Future Generation Computer Systems 136 (2022) 252–269

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

An energy efficient and runtime-aware framework for distributed stream computing systems

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

[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] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

A R T I C L E   I N F O

Article history:
Received 16 August 2021
Received in revised form 17 May 2022
Accepted 6 June 2022
Available online 16 June 2022

Keywords:
Stream computing
Runtime-aware
Reliable migration
Energy consumption
High throughput
Low latency

A B S T R A C T

Task scheduling in distributed stream computing systems is an NP-complete problem. Current scheduling schemes usually have a pause or slow start process due to the fluctuation of input data stream, which affects the performance stability, especially the high throughput and low latency goals. In addition, idle compute nodes at runtime may result in large idle load energy consumption. To address these problems, we propose an energy efficient and runtime-aware framework (Er-Stream). This paper thoroughly discusses the framework from the following aspects: (1) The communication between real-time data streaming tasks is investigated; stream application, resource and energy consumption are modeled to formalize the scheduling problem. (2) After an initial topology is submitted to the cluster, task pairs with high communication cost are processed on the same compute node through a lightweight task partitioning strategy, minimizing the communication cost between nodes and avoiding frequent triggering of runtime scheduling. (3) At runtime, reliable task migration is performed based on node communication and resource usage, which in turn helps the dynamic adjustment of the node energy consumption. (4) Metrics including latency, throughput, resource load and energy consumption are evaluated in a real distributed stream computing environment. With a comprehensive evaluation of variable-rate input scenarios, the proposed Er-Stream system provides promising improvements on throughput, latency and energy consumption compared to the existing Storm's scheduling strategies.

1. Introduction

Data-intensive services, such as social networking, stock trading and weather monitoring, are becoming increasingly common. They generate massive amounts of data every second. At the same time, more and more applications emphasize real-time and accuracy, putting higher demand on stream processing. For example, security issues face great challenges in large-scale computing environments [1,2], where timeliness is crucial in security check, and real-time computing allows for fast analysis and processing to obtain useful information. Besides, real-time computing is also used in various aspects such as education industry [3], road traffic [4], and environmental inspection [5].

To meet this demand, a variety of stream processing frameworks have emerged, including Spark [6], Heron [7], Samza [8] and Storm [9], etc. Built on batch processing [10], Spark divides incoming data stream into short batches; Heron is a real-time fault-tolerant distributed stream data processing system developed by Twitter [11] as an open-source project; Samza started out as a stream processing solution for LinkedIn [12]. Its most important feature is its construction relies heavily on the log-based Kafka [13]; Storm is one of the most popular open source big data stream computing systems and has been widely used by many well-known companies and organizations, such as Twitter and Alibaba [14].

A stream computing system has multiple compute nodes that collaborate to process tasks, where high data transfer latency between nodes may have a negative impact on system performance. The communication time and data transfer latency can be effectively reduced by restricting data transfer on the same node or between nearby nodes. In addition, capability differences between nodes result in different performance of task execution and data transfer. Task placement for streaming applications can be mapped to an NP-complete problem [15]. Given the computational resources of a node are limited, data loss may occur when

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

https://doi.org/10.1016/j.future.2022.06.007
0167-739X/© 2022 Elsevier B.V. All rights reserved.

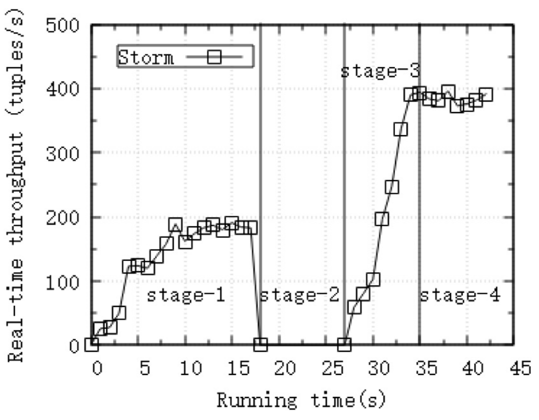


Fig. 1. System throughput at different times.

node resources cannot meet the computational demand. These factors make the scheduling of streaming applications challenging.

Triggering rescheduling [16,17] at runtime to reallocate tasks and resources for streaming applications is one of the ways to addressing this challenge. However, runtime scheduling also has problems, such as how to keep the sustainability of stream processing, how to effectively reduce the network delay of processing data tuples and balance the computational resources of nodes. Fig. 1 shows the throughput variation of an example application over time in Storm system. To create resource utilization bottleneck on some nodes, a smaller than required number of compute nodes are purposely used to run a streaming application. As can be seen, when the topology is running in [0s, 15s], its throughput stays at a relatively low level. The main reason for the low throughput may be because tasks with high communication load are on different nodes and/or some nodes are short of computing resources. To improve the throughput of the system, we can restart the whole topology to make the tasks with high communication load on the same node and deploy part of tasks from the nodes with limited computing resources to these with idle computing resources or to new nodes. However, this rescheduling process can seriously lower the throughput during interval [18s–27s] and a slow pickup during interval [27s–33s], which obviously affects user's experience. This is just a simple case, but it demonstrates the necessity of providing a runtime-aware mechanism which can monitor the node resource consumption and communication among tasks, dynamically balance the node resource load and deploy tasks based on their communication load at runtime.

In addition, after a streaming application is mapped to a directed acyclic graph (DAG), a critical path of the DAG can reflect the response time of the system. When none of the tasks running on a node is on a critical path, the node does not have to run the tasks to its full capability as it will inevitably result in high energy consumption. If there is a good method to dynamically adjust the working state of compute nodes, the energy consumption of the system may become more effective.

Based on the above observations and thoughts, this paper proposes an energy efficient and runtime-aware framework (Er-stream). It tries to resolve: (1) when and how to reschedule an application topology based on the fluctuation of data stream, (2) when to perform reliable task migration based on node resource consumption, and (3) how to dynamically adjust the frequency of node's CPU based on their resource load.

1.1. Paper contributions

As discussed, Er-Stream is proposed to improve the throughput and reduce the latency of a distributed stream computing system. Our contributions are summarized as follows:

(1) Investigate task placement, resource constraint and energy consumption of fluctuating data streams, and formalize the scheduling problem by modeling stream application, resource constraint and energy consumption;
(2) Propose a stream application scheduling algorithm that deploys tasks with potential communication load on the same node in the DAG initialization phase and evaluate the resource allocation scheme at runtime to determine the necessity of making partial task adjustments;
(3) Propose a run-time aware scheduling algorithm to avoid excessive consumption of node resources by determining the necessity of making task migration, and adjust node's CPU frequency dynamically based on the resource usage information to lower the energy consumption;
(4) Evaluate the system throughput, response time and energy consumption of the proposed scheduling framework.

Experiments are conducted on real data and the results demonstrate the effectiveness of the Er-stream framework.

1.2. Paper organization

The rest of this paper is organized as follows. Section 2 describes the background knowledge; Section 3 introduces the system models, including the DAG model, the resource model and the energy consumption model; Section 4 formalizes the scheduling problem and provides optimization schemes; Section 5 introduces the Er-Stream framework and its main algorithms; Section 6 evaluates the performance of the Er-Stream; Section 7 presents related work and Section 8 concludes our work.

2. Background

Scheduling strategies in a stream computing system determine the allocation of stream applications to compute nodes. In the process of creating a topology for a streaming application, user can define the parallelism of components and the number of resources to be used by the topology. Storm, as one of the most popular distributed streaming computing systems, provides four built-in scheduling strategies [9]: EvenScheduler, IsolationScheduler, MultitenantScheduler and ResourceAwareScheduler.

2.1. EvenScheduler

EvenScheduler releases resources that are no longer needed by other topologies before assigning tasks to them. The available resources in the system are therefore evenly distributed among the active topologies. As shown in Fig. 2, a streaming application G includes tasks $v_{1,1}$, $v_{1,2}$, $v_{3,1}$ and $v_{3,2}$, which are deployed on 4 compute nodes n_1, n_2, n_3, n_4. Tasks in dashed boxes implement the same function. There are two main steps to run tasks of the example streaming application G in a Storm cluster: (1) receive the tasks submitted by users, then distribute them evenly to four workers with two instances per worker. (2) retrieve the available resources of the current cluster, and place the four workers evenly on the nodes. This load distribution is not absolutely even. When the number of topologies submitted by users increases, uneven node load may occur.

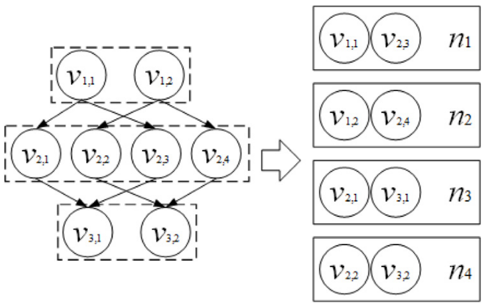


Fig. 2. Tasks in Topology are evenly distributed on nodes by EvenScheduler.

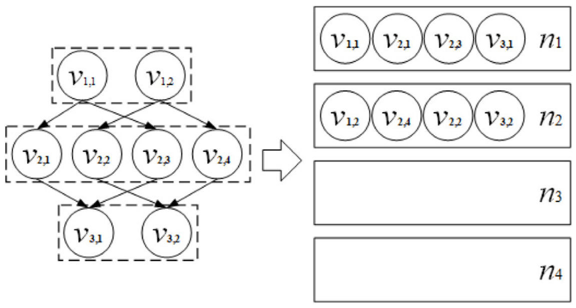


Fig. 3. Computing resource is isolated by IsolationScheduler.

2.2. IsolationScheduler

IsolationScheduler provides a mechanism that allows users to individually specify the node resources needed for certain topologies. The user needs to specify this information (topology names and the number of nodes they need) in the Storm configuration entry, and IsolationScheduler will prioritize the task assignment of these topologies, ensuring that the nodes assigned to a particular topology can only run that particular topology, as if these topologies were running in a separate environment from each other. After these specified topologies are assigned, the EvenScheduler assigns tasks of the remaining topologies using the remaining resources in the system.

As shown in Fig. 3, The streaming application G is assigned to two nodes n_1, n_2. The tasks in G will be evenly distributed over n_1 and n_2. This scheduler is designed to make the topology exclusive to the cluster nodes, so that different topologies are physically isolated from each other by occupying different cluster resources when they are released.

2.3. MultitenantScheduler

The MultitenantScheduler first constructs an isolated resource pool for each user exclusively, then it traverses the topology set and assigns nodes by creating topological associations to the resource pool. The resources are isolated from each other among the users. As shown in Fig. 4, two users *userA*, *userB* submit their streaming applications to the cluster. Assume the application submitted by *userA* is G and it is deployed on nodes n_1, n_2. The one submitted by *userB* is deployed on nodes n_3, n_4. As the applications are submitted by different users, there is no resource sharing between the two applications. If there are just right processing resources for G and *userA* wants to submit another streaming application, *userA* will have to wait for the resources of G to be completely released. If *userA* still has unused resources, other users cannot use them either. This scheduling strategy decreases resource utilization of the cluster, but provides an isolation mechanism for each application and fixes the allocation of computational resources.

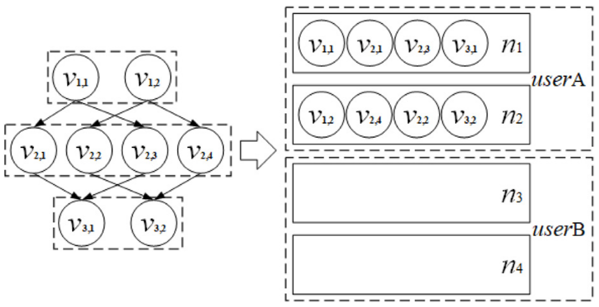


Fig. 4. Isolated resource pool is exclusively constructed for each user by MultitenantScheduler.

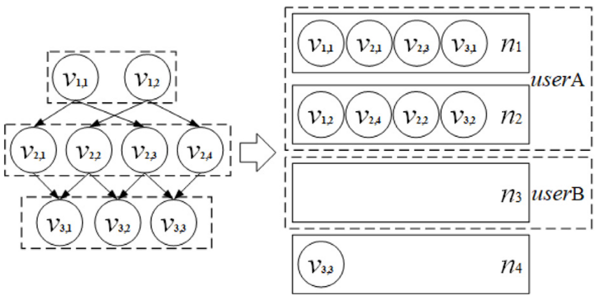


Fig. 5. Public resource pool can be used by ResourceAwareScheduler if users' computing resources are insufficient.

2.4. ResourceAwareScheduler

The ResourceAwareScheduler allocates resources on a per-user basis. Each user is guaranteed a certain number of resources to run their topology, and the ResourceAwareScheduler will guarantee the allocation as much as possible. When a Storm cluster has additional resources, the ResourceAwareScheduler will allocate the additional resources to users in a fair manner. The ResourceAwareScheduler can compensate for the shortcomings of MultitenantScheduler and improve the resource utilization of the entire cluster. As shown in Fig. 5, nodes n_1, n_2 can be used by *userA*, node n_3 can be used by *userB* and node n_4 is a public resource that can be used by both *userA* and *userB*. Assume *userA* submits a streaming application G which requires more than its available computational resources. As there exists an extra resource node n_4 in the cluster, G allocates its task to this extra node n_4. The unallocated resources of the cluster are public resources and can be used when needed.

The basic design idea behind all these stream computing scheduling strategies is to track the resource load of nodes. However, these strategies are static and cannot dynamically adapt to changing communication load between tasks or take into account the current load and energy consumption of each node. On Storm platform, topology rescheduling can be implemented through the IScheduler interface, but it requires that the entire topology be killed before the rescheduling strategy being applied.

In this paper, we focus on an energy efficient and runtime-aware framework for stream computing on the Storm platform, and further propose a scheme for reliable task migration upon rescheduling to improve both the performance and energy efficiency of the system.

3. System models

Before formalizing the scheduling problem and introducing our solution, we first model the stream application, resource and energy consumption in stream computing environments.

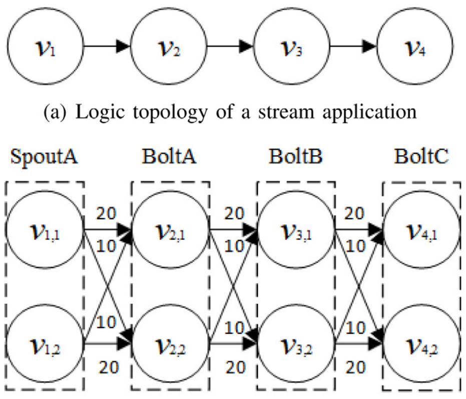


Fig. 6. Topology relationship diagram on Storm.

3.1. Stream application model

In a stream computing environment, users are able to define a topology themselves and submit it to the stream computing system. A topology consists of spout and bolt, where the spout task, the source of the data stream, can emit an unbounded number of tuples to downstream in the topology, and the bolt task, a component that consumes any number of tuples from spouts or other bolts, can process the received tuples, and potentially emit new tuples to downstream. The topology formed by spouts and bolts constitutes a complex streaming application and can be mapped to a directed acyclic graph $G = (V(G), E(G))$, where $V(G) = \{v_i | i \in 1, \ldots, n\}$ represents a finite set with n vertices. A vertex $v_i \in V(G)$ is an operation with a specific function and can be represented as multiple instances of a spout or bolt component. $E(G) = \{e_{v_{i,k},v_{j,m}} | v_{i,k}, v_{j,m} \in V(G)\}$ denotes a finite set of edges and an edge $e_{v_{i,k},v_{j,m}} \in E(G)$ denotes the existence of communication between instances $v_{i,k}$ and $v_{j,m}$.

When a topology is submitted to the cluster, a stream application model is first constructed based on the parallelism set by the user for each component, then the default EvenScheduler (if no other scheduler explicitly specified) steps in to allocate resources and distribute the corresponding tasks to run on nodes. As shown in Fig. 6, the illustrated topology contains one spout component and three bolt components, where each component has a parallelism of 2. This topology can be viewed as a directed acyclic graph with each component mapped as a vertex $v_i \in V(G)$ and each communication relationship between instances of the components mapped as $e_{v_{i,k},v_{j,m}} \in E(G)$.

We define $tr(v_{i,k}, v_{j,m})$, where $i, j = \{1 \ldots n \quad and \quad i \neq j\}, k = \{1 \ldots size(v_i)\}, m = \{1 \ldots size(v_j)\}$, is the data tuple transmission rate between instances $v_{i,k}$, $v_{j,m}$ of two vertices v_i and v_j, that is, the number of tuples passed per unit time. It satisfies (1).

$$tr(v_{i,k}, v_{j,m}) = \begin{cases} 0 & v_{i,k} \text{ and } v_{j,m} \text{ on the same node,} \\ w_{tr} & \text{otherwise,} \end{cases} \tag{1}$$

Where w_{tr} represents the average transmission rate in time interval $[t_s, t_e]$, t_s and t_e denote the start time and end time of a given short time period which can be set by user (e.g. 5 s). There may be transient fluctuations in the arrival stream rates, and w_{tr} can effectively lower its impact by deducting the max and min rates and calculating the average rate, easing the impact brought by sudden rate fluctuations. It can be calculated by (2).

$$w_{tr} = \frac{\int_{t_s}^{t_e} w_{tr_t} dt - \max(w_{tr_t}) - \min(w_{tr_t})}{t_e - t_s}. \tag{2}$$

where w_{tr_t} represents the transmission rate at time t, and $t \in [t_s, t_e]$.

The vertex instance $v_{i,k}$ outputs the processed data tuples to its downstream vertex instance set $D(v_{i,k})$, $o_{D(v_{i,k})} = |D(v_{i,k})|$ indicating the number of downstream instance set. If $o_{D(v_{i,k})} = 0$ indicates that the vertex is the last component of the stream topology. As shown in Fig. 6, the downstream instance set of $v_{2,1}$ is $D(v_{2,1}) = \{v_{3,1}, v_{3,2}\}$.

The vertex instance $v_{i,k}$ receives the input data stream from its upstream vertex instance set $U(v_{i,k})$, $o_U(v_{i,k}) = |U(v_{i,k})|$ indicating the number of upstream instance set. If $o_U(v_{i,k}) = 0$, it indicates that the vertex is the data source of the streaming application. As shown in Fig. 6, the upstream instance set of $v_{2,1}$ is $U(v_{2,1}) = \{v_{1,1}, v_{1,2}\}$.

3.2. Resource model

In a cluster, resources on a compute node can be measured in different dimensions, such as CPU, memory and I/O. In this paper, we consider the CPU resources. The complexity of a vertex $v_{i,k}$ is determined by the function it implements and measured by the time complexity and space complexity of the function algorithm implemented. The greater the complexity, the more CPU computational resources required. Not only the tuple processing, but also the tuple sending and receiving consume CPU computational resources.

Therefore, running an instance $v_{i,k}$ of vertex v_i on a compute node, the CPU consumption of this instance $v_{i,k}$, noted as $L_{v_{i,k}}$, can be calculated by (3).

$$L_{v_{i,k}} = l_{v_{i,k,c}} + l_{v_{i,k,in}} \cdot \varepsilon_{v_{i,k},v_{i-1,j}} + l_{v_{i,k,out}} \cdot \rho_{v_{i,k},v_{i+1,m}} \quad , \tag{3}$$

where $l_{v_{i,k,c}}$, $l_{v_{i,k,in}}$, $l_{v_{i,k,out}}$ denote the CPU resources consumed by the instance $v_{i,k}$ on tuple computation, tuple input and output on instance $v_{i,k}$, respectively. $\varepsilon_{v_{i,k},v_{i-1,j}}$ and $\rho_{v_{i,k},v_{i+1,m}}$ are decision variables and can be obtained by (4) and (5), respectively.

$$\varepsilon_{v_{i,k},v_{i-1,j}} = \begin{cases} 0, & v_{i,k} \text{ and } v_{i-1,j} \text{ run on the same node,} \\ 1, & \text{otherwise,} \end{cases} \tag{4}$$

$$\rho_{v_{i,k},v_{i+1,m}} = \begin{cases} 0, & v_{i,k} \text{ and } v_{i+1,m} \text{ run on the same node,} \\ 1, & \text{otherwise,} \end{cases} \tag{5}$$

where $v_{i-1,j} \in U(v_{i,k})$, $v_{i+1,m} \in D(v_{i,k})$.

At time t, there may be multiple instances $v_{i,k}$ running on compute node n_i, denoted as $C_{n_i,v}$. The CPU consumption of compute node n_i (denoted L_{n_i}) can be calculated by (6).

$$L_{n_i} = \sum_{v_{i,k} \in C_{n,v}} L_{v_{i,k}} \cdot w_{v_{i,k,t}}, \tag{6}$$

where $w_{v_{i,k,t}}$ is a coefficient, and can be calculated by (7).

$$w_{v_{i,k,t}} = \begin{cases} 1 & \text{at time t, } v_{i,k} \text{ is running,} \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

Since the CPU utilization of one compute node may vary, to ease the effect of such variation, all mathematical expectations in a statistical time interval $[t_s, t_e]$ can be defined as the load factor $Lr_{n_i,[t_s,t_e]}$ for compute node n_i. $Lr_{n_i,[t_s,t_e]}$ can be calculated by (8).

$$Lr_{n_i,[t_s,t_e]} = \frac{L_{n_i}}{t_e - t_s} = \frac{\sum_{v_{i,k} \in C_{n,v}} L_{v_{i,k}} \cdot w_{v_{i,k,t}}}{t_e - t_s}, \tag{8}$$

3.3. Energy consumption model

Power consumption [18,19] of a processor can be mainly divided into static power consumption and dynamic power consumption. Static power consumption refers to the standby power consumption of the physical machine. In general, the static power consumption of the same type of physical machine is a fixed constant. Dynamic power consumption refers to the power consumption of data processing when the physical machine performs a task. It is a variable and generally depends on the tasks. The power consumption of a processor can be denoted as (9)

$$P = c \cdot v^2 \cdot f, \tag{9}$$

Where P represents the power consumption, c represents a constant determined by the process and other factors, v represents the voltage and f represents the clock frequency. It is known that f is positively correlated with v. Therefore, for the convenience of discussion, the dynamic power consumption is modeled as cf^{κ}, κ is approximately equal to 3, and P_s is used to denote the static power consumption of the physical machine. The complete power consumption of the processor is denoted as (10)

$$P = P_s + c \cdot f^{\kappa}. \tag{10}$$

Assume the maximum frequency of a processor is f_{max} and $h_{n_i}^{max}$ is the execution time for node n_i to execute λ data tuples at the maximum frequency f_{max}. When the frequency of node n_i is f_i ($f_i < f_{max}$), the execution time of executing λ data tuples can be estimated by (11).

$$h_{n_i} = \frac{h_{n_i}^{max} \cdot f_{\max}}{f_i}. \tag{11}$$

Then, the energy consumption of node n_i to execute λ data tuples can be calculated by (12).

$$En_i(f_i) = P \cdot h_{n_i} = (p_s + c \cdot f_i^{\kappa}) \cdot \frac{h_{n_i}^{max} \cdot f_{\max}}{f_i}. \tag{12}$$

The energy consumption for the cluster deployed with a streaming application can calculated by (13).

$$En(f) = \sum_{i=1}^{s_c} En_i(f_i) \tag{13}$$

where s_c denotes the size of the cluster.

4. Problem statement and optimization

In this section, we formalize the scheduling problem in stream computing systems and present our optimization schemes for DAG initialization and runtime scheduling, and strategies for energy saving.

4.1. Scheduling problem

Latency and throughput are two important criteria for performance evaluation of a stream computing system. Assume the system latency to process a data tuple is dr, then the average delay $dr_{[1,\lambda]}$ when processing λ data tuples can be calculated by (14).

$$dr_{[1,\lambda]} = \frac{1}{\lambda} \cdot (\sum_{l=1}^{\lambda} dtr + \sum_{l=1}^{\lambda} dqu + \sum_{l=1}^{\lambda} dco), \tag{14}$$

where dtr, dqu, and dco denote the transmission delay, queuing delay, and computational delay of a data tuple in the system, respectively. In this paper, the queuing delay of a data tuple is not considered for the time being. When the system's data tuple processing delay $dr_{[1,\lambda]}$ is low, its throughput will be high. dtr, dco and $dr_{[1,\lambda]}$ are positively correlated. There exists a positive mathematical relationship between transmission delay of instances dtr and the data tuple transmission rate of instances $tr(v_{i,k}, v_{j,m})$, as represented by (15).

$$z(dtr) = \omega \cdot tr(v_{i,k}, v_{j,m}), \tag{15}$$

where ω is a coefficient and $\omega > 0$. Computational delay dco and a node's CPU consumption L_{n_i} are positively correlated, as represented by (16).

$$g(dco) = dco + \delta \cdot L_{n_i}, \tag{16}$$

where δ is a coefficient and can be calculated by (17).

$$\delta = \begin{cases} \mu & L_{n_i} > B, \\ 0 & \text{otherwise}, \end{cases} \tag{17}$$

where B is the maximum limit of the node resource load, μ is the factor that decides the change of dco, and μ is a positive number.

Suppose $Dn = \{n_1, n_2, n_3, \ldots, n_{num}\}$ is a cluster with num compute nodes. When a user submits an application topology $G = (V(G), E(G))$ (i.e., a DAG model), all vertex instances of $V(G)$ are deployed to Dn. For λ input tuples, the topology's network transmission latency $dtr_{[1,\lambda]}$ can be estimated by (18).

$$dtr_{[1,\lambda]} = \sum_{l=1}^{\lambda} z(dtr) = \omega \cdot \sum_{l=1}^{\lambda} tr(v_{i,k}, v_{j,m}). \tag{18}$$

From (18), we can see that the larger $\sum tr(v_{i,k}, v_{j,m})$ is, the greater the system transmission latency becomes. The main reason of the increase is that the two vertex instances with communication relationship are allocated to different compute nodes.

When a user submits multiple applications to a stream computing system, there may be an uneven distribution of vertex instances. This can result in some nodes being overloaded. Due to the relationship between the computational delay dco and node's CPU consumption L_{n_i} as indicated in (16), the dco of the overloaded node will increase.

In a time interval $[t_s, t_e]$, if a node's tuple processing rate is $cor_{[t_s,t_e]}$, and its tuple receiving rate is $inr_{[t_s,t_e]}$, ε can be calculated by (19).

$$\varepsilon = \frac{inr_{[t_s,t_e]}}{cor_{[t_s,t_e]}}, \tag{19}$$

where ε denotes the ratio between receiving and processing tuple rates in the $[t_s, t_e]$ time interval. When $\varepsilon \leq 1$, the data tuples of instance $v_{i,k}$ will not be backed up in cache, so the queuing delay will be small. When $\varepsilon > 1$, the data tuples for instance $v_{i,k}$ are continuously accumulated in the cache, and the queuing delay dqu increases over time.

In summary, if the scheduler of a stream computing system cannot effectively perceive the communication load between nodes and the resource usage of nodes, the dtr, dqu and dco of data tuples will grow.

4.2. DAG initialization optimization

The initial scheduling optimization is to find a reasonable allocation scheme that minimizes the communication cost between nodes, helping improve the throughput and lower the response latency.

The data tuple transmission rate between vertex instances cannot be estimated in the initialization phase because the communication load between vertex instances is not available yet. However, we can determine if communication exists between two adjacent instances, i.e. $tr(v_{i,k}, v_{i+1,m}) > 0$, we assume the average transmission rate $w_{tr(v_{i,k}, v_{i+1,m})} = 1$.

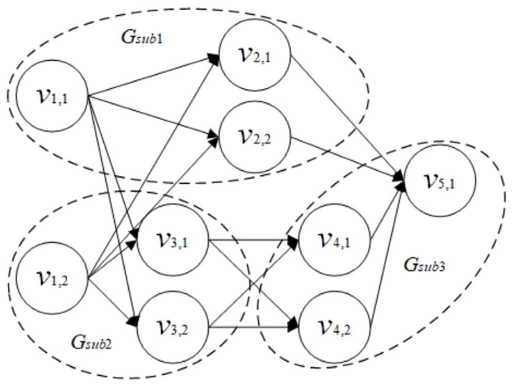


Fig. 7. Graph segmentation example.

For a cluster $Dn = \{n_1, n_2, n_3, \ldots, n_{num}\}$ and a streaming application G, we assign all instances in G to run on Dn. To reduce unnecessary network transmission overhead, the number of Dn used by G should be minimized, given allocated nodes are not overloaded.

Assume the set of compute nodes used by G is $Gn = \{n_i | 1 \leq i \leq num\}$, then the available resources of the cluster must be greater than the resources required for topology G, i.e., condition in (20) must be satisfied.

$$\sum_{j=1}^{i} C_{n_j,v} \geq \sum_{v_i \in V(G)} \sum_{v_{i,k} \in v_i} v_{i,k}. \tag{20}$$

As G is a directed acyclic graph and can be stored in a matrix, it can be further divided into $length_{Gn}$ number of subgraphs, denoted as (21).

$$G = \{G_{sub_1}, G_{sub_2}, \ldots, G_{sub_{length_{Gn}}}\}, \tag{21}$$

where G_{sub} denotes one subgraph of graph G. The size of G_{sub_i} corresponds to $C_{n_j,v}$, and each subgraph has a mapping relationship with compute node, denoted as $Map(sub_i, n_j)$. In addition, the subgraph must satisfy condition (22).

$$\min(\sum tr(G_{sub_i}, G_{sub_j})), \tag{22}$$

where $tr(G_{sub_i}, G_{sub_j})$ denotes the communication load between G_{sub_i} and G_{sub_j}.

As shown in Fig. 7, if the example application uses 3 compute nodes n_1, n_2 and n_3, the graph will be divided into 3 subgraphs G_{sub_1}, G_{sub_2} and G_{sub_3}. The dashed circle indicates the split boundary.

At runtime, the vertex instances of G_{sub_1}, G_{sub_2} and G_{sub_3} are placed on the corresponding compute nodes n_1, n_2 and n_3.

4.3. Runtime scheduling optimizer

The runtime reliable scheduling optimizer monitors the communication load and CPU load of nodes. When the communication load between vertex instances changes significantly, it produces an assessment result. Based on this result, a local adjustment is made to the allocation of vertex instances running on the compute nodes. If a node becomes CPU overloaded, a corresponding migration action is triggered, allowing the system computational latency to reach a balanced and stable state.

(1) The communication load. Given one stream application $G = (V(G), E(G))$ runs on node set $Gn = \{n_i | 1 \leq i \leq num\}$, each compute node n_i runs vertex instances $C_{n_i,v}$, and the communication load between each vertex instance can be collected by

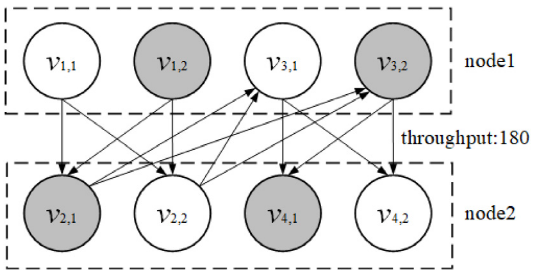


Fig. 8. A topology running on two nodes.

a monitoring module, the communication load between compute nodes in Gn can be calculated by (23).

$$tr(n_i, n_j) = \sum tr(v_{i,k}, v_{j,m}), \tag{23}$$

where $tr(v_{i,k}, v_{j,m})$ denotes the communication load between $v_{i,k}$ and $v_{j,m}$. $v_{i,k}$ and $v_{j,m}$ are running instances of the respective nodes.

When condition (24) is satisfied, a local adjustment to the vertex instance assignment on the compute nodes will be triggered.

$$\frac{old_{tr(n_i,n_j)}}{new_{tr(n_i,n_j)}} > \alpha, \quad (n_i, n_j) \in Gn, \tag{24}$$

where $old_{tr(n_i,n_j)}$ denotes the current allocation scheme on the compute node set Gn. $new_{tr(n_i,n_j)}$ denotes the proposed new allocation scheme generated by the background monitoring module. α denotes a user-defined trigger threshold, determining whether the vertex instance allocation needs adjustment.

There exists a mapping relationship between the position adjustment of vertex instances and the compute nodes, denoted as $fs(v_{i,k}, v_{j,m}) : n_i \rightarrow n_j$. When the position of one vertex instance changes, there must exist another instance to replace it, i.e. Changes occur in pairs and $fs(v_{i,k}, v_{j,m})$ is a one-to-one relationship.

When the mapping relationship fs is obtained, it does not immediately trigger a migration action on the vertex instances. We have to consider the state of the compute node's CPU load in a comprehensive way, e.g. whether the pair of instances can be migrated without overloading the CPU of the compute node. The migration action is triggered when condition (25) is satisfied.

$$DLr_{n_i,[t_s,t_e]} + L_{v_{i,k}} < B_{n_i}, \tag{25}$$

where $DLr_{n_i,[t_s,t_e]}$ denotes the CPU load ratio of the target node to be migrated to by $v_{i,k}$. $L_{v_{i,k}}$ denotes the CPU consumption of this instance $v_{i,k}$. B_{n_i} is the maximum CPU load ratio per node set by the user.

As shown in Fig. 8, the streaming application depicted in Fig. 6 runs on 2 compute nodes node1 and node2. It can be seen that there is a huge amount of communication load between node1 and node2 in this deployment. The monitoring module can sense this problem by collecting information about the communication load among vertex instances and propose a new allocation scheme. The new scheme is compared with the old one and generates the set of tasks to be migrated.

When the vertex instances in the set are migrated, the CPU loads of node1 and node2 are not evaluated as overload. After the task migration, the distribution of vertex instances on the nodes is shown in Fig. 9. It is clear that the communication load between node1 and node2 is decreased by roughly 66% ((180-60)/180 = 0.66) by adjusting the vertex instances deployed on the nodes, which contributes to the high throughput and low response time of the system.

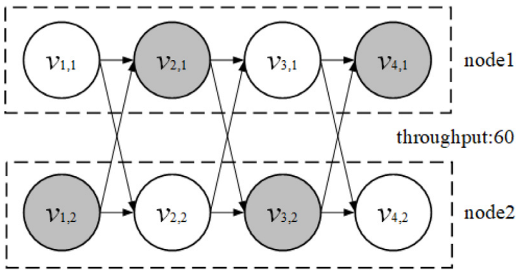


Fig. 9. Distribution of adjusted instances on nodes.

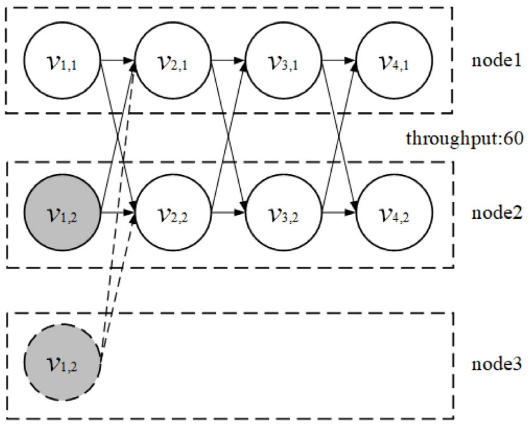


Fig. 10. Migration is triggered when a node's CPU is overloaded.

(2) The CPU load. The migration of vertex instances can also be triggered when the CPU load ratio of a compute node is greater than B_{n_i}. And the choice of the to-be-migrated vertex instance $v_{i,s}$ must satisfy condition (26) so that the instance migration cost is minimized.

$$v_{i,s} = \min(\sum_{v_{i,s}\in C_{n_i,v}} \sum_{v_{m,n}\in C_{n_i,v}} tr(v_{i,s}, v_{m,n})). \tag{26}$$

The vertex instance $v_{i,s}$ can choose a target node for migration where the upstream or downstream instance of $v_{i,s}$ is located on the target node. Set $M(n_i) = D(v_{i,s})\cup U(v_{i,s})$, where $M(n_i)$ denotes the migrated target set and n_i is the node that satisfies (25).

Consider migrating $v_{i,s}$ to an appropriate node in set $M(n_i)$ and obtain the reliability of node $n_j \in M(n_i)$ to get $r(n_j)$, where $r(n_j)$ denotes the reliability estimate of node n_j given by the system. Finally, migrate $v_{i,s}$ to the node with $max(r(n_j))$, where $n_j \in M(n_i)$. When set $M(n_i)$ is empty, obtain the reliability of all nodes $r(n_j)$, where $n_j \in Dn(n_i)$, and migrate $v_{i,s}$ to node $max(r(n_j))$.

As shown in Fig. 10, when node2's CPU is overloaded, instance $v_{1,2}$ running on node2 will be migrated to other node because it has minimal communication load with the other instances of the topology. The migration of instance $v_{1,2}$ follows two steps: (1) Local selection. As node1 has upstream or downstream instances of $v_{1,2}$, it is considered first. However, as node1 is a computationally intensive node, if instance $v_{1,2}$ is migrated to node1, it may overload node1's CPU according to condition (24). So no node is selected in this step. (2) Global selection. Global nodes in the cluster are considered in this step. Node3 has the lowest CPU load and is therefore selected. Instances $v_{1,2}$ is migrated to node3.

Task migration triggered by the communication load among nodes or by the CPU load of node can be accomplished by restarting the task. If there are stateful tasks in a streaming application, a checkpoint mechanism can be used to manage the states of these tasks. Before killing a stateful task, backing-up its state is required. After restarting the stateful task on a new node, its state needs to be recovered from the backup storage.

4.4. Energy saving strategies

CPU is the most important indicator of energy consumption for compute nodes [20,21]. In this paper, we therefore only focus on the CPU consumption. Optimizing node energy consumption requires sensing the CPU utilization of node on non-critical path and the amount of data processed by node on non-critical path. The computational performance of the node is known to be correlated with the frequency of the node's CPU f_i, the CPU utilization rate L_{n_i} and the amount of data processed by the node. Their correlation [18] is modeled in (27). Energy efficiency can be improved by making the node's CPU frequency equal to the frequency required to process data tuples, reducing the idle load energy consumption.

$$y_{n_i}(f_i) = \ln\left[f_i \cdot L_{n_i} + (1 - L_{n_i})\right] \cdot \theta, \tag{27}$$

where, $y_{n_i}(f_i)$ denotes the performance of node n_i, θ is a coefficient and can be estimated by (28). The minimum performance criterion of a given node is y_{min}, which satisfies $y_{n_i}(f_i) \geq y_{\min}$.

$$\theta = \frac{\sum_{v_{i,k}\in C_{n_i,v}} dco_{v_{i,k}}}{length_{C_{n_i,v}}}, \tag{28}$$

Where $dco_{v_{i,k}}$ denotes the computational delay of instance $v_{i,k}$ and $length_{C_{n_i,v}}$ denotes the size of set $C_{n_i,v}$.

Assuming that the streaming application has been deployed to the cluster, the resource usage and computational latency of each node is known, given the minimum computational performance of a node, the energy consumption of a node is minimized.

Then, our problem becomes:

$$MinimizeEn(f) = \sum_{i=1}^{length_{Dn}} En_i(f_i), \tag{29}$$

subject to

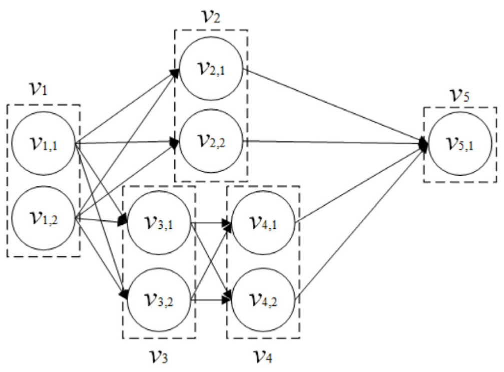
$$\begin{cases} y_{n_1}(f_1) \geq y_{\min}, \\ y_{n_2}(f_2) \geq y_{\min}, \\ \quad\cdots \\ y_n(f_n) \geq y_{\min}. \end{cases} \tag{30}$$

Please refer to Appendix for detail.

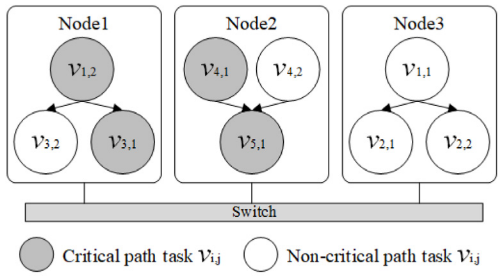
The minimum CPU frequency of a node can be calculated by considering the node's computing capability (Appendix). As shown in Fig. 11, a streaming application is deployed on 3 compute nodes, where $v_{1,2}$, $v_{3,1}$, $v_{4,1}$ and $v_{5,1}$ are on the critical paths calculated based on the stream application model. $v_{1,2}$ and $v_{3,1}$ are deployed on node1. $v_{4,1}$ and $v_{5,1}$ are on node2. None tasks running on node3 are on critical paths. Therefore, Node3's CPU frequency can be determined by the amount of tasks being performed on it. Given the information about node3's computing capability, the minimum CPU frequency of node3 can be obtained through the KKT mathematical model (in Appendix). Adjustment to the CPU frequency for nodes on non-critical paths do not have much impact on the system performance.

5. Er-Stream: Framework and algorithms

Based on the above formal modeling and analysis, we propose and implement Er-Stream, an energy efficient and runtime-aware framework for stream computing systems. To provide a better description of the proposal, this section discusses its overall framework and key algorithms, including DAG initialization algorithm, DAG runtime partial adjustment algorithm and energy saving algorithm.



(a) A sample streaming application.



(b) Task deployment.

Fig. 11. Frequency adjustment to nodes on non-critical paths.

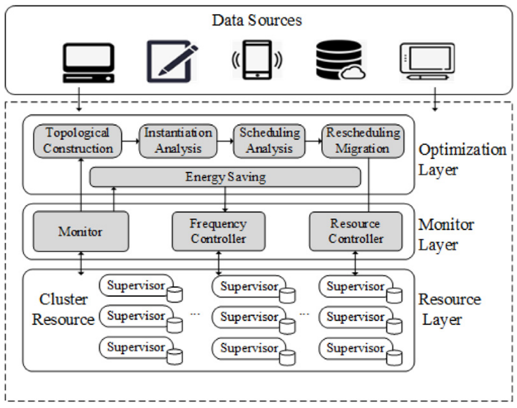


Fig. 12. Er-Stream framework.

5.1. System framework

The system framework of Er-Stream consists of an optimization layer and a monitoring layer, as shown in Fig. 12.

The monitor layer mainly includes a monitor module and two controller module. The monitor module collects various metrics of the cluster, such as communication load between nodes, rate of node resource utilization, etc. The collected data is submitted to the optimization layer. The controller module mainly includes a Resource Controller and a Frequency Controller. It receives decisions from the Scheduling Analysis module and the Energy Saving module in the optimization layer. The Resource Controller receives new scheduling information from the upper ReScheduling Migration module and deploys the new scheduling tasks onto the supervisors in the cluster. The Frequency Controller gets information about nodes on non-critical paths from the Energy Saving module and informs the compute nodes to adjust their CPU frequencies.

The optimization layer is mainly responsible for analyzing the data from the monitor layer and coming up with an adjustment suggestion. This layer mainly includes modules for topological construction, instantiation analysis, scheduling analysis, reliable migration and energy consumption analysis.

Topological construction refers to user designs the logical structure for a stream application, and determines the grouping strategy and the number of vertex instances for the stream application.

Instantiation analysis refers to the system creates one or more instances for each vertex. Instances of the same vertex have the same functionality and semantics. Multiple vertex instances can improve computational parallelism, but not the more the better.

Scheduling analysis refers to the system deploys vertex instances to compute nodes in the cluster, especially allocating the instances with potential communication load on the same compute node and keeping the instance number on each compute node balanced.

Reliable migration refers to the local adjustment of vertex instances using a reliable migration strategy to make the system sustainable when there are large fluctuations in data stream. In addition, the reliable migration strategy is also triggered to reduce system latency when the nodes are resource overloaded. It can be implemented via the IScheduler interface in Storm.

Energy consumption analysis refers to the system reduces the corresponding node's CPU frequency according to the computational load on the node. It opens a monitoring thread to listen to messages from the Frequency Controller. When the monitoring thread of node is notified by the Frequency Controller, the node triggers the CPU frequency adjustment strategy by using the CPUFreq tool [22].

Er-stream has the following advantages compared with traditional scheduling schemes. (1) When the input rate is stable, the traditional scheduling schemes do not consider the energy consumption of the cluster if the system has not reached its resource bottleneck. However, Er-stream can reduce the cluster's power consumption by adjusting the CPU frequency of nodes. (2) When there is high performance demand on the system, the traditional scheduling schemes will prolong nodes' downtime, causing the streaming application to work abnormally. Er-Stream can sense the bottleneck of the computing resources in the system. When the nodes are overloaded, a dynamic resource expansion can be conducted by Er-Stream.

5.2. DAG initialization algorithm

At the initial stage, the additional overhead caused by runtime scheduling can be reduced by graph partitioning of the DAG. The more instances are migrated during the runtime scheduling phase, the more costly the runtime scheduling will be. Therefore, DAG initialization using graph partitioning tries to control and reduce the number of runtime instance migrations.

Graph partitioning is an NP problem. Using heuristic genetic algorithm is one of the effective ways to solving this partitioning problem [23].

The heuristic genetic algorithm mainly consists of heuristic information and a valuation function. We further abstract the division of the DAG graph by maximizing the weights of the edges owned by each subgraph. The valuation function for graph partition is an objective function. It takes the average of the weights of each subgraph and is denoted as (31).

$$fit(pop_i) = \frac{\sum_{n_i \in Gn} \sum_{(v_{i,k}, v_{j,m}) \in C_{n_i,v}} sum + tr(v_{i,k}, v_{j,m})}{length_{Gn}}. \tag{31}$$

where $fit(pop_i)$ denotes the fitness value of the ith graph partition scheme pop.

Algorithm 1: Graph partition algorithm during initialization.

```
   Input: $G = (V(G), E(G))$, $Gn = \{n_i | 1 \le i \le num\}$;
   Output: $C_{n_i,v}$, $n_i \in Gn$;
1  while User set population size $n$ do
2      Generate a chromosome using a random method
         $cs(n) = \{v_{j1}, v_{j2}, \cdots, v_{jn}\}$;
3      Add this chromosome to the initial population
         $pop(1) \to cs(n)$;
4  end
5  Obtain the initial population $pop(1)$, $t := 1$, the
     number of iterations is 1;
6  while User sets the number of iterations $t$ do
7      For each chromosome $pop_i(t)$ in population $pop(t)$,
         calculate its fitness value.;
8      $u_i = fit(pop_i(t))$;
9      if $u_i < uf_i$, $uf_i$ is a user set value then
10         With probability: $p_i = \frac{u_i}{\sum_{j=1}^{N} u_j}$ randomly, use
             roulette wheel to randomly select some
             chromosomes from $pop_i(t)$ to form a new
             population
             $newpop(t + 1) = \{pop_j(t) | j = 1, 2, ..., N\}$;
11         Crossover with probability $P_c$ to generate some
             new chromosomes and get a new population
             $crosspop(t + 1)$;
12         With a small probability $P_m$ of mutating a gene
             on a chromosome to form
             $mutpop(t + 1)$; $t := t + 1$;
13         Form a new group $pop(t) = mutpop(t + 1)$;
14     end
15 end
16 return The chromosome with the maximum value of $u_i$
     in $pop(t)$;
```

The algorithm for graph partition in the initialization phase is described in Algorithm 1.

The input of this algorithm includes the stream application $G = (V(G), E(G))$ and the available nodes $Gn = \{n_i | 1 \le i \le num\}$. The output is the allocation $C_{n_i,v}$, $n_i \in Gn$ corresponding to each compute node.

Steps 1 to 4 initialize the population size $pop_i(t)$. A real number encoding is used, making each vertex instance $v_{i,k}$ of the graph G a gene fragment. Steps 7 to 8 calculate the fitness value $fitness(pop_i(t))$ of each chromosome, which is mainly used as a criterion to distinguish between good and bad individuals in a population, and is the driving force of the algorithmic evolutionary process, the only basis for natural selection. Step 10 is a selection operation, in which good individuals are randomly selected by a roulette wheel spin, so that each chromosome has a chance to enter the next generation. The fitness scaling method is used to set the magnitude of each individual probability P_i. A roulette wheel is constructed using the probability P_i of each individual.

Step 11 is the crossover operation, using two adjacent chromosomes for two-point crossover. Two-point crossover is to set two crossover points and swap the code strings between the crossover points with each other. The crossover points are randomly generated. In the process of crossover, gene conflicts may occur, and partial match crossover (PMX) is used to resolve gene conflicts. Step 12 is the mutation operation, which is mainly to maintain the diversity of the population. Here it is used to repair and replenish certain genes that may be lost during the selection crossover.

The algorithm uses swapping variants to generate two random numbers and swap their gene fragments. The deployment scheme generated by algorithm 1 costs 64 ms. Its time complexity is $O(n * m)$, where n is the number of iterations and m is the population size.

5.3. DAG runtime partial adjustment algorithm

To reduce the latency caused by data tuple processing, the monitor module must be able to sense (1) stream fluctuations of application G running on the set of compute nodes $Gn = \{n_i | 1 \le i \le num\}$, and (2) change of CPU load rate of compute node $n_i \in Gn$. To meet these two requirements, the Er-Stream decision algorithm is divided into two phases.

In the first phase, the monitor component scans the CPU load ratio $Lr_{n,[t_s,t_e]}$ of each compute node. When the condition $Lr_{n,[t_s,t_e]} > B_n$ is satisfied, Er-Stream triggers the instance migration strategy.

The node's CPU load detection algorithm is described in Algorithm 2.

Algorithm 2: CPU load detection algorithm.

```
   Input: $Lr_{n,[t_s,t_e]}$, $v_{i,k}$, $L_{v_{i,k}}$, $r(n_i)$, $B_n$;
   Output: $min_{v_{i,k}} \to tn$;
1  Declare migration target nodes $tn$;
2  $min_{v_{i,k}}$ denotes the minimum communication load
     with other vertex instances on node $n_i$;
3  for $l_i$ being the CPU load ratio for nodes in set $Gn$ do
4      Determine whether the CPU of each node is
         overloaded or not;
5      if $l_i > B_n$ then
6          Fetch downstream nodes $D(v_{i,k})$ and upstream
             nodes $U(v_{i,k})$ of the $min_{v_{i,k}}$, and assign them to
             $M(n)$;
7          for $n_i \in M(n)$ do
8              if $DLr_{n_i} + L_{\min v_{i,j}} > B_n$ then
9                  $Remove(M(n_i))$;
10             end
11         end
12         if $M(n_i)$ is empty then
13             $tn = max(r(n_i), n_i \in Dn(n_i))$;
14         end
15         $tn = max(r(n_i), n_i \in M(n_i))$;
16     end
17 end
18 return $min_{v_{i,k}} \to tn$ ;
```

The input of this algorithm includes the CPU load factor $Lr_{n,[t_s,t_e]}$ for each compute node, the CPU load factor $L_{v_{i,k}}$ for each vertex instance, reliability of each node $r(n_i)$ and CPU resource constraint threshold B_n. The output is the node to which the vertex instance is migrated. Step 3 to 5 select the compute nodes that are CPU overloaded and select the vertex instances from the nodes that will be moved out. Step 6 to 11 select the nodes that meet the conditions from the upstream or downstream nodes. In step 12 to 14, if the condition is not satisfied by either upstream or downstream nodes, the node with the highest reliability among all nodes will be selected. In step 15, the node with the highest reliability is selected from the upstream or downstream nodes. When a node is overloaded, a dynamic resource expansion can be conducted by the algorithm 2. The time complexity of Algorithm 2 is $O(n * m)$, where n denotes the size of Gn and m denotes the size of Dn.

In the second stage, the monitor module checks whether the communication load between compute nodes Gn changes significantly. If true, reevaluate the graph G to get $new_{tr(n_i,n_j)}$.

The algorithm for runtime scheduling is described in Algorithm 3.

Algorithm 3: Runtime scheduling algorithm.

```
     Input: $G$, $Lr_{n,[t_s,t_e]}$, $v_{i,k}$, $L_{v_{i,k}}$, $tr(v_{i,k}, v_{j,m})$, $\alpha$;
     Output: $Set(fs(v_{i,k}))$;
 1   Set $nos$ stores instances of vertices whose positions
       have changed;
 2   for $t_i \in G$ do
 3       Build Matrix $IEF_i \leftarrow tr(v_{i,k}, v_{j,m})$;
 4       Call the genetic algorithm interface, output the best
           allocation scheme and the fitness value
           $(BGrap\_e, bfitness) \leftarrow GAUtils(IET_i)$;
 5       if $\frac{oldFitness}{newFitness} > \alpha$ then
 6           Compare the old and new assignment schemes,
               and select the vertex instances $v_{i,k}$ whose
               positions have changed
               $fs(v_{i,k}) \leftarrow compareScheme(BGrap\_e, Grap)$;
 7           if $L_{v_{i,k}} + L_{n_j} - Lv_{k,i} < B_n$ and
               $L_{v_{k,i}} + L_{n_i} - Lv_{i,k} < B_n$ then
 8               $v_{i,k}$ and $v_{k,i}$ are found in pairs;
 9               $nos \leftarrow fs(v_{i,k}) : n_i \rightarrow n_j$;
10               $nos \leftarrow fs(v_{k,i}) : n_j \rightarrow n_i$;
11           end
12           $assigned(nos)$;
13       end
14   end
15   return $nos$;
```

The input of this algorithm includes the stream application G, the CPU load factor $Lr_{n,[t_s,t_e]}$ for each node $n_i \in Gn$, the vertex instance $v_{i,k}$ load factor $L_{v_{i,k}}$, the communication load between vertex instances $tr(v_{i,k}, v_{j,m})$ and the topology local adjustment trigger threshold α. The output is the set of position change for the vertex instances. In step 3 to 4, a matrix of application G is constructed and passed into Algorithm 1, which returns the new allocation scheme as well as the fitness value. In steps 5 to 6, if the ratio of the old and new schemes is greater than the threshold α, the vertex instance whose position has changed from the old to the new scheme is selected. Steps 7 to 11 determine whether the instances can be migrated to the destination node without overloading the CPU of the destination node. The time complexity of Algorithm 3 is $O(n)$, where n denotes the number of vertex instances.

5.4. Energy saving algorithm

When a node performs a task, its frequency can be dynamically adjusted to reduce its energy consumption according to the size of the processed task, providing that the node performance is not affected. Dynamic adjustment to node's CPU frequency can be achieved by calling the CPUFreq interface, which provides a lightweight CPU scaling monitor and is a powerful CPU frequency management tool.

As discussed above, a binary search algorithm is able to return optimal results under the constraints of keeping node performance. The algorithm for optimizing node energy consumption is described in Algorithm 4.

The input of this algorithm includes the correlation $y_{n_i}(f_i)$ between the CPU frequency f_i and CPU resource utilization L_{n_i} of node n_i, the minimum value of node performance y_{min} and the functional relationship $\gamma(f)$ between γ and f_i. The output is node's optimal frequency. Step 1 calculates the range value of γ and records the maximum and minimum. Step 4 calculates the frequency f corresponding to γ_{mid}. Step 5 to 7 determine whether the node performance at frequency f meets the requirements. In step 8 to 11, if the node performance requirements are not met, make adjustment to lb or rb. The time complexity of this algorithm is $O(log_2 n)$ where n is the range of γ.

Algorithm 4: Energy saving algorithm.

```
     Input: $y_{min}$, $y_{n_i}(f_i)$, $\gamma(f)$;
     Output: $f_i$;
 1   Compute $lb$ and $rb$ according to (A.13); // $lb$ and $rb$ are
       the upper and lower bounds of the search region $\gamma$ ,
       respectively;
 2   while $rb - lb > \mu$ do
 3       $mid \leftarrow {(lb + rb)}/_{2}$ ;
 4       $f \leftarrow \gamma_{mid}$;
 5       if $y_{\min} \le y_{n_i}(f) \le y_{\min} + c$ then
 6           return $f$;
 7       end
 8       if $y_{n_i}(f) \ge y_{\min} + c$ then
 9           $rb \leftarrow mid$;
10       else
11           $lb \leftarrow mid$;
12       end
13   end
```

Table 1
Software configuration of the Er-Stream.

Software	Version
Ubuntu	Ubuntu16.03
Storm	apache-storm-1.0.2
JDK	jdk1.7.64 bit
Zookeeper	zookeeper-3.4.6
Python	python 2.7.2
MySql	MySql-5.1.7
Power Detection	PowerTop 2.9

6. Performance evaluation

Experiments are conducted to evaluate whether the proposed scheduling algorithm can improve the system performance. In this section, the experimental environment and parameter settings are first discussed, followed by the result analysis of two stream applications, Top_N and WordCount.

6.1. Experimental environment and parameter setup

The Er-Stream framework is deployed above Storm-1.0.3 on Ubuntu 16.03. The experiments are conducted on a computing cluster in the Computer Architecture Laboratory of China University of Geosciences, Beijing. The cluster consists of twelve computers with two as nimbus nodes and ten as supervisor nodes, and the monitor component deployed together with the nimbus nodes. Three computers (three multiplexed with the supervisor) deploy the Zookeeper cluster. Detailed environment configuration is shown in Table 1. Detailed Server configuration is shown in Table 2.

Two stream applications Top_N and WorldCount are run to evaluate the system latency and throughput.

As shown in Fig. 13, the topology of Top_N is composed of four vertices v_1, v_2, v_3 and v_4, where the numbers of vertex instances for each vertex are 4, 8, 6 and 2, respectively. The function of each vertex is shown in Table 3. The grouping strategies between

Table 2
Server configuration.

Hardware	Performance
VCPU	2
Network Baseline Bandwidth	100 Mbps
Memory	1.2 GB
Hard Disk Drive	60 GB

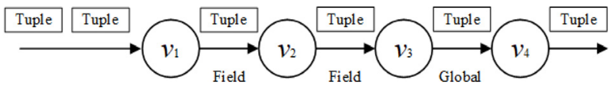


Fig. 13. Logical graph of Top_N.

Table 3
Vertex function of Top_N.

Vertex	Instances	Function
v_1	4	Read words from data stream
v_2	8	Count words
v_3	6	Rank words by count
v_4	2	Merge all ranks from upstream

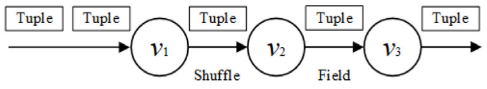


Fig. 14. Logical graph of WordCount.

Table 4
Vertex function of wordcount.

Vertex	Instances	Function
v_1	6	Read sentence from data stream
v_2	18	Split words of sentence
v_3	8	Count words

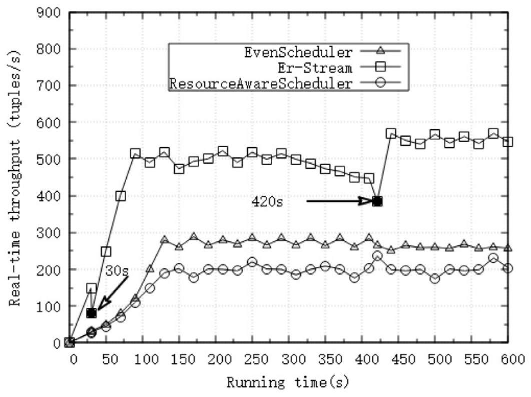


Fig. 15. System throughput of WordCount under stable input rate.

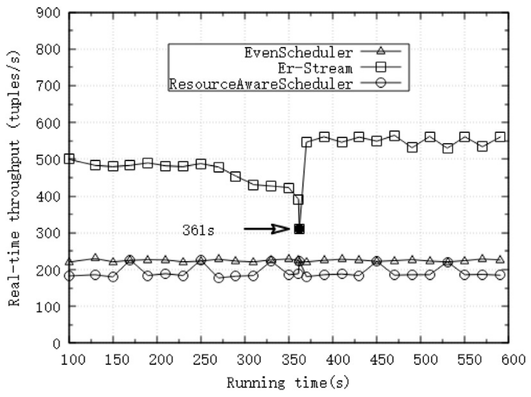


Fig. 16. System throughput of Top_N under stable input rate.

vertices are field grouping, field grouping, and global grouping from left to right.

As shown in Fig. 14, the topology of WordCount is composed of three vertices, v_1, v_2 and v_3, where the numbers of vertex instances for each vertex are 6, 18 and 8, respectively. The function of each vertex is shown in Table 4. The grouping strategies between vertices are shuffle grouping and field grouping from left to right.

Among the four built-in schedulers on Storm introduced in Section 2, EvenScheduler and ResourceAwareScheduler are frequently used in industrial practice. Therefore, in this experiment, we compare the proposed Er-Stream framework with them.

6.2. Performance results

The performance evaluation metrics adopted are system throughput ST and system response time RT.

6.2.1. System throughput

System throughput is calculated based on the number of data tuples processed per second. It reflects the cluster's processing capacity. The higher the system throughput, the more capable the system is of processing the data.

When the input rate remains stable, Er-Stream has higher real-time throughput compared to the EvenScheduler and ResourceAwareScheduler.

As shown in Fig. 15, the input rate of the data stream is 1000 tuples/s. In [100s–350s], the average throughput of Er-Stream is 506 tuples/s, and in [400s–600s], the average throughput of Er-Stream is 554 tuples/s. In [100s–600s], the throughputs of EvenScheduler and ResourceAwareScheduler are 271 tuples/s and 201 tuples/s, respectively. For the given WordCount application, the average throughput of Er-Stream is higher than those of the EvenScheduler and ResourceAwareScheduler when the input rate is stable.

As shown in Fig. 16, when Top_N is running, the throughput of the Er-Stream is higher compared to those of the EvenScheduler and ResourceAwareScheduler. In [100s–350s], the average throughput of Er-Stream is 486 tuples/s, and in [400s–600s], the average throughput of Er-Stream is 552 tuples/s. In [100s–600s], the throughputs of the EvenScheduler and resourceAwareScheduler are 224 tuples/s and 184 tuples/s, respectively.

There are inflection points at the 30s and 420s in Fig. 15 and 361s in Fig. 16. At the DAG initialization stage, since the communication load between vertex instances cannot be predicted, our division strategy is to minimize the number of edges connecting subgraphs, but this allocation may not be the best solution. After a period of time (which can be set, in this paper it is set to 30s), the communication load between instances can be obtained by the monitoring module. After 30s, The monitoring module continuously evaluates the system performance and decides whether to make a task migration. At the 420s in Fig. 15 and the 361s in Fig. 16, task rescheduling is triggered because the computational characteristics of the data may change significantly, resulting in changes in the data stream feed and the amount of data computed at each node.

Er-Stream has higher system throughput compared to EvenScheduler and ResourceAwareScheduler when the stream input rate is varying over time.

As shown in Fig. 17, when WorldCount is running, the data input rate increases from 1000 tuple/s to 2000 tuple/s at the

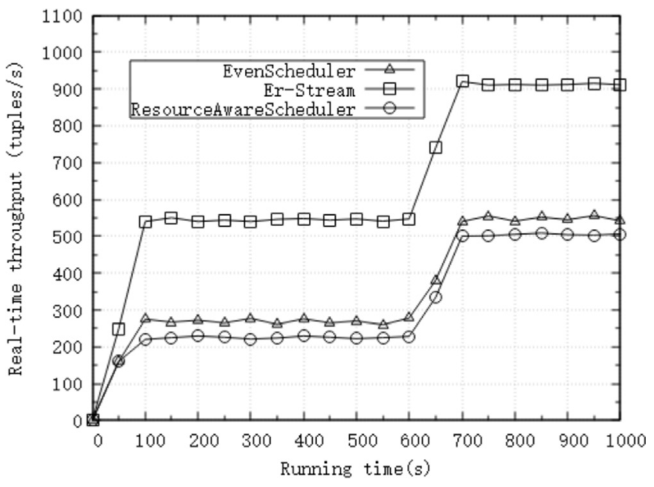


Fig. 17. System throughput of WordCount under increasing input rate.

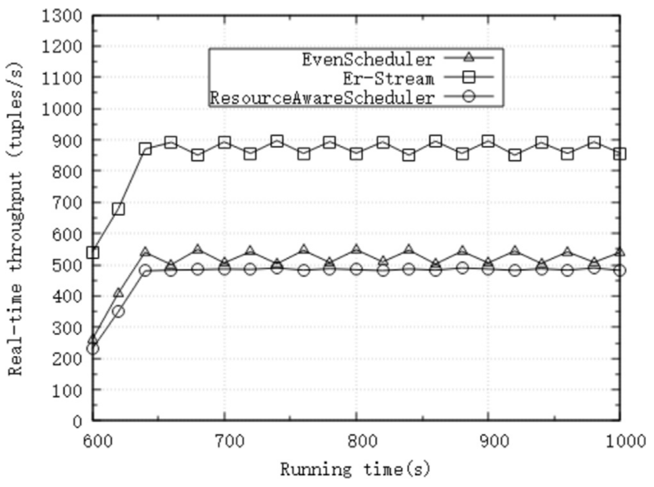


Fig. 18. System throughput of Top_N under increasing input rate.

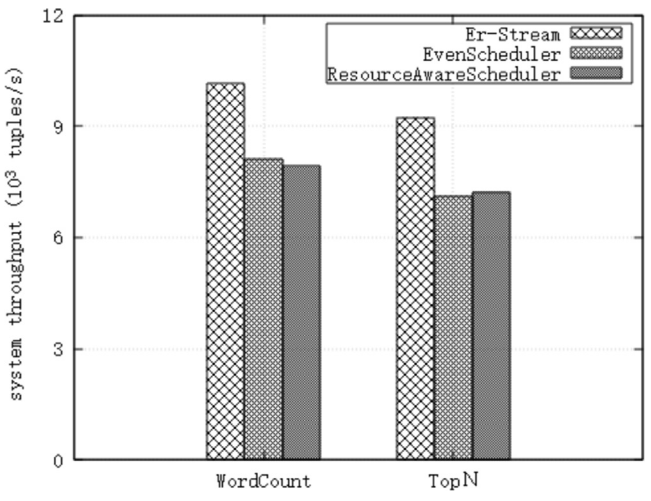


Fig. 19. Comparison of throughput's bottleneck under Er-Stream, EvenScheduler and ResourceAwareScheduler.

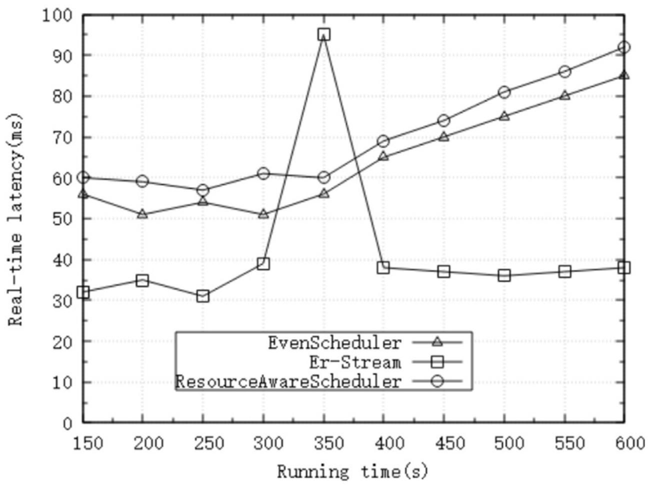


Fig. 20. System latency of WorldCount under stable input rate.

600s. The data input rate is 1000 tuple/s in [0s, 600s] and 2000 tuple/s in [600s, 1000s]. Under steady input, the throughput of Er-Stream varies from 506 tuple/s to 912 tuple/s and the throughputs of EvenScheduler and ResourceAwareScheduler range from 271 tuples/s to 546 tuples/s and from 201 tuples/s to 506 tuples/s, respectively. The average throughput of Er-Stream is still higher than those of the EvenScheduler and ResourceAwareScheduler when the input rate is increasing.

As shown in Fig. 18, when Top_N is running, the throughput of Er-Stream is still higher, compared to those of the EvenScheduler and ResourceAwareScheduler. They are 876 tuples/s, 526 tuples/s and 484 tuples/s, respectively.

The streaming applications WordCount and Top_N are deployed on the same 6 compute nodes by using Er-Stream, EvenScheduler and ResourceAwareScheduler, respectively. The bottleneck of system throughput is tested by continuously increasing the input rate. As shown in Fig. 19, it can be seen that Er-Stream has a stronger data processing capability than EvenScheduler and ResourceAwareScheduler when using the same amount of computational resources. It is because Er-Stream merges those operations requiring high communication load. Therefore, the bottleneck of system throughput under Er-Stream is greater than those under EvenScheduler and ResourceAwareScheduler, given the same amount of computing resources.

6.2.2. System response time

System response time RT is the time interval from the input of a topology to the output of result. It mainly includes processing delay, propagation delay and queuing delay. We mainly consider the propagation delay and processing delay in this paper. If the system response time is too long, it will greatly downgrade user experience. The shorter the system response time, the better the real-time system performance. Storm platform has a built-in UI allowing user to access RT data through a browser.

Er-Stream has lower latency compared to the Storm's two popular built-in scheduling strategies EvenScheduler and ResourceAwareScheduler under stable input. As shown in Fig. 20, the input rate for the WordCount is 1000 tuples/s, and the computation node is continuously pressurized. Er-Stream triggers the migration strategies at the 350s and stabilizes the computational latency of the system afterwards. At [400s–600s], the average latency of Er-Stream is 36 ms, while the computational latency of the built-in scheduling strategies of Storm is increasing over time.

As shown in Fig. 21, when Top_N is running, the real-time performance of Er-Stream is clearly better than those of the EvenScheduler and ResourceAwareScheduler. In [300s–600s], the average computation latency of Er-Stream, EvenScheduler and

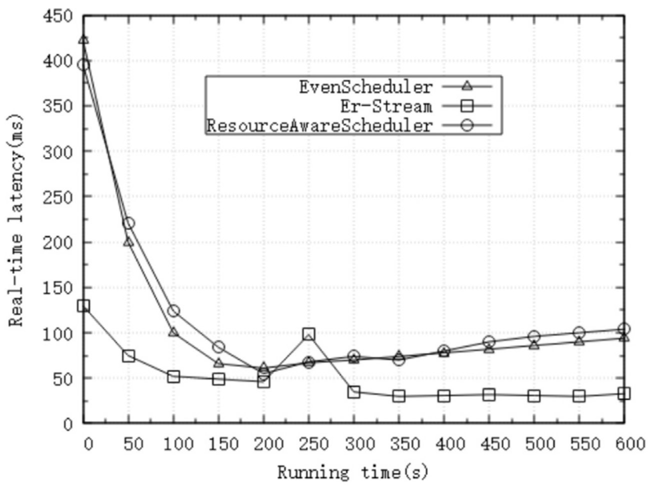


Fig. 21. System latency of Top_N under stable input rate.

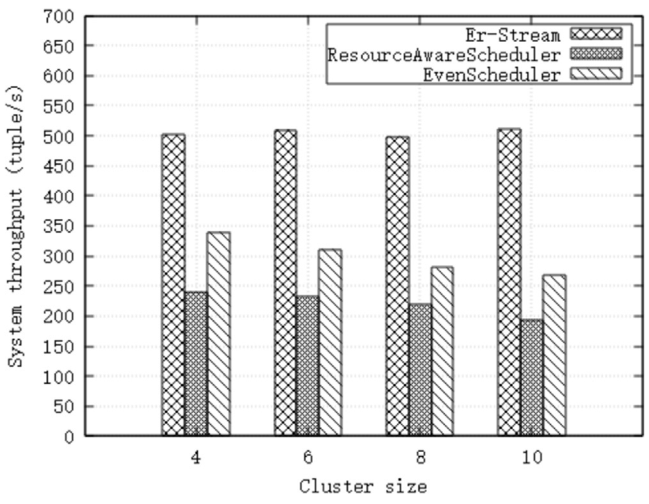


Fig. 22. System throughput with increasing cluster scale under stable input rate.

ResourceAwareScheduler are 34 ms, 79 ms and 87 ms, respectively.

6.3. Cluster size and CPU utilization of the compute nodes

Keeping the data input rate stable at 1000 tuples/s, we deploy the WordCount application on 4, 6, 8 and 10 compute nodes, respectively. When the cluster size increases, Er-Stream has higher throughput compared to the EvenScheduler and ResourceAwareScheduler. In addition, when Er-Stream deploys an application, it does not distribute the vertex instances evenly over all the compute nodes, but using the right number of compute nodes. As shown in Fig. 22, when the cluster size is 4 compute nodes, the average throughputs of Er-Stream, EvenScheduler and ResourceAwareScheduler are 502 tuples/s, 340 tuples/s and 240 tuples/s, respectively. When the cluster size is 6, Er-Stream's throughput remains stable, while the EvenScheduler's throughput drops to 311 tuple/s and ResourceAwareScheduler's throughput drops to 232 tuple/s. When the cluster size is 8, Er-Stream's throughput remains stable, the EvenScheduler's throughput drops to 281 tuple/s and the ResourceAwareScheduler's throughput drops to 219 tuple/s. When processing non-intensive tasks, Er-Stream remains stable performance even if the cluster size grows, while the EvenScheduler and ResourceAwareScheduler have the performance decreased even with an increased cluster size. Er-Stream remains stable as the cluster size grows. The reason behind is that the number of nodes used by Er-Stream remains constant regardless of the size of the cluster. When the computing resources used by the streaming application are sufficient, expanding the cluster size can trigger extra communication delays by EvenScheduler and ResourceAwareScheduler, leading to decreased throughput even with an increased cluster size.

To enable the compute nodes to better process data tuples, the CPU load of nodes is tested under pressure and their CPU utilization rate is calculated after implementing the reliable migration strategy.

As shown in Fig. 23, when the input rate is 1000 tuples/s, in [40s–160s] the CPU utilization rate of node1, node2, node3 and node4 are 60%, 74%, 57%, and 6%, respectively. It can be seen that each node is not overloaded in terms of resources.

At the 200s, the input rate is increased to 2000 tuples/s. It can be found that at the 213s, node2 is CPU overloaded and the CPU utilization rate reaches 91%. At this point, reliable migration is triggered on node2, and some tasks are migrated to node4 according to the reliability of the target migrating node (node4).

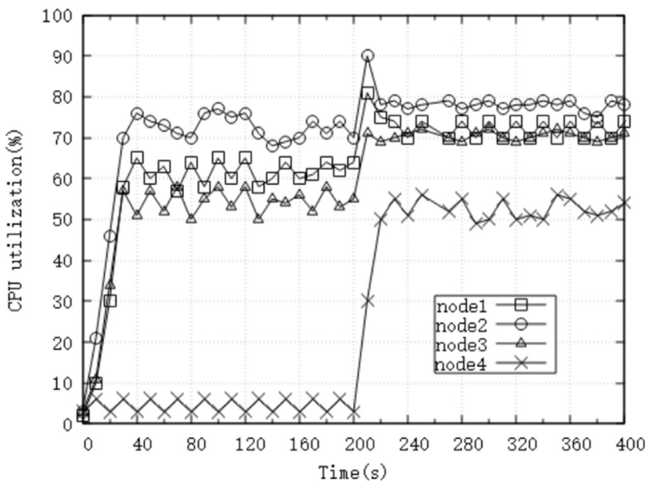


Fig. 23. CPU utilization of compute nodes under increasing input rate.

After that, the CPU utilization rate of node2 drops and is stabilized below 80% afterwards. It is proved that Er-Stream can monitor node's resources in real time and trigger task migration when a node becomes overloaded.

6.4. Energy saving strategy

The Top_N topology is deployed on 8 compute nodes by EvenScheduler. The power consumption of each node under a stable input rate is separately monitored by the POWERTOP tool. It can be observed that the power consumption is changed when the node CPU frequency is adjusted. Their correlation is shown in Fig. 24. When the system throughput is kept at a stable rate, the average power consumption of node1, node2 and node6 remains constant because the tasks running on these nodes are on critical paths and no CPU frequency changes are made. The average power consumption of node3, node4, node5, node7 and node8 decreases because their CPU frequencies are adjusted lower using the CPUFreq tool. The cluster's power consumption is therefore reduced due to the decrement of these 5 nodes. The frequency adjustments to these 5 nodes do not have much influence on the system response time because they are not on critical paths.

As shown in Fig. 25, the Top_N topology is deployed on 4 nodes (nodes 1–4) of the cluster by ResourceAwareScheduler. It

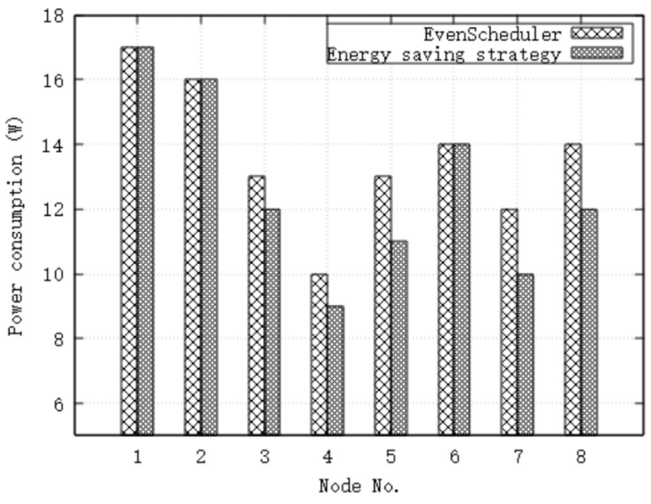


Fig. 24. Power consumption of TOP_N nodes under EvenScheduler.

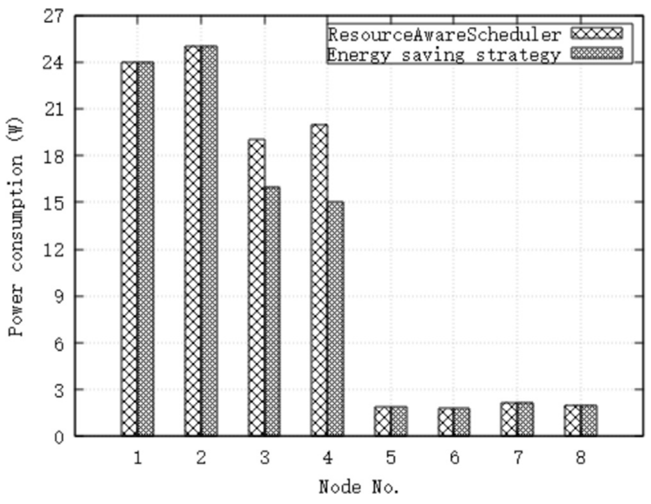


Fig. 25. Power consumption of TOP_N nodes under ResourceAwareScheduler.

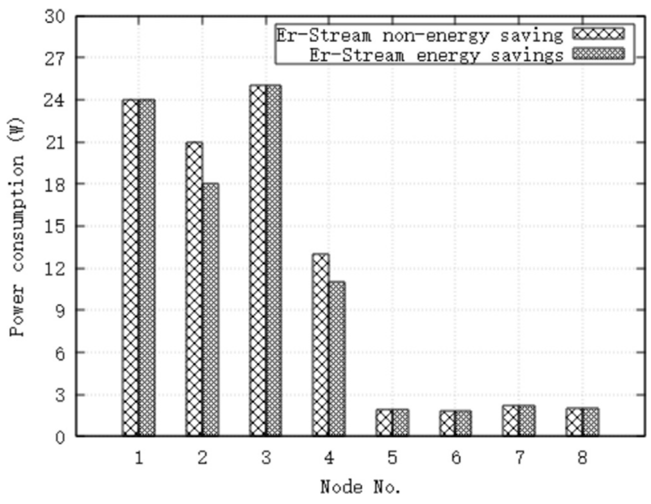


Fig. 26. Power consumption of TOP_N nodes under Er-Stream.

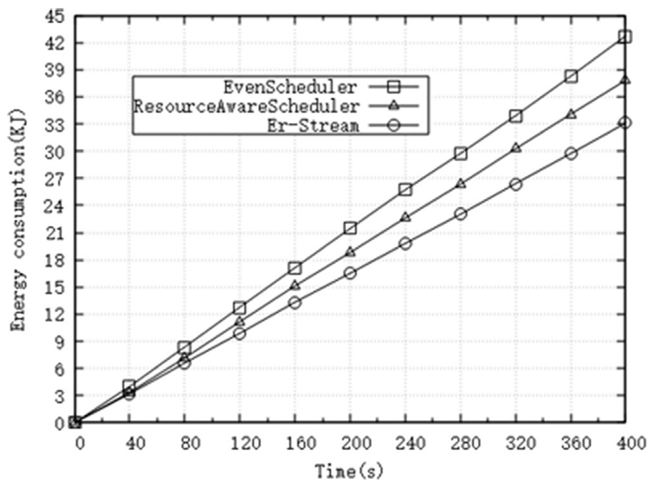


Fig. 27. Energy consumption of Top_N under stable input rate.

is clear that there is an apparent drop in power consumption for node3 and node4 because the tasks running on these 2 nodes are not on critical paths. Nodes 5–8 have low power consumption because they are in idle state.

As shown in Fig. 26, the Top_N topology is deployed on the same 8 compute nodes by Er-Stream. From this figure, we can observe that the average power consumption of node1 and node3 remains constant. For node2 and node4, their average power consumption decreases. For nodes 5–8, the consumption is significantly lower than the rest. Nodes 1–4 have higher average power consumption because they are the only nodes used to deploy the Top_N topology.

It can be found in Figs. 24–26 that the power consumption among the Top_N nodes under EvenScheduler, ResourceAware Scheduler and Er-Stream Scheduler is different because the dynamic CPU frequency scaling supported by Linux Operating System is used by the non-energy strategy based on resource load. The power consumption of Top_N nodes with our energy saving strategy is less than these with the default strategy, as Er-Stream reduces the CPU frequencies of these non-critical-paths nodes based on their minimum values.

The default dynamic CPU frequency scaling is used to adjust the CPU frequency of a node deployed with tasks in a distributed environment based on resource load. It is a better approach for a single node. However, for the whole cluster, it may cause extra power consumption. From Figs. 24–26, we get the cluster's power consumption for Top_N under EvenScheduler, ResourceAwareScheduler and Er-Stream energy is 109 W, 94 W and 85 W, respectively. The average power consumption under Er-Stream is lower than that of the EvenScheduler and ResourceAwareScheduler when the system throughput rate is stable.

The energy consumption of the cluster for Top_N under Er-Stream, EvenScheduler and ResourceAwareScheduler is separately tested. It is calculated by first working out the power consumption of each node, then summing it up. The energy consumption of each node can be obtained by multiplying the node power consumption by its running time. The energy consumption under the schedulers is shown in Figs. 27 and 28.

As shown in Fig. 27, when the data stream rate is stable at 2000 tuple/s, the energy consumption of the cluster for Top_N under Er-Stream is smaller than that of EvenScheduler and ResourceAwareScheduler. The longer the running time, the more obvious the gap. We analyze the reason behind Er-Stream's lower energy consumption in two main aspects: (1) The impact of

Table 5
Related work comparison.

Aspects	Related works												Our work
	[24]	[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	
Runtime-aware	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Communication-aware	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Resource constraint	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓
Reliable migration	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓
Energy consumption	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓

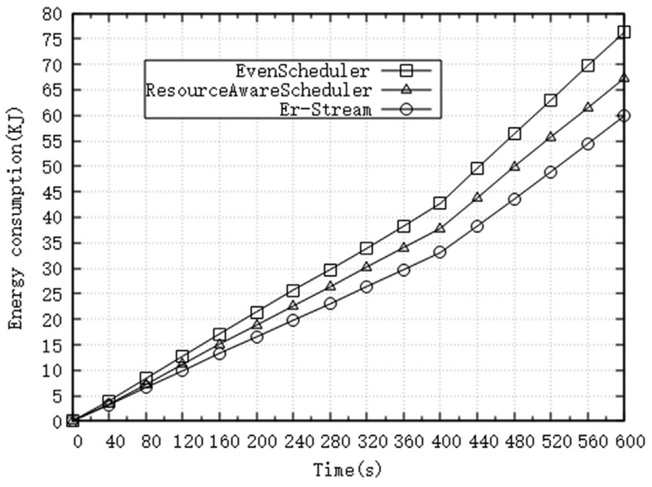


Fig. 28. Energy consumption of Top_N under increasing input rate.

compute node number on energy consumption under different scheduling schemes (Er-Stream, EvenScheduler and ResourceAwareScheduler). (2) The impact of CPU frequency under Er-Stream. It is found that the number of compute nodes employed by Er-Stream is reduced, compared with other schedulers, and Er-Stream has lower power consumption by adjusting node's CPU frequency.

As shown in Fig. 28, when Top_N is running with a low trigger rescheduling factor value (e.g. $\alpha = 0$), and the data input rate increases from 2000 tuple/s to 4000 tuple/s at the 400 s, it can be seen that the growth rate of the average energy consumption under Er-Stream, EvenScheduler and ResourceAwareScheduler becomes faster after 400 s. Moreover, Er-Stream remains a lower energy consumption compared to EvenScheduler and ResourceAwareScheduler.

7. Related work

In this section, we review three major categories of related work: stream application deployment optimization, resource constraints, and reliable scheduling & energy consumption in stream computing. The summary of the comparison between our work and other closely related works is given in Table 5.

7.1. Stream application deployment optimization

The deployment of tasks for streaming applications is critical in improving system throughput and reducing latency. In recent years, there has been great interest in stream application deployment on Storm. However, it is challenging to find an optimal deployment due to the fluctuation of data stream, the processing capacity of compute nodes and other factors.

In [24], a heuristic scheduling algorithm was proposed to place highly communicating tasks on the same compute node to find an optimal task allocation scheme in a heterogeneous cloud environment. However, data stream volatility and energy consumption of compute nodes was not considered.

In [25], a workload scheduling strategy based on graph partitioning algorithm was proposed to collect the runtime communication behavior of applications and create schedules using the METIS package for graph partitioning. Both computational resources of compute nodes and communication between tasks were considered, however, the scheduling scheme resulted in an equal amount of tasks processed by each compute node, which does not correspond to a realistic streaming environment.

In [26], an offline resource-aware scheduler was proposed. The algorithm used width-first search to rank the application topology to reduce inter-node communication and maximize throughput and resource utilization under user-specified resource constraints. However, the drawback is that users were frequently involved in this process.

In [27], the scheduling problem is modeled as a bin-packing variant and a heuristic-based algorithm is proposed to minimize the inter-node communication. The algorithm overcame the limitation of the static resource-aware scheduler. However, the impact of network characteristics on system performance was not considered.

In [36], a key/value store was added to each working node to reduce the granularity of task scheduling, from scheduling in processes to scheduling in threads. Each worker managed its own executor, reducing unnecessary overhead. However, the placement of communication-intensive tasks was not considered.

In [37], a novel predictive scheduling framework was proposed to enable fast and distributed stream data processing, featured with topology-aware modeling for performance prediction and predictive scheduling.

In [28], an adaptive online scheme was proposed to schedule and enforce resource allocation in stream processing systems, which guaranteed that the system could achieve less congestion under heavy load and less resource waste under light load.

In [29], the task assignment problem under stream computing systems was mainly studied by finding highly communicative tasks for which tasks were assigned by using graph partitioning algorithms and mathematical packages. However, the elastic variation of the data stream and the energy consumption of the compute nodes were likewise not considered.

Compared to them, the Er-Stream strategy uses heuristic algorithms to quickly partition the instance graph and uses fewer compute nodes to run streaming applications, reducing the latency of the system. It is able to dynamically sense and respond to changes of data stream.

7.2. Resource constraints of stream computing systems

While the impact of network latency on stream computing performance is substantial, CUP resources on compute nodes are also a factor that should not be ignored. When the CPU load of one compute node is too heavy, it will very likely affect the efficiency of data processing and even cause the node to go down. Therefore, it is helpful to optimize the load of a stream computing system.

In [38], an integrated solution is presented for load balancing, horizontal scaling and operator instance collocation, and the problem of load balance is modeled as a Mixed-Integer Linear Program to improve the load distribution in computing cluster. However, the number of nodes used by the application was not minimal, resulting in a waste of resources.

In [30], the allocation of priority and non-priority tasks was mainly considered, using the modified honey bee behavior inspired algorithm and the enhanced weighted round-robin algorithm, respectively, which improved the efficiency of resource utilization. However, the availability of computational nodes was not considered in the task migration process.

In [31], a new key-based workload partitioning framework was proposed. It assigned target worker threads to keys via hashing and routing strategies and supported dynamic workload assignment for stateful operators. However, this work only considered the short-term fluctuations of the data stream.

In [32], to load balance the system, a cloud-based resource scheduling scheme was proposed. It dynamically scheduled resources based on the current workload to avoid wasting resources and improved the operational efficiency of the system.

In [39], a new flow grouping scheme Partial Key Grouping (PKG) was proposed to optimize the standard hashing method using both key division and local load estimation. It solved the load imbalance problem to some extent.

Compared to them, Er-Stream is able to sense changes to the CPU load of compute nodes. It ensures the streaming applications use fewer compute nodes to process data without overloading CPU of the compute nodes.

7.3. Reliable scheduling and energy consumption in stream computing

When the deployment of a task is changed, the reliability of nodes is to be considered to keep the robustness of the system. The energy utilization can be improved if the energy consumption of nodes is controlled. In the past years, researchers have been optimizing reliable scheduling and energy consumption.

In [40], an Efficient Resource Allocation with Score scheme (ERAS) was proposed to provide reliable task scheduling for a limited number of heterogeneous virtual machines. It considered the earliest completion time of the task and the operational availability of the virtual machine to allocate resources to it. However, there was still room for improvement in terms of dynamic task scheduling efficiency.

In [33], an online adaptive framework for sensing runtime variations and environmental changes in real-time systems was proposed, taking soft-error reliability and lifetime reliability into account. In addition, dynamic migration tasks were performed under the condition that various constraints were satisfied.

In [34], an energy-aware real-time scheduling on a multi-core platform was proposed to minimize the expected energy consumption by considering processors on the same cluster running at the same speed and modeling the runtime energy consumption.

In [41], energy efficiency and resource utilization problems were studied. In a heterogeneous environment, the processing speed, energy consumption, and priority of compute nodes are different. An energy allocation and load balancing strategy based on data fitting and Lagrangian theory approach was used to solve these two problems.

In [35], the problem of allocating and placing resilient resources while satisfying energy cost minimization in a distributed system was investigated and a collaborative strategy was proposed from a system perspective. In addition, a prediction model was incorporated to estimate the system latency by predicting the future workload.

Compared to them, Er-Stream is able to evaluate node reliability based on factors such as node load. Karush–Kuhn–Tucker mathematical conditions are used to model node resource utilization and energy consumption to minimize node energy consumption of data center.

8. Conclusions and future work

In a fluctuating data stream environment, minimizing network communication and energy consumption, and keeping nodes meeting load constraints are the goals of a system implementation. Achieving these goals relies on the system that can intelligently monitor data stream size and node resource utilization to adaptively adjust instance deployment, and sense data center energy consumption to tailor the CPU frequency of node accordingly. In this work, we attempt to optimize the system performance from the perspective of minimizing the network transmission of data tuples by proposing an energy efficient and runtime-aware framework, Er-Stream. It handles changes in data stream and dynamically adjusts the instance migration on compute nodes, thus optimizing the system performance. At the same time, it senses the computational resources of each compute node to adjust the resource allocation, further improving the system performance. KKT mathematical constraints are used to model node resources and energy consumption to increase the energy utilization of the data center.

Currently, there is still room for Er-Stream to improve, e.g. it does not support state management. In the future, we will further:

(1) integrate state migration and fault tolerance into Er-Stream, making the system more efficient and reliable.
(2) consider more indicators when modeling energy consumption of nodes, such as I/O and memory.
(3) use Er-Stream in real-life large-scale stream computing environments, such as IoT, Telematics and financial risk control, to bring the system closer to the commercial sector and promote the development of community.

CRediT authorship contribution statement

Dawei Sun: Conceptualization, Methodology, Validation, Writing – original draft, Funding acquisition. **Yijing Cui:** Validation, Investigation, Writing – review & editing. **Minghui Wu:** Investigation, Data curation, Writing – review & editing. **Shang Gao:** Formal analysis, Investigation, Writing – review & editing. **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 under Grant No. 265QZ2021001; and MelbourneChindia Cloud Computing (MC3) Research Network.

Appendix. Energy saving strategy

In this appendix, we present the process of minimizing cluster energy consumption. As discussed in Section 4.4, the problem of energy consumption for cluster can be defined as

$$MinimizeEn(f) = \sum_{i=1}^{length_{Dn}} En_i(f_i), \tag{A.1}$$

subject to

$$\begin{cases} y_{n_1}(f_1) \geq y_{\min}, \\ y_{n_2}(f_2) \geq y_{\min}, \\ \quad \dots \\ \ y_n(f_n) \geq y_{\min}. \end{cases} \tag{A.2}$$

We just need to minimize En_i for each node n_i to finally minimize $En(f)$. This is a variable optimization problem and It can be well solved by meeting the Karush–Kuhn–Tucker conditions (KKT) [21], an optimization method for inequality constraints problems. For each node, the following is constructed:

$$L(f_i, \gamma) = En_i(f_i) + \gamma\left[y_{\min} - y_{n_i}(f_i)\right], \tag{A.3}$$

which can be differentiated to,

$$\frac{\partial L(f_i,\gamma)}{\partial f_i} = \frac{\partial En_i(f_i)}{\partial f_i} - \gamma \cdot \frac{\partial y_{n_i}(f_i)}{\partial f_i}, \tag{A.4}$$

where,

$$\frac{\partial En_i(f_i)}{\partial f_i} = h_{n_i} \cdot f_{\max}\left[c(\kappa-1)f_i^{\kappa-2} - \frac{P_s}{f_i^2}\right], \tag{A.5}$$

And,

$$\frac{\partial y_{n_i}(f_i, L_{n_i}, \theta)}{\partial f_i} = \frac{L_{n_i}}{f_i \cdot L_{n_i} + (1 - L_{n_i})}, \tag{A.6}$$

Equating (A.4) to 0

$$h_{n_i} \cdot f_{\max}\left[c(\kappa-1)f_i^{\kappa-2} - \frac{P_s}{f_i^2}\right] = \gamma \cdot \frac{L_{n_i}}{f_i \cdot L_{n_i} + (1 - L_{n_i})}. \tag{A.7}$$

We first analyze the relationship between γ and f_i. The Lagrangian factor γ can be treated as a function of f_i.

$$\gamma(f_i) = \frac{h_{n_i} \cdot f_{\max}}{L_{n_i}} \cdot \left[c(\kappa-1)\cdot f_i^{\kappa-2} - \frac{p_s}{f_i}\right] \cdot \left[f_i \cdot L_{n_i} + (1 - L_{n_i})\right], \tag{A.8}$$

where (A.9) is a factor of (A.8)

$$te(f_i) = \left[c(\kappa-1)\cdot f_i^{\kappa-2} - \frac{p_s}{f_i}\right]. \tag{A.9}$$

We have

$$te'(f_i) = c(\kappa-1)(\kappa-2)f_i^{\kappa-3} + \frac{p_s}{f_i^2} > 0. \tag{A.10}$$

Therefore, $te(f_i)$ is a monotonically increasing function with respect to f_i. Furthermore, (A.11) is another factor of (A.8).

$$f_i \cdot L_{n_i} + (1 - L_{n_i}), \tag{A.11}$$

where $L_{n_i} > 0$ and therefore (A.11) is also monotonically increasing with respect to f_i. We can consider $\gamma(f_i)$ as a monotonically increasing function with respect to f_i.

Considering Eq. (27), we get

$$y'_{n_i}(f_i) = \frac{L_{n_i}}{f_i \cdot L_{n_i} + (1 - L_{n_i})} > 0. \tag{A.12}$$

Therefore, $y_{n_i}(f_i)$ is also a monotonically increasing function with respect to f_i.

According to the above analysis, we can use a binary search to find the γ value. The frequency of the node corresponding to this γ satisfies $y_{\min} \leq y_{n_i}(f_i) \leq y_{\min} + c$, where c is the error factor. The search range of γ is.

$$\min\gamma(f_{i,\min}) \leq \gamma \leq \max\gamma(f_{i,\max}), \tag{A.13}$$

where $\min\gamma(f_{i,\min})$ is minimum frequencies of the nodes and $\max\gamma(f_{i,\max})$ is current frequency of the node.

References

[1] Y. Chen, X. Li, Research on big data application in intelligent safety supervision, in: 2017 IEEE 2nd International Conference on Big Data Analysis, ICBDA, 2017, pp. 473–477.

[2] J. Zhang, J. Chen, Y. Zheng, H. Yuan, Applications of big data in public safety emergency services, in: 2017 IEEE 2nd International Conference on Big Data Analysis, ICBDA, 2017, pp. 354–357.

[3] F. Matsebula, E. Mnkandla, A big data architecture for learning analytics in higher education, in: 2017 IEEE AFRICON, 2017, pp. 951–956.

[4] A. Imawan, J. Kwon, A timeline visualization system for road traffic big data, in: 2015 IEEE International Conference on Big Data, Big Data, 2015, pp. 2928–2929.

[5] A. Juneja, N.N. Das, Big data quality framework: Pre-processing data in weather monitoring application, in: 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing, COMITCon, 2019, pp. 559–563.

[6] S. Sameti, M. Wang, D. Krishnamurthy, Stride: Distributed video transcoding in spark, in: 2018 IEEE 37th International Performance Computing and Communications Conference, IPCCC, 2018, pp. 1–8.

[7] Twitter, Heron, https://github.com/apache/incubator-heron.

[8] Apache, Samza, http://samza.apache.org/.

[9] Apache, Storm, http://storm.apache/org/.

[10] J. Zhu, Y. Wang, D. Zhou, F. Gao, Batch process modeling and monitoring with lo.cal outlier factor, IEEE Trans. Control Syst. Technol. 27 (4) (2019) 1552–1565.

[11] Twitter, Twitter, https://twitter.com/.

[12] Linkedin, Linkedin, https://www.linkedin.com/.

[13] H. Wu, Research proposal: Reliability evaluation of the apache kafka streaming system, in: 2019 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, 2019, pp. 112–113.

[14] alibaba, aliyun, https://www.aliyun.com/.

[15] Y. Hou, X. Zhao, Q. Li, J. Chen, Y. Li, Z. Zheng, Solving large-scale NP-complete problem with an optical solver driven by a dual-comb 'clock', in: 2019 Conference on Lasers and Electro-Optics, CLEO, 2019, pp. 1–2.

[16] D. Sun, H. Yan, S. Gao, X. Liu, R. Buyya, Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams, J. Supercomput. 74 (2) (2018) 615–636.

[17] H. Cao, C.Q. Wu, L. Bao, A. Hou, W. Shen, Throughput optimization for storm-based processing of stream data on clouds, Future Gener. Comput. Syst. 112 (2020) 567–579.

[18] J. Huang, R. Li, X. Jiao, Y. Jiang, W. Chang, Dynamic DAG scheduling on multiprocessor systems: Reliability, energy, and makespan, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (11) (2020) 3336–3347.

[19] P. Jin, X. Hao, X. Wang, L. Yue, Energy-efficient task scheduling for CPU-intensive streaming jobs on hadoop, IEEE Trans. Parallel Distrib. Syst. 30 (6) (2019) 1298–1311.

[20] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, N. Guan, Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2019.

[21] Y. Ma, J. Zhou, T. Chantem, R.P. Dick, S. Wang, X.S. Hu, Online resource management for improving reliability of real-time systems on "Big–Little" type MPSoCs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (1) (2020) 88–100, http://dx.doi.org/10.1109/TCAD.2018.2883990.

[22] cpufreq, cpufreq, https://github.com/konkor/cpufreq.

[23] D. Chaudhary, A.K. Tailor, V.P. Sharma, S. Chaturvedi, HyGADE: Hybrid of genetic algorithm and differential evolution algorithm, in: 2019 10th International Conference on Computing, Communication and Networking Technologies, ICCCNT, 2019, pp. 1–4.

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

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

[26] Y. Peng, M. Hosseini, H. Hong, R. Farivar, R-Storm: Resource-aware scheduling in storm, in: Proceedings of the 16th Annual Middleware Conference, Association for Computing Machinery, New York, NY, USA, 2015, pp. 149–161.
[27] X. Liu, R. Buyya, D-Storm: Dynamic resource-efficient scheduling of stream processing applications, in: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems, ICPADS, 2017, pp. 485–492.
[28] S. Liu, J. Weng, J.H. Wang, C. An, Y. Zhou, J. Wang, An adaptive online scheme for scheduling and re-source enforcement in storm, IEEE/ACM Trans. Netw. 27 (4) (2019) 1373–1386.
[29] L. Eskandari, J. Mair, Z. Huang, D. Eyers, I-Scheduler: Iterative scheduling for distributed stream processing systems, Future Gener. Comput. Syst. 117 (2021) 219–233.
[30] K.D. Patel, T.M. Bhalodia, An efficient dynamic load balancing algorithm for virtual machine in cloud computing, in: 2019 International Conference on Intelligent Computing and Control Systems, ICCS, 2019, pp. 145–150.
[31] J. Fang, R. Zhang, T.Z.J. Fu, Z. Zhang, A. Zhou, X. Zhou, Distributed stream rebalance for stateful operator under workload variance, IEEE Trans. Parallel Distrib. Syst. 29 (10) (2018) 2223–2240.
[32] T.Z.J. Fu, J. Ding, R.T.B. Ma, M. Winslett, Y. Yang, Z. Zhang, DRS: Dynamic resource scheduling for real-time analytics over fast streams, in: 2015 IEEE 35th International Conference on Dis-Tributed Computing Systems, 2015, pp. 411–420.
[33] Y. Ma, J. Zhou, T. Chantem, R.P. Dick, S. Wang, X.S. Hu, Online resource management for improving reliability of real-time systems on "Big–Little"type MPSoCs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (1) (2020) 88–100.
[34] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, N. Guan, Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2019, pp. 156–168.
[35] X. Wei, L. Li, X. Li, X. Wang, S. Gao, H. Li, Pec: Proactive elastic collaborative resource scheduling in data stream processing, IEEE Trans. Parallel Distrib. Syst. 30 (7) (2019) 1628–1642.
[36] Z. Zhang, P. Jin, X. Wang, R. Liu, S. Wan, N-Storm: Efficient thread-level task migration in apache storm, in: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems, HPCC/SmartCity/DSS, 2019, pp. 1595–1602.
[37] T. Li, J. Tang, J. Xu, Performance modeling and predictive scheduling for distributed stream data processing, IEEE Trans. Big Data 2 (4) (2016) 353–364.
[38] K.G.S. Madsen, Y. Zhou, J. Cao, Integrative dynamic reconfiguration in a parallel stream processing engine, in: 2017 IEEE 33rd International Conference on Data Engineering, ICDE, 2017, pp. 227–230.
[39] M.A.U. Nasir, G.D.F. Morales, D. GarcíaSoriano, N. Kourtellis, M. Serafini, The power of both choices: Practical load balancing for distributed stream processing engines, in: 2015 IEEE 31st International Conference on Data Engineering, 2015, pp. 137–148.
[40] V.A. Lepakshi, C.S.R. Prashanth, Efficient resource allocation with score for reliable task scheduling in cloud computing systems, in: 2020 2nd International Conference on Innovative Mechanisms for Industry Applications, ICIMIA, 2020, pp. 6–12.
[41] J. Huang, R. Li, J. An, D. Ntalasha, F. Yang, K. Li, Energy-efficient resource utilization for heterogeneous embedded computing systems, IEEE Trans. Comput. 66 (9) (2017) 1518–1531.

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.

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.

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.

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 154 with 125,000+ 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.