Future Generation Computer Systems 184 (2026) 108616

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

A multi-domain cooperative scheduling framework for distributed stream computing systems

Dawei Sun [a],*, Zhongyuan Zhao [a], Yueru Wang [a], Shang Gao [b], Rajkumar Buyya [c]

[a] School of Artificial Intelligence, China University of Geosciences, Beijing, 100083, China
[b] School of Information Technology, Deakin University, Waurn Ponds, Victoria 3216, Australia
[c] Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing, and Information Systems, The University of Melbourne, Parkville, 3010, Victoria, Australia

ARTICLE INFO

Keywords:
Task dependency
Scheduling strategy
Stream computing
Multi-domain environment
Distributed systems

ABSTRACT

In multi-domain environments, task scheduling is one of the most critical elements in stream processing systems, as it directly determines the upper bound on overall system performance. Existing task scheduling approaches in multi-domain stream computing environments primarily focus on achieving load balancing and efficient resource allocation. However, they often overlook the performance degradation that occurs when highly interdependent tasks are assigned to different domains. To address this limitation, we propose Md-Stream, a multi-domain cooperative scheduling framework that improves task allocation efficiency and enables collaborative processing across domains. The framework is discussed from the following perspectives: (1) Impact Analysis: We analyze the negative effects of communication dependencies between tasks on system performance under traditional scheduling methods in multi-domain environments. (2) Model Construction: We design models that incorporate stream topology, task dependency, resource cost, and resource elasticity. (3) Heuristic Graph Partitioning: We propose a heuristic graph partitioning method (KFM) that initially partitions the task topology by extending the Kahn algorithm and subsequently refines it using an advanced move strategy, non-periodic fast checking, and an adaptive Fiduccia–Matthews algorithm to refine the partitioning based on computed gain values. (4) Task Allocation and Resource Elasticity: We develop a multi-domain dependent task allocation method combined with a resource elasticity mechanism to optimize both task distribution and resource utilization. Experimental evaluations demonstrate that Md-Stream significantly outperforms both Storm and Lc-Stream. It reduces average latency by 56.9% and 34.0%, increases throughput by 39.3% and 11.7%, and improves resource utilization by 65.8% and 33.0%, respectively.

1. Introduction

The value of big data is realized through the collection, storage, analysis, and in-depth mining of data within multi-domain environments [1]. Batch processing, as a traditional big data processing method, offers the advantage of handling multiple jobs simultaneously. However, it falls short in meeting the requirements of real-time big data analysis. Consequently, distributed stream processing systems (DSPS) such as Spark Streaming [2], Samza [3], Cao et al. [4], Apache Storm [5], and Apache Flink [6] have emerged. These systems incorporate specialized data processing architectures and drive advances in data processing technologies.

Current research on stream computing primarily focuses on five key challenges: resource scheduling [7], data migration [8], data partitioning [9], load balancing [10], and fault tolerance [11]. Among these, resource scheduling is particularly critical for ensuring efficient and stable system operation, especially in real-time environments.

In distributed cluster systems that span multiple domains, delays caused by frequent I/O operations present a major challenge, particularly when processing large volumes of streaming tasks with inter-task communication dependencies. Prior research often overlooks the impact of task dependencies on system performance. When highly dependent tasks are split and assigned to nodes across different domains, frequent cross-domain communication is triggered, resulting in significant performance degradation. By contrast, scheduling tasks with strong communication dependencies on processing nodes within the same domain can substantially reduce I/O transmission overhead and lower system latency.

* Corresponding author.
E-mail addresses: sundaweicn@cugb.edu.cn (D. Sun), zhaozhongyuan@email.cugb.edu.cn (Z. Zhao), wangyueru@email.cugb.edu.cn (Y. Wang), shang.gao@deakin.edu.au (S. Gao), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.future.2026.108616
Received 14 November 2025; Received in revised form 11 May 2026; Accepted 18 May 2026
Available online 27 May 2026

Scheduling policies embedded in widely used DSPS frameworks, such as Apache Storm [12], are typically general-purpose and domain-agnostic [13]. For instance, Storm distributes tasks across worker processes using round-robin strategies, which achieve superficial numerical load balancing while neglecting data dependencies and communication locality. As a result, interdependent tasks may be allocated to different workers, leading to excessive inter-process communication and reduced system efficiency. This oversight negatively impacts system performance, leaving opportunities for optimization in communication overhead [14], resource allocation [15], and related areas. Furthermore, these strategies lack adaptability to dynamic multi-domain computing environments, where network conditions, task graphs, and resource states vary over time.

Several studies have explored heuristic and approximate scheduling algorithms [16,17]. However, most heuristic-based formulations remain tailored to a single administrative domain and do not account for holistic resource elasticity dynamics and cross-domain cooperation in multi-domain stream deployments.

In multi-domain environments, computing resources are heterogeneous, and network latency and bandwidth are often non-uniform. These factors violate the uniform-resource assumptions of traditional schedulers. As a result, task placement becomes inefficient and communication overhead increases due the need for handling heterogeneity issues explicitly. Therefore, there is a pressing need for scheduling frameworks that go beyond load balancing and address the unique challenges of multi-domain dependency-aware scheduling to achieve more efficient and precise data processing [18].

To this end, we propose Md-Stream, a multi-domain cooperative scheduling framework designed to reduce communication overhead, improve resource utilization, and adapt to runtime variations by integrating task dependency modeling, domain-aware partitioning, and elastic resource management. The framework comprises four key components:

(1) **Refined Partitioning and Dependency Modeling**: Constructs stream topology, quantifies task dependencies, represents the application as a graph, and introduces resource elasticity for real-time monitoring.
(2) **Topology Graph Partitioning**: Partitions the application's directed acyclic graph (DAG) using an enhanced version of Kahn's algorithm that arranges tasks in topological execution order. Communication-intensive tasks are grouped into initial blocks, and the Fiduccia–Matthews algorithm is used to refine partitions based on communication dependency degree and resource constraints.
(3) **Multi-Domain Task Dependency-Aware Allocation**: Maps partitioned task blocks to resource nodes using a lightweight greedy strategy. This strategy ranks candidate nodes based on communication overhead and resource availability, prioritizing resource utilization while minimizing cross-node and cross-domain data transmission.
(4) **Resource Elasticity Mechanism**: Detects bottleneck nodes with high resource usage and applies a heuristic parallelism adjustment algorithm to iteratively reconfigure executors, reduce system latency and improve performance.

This paper builds upon the earlier Td-Stream work [19], which proposed a task dependency-aware scheduling strategy for cross-domain Storm deployments, including dependency modeling, dependency-aware task allocation, and an initial elasticity mechanism for runtime parallelism adjustment.

Building on Td-Stream, Md-Stream extends the framework in three aspects. First, it introduces a heuristic topology-partitioning algorithm, KFM, which constructs an initial partition via a modified Kahn-based topological ordering and refines partitions using an adaptive Fiduccia–Mattheyses move strategy with non-periodic fast checking. Second, it strengthens the resource elasticity design by incorporating an Orl-based capacity model, an online parallelism update algorithm for operator reconfiguration, and an idle-node hibernation mechanism to further improve resource utilization. Third, it expands the experimental evaluation with additional topologies, broader performance metrics, and extended comparisons against Storm, R-storm and Lc-Stream.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 defines the problem and Section 4 focuses on model construction. Section 5 details the Md-Stream scheduling framework. Section 6 presents experimental results and performance analysis. Finally, Section 7 concludes the paper and outlines future work.

2. Related work

This section reviews recent research in two relevant areas: resource scheduling strategies and multi-domain cooperation frameworks for distributed stream computing systems. Table 1 summarizes a comparison between Md-Stream and representative related works across several key dimensions.

2.1. Resource scheduling

Resource scheduling is the process of ensuring that the resource requirements of a converged streaming task are aligned with the resource capacities of distributed hosts. Its objectives involve minimizing communication dependencies between tasks, optimizing overall resource utilization, and maximizing cost-effectiveness under a set of system constraints.

R-Storm [27] is the built-in resource-aware scheduler for Apache Storm. It aims to enhance processing performance by optimizing resource utilization and minimizing internal network latency. The scheduler formulates resource-aware scheduling as a quadratic multivariate three-dimensional knapsack problem.

P-Scheduler [20] introduces an adaptive hierarchical scheduling scheme that estimates the number of nodes required for a topology and co-locates high-traffic components on the same JVM or node to reduce inter-task communication overhead. To improve adaptability, Leila Eskandari et al. later proposed T3-Scheduler [21], a two-stage topological traffic-aware approach that co-locates communication-intensive executors to further minimize communication costs. Building upon these, I-Scheduler [22] adopts a heuristic mechanism that compresses task graphs and leverages optimization software for task assignment. In the absence of such software, it uses a backtracking method to partition the application graph based on node heterogeneity.

Similarly, several works have investigated operator placement from an optimization perspective. Cardellini et al. [28] formulated operator placement as an integer linear programming problem, accounting for heterogeneous computational and network resources. Liu et al. [29] proposed a selectivity-based partitioning method that determines operator parallelism by evaluating the influence domain of upstream operators. SP-Ant [23], on the other hand, applies an ant colony optimization algorithm to allocate executors across heterogeneous compute nodes, balancing local and global optimization objectives. ER-Storm [24] focuses on resource elasticity and scalable decision-making by introducing model-free reinforcement learning to guide runtime operator replication and relocation under dynamic workloads.

Wang et al. proposed CAPSys [26], an adaptive resource controller for Apache Flink that jointly optimizes task placement and auto-scaling. It models compute, state-access (I/O), and network contention, and uses empirically validated heuristic pruning to search the NP-hard placement space. CAPSys can generate a feasible placement within 100 ms even for queries with hundreds of parallel tasks. However, it is designed for datacenter settings where propagation delay is assumed negligible, and thus does not explicitly address cross-domain cooperative scheduling. It also assumes homogeneous tasks within the

Table 1
Comparison between Md-Stream and related work.

Work	Year	Performance		Method			
		Resource utilization efficiency	Real-time responsiveness	Heuristic strategy	Multi-domain cooperation	Task dependency awareness	Operator parallelism
P-Scheduler [20]	2016	✓	×	×	×	×	×
T3-Scheduler [21]	2018	✓	×	×	×	×	✓
I-Scheduler [22]	2021	✓	×	✓	×	×	×
SP-Ant [23]	2022	✓	×	✓	×	×	✓
ER-Storm [24]	2023	✓	×	✓	✓	✓	✓
Chunlin et al. [25]	2022	✓	×	✓	✓	×	✓
CAPSys [26]	2025	✓	✓	✓	×	×	✓
Md-Stream (Our work)		✓	✓	✓	✓	✓	✓

same operator and treats data skew as out of scope, requiring separate mitigation before placement.

Despite these contributions, several limitations remain. R-Storm suffers from high computational overhead in distributed scenarios. P-Scheduler, T3-Scheduler, and I-Scheduler incur substantial computational delays during node prediction and depend heavily on external optimization software such as METIS [30], which adds latency and complexity during runtime scheduling. Moreover, while SP-Ant and ER-Storm incorporate optimization and elasticity considerations, they provide limited modeling of task dependencies and often introduce significant decision-making overhead, hindering their real-time adaptability in large-scale, heterogeneous stream processing environments.

2.2. Multi-domain cooperation

Multi-domain cooperation serves as a fundamental model for scaling distributed systems. By distributing system components, data storage, computing power, and network resources across multiple domains, it overcomes single-node bottlenecks. This model supports parallel task execution, enhances resource utilization, and improves data security and service availability, making it a key area of research.

Chunlin et al. [25] proposed a data placement algorithm based on Lagrangian relaxation to optimize massive data transfers across geographically distributed domains. The algorithm minimizes transmission costs by jointly considering data center capacity and load balancing. It reformulates the bandwidth cost problem as a multi-source shortest path problem and derives the optimal placement scheme using linear programming and Lagrangian relaxation.

Xu et al. [31] developed a bandwidth-aware algorithm that integrates convex optimization with random sampling to minimize bandwidth consumption while meeting latency constraints. This approach effectively balances user requests across data centers but degrades in performance when facing irregular or rapidly changing request patterns.

In parallel, growing environmental concerns have motivated energy-aware multi-domain scheduling studies. Ehsan et al. [32] presented ECAIVMP, a heuristic approach designed to reduce cross-domain energy consumption and carbon emissions while maintaining service quality. The strategy accounts for both IT and non-IT energy usage in virtual machine placement. Similarly, Hosseinalipour et al. [33] introduced a graph-based job assignment method that models power consumption in data centers. Their scheme utilizes low-complexity subgraph extraction for scheduling under both fixed and adaptive pricing models and incorporates online learning algorithms to improve adaptability in dynamic environments.

Despite these advancements, existing solutions share several limitations. Most exhibit limited adaptability to dynamic network topologies and fluctuating workloads, resulting in inefficient resource allocations under varying traffic conditions. Furthermore, they generally lack awareness of task dependencies and provide weak integration between communication-aware scheduling and elasticity mechanisms, restricting their effectiveness in large-scale, heterogeneous, and rapidly changing stream processing systems.

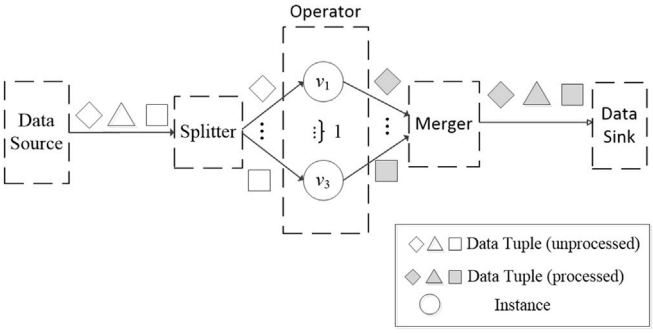


Fig. 1. Operator instantiation.

To overcome these challenges, Md-Stream is proposed as a cooperative multi-domain scheduling framework that integrates task dependency modeling, adaptive task allocation, and resource elasticity. It is designed to minimize communication overhead, reduce latency, and improve resource utilization under dynamic conditions.

3. Problem statement

This section first analyzes the task dependency challenge in multi-domain computing environments, then presents the task allocation strategy and the construction of our optimization model.

3.1. Task dependency

Stream computing systems need to address challenges posed by large volumes of mixed data and complex computations [34]. The instantaneous and non-uniform inflow of data complicates resource prediction and often leads to unbalanced resource allocation. Insufficient resources and stream congestion can slow down processing and negatively impact performance. To optimize performance, both task dependency and resource allocation must be considered in scheduling decisions.

Modern stream systems use operator instantiation to enhance resource scalability and distribute input load. The parallelization strategy splits complex computational tasks into subtasks, which are executed in parallel on different processing units. Each operator instance independently processes fragments of data stream across processors, threads, or compute nodes.

As shown in Fig. 1, data from the source is passed to a Splitter component. The Splitter component divides it into segments and passes them to downstream operator instances. Each instance processes its segment in parallel and returns results to a Merger component, which aggregates outputs before sending them to the data sink.

In Storm systems, multiple instances of spouts (data sources) and bolts (data processors) are executed in parallel. Users configure the number of executors to set the degree of parallelism, and resource elasticity can be achieved by increasing the number of executors when

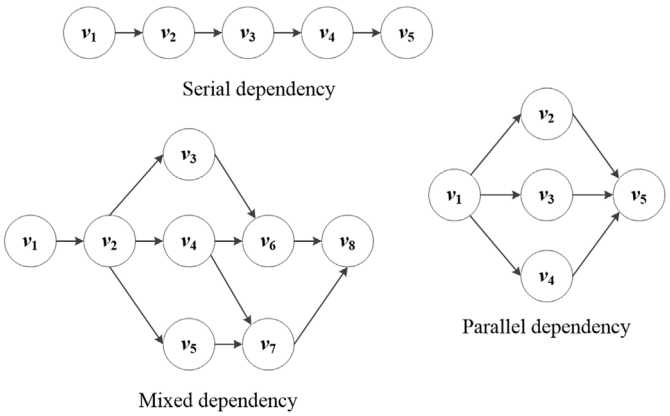


Fig. 2. Types of task dependencies.

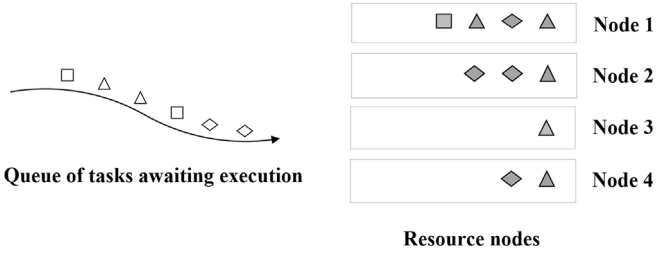


Fig. 3. Task allocation in Storm.

congestion is detected. Although topologies are configured statically, Storm allows re-balancing during runtime to adjust parallelism levels.

Task dependencies define constraints on data flow and execution order between tasks [35]. In stream systems, such dependencies primarily involve communication dependencies—i.e., the need for tasks to exchange data rather than to execute in strict order. Fig. 2 illustrates three typical dependency types: serial, parallel, and mixed.

Serial dependency: one task must complete before another task can begin, enforcing sequential execution. **Parallel dependency**: Independent tasks can execute simultaneously, with no mutual constraints. **Mixed dependency**: A combination of both, involving partially parallel and partially sequential task flows.

Unlike batch systems, stream task execution is loosely coupled. Component instances are relatively independent, and task dependencies primarily impact data transfer (not execution correctness), making communication latency a critical factor.

3.2. Task allocation

In Apache Storm, the default scheduler (DefaultScheduler) attempts to balance load by assigning tasks to resource nodes with the highest remaining capacity. Fig. 3 shows a scenario in which tasks (represented by diamond shapes) with mutual dependencies are assigned to the third resource node based solely on capacity. Since dependent tasks reside on different nodes (e.g., nodes 1, 2, and 4), communication occurs across nodes, significantly increasing latency—especially in multi-domain settings.

While DefaultScheduler achieves basic load balancing by evenly distributing executors, it does not account for inter-task dependencies. As a result, related tasks may be assigned to distant nodes, leading to excessive communication overhead and increased processing latency [36].

To further analyze this issue, a cross-domain cluster environment was emulated using Apache Storm on local virtual machines. Two

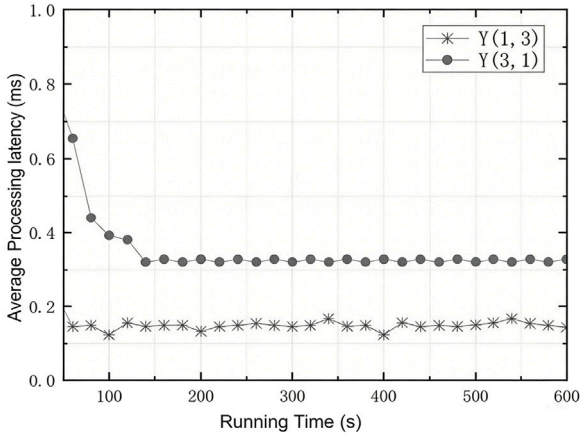


Fig. 4. Effect of task dependencies on latency in WordCount.

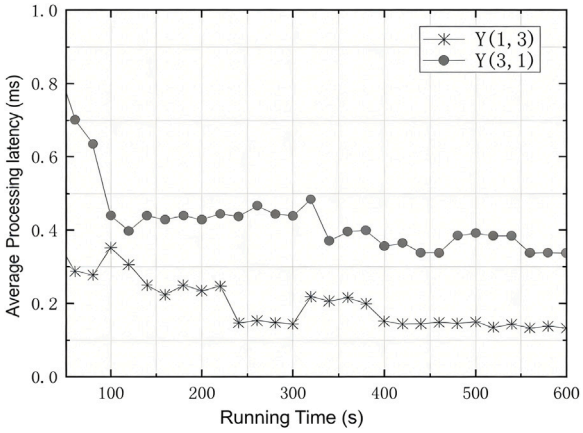


Fig. 5. Effect of task dependencies on latency in TOP-N.

representative stream applications, WordCount and Top-N, were deployed to evaluate average processing latency — measured as the time interval between tuple generation and processing completion — under two configurations: Y(1,3) (one domain with three resource nodes) and Y(3,1) (three domains with one resource node each).

As shown in Figs. 4 and 5, latency in the Y(3,1) configuration is consistently higher than in Y(1,3), due to additional cross-domain communication overhead. Although DefaultScheduler distributes executors evenly, it fails to consider task dependencies, which leads to suboptimal placements in terms of latency. If the actuators are unnecessarily dispersed to resource nodes in different domains, the communication due to task dependencies introduces additional latency. The DefaultScheduler used in the experiments distributes component executors evenly across resource nodes and achieves a basic level of load balancing. However, it may not provide optimal task placement when task communication dependencies are considered. It is necessary to fully consider the dependencies between tasks and choose appropriate configuration and scheduling strategies to reduce communication delays and thus improve system performance.

In real-world multi-domain environments, where resource nodes may be deployed in geographically distributed environments, cross-node communication introduces non-negligible delays. Factors such as physical transmission distance, network infrastructure quality, congestion, and protocol overhead contribute to this latency. These delays not

Table 2
Main notations used in this paper.

Symbol	Description
G	Directed acyclic graph
$V(G)$	Set of graph vertices
$E(G)$	Set of connected edges
V_i	Component vertex i
$V_{i,k}$	kth instance of component vertex i
$e_{v_{i,k},v_{j,m}}$	Edge between instances $v_{i,k}$ and $v_{j,m}$
$Iael_j$	Average execution latency of instance j
$Iemn_j$	Number of messages processed by instance j
dc_{v_i,v_j}	Dependency between components v_i and v_j
$de_{v_{i,k},v_{j,m}}$	Dependency between task instances $v_{i,k}$ and $v_{j,m}$
$ds_{v_{i,k},v_{j,m}}$	Dependency strength between $v_{i,k}$ and $v_{j,m}$
$f_{ds}(g_i, g_j)$	Communication dependency overhead between subgraphs g_i and g_j
$\boldsymbol{R}$	Integrated resource entity comprising fully-interconnected nodes
r_i	Resource node ith within the integrated resource entity
$vr_{v_{j,m},i}$	Resources consumption of instance $v_{j,m}$ at node i
$f_r(g_i)$	Resource overhead of subgraph g_i
Orl_i	Resource level (capacity) of operation node i
$Orl_{\max}$	Maximum capacity threshold of operation node
$Orl_{\min}$	Minimum capacity threshold of operation node

only increase the execution time of individual tasks but also degrade the overall throughput and real-time responsiveness of the system.

4. System model

To address these issues, four interrelated models are established: (1) a stream topology model to describe the structure of stream applications, (2) a task dependency model to reveal inter-task dependencies and communication requirements, (3) a resource cost model to quantify the computational cost of task execution, and (4) a resource elasticity model to evaluate node capacity and trigger adaptive resource adjustments when necessary.

For clarity, the main notations used throughout the paper are summarized in Table 2.

4.1. Stream topology model

A stream application can be represented as a topology with two views: logical view and physical view.

The logical view consists of operators and stream. Operators are self-contained processing units responsible for performing specific operations, such as filtering or labeling. Streams represent unbounded sequences of tuples exchanged between operators. Each operator performs partial computation on incoming tuples and sends the partial processing results to downstream operators.

The physical view is represented by a directed acyclic graph $G = (V(G), E(G))$, where $V(G)$ is a set of vertices consisting of data sources, operators, and receivers, and $E(G)$ is a set of edges along which streams flow between vertices. Each operator can have one or more tasks, which are instance of the operator performing identical computations on different data streams, enabling parallel execution. The number of tasks can dynamically change during runtime based on system states.

When multiple tasks exist within an operator, a stream grouping mechanism defines how data from sender tasks is partitioned and distributed among receiver tasks. As shown in Fig. 6, a simple example topology contains five components (one spout and four bolts). Squares represent components, circles denote task instances, and the number of circles corresponds to component parallelism. In this example, components A, B, C, and D have 1, 3, 2, and 1 task instances, respectively.

Formally, $V(G) = \{v_i | i \in 1, \ldots, n\}$ denotes a finite set of n vertices, where each vertex v_i corresponds to an operation. $E(G) = \left\{ e_{v_{i,k},v_{j,m}} | v_{i,k}, v_{j,m} \in V(G) \right\}$ denotes the set of directed edges between instances, with $e_{v_{i,k},v_{j,m}}$ representing communication between task instances $v_{i,k}$ and $v_{j,m}$.

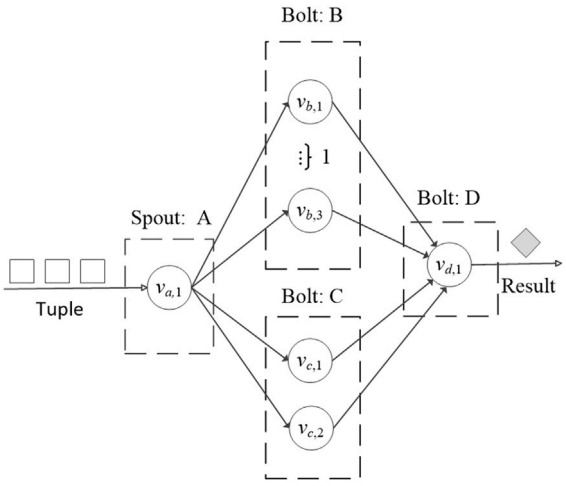


Fig. 6. Stream topology structure.

4.2. Task dependency model

In streaming systems, task dependencies are defined by data transfer relationships rather than execution order. In the topology graph, upstream and downstream operators are connected by edges wherever data transmission occurs, as defined in Eq. (1).

$$dc_{v_i,v_j} = \begin{cases} 1, & \text{if data transfer exists between } v_i \text{ and } v_j, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Similarly, dependencies between task instances are represented as in Eq. (2).

$$de_{v_{i,k},v_{j,m}} = \begin{cases} 1, & \text{if data transfer exists between } v_{i,k} \text{ and } v_{j,m}, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

The dependency strength ds quantifies the communication intensity between tasks, expressed as the product of the dependency indicator and the average transmission rate (atr) between two tasks, as shown in Eq. (3).

$$ds_{v_{i,k},v_{j,m}} = de_{v_{i,k},v_{j,m}} \cdot atr_{v_{i,k},v_{j,m}}, \tag{3}$$

Here, atr represents the expected number of tuples transmitted from the upstream to downstream instance per unit time.

In stream computing, communication can occur (i) within nodes, (ii) across processes on the same node, or (iii) across nodes—with cross-node communication being the most expensive. Therefore, the objective is to maximize the sum of dependency strengths within nodes and minimize those across nodes.

Let the topology be divided into k partition blocks, where g_i denotes the ith block ($g_i \subseteq G, (i = 1, 2, \ldots, k)$), and blocks are non-overlapping. The inter-block dependency strength between subgraphs g_i and g_j is defined in Eq. (4).

$$f_{ds}(g_i, g_j) = \sum_{v_{i,k} \in g_i} \sum_{v_{j,m} \in g_j} ds_{v_{i,k},v_{j,m}}, \tag{4}$$

The objective function for minimizing cross-block dependency strength is expressed in Eq. (5).

$$F = \min \sum_{i=1}^{k} \sum_{j=1}^{k} f_{ds}(g_i, g_j), \tag{5}$$

4.3. Resource cost model

Each component has distinct computational requirements, leading to varying runtime resource consumption. Even within the same component, tasks may differ in cost due to data characteristics or stream grouping patterns.

The resources of a node can be measured in dimensions such as CPU and memory. Define a resource set $\boldsymbol{R}$ consisting of l fully interconnected nodes, where $\boldsymbol{R} = \{r_1, \ldots, r_l\}$. The CPU resource occupancy of node i is denoted as r_i^C, and its memory resource occupancy is denoted as r_i^M. At a certain point in time, there may be multiple task instances running at node i. Let $vr_{v_{j,m},i}$ denote the amount of computational resources consumed by the task $v_{j,m}$ at node i. The value of $vr_{v_{j,m},i}$ can be obtained by proportionally calculating the resource occupancy of the resource nodes collected during the topology runtime, and this relationship is expressed by Eqs. (6) and (7).

$$vr^C_{v_{j,m},i} = r_i^C \cdot \delta_{v_{j,m},i}, \tag{6}$$

$$vr^M_{v_{j,m},i} = r_i^M \cdot \delta_{v_{j,m},i}, \tag{7}$$

where $\delta_{v_{j,m},i}$ represents the ratio between the number of data tuples transferred by this instance and the number of data tuples transferred by all tasks of node i, as represented by Eq. (8).

$$\delta_{v_{j,m},i} = \frac{tp_{v_{j,m}}}{\sum_{v_{k,p} \in r_i} tp_{v_{k,p}}}, \tag{8}$$

The total resource consumption of a task is defined in Eq. (9), where α is the weighted value of CPU resources.

$$vr_{v_{j,m}} = \alpha \cdot vr^C_{v_{j,m},i} + (1-\alpha) \cdot vr^M_{v_{j,m},i}, \tag{9}$$

The resource overhead for a single subgraph is calculated by Eq. (10).

$$f_r(g_i) = \sum_{v_{j,m} \in g_i} vr_{v_{j,m}}, \tag{10}$$

4.4. Resource elasticity model

To evaluate whether a node has reached its capacity limit, we define a capacity parameter for each operational node. This parameter indicates whether a node is becoming a performance bottleneck or is underutilized.

Let instance j of a component have an average execution latency $Iael_j$ and a total number of processed messages $Iemn_j$. The capacity of instance j is given by Eq. (11), where t_s and t_e denote the start and end times of the sampling window.

$$Irl_j = \frac{Iael_j \cdot Iemn_j}{t_e - t_s}, \tag{11}$$

As multiple instances execute in parallel, the component-level capacity Orl_i is determined by the maximum instance capacity, as shown in Eq. (12).

$$Orl_i = \max(Irl_j), j \in \{1, 2, \ldots, M\}, \tag{12}$$

When Orl_i approaches 1, the node is near full capacity, and parallelism should be increased. Conversely, when Orl_i approaches 0, the node is underutilized, and parallelism can be reduced. Two thresholds, Orl_{max} and Orl_{min} are defined to trigger elastic resource adjustments when these bounds are reached.

5. Md-stream: architecture and algorithms

Building on the models presented above, we propose Md-Stream, a multi-domain cooperative scheduling framework for distributed stream computing systems.

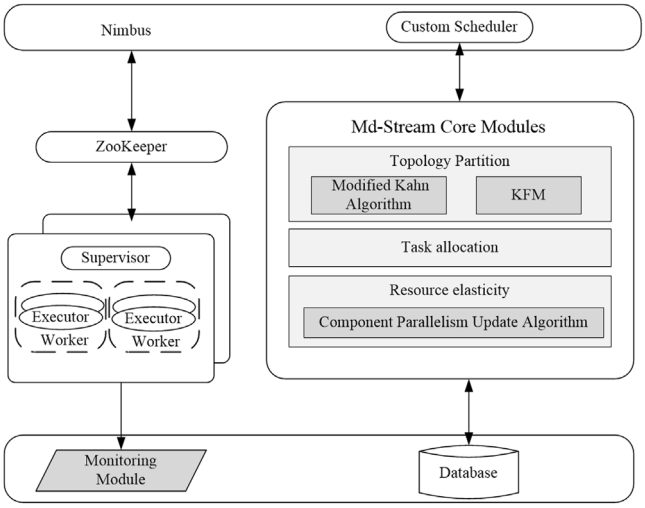


Fig. 7. Md-Stream architecture.

5.1. System architecture

Md-Stream optimizes task scheduling by integrating topology graph partitioning, multi-domain task allocation, and a resource elasticity model. It accounts for task dependency strength, cross-domain communication overhead, and available resources to improve scheduling decisions in stream computing environments.

Fig. 7 illustrates the overall architecture of Md-Stream, which is implemented on top of Apache Storm. The system consists of core Storm components (Nimbus, Zookeeper, and Supervisors) and three additional modules: a Monitoring module, a Database, and the Core Scheduling module introduced by Md-Stream.

Among them, Nimbus is responsible for coordinating and managing the entire Storm system. It assigns tasks to worker nodes and manages their lifecycle.

Zookeeper provides distributed coordination services. It maintains global configuration parameters, ensures consistency across components, and simplifies resource discovery through unified naming services.

Supervisor runs on each compute node and receives task instructions from Nimbus. It starts or stops worker processes accordingly, enabling dynamic adaptation to task changes and ensuring stable execution.

To enable real-time optimization of task placement and resource allocation, Md-Stream introduces three key modules:

Monitoring module: Continuously collects system metrics such as CPU usage and data stream characteristics, enabling real-time assessment of system load and resource demands.

Database module: Stores historical and current task information along with monitoring data. It provides a persistent backend for data-driven scheduling decisions.

Core scheduling module: The central component of Md-Stream, this module replaces Storm's default scheduler by implementing the `IScheduler` interface. It integrates real-time monitoring and historical data to perform dependency-aware and resource-optimized task scheduling.

Together, these enhancements enable Storm to support real-time monitoring, data-driven task allocation, and adaptive scheduling, significantly improving responsiveness, scalability, and overall system performance in complex and dynamic stream processing scenarios.

5.2. Topology partitioning

Topological graph partitioning seeks to divide a task topology graph by considering task dependencies and resource consumption. However, this problem is theoretically NP-hard, making it infeasible to search exhaustively for an optimal partitioning scheme. To address this, we adopt a heuristic graph partitioning algorithm, KFM. This algorithm first generates an initial partition using a modified Kahn's algorithm for topological sorting. It then combines an adapted Fiduccia–Mattheyses algorithm for gain computation with a non-periodic fast-check vertex movement strategy to refine the partitioning.

The stream topology is modeled as a directed acyclic graph (DAG) $G = (V(G), E(G))$. Let $g_1, \ldots, g_k$ denote the k partition blocks such that $g_1 \cup \cdots \cup g_k = G$, and $g_i \cap g_j = \emptyset$ for all $i \neq j$.. A threshold $L_{\max}$ is defined for the aggregate weight of vertices in each partition. A block g_i is considered overloaded if its total vertex weight exceeds this threshold, i.e., $f_r(g_i) > L_{\max}$; otherwise, it is balanced if $f_r(g_i) \leq L_{\max}$. A vertex v is termed a boundary node if it has neighbors in a different partition.

The partitioned graph can be abstracted into a quotient graph, in which each node represents a partition block and edges denote inter-block dependencies. In a weighted quotient graph, node weights reflect the total weight of the block, and edge weights correspond to the cumulative weights of inter-block edges. This abstraction helps capture the relative importance of each partition and the strength of connections between them.

The KFM partitioning approach must satisfy two constraints:

(1) **Balance constraint:** $\forall i \in \{1, \ldots, k\}$, the total weight must not exceed a relaxed threshold: $f_r(g_i) \leq L_{\max} = (1+\chi)\left\lceil \frac{f_r(G)}{k} \right\rceil$, where $\chi \in (0, 1)$ is a user-defined imbalance factor.
(2) **Acyclic constraint:** The input graph must be acyclic, a condition inherently satisfied by stream processing topologies.

The objective function of KFM aims to minimize the total weight of edges cut across partitions: $\min \sum_{i=1}^{k} \sum_{j=1}^{k} f_{ds}(g_i, g_j)$, where $f_{ds}(g_i, g_j)$ denotes the total weight of edges connecting blocks g_i and g_j.

In summary, the goal is to find a partitioning $\prod = g_1, \ldots, g_k$ such that: the balance constraint is satisfied, the graph remains acyclic, and the total weight of inter-partition edges is minimized.

The balance constraint ensures that the aggregated resource demand of each partition does not exceed the domain's processing capacity, which is critical for avoiding resource contention. Minimizing the edge cuts reduces inter-domain task dependencies, which in turn decreases communication overhead and enhances the efficiency of stream application processing.

Algorithm 1: Modified Kahn Algorithm

```
Input: Stream application G = (V, E)
Output: Topologically sorted list T
Initialize list S containing all vertices with in-degree of 0;
Initialize empty list T to store the topological ordering;
while S ≠ ∅ do
    Select u ∈ S;
    S.remove(u) /* Remove vertex u from S */;
    T.append(u) /* Add vertex u to sorted list T */;
    for each neighbor v of u do
        indegree[v] = indegree[v] − 1 ;
        if indegree[v] == 0 then
            S.append(v)
        end
    end
end
return T;
```

As shown in Algorithm 1, the algorithm begins by initializing two lists: S, which stores vertices with zero in-degree, and T, which will hold the final topological order. In each iteration of the loop, a vertex u is selected from S, appended to T, and its outgoing edges are removed. If this results in any of u's neighboring vertices reaching zero in-degree, they are added to S. The process repeats until S becomes empty. The final list T represents a valid topological sort of the input graph.

The modified Kahn algorithm achieves a time complexity of $O(|V| + |E|)$, making it efficient for sorting large-scale graphs. During initialization, all vertices are scanned once to compute in-degrees and identify those with zero in-degree, yielding $O(|V|)$ time. Each edge is processed exactly once across the loop, leading to $O(|E|)$ time in total.

Using Algorithm 1, we obtain the topological list T that includes all vertices in the graph. Based on this list, the graph is divided into a sequence of blocks composed of consecutive nodes, forming the initial partitioning scheme. The design of the improved Kahn algorithm ensures that vertices in block g_j do not have outgoing edges to any block g_i for $i < j$, thus maintaining acyclicity across partitions.

Due to the randomized nature of the initial partitioning, the algorithm is executed τ times, each with a different random seed. Among these candidates, the partition with the best performance (e.g., lowest inter-block communication cost) is selected as the final initial solution.

Consider the example topology graph shown in Fig. 8, where all vertex weights are set to 1. The weights on the edges denote communication costs between connected tasks. Suppose we aim to divide this graph into $k = 3$ partitions, with an imbalance tolerance $\chi = \frac{2}{3}$ and a maximum partition capacity $L_{\max} = 5$.

Fig. 9 illustrates the resulting initial partition layout obtained through the modified Kahn-based method.

Using the modified Kahn algorithm, we obtain a valid topological ordering of the example graph: $T = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$. We aim to divide the graph into $k = 3$ partitions, ensuring that the total vertex weights across the partitions are balanced and meet the equilibrium constraint. One feasible initial partitioning is: $g_1 = \{v_1, v_2, v_3\}, g_2 = \{v_4, v_5, v_6\}, g_3 = \{v_7, v_8, v_9\}$, This initial division is illustrated in Fig. 9.

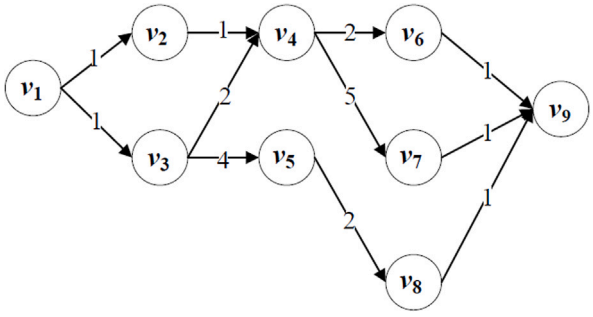


Fig. 8. Example stream topology.

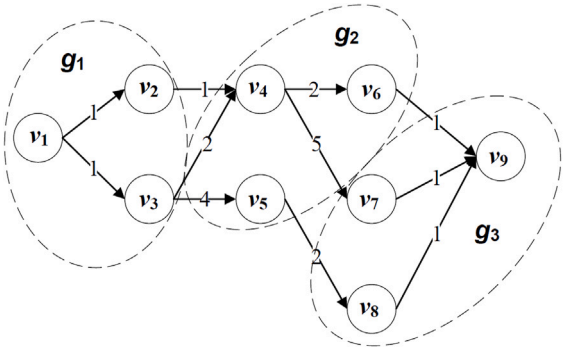


Fig. 9. Initial partitioning result of the stream topology.

5.3. Heuristic graph partitioning

To facilitate reproducibility, we provide a standalone implementation of the KFM algorithm and make the source code publicly available.[1]

The KFM algorithm reduces the number of cut edges by migrating nodes between blocks given an initial solution. This reduction in the number of cut edges after the migration operation is called the gain of that migration. To evaluate the migration gain of node v, we define two functions for assessing the effects of different vertex migration strategies on the partitioning quality of KFM, as shown in Eqs. (13) and (14).

$$C_{in}(v, g_i) = \sum_{u \in g_i} ds_{u,v}, \tag{13}$$

$$C_{out}(v, g_i) = \sum_{u \in g_i} ds_{v,u}, \tag{14}$$

C_{in} denotes the sum of dependency weights from all nodes in partition g_i to node v; C_{out} is sum of dependency weights from node v to all nodes in partition g_i.

Here, $ds_{u,v}$ denotes the dependency strength from node u to v, as previously defined. If v currently resides in g_i, these edges are internal. Moving v to another block would convert them into external edges, increasing the inter-block communication cost. Conversely, if v belongs to g_j and is moved to g_i, external edges become internal, reducing the cut-edge cost.

For the given partitioning g_1, g_2, and g_3, suppose we consider a refinement step between g_1 and g_2. Candidate vertices for migration are selected using the following criteria: (1) From g_1: vertices with no outgoing edges pointing to any earlier partition than g_2. (2) From g_2: vertices with no incoming edges from any later partition than g_1.

Then the candidate vertices in g_1 are v_2 and v_3, and the candidate vertices in g_2 are v_4 and v_5. The gain values of these candidate vertices are computed separately. The gain for a candidate vertex v_i is calculated as the edge weights between the blocks v_i is moving to minus the edge weights inside the block v_i is located. It is expressed by Eq. (15), which is the sum of the edge weights of this vertex inside the partition block minus the sum of the edge weights of this vertex connected between another partition block.

$$\begin{cases} func(v, g_A \to g_B) = C_{out}(v, g_B) - C_{in}(v, g_A) \\ func(v, g_B \to g_A) = C_{in}(v, g_A) - C_{out}(v, g_B) \end{cases}, \tag{15}$$

This gain function effectively evaluates the trade-off between decreasing cut edges (by converting external to internal edges) and increasing them (by converting internal to external edges). The KFM refinement process uses this gain value to determine which node movements improve the overall partition quality while still satisfying resource balance constraints.

The candidate vertices are sorted into a priority queue based on their gain values. If two candidates share the same gain value, a random comparator determines their order in the queue. Based on the topology graph and the gain calculation defined in Eq. (15), the initial priority queue becomes:

For each candidate vertex v_i, state modification rules are applied as follows:

(1) Forward migration ($g_A \xrightarrow{v_i} g_B$): Vertex v_i is moved from partition g_A to g_B. After the move: v_i and all its neighbors in g_B (with incoming edges from v_i) are locked (disabled). Vertices in g_A that either point to v_i or no longer have successors in earlier partitions are enabled.

[1] Source code available at: https://github.com/695200453/kfm.

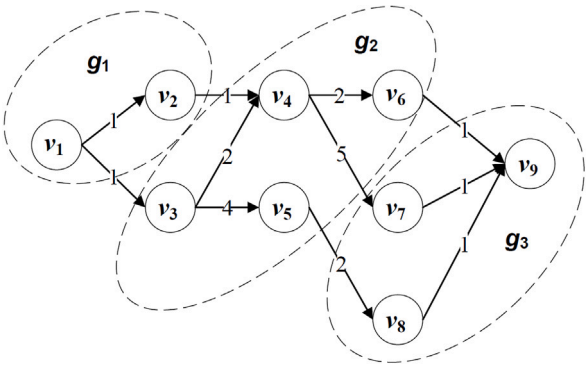


Fig. 10. Division after moving v_3.

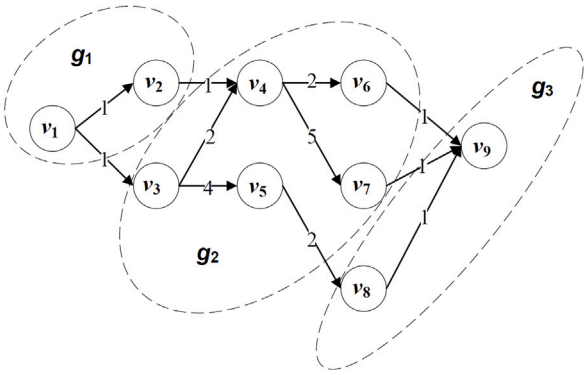


Fig. 11. Partitioning results for example topology.

(2) Reverse migration ($g_B \xrightarrow{v_i} g_A$): Vertex v_i is moved from g_B to g_A. After the move: v_i and all its neighbors in g_A (with outgoing edges to v_i) are locked. Vertices in g_B that are pointed to by v_i or that no longer have predecessors in later partitions are enabled.

Next, the highest-priority enabled vertex in the queue is selected. If a vertex is already disabled or its move violates the balance constraint, the algorithm skips it. Otherwise, it is moved to the target block, and the enable/disable states of related vertices are updated. Even vertices with negative gain values are processed (moved and disabled) to ensure completeness. Newly enabled vertices have their gain recalculated and are added back into the priority queue. Previously computed gains for other candidates remain valid unless their connectivity changes.

Fig. 10 shows the updated partitioning after moving v_3 from g_1 to g_2, a change that satisfies the balance constraint and yields a positive gain. According to the update rules: v_3, v_4, and v_5 are disabled. v_1, a newly enabled vertex, is evaluated and found to have a gain of 0. The new priority queue becomes: v_3 (disabled), v_5 (disabled), v_4 (disabled), v_2 (enabled), v_1 (enabled).

The algorithm proceeds with v_2. Since v_2 has a gain of 0 and its movement does not benefit the balance constraint, it is disabled. Similarly, v_1 is also disabled for the same reason. Once all candidates are disabled, the current internal pass between g_1 and g_2 concludes.

Next, the algorithm randomly selects another pair of partitions for a new internal pass, provided that at least one of them is active (i.e., contains an enabled vertex). If no active partitions remain, the process terminates. To avoid local minima, the algorithm performs multiple iterations (with all vertices re-enabled) until a predefined

iteration limit is reached. The final result for the example is shown in Fig. 11.

The total cut edge weight is reduced from 15 to 6, demonstrating that the KFM algorithm significantly decreases cross-block communication costs and inter-task dependencies.

When partitioning a stream topology graph, both vertex weight balancing and cluster resource constraints must be considered. Each task may require different resources, and available node resources change dynamically during runtime. Since the number of workers per node is limited by the physical number of available slots, the number of partition blocks k is calculated using Eq. (16):

$$k = \frac{\frac{\mu \cdot r_{max}}{Num_{slot}}}{\frac{f_r(G)}{Num_{task}}} = \frac{\mu \cdot r_{max} \cdot Num_{task}}{Num_{slot} \cdot f_r(G)}, \tag{16}$$

Eq. (16) is derived under a full-utilization assumption and represents the ratio between allocatable resources and expected per-task demand. In practice, workload intensity may fluctuate, and operating at theoretical capacity can increase the risk of transient overload. To provide headroom, we introduce a safety-margin factor $\mu \in (0, 1)$. In our experiments, we set $\mu = 0.7$ (reserving approximately 30% slack capacity) as a balanced choice between resource efficiency and runtime stability, and keep this value fixed across all experiments.

The pseudo-code of the KFM algorithm for stream topology graph partitioning is shown in Algorithm 2.

Definition (*AcyclicCheck*)**.** Let the current partition be $\{g_1, \ldots, g_k\}$ with total order $1 < 2 < \cdots < k$. For a candidate vertex v considered for movement between blocks g_A and g_B $(A < B)$:

- If $v \in g_A$ and is considered to move to g_B, the move is allowed only if

  $\nexists (v \to u) \in E$ such that $block(u) < B$.

- If $v \in g_B$ and is considered to move to g_A, the move is allowed only if

  $\nexists (u \to v) \in E$ such that $block(u) > A$.

Here $block(u)$ denotes the index of the partition currently containing vertex u.

Proposition (*Acyclicity Preservation*)**.** *The quotient graph induced by the partition remains a directed acyclic graph after each refinement move.*

Proof Sketch. The initial partition is constructed from a Kahn-based topological ordering; therefore, all inter-block edges follow the block order $(i < j)$, and the quotient graph is acyclic.

During refinement between blocks g_A and g_B $(A < B)$, the AcyclicCheck explicitly prevents the creation of any backward inter-block edge (i.e., from a higher-index block to a lower-index block). Because any directed cycle in a totally ordered graph must contain at least one backward edge, such cycles cannot arise. Hence, the quotient graph remains acyclic after every admissible move.

The KFM algorithm takes the stream application graph G, vertex weights $vr_{v_{i,k}}$, edge weights $ds_{v_{i,k},v_{j,m}}$, per-node resource capacity $r_{\max}$, scaling factor μ, number of workers Num_{slot}, and number of tasks Num_{task} as input. The output is the optimal partitioning scheme $P_{optimal}$ for the stream application. In Step 1, the number of partitions k is computed using Eq. (16). The algorithm then iterates τ times, generating a candidate partition P_t in each iteration. After completing all iterations, it compares the candidate partitions and returns the best-performing one as $P_{optimal}$. To prevent deadlocks in dependency-aware scheduling, the refinement stage is guarded to preserve acyclicity of the partition quotient graph [37].

Within each outer iteration, the algorithm first obtains the topologically sorted list T (using Algorithm 1) and partitions T into k initial blocks $g_1, \ldots, g_k$ under the balance constraint, then enables all vertices for refinement. Because the initialization follows the order of T, inter-block dependencies are consistent with the block order, yielding an acyclic quotient graph at initialization.

During refinement, the algorithm repeatedly selects two blocks g_A and g_B such that at least one contains enabled vertices. It enforces a consistent direction by ensuring $A < B$ (swapping the selected blocks if necessary). It then identifies candidate vertices between the two blocks, computes their gain values, and filters candidates using AcyclicCheck$(\cdot, g_A, g_B)$ before inserting them into the priority queue Q. The queue is processed in descending gain order using PopMax to extract the current best vertex. If the selected vertex is disabled, it is skipped. If the gain condition holds (positive gain, or zero gain but improving balance), the vertex is moved only if AcyclicCheck$(v_{j,m}, g_A, g_B)$ passes and the balance constraint remains satisfied. After a move, states of affected vertices are updated, and newly enabled candidates are inserted into Q only after passing AcyclicCheck. Each processed vertex is then disabled to ensure termination.

Algorithm 2: Improved Heuristic Graph Partitioning Algorithm (KFM)

Input: Stream application $G = \{V, E\}$, vertex weights $vr_{v_{i,k}}$, edge weights $ds_{v_{i,k},v_{j,m}}$, per-node resource capacity $r_{\max}$, scaling factor μ, number of workers Num_{slot}, number of tasks Num_{task}
Output: Optimal subgraph partition result $P_{optimal}$

```
Compute number of partitions k using Eq. (16);
for t = 0 to τ do
    Obtain topological list T // Using Algorithm 1
    Partition T into g_1, ..., g_k satisfying balance constraints;
    Enable all vertices;
    while there exists at least one enabled vertex in any g_1, ..., g_k do
        Select g_A and g_B // At least one must contain enabled vertices
        if A > B then
            swap (g_A, A) and (g_B, B) // enforce A < B
        end
        Initialize empty priority queue Q;
        Identify candidate vertices satisfying AcyclicCheck(·, g_A, g_B);
        Compute their gain and insert candidate vertices into Q based on gain values;
        while Q ≠ ∅ do
            v_{j,m} ← PopMax(Q);
            if v_{j,m} is disabled then continue;
            if v_{j,m}.gain > 0 or (v_{j,m}.gain = 0 and its move improves balance) then
                if AcyclicCheck(v_{j,m}, g_A, g_B) and balance constraint is satisfied then
                    Move v_{j,m} between partitions;
                    UpdateStates();
                    // Update states of affected vertices
                    if new candidate vertices are enabled then
                        Identify new candidates satisfying AcyclicCheck;
                        Compute gain and insert new candidates into Q;
                    end
                end
            end
            Disable v_{j,m};
        end
    end
    P_t ← {g_1, ..., g_k};
    Store P_t into set P;
end
P_optimal ← Compare(P);
return P_optimal;
```

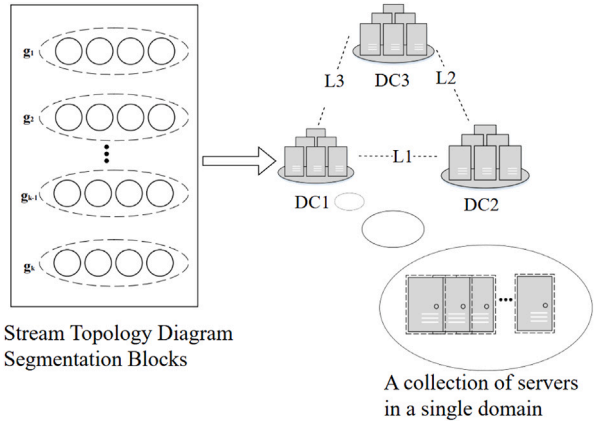


Fig. 12. Multi-domain task allocation.

Unlike classical partitioners such as METIS, which mainly target general or undirected graph cut minimization, KFM is designed for constrained partitioning on stream DAGs, where dependency order and quotient-graph acyclicity must be preserved. the time complexity of the KFM algorithm is $O(\tau(|V|\log|V|+|E|))$, where τ is the number of outer iterations, $|V|$ is the number of vertices, and $|E|$ is the number of edges in the graph. Each iteration involves generating the topological sort, partitioning, state updates, vertex migration decisions, and queue operations.

5.4. Multi-domain task allocation

In multi-domain distributed environments, dependent tasks are allocated to processing nodes based on both resource utilization and domain locality. The structure of the multi-domain task allocation framework is illustrated in Fig. 12.

The allocation process follows the principle of resource-prioritized assignment while considering domain boundaries. Priority is given to nodes within the domain of the master node (e.g., $DC1$). When a node's task load exceeds a predefined threshold, tasks are allocated to other nodes within the same domain to minimize cross-domain communication. Only when local resources are exhausted does the scheduler extend allocation to other domains.

Cross-domain communication adheres to the Round-Trip Delay (RTD) model [38], which factors in latency, cost, server load, and path reliability. To meet stream processing latency requirements, allocation prioritizes domains with the lowest propagation delays. Nodes are ranked based on communication latency, and tasks are assigned sequentially.

The task dependency between two partitions is denoted in Eq. (17):

$$e_{g_i,g_j} = f_{ds}(g_i, g_j), \tag{17}$$

Algorithm 3 describes the task dependency-aware allocation strategy in a multi-domain environment.

Algorithm 3 takes the optimal partitioning result as input and produces a task scheduling strategy based on task dependencies. Given P_{optimal}, all the cut edges between partition blocks, denoted as $e_{g_1,g_2}, e_{g_1,g_3}, \ldots, e_{g_{n-1},g_n}$, are extracted from the partition set $G = g_1, \ldots, g_k$ and sorted in descending order of edge weight (Step 1). In parallel, the resource utilization of each computing node is obtained as $R_c = rc_1, \ldots, rc_n$ (Step 2).

To support locality-aware scheduling, domains are first ordered by inter-region propagation latency. Within each domain, nodes are sorted in ascending order of utilization, resulting in the final scheduling list $R = r_1, \ldots, r_n$ (Step 3). This ordering ensures that tasks are preferentially assigned to low-latency, low-utilization nodes, thereby reducing cross-domain transmission while maximizing resource efficiency. If no partition blocks or resource nodes are available, the algorithm terminates early by returning null (Steps 5–7).

A counter is then initialized to index the resource nodes (Step 8). As long as there are remaining partition blocks, the algorithm iteratively selects the two blocks linked by the highest-weight cut edge and assigns them to the current node r_j. These blocks and their corresponding edge are removed from the respective sets, and the utilization of r_j is updated accordingly.

Subsequently, the algorithm continues assigning additional blocks to r_j by selecting those connected via the next highest-weight edges, repeating the process until either the node's capacity threshold is exceeded or no unassigned blocks remain. Each resulting allocation is recorded as a mapping map_j, and once all partition blocks are assigned, a complete mapping from partition blocks to resource nodes denoted as Map is obtained.

Algorithm 3: Task Dependency-Aware Allocation in Multi-Domain Environments

Input: Optimal partitioning result P_{optimal}
Output: Task-to-node scheduling plan

1 Extract partition blocks $g_1, \ldots, g_k$ from P_{optimal};
2 Compute dependency weights between partitions and sort in descending order: $E_g = \{e_{g_i,g_j}\}$;
3 Obtain resource utilization of each node: $R_c = \{rc_1, \ldots, rc_n\}$;
4 Sort all nodes based on domain latency and utilization: $R = \{r_1, \ldots, r_n\}$;
5 **if** $G = \emptyset$ *or* $R = \emptyset$ **then**
6 **return** null;
7 **end**
8 Initialize node counter $j = 1$;
9 **while** $G \neq \emptyset$ **do**
10 Initialize mapping map_j;
11 **for** ***each*** *node* $r_j \in R$ **do**
12 Assign the two blocks linked by the largest e_{g_A,g_B} to node r_j;
13 Remove g_A, g_B from G and e_{g_A,g_B} from E_g;
14 Update rc_j;
15 **while** $rc_j \leq \mu$ *and* $G \neq \emptyset$ **do**
16 Assign next most dependent block g_c to r_j;
17 Remove g_c from G and corresponding e from E_g;
18 **end**
19 Record current mapping map_j;
20 Append map_j to Map;
21 $j = j + 1$;
22 **end**
23 **end**
24 **return** Final task scheduling map Map;

5.5. Resource elasticity mechanism

While initial topology partitioning provides a balanced structure, it may not fully address dynamic workload conditions or avoid resource underutilization. In real deployments, component bottlenecks and fluctuating throughput may result in thread congestion or idle resources. The resource elasticity mechanism addresses these inefficiencies by dynamically adjusting the parallelism of operators.

This mechanism is triggered after the system reaches a steady state, avoiding frequent thread reallocation during early-stage configuration. When system latency exceeds the defined threshold, a heuristic algorithm adjusts operator parallelism to maintain a smooth data flow.

The parameter parallelism degree PD_i of the operation node i can be adjusted according to Eq. (18). Here, RT_i represents the rate of receiving data tuples by operation node i, and RAP_i represents the average processing rate of operation node i. To avoid sudden abnormal

changes during adjustment of the parallelism degree of the operation node, the parameter ω is set.

$$PD_i = PD_i + \left[\omega \cdot \frac{RT_i - RAP_i}{RT_i}\right], \tag{18}$$

Here, RT_i denotes the input tuple rate, RAP_i is the average processing rate, and ω is a dampening factor to prevent abrupt changes.

The update process is described in Algorithm 4.

Algorithm 4: Component Parallelism Update Algorithm

Input: Operator capacity thresholds $Orl_{\min}$, $Orl_{\max}$
Output: Updated parallelism levels for all components

$time \leftarrow$ uptime();
if $time > Time_{stability}$ **then**
  $SL \leftarrow$ Average system latency;
  if $SL > SL_{\max}$ **then**
    for $i = 1$ *to* N *operators* **do**
      for $j = 1$ *to* M *executors* **do**
        $Iael_j \leftarrow$ Avg. execution latency of executor j;
        $Iemn_j \leftarrow$ Number of messages executed by j;
        $Irl_j \leftarrow \frac{Iael_j \cdot Iemn_j}{t_e - t_s}$;
      end
      $Orl_i \leftarrow \max(Irl_1, \ldots, Irl_M)$;
      if $Orl_i > Orl_{\max}$ *or* $Orl_i < Orl_{\min}$ **then**
        $RT_i \leftarrow$ Input rate of component i;
        $PD_i \leftarrow$ Current parallelism;
        $RAP_i \leftarrow$ Avg. processing rate of component i;
        Recalculate PD_i using (18);
      end
    end
  end
end
return Updated component parallelism;

In Algorithm 4, a threshold value for the capacity of the operation nodes is entered and the new degree of parallelism is output. After the system reaches a stable state, the system latency is measured. If the latency exceeds the threshold, the topology is traversed to identify operators whose parallelism should be adjusted. Calculate the capacity of each actuator by traversing all the actuators of the current operation node, obtaining the average execution latency of each actuator and the number of messages executed, and take the maximum value of all actuator capacities as the current operation node capacity parameter. Determine whether this value meets the determination of the need to adjust the parallelism resources, if it meets the conditions, the number of thread resources of the operation node is adjusted by Eq. (18). At the end of all traversals, the topology optimization scheme after resource elasticity adjustment is obtained.

Although $Orl_{\max}$ and $Orl_{\min}$ are fixed during controlled experiments to ensure fair comparison, the framework does not require static thresholds. In practice, these parameters can be adjusted dynamically based on recent monitoring statistics (e.g., moving averages or percentile-based load estimates) and combined with hysteresis or cool down intervals to prevent oscillatory scaling. This extension would improve responsiveness under highly volatile stream workloads while preserving stability.

The time complexity is $O(N \times M)$, where N is the number of operators and M is the number of executors. This complexity reflects the nested traversal of all operator–executor pairs and parallelism updates. Other operations have constant or negligible cost.

Task dependency-aware scheduling often results in fewer active computing nodes. Once task allocation is completed, unused cluster nodes can be hibernated or shut down. In multi-domain systems, where energy prices and power availability may vary geographically, this practice can reduce energy consumption and operational costs.

While predictive elasticity mechanisms can be effective under stable and regular workload patterns, they often rely on accurate forecasting models and incur additional training and maintenance overhead. In contrast, Md-Stream adopts a measurement-driven elasticity strategy that reacts directly to observed operator load and system latency, enabling fast adaptation to bursty and non-stationary stream workloads. This lightweight design improves robustness and reduces control complexity in latency-sensitive multi-domain environments.

Table 3
Software configuration summary.

Software	Version
OS	CentOS 7
Apache Storm	Apache-Storm-2.4.0
JDK	JDK-8u131-linux-x64
Apache Zookeeper	Apache-Zookeeper-3.5.10
Python	Python-3.10.6
MySQL	MySQL-5.7

6. Performance evaluation

6.1. Experimental setup

Md-Stream is implemented on Apache Storm 2.4.0 and deployed on a CentOS 7-based cluster with 15 nodes, including 1 Nimbus node, 2 ZooKeeper nodes for coordination, and 12 Supervisor nodes for stream processing. To emulate a multi-domain environment, the 12 Supervisor nodes are logically partitioned into four domains (three nodes per domain). Cross-domain communication is defined as traffic between Supervisor nodes belonging to different domains. Additional network latency is selectively injected into inter-domain traffic using Linux Traffic Control (tc), while intra-domain communication remains unchanged. Each node is equipped with a 2-core 2.67 GHz CPU, 2 GB of memory, 40 GB of disk storage, and a 100 Mbps Ethernet interface.

During the experiments, real-time monitoring of the system runtime status is performed via the native Storm UI component of Apache Storm, and an automated high-frequency acquisition and recording pipeline for core performance metrics is implemented with Python. All experiments are conducted with each experimental configuration executed independently three times, and every individual run is performed continuously for 60 min after topology submission. We extract and analyze two sets of experimental data: the runtime data of the first 700 s before the system enters the steady state, and the full-cycle data of the complete 60-minute run.

The detailed experimental environment configuration is shown in Table 3.

To evaluate Md-Stream's scheduling policy, we use two commonly studied Storm topologies: WordCount and Top-N.

The WordCount topology reads text from a stream, counts the number of occurrences of each word, and outputs the results. It consists of three main components: the Sentence Spout, the Split Sentence Bolt, and the Word Count Bolt. The Sentence Spout reads textual data from the stream. The Split Sentence Bolt identifies word boundaries and splits the text into individual words using regular expressions or string processing techniques. The Word Count Bolt counts word occurrences in real time.

The logical structure of WordCount is shown in Fig. 13.

The Top-N topology processes large-scale data to identify the N most frequent elements. It includes four main components: a data source, a counting component, and ranking and merging components. The data source reads input data; the counting component aggregates frequency statistics; and the ranking/merging component sorts the data to extract the top-N elements.

The logical structure of Top-N is shown in Fig. 14.

Both topologies are submitted to the cluster to evaluate Md-Stream's internal system performance.

Although WordCount and Top-N appear relatively simple at the logical topology level, their physical execution graphs under realistic

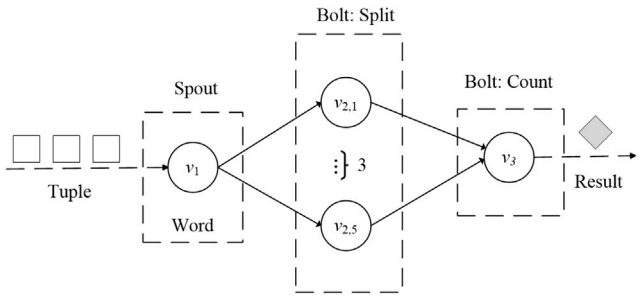


Fig. 13. WordCount topology.

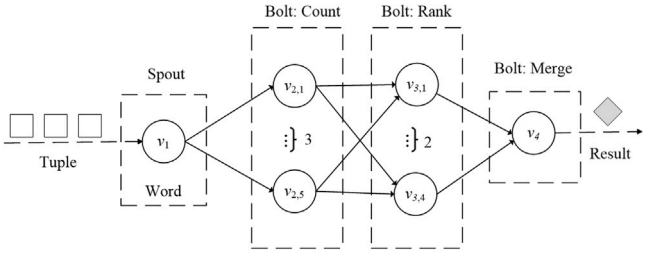


Fig. 14. Top-N topology.

parallel deployment are significantly more complex due to parallel executors, inter-operator communication, and cross-domain task interactions. As a result, the runtime execution DAG contains dense dependency relationships and communication-intensive execution patterns, making these workloads suitable for evaluating dependency-aware multi-domain scheduling.

In particular, prior workload characterization and benchmark taxonomy studies [7,39] have shown that Top-N belongs to the category of stateful, communication-intensive parallel aggregation workloads. This type of workload is characterized by window-based aggregation, dense inter-executor communication, frequent state access, and strong sensitivity to scheduling locality and task placement. Similar execution-level characteristics have also been identified in industrial streaming benchmarks such as the Yahoo Streaming Benchmark (YSB) [40,41]. Although the application semantics differ, Top-N and YSB exhibit fundamentally similar execution-level dependency structures relevant to scheduling optimization.

6.2. Parameter tuning

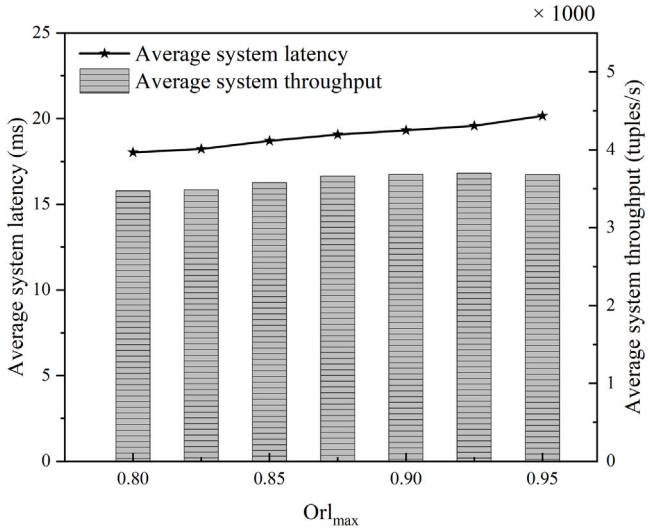
The operator node capacity threshold parameters $Orl_{\max}$ and $Orl_{\min}$ determine whether the parallelism of a node should be adjusted. Their values significantly affect the algorithm's performance and must be tuned experimentally.

To determine appropriate values for $Orl_{\max}$ and $Orl_{\min}$, we adopt a composite performance metric jointly evaluating system latency and throughput, and define a comprehensive evaluation function f to reflect the overall performance of the Storm system, as represented by Eq. (19).

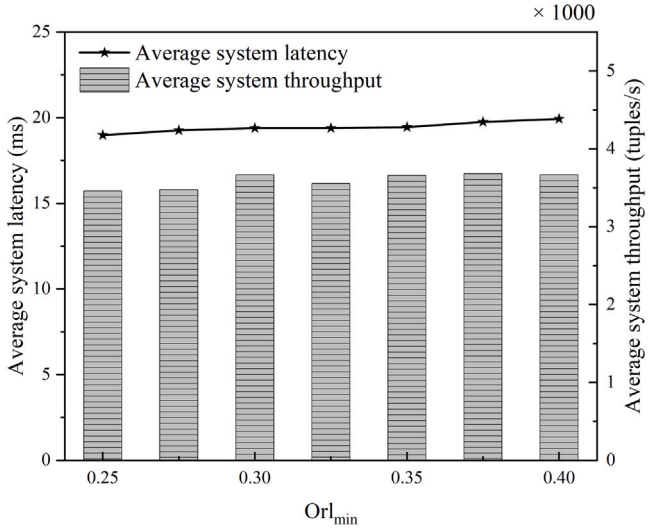
$$f = (\varphi_1 + \varphi_2)\sqrt{\frac{(\partial \cdot T)^{\varphi_2}}{L^{\varphi_1}}}, \tag{19}$$

where L denotes average system latency, T denotes average throughput, φ_1 and φ_2 are weighting factors, and ∂ is a scaling constant. In our experiments, we set $\varphi_1 = \varphi_2 = 1$, giving equal importance to latency and throughput. Under this setting, the evaluation function simplifies to Eq. (20).

$$f = 2\sqrt{\frac{(\partial \cdot T)}{L}}, \tag{20}$$



(a) Average system latency and throughput for different values of $Orl_{\max}$.



(b) Average system latency and throughput for different values of $Orl_{\min}$.

Fig. 15. Performance metrics under different threshold parameters.

Thus, maximizing f corresponds to maximizing throughput while minimizing latency in a balanced manner.

To select the thresholds, we adopt a control-variable method to evaluate performance under different values of $Orl_{\max}$ and $Orl_{\min}$, and the latency and throughput are recorded after the system reaches a steady state.

Fig. 15 illustrates how the average system latency and throughput change under different threshold settings. Because latency and throughput respond differently to threshold changes, a single metric is insufficient for parameter selection. Accordingly, $Orl_{\max}$ and $Orl_{\min}$ are selected by maximizing the composite evaluation score f in Eq. (19). In this procedure, one threshold is varied while the other is fixed. After the system reaches a steady state, the average latency L and throughput T are measured, and the corresponding evaluation score f is computed. The setting that yields the highest f is chosen as the optimal configuration.

Table 4 lists the resulting f values for different $Orl_{\max}$ candidates, and the maximum is attained at $Orl_{\max} = 0.875$. Similarly, Table 5 reports the scores for different $Orl_{\min}$ values, where the maximum occurs at $Orl_{\min} = 0.30$.

Table 4
Evaluation function f for varying $Orl_{\max}$.

$Orl_{\max}$	0.80	0.825	0.85	0.875	0.90	0.925	0.95
f	437.94	435.73	436.58	438.08	435.65	432.82	425.63

Table 5
Evaluation function f for varying $Orl_{\min}$.

$Orl_{\min}$	0.25	0.275	0.30	0.325	0.35	0.375	0.40
f	426.26	423.50	434.73	427.67	433.40	431.28	428.72

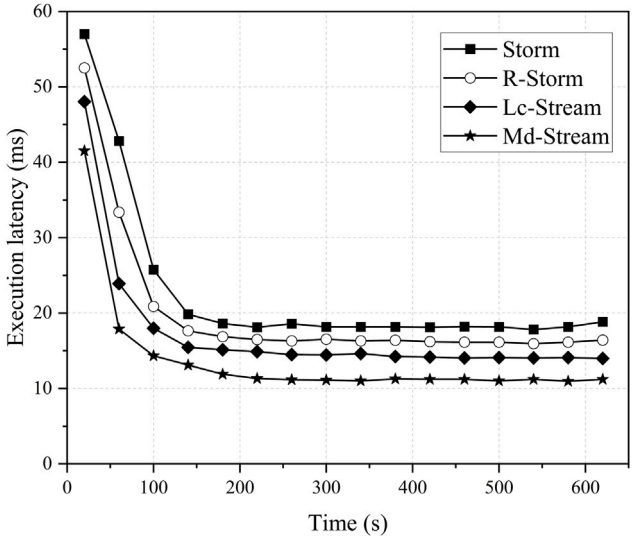


Fig. 16. Execution latency comparison between Md-Stream and other frameworks.

For the stability threshold $Time_{stability}$, latency monitoring over 60 min indicates that system performance stabilizes after 700 s. Thus, $Time_{stability} = 700$ s is used for all experiments.

Comparative evaluations are conducted between Md-Stream and the following baselines: **DefaultScheduler**, the built-in scheduler provided by Apache Storm; **Lc-Stream** [42], a recent multi-domain scheduling strategy with a comparable design; **R-Storm**, a resource-aware scheduler for Apache Storm that considers multi-resource constraints (e.g., CPU and memory) and seeks to reduce intra-cluster communication overhead through resource-aware executor placement; and **SP-Ant**, an optimization-based scheduler that combines a co-location (bin-packing-style) step with an ant colony optimization procedure to iteratively refine executor placement and reduce communication cost. Both WordCount and Top-N topologies are used for this comparison. Performance is evaluated using three metrics: latency, throughput, and resource utilization.

6.3. System latency

System latency is a key indicator of scheduling strategy effectiveness. Lower latency translates to faster processing and more responsive feedback. By analyzing latency metrics, we can assess the strengths and limitations of different scheduling approaches.

In this experiment, system latency is observed using the Storm UI, with a Python crawler used to record and extract metrics. The latency data were collected during a 60-minute execution window, with key results reported for the first 700 s and the entire 60-minute run.

As shown in Fig. 16, the execution latency of all components over the first 700 s is compared among Md-Stream, the Storm default scheduler, R-Storm, and Lc-Stream, where the x-axis represents time (seconds) and the y-axis represents execution latency (milliseconds).

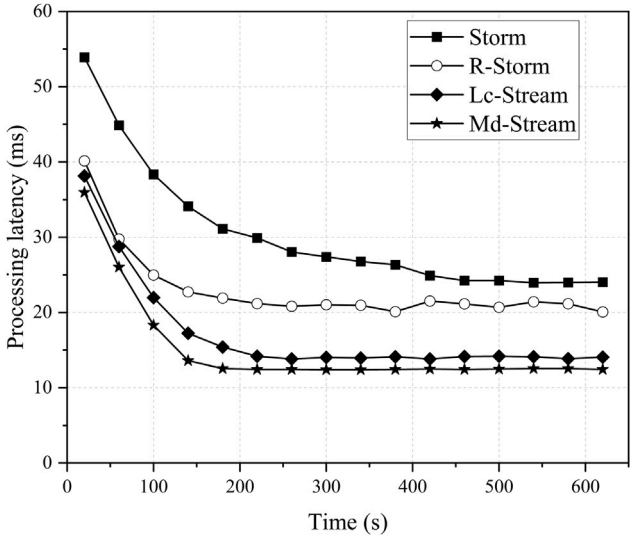


Fig. 17. Processing latency comparison between Md-Stream and other frameworks.

Execution latency measures the time between when a tuple is received and when processing begins, primarily reflecting queuing delay.

At startup, all policies exhibit high latency due to system initialization and configuration loading. After about 120 s, the Storm default policy stabilizes with an average latency of roughly 18 ms. R-Storm converges to a stable state after approximately 130 s, with an average execution latency maintained at around 16 ms, realizing a slight optimization over the native Storm policy. Lc-Stream drops to about 15 ms after around 140 s and then remains near 14.4 ms. In contrast, Md-Stream reaches a steady state after about 200 s with a lower average execution latency of approximately 12 ms, reducing latency by about 30%–40% compared with Storm's default scheduler, by around 25% compared with R-Storm, and by about 10%–20% compared with Lc-Stream within the first 700 s.

As shown in Fig. 17, processing latency (i.e., the total time taken to handle a tuple, including all method execution and associated operations) is plotted over the first 700 s for Md-Stream, Storm, R-Storm, and Lc-Stream.

All strategies exhibit high initial latency that gradually declines as the system warms up: Storm and Md-Stream begin to decrease after roughly 120 s, R-Storm maintains a continuous downward trend until approximately 200 s, while Lc-Stream continues decreasing until about 220 s and Md-Stream until about 160 s. Once the system reaches a steady state, Storm fluctuates around 25 ms, R-Storm stabilizes at approximately 21 ms, whereas Md-Stream stabilizes at approximately 13 ms, corresponding to about a 50% reduction compared with Storm and a roughly 38% reduction compared with R-Storm. In comparison with Lc-Stream, which stabilizes at around 14 ms, Md-Stream remains lower at about 13 ms, reducing average processing latency by roughly 10% within 700 s.

As shown in Fig. 18, the experimental comparison in the multi-domain scheduling setting indicates that SP-Ant achieves lower response time in a single-domain environment, mainly because its ant colony optimization (ACO)–based iterative search, together with the co-location (pinning) of communication-intensive operators, effectively reduces inter-operator communication overhead.

However, when extended to multi-domain deployments, the additional cross-domain network latency is explicitly amplified, while SP-Ant lacks mechanisms for multi-domain cooperative placement and dependency-aware control of cross-domain cuts; as a result, critical dependency chains are more likely to span domains, leading to a

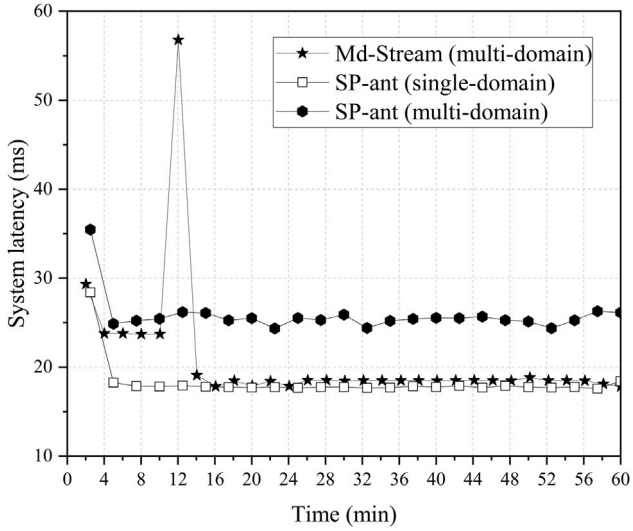


Fig. 18. Total system latency comparison between multi-domain Md-Stream and SP-ant frameworks.

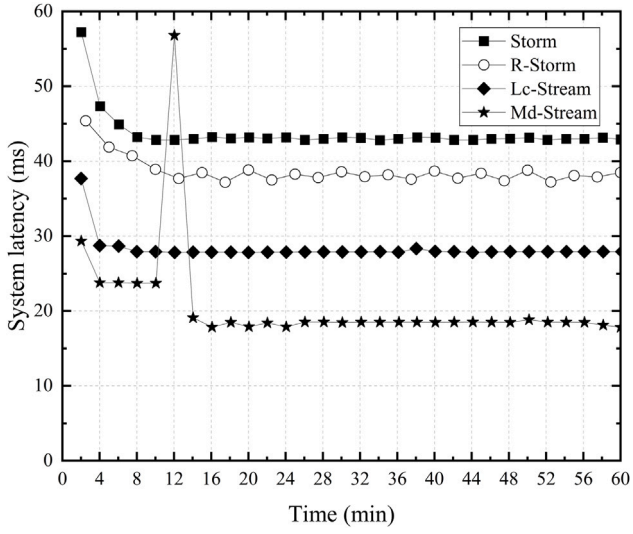


Fig. 19. Total system latency comparison between Md-Stream and other frameworks.

noticeable performance degradation compared with the single-domain case.

In contrast, Md-Stream leverages dependency-aware partitioning and multi-domain cooperative scheduling to consolidate strongly dependent tasks within the same domain or low-latency domains, thereby mitigating cross-domain communication waiting. Moreover, once the system reaches a stable operating regime, Md-Stream activates elastic mechanisms, such as dynamic parallelism adjustment and idling of underutilized nodes, to alleviate runtime bottlenecks and improve effective resource utilization. Consequently, Md-Stream demonstrates a clearer advantage over SP-Ant in multi-domain scheduling, achieving lower end-to-end latency.

As shown in Fig. 19, Md-Stream, Storm, R-Storm, and Lc-Stream are the four scheduling strategies for 60 min of total system latency. In order to avoid a large fluctuation of data in a certain 10 s which is not favorable for observing the results, it shows the average system latency for each 120 s, totaling 60 min. The total system latency is calculated as

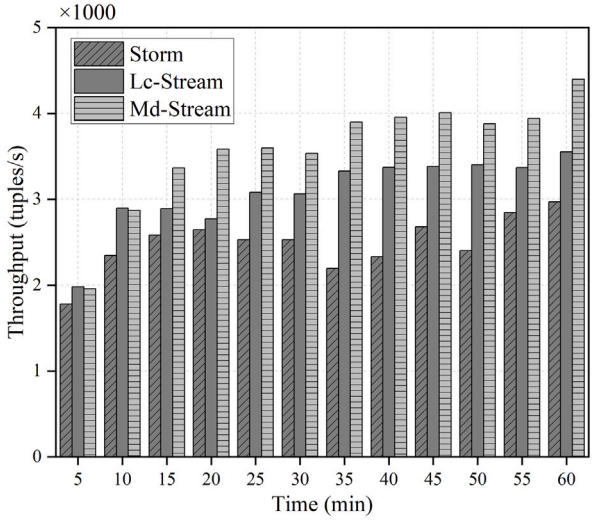


Fig. 20. WordCount throughput comparison between Md-Stream and other frameworks.

the sum of the execution latency of all components and the processing latency of all components. From Fig. 19, it can be seen that when the total system latency enters into stabilization, the system latency of the Storm policy stays around 43 ms, the R-Storm policy maintains a stable system latency at approximately 38 ms, and the total system latency of the Lc-Stream policy stays around 28 ms. Before 12 min, the stabilized total system latency of the Md-Stream policy is about 24 ms.

Notably, after the system reaches the defined stability time ($Time_{stability}$ = 700 s), Md-Stream's resource elasticity mechanism is activated. This mechanism dynamically adjusts resource allocations (e.g., thread counts for specific components), further optimizing performance.

Compared to the Storm default policy, Md-Stream reduces total system latency by approximately 56.9%. When compared to the R-Storm policy, Md-Stream achieves a significant latency reduction of around 52.6%. When compared to Lc-Stream, it still achieves a substantial reduction of around 34.0%.

Overall, the latency reduction is primarily attributed to Md-Stream's dependency-aware scheduling pipeline. The KFM-based partitioning minimizes the weight of cross-partition dependency edges, thereby reducing inter-partition communication. The subsequent multi-domain allocation co-locates communication-intensive partitions within low-latency domains, further suppressing cross-domain transmissions among strongly dependent operators. As a result, tuples experience reduced communication waiting and queuing contention, leading to lower execution latency and improved end-to-end processing delay in steady state.

6.4. System throughput

System throughput reflects a system's data processing capability per unit of time. In this experiment, we quantify the throughput under Md-Stream, Storm, and Lc-Stream strategies. After submitting WordCount and Top-N topologies to the cluster, we observe system behavior through Storm UI, and record data using a Python crawler. The experiment duration is 60 min, with system throughput measured every 300 s. The experimental results are shown in Figs. 20 to 21.

As shown in Fig. 20, the system throughput of WordCount over 60 min is compared under Md-Stream, Storm's default scheduling strategy, and Lc-Stream. Throughput is calculated as the ratio of the change in Acked tuples to the time window of the counting component, representing the number of tuples successfully processed per unit time.

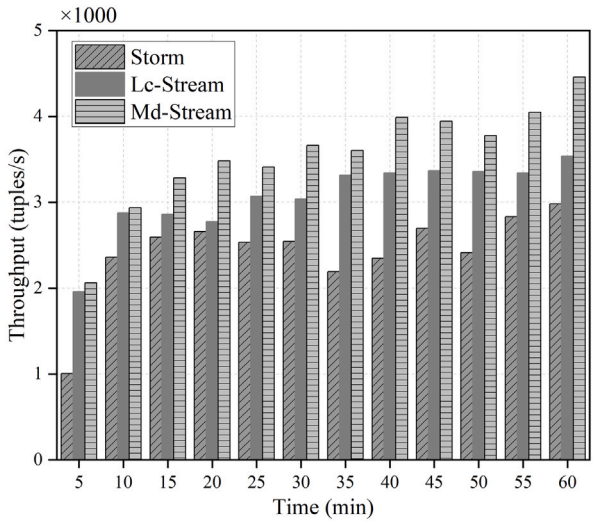


Fig. 21. Top-N throughput comparison between Md-Stream and other frameworks.

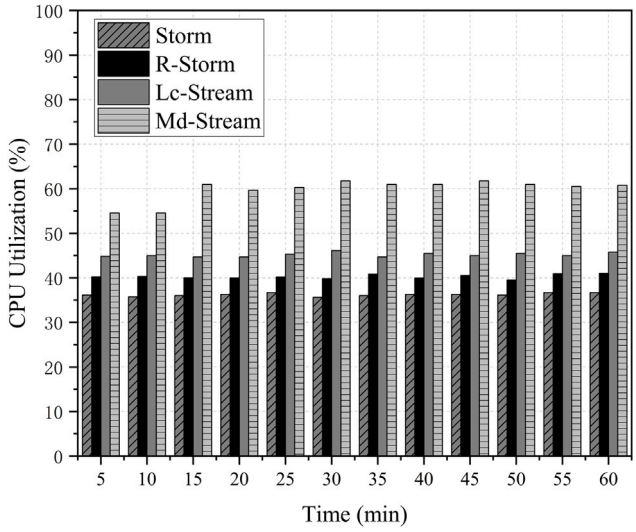


Fig. 22. Average CPU utilization comparison between Md-Stream and other frameworks.

In the first 5 min, due to configuration loading, throughput for all strategies remains below 2000 tuples/s; afterward, Storm stabilizes with an average throughput of 2565 tuples/s. Around 700 s, Md-Stream triggers its resource elasticity mechanism, leading to a noticeable throughput increase between 10 and 15 min, and achieving an average of 3589 tuples/s over the full duration, an improvement of 39.9% compared with Storm. Compared with Lc-Stream, which averages 3200 tuples/s, Md-Stream still attains higher throughput, reflecting a 12.2% improvement.

As shown in Fig. 21, Top-N throughput is compared among Md-Stream, Storm, and Lc-Stream. Storm maintains an average throughput of 2565 tuples/s, while Md-Stream reaches an average of 3554 tuples/s, improving upon Storm by 38.6%; although Md-Stream exhibits a modest increase at around 15 min, the gain is limited because Top-N already has sufficient executor resources. Compared with Lc-Stream, which achieves an average throughput of 3198 tuples/s, Md-Stream still attains higher throughput, yielding an improvement of 11.1%. Overall, Md-Stream improves throughput by 39.3% relative to Storm and by 11.7% relative to Lc-Stream.

The throughput improvement mainly arises from reduced communication-induced stalls and improved alignment between operator parallelism and node capacity. By minimizing dependency cuts and co-locating tightly coupled tasks, Md-Stream reduces costly remote data transfers, enabling executors to spend more time on effective computation. After the system reaches the stability threshold, the elasticity mechanism dynamically adjusts operator parallelism based on capacity indicators, mitigating runtime bottlenecks and sustaining a higher processing rate, which explains the observed throughput gains.

6.5. Resource utilization

As shown in Fig. 22, the average CPU utilization over a 60-minute monitoring window for Md-Stream, alongside the baseline Storm, R-Storm, and Lc-Stream, is presented. Utilization is computed as the expected value across all active compute nodes, excluding dormant nodes.

Storm's average CPU utilization is approximately 36.5%, while R-Storm maintains a stable average CPU utilization of around 40.5% throughout the entire test period, realizing a moderate improvement over native Storm through optimized resource scheduling. Md-Stream reaches around 55% CPU utilization at the 5th and 10th min, and stabilizes at about 60.5% from the 15th min onward. This improvement is attributed to Md-Stream's resource elasticity mechanism, which dynamically adjusts operator parallelism and hibernates idle nodes, resulting in a 65.8% increase in average CPU utilization compared with Storm, and a 49.4% uplift relative to R-Storm. Lc-Stream maintains an average utilization of about 45.5% by similarly hibernating some idle nodes, delivering a moderate performance gain over both Storm and R-Storm; nevertheless, Md-Stream further improves average CPU utilization by approximately 33.0% over Lc-Stream.

Higher resource utilization is achieved not by increasing system load, but by improving effective utilization. Dependency-aware partitioning and domain-aware allocation reduce communication overhead and idle waiting, while the elasticity mechanism continuously aligns parallelism with runtime capacity. In addition, idle nodes are hibernated after scheduling rounds to prevent prolonged underutilization. In contrast, dependency-unaware schedulers may scatter correlated tasks across domains, increasing communication overhead and idle CPU gaps, resulting in lower overall utilization.

7. Conclusions and future work

The proposed multi-domain cooperative scheduling framework, Md-Stream, is designed for cross-domain stream computing environments. By partitioning tasks into fine-grained graphs and assigning highly dependent tasks to the same domain, Md-Stream effectively reduces inter-domain I/O overhead. Additionally, the resource elasticity mechanism dynamically adjusts operator parallelism, thereby lowering system latency and improving throughput and resource utilization.

Experimental results demonstrate that Md-Stream delivers substantial performance gains. However, certain limitations remain: (1) Sensitivity to data surges: Md-Stream performs well under stable data rates but may struggle during burst events that cause sudden traffic spikes. (2) Limited load balancing: The current scheduler prioritizes communication efficiency and dependency constraints, yet overlooks global load balance, leading to potential node overloads.

Future work will enhance the dataflow monitoring module to support adaptive responses to real-time traffic variations. This improvement also support the extension of Md-Stream to large-model-native stream computing systems. In such scenarios, streaming applications commonly exhibit complex DAG dependencies, dynamic workload fluctuations, and heterogeneous resource requirements, and incorporating global load balancing to ensure fair workload distribution while minimizing communication overhead. With further support from cloud-native runtime monitoring and finer-grained scheduling coordination, Md-Stream can serve as a feasible foundation for efficient scheduling and adaptive optimization in large-model-native stream computing environments.

CRediT authorship contribution statement

Dawei Sun: Writing – review & editing, Methodology, Funding acquisition. **Zhongyuan Zhao:** Writing – original draft, Validation, Methodology, Conceptualization. **Yueru Wang:** Writing – original draft, Validation, Conceptualization. **Shang Gao:** Writing – review & editing, Investigation, Formal analysis. **Rajkumar Buyya:** Writing – review & editing, Methodology.

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. 62372419; the Fundamental Research Funds for the Central Universities, China under Grant No. 265QZ2021001;

This paper is a substantial extension of a 6-page paper [19] presented at HPCC 2024.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.future.2026.108616.

Data availability

Data will be made available on request.

References

[1] Q.-C. To, J. Soto, V. Markl, A survey of state management in big data processing systems, VLDB J. 27 (6) (2018) 847–872, http://dx.doi.org/10.1007/s00778-018-0514-9.
[2] Apache spark, 2026, https://spark.apache.org/streaming/.
[3] Apache samza, 2026, http://samza.apache.org/.
[4] 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, http://dx.doi.org/10.1016/j.future.2020.06.009.
[5] Apache storm, 2026, https://storm.apache.org/.
[6] Apache flink, 2026, https://github.com/apache/flink.
[7] X. Liu, R. Buyya, Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions, ACM Comput. Surv. 53 (3) (2020) http://dx.doi.org/10.1145/3355399.
[8] M. Bansal, I. Chana, S. Clarke, A survey on IoT big data: Current status, 13 V's challenges, and future directions, ACM Comput. Surv. 53 (6) (2020) http://dx.doi.org/10.1145/3419634.
[9] C. Li, Q. Cai, Y. Luo, Data balancing-based intermediate data partitioning and check point-based cache recovery in Spark environment, J. Supercomput. 78 (3) (2022) 3561–3604, http://dx.doi.org/10.1007/s11227-021-04000-2.
[10] A. Kishor, R. Niyogi, B. Veeravalli, Fairness-aware mechanism for load balancing in distributed systems, IEEE Trans. Serv. Comput. 15 (4) (2022) 2275–2288, http://dx.doi.org/10.1109/TSC.2020.3044104.
[11] Z. Cheng, Q. Huang, P.P.C. Lee, On the performance and convergence of distributed stream processing via approximate fault tolerance, VLDB J. 28 (5) (2019) 821–846, http://dx.doi.org/10.1007/s00778-019-00565-w.
[12] 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, http://dx.doi.org/10.1109/TPDS.2018.2881176.
[13] D. Cheng, X. Zhou, Y. Wang, C. Jiang, Adaptive scheduling parallel jobs with dynamic batching in spark streaming, IEEE Trans. Parallel Distrib. Syst. 29 (12) (2018) 2672–2685, http://dx.doi.org/10.1109/TPDS.2018.2846234.
[14] A. Muhammad, M. Aleem, BAN-storm: a bandwidth-aware scheduling mechanism for stream jobs, J. Grid Comput. 19 (3) (2021) http://dx.doi.org/10.1007/s10723-021-09567-x.
[15] 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, http://dx.doi.org/10.1109/TPDS.2019.2891587.
[16] H. Li, J. Xia, W. Luo, H. Fang, Cost-efficient scheduling of streaming applications in apache flink on cloud, IEEE Trans. Big Data 9 (4) (2023) 1086–1101, http://dx.doi.org/10.1109/TBDATA.2022.3233031.
[17] J. Chen, Y. He, Y. Zhang, P. Han, C. Du, Energy-aware scheduling for dependent tasks in heterogeneous multiprocessor systems, J. Syst. Archit. 129 (2022) 102598, http://dx.doi.org/10.1016/j.sysarc.2022.102598.
[18] V. Cardellini, F. Lo Presti, M. Nardelli, G.R. Russo, Runtime adaptation of data stream processing systems: The state of the art, ACM Comput. Surv. 54 (11s) (2022) http://dx.doi.org/10.1145/3514496.
[19] D. Sun, Z. Zhao, Y. Wang, S. Gao, R. Buyya, A task dependency-aware scheduling strategy for cross-domain stream computing environments, in: 2024 IEEE International Conference on High Performance Computing and Communications, HPCC, 2024, pp. 450–455, http://dx.doi.org/10.1109/HPCC64274.2024.00067.
[20] L. Eskandari, Z. Huang, D. Eyers, P-scheduler: Adaptive hierarchical scheduling in apache storm, in: Proceedings of the Australasian Computer Science Week Multiconference, ACSW'16, 2016, pp. 26:1–26:10, http://dx.doi.org/10.1145/2843043.2843056.
[21] 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, http://dx.doi.org/10.1016/j.future.2018.07.011.
[22] 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, http://dx.doi.org/10.1016/j.future.2020.11.011.
[23] M. Farrokh, H. Hadian, M. Sharifi, A. Jafari, SP-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters, Expert Syst. Appl. 191 (2022) 116322, http://dx.doi.org/10.1016/j.eswa.2021.116322.
[24] H. Hadian, M. Farrokh, M. Sharifi, A. Jafari, An elastic and traffic-aware scheduler for distributed data stream processing in heterogeneous clusters, J. Supercomput. 79 (1) (2023) 461–498, http://dx.doi.org/10.1007/s11227-022-04669-z.
[25] C. Li, Q. Cai, Y. Lou, Optimal data placement strategy considering capacity limitation and load balancing in geographically distributed cloud, Future Gener. Comput. Syst. 127 (2022) 142–159, http://dx.doi.org/10.1016/j.future.2021.08.014.
[26] Y. Wang, L. Huang, Z. Wang, V. Kalavri, I. Matta, CAPSys: Contention-aware task placement for data stream processing, in: Proceedings of the Twentieth European Conference on Computer Systems, EuroSys '25, Association for Computing Machinery, New York, NY, USA, ISBN: 9798400711961, 2025, pp. 654–670, http://dx.doi.org/10.1145/3689031.3696085.
[27] B. Peng, M. Hosseini, Z. Hong, R. Farivar, R. Campbell, R-storm: Resource-aware scheduling in storm, in: Proceedings of the 16th Annual Middleware Conference, 2015, pp. 149–161, http://dx.doi.org/10.1145/2814576.2814808.
[28] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator replication and placement for distributed stream processing systems, SIGMETRICS Perform. Eval. Rev. 44 (4) (2017) 11–22, http://dx.doi.org/10.1145/3092819.3092823.
[29] F. Liu, W. Zhu, W. Mu, Y. Zhang, M. Li, Z. Zhu, W. Wang, Elastic resource allocation based on dynamic perception of operator influence domain in distributed stream processing, in: Computational Science - ICCS 2022, PT I, 2022, pp. 734–748, http://dx.doi.org/10.1007/978-3-031-08751-6_53.
[30] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392, http://dx.doi.org/10.1137/S1064827595287997.
[31] X. Xu, W. Li, H. Qi, J. Wang, K. Li, Latency-constrained cost-minimized request allocation for geo-distributed cloud services, IEEE Open J. Commun. Soc. 1 (2020) 125–132, http://dx.doi.org/10.1109/OJCOMS.2020.2964303.
[32] E. Khodayarseresht, A. Shameli-Sendi, Q. Fournier, M. Dagenais, Energy and carbon-aware initial VM placement in geographically distributed cloud data centers, Sustain. Comput.: Inform. Syst. 39 (2023) 100888, http://dx.doi.org/10.1016/j.suscom.2023.100888.
[33] S. Hosseinalipour, A. Nayak, H. Dai, Power-aware allocation of graph jobs in geo-distributed cloud networks, IEEE Trans. Parallel Distrib. Syst. 31 (4) (2020) 749–765, http://dx.doi.org/10.1109/TPDS.2019.2943457.
[34] D. Wladdimiro, L. Arantes, P. Sens, N. Hidalgo, PA-SPS: A predictive adaptive approach for an elastic stream processing system, J. Parallel Distrib. Comput. 192 (2024) 104940, http://dx.doi.org/10.1016/j.jpdc.2024.104940.
[35] X. Li, R. Fan, H. Hu, N. Zhang, Joint task offloading and resource allocation for cooperative mobile-edge computing under sequential task dependency, IEEE Internet Things J. 9 (23) (2022) 24009–24029, http://dx.doi.org/10.1109/JIOT.2022.3188933.
[36] A. Muhammad, M. Aleem, A3-storm: topology-, traffic-, and resource-aware storm scheduler for heterogeneous clusters, J. Supercomput. 77 (2) (2021) 1059–1093, http://dx.doi.org/10.1007/s11227-020-03289-9.
[37] J. Herrmann, J. Kho, B. Uçar, K. Kaya, Ü.V. Çatalyürek, Acyclic partitioning of large directed acyclic graphs, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID, 2017, pp. 371–380, http://dx.doi.org/10.1109/CCGRID.2017.101.
[38] I. Dimolitsas, D. Dechouniotis, S. Papavassiliou, Time-efficient distributed virtual network embedding for round-trip delay minimization, J. Netw. Comput. Appl. 217 (2023) 103691, http://dx.doi.org/10.1016/j.jnca.2023.103691.

[39] W. Yue, M. Boissier, T. Rabl, A survey of stream processing system benchmarks, in: R. Nambiar, M. Poess (Eds.), Performance Evaluation and Benchmarking, Springer Nature Switzerland, Cham, 2026, pp. 24–43, http://dx.doi.org/10.1007/978-3-031-93858-0_2.
[40] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B.J. Peng, P. Poulosky, Benchmarking streaming computation engines: Storm, flink and spark streaming, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2016, pp. 1789–1792, http://dx.doi.org/10.1109/IPDPSW.2016.138.
[41] Yahoo! Inc., Yahoo streaming benchmarks, 2026, https://github.com/yahoo/streaming-benchmarks.
[42] D. Sun, Y. Wang, J. Sui, S. Gao, J. Rong, R. Buyya, Lc-stream: An elastic scheduling strategy with latency constraints in geo-distributed stream computing environments, Concurr. Comput.-Pract. Exp. 36 (14) (2024) http://dx.doi.org/10.1002/cpe.8085.

Dawei Sun is a Professor in the School of Artificial Intelligence, China University of Geosciences, Beijing, PR 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 100 journal and conference papers

Zhongyuan Zhao is a postgraduate student in the School of Artificial Intelligence, China University of Geosciences, Beijing, China. He received the B.E. Degree in Computer Science and Technology from Hebei Agricultural University, BaoDing, China in 2024. His research interests include big data stream computing and distributed systems.

Yueru Wang received her Bachelor of Engineering in Computer Science and Technology from China University of Geosciences (Beijing) in 2021, followed by a Master of Engineering in Computer Technology from the same institution in 2024. Her research focuses on big data distributed computing and cloud computing, emphasizing the development of innovative solutions to enhance scalability and operational efficiency in these domains.

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 Quantum Cloud Computing and Distributed Systems (qCLOUDS) 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 850 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 181 with 170,200+citations). He is among the world' s top 2 most influential scientists in distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. 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.