Hierarchical Dependency-Aware Scheduling for Distributed Stream Computing Systems

Yinuo Fan[1], Dawei Sun[1⋆], Shuaiyi Zou[1], Jonathan Kua[2], and Rajkumar Buyya[3]

[1] School of Information Engineering, China University of Geosciences, Beijing 100083, China
fanyinuocn@email.cugb.edu.cn, sundaweicn@cugb.edu.cn, zoushuaiyi@email.cugb.edu.cn

[2] School of Information Technology, Deakin University, Geelong, VIC 3220, Australia
jonathan.kua@deakin.edu.au

[3] Quantum Cloud Computing and Distributed Systems (qCLOUDS) Lab, School of Computing and Information Systems, The University of Melbourne, Parkville, Victoria 3010, Australia
rbuyya@unimelb.edu.au

Abstract. Scheduling strategies play a crucial role in distributed stream computing systems. Many state-of-the-art works focus on load balancing of computing nodes and dependencies of stream application tasks, However, these methods do not sufficiently consider differentiated communication costs between computing nodes. Scheduling computing nodes with significant differences in communication costs to high-dependency task leads to increased communication latency, thereby degrading the overall system latency. To overcome these limitations, we propose Hd-Stream, which is a hierarchical dependency-aware scheduling mechanism for distributed stream systems. Our Hd-Stream mechanism comprises: (1) The dependency relationships of stream application tasks and computing nodes are hierarchically quantified by constructing a stream application model, a task dependency model, and a resource dependency model. (2) A Communication Dependency-aware virtual resource node (VRN) scheduling algorithm is proposed to minimize the communication costs between computing nodes. (3) A hierarchical dependency-aware scheduling algorithm is proposed for reducing communication latency and optimize resource utilization based on maximizing migration benefits. (4) Multiple metrics are evaluated, including system latency, system throughput, and resource utilization n real distributed stream computing scenarios. Our experimental results demonstrate that Hd-Stream reduces system latency by 42%, increases average throughput by 25%, and achieves efficient resource utilization.

Keywords: Hierarchical dependency-aware · Scheduling · Communication latency · Distributed stream computing

⋆ Dawei Sun, sundaweicn@cugb.edu.cn

1 Introduction

Distributed stream computing systems (DSC), such as Apache Storm and Spark Streaming [1], are widely used in various applications (e.g., fraud detection [2]) due to their ability to process dynamic, continuous, and unbounded data streams with millisecond-level latency. Processing data streams requires not only the rapid execution of complex computations such as filtering, aggregation, and correlation [3], but also the assurance of excellent system performance. Although DCS systems provide rich stream data processing capabilities and efficient scheduling mechanisms [4], they still face many challenges in more complex application scenarios, such as when dealing with computational resource heterogeneity.

DSC systems' key optimization includes data stream grouping [5], scheduling [6], and fault tolerance management [7]. The optimization objectives are decreasing latency, increasing throughput and resource utilization, while ensuring system stability. Scheduling involves allocating computational node resources to tasks within stream applications while avoiding frequent network transmissions caused by task communication, which can lead to high communication latency or performance bottlenecks due to uneven resource load. Additionally, the system must dynamically monitor and adjust scheduling schemes to respond to fluctuating data streams, ensuring system performance and data integrity [8].

In DSC systems, stream applications are typically represented as directed acyclic graphs (DAGs) connected by data dependencies [9]. The vertices represent data processing operations, the edges indicate the direction of data stream, and the weights reflect the communication traffic between vertices. The critical path of the DAG reflects the system's response latency [10]. To improve the computational efficiency of stream applications, it is essential to utilize runtime context information of DSC systems and minimize the latency of cross-network communication. This is achieved by reducing the communication traffic between computing nodes caused by interactions between tasks. Since scheduling is an NP-hard problem [11] and the data stream continuously fluctuating, it is crucial to balance the scheduling overhead with the scheduling effectiveness. This is necessary to prevent system crashes due to message queue failures during the execution of scheduling.

In complex network environments, a key research challenge is to comprehensively integrate runtime information from both the application logic layer and the physical resource layer in order to achieve a balance between scheduling overhead and scheduling benefits. Although existing scheduling strategies [12, 13] have achieved some success in optimizing operator dependencies and resource management, they do not sufficiently consider the differences in communication costs and distribution characteristics of physical computing nodes. As a result, even with optimized task dependencies at the logical level, actual deployments may still incur additional system performance overhead due to communication latency. Therefore, scheduling strategies need to further integrate context information, particularly by incorporating awareness of communication latency and optimizing resource management. This approach aims to effectively mitigate the

aforementioned issues while ensuring system performance, achieving low latency, high throughput, and more balanced and efficient resource utilization.

To address the aforementioned challenges, we propose Hd-Stream, which is a hierarchical dependency-aware scheduling for distributed stream computing systems. Hd-Stream leverages the dependencies between stream applications at the logical layer and computing nodes at the physical layer to achieve efficient scheduling. The contributions of this paper are as follows:

(1) We constructed a stream application model, a task dependency model, and a resource dependency model to hierarchically quantify the dependencies between stream applications and computing node.
(2) We proposed a communication dependency-aware virtual resource node scheduling (VRN) algorithm to minimize the communication costs between computing nodes.
(3) We proposed a hierarchical dependency-aware scheduling algorithm to reduce communication latency and optimize resource utilization according to maximum migration benefits.
(4) We implemented Hd-Stream and integrated it into Apache Storm, and evaluated its performance in a real distributed stream computing environment.

1.1 Paper Organization

The rest of this paper is organized as follows: Section 2 reviews and analyzes related work in the field of distributed stream computing scheduling. Section 3 constructs the stream application model, the task dependency model and the resource dependency model. Section 4 introduces Hd-Stream, including its system architecture and algorithms. Section 5 evaluates the performance of Hd-Stream in a real-world distributed stream computing environment. Section 6 concludes the paper and outlines future work.

2 Related Work

Scheduling is one of the crucial research areas in DSC systems [14]. Since the complexity of streaming applications and the heterogeneity of computing resources of computing resources, it is challenging to find an optimal scheduling scheme. We reviewed existing scheduling strategies and analyzed their advantages and limitations.

I-Scheduler [15] first divides the streaming application into subgraphs by graph partitioning technique, subsequently prioritizes the task subgraphs with higher communication dependency to the most resource-sufficient computing node based on the inverse order of resource node capacity. However, I-Scheduler heavily relies on the optimization software and the empirical partition capacity threshold definition.

SP-Ant [16] used the ACO algorithm to iteratively optimize the scheduling scheme and adopted the boxing algorithm as the initial scheduling scheme to

accelerate the convergence process of the ACO algorithm. However, since the ACO algorithm requires multiple iterations to approach the optimal solution, its overall convergence speed is slow and may introduce additional scheduling delays in dynamic data flow environments.

The work in [17] proposed Cost-Efficient Task Scheduling Algorithm (CETSA) and Cost-Effective Load Balancing Algorithm (LBA-CE) to optimize load balancing while reducing the cost of job execution. These algorithms ensure workload balancing in heterogeneous clusters while minimizing cost. However, these algorithms do not adequately consider the real-time fluctuations in the data stream rate.

The method presented in [18] is a resource scheduling and provisioning approach under latency constraints, designed to address the complexity and unpredictability of dynamic streaming workflow scenarios. This approach assumes that the data communication overhead is negligible. However, the communication overhead is an important factor that cannot be ignored in the real-world.

D-Storm [19] abstracts the scheduling problem into a crating problem and proposed an algorithm based on a greedy policy to minimize the communication latency between nodes. The algorithm is able to dynamically acquire resource requirements and dynamically make scheduling decisions at system runtime. However, the algorithm only focuses on the communication volume when ordering the cluster arithmetic tasks and does not fully consider the heterogeneous arithmetic characteristics of the nodes.

The work in [20] proposed a method to reduce network latency caused by distributed computing across multiple devices by formulating the complete continuity of computing resources through a formal scheduling optimization problem. However, the method does not adjust the utilization of computing resources according to the changes in data stream.

Table 1: Comparison of Ra-Stream and related work.

Scheduler	**Aspects**		
	Heterogeneous cluster	Communication cost	Load balancing
I-Scheduler [15]	✓	✓	×
SP-Ant [16]	✓	✓	×
VM provisioner [18]	✓	×	×
D-Storm [19]	×	✓	×
Ra-Stream(Ours)	✓	✓	✓

The aforementioned approaches have improved the scheduling strategies and have achieved excellent results. However, most of them do not comprehensively consider the complexity of the computing environment, especially in data streams fluctuation and the differentiated communication cost of computing nodes. We systematically compare and analyze Hd-Stream with existing studies in Table 1.

3 System Model

The scheduling effectiveness and stability of DSC systems are influenced by factors such as communication costs between computational nodes, available resources, and data stream rates. In this work, we constructed a stream application model, a task dependency model, and a resource dependency model to provide a solid foundation for the scheduling strategy.

3.1 Stream Application Model

The logical topology of stream applications is often abstracted as $G = \{V(G), E(G)\}$, where $V(G) = \{v_1, v_2, \cdots, v_n\}$ represent n vertices within G, each vertex $v_i \in V(G)$ denotes a specific logical computation, responsible for sending or processing data streams. $E(G)$ represent the directed edges in G, and $E(G) = \{e_{v_i,v_j} | i < j < n\}$, where e_{v_i,v_j} represents a set of directed edges between v_i and v_j, indicating the data stream transmission.

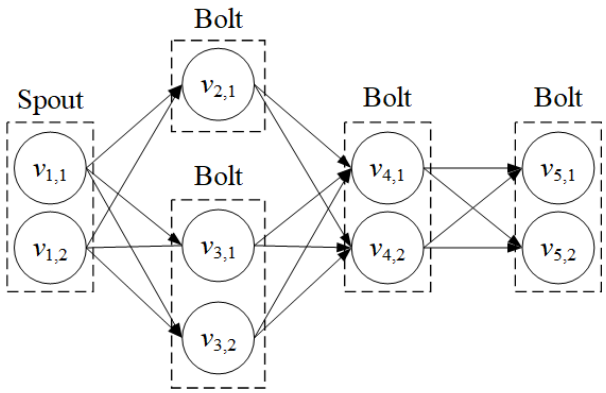

Fig. 1: The task topology of a stream application

When a user constructs a stream application and submits it to the DSC system, the DSC system creates the corresponding number of task instances based on the default parallelism settings for each vertex in the submitted stream application. Each vertex can have one or more task instances. As shown in Fig. 1, the stream application consists of four vertices: one Spout and three Bolts. The parallelism, or number of task instances for each vertex, is 2, 3, 1, and 2, respectively. The system applies the specified scheduling strategy, placing the task instances to physical resource nodes within the cluster according to the task topology of the stream application.

3.2 Task Dependency Model

Due to the differing grouping strategies among task instances, there may be significant variations in the communication traffic between instances of the same vertex with upstream vertex task instances. Consequently, the dependencies among the various vertex task instances will differ. During the scheduling process, it is essential to consider the communication traffic between task instances to quantify their dependencies. This helps optimize the scheduling strategy, reduce inter-node communication, and enhance overall system performance.

When task instances with communication connections are deployed on the same computing node, their dependency is considered to be 0, as there is no data transmission across the network. We define $w_{v_{i,k},v_{j,m}}$ as the average data transfer rate between upstream task instance $v_{i,k}$ and downstream task instance $v_{j,m}$, which represents the mathematical expectation of the data transfer rate between the two task instances over a unit of time. We define $d_{v_{i,k},v_{j,m}}$ to represent the dependency between $v_{i,k}$ and $v_{j,m}$, which can be calculated by (1):

$$d_{v_{i,k},v_{j,m}} = \begin{cases} 0, & v_{i,k}, v_{j,m} \text{ are deployed in the same node,} \\ w_{v_{i,k},v_{j,m}}, & \text{Otherwise.} \end{cases} \tag{1}$$

Communication between task instances includes inter-node communication and intra-node communication. The latency for inter-node communication is significantly higher than the latency for intra-node communication [21]. The scheduling strategy should consider the communication type when partitioning task instances in the stream application. The goal is to maximize the total task instance dependencies within computing nodes while minimizing the total task instance dependencies between computing nodes.

3.3 Resource Dependency Model

The communication cost between computing nodes reflects their dependencies. The design of the scheduling strategy should fully consider dependency differences to minimize communication latency. Additionally, when the dependencies between computing nodes vary significantly, the system needs to adjust the scheduling scheme in a timely manner to avoid performance degradation. Balancing scheduling benefits and scheduling overhead is an important consideration.

The dependency of cn_i and cn_j are denoted as $X_{i,j}$, We construct the dependency feature matrix X for the computing nodes in the cluster, which satisfies $N = |X|$, N is the total number of compute nodes. We introduce the concept of resource clustering for computing nodes, grouping nodes with similar dependency differences into the same cluster, referred to as virtual resource nodes VRN.

We leverage the K-means clustering algorithm combined with the elbow method to cluster the computing nodes. First, we randomly initialize k feature vectors of the computing nodes as the initial cluster centers. We then calculate the Euclidean distance between each computing node and the cluster centers to

measure the differences in communication dependency. The Euclidean distance can be calculated by (2):

$$d(X_i, C_m) = \sqrt{\sum_{j=1}^{N}(X_{i,j} - C_{m,j})^2} \tag{2}$$

where X_i is the feature vector of the ith computing node, C_m is the mth cluster center in X. We assign each computing node to the cluster whose cluster center has the smallest Euclidean distance. After each iteration, we re-calculate the center of each cluster to update it as the new cluster center. This process is repeated until the cluster centers no longer change or until the maximum number of iterations is reached.

We experimented with different numbers of clusters from $k = 2$ to $k = N - 1$ and calculated the sum of squared errors (SSE) for each clustering result. Subsequently, we used the elbow method to analyze the trend of SSE changes corresponding to different k values, identifying the inflection point where the decrease in SSE slows down significantly. The k value at this inflection point is taken as the optimal number of clusters, resulting in the final set of virtual resource nodes VRN. The sum of squared errors can be calculated as (3):

$$SSE = \sum_{i=1}^{k} \sum_{X_j \in VRN_i} (X_j - C_i)^2, \tag{3}$$

where VRN_i represents the ith virtual resource node, which is the set of compute nodes in the ith cluster. To enhance system stability and avoid performance degradation due to significant fluctuations in communication dependencies of computing nodes, we set thresholds ψ to represent the maximum tolerable network fluctuation range. We quantify this using the Frobenius norm $\|X\|_F$ of the X at the current time window T_i and the Frobenius norm of X at the adjacent time window T_{i-1}, $\|X\|_F$ can be calculated by (4):

$$\|X\|_F = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}|X_{i,j}|^2}. \tag{4}$$

The communication fluctuation of computing nodes is calculated using (5):

$$\Delta = \frac{\left\|X_{T_i} - X_{T_{i-1}}\right\|_F}{\left\|X_{T_{i-1}}\right\|_F}. \tag{5}$$

If $\psi > \Delta$ is detected within the current time window, it indicates a significant fluctuation in the communication dependencies of computing nodes, necessitating a rescheduling of the stream application.

4 Hd-Stream: Architecture and Algorithms

In this section, we provide an overview of Hd-Stream based the constructed models presented in the previous section, including its system architecture and algorithms.

4.1 System Architecture

Based on the aforementioned models, we implemented Hd-Stream and integrated it into the mainstream DSC framework Apache Storm. As shown in Fig. 2, Hd-Stream consists of four main components: **Scheduling Trigger**, VRN **Generation**, **Task Deployment** and **Data Monitor**.

Scheduling Trigger reads the current status of the running system from the Database and determines whether to trigger re-scheduling. VRN **Generation** clusters physical computing nodes into virtual resource nodes (VRN)s based on communication dependencies and generates a scheduling sequence of VRNs. **Task Deployment** generates scheduling solutions based on the dependencies of stream application tasks and the VRN scheduling sequence. **Data monitor** is responsible for the real-time collection of the resource utilization of computing nodes, the resource requirements of tasks, and the communication traffic between tasks.

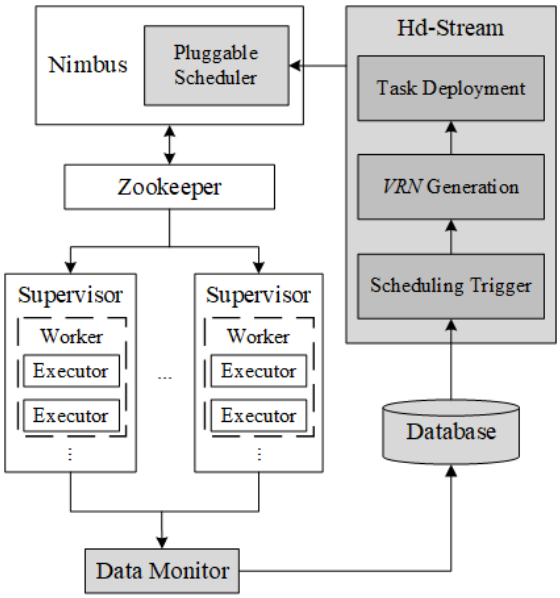

Fig. 2: Hd-Stream's architecture

4.2 Communication Dependency-aware VRN Scheduling

In scenarios where there are significant differences in communication dependencies among computing nodes, if the communication dependencies of compute nodes are overly focused, the scheduling process will involve excessive iterations, leading to high scheduling overhead. To address this problem, we proposed a method of fusion for virtual resource nodes (VRN) to cluster physical resource nodes and generate a scheduling sequence for the VRNs. This method reduces the search space for computing nodes during the scheduling process, thereby reducing scheduling overhead.

To improve the execution efficiency of the scheduling strategy and reduce the overhead in selecting computing nodes for deploying tasks, we calculate the currently available resource amount for each VRN. The VRN with the maximum available resources is selected as the starting node of the VRN scheduling sequence. Subsequently, the scheduling order of VRNs is determined based on the communication costs between them. The communication dependency-aware VRN fusion process is outlined in Algorithm 1.

The input to Algorithm 1 includes the dependency feature matrix X and the maximum number of clusters k_{max}. The output is the VRN scheduling sequence P. Steps 1–7 initialize data structures to store clustering metrics (WCSS) and iterate through cluster numbers k from 2 to $k_{max} - 1$, and computing WCSS for each k by applying K-Means clustering. Steps 9–17 generate clusters V, compute inter-VRN dependency matrices D, and calculate total resources R for each VRN. Steps 18–22 construct the scheduling sequence P by greedily selecting the VRN with the highest resource (m) and iteratively appending the next VRN with the minimum dependency to the current sequence (t) until all VRNs are included. The time complexity of Algorithm 1 is $O(m^2 \cdot d)$, where m and d are the number of rows and columns in the dependency feature matrix X, respectively.

4.3 Hierarchical Dependency-aware Scheduling

Static scheduling lacks prior knowledge task dependencies, making it difficult to fully utilize runtime information to optimize scheduling solutions. Therefore, we propose a hierarchical dependency-aware strategy that comprehensively considers factors such as task migration benefits, task dependencies, computing node dependencies, and system resource constraints. This approach optimizes the scheduling solutions and reduces unnecessary communication overhead across computing nodes.

For task migration between VRNs, we select tasks that can yield migration benefits and migrate them from the current VRN to upstream or downstream VRNs. We define the change in communication traffic between VRNs after task migration as the migration benefit. A cost function is established to quantify this benefit, which is used to calculate both the external and internal costs of task

Algorithm 1 Communication dependency-aware VRN scheduling sequence

Input: Dependency feature matrix X , maximum number of clusters k_{max}
Output: VRN scheduling sequence P

Initialize the list T to store the Within-Cluster Sum of Squares (WCSS) for each clustering;
Initialize the list R to store the total resources of each VRN;
for $k = 2$ to $k_{max} - 1$ **do**
    Run K-Means on X with k clusters for max_{iter} iterations;
    $t \leftarrow$ Compute the WCSS for the current clustering;
        // Store the WCSS and corresponding k
    T.append((t, k));
end for
$n \leftarrow$ Find the optimal number of clusters using elbow rule on WCSS curve T;
$V \leftarrow$ Run K-Means on X with n clusters for max_{iter} iterations;
        // Compute the dependency feature matrix between VRNs
for $i = 1$ to $|V|$ **do**
    for $j = i + 1$ to $|V|$ **do**
        $D_{i,j} \leftarrow$ getDependencyOf(V_i, V_j, X);
        $D_{j,i} \leftarrow Di, j$;
    end for
end for
        // Compute the total resources of each VRN
for $i = 1$ to $|V|$ **do** $R_i \leftarrow$ sumOfResource(V_i);
end for
        // Generate the scheduling sequence
$m = \text{argmax}_j R_j$;
P.append(m);
while $|P| \neq |V|$ **do**
        // Find the minimum dependency between VRNs
    $t \leftarrow$ findMinDepend(D_m), where $t \neq m$ and t not in P; P.append(t); $m = t$;
end while
return P

migration, as shown in (6) and (7):

$$Ex(v) = \sum_{Pa(j) \neq Pa(v)} \left(w_{v,j} + w_{j,v}\right), \tag{6}$$

$$In(v) = \sum_{Pa(j) = Pa(v)} \left(w_{v,j} + w_{j,v}\right), \tag{7}$$

where $Pa(v)$ is the partition to which task v belongs, specifically the VRNs associated with v. $Ex(v)$ is external cost, representing the change in communication traffic when v is moved into the adjacent partition $Pa(j)$. $In(v)$ is internal cost, representing the change in communication traffic resulting from moving task v out of $Pa(v)$.

Based on the existing deployment scheme of stream applications, we obtain a task subgraphs sequence $R = \{R_i, R_2, \cdots, R_m\}$ in VRNs, where m is the

number of utilized VRN. Following the order of subgraphs in R, we first calculate the boundary task sequence between the subgraphs to migration tasks. The boundary tasks in the upstream and downstream task subgraphs are defined as $B_i = \{v \in R_i | \exists j \in R_{i+1}, w_{v,j} > 0\}$ and $B_{i+1} = \{j \in R_{i+1} | \exists v \in R_i, w_{v,j} > 0\}$. B_i and B_{i+1} are the boundary tasks of R_i and R_{i+1}, respectively. R_i and R_{i+1} are two adjacent subgraphs. Subsequently we calculate the task migration benefit by using (8).

$$g(v) = Ex(v) - In(v), v \in \{B_i \cup B_{i+1}\}, \tag{8}$$

Additionally, the task migrations satisfy resource constraint conditions, that is the resource requirements of the tasks do not exceed the resource constraint of the VRN, thereby achieving load balancing of VRNs. The workflow of hierarchical dependency-aware scheduling is outline in Algorithm 2.

Algorithm 2 Hierarchical dependency-aware scheduling

Input: Stream application G , The task subgraph list R
Output: Scheduling scheme P^*

1: Initialize list L to store available resources of VRNs, priority queue Ga to store gain value of boundary tasks;
2: **for** $i = 0$ to $|R| - 1$ **do**
3: $B_i, B_{i+1} \leftarrow$ find boundary tasks of R_i and R_{i+1};
4: $disableTask \leftarrow$ initialize list to store disabled task;
// Calculate the gain value of the boundary tasks
5: **for** task $v \in \{B_i \cup B_{i+1}\}$ **do**
6: $Ga_v = Ex(v) - In(v)$;
7: **end for**
// Move the boundary tasks that generate positive returns to the adjacent partitions
8: **while** $|Ga| \neq 0$ **do**
9: $gain_v \leftarrow Ga$.pop();
10: **if** $gain_v \geq 0$&moving v does not violate resource constraints &$v \notin disableTask$ **then**
11: move v to other partition;
12: update B_i, B_{i+1};
13: $disableTask$.add(v);
14: **else**
15: P^*.append(R_i);
16: Break;
17: **end if**
18: Ga.upadateGainsOf(B_i, B_{i+1});
19: **end while**
20: **end for**
21: **return** P^*

The input to Algorithm 2 includes the stream application G and the task subgraph list R. The output is the Scheduling scheme P^*. Steps 1-4 initialize

a resource availability list L for VRNs and a priority queue Ga of boundary tasks. Steps 5-6 calculate the benefit of boundary tasks. Step 8-19 is the process of task migration. The time complexity of Algorithm 2 is $O(n \cdot k \cdot logk)$, where n is the size of the task subgraph list R, and k is the number of boundary tasks.

5 Performance Evaluation

In this section, we evaluate the performance of Hd-Stream by comparing it with the mainstream works Storm [22] and R-Storm [23] in real-world distributed stream computing environment.

5.1 Experimental Configuration

We configure a resource-heterogeneous computing cluster consisting of 13 computing nodes, where 1 node is the master node running the Nimbus and Zookeeper components to maintain system operation, while the remaining nodes are worker nodes running the Supervisor to process data streams. The hardware configuration and software configuration are presented in Tables 2 and 3, respectively.

Table 2: Hardware Configuration

Parameter	**Master Node**	**Worker Node 1**	**Worker Node 2**	**Worker Node 3**
Quantity	1	3	4	5
VCPU	2GHz, 2 cores	2GHz, 1 core	2GHz, 1 core	2GHz, 4 cores
Memory	4G	2G	4G	4G
Disk	40G	40G	40G	40G

Table 3: Software Configuration

Software	**Version**
OS	CentOs 7
Apache Storm	Apache-Storm-2.4.0
JDK	JDK 1.8
Zookeeper	Zookeeper-3.5.7
Python	Python-3.8.10
Redis	Redis-5.0.7
Kafka	Kafka-3.6.2

We used Yahoo Webscope S5 Dataset [24] for the stream application Yahoo Streaming Benchmark [25] across the experiments. The architecture task

instance parallelism configurations is shown in Fig. 3. *Spout* reads data from external data sources. De-serialization processes the read byte arrays for de-serialization. *Filter* filters tuples that meet the specified criteria. *Projection* extracts the fields of interest from the tuples to create new tuples. *Join* associates with external data based on the tuple ID field to form new tuples. Finally, *Aggregation* groups and aggregates tuples based on active fields and stores them in an external storage system for convenient querying and analysis.

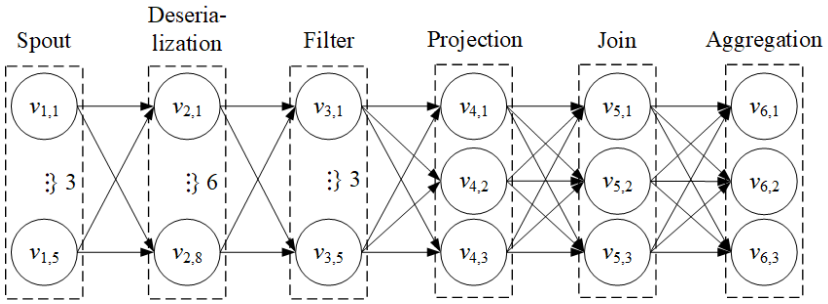

Fig. 3: Task topology of Yahoo Streaming Benchmark

5.2 Resource Utilization

CPU and memory resources are the primary computational resources for distributed clusters processing data, closely related to the cluster's energy consumption and costs. We assess the overall utilization of CPU and memory resources under different scheduling strategies by calculating the average utilization rate, defined as (CPU resource utilization + memory resource utilization) / 2, to evaluate how effectively each scheduling strategy utilizes cluster resources.

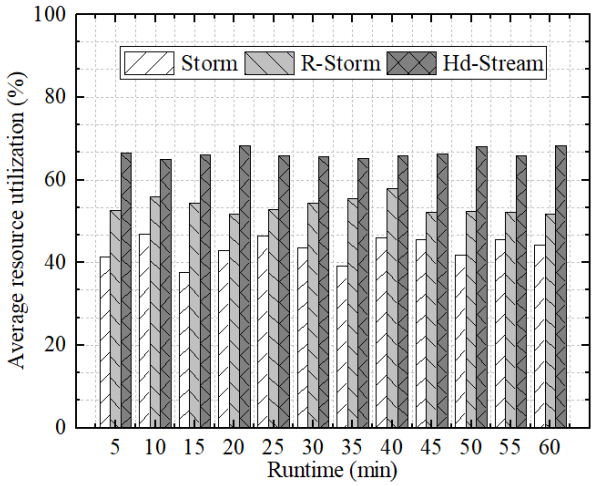

Fig. 4: Resource utilization of Yahoo Streaming Benchmark under a stable stream rates

Under a stable data stream rate of 1000 tuples/s, Hd-Stream improved the cluster's resource efficiency through dynamic scheduling. As shown in Fig. 4, the average resource utilization for Storm, R-Storm, and Hd-Stream across 60 minutes in runtime is 43.45%, 53.66%, and 66.35%, respectively.

To observe the changes in resource utilization under fluctuating data stream rates, the data stream rate is changed from 1000 tuples/s to 2500 tuples/s at the 30th minute. As shown in Fig. 5, after the 30th minute, Storm and R-Storm experienced severe resource overload. In contrast, Hd-Stream maintained load balancing due to its dynamic scheduling mechanism, achieving an overall average resource utilization of 83.03%. This effectively avoided performance bottlenecks caused by imbalance resource distribution and efficiently utilized the resources of the computing nodes in the cluster.

5.3 System Latency

System latency is defined as the time interval between when a tuple enters the system and until it is fully processed. We evaluated the system latency for Hd-stream, Storm and R-Storm and analyzed the reasons behind their system latency trend under stable data stream rates and fluctuating data stream rates, respectively.

Under a stable data stream rate of 3,000 tuples/s, Hd-Stream not only significantly reduces the overall system latency but also achieves a more stable latency trend. As shown in Fig. 6, during the first 24 minutes of runtime, Hd-Stream, R Storm, and Storm maintain low system latency around 25 ms. However, as the stream application is running, data backlog occurs after the 24th minute. Due

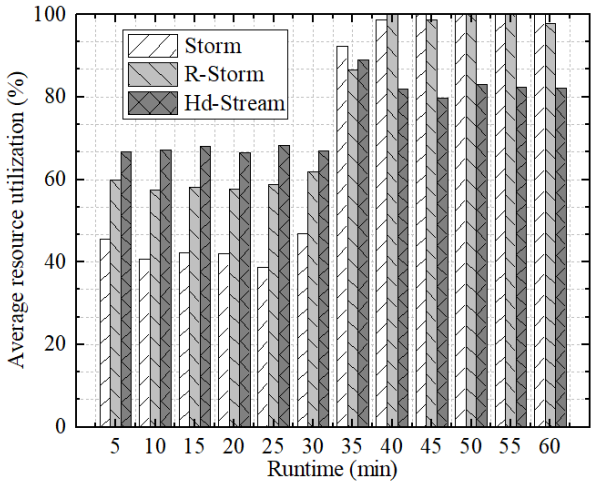

Fig. 5: Resource utilization of Yahoo Streaming Benchmark under fluctuating stream rates.

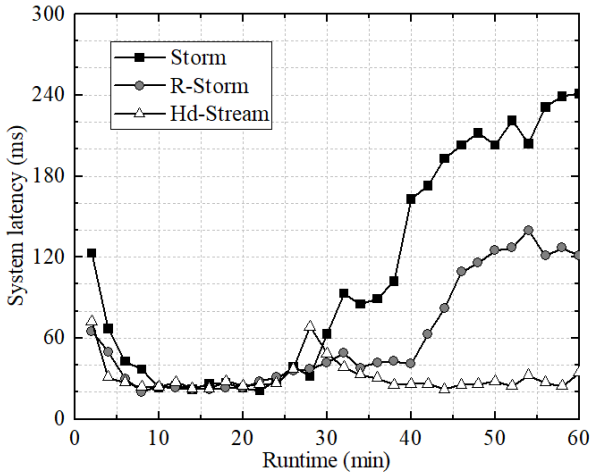

Fig. 6: System latency of Yahoo Streaming Benchmark under a stable stream rate.

to the lack of dynamic scheduling capabilities of Storm and R-Storm, system latency gradually increases to 241 ms and 121 ms at 60th minutes, respectively. In contrast, Hd-Stream dynamically re-schedule the stream application at the 28th minute. Although this introduces scheduling overhead, causing a temporary increase in system latency at 68.2 ms, system latency gradually decreases and stabilizes after the re-scheduling is completed.

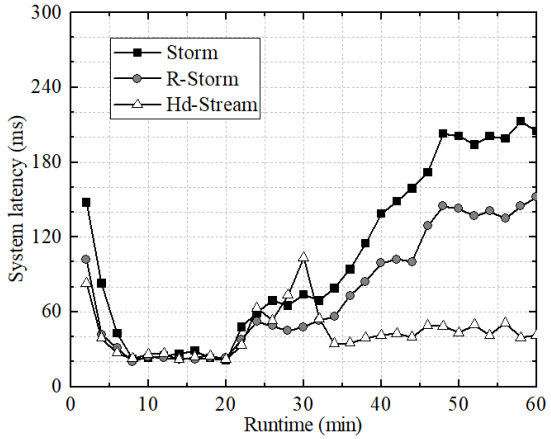

Fig. 7: System latency of Yahoo Streaming Benchmark under fluctuating stream rates.

To access the performance of Hd-Stream under fluctuating stream rates, the data stream rate is gradually increased from 3000 tuples/s to 9000 tuples/s at the 20th minute. As shown in Fig. 7, Hd-Stream rescheduled the stream application at the 28th minute to address the fluctuating data stream, causing a temporary increase in system latency to 103.3 ms. After the rescheduling is completed, system latency decreased and gradually stabilized around 45 ms. In contrast, Storm and R-Storm are unable to cope with the fluctuating data stream, resulting in a gradual increase in system latency.

5.4 System Throughput

System throughput reflects the system's ability to process data tuples, specifically the number of tuples processed per unit time. In the experiment, the system throughput of Hd-Stream, R-Storm, and Storm individually was measured and recorded every five minutes under stable and fluctuating data streams.

Under a stable data stream rate of 1000 tuples/s, Hd-Stream's average throughput is higher than others' throughput. As shown in Fig. 8, the average throughput of Storm, R-Storm and Hd-stream is 888.78 tuples/s, 913.71 tuples/s and 983.73 tuples/s, respectively. Although the average throughput of Hd-Stream is slightly

higher than others' throughput, the differences are not significant due to the low resource load on the system at this data stream rate.

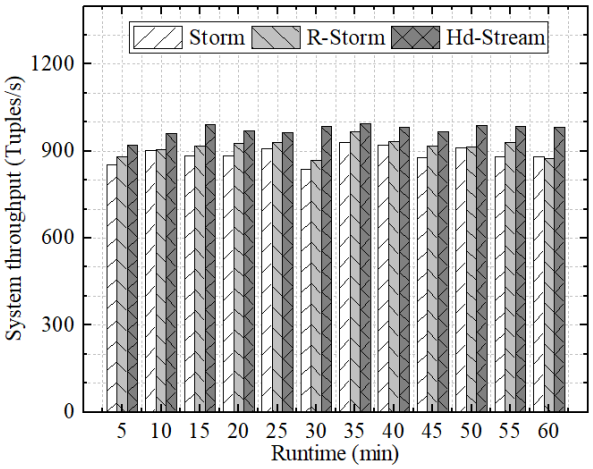

Fig. 8: Average throughput of Yahoo Streaming Benchmark under a stable stream rate.

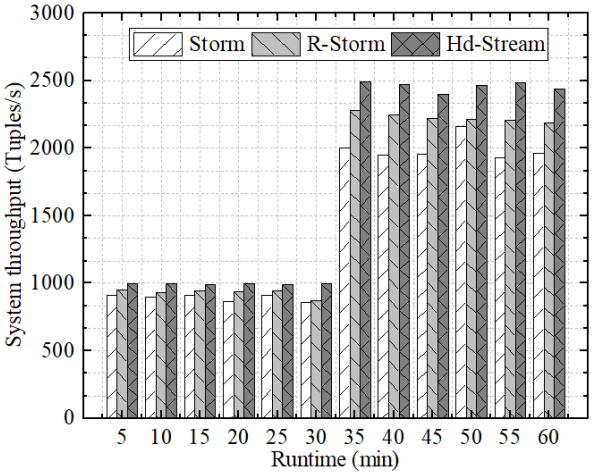

Fig. 9: Average throughput of Yahoo Streaming Benchmark under fluctuating stream rates.

To observe the differences in average throughput under fluctuating data stream rates, the data stream rate was changed from 1000 tuples/s to 2500 tuples/s at the 30th minute. As shown in Fig. 9, Hd-Stream's average throughput was significantly higher than that of Storm and R-Storm, with average throughput of 2458.38 tuples/s, 2224.55 tuples/s, and 1991.8 tuples/s respectively after 30 minutes. Due to Hd-Stream's dynamic scheduling capabilities, when the tuple input rate stabilized at 2500 tuples/s, Hd-Stream exhibited superior average throughput compared to both Storm and R-Storm.

6 Conclusions and Future Work

To address scenarios characterized by significant communication dependency disparities of computing nodes, we proposed Hd-Stream, which is a hierarchical dependency-aware scheduling strategy. At the computing resource level, Hd-Stream transforms physical computing nodes into virtual resource nodes VRNs and generates a scheduling sequence for VRNs based on the communication dependencies between these nodes. Concurrently, at the stream application level, it dynamically schedules stream applications according to the inter-task dependencies. Hd-Stream not only reduces system latency and enhances average throughput but also achieves load balancing of computing nodes. Furthermore, in the face of fluctuating data stream scenarios, Hd-Stream demonstrates superior stability. In the future, we aim to further explore the following areas:

(1) Implement a fair scheduling strategy for multi-stream application scenarios based on the runtime information of the system and stream applications.
(2) Achieve superior and stable system performance through the elastic scaling of operators within the stream applications.

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 under Grant No. 265QZ2021001.

References

1. Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B.J., et al.: Benchmarking streaming computation engines: Storm, flink and spark streaming. In: 2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW). pp. 1789–1792. IEEE (2016)
2. Carcillo, F., Dal Pozzolo, A., Le Borgne, Y.A., Caelen, O., Mazzer, Y., Bontempi, G.: Scarff: a scalable framework for streaming credit card fraud detection with spark. Information fusion **41**, 182–194 (2018)
3. Du, S., Wang, S.: An overview of correlation-filter-based object tracking. IEEE Transactions on Computational Social Systems **9**(1), 18–31 (2021)
4. Fragkoulis, M., Carbone, P., Kalavri, V., Katsifodimos, A.: A survey on the evolution of stream processing systems. The VLDB Journal **33**(2), 507–541 (2024)

5. Li, W., Liu, D., Chen, K., Li, K., Qi, H.: Hone: Mitigating stragglers in distributed stream processing with tuple scheduling. IEEE Transactions on Parallel and Distributed Systems **32**(8) (2021)
6. Tan, J., Tang, Z., Cai, W., Tan, W.J., Xiao, X., Zhang, J., Gao, Y., Li, K.: A cost-aware operator migration approach for distributed stream processing system. IEEE Transactions on Cloud Computing (2025)
7. Xu, H., Liu, P., Ahmed, S.T., Da Silva, D., Hu, L.: Adaptive fragment-based parallel state recovery for stream processing systems. IEEE Transactions on Parallel and Distributed Systems **34**(8), 2464–2478 (2023)
8. Barika, M., Garg, S., Zomaya, A.Y., Ranjan, R.: Online scheduling technique to handle data velocity changes in stream workflows. IEEE Transactions on Parallel and Distributed Systems **32**(8), 2115–2130 (2021)
9. Fu, X., Tang, B., Guo, F., Kang, L.: Priority and dependency-based dag tasks offloading in fog/edge collaborative environment. In: 2021 IEEE 24th international conference on computer supported cooperative work in design (CSCWD). pp. 440–445. IEEE (2021)
10. Al-Sinayyid, A., Zhu, M.: Job scheduler for streaming applications in heterogeneous distributed processing systems. The Journal of Supercomputing **76**(12), 9609–9628 (2020)
11. Röger, H., Mayer, R.: A comprehensive survey on parallelization and elasticity in stream processing. ACM Computing Surveys (CSUR) **52**(2), 1–37 (2019)
12. Mortazavi-Dehkordi, M., Zamanifar, K.: Efficient deadline-aware scheduling for the analysis of big data streams in public cloud. Cluster Computing **23**(1), 241–263 (2020)
13. Mao, Y., Zhao, J., Zhang, S., Liu, H., Markl, V.: Morphstream: Adaptive scheduling for scalable transactional stream processing on multicores. Proceedings of the ACM on Management of Data **1**(1), 1–26 (2023)
14. Liu, X., Buyya, R.: Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions. ACM Computing Surveys (CSUR) **53**(3), 1–41 (2020)
15. Eskandari, L., Mair, J., Huang, Z., Eyers, D.: I-scheduler: Iterative scheduling for distributed stream processing systems. Future generation computer systems **117**, 219–233 (2021)
16. Farrokh, M., Hadian, H., Sharifi, M., Jafari, A.: Sp-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters. Expert Systems with Applications **191**, 116322 (2022)
17. Li, H., Xia, J., Luo, W., Fang, H.: Cost-efficient scheduling of streaming applications in apache flink on cloud. IEEE Transactions on Big Data **9**(4), 1086–1101 (2022)
18. Brown, A., Garg, S., Montgomery, J., KC, U.: Resource scheduling and provisioning for processing of dynamic stream workflows under latency constraints. Future Generation Computer Systems **131**, 166–182 (2022)
19. Liu, X., Buyya, R.: D-storm: Dynamic resource-efficient scheduling of stream processing applications. In: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS). pp. 485–492. IEEE (2017)
20. Ecker, R., Karagiannis, V., Sober, M., Schulte, S.: Latency-aware placement of stream processing operators in modern-day stream processing frameworks. Journal of Parallel and Distributed Computing p. 105041 (2025)
21. Wu, M., Sun, D., Cui, Y., Gao, S., Liu, X., Buyya, R.: A state lossless scheduling strategy in distributed stream computing systems. Journal of Network and Computer Applications **206**, 103462 (2022)

22. Apache: Storm (2025), http://storm.apache.org/
23. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.: R-storm: Resource-aware scheduling in storm. In: Proceedings of the 16th annual middleware conference. pp. 149–161 (2015)
24. Yahoo: webscope (2025), https://webscope.sandbox.yahoo.com/
25. Yahoo: streaming benchmark (2025), https://github.com/yahoo/streaming-benchmarks/