where μ is the weight between CPU resources and memory resources, and $0 < \mu < 1$.

As shown in Fig. 2, there are multiple deployment options between subgraphs and nodes. We have to find out the one with the minimum cost to the data center when making a selection.

4.3. Stateful scheduling problem

At runtime, triggering task rescheduling can have a serious impact on the system performance, making it difficult to guarantee the reliability of state data. In Fig. 3, topology T runs in a data center and the state data of the vertices are backed up to remote storage. When manually or automatically rescheduling tasks running on nodes, we have to kill the tasks and restart. This has two negative effects.

First, when a task is restarted, the state data of the task needs to be pulled from the remote storage. However, due to factors like network packet loss or failure of storage nodes with the state data, the state of the task may be lost, causing incompleteness of the state data.

Second, if the volume of task state data is large, it takes a long time to pull data from the remote storage during state recovery, resulting in a slow start-up.

Sl-Stream tries to ease the negative effects by dividing the state data into blocks using erasure code technology, which ensures the reliability of state data and reduces the data pulling time during state recovery.

5. Sl-stream: optimizer model

In this section, we propose four optimizer models for the three problems identified above, i.e. optimizers for subgraph partitioning, task deployment, hierarchical scheduling and state management.

5.1. Subgraph partitioning optimizer

Subgraph partitioning is an NP problem (Liu et al., 2017), which a heuristic algorithm usually suits better. In this paper, we use the quantum particle swarm algorithm (Yang et al., 2004) to solve it. A set of solutions $X = (x_1, x_2, \dots, x_m)$ is generated by randomly arranging natural numbers from 1 to m for the instances in a stream application G using the common natural number encoding technique, where x_i is a way to randomizing the natural numbers from 1 to m, and p_i denotes an adaptation value at the current position of x_i, which can be calculated by (15).

$$q_i^1 = \frac{size_{G_{sub_i}}}{\sum_{j=1}^{k} size_{G_{sub_j}}}, q_i^2 = \frac{\sum_{v_{i,j} \in G_{sub_i}} R_{v_{i,j}}}{\sum_{v_{i,j} \in G} R_{v_{i,j}}},$$
$$p_i = \sum_{i=1}^{k} \sum_{j=1}^{k} r(G_{sub_i}, G_{sub_j}) + \rho \sum_{i=1}^{k} |q_i^1 - q_i^2|. \tag{15}$$

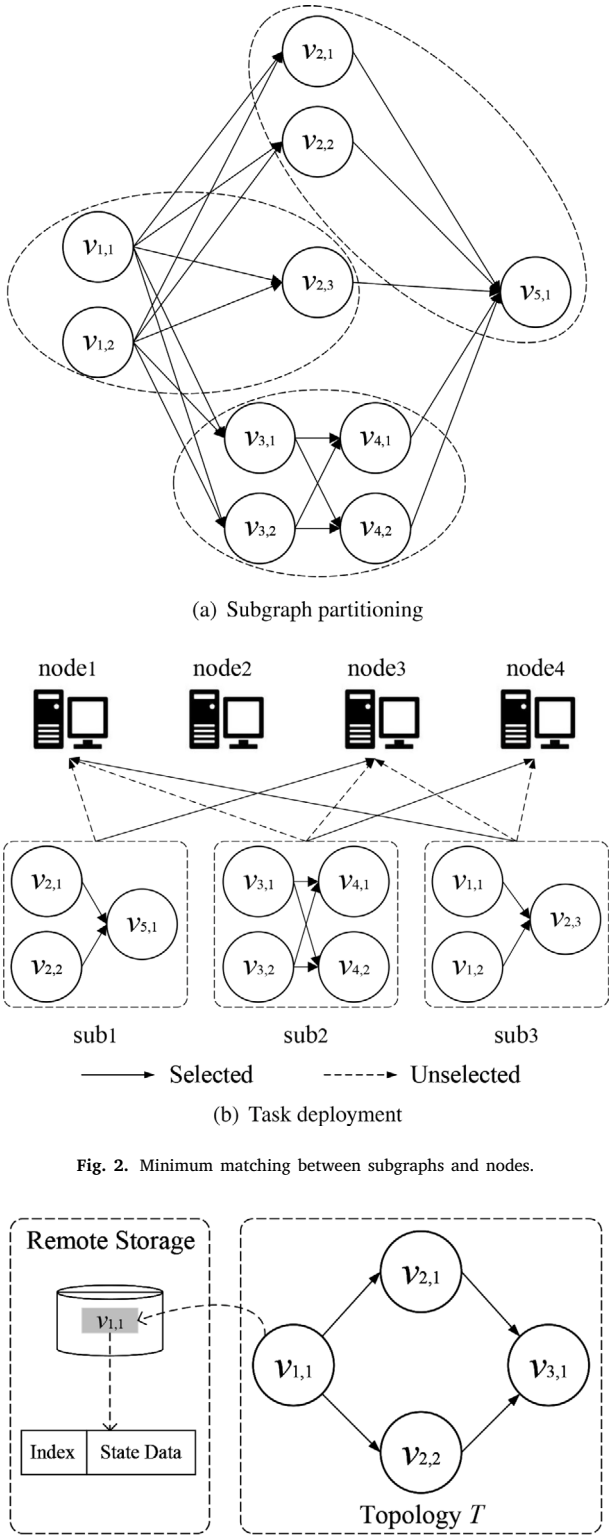


Fig. 2. Minimum matching between subgraphs and nodes.

Fig. 3. Stateful topology running on Storm.

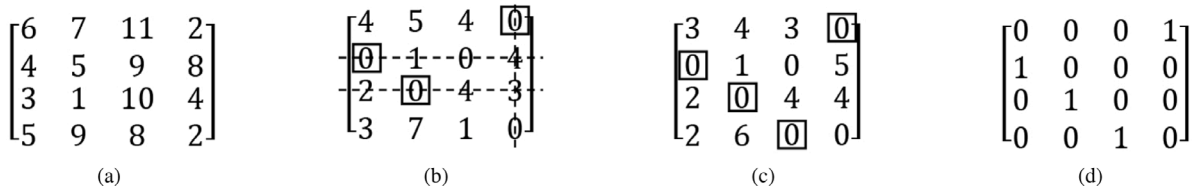


Fig. 4. Matching process between nodes and subgraphs.

where $size_{G_{sub_i}}$ represents the size of subgraph G_{sub_i}, q_i^1 denotes the ratio of the number of instances owned by the subgraph against the topology, and q_i^2 denotes the ratio of the resource consumption of the subgraphs against the topology.

The particle wave function $\varphi(x)$ of the quantum particle swarm (Yang et al., 2004) is known to be (16).

$$\varphi(x) = \frac{1}{\sqrt{L}} e^{-\frac{\|p-x\|}{L}}. \tag{16}$$

Its probability density function $Q(x)$ is (17).

$$Q(x) = |\varphi(x)|^2 = \frac{1}{L} e^{-2\frac{\|p-x\|}{L}}, \tag{17}$$

where p denotes the best adaptation value in each particle history and the parameter L can be calculated by (18).

$$L(t+1) = 2\psi \cdot |p - x(t)|. \tag{18}$$

Based on the probability density of the wave function (17), we are able to calculate the current wave value of $x(t)$ by the Monte Carlo algorithm (Traub and Wozniakowski, 1992).

$$x(t) = p \pm \frac{L}{2\ln(\frac{1}{\eta})}, \tag{19}$$

where L can be obtained by (18) and η takes value in range $rand(0,1)$. The L parameter is expressed as (20).

$$L(t+1) = 2\beta \cdot |mbest - x(t)|, \tag{20}$$

where β is the convergence coefficient, and generally takes the value of (21). Different β affects the convergence speed of the algorithm. σ takes value in range $rand(0.5 \sim 1.0)$. $mbest$ denotes the center of all particles, which can be calculated by (22).

$$\beta = \frac{\sigma \cdot (MAXITER - T)}{MAXITER} + 0.5, \tag{21}$$

$$mbest = \sum_{i=1}^{M} \frac{p_i}{M}, \tag{22}$$

where M is the number of populations, representing the size of the partition schemes set for the topology. p_i is the historical optimal fitness value of the ith particle. A particle represents one partition scheme.

Based on the above description, the iteration function of the topology partitioning algorithm is (23).

$$x(t+1) = p \pm \beta |mbest - x(t)| \cdot \ln u. \tag{23}$$

Keep executing (23) until the maximum number of iterations is reached. The output is then the divided subgraphs.

The deployment scheme from the divided subgraphs to compute nodes is to be discussed next.

5.2. Task deployment optimizer

Based on the above subgraph partitioning method, we partition the stream application G into k subgraphs $\{G_{sub_1}, G_{sub_2}, \ldots, G_{sub_k}\}$. G is deployed to a data center with m available compute nodes. We abstract the relationship between subgraphs and compute nodes in Fig. 5, where rows correspond to subgraphs, columns correspond to nodes, w_{km} denotes the cost of running subgraph G_{sub_k} on node n_m. We try to find one deployment with the minimum cost.

$$G_{sub} \times node = \begin{bmatrix} w_{11} & w_{12} & & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ w_{31} & w_{32} & & w_{3m} \\ \vdots & & \ddots & \vdots \\ w_{k1} & w_{k2} & \cdots & w_{km} \end{bmatrix}$$

Fig. 5. The cost matrix for assignment between nodes and subgraphs.

For example, Fig. 4(a) represents the cost matrix for assignment between subgraphs and nodes. First, we make row and column changes to Fig. 4(a) by subtracting the smallest element of one row for each row and the smallest element of one column for each column. A matrix can be obtained in Fig. 4(b). Next, determine the independent 0 elements for trial deployment. Start with a row that has only one 0 element, mark that 0 element in that row, and then cross out the other 0 elements in the column where the marked 0 element is located. Similarly, start with a column that has only one 0 element, mark the 0 element in that column, and then cross out the other 0 elements in the row where the marked 0 element is located. The final established independent 0 element is the box-marked 0 in Fig. 4(b). Since the number of independent 0 elements is 3, which is less than the matrix dimension 4, the matrix needs further adjustment.

Mark a row without an independent 0 element, mark the column with the 0 element crossed out of this row, and mark the row with an independent 0 element in this column. Then, the unmarked rows are delineated and the marked columns are delineated. The final delineation result is shown in Fig. 4(b). Then, the minimum value of the undrawn line is selected, the value is subtracted from the elements of the undrawn line, and the value is added to the elements of the intersection of the lines to finally obtain Fig. 4(c). Determine whether Fig. 4(c) satisfies the number of independent 0 elements equals to 4 (the dimension of matrix). If not, repeat the process of Fig. 4(a,b,c). The final deployment result is obtained as Fig. 4(d).

Sl-Stream triggers local adjustment to tasks based on the final deployment scheme, which is to be discussed next.

5.3. Hierarchical scheduling optimizer

The main process of Sl-Stream runtime scheduling optimizer is to monitor in real time whether the amount of communication between vertex instances and their resource consumption have changed significantly, and provide an evaluation result. Decision on whether rescheduling is made is based on this evaluation. Vertex instances are adjusted layer-by-layer during rescheduling. Triggering rescheduling consumes certain system resources, but the cost can be justified in the long run.

At runtime, the monitoring module of Sl-Stream continuously collects the metrics of the vertex instances and obtains the fitness value p_{new} for the new partition scheme x_{new} according to algorithm 1. It

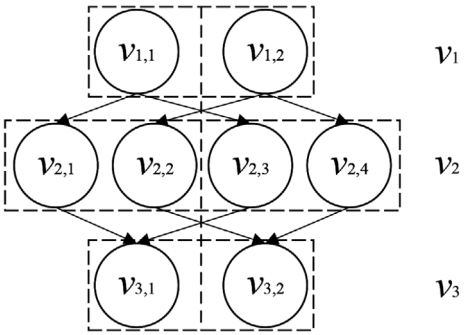


Fig. 6. Layer-by-layer adjustment.

compares p_{new} with the fitness value p_{old} of the running partition scheme x_{old} and triggers rescheduling when $\frac{p_{old}}{p_{new}} < \xi, 0 < \xi < 1$ is satisfied, where ξ denotes a user-defined factor that triggers task rescheduling. The higher ξ is, the more frequent the rescheduling is triggered.

When rescheduling is triggered, it does not kill the entire topology, but is done gradually within a certain period of time. According to Sl-Stream's state management strategy, a layer-by-layer instance migration is performed to avoid state loss of the instances. To ensure that the system provide services continuously, half instances of each layer are adjusted at a time.

As shown in Fig. 6, the stream application consists of v_1, v_2 and v_3 three layers. When rescheduling occurs, each layer has half instances adjusted each time. E.g. the instances of v_1 are adjusted first, and the adjustment is conducted in two steps: one instance one step. Next, the instances of v_2 are adjusted. Two instances each step for two steps. Finally, v_3 is done, which is the same as v_1. The timing of this process is user-defined.

When a task is migrated to a new node, it needs to restart and recover its state. An effective state management can help with the fast start-up on the new node, which is to be discussed next.

5.4. State management optimizer

State management can affect the system computational performance during instance migration. For example, if a stateful vertex is migrated, its state may be lost, disrupting the continuity of data processing. Storm system uses a checkpointing mechanism to maintain stateful vertices and store check-point (Tian et al., 2018; Zhuang et al., 2020) data to a remote storage system (e.g. Hadoop Distributed File System (HDFS) Lu et al., 2013). This may lead to longer time to recover the system state. It becomes even worse if packets are lost during network transmission, ultimately leading to incomplete state. We implement an online state backup and recovery mechanism to address these problems. The mechanism first divides the state data into several raw blocks, and encode the raw blocks to generate the check blocks. Then these data blocks are backed up to different upstream vertex instances for synchronization, which will be pulled for parallel processing during state recovery.

Suppose there is a stream application $G = (V(G), E(G))$, where $V(G) = \{v_1, v_2, \ldots, v_n\}$. Each vertex instance manages its own state and periodically generates the raw and check blocks. As shown in Fig. 7, the synchronization flow of state information among vertex instances forms a logical ring. Fig. 8 indicates that in the sth vertex, each instance $v_{s,k}$ manages its own state while partitioning its state data into raw blocks and generating check blocks by encoding raw blocks for synchronization with the instances in the $(s-1)$th vertex. Moreover, the sum of number of check blocks and raw blocks must be equal to the number of upstream instances of $v_{s,k}$.

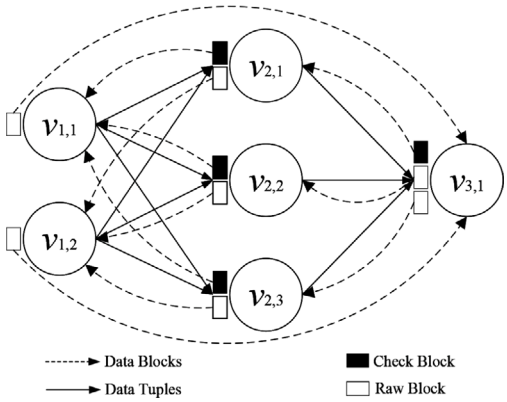


Fig. 7. Logical rings formed among stateful vertex instances.

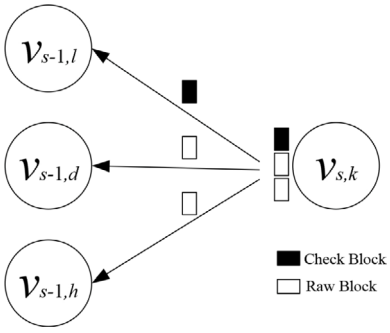


Fig. 8. Data block backup on upstream vertex instances.

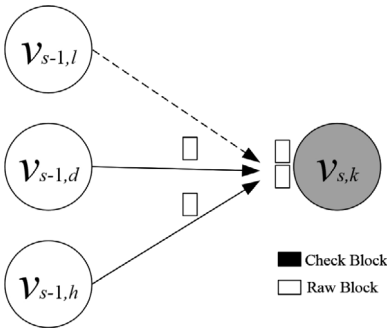


Fig. 9. State recovery for vertex instance.

If a vertex instance needs to restart, its state can be retrieved. As shown in Fig. 9, when a vertex instance $v_{s,k}$ needs to restart on another node, it can pull the state from the upstream vertex instances in parallel and then resume work. If several upstream vertex instances fail or stop running, the state recovery of downstream instance is not affected, given the number of failed instances less than the number of check blocks to be discussed in Section 6(E). As shown in Fig. 10, when an upstream vertex instance $v_{s-1,k}$ fails, the downstream instance $v_{s,k}$ can still use the remaining data blocks to recover the complete state through algorithm 5.

An instance needs to maintain state data blocks for multiple instances, so organizing different data blocks into a prefix tree can improve the locating efficiency. Nodes in the prefix tree mainly contain routing index information and data blocks. The routing index is composed of topology ID, component name (vertex name) and executor ID (instance ID). Fig. 11 shows how each instance organizes data blocks. Leaf nodes mainly store the data blocks and non-leaf nodes store the routing information.

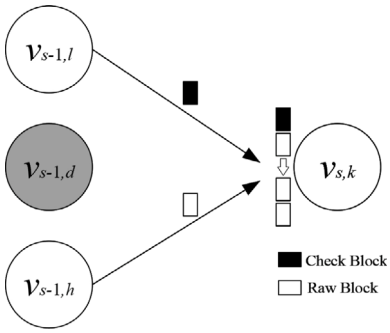


Fig. 10. Fault-tolerant state recovery for vertex instance.

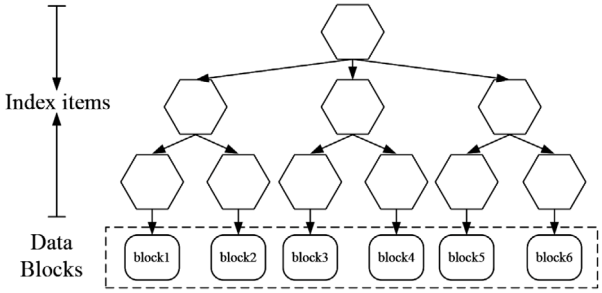


Fig. 11. Tree structure for data block management on vertex instance.

6. Sl-stream: framework and algorithms

Based on the above analysis, we propose and implement a state lossless scheduling strategy. In this section, we introduce the system framework and algorithms for subgraph partitioning, task deployment, hierarchical migration and state management.

6.1. System framework

The framework of Sl-Stream is shown in Fig. 12. The process of the state lossless scheduling strategy can be decomposed into the following five steps: Step 1: monitor communication E_r, CPU Ln_c and memory M_{n_c} load. This step collects information about the amount of communication E_r between tasks and the load on the nodes, and predicts the CPU $R^c_{v_{i,j},n_c}$ and memory $R^m_{v_{i,j},n_c}$ load of each task. Step 2: trigger scheduling. When the ratio between old p_{old} and new p_{new} scheduling is less 0.7 (defined by user, the higher this value is, the more frequent the rescheduling is triggered), new scheduling can be triggered at runtime based on the workload and communication E_r information. Step 3: partition subgraphs. Suppose that a streaming application requires 3 compute nodes to run, the corresponding DAG graph will be divided into 3 subgraphs. Communication-intensive tasks are placed on the same subgraph by algorithm 1 and the workload of each subgraph is guaranteed to be relatively balanced. Step 4: assign tasks. Suppose that there are 5 compute nodes in the cluster. 3 of 5 nodes will be selected by algorithm 2 to run the streaming application based on Step 3, and the algorithm guarantees the selected node costs are minimal. Step 5: generate a scheduling scheme and notify the data center of the scheme.

6.2. Subgraph partitioning algorithm

If the response time is higher or the throughput is lower than user's expectations, part of vertices of the DAG running online need to be rescheduled to improve the system performance.

In the online rescheduling phase, the DAG needs to be partitioned into the subgraphs to ensure that the system communication delay is minimized and the workload of the nodes are relatively balanced. This process is described in algorithm 1.

Algorithm 1: subgraph partition.

Input: Stream application G, predicted workload for each instance $v_{i,j}$, communication load between instances $r(v_{i,j}, v_{k,m})$;
Output: subgraph partition scheme x_i;
1 Initialize the maximum number of iteration defined by user, noted as *count* ;
2 Initialize the population size defined by user, noted as M ;
3 Initialize the set of partition schemes, noted as X ;
4 **for** *each* M **do**
5 G's instances are encoded with natural numbers, denoted as x_i;
6 $X \leftarrow x_i$;
7 **end**
8 **while** $count > 0$ **do**
9 Initialize the population optimum $gbest$, the center of gravity $mbest$ of x_i ;
10 **for** *each* x_i *in* X **do**
11 Calculate the historical optimum $pbest$ for x_i. Calculate the fitness value p_i for each x_i according to (15), and compare it with the historical optimum $pbest$. $pbest$ keeps the maximum value of both;
12 Calculate the population optimal $gbest$ for X. Calculate the fitness value p_i for each x_i and compare it with the $gbest$. $gbest$ keeps the maximum value of all p_i;
13 Update the center of gravity $mbest$ of all x_i according to Eq. (22);
14 **end**
15 **for** *each* x_i *in* X **do**
16 Update the position of each x_i according to (23) to generate new populations;
17 **end**
18 $count = count - 1$;
19 **end**
20 **return** *subgraph partition scheme* x_i *with maximum fitness value*

The input of algorithm 1 includes a DAG, workload of each instance and communication load between instances. Its output is a set of instances to be run on each node. Step 4 to step 7 initialize a population at random. Step 10 to step 14 calculate the historical optimal fitness value of subgraph partition scheme x_i, the global optimal value and the center of gravity of all x_i. Step 15 to step 17 update the positions of all x_i. The time complexity of algorithm 1 is $O(p \cdot n)$, where p is the population size defined by user, and n is the maximum number of iteration.

In algorithm 1, the node where a vertex instance is deployed is very likely to change, resulting in the redeployment of the online DAG. In this case, both the communication load and the resource consumption of the vertex instance are considered. The resource consumption of instance can be computed at DAG runtime.

6.3. Task deployment algorithm

Based on algorithm 1, when a stream application G is successfully partitioned into multiple subgraphs, each subgraph may have multiple nodes satisfying the subgraph deployment condition, which means that the computational resources required by the subgraph are less than the available resources of the nodes. If subgraphs are improperly assigned to nodes, a large amount of resources might be left idle in the data center.

When assigning subgraphs to nodes, the deployment with the minimum cost is preferred to ensure the maximum resource utilization in the data center. This is described in algorithm 2.

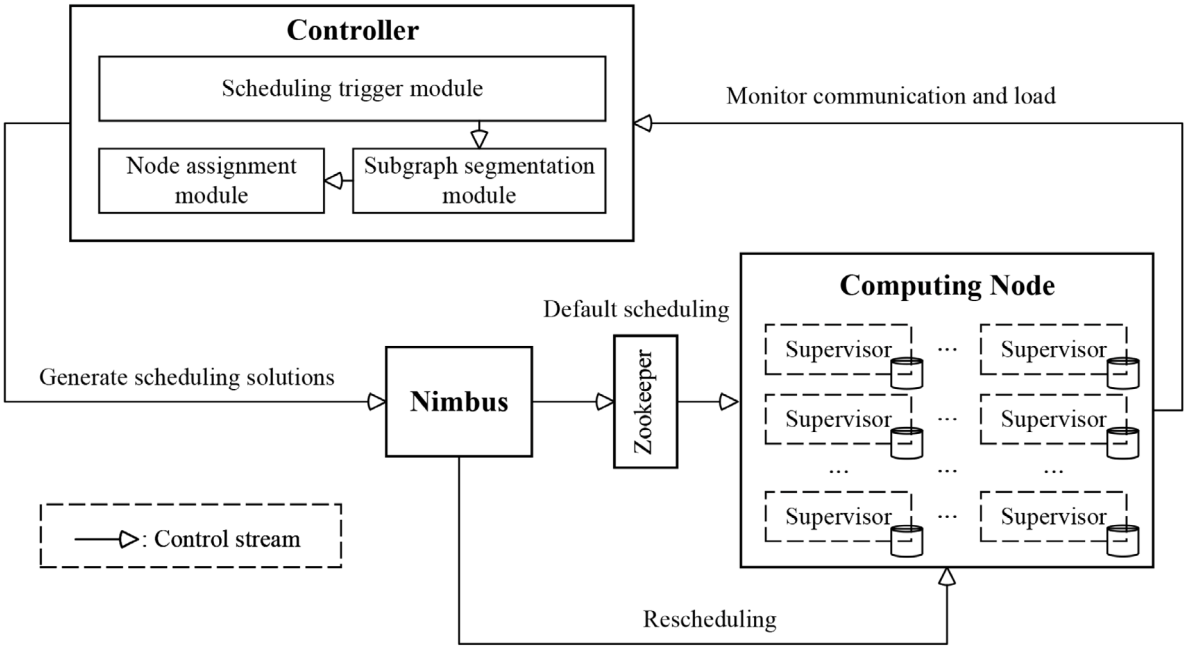


Fig. 12. The framework of Sl-Stream.

Algorithm 2: Minimum Cost Matching.

Input: Subgraphs of stream application G, predicted workload of each subgraph, and workload of each node;
Output: A set of one-to-one mappings between subgraphs and nodes;

1 Initialize the cost matrix M between subgraphs and nodes. The element value of the matrix M is the workload of each node minus the workload of each subgraph, and the value must be greater than 0. If it is less than 0, the element value is set to 1000 representing "infinity" here;
2 **while** *true* **do**
3 **for** *each row of matrix M* **do**
4 Row minus the minimum value of this row;
5 **end**
6 **for** *each column of matrix M* **do**
7 Column minus the minimum value of this column;
8 **end**
9 Perform a trial deployment. Find as many independent 0 elements as possible;
10 Starting from the row with only one 0 element, mark the 0 element in the row, and cross out the other 0 elements in the column where the marked 0 element is located;
11 Starting from a column with only one 0 element, mark the 0 element in that column , and cross out the other 0 elements in the row where the marked 0 element is located;
12 **if** *the number of independent 0 elements == the number of dimensions of the square matrix* **then**
13 End the program, break;
14 **end**
15 Pass all 0 elements with the least number of straight lines;
16 The minimum value *min* of the unpassed elements of the line is selected, and *min* is subtracted from all the unpassed elements of the line, and *min* is added to the intersecting elements of the line to obtain a new matrix;
17 **end**
18 **return** *matrix M*

The input of algorithm 2 includes subgraphs of stream application G, predicted workload of each subgraph and workload of each node. The output is a set of one-to-one mappings between subgraphs and nodes. Step 3 to 14 find the independent 0 elements and determine if the end condition of the program is satisfied. Step 15 to 16 adjust the matrix to produce more independent 0 elements. The time complexity of algorithm 2 is $O(m^3)$, where m is the number of compute node.

In algorithm 2, the matrix composed of subgraphs and nodes may not be a square matrix, so the maximum number of rows and columns of the matrix are chosen as the dimension of the square matrix. The missing part of the square matrix is filled with value 10 000 ("infinity") for supplementation.

6.4. Hierarchical migration algorithm

If rescheduling is triggered, an efficient local task migration method is available to enable the sustainable processing of data stream. Based on algorithm 1, the adaptation value p_{new} for a new partition scheme is continuously obtained. Based on algorithm 2, the mapping relationships between tasks and nodes in an deployment scheme can be obtained.

In task migration phase, all the tasks of the application are adjusted in batches layer by layer to ensure their state data integrity. This process is described in algorithm 3.

The input of algorithm 3 includes current state of G, predicted workload of vertex instances and workload of each node. The output is a boolean of Hierarchical migration. Step 2 and step 4 invoke algorithm 1 and algorithm 2 to get the necessary data. Step 6 to 16 find the vertex instances to be changed in each layer and process the changed instances in two batches for each layer. Step 17 to 23 migrate the changed instances in batches. The time complexity of algorithm 3 is $O(s)$, where s is the number of instances in the topology.

Algorithm 3 perceives fluctuations in data stream and responds in a timely manner. We perform the migration in batches according to the changes of resource consumption and communication load of tasks in each vertex. It ensures the system continue to provide services and the task state be recovered quickly in case of rescheduling.

Algorithm 3: Hierarchical migration.

Input: Current state of stream application G, predicted workload of vertex instances and workload of each node;
Output: The boolean of Hierarchical migration;

while *true* **do**
  Call algorithm 1 to get the current partition scheme and the fitness value (x_{new}, p_{new});
  if $\frac{p_{old}}{p_{new}} < \xi$ **then**
    Input x_{new} into algorithm 2 to get the mapping relationship (deployment scheme) between subgraphs and nodes $(G_{newsub}, nodeId)$;
    Use *Batches* to represent the set of migration batches for tasks;
    for *each* v_i *in* G **do**
      Use *SubBatch* to represent the set of tasks $v_{i,j}$ of the current vertex v_i that have changed position;
      for *each instance* $v_{i,j}$ *in* v_i **do**
        Find the location of $v_{i,j}$ in G_{newsub} and mark the $newNodeId$ corresponding to G_{newsub};
        Find the location of $v_{i,j}$ in G_{oldsub} and mark the $oldNodeID$ corresponding to G_{oldsub};
        Establish the relationship $f(v_{i,j})$ for the position change of $v_{i,j}$,
        $f(v_{i,j}) = oldNodeId(v_{i,j}) \rightarrow newNodeId(v_{i,j})$;
        $SubBatch \leftarrow f(v_{i,j})$;
      end
      SubBatch is divided into *SubBatch*1 and *SubBatch*2;
      $Batches \leftarrow SubBatch1 \quad Batches \leftarrow SubBatch2$;
    end
    for *each subBatch in Batches* **do**
      for *Batches in each change of task* $f(v_{i,j})$ **do**
        Kill $v_{i,j}$ task on $oldNodeId$ node;
        Start the $v_{i,j}$ task on $newNodeId$ node;
        The latest data block is pulled from the upstream instances of $v_{i,j}$ task in parallel and the state data is restored according to algorithm 4 to resume $v_{i,j}$'s work;
      end
    end
    return *True*
  end
end
return *False*

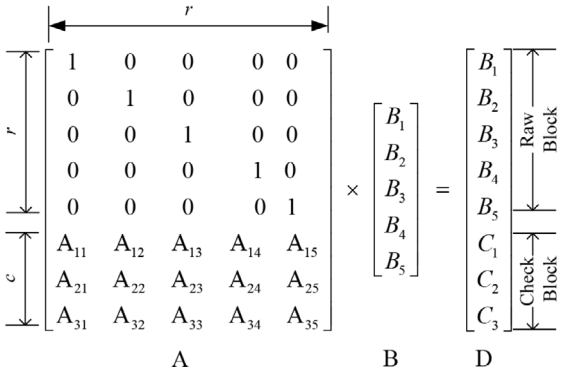
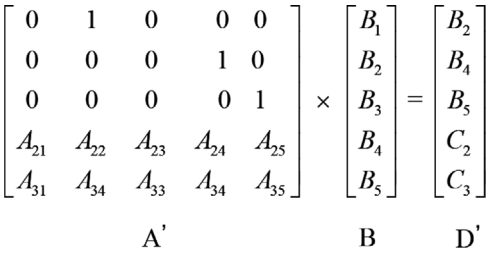
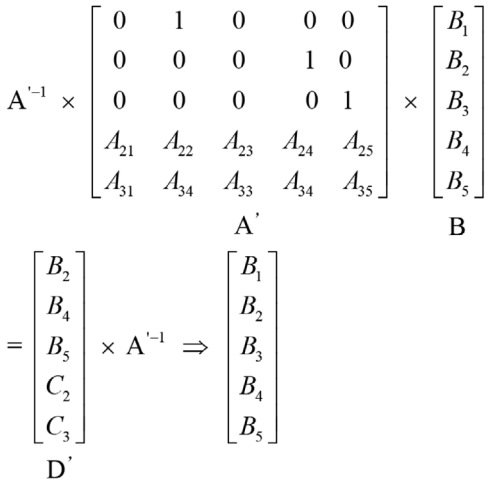


Fig. 13. Encoding of stateful data.



(a) The remaining data blocks D'



(b) Solve for vector B

Fig. 14. Decoding of stateful data.

6.5. State management algorithm

Fault-tolerant state backup can be implemented with the jerasure tool (Plank and Greenan, 2022). The whole process can be divided into two steps: encoding the state data after partition and decoding the encoded data blocks.

(1) Encoding step: if the number of upstream vertex instances for vertex instance $v_{s,k}$ is u, the state data of $v_{s,k}$ can be partitioned into r raw blocks and c check blocks, and r and c satisfy (24). These $(r + c)$ data blocks are stored respectively in the upstream vertex instances of $v_{s,k}$. As long as the number of failed upstream instances is less than c, instance $v_{s,k}$ is able to recover the complete state data from the remaining r copies of data blocks (no matter raw or check), enabling the state data tolerate loss of c number of blocks.

$$u = r + c \tag{24}$$

Assume that the number of upstream vertex instances for instance $v_{s,k}$ is 8, and the $v_{s,k}$ state is partitioned into 5 raw blocks and 3 check blocks, i.e., $r = 5$, $c = 3$. As shown in Fig. 13, the r raw blocks $B_1 \sim B_5$ are arranged into B vectors and a $(r + c) * r$ encoding matrix A is constructed. Where matrix A must satisfy: (1) the first r rows are unit matrices; (2) the $r * r$ square matrix consisting of any r row vectors in matrix A must be invertible. In this work, the encoding matrix technique used is the Vandermonde matrix (Klinger, 1967). The u data blocks, denoted as vector D, are obtained by multiplying the encoding matrix A with the matrix B.

(2) Decoding step: Suppose the upstream instances storing the data blocks B_1, B_3 and C_1 for $v_{s,k}$ are faulty, the state of $v_{s,k}$ can be recovered from the data blocks stored in the remaining instances. As shown in Fig. 14(a), $v_{s,k}$ pulls the remaining data blocks to produce vector D' from square matrix A' and vector B. Our goal is to solve for vector B, which represents the complete state of $v_{s,k}$.

As the $r * r$ square matrix consisting of any r row vectors of matrix A is invertible, the square matrix A' is invertible. As shown in

Fig. 14(b), it is only necessary to multiply the matrix A'^{-1} on both sides of the equation in Fig. 14(a) to solve for the vector B. Then the data block integration is performed and the complete state data $B_1 \sim B_5$ is obtained.

In the encoding and decoding steps for the state data, the interface to the jerasure tool is called. The encoding step is described in algorithm 4.

Algorithm 4: Encoding of stateful data.

Input: Storage location of state data, number of raw blocks and number of check blocks;
Output: The boolean of encoding stateful data;
1 Construct encoding matrix A;
2 Access the jerasure toolkit, call the encoding interface jerasure_matrix_encode to generate data blocks. Pass in the parameters A, number of raw blocks, number of check blocks and state data location;
3 Pull the metadata (especially the node location data) of the upstream vertex instances for the instance;
4 **for** *each block of data* **do**
5 Send data blocks to the upstream vertex instances based on the location information in metadata;
6 **end**
7 **return** $True$

The input of algorithm 4 includes state data storage location, number of raw blocks and number of check blocks. The output is a boolean of encoding stateful data. Step 2 partitions and encodes the state data. Step 3 to step 6 synchronize the raw blocks and check blocks with the upstream vertices. The time complexity of algorithm 4 is $O(e)$, where e is the number of blocks of the state data. The decoding step is described in algorithm 5.

Algorithm 5: Decoding of stateful data.

Input: Upstream instances information for the instance;
Output: Complete state data;
1 Initialize the number of raw blocks of the instance, noted as r ;
2 Pull the surviving data block b from the upstream vertex instances;
3 **if** $b > r$ **then**
4 Construct the encoding square matrix A' based on the surviving data blocks;
5 Initialize the $jerasure$ class by the jerasure toolkit;
6 Call the decoding interface of the $jerasure$ class by passing in the parameter square matrix A' and the data block location;
7 Construct state data B;
8 **return** B;
9 **else**
10 **return** *null*
11 **end**

The input of algorithm 5 includes information of the upstream vertices for the instance. The output is the complete state data. Step 1 pulls the surviving data blocks. Step 3 to step 9 recover from the surviving data blocks. The time complexity of algorithm 5 is $O(d)$, where d is the number of blocks of the state data.

Since each vertex instance keeps the state data blocks for multiple downstream instances, organizing and managing these data blocks using a prefix tree can improve the efficiency of finding data blocks. Once the prefix tree is constructed, it is rarely changed. Algorithm 6 describes the data block querying process.

Algorithm 6: Querying data block location.

Input: Prefix of the data block pd and root node of the prefix tree rn;
Output: Location of data blocks;
1 **while** rn *is not a leaf node* **do**
2 **for** *Each child node* e *of* rn **do**
3 $flag = flase$;
4 **if** $e \rightarrow blockData == pd$ **then**
5 $flag = ture$;
6 $rn \leftarrow e$;
7 break;
8 **end**
9 **end**
10 **if** $flag == true$ **then**
11 **return** *null*
12 **end**
13 **end**
14 **if** $rn \rightarrow blockData == pd$ **then**
15 **return** rn
16 **end**
17 **return** *null*

Table 3
Software configuration of the Sl-Stream.

SoftWare	Version
Ubuntu	Ubuntu 16.04 64 bit
Storm	Apache-Storm-1.0.2
JDK	Jdk1.8
Zookeeper	Zookeeper-3.4.6
Python	Python 2.7.2
MySql	MySql-5.1.7

The input of algorithm 6 is prefix of the data block queried by user and root node of the prefix tree. The output is the location of the data blocks. Step 2 to step 12 traverse the index information of the prefix tree. If the index information is not found in non-leaf nodes, null is returned. Step 14 to step 16 return data blocks. The time complexity of algorithm 6 is $O(h)$, where h is the height of the prefix tree. This algorithm can quickly find the location of data blocks based on the index information. Once the data blocks are obtained, they can be modified or transferred over the network.

7. Performance evaluation

In this section, we evaluate the performance of Sl-Stream system. The experimental environment and parameter settings are first discussed, followed by the analysis of the evaluation results and the fault-tolerant state management algorithms.

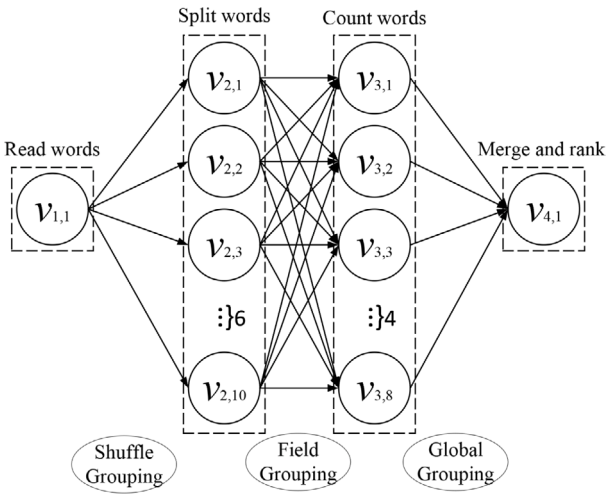
7.1. Experimental environment and parameter setup

The Sl-Stream system is developed based on Storm 1.0.2 and deployed on Ubuntu 16.04. Real-world data experiments are conducted on a cluster of AliCloud. The cluster consists of 10 machines, 2 of which run Storm nimbus as master nodes and 3 deploy zookeeper. The nodes deploying nimbus and zookeeper also deploy supervisor nodes, while the other 5 deploy only supervisor nodes. The software configuration of Sl-Stream is shown in Table 3.

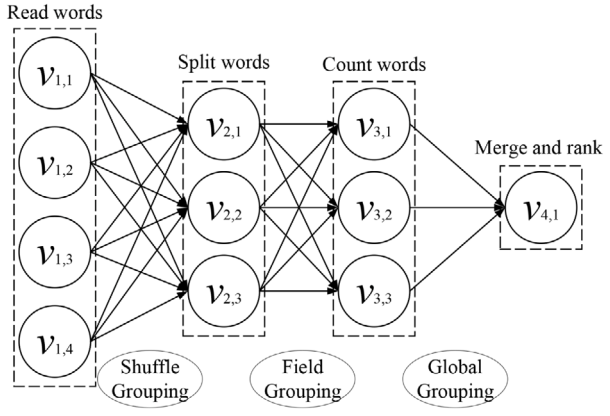
In addition, Top_N DAGs, one of the commonly used test applications, are submitted to the data center as the stream application. Two logic graphs of Top_N are shown in Fig. 15. The vertex functions and the number of instances are shown in Table 4.

Table 4
Vertex functions of Top_N.

Vertex	Instances	Function
v_1	1, 4	Read words from data stream
v_2	10, 3	Split words
v_3	8, 3	Count words
v_4	1, 1	Merge all ranks from upstream



(a) Topology 1 of Top_N.



(b) Topology 2 of Top_N.

Fig. 15. Two topologies of Top_N.

7.2. Performance results

The experiments focus on three metrics: system throughput, system response time and cluster size.

(1) System Throughput.

System throughput reflects the performance of a system and is measured by the number of output tuples per second for a DAG the system is running. The higher the system throughput, the more capable the system is of processing data. In this set of experiments, we set the input rates of the data stream to 1000 tuples/s and 2000 tuples/s to test the system throughput. The experimental results are the average throughput of topology 1 and topology 2.

The Sl-Stream strategy has higher system throughput than the Storm EvenScheduler and ResourceAwareScheduler when the input rate of data stream is kept stable at 1000 tuples/s. As shown in Fig. 16, the system throughput basically remains stable after 200 s. The average throughput of Sl-Stream is 523 tuples/s, roughly two times

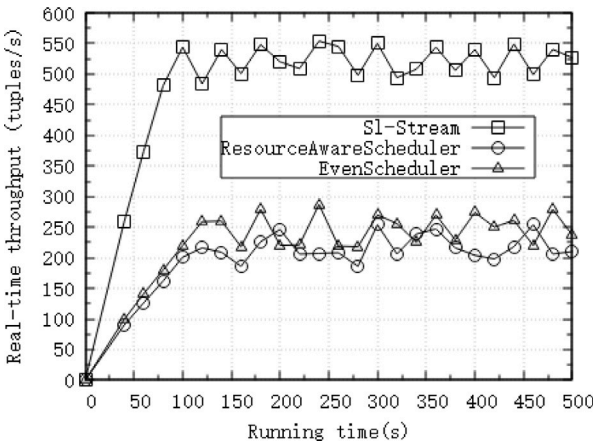


Fig. 16. System throughput under data rates of 1000 tuples/s.

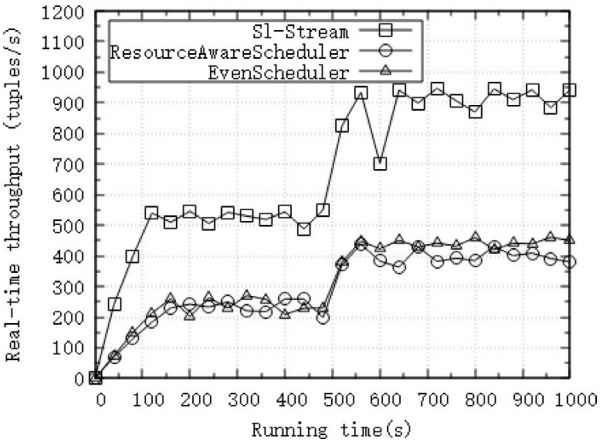


Fig. 17. System throughput under data rates of 2000 tuples/s.

more than the EvenScheduler's 254 tuples/s and ResourceAwareScheduler's 221tuples/s. Experiments show the average throughput of the Sl-Stream system is higher than that of the EvenScheduler and ResourceAwareScheduler when the input rate is stable for the given application.

When the input rate fluctuates over time, as shown in Fig. 17, ramping up to 2000 tuples/s at the 500 s, Sl-Stream system still has a higher system throughput than the EvenScheduler and ResourceAwareScheduler. At around 700 s, the system throughput remains stable again. The throughput of Sl-Stream changes from 523 tuples/s to 922 tuples/s, the throughput of the EvenScheduler changes from 254 tuples/s to 431 tuples/s, and that of the ResourceAwareScheduler changes from 212 tuples/s to 405 tuples/s. Experiments show that the average throughput of Sl-Stream is still greater than those of the EvenScheduler and ResourceAwareScheduler when the input rate fluctuates for the given application.

From Fig. 17, it can be observed that the throughput of Sl-Stream drops sharply at 593 s after the rate increases. The main reason for this is that the communication between tasks may shift when the data stream rate changes, causing the monitoring module to trigger a rescheduling command.

(2) System Response Time

As the response time of a system can directly affect user's experience, it is considered as an important evaluation metric. The shorter the response time of a system, the better the user experience and the better the real-time performance.

When the input rate is kept stable at 2000 tuples/s for topology 1 of Top_N, Sl-Stream has a lower response time compared to the

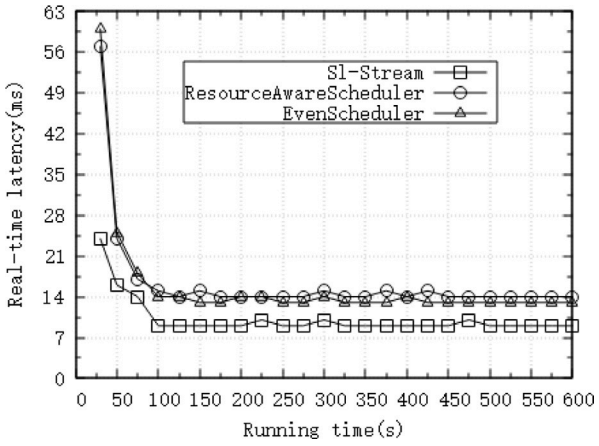


Fig. 18. System response time of topology 1 under data rates of 2000 tuples/s.

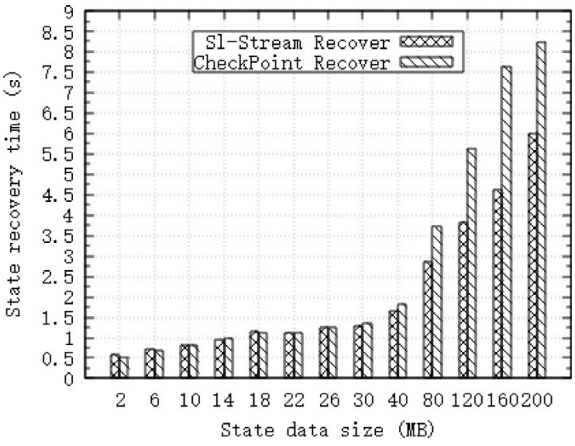


Fig. 20. Recovery time under different state sizes.

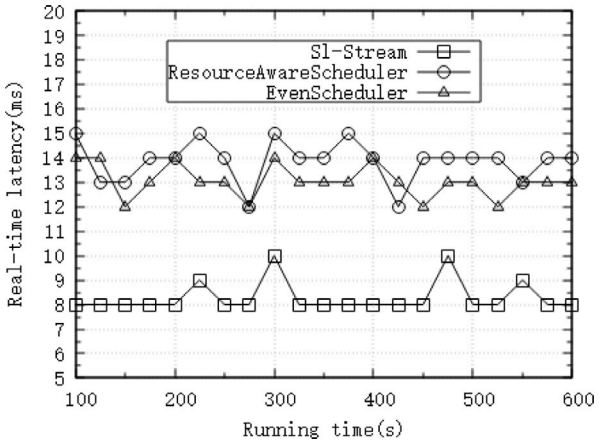


Fig. 19. System response time topology 2 under data rates of 2000 tuples/s.

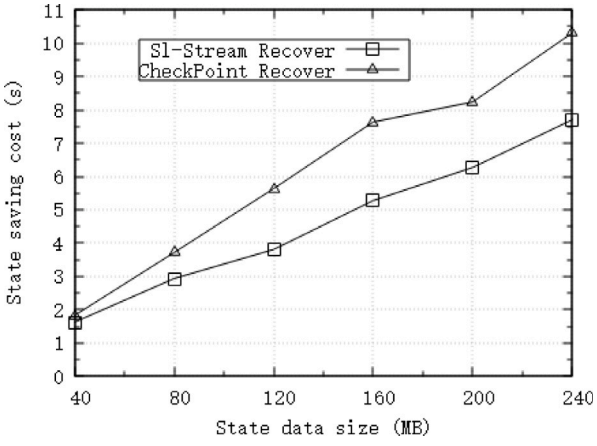


Fig. 21. State saving time under different state sizes.

EvenScheduler and ResourceAwareScheduler. As shown in Fig. 18, the average response time for Sl-Stream, the EvenScheduler and ResourceAwareScheduler are 9.3 ms, 13.2 ms and 14.1 ms, respectively after the system becomes stable. The experiments clearly show that the average response time of Sl-Stream is lower than those of the EvenScheduler and ResourceAwareScheduler when the input rate is stable.

When the input rate is kept stable at 2000 tuples/s for topology 2 of Top_N, Sl-Stream also has a lower response time compared to the EvenScheduler and ResourceAwareScheduler. As shown in Fig. 19, the average response time are 8.6 ms, 12.9 ms and 13.7 ms for Sl-Stream, EvenScheduler and ResourceAwareScheduler when the system becomes stable. It is obvious that the average response time of Sl-Stream is lower than those of the EvenScheduler and ResourceAwareScheduler when the input rate is stable.

7.3. State recovery time evaluation

We compare the state recovery technique of Sl-Stream with the checkpoint recovery of Storm by changing the size of state. In this experiment, we focus on a stream application where only one task needs to be migrated when rescheduling is triggered. The number of check blocks and raw blocks are set to 3 and 7, respectively.

As shown in Fig. 20, the state recovery time for Sl-Stream and checkpoint are similar when the state data size falls in range [2 MB, 30 MB]. However, the state recovery time of Sl-Stream is reduced roughly from 5.2% to 31.4% when the state data size varies in range [40 MB, 200 MB]. And this difference keeps getting larger as the amount of data in the state increases. The state recovery time for Sl-Stream mainly includes data blocks pulling time and data blocks computing time. Given the network bandwidth resources are sufficient, the time difference between Sl-Stream and checkpoint for pulling the state data is not significant when the state data size is small. Therefore, the state recovery time of both is similar. When the state data size becomes larger, it becomes a major factor affecting the state recovery time. Sl-Stream simultaneously pulls multiple state data blocks when performing state recovery. Storm's check-point recovery, on the other hand, can only use single-threaded state data pulling. Therefore, the state recovery time of Sl-Stream is much shorter than that of Storm when the size of the state data is large.

We also compare the state saving cost of Sl-Stream with that of the checkpoint recovery by changing the size of state data. As shown in Fig. 21, when the size of state data reaches 80 MB, the state saving time for Sl-Stream and for Storm is 2.9 s and 3.7 s, respectively. When the size of state data is 200 MB, the time is 6.2 s and 8.2 s for Sl-Stream and Storm. Sl-Stream always takes much less time than Storm to save the state data with increasing sizes. And the time gap increases as the size increases.

The total state recovery time of Sl-Stream and of the Storm checkpoint recovery is evaluated by changing the number of migrated instances when rescheduling is triggered. In this experiment, the minimum and maximum sizes of the migrated instance state data are 10 MB and 80 MB, respectively. As shown in Fig. 22, when the number of migrated instances is 4, the total state recovery time of Sl-Stream and

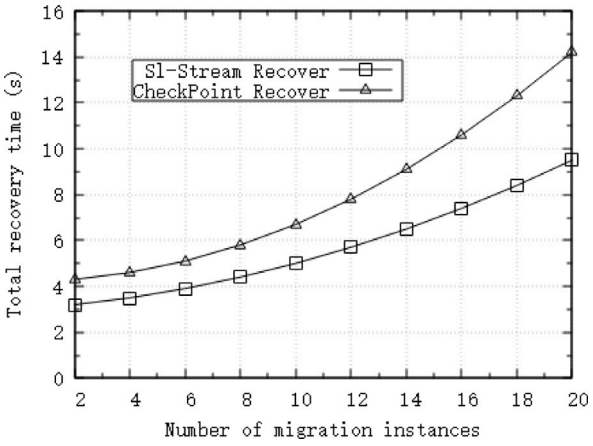


Fig. 22. Total recovery time for migrating different number of instances.

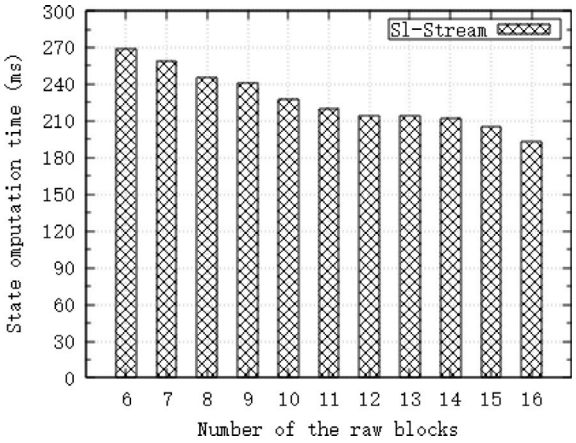


Fig. 23. Computation time for different number of raw blocks.

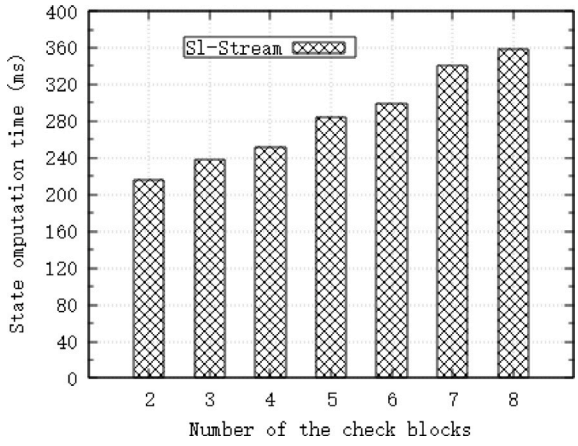


Fig. 24. Computation time for different number of check blocks.

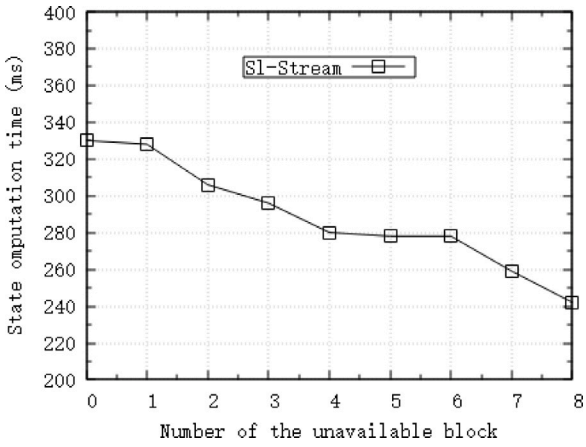


Fig. 25. Computation time for different number of unavailable blocks.

of Storm checkpoint recovery is 3.5 s and 4.6 s, respectively. When the number is 8, the total time for Sl-Stream and for Storm is 4.4 s and 5.8 s, respectively. When the number is 16, the time is 7.4 s and 10.6 s for Sl-Stream and Storm. Sl-Stream always takes much less time than Storm to recover the state.

We evaluate the performance impact by the number of raw blocks between 6 and 16 on state recovery, where the state data size is 10 MB and the number of check blocks is 3. As shown in Fig. 23, the system computation time to recover the state data is evaluated by varying the number of raw blocks. It can be observed that as the number of raw blocks increases, the required computation time decreases.

We also evaluate the performance impact by the number of the check blocks between 2 and 8 on state recovery, where the state data size is 10 MB and the number of raw blocks is 8. As shown in Fig. 24, the system computation time to recover the state data is evaluated by varying the number of check data blocks. It can be observed that as the number of checksum data blocks increases, the required computation time increases as well.

Therefore, reasonable numbers of raw blocks and check blocks can improve the system performance when recovering state data.

We also evaluate the performance impact by number of unavailable data blocks between 1 and 8 on state recovery, where the size of state data is 10 MB, the number of raw blocks is 8, and the number of check blocks is 8. As shown in Fig. 25, the system computation time to recover state data is evaluated by varying the number of unavailable data blocks. It can be observed that as the number of unavailable data blocks increases, the computation time required gradually decreases. Therefore, the loss of data blocks does not consume longer time for state data recovery.

8. Conclusions and future work

In this paper, we propose a state lossless scheduling strategy. This strategy is divided into three main phases. First, one stream application is divided into different subgraphs based on the amount of communication between tasks and their potential resource consumption. Second, the nodes and subgraphs are matched at minimum cost based on the predicted cost of nodes running the subgraph. Third, local instance adjustment is conducted for each layer of the topology and its state is restored in parallel for the adjusted vertex instance. Sl-Stream provides distributed state management measures to enhance the system reliability. The experiments show that our Sl-Stream scheduling strategy is effective.

In our strategy, the subgraph partition phase, task deployment phase and stateful scheduling phase run sequentially. The solutions are proposed to address the problems in each phase and they run independent from each other. Suboptimal results of a predecessor phase should not affect much the following phases as they try to optimize each phase independently.

As partly the future work, we will further investigate the following areas.

(1) Consider the parallelism of DAG to further reduce the data processing latency of the system.
(2) Integrate cluster energy consumption into Sl-Stream to improve the energy efficiency of the system.

CRediT authorship contribution statement

Minghui Wu: Conceptualization, Methodology, Validation, Writing – original draft. **Dawei Sun:** Methodology, Validation, Writing – original draft, Investigation, Funding acquisition. **Yijing Cui:** Validation, Investigation, Writing – review & editing. **Shang Gao:** Formal analysis, Investigation, Writing – review & editing. **Xunyun Liu:** Validation, Data curation, Investigation. **Rajkumar Buyya:** Methodology, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 61972364; the Fundamental Research Funds for the Central Universities, PR China under Grant No. 265QZ2021001; and MelbourneChindia Cloud Computing (MC3) Research Network, PR China.

References

Al-Maytami, B.A., Fan, P., Hussain, A., Baker, T., Liatsis, P., 2019. A task scheduling algorithm with improved makespan based on prediction of tasks computation time algorithm for cloud computing. IEEE Access 7, 160916–160926.

Al-Sinayyid, A., Zhu, M., 2020. Job scheduler for streaming applications in heterogeneous distributed processing systems. J. Super Comput. 76, 9609–9628.

Alghamdi, M.I., Jiang, X., Zhang, J., Zhang, J., Jiang, M., Qin, X., 2017. Towards two-phase scheduling of real-time applications in distributed systems. J. Netw. Comput. Appl. 84, 109–117.

Apache, 2022a. Flink. http://flink.apache.org/.

Apache, 2022b. Samza. https://github.com/apache/incubator-heron/.

Apache, 2022c. Storm. http://storm.apache/org/.

Barika, M., Garg, S., Zomaya, A.Y., Ranjan, R., 2021. Online scheduling technique to handle data velocity changes in stream workflows. IEEE Trans. Parallel Distrib. Syst. 32 (8), 2115–2130.

Cardellini, V., Nardelli, M., Luzi, D., 2016. Elastic stateful stream processing in storm. In: 2016 International Conference on High Performance Computing & Simulation (HPCS). pp. 583–590.

Djigal, H., Feng, J., Lu, J., Ge, J., 2021. IPPTS: An efficient algorithm for scientific workflow scheduling in heterogeneous computing systems. IEEE Trans. Parallel Distrib. Syst. 32 (5), 1057–1071.

Ebadifard, F., Babamir, S.M., 2018. A modified black hole-based multi-objective workflow scheduling improved using the priority queues for cloud computing environment. In: 2018 4th International Conference on Web Research (ICWR). pp. 162–167.

Ebadifard, F., Babamir, S.M., Barani, S., 2021. A dynamic task scheduling algorithm improved by load balancing in cloud computing. IEEE Trans. Parallel Distrib. Syst. 117–183.

Ebadifard, F., Doostali, S., Babamir, S.M., 2018. A firefly-based task scheduling algorithm for the cloud computing environment: Formal verification and simulation analyses. In: 2018 9th International Symposium on Telecommunications (IST). pp. 664–669.

Eskandari, L., Huang, Z., Eyers, D., 2016. P-Scheduler: adaptive hierarchical scheduling in apache storm. In: Proceedings of the Australasian Computer Science Week MulticonferenceFebruary, pp. 1–10.

Eskandari, L., Mair, J., Huang, Z., Eyers, D., 2018. T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster. Future Gener. Comput. Syst. 89, 617–632.

Farrokh, M., Hadian, H., Sharifi, M., Jafari, A., 2022. SP-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters. Expert Syst. Appl. 191, 116322.

Fischer, L., Bernstein, A., 2015. Workload scheduling in distributed stream processors using graph partitioning. In: 2015 IEEE International Conference on Big Data (Big Data). pp. 124–133.

Fu, X., Tang, B., Guo, F., Kang, L., 2021. Priority and dependency-based DAG tasks offloading in fog/edge collaborative environment. In: 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD). pp. 440–445.

Gedik, B., Schneider, S., Hirzel, M., Wu, K., 2014. Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. 1447–1463.

Jiang, J., Zhang, Z., Cui, B., Tong, Y., Xu, N., 2017. StroMAX: Partitioning-based scheduler for real-time stream processing system. In: Database Systems for Advanced Applications - 22nd International Conference. pp. 269–288.

Klinger, A., 1967. The vandermonde matrix. Amer. Math. Monthly 74 (5), 571–574.

Li, C., Tang, J., Ma, T., Yang, X., Luo, Y., 2019a. Load balance based workflow job scheduling algorithm in distributed cloud. J. Netw. Comput. Appl. 152 (4), 102518.

Li, H., Wu, J., Jiang, Z., X. Li, X.W., 2017b. Task allocation for stream processing with recovery latency guarantee. In: 2017 IEEE International Conference on Cluster Computing, CLUSTER 2017. IEEE Press, pp. 379–383.

Li, C., Zhang, J., Luo, Y., 2017a. 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.

Li, J., Zheng, G., Zhang, H., Shi, G., 2019b. Task scheduling algorithm for heterogeneous real-time systems based on deadline constraints. In: IEEE International Conference on Electronics Information and Emergency Communication. pp. 113–116.

Liu, Y., Chao, G., Zhang, Z., Lu, Y., Shi, C., Liang, M., Li, T., 2017. Solving NP-hard problems with physarum-based ant colony system. IEEE/ACM Trans. Comput. Biol. Bioinform. 14 (1), 108–120.

Liu, P., Xu, H., Silva, D.D., Wang, Q., Ahmed, S.T., Hu, L., 2020. FP4S: Fragment-based parallel state recovery for stateful stream applications. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1537–1548.

Lu, K., Dai, D., Sun, M., 2013. HDFS+: Concurrent writes improvements for HDFS. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. pp. 182–183.

Marchal, L., Nagy, H., Simon, B., Vivien, F., 2018. Parallel scheduling of DAGs under memory constraints. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 204–213.

Pathan, R., Voudouris, P., Stenström, P., 2018. Scheduling parallel real-time recurrent tasks on multicore platforms. IEEE Trans. Parallel Distrib. Syst. 29 (4), 915–928.

Plank, S., Greenan, M., 2022. Jerasure. https://github.com/tsuraan/Jerasure/.

Rho, J., Azumi, T., Nakagawa, M., Sato, K., Nishio, N., 2017. Scheduling parallel and distributed processing for automotive data stream management system. J. Parallel Distrib. Comput 286–300.

Runsewe, O., Samaan, N., 2019. CRAM: a container resource allocation mechanism for big data streaming applica-tions. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 321–3320.

Sainik, L., Khajuria, D., 2014. Fault tolerant data flow using curator — Storm. In: 2014 IEEE 5th International Conference on Software Engineering and Service Science. pp. 472–475.

spark, 2022. streaming. https://spark.apache.org/.

Tian, Y., Shen, Q., Zhu, Z., Yang, Y., Wu, Z., 2018. Non-authentication based checkpoint fault-tolerant vulnerability in spark streaming. In: 2018 IEEE Symposium on Computers and Communications (ISCC). pp. 00783–00786.

Traub, J.F., Wozniakowski, H., 1992. The Monte Carlo algorithm with a pseudorandom generator. Math. Comp. 58 (197), 323–339.

Twitter, 2022. Heron. https://github.com/apache/incubator-heron.

Wu, Y., Tan, K., 2015. ChronoStream: Elastic stateful stream computation in the cloud. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 723–734.

Yang, S., Wang, M., jiao, L., 2004. A quantum particle swarm optimization, In: Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat), 1, pp. 320–324.

Zhang, S., Wu, Y., Zhang, F., He, B., 2020. Towards concurrent stateful stream processing on multicore processors. IEEE Trans. Parallel Distrib. Syst. 1537–1548.

Zhao, Y., Liu, Z., Wu, Y., Jiang, J., Cheng, J., Liu, K., Yan, X., 2021. Timestamped state sharing for stream analytics. IEEE Trans. Parallel Distrib. Syst. 32 (11), 2691–2704.

Zhuang, Y., Wei, X., Li, H., Hou, M., Wang, Y., 2020. Reducing fault-tolerant overhead for distributed stream processing with approximate backup. In: 2020 29th International Conference on Computer Communications and Networks (ICCCN). pp. 1–9.

Minghui Wu is a postgraduate student at the School of Information Engineering, China University of Geosciences, Beijing, China. He received his Bachelor Degree in Network Engineering from Zhengzhou University of Aeronautics, Zhengzhou, China in 2020. His research interests include big data stream computing, distributed systems and blockchain.

Dawei Sun is an Associate Professor in the School of Information Engineering, China University of Geosciences, Beijing, P.R. China. He received his Ph.D. degree in computer science from Northeastern University, China in 2012, and conducted the Postdoctoral research in the department of computer science and technology at Tsinghua University, China in 2015. His current research interests include big data computing, cloud computing and distributed systems. In these areas, he has authored over 70 journal and conference papers.

Yijing Cui is a postgraduate student at the School of Information Engineering, China University of Geosciences, Beijing, China. She received her Bachelor Degree in Network Engineering from Zhengzhou University of Aeronautics, Zhengzhou, China in 2020. Her research interests include big data stream computing, data analytics and distributed systems.

Shang Gao received her Ph.D. degree in computer science from Northeastern University, China in 2000. She is currently a Senior Lecturer in the School of Information Technology, Deakin University, Geelong, Australia. Her current research interests include distributed system, cloud computing and cyber security.

Xunyun Liu received the B.E. and M.E degree in Computer Science and Technology from the National University of Defense Technology in 2011 and 2013, respectively. He obtained the Ph.D. degree in Computer Science at the University of Melbourne in 2018. His research interests include stream processing and distributed systems.

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored over 750 publications and four text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 153 with 123,500+ citations). He served as the founding Editor-in-Chief (EiC) of IEEE Transactions on Cloud Computing and now serving as EiC of Journal of Software: Practice and Experience.